Applications of Matching Theory in Constraint Programming

Von der Fakultät für
Elektrotechnik und Informatik der
Gottfried Wilhelm Leibniz Universität Hannover

zur Erlangung des Grades
Doktor der Naturwissenschaften
Dr. rer. nat.

genehmigte Dissertation

von
Dipl.-Inform. Radosław Cymer
geboren am 18.06.1970 in Łódź

2013

Referent: Heribert Vollmer, Leibniz Universität Hannover
Korreferent: Ulrich Hertrampf, Universität Stuttgart

Tag der Promotion: 16.12.2013

Einfachheit ist das Resultat der Reife.
Simplicity is a result of maturity.

Friedrich Schiller

I would like to thank my supervisor, Prof. Heribert Vollmer, for his assistance in the preparation of this thesis. Our interesting discussions and his patient guidance were a great support. This thesis would not have been written if he had not exposed me to the world of constraint programming. At our first meeting he recommended that I read the book "Constraint Processing" by Rina Dechter in order to get to know constraint programming. This reading fascinated me and gave me the impetus to further research this interesting subject matter. In the course of time I have come to realize that constraint programming has a connection with graph theory and I have deeply researched this area. When I had thoroughly studied the book "Matching Theory" by László Lovász and Michael D. Plummer I perceived that there exists a very good field of research in matching theory in order to connect the results which can help to solve the global constraints. I have read many papers that described an algorithm that could only solve one specific kind of global constraint. Having seen the similarity between the constraints I figured that it is highly possible to discover a universal method to solve some of them at once. This thesis, as the result of my many years of research, describes how this is possible.

I would also like to thank Prof. Qinglin Roger Yu for several helpful comments about the decomposition of graphs. This work could not have been fully achieved without his support.

I also thank Prof. David S. Johnson for his very exciting discussions, advice, his suggestions given and my answered questions regarding the complexity of some non-trivial problems.

Last and by no means least, I am very grateful to my dear wife Anna (to whom this thesis is dedicated) for just being there, staying sane and keeping our family together during the past few years while I was writing this thesis. Without her help, patience and support, the achievement of this goal would have been simply impossible.

I apologize to all those omitted who helped me in one way or another. Even though I have unconsciously forgotten them, I am very grateful for the aid they have given.

Zusammenfassung

Thema: Anwendungen der Matchingtheorie in der Constraint-Programmierung.

Eines der wichtigsten und meist studierten Gebiete der Constraint-Programmierung sind globale Constraints. Globale Constraints sind Klassen von Constraints, die mit einer beliebigen Anzahl von Variablen definiert sind. Diese Dissertation beschäftigt sich mit der Matchingtheorie und deren Anwendung in der Constraint-Programmierung. Es wird hierbei eine universelle Methode entwickelt, um verschiedene globale Constraints zu behandeln.

Die globalen Constraints sollten eine gemeinsame Eigenschaft haben, damit sich ihre Lösung mittels eines Matchingproblems darstellen lässt. Die Grundidee ist, den globalen Constraint als einen Hilfsgraphen zu modellieren, in dem jede Lösung einem zulässigen Matching entspricht. Kanten, die zu keinem Matching gehören, können aus dem Graphen entfernt werden, da die Werte, die sie repräsentieren, keine Lösung darstellen. Solche Kanten können mit der sogenannten alternierenden Breiten- und Tiefensuche ermittelt werden.

Zuerst werden all jene globalen Constraints erörtert, die sich durch bipartite Graphen darstellen lassen. Die nicht erlaubten Kanten können mit Hilfe der Dulmage-Mendelsohn Zerlegung gefunden werden. In diesem Kapitel wird ein universeller Propagierungsalgorithmus für die folgenden Constraints (und ihre weichen Versionen) beschrieben: `alldifferent`, `correspondence`, `inverse`, `same`, `used_by`, `gcc`, und `symmetric_gcc`.

Das nächste Kapitel betrachtet Constraints, die sich mittels Matching in einem einfachen Graphen lösen lassen. Hier wird u.a. der Struktursatz von Gallai-Edmonds verwendet, um die nicht erlaubten Kanten zu ermitteln. Ein Constraint, das sich mit dieser Methode lösen lässt, ist z.B. das globale Constraint `symmetric_alldifferent`, seine Familie, d.h. `symmetric_alldifferent_except_0`, `symmetric_alldifferent_loop`, dessen weiche Version `soft_symmetric_alldifferent_var`, aber auch `2-cycle`, `proper_forest`, `tour` und `undirected_path` Constraints. Die zwei letzten Constraints sind NP-hart.

Als nächstes wird das gerichtete Matching vorgestellt. Dieses Matching lässt sich mittels eines bipartiten Graphen berechnen, wobei hier die bereits entwickelte Methode angewandt werden kann. Hiermit können folgende globale Constraints gelöst werden: `circuit`, `cycle`, `derangement`, sowie dessen weiche Version `soft_derangement_var`, aber auch `binary_tree`, `tree`, `path` und `map`. Meistens repräsentieren diese Constraints jedoch NP-vollständige Probleme und die Domänen der Variablen können nur partiell reduziert werden.

Zuletzt soll die Lösung von Optimierungsproblemen mittels gewichtetem Matching erörtert werden. Bei einem Graphen können sowohl Kanten als auch Knoten mit Gewichten versehen werden. In der Dissertation werden Optimization-Constraints betrachtet, die entweder durch knotengewichtete Graphen (wie z.B. `wpa`, `nvalue` und `swdv`) oder durch kantengewichtete Graphen (wie z.B. `cost_alldifferent`, `soft_inverse_var`, `cost_gcc`, `cost_symmetric_gcc`, `cost_symmetric_alldifferent`, `cost_tour` und `cost_path`) modelliert werden können. Auch hier sind einige Probleme NP-vollständig und erlauben nur partielle Propagation.

Schlagworte: Matchingtheorie, Zerlegungstheorie, extreme Mengen, Dominatoren, Constraint-Programmierung, globale Constraints, Filteralgorithmen, Komplexität

Abstract

One of the most important and most studied areas in constraint programming is global constraints. Global constraints are classes of constraints defined by a relation between a non-fixed number of variables. This thesis deals with matching theory and its application in constraint programming. Based on the existing results of matching and decomposition theory, a universal method will be devised for solving various global constraints.

All constraints should have the property that their solution is representable as a matching problem in a particular graph. The basic idea is to model the solution to the global constraint as a maximum, perfect or complete matching. The edges belonging to no matching (forbidden edges) can be removed from the auxiliary graph because the corresponding values do not belong to a solution. The partition of edges can be computed by means of the so-called alternating breadth-first search and alternating depth-first search.

First, the bipartite matching will be discussed and some constraints representable by a matching in a bipartite graph will be considered. The mandatory and forbidden edges can be found by means of the Dulmage-Mendelsohn Canonical Decomposition. In this chapter we describe a generic propagation routine for the following constraints and their soft versions: `alldifferent`, `correspondence`, `inverse`, `same`, `used_by`, `gcc`, and `symmetric_gcc`.

The next chapter examines constraints solvable by the matching for general graphs. The partition of edges can be determined by means of the Gallai-Edmonds Structure Theorem. A constraint modeled as a maximum matching problem is the well-known constraint `symmetric_alldifferent` and its variants, such as `symmetric_alldifferent_except_0`, `symmetric_alldifferent_loop` and `soft_symmetric_alldifferent_var`, and `2-cycle`, as well the `proper_forest` constraint. By means of the Lovasz-Plummer Decomposition the constraints `tour` and `undirected_path` can be partially solved.

The next chapter introduces a directed matching. This matching can be found by means of the bipartite graph so the method proposed for the bipartite matching can be easily applied. The propagation rules described in this chapter can have an application to some graph partitioning constraints such as `circuit`, `cycle`, `derangement`, `soft_derangement_var`, `tree`, `binary_tree`, `path` and `map`. These constraints are mostly intractable and therefore the domains can be only partially reduced.

In the last chapter of the thesis the weighted matching is disputed. This matching can have an application to solve optimization constraints representable as a weighted matching problem. There are two kinds of weighted matchings: a vertex-weighted matching and an edge-weighted matching. By the first matching the constraints such as `wpa`, `nvalue` or `swdv` can be solved. By the second matching the constraints `cost_alldifferent`, `soft_inverse_var`, `cost_gcc`, `cost_symmetric_gcc`, `cost_symmetric_alldifferent`, as well as `cost_tour` and `cost_path` can be modeled. Some of the constraints represent NP-complete problems so only incomplete filtering is possible.

Keywords: matching theory, decomposition theory, extreme sets, dominators, constraint programming, global constraints, filtering algorithms, computational complexity

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

In this chapter we give a short introduction to the field of constraint programming. We briefly discuss what a constraint satisfaction problem is and how it can be handled. In the next section, a small example is presented that demonstrates how constraint programming works. In the final part of this chapter further details of this thesis, including the contributions, are described and the main publications, on which the thesis is based, are listed.

Since this thesis is about matchings, it is therefore appropriate to start by defining a matching in a graph. Informally, a graph is a fundamental concept that can model binary relations between objects. Each object corresponds to a *vertex* in the graph and the relationships are modeled by *edges*. A *matching* is a set of edges such that no two of them are adjacent. A *maximum matching* is a matching that contains a maximum number of edges. A *perfect matching* is a matching that covers all vertices of the graph. Finding a maximum/perfect matching in a graph is a problem that appears in numerous situations.

Constraint programming is a powerful approach to model and solve hard combinatorial problems. We enumerate some fields where constraint programming can be used (cf. [13]):

- relational databases
- artificial intelligence
- operations research
- combinatorial optimization
- molecular biology
- electrical engineering
- numerical analysis
- natural language processing
- computer algebra

A *constraint satisfaction problem* (abbreviated as *CSP*) consists of a finite set $X = \{x_1, \ldots, x_n\}$ of variables, a set $D = \{D_{x_1}, \ldots, D_{x_n}\}$ of finite domains which represent the set of possible values that each variable can take, and a set of constraints $C \subseteq D_{x_1} \times \ldots \times D_{x_n}$ which link up the variables and define the set of combinations of values that are allowed.

A *solution* to a CSP is an assignment of a single value j from its domain D_{x_i} to each variable $x_i \in X$, such that all constraints are satisfied simultaneously (no constraint is violated). The set of all solutions to a constraint problem P is denoted by $sol(P)$.

A *propagation technique* (or *inference*) is a function that maps a constraint problem $P = (X, D, C)$ to a new constraint problem $P' = (X, D', C)$, where for every $x \in X$ holds $D'_x \subseteq D_x$, and where $sol(P) = sol(P')$.

A CSP is *hyper-arc consistent* with respect to the constraint C on variables $x_1, \ldots, x_k$ if for each index $i \in \{1, \ldots, k\}$, and each value $d \in D_{x_i}$, there exists an element $(d_{x_1}, \ldots, d_{x_{i-1}}, d, d_{x_{i+1}}, \ldots, d_{x_k}) \in C$. A CSP is hyper-arc consistent if it is hyper-arc consistent with respect to all of its constraints. A hyper-arc consistent propagation is a propagation technique that turns a given CSP into a hyper-arc consistent CSP.

In the literature several other names for hyper-arc consistency have often been used such as *generalized arc consistency* (often abbreviated as GAC) or *domain consistency.* This could confuse newcomers entering the field.

A filtering algorithm associated with a CSP aims at removing all the values that do not participate in any solution (are not consistent with the CSP). When all infeasible values are deleted by the filtering algorithm we say that it achieves hyper-arc consistency.

Constraint programming often makes use of global constraints to increase its efficiency and to determine important subproblems of the model. A *global constraint* is a constraint with an arbitrary number of variables often being given as a parameter. Modeling by means of global constraints is therefore more complex but they yield a stronger pruning. The global constraints are characterized in [43]. An extensive list of global constraints is contained in the Global Constraint Catalog [27]. Because of the NP-hardness of many global constraints, it is highly unlikely that a polynomial-time solving technique will ever be found for them.

The subject of this thesis is an algorithmic method for solving global constraints. We will focus in this work on the matching problem and we will discuss how to create a general propagation algorithm for many common constraints. We will develop a tool box for various important global constraints. The filtering algorithm is based on the results of decomposition theory and a modified version of the breadth-first and depth-first traversals.

Matching theory is one of the classical and most important topics in combinatorial optimization. The fundamentals of this field were established by Julius Petersen in 1891. The matching problems have many important applications in graph theory and computer science.

This thesis is devoted to constraint propagation and will typically discuss theoretical results. We investigate the application of matching theory in constraint programming. We introduce a generic method for global constraints that can be represented by a matching in a graph associated with the constraint. We propose to use different matchings as a pruning technique in constraint programming.

We intend to apply matching theory both by checking feasibility and during the propagation phase of constraint programming. The combination of matching theory and constraint programming is a common idea because both methods can be combined to solve the problem.

The algorithmic methods for solving global constraints are plentiful. Therefore, a detailed overview thereof is obviously far out of the scope of a doctoral thesis chapter, or even a handbook. In this thesis we present a method for matching-based constraints. We will consider different versions of the matching problems, depending on whether the graph is bipartite, general or directed, and whether the graph is weighted or not.

Decomposition theory is the most developed area in matching theory. The origin of decomposition theory can be traced back to the book "Matching Theory" by Lovasz and Plummer. Dulmage and Mendelsohn established the existence of a canonical decomposition for bipartite graphs. A few years later, Gallai and Edmonds provided an efficient algorithm for obtaining the decomposition of general graphs. Lovasz and Plummer established the existence of a canonical decomposition in terms of degree-matchings.

Our work proposes a generic filtering algorithm for matching-based constraints. It relies on decomposition theory. The edges of the graph will be divided into edges belonging to no, some or all maximum/perfect matchings. This thesis deals with such a partition, and shows how it can be used to improve the practical results of matching-based global constraints.

This work is an important contribution in establishing a clear link between results in matching theory and the filtering algorithms of several global constraints based on matching. In particular, it gives a distinctive and clear algorithmic framework to implement these algorithms. The exposition of such a theory is very relevant for the constraint programming community. The solving approach presented has the potential to become an essential resource for the solver developers, reducing their work time by efficiently implementing various global constraints.

We would like to point out that most of the results presented here are well known in decomposition theory, but they seem to be ignored in the constraint programming community. The thesis intends to be a comprehensive introduction to this theory, as well as a showcase of its practical applications for pruning algorithms.

An Illustrative Example

As a simple example of pruning, consider the ALLDIFFERENT constraint, which states, as the name indicates, that all variables in this constraint must be pairwise distinct. This global constraint can clearly be replaced by the set of $\binom{n}{2}$ disequalities of the form $x_i \neq x_j$. The standard filtering algorithm for the decomposed ALLDIFFERENT constraint works as follows. Whenever the domain of the variable contains only one value, remove this value from the domains of the other variables that occur in the ALLDIFFERENT constraint. This procedure is repeated as often as possible and has the following domain reduction rules [13, page 85]:

$$\frac{\langle x \neq y; x \in D_x, y = a\rangle}{\langle x \neq y; x \in D_x \setminus \{a\}, y = a\rangle}; \quad \frac{\langle x \neq y; x = b, y \in D_y\rangle}{\langle x \neq y; x = b, y \in D_y \setminus \{b\}\rangle}.$$

One of the disadvantages of this method is that one needs $\frac{n^2-n}{2}$ disequalities to express the ALLDIFFERENT constraint on n variables. Moreover, the worst-case time complexity of the algorithm is $\mathcal{O}(n^2)$. Another, even more important, drawback of the above method is the loss of information: hyper-arc consistency over the corresponding set of binary inequalities is insufficient to detect that the constraint is inconsistent. So, the given pruning routine is inefficient.

However, the more appropriate hyper-arc consistent propagation technique can be efficiently implemented. A very useful theorem exists to derive algorithms that ensure consistency for the ALLDIFFERENT constraint. In order to state it in the terms we have defined, we will introduce convenient notation. Let the cardinality of a set A be denoted by $|A|$ and let S be a family of m sets $S_1, \ldots, S_m$. A *transversal* or *system of distinct representatives* (usually abbreviated as SDR) is a collection of m distinct elements $t_1, \ldots, t_m$ such that $t_i \in S_i$ for $i = 1, \ldots, m$. This being given, we can now state the necessity of whether our constraint can be fulfilled.

Theorem 1.0.1 (Hall [149]) *In order that an SDR will exist, it is sufficient that for each $k = 1, \ldots, m$, any selection of k of the sets will contain between them at least k elements.*

In other words, there exists an SDR if the union of any k sets among $S_1, \ldots, S_m$ contains at least k elements. More formally,

$$\underset{I \subseteq \{1,\ldots,m\}}{\forall} |\bigcup_{i \in I} S_i| \geq |I|.$$

From Hall's Theorem we can now deduce the following result which checks whether the ALLDIFFERENT constraint is satisfiable.

Theorem 1.0.2 ([300, Theorem 5]) *The* ALLDIFFERENT *constraint on the variables x_1, $\ldots$, x_n with respective domains $D_{x_1}, \ldots, D_{x_n}$ has a solution if no subset $K \subseteq \{x_1, \ldots, x_n\}$ exists such that $|K| > |\cup_{x_i \in K} D_{x_i}|$.*

This method only makes it possible to decide whether the global constraint has a solution; it does not find the solution. The following example shows its application.

Example Consider the ALLDIFFERENT constraint on the variables $x_1, \ldots, x_5$ with the following domains: $D_{x_1} = D_{x_2} = D_{x_3} = \{2, 3\}$, $D_{x_4} = \{3, 4\}$ and $D_{x_5} = \{4, 5, 6\}$. This global constraint is obviously inconsistent since there are only two values, namely 2 and 3, available for three variables x_1, x_2 and x_3 that must be mutually different. This can also be detected by Hall's Theorem. Take a subset $K = \{x_1, x_2, x_3\}$, then $|K| = 3$. Furthermore, $|\cup_{x_i \in K} D_{x_i}| = |\{2, 3\}| = 2$. For this subset K, Hall's condition does not hold; hence this constraint has no solution.

Observe that the number of generated sets is exponential, so this method is not practical. An efficient approach for characterizing the values we have to remove from the variable domains to ensure hyper-arc consistency is based on the technique introduced by Régin [255].

We can use one of the constraint propagation algorithms for the global constraint ALLDIFFERENT to prune the domains of variables. For this constraint many specialized filtering methods have been developed. For example, if we apply a filtering algorithm for achieving hyper-arc consistency [255], inconsistency can be detected in time complexity $\mathcal{O}(\sqrt{n} \cdot m)$, where n is the number of variables inside the ALLDIFFERENT constraint and m is the sum of the cardinalities of the domains.

In order to check the feasibility of the ALLDIFFERENT constraint, it is helpful to recognize, that the constraint essentially represents a maximum bipartite matching between variables and values in which each variable must be assigned to a different value. We construct a bipartite graph with vertices on one side that correspond to variables, and vertices on the other side that correspond to values in the domains. There is an edge $\{x_i, j\}$ iff $j \in D_{x_i}$. The ALLDIFFERENT constraint is satisfiable if each vertex representing a variable is incident to an edge belonging to some maximum matching. For instance, the constraint with the domains from our running example can be represented by the bipartite matching problem shown in Figure 1.1. The heavy lines indicate the maximum matching.

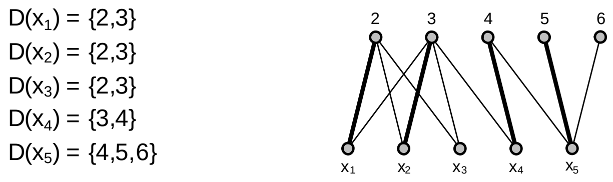


Figure 1.1: Checking feasibility of the ALLDIFFERENT constraint

A matching algorithm for this bipartite graph will determine a maximum matching of cardinality 4 indicating that at most four variables can be matched and so the constraint is unsatisfiable (note that there is no assignment to x_1, x_2 and x_3 which satisfies the constraint).

Theorem 1.0.3 *The* ALLDIFFERENT *constraint on variables $x_1, \ldots, x_n$ has a solution if a maximum matching of the bipartite graph associated with the constraint has cardinality n. Furthermore, value j can be deleted from the domain of the variable x_i if the corresponding edge $\{x_i, j\}$ belongs to no maximum matching.*

Proof See Theorem 1 in [255] and/or Theorem 4 in [300]. □

From the above result the following filtering algorithm can be devised. First, we create the bipartite graph associated with the ALLDIFFERENT constraint. Next, we compute a maximum matching. If the matching does not cover all vertices representing variables then the constraint is inconsistent. Otherwise, we remove edges that belong to no maximum matching. The remaining edges correspond to the feasible values in the domains. This follows from the fact that variable x_i can take value j iff edge $\{x_i, j\}$ is a part of some maximum matching.

In the bipartite graph of Figure 1.1 edge $\{x_4, 4\}$ belongs to all maximum matchings, edges $\{x_4, 3\}$ and $\{x_5, 4\}$ belong to no maximum matching. The remaining edges are part of at least one maximum matching but not all of them.

Outline of the Thesis

In this thesis we investigate the application of matching theory in constraint programming. We propose to apply the results of decomposition theory during the propagation phase.

In Chapter 2 the definitions of general terms are given, the notation used throughout the thesis is fixed, and the basic definitions from graph theory and matching theory are recalled. An experienced reader can skim this chapter.

Chapter 3 discusses algorithms based on searching a graph using either a depth-first search or a breadth-first search. We present Gray-Path Theorem for a depth-first search and an algorithm for the classification of edges in a breadth-first search. Later in this thesis it will be shown how these results can be combined to create a generic filtering algorithm.

In Chapter 4 we propose a general framework to treat global constraints which are representable by matching in a bipartite graph. The algorithms we present can be considered as a generalization of a breadth-first and a depth-first search. They find alternating cycles and alternating paths, which can be used to partition edges in the graph, and thus to prune the domains of the variables. The new results are the Dulmage-Mendelsohn Canonical Decomposition for degree-matchings and the continuity property for degree-factors in bipartite graphs. The bipartite graphs with positive surplus are defined with respect to degree-matchings.

Chapter 5 treats the matching problem in general graphs. A slight modification of the algorithm presented in the previous chapter allows us to find the partition of edges. The contributions of this chapter are the Structure Theorem for Lovasz-Plummer Canonical Decomposition and the better filtering algorithm for the family of SYMMETRIC_ALLDIFFERENT constraint. The devised algorithm iterates over extreme sets and not over the vertices as does the algorithm by Jean-Charles Régin. The theory of extreme sets is extended to f-matchings.

Chapter 6 deals with a so-called directed matching. First we reduce it to a matching problem on a certain bipartite graph, then we show how to apply to this problem the above framework. The algorithm we get is essentially the same as in Chapter 4. The theory of strongly connected components and dominators is specialized to handle global constraints representable by directed graphs. For each graph partitioning constraint, we provide a pruning algorithm which performs optimal filtering or prove that it is intractable.

Chapter 7 is devoted to weighted matching problems. In this chapter we are concerned with costs associated with vertices or edges. The filtering technique is based on the weighted version of the alternating breadth-first and depth-first traversals. Some intractability results belong to the contribution of this chapter.

Chapter 8 concludes the thesis by emphasizing the results and proposing some further fields of research.

To a large extent this thesis is self-contained. Theorems are proved with all details where possible. Further, it contains many figures and examples, which clarify the theory. However, we will assume that the reader is familiar with the basic concepts of algorithms, matching theory and constraint programming. Excellent references are [67], [212] and [13].

Publications

The papers of the work presented in this thesis have been published in several journals. The fundamental manuscript "Dulmage-Mendelsohn Canonical Decomposition as a generic pruning technique" has been published in the journal "Constraints". In this paper we present a general hyper-arc consistency filtering algorithm for matching-based constraints which can be modeled by bipartite graphs. We have exhibited there the following results:

- Positive surplus bipartite graphs with respect to (g, f)-matchings
- Dulmage-Mendelsohn Canonical Decomposition for (g, f)-matchings
- Gray-Path-Theorem for depth-first search
- Alternating breadth-first search for determination of partition of vertices
- Alternating depth-first search for determination of partition of edges
- Alternating paths for perfect (g, f)-matchings
- Shrinking the bounds of degree conditions
- Continuity property in (g, f)-factors
- General propagation routine for hard and soft global constraints

The second manuscript "Gallai-Edmonds Decomposition as a pruning technique" has been published in "Central European Journal of Operations Research". In this paper we extend our work to general graphs. The main contribution of this manuscript is:

- Canonical Decomposition for f-matchings
- Extreme sets with respect to f-matchings
- Some properties of extreme sets with respect to f-matchings
- Transformation from the Lovász-Plummer decomposition into the Gallai-Edmonds one
- The divide-and-conquer approach to determine the partition of edges

The third manuscript "Propagation rules for graph partitioning constraints" has been submitted to the journal "Annals of Operations Research". In this paper we extend our work to directed graphs and present filtering algorithms for graph partitioning constraints. The results can be summarized as follows:

- Transformation of a directed matching into a bipartite matching
- Pruning according to a directed matching, strong connectivity and dominators

The fourth manuscript "Weighted matching as a generic pruning technique applied to optimization constraints" has also been submitted to the journal "Annals of Operations Research". In this paper we extend our work to weighted graphs. The main contribution of this manuscript is:

- Classification of edges in a breadth-first search into tree, back and cross edges
- Vertex-weighted matching problem for (g, f)-matchings
- Weighted alternating depth-first search for vertex-weighted graphs
- Weighted alternating breadth-first search for edge-weighted graphs
- Cost-based propagation algorithm for optimization constraints

The following intractability results have been obtained:

- The interval subset sum problem is NP-complete
- The general minimum vertex-weighted (g, f)-matching problem is NP-hard
- Vertex-weighted (g, f)-matching with cost lying between given bounds is NP-complete
- Edge-weighted (g, f)-matching with cost lying between given bounds is NP-complete

Chapter 2

Preliminaries

In this chapter we present a concise collection of those basic definitions we need to get started. Additional terminology will be presented later in the sequel as needed.

2.1 Some basic concepts

In this section we briefly mention some of the mathematical concepts and notations which will be used throughout the thesis.

We assume that the reader is familiar with the set-theoretical concepts such as sets, subsets, proper subsets, union, intersection, difference and cross product of sets.

If S is a finite set, we will denote the number of elements in S (the cardinality of the set S) by $|S|$; the empty set will be denoted by $\emptyset$.

For any sets A and B, the symmetric difference $(A \setminus B) \cup (B \setminus A) = (A \cup B) \setminus (A \cap B)$ will be denoted by $A \oplus B$.

A multiset is a set in which an element may occur more than once.

For the sets of integer and rational numbers we will use symbols $\mathbb{Z}$ and $\mathbb{Q}$, respectively. The superscript $+$ restricts the sets to the non-negative numbers, i.e. $\mathbb{Z}^+ = \{x \in \mathbb{Z} : x \geq 0\}$. The completion of a proof will be indicated by the symbol $\Box$.

We will use $\lfloor x \rfloor$ for the largest integer not greater than x, and $\lceil x \rceil$ for the smallest integer not less than x (e.g. $\lfloor \pi \rfloor = 3$ and $\lceil \pi \rceil = 4$). Clearly, $\lceil x \rceil = -\lfloor -x \rfloor$.

We denote the interval of integers $\{x : l \leq x \leq u\}$ by $[l, u]$.

In an implication $p \to q$, if p implies q then p is called the *antecedent* and q is called the *consequent*.

An invariant is a parameter or property of graphs that is preserved by isomorphisms (bijections with respect to vertices and edges).

We begin by recalling some elementary notions concerning relations. There are two main kinds of relations that play a very important role in mathematics and computer science: equivalence relations and partial orderings. In this section we define them and investigate some of their properties.

Some typical properties of binary relations are (for all x, y and z in X):

reflexivity:	$(x,x) \in R$
irreflexivity:	$(x,x) \notin R$
symmetry:	$(x,y) \in R \to (y,x) \in R$
asymmetry:	$(x,y) \in R \to (y,x) \notin R$
antisymmetry:	$(x,y) \in R \wedge (y,x) \in R \to x = y$
transitivity:	$(x,y) \in R \wedge (y,z) \in R \to (x,z) \in R$
intransitivity:	$(x,y) \in R \wedge (y,z) \in R \to (x,z) \notin R$

These properties are not independent of each other: an irreflexive and transitive relation is always asymmetric and every asymmetric relation is always irreflexive. Note that irreflexivity is the negation of reflexivity, but intransitivity is not the negation of transitivity, and neither asymmetry nor antisymmetry is the negation of symmetry.

We consider now some especially important combinations of relations. A relation R which is reflexive, symmetric and transitive is called an *equivalence relation*. Such a relation uniquely determines a decomposition of the set X into pairwise disjoint classes: two elements are equivalent if they belong to the same class of the decomposition.

A relation R on a set X is a (reflexive) partial order (or partial ordering) on X if R is reflexive, antisymmetric and transitive. A set X with a partial order is called a *partially ordered set*, also called a *poset* for short.

By a *strict partial order* we mean a relation which is irreflexive and transitive (and thus asymmetric). A partial ordering R on a set X is called a linear order (or total order) on X if any two elements are comparable, i.e. for any two elements x and y of X, either $(x,y) \in R$ or $(y,x) \in R$ (both may hold).

An *undirected graph* is an irreflexive symmetric relation on a finite set of elements called *vertices.* A complete graph is a linear symmetric relation.

A *tournament* is an oriented complete graph, i.e. it is a directed graph in which every pair of vertices is joined by exactly one arc. Thus, assigning an orientation to each edge of a complete graph results in a tournament. A tournament represents a linear irreflexive asymmetric relation. Thus, it is a complete asymmetric digraph.

There is a very close connection between acyclic digraphs and posets. Every directed acyclic graph (DAG) represents a strictly partial order and every poset can be represented in several ways by the so-called comparability digraphs.

Invariant(s)	Property	Relation	Discussed in
connected components	connectivity	equivalence	Chapter 4
strongly connected components	strong connectivity	equivalence	Chapter 6
extreme sets in elementary graphs	canonical partition	equivalence	Chapter 5
directed acyclic graph	acyclicity	poset	Chapter 6
canonical partition of bigraphs	irreducibility	poset	Chapter 4
dominators	reachability	poset	Chapter 6

Table 2.1: Correspondences between relations and graph invariants

A binary operation $\diamond$ on a set S is called *commutative* if $x \diamond y = y \diamond x$ for all $x, y \in S$. It is *associative* if $x \diamond (y \diamond z) = (x \diamond y) \diamond z$ for all $x, y, z \in S$. Operations that do not satisfy these properties are called non-commutative and non-associative, respectively.

A function f from X to Y is a mapping whose domain is X and whose range is contained in Y. A *surjective* function f from X onto Y is a function whose range is Y. An *injective* function f from X into Y assigns different values for every two distinct elements of its domain. A *bijective* function is both surjective and injective.

The function $\log n$ denotes logarithm to the base two, unless explicitly stated otherwise. The *monus* function $\dot{-}$ is defined for numbers n, m by $n \dot{-} m = \max\{0, n - m\}$.

2.2 Complexity of algorithms

In most cases it is impossible to compute exactly the complexity of an algorithm. If f and g are functions assigning real values to natural numbers, then we will use the following notation (called *Landau symbols*):

- $f(n) = \mathcal{O}(g(n))$ if there exists a natural number n_0 and a constant $K > 0$ such that for each $n \geq n_0$ we have $f(n) \leq Kg(n)$
- $f(n) = \Omega(g(n))$ if there exists a natural number n_0 and a constant $K > 0$ such that for each $n \geq n_0$ we have $f(n) \geq Kg(n)$
- $f(n) = \Theta(g(n))$ if both $f(n) = \mathcal{O}(g(n))$ and $f(n) = \Omega(g(n))$
- $f(n) = \tilde{\mathcal{O}}(g(n))$ if $f(n) = \mathcal{O}(g(n) \log^k g(n))$ for some k (the so-called *soft-$\mathcal{O}$* notation)
- $f(n) = o(g(n))$ if $f(n) = \mathcal{O}(g(n))$ and $f(n) \neq \Theta(g(n))$
- $f(n) = \omega(g(n))$ if $g(n) = o(f(n))$.

By *amortized time* we mean the time per operation averaged over a worst-case sequence of operations performed (see survey [286] or [67, Chapter 17]).

We now give a brief summary of the fundamental ideas of complexity theory. A decision problem belongs to the class P if there exists a polynomial algorithm to solve it, and belongs to the class NP if there exists a polynomial algorithm to verify a solution. A basic question in the complexity theory is whether the classes P and NP are equal. An NP-hard problem has the property that any problem in NP can be polynomially reduced to it. An NP-complete problem is an NP-hard problem which belongs to the class NP. NP-hard problems have the property that if they belong to the class P then every NP-complete problem belongs to P. For a detailed discussion of this theory we refer the reader to the books by Garey & Johnson [131], Papadimitriou [241] and/or Sipser [272].

Ackermann's function is defined recursively as

$$\begin{aligned} A(1,j) &= 2^j && \text{for } j \geq 1, \\ A(i,1) &= A(i-1,2) && \text{for } i \geq 2, \\ A(i,j) &= A(i-1, A(i,j-1)) && \text{for } i,j \geq 2. \end{aligned}$$

From this formula we can define the inverse function $\alpha(m,n)$ for $m \geq n \geq 1$ by

$$\alpha(m,n) = \min\{i \geq 1 \mid A(i, \lfloor m/n \rfloor) > \lceil \log n \rceil\}.$$

The function A grows very rapidly; on the other hand, the function α grows extremely slowly. The reader may want to compute some values, but for all practical purposes, $\alpha(m,n) \leq 4$.

The single-variable inverse Ackermann function, written as $\log^* n$, is the number of times the logarithm of n needs to be applied until $n \leq 1$. Thus, $\log^* 65536 = 4$, because $\log\log\log\log 65536 = 1$. Although $\alpha(m,n)$ grows more slowly than $\log^* n$, $\alpha(m,n)$ is not a constant, so the running time $\mathcal{O}(m \cdot \alpha(m,n))$ is not linear.

We will use $f^{(i)}(n)$ to denote the i-fold application of the function f to the input n. The function $\log^* n$ is defined in terms of the $\log^{(i)} n$ function, where

$$\log^{(i)} n = \begin{cases} n & \text{if } i = 0, \\ \log\log^{(i-1)} n & \text{if } i > 0 \text{ and } \log^{(i-1)} n > 0, \\ undefined & \text{if } i > 0 \text{ and } \log^{(i-1)} n < 0 \text{ or } \log^{(i-1)} n \text{ is undefined.} \end{cases}$$

Then, we let

$$\log^* n = \min\{i \geq 0 \mid \log^{(i)} n \leq 1\} = \min\{i \geq 0 \mid \underbrace{\log\log\ldots\log}_{i \text{ times}} n \leq 1\}.$$

2.3 Graph Theory

For background on graph theory we recommend the books by Harary [150], Berge [39,40], Bollobás [49] and/or Gondran & Minoux [143].

The definitions of the terms used are essentially the same as those of [212], but have been modified in detail so as to be applicable within modern terminology. In particular, "point" and "line" have been replaced by the more conventional notations "vertex" and "edge", respectively.

An *undirected graph* (or simply a *graph*) G consists of a finite non-empty set of elements $V(G)$ called *vertices* and a set of unordered pairs of vertices $E(G)$ called *edges*. We allow *multiple* or *parallel* edges here, unless otherwise specified. When parallel edges are not allowed, we will call the corresponding graph *simple*. Also, we will not allow *loops* unless otherwise stated.

Let G be an undirected graph with the vertex set $V(G)$ and the edge set $E(G)$. The number of vertices, denoted by $n = |G| = |V(G)|$, is called the *order* of G, and the number of edges, denoted by $m = ||G|| = |E(G)|$, is called the *size* of G. A graph G is *dense* when the number of edges is close to n^2. A graph G is *sparse* when the number of edges is much less than n^2. More formally, graphs with $|E| = \Theta(|V|^2)$ are often called *dense*, while graphs with $|E| = \Theta(|V|)$ are called *sparse*.

An edge $\{x,y\}$ is usually written xy. There are likely the same number of people using xy rather than $\{x,y\}$, so we will use them without discrimination. A graph with no vertices and hence no edges is called a *null graph* and will be denoted by $\emptyset$. A *trivial graph* is a graph consisting of one vertex and no edges.

If xy is an edge in graph G, edge xy is said to *join* vertices x and y, to be *incident* with vertices x and y, and vertices x and y are said to be *adjacent*. Two edges which share a vertex are also said to be *adjacent*.

The set of edges with exactly one endpoint incident with a vertex in X will be written $\nabla(X)$ and the set of edges with one endpoint in X and the other in Y will be written $\nabla(X, Y)$. More formally, $\nabla(X, Y) = \{xy \mid x \in X \wedge y \in Y\}$. For $X \subseteq V(G)$, the *neighbor set* of X, denoted by $\Gamma(X)$, is the set of all vertices adjacent to at least one vertex of X. More formally, $\Gamma(X) = \{y \mid xy \in E \wedge x \in X\}$. When X is a singleton $\{x\}$, we write $\nabla(x)$ and $\Gamma(x)$ instead of $\nabla(\{x\})$ and $\Gamma(\{x\})$, respectively.

The number of edges in a graph G, incident with a vertex x, is called the *degree* (or *valency*) of x in G and denoted by $deg_G(x)$ (or shortly $d_G(x)$). If graph G is understood, we will sometimes abbreviate this to $d(x)$. Clearly, the degree $d(x)$ is equal to $|\Gamma(x)|$. A loop is considered to have degree 2.

A vertex of odd degree is called an *odd vertex*, and a vertex of even degree is called an *even vertex*. Moreover, a vertex of degree zero is called an *isolated vertex*, and a vertex of degree one is called an *endpoint* (or a *leaf*). An edge incident with an endpoint is called a *pendant edge*.

The earliest result on graph theory is essentially due to Swiss mathematician Leonhard Euler (1707-1783) in 1735 [96], although he did not express it in the language of graphs. It is often called *Handshaking Lemma*:

Lemma 2.3.1 (Handshaking Lemma) *The sum of degrees of all vertices in any graph $G = (V, E)$ is equal to twice the number of its edges. Namely,*

$$\sum_{x \in V} d(x) = 2|E|.$$

Corollary 2.3.2 *Every graph has an even number of odd vertices.*

A graph in which all degrees are equal to k is said to be *k-regular* and if G is k-regular for some k, we simply say that G is *regular*. A graph which is 3-regular is often called *cubic*.

We introduce now some notions concerning paths and cycles in a graph. A sequence $v_0e_1v_1e_2v_2..v_{n-1}e_nv_n$ of vertices and edges, such that every e_i, $1 \leq i \leq n$, is an edge joining vertices v_{i-1} and v_i , starting at the *initial vertex* v_0 and terminating at the *final vertex* v_n, is called a *walk*. Note that in a walk edges and hence vertices can appear more than once. If, however, all edges in a walk are distinct, the walk is called a *trail*, and if, in addition, the vertices are also distinct, the trail is a *path*. We often denote a trail by a sequence $e_1e_2..e_n$ of edges and a path by a sequence $v_0v_1..v_n$ of vertices. The *length of a walk* is the number of occurrences of edges appearing in it, counting repetitions of edges multiple times. The length can be 0 for the case of a single vertex.

An *internal vertex* of a path (trail or walk) is a vertex that is neither the initial nor the final vertex of this path (trail or walk).

A walk or trail which begins and ends at the same vertex will be said to be *closed*. We will define a *cycle* to be any (closed) path of length at least one, together with an edge joining the first and the last vertex. The *length* of a cycle will also be the number of edges it contains. A cycle of length n will be called an *n-cycle*. A *loop* is a cycle of length 1. A walk (trail, path or cycle) of even length is called an *even walk (trail, path or cycle)*, respectively. An *odd walk (trail, path or cycle)* can be defined similarly.

If G is a graph and H is also a graph whose vertices and edges are the vertices and edges of G, then H will be called a *subgraph* of G. If H is a subgraph of G and if every edge joining two vertices of H which lies in G also lies in H, we call H an *induced* subgraph of G. If X is a set of vertices in graph G, then $G[X]$, the subgraph of G *induced by* X, is the induced subgraph of G having vertex set X. A subgraph H of G is said to be *spanning* if $V(H) = V(G)$. A spanning regular subgraph of degree k is called a *k-factor*.

Removal of certain vertices means their removal together with the edges incident with them. If X is a set of certain vertices of G then the graph remaining after the removal of X is denoted by $G - X$. In case $X = \{x\}$ is a singleton set we simply write $G - x$. Similarly, if X is a set of edges of G then $G - X$ denotes the graph arising by omitting the edges of X.

A graph in which every pair of vertices is adjacent is said to be *complete*, and the complete graph on n vertices is denoted by K_n. A maximal complete subgraph of graph G is called a *clique* of G.

A graph is *connected* if every two vertices are joined by a path. A maximal connected subgraph of G is called a *component* of G. Components are *even* or *odd* according to whether their vertex sets have even or odd cardinality.

If the vertex set of a graph G can be partitioned into two disjoint non-empty sets, $V(G) = V_1 \cup V_2$, such that all edges of G join a vertex of V_1 to a vertex of V_2, we call G *bipartite* and refer to (V_1, V_2) as the *bipartition* of G. In this case we will also sometimes call the sets V_1 and V_2 the *color classes* of G. A bipartite graph is often also referred to as a *bigraph*.

A special bipartite graph which we will have occasion to use is K_{n_1,n_2}, the *complete bipartite graph* having color classes of size n_1 and n_2 and in which every vertex in each color class is adjacent with every vertex in the other. In particular, $K_{1,n}$ is called an *n-star* (or sometimes simply, a *star*), and is denoted by S_n. The star S_3 is often called a *claw*.

A graph is *r-partite* if its vertices can be partitioned into r sets, called *partite sets*, in such a way that no edge joins two vertices in the same set. A *complete r-partite graph* is an r-partite graph obtained by joining two vertices iff they lie in different partite sets. If the r-partite sets have sizes $n_1, n_2, \ldots, n_r$, then the resulting graph is denoted by $K_{n_1,n_2,\ldots,n_r}$, and if all of these partite sets have size s, then the graph is denoted by $K_{r(s)}$.

A *wheel* W_{n+1} is a graph that consists of an n-cycle C_n (called a *rim*) every vertex of which is joined to a single common vertex (called a *hub*) by an edge (called a *spoke*).

A graph containing no cycles is called *acyclic*. An acyclic graph is called a *forest*, and if the acyclic graph is also connected, it is called a *tree*. If tree T is a subgraph of graph G and if $V(T) = V(G)$, we call T a *spanning tree* of G.

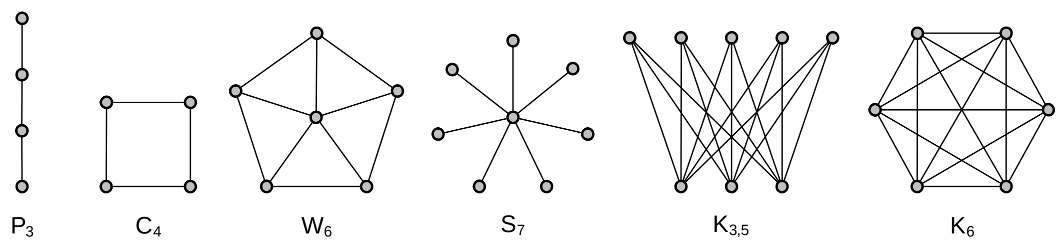


Figure 2.1: Examples of commonly used graphs

- Every tree is a bipartite graph.
- Every connected graph has a spanning tree.
- A connected graph on n vertices is a tree iff it has $n-1$ edges.
- Every non-trivial tree has at least two leaves.

Theorem 2.3.3 *The following statements are equivalent for a graph G:*

1. *G is a tree.*
2. *Every two vertices of G are joined by a unique path.*
3. *G is connected, but if any edge is removed from G, the resulting graph is disconnected.*
4. *G is acyclic, but if any two non-adjacent vertices of G are joined by an edge e, then $G+e$ has exactly one cycle.*

Proof See Theorem 4.1 in [150, Chapter 4]. □

If the edges of a graph have a direction assigned to them, we have what is known as a *directed graph*. More precisely, a *directed graph*, or *digraph*, $D = (V, E)$ consists of a set of nodes $V(D)$ and a set of ordered pairs of nodes $E(D)$ called *directed edges* (or *arcs*). For an arc (u, v) the first node u is its *tail* and the second node is its *head*. The first node u is also called the *source* of the arc (u, v) and the second node is called the *target* of the arc.

The number of arcs having v as their second node is called the *in-degree* of v and is denoted by $deg^{-}(v)$ (or shortly $d^{-}(v)$). Similarly, the *out-degree* of node v is the number of arcs having v as their first node and is written $deg^{+}(v)$ (or shortly $d^{+}(v)$). We use subscripts (e.g. $d^{+}_{D}(v)$) to specify the digraph D if the usage is not clear from the context.

The definitions of walk, trail, path and cycle must be modified somewhat in the case of directed graphs. In each of these alternating sequences of nodes and arcs, we will insist that each (directed) edge join the node before it to the node after it in the sequence. An *acyclic* digraph is one containing no (directed) cycles. A digraph is *strongly connected* if given every ordered pair of nodes (u, v), there is a (directed) path from u to v and from v to u.

There is a digraph version of Handshaking Lemma (see Lemma 2.3.1), which we call the Handshaking Dilemma:

Lemma 2.3.4 (Handshaking Dilemma) *In any digraph, the sum of out-degrees of all nodes and the sum of all the in-degrees are both equal to the number of arcs. Namely,*

$$\sum_{x \in V} d^+(x) = \sum_{x \in V} d^-(x) = |E|.$$

It will be useful to classify each node of a digraph according to the combination of its out-degree and in-degree:

- x is *isolated* if $d^+(x) = d^-(x) = 0$,
- x is a *source* (or *transmitter*) if $d^+(x) > 0$ and $d^-(x) = 0$,
- x is a *sink* (*target* or *receiver*) if $d^+(x) = 0$ and $d^-(x) > 0$,
- x is a *carrier* if $d^+(x) = d^-(x) = 1$,
- x is *ordinary* otherwise, i.e. if $d^+(x) \cdot d^-(x) > 1$.

A *weighted graph* is a graph G together with a function associating a real number $w[e]$ to each edge e, called its *length*, *cost*, *weight*, *capacity* or *penalty* according to context.

We introduce now some graph operations.

The *complementary graph* (or *complement*) of a graph G is the graph $\overline{G}$ having the same vertex set as G, but containing exactly those edges that are not in G.

For two graphs G_1 and G_2, the *union* $G_1 \cup G_2$ is the graph with vertex set $V(G_1) \cup V(G_2)$ and edge set $E(G_1) \cup E(G_2)$, where $V(G_1) \cap V(G_2) = \emptyset$. The union $G \cup G$ is often denoted by $2G$, and for any integer $n \geq 3$, we inductively define nG by $(n-1)G \cup G$. Note that $K_{r(s)}$ is the complement of rK_s (i.e. $K_{r(s)} = \overline{rK_s}$).

The *join* (or *suspension*) $G_1 + G_2$ denotes the graph with vertex set $V(G_1) \cup V(G_2)$ and edge set $E(G_1 + G_2) = E(G_1) \cup E(G_2) \cup \{xy \mid x \in E(G_1) \wedge y \in E(G_2)\}$, i.e. $G_1 + G_2$ is obtained from $G_1 \cup G_2$ by adding all the edges joining a vertex of G_1 to a vertex of G_2. In particular, a complete bipartite graph $K_{r,s} = rK_1 + sK_1$, a wheel graph $W_{n+1} = C_n + K_1$, and a star graph $S_{n+1} = nK_1 + K_1$.

The *Cartesian product* $G_1 \times G_2$ of G_1 and G_2 (also called the *Cartesian sum*) has vertex set $V(G_1) \times V(G_2)$ and the vertex (u_1, u_2) is adjacent to (v_1, v_2) if either $u_1 = v_1$ and u_2 is adjacent to v_2 in G_2, or $u_2 = v_2$ and u_1 is adjacent to v_1 in G_1. Observe that $nK_1 \times G = nG$. Clearly, the 4-cycle $C_4 = K_2 \times K_2$, and the cube $Q_3 = C_4 \times K_2$.

We illustrate in the below figure the graph operations for the case $G_1 = P_2$ and $G_2 = K_2$.

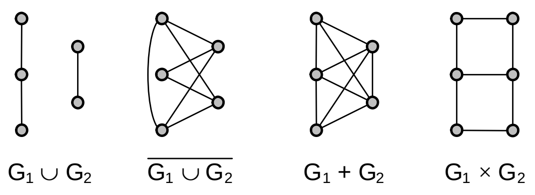


Figure 2.2: Operations on graphs

It is easy to check that all operations are associative and commutative. The above described graph operations are summarized in the following table.

G	$\overline{G}$	$G_1 \cup G_2$	$G_1 + G_2$	$G_1 \times G_2$
n	n	$n_1 + n_2$	$n_1 + n_2$	$n_1 n_2$
m	$\binom{n}{2} - m$	$m_1 + m_2$	$m_1 + m_2 + n_1 n_2$	$n_1 m_2 + m_1 n_2$
$d(x)$	$n - 1 - d(x)$	$d(x)$	$d(x) + n_1$ or $d(x) + n_2$	$d(x) + \nabla(y)$

Table 2.2: The number of vertices and edges in graph operations

2.4 Matching Theory

In this section we recall some of the basic concepts regarding matchings. We will use standard terminology but for the sake of clarity we will repeat the most important definitions and notations from [212].

Formally, the matching problem can be described as follows. Let $G = (V, E)$ be a graph with the vertex set V and the edge set E. A set of vertices in a graph G is said to be *independent* if no two of them are adjacent. A set of edges in a graph G is called a *matching* if no two edges share a vertex in common. The *size* (or *cardinality*) of a matching M is the number of edges in M. Clearly, the cardinality of a matching in a graph G of order n cannot exceed $\lfloor \frac{n}{2} \rfloor$. A *maximum matching* of G is a matching M having the largest cardinality. The number of edges in a maximum matching of G is called the *matching number* of G and is denoted by $\nu(G)$.

Now suppose $M \subseteq E$ is a fixed matching in graph G. Relative to a matching M in G, edges that belong to M are called *matched edges*, while edges not in M are *free* (or *unmatched*) *edges*. A vertex v is said to be *saturated* (*covered* or *matched*) by M if it is incident with a matched edge; otherwise vertex v is *exposed* (or *free*). Non-matched vertices are also called *unsaturated* or *uncovered*. Every saturated vertex v has a *mate*, the other endpoint of the matched edge incident with v. If a matching M in a graph G saturates no vertex in a subset $X \subset V(G)$, then we say that M *avoids* X.

An *alternating trail* (or *alternating path*) is a trail (or path) whose edges are alternately free and matched. An *alternating cycle* is defined similarly. The *length* of an alternating trail, path, or cycle is the number of edges it contains. An alternating path may have length 0; in this case the path contains exactly one exposed vertex. Note that an alternating path may begin with an edge in M or with an edge not in M. If, however, an alternating path P begins and ends at different exposed vertices, we call P an *augmenting path*. Obviously, every augmenting path has an odd number of edges. If M is a matching and P is an augmenting path relative to M then the augmenting operation (symmetric difference) $M \oplus P$ is a matching of size $|M| + 1$. As usual, M is a maximum matching if there is no matching M' in G with $|M'| > |M|$.

A *perfect matching* of a graph G is a matching covering all vertices of $V(G)$, that is, if each vertex of G is incident with exactly one edge of M. Clearly, a perfect matching, when it exists, is maximum, but the converse does not hold in general. A perfect matching of G is sometimes called a *1-factor* of G and a graph with a collection of perfect matchings is said to be a *1-factorable graph*. Similarly, a *2-factor* is a union of vertex-disjoint cycles, and a *2-factorization* of a graph G is a decomposition of G into 2-factors.

A *near-perfect matching* in a graph G is a matching with exactly one exposed vertex. This can only occur when the graph has an odd number of vertices and such a matching must be maximum.

A *complete matching* from X into Y in a bipartite graph G with bipartition (X, Y) is a matching M in which each vertex of X is incident with an edge of M. It is a matching of size $|X|$. Such a matching is also called an *assignment*. Clearly, a complete matching must necessarily be maximum. It is defined as a spanning subgraph such that the degree of each vertex of X is 1 and the degree of each vertex of Y is at most 1.

The following result characterizes a maximum matching (see [212, Theorem 1.2.1]):

Theorem 2.4.1 (Augmenting Path Theorem) *Let M be a matching in a graph G. Then M is a matching of maximum size iff there is no augmenting path relative to M.*

This fact was obtained independently by Claude Berge [38] in 1957 (for the maximum matching problem) and by Robert Z. Norman and Michael O. Rabin [231] in 1959 (for the minimum covering problem). The result was also recognized in 1891 by Julius Petersen [243].

Theorem 2.4.1 is the basis of most algorithms for determining maximum matchings in arbitrary graphs. The basic idea is obvious: we start with any given matching and try to find an augmenting path relative to the present matching in order to enlarge the matching until no such paths remain. Intuitively, an augmenting path can be used to increase the number of edges that belong to a matching. If P is an augmenting path relative to M then the augmenting operation increases the cardinality of the matching M by one.

The following theorem identifies edges that belong to a maximum matching. The proof was given by Julius Petersen [243].

Theorem 2.4.2 *An edge belongs to some, but not to all maximum matchings, iff, for an arbitrary maximum matching M, it belongs either to an alternating path of even length which begins at an exposed vertex, or to an alternating cycle of even length.*

We now state the following corollary which is a direct consequence of Theorem 2.4.2:

Corollary 2.4.3 *An edge belongs to some perfect matching, iff, for a given arbitrary perfect matching M, it either belongs to M or to an even alternating cycle.*

Proof The statement follows immediately from Theorem 2.4.2, because there are no exposed vertices if a perfect matching exists. □

We will call an edge of a graph G *allowed* (or *admissible*) if it occurs in some matching of G and any edge which is not allowed will be called *forbidden*. An edge which belongs to every perfect (or maximum) matching will be called *mandatory*. Clearly, if an allowed edge is isolated, that is, not adjacent to any other allowed edge, then it lies in every matching.

We will call a vertex of a graph G *vital* if it is saturated by every maximum matching. A vertex v is *totally covered* if every edge incident to it is admissible.

A graph G is said to be *elementary* if its allowed edges form a connected spanning subgraph of G. If G has a perfect matching, is connected and all of its edges are allowed, G is said to be *1-extendable* (or sometimes *matching covered*).

Any edge belonging to no maximum matching (any non-admissible edge) can be removed from G, so that the graph can be reduced to a partial graph in which all the edges belong to at least one maximum matching. As the graph can no longer be connected, it will be broken up into matching covered subgraphs, and we will have the canonical decomposition of the initial problem.

For a general graph G let us define the *deficiency* of G, denoted by $\delta(G)$, by the equation

$$\delta(G) = |V| - 2\nu(G).$$

Note that G has a perfect matching iff $\delta(G) = 0$. Hence, $\delta(G)$ is the number of vertices left unmatched by any arbitrary maximum matching, i.e. it is the number of exposed vertices.

Throughout this work we will use the convention that in the figures the solid lines indicate the edges of the graph. The matched edges are depicted as *thick* edges, while the free edges are depicted as *thin* edges. The dashed lines show the forbidden edges. The edges marked with crosses $\times$ are forbidden, as well.

Degree-matchings

A matching defined only in terms of the degrees of its vertices is called a *degree-matching*. Consider a graph $G = (V, E)$ and two non-negative integer-valued functions g and f, called *degree conditions*, defined on the vertex set $V(G)$, such that $0 \leq g(x) \leq f(x) \leq d(x)$ holds for every $x \in V(G)$.

For convenience, we write $f(S)$ instead of $\sum_{x \in S} f(x)$ where $S \subseteq V$. We also define $f^*(S) = \sum_{x \in \Gamma(S)} \min\{f(x), |\nabla(x, S)|\}$ for any subset $S \subseteq V(G)$. With respect to a degree-matching M we introduce the term of degree of a vertex x, denoted by $d_M(x)$, as the number of edges belonging to M and adjacent to x (for an ordinary matching $d_M(x) \in \{0, 1\}$).

We say that a graph G has a perfect f-matching if there exists a spanning subgraph $F \subseteq G$, called an f-factor of G, such that $d_F(x) = f(x)$ for every vertex $x \in V(G)$. In particular, a 1-factor is simply a perfect matching and a 2-factor is a spanning subgraph whose components are cycles.

We say that a graph G has a perfect (g, f)-matching if there exists a spanning subgraph $F \subseteq G$, called a (g, f)-factor of G, such that $g(x) \leq d_F(x) \leq f(x)$ for each vertex $x \in V(G)$. Note that a $(0, 1)$-factor is just a matching and a $(1, 1)$-factor is simply a perfect matching.

In summary, the degree-matching M is called

- matching, if $g(x) = f(x) = 1$ for some x,

- (0,1)-matching, if $g(x) = 0$, $f(x) = 1$ (i.e. $d_M(x) \in \{0, 1\}$) for some x,
- f-matching, if $g(x) = f(x) = d_M(x)$ for some x,
- (g, f)-matching, if $g(x) \leq d_M(x) \leq f(x)$ for some x,
- parity (g, f)-matching, if $g(x) \equiv d_M(x) \equiv f(x) \pmod 2$ for some x.

The degree-matching is called perfect if the above conditions hold for all x. If there exists no perfect degree-matching in G, one could try to minimize the number of exposed vertices. Another way is to measure how much the degree conditions are violated by any degree-matching M. For an arbitrary degree-matching M of a graph G with degree conditions g and f we define the *deficiency* (or *deviation*) of M as follows

$$\delta_M(G, (g, f)) = \sum_{x \in V(G)} \max\{g(x) - d_M(x), 0, d_M(x) - f(x)\}.$$

The deficiency of G with respect to degree conditions g and f is defined as

$$\delta(G, (g, f)) = \min_M \delta_M(G, (g, f)),$$

where the minimum is taken over all degree-matchings M of G. An optimal degree-matching is a matching M such that $\delta_M(G, (g, f)) = \delta(G, (g, f))$. A degree-factor is a matching of deficiency 0. If the degree-matching and the degree conditions are clear from the context, we will denote the deficiency of G shortly by $\delta(G)$.

If the graph induced by all matched edges is connected, then we call the degree-matching the *connected degree-matching*. Connected (g, f)-matchings and connected f-matchings in a graph G can be defined in a similar way. Clearly, G can have a degree-matching but it may not necessarily be connected. The problem of determining if a graph has a degree-factor is polynomially solvable, but determining whether a graph has a connected degree-matching is intractable [131, Problem GT26] (see also survey [196]).

Now suppose $M \subseteq E$ is a fixed degree-matching in graph G. With respect to M we can classify the vertices of G into the following types:

- exposed vertex, if $d_M(x) < g(x)$,
- saturated vertex, if $g(x) \leq d_M(x) \leq f(x)$,
- supersaturated vertex, if $f(x) < d_M(x)$.

In this work, without loss of generality, we will only consider degree-matchings without supersaturated vertices.

Shrunken degree conditions Let G be an arbitrary graph with degree conditions g and f. Denote by $\delta(G)$ the deficiency of G with respect to (g, f). Then the degree conditions are called *shrunken* if for each vertex $x \in V(G)$ there exists a (g, f)-optimal subgraph H with deficiency $\delta(H) = \delta(G)$ such that $d_H(x) = g(x)$ and there exists a (g, f)-optimal subgraph H' with deficiency $\delta(H') = \delta(G)$ such that $d_{H'}(x) = f(x)$.

In the rest of this section we give a construction of all degree-factors of a graph, or, in case that no degree-factors exist, of all optimal degree-factors which come as close to degree-factors from below as possible.

If we want to investigate degree-matchings, then we may try to reduce the degree-matching problem to the maximum matching problem, by constructing from the given graph G another graph G^*, called an *incremental graph* [143], or *inflated graph* [75], such that G^* has a perfect matching iff G has a perfect (g, f)-matching. That such a construction is possible, was first observed by William T. Tutte, who in 1954 gave a reduction from the perfect f-matching problem to the perfect matching problem [293] and in 1981 a reduction from the perfect (g, f)-matching problem to the perfect f-matching problem [294] (see also [212, Section 10.1]).

In fact, a perfect (g, f)-matching problem can be reduced to a perfect f-matching problem and hence to a perfect matching problem. Clearly, the reduction of a perfect degree-matching problem to a perfect matching problem can be realized in polynomial time. This reduction shows also that a degree-matching can be computed in polynomial time. Thus, degree-matching problems can be solved by applying algorithms for maximum matching on the incremental graph. We can also use the results to let the algorithm work directly on the initial graph, and possibly obtain more efficient implementation.

We will now sketch the reduction routine for degree-matchings that forms the main theme of this section. Let us consider now various types of matchings and their gadgets. Our notation and description will be compatible with that of [220], except that "inner" and "outer" will be replaced by the conventional terminology "internal" and "external", respectively, in order to avoid confusion with standard terminology of matching theory.

In this method, every vertex x of the input graph G is replaced by a particular, more complex subgraph, called a *gadget* [220], *substitute* [115], or *template* [75].

As stated in [220]: every gadget consists of *external vertices*, *core vertices*, *external edges* and *core edges*. Core edges join core vertices to external vertices and external edges join external vertices of different gadgets. Each core vertex is adjacent to all external vertices resulting in a complete bipartite graph. Vertices of the same class are not adjacent to one another, unless loops attached to the replaced vertex are modeled by joining two of the external vertices. The second type of gadgets contains additionally *internal vertices* and *internal edges*. Internal edges directly join internal vertices to external vertices. Each internal vertex is adjacent to exactly one external vertex, while each core vertex is adjacent to all internal vertices resulting in a complete bipartite graph.

The incremental graph G^* of the graph G with degree condition f is obtained from G by replacing every vertex x by a complete bipartite graph $K_{d(x),d(x)-f(x)}$. Hence, the gadget has $d(x)$ external vertices, $d(x) - f(x)$ core vertices, and $d(x) \cdot (d(x) - f(x))$ core edges.

Obviously, the smaller the value of $f(x)$, the more complex the gadget and, therefore, the slower the computation of perfect f-matching. Consequently, for small values of $f(x)$, such that $2 \cdot f(x) < d(x)$, an alternative and more appropriate set of gadgets exist. This set of gadgets has been investigated by Meijer, Núñez-Rodríguez, & Rappaport [220]. They present gadgets that are bipartite complete graphs $K_{d(x),f(x)}$ with a pendant edge attached at each external vertex. The external vertices then become internal vertices, and the leaves become external vertices. An obvious consequence of this definition is that the gadget has $d(x)$ external vertices, $d(x)$ internal vertices, $f(x)$ core vertices, $d(x)$ internal edges, and $d(x) \cdot f(x)$ core edges.

Observe that it is superfluous to transform the vertices with $f(x) = 1$. Thus, in our transformation only vertices x, such that $f(x) \geq 2$, are replaced by a gadget. Note that for our algorithm, both sets of gadgets may be combined together resulting in a graph with the minimized number of edges. The best performance is obtained when a valid vertex x of the input graph G is replaced by a gadget with the smaller number of edges and vertices.

Clearly, this reduction of the f-factor problem to the perfect matching problem is a polynomial-bounded reduction. It is not difficult to verify that the obtained graph G^* indeed satisfies perfect matching property: there exists a perfect f-matching in the original graph G iff there exists a perfect matching in the corresponding graph G^*.

However, these constructions are very inefficient, since they increase the number of vertices and edges in G^* to $\mathcal{O}(m)$ and $\Omega(m \cdot n)$ respectively, which leads to an overall slow algorithm in the worst case. Therefore, one way to reduce the time complexity would be to construct a graph G^* with the number of edges estimated by $\mathcal{O}(m)$. Such a reduction method has been discovered by Gabow [115], who presented an efficient method for reducing perfect (g, f)-matchings to perfect matchings (in fact, a perfect (g, f)-matching problem must be first reduced to a perfect f-matching problem and then to a perfect matching problem). He follows the approach of Tutte but utilizes the so-called *sparse substitutes*. These serve to reduce the size of the transformed graph and lead to an $\mathcal{O}(\sqrt{f(V)} \cdot m)$ algorithm for the construction of perfect (or optimal) (g, f)-matching.

Theorem 2.4.4 *An optimal (g, f)-matching in a graph G can be found in $\mathcal{O}(\sqrt{f(V)} \cdot m)$ time and $\mathcal{O}(m)$ space.*

Proof See Theorem 3.1 in [115]. □

The algorithm presented in [115] which computes the perfect degree-matching, can be easily modified to give an optimal degree-matching or a perfect degree-matching with a minimum/maximum number of edges.

Clearly, a maximum matching in G^* corresponds to an optimal degree-matching in G. In order to compute an optimal degree-matching we form an initial matching obtained by repeatedly selecting core vertices of all gadgets, choosing a free adjacent edge and marking it as matched. This results in a complete matching of all gadgets. The entire procedure can be implemented to run in linear time. We can therefore assume that any initial matching which we construct on an incremental graph G^* covers all internal and core vertices in every gadget. Observe that this is always possible, so we can easily start with such an initial matching. Clearly, augmenting the matching never exposes a vertex that is already saturated. Then every maximum matching of the incremental graph G^* which saturates all core and internal vertices in every gadget induces an optimal degree-matching in a graph G, and the converse is also true. Note that this correspondence between optimal degree-matchings and maximum matchings is, in general, not bijective. Thus, there exist several maximum matchings of G^* which correspond to the same optimal degree-matching of G. It should be also noted that the construction of G^* is not unique. Clearly, if graph G is a multigraph then G^* is a simple graph. On the other hand, if G is bipartite and every gadget in G^* is bipartite then G^* is also bipartite.

The idea to make use of gadgets in the reduction from a degree-factor problem into a perfect matching problem is not new. There are monographs in the literature containing this approach (see, for example, Berge [40, Chapter 8], Bollobás [49, Chapter II.3], Gondran & Minoux [143, Chapter 7 (Section 5.2)], Lovász & Plummer [212, Section 10.1]). But none of them gives an efficient algorithm for their use with the problem of determining degree-factors, including systematic study.

Year	Author(s)	Complexity	Strategy/Remarks
1964	Goldman [142]		reduction to ordinary matchings
1967	Urquhart [295]	$\mathcal{O}(f(V) \cdot n^3)$	for weighted degree-matchings
1976	Goodman et al. [144]	$\mathcal{O}(n)$	for unweighted/weighted trees
1983	Gabow [115]	$\mathcal{O}(\sqrt{f(V)} \cdot m)$	sparse substitutes
1983	Gabow [115]	$\mathcal{O}(f(V) \cdot m \cdot \log n)$	for weighted degree-matchings
1984	Hartvigsen [153]	$\mathcal{O}(n^3)$	for weighted (1,2)-matchings
1985	Anstee [12]	$\mathcal{O}(n^3)$	alternating walks
1990	Heinrich et al. [155]	$\mathcal{O}(\sqrt{g(V)} \cdot m)$	criterion of Lovász [210]
1993	Hell & Kirkpatrick [158]	$\mathcal{O}(\sqrt{g(V)} \cdot m)$	for bipartite graphs
2009	Meijer et al. [220]	$\mathcal{O}(n^3)$	gadgets

Table 2.3: History of algorithms for subgraphs with prescribed degrees

We conclude this section with the following demonstration. It is well known that problems involving 2-matchings are much easier to solve than the corresponding problems for ordinary matchings. In fact, a 2-matching problem can be transformed to an ordinary matching problem and a 2-factor problem can be reduced to finding a perfect matching in a bipartite graph (see Theorem 6.1.4 in [212]). However, a 2-factor can be degenerated, since it may include the same edge twice in a cycle (in fact, it is a perfect $(1,2)$-matching).

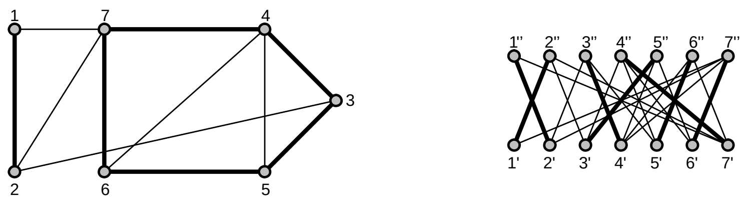


Figure 2.3: A perfect $(1,2)$-matching (degenerated 2-factor)

A simple 2-factor is a 2-factor which is not degenerated. In the figure below we present how we can compute a simple 2-factor using the method presented in this section. On the right side of the figure the incremental graph of the graph given on the left side of the figure is depicted. The perfect matching of the incremental graph corresponds to the perfect 2-matching of the original graph.

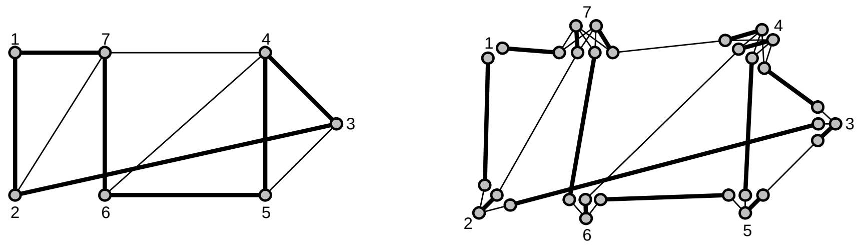


Figure 2.4: A perfect 2-matching (simple 2-factor)

2.5 Decomposition Theory (bipartite graphs)

Recall that a graph G is *bipartite* if the vertex set $V(G)$ can be partitioned into two sets V_1 and V_2 in such a way that no two vertices from the same set are adjacent. The sets V_1 and V_2 are called the *color classes* of G and (V_1, V_2) is a *bipartition* of G.

Theorem 2.5.1 (König [191]) *A graph G is bipartite iff G has no cycle of odd length.*

Theorem 2.5.2 (Hall [149]) *A bipartite graph G with bipartition (V_1, V_2) has a matching covering V_1 iff $|X| \leq |\Gamma(X)|$ holds for each $X \subseteq V_1$.*

Theorem 2.5.3 (Frobenius [112]) *A bipartite graph G with bipartition (V_1, V_2) has a perfect matching iff $|V_1| = |V_2|$ and $|X| \leq |\Gamma(X)|$ holds for each $X \subseteq V_1$.*

The following results can be easily obtained as a generalization of Hall's Theorem:

Theorem 2.5.4 (Yuting & Kano [311]) *Let $G = (V_1 \cup V_2, E)$ be a bipartite graph, and let f be an integer-valued function defined on V_1. Then G has a spanning subgraph H such that $d_H(x) = f(x)$ for all $x \in V_1$ and $d_H(y) = 1$ for all $y \in V_2$ iff $f(V_1) = |V_2|$ and $f(X) \leq |\Gamma(X)|$ for all $X \subseteq V_1$.*

Theorem 2.5.5 *Let $G = (V_1 \cup V_2, E)$ be a bipartite graph with degree condition f. If G has a perfect f-matching then $f(V_1) = f(V_2)$ and $f(X) \leq f(\Gamma(X))$, for all $X \subseteq V_1$.*

Proof See Theorem 2.4.4 in [212]. □

Theorem 2.5.6 *Let G be a bipartite graph with bipartition (V_1, V_2) and degree conditions g and f. If G has a perfect (g, f)-matching then* $\max\{g(V_1), g(V_2)\} \leq \min\{f(V_1), f(V_2)\}$ *and $g(X) \leq f(\Gamma(X))$ holds for every $X \subseteq V_1$ and $X \subseteq V_2$.*

Elementary bipartite graphs and bipartite graphs with positive surplus play an important role as building blocks in decomposition theory with respect to degree-matchings.

2.5.1 Elementary bipartite graphs

An elementary bipartite graph is a graph in which every edge is contained in some perfect matching. An example is a simple cycle C_n of even length or a complete bipartite graph with the same number of vertices in its color classes ($K_{r,s}$ with $r = s$).

In a similar way we can define elementary bipartite graphs with respect to perfect f-matchings and perfect (g, f)-matchings.

Let D be the directed graph obtained from G by replacing each matched edge by an arc leading from V_1 to V_2 and by orienting all other edges in the opposite direction. Then the strongly connected components of D are exactly the elementary bipartite subgraphs of G. However, this property does not hold for elementary graphs with respect to perfect (g, f)-matchings.

The following properties characterize elementary bipartite graphs [212, Theorem 4.1.1]:

- Every elementary graph with at least four vertices is 2-connected.
- A bipartite graph G with bipartition (V_1, V_2) is elementary iff $|V_1| = |V_2|$ and $|\Gamma(X)| \geq |X| + 1$ for every non-empty proper subset X of V_1 or V_2.
- $G = K_2$ is the only elementary bipartite graph with the unique perfect matching.
- For any $x \in V_1$ and $y \in V_2$ if $|V(G)| \geq 4$ then $G - x - y$ has a perfect matching.
- For any perfect matching M digraph D is strongly connected.

The last four properties were independently discovered first by Hetyei [162] in 1964, and later by Little, Grant and Holton [207] in 1975, providing criteria for a graph to be elementary (see also [212, Theorem 4.1.1]).

Extending elementary bipartite graphs with perfect matching to elementary bipartite graphs with perfect f-matching we receive the following properties:

- $f(V_1) = f(V_2)$ and for $\emptyset \neq X \subset V_1$ or $\emptyset \neq X \subset V_2$ holds $f(X) < f^*(X)$.
- Edges adjacent to vertices with $f(x) = d(x)$ belong to all perfect f-matchings.
- For any perfect f-matching M digraph D is strongly connected.

The following properties characterize elementary bipartite graphs with respect to perfect (g, f)-matchings:

- $\max\{g(V_1), g(V_2)\} \leq \min\{f(V_1), f(V_2)\}$,
- For every non-empty proper subset X of V_1 or V_2 holds $g(X) < f^*(X)$.

Figure 2.5 shows an elementary bipartite graph with respect to perfect (g, f)-matchings. The pairs of numbers (g_i, f_i) beside the vertices indicate the degree conditions. This graph has the following properties (other than properties of elementary graphs with perfect f-matchings):

- Edges x_1y_1, x_1y_2, x_5y_5 and x_6y_7 belong to all perfect (g, f)-matchings,
- Digraph D has two strongly connected components $\{x_1, x_2, x_3, y_1, y_2, y_3\}$ and $\{x_4, x_5, x_6, y_4, y_5, y_6, y_7\}$ although edges x_4y_2 and x_5y_3 belong to some perfect (g, f)-matching.

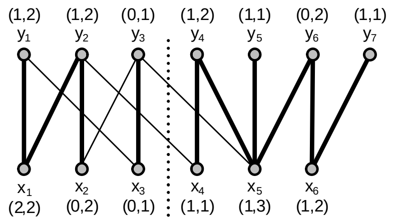


Figure 2.5: Elementary bipartite graph with respect to (g, f)-matchings

2.5.2 Bipartite graphs with positive surplus

A graph G with bipartition (V_1, V_2) has positive surplus (as viewed from V_1) if the number of neighbors of S is bigger than the size of S for any non-empty subset S of V_1, i.e. $|S| < |\Gamma(S)|$ for any $\emptyset \neq S \subseteq V_1$. The surplus of the set S is defined by $|\Gamma(S)| - |S|$. The surplus of G is the minimum surplus of all non-empty subsets of V_1 and will be denoted by $\sigma(G)$.

Bipartite graphs with positive surplus are connected. Simple examples include stars S_k (as viewed from center vertex) and the complete bipartite graphs with different cardinalities of their color classes ($K_{r,s}$ with $r \neq s$). A more elaborate example is presented in Figure 2.6.

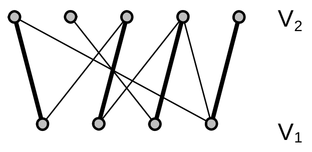


Figure 2.6: Positive surplus bipartite graph

Lemma 2.5.7 *Let $G = (V_1 \cup V_2, E)$ be a bipartite graph with positive surplus (as viewed from V_1). Then the following holds:*

- *Vertices in V_1 are saturated by every maximum matching,*
- *Vertices in V_2 are exposed by at least one maximum matching,*
- *Every edge of G does not belong to at least one maximum matching.*

Moreover, $\delta(G) = |V_2| - |V_1|$.

Proof Since $|X| < |\Gamma(X)|$ for all $\emptyset \neq X \subseteq V_1$, according to the well-known Hall's Theorem (Theorem 2.5.2), every maximum matching of G saturates all vertices in V_1. Let $\{x, y\} \in E$, where $x \in V_1$ and $y \in V_2$, and let M be a maximum matching of G not covering y. Since

M saturates x we can assume that $\{x, z\} \in M$ because M is a complete matching. Let $M' = M - \{x, z\} + \{x, y\}$. Then M' is the desired maximum matching of G containing $\{x, y\}$. □

Theorem 2.5.8 *The following statements are equivalent for a bipartite graph with bipartition (V_1, V_2):*

1. *G has positive surplus (as viewed from V_1),*
2. *G contains a spanning forest F such that $d_F(x) = 2$ for every vertex $x \in V_1$,*
3. *$|V_1| < |V_2|$ and $|X| < |\Gamma(X)|$ for every non-empty subset X of V_1,*
4. *for every $x \in V_1$ the graph obtained by adding a vertex to V_1 connected to the vertices of $\Gamma(x)$ has a matching of cardinality $|V_1| + 1$,*
5. *inserting $\delta(G) = |V_2| - |V_1|$ new vertices to V_1 and joining them to the vertices in V_2 results in a graph with a perfect matching.*

It follows that a bipartite graph G is elementary iff $G - x$ has positive surplus for any vertex x of G.

Positive surplus bipartite graphs with respect to f-matchings Let G be a bipartite graph with bipartition (V_1, V_2) and let $f(x)$ be an integer-valued function on the set $V(G)$ such that $0 \leq f(x) \leq d(x)$ for each $x \in V(G)$. Graph G is said to have positive surplus (as viewed from V_1) if $f(X) < f^*(X)$ for any $\emptyset \neq X \subseteq V_1$.

An example of a positive surplus bipartite graph with respect to f-matchings is shown in Figure 2.7:

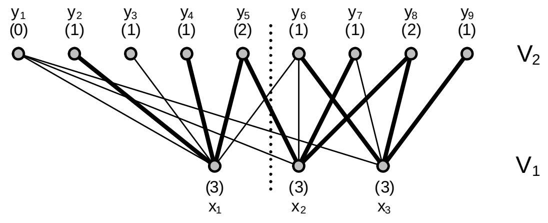


Figure 2.7: Positive surplus bipartite graph with respect to f-matchings

The graph consists of two parts: a bipartite subgraph with positive surplus (depicted on the left side of the figure) and an elementary bipartite subgraph with perfect f-matching (depicted on the right side of the figure). Some properties are immediate:

- Vertices in V_1 are saturated by every optimal f-matching
- Every edge $\{x, y\}$ with $x \in V_1$ and $f(x) = d(x)$ is contained in all optimal f-matchings
- The deficiency of G is equal to $\delta(G) = f(V_2) - f(V_1)$

- Inserting a new vertex x to V_1 with $f(x) = f(V_2) - f(V_1)$ and connecting it to all vertices in V_2 results in a graph with a perfect f-matching.

Note that positive surplus bipartite graphs with respect to degree-matchings may have edges contained in every optimal matching and/or edges contained in no optimal matching. For example, the graph depicted in Figure 2.7 has the following properties (other than properties of bipartite graphs with positive surplus):

- Vertex y_1 of V_2 is neither exposed nor saturated
- Vertices y_6, y_7, y_8 and y_9 of V_2 are saturated by every optimal f-matching
- The remaining vertices of V_2 are exposed by at least one optimal f-matching
- Edges x_2y_5, x_2y_8, x_3y_8 and x_3y_9 belong to all optimal f-matchings
- Edges x_1y_1, x_1y_6, x_2y_1 and x_3y_1 belong to no optimal f-matching

In a similar way we can define positive surplus bipartite graphs with respect to (g, f)-matchings.

Positive surplus bipartite graphs with respect to (g, f)-matchings Let G be a bipartite graph with degree conditions g and f. Then the graph G is said to have positive surplus (as viewed from V_1) if $g(X) < f^*(X)$ holds for any $\emptyset \neq X \subseteq V_1$.

An example of positive surplus bipartite graph with respect to (g, f)-matchings is presented in Figure 2.8:

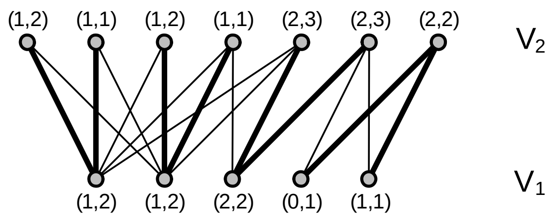


Figure 2.8: Positive surplus bipartite graph with respect to (g, f)-matchings

Let $G = (V_1 \cup V_2, E)$ be a positive surplus bipartite graph (as viewed from V_1) with respect to (g, f)-matchings. For shrunken degree conditions $\hat{g}$ and $\hat{f}$ it holds that $\hat{g}(x) = f(x)$ for every $x \in V_1$ and $\hat{f}(x) = g(x)$ for every $x \in V_2$. Thus, each property for any positive surplus bipartite graph with respect to f-matchings holds also for positive surplus bipartite graphs with respect to (g, f)-matchings. In particular, G has the following, similar properties:

- Vertices in V_1 are saturated by every optimal (g, f)-matching
- Every edge of G belongs to at least one optimal (g, f)-matching
- The deficiency of G is equal to $\delta(G) = g(V_2) - f(V_1)$

2.6 Decomposition Theory (general graphs)

Recall that a *near-perfect matching* in a graph G is one in which exactly one vertex is exposed. This can only occur when the graph has an odd number of vertices, and such a matching must be maximum. If, for every vertex in a graph, there is a near-perfect matching that omits only that vertex, the graph is also called factor-critical. In a similar way as degree-matchings for bipartite graphs do, elementary general graphs and factor-critical graphs play an important role as building blocks in decomposition theory of general graphs.

2.6.1 Elementary general graphs

Recall that a connected graph G with a perfect matching is *elementary* if its allowed edges form a connected spanning subgraph. If every edge of G is contained in a perfect matching, then G is called 1-extendable (or matching covered). Simple examples include complete graphs with an even number of vertices (K_{2n}). Every 1-extendable graph is elementary, but the converse is not true in the general case. On the other hand, every elementary bipartite graph is 1-extendable.

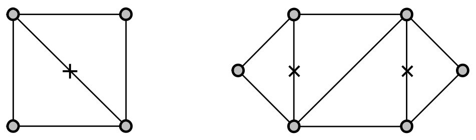


Figure 2.9: Non-matching covered elementary general graphs

A graph is *k-extendable* if each matching of size k in G can be extended to a perfect matching. The family of k-extendable graphs is quite large. For example, the (trivial) tetrahedron (K_4), the cube (Q_3), the dodecahedron and the icosahedron (but not the octahedron) are 2-extendable as well as all complete bipartite graphs K_{n_1,n_2} are such. The Petersen graph is 1-extendable, but not 2-extendable. If a graph is bicritical, it is 1-extendable. If a graph is 2-extendable, it is either bipartite or bicritical. It should be also clear that if a graph is k-extendable, it is also $(k-1)$-extendable.

A graph G is said to be *bicritical* if $G - x - y$ has a perfect matching for each pair of distinct vertices x and y in G, not necessarily adjacent. A simple example of the bicritical graph is a (trivial) dipole D_n or an odd wheel W_n. Bicritical graphs play a central role in decomposition theory of graphs in terms of their maximum matchings. The structure of bicritical graphs is far from being completely understood [212]. Obviously, a graph G is bicritical iff $G-x$ is factor-critical for every $x \in V(G)$, every bicritical graph is also matching covered (1-extendable) and no bipartite graph is bicritical.

A 3-connected bicritical graph is called a *brick*. For example, every odd wheel is a brick. Four bricks play a special role in the theory of matching covered graphs: K_4 - the complete graph on 4 vertices, $\overline{C_6}$ - the triangular prism, R_8 and the Petersen graph (see Figure 2.10). It should be pointed out that in any brick which is large enough, there exists an edge whose removal leaves a graph which is still matching covered.

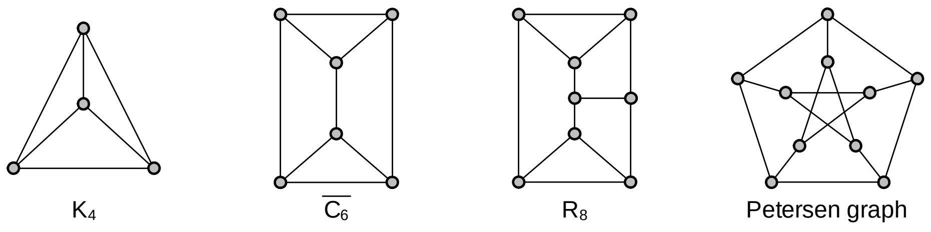


Figure 2.10: Four important matching covered graphs

Note that bicritical graphs cannot be bipartite. For bipartite graphs, we modify the definition slightly. A *brace* is a connected bipartite graph such that the deletion of any two vertices from each color class results in a bipartite graph with a perfect matching. Clearly, every *brace* is a 2-extendable bipartite graph. For example, every complete bipartite graph with the same number of vertices in its color classes is a brace.

2.6.2 Factor-critical graphs

A graph G is said to be *factor-critical* (or *hypomatchable*) if G has no perfect matching, but $G - x$ has a perfect matching for every $x \in V(G)$. Simple examples include odd-length cycle C_n and the complete graph K_n of odd order. It is easy to see that factor-critical graphs are connected, must always have an odd number of vertices and cannot be bipartite. A factor-critical graph is *trivial* if it is a singleton, and *non-trivial* if it has at least three vertices.

Testing for a graph being factor-critical is easier than the definition might indicate. Just apply the blossom shrinking algorithm. The graph is factor-critical iff the algorithm terminates with a single shrunken vertex.

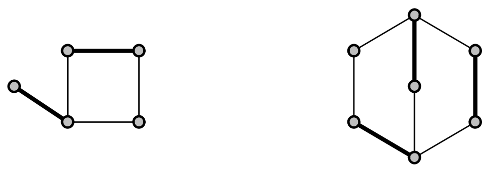


Figure 2.11: Near-perfect matching and factor-critical graph

In the above figure, the left part shows a graph with a near-perfect matching and the right part shows a factor-critical graph. Observe that every factor-critical graph has a near-perfect matching, but the converse is not true.

Lemma 2.6.1 *Let $G = (V, E)$ be a factor-critical graph. Then every edge of G belongs to some maximum matching.*

Proof Let $\{x, y\} \in E(G)$ and let M be a maximum matching of G avoiding x. Since M covers y we can assume that there exists z such that $\{y, z\} \in M$ since M is a perfect matching in $G - x$. Let $M' = M - \{y, z\} + \{x, y\}$. Then M' is the desired perfect matching of $G - z$ containing $\{x, y\}$. □

Let us summarize the following properties of factor-critical graphs:

- Every vertex is exposed by at least one maximum matching
- Every edge does not belong to at least one maximum matching
- A graph G is factor-critical iff $\nu(G-x) = \nu(G)$ for all $x \in V(G)$
- A graph G is factor-critical iff the join of G and K_1, $G + K_1$, is bicritical

In a similar way we can define critical graphs with respect to degree-matchings.

Factor-critical graphs with respect to f-matchings A graph G is said to be *critical with respect to f-matchings* if for every vertex v of G, $G + vw$ with $f(w) = 1$ has an f-factor. It is obvious that if G is critical with respect to f-matchings then G is connected, $\delta(H, f) = 1$ for any spanning f-optimal subgraph H of G and $f(V)$ is odd.

Factor-critical graphs with respect to (g, f)-matchings A graph G is said to be *critical with respect to (g, f)-matchings*, shortly *(g, f)-critical*, if for every vertex $v \in V(G)$ there exists in G a perfect (g, f^v)-matching M such that $d_M(v) = f(v) + 1$ and $g(x) \le d_M(x) \le f(x)$ for all $x \ne v$ and there exists a perfect (g^v, f)-matching M' such that $d_{M'}(v) = g(v) - 1$ and $g(x) \le d_{M'}(x) \le f(x)$ for all $x \ne v$. It can be proven that in the case of (g, f)-critical graphs, we must always have $g(x) \equiv f(x)$ (see Theorem 10.2.12c in [212]). It is also easy to show that, similarly as for the f-critical graphs, a graph G is (g, f)-critical if it is connected, $\delta(G, (g, f)) = 1$ and $g(V) = f(V)$ is odd (see Theorem 10.2.16 in [212]).

The following table summarizes properties of canonical subgraphs:

bipartite graph	$\nu(G)$	$\delta(G)$	matching
elementary	$\max\{g(V_1), g(V_2)\}..\min\{f(V_1), f(V_2)\}$	0	perfect
positive surplus	$f(V_1)$	$g(V_2) - f(V_1)$	complete
general graph			
elementary	$\lceil \frac{g(V)}{2} \rceil..\lfloor \frac{f(V)}{2} \rfloor$	0	perfect
factor-critical	$\frac{f(V)-1}{2}$	1	near-perfect

Table 2.4: Properties of canonical subgraphs

Factor-critical graphs, bipartite graphs with positive surplus, and elementary graphs play an important role in decomposition theory as building blocks in the structure theorem of Gallai and Edmonds. Clearly, the list of these canonical blocks is uniquely determined by G, i.e. it does not depend on some arbitrary choices made during the decomposition procedure.

A graph G is said to be *k-factor-critical* if the subgraph $G - S$ has a perfect matching for any subset S of $V(G)$ with $|S| = k$. Clearly, factor-critical graphs and bicritical graphs are k-factor-critical graphs when $k = 1$ and $k = 2$, respectively.

For further reading on decomposition theory we recommend Lovász & Plummer [212], Yu & Liu [309], and Akiyama & Kano [8].

2.7 Constraint Programming

In this section we summarize the main notions and associated notations that will be used in this work. We assume familiarity with the basic concepts of constraint satisfaction problems. For a thorough explanation of this area, we refer the reader to the monographs by Van Hentenryck [159], Tsang [291], Marriott & Stuckey [215], Apt [13], and/or Dechter [74].

A *constraint network* R consists of a finite set of *variables* $X = \{x_1, \ldots, x_n\}$, with respective *domains* $D = \{D_1, \ldots, D_n\}$, which represent the possible values for each variable $D_i = \{v_1, \ldots, v_k\}$, and a set of *constraints* $C = \{C_1, \ldots, C_t\}$ which link up the variables and define the set of combinations of values that are allowed. Thus, a constraint network can be viewed as a triple (X, D, C).

A constraint C_i is a relation R_i defined on a subset of variables S_i, $S_i \subseteq X$. The relation denotes the variables' simultaneous legal value assignments. The set S_i is called the *scope* of R_i. If $S_i = \{x_{i_1}, \ldots, x_{i_r}\}$, then R_i is a subset of the Cartesian product $D_{i_1} \times \ldots \times D_{i_r}$. Thus, a constraint can also be viewed as a pair $C_i = \langle S_i, R_i \rangle$. When the scope's identity is clear, we will often identify the constraint C_i with its relation R_i. Alternatively, for clarity, a constraint relation may be denoted R_{S_i}.

The *arity* of a constraint refers to the cardinality, or size, of its scope. A *unary constraint* is defined on a single variable; a *binary constraint*, on two variables; a *ternary constraint*, on three variables. A *binary constraint network* has only unary and binary constraints.

Typical tasks over constraint networks are determining whether a solution exists, finding one or all solutions, finding whether a partial instantiation can be extended to a full solution, and finding an optimal solution relative to a given cost function. Such tasks are referred to as *constraint satisfaction problems* (abbreviated as CSP), which are known in general to be intractable, even when the constraints are restricted to binary constraints.

A problem that has one or more solutions is called *satisfiable* or *consistent.* If there is no possible assignment of values to variables that satisfies all the constraints, then the network is called *unsatisfiable* or *inconsistent.*

A *filtering algorithm* for a constraint C is an algorithm that prunes the domain of each variable that C is defined on. The algorithm performs *partial filtering* if it removes only some of the inconsistent values. If the algorithm removes every inconsistent value we say that the algorithm achieves *complete filtering*.

The pruning is a task which shrinks the domain of each variable without changing the set of solutions. We say that the pruning is incomplete if it removes some inconsistent values but not every inconsistent value. A pruning is complete if the removal of any additional value from any domain would change the set of solutions.

A binary constraint is *arc consistent* if for every value in the domain of the variable, there exists a value in the domain of the other one such that the pair of values satisfied the constraint. A non-binary constraint is *hyper-arc consistent* iff for any value of a variable in its domain, there exists a value for every other variable in the domain such that the tuple satisfies the constraint. In the literature, hyper-arc consistency is also referred to as *generalized arc consistency* (often abbreviated as GAC) or *domain consistency*.

We can enforce hyper-arc consistency on a constraint by shrinking the domains of the variables. If a value does not participate in a solution to a subnetwork generated by this constraint, it will clearly not be part of a complete solution.

Briefly, hyper-arc consistency means that the domains of the variables are such that for any value in a domain, setting the variable to that value allows settings for all the other variables so that the constraint is satisfied.

Achieving hyper-arc consistency can be sometimes too costly or even intractable, and it makes sense to choose a weaker level of filtering.

Global constraints

A constraint C is often called a *global constraint*, if C is defined on an arbitrary subset of variables and when processing C as a whole gives a better result than processing any conjunction of constraints of smaller arity that is semantically equivalent to C.

A *domain variable* x is a variable ranging over a finite set of integers denoted by D_x. The minimum and maximum values of D_x are denoted by $\min(D_x)$ and $\max(D_x)$, respectively, while $|D_x|$ designates the number of elements in D_x.

Domain propagation on a constraint removes values, which cannot be extended to a pair of tuples of values satisfying the constraints from the domains of the variables until the constraint is hyper-arc consistent.

A global constraint can be characterized by various parameters:

- n : the number of variables, $|X|$
- d: the size of the largest domain, $\max_{x \in X} |D_x|$
- k: the cardinality of the union of all domains, $| \cup_{x \in X} D_x|$
- m: the sum of the cardinalities of the domains, $\sum_{x \in X} |D_x|$

A global constraint is

- bounds consistent iff:

  $\forall d_i \in \{\min(D_i), \max(D_i)\}, \exists d_j \in [\min(D_j), \max(D_j)] : (d_1, \ldots, d_n) \in C$
- range consistent iff:

  $\forall d_i \in D_i, \exists d_j \in [\min(D_j), \max(D_j)] : (d_1, \ldots, d_n) \in C$
- hyper-arc consistent iff:

  $\forall d_i \in D_i, \exists d_j \in D_j : (d_1, \ldots, d_n) \in C$

Chapter 3

Graph Traversal Algorithms

In this chapter we focus on two traditional search algorithms called *depth-first search* (DFS) and *breadth-first search* (BFS), which are integral parts of numerous algorithms used in computer science, operations research and constraint programming.

A graph G can be traversed in a depth-first manner or a breadth-first manner [67]. A depth-first search follows untraversed edges from the most recently visited vertex, whereas a breadth-first search traverses all adjacent edges of the currently visited vertex. A depth-first search explores a graph in linear time using an auxiliary stack. On the other hand, a breadth-first search explores a graph using an auxiliary queue.

The depth-first search is based on the vertex reached last, whereas the breadth-first search traverses the vertices systematically in the order in which they have been reached. The three possible colors of each vertex reflect if it is unvisited (white), visited but unexplored (gray) or completely explored (black).

It is well known that during the depth- and breadth-first search each vertex of the traversed graph can be classified into one of the following three states:

1. **White vertex** indicates that the vertex has not been discovered yet.
2. **Gray vertex** indicates the vertex already discovered but not yet finished.
3. **Black vertex** indicates the finished vertex.

With each vertex x we associate the following arrays (vectors): $parent[x]$, $level[x]$, $color[x]$, pre-order$[x]$ and post-order$[x]$.

3.1 Depth-First Search

In this section the important structures of the depth-first search and their powerful properties will be recalled.

A depth-first forest is an ordered collection of (vertex-disjoint) depth-first trees, each rooted at some vertex of G, so that every vertex in G and each tree edge of G belongs to exactly one tree.

Depth-first search uses $parent[v]$ to record the father of vertex v. We have $parent[v] = v$ iff vertex v is the root of a depth-first tree and $parent[v] = NIL$ for all white vertices. Depth-first search timestamps each vertex when its color is changed. When vertex v changes its color from white to gray, the time is recorded in $discovery[v]$. When vertex v changes its color from gray to black, the time is recorded in $finishing[v]$. These timestamps are integers between 1 and $2|V|$, since there is one discovery event and one finishing event for every vertex. Thus, the state of a vertex v is stored in a color variable as follows:

color	state	discovery time	finishing time
white	undiscovered	$discovery[v] = 0$	$finishing[v] = 0$
gray	discovered but not fully explored	$discovery[v] > 0$	$finishing[v] = 0$
black	fully explored	$discovery[v] > 0$	$finishing[v] > 0$

Table 3.1: Classification of vertices in a depth-first search

For every vertex v holds $discovery[v] < finishing[v]$. Vertex v is white before time $discovery[v]$, gray between time $discovery[v]$ and $finishing[v]$, and black thereafter. The vertex v is a leaf of a depth-first tree iff $discovery[v] + 1 = finishing[v]$ (unless the tree is trivial).

Every vertex in a depth-first tree can be reached from the root r by a unique path. Any vertex w on the unique path from r to v is called an *ancestor* of v and v is called a *descendant* of w.

Recall that in a depth-first search we can classify each edge into one of the four classes:

1. **Tree edge** is an edge in a depth-first tree. It connects a vertex to its child forming the forest.

2. **Back edge** connects a vertex to its ancestor in a depth-first tree.

3. **Forward edge** is a non-tree edge that connects a vertex to its descendant in a depth-first tree.

4. **Cross edge** connects two (unrelated) vertices in the same depth-first tree neither of which is an ancestor of the other or joins two vertices in two different depth-first trees.

Thus, the class of an edge vw can be determined in the following way:

class	v	w	discovery time	finishing time
tree edge	gray	white	$discovery[w] = 0$	$finishing[w] = 0$
back edge	gray	gray	$discovery[w] > 0$	$finishing[w] = 0$
forward edge	gray	black	$discovery[v] < discovery[w]$	$finishing[w] > 0$
cross edge	gray	black	$discovery[v] > discovery[w]$	$finishing[w] > 0$

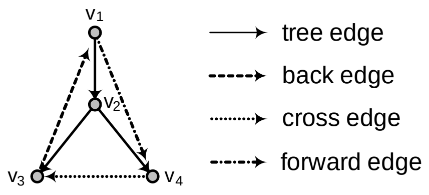
Table 3.2: Classification of edges in a depth-first search

Because the search strategy ensures that each edge is traversed only once, the total time to perform the depth-first search is $\mathcal{O}(m+n)$. Note also that the vertex u is an ancestor of the vertex v in the depth-first tree if the vertex v was discovered during the time in which the vertex u was gray. All gray vertices lie on a unique path in the depth-first tree and this path corresponds to the recursion stack of the depth-first search. The last observation is so important that it deserves to be stated as a theorem.

Theorem 3.1.1 (Gray-Path Theorem) *Vertex u is an ancestor of the current vertex v iff vertex v can be reached from u along a path consisting entirely of gray vertices.*

In our scheme we are going to use the notion of pre-order and post-order numbers. Pre-order, post-order, and in-order numberings are discussed in [2, Chapter 3] and [67, Chapter 12] (see also [285, Chapter 1]). While pre-order and post-order traversals are defined for all types of trees, in-order traversal is defined only for binary trees.

The pre-order is a numbering of the vertices from 1 to n in the order they are discovered. In the post-order the vertices are numbered from 1 to n by increasing finishing time. An example of a depth-first tree for a digraph is shown in Figure 3.1 with the traversal numbers given in the table on the right side.



vertex	pre-order	post-order
v_1	1	4
v_2	2	3
v_3	3	1
v_4	4	2

Figure 3.1: Classification of edges in a depth-first search

There are the following properties of traversal numbers:

- The pre-order number of a vertex is greater than that of its parent.
- The in-order number of a vertex in a binary tree is greater than that of its left child and smaller than that of its right child.
- The post-order number of a vertex is smaller than that of its parent.
- A specific traversal number of a vertex is always greater than that of its left sibling and smaller than that of its right sibling.

A recursive procedure for carrying out a depth-first search is shown in Algorithm 1. The algorithm processes all vertices and edges of a given digraph, identifies the edge by specifying whether it is a back edge, forward edge or cross edge, and prints its kind in order in which it has been determined (cf. [281, procedure `CLASSIFY`]).

The following properties characterize depth-first search:

- In a depth-first search of an undirected graph G every edge of G is either a tree edge or a back edge. There are no forward edges and no cross edges.

Algorithm 1 Depth-First Search of G

```
Require: Directed graph $G = (V, E)$
Ensure: Depth-First Forest
  set $time \leftarrow 1$
  for each vertex $v \in V$ do
    set $parent[v] \leftarrow NIL$
  end for
  for each vertex $r \in V$ do
    if $discovery[r] = 0$ then {white vertex}
      set $parent[r] \leftarrow r$ {make $r$ the root of $T$}
      perform DFS(r)
    end if
  end for

  procedure DFS(s)
  set $discovery[s] \leftarrow time \leftarrow time + 1$
  for every arc $(s, t) \in E$ do
    if $discovery[t] = 0$ then {white vertex}
      print (s,t) - tree edge
      set $parent[t] \leftarrow s$
      perform DFS(t)
    else if $finishing[t] = 0$ then {gray vertex}
      print (s,t) - back edge
    else {black vertex}
      if $discovery[s] < discovery[t]$ then
        print (s,t) - forward edge
      else
        print (s,t) - cross edge
      end if
    end if
  end for
  set $finishing[s] \leftarrow time \leftarrow time + 1$
```

- When a depth-first search is executed on a digraph each cross edge (x, y) has x to the right of y (there are no left-to-right cross edges).
- A directed graph D is acyclic iff a depth-first search of D yields no back edges.
- The post-order number in a depth-first search yields a reverse topological ordering for any directed acyclic graph.

3.2 Breadth-First Search

Depth-first search classifies the vertices of the traversed graph into white, gray and black vertices. To keep track of progress, breadth-first search can also be used to classify the vertices into the same three categories:

1. **White vertex** indicates a vertex that has not yet been visited.
2. **Gray vertex** indicates a vertex that has been visited but its children have not been visited yet.
3. **Black vertex** indicates a vertex that has been visited and its children have been visited as well.

White vertices are undiscovered while gray vertices are discovered but have undiscovered adjacent vertices. Black vertices are discovered and are adjacent to only other black or gray vertices.

Whenever a white vertex is discovered, it is colored gray and placed in the queue. The color gray indicates the vertex that is currently in the queue. The color black indicates the vertex that is no longer in the queue. The following table summarizes the basic features of the vertices.

color	state	condition	queue
white	undiscovered	$parent[v] = NIL$ or $level[v] = \infty$	-
gray	discovered but not fully explored	$parent[v] \neq NIL$ or $level[v] < \infty$	+
black	fully explored	$parent[v] \neq NIL$ or $level[v] < \infty$	-

Table 3.3: Classification of vertices in a breadth-first search

Recall that a depth-first search classifies each edge of a digraph into tree, back, forward and cross edge [67, Section 22.3]. A breadth-first tree can also be used to classify edges into the following categories:

1. **Tree edge** leads from a parent to its child.
2. **Back edge** leads from a descendant to its ancestor.
3. **Cross edge** connects two tree vertices that are not directly related.

Note that a breadth-first search explores no forward edges. Observe also that, although the properties of these edges remain the same, the breadth-first tree is different from the depth-first tree on the same graph. A back edge in the depth-first tree may not be a back edge in the breadth-first tree. A similar claim holds for cross edges. While cross edges in the depth-first search are only directed from right to left, in the breadth-first search they can be directed from right to left or from left to right.

The breadth-first search procedure develops a breadth-first tree with the source vertex r as its root. The parent of any other vertex in the tree is the vertex from which it was first discovered. For each vertex v the parent of v is stored in the variable $parent[v]$. Obviously, we have $parent[v] = v$ iff vertex v is the root of a breadth-first tree. Another variable, $level[v]$, computed by the search procedure, contains the number of tree edges on the path from r to v. These values are integers between 0 and the diameter[1] of the breadth-first tree.

Every vertex in a breadth-first tree can be reached from the root r by a shortest path. Any vertex w on the unique path from r to v is called an *ancestor* of v and v is called a *descendant* of w.

The determination of non-tree edges during a breadth-first search seems to be more complex than in a depth-first search. Observe that every tree edge leads to a white vertex, every back edge leads to a black vertex, whereas a cross edge can lead either to a gray or a black vertex. This observation yields the following method.

The computational effort associated with our algorithm can be split into two stages: executing a breadth-first search and then classification of edges. In the first stage of the algorithm we build a breadth-first forest consisting entirely of tree edges and we compute shortest paths from the root to every other vertex in the tree. In the second stage of the algorithm we want to determine back and cross edges. We start from the leaf and examine each outgoing/incoming edge on the path to the root. While the procedure in the first stage was growing a tree, it stored the father of every vertex v in the variable $parent[v]$. With this information we can easily find the path from leaf to the root of its tree. We detect that we have reached the root when we hit a vertex v with $parent[v] = v$. Whether it is a back edge or a cross edge, will be determined in the following way.

A breadth-first search of G corresponds to some kind of tree traversal on T. However, it is not pre-order, post-order, or even in-order traversal; instead, the traversal goes one level at a time, left to right within a level (where a level is defined simply in terms of distance from the root of the tree). This traversal is called level-order traversal. Note that the pre-order/post-order traversal is a list of the vertices in the order that they were first/last visited by a depth-first search. We can simulate both these traversals in a breadth-first search as follows.

Finding the pre-order or post-order numberings can be achieved through a depth-first search on T in $\mathcal{O}(m + n)$ time. Once pre-order and post-order numbers are assigned to the vertices, we can correctly identify the structure of a breadth-first tree, because the traversal numbers can be used to determine the various types of edges. What remains to be shown

[1]The *diameter* of a digraph D is the maximal distance between two vertices in D.

is that this method correctly identifies the edges. This will be done in the proof of the following theorem.

Theorem 3.2.1 *Let T be a breadth-first tree of a directed graph $G = (V, E)$. Let $x, y \in V$ and $(x, y) \in E$. Let traversal numbers pre-order and post-order be assigned to the vertices. These two numberings of the vertices can be used to classify the edges for the breadth-first search. With respect to the breadth-first tree T, an arc (x, y) is a*

1. *tree edge, iff $pre\text{-}order[x] < pre\text{-}order[y] \wedge post\text{-}order[x] > post\text{-}order[y]$*
2. *back edge, iff $pre\text{-}order[x] > pre\text{-}order[y] \wedge post\text{-}order[x] < post\text{-}order[y]$*
3. *right-to-left cross edge, iff $pre\text{-}order[x] > pre\text{-}order[y] \wedge post\text{-}order[x] > post\text{-}order[y]$*
4. *left-to-right cross edge, iff $pre\text{-}order[x] < pre\text{-}order[y] \wedge post\text{-}order[x] < post\text{-}order[y]$.*

Proof Let us see why this is true. We only give an intuitive argument and invite the reader for further detailed discussion. Firstly, observe that the pre-order (post-order) numbering of a vertex is higher (lower) than those of its tree descendants. This fact together with the observation that a back edge always goes from vertices to their ancestors with respect to tree edges explains the characterization of back edges. Secondly, observe that cross edges directed from right to left go from vertices with larger pre-order and post-order numbering to vertices with smaller pre-order and post-order numbering. For cross edges directed from left to right the converse holds - they lead from smaller to larger pre-order and post-order numbering. This explains the characterization of cross edges. □

The following table summarizes the basic features of edges in a breadth-first search (here, symbol $\oplus$ denotes an exclusive-or logical operation).

class	v	w	condition
tree edge	gray	white	$parent[w] = NIL$ or $level[w] = \infty$
back edge	gray	root	$parent[w] = w$
		black	$pre\text{-}order[v] > pre\text{-}order[w] \wedge post\text{-}order[v] < post\text{-}order[w]$
cross edge	gray	gray	$level[v] \leq level[w] \leq level[v] + 1$
		black	$pre\text{-}order[v] < pre\text{-}order[w] \oplus post\text{-}order[v] > post\text{-}order[w]$

Table 3.4: Classification of edges in a breadth-first search

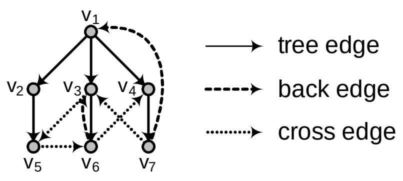
Thus, the algorithm to determine the partition of edges works in the following manner. It first performs a breadth-first search, followed next by a depth-first search on the generated shortest-path tree T. The first search discovers which vertices can reach which others and finds shortest paths from a given vertex of a graph to other vertices. The second search classifies non-tree edges. Then the condition $pre\text{-}order[v] > pre\text{-}order[w] \wedge post\text{-}order[v] < post\text{-}order[w]$ either holds or does not hold. In the former case $\{v, w\}$ is a back edge and in the latter case $\{v, w\}$ is a cross edge.

Here is an iterative procedure for carrying out a breadth-first search:

Algorithm 2 Breadth-First Search of G

```
Require: Directed graph $G = (V, E)$
Ensure: Breadth-First Forest
  for each vertex $v \in V$ do
    set $color[v] \leftarrow WHITE$
    set $parent[v] \leftarrow NIL$
  end for
  set $Q \leftarrow \emptyset$
  set $r \leftarrow$ the first vertex of $G$
  perform BFS(r)

  procedure BFS(r)
  set $color[r] \leftarrow GRAY$
  set $parent[r] \leftarrow r$ {make $r$ the root of $T$}
  set $Q \leftarrow \{r\}$
  while $Q \neq \emptyset$ do
    set $s \leftarrow Q$
    set $Q \leftarrow Q \setminus \{s\}$ {DEQUEUE}
    for every arc $(s, t) \in E$ do
      if $color[t] = WHITE$ then {white vertex}
        print (s,t) - tree edge
        set $color[t] = GRAY$
        set $parent[t] \leftarrow s$
        set $Q \leftarrow Q \cup \{t\}$ {ENQUEUE}
      else if $color[t] = GRAY$ then {gray vertex}
        print (s,t) - cross edge
      else if $t = r$ then {root}
        print (s,t) - back edge
      else {black vertex}
        print (s,t) - cross edge or back edge
      end if
    end for
    set $color[s] \leftarrow BLACK$
  end while
```

The figure below gives an example of a breadth-first tree and the resulting back and cross edges with the pre-order and post-order numbers for each vertex being given in the table on the right side of the figure. Arcs (v_6, v_3) and (v_7, v_1) are back edges, arcs (v_3, v_5) and (v_7, v_3) are right-to-left cross edges, and arcs (v_5, v_6) and (v_6, v_4) are left-to-right cross edges.



vertex	pre-order	post-order
v1	1	7
v2	2	2
v3	4	4
v4	6	6
v5	3	1
v6	5	3
v7	7	5

Figure 3.2: Classification of edges in a breadth-first search

The queue operations are **enqueue(v)** which inserts v at the rear of the queue, and **dequeue()** which removes and takes the value of the first element of the queue. Each edge is scanned at most twice. Thus the algorithm takes $\mathcal{O}(m)$ time. In the queue each vertex may appear at most once. Therefore, the algorithm requires $\mathcal{O}(n)$ space in addition to the input.

The overall running time complexity of breadth-first search is $\mathcal{O}(m + n)$. Each vertex is put on the queue exactly once and extracted exactly once; thus there are $2n$ queue operations. Each edge is considered only once and, therefore, takes $\mathcal{O}(m)$ time.

The following properties characterize breadth-first search:

- In a breadth-first search of an undirected graph G every edge of G is either a tree edge or a cross edge. There are no back edges and no forward edges.
- There are two kinds of cross edges: a left-to-right cross edge is directed from a smaller discovery number to a larger one; a right-to-left cross edge is the opposite.
- During a search vertices leave the queue in order of their distances from the root.
- Any breadth-first tree is a shortest-path tree.
- A graph is bipartite if there are no cross edges with both endpoints on the same level.

Depth-first search is used for such diverse applications as connectivity and planarity of undirected graphs, and cycle detection and topological ordering over the strongly connected components of directed graphs, whereas breadth-first search is applied to shortest path problems, network flows and the recognition of various graph classes. The following table presents some applications of graph traversal algorithms:

Problem	DFS	BFS	reference
testing bipartiteness		+	
spanning forest	+	+	
connectivity of an undirected graph	+	+	Hopcroft & Tarjan [165]
strong connectivity of a directed graph	+		Tarjan [280]
shortest paths		+	Dijkstra [76]
biconnectivity of an undirected graph	+		Tarjan [280]
triconnectivity of an undirected graph	+		Hopcroft & Tarjan [166]
computing an *st*-numbering	+		Even & Tarjan [98]
planarity testing	+		Hopcroft & Tarjan [167]
topological ordering	+		Tarjan [284]
transitive closure	+		Ioannidis et al. [171]
testing flow graph reducibility	+		Tarjan [282]
finding dominators in a directed graph	+		Tarjan [281]
maximum matching in bipartite graphs	+	+	Hopcroft & Karp [164]
maximum matching in general graphs	+	+	Micali & Vazirani [221]

Table 3.5: Some applications of graph traversal algorithms

Chapter 4

Bipartite Graphs

In this chapter we introduce a generic propagation mechanism for constraint programming. A first advantage of our pruning technique stems from the fact that it can be applied to various global constraints. We describe a filtering scheme for such a family based on the Dulmage-Mendelsohn Structure Theorem. Our method checks the feasibility in polynomial time and then ensures hyper-arc consistency in linear time. It is also applicable to any soft version of global constraint expressed in terms of a maximum matching in a bipartite graph and remains of linear complexity.

This chapter suggests using matching theory related to the Dulmage-Mendelsohn Canonical Decomposition of a bipartite graph as a framework to derive filtering algorithms for a range of constraints based on matching. It presents the canonical decomposition and shows how to apply it to global constraints whose solutions can be mapped to a matching problem.

The decomposition naturally identifies edges that are in all, some or none of the maximum matchings of the bipartite graph. As solutions of the constraint can be mapped to such matchings, the pruning of the domains can be done according to the meaning of the respective edge in the bipartite graph.

This chapter proposes a generic filtering algorithm for bipartite matching-based constraints. The algorithm is simple and intuitive. It relies on the Dulmage-Mendelsohn Canonical Decomposition of bipartite graphs. The chapter shows how such a decomposition can be used to improve the practical results of global constraints such as the well-known ALLDIFFERENT or GLOBAL_CARDINALITY constraints.

The chapter also shows that the Dulmage-Mendelsohn decomposition can be applied to the soft version of bipartite matching-based constraints. We represent the cost of violation as a deficiency of a matching and compute an optimal degree-matching in the auxiliary graph. This can be done efficiently with matching theory. We apply our method to a number of soft global constraints and obtain an efficient filtering algorithm.

We feel bipartite graphs deserve special treatment because it is the case that the majority of real-world applications of matching theory deal with graph models which are bipartite. The problem of finding a maximum matching in a bipartite graph arises in many areas of operations research and other fields, and the search for efficient algorithms for the problem has received a great deal of attention. Extensive discussion of the problem and its applications can be found in the monographs of Lawler [201, Chapter 5], Papadimitriou & Steiglitz [242, Chapter 10]. The book by Asratian, Denley, & Häggkvist [15] is an excellent reference which deals solely with bipartite graphs.

Now we survey the algorithm for finding in bipartite graphs a matching of maximal cardinality. One of the best algorithms for our problem is by John E. Hopcroft and Richard M. Karp [164]. The main idea is to find many augmenting paths in one traversal of the graph. Their algorithm is divided into phases. In each phase a maximal set of vertex-disjoint augmenting paths of shortest length is found and is used to augment the matching. The number of phases is at most $\mathcal{O}(\sqrt{n})$ (see Proposition 5.2.4 in [15]).

Table 4.1 summarizes known polynomial-time algorithms for the problem (cf. [264, Section 16.7a]). Here, ω is any real number such that any two $n \times n$ matrices can be multiplied by $\mathcal{O}(n^\omega)$ arithmetic operations (e.g. $\omega = 2.376$).

Year	Author(s)	Complexity	Strategy/Remarks
1931	König [192]	$\mathcal{O}(n \cdot m)$	Hungarian method
1955	Kuhn [198]	$\mathcal{O}(n \cdot m)$	Hungarian method
1962	Ford & Fulkerson [106]	$\mathcal{O}(n \cdot m)$	network flow
1971	Hopcroft & Karp [164]	$\mathcal{O}(\sqrt{n} \cdot m)$	shortest paths
1973	Karzanov [183]	$\mathcal{O}(\sqrt{n} \cdot m)$	based on Dinits [78]
1981	Ibarra & Moran [170]	$\tilde{\mathcal{O}}(n^\omega)$	matrix multiplication
1991	Balinski & Gonzalez [19]	$\mathcal{O}(n \cdot m)$	strong spanning trees
1991	Alt, Blum, Mehlhorn, & Paul [10]	$\mathcal{O}(n \cdot \sqrt{\frac{n \cdot m}{\log n}})$	matrix scanning
1991	Feder & Motwani [102]	$\mathcal{O}(\sqrt{n} \cdot m \log_n \frac{n^2}{m})$	graph compression
1997	Goldberg & Kennedy [141]	$\mathcal{O}(\sqrt{n} \cdot m \log_n \frac{n^2}{m})$	push-relabel method
2004	Mucha & Sankowski [224]	$\tilde{\mathcal{O}}(n^\omega)$	matrix multiplication

Table 4.1: History of algorithms for the cardinality bipartite matching problem

The approach presented in this chapter was published in [70]. The version showed here generalizes and improves the original results. We will be able to describe the algorithms more simply and the presentation of the propagation algorithm should be clearer than in the paper. There are some new results. We state the principle of duality (see Theorems 4.1.7 and 4.1.8). Section 4.5 is new, as well.

4.1 Preliminaries

Let G be a bipartite graph with degree conditions g and f, and suppose that M is any (g, f)-matching of G. A sequence of vertices and edges $P = v_0 e_1 v_1 .. e_t v_t$ is said to be an *augmenting trail* in G relative to M if

i) P is an alternating trail, i.e. $e_i \notin M$ for odd i and $e_i \in M$ for even i $(1 \leq i \leq t)$,

ii) v_0 is an exposed vertex, i.e. $d_M(v_0) < g(v_0)$,

iii) $\delta(G[M \oplus P]) < \delta(G[M])$, i.e. $|M \oplus P| = |M| + 1$.

Observe that the augmenting trails may terminate at saturated vertices (either t is odd and $g(v_t) \leq d_M(v_t) < f(v_t)$ or t is even and $g(v_t) < d_M(v_t) \leq f(v_t)$).

From the definition stated above, we can easily see the following result (see also [163]).

Theorem 4.1.1 *The (g, f)-matching M in a graph is optimal iff there is no augmenting trail relative to M.*

The following result follows easily from the assumption that G is bipartite.

Corollary 4.1.2 *The (g, f)-matching M in a bipartite graph is optimal iff there is no augmenting path relative to M.*

A perfect (g, f)-matching of G with a maximum number of edges is called a *maximum* perfect (g, f)-matching. Analogously, a perfect (g, f)-matching of G with a minimum number of edges is called a *minimum* perfect (g, f)-matching.

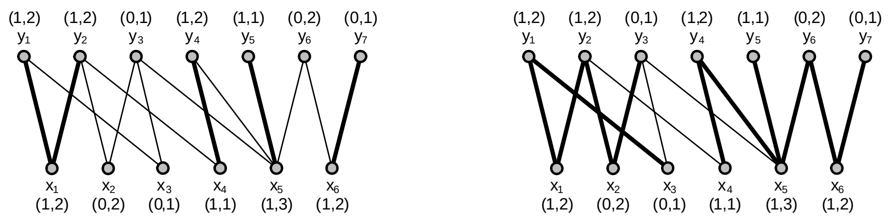


Figure 4.1: Minimum and maximum perfect (g, f)-matching

The following properties characterize graphs with perfect (g, f)-matchings.

- The size of the minimum perfect (g, f)-matching is at least $\max\{g(V_1), g(V_2)\}$.
- The size of the maximum perfect (g, f)-matching is at most $\min\{f(V_1), f(V_2)\}$.

Theorem 4.1.3 *A maximum matching can be obtained from any other maximum matching by a sequence of transfers along alternating cycles and paths of even lengths.*

Proof See Theorem 5.1.7 in [15]. □

Theorem 4.1.4 (Petersen [243]) *Every perfect f-matching can be transformed into another perfect f-matching by a series of transformations along even alternating cycles.*

Proof See Theorem 7.2.4 in [15]. □

Just as for perfect matchings and f-matchings, we can transform any perfect (g, f)-matching into any other by a series of transformations. These transformations are based on alternating paths and cycles. But first, let us introduce some new terms.

A graph G will be called *normalized* if $f(x) \neq 0$ and $g(x) \neq d(x)$. We can make any graph G with a perfect (g, f)-matching normalized in the following manner. A vertex x for which $g(x) = f(x) = 0$ will be deleted from G since obviously such a vertex cannot be a part of any perfect degree-matching; we mark all adjacent edges as forbidden. Likewise, we can also eliminate the case in which G contains a vertex of degree $d(x) = g(x) = f(x)$ since we can remove this vertex, decrease by 1 the degree conditions g and f of all incident vertices and mark all adjacent edges as mandatory. In other words, we may assume in the sequel that for all $x \in V(G)$ we always have $0 \leq g(x) < f(x) \leq d(x)$ or $0 < g(x) = f(x) < d(x)$.

Let G be a normalized bipartite graph with an arbitrary perfect (g, f)-matching M. The vertex x will be labeled as *positive* if $d_M(x) < f(x)$, or as *negative* if $g(x) < d_M(x)$. Clearly, a vertex can be labeled both positive and negative (when $g(x) < d_M(x) < f(x)$). Intuitively, the positive (or negative) vertex x means that, without changing the deficiency of G, it is possible to insert (or remove) the matched edge at x along an alternating path. Further, let us call the vertex *neutral* when $g(x) = d_M(x) = f(x)$. Then any perfect (g, f)-matching M' can be obtained from any other perfect (g, f)-matching M by a sequence of transfers along alternating paths or cycles of the following types:

1. an alternating path of even length, called a *negative alternating path*, connecting two distinct vertices, starting from a negative vertex and a matched edge and terminating with a free edge and a positive vertex;

2. an alternating path of even length, called a *positive alternating path*, connecting two distinct vertices, starting from a positive vertex and a free edge and terminating with a matched edge and a negative vertex;

3. an alternating path of odd length, called a *decreasing alternating path*, between two distinct negative vertices and matched edges (in this case the deficiency of the graph remains the same but the number of matched edges decreases);

4. an alternating path of odd length, called an *increasing alternating path*, between two distinct positive vertices and free edges (in this case the deficiency of the graph remains the same but the number of matched edges increases);

5. an alternating cycle (of even length).

Figure 4.2 illustrates alternating paths of various types (negative and positive endpoints are designated by [−] and [+], respectively):

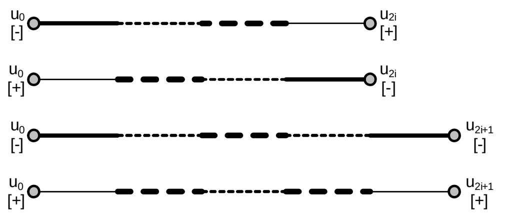


Figure 4.2: Alternating paths defined for perfect (g,f)-matchings

Observe that each transfer along any alternating path does not change the degrees of internal vertices but only changes the degrees of endpoints by one.

Next, we investigate the properties of matching transformation in bipartite graphs. That is, given two matchings, we can start with one and transform to another one through the operation of symmetric difference. Combining theorems with properties we can deduce the following results.

Theorem 4.1.5 *A perfect (g,f)-matching M of a bipartite graph G can be obtained from any other perfect (g, f)-matching M' by a sequence of transfers along alternating cycles of even length and alternating paths of even and odd length.*

Proof The proof of this theorem is similar to that of Theorem 7.2.4 in [15]. □

Theorem 4.1.6 *A perfect (g, f)-matching M in a bipartite graph G is minimum iff there exists in G no decreasing alternating path relative to M; it is maximum iff G has no increasing alternating path relative to M.*

Principle of duality

Now we introduce the dual structures in the (g, f)-matching problem needed by the framework of this chapter (see Section 4.3.4). We consider a global degree-matching problem thus also the dual is defined globally. Some first steps in this direction have been made in [234],[236] and [294]. It is convenient to define other functions g' and f' on $V(G)$ by the rule $g'(x) = d_G(x) - g(x)$ and $f'(x) = d_G(x) - f(x)$. Then all associated properties of (g, f)-matchings also hold with f' replacing f and g' replacing g. The correspondence $(g, f) \to (f', g')$ imposes a kind of duality in matching theory.

The *dual factor* F' of F has the same vertices as F and the edge $e \in E(G)$ is in F' iff e is not in F. Thus the dual of F is obtained by interchanging all occurrences of the matched edges with the free edges. We have already encountered other dual concepts, such as duality theory in linear programming [242, Chapter 3], matroid duality [201, Chapters 7 and 8], directional duality [151, Chapter 2], and so on. The latter is a classical result in the theory of binary relations.

For each theorem about factors (degree-matchings), there is a corresponding theorem obtained by replacing every concept by its dual. The concept of dual factors provides a duality in matching theory. The dual theorem follows immediately after applying the principle of duality. We now illustrate how this principle generates new results. Here are some very simple ones

Theorem 4.1.7 *Let g and f be integer-valued functions on the vertex set $V(G)$ of a graph G and $0 \leq g(x) \leq f(x) \leq d(x)$ for each vertex $x \in V$. Then G has a perfect (g, f)-matching iff it has a perfect (f', g')-matching, whose edges are those edges of G not belonging to the perfect (g, f)-matching. In particular, an edge is mandatory concerning perfect (g, f)-matching iff this edge is forbidden concerning perfect (f', g')-matching, and vice versa.*

Proof Let M be a perfect (g, f)-matching in G. Then $g(x) \leq d_M(x) \leq f(x)$ for every vertex $x \in V(G)$. From the definition of the dual matching M' holds $d_{M'}(x) = d_G(x) - d_M(x)$.

This being given we have $f'(x) = d_G(x) - f(x) \leq d_G(x) - d_M(x) \leq d_G(x) - g(x) = g'(x)$. Thus, every dual matching M' is a perfect (f', g')-matching. □

Theorem 4.1.8 *The problem of finding a perfect (g, f)-matching in G with the minimum number of edges is equivalent to the problem of finding a perfect (f', g')-matching in G with the maximum number of edges, and vice versa.*

Proof According to Theorem 4.1.6 a perfect (g, f)-matching is minimum/maximum iff there is no decreasing/increasing alternating path in G. Because of this fact we will prove our theorem by contradiction. Assume that M is a minimum perfect (g, f)-matching but its dual M' does not have the maximum number of edges. This means that there exists in G an increasing alternating path with respect to M'. Since this path will increase the number of edges in M', then, according to Theorem 4.1.7, there exists in G the dual perfect (g, f)-matching with the smaller number of edges than M. But this is a contradiction of our assumption that M has the minimum number of edges. This contradiction proves our theorem. □

Observe that the last result is a *minimax theorem*, a theorem which states that the minimum of one thing is equal to the maximum of another. Every minimax theorem in graph theory can be derived as a direct consequence of the duality theorem in linear programming (see [212, Theorem 7A.2]). There are some classic minimax theorems known, such as, for example, Menger's Theorem (see [212, Theorem 2.4.1]), Dilworth's Chain-Decomposition Theorem (see Theorem 1.4.8 and Theorem 1.4.12 in [212]), Max-Flow Min-Cut Theorem (see [212, Theorem 2.1.4]), Egerváry's Theorem (see [212, Theorem 7.1.4]). Recall that the dual of the maximum matching problem in bipartite graphs is the minimum vertex cover problem (see König's Minimax Theorem [212, Theorem 1.1.1]).

4.2 Dulmage-Mendelsohn Canonical Decomposition

This section is devoted to a comprehensive survey of the Dulmage-Mendelsohn Canonical Decomposition, a unique decomposition of a bipartite graph with respect to maximum matchings. Later in this work we will see that it has an attractive application in efficient filtering algorithms. A standard reference for matching theory, with emphasis on structures rather than algorithms, is [212].

Let G be a bipartite graph with bipartition (V_1, V_2). Furthermore, let M be an initial maximum matching in G. Denote by $\langle A_1, B_1, C_1 \rangle$ the decomposition of V_1 (relative to M) into three disjoint subsets A_1, B_1, and C_1, where

$$B_1 = \{x \in V_1 | x \text{ is reachable from at least one exposed vertex in } V_1 \text{ via alternating paths}\},$$
$$A_1 = \{x \in V_1 | x \text{ is reachable from at least one exposed vertex in } V_2 \text{ via alternating paths}\},$$
$$C_1 = V_1 \setminus (A_1 \cup B_1).$$

Symmetrically, there exists a similar decomposition $\langle A_2, B_2, C_2 \rangle$ of V_2, the other color class of the bipartite graph, where

$B_2 = \{y \in V_2 | y$ is reachable from at least one exposed vertex in V_2 via alternating paths$\}$,

$A_2 = \{y \in V_2 | y$ is reachable from at least one exposed vertex in V_1 via alternating paths$\}$,

$C_2 = V_2 \setminus (A_2 \cup B_2)$.

4.2.1 Bipartite Graphs with Imperfect Maximum Matchings

The structure of bipartite graphs was considered by Andrew L. Dulmage and Nathan S. Mendelsohn [83–86] before the results were derived for general graphs by Tibor Gallai and Jack Edmonds [90]. The authors published a series of papers in which they worked out the canonical decomposition for bipartite graphs in terms of maximum matchings and minimum vertex covers[1]. Some of their results are summarized in the next theorem.

Theorem 4.2.1 (Dulmage-Mendelsohn Structure Theorem) *Let $G = (V_1 \cup V_2, E)$ be a bipartite graph and let $\langle A_1, B_1, C_1\rangle$ and $\langle A_2, B_2, C_2\rangle$ are defined as above. Then*

1. *$B = B_1 \cup B_2$ is an independent set of vertices,*
2. *The subgraph $G[C_1 \cup C_2]$ has a perfect matching, and hence $|C_1| = |C_2|$,*
3. *$\Gamma(B_1) = A_2$ and $\Gamma(B_2) = A_1$,*
4. *Every maximum matching of G consists of a perfect matching of $G[C_1 \cup C_2]$, a complete matching from A_1 into B_2 and a complete matching from A_2 into B_1,*
5. *The subgraphs induced by $A_1 \cup B_2$ and $A_2 \cup B_1$ have positive surplus when viewed from A_1 and A_2, respectively,*
6. *The size of a maximum matching is $\frac{1}{2}(|V| - |B| + |A|)$.*

Proof See Theorem 3.2.4 in [212]. □

We call the decomposition given in this theorem the *Dulmage-Mendelsohn Canonical Decomposition* of the bipartite graph. An example of such decomposition is illustrated in Figure 4.3 (this and the next image is taken from [246]).

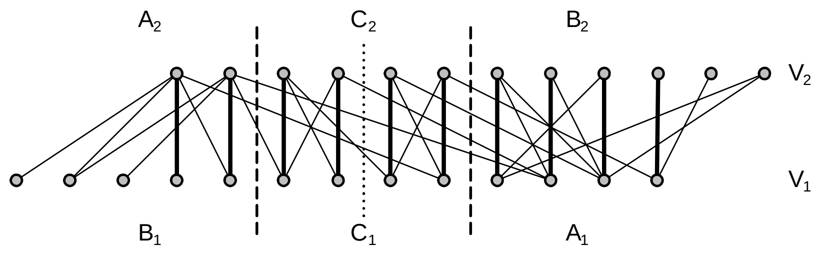


Figure 4.3: Dulmage-Mendelsohn decomposition of a bigraph with a maximum matching

The reader is invited to verify the sets in terms of some other maximum matching and to check that, by Theorem 4.2.1, the decomposition has the following properties:

[1]Recall that in bipartite graphs there is always a vertex cover and a matching of the same cardinality.

- There is no edge between B_1 and C_2, between B_2 and C_1, and between B_1 and B_2.
- Edges between C_1 and A_2, between C_2 and A_1, and between A_1 and A_2 never belong to a maximum matching.
- Every edge incident with a vertex of B_1 or B_2 lies in some maximum matching of G (see Lemma 2.5.7).
- Vertices of A_1 and A_2 are saturated by every maximum matching (see Lemma 2.5.7).
- Vertices of $B_1 \cup B_2$ are exposed by at least one maximum matching (see Lemma 2.5.7).

With these concepts every bipartite graph has a unique decomposition into three induced subgraphs (not necessarily connected): $G[C_1 \cup C_2]$, $G[A_1 \cup B_2]$ and $G[A_2 \cup B_1]$ together, perhaps, with some additional edges between A_1 and A_2, A_1 and C_2, and between A_2 and C_1, joining these three subgraphs in certain ways. These additional edges do not occur in any maximum matching. Two of these three subgraphs, $G[A_1 \cup B_2]$ and $G[A_2 \cup B_1]$, are bipartite graphs with positive surplus. Graph $G[C_1 \cup C_2]$, on the other hand, is a subgraph with a perfect matching. It has an interesting further decomposition which will be discussed in the next subsection.

The subgraph $G[A_1 \cup B_2]$ with vertex sets A_1 and B_2 represents the underdetermined part of the decomposition, the subgraph $G[C_1 \cup C_2]$ with vertex sets C_1 and C_2 represents the determined part, and the subgraph $G[A_2 \cup B_1]$ with vertex sets A_2 and B_1 represents the overdetermined part.

4.2.2 Bipartite Graphs with Perfect Matchings

Recall that a bipartite graph G is elementary if every edge of G is contained in at least one perfect matching of G. There exists a decomposition of bipartite graphs with a perfect matching into elementary bipartite subgraphs.

Let us now consider the bipartite graph G with a perfect matching M. The following result due to König [191] and Dulmage & Mendelsohn [83,84] gives the canonical decomposition of any bipartite graph with a perfect matching into elementary bipartite subgraphs.

Theorem 4.2.2 (Decomposition into elementary bipartite subgraphs) *Let G be a bipartite graph with a perfect matching M. Denote by S_i and T_i $(i = 1, \dots, k)$ the set of vertices from V_1 and V_2, respectively, joined by a matched edge or reached via alternating cycles (of even length). Then:*

1. *Each $H_i = G[S_i \cup T_i]$ is an elementary subgraph with a perfect matching, so $|S_i| = |T_i|$,*
2. *$|\Gamma(X)| > |X|$ for every non-empty proper subset X of S_i or T_i,*
3. *The bipartite graph obtained by shrinking every S_i and T_i to a vertex and replacing each set of parallel edges by a single edge has a unique perfect matching of size k,*

4. *Elementary subgraphs $H_i = (S_i \cup T_i, E_i)$ can be labeled in such a way that every edge in G from a subgraph H_i to a subgraph H_j with $i < j$ has one endpoint in T_i and the other one in S_j.*

Proof See Theorem 4.1.1, Lemma 4.3.1 and Lemma 4.3.2 in [212]. □

Of course, this decomposition is unique and does not depend on the initial perfect matching. An example of such decomposition is shown in Figure 4.4.

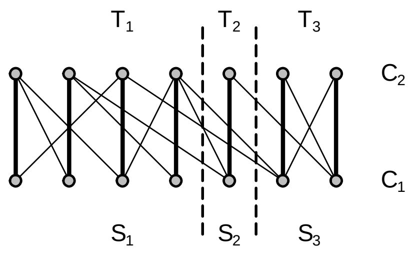


Figure 4.4: Dulmage-Mendelsohn decomposition of a bigraph with a perfect matching

In other words, from the previous theorem it follows that the decomposition has the following properties:

- Edges between H_i and H_j with $i \neq j$ never belong to a perfect matching
- if $|H_i| \geq 4$ then every edge of H_i does not belong to at least one perfect matching
- if $H_i = K_2$ then the only edge of H_i is contained in every perfect matching

By this theorem every bipartite graph with a perfect matching has a unique decomposition into elementary bipartite subgraphs together with some additional edges between them joining these subgraphs in certain ways. These additional edges do not occur in any perfect matching.

4.2.3 Bipartite Subgraphs with prescribed Degrees

Let G be a graph with degree conditions g and f. Recall that a perfect (g, f)-matching in G is a subset of $E(G)$ such that for each vertex x at least $g(x)$ and at most $f(x)$ edges incident to x are in the subset. An f-matching is called perfect if, for each vertex x, there are exactly $f(x)$ edges incident to x. An optimal (g, f)-matching is a matching with the smallest deficiency.

The existence of an optimal (g, f)-matching is guaranteed by Theorem 10.2.5 in [212]. The key is to find a $(0, f)$-factor F minimizing deficiency. This can be done in a way similar to the algorithm due to Hopcroft & Karp. If for a vertex x it holds that $d_F(x) < g(x)$, then the graph has no perfect (g, f)-matching. Otherwise, F itself is a (g, f)-factor of G. Hell & Kirkpatrick [158] have shown that this method can be realized in $\mathcal{O}(\sqrt{g(V)} \cdot m)$ operations.

An algorithm to compute an optimal f-matching was given by Quimper et al. in [253]. It is a generalization of the algorithm due to Hopcroft & Karp with a similar $\mathcal{O}(\sqrt{f(V)} \cdot m)$ time complexity.

Although the Dulmage-Mendelsohn decomposition is defined only for bipartite graphs with a maximum matching, we are able to extend it to a class of bipartite graphs with an optimal (g, f)-matching. We now state the generalized version of the Dulmage-Mendelsohn canonical decomposition. Consider the following decomposition of V_1 into three disjoint subsets:

$B_1 = \{$vertices in V_1 reachable by alternating trails from some exposed vertex in $V_1\}$,

$A_1 = \{$vertices in V_1 reachable by alternating trails from some exposed vertex in $V_2\}$,

$C_1 = V_1 \setminus (A_1 \cup B_1)$.

The decomposition $\langle A_2, B_2, C_2\rangle$ of V_2 can be similarly defined. The decomposition is unique and independent of the initial optimal (g, f)-matching M. In Figure 4.5 we illustrate the decomposition.

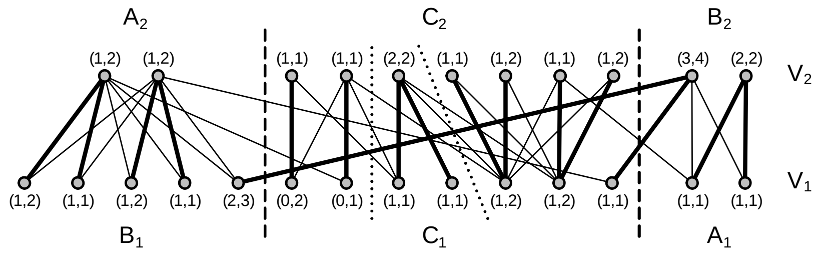


Figure 4.5: Dulmage-Mendelsohn decomposition of a bigraph with an optimal (g,f)-matching

The reader is asked to check the decomposition of the graph G in the figure. Note that $\delta(G, (g, f)) = 2$. According to observations we can formulate the following result.

Theorem 4.2.3 *Let G be any bipartite graph with degree conditions g and f. Then, there is a partition $\langle A_1, B_1, C_1\rangle$ and a partition $\langle A_2, B_2, C_2\rangle$ so that the following statements hold:*

1. *If $x \in A_1 \cup A_2$, then $d_M(x) \geq f(x)$ for every optimal (g, f)-matching M.*
2. *If $x \in B_1 \cup B_2$, then $d_M(x) \leq g(x)$ for every optimal (g, f)-matching M.*
3. *No edge connecting A_1 to C_2, A_2 to C_1, or A_1 to A_2 belongs to any optimal (g, f)-matching.*
4. *Every edge connecting B_1 to C_2, B_2 to C_1, or B_1 to B_2 belongs to every optimal (g, f)-matching.*

Proof See Theorem 10.2.12 and Theorem 10.2.13 in [212]. □

We are now ready to formulate the main structure theorem, which, together with the previous theorem, establishes the canonical decomposition of bipartite graphs with optimal (g, f)-matchings. Let us define the functions $\bar{g}$ and $\bar{f}$ by $\bar{g}(x) = g(x) \dot{-} |\nabla(x, B)|$, $\bar{f}(x) = f(x) \dot{-} |\nabla(x, B)|$ for every $x \in C_1 \cup C_2$.

Theorem 4.2.4 *Let $A = A_1 \cup A_2$, $B = B_1 \cup B_2$ and $C = C_1 \cup C_2$, whereby $\langle A_1, B_1, C_1\rangle$ and $\langle A_2, B_2, C_2\rangle$ are the three sets of the Dulmage-Mendelsohn decomposition for bipartite graph G with an optimal (g, f)-matching M. Then this decomposition has the following properties:*

1. *The subgraph $G[C_1 \cup C_2]$ (not necessarily connected) has a perfect $(\bar{g}, \bar{f})$-matching.*
2. *The subgraphs $G[A_1 \cup B_2]$ and $G[A_2 \cup B_1]$ have positive surplus (as viewed from A_1 and A_2, respectively).*
3. *Every optimal (g, f)-matching of $G[A_1 \cup B_2]$, or $G[A_2 \cup B_1]$, saturates A_1, or A_2, respectively.*
4. *Every optimal (g, f)-matching of G splits into a perfect $(\bar{g}, \bar{f})$-matching of $G[C_1 \cup C_2]$ and complete (g, f)-matchings of $G[A_1 \cup B_2]$ and $G[A_2 \cup B_1]$.*
5. *Subgraph $G[C_1 \cup C_2]$ has further decomposition into elementary bipartite subgraphs with respect to (g, f)-matchings. Edges between these subgraphs never belong to any perfect (g, f)-matching.*

Proof See Theorem 10.2.18 in [212]. □

4.3 Partition of vertices and edges

In the following, we assume that one maximum matching is initially established. Suppose we have a maximum matching M in a bipartite graph $G = (V_1 \cup V_2, E)$. The overall aim is to specify some procedures that allow us to efficiently make the following partition of the vertex set:

- the set of vertices exposed by at least one maximum matching (allowed vertices)
- the set of vertices saturated by every maximum matching (mandatory vertices)

and the following partition of the edge set:

- the set of edges belonging to no maximum matching (forbidden edges)
- the set of edges not belonging to at least one maximum matching (allowed edges)
- the set of edges belonging to all the maximum matchings (mandatory edges)

Any forbidden edge e can be deleted from graph G without affecting the deficiency of G. Similarly, we can delete any mandatory edge $e = \{v, w\}$ together with its endpoints v and w and all edges incident with them without changing the deficiency.

4.3.1 Alternating breadth-first search

The canonical decomposition $\langle A_1, B_1, C_1\rangle$ and $\langle A_2, B_2, C_2\rangle$, and thus the partition of vertices, can be determined by performing a special kind of breadth-first search, called an *alternating breadth-first search*, starting from exposed vertices of G. The search is restricted to vertices reachable via alternating paths. Then B_1 and A_2 are given by the vertices of G appearing in *breadth-first trees* rooted at exposed vertices of V_1. Since we only consider alternating paths in the traversal, we will refer to these trees as an *alternating breadth-first forest*. The set B_1 is the union of vertices occurring on the even levels of the forest, while the set A_2 is the union of vertices occurring on the odd levels of the forest. The sets B_2 and A_1 can be determined in the same way as B_1 and A_2, by forming the alternating breadth-first forest from the set of exposed vertices in V_2. It is obvious that the alternating breadth-first search only operates along alternating paths.

The vertices in $B_1 \cup B_2$ are allowed; all the remaining vertices are mandatory. The edges between C_1 and A_2, between C_2 and A_1, and between A_1 and A_2 are forbidden. The edges between C_1 and B_2, between C_2 and B_1, and between B_1 and B_2, if any, are mandatory (this situation occurs only in the case of the degree-matching). All the edges traversed by the alternating breadth-first search are allowed. The subgraphs $G_U = G[A_1 \cup B_2]$ and $G_O = G[A_2 \cup B_1]$ have as many connected components as the number of trees in the alternating breadth-first forest and we have made the first decomposition step in the graph. We now have to consider the decomposition of $G[C_1 \cup C_2]$.

4.3.2 Alternating depth-first search for perfect matchings

Now the question is how to have an efficient way to find the decomposition of edges in $G[C_1 \cup C_2]$. The following theorem gives an answer to this question (cf. [212, Exercise 4.1.5]).

Theorem 4.3.1 *Consider a bipartite graph $G = (V_1 \cup V_2, E)$ with an initial perfect matching M. Let D be the directed graph obtained from G by replacing each matched edge by an arc leading from V_1 to V_2 and by orienting all other edges in the opposite direction. Then the strongly connected components of D are exactly the elementary bipartite subgraphs of G.*

Proof See Theorem 5.1.1 in [309]. □

We can use the auxiliary digraph to decide whether a given edge e in G can belong to some perfect matching in G. In order to realize it we will need the following criterion:

Lemma 4.3.2 *Let G be a graph and M be a perfect matching in G. Let $e \notin M$ be an edge in G. Then $e = \{x_i, y_j\}$ can belong to some perfect matching in G iff x_i and y_j lie in the same strongly connected component of the auxiliary digraph D.*

Proof See Proposition 1 in [255]. □

This observation yields the following simple algorithm to compute the partition of edges in a graph with an initial perfect matching. We first compute the strongly connected components in the auxiliary digraph D. Edges whose endpoints are both in the same strongly connected component belong to an even alternating cycle in G, and thus are allowed. Edges that are not allowed but belong to M are mandatory. All other remaining edges, with endpoints in two distinct strongly connected components, as non-admissible and not belonging to M, are forbidden.

A high-level description of the procedure that detects partition of vertices and edges is described below (cf. Algorithm 2 in [255]):

Algorithm 3 Dulmage-Mendelsohn Canonical Decomposition of G

```
Require: Bipartite graph $G = (V_1 \cup V_2, E)$
Ensure: Partition of vertices and edges
  Compute the maximum matching $M$ of $G$
  Mark all vertices as MANDATORY and all edges as FORBIDDEN
  Perform a breadth-first search starting from exposed vertices
  Mark all traversed edges as ALLOWED
  Mark all visited vertices on the even level as ALLOWED
  Find partition of $V_1$ into the sets $A_1$, $B_1$, and $C_1$
  Find similarly partition of $V_2$ into the sets $A_2$, $B_2$, and $C_2$
  The Dulmage-Mendelsohn Decomposition is given by $\langle A_1, B_1, C_1 \rangle$ and $\langle A_2, B_2, C_2 \rangle$
  Let $G_U = G[A_1 \cup B_2]$ and $G_O = G[A_2 \cup B_1]$
  Calculate $G_W = G[C_1 \cup C_2] = G - G_U - G_O$
  Construct the auxiliary directed graph $D$ from $G_W$ as described in the text
  Compute all strongly connected components $SCC_1, \ldots, SCC_k$ of $D$
  Find the partition of edges as explained in the text
  Remove all forbidden edges from $G$
```

We now consider the complexity of this procedure. Before the execution of the first step we have to construct a maximum matching M in the graph G. This takes $\mathcal{O}(\sqrt{n} \cdot m)$ time and $\mathcal{O}(m \cdot n)$ memory by using the algorithm of Hopcroft & Karp [164]. The other steps can be done in a total time of $\mathcal{O}(m + n)$, since the computation of the partition of vertices and edges is made by a classical breadth-first/depth-first search in linear time using the algorithm by Tarjan [280] (see also [67, Chapter 22]). Since the complexity of identifying the strongly connected components can be done in time $\mathcal{O}(m + n)$, we conclude that, with an initial maximum matching M, we can establish the partition of edges in linear time.

Theorem 4.3.3 *Let G be a bipartite graph with an initial maximum matching M. Then the Dulmage-Mendelsohn Canonical Decomposition of G can be computed in linear time.*

Observe that if we need to split the graph G (based on the bipartite subgraph of C_1 and C_2) into two classes: admissible and non-admissible edges, then we have only to construct a directed auxiliary graph D obtained from G as follows. The directed graph consists of vertices from C_1. From two vertices x_i and x_j in C_1 there is a directed edge x_i to x_j in this new directed graph iff there is an edge from x_i to the $mate(x_j)$ in C_2. In order to obtain the desired decomposition apply now thereon an algorithm for finding strongly connected components. The trivial components represent the then mandatory edges. Note also that, with a small modification, our procedure works on the bipartite graphs with parallel edges.

We have seen that by performing three steps:

1. Construct the auxiliary directed graph D,

2. Compute the strongly connected components of D,

3. Determine the partition of edges,

it is possible to find the decomposition of the graph. It is not so obvious that this can also be done in only one step. In the rest of this subsection we present a linear time algorithm based on the depth-first search that maintains a representation of alternating cycles. This approach is important for speeding up some algorithms on graph G (e.g. for the Gallai-Edmonds Canonical Decomposition).

Another algorithm

We hereby present a procedure that allows us to classify the edges of several alternating cycles with only one search phase and to simply derive the decomposition of the graph.

In order to give the faster method for computing the decomposition into elementary bipartite subgraphs, and thus the partition of edges, we need to construct a special kind of depth-first forest called an *alternating depth-first forest*. Let $G = (V_1 \cup V_2, E)$ be an arbitrary bipartite graph with an initial perfect matching M. The routine will traverse the bipartite graph in the following manner. If the level is odd, all adjacent edges (except matched ones) leave the vertex, whereas when the level is even, only a matched edge leaves the vertex. Thus, an alternating depth-first search, similar to an alternating breadth-first search, simulates the traversing on the directed graph and constructs layers that alternately use matched and free edges. The difference is that the alternating breadth-first search starts from the exposed vertex and all the even levels are governments by the current matching M but the alternating depth-first search starts from the saturated vertex and all matched edges are responsible for odd levels. For odd levels, the vertices are simply given by the mates of the vertices from the previous even level. For even levels, the next vertex is some unvisited neighbor of a vertex from the previous odd level. Therefore, every vertex on the even level must have its mate on the odd level. This implies that the alternating depth-first tree must have leaves on odd levels. It should be obvious that an alternating depth-first search maintains alternating cycles.

An alternating depth-first forest is an ordered collection of (vertex-disjoint) depth-first trees, each rooted at some vertex of G with a matched edge from M, so that every vertex in G and each matched edge from M belongs to exactly one tree.

We now give the following result, which is a direct consequence of Lemma 4.3.2, Corollary 2.4.3 and Theorem 4.2.2 (see also [25]).

Corollary 4.3.4 *Let M be a perfect matching in a bipartite graph G and let $C_1, \ldots, C_k$ be the cycles (relative to M) determined by an alternating depth-first search on G. Then*

- *Edges included in any alternating cycle are allowed.*
- *Matched edges included in no alternating cycle are mandatory.*
- *Free edges included in no alternating cycle are forbidden.*

In order to derive the algorithm we first introduce some notations. We call a cycle *completed* if all its vertices are black and we call it *active* if this is not the case. Among the set of vertices that currently belong to an alternating cycle C_i, let c_i be the vertex whose discovery timestamp is minimum. Then the vertices of C_i form a subtree of an alternating depth-first tree. Vertex c_i is called the *core* of the cycle C_i. A *nested cycle* is a cycle whose core belongs to another alternating cycle. A cycle can be nested in more than one cycle or may involve several cycles. These cycles are merged into one, with the core closest to the root of the tree. Clearly, all vertices reachable from vertices in a completed cycle are black. In particular, a cycle is active until its core is gray.

We now describe the algorithm in more detail. If $\{v, w\}$ is a tree edge we simply initiate a recursive call of depth-first search. If $\{v, w\}$ is a non-tree edge and w is black or belongs to a completed cycle then no action is required to maintain the alternating cycles. If $\{v, w\}$ is a non-tree edge and w is gray (this is the case with back edges) or belongs to an active cycle then an alternating cycle is detected. For simplicity of routine, we allow (alternating) cycles to have length 0 so that each vertex will be marked (at start) as belonging to a (trivial) cycle.

We use an idea of *contracting* cycles obtained by replacing the vertices defining cycle with its core. If the alternating depth-first search encounters a vertex belonging to a cycle, then the search jumps to the core of the cycle. If the search finds a new alternating cycle then this cycle is embedded to the contracted cycle. Clearly, the tree T' formed from T by collapsing the cycle C contains an alternating cycle iff T does, since bipartite graphs have only cycles of even length (see Theorem 2.5.1) and every traversed edge connects two vertices on levels with different parities.

We now prove the correctness of our algorithm by the following results.

Lemma 4.3.5 *If an edge is on a closed alternating trail then it is on an alternating cycle.*

Proof Let $C = v_0e_1v_1 \ldots v_{t-1}e_tv_0$ be the shortest closed alternating trail. Then C is an alternating cycle. In order to see it, suppose conversely that there exists $1 \leq i < j < t$ such that $v_i = v_j$. Since bipartite graphs have only cycles of even length (see Theorem 2.5.1) then $C' = v_0e_1v_1 \ldots e_iv_ie_{j+1} \ldots v_{t-1}e_tv_0$ is also a closed alternating trail, but shorter than C. This assumption leads to a contradiction. Thus, C is an alternating cycle. □

Theorem 4.3.6 *Alternating cycles are detected by back edges.*

Proof Assume that there is a back edge $\{v, w\}$. Then vertex w is an ancestor of vertex v in the depth-first tree. According to the Gray-Path Theorem there exists a path from w to v. Coupled with the back edge $\{v, w\}$ this defines an alternating cycle. $\square$

Theorem 4.3.7 *Alternating cycles are detected by cross (or forward) edges leading to an active cycle.*

Proof Suppose that there is a cross (or forward) edge $\{v, w\}$, whereby w belongs to any active cycle (with a gray core c). Then, according to the Gray-Path Theorem, there exists a path from c to v. Since vertex w belongs to a cycle with the core c, there exists a path from w to c. Thus, the edge $\{v, w\}$ completes an alternating cycle. $\square$

Let us summarize some important properties of an alternating depth-first search:

- Every vertex on the even level has its mate on the odd level.
- Trees have leaves on odd levels.
- Every matched edge belongs to a tree.
- Cycles are detected by back edges and (allowed) cross edges.
- Every non-tree edge (forward or cross) is allowed when it leads to the collapsed cycle whose core is not yet finished (gray vertex); otherwise it is forbidden.

An efficient implementation of the algorithm is dominated by the time to keep core of cycles formed by the contraction operation. Any data structure for disjoint set merging can be used for this purpose (see, for example, [67, Chapter 21]). In fact, the incremental tree set union algorithm of Gabow & Tarjan presented in [121] can be used.

Recall that the disjoint set is a data structure that supports the following operations:

makeset(x) – creates a new set whose leader member is x,

unionset(x,y) – creates the union of the sets containing two elements x and y,

findset(x) – returns the leader element of the unique set containing x.

Table 4.2 presents history of algorithms for the disjoint set union problem[2] (here, m denotes the number of operations and n denotes the number of elements).

Note that the same data structure is used in the blossom shrinking algorithm for cardinality maximum matching of general graphs to keep tracks of blossoms. From an interpretation point of view the base of a blossom is equivalent to the core of an alternating cycle. The paper [125] surveys data structures and algorithms, which have been proposed in the literature to solve the set union problem and some of its variants.

Because our search strategy guarantees that each edge is encountered at most twice (the second time when cycles are being contracted) this gives the linear-time algorithm of complexity $\mathcal{O}(m + n)$ to find the partition of edges.

[2]The *disjoint set union problem* is sometimes called the *equivalence problem*, because of the fact that the sets define a partition of the elements into equivalence classes.

Algorithm 4 Alternating Depth-First Search of G

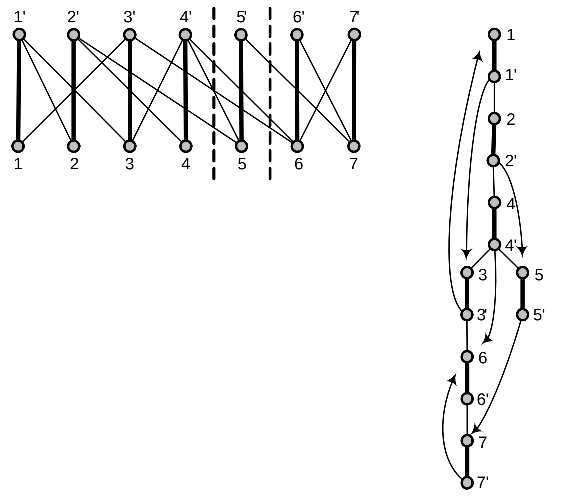
```
Require: Bipartite graph $G = (V_1 \cup V_2, E)$ with a perfect matching $M$
Ensure: Partition of edges into MANDATORY, ALLOWED and FORBIDDEN
  set $time \leftarrow 1$
  mark all edges in $G$ as FORBIDDEN
  mark all edges in $M$ as MANDATORY
  for each vertex $v \in V$ do
    set $parent[v] \leftarrow NIL$
    $MAKESET(v)$ {trivial cycle}
  end for
  for each vertex $s \in V_1$ do
    if $discovery[s] = 0$ then {white vertex}
      set $parent[s] \leftarrow s$ {make $s$ the root of $T$}
      set $discovery[s] \leftarrow time \leftarrow time + 1$
      set $t \leftarrow mate(s)$
      set $parent[t] \leftarrow s$
      perform AlternatingDFS(t)
    end if
  end for

  procedure AlternatingDFS(v)
  set $discovery[v] \leftarrow time \leftarrow time + 1$
  for every edge $\{v, w\} \notin M$ do
    if $discovery[w] = 0$ then {tree edge}
      set $parent[w] \leftarrow v$
      make $t = mate(w)$ the next child of $w$
      set $discovery[w] \leftarrow time \leftarrow time + 1$
      set $parent[t] \leftarrow w$
      perform AlternatingDFS(t)
    else if $finishing[FINDSET(w)] = 0$ then {gray vertex}
      mark edge $\{v, w\}$ as ALLOWED
      if back edge or cross edge then {cycle detected}
        perform ContractCycle(v,w)
      end if
    end if
  end for
  set $finishing[v] \leftarrow time \leftarrow time + 1$
  set $finishing[parent[v]] \leftarrow time \leftarrow time + 1$

  procedure ContractCycle(v,w)
  set $c \leftarrow FINDSET(w)$
  set $x \leftarrow FINDSET(v)$
  while $x \neq c$ do
    UNIONSET($x$,$c$)
    mark edge $\{x, parent[x]\}$ as ALLOWED
    set $x \leftarrow FINDSET(parent[x])$
  end while
  MAKESET($c$)
```

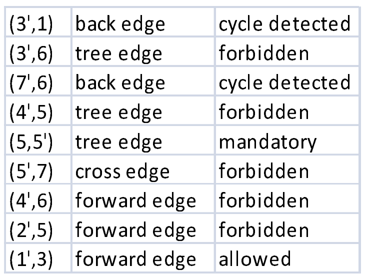
Year	Author(s)	Complexity	Heuristic/Method
1961	Arden, Galler, & Graham [14]	$\Theta(n + m \cdot n)$	naive find
1964	Galler & Fischer [129]	$\Theta(n + m \cdot \log n)$	linking by size or rank
1972	Fischer [103]	$\mathcal{O}(n \cdot \log\log n)$	collapsing with weighting
1973	Hopcroft & Ullman [168]	$\mathcal{O}(m \cdot \log^* n)$	path compression
1974	McIlroy & Morris [1]	$\Theta(m + n \cdot \log n)$	collapsing with linking
1975	Tarjan [283]	$\mathcal{O}(m \cdot \alpha(m, n))$	multiple partition
1976	Rem [77, Chapter 23]	$\Theta(m \cdot \log_{(2+m/n)} n)$	naive splicing
1976	Rem [77, Chapter 23]	$\Theta(m \cdot \alpha(m + n, n))$	splicing by rank
1977	van Leeuwen & Weide [301]	$\mathcal{O}(m \cdot \log^* n)$	path halving
1977	van Leeuwen & Weide [301]	$\mathcal{O}(m \cdot \log^* n)$	path splitting
1977	van Leeuwen & Weide [301]	$\mathcal{O}(n + m \cdot \log n)$	reversal with linking
1984	Tarjan & van Leeuwen [287]	$\mathcal{O}(m \cdot \alpha(m, n))$	path compression
1985	Gabow & Tarjan [121]	$\mathcal{O}(m + n)$	microsets
1986	Blum [45]	$\mathcal{O}(\log n / \log\log n)$	k-UF trees

Table 4.2: History of algorithms for the disjoint set union problem

We have found the partition of edges and from the above algorithm we have obtained forbidden, allowed and mandatory edges. Let G_W denote the partial graph of $G[C_1 \cup C_2]$ obtained by removing all edges which cannot belong to any perfect matching. Then G_W has as many connected components as the number of cores in the alternating depth-first forest plus the number of mandatory edges. The canonical decomposition and thus the partition of vertices and edges is easy to perform.

Figure 4.6 illustrates the progress of an alternating depth-first search on the graph shown in Figure 4.4.





(3',1)	back edge	cycle detected
(3',6)	tree edge	forbidden
(7',6)	back edge	cycle detected
(4',5)	tree edge	forbidden
(5,5')	tree edge	mandatory
(5',7)	cross edge	forbidden
(4',6)	forward edge	forbidden
(2',5)	forward edge	forbidden
(1',3)	forward edge	allowed

Figure 4.6: Alternating depth-first search for perfect matching

Let us demonstrate in more detail how our procedure works. We consider the graph on the left side of the figure and perform an alternating depth-first search beginning at the vertex 1. The edges are traversed in the following order: $\{1, 1'\}$, $\{1', 2\}$, $\{2, 2'\}$, $\{2', 4\}$, $\{4, 4'\}$, $\{4', 3\}$, $\{3, 3'\}$. After that, the back edge $\{3', 1\}$ detects the alternating cycle $C_1 = 11'22'44'33'$. This cycle is contracted to the core 1 and all edges contained in it are marked as allowed. The search continues with the edges $\{3', 6\}$, $\{6, 6'\}$, $\{6', 7\}$, $\{7, 7'\}$, and the back edge $\{7', 6\}$ detects the new alternating cycle $C_2 = 66'77'$ with the core 6. Then the algorithm backtracks from $7'$ to $4'$. Now the edges $\{4', 5\}$ and $\{5, 5'\}$ are traversed. The cross edge $\{5', 7\}$ leads to the completed cycle C_2 and thus it remains marked as forbidden. The routine backtracks again to the vertex $4'$ and the forward edge $\{4', 6\}$ is traversed. Now the vertex $2'$ is reached and the forward edge $\{2', 5\}$ is traversed. Finally, the algorithm backtracks to the vertex $1'$ and the forward edge $\{1', 3\}$ is marked as allowed since it connects two vertices of the active cycle C_1. Now the root vertex 1 is reached and the algorithm terminates.

The alternating tree constructed by the alternating depth-first search is depicted in the middle of the figure. Since two disjoint alternating cycles C_1 and C_2 during the search have been detected and the tree edge $\{5, 5'\}$ is mandatory, three connected components are shown. The table on the right side of the figure summarizes the most important steps of the procedure and shows the partition of edges. All the remaining tree edges are classified as allowed.

4.3.3 Alternating depth-first search for perfect (g, f)-matchings

Recall that we can transform any perfect (g, f)-matching into any other by a series of transformations. These transformations are based on alternating paths and cycles (cf. Figure 4.2). Additionally, if there exists an edge that does not occur in any of these types of alternating paths and cycles then this edge is forbidden if the edge is free, and mandatory if the edge is matched.

Motivated by these observations, we can now state the similar result as for the bipartite graphs with a perfect matching. The following corollary extends the result of Corollary 4.3.4 to the case of the bipartite graphs with perfect (g, f)-matchings.

Corollary 4.3.8 *Let M be a perfect (g, f)-matching in a bipartite graph G, let $C_1, \ldots, C_k$ be the cycles (relative to M) and let $P_1, \ldots, P_t$ be the paths (relative to M) determined by an alternating depth-first search on G. Then*

- *Edges included in any alternating cycle or alternating path are allowed.*
- *Matched edges included neither in alternating cycle nor alternating path are mandatory.*
- *Free edges included in no alternating cycle nor alternating path are forbidden.*

Proof This corollary is a generalization of Ore's Theorem to (g, f)-matchings (see Theorem 4.1.2 in [238]). □

Let $C = C_1 \cup C_2$ be the set of vertices from the canonical decomposition of G with the perfect (g, f)-matching M. Let us define the following sets:

$$\begin{aligned} C^- &= \{\text{the set of negative vertices, i.e. } g(x) < d_M(x) \leq f(x)\} \\ C^+ &= \{\text{the set of positive vertices, i.e. } g(x) \leq d_M(x) < f(x)\} \\ C^0 &= \{\text{the set of neutral vertices, i.e. } g(x) = d_M(x) = f(x)\} \end{aligned}$$

Recall that our task consists of finding the edges that cannot be a part of an optimal (g, f)-matching. Therefore, detecting edges that cannot be a part of an alternating cycle and an alternating path would be sufficient. We show how to make every vertex of the graph totally covered and how to shrink the bounds of degree conditions. Obviously, it is equivalent to enforce hyper-arc consistency on variables and bounds consistency on cardinality variable domains associated with the bipartite graph.

In order to establish the partition of vertices and edges algorithmically, we first compute an optimal (g, f)-matching in the bipartite graph G. This can be realized in $\mathcal{O}(\sqrt{g(V)} \cdot m)$ operations by an algorithm due to Hell & Kirkpatrick [158]. In this way we get a canonical decomposition $\langle A_1, B_1, C_1\rangle$ and $\langle A_2, B_2, C_2\rangle$. Next, we filter the inconsistent edges and corresponding domain values in G. Using the standard approach of the Dulmage-Mendelsohn Decomposition we can easily determine the set of allowed, forbidden and mandatory edges.

The subgraphs G_U, G_O and G_W can be obtained in linear time by using a modified alternating breadth-first search restricted to alternating trails. The partition of vertices can be determined by forming the alternating breadth-first forest from the set of exposed vertices. Similarly, as for the perfect matchings, the components of G_W can be determined by using a modified alternating depth-first search restricted to closed alternating trails.

Unfortunately, in the current case both positive surplus bipartite graphs and elementary bipartite graphs may contain mandatory edges and the auxiliary digraph associated with the latter must not be, in general, strongly connected, so we cannot blindly apply the machinery from the previous subsection for the filtering algorithm.

There are some significant differences. Alternating cycles for perfect matchings start and terminate on the same side of the bipartite graph. Their symmetric structure makes it sufficient to search for alternating cycles starting from only one of the two sides. Unfortunately, the alternating paths for perfect (g, f)-matching are not symmetric: they can be of either even or odd length and may start and terminate on different sides of the bipartite graph. The first approach that suggests itself is to look for alternating cycles and paths with respect to perfect (g, f)-matchings in three phases, each of which restricts attention to starting vertices on a single side. Of course, it remains to be demonstrated that such a strategy achieves the partition of edges. Our algorithm has the following form:

I Find alternating paths and alternating cycles starting from unvisited negative vertices and matched edges on the left side of the graph, which contains at least one negative vertex.

II Find alternating paths and alternating cycles starting from unvisited positive vertices and free edges on the right side of the graph.

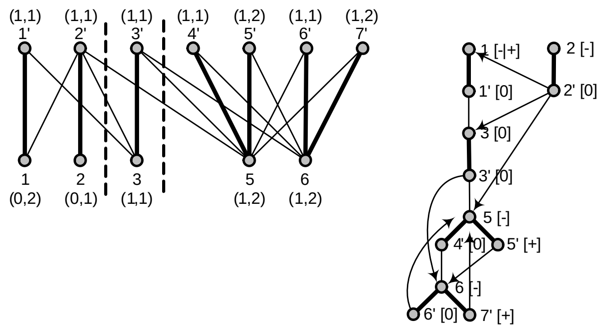
III Find alternating cycles starting from unvisited neutral vertices and matched edges on the left side of the graph.

Note that the first two phases of the algorithm are structurally identical. The only difference is the level of starting vertices in the depth-first search: the negative (or neutral) roots have the even level, the positive roots have the odd level. The third phase of the algorithm is identical with the algorithm from the previous subsection for bipartite graphs with perfect matchings, because an optimal (g, f)-matching is simply a maximum matching, if $g(x) \equiv f(x) \equiv 1$, and in this case all (saturated) vertices will be labeled as neutral.

Given a bipartite graph with an initial perfect (g, f)-matching, we wish to compute the partition of edges. The partition can be found using a modified version of the alternating depth-first search. For alternating cycles the algorithm proceeds in the same way as in the case of perfect matchings. Hence, we need only to describe the routine for computation of alternating paths.

If the alternating depth-first search detects an alternating path, then it marks all edges on this path as allowed and vertices as belonging to the path. All detected alternating paths lead always from the current vertex t_i to the root r_i. If the search encounters a vertex belonging to any alternating path, then a new alternating path is detected. As before, we allow (alternating) paths to have length 0 so that each vertex, which is both positive and negative, belongs to a positive/negative alternating path. In particular, positive/negative roots of trees will be marked (at start) as belonging to a (trivial) path.

Figure 4.7 illustrates the progress of the alternating depth-first search on the normalized subgraph G_W shown in Figure 4.5.



(1,1')	tree edge	mandatory
(1',3)	tree edge	forbidden
(3,3')	tree edge	mandatory
(3',5)	tree edge	forbidden
(6',5)	back edge	cycle detected
(7',5)	back edge	cycle detected
(5',6)	cross edge	allowed
(3',6)	forward edge	forbidden
(2',1)	cross edge	path detected
(2',3)	cross edge	forbidden
(2',5)	cross edge	forbidden

Figure 4.7: Alternating depth-first search for perfect (g,f)-matching

Let us demonstrate how our algorithm works. In the same way as with the perfect matching we apply an alternating depth-first search to the graph depicted on the left side of the figure. The first phase of the algorithm starts from the negative/positive vertex 1 and the matched edge $\{1, 1'\}$. The successive edges are traversed in the following order: $\{1', 3\}$, $\{3, 3'\}$, $\{3', 5\}$, $\{5, 4'\}$, $\{4', 6\}$, $\{6, 6'\}$. After that, the back edge $\{6', 5\}$ detects the alternating cycle $C_1 = 54'66'$ with the core vertex 5. Then, the algorithm backtracks to 6 and the matched edge $\{6, 7'\}$ is traversed. The back edge $\{7', 5\}$ detects the nested cycle

$C_2 = 54'67'$ which is embedded into the cycle C_1. Now the procedure backtracks from $7'$ to 6, to $4'$, then to 5 and the matched edge $\{5, 5'\}$ is traversed. The cross edge $\{5', 6\}$, as leading to the active cycle C_2, is marked as allowed and a new nested cycle is detected. Next, the search backtracks to the vertex $3'$, the core vertex 5 becomes black and thus the forward edge $\{3', 6\}$, as leading to the completed cycle C_1, remains classified as forbidden. The algorithm backtracks to the root vertex 1. The next (not yet discovered) negative vertex 2 becomes the root of a new alternating tree and the matched edge $\{2, 2'\}$ is traversed. All the remaining edges are cross edges. The cross edge $\{2', 1\}$, as leading to the (trivial) alternating path, is marked as allowed, the cross edges $\{2', 3\}$ and $\{2', 5\}$ remain forbidden. Finally, the algorithm backtracks to the root vertex 2 and the matched edge $\{2', 2\}$, as belonging to the alternating path $P_1 = 12'2$, is marked as allowed. Since all vertices of the graph have been visited, the algorithm terminates.

The alternating forest constructed by the alternating depth-first search is depicted in the middle of the figure. The table on the right side shows the classification of edges. All the remaining edges are allowed.

The following results verify that our algorithm works correctly.

Lemma 4.3.9 *If an edge is on an alternating trail then it is on an alternating path.*

Proof Let $P = v_0 e_1 v_1 .. v_{t-1} e_t v_t$ be the shortest alternating trail. Then P is an alternating path. In order to see it, suppose conversely that there exists $1 \leq i < j < t$ such that $v_i = v_j$. Since bipartite graphs have only cycles of even length (see Theorem 2.5.1) then $P' = v_0 e_1 v_1 .. e_i v_i e_{j+1} .. v_{t-1} e_t v_t$ is also an alternating trail, but shorter than P. This assumption leads to a contradiction. Thus, P is an alternating path. □

Theorem 4.3.10 *Alternating paths are detected by tree edges leading either to a negative vertex on the odd level or to a positive vertex on the even level.*

Proof Immediate from the definition of alternating paths with respect to perfect (g, f)-matchings and the fact that the trees have negative roots on the even level or positive roots on the odd level. □

Theorem 4.3.11 *Alternating paths are detected by non-tree (forward or cross) edges leading to a vertex belonging to any alternating path.*

Proof Suppose that there is a forward edge $\{v, w\}$. Then vertex v is a descendant of vertex w in the depth-first tree. Since vertex w belongs to a path, thus there exists a path $r..v..w..t$ from the root r to the vertex t connecting vertex v to w. Then the desired alternating path is $r..vw..t$. The proof for cross edges is similar. □

Theorem 4.3.12 *Alternating paths are detected by non-tree (forward or cross) edges leading to a cycle whose core belongs to any alternating path.*

Proof Suppose that there is a cross edge $\{v, w\}$. We have to distinguish two cases: vertex v belongs to the same tree as vertex w or vertices v and w belong to two distinct trees. In both cases there exists a path from the root r (or r') to the vertex v. Since the vertex w lies on a cycle whose core c belongs to a path, there exists a path from w to c, and a path from c to t. This results in a path from w to t (when t is an ancestor of w) or in a trail from w to t (when t is a descendant of w), and a cross edge completes an alternating path or trail $r..vw..c..t$ (or $r'..vw..c..t$). The proof for forward edges is similar. □

Let us summarize some important properties of the alternating depth-first search for perfect (g, f)-matching:

- Each vertex on the even level is adjacent by a matched edge with a vertex on the odd level.
- Every tree has a negative (or neutral) root on the even level, or a positive root on the odd level.
- Cycles are determined in the same way as by the alternating depth-first search for perfect matchings.
- Paths are detected by (allowed) tree edges or non-tree edges leading to a vertex belonging to an alternating path.
- The tree edge is allowed when it leads either to a negative vertex on the odd level, or to a positive vertex on the even level.
- Trees with neutral roots admit no alternating paths.

We obtain the following theorem.

Theorem 4.3.13 *A partition of vertices and edges for a bipartite graph with a given initial perfect (g, f)-matching can be found in linear time.*

4.3.4 Shrinking the bounds of degree conditions

For a specific vertex x we want to find the lower bound of $g(x)$ and the upper bound of $f(x)$ such that there exists an optimal (g, f)-matching M with the same deficiency.

In order to establish bounds of the degree conditions, we will run on the graph G the same routine twice, first for the degree condition g to obtain its lower bound and then for the dual degree condition f' to compute the upper bound of the degree condition f.

For each saturated vertex x in G we construct a graph $G-x$ with a partial (g, f)-matching and find an optimal (g, f)-matching M_x^g. Then the number of edges in M_x^g adjacent to x must be at least $\delta(G-x) - \delta(G)$. In a similar way we can find the upper bound of $f(x)$. We remove the vertex x from the graph G and compute a dual optimal (f', g')-matching $M_x^{f'}$. Then the maximum number of edges adjacent to x in an optimal (g, f)-matching equals $d(x) - \delta'(G-x) + \delta'(G)$. Observe that the narrowing of the degree conditions can be an expensive task requiring as many as $\mathcal{O}(m \cdot n)$ steps in the worst case. The following result allows us to detect in linear time when the bounds should be shrunken to the same value.

Algorithm 5 Degree Alternating Depth-First Search of G

```
Require: Bipartite graph $G = (V_1 \cup V_2, E)$ with a perfect $(g, f)$-matching $M$
Ensure: Partition of edges into MANDATORY, ALLOWED and FORBIDDEN
  Let $V_1$ denote the side of $G$ with at least one negative vertex; otherwise
  Let $V_2$ denote the side of $G$ with at least one positive vertex
  [Initialization of variables]
  set $time \leftarrow 1$
  mark all edges in $G$ as FORBIDDEN
  mark all edges in $M$ as MANDATORY
  for each vertex $v \in V$ do
    set $parent[v] \leftarrow NIL$
    set $path[v] \leftarrow FALSE$
    $MAKESET(v)$ {trivial cycle}
  end for
  [Phase I]
  for each negative vertex $r \in V_1$ do
    if $discovery[r] = 0$ then {white vertex}
      set $parent[r] \leftarrow r$ {make $r$ the root of $T$}
      set $level[r] \leftarrow EVEN$
      if $POSITIVE \in sign[r]$ then {trivial path}
        set $path[r] \leftarrow TRUE$
      end if
      perform AlternatingDFS(r,r)
    end if
  end for
  [Phase II]
  for each positive vertex $r \in V_2$ do
    if $discovery[r] = 0$ then {white vertex}
      set $parent[r] \leftarrow r$ {make $r$ the root of $T$}
      set $level[r] \leftarrow ODD$
      if $NEGATIVE \in sign[r]$ then {trivial path}
        set $path[r] \leftarrow TRUE$
      end if
      perform AlternatingDFS(r,r)
    end if
  end for
  [Phase III]
  for each neutral vertex $r \in V_1$ do
    if $discovery[r] = 0$ then {white vertex}
      set $parent[r] \leftarrow r$ {make $r$ the root of $T$}
      set $level[r] \leftarrow EVEN$
      perform AlternatingDFS(r,r)
    end if
  end for
```

```
procedure AlternatingDFS(r,s)
set $discovery[s] \leftarrow time \leftarrow time + 1$
if $level[s] = EVEN$ then
  for every edge $\{s, t\} \in M$ do
    if $discovery[t] = 0$ then {tree edge}
      set $parent[t] \leftarrow s$
      set $level[t] \leftarrow ODD$
      if $NEUTRAL \notin sign[r]$ and $NEGATIVE \in sign[t]$ then {path detected}
        set $path[t] \leftarrow TRUE$
      end if
      perform AlternatingDFS(r,t)
    else {gray vertex}
      if $finishing[FINDSET(t)] = 0$ then {cycle detected}
        mark edge $\{s, t\}$ as ALLOWED
        perform ContractCycle(s,t) (see Algorithm 4)
      else if $NEUTRAL \notin sign[r]$ and $path[FINDSET(t)]$ then {path detected}
        mark edge $\{s, t\}$ as ALLOWED
        set $path[s] \leftarrow TRUE$
      end if
    end if
  end for
else {level[s] = ODD}
  for every edge $\{s, t\} \notin M$ do
    if $discovery[t] = 0$ then {tree edge}
      set $parent[t] \leftarrow s$
      set $level[t] \leftarrow EVEN$
      if $NEUTRAL \notin sign[r]$ and $POSITIVE \in sign[t]$ then {path detected}
        set $path[t] \leftarrow TRUE$
      end if
      perform AlternatingDFS(r,t)
    else {gray vertex}
      if $finishing[FINDSET(t)] = 0$ then {cycle detected}
        mark edge $\{s, t\}$ as ALLOWED
        perform ContractCycle(s,t) (see Algorithm 4)
      else if $NEUTRAL \notin sign[r]$ and $path[FINDSET(t)]$ then {path detected}
        mark edge $\{s, t\}$ as ALLOWED
        set $path[s] \leftarrow TRUE$
      end if
    end if
  end for
end if
set $finishing[s] \leftarrow time \leftarrow time + 1$
if $s \neq r$ and $path[s] = TRUE$ then
  set $p \leftarrow parent[s]$
  mark edge $\{s, p\}$ as ALLOWED
  set $path[p] \leftarrow TRUE$
end if
```

Theorem 4.3.14 *Let $A = A_1 \cup A_2$, $B = B_1 \cup B_2$ and $C = C_1 \cup C_2$, whereby A_i, B_i and C_i (for $i = 1, 2$) are the three sets of the Dulmage-Mendelsohn decomposition for bipartite graph G with an optimal (g, f)-matching M. Then the lower bounds $\hat{g}$ and the upper bounds $\hat{f}$ of the degree conditions g and f are as follows:*

$$\hat{g}(x) = \begin{cases} f(x), & \textit{if } x \in A \\ g(x), & \textit{if } x \in B \\ max\{g(x), \delta(G_W - x)\}, & \textit{if } x \in C^- \\ g(x), & \textit{if } x \in C \setminus C^- \end{cases}$$

$$\hat{f}(x) = \begin{cases} f(x), & \textit{if } x \in A \\ g(x), & \textit{if } x \in B \\ d(x) - max\{f'(x), \delta'(G_W - x)\}, & \textit{if } x \in C^+ \\ f(x), & \textit{if } x \in C \setminus C^+ \end{cases}$$

Proof According to Theorem 4.2.3 the allowed degrees for the vertices in A and B equal $f(x)$ and $g(x)$, respectively, so we only need to prove our theorem for the vertices in C. Consider the graph G_W with a perfect (g, f)-matching. Clearly, $\hat{g}(x) \equiv g(x) = f(x) \equiv \hat{f}(x)$ for every neutral vertex $x \in C^0$. If the vertex x is strictly positive (i.e. $g(x) = d_M(x) < f(x)$) then it is not necessary to compute the optimal (g, f)-matching of $G_W - x$, since the vertex x has already achieved the lower bound of the degree condition and it is impossible to construct a matching that has one fewer edge adjacent to the vertex x. Therefore, we can set $\hat{g}(x) \equiv g(x)$ for every $x \in C^+ \setminus C^-$. On the other hand, the same applies to the set $C^- \setminus C^+$: if the vertex x is strictly negative (i.e. $g(x) < d_M(x) = f(x)$) then it is not necessary to compute the optimal (f', g')-matching of $G_W - x$ and thus we can set $\hat{f}(x) \equiv f(x)$ for each $x \in C^- \setminus C^+$. Hence, we need only to determine the bounds of the degree conditions for the vertices $x \in C^- \cup C^+$: the lower bound for the vertices in C^- and the upper bound for the vertices in C^+. □

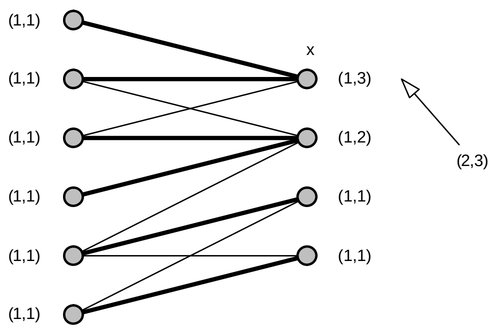


Figure 4.8: Shrinking the bounds of degree conditions

We will now compare our result with the algorithm due to Quimper et al. [253]. Their method prunes the cardinality variables in $\mathcal{O}(m \cdot n)$ steps (for lower bounds) + $\mathcal{O}(m \cdot n^{1.66})$

steps (for upper bounds). However, the time required to compute the allowed bounds of the degree conditions by our algorithm is $\mathcal{O}(m \cdot (|C^-| + |C^+|))$. This is an improvement of the algorithm due to Quimper et al. by a factor of $n^{0.66}$ in the worst case. In the best case our algorithm does not need to iterate over all vertices and that might be greatly efficient.

In addition, it is an interesting open problem whether there exists an algorithm that prunes the domains of the cardinality variables to hyper-arc consistency. We now prove that our algorithm achieves hyper-arc consistency for the cardinality variables when their domains are intervals.

Theorem 4.3.15 *Let G be any bipartite graph with shrunken degree conditions g and f. For any vertex $x \in V(G)$ the degrees of x in perfect (g, f)-matchings form a sequence of consecutive integers.*

Proof Let $M^{g(x)}$ be a perfect (g, f)-matching such that $d_M(x) = g(x)$ and let $M^{f(x)}$ be a perfect (g, f)-matching such that $d_M(x) = f(x)$. According to Theorem 4.1.5 we can transform $M^{g(x)}$ into $M^{f(x)}$ by a sequence of transfers along alternating cycles and alternating paths. Since any alternating path changes the degree of x by at most 1, the degrees of x in the intermediate perfect (g, f)-matchings M will cover every integer in the interval $[g(x), f(x)]$. □

Let us remark that the above theorem does not hold for general graphs. As a counterexample let us take the cycle graph C_3 with the following degree conditions: $f(x_1) = g(x_1) = f(x_2) = g(x_2) = 1$, $f(x_3) = 0$ and $g(x_3) = 2$. There exists no perfect (g, f)-matching with $d_M(x_3) = 1$ (see Figure 10.2.2 in [212]).

4.4 Application to Global Constraints

In this section we consider application of the Dulmage-Mendelsohn Canonical Decomposition to several global constraints. A structural decomposition of a bipartite graph associated with a global constraint canonically decomposes it into three distinct parts: *over-constrained*, *under-constrained*, and *well-constrained*. We use the notation G_U to represent an under-constrained part, G_O denotes an over-constrained part and G_W stands for the remaining well-constrained part. At this point, note that one or two of these three subgraphs may be empty in general. Intuitively, if the bipartite graph $G = (X \cup Y, E)$ represents a global constraint, then the constraint is under-constrained if the surplus of Y is negative and it is over-constrained if this surplus is positive.

In an *over-constrained* part the number of variables is greater than the number of values. The additional variables cause the constraint to be violated and thus a maximum matching yields no solution. We can deal with this situation in a number of ways. Firstly, through a search process, we could remove variables in order to make the part well-constrained. Alternatively, contradictory constraints might be handled as soft constraints associated with the so-called *variable-based violation measure*. In this way, the over-constrained part of the constraint can be isolated from parts that must be necessarily satisfied.

An *under-constrained* part has more values than variables. It is the underdetermined part that contains redundancy. Thus, for under-constrained constraints we do not have enough information to set all the values. The additional values are either redundant or contradictory and thus a perfect matching in the graph does not exist. A possible way of making the constraint satisfiable would be to consider some of them as parameters in order to obtain a well-constrained part, add additional values to the global constraint or remove unsatisfied constraints from the global constraint which corresponds to the so-called *value-based violation measure*.

In a *well-constrained* part the number of variables is equal to possible assignments of values to variables. As we have seen, this part can be further decomposed into smaller canonical parts. A failure in computing the well-constrained part means that no valid solution exists and the constraint is unsatisfiable. The existence of a single solution requires that the number of variables is equal to the number of constraints in the global constraint.

A global constraint is representable by a bipartite graph if the constraint can be modeled as a matching problem in a bipartite graph. A solution can be represented by a maximum, complete or perfect matching and there is a one-to-one correspondence between the matching and the solution of the constraint. The pruning of the domain can be done according to the meaning of the respective edge in the graph.

4.4.1 Hard Global Constraints

The presented approach allows us to implement an efficient generic propagation algorithm for various global constraints representable by a bipartite graph, making use of matching and decomposition theory. We apply our method to several global constraints that are well known to the constraint programming community.

We now provide the skeleton of our general propagation routine.

Algorithm 6 General propagation routine for hard global constraints representable by bipartite graphs

```
Require: Hard global constraint
Ensure: Hyper-arc consistency or constraint inconsistent
  Normalize the domains of the variables {quick elimination}
  Create an auxiliary bipartite graph $G$ associated with the global constraint
  Compute the maximum matching $M$ in $G$
  Compute the Dulmage-Mendelsohn Canonical Decomposition of $G$
  Determine subgraphs $G_O$, $G_U$ and $G_W$ corresponding to the solution
  If $G_O \neq \emptyset$ then return FALSE {constraint not consistent}
  Find the partition of mandatory, allowed and forbidden edges
  Remove all forbidden edges from the graph $G$
  Prune the domains of the variables according to the partition of edges
  If any of the domains becomes empty then return FALSE {constraint not satisfied}
  return TRUE
```

The following result is immediate:

Theorem 4.4.1 (Complexity of the algorithm) *Assume that the global constraint is representable by a bipartite graph G with n vertices and m edges. Then the feasibility of the constraint can be checked in $\mathcal{O}(\sqrt{n} \cdot m)$ time and hyper-arc consistency can be established in $\mathcal{O}(m + n)$ time.*

We now demonstrate our idea on concrete examples. Some of the examples are illustrated by a figure. On the left side of each figure the domains of the variables are given. In the middle of each figure a bipartite graph corresponding to the global constraint is depicted and the decomposition is shown. The thick edges indicate a matching, while the vertical dashed lines show forbidden edges. On the right side of each figure the reduced domains after pruning are presented.

ALLDIFFERENT The ALLDIFFERENT constraint is defined by

$$\text{ALLDIFFERENT}(\langle x_1, \ldots, x_n \rangle) = \{(d_1, \ldots, d_n) \in D_{x_1} \times \cdots \times D_{x_n} \mid \underset{\substack{i,j\\ i \neq j}}{\forall} (d_i \neq d_j)\}.$$

It is well known that the constraint ALLDIFFERENT can be modeled as a matching problem in a bipartite graph G called a *value graph* [255]. On the left side we have a vertex for every variable x_i and on the right side we have a vertex d_j for every value that occurs in some domain of the variables. We draw an edge between x_i and d_j iff the value d_j is contained in the domain of x_i. Then there is a one-to-one correspondence between the maximum matching of G covering all vertices representing variables and the assignment of variables satisfying the constraint (see Theorem 1 in [255]). The constraint is consistent iff $G_O = \emptyset$. Hyper-arc consistency can be established in linear time. However, by means of our technique it is possible to do it better by a constant factor. This follows from the fact that our algorithm does not need to iterate over all values in the domain. The alternating depth-first search simultaneous finds alternating cycles and classifies the edges (Figure 4.9). Indeed, it is not necessary, as in the algorithm due to Régin [255], to identify strongly connected components in order to determine partition of edges. Combining the computation of alternating cycles (= strongly connected components in the auxiliary digraph) at the same time as categorizing edges seems to be an effective way to save time during the computation.

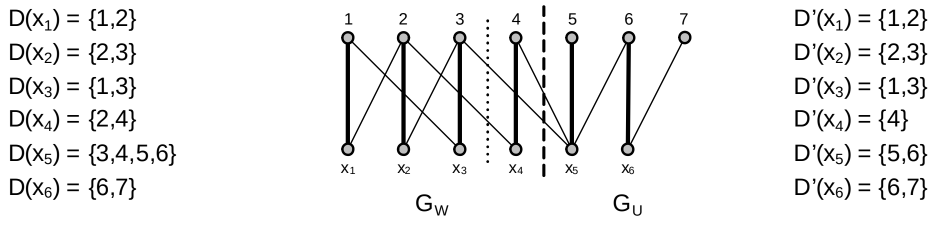


Figure 4.9: Value graph and pruning of the ALLDIFFERENT constraint

The following table presents the time complexity survey of the algorithms for the ALLDIFFERENT constraint.

consistency	complexity	approach	author(s)
bounds consistency	$\mathcal{O}(n \cdot \log n)$	Hall intervals	Puget [248]
	$\mathcal{O}(n \cdot \log n)$	Hall intervals	López-Ortiz et al. [208]
	$\Theta(n)$	Hall intervals	Quimper [252, Section 4.2]
	$\mathcal{O}(n)$	convex bigraph	Mehlhorn & Thiel [219]
range consistency	$\Theta(n^2)$	Hall sets	Leconte [203]
	$\mathcal{O}(n)$	Hall intervals	Quimper [252, Section 4.3]
hyper-arc consistency	$\mathcal{O}(\sqrt{n} \cdot m)$	matching theory	Régin [255]

Table 4.3: Algorithms for the ALLDIFFERENT constraint

For information concerning the empirical behavior of filtering algorithms for the ALLDIFFERENT constraint, see [134]. Theses dealing with the ALLDIFFERENT constraint have been written by Sven Thiel [288], Willem-Jan van Hoeve [299], and Claude-Guy Quimper [252].

ALLDIFFERENT_EXCEPT_0 The ALLDIFFERENT_EXCEPT_0 constraint [27] is a special case of the ALLDIFFERENT constraint. Quite often it appears, that for some modeling reason, we do not want the normal constraint to hold for variables that take value 0. This constraint is useful for this purpose by enforcing all variables of the constraint to take distinct values, except those variables which are assigned to 0.

The filtering achieving hyper-arc consistency can be realized by means of our technique as follows. We construct, as previously described, a bipartite graph $G = (X \cup Y, E)$ with vertices on one side that correspond to variables, and vertices on the other side that correspond to elements in the variable domains plus some additional vertices 0_i representing variables with joker values. The graph contains an edge $\{x_i, d_j\}$ whenever $d_j \in D(x_i)$ $(d_j \neq 0)$ and an edge $\{x_i, 0_i\}$ whenever $0 \in D(x_i)$. It is not difficult to observe that there is a one-to-one correspondence between the solution of the constraint and the maximum matching in the constructed graph saturating all vertices representing variables.

CORRESPONDENCE The CORRESPONDENCE constraint [27] is defined on three collections of variables. It is derived from the SORT_PERMUTATION constraint [315] by removing the sorting condition. This constraint has the form CORRESPONDENCE(X,Y,Z) and states that the variables of the third collection Z correspond to the variables of the first collection X according to the permutation expressed by the second collection Y.

We represent the CORRESPONDENCE constraint as a bipartite graph $G = (X \cup Z, E)$ which we call a *restricted intersection graph*. The vertices on one side represent the variables X, the vertices on the other side represent the variables Z and there is an edge $\{x_i, z_j\}$ iff $D(x_i) \cap D(z_j) \neq \emptyset \wedge j \in D(y_i)$. Note that the second condition is usually called an ELEMENT constraint [160].

Looking at the definition of the CORRESPONDENCE constraint, it is easy to derive a tight correlation between the solution of the constraint and the perfect matching in its auxiliary graph. This observation motivates the following pruning algorithm. We first compute the perfect matching M in the graph G. If it does not exist, then the constraint is inconsistent.

Otherwise, we need to detect all edges that can never belong to a perfect matching. This can be simply done by means of our routine. Hyper-arc consistency can be obtained by the following narrowing of variable domains:

$$\begin{aligned} D'(x_i) &= D(x_i) \cap \bigcup_{\{x_i,z_j\} \in E(G_W)} D(z_j) \\ D'(y_i) &= D(y_i) \cap \bigcup_{\{x_i,z_j\} \in E(G_W)} \{j\} \\ D'(z_j) &= D(z_j) \cap \bigcup_{\{x_i,z_j\} \in E(G_W)} D(x_i) \end{aligned}$$

INVERSE The INVERSE constraint [27] is a particular case of the CYCLE constraint, a case which can be modeled by means of the system of two dependently ALLDIFFERENT constraints. This constraint requires that each vertex of an associated digraph has exactly one predecessor and one successor and that if the successor of the vertex i is the vertex j then the predecessor of the vertex j is the vertex i.

The constraint is defined more formally as follows

$$\text{INVERSE}(\langle x_1, \ldots, x_n \rangle, \langle y_1, \ldots, y_n \rangle) =$$
$$\{(d_1, \ldots, d_n) \in D_{x_1} \times \cdots \times D_{x_n}, (d'_1, \ldots, d'_n) \in D_{y_1} \times \cdots \times D_{y_n} \mid \underset{i \neq j}{\forall} (d_i = j \Leftrightarrow d'_j = i)\}.$$

This constraint is present in problems sometimes required by specific heuristics that use both predecessor and successor variables. In this example we show how to simply achieve hyper-arc consistency for this constraint.

The constraint must first be normalized as follows. Suppose that there exists a value $j \in D(x_i)$, while $i \notin D(y_j)$. Then we can immediately remove value j from $D(x_i)$. Similarly, we have to remove value i from $D(y_j)$ if $j \notin D(x_i)$. Hence, we assume that such situations do not occur during the creating of an auxiliary graph.

The INVERSE(X,Y) constraint can be expressed by a bipartite graph $G = (X \cup Y, E)$, called a *variable graph*, in which the vertices of X correspond to the predecessor variables, the vertices of Y correspond to the successor variables, and there is an edge between two vertices x_i and y_j iff $i \in D(y_j)$ and $j \in D(x_i)$.

It is straightforward to see that there is a one-to-one correspondence between the perfect matching in the corresponding graph and the solution of the constraint. For every forbidden edge $\{x_i, y_j\}$ we have to delete the value i from $D(y_j)$ and the value j from $D(x_i)$.

SAME The SAME constraint is defined on two sequences of variables and states that the variables in one sequence use the same values as the variables in the other sequence. The constraint was introduced by Beldiceanu [27]. One can also view the SAME constraint as requesting that one sequence is a permutation of the other. A hyper-arc consistency algorithm for the SAME constraint has been presented by Beldiceanu, Katriel, & Thiel [33], making use of flow theory.

The SAME constraint [27] is a relaxed version of the SORT constraint introduced in [233] in which we do not enforce the second collection of variables to be sorted in increasing order.

The constraint is defined on two sets X and Y of distinct variables such that $|X| = |Y|$ and each $v \in X \cup Y$ has a domain $D(v)$. A solution is an assignment of values to the variables such that the value assigned to each variable belongs to its domain and the multiset of values assigned to the variables of X is identical to the multiset of values assigned to the variables of Y.

We represent the SAME constraint as a bipartite graph $G = (X \cup Y, E)$ which we call the *intersection graph*. The vertices on one side represent the variables X, the vertices on the other side represent the variables Y, and there is an edge $\{x_i, y_j\}$ iff $D(x_i) \cap D(y_j) \neq \emptyset$. There is a one-to-one correspondence between the perfect matching in G and the solution of the constraint (Figure 4.10). Hyper-arc consistency can be established by the following narrowing of variable domains:

$$\begin{aligned} D'(x_i) &= D(x_i) \cap \bigcup_{\{x_i,y_j\} \in E(G_W)} D(y_j) \\ D'(y_j) &= D(y_j) \cap \bigcup_{\{x_i,y_j\} \in E(G_W)} D(x_i) \end{aligned}$$

.

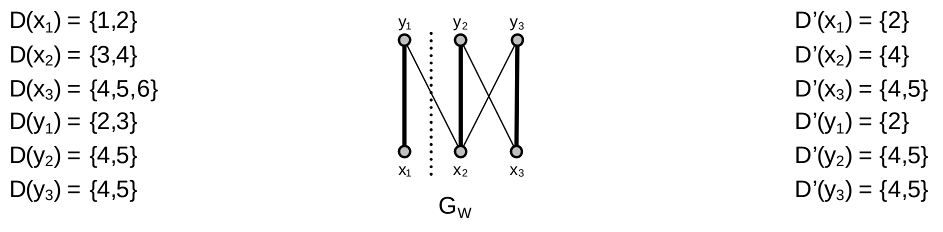


Figure 4.10: Intersection graph and pruning of the SAME constraint

USED_BY The USED_BY constraint is an under-constrained version of the SAME constraint. It is defined on two sets of variables X and Y such that $|X| \geq |Y|$ and a solution is an assignment of values to the variables such that the multiset of values assigned to the variables in Y is contained in the multiset of values assigned to the variables in X.

As in the case of the SAME constraint, we construct the intersection graph $G = (X \cup Y, E)$. There is a one-to-one correspondence between the solution of the USED_BY constraint and the complete matching in G from Y into X (Figure 4.11). Hyper-arc consistency can be established by the following narrowing of variable domains (we assume that $D(x_i) \cap \bigcup_{\emptyset} D(y_j) = D(x_i)$. This can only occur when $x_i \in V(G_U)$):

$$\begin{aligned} D'(x_i) &= D(x_i) \cap \bigcup_{\{x_i,y_j\} \in E(G_W)} D(y_j) \\ D'(y_j) &= D(y_j) \cap \bigcup_{\{x_i,y_j\} \in E(G_W) \cup E(G_U)} D(x_i) \end{aligned}$$

.

GLOBAL_CARDINALITY The GLOBAL_CARDINALITY [256] constraint (abbreviated shortly as GCC) is a generalization of the ALLDIFFERENT constraint enforcing bounds on

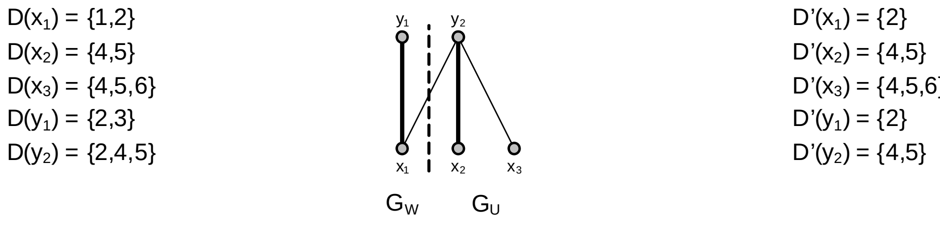


Figure 4.11: Intersection graph and pruning of the USED_BY constraint

the number of occurrences of values in a set of variables. Its filtering algorithm is based on a flow computation. The constraint holds if the corresponding flow network models an admissible flow. The constraint is defined by

$$\textsc{global\_cardinality}(\langle x_1,\dots,x_n\rangle,\langle l_{v_1},\dots,l_{v_k}\rangle,\langle u_{v_1},\dots,u_{v_k}\rangle) =$$
$$\{(d_1,\dots,d_n)\in D_{x_1}\times\cdots\times D_{x_n} \mid \forall_j \left(l_{v_j} \le \left| \bigcup_{\substack{1\le i\le n\\ v_j=d_i}} \{i\} \right| \le u_{v_j} \right) \}.$$

For our example we will need to construct a special graph, called a *variable-value graph*. Following Quimper et al. [253], let $G = (X \cup D, E)$ be a bipartite graph such that the vertices on the left side represent the variables and the vertices on the right side represent the values. There is an edge $\{x_i, d_j\}$ in E iff the value d_j is in the domain $D(x_i)$ of the variable x_i. Let g and f be two functions associated with vertex v such that $g(x_i) = f(x_i) = 1$ for all variable-vertices $x_i \in X$ and $g(d) = l(d)$, $f(d) = u(d)$ for all value-vertices d in D. Then it is straightforward to check that there is a one-to-one correspondence between the solution of the GCC constraint and the perfect (g, f)-matching of the constructed graph (Figure 4.12).

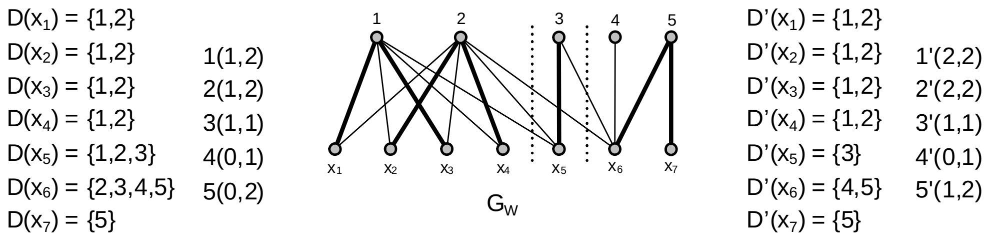


Figure 4.12: Variable-value graph and pruning of the GCC constraint

The following table presents algorithms for the GCC constraint.

Empirical studies concerning the quality of various filtering algorithms for the GCC constraint can be found in [230]. A thesis dealing with the GLOBAL_CARDINALITY constraint is due to Claude-Guy Quimper [252].

SYMMETRIC_CARDINALITY The SYMMETRIC_CARDINALITY constraint is specified in terms of a set of variables X which take their values in the subset V. It constrains the number of times a value can be assigned to a variable x_i in X to be in an interval

consistency	complexity	approach	author(s)
bounds consistency	$\mathcal{O}(n)$	Hall intervals	Quimper et al. [254]
	$\mathcal{O}(n+k)$	convex bigraph	Katriel & Thiel [187]
range consistency	$\mathcal{O}(k \cdot n)$	unstable sets	Quimper et al. [253]
	$\mathcal{O}(n)$	unstable sets	Quimper [252, Section 5.5]
hyper-arc consistency	$\mathcal{O}(k \cdot n^2)$	flow theory	Régin [256]
	$\mathcal{O}(k \cdot n \cdot \sqrt{n})$	matching theory	Quimper et al. [253]

Table 4.4: Algorithms for the GLOBAL_CARDINALITY constraint

$[l(x_i), u(x_i)]$, where $l(x_i)$ and $u(x_i)$ are non-negative integers and at the same time it restricts the number of occurrences of each value $v_j \in V$ in the sets assigned to variables in X to be an interval $[l(v_j), u(v_j)]$. This constraint was introduced in [189] where, moreover, a filtering method for this constraint based on flow theory was presented.

The constraint is defined as

$$\begin{aligned}&\textsc{symmetric\_cardinality}(\langle x_1, \dots, x_n \rangle, \mathit{bounds}) = \\ &\qquad \{((d_1^1, \dots, d_1^{k_1}), \dots, (d_n^1, \dots, d_n^{k_n})) \in D_{x_1}^{k_1} \times \cdots \times D_{x_n}^{k_n} \mid \\ &\qquad\qquad \forall_i (l_{x_i} \leq k_i \leq u_{x_i}) \wedge \forall_j \left(l_{v_j} \leq \left| \bigcup_{\substack{1 \leq i \leq n \\ v_j = d_i}} \{i\} \right| \leq u_{v_j} \right) \}.\end{aligned}$$

In this example we show how this constraint can be modeled as a perfect (g, f)-matching problem. We describe both a method for checking consistency of this constraint and a filtering method thereof. For the SYMMETRIC_CARDINALITY constraint we propose to build a value graph G as described in [255]. Let $G = (X \cup V, E)$ be a bipartite graph such that vertices on the left side represent variables and vertices on the right side represent values. There is an edge $\{x_i, v_j\}$ in E iff the value v_j is in the domain $D(x_i)$ of the variable. Let g and f be two functions associated to vertices of G such that $g(x_i) = l(x_i)$ and $f(x_i) = u(x_i)$ for all variable-vertices $x_i \in X$; and $g(v_j) = l(v_j)$ and $f(v_j) = u(v_j)$ for all value-vertices v_j in V. Then it can be easily shown that there is an equivalence between the existence of a perfect (g, f)-matching of the constructed graph and the consistency of the SYMMETRIC_CARDINALITY constraint.

In addition, our algorithm narrows the bounds of the occurrence variables. This is the first efficient filtering algorithm for this constraint that achieves bounds consistency (which is, in fact, hyper-arc consistency; see Theorem 4.3.15) for all cardinality variables, and not only hyper-arc consistency for the domain variables.

Observe that the SYMMETRIC_CARDINALITY constraint might be viewed as an extension of some previously mentioned constraints. For example, a constraint ALLDIFFERENT can be modeled as a SYMMETRIC_CARDINALITY constraint where the cardinality of sets assigned to variables is set to be exactly 1 and the occurrence of each value is restricted to the interval $[0, 1]$. A constraint ALLDIFFERENT_EXCEPT_0 can be modeled in two different ways. Firstly, when with each vertex of value graph G we associate two functions g and f such that $g(x_i) = f(x_i) = 1$ for all variable-vertices $x_i \in X$; $g(v_j) = 0$ and $f(v_j) = 1$ for all value-vertices $v_j \neq 0$ in V, and we set $g(0) = 0$ and $f(0) = k$, where k is the number of variables with value 0 in their domains. Secondly, when instead of the value-vertex 0 for every variable-vertex x_i, such that $0 \in D(x_i)$, we set $g(x_i) = 0$ and $f(x_i) = 1$, and $g(x_i) = f(x_i) = 1$ otherwise. Moreover, a GLOBAL_CARDINALITY constraint can be modeled as a SYMMETRIC_CARDINALITY constraint by restricting the cardinality of sets assigned to problem variables to value 1.

4.4.2 Soft Global Constraints

A global constraint is over-constrained when no assignment of values to variables makes it consistent. In this situation, the goal is to find the set of variables violating the global constraint. It is then natural to identify soft constraints that are allowed to be violated, and the objective is to optimize the degree of satisfaction of the global constraint according to some criteria. A standard measure of violation is the variable-based violation measure which is defined as the minimum number of variables that need to change their values in order to satisfy the constraint. This measure is represented by the cost variable z, which is to be minimized during the solution process.

In this subsection we propose an efficient hyper-arc consistency algorithm for this case. As a consequence, without any modification, our method can solve over-constrained problems modeled as a matching problem for bipartite graphs. This approach is quite general and can be applied to a wide range of global constraints.

The proposed algorithm for soft global constraints with variable-based violation measure works as follows. We first compute the canonical decomposition of the corresponding bipartite graph. Let $\delta(G_O)$ denote the deficiency of graph G_O; in other words it is a lower bound of the cost variable z. Thus, if $\delta(G_O) > \max(D(z))$ then the constraint is not consistent. Otherwise, we distinguish two situations. If $\delta(G_O) = \max(D(z))$ then all the values of variables that corresponding edges are forbidden must be removed from their domains. Otherwise, if $\delta(G_O) < \max(D(z))$ then all values in their domains are allowed. Finally, we update the lowest possible value of violation to $\min(D(z)) \leftarrow \max(\min(D(z)), \delta(G_O))$. Clearly, if at least one domain of some variable becomes empty then the constraint is inconsistent.

The algorithm solving the constraint is summarized below (see Algorithm 7).

We show that this algorithm is correct.

Theorem 4.4.2 (Correctness of the algorithm) *Assume that a soft global constraint can be represented by a bipartite graph G and there exists a one-to-one correspondence between the solution of the constraint and the maximum matching in G. Then the above algorithm makes the global constraint hyper-arc consistent.*

Proof Let G_O, G_W and G_U be subgraphs of G obtained from the Dulmage-Mendelsohn Canonical Decomposition. The minimal number of variables from $G_O = G[A_2 \cup B_1]$ to change their value in order to make this part of graph G well-constrained is equal to the deficiency

Algorithm 7 General propagation routine for soft global constraints representable by bipartite graphs

```
Require: Soft global constraint with variable-based violation measure
Ensure: Hyper-arc consistency or constraint inconsistent
  Normalize the domains of the variables {quick elimination}
  Create an auxiliary bipartite graph $G$ associated with the global constraint
  Compute the maximum matching $M$ in $G$
  Compute the Dulmage-Mendelsohn Canonical Decomposition of $G$
  Determine subgraphs $G_O$, $G_U$ and $G_W$ corresponding to the solution
  Let $\delta(G_O)$ be the deficiency of the subgraph $G_O$
  if $\delta(G_O) > \max(D(z))$ then {constraint not consistent}
    return FALSE
  else if $\delta(G_O) = \max(D(z))$ then
    Find the partition of mandatory, allowed and forbidden edges
    Remove all forbidden edges from the graph $G$
    Prune the domains of the variables according to the partition of edges
  else
    all values in the domain of each variable are allowed
  end if
  Update $\min(D(z)) \leftarrow \max(\min(D(z)), \delta(G_O))$
  if any of the domains becomes empty then {constraint not satisfied}
    return FALSE
  end if
  return TRUE
```

of G_O. This follows from the fact that inserting $\delta(G_O)$ new vertices to A_2 and connecting them to the vertices in B_1 results in a graph with a perfect matching (see Theorem 2.5.8). Therefore, in the case when $\delta(G_O) = \max(D(z))$ all forbidden edges must be deleted from the graph G in order to make a constraint hyper-arc consistent since they belong to no maximum matching. However, when the upper bound of the cost variable z is greater then $\delta(G_O)$ we can further delete matched edges to make vertices from G_W and G_U exposed. In this case every edge belongs to some maximum matching. In order to see this, we only need to consider free edges. Let $\{v, w\}$ be an arbitrary free edge and $\{v, mate(v)\}$ or $\{w, mate(w)\}$ be its adjacent matched edge(s) (at least one must exist). Assume that we want to make the vertex $mate(w)$ exposed. But then $M - \{w, mate(w)\} + \{v, w\}$ is the desired maximum matching missing the vertex $mate(w)$. □

The following result is immediate:

Theorem 4.4.3 (Complexity of the algorithm) *Hyper-arc consistency for a global constraint representable by a bipartite graph with an initial maximum matching can be established in linear time.*

We now illustrate our idea with concrete examples.

SOFT_ALLDIFFERENT_VAR This (soft) constraint is designed for over-constrained problems in which the task is to obtain a solution that is close to feasibility of the well-known ALLDIFFERENT constraint. The constraint limits the degree to which a set of variables fails to take all distinct values. A complete pruning algorithm establishing hyper-arc consistency is given in [245].

The constraint SOFT_ALLDIFFERENT_VAR can be filtered to hyper-arc consistency by means of our technique as follows. The lower bound of the cost variable z is $|X| - |M|$, where X is the set of variables and M is the maximum matching in the corresponding value graph associated with the constraint. It is equal to the deficiency of G_O. The variables that need to be changed in order to obtain the solution satisfying an ALLDIFFERENT constraint are exposed vertices in X (Figure 4.13).

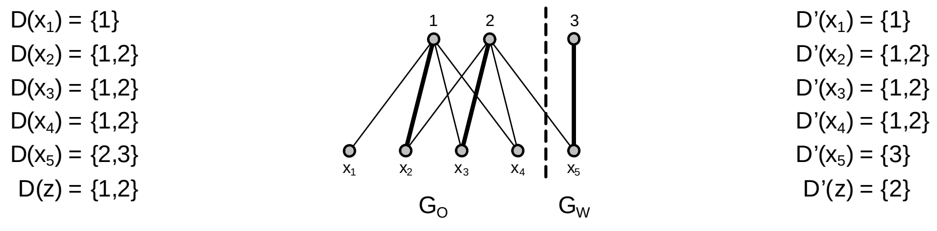
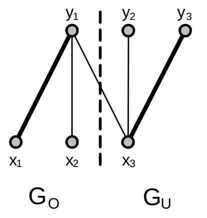


Figure 4.13: Value graph and pruning of the SOFT_ALLDIFFERENT_VAR constraint

SOFT_SAME_VAR The constraint SOFT_SAME_VAR can be filtered to hyper-arc consistency by an algorithm that is similar to the one for the SAME constraint. We first compute the canonical decomposition of an auxiliary graph G, which is an intersection graph. The lower bound of the cost variable z is $\delta(G_O)$. If $\delta(G_O) > \max(D(z))$ we know that the constraint is inconsistent. Otherwise, if $\delta(G_O) = \max(D(z))$, we remove all invalid values from the domains of variables which corresponding edge is forbidden (Figure 4.14).

$D(x_1) = \{1,2\}$
$D(x_2) = \{2\}$
$D(x_3) = \{2,3\}$
$D(y_1) = \{2,3\}$
$D(y_2) = \{3\}$
$D(y_3) = \{3,4\}$
$D(z) = \{0,1\}$

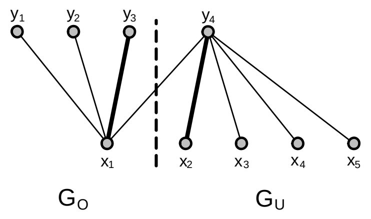


$D'(x_1) = \{1,2\}$
$D'(x_2) = \{2\}$
$D'(x_3) = \{3\}$
$D'(y_1) = \{2\}$
$D'(y_2) = \{3\}$
$D'(y_3) = \{3,4\}$
$D'(z) = \{1\}$

Figure 4.14: Intersection graph and pruning of the SOFT_SAME_VAR constraint

SOFT_USED_BY_VAR The constraint SOFT_USED_BY_VAR is a soft version of the constraint USED_BY and can be filtered to hyper-arc consistency by the same algorithm as for the previously described soft global constraints (Figure 4.15). This constraint can be interpreted as an under-constrained version of the constraint SOFT_SAME_VAR and can thus be filtered to hyper-arc consistency by an analogous algorithm to the latter one.

$D(x_1) = \{1,2\}$
$D(x_2) = D(x_3) = \{2\}$
$D(x_4) = D(x_5) = \{3\}$
$D(y_1) = D(y_2) = D(y_3) = \{1\}$
$D(y_4) = \{1,2,3\}$
$D(z) = \{1,2,3\}$



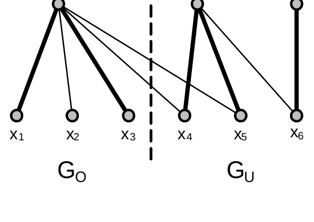
$D'(x_1) = \{1,2\}$
$D'(x_2) = D'(x_3) = \{2\}$
$D'(x_4) = D'(x_5) = \{3\}$
$D'(y_1) = D'(y_2) = D'(y_3) = \{1\}$
$D'(y_4) = \{1,2,3\}$
$D'(z) = \{2,3\}$

Figure 4.15: Intersection graph and pruning of the SOFT_USED_BY_VAR constraint

SOFT_GCC_VAR Note that filtering algorithms for soft versions of the GCC and the SYMMETRIC_CARDINALITY constraints are immediate. When $G_O \neq \emptyset$ then the constraints are over-constrained and the variable-based violation measure will be useful (Figure 4.16). When $G_U \neq \emptyset$ then the constraints are under-constrained and the value-based violation measure will help to solve the constraint.

$D(x_1) = \{1\}$
$D(x_2) = \{1\}$
$D(x_3) = \{1\}$
$D(x_4) = \{1,2\}$
$D(x_5) = \{1,2\}$
$D(x_6) = \{2,3\}$
$D(z) = \{0,1\}$

1(1,2)
2(3,3)
3(1,1)



$D'(x_1) = \{1\}$
$D'(x_2) = \{1\}$
$D'(x_3) = \{1\}$
$D'(x_4) = \{2\}$
$D'(x_5) = \{2\}$
$D'(x_6) = \{2,3\}$
$D'(z) = \{1\}$

1'(2,2)
2'(3,3)
3'(1,1)

Figure 4.16: Variable-value graph and pruning of the SOFT_GCC_VAR constraint

A thesis dealing with the SOFT_GCC_VAR constraint is by Alessandro Zanarini [313].

4.5 Convex bipartite graphs

In this section we study constraints representable by convex bipartite graphs. For such constraints we will develop a filtering algorithm that achieves bounds consistency. In this section we assume that all variable domains are intervals and we will show how to prune them to the possible smallest intervals.

Recall that a bipartite graph G with bipartition (V_1, V_2) is said to be V_1-convex if there is an ordering on V_1 such that for all $v \in V_2$ the vertices adjacent to v are consecutive. Convexity over V_2 is defined analogously. A bipartite graph that is convex over both V_1 and V_2 is said to be doubly convex. Any given bipartite graph can be tested for convexity in linear time using an algorithm devised by Kellogg S. Booth & George S. Lueker [50].

Complexity surveys for maximum matchings in convex bipartite graphs:

Year	Author(s)	Complexity	Strategy/Remarks
1967	Glover [138]	$\mathcal{O}(n_1 \cdot n_2)$	ordered vector
1981	Lipski & Preparata [206]	$\mathcal{O}(n_2 + n_1 \cdot \alpha(n_1))$	disjoint set union
1984	Gallo [130]	$\mathcal{O}(n_1 \cdot \log n_1)$	binary heap
1988	Scutellà & Scevola [265]	$\mathcal{O}(n_1 \cdot \log p + n_1 \cdot \alpha(n_1))$	balanced binary trees
1996	Steiner & Yeomans [276]	$\mathcal{O}(n_1)$	off-line minimum
2008	Katriel [186]	$\mathcal{O}(m + n_2 \cdot \log n_1)$	vertex-weighted matching

Table 4.5: History of algorithms for the convex bipartite matching problem

Here, $p = \min\{\lceil \frac{n_2}{n_1} \rceil, n_1\}$. Recall that a *greedy algorithm* at any individual stage selects, without regard for future consequences, that option, which is, in some particular sense, locally optimal. When the algorithm terminates the local optimum is equal to the global optimum. Note that the greedy algorithm produces an optimal solution only for a special class of structures called matroids. We already know some greedy algorithms, such as Dijkstra's shortest path algorithm [76] or Kruskal's minimum spanning tree algorithm [197]. The next example is an algorithm for finding a matching in a convex bipartite graph. Glover's algorithm (see also [201, Section 5.6] and [15, Section 5.3] for further reading) is greedy in the sense that it always chooses an edge among those that belong to an actual maximal matching, i.e. it picks an allowed edge.

SORT The constraint SORT(X,Y) is defined on two collections of variables. The constraint states that the variables of the second collection correspond to the variables of the first collection according to a permutation sorted in non-decreasing order.

The constraint is introduced in [233]. The filtering algorithm achieving bounds consistency is presented in [44] and in [219]. The complexity of the first algorithm is $\mathcal{O}(n \cdot \log n)$, the running time of the second algorithm is $\mathcal{O}(n)$ plus the time for sorting the bounds of the variable domains.

Now we show how the algorithm can be implemented in linear time if the sorting of the variables in the collection X is known according to both the lower and upper bounds of their domains.

First, the domains of Y should be normalized. Normalization can be achieved by setting

$$\begin{aligned}\min(D'(y_i)) &= \max(\min(D(y_{i-1})), \min(D(y_i))), \text{ where } 2 \leq i \leq n \\ \max(D'(y_i)) &= \min(\max(D(y_i)), \max(D(y_{i+1}))), \text{ where } n-1 \geq i \geq 1\end{aligned}$$

Next, in the same way as in the case of the SAME constraint, we create an intersection bipartite graph G. In fact, it is a convex bipartite graph. Suppose that our intersection convex bipartite graph contains a perfect matching. In order to narrow the variable domains of X we use the following reduction rule:

$$D'(x_i) = D(x_i) \cap \bigcup_{\{x_i, y_j\} \in E(G_W)} D(y_j)$$

However, before we explain how to further narrow the domains of Y we must introduce two kinds of perfect matchings in the convex bipartite graphs. A perfect matching with respect to the upper, or lower, bound of X is obtained when from the set of free vertices we choose such a vertex x_i that $\max(D(x_i))$ is minimal, or $\min(D(x_i))$ is maximal. We denote such perfect matchings by M_{min} and M_{max}, respectively. Then we get the following propagation rules:

$$\begin{aligned}\min(D'(y_j)) &= \max(\min(D(y_j)), \min(D(x_i))), \text{ where } \{x_i, y_j\} \in M_{min} \\ \max(D'(y_j)) &= \min(\max(D(y_j)), \max(D(x_i))), \text{ where } \{x_i, y_j\} \in M_{max}\end{aligned}$$

A thesis dealing with the SORT constraint has been written by Sven Thiel [288].

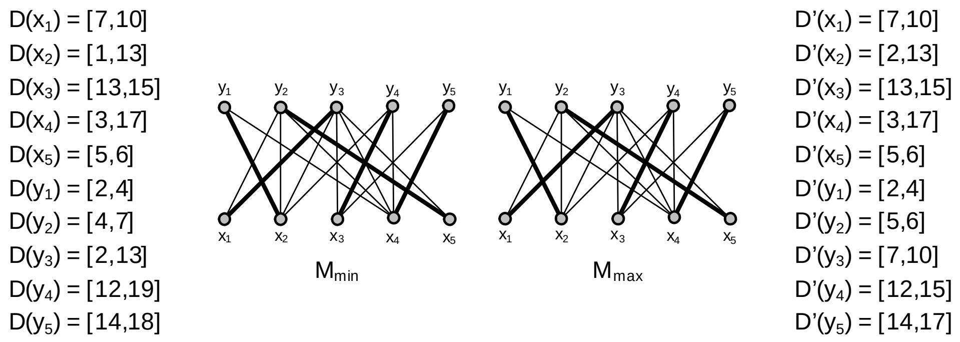


Figure 4.17: Intersection convex bipartite graph and pruning of the SORT constraint

SORT_PERMUTATION The SORT_PERMUTATION constraint was introduced in [315]. It is defined on three collections of variables. The variables of the first collection correspond to the variables of the third collection according to the permutation of the second collection. The variables of the third collection are sorted in increasing order.

In the same way as in the CORRESPONDENCE constraint, the constraint can be modeled by means of the restricted intersection graph: we have an edge $\{x_i, z_j\}$ between X and Z iff $D(x_i) \cap D(z_j) \neq \emptyset \wedge j \in D(y_i)$. If the variable domains are intervals the generated graph is convex.

In order to narrow the domains of the variables X and Z we would do the same as in the previous example. But our approach achieves bounds consistency only on the domains of Z. This is not the case for the variables of X and Y (cf. [288, Section 3.1.4]).

4.6 Summary

In this chapter, we have developed a generic filtering technique, based on a breadth-first search and depth-first search, called alternating breadth-first search and alternating depth-first search, that runs in linear time assuming that the appropriate matching is given. The algorithm can be efficiently used as a propagation routine when solving a global constraint representable by a bipartite graph.

We believe that this chapter illustrates that decomposition theory of bipartite graphs and the Dulmage-Mendelsohn Canonical Decomposition have a practical application for describing filtering algorithms for global constraints representable by bipartite graphs regardless of the underlying model used for the constraint (value, variable, or intersection graph, convex bipartite graph, and so on) and its solution (perfect, complete, maximum or optimal matching).

The following table summarizes time complexities of global constraints representable by a bipartite graph which were discussed in this chapter.

global constraint	model	checking feasibility	hyper-arc consistency	reference
ALLDIFFERENT	value graph	$\mathcal{O}(\sqrt{n} \cdot m)$	$\mathcal{O}(m + n + k)$	Régin [255]
ALLDIFFERENT_EXCEPT_0	value graph	$\mathcal{O}(\sqrt{n} \cdot m)$	$\mathcal{O}(m + n + k)$	here
CORRESPONDENCE	intersection graph	$\mathcal{O}(\sqrt{n} \cdot m)$	$\mathcal{O}(m + n)$	here
INVERSE	variable graph	$\mathcal{O}(\sqrt{n} \cdot m)$	$\mathcal{O}(m + n)$	here
SAME	intersection graph	$\mathcal{O}(\sqrt{n} \cdot m)$	$\mathcal{O}(m + n)$	here
USED_BY	intersection graph	$\mathcal{O}(\sqrt{n} \cdot m)$	$\mathcal{O}(m + n)$	here
GCC	variable-value graph	$\mathcal{O}(\sqrt{g(V)} \cdot m)$	$\mathcal{O}(m + n + k)$	Quimper [253]
SYMMETRIC_CARDINALITY	variable-value graph	$\mathcal{O}(\sqrt{g(V)} \cdot m)$	$\mathcal{O}(m + n + k)$	here
SOFT_ALLDIFFERENT_VAR	value graph	$\mathcal{O}(\sqrt{n} \cdot m)$	$\mathcal{O}(m + n + k)$	Petit [245]
SOFT_SAME_VAR	intersection graph	$\mathcal{O}(\sqrt{n} \cdot m)$	$\mathcal{O}(m + n)$	here
SOFT_USED_BY_VAR	intersection graph	$\mathcal{O}(\sqrt{n} \cdot m)$	$\mathcal{O}(m + n)$	here
SOFT_GCC_VAR	variable-value graph	$\mathcal{O}(\sqrt{g(V)} \cdot m)$	$\mathcal{O}(m + n + k)$	Zanarini [314]
SOFT_SYMMETRIC_ CARDINALITY_VAR	variable-value graph	$\mathcal{O}(\sqrt{g(V)} \cdot m)$	$\mathcal{O}(m + n + k)$	here

Table 4.6: Summary of results for bipartite graphs

Chapter 5

General Graphs

In the current chapter traditional results in matching theory and recently developed techniques in decomposition theory are conveniently combined and extended. We use the Gallai-Edmonds Structure Theorem to decompose a general graph with perfect matching into elementary subgraphs. This decomposition is a stronger version of the Dulmage-Mendelsohn Canonical Decomposition described in the previous chapter. In particular, we observe that the Gallai-Edmonds Decomposition applied to constraint programming will allow us to concentrate on the global constraints representable by general graphs with perfect matching.

In this chapter we present an algorithm for finding the partition of edges in general graphs. The algorithm has $\mathcal{O}(p \cdot m)$ time complexity, where p is the number of maximal extreme sets in G. Our algorithm is based on the notion of extreme sets introduced by Anton Kotzig [193–195]. The extreme sets will be defined for f-matchings.

The maximum matching problem in general graphs is one of the most fundamental problems in matching theory and has been investigated extensively. This problem was first formulated and solved by Edmonds [90] using his well-known blossom shrinking algorithm. Since a number of algorithms have been developed for this problem some of them are listed in Table 5.1 (cf. [264, Section 24.4a]).

Several algorithms have been proposed to solve the maximum matching problem on general graphs. These algorithms are considerably more intricate than for bipartite graphs. The first polynomial algorithm, called the *blossom shrinking algorithm*, for determining maximal matchings, based on Augmenting Path Theorem (Theorem 2.4.1), was initially found by Jack Edmonds [90] with a complexity of $\mathcal{O}(n^4)$, although he did not state this formally. The complexity of this algorithm has been improved from $\mathcal{O}(n^4)$ to $\mathcal{O}(n^3)$ by Gabow [114] and using union-find data structure to $\mathcal{O}(n \cdot m \cdot \alpha(m, n))$ by Tarjan [283], where α is the inverse of Ackermann's function, which is very slow growing. This can be further reduced to $\mathcal{O}(n \cdot m)$ time using the disjoint-set result of Gabow & Tarjan [121].

These algorithms follow the general paradigm for matching algorithms: repeated augmentation by augmenting paths until a maximum matching is obtained. We assume that the reader is familiar with this paradigm, which can, for example, be obtained by reading standard references [212, Section 9.1], [218, Section 7.6.2] or [285, Chapter 9].

Year	Author(s)	Complexity	Strategy/Remarks
1965	Edmonds [90]	$\mathcal{O}(n^4)$	blossom shrinking
1965	Witzgall & Zahn [307]	$\mathcal{O}(n^2 \cdot m)$	augmenting paths
1969	Balinski [17]	$\mathcal{O}(n^3)$	labeling technique
1973	Gabow [114]	$\mathcal{O}(n^3)$	labeling technique
1974	Kameda & Munro [176]	$\mathcal{O}(n \cdot m)$	better blossom handling
1975	Even & Kariv [97]	$\mathcal{O}(n^{5/2})$	extremely complicated
1976	Karzanov [184], Lawler [201]	$\mathcal{O}(n^3)$	no explicit shrinking
1976	Gabow [114] + Tarjan [283]	$\mathcal{O}(n \cdot m \cdot \alpha(m,n))$	disjoint set union
1980	Micali & Vazirani [221,302],[244]	$\mathcal{O}(\sqrt{n} \cdot m)$	shortest paths
1985	Gabow [114] + Tarjan [121]	$\mathcal{O}(n \cdot m)$	linear disjoint set union
1990	Blum [46,48]	$\mathcal{O}(\sqrt{n} \cdot m)$	reachability in digraphs
1991	Gabow & Tarjan [124]	$\mathcal{O}(\sqrt{n} \cdot m)$	ϵ-optimality
1995	Goldberg & Karzanov [139,140]	$\mathcal{O}(\sqrt{n} \cdot m \cdot \log_n \frac{n^2}{m})$	graph compression
2003	Fremuth-Paeger & Jungnickel [111]	$\mathcal{O}(\sqrt{n} \cdot m \cdot \log_n \frac{n^2}{m})$	graph compression
2004	Mucha & Sankowski [224]	$\mathcal{O}(n^\omega)$	matrix multiplication
2006	Harvey [154]	$\mathcal{O}(n^\omega)$	matrix multiplication

Table 5.1: History of algorithms for the cardinality non-bipartite matching problem

The fastest known algorithm with the best asymptotic running time is given in [221] by Silvio Micali and Vijay V. Vazirani. The algorithm generalizes the method of Hopcroft & Karp [164] for bipartite graphs. Although an extensive discussion of this algorithm was given in [244], a formal proof of the correctness appeared nearly 10 years after the algorithm had been designed [302]. Recently, Vazirani has written a new manuscript about his algorithm, which contains the simplified version of correctness proof [303].

The algorithm works in phases. In each phase a maximal set of vertex-disjoint shortest augmenting paths is found and the existing matching is increased along these paths. As in the bipartite case, it can be shown (see, for example, [4, Section 8.2], [164] or [217, Section IV.9.2]) that the number of phases is $\mathcal{O}(\sqrt{n})$ and hence the total running time is $\mathcal{O}(\sqrt{n} \cdot m)$. On many graphs the number of phases is smaller. In particular, Rajeev Motwani [223] has shown that the number of phases is $\mathcal{O}(\log n)$ for random graphs (see also [23]).

Unfortunately, the algorithm of Micali & Vazirani is rather difficult to understand and implement. The procedure they present turns out to be too complex for efficient implementation. In LEDA, the popular library framework for efficient data types and algorithms, developed by Kurt Mehlhorn and Stefan Näher [218], the algorithm of Gabow & Tarjan [285],[121] with running time $\mathcal{O}(n \cdot m)$, improved from $\mathcal{O}(n \cdot m \cdot \alpha(m,n))$, is used for the implementation of the maximum matching algorithm on general graphs. That applies to perfect matchings as well because a perfect matching is a matching of maximum size.

We conclude this section by remarking that while most of the results in this chapter have appeared before in [72], there are some new results. In particular, we present the degree-version of the divide-and-conquer approach, which we left as an exercise for the reader. In addition, some new additional properties of extreme sets are discovered (see Property 5.2.9 and 5.2.10). Theorem 5.3.4 is new. It clarifies the number of maximal extreme sets in a general graph, which was lacking in the paper. Additionally, we prove the correctness of the pruning algorithm for the SOFT_SYMMETRIC_ALLDIFFERENT_VAR constraint (see Lemma 5.4.4). An example describing the CLIQUE constraint is new. Section 5.5 is new.

5.1 Preliminaries

In order to make our thesis self-contained, we briefly introduce some terminology. This botanical flavor comes from [90]. The important concept is that of a *blossom* in a graph. Recall that blossoms play a key role in Edmonds' algorithm for maximum matchings and weighted perfect matchings in general graphs.

Let G be a graph and let M be a matching in G. A *blossom* B in G is an alternating path relative to M that forms a cycle of odd-length with $|M \cap E(B)| = \frac{|V(B)-1|}{2}$ matched edges. It is convenient to consider also the vertices of the graph as (trivial) blossoms of size one. Note that every blossom is a factor-critical subgraph of G, but the converse does not hold in general, since a blossom is not an induced subgraph. A vertex of B not covered by $M \cap E(B)$ is called the *base* of B.

A *shrunken blossom* results when a blossom B is collapsed into a single vertex b, the base of the blossom, whereby every edge $\{x, y\}$ with $x \notin B$ and $y \in B$ is transformed into the edge $\{x, b\}$. The reverse of this process gives an *expanded blossom*. More formally,

Shrunken graph Given a graph $G = (V, E)$ with a matching M and a blossom B, the shrunken graph G/B with matching M/B is defined as follows:

- $V(G/B) = (V \setminus B) \cup \{b\}$
- $E(G/B) = E \setminus E[B]$
- $M/B = M \setminus E[B]$

where $E[B]$ denotes the set of edges within B, and b is the base of B.

Observe that G/B may contain parallel edges between vertices, if G contains a vertex which is joined to B by more than one edge. If G/B is the graph obtained from G after shrinking the blossom B, then such an operation is justified by the following property:

Theorem 5.1.1 *There is an augmenting path in G iff there is an augmenting path in G/B.*

Proof See Lemma 9.1.1 in [212]. □

Two blossoms are either disjoint or one is contained in the other. In the latter case the first blossom is said to be nested in the second. Since a blossom is not an induced subgraph, we may have two vertices v and w belonging to the same blossom B without edge $\{v, w\}$ being in B. However, a simple analysis shows that except for one vertex $b \in B$, called the base of the blossom, for matched edge $\{u, v\}$ if u and v belong to some blossoms, then u and v belong to the same blossom.

A *stem* is an even alternating path from an exposed vertex to the base of B. The stem of the blossom may be empty. If the stem is not empty, it contains a matched edge incident to the base. The base of a blossom is exposed iff the stem is empty.

There are no blossoms in bipartite graphs because blossoms have an odd number of edges and there does not exist a cycle with an odd number of edges in a bipartite graph (see Theorem 2.5.1). In the case of a non-bipartite graph whenever we come across a blossom we reduce it to a single vertex. And any edge that does not belong to a blossom but is incident on one of its vertices is now incident on this new vertex formed by shrinking the blossom.

Figure 5.1 shows a blossom. The blossom consists of nine edges, four of which are matched. The base of the blossom is x_2. The stem of the blossom is the path $x_0x_1x_2$. Edge $\{x_3, x_8\}$ does not belong to this blossom, since there exists no alternating path in B involving this edge. We can see that vertices x_0 and x_{11} are exposed. So the augmenting path $x_0x_1x_2x_3x_5x_7x_9x_{11}$ will increase the size of the matching by 1.

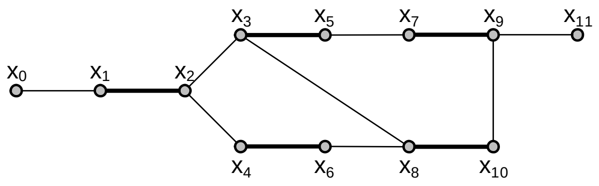


Figure 5.1: A blossom

Every vertex v in the blossom, except its base, is reachable from the root r, or from the base of the blossom, through two vertex-disjoint alternating paths; one has even length and the other has odd length. The even alternating path to vertex v terminates with a matched edge, and the odd alternating path to vertex v terminates with a free edge.

The following facts characterize blossoms:

- A stem spans $2l + 1$ vertices and contains l matched edges for some integer $l \geq 0$.
- A blossom spans $2k + 1$ vertices and contains k matched edges for some integer $k \geq 1$. The matched edges cover all vertices of the blossom except the base.
- The base of a blossom is always an outer (even) vertex.
- Every vertex in a blossom (except its base) is reachable from the root (or from the base) via two distinct alternating paths; one with an even and one with an odd length.
- An even alternating path in a blossom terminates with a matched edge and an odd alternating path terminates with a free edge.
- For a matched edge both its endpoints belong to the same blossom.
- Two blossoms are either disjoint or one is contained in the other.

A *rooted alternating forest* with respect to a given matching M is a forest F in G with the following properties:

- F contains all vertices not covered by M.
- Each connected component of F contains exactly one exposed vertex, its root.
- Each root is an outer (even) vertex.
- Every inner (odd) vertex is adjacent in the tree with an outer (even) vertex.
- All inner (odd) vertices have degree two in F.
- The unique path from any vertex x in F to the root of the connected component containing x is alternating.
- The number of outer vertices that are not a root equals the number of inner vertices.

Let G be a graph with degree conditions g and f, and M be a degree-matching in G. An f-blossom in G with respect to M is an f-critical subgraph B of G (cf. [153, Theorem 2.5]) with $|M \cap E(B)| = \frac{f(B)-1}{2}$ and the base b (see Figure 5.2).

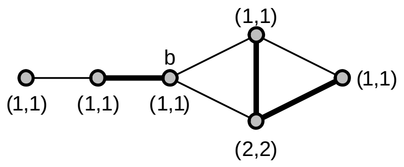


Figure 5.2: An f-blossom

5.2 Gallai-Edmonds Canonical Decomposition

In this section we investigate general graphs with degree-matchings. The obtained results are mainly based on the structure theorem of Gallai and Edmonds (for maximum matchings), and of Lovász and Plummer (for optimal degree-constrained matchings). This section is mainly expository in nature. At this point we collect some basic material that will be needed later on and include proofs of almost all the results.

5.2.1 General Graphs with Maximum Matchings

We now present a very important result in matching theory which can have useful applications in constraint programming. This is the so-called Gallai-Edmonds Structure Theorem. It was independently discovered by Tibor Gallai [127,128] (who published it in some Hungarian journals) and by Jack Edmonds [90]. Incidentally, their methods of proofs were quite different. The reader is referred to [212, Section 3.2] for a detailed discussion of this theorem, which characterizes the structure of maximum matchings in graphs. It is comforting to know that we can obtain this canonical decomposition efficiently, that is, in linear time, via the blossom shrinking algorithm.

First, let us determine the important properties of this canonical decomposition. Consider a simple graph $G = (V, E)$ with an initial maximum matching M. Then V can be decomposed into three disjoint subsets $V = A \cup C \cup D$, where

$D = \{$the set of vertices which are not covered by at least one maximum matching$\}$,

$A = \{$the set of vertices in $V \setminus D$ that are adjacent to at least one vertex in $D\}$,

$C = V \setminus (A \cup D)$.

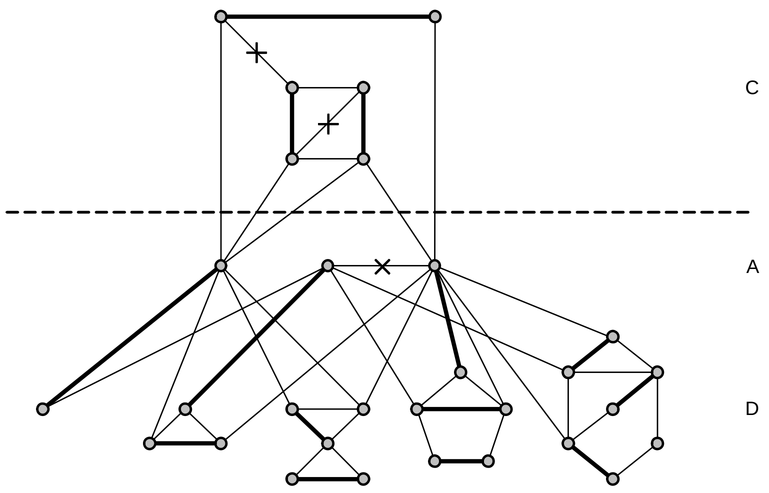


Figure 5.3: Gallai-Edmonds decomposition of a general graph with a maximum matching

Figure 5.3 illustrates an example of such a decomposition. Some properties of this decomposition are given in the following result.

Theorem 5.2.1 (Gallai-Edmonds Structure Theorem [127,128],[90]) *If G is a graph and A, C and D are defined as above, then the following statements hold:*

1. *The components of the subgraph induced by D are factor-critical,*
2. *The subgraph induced by C has a perfect matching,*
3. $A = \Gamma(D) \setminus D$,
4. *The bipartite multigraph obtained from G by deleting the vertices of C and the edges spanned by A, and by contracting each component of D to a single vertex has positive surplus (as viewed from A),*
5. *Every maximum matching of G splits into a perfect matching of each connected component of C, a near-perfect matching of each component of D, and a complete matching from A into distinct components of D.*

Proof See Theorem 3.2.1 in [212]. □

Some important consequences of this theorem are immediate:

- No edge spanned by A belongs to any maximum matching.
- No edge connecting A to C belongs to any maximum matching.
- Every edge incident with a vertex of D lies in some maximum matching (Lemma 2.6.1).
- There is no edge between C and D.
- Vertices of A and C are saturated by every maximum matching.
- Vertices of D are exposed by at least one maximum matching.
- Each connected component of C has even cardinality.
- Each connected component of D has odd cardinality.

Thus, the *Gallai-Edmonds Canonical Decomposition* is very useful to determine all edges belonging to no maximum matching. The partition of vertices can be obtained by applying Edmonds' blossom shrinking algorithm. Consider the alternating forest at the termination of the algorithm. The set of vertices which are either outer (even) vertices of the forest or inside shrunken outer vertices (blossoms) forms the set D. The set of inner (odd) vertices of the forest is the set A. The rest of the vertices of G (out of the forest) belongs to the set C. In summary, we have the following possibilities:

x is outer (even)	$\mathit{level}[x]$ is even and $\forall_{y \neq x}\, \mathit{blossom}(y) \neq \mathit{blossom}(x)$
x is inner (odd)	$\mathit{level}[x]$ is odd and $\forall_{y \neq x}\, \mathit{blossom}(y) \neq \mathit{blossom}(x)$
$x \in$ blossom	($\mathit{level}[x]$ is even or odd) and $\exists_{y \neq x}\, \mathit{blossom}(y) = \mathit{blossom}(x)$
out of forest	otherwise

5.2.2 General Graphs with Perfect Matchings

Note that if G has a perfect matching, then the Gallai-Edmonds decomposition gives no information about the allowed edges because the partition $\langle A, C, D\rangle$ is trivial with $C = V(G)$ and hence $A = D = \emptyset$. However, the linear-time method for finding the allowed edges incident to a vertex v extends to any graph that has a perfect matching as follows. Before we present such an algorithm we need some additional definitions. Our description is based on [212, Section 3.3].

We will call a set of vertices X in $V(G)$ *extreme* if the following equality holds

$$\delta(G - X) = \delta(G) + |X|.$$

Note that the empty set $\emptyset$ is always extreme. The singleton set may or may not be extreme in general. However, if G has a perfect matching then every singleton subset of V is extreme. We refer to such an extreme set as a *trivial* one.

An extreme set X of G is maximal if all components of $G - X$ are factor-critical. If G is bipartite with a perfect matching then both of its color classes form maximal extreme sets.

According to the Gallai-Edmonds Structure Theorem the set A is itself extreme (cf. [212, Exercise 9.1.2]) and the deficiency of G, $\delta(G) = t - |A|$, where t denotes the number of connected components of D. If $C \neq \emptyset$ then A is not a maximal extreme set. In general, A is not a unique extreme set and one of our concerns will be to characterize it among all extreme sets. The relation

$$\{(x, y) : x, y \in V \text{ and either } x = y \text{ or } \{x, y\} \text{ is extreme in } G\}$$

is an equivalence relation in elementary graphs. Finding a maximal extreme set can be accomplished in linear time. Now we point out the following few additional simple properties of extreme sets, which are fundamental for our purposes and are presented in [212, Section 3.3] as exercises for the reader. For the sake of completeness, we will prove some of them.

Property 5.2.1 *Every subset of an extreme set is extreme.*

Property 5.2.2 *If Y is extreme in G and $X \subseteq Y$ then $Y - X$ is extreme in $G - X$.*

Property 5.2.3 *If X is an extreme set in G and Z is an extreme set in $G - X$, then $X \cup Z$ is an extreme set in G.*

Proof Let X be any extreme set in G and Z be any extreme set in $G - X$. Then the following holds: $\delta(G - (X \cup Z)) = \delta(G - X - Z) = \delta(G - X) + |Z|$ (since Z is extreme in $G - X$) $= \delta(G) + |X| + |Z|$ (since X is extreme in G). The result follows now from the definition of an extreme set and the fact that $X \cap Z = \emptyset$. $\square$

Property 5.2.4 *A singleton set $\{v\} \subseteq V(G)$ is extreme iff $v \in A(G) \cup C(G)$.*

Property 5.2.5 *Let $e = xy$ be an edge in G and suppose G has a perfect matching. Then the set $\{x, y\}$ is extreme in G iff e lies in no perfect matching of G.*

Proof Let x and y be any two vertices in G. Then $G - x - y$ has no perfect matching iff $\delta(G - x - y) = 2$. But this in turn holds iff $\{x, y\}$ is extreme, that is $2 = \delta(G - x - y) = \delta(G) + |\{x, y\}| = 2$. $\square$

The last property immediately gives us a simple algorithm for edge partition. For each edge $\{x, y\}$, we temporarily delete it and test if the resulting graph $G - x - y$ still has a perfect matching. If so, we restore the edge; otherwise we remove the edge permanently. To compute the perfect matching we start from $M - \{\{x, mate(x)\}, \{y, mate(y)\}\}$ and search for an augmenting path with endpoints $mate(x)$ and $mate(y)$. To do this we need only $\mathcal{O}(m)$ operations. Since there are m edges in G, the complexity of the algorithm is $\mathcal{O}(m^2)$. We can improve this complexity by using the following result.

Proposition 5.2.2 *Let G be any graph and $x \in A(G) \cup C(G)$. Then $A(G - x) \cup \{x\}$ is extreme in G. Moreover, if $x \in C(G)$ and X is any extreme set in $G - x$, then $X \cup \{x\}$ is extreme in G.*

Proof The proof follows from Lemma 3.3.10 in [212] and stems from the fact that every barrier[1] is an extreme set (see Lemma 3.3.8 in [212]). □

We now give a structure theorem for perfect matchings, which generalizes the fundamental Gallai-Edmonds Structure Theorem for maximum matchings. In order to describe it more formally, we first need to introduce some terminology.

Assume that G has a perfect matching and let X be an extreme set in G. Let $G' = G - C(G-X)$, $t = |X|$, and let $D_1, \dots, D_t$ be the connected components of $D(G-X)$. Let G_0 denote the bipartite multigraph obtained from G' by contracting each component D_i to a single vertex and deleting each edge spanned by X. Let G_i be the multigraph obtained from G' by shrinking all vertices of $V(G) - D_i$ to a single vertex. We call G_0 the *gluing bipartite multigraph* and the multigraphs $G_1, \dots, G_t$ the *pieces* of G at extreme set X (these terms come from [211]).

We will now prove a very useful theorem which can have an application to propagation of global constraints representable by a perfect matching in a general graph.

Theorem 5.2.3 *Let $G = (V, E)$ be any graph with a perfect matching M, $v \in V$, and let $\langle A, C, D\rangle$ be the Gallai-Edmonds Canonical Decomposition of $G - v$. Then the following holds:*

1. *The set $X = A \cup \{v\}$ is extreme in G,*
2. *The edges in $G[X]$ are forbidden in G,*
3. *The edges in $\nabla(X, C)$ are forbidden in G,*
4. *The bipartite multigraph G_0 obtained from $G - C$ by contracting each connected component of D to a single vertex and by deleting each edge spanned by X has a perfect matching,*
5. *The multigraph G_i obtained from $G - C$ by contracting the set $V(G) - D_i$ to a single vertex has a perfect matching,*
6. *The mandatory, allowed, or forbidden edges of G are precisely those edges which are, respectively, mandatory, allowed, or forbidden in one of the graphs G_i, $i = 0, \dots, t$, where $t = |X|$.*

Proof Let M be a perfect matching of G and let $\langle A, C, D\rangle$ be the Gallai-Edmonds Canonical Decomposition of $G - v$. Then the theorem can be proven in the following way:

1. Since every vertex is a (trivial) extreme set and the set A of the Gallai-Edmonds decomposition of G is always extreme, the claim follows immediately from Property 5.2.3.
2. According to the Gallai-Edmonds Structure Theorem, every maximum matching contains a complete matching from A into distinct components of D. Since M matches all vertices of A with all connected components of D, thus M cannot contain any edge spanned by A.

[1]A *barrier* is a set $X \subseteq V$ such that the number of odd components of $G - X$ is equal to $|X| + \delta(G)$.

3. It can be proven analogously as Claim 2.

4. According to the Gallai-Edmonds Structure Theorem, D consists of factor-critical components and every maximum matching of G contains a complete matching from A into distinct components of D. Since X is an extreme set in G, thus it must be precisely $|X|$ connected components of D. So each perfect matching of G is mapped onto a perfect matching of G_0.

5. Analogously, as for Claim 4.

6. It follows from Claims 4 and 5.

The theorem is established. □

This theorem describes a way to decompose general graphs with perfect matchings into matching covered (1-extendable) subgraphs. It is instructive to study the example shown in Figure 5.4, which demonstrates how the decomposition by means of extreme set works (the non-trivial blossoms are circled).

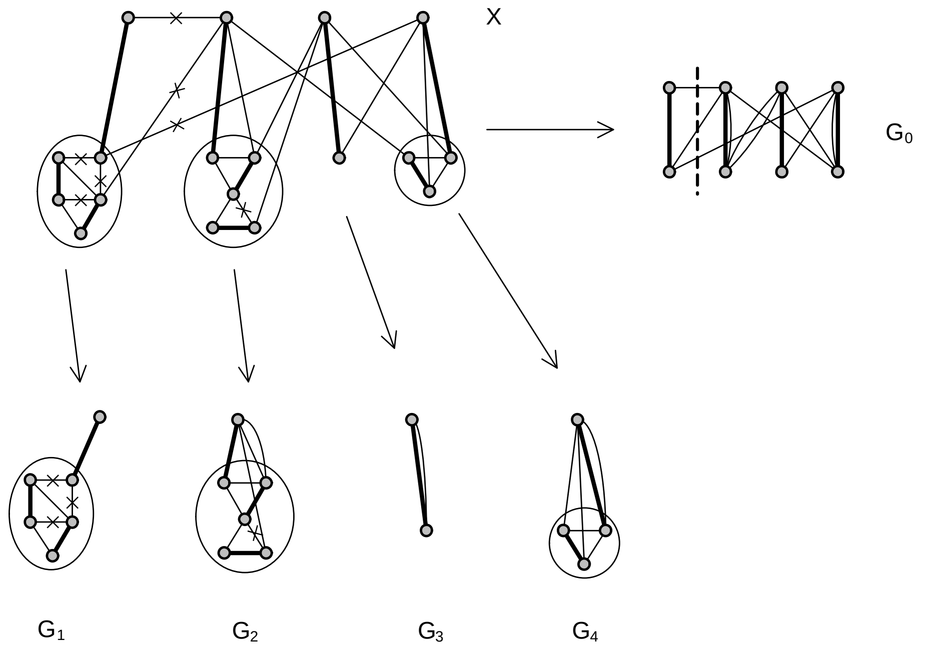


Figure 5.4: The canonical decomposition of a general graph with a perfect matching

The reader is invited to convince himself about the truth of the following very simple result:

Lemma 5.2.4 *Any matched edge in the gluing bipartite multigraph G_0 is incident with the base of the respective blossom.*

Thus, the connected components of D are the blossoms and the gluing bipartite multigraph G_0 has the form $G[\nabla(A, base(D_i))]$, where $base(D_i)$ denotes the base of the blossom D_i, $i = 1, \ldots, t$, $t = |X|$.

5.2.3 General Graphs with Degree-Matchings

For a general graph G with an optimal degree-matching we define subsets A, B, C and D of $V(G)$ as follows

$$D = \{v \in V | \text{there exist both even and odd alternating trails from exposed vertices to } v\}$$
$$B = \{v \in V | \text{there exists an even alternating trail from some exposed vertex to } v\} \setminus D$$
$$A = \{v \in V | \text{there exists an odd alternating trail from some exposed vertex to } v\} \setminus D$$
$$C = V \setminus (A \cup B \cup D)$$

Clearly, A, B, C and D are pairwise disjoint. We call the partition $\langle A, B, C, D\rangle$ of V the *Lovász-Plummer Canonical Decomposition*. It is the generalization of the Gallai-Edmonds Canonical Decomposition for maximum matchings. Note that $C = V(G)$ iff a graph G has a perfect (g, f)-matching. The decomposition has the following properties (see Theorem 10.2.13 in [212]):

- No edge connecting A to C or spanned by A belongs to any optimal (g, f)-matching
- Each edge connecting B to C or spanned by B belongs to every optimal (g, f)-matching
- There is no edge connecting C to D
- Vertices of $A \cup C$ are saturated by every optimal (g, f)-matching
- Vertices of $B \cup D$ are exposed by at least one optimal (g, f)-matching

As an example, consider the graph G depicted in Figure 5.5 and its optimal f-matching (this illustration is adapted from [110]). Here, $f \equiv 3$. Observe that $\delta(G) = 1$.

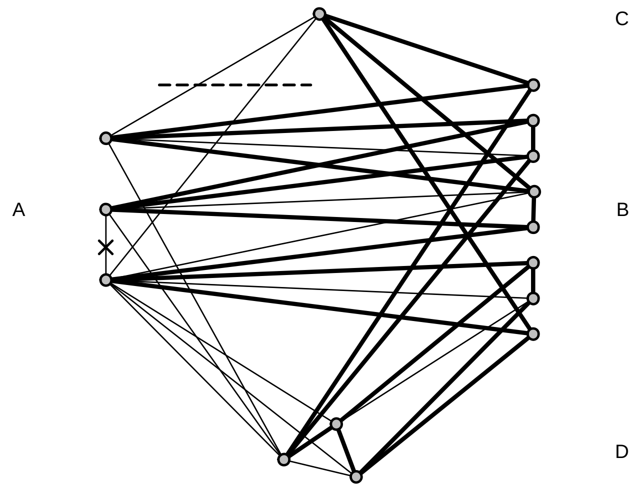


Figure 5.5: Lovász-Plummer decomposition of a general graph with an optimal f-matching

Theorem 5.2.5 *The following holds for every optimal (g, f)-matching M of G:*

(a) If $x \in A$, then $d_M(x) \geq f(x)$,

(b) If $x \in B$, then $d_M(x) \leq g(x)$,

(c) If $x \in C$, then $g(x) \leq d_M(x) \leq f(x)$,

(d) If $x \in D$, then $g(x) = f(x)$ and $f(x) - 1 \leq d_M(x) \leq f(x) + 1$.

Proof See Theorem 10.2.12 in [212]. □

We are now ready to formulate the main structure theorem which establishes the canonical decomposition of general graphs with degree-matchings. Let us define, using a given pair of degree conditions (g, f), further pair $(\hat{g}, \hat{f})$ as follows:

$$\hat{g}(x) = \begin{cases} f(x), & \text{for every } x \in A \\ g(x) \dot{-} |\nabla(x, B \cup C \cup D)|, & \text{for every } x \in B \\ g(x) \dot{-} |\nabla(x, B)|, & \text{for every } x \in C \cup D \end{cases}$$

$$\hat{f}(x) = \begin{cases} f(x), & \text{for every } x \in A \\ f(x) \dot{-} |\nabla(x, B \cup C)|, & \text{for every } x \in B \\ f(x) \dot{-} |\nabla(x, B)|, & \text{for every } x \in C \cup D \end{cases}$$

Theorem 5.2.6 *Let G be any graph, $\langle A, B, C, D\rangle$ – the Canonical Decomposition and let $\hat{g}, \hat{f}$ be defined as above. Then the following statements hold:*

1. *The connected components of $G[C]$ admit perfect $(\hat{g}, \hat{f})$-matching,*

2. *The connected components of $G[D]$ are $(\hat{g}, \hat{f})$-critical,*

3. *Every optimal (g, f)-matching of $G[A \cup B \cup D]$ saturates A,*

4. *The bipartite multigraph $G[A \cup B \cup D]$ obtained from G by deleting the vertices of $G[C]$ and the edges joining B and D or spanned by A or B and by contracting each component of $G[D]$ to a single vertex has positive surplus (viewed from A),*

5. *Every optimal (g, f)-matching of G splits into a perfect $(\hat{g}, \hat{f})$-matching of $G[C]$, a near-perfect $(\hat{g}, \hat{f})$-matching of the connected components of $G[D]$, and a complete $(\hat{g}, \hat{f})$-matching from A to $B \cup D$.*

Proof See Theorem 10.2.18 in [212]. □

Note that every component of $G[D]$ is (g, f)-critical. So if G is a bipartite graph, then $D = \emptyset$. Moreover, the set B is the union of all singleton (trivial) components of D in the Gallai-Edmonds decomposition.

Let G^* be an incremental graph with a maximum matching M^* (which corresponds to an optimal degree-matching in the graph G). If we have found the canonical decomposition of $V(G^*)$ it is possible to obtain the canonical decomposition of $V(G)$ in the following way.

Theorem 5.2.7 *Let G be any graph with an optimal degree-matching M, G^* be an incremental graph of G, and let $\langle A^*, B^*, C^*, D^*\rangle$ be the Gallai-Edmonds Decomposition of G^*. Then the Lovász-Plummer Decomposition $\langle A, B, C, D\rangle$ of G can be determined as follows*

(a) $x \in A$, if every external vertex in the gadget of x belongs to A^* or C^*,

(b) $x \in B$, if every external vertex in the gadget of x belongs to B^* or C^*,

(c) $x \in C$, if every vertex in the gadget of x belongs to C^*,

(d) $x \in D$, otherwise.

Proof We will prove our theorem for the first case as proof for the other cases is almost identical. Consider a vertex x and its corresponding gadget. According to the definition of the set A^* there exists an odd alternating path from an exposed vertex to the vertices of A^*. It is now not difficult to check that every alternating path in the incremental graph G^* corresponds exactly to an alternating trail in the original graph G (by contracting each gadget to a single vertex), and conversely, any alternating trail in G can be easily converted into an alternating path in G^* (by expanding every vertex of the trail to a gadget). Since an odd alternating path from an exposed vertex to an external vertex in the gadget of x corresponds to an odd alternating trail from an exposed vertex to x then, according to the definition of the set A, $x \in A$. Observe that some external vertices in the gadget of x may belong to C^* but this fact has no influence on the category of x. $\square$

Let G^* be an incremental graph with a perfect matching M^* (which corresponds to a perfect (g, f)-matching in the graph G). The set X is said to be extreme with respect to (g, f)-matchings if the set X^* is extreme in the incremental graph G^*. Here, X^* denotes the union of all external vertices belonging to a gadget corresponding to every $x \in X$ of $V(G)$.

In particular, a set of vertices X in $V(G)$ is *extreme* with respect to f-matchings if the following equality holds

$$\delta(G - X) = \delta(G) + f(X).$$

Unfortunately, the situation here is more complicated than it is for extreme sets in the case of the ordinary matching. The difficulty in processing extreme sets with respect to f-matchings is due to the fact that any singleton set must not be in general extreme. For example, in Figure 5.12, the singleton $\{1\}$ is not extreme with respect to 2-matchings, and neither $\{4\}$, $\{6\}$, $\{7\}$ nor $\{8\}$ are such.

We now point out a few simple properties of extreme sets with respect to f-matchings.

Property 5.2.6 *Every subset of an extreme set is extreme.*

Property 5.2.7 *Let G be any graph with a perfect f-matching. Assume that $\{x\}$ is not extreme in G. Then x belongs to no extreme set.*

Proof Clearly, $\delta(G - x) < f(x)$ (cf. Lemma 3.3.1 in [212]). Assume that $\{x, y\}$ is extreme in G. Then we have $\delta(G-\{x, y\}) = f(x)+f(y)$. On the other hand, we have $\delta(G-\{x, y\}) \leq \delta(G - x) + f(y) < f(x) + f(y)$, which is a contradiction to our hypothesis. Thus, x belongs to no extreme set. $\square$

For (g, f)-matchings the situation is considerably more complex. In fact, the above property does not hold (cf. Figure 5.10). An alternative way is based on a reduction to the f-matching problem.

It is sensible to try to apply matching theory to the theory of f-matchings. We now define 2-bicritical graphs and give several characterizations of them. A graph G is said to be *2-bicritical* if $G - v$ contains a perfect 2-matching for every vertex $v \in V(G)$. It turns out that the properties of 2-bicritical graphs seem to be analogous to the properties of bicritical graphs. For example, if G is a 2-bicritical graph then G has a perfect 2-matching (see Corollary 6.2.2 in [212]), and every edge of G is allowed (see Theorem 6.2.9 in [212]). This result can be generalized to f-matchings.

Property 5.2.8 *If graph G with a perfect f-matching has no extreme sets then all edges of G are allowed.*

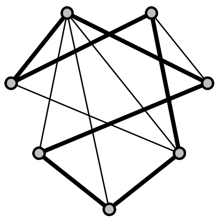


Figure 5.6: 2-bicritical graph

From [185, Theorem 3] we can deduce the following property (cf. [107]):

Property 5.2.9 *If graph G has no extreme sets with respect to a certain f-matching, such that $\delta(G - x) = 0$ holds for every vertex x, then it has a perfect f-matching.*

We now prove the following property from which Property 5.2.8 can be easily obtained.

Property 5.2.10 *Every edge connecting two vertices belonging to no extreme set is allowed.*

Proof We show how finding two odd augmenting trails reduces to finding a closed alternating trail of even length. Let $e = \{x, y\}$ be an edge connecting vertices x and y belonging to no extreme set. Since set $\{x\}$ is not extreme, thus there exists in $G - x$ an augmenting trail from some x' and x'' matched with x in the original graph G. Analogously, there exists in $G - y$ an augmenting trail from some y' and y''. Without loss of generality, we can assume that there is an odd alternating trail from x' to y' in G; otherwise, the vertices can be exchanged. Under assumption that G is 2-connected, such an alternating trail must exist, because all the remaining vertices are saturated. We can complete this alternating trail to a closed alternating trail of even length through edge e and two adjacent edges incident to x and y, i.e. $\{x, x'\}$ and $\{y, y'\}$. $\square$

The following result is a generalization of Theorem 5.2.3. Note that in our presentation, it is a straightforward consequence of the preceding results (see also [163]).

Theorem 5.2.8 (Decomposition into matching covered subgraphs) *Let $G = (V, E)$ be any graph with a perfect (g, f)-matching M, S – an extreme set of G, $\langle A, B, C, D\rangle$ – Lovász-Plummer Decomposition of $G - S$, and let D_i be connected components of D. Define the degree conditions $(\bar{g}, \bar{f})$ by*

$$\bar{g}(x) = \begin{cases} g(x), & \text{for every } x \in A \\ g(x) \dot{-} |\nabla(x, B \cup C \cup D)|, & \text{for every } x \in B \\ g(x) \dot{-} |\nabla(x, B)|, & \text{for every } x \in C \\ 1 \dot{-} |\nabla(D_i, B)|, & \text{for each connected component } D_i \end{cases}$$

$$\bar{f}(x) = \begin{cases} f(x), & \text{for every } x \in A \\ \min\{|\nabla(x, A \cup S)|, f(x) \dot{-} |\nabla(x, B \cup C)|\}, & \text{for every } x \in B \\ f(x) \dot{-} |\nabla(x, B)|, & \text{for every } x \in C \\ 1, & \text{for each connected component } D_i \end{cases}$$

Then the following holds:

1. *The set $X = A \cup S$ is extreme in G,*
2. *The edges in $G[X]$ are forbidden in G,*
3. *The edges in $\nabla(X, C)$ are forbidden in G,*
4. *The bipartite multigraph G_0 obtained from $G - C$ by contracting each connected component of D to a single vertex and by deleting each edge spanned by X or B, and removing all edges joining B and D has a perfect $(\bar{g}, \bar{f})$-matching,*
5. *The edges mandatory, allowed or forbidden in the gluing bipartite multigraph G_0 are, respectively, mandatory, allowed or forbidden edges in G,*
6. *Let T_i, $i = 1, \ldots, t$, where $t \leq |X|$, be connected components of $G - C - X$. The multigraph G_i obtained from $G - C$ by contracting the set $V(G) - T_i$ to a single vertex, has a perfect (g, f)-matching,*
7. *The edges mandatory, allowed or forbidden in the pieces of G at extreme set X are precisely those edges which are, respectively, mandatory, allowed or forbidden in G.*

Observe that this theorem is general and, for $g \equiv f \equiv 1$, it is equivalent to Theorem 5.2.3. We illustrate the theorem using the graph in Figure 5.7, where $g \equiv f \equiv 2$.

5.3 Computing the partition of vertices and edges

Suppose we have a maximum matching M in a graph $G = (V, E)$ and we want to search for all edges of the graph which do not appear in any maximum matching. An additional step would be to identify the edges that participate in every maximum matching. The overall aim is to establish a partition of the edge set E into three disjoint subsets:

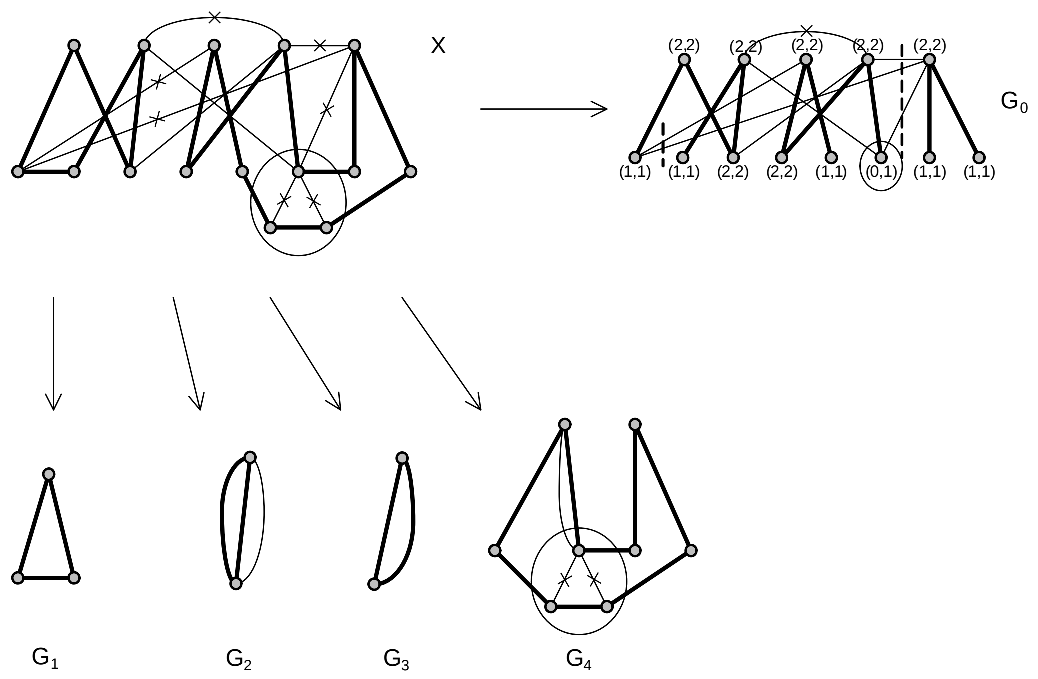


Figure 5.7: The canonical decomposition of a general graph with a perfect f-matching

- the set of edges belonging to no maximum matching (forbidden edges),
- the set of edges not belonging to at least one maximum matching (allowed edges),
- the set of edges belonging to all the maximum matchings (mandatory edges).

5.3.1 Partition of vertices

Our goal now is to implement the Gallai-Edmonds decomposition algorithm. As we have already mentioned, it is known that using the method of [115], the most efficient implementation of the algorithm for computing degree-matchings runs in $\mathcal{O}(\sqrt{f(V)} \cdot m)$ time. In order to find a perfect (or optimal) degree-matching at most $\sqrt{f(V)}$ phases are needed, and each phase is of complexity $\mathcal{O}(m)$. This algorithm does not only find a degree-matching for G, but it also constructs the Gallai-Edmonds decomposition of G.

There is a strong relationship between this algorithm and the Gallai-Edmonds Structure Theorem. This presentation is inspired by [218, Section 7.7.2] and [285, Chapter 9]. Given any matching M, the algorithm assigns labels to the vertices as follows. Every vertex is labeled as either `EVEN`, `ODD` or `UNLABELED`. A vertex is labeled `UNLABELED` (or out-of-forest) if it does not belong to any alternating tree and it is labeled `EVEN` or `ODD` otherwise. A vertex v is labeled `EVEN` (or outer) if there is an alternating trail of even length from an exposed vertex to v, and vertex v is labeled `ODD` (or inner) if there is an odd alternating trail from an exposed vertex to v. Clearly, if M is any maximum matching, then this labeling corresponds to the Gallai-Edmonds decomposition (see [212, Section 3.2]). The set of vertices belonging to any (proper) blossom is the set D, the sets of vertices labeled exclusively `EVEN` or `ODD` are the sets B and A, respectively, and the set of `UNLABELED` vertices is the set C.

Motivated by this description we obtain the following procedure:

Algorithm 8 Computing the Gallai-Edmonds Decomposition of G

```
Require: General graph $G = (V, E)$ with an initial maximum matching $M$
Ensure: Partition of vertices
  Make one iteration of blossom shrinking algorithm starting from exposed vertices
  Find the set $C$ of unlabeled vertices
  Let $B$ be the set of even vertices not belonging to any blossom
  Let $A$ be the set of odd vertices not belonging to any blossom
  Let $D$ be the set of vertices belonging to any blossom
  The Gallai-Edmonds Decomposition is given by $\langle A, B, C, D\rangle$
```

The next theorem is an immediate result:

Theorem 5.3.1 *The partition of vertices in a graph with an initial maximum matching can be determined in linear time.*

Proof One iteration of the blossom shrinking algorithm takes $\mathcal{O}(m)$ time. This can be seen as follows. Let M be an initial maximum matching in a graph G. Recall that the blossom shrinking algorithm is based on the breadth-first search strategy. During the breadth-first search (with respect to M) each edge is examined at most twice (in the bipartite case, only once). This gives the required complexity of $\mathcal{O}(m)$. Classification of vertices takes $\mathcal{O}(n)$ time. This leads to an overall linear complexity in the number of edges and vertices. □

5.3.2 Partition of edges

Theorems 5.2.3 and 5.2.8 suggest an algorithm for computing the partition of edges and we are ready to present the filtering method. Our algorithm is based on a simple, but subtle, *divide-and-conquer* approach [1, Section 2.6],[2, Section 10.1],[67, Section 2.3]. Recall that divide-and-conquer algorithms consist of two parts:

- *Divide:* Smaller subproblems are solved recursively and independently;
- *Conquer:* The solution to the original problem is combined from the solutions to the subproblems.

The algorithm has the following form (at the start of the algorithm all vertices are marked as UNSCANNED and all edges as UNTRAVERSED):

Let us describe our algorithm in more detail. Each perfect matching of G is mapped into a perfect matching of G_0 by the contraction of every connected component of D to a single vertex, being the base of the blossom. So the mandatory, allowed or forbidden edges of G, if not contracted, correspond to mandatory, allowed and forbidden edges in G_0, respectively. Every perfect matching of G consists of a perfect matching of each connected component of

Algorithm 9 The divide-and-conquer approach to determine the partition of edges

```
Require: General graph $G = (V, E)$ with an initial perfect $(g, f)$-matching $M$
Ensure: Partition of edges
  repeat {find extreme set}
    Choose one UNSCANNED vertex $v$
    Relabel $v$ as SCANNED
    Compute the Gallai-Edmonds Decomposition $\langle A, B, C, D\rangle$ of $G - v$ (see Algorithm 8)
  until $\{v\}$ is extreme or all vertices are SCANNED
  if there are no extreme sets in $G$ then
    Mark all edges of $G$ as TRAVERSED
    Mark all edges of $G$ as ALLOWED (see Property 5.2.8)
    return
  end if
  Let $X = A \cup \{v\}$ {extreme set}
  Determine the degree conditions $\bar{g}$ and $\bar{f}$ (see Theorem 5.2.8)
  Find connected components $C_1, C_2, \ldots, C_r$ of $G[C]$
  for every connected component $C_i$ with at least one UNSCANNED vertex do
    Let $M_i = M \cap E(C_i)$
    Recursive call of this procedure with $G = C_i$, $M = M_i$, $g = \bar{g}$ and $f = \bar{f}$
  end for
  Form the gluing bipartite multigraph $G_0$ with bipartition $(X, B \cup base(D_i))$
  Let $M_0 = M \cap E(G_0)$
  Perform an alternating depth-first search (with respect to $M_0$) on $G_0$
  Determine the partition of edges in $G_0$ (see Algorithm 4 and 5)
  Remove forbidden edges from $G$
  Mark all vertices of $X$ as SCANNED
  Mark all edges in $G_0$ as TRAVERSED
  Mark vertices incident with all TRAVERSED edges as SCANNED
  Let $t$ be the number of connected components of $G[B \cup D]$
  Form the pieces $G_1, G_2, \ldots, G_t$ of $G$ at extreme set $X$
  for every piece $G_i$ with at least one UNSCANNED vertex do
    Let $M_i = M \cap E(G_i)$
    Recursive call of this procedure with $G = G_i$, $M = M_i$, $g$ and $f$
  end for
```

$G[C]$, a perfect matching of the gluing bipartite multigraph G_0, and a perfect matching of each piece of G at extreme set X.

We use a depth-first search to traverse the multigraph G_0 and to look for alternating cycles (cf. Corollary 2.4.3) and paths. This step of our routine is a slightly modified version of the algorithm presented in Section 4.3 for a related problem. We choose any unvisited vertex belonging to $B \cup base(D_i)$ and form a depth-first tree starting from a matched edge adjacent to this vertex. The depth-first tree grows vertex by vertex in the following manner. The vertices on the odd level are simply given by the mates of the vertices from the previous even level. The vertices on the even level are non-scanned neighbors of vertices from the previous odd level. Thus, edges alternate from the matched edge on the even level to the free edge on the odd level. If a tree edge on the even level encounters a vertex belonging to a blossom, then the search jumps to the base of the blossom. On the other hand, if a non-tree edge leads to a gray vertex an alternating cycle is discovered. Then we shrink the vertices of the alternating cycle to a single pseudovertex, in the same way as in the shrinking blossom algorithm. If the vertices in each alternating cycle are contracted into a single vertex, the remaining edges in the depth-first tree form an alternating path. A very important property of our algorithm is the fact that we can run it without constructing the bipartite multigraph G_0 explicitly.

The following results verify that our algorithm works correctly.

Theorem 5.3.2 *An alternating cycle is discovered by a non-tree edge leading to a blossom with the gray base.*

Proof Let a non-tree edge $\{v, w\}$ leads to a blossom B with the gray base b. Then the vertex v is an ancestor of the vertex b in the depth-first tree. Thus, there exists an alternating path from b to v. Since the vertex w belongs to the blossom B with the base b, there exists an alternating path from w to b. The non-tree edge $\{v, w\}$ completes the alternating cycle $b..vw..b$. □

Theorem 5.3.3 *Every non-tree edge leading to a blossom with the black base is forbidden.*

Proof This follows from the fact that there exists no alternating cycle including this edge. □

Observe that the worst time complexity of our algorithm is $\mathcal{O}(n \cdot m)$ in the case when the input graph G is bicritical (cf. [212, Theorem 5.2.5]). The same complexity will be achieved when the graph is 2-bicritical. Clearly, the best time complexity of this algorithm is $\mathcal{O}(m)$ when the initial graph has only two disjoint maximal extreme sets. In general holds:

Theorem 5.3.4 *If G is a non-elementary graph, then the maximal extreme set of G is the union of some maximal extreme sets of the elementary components of G.*

Proof This is an immediate consequence of the properties of maximal extreme sets of elementary graphs: the maximal extreme set of G is the product of the maximal extreme sets of the elementary components of G. □

Theorem 5.3.5 *Let p denote the number of maximal extreme sets in a graph. Then the partition of edges can be determined in $\mathcal{O}(p \cdot m)$ time.*

Proof We assume that if a graph in the theorem is not connected, we apply this proof to each of its connected components. Let M be an initial perfect matching in G, and let $\langle A, B, C, D \rangle$ be the Gallai-Edmonds Decomposition of $G - v$. Since we consider maximal extreme sets we can assume that $C = \emptyset$. Recall that for an elementary graph G (not necessarily matching covered), the canonical partition of maximal extreme sets forms an equivalence class of its vertex set. We know that an edge $\{x, y\}$ is allowed in G iff x and y belong to different classes of the canonical partition (see Theorem 5.2.2 (b) in [212]). On the other hand, every edge induced by the vertices of the same class is forbidden (see Property 5.2.5). Thus, in order to detect all these edges we need at most p iterations of the blossom shrinking algorithm. In particular, we need only one iteration when G precisely has two maximal extreme sets (e.g. if G is bipartite), and we need $n - 1$ iterations when all classes of G are singletons (e.g. if G is bicritical). We show this in a more formal way.

1. $p = 2$ (the best case). Here, we have $V(G_0) = V(G)$. In this situation we need only to perform two graph traversals in order to compute the partition of edges. The first traversal is the breadth-first search to find the extreme set X. The second traversal is a modified version of the alternating depth-first search on G_0 with respect to M.

2. $p = n$ (the worst case). Here, we have $X^{(i)} = \{v_i\}$, $V(G_0^{(i)}) = \{v_i, mate(v_i)\}$, $E(G_0^{(i)}) = \nabla(v_i)$ and $G_1^{(i)} = G$ for $i = 1, \ldots, n$. In this situation we need to call $n - 1$ times the blossom shrinking algorithm. Since in every step $|V(G_0)| = 2$, $|E(G_0)| = d_G(v_i)$, and $\sum d_G(v_i) = 2m$ (see Lemma 2.3.1) we have to traverse every edge of G at most twice, in order to compute the partition of edges.

3. In the average case for every step the following holds: $|V(G_0)| < \sum |V(G_i)| = |V(G)|$ and $|E(G_0)| < |\sum |E(G_i)| \leq |E(G)|$ for $i = 1, \ldots, t$. Every step reduces the task of finding a partition of edges in G to the task of finding a partition of edges in the smaller graph G_i, for $i = 0, 1, \ldots, t$. Recall that for any non-elementary graph, the family of maximal extreme sets never gives a partition of its vertex set. However, since each elementary subgraph of G is determined by all the allowed edges of G we can assume, without loss of generality, that G is elementary. Observe that, in elementary graphs, the notion of maximal extreme sets and extreme sets coincide (see Lemma 5.1.1 and Theorem 5.1.3 in [212]). If every edge of G has been traversed, then we claim that the partition of edges is already determined. This can be realized for every (maximal) extreme set. Since every case can be accomplished in linear time by Theorem 5.3.1, the total running time is $\mathcal{O}(p \cdot m)$.

These observations yield the assertion about the complexity. □

Let G be an elementary graph with p (maximal) extreme sets. The above theorem intuitively means that we have to call at most $p - 1$ times the blossom shrinking algorithm in order to classify all edges of G. Observe further that the canonical partition of a matching covered graph canonically decomposes it into a p-partite graph.

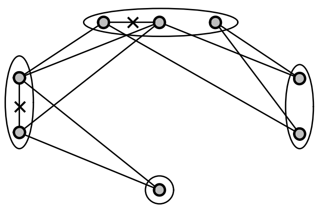


Figure 5.8: The canonical partition of an elementary graph

In this section we have analyzed a polynomial time algorithm for a partition of edges in a general graph. The question of whether there exists a linear time algorithm remains open.

5.4 Application to Global Constraints

In order to define global constraints representable by general graphs we first introduce a graph associated with any instance of these constraints. We assume that the variables and their domain values represent the same set of elements. Let $X = \{x_1, x_2, \ldots, x_n\}$ be a set of n variables with respective finite domains $D_{x_i} \subseteq \{0, 1, 2, \ldots, n\}$ for $i = 1, 2, \ldots, n$. To these variables we can associate the graph $G = (V, E)$, called a *value graph*, with vertex set $V = \{v_i : 1 \leq i \leq n\}$ and edge set $E = \{\{v_i, v_j\} : i \in D_{x_j} \wedge j \in D_{x_i}, 1 \leq i \leq n\}$. Clearly, $n = |V|$, $m = |E| \leq \frac{1}{2} \sum |D_{x_i}|$, and $d(v_i) \leq |D_{x_i}|$ for all $x_i \in X$.

Observe that the following property must hold for the constraint: if we have a connection from the vertex v_i to the vertex v_j then we must have also a connection from the vertex v_j to the vertex v_i. Hence, an edge $\{v_i, v_j\}$ exists iff j is in the domain of variable x_i and i is in the domain of variable x_j. Moreover, elimination of an edge $\{v_i, v_j\}$ from the associated graph during the pruning means elimination of the value j from the domain of variable x_i and elimination of the value i from the domain of the variable x_j. Without loss of generality, we can identify vertex v_i with value i.

The domains have to be first normalized. Suppose that there exists a value $j \in D_i$, while $i \notin D_j$. Then we can immediately remove value j from D_i. Hence, we assume that such situations do not occur during the creating of the corresponding graph, unless otherwise stated.

We now provide the skeleton of our general propagation routine (see Algorithm 10). The details of each part of the routine will be explained in the examples (cf. Algorithm 6).

Our routine first builds up an auxiliary graph associated with a global constraint, then constructs an appropriate matching, then decomposes the graph according to the Gallai-Edmonds Decomposition (Theorem 5.2.1), and successively identifies allowed edges and eliminates forbidden edges, reducing the remainder global constraint in a suitable way.

We now demonstrate our idea on concrete examples. Some of the examples are illustrated by figures. On the left half of the figure the domains are given. Next, the graph representing

Algorithm 10 Propagation routine for global constraints representable by general graphs

```
Normalize the domains of the variables {quick elimination}
Create an auxiliary graph $G$ associated with the global constraint
Compute an optimal degree-matching $M$ in $G$
If appropriate matching does not exist then return FALSE {constraint inconsistent}
Find the partition of mandatory, allowed and forbidden edges (see Algorithm 9)
Prune the domains of the variables according to the partition of edges
If any of the domains become empty then return FALSE
return TRUE
```

the constraint is shown. On the right half of the figure the canonical decomposition is depicted and the reduced domains are presented. Blossoms are denoted by a list of their vertices (enclosed within curly brackets { and }). The base of the blossom is on the first position in the sequence. We only show the first step of the decomposition and leave the details of the next steps as an exercise for the reader.

SYMMETRIC_ALLDIFFERENT In order to keep this work self-contained, we briefly recall the well-known global constraint SYMMETRIC_ALLDIFFERENT. Our description is based on [257] and [300].

The SYMMETRIC ALLDIFFERENT constraint is a particular case of the ALLDIFFERENT constraint and was introduced by Jean-Charles Régin [257]. This constraint is useful to be able to express certain items that should be grouped as pairs, for example in the problems of round-robin tournament scheduling or rostering. It is referenced under the name ONE_FACTOR in [161] and [290]. The constraint states that all variables must take different values, and if the variable representing element i is assigned to the value representing element j, then the variable representing element j must be assigned to the value representing element i. A more formal definition is presented below.

$$\text{SYMMETRIC\_ALLDIFFERENT}(x_1,\dots,x_n) = \{(d_1,\dots,d_n) \in D_{x_1} \times \cdots \times D_{x_n} \mid \underset{\substack{i,j\\ i\neq j}}{\forall} (d_i = j \Leftrightarrow d_j = i)\}.$$

We propose the following algorithm to achieve hyper-arc consistency. First, we compute a maximum matching M in the graph G. This can be done in $\mathcal{O}(\sqrt{n}\cdot m)$ time by applying the algorithm of Micali & Vazirani described in [221]. If $|M| < \frac{n}{2}$, then the perfect matching does not exist and the constraint is not satisfiable. Otherwise, we need to detect all edges that can never belong to a perfect matching. Clearly, the constraint is hyper-arc consistent iff every edge in the corresponding graph G belongs to some perfect matching (see Proposition 1 in [257]). Hence, we only need to check whether an edge that does not belong to M is a part of an even alternating cycle. This can be easily done by means of our routine. For every pair of values i and j, such that $\{v_i, v_j\}$ is forbidden we remove value i from the domain of the variable x_j and remove value j from D_{x_i}.

Note also that Régin in his algorithm explicitly used the property of the Gallai-Edmonds Decomposition although he did not point it out (cf. [257, Proposition 3]). His algorithm classifies only one vertex to an extreme set and all incident edges not traversed by the blossom shrinking algorithm are to be removed. However, by means of our technique it is possible to detect forbidden edges with respect to the entire extreme set. This follows from the fact that our algorithm does not iterate over all vertices in G but only iterates over (maximal) extreme sets.

The following example and explanation is taken from the paper by Régin [257]. The complexity of his (first) hyper-arc consistency algorithm is $\mathcal{O}(n \cdot m)$ because there are n calls to the blossom shrinking algorithm. The (second) filtering algorithm that does not achieve hyper-arc consistency has complexity $\mathcal{O}(m)$. However, in the first step starting the search from the vertex j, it will remove only the edge $\{f, h\}$ as it has been not traversed. Observe that by means of our routine, in the first step more edges are removed, as in both algorithms due to Régin. Our routine needs only two calls of the blossom shrinking algorithm, in order to ensure hyper-arc consistency.

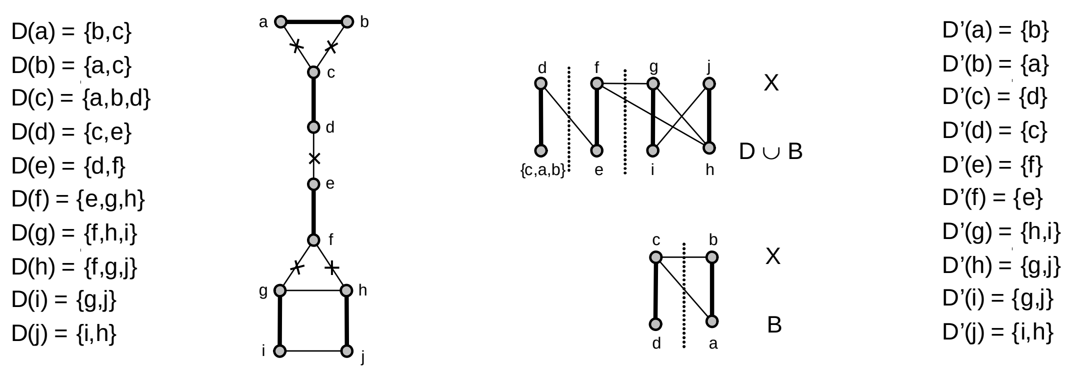


Figure 5.9: Pruning of the SYMMETRIC_ALLDIFFERENT constraint

Let us demonstrate with this example how our algorithm works. Consider the gluing bipartite multigraph arising from the value graph associated with the global constraint depicted on the left side in the Figure 5.9 above. We remove vertex j from the graph (as in the example given by Régin), make one iteration of the blossom shrinking algorithm starting from exposed vertex h and compute the Gallai-Edmonds Decomposition. Hence, here: $A = \{d, f, g\}$, $B = \{e, i, h\}$, $C = \emptyset$ and $D = \{c, a, b\}$. Thus, the extreme set $X = \{d, f, g, j\}$. Then, we perform an alternating depth-first search starting from the vertices of $B \cup base(D_i)$ and remove from the graph the detected forbidden edges, i.e. $\{d, e\}$ and $\{f, h\}$ (as not belonging to any alternating cycle or path), and $\{f, g\}$ (as belonging to $G[X]$). The mandatory (matched) edges are $\{c, d\}$ and $\{e, f\}$. We next mark all vertices of X and B as scanned and examine the pieces at extreme set X. Since only the subgraph with vertices a, b, c and d contains not yet traversed edges we choose an unvisited vertex b and repeat the whole routine. In the second step of the algorithm the extreme set $X = \{c, b\}$ is found and two forbidden edges are detected: $\{c, b\}$ (as spanned by X), and $\{a, c\}$ (as belonging to neither alternating path nor cycle). Since all vertices have been marked as scanned the algorithm terminates and our constraint is hyper-arc consistent.

SYMMETRIC_ALLDIFFERENT_EXCEPT_0 In this example we discuss a constraint SYMMETRIC_ALLDIFFERENT_EXCEPT_0. The arguments of this constraint are n assignment variables (similar to the classical SYMMETRIC_ALLDIFFERENT constraint) and the constraint states that all variables, except those which are assigned to value 0, must be grouped by pairs. As far as we know this constraint has not been treated before and no propagation algorithm has been proposed. The constraint is defined as follows

$$\text{SYMMETRIC\_ALLDIFFERENT\_EXCEPT\_0}(x_1, \ldots, x_n) = \{(d_1, \ldots, d_n) \in D_{x_1} \times \cdots \times D_{x_n} \mid \underset{\substack{i,j \\ i \neq j}}{\forall}\ d_i = 0 \vee d_j = 0 \vee (d_i = j \Leftrightarrow d_j = i)\}.$$

The constraint has been recently introduced within the Global Constraint Catalog [27]. In contrast to the previous constraint this constraint can be defined on the odd number of variables. It can have an application in a number of real sport scheduling problems. Value 0 can be used, for instance, to model the fact that a team does not play. We now show how one can filter this constraint in two distinct, but equivalent, ways.

The constraint can be expressed by a graph with degree conditions g and f. In the same way as for the common SYMMETRIC_ALLDIFFERENT constraint, we construct the graph G with the vertex set $V = \{v_1, \ldots, v_n\}$ and the edge set E, such that $\{v_i, v_j\} \in E$ iff $i \in D_{x_j} \wedge j \in D_{x_i}$. Further, for every vertex v_i we set $g(v_i) = f(v_i) = 1$ if $0 \notin D_{x_i}$ and $g(v_i) = 0$, $f(v_i) = 1$ if $0 \in D_{x_i}$.

Then it is easy to see that the constraint is satisfied iff G has a perfect (g, f)-matching and it is hyper-arc consistent iff every edge in the corresponding graph G belongs to some perfect (g, f)-matching.

A perfect $(0, 1)$-matching is a special case of a perfect degree-matching for which holds the condition $0 = g(x) < f(x) = 1 \leq d(x)$ for some vertices $x \in V(G)$, called *semi-saturated* vertices. There is a procedure for reducing a perfect $(0, 1)$-matching problem on a graph G to a perfect matching problem on a larger graph. The procedure looks as follows:

- Join by an edge every pair of non-adjacent semi-saturated vertices.
- The subgraph (induced by semi-saturated vertices) results in a complete graph.
- If G has an odd number of vertices then add one (dummy) vertex and connect it with all semi-saturated vertices.

These operations are well-defined. Let us denote the resulting graph by G^*. The spanning subgraph induced by semi-saturated vertices and dummy vertex, if added, will be called the *gadget*. We have the following result

Theorem 5.4.1 *Graph G has a perfect $(0, 1)$-matching iff G^* has a perfect matching.*

Proof If G^* has a perfect matching M^* then removing from G^* all the inserted edges belonging to the gadget results in G with a perfect $(0,1)$-matching. Conversely, if G has a perfect $(0,1)$-matching M then it is always possible to extend M to a perfect matching of G^* by saturating the exposed vertices of G^*. □

We have given a complete description of the structure of elementary components in a graph G having a perfect $(0,1)$-matching. We proved that the augmentation of G by gadgets does not change the canonical decomposition of the graph.

We use this theorem to make the SYMMETRIC_ALLDIFFERENT_EXCEPT_0 constraint hyper-arc consistent.

Theorem 5.4.2 *The global constraint* SYMMETRIC_ALLDIFFERENT_EXCEPT_0 *is hyper-arc consistent iff every edge in the corresponding graph G^* belongs to a perfect matching.*

Proof By the definition of hyper-arc consistency and application of Theorem 5.4.1. □

If every edge belonging to the gadget and incident with the vertex v_i is forbidden then value 0 is infeasible and can be deleted from the domain of the variable x_i (see Figure 5.10).

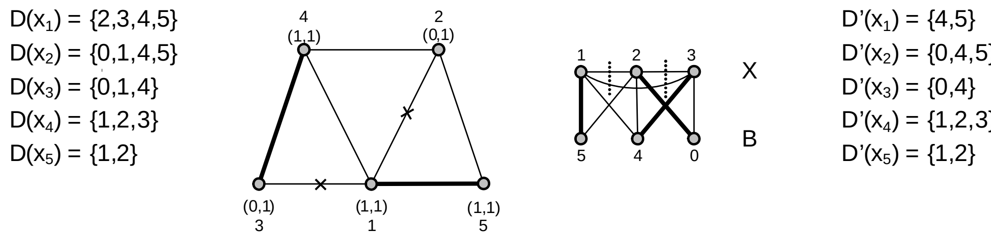


Figure 5.10: Pruning of the SYMMETRIC_ALLDIFFERENT_EXCEPT_0 constraint

The second way can be schematized as follows. We add to G a new dummy vertex labeled v_0 that is joined by an edge to each vertex of which a corresponding variable has a value 0 in its domain. Finally, we define the degree conditions $g(v_0) = 0$ and $f(v_0) = d(v_0)$ for the dummy vertex v_0, and $g(v_i) = f(v_i) = 1$ for all the remaining vertices. Then, in a similar way as before, there is a strong relationship between a perfect (g, f)-matching and a solution of the constraint. Of course, there is the same propagation algorithm which achieves hyper-arc consistency.

SYMMETRIC_ALLDIFFERENT_LOOP The SYMMETRIC_ALLDIFFERENT_LOOP constraint extends the common SYMMETRIC_ALLDIFFERENT constraint by allowing some kind of polygamy between the pairing of objects. This corresponds to occurring loops in a value graph associated with the global constraint.

This variant of the SYMMETRIC_ALLDIFFERENT constraint has been recently introduced within the Global Constraint Catalog [27]. Using this constraint the Dürer's Magic Square and Survo Puzzle problems can be easily modeled [36]. The constraint can also be expressed as an ALLDIFFERENT constraint together with constraints that maintain the symmetry. Another representation can be made that uses the so-called CYCLE constraint, where each cycle consists of at most two vertices. Note that loops are considered to be cycles of length 1 and a task would be to find in a digraph associated with the constraint a partition with cycles of length at most 2. The constraint is defined as follows

$$\begin{aligned}&\textsc{symmetric\_alldifferent\_loop}(x_1,\dots,x_n) = \\ &\qquad\{(d_1,\dots,d_n)\in D_{x_1}\times\cdots\times D_{x_n} \mid \underset{\substack{i,j\\ i\neq j}}{\forall}\, d_i = i \vee d_j = j \vee (d_i = j \Leftrightarrow d_j = i)\}.\end{aligned}$$

It is straightforward to check that our constraint is strongly related with the global constraint SYMMETRIC_ALLDIFFERENT_EXCEPT_0, which we discussed in the previous example. Of course, if we replace the domain of every variable x_i with value i in its domain by $D_{x_i} \setminus \{i\} \cup \{0\}$ then the SYMMETRIC_ALLDIFFERENT_EXCEPT_0 constraint becomes equivalent to the SYMMETRIC_ALLDIFFERENT_LOOP constraint and there is the same propagation algorithm which achieves hyper-arc consistency.

The following result is an immediate consequence of our considerations.

Theorem 5.4.3 *The constraint* SYMMETRIC_ALLDIFFERENT_LOOP *is hyper-arc consistent iff the corresponding* SYMMETRIC_ALLDIFFERENT_EXCEPT_0 *constraint is hyper-arc consistent.*

SOFT_SYMMETRIC_ALLDIFFERENT_VAR When a global constraint has no solution it is also said to be over-constrained. It is then natural to identify soft constraints that are allowed to be violated, and minimize the total violation according to some criteria.

For a set C of constraints, let $C_h \subseteq C$ be the set of *hard constraints*, that is, the constraints that must necessarily be satisfied. Then $C_s = C\setminus C_h$ is the set of *soft constraints*. For the set C of constraints we introduce a function that measures the violation and has the following form

$$\mu : D_{x_1}\times\cdots\times D_{x_n} \mapsto \mathbb{Z}^+.$$

This approach has been introduced in [245] and was developed further in [35] (see also [261]). Hence, our constraint can be defined as follows

$$\begin{aligned}&\textsc{soft\_symmetric\_alldifferent\_var}(x_1,\dots,x_n,z,\mu) = \\ &\qquad\{(d_1,\dots,d_n)\in D_{x_1}\times\cdots\times D_{x_n} \mid \mu(d_1,\dots,d_n)\in D_z\}.\end{aligned}$$

In this example it will be shown how the classical Gallai-Edmonds Decomposition can be useful to solve the soft version of the SYMMETRIC_ALLDIFFERENT constraint. As far as we know such a constraint has not been treated before and no propagation algorithm has been proposed.

A structural decomposition of a general graph associated with a global constraint canonically decomposes it into two parts: over-constrained and well-constrained. We will use the notation $G_O = G[A\cup B\cup D]$ to represent an over-constrained part and $G_W = G[C]$ denotes the remaining well-constrained part. Intuitively, if the hard global constraint can be modeled as a matching problem in a graph then this constraint is satisfiable iff $G_O = \emptyset$.

Loosely speaking a well-constrained problem has at least one solution (or exactly one solution if the perfect matching is unique), whereas an over-constrained problem has no solution. In a well-constrained part the number of variables equals the number of satisfied constraints. This part can be further decomposed into smaller canonical parts (matching covered subgraphs).

In an over-constrained part the number of variables is greater than the number of satisfied constraints. The additional variables are redundant and thus the graph has no perfect matching and the constraint is inconsistent. A possible way to make the constraint satisfiable would be a transformation of the over-constrained part into a well-constrained one. This can be realized by omitting the conflicting variables from the global constraint or assigning other values to them. The latter operation corresponds to the so-called variable-based violation measure. The number of contradictory variables equals the deficiency of G_O.

The variable-based violation measure counts how many variables need to change their values in order for the constraint to be satisfied. In this model the cost of an assignment is defined to be the minimal number of vertices not covered by any maximum matching.

We can make the SOFT_SYMMETRIC_ALLDIFFERENT_VAR constraint hyper-arc consistent in the following way. Let $D = (N, A)$ be a digraph such that $N = V(G)$ and $A = \{(v_i, v_j) \mid i \in D_{x_j}\}$. We construct a complete graph K_n on the same set of vertices as the digraph D. We next extend the graph K_n by applying a 'weight' function w to its edges. For every edge in K_n with no, one or two corresponding arcs in D we give it a weight of 2, 1 and 0, respectively.

Let K_n be the weighted value graph associated with the constraint. The global constraint SOFT_SYMMETRIC_ALLDIFFERENT_VAR is satisfied iff there exists a weighted perfect matching in K_n with cost w such that $\min(D_z) \leq w \leq \max(D_z)$ (we assume that the domain of the cost variable z is an interval).

We can now define a simple, brute-force filtering algorithm for computing the partition of edges. Let M_{min} be a minimum-weight perfect matching in K_n with cost w_{min} (clearly, every complete graph of even order contains a perfect matching). It is known that if arbitrary changes are made to the edges incident to one vertex, a new minimum-weight perfect matching can be constructed by finding one weighted augmenting path. This can be done in time $\mathcal{O}(m + n \cdot \log n)$ [118].

In order to find all forbidden edges we choose an arbitrary free edge $e = \{v_i, v_j\}$ such that $w(e) \neq 2$ and recompute a minimum-weight matching M_e in the graph $K_n - v_i - v_j$. Then the cost of the minimum-weight perfect matching involving edge e is $w_e = w(M_e) + w(e)$. We mark edge e as allowed if $\min(D_z) \leq w_e \leq \max(D_z)$; otherwise, edge e is forbidden.

In order to find all mandatory edges we choose an arbitrary matched edge e such that $w(e) \neq 2$ and recompute a minimum-weight perfect matching $M_{\bar{e}}$ in the graph $K_n - e$. Then the cost of the minimum-weight perfect matching avoiding edge e equals $w_{\bar{e}} = w(M_{\bar{e}})$. We mark edge e as mandatory if $\max(D_z) < w_{\bar{e}}$.

Note that the procedure, which allows us to transform the solution of the constraint into the weighted matching problem, is very inefficient, because it greatly increases the number of edges in the auxiliary graph. Since there are $\frac{1}{2}n(n-1)$ edges in a complete graph, using the weighted matching algorithm improved by Gabow [118] yields an $\mathcal{O}(n \cdot (n^2 + n \cdot \log n))$ running time (for checking the feasibility) plus an $\mathcal{O}(m \cdot (n^2 + n \cdot \log n))$ running time (for achieving hyper-arc consistency). For certain problems, this complexity may be prohibitively expensive and can hinder this method from being systematically used during the search for a solution and pruning of the SOFT_SYMMETRIC_ALLDIFFERENT_VAR constraint. Therefore, we now present the efficient (but more involved) method based on the Gallai-Edmonds decomposition. From an interpretation point of view this model can be expressed with the SOFT_2-CYCLE_VAR constraint.

We now sketch a method to make our over-constrained constraint satisfied. Let G be any graph with a maximum matching M, and let $\langle A, B, C, D \rangle$ denote the canonical decomposition of G. Further, let $C_1, C_2, \ldots, C_r$ and $D_1, D_2, \ldots, D_t$, where $t > |A|$, be connected components of C and $B \cup D$, respectively.

In order to make the constraint satisfied we must first add edges to the associated graph G in such a way, that the resulting graph will have a perfect matching. First, observe that forcing a maximum matching to use forbidden (or not existing) edges in G can increase (or decrease) the deficiency by 2. We prove this in a more formal way:

Lemma 5.4.4 *Forcing a maximum matching to use forbidden (or not existing) edges in G can increase (or decrease) the deficiency by* 2.

Proof Obviously, forcing the use of any forbidden edge decreases the deficiency by 2. There are three possibilities here. We can use an edge in a connected component of C, an edge between C and A, or an edge spanned by A.

The case of not existing edges is more complicated. There are several possibilities here. The proof follows from a case-by-case analysis. All the possibilities are shown in the below table together with the resulting difference of the deficiency and the cost function of the corresponding maximum matching. The straightforward but tedious verification is omitted.

edge	deficiency δ	violation μ	remark
$e \in \nabla(A, A)$	+2	+1, +2	forbidden edge
$e \in \nabla(A, C)$	+2	-	forbidden edge
$e \in \nabla(A, D)$	0	+2	allowed edge
$e \in \nabla(C_i, C_i)$	0, +2	+2	
$e \in \nabla(C_i, C_j)$	+2	-	forbidden edge
$e \in \nabla(C_i, D_j)$	0	-	
$e \in \nabla(D_i, D_i)$	0	-	allowed edge
$e \in \nabla(D_i, D_j)$	-2, 0	-2, -1	decreased deficiency

Table 5.2: Scenarios for the SOFT_SYMMETRIC_ALLDIFFERENT_VAR constraint

Note that the edges of the maximum matching M constructed in the proof of the above theorem have costs either 1 or 2. □

For example, we can decrease the deficiency of G with a maximum matching M by

- connecting two distinct components D_i and D_j of $B \cup D$ by an edge, or
- joining C_k to D_i and D_j by two edges, in such a way that there exists an odd alternating path connecting adjacent vertices in the matching covered subgraph of $G[C_k]$, or
- inserting three edges among $C_k{-}C_l$, $C_k{-}D_i$, and $C_l{-}D_j$, in such a way that there is an odd alternating path joining adjacent vertices in the matching covered subgraphs of $G[C_k]$ and $G[C_l]$.

We want to point out that it is possible to make a forbidden edge admissible in the following situations:

- taking a forbidden edge e spanned by C and joining by an edge two vertices connecting the endpoints of e by odd alternating paths, or
- taking a forbidden edge e between C and A and joining by an edge vertex of C with vertex of $B \cup D$, in such a way that these vertices are reachable by an odd alternating path, or
- taking a forbidden edge e spanned by A and connecting by two edges four distinct components of D.

We have already reached the perfect matching and our next step will be to show how to build a next perfect matching involving additional edges. Deciding if an edge e inserted to $G[C]$ is allowed or not seems to be a difficult task. Let C_i be a connected component of $G[C] + e$ with a perfect matching M_i. Consider an alternating depth-first forest after the algorithm (alternating depth-first search with respect to M_i) terminates. By Corollary 2.4.3 an edge is allowed if it is contained in an even alternating cycle. Thus, every back (or forward) edge connecting two vertices on the levels with different parities is allowed. This can be checked in $\mathcal{O}(1)$ time. If an edge joins two vertices connected by an odd alternating path, then it creates an alternating cycle involving this edge. This costs $\mathcal{O}(m_i)$ time, where m_i is the number of edges in C_i. On the other hand, the edge is forbidden if it connects two vertices belonging to the same extreme set. This takes $\mathcal{O}(1)$ time, assuming that the canonical partition is known.

We will now consider the problem to determine whether two arbitrary edges belong to any perfect matching. In order to describe our idea in modern terminology, we need a few definitions. If G is a graph and H is a subgraph of G then H is said to be *nice* if $G - V(H)$ has a perfect matching. It is known that any two edges of an elementary bipartite graph lie on a nice cycle (see Corollary 4.2.10 in [212]). Further, if G (not necessarily bipartite) is matching covered, then any two edges of G are contained in a nice (alternating) cycle (see Theorem 5.4.4 in [212]). In particular, for any edge e there exists a perfect matching containing e and another perfect matching avoiding e. These results give us immediately an algorithm for finding all admissible edges in $G[C]$.

In order to check the satisfiability and to achieve hyper-arc consistency we first construct two auxiliary graphs associated with the SOFT_SYMMETRIC_ALLDIFFERENT_VAR constraint. The first graph is the *hard value graph* G_h: there is an edge between v_i and v_j if $i \in D_{x_j} \wedge j \in D_{x_i}$. The second graph, the *soft value graph* G_s, is constructed in such a way that there is an edge between v_i and v_j if $i \in D_{x_j} \vee j \in D_{x_i}$. Clearly, $|V(G_h)| = |V(G_s)|$, $|E(G_h)| \leq |E(G_s)|$, and $\delta(G_h) \geq \delta(G_s)$. Next, we compute a maximum matching M_h in the graph G_h. This can be carried out in polynomial time by using the algorithm of Micali & Vazirani [221]. If G_h has no perfect matching then the constraint is over-constrained. Let $\delta_h = \delta(G_h)$. We then distinguish two situations:

1. $|E(G_h)| = |E(G_s)|$. We consider the following cases:

   (a) If $\max(D_z) < \delta_h$ then the constraint is inconsistent.

   (b) If $\min(D_z) \leq \delta_h < \max(D_z) - 1$ then the constraint is consistent, and all domain values are allowed. Namely, if we change any variable x_j to the value i, then we have to assign the value j to the variable x_i. From an interpretation point of view this corresponds to adding an edge between two vertices.

   (c) If $\delta_h = \max(D_z)$ or $\delta_h = \max(D_z) - 1$ then the constraint is consistent, and only those domain values whose corresponding edge belongs to a maximum matching are feasible. According to the Gallai-Edmonds Structure Theorem every edge in $G_O = G_h[A \cup B \cup D]$ (except that spanned by A) is allowed (see Lemma 2.6.1). Thus, we have only to determine admissible and forbidden edges in $G_W = G_h[C]$. We can make it in the same way as for the standard SYMMETRIC_ALLDIFFERENT constraint. All these edges can be determined, and the corresponding domain values can be removed, in $\mathcal{O}(p' \cdot m')$ time, where p' is the number of maximal extreme sets in G_W and m' is the number of edges in G_W (see Theorem 5.3.5).

   Finally, we can update $\min(D_z)$ to be the maximum of its current value and δ_h. Additionally, we must remove from D_z all values being of different parity than δ_h (all odd values). This follows from the fact that adding an edge to the graph (which corresponds to assigning other values to variables) can only change its deficiency by 2. Then our constraint is hyper-arc consistent.

2. $|E(G_s)| > |E(G_h)|$. We proceed in the following manner.

   Let A_h, B_h, C_h and D_h be the sets of the Gallai-Edmonds decomposition of G_h. From the subgraph G_O we create the graph $G_s[\nabla(A_h \cup D_h, B_h \cup D_h)]$, derived from the subgraph $G_h[A_h \cup B_h \cup D_h]$ by removing edges spanned by A_h and inserting soft edges (i.e. $i \in D_{x_j} \wedge j \notin D_{x_i} \vee j \in D_{x_i} \wedge i \notin D_{x_j}$). This allows us to compute a maximum matching $M_s[A_h \cup B_h \cup D_h]$ and the deficiency of $G_s[A_h \cup B_h \cup D_h]$. Clearly, $\delta(G_s) \leq \delta(G_s[A_h \cup B_h \cup D_h]) \leq \delta(G_h)$. Let $\delta_s = \frac{1}{2}(\delta(G_h) + \delta(G_s[A_h \cup B_h \cup D_h]))$. This is the minimal number of variables that need to be changed in order to satisfy the constraint. This follows from the fact that from any graph G with a maximum matching we can obtain a perfect matching by connecting the pairs of vertices from two distinct components of $B_h \cup D_h$. Clearly, $\delta_h \geq \delta_s = w_{min}$, where w_{min} is the cost of the minimum-weight perfect matching in the weighted value graph associated with the global constraint.

We can now prune our constraint in the following way (in general, we will not achieve hyper-arc consistency). We have two cases to consider.

(a) $\delta_s = \delta_h$.

From the graph G_h we create an auxiliary graph G'_s by inserting all soft edges between C_h and $B_h \cup D_h$. Next, we join by an edge all pairs of subsequent vertices forming a complete graph $K_{r'}$, where r' is the number of vertices in $B_h \cup D_h$ adjacent to $\nabla(C_h, B_h \cup D_h)$. Let us denote the number of edges in G'_s by m'_s. Observe that all hard edges in $\nabla(D_h, D_h)$ are allowed.

i. If $\max(D_z) < \delta_s$ then the constraint is inconsistent.

ii. If $\delta_s = \max(D_z)$ then an edge e in G'_s is admissible if it lies in some maximum matching of G'_s. With the help of our technique we can find these edges in $\mathcal{O}(p'_s \cdot m'_s)$ time.

iii. If $\delta_s = \max(D_z) - 1$ then an edge is allowed if it is allowed in G'_s, or $G'_s + e_s$. In particular, all soft edges in $\nabla(C_h, B_h \cup D_h)$ are allowed.

iv. If $\delta_s = \max(D_z) - 2$ then all hard edges are admissible; a soft edge is allowed if it is allowed in G'_s, $G'_s + e_s$, or $G_s[C_h]$.

v. If $\min(D_z) \leq \delta_s < \max(D_z) - 2$ then all edges in G_s are allowed.

(b) $\delta_s < \delta_h$.

Let A_s, B_s, C_s and D_s be the sets of the Gallai-Edmonds decomposition of $G_s[A_h \cup B_h \cup D_h]$. Clearly, $A_s \subseteq A_h$, $B_s \subseteq B_h$, $C_s \subseteq A_h \cup B_h \cup D_h$, $D_s \subseteq D_h$ and $A_h \cup B_h \cup D_h = A_s \cup B_s \cup C_s \cup D_s$. In the same way as for the previous case, we create an auxiliary graph G''_s from the graph G_h by inserting all soft edges between C_h and $B_s \cup D_s$, and additionally, by inserting all soft edges between two distinct connected components of D_h. Next, we join by an edge all pairs of adjacent vertices forming a complete graph $K_{r''}$, where r'' is the number of vertices in $B_s \cup D_s$ incident with $\nabla(C_h, B_s \cup D_s)$. Let us denote the number of edges in G''_s by m''_s. Note that all hard edges in $\nabla(D_s, D_s)$ are allowed.

i. If $\max(D_z) < \delta_s$ then the constraint is inconsistent.

ii. If $\delta_s = \max(D_z)$ then an edge e in G''_s is admissible if it lies in some maximum matching of G''_s (see Figure 5.11). By means of our technique we can find these edges in $\mathcal{O}(p''_s \cdot m''_s)$ time.

iii. If $\delta_s = \max(D_z) - 1$ then an edge is allowed if it is allowed in G''_s, G'_s, or $G''_s + e_s$. In particular, all hard edges in $\nabla(A_h, B_h \cup D_h)$ are allowed.

iv. If $\delta_s = \max(D_z) - 2$ then all hard edges are admissible; a soft edge is allowed if it is allowed in G''_s, G'_s, $G''_s + e_s$, $G'_s + e_s$, or $G_s[C_h]$.

v. If $\min(D_z) \leq \delta_s < \max(D_z) - 2$ then all edges in G_s are allowed.

Finally, we update $\min(D_z)$ to δ_s if $\min(D_z) < \delta_s$ and remove from D_z all odd values if there exists no maximum matching in G'_s (case (a)) or G''_s (case (b)) with the odd number of soft edges; otherwise all (odd and even) values are feasible.

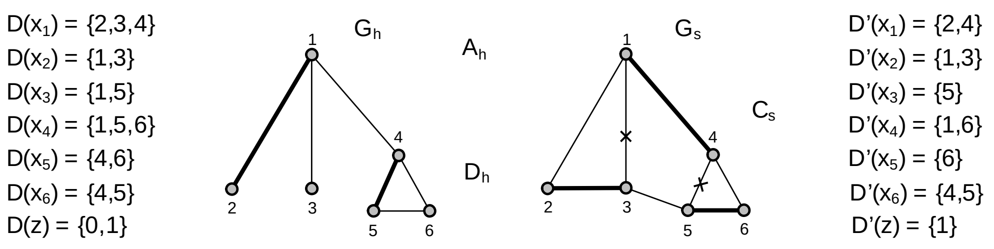


Figure 5.11: Pruning of the SOFT_SYMMETRIC_ALLDIFFERENT_VAR constraint

2-CYCLE The 2-CYCLE constraint is a particular case of the CYCLE constraint which was introduced in CHIP [29]. When we impose an additional condition that the number of cycles is exactly half of the number of vertices and each cycle has only two vertices then the CYCLE constraint can be expressed by means of the SYMMETRIC_ALLDIFFERENT constraint. This constraint can then be interpreted as covering the associated digraph with disjoint cycles of length two. Clearly, there exists in our case no solution of the CYCLE constraint when the number of variables is odd or NCYCLE $\neq \frac{n}{2}$.

PROPER_FOREST This constraint was proposed by Nicolas Beldiceanu, Irit Katriel, and Xavier Lorca in [32]. The constraint has the form PROPER_FOREST(NTREE, NODES) and requires that a graph G associated with the constraint can be covered by a set of NTREE proper trees, i.e. trees that contain at least two vertices, in such a way that each vertex of G belongs to one distinct tree. A filtering algorithm for this constraint is described in [32]. It achieves hybrid consistency and its running time is dominated by the complexity of finding all edges that do not belong to any maximum matching in G.

In appendix B an algorithm for detecting always saturated vertices is given. However, we know that the vertices of the graph which are saturated by every maximum matching are vertices in $C \cup A$. These vertices can be detected in linear time.

In appendix C the authors outline an $\mathcal{O}(n \cdot m)$ algorithm that determines which edges of the graph belong to at least one maximum matching. We know that this can be performed in $\mathcal{O}(p \cdot m)$ time.

CLIQUE In this example we consider the CLIQUE constraint, which has been introduced by Torsten Fahle [101]. The constraint models the maximum clique problem. It asks for a largest clique Q in a graph G associated with the constraint. It is known that this problem is NP-hard [182],[131, Problem GT19]. Although CLIQUE is not directly connected with the Gallai-Edmonds Decomposition, our technique can be useful to compute an upper bound for the size of the maximum clique. The filtering rule is based on the relation between the maximum clique and the maximum matching: $|Q| \leq |G| - \nu(\overline{G})$ (cf. [259, Property 4]). This upper bound can be easily checked by means of the maximum matching algorithm.

TOUR The global constraint TOUR is an undirected version of the CIRCUIT constraint. It describes a Hamiltonian cycle on an undirected graph associated with the constraint. The constraint is defined as TOUR(NODES) where NODES is a collection of n variables whose domains are subsets of $\{1, \ldots, n\}$. The constraint requires that $y_1, \ldots, y_n$ is a permutation of $1, \ldots, n$, where $y_1 = x_{y_n}$ and $y_{i+1} = x_{y_i}$ for $i = 1, \ldots, n-1$. The constraint can be viewed as describing a Hamiltonian cycle in an associated undirected graph G that contains an edge $\{v_i, v_j\}$ iff j belongs to the domain of x_i and i belongs to the domain of x_j. An edge $\{v_i, v_j\}$ of G is selected when $x_i = j \wedge i = x_j$ and TOUR requires that the selected edges form a Hamiltonian cycle.

Obviously, checking a TOUR constraint for satisfiability is equivalent to checking whether an associated graph has a Hamiltonian cycle, which is an NP-complete problem [182],[131, Problem GT37]. Achieving hyper-arc consistency for TOUR is thus NP-hard. There are, however, useful incomplete filtering methods that run in polynomial time.

A Hamiltonian cycle is a special (i.e. connected) perfect 2-matching. Hence, a necessary condition to satisfiability of this constraint is to have a perfect 2-matching. The second necessary condition is that the graph must be 2-connected. So, if G has a connected perfect 2-matching, then $C(G - v) = \emptyset$ for all $v \in V(G)$, where $C(G - v)$ is the set C of the Gallai-Edmonds decomposition for $G - v$. Also, if the associated graph is bipartite, it must have the same number of vertices in each color class.

To our knowledge, there are no published filtering algorithms for pruning of the TOUR constraint. We present two different methods. The first method consists of computing a perfect matching on the graph associated with the global constraint and removing all forbidden edges. This follows from the fact that a forbidden edge does not belong to any alternating cycle and thus cannot be a part of a Hamiltonian cycle. There are no mandatory edges in a Hamiltonian graph too, because it must be 2-connected.

However, this method will not work on graphs with an odd order. Observe that every Hamiltonian graph with an odd number of vertices must be necessarily factor-critical. This follows from the fact that if G contains a non-empty set X of vertices whose deletion breaks the rest of the graph into more than $|X|$ connected components then G has no Hamiltonian cycle. According to the Gallai-Edmonds Structure Theorem such a set is the extreme set A, and $A = \emptyset$ if the graph is factor-critical.

In this example we now demonstrate a new partial filtering method not presented yet. This is one of the examples which explicitly uses the properties of the Lovász-Plummer Canonical Decomposition and the results of Theorem 5.2.8. Our idea is based on identifying perfect matchings in the incremental graph corresponding to the graph associated with the constraint. This follows from the fact that every Hamiltonian cycle is a connected perfect 2-matching. Our routine looks for a perfect matching in the incremental graph and any edge not belonging to it will be removed from the associated graph. This will result in deleting some edges from the associated graph, which cannot be a part of any Hamiltonian cycle (see Figure 5.12). By using the reduction technique due to Gabow [115] it is possible to compute a perfect 2-matching in $\mathcal{O}(\sqrt{n} \cdot m)$ time.

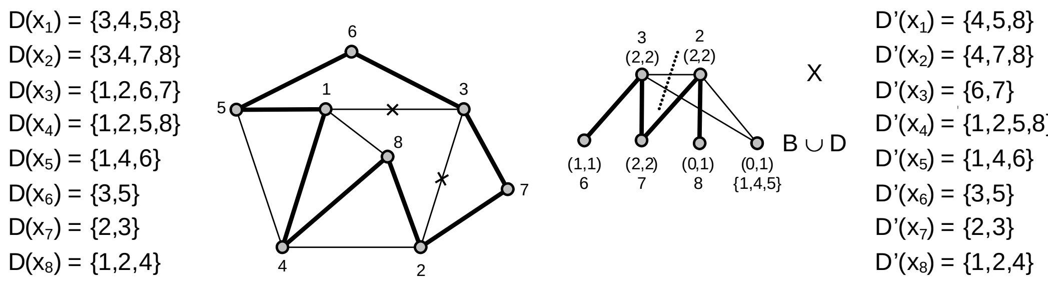


Figure 5.12: Pruning of the TOUR constraint

To summarize, three of the easily verifiable ways to justify the claim that TOUR constraint has no solution, are as follows (G denotes the graph associated with the constraint):

1. if G is not 2-connected,
2. (a) if G with an even number of vertices has no perfect matching,
   (b) if G with an odd number of vertices is not factor-critical,
3. if G has no perfect 2-matching.

UNDIRECTED_PATH This is an undirected version of the constraint PATH_FROM_TO that was proposed by Althaus et al. in [11]. The constraint holds if some edges of a graph G associated with the constraint form a path between two given vertices of G.

This is the second example that demonstrates the application of Theorem 5.2.8. Recall that a path P is Hamiltonian if it passes through every vertex exactly once. Obviously, a Hamiltonian path is a connected perfect $(1, 2)$-matching.

We propose the following incomplete filtering algorithm. Without loss of generality, we can assume that we wish to find a path between vertices v_1 and v_n. First, we construct an auxiliary graph G. We add a loop to G for each vertex v_i such that $0 \in D_{x_i}$. Next, we define the degree condition f as follows. We set $f(v_1) = f(v_n) = 1$, and $f(v) = 2$ for all the remaining vertices. Then it is easy to see that any path between v_1 and v_n represents a perfect f-matching in G (see Figure 5.13). If we assume that the loops are excluded from the matching then the generated path between two distinguished vertices forms a connected $(1, 2)$-factor.

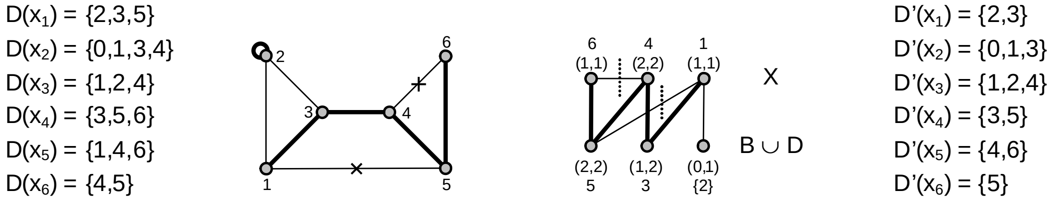


Figure 5.13: Pruning of the UNDIRECTED_PATH constraint

5.5 Kano Canonical Decomposition

This section is mostly based on the book "Factors and Factorizations of Graphs" [8].

In this section we consider the so-called parity factors. Let G be any graph, and let g and f be integer-valued functions such that $0 \leq g(x) \leq f(x) \leq d_G(x)$ and $g(x) \equiv f(x) \pmod 2$ for each $x \in V(G)$. Then a spanning subgraph F of G is called a *parity* (g, f)*-factor* if $g(x) \leq d_F(x) \leq f(x)$ and $d_F(x) \equiv f(x) \pmod 2$ for all $x \in V(G)$.

A spanning subgraph F of a graph G is called an *even factor* of G if every vertex has an even degree in F. Analogously, a spanning subgraph F of a graph G is called an *odd factor* of G if every vertex has an odd degree in F. In particular, for an odd integer-valued function f, a spanning subgraph F of G is called a $(1, f)$*-odd factor* if $d_F(x) \in \{1, 3, \ldots, f(x)\}$ for every $x \in V(G)$.

Jin Akiyama & Mikio Kano in [8] proposed the following construction for a parity (g, f)-factor. We construct an auxiliary graph G^* from G by adding $(f(x) - g(x))/2$ loops to every vertex $x \in V(G)$. We can easily see that G has a parity (g, f)-factor iff G^* has an f-factor.

For a $(1, f)$-odd parity factor they give the following construction: replace every vertex $x \in V(G)$ by the complete graph $K_{f(x)}$ on $f(x)$ vertices and for every edge $\{x, y\}$ of G, join every vertex of $K_{f(x)}$ to every vertex of $K_{f(y)}$. Then there is a strong relationship between perfect matching in G^* and $(1, f)$-odd factor of G.

Another transformation to an ordinary matching has been proposed by Gérard Cornuéjols in [68]. In his construction of the incremental graph G^* each gadget is a complete bipartite graph $K_{d(x),d(x)-g(x)}$ with additional edges connecting $f(x) - g(x)$ distinct core vertices. It follows from this definition that the gadget is no more bipartite and has $2 \cdot d(x) - g(x)$ vertices and $d(x) \cdot (d(x) - g(x)) + \frac{1}{2} \cdot (g(x) - f(x))$ edges.

An *augmenting walk* with respect to parity (g, f)-matching M is an alternating walk W such that $\delta(G[M \oplus W]) < \delta(G[M])$.

Theorem 5.5.1 (Kano & Katona [177]) *A parity (g, f)-matching M in a graph G is optimal iff there is no augmenting walk relative to M.*

A double alternating path is a pair of vertex-disjoint alternating paths connecting two different vertices. An even double alternating path leading to a negative/positive vertex is called a positive/negative double alternating path. An odd alternating cycle starting and terminating at v is an alternating cycle of odd length around a given vertex v. An odd alternating cycle is called increasing/decreasing if the first and the last edge is free/matched.

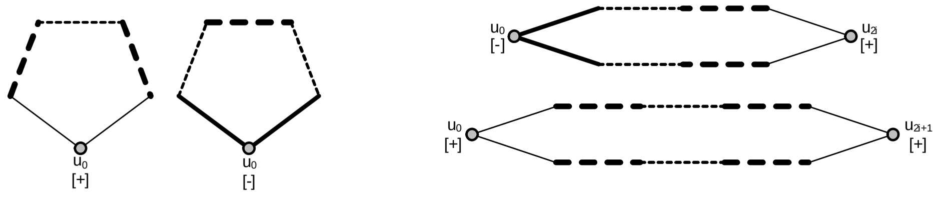


Figure 5.14: Odd alternating cycles and double alternating paths defined for parity factors

Year	Author(s)	Complexity	Strategy/Remarks
1980	Ebert [88]	$\mathcal{O}(m)$	depth-first search, maximal parity factors
1988	Cornuéjols [68]	$\mathcal{O}(n^4)$	reduction to ordinary matchings
1994	Nam [228]	$\mathcal{O}(n^3)$	gadgets, augmenting walks
2007	Kano & Katona [178]	$\mathcal{O}(n^3)$	for smallest optimal $(1, f)$-odd subgraphs

Table 5.3: History of algorithms for the parity matching problem

A graph G is said to be *elementary with respect to parity* (g, f)*-matchings* if the allowed edges induce a connected spanning subgraph of G. Analogously, a graph G is said to be *critical with respect to parity* (g, f)*-matchings* if $G + vw$ with $g(w) = f(w) = 1$ has a perfect parity (g, f)-matching for all vertices v of G.

We are now ready to give a structure theorem on $(1, f)$-odd subgraphs. Let G be a simple graph and $f : V \mapsto \{1, 3, 5, \ldots\}$. Define

$$\begin{aligned} D &= \{v \in V \mid \nu(G + vw) = \nu(G) + 2\}, \\ A &= \Gamma(D) \setminus D, \\ C &= V \setminus (A \cup D). \end{aligned}$$

Theorem 5.5.2 (Structure Theorem on $(1, f)$-odd matchings) *Let $G = (V, E)$ be a graph, f be any odd-valued function, and $\langle A, C, D\rangle$ be the decomposition defined as above. Then the following statements hold:*

1. *Every component of $G[D]$ is critical with respect to $(1, f)$-odd matchings,*
2. *Subgraph $G[C]$ has a perfect $(1, f)$-odd matching,*
3. *Every optimal $(1, f)$-odd matching M saturates $A \cup C$, and*
4. *$d_M(u) = f(u)$ for every vertex $u \in A$.*

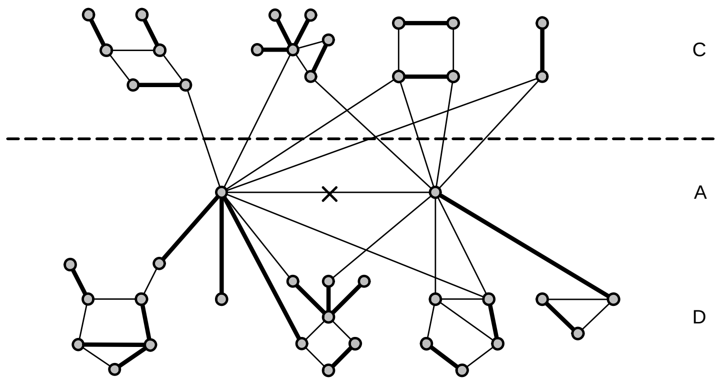


Figure 5.15: Kano canonical decomposition of a general graph with a parity $(1, f)$-matching

Theorem 5.5.3 (Ebert [88]) *Every perfect even-matching M contains at least one cycle.*

Theorem 5.5.4 (Yu & Zhang [310]) *A graph G has a unique odd factor iff G is a tree of even order.*

TOUR_EXCEPT_0 Within the TOUR constraint, we introduce the TOUR_EXCEPT_0 constraint since it can be seen as a kind of relaxation of the TOUR constraint where we allow the use of value 0 several times. The constraint enforces to cover an undirected graph G described by the NODES collection with one cycle. The required pattern is a connected perfect parity (0,2)-matching (i.e. a connected (0-2)-even factor).

FULL_BINARY_TREE A *full binary tree* is a tree in which every vertex is either a leaf or has exactly two children. If we attach to every root a pendant edge then the full binary forest forms an acyclic perfect parity (1,3)-matching (i.e. an acyclic (1,3)-odd factor).

5.6 Summary

Matching problems, in both bipartite and non-bipartite graphs, are useful models for a number of global constraints. This chapter has presented some representative applications of matching and decomposition theory in constraint programming.

This chapter illustrated the use of matching theory related to the Gallai-Edmonds canonical decomposition of a general graph as a framework to derive filtering algorithms for a range of constraints based on matching such as the well-known SYMMETRIC_ALLDIFFERENT, TOUR or the newly introduced UNDIRECTED_PATH constraint. The chapter also showed that the Gallai-Edmonds decomposition can be applied to the soft version of matching-based constraints.

Our pruning algorithm, based on the degree-matching paradigm, performs global constraint propagation, computes and generates the forbidden values for each variable of global constraint. This method is quite general and can be applied on a wide range of global constraints whose solution can be mapped to a particular matching in the general graph. We have presented applications of our procedure to some important and well-studied global constraints.

Two open problems concerning the SOFT_SYMMETRIC_ALLDIFFERENT_VAR constraint remain to be solved. Does a hyper-arc consistency filtering algorithm with the complexity of $\mathcal{O}(\sqrt{n} \cdot m)$ for the case $|E(G_s)| > |E(G_h)|$ exist? What applications could there be for this constraint?

The following table summarizes all the results for global constraints representable by a general graph that were discussed in this chapter. Here, n denotes the number of vertices, m is the number of edges, and p denotes the number of maximal extreme sets in the graph. The constraints designated with an asterisk are NP-hard.

global constraint	model	matching	checking feasibility	hyper-arc consistency	reference
SYMMETRIC_ ALLDIFFERENT	value graph	perfect	$\mathcal{O}(\sqrt{n} \cdot m)$	$\mathcal{O}(p \cdot m)$	Régin [257]
SYMMETRIC_ALLDIF-FERENT_EXCEPT_0	value graph	(0,1)-factor	$\mathcal{O}(\sqrt{n} \cdot m)$	$\mathcal{O}(p \cdot m)$	here
SYMMETRIC_ALL-DIFFERENT_LOOP	value graph	(0,1)-factor	$\mathcal{O}(\sqrt{n} \cdot m)$	$\mathcal{O}(p \cdot m)$	here
SOFT_SYMMETRIC_ ALLDIFFERENT_VAR	hard and soft value graph	maximum	$\mathcal{O}(\sqrt{n} \cdot m)$	$\mathcal{O}(p \cdot m)$	here
2-CYCLE	value graph	perfect	$\mathcal{O}(\sqrt{n} \cdot m)$	$\mathcal{O}(p \cdot m)$	here
PROPER_FOREST	solid and dotted value graph	maximum	$\mathcal{O}(\sqrt{n} \cdot m)$	$\mathcal{O}(p \cdot m)$	Beldiceanu [32]
CLIQUE(*)	complementary graph	maximum	–	–	Fahle [101]
TOUR(*)	value graph	connected 2-factor	–	–	here
UNDIRECTED_PATH(*)	value graph	connected (1,2)-factor	–	–	here

Table 5.4: Summary of results for general graphs

Chapter 6

Directed Graphs

In previous chapters we discussed matchings in bipartite and general graphs and studied their applications in constraint programming. In this chapter we introduce directed matchings. This extension also plays an important role in matching theory and constraint programming. Although there is not much coverage of this topic in the literature, the reader can refer to the works of William T. Tutte [292] and Oystein Ore [234,235,237].

In this chapter we review existing methods and present a generic propagation mechanism for graph partitioning constraints based on directed matchings. The task is also to give a set of several propagation rules according to specific partition properties. Every solution of the global constraint corresponds to a subgraph of the corresponding digraph associated with the constraint. The filtering identifies the arcs of the digraph that do not belong to a solution. We illustrate this principle on some common global constraints.

Directed graphs are often used to model different problems. Constraints that describe partitions of the nodes in a given initial digraph have been considered from an early stage of constraint programming research. Some examples include the CIRCUIT [200],[188], CYCLE [29], TREE [30],[100], PATH [34],[268] and CLIQUE [101],[259] constraints. Problems involving these constraints are intractable in general. This work goes one step further by introducing a set of specific propagation rules for these global constraints.

In Chapters 4 and 5 filtering algorithms were introduced that use the semantic of the constraint in terms of maximum matching. In this chapter, we extend this concept to global constraints representable by directed graphs. We present a paradigm based on a directed matching. We illustrate our method with a complete study of specific global constraints.

In the context of graph partitioning constraints, the contribution of this chapter is to show how to combine directed graphs with decomposition theory to get a general propagation technique for graph partitioning constraints. We want to point out that our goal is not to partition a given digraph D associated with a global constraint, but rather to find out whether it is possible to make and detect those arcs of D that do not belong to any partition corresponding to a specific pattern.

Directed graphs probably form the most interesting class of graphs and have a very rich theory, a theory which has no analog in the theory of undirected graphs. A standard reference for the theory of directed graphs, with emphasis on structures and concepts, rather than algorithms, is [151]. An excellent reference to algorithms is [21].

Throughout this chapter we use round brackets ( ) to enclose ordered pairs of vertices (arcs) and curly brackets { } to enclose unordered pairs (edges). A directed path connecting v_1 to v_n is a set of arcs $(v_1, v_2), (v_2, v_3), \ldots, (v_{n-1}, v_n)$ ordered in such a way that the tail of any arc is the head of its successor.

The work in this chapter is based on the paper [73]. But we also present some results which have not been published yet. In particular, in Section 6.3 we derive a method for the PROPER partitioning constraints, in which every pattern must consist of at least two nodes. Our technique can be then applied to solve the proper versions of graph partitioning constraints. Section 6.2 deals with an algorithmic method based on the decomposition theory of directed graphs. We give a new linear time algorithm for the detection of strongly connected components in a directed graph. We will see that the strong components of a directed graph become the elementary subgraphs of an associated bipartite graph. Section 6.4 is new.

6.1 Preliminaries

We start with formal definitions of the central concepts. We first recall some necessary terminology of the theory of digraphs that we will use in this chapter.

A *digraph* (directed graph) D is a pair (V, E), where V is a finite set of elements, called *vertices* (or *nodes*), and $E \subseteq V \times V$ is a set of ordered pairs (v_i, v_j) of vertices, called *arcs* (or *directed edge*). The number of vertices $n = |V|$ is the *order* of D. The number of arcs $m = |E|$ is the *size* of D.

A digraph H is a *subdigraph* of a digraph D if $V(H) \subseteq V(D)$ and $E(H) \subseteq E(D)$. If $V(H) = V(D)$, then H is called a *spanning subdigraph* (or a *factor*) of D.

If (v_i, v_j) is an arc of D, then v_i is called the *head* (or *initial endpoint*) and v_j is called the *tail* (or *terminal endpoint*). Graphically, the vertices can be represented by points, and (v_i, v_j) will be represented by an arrow connecting the points v_i and v_j, v_j being at the tip of the arrow.

An arc whose endpoints coincide is called a *loop*. Two arcs, or edges, are called *adjacent* if they have at least one endpoint in common. In this chapter we consider digraphs without multiple (parallel) arcs but which may contain loops.

Vertex u is called a *successor* of vertex v if there is an arc with v as its initial endpoint and u as its terminal endpoint. The set of all successors of v is denoted by

$$\Gamma^+(v) = \{u \in V : (v, u) \in E\}.$$

Similarly, vertex u is called a *predecessor* of vertex v if there exists an arc of the form (u, v). The set of all predecessors of vertex v is denoted by

$$\Gamma^-(v) = \{u \in V : (u, v) \in E\}.$$

The set of all *neighbors* of v is denoted by

$$\Gamma(v) = \Gamma^+(v) \cup \Gamma^-(v).$$

For a set $X \subseteq V$, we let

$$\Gamma^+(X) = \bigcup_{x \in X} \Gamma^+(x) \text{ and } \Gamma^-(X) = \bigcup_{x \in X} \Gamma^-(x).$$

The *outward demi-degree* (or *out-degree* for short) of a vertex v in D, denoted by $d^+(v)$, is the number of arcs starting at (leaving) v and the *inward demi-degree* (or *in-degree* for short) of a vertex v, denoted by $d^-(v)$, is the number of arcs terminating at (entering) v. The *total degree* (or just *degree*) of a vertex v, denoted by $d(v)$, is defined by

$$d(v) = d^+(v) + d^-(v).$$

Clearly, we have:

$$d^+(v) = \left|\Gamma^+(v)\right| \text{ and } d^-(v) = \left|\Gamma^-(v)\right|.$$

A *directed path* P of length k is a sequence of $k+1$ distinct vertices $v_0, v_1, \ldots, v_k$ together with the k distinct arcs $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$. The vertex v_0 is the *initial endpoint* (or *source*) of the path P, the vertex v_k is the *terminal endpoint* (or *target*) of the path P, and the remaining vertices are the *internal nodes* of the path P. A *directed circuit* is a (directed) path that begins and ends at the same vertex.

A digraph is called *strongly connected* (or *strong*) if, for any two vertices v_i and v_j, there exists a path from v_i to v_j and a path from v_j to v_i. The decomposition of a directed graph into strongly connected components is a fundamental tool in graph theory with various applications. There exist following algorithms for strong connectivity:

Year	Author(s)	Strategy/Remarks
1970	Purdom [249]	stack
1971	Munro [227]	disjoint set union
1972	Tarjan [280]	one depth-first search traversal
1976	Dijkstra [77, Chapter 25]	top-down development
1980	Kosaraju [2, Section 6.7],[67, Section 22.5]	unpublished
1981	Sharir [269]	two depth-first search traversals
1996	Cheriyan & Mehlhorn [62],[218, Section 7.4]	depth-first spanning tree
2000	Gabow [119]	path-based depth-first search
2013	here	matching theory

Table 6.1: History of algorithms for strongly connected components

A *strong component* of a digraph D is a maximal strongly connected subdigraph of D. We assume that the *trivial digraph*, consisting of exactly one vertex, is vacuously strong since it does not contain two distinct vertices. Corresponding to any digraph D, there is a new digraph whose definition is based on the strong components of D. Let $S_1, S_2, \ldots, S_p$ be the strong components of a digraph D. The *strong component graph* (also called the *condensation*) of D, is the simple digraph $SC(D)$ with vertex set $V(SC(D)) = \{s_1, s_2, \ldots, s_p\}$, such that there is an arc in digraph $SC(D)$ from vertex s_i to vertex s_j iff there is an arc in digraph D from a vertex in strong component S_i to a vertex in strong component S_j.

Multiple arcs from a given strongly connected component to another strongly connected component are merged.

Notice that the strong component graph of any digraph (some authors use the term *reduced digraph* to describe such digraphs) is a directed acyclic graph, that is, it contains no directed cycles. Note that at this point we allow acyclic digraphs to contain loops.

An *acyclic digraph* is often referred to by its abbreviation, DAG. The term DAG is typically pronounced as a word, not spelled out as an acronym.

Acyclic digraphs play a very important role in both theory and applications of digraphs and form a well-studied family of digraphs, in particular, due to the following important properties:

- In every directed acyclic graph there is at least one source (a vertex with no incoming edges: $d^-(v) = 0$) as well as at least one sink (a vertex of out-degree zero: $d^+(v) = 0$),
- Every acyclic digraph has a topological ordering of its vertices,
- For every vertex v there is a path from any source to v and a path from v to any sink.

Clearly, in any digraph all the vertices on a cycle belong to the same strongly connected component. A strongly connected component will be called a *source component* if it corresponds to a source vertex in a strong component graph. Analogously, a strongly connected component of D that corresponds to a sink of $SC(D)$ is called a *sink component*. A strongly connected component is *trivial* if it consists of one vertex without a loop, and is *non-trivial* otherwise.

In order to obtain short proofs of various results on subdigraphs or efficiently solve many problems dealing with directed graphs the following transformation of a directed graph D to a bipartite graph is extremely useful [143, page 411]. Let $BR(D)$ denote the bipartite graph with bipartition (V_1, V_2) defined as follows. Let $V(D) = \{v_1, \dots, v_n\}$, and put $V_1 = \{v'_1, \dots, v'_n\}$ and $V_2 = \{v''_1, \dots, v''_n\}$. The elements of V_1 are called *outward*, and the elements of V_2 are called *inward*. We next join v'_i to v''_j by an edge in $BR(D)$ iff there exists an arc of the form (v_i, v_j) in D. We call $BR(D)$ the *bipartite representation* of D.

Recall that a matching in an undirected graph $G = (V, E)$ is a set of edges from E, no two of which share a vertex, a maximum matching of G is a matching of maximal cardinality among all matchings of G and a perfect matching is a set of pairwise disjoint edges that cover all the vertices of G.

Let $D = (V, E)$ be a digraph with vertex set $V(D)$ and arc set $E(D)$. Further, let $\vec{g} = (g^-, g^+)$ and $\vec{f} = (f^-, f^+)$ be pairs of non-negative integer-valued functions defined on $V(D)$ such that $0 \le g^-(x) \le f^-(x) \le d^-_D(x)$ and $0 \le g^+(x) \le f^+(x) \le d^+_D(x)$ for every $x \in V(D)$. We say that the digraph D has a directed perfect $\vec{f}$-matching if there exists a spanning subdigraph $F \subseteq D$ such that $d^-_F(x) = f^-(x)$ and $d^+_F(x) = f^+(x)$ for all $x \in V(D)$. Analogously, a directed perfect $(\vec{g}, \vec{f})$-matching of D is defined to be a spanning subdigraph $H \subseteq D$ such that $g^-(x) \le d^-_H(x) \le f^-(x)$ and $g^+(x) \le d^+_H(x) \le f^+(x)$ for all $x \in V(D)$.

The existence of a directed perfect matching in a digraph is equivalent to the existence of a perfect matching in its corresponding bipartite representation. For given pairs of functions

$\vec{g} = (g^-, g^+)$ and $\vec{f} = (f^-, f^+)$ defined on $V(D)$, we define two functions $g, f : V(BR(D)) \mapsto \mathbb{Z}^+$ by

$$g(x) = \begin{cases} g^+(x), & \text{if } x \in V_1 \\ g^-(x), & \text{if } x \in V_2 \end{cases}$$

and

$$f(x) = \begin{cases} f^+(x), & \text{if } x \in V_1 \\ f^-(x), & \text{if } x \in V_2. \end{cases}$$

Then it is easy to see that D has a directed perfect $\vec{f}$-matching iff $BR(D)$ has a perfect f-matching, and that D has a directed perfect $(\vec{g}, \vec{f})$-matching iff $BR(D)$ has a perfect (g, f)-matching.

The following necessary and sufficient conditions for the existence of a directed perfect $(\vec{g}, \vec{f})$-matching are easily verified using matching theory (for bipartite graphs).

Theorem 6.1.1 *Let $D = (V, E)$ be a digraph with non-negative integer-valued functions $\vec{g} = (g^-, g^+)$ and $\vec{f} = (f^-, f^+)$ defined on $V(D)$. If D has a directed perfect $(\vec{g}, \vec{f})$-matching then*

$$\sum_{x \in X} g^+(x) \leq \sum_{x \in \Gamma^+(X)} f^-(x)$$

and

$$\sum_{x \in X} g^-(x) \leq \sum_{x \in \Gamma^-(X)} f^+(x)$$

for all $X \subseteq V$.

Proof The proof of this theorem is analogous to that of Theorem 2.4.5 in [212]. □

A cycle cover of an undirected graph is a spanning subgraph that consists solely of single cycles in which every vertex is a part of exactly one cycle. Cycle covers are also known as 2-factors since every vertex has degree two in a cycle cover.

A *cycle factor* in a directed graph $D = (V, E)$ is a spanning subdigraph of D in which the inward and outward degree of every vertex v is equal to 1:

$$d^+(v) = d^-(v) = 1.$$

Observe that a strongly connected digraph needs not necessarily have a cycle factor. We can use matching theory to find a cycle factor in a given digraph or to prove that none exists. We now begin with the necessary and sufficient condition for the existence of a cycle factor in a digraph.

Theorem 6.1.2 *A directed graph $D = (V, E)$ has a cycle factor iff*

$$|X| \leq \left| \bigcup_{x \in X} \Gamma^+(x) \right|$$

and

$$|X| \leq \left| \bigcup_{x \in X} \Gamma^-(x) \right|$$

for each $X \subseteq V$.

This looks, in fact, very much like a translation of Hall's Theorem (Theorem 2.5.2) into the language of directed graphs. Indeed, it is practically the same result. It can be proven by applying the Frobenius' Theorem (Theorem 2.5.3) to a bipartite representation $BR(D)$ constructed from digraph D. It is easy to see that D has a cycle factor iff the bipartite graph $BR(D)$ contains a perfect matching.

A *path factor* of a digraph D is a spanning subdigraph, each of whose components is a path. Note that a directed matching does not exclude cycles, whereas a path factor is acyclic. It is obvious that if a digraph has a cycle factor, then it also has a path factor. The converse is not true.

6.2 Canonical Decomposition

In this section we consider the canonical decomposition of directed graphs with respect to strong connectivity. We review the characterization of strong digraphs in terms of elementary graphs. The results presented in this section are mainly based on the joint work of Diane M. Johnson, Andrew L. Dulmage, & Nathan S. Mendelsohn [173].

Let D be a directed graph of order n and size m. We need the definition of the following special graph related to the bipartite graph $BR(D)$ introduced in the previous section.

The *augmented bipartite graph* corresponding to the directed graph D is a graph $BR^*(D)$ defined as the graph with the same vertex sets V_1 and V_2 as $BR(D)$. Every edge of $BR(D)$ is an edge of $BR^*(D)$ and in addition the n unordered pairs $\{v_i', v_i''\}$, $i = 1, 2, \ldots, n$ are edges of $BR^*(D)$.

Theorem 6.2.1 *If $BR^*(D)$ is the augmented bipartite graph corresponding to the directed graph D, then D is strongly connected iff $BR^*(D)$ is elementary.*

Proof See Theorem 4 in [173]. □

The last property gives us immediately a simple filtering algorithm with respect to the constraint STRONGLY_CONNECTED [11]. A brute force approach with complexity of $\mathcal{O}(m^2)$ has been described in [80]. A filtering algorithm according to a strong connectivity looks as follows. First, we create graph $BR^*(D)$. Next, we start with an initial empty matching and iterate over $i = 1, \ldots, n$. We add to the initial matching the edge $\{v_i', v_i''\}$. Then, the application of the alternating depth-first search to the initial matching M results in hyper-arc consistency with respect to the STRONGLY_CONNECTED constraint (Figure 6.1).

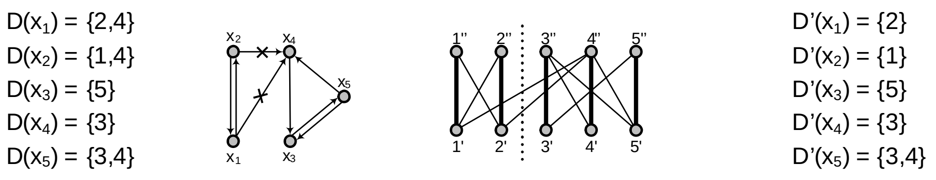


Figure 6.1: Pruning according to a strong connectivity

Theorem 6.2.2 *For any directed graph D, let $G_1, \dots, G_k$ be the canonical decomposition of $BR^*(D)$ into elementary subgraphs. If D_i, $i = 1, \dots, k$, is the subdigraph of D such that $BR^*(D_i) = G_i$, then $D_1, \dots, D_k$ are the strongly connected components of D.*

Proof See Theorem 6 in [173]. □

The following is the property of the augmented bipartite graphs:

- It is possible for $BR^*(D)$ to be connected without D being strongly connected.

6.3 Graph Partitioning Constraints

In this section we present a graph-theoretic analysis of a directed matching, using matching theory. We show a direct reduction of the directed matching problem on a digraph to the maximum matching problem on a bigraph. This reduction yields an algorithm to determine the partition of edges in the directed matching making use of decomposition theory for bipartite graphs. In this section examples are given and filtering algorithms are developed.

In order to investigate global constraints in the sequel we first introduce the digraph associated with any instance of these constraints[1]. Let $X = \{x_1, x_2, \dots, x_n\}$ be a set of n variables with respective finite domains $D_{x_i} \subseteq \{1, 2, \dots, n\}$ for $i = 1, 2, \dots, n$. To these variables we can associate the digraph $D = (V, E)$ with vertex set $V = \{v_i : 1 \leq i \leq n\}$ and arc set $E = \{(v_i, v_j) : j \in D_{x_i}, 1 \leq i, j \leq n\}$. Observe that the number of vertices is equal to the number of variables, and the number of arcs equals the sum of domain cardinalities. Thus, $n = |V|$ and $m = |E| = \sum |D_{x_i}|$ for all $x_i \in X$. Further, we have

$$\begin{aligned} d^+(v_i) &= |D_{x_i}|, \\ d^-(v_i) &= |D_{x_i}^{-1}| = |\{j : i \in D_{x_j}\}|. \end{aligned}$$

Hence, a directed edge (v_i, v_j) exists iff j is in the domain of variable x_i. Moreover, elimination of an arc (v_i, v_j) from the associated digraph during the pruning means removing of the value j from the domain of variable x_i.

A digraph associated with a global constraint can be viewed as an undirected graph by forgetting the orientation of its arcs, removing loops and merging all multiple resulting edges. We call this graph the *underlying graph* associated with the global constraint (the *undirected version* of a directed graph). Clearly, elimination of an edge $\{v_i, v_j\}$ from the underlying graph during the pruning is equivalent both to the removing of the value j from the domain of variable x_i (if it exists) and the removing of the value i from the domain of variable x_j (if it exists).

Our algorithm to find a partition of edges is based on the following simple observation.

Theorem 6.3.1 *Let D be a digraph associated with the global constraint and assume that there exists a one-to-one correspondence between the solution of the constraint and the directed perfect $(\vec{g}, \vec{f})$-matching in D. Further, let $BR(D)$ denotes the bipartite representation*

[1] We assume that the variables and their domain values represent the same set of elements.

of D. Then, the necessary condition to satisfiability of the constraint is that $BR(D)$ must have a perfect (g, f)-matching. Moreover, the mandatory (or forbidden) edges in a perfect (g, f)-matching of $BR(D)$ are mandatory (or forbidden) arcs in a directed perfect $(\vec{g}, \vec{f})$-matching of the digraph D.

Proof Suppose that $BR(D)$ has a perfect (g, f)-matching M consisting of edges $e_1, \ldots, e_{|M|}$. Then the arcs form a directed perfect $(\vec{g}, \vec{f})$-matching. Indeed, in the subdigraph D' induced by these arcs every vertex v_i has an out-degree and an in-degree equal to the number of matched edges incident to x_i on the outward side and to y_i on the inward side of $BR(D)$, respectively, and, according to the definition, such a subdigraph is precisely a directed perfect $(\vec{g}, \vec{f})$-matching in D. □

To illustrate this theorem, consider Figure 6.2.

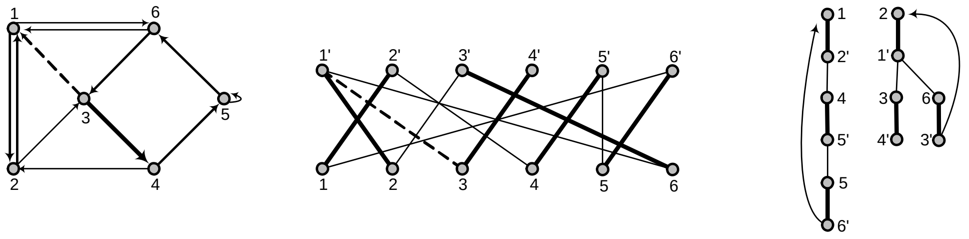


Figure 6.2: Pruning according to a directed matching

On the left side of the figure a directed graph D in which we wish to find a cycle factor is presented. In the middle of the figure the bipartite representation $BR(D)$ is depicted and a perfect matching M is shown. Observe that the perfect matching in $BR(D)$ corresponds to a cycle factor $1 \rightarrow 2 \rightarrow 1 \ \cup \ 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 3$ in D. In order to obtain the desired partition of edges we need only to apply an alternating depth-first search on $BR(D)$ with respect to M. The alternating depth-first forest (computed by the algorithm devised in Chapter 4) is shown on the right side of the figure. Since edge $\{1', 3\}$ in $BR(D)$ is forbidden and edge $\{3, 4'\}$ is mandatory, arc $(3, 1)$ in D is forbidden (drawn as a dashed arrow), and arc $(3, 4)$ is mandatory (drawn as a bold arrow).

A graph partitioning constraint can be seen as a problem for finding a partial graph of a given digraph associated with the constraint. Graph partitioning constraints are the main subject of the thesis written by Xavier Lorca [209]. From an interpretation point of view the subdigraphs so obtained are called *functional graphs* and they have the characteristic property that the out-degree of each vertex is equal to 1 (i.e. every vertex of the partition has exactly one successor). Thus, we will always have $g^+(v) = f^+(v) = 1$ for all vertices of D which corresponds to $g(v) = f(v) = 1$ for all vertices on the outward side of the corresponding bipartite graph $BR(D)$.

By the *continuity property* for the count variable of a graph partitioning constraint we mean that if a digraph associated with the constraint can be decomposed into r and s connected components, where $r \leq s$, then it can be decomposed into k components for all k such that $r \leq k \leq s$.

In the following we will demonstrate the reduction technique on some graph partitioning constraints. The method is composed of two main phases. The first one focuses on checking if a constraint has a solution. The second phase makes it possible to find some arcs that do not participate in any solution. This phase can be split into three steps: bounds filtering of count variable(s), pruning forbidden arcs when the count variable is instantiated to one of its extrema, and structural filtering. The bounds filtering concentrates on the cardinality of the expected partition. It consists of removing the values of count variable that are out of range. The structural filtering detects the arcs that do not belong to the expected partition and reduces the domain variables of constraint. All steps are complementary and together form a (partial) filtering for a graph partitioning constraint.

CIRCUIT The CIRCUIT constraint was first formulated by Jean-Louis Laurière [200]. It can be viewed as describing a Hamiltonian circuit[2] on a directed graph D associated with the constraint. The constraint is defined as CIRCUIT(NODES), where NODES is a collection of variables $\{x_1, \ldots, x_n\}$ whose domains are subsets of $\{1, \ldots, n\}$. It requires that a tuple $(d_1, \ldots, d_n)$ be a cyclic permutation of $(1, \ldots, n)$, where each $d_{i+1} = x_{d_i}$ and $x_{d_n} = d_1$. An associated directed graph D contains an arc (x_i, x_j) iff j belongs to the domain of x_i. An arc (x_i, x_j) of D is selected when $x_i = j$ and CIRCUIT requires that the selected arcs form a directed Hamiltonian circuit.

For the global constraint CIRCUIT there is a very immediate reduction from the Hamiltonian circuit, which demonstrates that reasoning with this constraint is generally intractable. For this reason, it is perhaps not surprising that, in the past, there has been little comment on it. Therefore, we now formally prove its computational intractability.

Theorem 6.3.2 *Deciding whether the* CIRCUIT *constraint has a solution is* NP*-complete.*

Proof We use a transformation from DIRECTED HAMILTONIAN CIRCUIT [131, Problem GT38] into the CIRCUIT constraint. Given a digraph $D = (V, E)$ with $n = |V|$ vertices and $m = |E|$ edges. We construct a CIRCUIT constraint, CIRCUIT$(x_1, \ldots, x_n)$ in which $D_{x_i} = \{j : (v_i, v_j) \in E\}$. The constructed CIRCUIT constraint has a solution iff the original DIRECTED HAMILTONIAN CIRCUIT problem has a solution. □

A famous combinatorial problem that can be modeled with the CIRCUIT constraint is the *Traveling Salesperson Problem* (abbreviated as TSP) [202]. Many other problems can be expressed in terms of Hamiltonian cycles, such as the Euler Knight's Tour Problem on a chessboard, or the Chinese Postman Problem [212, Section 6.5].

Clearly, checking a CIRCUIT constraint for satisfiability is equivalent to checking if an associated digraph has a Hamiltonian circuit, which is an NP-complete problem [182]. Achieving hyper-arc consistency for CIRCUIT is thus NP-hard. There exist, however, several necessary conditions that can be verified in polynomial time.

An obvious necessary condition for a digraph to be Hamiltonian is that the graph must be strongly connected. However, this condition is not sufficient. Another obvious and quite

[2]A digraph is *Hamiltonian* if it contains a directed circuit that visits each vertex once without touching any vertex more than once.

powerful necessary condition for a digraph to be Hamiltonian is the existence of a cycle factor. Both conditions can be checked in polynomial time. Clearly, a Hamiltonian circuit is a cycle factor but the converse is not necessarily true because some cycle factors may consist of several disjoint circuits.

We know that achieving hyper-arc consistency for CIRCUIT is NP-hard. Therefore, we will now describe some useful incomplete filtering methods, that run in polynomial time, which are only partially related to those presented in [57],[188] and [271].

One of the elementary filtering methods for CIRCUIT is based on the ALLDIFFERENT constraint [255]. The ALLDIFFERENT filtering method can be applied because all the variables of the NODES collection have to take distinct values. A further necessary condition to satisfiability of this constraint is to have at most one single strongly connected component.

The filter removes inconsistent values by eliminating non-Hamiltonian arcs from the associated digraph, that is, arcs that belong to no Hamiltonian circuit. Filtering can also be based on sufficient conditions for non-Hamiltonicity of a digraph, some of which appear in [202, Chapter 11]. Most of the known sufficient conditions for a digraph D to be Hamiltonian assert that if the degrees of the vertices of D are sufficiently large [22],[137],[214],[308], or D has enough arcs [205], then D is Hamiltonian. Unfortunately, the number of arcs must be nearly as large as the number of edges in a complete graph with n vertices.

We have taken care of many sufficient conditions for a digraph to have a Hamiltonian circuit. However, none of these are necessary conditions. For example, the oriented cycle C_n, the simplest Hamiltonian digraph of all, does not satisfy any of these conditions when n is large.

Latife Genç Kaya and John N. Hooker presented in [188] a recursive algorithm that eliminates non-Hamiltonian arcs from the graph via vertex separators. The filter is based on an idea put forward by Václav Chvátal that every Hamiltonian graph is 1-tough[3] (for details, see [202, page 405]). Their algorithm identifies almost all unsatisfiable instances and eliminates about one-third of the inconsistent values from the variable domains.

We now demonstrate a partial filtering method which explicitly uses the properties of the Gallai-Edmonds Canonical Decomposition and the results presented in the previous chapter (see Theorem 5.2.8). Our idea is based on identifying a perfect 2-matching in the underlying graph associated with the constraint. This follows from the fact that every Hamiltonian circuit is a 2-factor and each 2-factor can be considered as a generalization of a Hamiltonian circuit. Our routine looks for a perfect 2-matching in the underlying graph and the corresponding edge not belonging to it will be removed from the associated digraph (together with an opposite arc, if it exists). This will result in deleting some 2-cycles from the associated digraph, which cannot be a part of any Hamiltonian circuit. Note that the problem of removing all cycles of length two from the Hamiltonian digraph is NP-hard [131, Problem GT13]. However, our method coupled with the ALLDIFFERENT constraint gives a more effective pruning.

[3]A graph G is called *t-tough* if the deletion of an arbitrary set S of vertices leaves the rest of the graph either connected or else broken into no more than $|S|/t$ connected components.

Observe that neither filtering method is redundant of the other, but both combined together improve the propagation behavior in some cases, as can be seen in the following example. Let CIRCUIT$(x_1, \ldots, x_8)$ constraint have the following domains $D_{x_1} = \{3,4,5\}$, $D_{x_2} = \{4,7,8\}$, $D_{x_3} = \{2,7\}$, $D_{x_4} = \{1,5\}$, $D_{x_5} = \{6\}$, $D_{x_6} = \{3,5\}$, $D_{x_7} = \{2\}$ and $D_{x_8} = \{1,4\}$. Then pruning based on ALLDIFFERENT removes values 4 and 7 from D_{x_2} and value 2 from the domain of variable x_3. But filtering based on our method removes first value 3 from D_{x_1} and value 2 from the domain of variable x_3 (see Figure 6.3). Then the next step with the ALLDIFFERENT constraint removes values 4 and 7 from D_{x_2} and value 5 from the domain of variable x_6. Hence, the filter has deleted 2-cycle $6 \to 5 \to 6$ from the associated digraph, as belonging to no Hamiltonian circuit. The domains after two steps of the algorithm look as follows: $D_{x_1} = \{4,5\}$, $D_{x_2} = \{8\}$, $D_{x_3} = \{7\}$, $D_{x_4} = \{1,5\}$, $D_{x_5} = \{6\}$, $D_{x_6} = \{3\}$, $D_{x_7} = \{2\}$ and $D_{x_8} = \{1,4\}$.

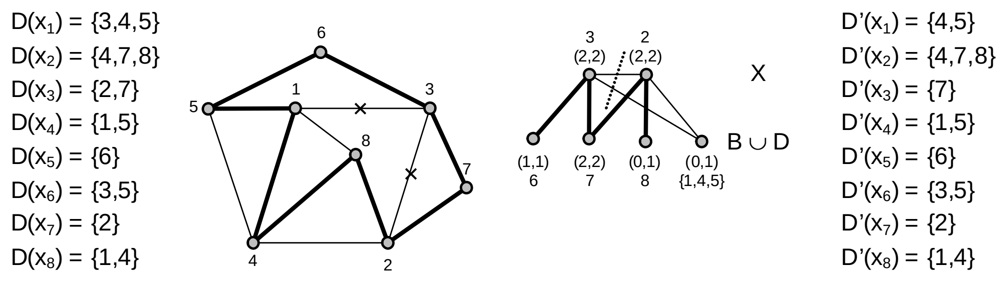


Figure 6.3: Pruning of the CIRCUIT constraint

We give a summary of our algorithm:

Algorithm 11 Partial filtering algorithm for the CIRCUIT constraint

```
Require: Digraph $D$ associated with the global constraint CIRCUIT(NODES)
Ensure: Incomplete pruning
  Remove all loops from the associated digraph $D$
  Check the necessary condition whether $D$ is strongly connected
  If $D$ has more than one strongly connected component, then the constraint is inconsistent
  If any of the sufficient conditions holds, then the constraint is satisfiable
  If the underlying graph has no perfect 2-matching, then the constraint is unsatisfiable
  Pruning according to the Gallai-Edmonds Canonical Decomposition
  Pruning according to the ALLDIFFERENT(NODES) constraint
```

All steps have a polynomial complexity. The removal of all loops takes $\mathcal{O}(n)$ time. Finding the strongly connected components of D requires $\mathcal{O}(m + n)$ time and space (see Table 6.1). The checking of any the sufficient conditions takes minimal $\mathcal{O}(1)$ [205] and maximal $\mathcal{O}(n^3)$ time [214]. A directed perfect matching M can be computed from scratch in time $\mathcal{O}(\sqrt{n} \cdot m)$. A perfect 2-matching of the underlying graph can then be found from M in time $\mathcal{O}(\sqrt{k} \cdot m)$, where k is the number of exposed vertices in G. Pruning according to the Gallai-Edmonds Canonical Decomposition can be performed in $\mathcal{O}(p \cdot m)$ time (see

Theorem 5.3.5), where p denotes the number of maximal extreme sets in the underlying graph. Pruning according to the Dulmage-Mendelsohn Canonical Decomposition can be realized in linear time (see Theorem 4.3.3).

CYCLE The CYCLE is a useful constraint that was introduced in CHIP [29] in order to tackle hard combinatorial problems. It can be used for modeling various problems such as the multiple traveling salesmen problem [202, Chapter 2 (Section 3.3) and Chapter 5 (Section 6.1)], the vehicle routing problem [202, Chapter 12],[29], and the balanced Euler knight problem [51].

The constraint has the form CYCLE(NCYCLE, NODES), where NCYCLE is a domain variable and NODES is a collection of domain variables. The CYCLE constraint partitions a given associated digraph described by the NODES collection into a set of vertex-disjoint cycles. The constraint requires that in the digraph there are exactly NCYCLE directed circuits, such that every vertex belongs to exactly one cycle.

As the first interpretation, the CYCLE(NCYCLE, NODES) constraint can be seen as the problem of finding NCYCLE distinct cycles in a directed graph in such a way that each vertex is visited exactly once. In the second interpretation, this constraint can be considered as the number of NCYCLE cycles of a permutation $\langle x_1, \ldots, x_n \rangle$.

Both observations are equivalent to the formulation of the ALLDIFFERENT constraint. This can be seen from the fact that a cycle in a directed graph D is a spanning subdigraph of D in which the in-degree and out-degree of every vertex v is equal to 1:

$$d^+(v) = d^-(v) = 1.$$

Note that the global constraint CIRCUIT is a special case of the global constraint CYCLE in which the first parameter NCYCLE is fixed to 1. Thus, the elementary filtering methods for the CIRCUIT constraint presented in the last example can be simply adapted to the CYCLE constraint. Moreover, a necessary condition to satisfiability of this constraint is to have at most $\max(D_{\text{NCYCLE}})$ strongly connected components. Clearly, if the number of strongly connected components equals NCYCLE, then, for each connected component, we can enforce pruning according to the global constraint CIRCUIT, as discussed earlier.

Nicolas Beldiceanu, within his Global Constraint Catalog [27], proposes the following algorithm: Since all variables in NODES have to take distinct values one can reuse the algorithms associated with the ALLDIFFERENT constraint. A second necessary condition is to have no more than $\max(D_{\text{NCYCLE}})$ strongly connected components. Since all the vertices of a circuit belong to the same strongly connected component, an arc going from one strongly connected component to another strongly connected component has to be removed.

This method is redundant. We will prove that an arc going from one strongly connected component to another one will be detected during pruning according to a directed matching.

Theorem 6.3.3 *Let $D = (V, E)$ be a directed graph associated with the global constraint* CYCLE*, a bipartite graph $BR(D)$ be the bipartite representation of D, and let M be a perfect matching in $BR(D)$. Then an edge (v_i, v_j) belongs to some perfect matching in $BR(D)$ iff vertices v_i and v_j belong to the same strongly connected component of D.*

Proof Since bipartite representation $BR(D)$ contains a perfect matching iff a digraph D has a cycle factor, thus when we take an arc e going from a strongly connected component S_i to another strongly connected component S_j, then we can never return to S_i in order to create a cycle involving the arc e. Hence, all such connecting arcs do not belong to any perfect matching. □

Moreover, a lower bound on the number of vertex-disjoint cycles in a digraph D is equal to the number of strongly connected components. However, the problem of finding an upper bound on the number of vertex-disjoint cycles in a given digraph D is NP-hard to compute. We will see that the maximum number of vertex-disjoint cycles in a digraph D is related to the minimum number of vertices in D needed to eliminate all cycles of D.

An upper bound on the number of the disjoint cycles can be obtained by solving the feedback vertex set problem. We will use some additional notation and terminology. Given a directed graph $D = (V, E)$, a *feedback vertex set* (abbreviated as FVS) is a set of vertices whose removal leaves an acyclic digraph. The problem is to find such a set with minimum cardinality. Obviously, forests and acyclic digraphs have a value of 0 since they have no cycles. In the literature, the term *cycle cutset* (or *cutset* in the short) has appeared as a synonym for feedback vertex set.

For a digraph D we denote by $\nu(D)$ the maximum number of vertex-disjoint cycles, and by $\tau(D)$ the minimum number of elements in a feedback vertex set of D. Clearly, we have $\nu(D) \leq \tau(D)$ and it is easy to construct an infinite family of digraphs such that only inequalities hold. Indeed, let $D = (V, E)$ be a digraph with vertex set $V(D) = \{x_1, x_2, x_3, x_4, y_1, y_2, y_3, y_4\}$ and arc set $E(D) = \{(y_i, x_i) : 1 \leq i \leq 4\} \cup \{(x_i, y_j) : 1 \leq i \neq j \leq 4\}$. Then $\nu(D) = 2$, but $\tau(D) = 3$.

In summary, we have the following results:

Theorem 6.3.4 *A lower bound on the number of cycles in the digraph D associated with the global constraint* CYCLE *equals the number of strongly connected components in D.*

Theorem 6.3.5 *The problem of determining the minimum number of cycles in the digraph D associated with the global constraint* CYCLE *is* NP*-hard.*

Proof The problem is clearly NP-hard as the answer is 1 if the digraph D has a Hamiltonian circuit, which is known to be an NP-complete task. □

Theorem 6.3.6 *The problem of determining the maximum number of cycles in the digraph D associated with the global constraint* CYCLE *is* NP*-hard.*

A natural greedy algorithm for finding the maximum number of vertex-disjoint cycles is to, repeatedly, find and remove the vertices belonging to the smallest cycle in the current digraph, until there are no more cycles left.

Richard M. Karp [182] was the first to prove that the FVS problem is NP-complete (see also [131, Problem GT7]). It is thus not surprising that the above mentioned decision problem for the maximum number of vertex-disjoint cycles is also NP-complete.

Notice that the computing of the maximal number of vertices such that the corresponding induced subdigraph forms a directed acyclic graph is equivalent to enforcing satisfiability of the global constraint CUTSET [99], which holds if its corresponding digraph possesses no vertex-disjoint cycles. Because it is an NP-complete problem, recent research in this area has been concentrated on designing algorithms finding a minimum cutset for a restricted class of digraphs and a relatively small cutset for general digraphs. The algorithm described in [99] returns two vertex sets, S_1 and S_2, such that $S_1 \cap S_2 = \emptyset$, $S_1 \cup S_2$ is a cutset of a digraph D and such that $|S_1| \leq \tau(D) \leq |S_1 \cup S_2|$. Note that if $S_2 = \emptyset$ then S_1 is guaranteed to be a minimum cutset.

Theorem 6.3.7 *Enforcing hyper-arc consistency on the count variable* NCYCLE *of the global constraint* CYCLE *is* NP*-hard.*

Proof It is easy to see that the problem is equivalent to the cycle cover problem. This employs a reduction from the PARTITION INTO HAMILTONIAN SUBGRAPHS problem [131, Problem GT13], originally shown NP-complete by Leslie G. Valiant [296]. □

When we impose the additional condition that each cycle has only two vertices, then the CYCLE constraint can be expressed by means of the constraint SYMMETRIC_ALLDIFFERENT. This constraint can be then interpreted as covering the associated graph with disjoint circuits of length two. Clearly, there exists in this case no solution to the CYCLE constraint when the number of variables is odd or NCYCLE $\neq \{\frac{n}{2}\}$. A complete filtering algorithm achieving hyper-arc consistency for the SYMMETRIC_ALLDIFFERENT constraint was proposed by Régin in [257]. Its running time is $\mathcal{O}(n \cdot m)$. This complexity has been improved in this thesis to $\mathcal{O}(p \cdot m)$ by making use of decomposition theory, where p is the number of maximal extreme sets in the underlying graph (see Theorem 5.3.5).

We now give a summary of the algorithm:

Algorithm 12 Partial filtering algorithm for the CYCLE constraint

```
Require: Digraph $D$ associated with the global constraint CYCLE(NCYCLE, NODES)
Ensure: Incomplete pruning
  If NCYCLE $= \{1\}$ then the constraint is equivalent to CIRCUIT(NODES)
  Pruning according to the perfect matching on $BR(D)$ (ALLDIFFERENT(NODES))
  Compute MINCYCLE as the number of strongly connected components of $D$
  Estimate MAXCYCLE (global constraint CUTSET(NCYCLE, NODES))
  Update variable NCYCLE according to MINCYCLE and MAXCYCLE values
  If there are no loops in $D$ and min(NCYCLE) $= \frac{n}{2}$ then the constraint is equivalent to
  SYMMETRIC_ALLDIFFERENT(NODES)
  If NCYCLE $= \{$MINCYCLE$\}$ then for each strong component $S_i$ pruning associated with the
  CIRCUIT($S_i$) constraint
```

All steps have a polynomial complexity. Finding the strongly connected components of D requires $\mathcal{O}(m+n)$ time and space (see Table 6.1). The existence of a cycle factor in a digraph can be checked and a cycle factor found, if it exists, in time $\mathcal{O}(\sqrt{n} \cdot m)$ [164]. The incomplete filtering algorithm for a CUTSET constraint has $\mathcal{O}(m + n \cdot \log n)$ time complexity [99]. Hyper-arc consistency for a SYMMETRIC_ALLDIFFERENT constraint can be achieved in polynomial time (see Section 5.4).

A thesis dealing with the CYCLE constraint is written by Eric Bourreau [51].

DERANGEMENT The DERANGEMENT constraint is a special case of the CYCLE constraint. It enforces the covering of an associated digraph by a set of vertex-disjoint proper cycles. In another interpretation it is required to have a permutation with no fixed points.

The pruning for achieving hyper-arc consistency is simple. From a digraph D associated with the global constraint DERANGEMENT just remove all loops in an iterative way to obtain a reduced digraph D'. This step corresponds to the normalization of the variable domains. Then construct an auxiliary bipartite graph $BR(D')$ and compute a perfect matching in it. There is a one-to-one correspondence between the solution of the constraint and the existence of the perfect matching. Checking the feasibility can be realized in $\mathcal{O}(\sqrt{n} \cdot m)$ time, hyper-arc consistency can be established in $\mathcal{O}(m)$ time (see Theorem 4.4.1).

From an interpretation point of view this constraint is related to ALLDIFFERENT with unary constraints $x_i \neq i$ for all i, since the number of cycles is free, and the variables and their domains represent the same set of elements. Observe that the bipartite graph associated with the directed perfect matching is equivalent to the value graph associated with the ALLDIFFERENT constraint.

SOFT_DERANGEMENT_VAR A problem is over-constrained when no assignment of values to variables is possible to satisfy the constraint. In this situation the goal is to find a compromise which allows some constraints to be violated and search for solutions that violate as few constraints as possible. The cost of the violation can be defined as the number of assigned values that should change in order to make the constraint satisfied. This measure is represented by the cost variable z which is to be minimized.

In this example we apply our method to the soft version of the DERANGEMENT constraint by introducing the notion of deficiency to directed graphs.

Some preliminary terminology is needed. Recall that in the maximum matching of G the number of exposed vertices is called the deficiency of G and is denoted by $\delta(G)$. Let the deficiency of D be the number of exposed vertices on the outward side of its bipartite representation $BR(D)$.

We aim at computing a lower bound of z in order to check the consistency of the global constraint. The following result is a direct consequence of Theorem 4.4.2.

Corollary 6.3.8 *Assume that a global constraint can be represented by a directed graph D and there exists a one-to-one correspondence between the solution of the constraint and the directed perfect matching in D. Then a lower bound of the cost variable z equals the deficiency of D. Further, if $\delta(D) < \max(D_z)$ then all the values of domains of variables are consistent with the global constraint. If $\delta(D) = \max(D_z)$ then values of the domains which* *are represented by forbidden arcs can be removed. Otherwise, if* $\delta(D) > \max(D_z)$ *then the constraint is inconsistent.*

Using the above result, we can formulate the following filtering algorithm that enforces hyper-arc consistency on the SOFT_DERANGEMENT_VAR constraint (cf. Algorithm 7):

Algorithm 13 Filtering algorithm for the SOFT_DERANGEMENT_VAR constraint

```
Require: Digraph $D$ associated with SOFT_DERANGEMENT_VAR($z$, NODES)
Ensure: Hyper-arc consistency or constraint not satisfied
  Compute a maximum matching in the bipartite graph $BR(D)$
  Compute the Dulmage-Mendelsohn Canonical Decomposition
  Determine subgraphs $G_O$, $G_U$ and $G_W$
  Determine the partition of edges
  Let $\delta$ denote the deficiency of the subgraph $G_O$
  If $\delta > \max(D_z)$ then the constraint is inconsistent
  If $\delta = \max(D_z)$ then all forbidden arcs must be removed from the digraph $D$
  If $\delta < \max(D_z)$ then all arcs in $D$ are allowed
  Update the domain of the cost variable $z$
```

The algorithm first computes a maximum matching in the bipartite graph $BR(D)$. This takes $\mathcal{O}(\sqrt{n} \cdot m)$ time. The next steps are of linear complexity.

The proof of Corollary 6.3.8 applies to any constraint whose graph representation resembles D and a solution corresponds to a directed perfect matching. For all such constraints that are consistent, hyper-arc consistency can be achieved in linear time, assuming that the maximum matching in $BR(D)$ is known. Note that this is equal to the complexity of achieving hyper-arc consistency on the hard version of these constraints.

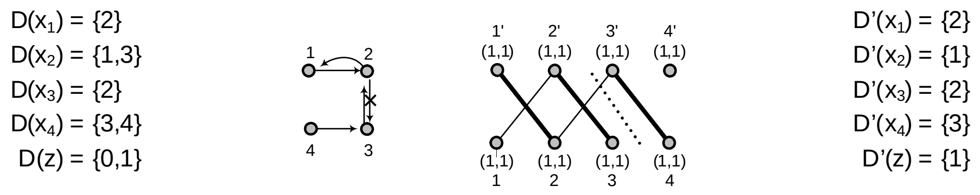


Figure 6.4: Pruning of the SOFT_DERANGEMENT_VAR constraint

TREE In this example we provide a quick description of the TREE constraint. The TREE constraint partitions a given directed graph into a forest of vertex-disjoint directed trees, where only certain vertices can be tree roots. More precisely, the digraph is partitioned into a set of vertex-disjoint anti-arborescences[4]. The constraint has the form TREE(NTREE, NODES), where NTREE is a domain variable specifying the number of trees in the tree partition, and NODES is a collection of n variables whose domains consist of elements of $\{1, \ldots, n\}$.

[4]A digraph D is an *anti-arborescence* with anti-root r iff for each vertex v in D there is a (directed) path from v to r and the underlying undirected graph of D is a tree.

The constraint holds if the associated digraph D is covered by a set of NTREE trees in such a way that each vertex of D belongs to one distinct tree. The arcs of the trees are directed from their leaves to their respective roots.

A hyper-arc consistency filtering algorithm for the global constraint TREE is described in [30]. This algorithm is based on the necessary and sufficient conditions that we now very briefly describe.

Before sketching a filtering algorithm for pruning the TREE constraint, we introduce some terminology regarding digraph $D = (V, E)$ and strong component graph $SC(D)$ associated with the TREE constraint. These definitions and notations are introduced in the original version of [30]:

- A vertex v such that $(v, v) \in E$ is called a *potential root*.
- A strongly connected component of D that contains at least one potential root is called a *rooted component*.
- A vertex v is a *door* of the strongly connected component if there exists an arc $(v, w) \in E$ such that v and w do not belong to the same strongly connected component of D.
- A vertex v is a *winner* if v is a door or a potential root.
- An arc $(v, w) \in E$ such that v and w do not belong to the same strongly connected component is called a *connecting arc*.
- Similarly, an arc $(v, w) \in E$ such that v and w belong to the same strongly connected component is called a *non-connecting arc*.
- *Enforcing an arc* (v, w) of D corresponds to removing from D all arcs (v, u) such that $u \neq w$.

Let D be a digraph associated with the TREE(NTREE, NODES) constraint and let $SC(D)$ be a strong component graph of D. Let MINTREE and MAXTREE respectively denote a lower and an upper bound on the number of trees for partitioning the digraph D into a set of vertex-disjoint anti-arborescences. MINTREE is equal to the number of sink components in $SC(D)$ (the number of strongly connected components in D with no outgoing arcs) and MAXTREE is equal to the number of potential roots in D. These bounds are sharp, this means that, for every MINTREE $\leq$ NTREE $\leq$ MAXTREE, we can construct the partition of edges into NTREE vertex-disjoint trees. The constraint TREE has at least one solution iff all sink components of D contain at least one potential root and $D_{\text{NTREE}} \cap [\text{MINTREE}, \text{MAXTREE}] \neq \emptyset$.

In the original filtering algorithm proposed in [30] the constraint is propagated according to the *strong articulation points* of D. Recall that a strong articulation point of a strongly connected component S is such a vertex s that if we remove it then S will be broken into at least two strongly connected components. Equivalently, s is a strong articulation point of D iff $D - \{s\}$ has more strongly connected components than D. However, it was shown in [100] that the concept of strong articulation points is not practical and the authors propose a new formulation of pruning rules based on dominators.

Recall that a vertex v dominates another vertex w with respect to a designated start vertex s if every directed path in D from s to w contains v. From the above definition, it can be easily seen that every vertex dominates itself. Also, it can be seen that the entry vertex s dominates all the vertices in the digraph. Hence, if both u and v dominates w, one of u and v dominates the other.

Here are some properties of the dominance relation.

- For all x, x dominates itself (reflexivity).
- If x dominates y, then y does not dominate x (asymmetry).
- If x dominates y and y dominates x, then $x = y$ (antisymmetry).
- If x dominates y and y dominates z, then x dominates z (transitivity).
- If x and y both dominate z, then either x dominates y or conversely.
- There may exist vertices x and y such that neither x dominates y nor y dominates x.

Hence, dominance relation is a partial order (for short, a *poset*). Improving on a previous work by Tarjan [281], who discovered an $\mathcal{O}(n \cdot \log n + m)$-time algorithm for finding dominators in an arbitrary digraph, Lengauer & Tarjan [204] proposed an $\mathcal{O}(m \cdot \log n)$-time algorithm and a more complicated $\mathcal{O}(\alpha(m,n) \cdot m)$-time version, where $\alpha(m,n)$ is an extremely slow-growing functional inverse of the Ackermann function [283]. An implementation of this algorithm in linear time is presented in [9].

Theses dealing with dominators are [135] (see also [251]). Table 6.2 presents a complexity survey for dominators (almost all algorithms are based on the depth-first search [280]).

Year	Author(s)	Complexity	Strategy/Remarks
1969	Lowry & Medlock [213]	$\mathcal{O}(n^4)$	brute-force
1972	Purdom & Moore [250]	$\mathcal{O}(n \cdot m)$	connectivity, shortest paths [76]
1974	Tarjan [281]	$\mathcal{O}(n \cdot \log n + m)$	disjoint set union, priority queue
1977	Aho & Ullman [3]	$\mathcal{O}(n \cdot (n + m))$	set definition of dominance
1979	Lengauer & Tarjan [204]	$\mathcal{O}(m \cdot \log n)$	link-eval without balancing
1979	Lengauer & Tarjan [204]	$\mathcal{O}(\alpha(m,n) \cdot m)$	link-eval with balancing
1983	Ochranová [232]	$\mathcal{O}(m)$	for control flow graphs
1985	Harel [152]	$\mathcal{O}(m + n \cdot \log^{(3)} n)$	linear disjoint set union [121]
1998	Buchsbaum et al. [53,54]	$\mathcal{O}(m + n)$	microtrees, bottom-up linking
1999	Alstrup et al. [9]	$\mathcal{O}(m + n)$	divide-and-conquer, microtrees
2001	Cooper et al. [66]	$\mathcal{O}(n^2)$	clever tree-based
2004	Georgiadis & Tarjan [136]	$\mathcal{O}(m + n)$	off-line nearest common ancestors

Table 6.2: History of algorithms for dominators

Clearly, a directed graph can have at most n dominators (strong articulation points). This bound is tight and is realized by the directed cycle C_n. Indeed, in this digraph each vertex is a dominator (strong articulation point).

In terms of the graph partitioning constraints, dual concepts and algorithms relating to dominators can be employed. A vertex d is a dominator of v with respect to a winner w iff there is no path from v to w in $D-\{d\}$. Other variants of the notion are defined analogously. In particular, we can exploit properties for edge dominators. We say that an arc (u,v) is an *edge dominator* of vertex w if every path from vertex s to vertex w contains arc (u,v). In a similar way as for dominators, all edge dominators of a given digraph D with a designated start vertex s can be computed in linear time.

Clearly, every dominator is a strong articulation point, but not conversely (cf. Figure 6.8 and Figure 8.1 in [251]). Hence, using dominators instead of strong articulation points leads to the better filtering. In general, if vertex u dominates vertex v then arc (v,u) does not belong to any solution. This follows from the fact that if every path from u to w requires v, then any path from v to u has to be forbidden.

Let us consider $S_i, 1 \leq i \leq p$, a strongly connected component of D, and let D_i be a set of dominators of S_i defined with respect to the winners in S_i. The removal of any dominator $d \in D_i$ creates two kinds of strongly connected components (for more details, see [34]):

- Δ_d is the (possibly empty) set of strong components from which no winner of S_i can be reached by a path that does not contain the dominator d,
- $\bar{\Delta}_d$ is the (possibly empty) set of strong components from which at least one winner of S_i can be reached by at least one path that does not contain the dominator d.

In addition, among the strongly connected components of Δ_d three types thereof, possibly empty, may be further distinguished:

- Δ_d^+ is the set of strong components corresponding to sources in $SC(\Delta_d)$,
- Δ_d^- is the set of strong components corresponding to sinks in $SC(\Delta_d)$,
- $\Delta_d^{\mp}$ is the set of the remaining strong components (neither sources nor sinks).

Let us consider some interesting properties of these strongly connected components.

Property 6.3.1 *Let d be a dominator in strongly connected component S_i with respect to winners. Then d belongs to all paths from any vertex of Δ_d to any vertex of $\bar{\Delta}_d$.*

Property 6.3.2 ([34, Proposition 1]) *If there exists a path factor in D then there exists a Hamiltonian path in Δ_d leading from the source component Δ_d^+ to the sink component Δ_d^-, and finishing on the dominator d.*

Property 6.3.3 *Let $dom(v)$ be the set of vertices dominated by a vertex v and let d be a dominator in S_i with respect to winners. Then $d \in dom(v)$ for every $v \in \Delta_d$.*

Pruning is then performed according to the following rule, which prevents the creation of (proper) cycles:

Theorem 6.3.9 *An arc (d,i) of a dominator d that reaches a vertex i of Δ_d is forbidden.*

Proof The claim follows from the fact that enforcing such an arc would lead to some strong components with no winners, and hence creating a cycle, which is a contradiction. □

We now give a summary of the full algorithm:

Algorithm 14 Filtering algorithm for the TREE constraint

```
Require: Digraph $D$ associated with the global constraint TREE(NTREE, NODES)
Ensure: Hyper-arc consistency or constraint not satisfied
  Compute MINTREE and MAXTREE
  Update variable NTREE according to MINTREE and MAXTREE values
  Check the conditions for satisfiability
  If NTREE $= \{$MINTREE$\}$ then any potential root in a non-sink component is forbidden
  If NTREE $= \{$MAXTREE$\}$ then every outgoing non-loop arc of each potential root is forbidden
  Pruning according to dominators of $D$ (see Theorem 6.3.9)
```

The presented filtering algorithm has a linear time complexity [100]. Computing the strongly connected components of D takes $\mathcal{O}(m+n)$ time (see Table 6.1). Checking that each sink component of D contains at least one potential root takes $\mathcal{O}(n)$ time. Testing whether MAXTREE $<$ min(NTREE) or max(NTREE) $<$ MINTREE takes $O(1)$ time. Pruning according to dominators (Theorem 6.3.9) can be easily performed in linear time (see Table 6.2).

BINARY_TREE The BINARY_TREE constraint is derived from the TREE constraint, which enforces the partitioning of an associated digraph into a set of vertex-disjoint binary trees. The arcs of the binary trees are directed from their leaves to their respective roots.

The constraint has the form BINARY_TREE(NTREE, NODES), where NTREE is a domain variable specifying the number of binary trees in the tree partition, and NODES is a collection of n variables. The constraint holds if the associated digraph D is covered by a set of NTREE binary trees in such a way that each vertex of D belongs to one distinct tree.

The filtering algorithm for the BINARY_TREE constraint is not known. Currently for this constraint one can use the algorithm associated with the general TREE constraint [30],[100] or a modified GLOBAL_CARDINALITY constraint for handling the fact that a successor has at most two predecessors. We show how to handle this constraint by means of the method described in this thesis. Although the proposed algorithm does not achieve hyper-arc consistency (the problem to find a spanning tree in which no vertex has degree larger than some given integer is NP-complete [131, Problem ND1]), it is relatively simple to implement.

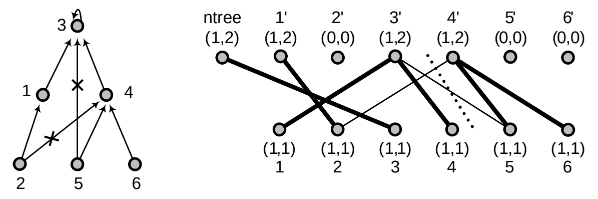
For any proper forest of binary trees the following holds:

$$\vec{g}(x) = \begin{cases} (0,1) \text{ for leaves} \\ (1,1) \text{ for internal nodes} \\ (1,0) \text{ for roots} \end{cases} \qquad \vec{f}(x) = \begin{cases} (0,1) \text{ for leaves} \\ (2,1) \text{ for internal nodes} \\ (2,0) \text{ for roots} \end{cases}$$

In the underlying graph a tree is binary if all internal vertices have degree at most three except the roots which have degree at most two and the leaves which have degree one. For proper trees all vertices have degree at least one (there are no isolated points). We model the TREE constraint by the associated digraph D in which the vertices represent the variables and the arcs represent the successor relation between them. Let R be the set of potential roots. In order to obtain the digraph D' associated with the BINARY_TREE constraint we add one dummy vertex v_0 to the input digraph D and declare that each of its predecessors is a potential root in R.

We construct a bipartite graph associated with the global constraint as described in the sequel with the following minor modifications. The vertices on both sides correspond to variables and there is an edge $\{v_i, v_j\}$ iff $j \in D_{x_i}$ and $i \neq j$. With every vertex we associate two functions g and f such that for each vertex v_i on the outward side we set $g(v_i) = f(v_i) = 1$ (since every vertex must have only one successor) and for the vertex v_j on the inward side we set $g(v_j) = 1$ and $f(v_j) = 2$ for roots and internal nodes (since every vertex can have at most two predecessors) or $g(v_j) = f(v_j) = 0$ for the vertices representing leaves (if they are initially fixed). Additionally, we connect all the vertices representing potential roots (these are variables with $i \in D_{x_i}$) to a single vertex labeled *ntree* and set $g(ntree) = \min(D_{\text{NTREE}})$ and $f(ntree) = \max(D_{\text{NTREE}})$. If we wish to find proper trees, then we set $g(r_i) = 1$ if in a sink component there exists only one potential root r_i. Otherwise, we set $g(r_i) = 0$. Then, we use the filtering algorithm described in Chapter 4 to determine the partition of edges and the bounds of the NTREE variable.

D(x_1) = {3}
D(x_2) = {1,4}
D(x_3) = {3}
D(x_4) = {3}
D(x_5) = {3,4}
D(x_6) = {4}
D(ntree) = {1,2}



D'(x_1) = {3}
D'(x_2) = {1}
D'(x_3) = {3}
D'(x_4) = {3}
D'(x_5) = {4}
D'(x_6) = {4}
D'(ntree) = {1}

Figure 6.5: Pruning of the BINARY_TREE constraint

On acyclic digraphs, it is easy to see that the feasibility can be checked in polynomial time by computing the perfect (g, f)-matching in the bipartite graph associated with the constraint. Hyper-arc consistency can then be achieved in linear time by determining the partition of edges. Since there are no cycles in the digraph D, the matched edges in the bipartite graph $BR(D)$ will form the partition into the forest of vertex-disjoint binary trees (see Figure 6.5).

Our algorithm has the following form:

Algorithm 15 Partial filtering algorithm for the BINARY_TREE constraint

```
Require: Digraph $D$ associated with the global constraint BINARY_TREE(NTREE, NODES)
Ensure: Incomplete pruning
  Pruning associated with the TREE constraint
  Pruning according to a directed matching
```

PATH From an interpretation point of view, the PATH constraint is the unary TREE constraint. This constraint requires the partitioning of a directed graph D into a set of vertex-disjoint (directed) paths.

The constraint has the form PATH(NPATH, NODES), where NPATH is a domain variable specifying the number of paths, and NODES is a collection of n variables. The constraint holds if the associated digraph D is covered by a set of NPATH paths in such a way that each vertex of D belongs to a single path.

A directed Hamiltonian path problem is NP-complete [131, Problem GT39]. The problem can be solved in polynomial time for acyclic digraphs.

For any proper path the following holds:

$$\vec{g}(x) = \begin{cases} (0,1) \text{ for sources} \\ (1,1) \text{ for internal nodes} \\ (1,0) \text{ for targets} \end{cases} \quad \vec{f}(x) = \begin{cases} (0,1) \text{ for sources} \\ (1,1) \text{ for internal nodes} \\ (1,0) \text{ for targets} \end{cases}$$

In an undirected graph each internal vertex of the path has degree two and each endpoint has degree one. The number of edges in the path equals the number of vertices minus 1.

Let MINPATH and MAXPATH denote the minimum and the maximum number of paths in a path factor of D. Clearly, MAXPATH is the number of potential roots for paths (i.e. MAXPATH $= |\{i : i \in D_{x_i}\}|$). When the number of paths is not fixed (i.e. $|D_{\text{NPATH}}| > 1$), the key point of any approach solving the PATH constraint is the evaluation of the lower bound on the number of paths for partitioning the digraph D associated with the global constraint.

Theorem 6.3.10 *Let D be a digraph associated with the* PATH *constraint. A lower bound on the number of vertex-disjoint paths for partitioning D is the number of sink components in $SC(D)$.*

However, the number of sink components in $SC(D)$ is not a sharp lower bound for the number of paths in D. In fact, finding a sharp lower bound makes the problem NP-complete, since we can easily reduce the Hamiltonian path problem to this problem. A sharper lower bound on the number of disjoint paths is introduced by the following result.

Theorem 6.3.11 *A lower bound on the number of vertex-disjoint paths in the digraph D associated with the* PATH *constraint is equal to* $\max\{1, \delta(D)\}$.

Both the directed perfect matching in D and the smallest possible number MINPATH of paths can be found in $\mathcal{O}(\sqrt{n} \cdot m)$ time [164]. Observe that in the case of non-acyclic digraphs the non-sharp lower bound on the number of vertex-disjoint paths introduced by Theorem 6.3.11 can be generalized to the sharp lower bound. This follows from the fact that some matched edges in $BR(D)$ form a cycle in D, which reduces the minimum number of vertex-disjoint paths at the most by 1.

We create a directed graph D associated with the PATH constraint as follows. The vertices correspond to variables and there is an arc (v_i, v_j) iff $j \in D_{x_i}$ and $i \neq j$. We add a new dummy vertex v_0 representing count variable NPATH. There is an arc from v_i to v_0 iff $i \in D_{x_i}$.

With every vertex we associate two pairs of functions $\vec{g} = (g^-, g^+)$ and $\vec{f} = (f^-, f^+)$ such that for each vertex v_i we set $g^-(v_i) = 0$, $f^-(v_i) = 1$ and $g^+(v_i) = f^+(v_i) = 1$. If we want to distinguish proper paths from isolated loops then we set $g^-(r_i) = 1$ if in the sink component S_i of $SC(D)$ only one potential root r_i exists. Additionally, we set $g^-(v_0) = \min(D_{\text{NPATH}})$, $f^-(v_0) = \max(D_{\text{NPATH}})$ and $g^+(v_0) = f^+(v_0) = 0$.

We construct a bipartite graph associated with the PATH constraint as follows. The vertices on both sides correspond to variables and there is an edge $\{v_i, v_j\}$ iff $j \in D_{x_i}$ and $i \neq j$. With every vertex we associate two functions g and f such that for each vertex v_i on the outward side we set $g(v_i) = f(v_i) = 1$ and for the vertex v_j on the inward side we set $g(v_j) = 0$ and $f(v_j) = 1$. Additionally, we connect all the vertices representing potential roots to a single vertex labeled *npath* and set $g(npath) = \min(D_{\text{NPATH}})$ and $f(npath) = \max(D_{\text{NPATH}})$. Then, we use the filtering algorithm described in Chapter 4 to determine the partition of edges and the bounds of the NPATH variable (see Figure 6.6). Since a path factor in an acyclic digraph has no cycles, this implies that the path factor for acyclic digraphs is easy to find.

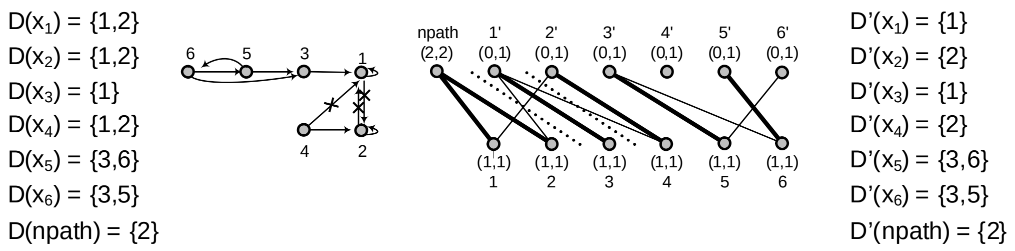


Figure 6.6: Pruning of the PATH constraint

Observe that the PATH constraint is very similar to the ALLDIFFERENT constraint, except that the potential roots have to be handled differently. In order to avoid cycles we have the additional restriction that each vertex on the path is not visited more than once, initial endpoints are excluded from the set, every element is distinct and must appear once yet the numbers representing potential roots appear exactly twice. On the bipartite representation $BR(D)$ of D we use Algorithm 5 to prune every arc of D that is incompatible with the PATH constraint.

Recall that vertex d is a dominator of vertex v with respect to vertex w iff there is no path from v to w in $D - \{d\}$. Thus, if vertex i dominates vertex j then arc (j, i) does not belong to any solution. But there are more propagation rules and according to dominators of D the pruning is performed in the following way (see [34]):

Theorem 6.3.12 *Let d be a dominator in D. Then the following arcs are forbidden in D:*

1. *An arc (d, i) going from the dominator d to Δ_d,*

2. *An arc (j, i) going from $\bar{\Delta}_d$ to Δ_d such that the strong component containing i is not a source,*

3. *An arc (i, d) going from Δ_d such that the strong component containing i is not a sink,*

4. *An arc (j, d) going from $\bar{\Delta}_d$ such that the strong component Δ_d is not empty.*

Proof These rules have not been proven in [34]. Thus, we formally do it.

1. Proof analogous as for Theorem 6.3.9.
2. The claim follows from the fact that there would be no way to visit some of the vertices of Δ_d if the strong component containing i were not to be a source.
3. The claim follows from the fact that there would be no way to visit some of the vertices of Δ_d if the strong component containing j were not to be a sink.
4. The claim follows from the fact that there would be no way to visit all the vertices of Δ_d if the arc (j, d) were to be enforced.

□

These propagation rules prevent the creation of (proper) cycles and enforce one single predecessor for each vertex of the strongly connected component. We demonstrate the theorem with Figure 6.7:

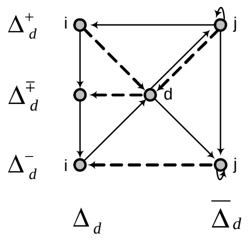


Figure 6.7: Pruning according to a dominator d

Theorem 6.3.13 ([34, Proposition 2]) *A lower bound on the number of vertex-disjoint paths partitioning the strongly connected component S_i with respect to a dominator d is provided by the minimum number of paths partitioning $\bar{\Delta}_d$ (the number of rooted components in $\bar{\Delta}_d$) minus 1 if there exists an arc $(u, v) \in S_i$ such that $u \in \bar{\Delta}_d$ and $v \in \Delta^+_d$.*

The authors of [34] do not include the filtering for arcs between two strongly connected components because they do not know how to do this efficiently by computing one single feasible flow. However, by means of our technique it is possible to make it during the pruning according to a directed matching. For example, our algorithm will detect the following mandatory arcs: $(0, 1)$, $(1, 2)$, $(3, 3)$, $(7, 7)$, $(8, 8)$, $(9, 9)$, $(10, 10)$, $(11, 13)$, $(13, 12)$ and the following forbidden arcs: $(2, 0)$, $(3, 2)$, $(3, 7)$, $(3, 8)$, $(4, 2)$, $(5, 1)$, $(6, 1)$, $(6, 7)$, $(6, 8)$, $(7, 12)$, $(8, 12)$, $(9, 13)$, $(10, 13)$, $(12, 7)$, $(12, 8)$, $(12, 11)$ (see Example 5 and Figure 6 in [34]).

This will be realized in the following way. First, according to dominator 2 of C_0, the arc $(2, 0)$, as leading to Δ_0, is detected by Case 1, the arcs $(5, 1)$ and $(6, 1)$ are detected by Case 2, the arcs $(3, 2)$ and $(4, 2)$ are detected by Case 4. Analogously, according to dominator 13 of C_1, the arc $(12, 11)$, as leading to Δ_1, is detected by Case 1, the arcs $(9, 13)$ and

$(10, 13)$ are detected by Case 2, the arcs $(7, 12)$ and $(8, 12)$ are detected by Case 4. Next, according to the directed matching, the arcs $(3, 7)$, $(3, 8)$, $(6, 7)$, $(6, 8)$, $(12, 7)$, $(12, 8)$ are detected. Additionally, the minimum/maximum number of vertex-disjoint paths is 5 and 7, respectively. Internal nodes are 1 and 13.

Our algorithm has the following form:

Algorithm 16 Partial filtering algorithm for the PATH constraint

```
Require: Digraph $D$ associated with the global constraint PATH(NPATH, NODES)
Ensure: Incomplete pruning
  Compute MINPATH and MAXPATH
  Adjust variable NPATH according to MINPATH and MAXPATH values
  If NPATH = {MINPATH} then any potential root in a non-sink component is forbidden
  If NPATH = {MAXPATH} then all outgoing non-loop arcs for each potential root are forbidden
  If $D$ is not acyclic then pruning according to dominators of $D$ (see Theorem 6.3.12)
  Pruning according to a directed matching
```

Theorem 6.3.14 *The count variable* NPATH *has the continuity property.*

Proof Consider a digraph D and a set of potential roots R. Assume that we have found a path factor of size $r < |R|$. We can build a path factor of size $r + 1$ by decomposing one of its paths with respect to its potential roots. These steps may be continued until a path factor of size $|R|$ is achieved. □

MAP In this example we provide a quick description of the MAP constraint. Next, we show how the propagation rules used for the constraints CIRCUIT, CYCLE and TREE can be implemented to generate a partial filtering algorithm.

The MAP constraint is a useful global constraint that can be used for covering a graph by a set of disjoint cycles and trees, and for modeling various problems such as random mappings [104], or graph related problems for the vertex-disjoint partitioning of graphs. However, before we define it more formally we need the description of the map. For a map, we take the definition from [266, page 459]:

> Every map decomposes into a set of connected components, also called *connected maps*. Each component consists of the set of all points that wind up on the same cycle, with each point on the cycle attached to a tree of all points that enter the cycle at that point.

The global constraint MAP has the form MAP(NBCYCLE, NBTREE, NODES), where NBCYCLE and NBTREE are domain variables, and NODES is a collection of vertices, which domain designates the successor vertex that will be used in the covering. The variables NBCYCLE and NBTREE are respectively equal to the number of cycles and the number of trees in the partition that can be interpreted as a map.

For any map the following holds (here, k denotes the inward degree of the vertex x):

$$\vec{g}(x) = \begin{cases} (0,1) \text{ for leaves} \\ (1,1) \text{ for internal nodes of trees} \\ (1,1) \text{ for internal nodes of cycles} \\ (1,0) \text{ for enter points} \end{cases} \qquad \vec{f}(x) = \begin{cases} (0,1) \text{ for leaves} \\ (k,1) \text{ for internal nodes of trees} \\ (1,1) \text{ for internal nodes of cycles} \\ (k,1) \text{ for enter points} \end{cases}$$

The MAP constraint was introduced within the Global Constraint Catalog [27] but no filtering algorithm is known. The purpose of this example is to present an incomplete filtering algorithm for the MAP constraint. The filter removes inconsistent values by eliminating arcs from the associated digraph, which do not belong to a map. We prove the necessary condition for an arc to be part of the MAP constraint, which provides the basis for eliminating arcs.

Before we more formally describe a filtering algorithm for the MAP constraint, we first need to introduce some terminology that will be used throughout this example. We will, as far as possible, use the notation introduced in [30], which we now extend:

- A strongly connected component that has a cycle factor is called a *cycle component*.
- A strongly connected component that has no cycle factor is called a *tree component*.

Observe that every tree component contains at least one tree. Thus, the number of trees in the MAP constraint is related to the number of sink components that have no cycle factor. Since a path is a degenerated tree the maximum number of paths in a MAP constraint is related to the number of trees.

We need to slightly modify the definition of the strong component graph associated with the MAP constraint. The strong component graph $SC(D)$ is derived from D with the following modification: to each strongly connected component of D that is a cycle component, we associate a vertex with a loop. The vertices without loops represent tree components.

It can be easily shown that if D contains a tree component then it must necessarily contain a tree. Thus, presence of trivial vertices in a digraph associated with the MAP constraint implies the presence of at least one tree.

We now introduce a theorem that will allow us to reduce the problem of finding the partition of a directed graph to the problem of estimating the bounds on the minimal and the maximal number of cycles and trees.

Theorem 6.3.15 *Let $D = (V, E)$ be an arbitrary finite (not necessarily connected) digraph such that every vertex has at least one successor. Then there exists a partition of D consisting of cycles, possibly loops, with trees having roots on their vertices.*

Proof One can construct the partition of D from its arcs by first selecting an arbitrary arc among them and then successively adding a new arc in such a way that it has at least one endpoint in common with the arcs already selected. Since the domains are finite, each such sequence must eventually loop back on itself. □

Hence, according to this theorem, no pruning is required for the MAP constraint when there are no given bounds on the number of cycles and the number of trees (e.g. $D_{\text{NBCYCLE}} = D_{\text{NBTREE}} = \{0, \ldots, n\}$). Further, every map has at least one cycle (including loops). When the above operation is repeated, starting each time from an element not previously hit, the vertices group themselves into components. This leads to a valuable characterization of such a partition: a map is a set of connected components that are cycles of trees. Thus, every connected component is a collection of rooted trees arranged in a cycle.

Theorem 6.3.16 *The minimum number of cycles in the digraph D associated with the global constraint* MAP *equals the number of sink components in $SC(D)$.*

Proof The claim follows from the fact that every sink component is strongly connected; this means it contains at least one cycle, and there is no path between two vertices that belong to two distinct sink components of D. □

Observe that the non-sharp lower bound on the number of cycles in the CYCLE constraint introduced by Theorem 6.3.4 is now generalized to the sharp lower bound on the number of cycles in the MAP constraint. But the results of Theorem 6.3.6 remain still valid.

Theorem 6.3.17 *The problem of determining the maximum number of cycles in the digraph D associated with the global constraint* MAP *is* NP*-hard.*

Proof The problem of finding the maximum number of vertex-disjoint cycles is related to the problem of determining a feedback vertex set of minimum cardinality. The claim follows now from the fact that the minimum cutset problem is an NP-hard task. □

Theorem 6.3.18 *Given a digraph D associated with the* MAP *constraint, an upper bound on the number of cycles partitioning D is given by the minimum feedback vertex set of D.*

Proof The claim follows from the fact that the size of the minimum feedback vertex set in a digraph D is no less than the maximum number of vertex-disjoint cycles in D. □

It is easy to see that a map where all the variables have distinct values leads to a set of cycles. Therefore, if digraph D associated with the MAP constraint has a cycle factor then we can immediately set MINTREE $= 0$. On the other hand, if digraph D has only cycles of length 1 (i.e. loops), we have a map that corresponds to a forest of trees and paths, and the algorithm for the TREE or PATH constraint can be used.

Theorem 6.3.19 *A lower bound on the number of trees in the digraph D associated with the global constraint* MAP *is equal to the minimal number of vertex-disjoint proper trees rooted on the sink components of $SC(D)$ and covering all tree components of $SC(D)$.*

Proof According to the definition of the MAP constraint the number of trees equals the number of arcs that do not belong to any cycle yet their tails are located on a cycle. Therefore, the minimal number of trees is equal to the minimal number of trees in D rooted at the sink components of $SC(D)$. Thus, the claim is trivially derived from the definition of the TREE constraint. □

Recall that in a topological ordering of a given directed acyclic graph D, each vertex v is associated with a value $ord(v)$, such that for each arc (u, v) we have $ord(u) < ord(v)$ and for each arc (v, w) we have $ord(v) < ord(w)$. The topological ordering can be found in linear time [67, Section 22.4].

In order to compute a lower bound on the number of trees for the MAP constraint we will impose a constraint INCREASING_NVALUE(NVAL, VARIABLES) with the following variables and domains. Let VARIABLES be a set of tree components of $SC(D)$. For every tree component s_i we put into the domain D_{s_i} the topological number $ord(s_j)$ of the sink component s_j in $SC(D)$ if there is a directed path from s_i to the sink component s_j. For variable NVAL we set $D_{\text{NVAL}} = \{1, \dots, |\text{VARIABLES}|\}$. Since enforcing hyper-arc consistency for the INCREASING_NVALUE constraint is $\mathcal{O}(m)$ [31] the computation of a lower bound on the number of trees is of linear complexity.

According to the property of DAGs we know that from every vertex v there exists a directed path in $SC(D)$ to at least one of its sinks. Thus, we make sure that we will explore a tree and all the trivial vertices of $SC(D)$ will be visited.

We know that for the MAP constraint the number of trees is the number of vertices directly connected to a cycle. According to this definition a trivial upper bound could be computed as follows: count the number of vertices for which at least one successor (that is not the vertex itself) is a part of a potential cycle. This can be made faster by using the following idea: every vertex that belongs to a strongly connected component containing more than one vertex or having a loop is a vertex that can be on a cycle. Therefore, we should find a vertex that has at least one successor different from those on the cycle.

It turns out that a sharper bound on the maximal number of trees can be obtained by solving the NVALUE constraint. More formally, an upper bound on the number of trees in the MAP(NBCYCLE, NBTREE, NODES) constraint equals n minus the minimum number of distinct values in the NVALUE(NVAL, NODES) constraint, where n is the number of variables and $D_{\text{NVAL}} = \{0, \dots, n\}$. Since the computing of the minimum number of distinct values for the NVALUE constraint is an NP-hard task [41] then the computing of the maximum number of trees for the MAP constraint is NP-hard, as well.

Theorem 6.3.20 *An upper bound on the number of trees for the global constraint* MAP *equals n minus the minimum number of distinct values in the* NVALUE(NVAL, NODES) *constraint, where $D_{\text{NVAL}} = \{0, \dots, n\}$. More formally,* $\text{MINTREE} = n - \min(D'_{\text{NVAL}})$.

Proof Let D be a directed graph associated with the MAP constraint. Suppose that we start from $\text{MINTREE} = 0$. In this case digraph D has, obviously, a cycle factor. This is the same as the NVALUE constraint with $D_{\text{NVAL}} = \{n\}$, since each value in NVAL must appear once, which leads to a set of cycles. Similarly, the case that $\text{MINTREE} = 1$ is the same as the NVALUE constraint with $D_{\text{NVAL}} = \{n - 1\}$. This follows from the fact that when $n - 1$ variables are distinct, two variables must take the same value. Since one value belongs to some cycle the second value is a terminal endpoint of a directed path leading to this cycle. Other cases can be handled similarly. Continuing this process, by the pigeonhole principle[5],

[5]The *pigeonhole principle*, called also *Dirichlet drawer principle*, states that if $n + 1$ objects (pigeons)

we find that a map having exactly k distinct values has at most $n-k$ trees. A bound is non-sharp since a vertex must not necessarily belong to a cycle. The same argument holds for each of the $1,\dots,n$ distinct values in the map, so the expected number of trees is obtained by the formula given in the theorem. Thus, we have proven the result. □

Theorem 6.3.21 *The problem of determining the maximum number of trees in the digraph D associated with the global constraint* MAP *is* NP*-hard.*

Proof The claim follows from the fact that computing the lower bound on NVAL variable of the NVALUE constraint is NP-hard. Such a constraint is called ATMOST_NVALUE [41]. □

Since the MAP constraint is satisfiable when the values NBCYCLE and NBTREE are of allowed range the incomplete pruning algorithm consists of detection forbidden arcs when NBCYCLE and NBTREE are instantiated to one of their extrema.

We demonstrate our algorithm with the following example (the sample digraph is taken from [65, page 23]).

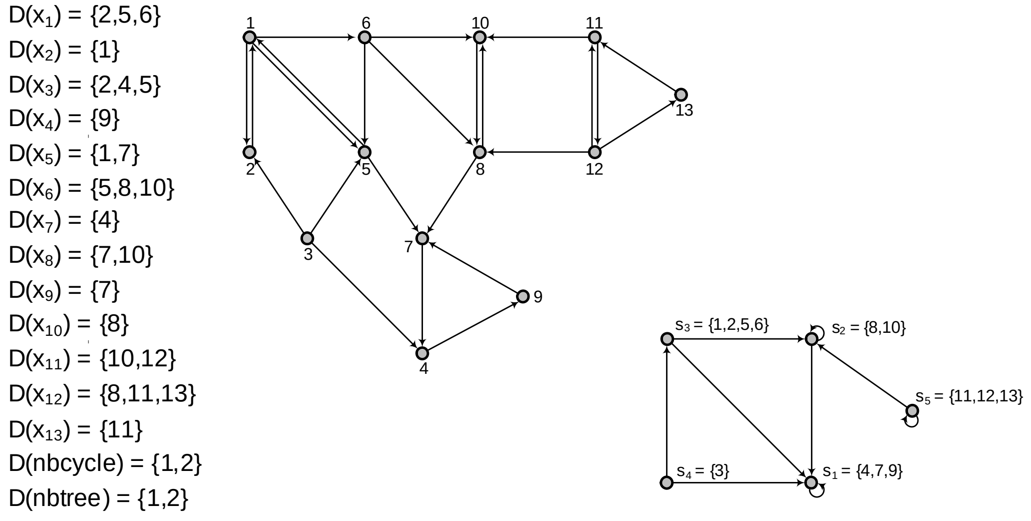


Figure 6.8: Checking feasibility of the MAP constraint

Figure 6.8 illustrates the different terms related to the MAP constraint. The MAP constraint is stated with the variable domains given on the left side of the figure. In the middle of the figure the digraph D associated with the MAP constraint is depicted. On the right side of the figure the strong component graph $SC(D)$ is shown. To each strongly connected component S_i of D corresponds a vertex s_i of $SC(D)$. Vertices s_1, s_2 and s_5 with loops represent cycle components, vertices s_3 and s_4 represent tree components. Further, vertices s_4 and s_5 represent source components and vertex s_1 represents a sink component.

The minimum number of cycles is 1 since there exists one sink component S_1. An upper bound on the number of cycles is 4 since $\tau(D) = 4$. A cutset is, for example, $\{1, 4, 10, 11\}$. Note that in our case $\nu(D) = \tau(D)$. The minimum number of trees is 1 since there exists

are placed into n boxes (pigeonholes), then some box contains more than one object.

one path leading to the sink component and covering all tree components of $SC(D)$. An upper bound on the number of trees is $5(= 13 - 8)$, since $\min(D'_{\text{NVAL}}) = 8$ for the constraint ATMOST_NVALUE. The constraint MAP holds since the values of count variables NBCYCLE and NBTREE lie within the computed extrema. The reader is invited to check some possible solutions with an arbitrary number of cycles and trees from range $1 \ldots 2$.

We conclude this subsection with a summary of the incomplete filtering algorithm:

Algorithm 17 Partial filtering algorithm for the MAP constraint

```
Require: Digraph $D$ associated with the constraint MAP(NBCYCLE, NBTREE, NODES)
Ensure: Incomplete pruning
  If NBCYCLE $= \{1\}$ and NBTREE $= \{0\}$ then the constraint specializes into CIRCUIT(NODES)
  If max(NBCYCLE) $> 1$ and NBTREE $= \{0\}$ then the constraint specializes into CYCLE(NBCYCLE, NODES)
  Compute MINCYCLE and MINTREE
  Estimate MAXCYCLE and MAXTREE
  If all sink components of $D$ are loops and their number is equal to NBCYCLE, and the minimum number of trees is equal to min(NBTREE), and the number of vertices in $D$ which have a successor which is located on a sink component of $D$ equals max(NBTREE), then the constraint is equivalent to TREE(NBCYCLE, NODES)
  If NBCYCLE $=$ {MINCYCLE} then the pruning is equivalent to the pruning for TREE(NBTREE, TC(D)), where TC(D) denotes a tree component of $D$, which is created in the following way: every strong component representing sink of $SC(D)$ is contracted to a single vertex with a loop
  If NBTREE $=$ {MINTREE} then the pruning is equivalent to the pruning for TREE(NBTREE, TT(D)), where TT(D) denotes a tree component of $D$, which is created in the following way: the tail of every arc going to a strong component representing sink of $SC(D)$ is replaced by a loop
```

Evaluating the complexity of the algorithm is done by analyzing the following steps. In order to compute the MINCYCLE value we need to find strongly connected components of D. This takes $\mathcal{O}(m + n)$ time (see Table 6.1). In order to estimate the MAXCYCLE value we use the algorithm for the CUTSET constraint that is of $\mathcal{O}(m + n \cdot \log n)$ time complexity [99]. The existence of a cycle factor can be decided in time $\mathcal{O}(\sqrt{n} \cdot m)$ [164]. The construction of the strong component graph $SC(D)$ can be easily done in linear time. The topological ordering of the vertices of $SC(D)$ may be obtained by performing a depth-first search [67, Section 22.4]. The minimum number of trees can be computed in linear time [31]. Estimating the maximum number of trees takes $\mathcal{O}(n^2)$ time when we use one of the approximation algorithms described in [41]. Adjusting the variables NBCYCLE and NBTREE can be carried out in constant time.

We analyzed the use of the global constraint MAP for modeling various problems and for partitioning graphs. We have shown that the constraint can be solved by using the other global constraints such as CUTSET, NVALUE, CIRCUIT, CYCLE, TREE or even SYMMET-

RIC_ALLDIFFERENT.

6.4 Counting the number of solutions

Enumeration problems, such as "how many solutions has a given problem", associated with NP-complete problems, are #P-complete. The concept of #P-completeness was introduced by Leslie G. Valiant [296,297] (see also [131, Section 7.3] and/or [241, Chapter 18]).

The problem of determining the number of matchings in a given graph is #P-complete. However, one might ask whether the number of solutions for a given problem instance can be determined in polynomial time, and this question is one of the central issues that we will investigate in this section.

CIRCUIT It is obvious that every strongly connected complete digraph K_n is Hamiltonian. The number of Hamiltonian circuits in K_n is equal to $(n-1)!$. For example, all circuits in K_4 are the following 3!=6 permutations with one single cycle: (2,3,4,1), (2,4,1,3), (3,1,4,2), (3,4,2,1), (4,1,2,3), (4,3,1,2).

CYCLE Observe that if the domains of the variables $x_1, \ldots, x_n$ range from 1 to n, then the total number of solutions of the CYCLE$(k, \{x_1, \ldots, x_n\})$ constraint corresponds to the number of ways to partition n objects into k non-empty cycles. These numbers are called the *Stirling numbers of the first kind* [7, Chapter III] and are denoted by $\left[{n \atop k}\right] = s(n,k)(-1)^{n-k}$, where $s(n,k)$ are the coefficients in the expansion:

$$\sum_{k=0}^{n} s(n,k)x_k = x(x-1)(x-2)\ldots(x-n+1).$$

For example, $x(x-1)(x-2)(x-3) = x^4 - 6x^3 + 11x^2 - 6x$, and hence there are $\left[{4 \atop 2}\right] = 11$ permutations of $\{1,2,3,4\}$ with 2 cycles: (1,3,4,2), (1,4,2,3), (2,1,4,3), (2,3,1,4), (2,4,3,1), (3,1,2,4), (3,2,4,1), (3,4,1,2), (4,1,3,2), (4,2,1,3), (4,3,2,1).

This correspondence between the number of solutions to the CYCLE constraint and the Stirling numbers of the first kind was primarily observed by Nicolas Beldiceanu and Evelyne Contejean [29]. Note that we get

$$\left[{n \atop 1}\right] = (n-1)!$$

$$\sum_{k=0}^{n} \left[{n \atop k}\right] = n!$$

$$\sum_{k=0}^{n} s(n,k) = 0$$

TREE In order to define an exact formula for the total number of solutions we first need to give the classical Faá di Bruno's formula for the number of ways to partition a set of n distinct elements into k non-empty subsets, such that each partition contains exactly λ_i sets of cardinality i for $i = 1, \ldots, n$:

$$n! \prod_{i=1}^{n} \frac{1}{\lambda_i!(i!)^{\lambda_i}}.$$

The following formula computes the number of partitions of a set with n elements into k non-empty subsets:

$$\sum n! \prod_{i=1}^{n} \frac{1}{\lambda_i!(i!)^{\lambda_i}},$$

where the sum is over all different solutions in non-negative integer n-tuples $(\lambda_1, \dots, \lambda_n)$ satisfying the conditions $\sum_{i=1}^{n} \lambda_i = k$ and $\sum_{i=1}^{n} i\lambda_i = n$.

These numbers are called the *Stirling numbers of the second kind* [7, Chapter III] and are denoted by $\left\{ {n \atop k} \right\} = S(n,k)$, where $S(n,k)$ are defined implicitly by the equation:

$$x^n = \sum_{k=0}^{n} S(n,k)x(x-1)(x-2)\dots(x-k+1).$$

It is straightforward to prove by induction or usage of the algebra of generating functions, that, if the domains of the variables $x_1, \dots, x_n$ range from 1 to n, then the total number of solutions of the TREE$(k, \{x_1, \dots, x_n\})$ constraint is equal to:

$$\sum_{\substack{\lambda_1+\lambda_2+\ldots+\lambda_n=k \\ \lambda_1+2\lambda_2+\ldots+n\lambda_n=n \\ \lambda_1,\lambda_2,\ldots,\lambda_n\geq 0}} n! \prod_{i=1}^{n} \frac{(i^{i-1})^{\lambda_i}}{\lambda_i!(i!)^{\lambda_i}}.$$

For example, when $n = 4$ and $k = 2$ the total number of solutions equals $4!\frac{(1^0)^1}{1!(1!)^1}\frac{(3^2)^1}{1!(3!)^1} + 4!\frac{(2^1)^2}{2!(2!)^2} = 4 \cdot 9 + 3 \cdot 4 = 48$. These are the following solutions: (1,1,1,4), (1,1,2,4), (1,1,3,1), (1,1,3,2), (1,2,1,1), (1,2,1,3), (1,2,2,2), (1,2,2,3), (1,2,4,1), (1,2,4,2), (1,3,1,4), (1,3,3,2), (1,3, 3,3), (1,3,4,4), (1,4,2,4), (1,4,3,1), (1,4,3,3), (1,4,4,4), (2,2,1,4), (2,2,2,4), (2,2,3,1), (2,2,3,2), (2,3,3,4), (2,4,3,4), (3,1,3,4), (3,2,2,4), (3,2,3,1), (3,2,3,3), (3,2,4,4), (3,3,3,4), (4,1,3,4), (4,2, 1,4), (4,2,3,2), (4,2,3,3), (4,2,4,4), (4,4,3,4) and (1,1,3,3), (1,1,4,4), (1,2,1,2), (1,2,2,1), (1,3, 3,1), (1,4,1,4), (2,2,3,3), (2,2,4,4), (3,2,3,2), (3,4,3,4), (4,2,2,4), (4,3,3,4). Note that we have $4 + 3 = \left\{ {4 \atop 2} \right\} = 7$.

Further, it can be proven that the number of all rooted trees over n vertices is given by:

$$(n+1)^{n-1}\ .$$

Observe that the above formula is very similar to the well-known Cayley's formula n^{n-2} counting labeled trees with n vertices [59].

BINARY_TREE It is an open problem to obtain explicit formula for the number of solutions to a BINARY_TREE constraint with all domains $[1, n]$. From an interpretation point of view this is equivalent to giving a formula for the number of forests on n nodes of rooted labeled binary trees.

PATH Analogously as for the TREE constraint, it can be easy proven by induction that the number of solutions of the PATH$(k, \{x_1, \dots, x_n\})$ constraint equals:

$$\sum_{\substack{\lambda_1+\lambda_2+\ldots+\lambda_n=k \\ \lambda_1+2\lambda_2+\ldots+n\lambda_n=n \\ \lambda_1,\lambda_2,\ldots,\lambda_n\geq 0}} n! \prod_{i=1}^{n} \frac{1}{\lambda_i!}.$$

For example, when $n = 4$ and $k = 2$ the total number of solutions is equal to $4!\frac{1}{1!}\frac{1}{1!}+4!\frac{1}{2!} = 24 + 12 = 36$. These are the following solutions: (1,1,2,4), (1,1,3,2), (1,2,1,3), (1,2,2,3), (1,2,4,1), (1,2,4,2), (1,3,1,4), (1,3,3,2), (1,3,4,4), (1,4,2,4), (1,4,3,1), (1,4,3,3), (2,2,1,4), (2,2, 3,1), (2,3,3,4), (2,4,3,4), (3,1,3,4), (3,2,2,4), (3,2,3,1), (3,2,4,4), (4,1,3,4), (4,2,1,4), (4,2,3,2), (4,2,3,3) and (1,1,3,3), (1,1,4,4), (1,2,1,2), (1,2,2,1), (1,3,3,1), (1,4,1,4), (2,2,3,3), (2,2,4,4), (3,2,3,2), (3,4,3,4), (4,2,2,4), (4,3,3,4). In particular, the number of Hamiltonian paths in a complete digraph of order n is equal to $n!$.

MAP Clearly, there are n^n different mappings from the set of integers between 1 and n onto itself. A simple counting argument says that the number of maps with k different values in the image is given by [266, page 454]:

$$k!\binom{n}{k}\left\{ {n \atop k} \right\}.$$

6.5 Summary

In this chapter we have studied pruning algorithms for various graph partitioning constraints. While some propagation rules for some of these constraints can be found in previous works, this study establishes a corpus of filtering rules which can be enforced in constraint programming community using graph partitioning constraints. This study provides optimal filtering rules for most of the constraints representable by directed acyclic graphs or proves their NP-hardness. The pruning is performed according to strongly connected components, directed matchings and dominators. Filtering algorithms are presented and their complexity is discussed. For some of the constraints only incomplete filtering algorithms are provided, leaving the design of more efficient algorithms as an open problem. We believe that this chapter illustrates and shows that other global constraints (such as ALLDIFFERENT, SYMMETRIC_ALLDIFFERENT, CUTSET or NVALUE) are of practical interest for describing pruning techniques for graph partitioning constraints.

The following tables summarize all the theoretical results for graph partitioning constraints we dealt with in this chapter. The first table gives an overview of complexities. The second table gives an exhaustive list of the basic properties for all graph partitioning constraints discussed in this chapter.

global constraint	model	complexity	checking feasibility	hyper-arc consistency	reference
CIRCUIT	digraph/graph	NP-hard	–	–	here
CYCLE	digraph/bigraph	NP-hard	–	–	here
DERANGEMENT	digraph/bigraph	tractable	$\mathcal{O}(\sqrt{n} \cdot m)$	$\mathcal{O}(m+n)$	here
SOFT_DERANGE-MENT_VAR	digraph/bigraph	tractable	$\mathcal{O}(\sqrt{n} \cdot m)$	$\mathcal{O}(m+n)$	here
TREE	digraph	tractable	$\mathcal{O}(m+n)$	$\mathcal{O}(m+n)$	[100]
BINARY_TREE	digraph/bigraph	NP-hard	–	–	here
	DAG/bigraph	tractable	$\mathcal{O}(\sqrt{n} \cdot m)$	$\mathcal{O}(m+n)$	here
PATH	digraph/bigraph	NP-hard	–	–	[34]
	DAG/bigraph	tractable	$\mathcal{O}(\sqrt{n} \cdot m)$	$\mathcal{O}(m+n)$	here
MAP	digraph	NP-hard	–	–	here

global constraint	pattern	number of components	count variable	lower bound	upper bound	continuity property
CIRCUIT	cycle factor	1	–			
CYCLE	cycle factor	NCYCLE	NCYCLE	non-sharp	non-sharp	-
DERANGEMENT	cycle factor		–			
SOFT_DERANGE-MENT_VAR	cycle factor		–			
TREE	anti-arbo-	NTREE	NTREE	sharp	sharp	+
BINARY_TREE	rescence	NTREE	NTREE	non-sharp	sharp	+
PATH	path factor	NPATH	NPATH	non-sharp	sharp	+
MAP	functional graph	NBCYCLE	NBCYCLE	sharp	non-sharp	+
			NBTREE	non-sharp	non-sharp	+

Table 6.3: Summary of results for directed graphs

Chapter 7

Weighted Graphs

This chapter presents a filtering technique based on weighted matching that uses decomposition techniques presented in previous chapters. In this chapter we will extend our considerations to the weighted case and we will again obtain a polynomial complexity.

The design of specific filtering algorithms to solve global constraints is necessary for the constraint programming community. It has been pointed out in [58] (see also [105]) that a weakness of constraint programming is related to combinatorial optimization problems. In this chapter we present an efficient way of implementing the pruning for the optimization constraints. The propagation mechanism is based on the algorithms computing weighted matchings. It will turn out that techniques for the weighted matching problems can be applied to filtering algorithms for global constraints much more explicitly than has been done before.

The work, which we present in this chapter, is based on the paper [71]. It is structured as follows. In Section 7.1 we formally introduce the approach. Section 7.2 provides a clear and complete description of the generic version of the algorithm to prune domains of variables in some global optimization constraints. In a number of important cases our method achieves hyper-arc consistency in polynomial time. A key feature in our implementation is the use of decomposition theory. Section 7.3 compares some of the optimization constraints reported in the literature and shows how they can be solved by means of the weighted matching. The fundamental idea of our algorithm is to apply the modified algorithm of Dijkstra for finding the shortest path.

7.1 Preliminaries

In this section we give some preliminaries on graph and matching theory.

For a graph G and $S \subseteq V(G)$, if $G - S$ is acyclic, then S is said to be a *feedback vertex set* of G. The size of the smallest feedback vertex set is called the *cycle cover number* of G and is denoted by $\tau(G)$. The corresponding problem of eliminating all cycles from a graph by means of the deletion of vertices does not have a simple solution. The latter question is difficult for both undirected and directed graphs, and even for bipartite graphs (see [182],[131, Problem GT7]). Clearly, $\tau(G) = 0$ iff G is a forest. For some common families of graphs there are formulas; e.g. $\tau(P_n) = 0, \tau(C_n) = 1$, $\tau(W_n) = 2$, $\tau(K_{r,s}) = r-1$ if $r \leq s$, $\tau(K_n) = n-2$, and so on. Thus, it holds $0 \leq \tau(G) \leq n-2$.

Let us now consider a graph G with a given optimal (g, f)-matching M. With respect to M we have five possible constellations for an alternating path P in G (cf. Figure 4.2):

1. an even alternating path leading from an exposed vertex and a free edge to a matched edge and a saturated (neutral or strictly positive) vertex, or conversely,
2. an even alternating path starting from a negative vertex and a matched edge and terminating with a free edge and a positive vertex, or conversely,
3. an odd alternating path between two distinct negative vertices and matched edges,
4. an odd alternating path between two distinct positive vertices and free edges.

All these possible alternating paths are determined by the function `pathDetected` in our algorithm. They may occur merely (except the first one) in the perfect (g, f)-matchings.

Recall that there exists a procedure for reducing the problem of finding a perfect degree-matching on a graph G into the problem of finding a perfect matching in an incremental graph G^*. The problem for weighted degree-matchings can be solved by using a technique discovered by Gabow, who in [115] presented an efficient $\mathcal{O}(f(V) \cdot \min\{m \cdot \log n, n^2\})$ algorithm for the construction of a maximum edge-weighted degree-constrained subgraph (see also [264, Section 33.6a]).

The *cost* of an edge is a value, called a *weight*, associated with the edge. Edge weights are assumed to be integers. In general, there is no assumption made on the signs of weights. However, the complexity of weighted matching algorithms depends on the signs of the costs, because the most efficient algorithms for searching of shortest paths deals only with non-negative weights.

The cost of a matching M in G is defined as the sum of the weights of all its edges:

$$\mathit{cost}(M) = \sum_{e \in M} w(e).$$

Throughout this chapter we will consider the following three problems of matching theory:

- the cardinality degree-matching problem
- the vertex-weighted degree-matching problem
- the edge-weighted degree-matching problem

The reader may be more familiar with two related problems: *maximum cardinality matching* and *maximum edge-weighted matching*, the latter usually known as a *maximum-weight matching*, but renamed here in order to distinguish it from its vertex-weighted counterpart. In the maximum cardinality matching problem we want to find a matching of maximum cardinality in a given graph G without any weight function. For the maximum edge-weighted matching problem a graph G and a weight function $w : E \mapsto \mathbb{Z}$, which assigns a weight to every edge, are given. The goal is then to find in G a matching of maximum cost. Obviously, the maximum cardinality matching problem is a particular case of the maximum edge-weighted matching problem, where all edges are assigned the same weight and, therefore, this problem is easier to solve than the more general maximum edge-weighted matching problem. The best known time complexity of algorithms based on the technique of augmentation that solves the former is $\mathcal{O}(\sqrt{n} \cdot m)$ [164],[221] while algorithms that solve the latter have a best time complexity of $\mathcal{O}(n \cdot (m + n \cdot \log n))$ [118], although algorithms of time complexity $\mathcal{O}(m \cdot n \cdot \log n)$ [20],[126] tend to be more practical.

A *maximum-weight perfect matching* is a perfect matching whose edges have the greatest total weight possible. A *maximum-weight matching* is defined similarly but is not required to be perfect. Since there may be negative weights, a maximum-weight matching does not have to be in general a maximum-size matching. But even under the assumption that all weights are positive the maximum-weight matching is not necessarily also the matching of maximum cardinality. However, in view of the preceding considerations, we will restrict our attention to the problem of determining the maximum-weight matching with maximum cardinality. It is intuitively clear that in order to find a maximum-weight matching among the matchings of maximum cardinality we only have to apply an algorithm to compute a maximum-weight matching with respect to the cost function defined by adding a large enough constant K (i.e. $K = 1 + n \cdot \max\{\cup_{e\in E}|w(e)|\}$) to the weight of every edge, but this is not required by the algorithms below because we will always consider maximum matchings.

In the *weighted matching problem with lower and upper bounds on the cost*, we want to determine whether we can find a maximum matching with the cost lying between the lower and upper bounds c_{min} and c_{max}. The weighted matching M is said to be an *admissible weighted matching* if the condition $c_{min} \leq cost(M) \leq c_{max}$ holds. The minimization problem can be then solved by setting $c_{min} = -\infty$ (we impose only an upper bound on the cost), while the maximization problem can be solved by setting $c_{max} = \infty$ (the cost is only bounded from below). If both $c_{min} = -\infty$ and $c_{max} = \infty$ then we have the classical maximum matching problem. Throughout this chapter, we will use the phrase "admissible weighted matching" to refer to an admissible weighted matching among the matchings of minimum deficiency.

We want to point out that it suffices to restrict our attention to maximum-weight matchings, because the respective minimum can be obtained if all weights $w(e)$ are replaced by $K - w(e)$, for some large positive constant K (i.e. $K = 1 + \sum_{e\in E}|w(e)|$).

Suppose now that some weights are negative. The edge-weighted graph can then be transformed into an equivalent one in which there are only non-negative weights. A tempting solution is to add a constant K (i.e. $K = 1 - \min\{\cup_{e\in E} w(e)\}$) to each edge weight, thus removing negative weights, calculate a minimum/maximum edge-weighted matching on the new graph, and then use that result in the original graph after adjusting the obtained cost accordingly (see [258, Discussion 7.1]). However, the naive implementation of this strategy does not work in general because matchings with many edges (e.g. maximum perfect (g, f)-matchings) become more costly than matchings with few edges (e.g. minimum perfect (g, f)-matchings). We are not aware of any method to solve this problem. However, in most real-life situations, the weights are of the same sign and hence this problem is non-existent. Therefore, we assume that in perfect (g, f)-matchings all the edges have non-negative weights.

A maximum-weight perfect (g, f)-matching must not be, in general, a maximum perfect (g, f)-matching. Analogously, a minimum-weight perfect (g, f)-matching must not be a perfect (g, f)-matching with the minimum number of edges (cf. Figure 7.1).

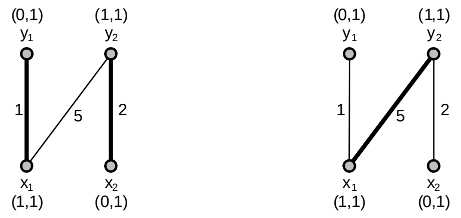


Figure 7.1: Minimum and maximum weighted perfect (g, f)-matching

A *weighted augmenting path* is an alternating path in which the total weight of the free edges exceeds the total weight of the matched edges. More formally, for a weighted augmenting path we have $cost(M \oplus P) > cost(M)$. A *weighted augmenting cycle* can be similarly defined. A matching M is a maximum-weight matching iff M has no weighted augmenting cycles or paths.

We know that the key paradigm in finding a maximum cardinality matching is the existence of augmenting paths. An analogous result holds for weighted matchings:

Theorem 7.1.1 (Minieka [222, Theorem 5.3]) *A weighted matching M in a graph G is optimal iff no weighted augmenting path relative to M exists.*

Here, an augmenting path for a weighted maximum/minimum matching is an alternating path in which the total weight of free edges is greater/lower than the total weight of matched edges.

Now let C be a weighted augmenting cycle which is used to augment M to M'. Then the cost of C is not defined as the sum of the weights of each of its edges but rather by

$$cost(C) = \sum_{e \in C \setminus M} w(e) - \sum_{e \in C \cap M} w(e),$$

i.e., it is the total weight of the free edges in C minus the weight of the matched edges. Thus, the cost of the matching $M \oplus C$ is enlarged by the cost of the (alternating) cycle C. Many of the difficulties associated with the above formula are overcome by working with a transformed set of costs. This motivates the use of the reduced weight w' instead of the actual weight w since because of the fact that $w'(e) = 0$ if $e \in M$, the cost definition becomes a simple sum:

$$cost(C) = \sum_{e \in C} w'(e).$$

Recall that the most efficient algorithms for the minimum/maximum edge-weighted matching problem follow the primal-dual paradigm. These algorithms construct a matching and a dual solution simultaneously. A potential function assigns an integer to every vertex. For a potential function $\pi : V \mapsto \mathbb{Q}$ and a weight function $w : E \mapsto \mathbb{Z}^+$, the *reduced weight* of an edge $e = \{v, w\}$ is defined as $w'(e) = w(e) - \pi(v) - \pi(w)$ (or $w'(e) = \pi(v) + \pi(w) - w(e)$, respectively). Observe that $w(e) \geq \pi(v) + \pi(w)$ (or $w(e) \leq \pi(v) + \pi(w)$, respectively) for every edge $e = \{v, w\}$. The algorithms maintain a matching M of minimum/maximum cost with the property that all edges have non-negative reduced weights, whereas every matched edge has reduced weight 0, and all exposed vertices have potential 0. The cost of the minimum/maximum edge-weighted matching equals the sum of the potentials of all saturated vertices. The cost of an alternating cycle is simply the sum of the reduced weights of all edges contained in it. However, for augmenting paths, the following result holds (the similar conditions can be proven for the remaining alternating paths):

Lemma 7.1.2 *Let M be any minimum-weight matching, and let any potential function π that proves the optimality of the matching M be given. For any even augmenting path P, with respect to M, leading from a positive vertex v_0 to a negative vertex v_t, the following holds:*

$$cost(P) = \sum_{e \in P} w'(e) + \pi(v_0) - \pi(v_t).$$

Proof (cf. Property 2.4 in [4]) Let $P = v_0..v_t$ be any even alternating path P from a positive vertex v_0 to a negative vertex v_t. Then

$$\begin{aligned} cost(P) &= \sum_{e \in P \setminus M} w(e) - \sum_{e \in P \cap M} w(e) \\ &= \sum_{e \in P \setminus M} (w'(e) + \pi(v_i) + \pi(v_j)) - \sum_{e \in P \cap M} (w'(e) + \pi(v_i) + \pi(v_j)) \\ &= \sum_{e \in P} w'(e) + \sum_{e \in P} (\pi(v_i) - \pi(v_j)) \\ &= \sum_{e \in P} w'(e) + \pi(v_0) - \pi(v_t). \end{aligned}$$

Notice that the last equality follows from the fact that for any alternating path P the expression $\sum_{e \in P} (\pi(v_i) - \pi(v_j))$ is equal to $\pi(v_0) - \pi(v_t)$ because for each vertex v_i on the path P, other than the endpoints v_0 and v_t, the term $\pi(v_i)$ occurs once with a positive sign and once with a negative sign. $\square$

Unfortunately, it is in general not true that the above given formulas for the reduced weights are valid. The situation here seems to be more complicated because in case of degree-matchings we have that $w(e) - \pi(v) - \pi(w) \leq 0$ (or $\pi(v) + \pi(w) - w(e) \leq 0$, respectively) holds for all matched edges, and $w(e) - \pi(v) - \pi(w) \geq 0$ (or $\pi(v) + \pi(w) - w(e) \geq 0$, respectively) for all free edges [24],[304]. By these conditions the reduced weights of matched edges can be negative, so the above lemma does not hold. An alternative way is based on a transformation to the ordinary weighted matching problem.

7.2 Computing the partition of edges

Suppose we have a weighted matching M in a graph $G = (V, E)$ and we want to search for all edges of the graph which do not appear in any admissible weighted matching. An additional step would be to identify the edges that participate in every admissible weighted matching. The overall aim is to establish a partition of the edge set E into three disjoint subsets:

- the set of edges belonging to no admissible weighted matching (forbidden edges),
- the set of edges belonging to some admissible weighted matching (allowed edges),
- the set of edges belonging to every admissible weighted matching (mandatory edges).

It is well known that when we solve the weighted matching problem we obtain a lower and an upper bound on the objective function and reduced weights[1]. In fact, weighted matching problems are problems of linear programming. Recall that the reduced weights estimate the increase of the objective function by at least $w'(x_i, d_j)$ when we enforce the assignment $x_i = d_j$. A huge advantage of this approach is that it can be applied very efficiently. Namely, reduced weights are obtained automatically when solving a linear program.

When bounds on the objective function are known, reduced weights resulting from linear solving can be used to adjust variable domains. The reduced weight of a variable is an estimate of how much worse the optimal solution becomes under changes to the value of that variable. Thus, our algorithm is based on the following observation. Edges whose reduced weights are sufficiently large (i.e. $cost(M_{min}) + w'(e) > c_{max}$) can be safely ignored, since they will never be part of an admissible edge-weighted matching. Analogously, edges whose reduced weights become sufficiently small in a maximum edge-weighted matching (i.e. $cost(M_{max}) - w'(e) < c_{min}$) are forbidden. This step of the algorithm is the so-called *quick elimination*. Hence, the filtering rule can be applied without additional computational costs. However, such a filtering algorithm does not, in general, establish hyper-arc consistency.

7.2.1 Graphs with vertex-weighted matchings

In this subsection we discuss polynomial algorithms for finding a maximum vertex-weighted matching and for computing a decomposition of edges in the graph G. We present our work in a bipartite matching context in order to relate the maximum vertex-weighted matching problem to other matching problems, comparing them from the algorithmic perspective.

Given a graph $G = (V, E)$ and a weight function $w : V \mapsto \mathbb{Z}$ for each vertex of G (weights are assigned to vertices, not edges), the cost of a matching M is defined to be the sum of the weights of the vertices covered by the matching M (we assume that $x \in M$ iff $d_M(x) > 0$ and x is saturated):

$$cost(M) = \sum_{x \in M} w(x).$$

[1]In terms of linear programming potentials are called *dual variables* and reduced weights are *slacks*.

Therefore, an *admissible vertex-weighted matching* is one which has the feasible cost among all possible matchings of G. Observe that if all the weights assigned to the vertices of the graph are equal, the problem considered is simply the problem of finding a maximum matching, and a maximum vertex-weighted matching is simply a maximum matching.

However, a reduction from the maximum edge-weighted matching problem to the maximum vertex-weighted matching problem is not possible in general. We are, therefore, inclined to think that the maximum vertex-weighted matching problem, as a particular case, is easier to solve than the more general maximum edge-weighted matching problem.

Of course, this also makes the maximum vertex-weighted matching problem more difficult to solve than the maximum cardinality matching problem, although the two latter problems are closely related. Indeed, the best known time complexity of algorithms that solve the maximum vertex-weighted matching problem is $\mathcal{O}(\sqrt{n} \cdot m \cdot \log n)$ [274], with algorithms of time complexity $\mathcal{O}(n \cdot m)$ [288, Chapter 4] being more practical.

Year	Author(s)	Complexity	Strategy/Remarks
1978	Megiddo & Tamir [216]	$\mathcal{O}(n \cdot \log n)$	edge-weighted graph
1984	Spencer & Mayr [274]	$\mathcal{O}(\sqrt{n} \cdot m \cdot \log n)$	divide-and-conquer
1987	Mulmuley & Vazirani [225]	-	theoretical
1998	Campêlo Neto & Klein [229]	$\mathcal{O}(m \cdot \log n)$	for chordal graphs
1998	Ahuja & Orlin [6]	$\mathcal{O}(n^3)$	for path graphs
1998	Ahuja & Orlin [6]	$\mathcal{O}(n^2 \cdot \log n)$	for path graphs
2001	Tabatabaee et al. [277]	$\mathcal{O}(n \cdot m)$	alternating paths
2004	Thiel [288, Chapter 4]	$\mathcal{O}(n \cdot m)$	weight-augmenting paths
2008	Katriel [186]	$\mathcal{O}(m + n_2 \cdot \log n_1)$	for convex bigraphs
2009	Halappanavar [148, Chapter III]	$\mathcal{O}(n \cdot \log n + n \cdot m)$	sort- and search-based

Table 7.1: History of algorithms for the vertex-weighted matching problem

In the same way as in the case of the edge-weighted matching, without loss of generality, it suffices to restrict our attention to maximum cost, because the respective minimum can be obtained by negating the weights, computing the maximum, and taking the opposite. However, a significant difference in the case of edge-weighted matchings is the fact that when all weights are positive then the maximum vertex-weighted matching is always a matching of maximum cardinality (see, for example, [274, Lemma 1],[225], or [148, Theorem III.2.1]). The converse clearly does not hold, since a maximum matching does not necessarily need to be a vertex-weighted matching of maximum cost.

Assume that we have a maximum cardinality matching M and we want to solve the problem of maximizing its cost. Let us consider alternating cycles and paths. An alternating cycle matches exactly the same vertices, and hence it has the same cost. For an even alternating path P with endpoints x and y (starting from a free edge and terminating with a matched edge) we have the following possibilities:

$$cost(P) = \begin{cases} 0, & \text{if } \begin{pmatrix} 0 < g(x) \le d_M(x) < f(x) \\ 0 < g(y) < d_M(y) \le f(y) \end{pmatrix} \bigwedge \\ w(x), & \text{if } \begin{pmatrix} 0 = g(x) = d_M(x) < f(x) \vee 0 < g(x) = d_M(x) + 1 \\ 0 < g(y) < d_M(y) \le f(y) \end{pmatrix} \bigwedge \\ -w(y), & \text{if } \begin{pmatrix} 0 < g(x) \le d_M(x) < f(x) \\ 0 = g(y) = d_M(y) - 1 \vee 0 < g(y) = d_M(y) \end{pmatrix} \bigwedge \\ w(x) - w(y), & \text{if } \begin{pmatrix} 0 = g(x) = d_M(x) < f(x) \vee 0 < g(x) = d_M(x) + 1 \\ 0 = g(y) = d_M(y) - 1 \vee 0 < g(y) = d_M(y) \end{pmatrix} \bigwedge \end{cases}$$

Hence, the cost of the alternating path P depends only on its endpoints. This follows from the fact that augmenting the matching along the alternating path P never exposes internal vertices of P that are already saturated. The formulas for odd alternating paths can be obtained in a similar way.

Unfortunately, the minimum vertex-weighted (g, f)-matching is quite different from the maximum one and seems to be more difficult to maintain. It turns out that the problem of determining the minimum vertex-weighted (g, f)-matching, when some weighted vertices with $g(x) = 0 \wedge f(x) > 2$ exist in the graph, is intractable in general.

Theorem 7.2.1 *The general minimum vertex-weighted (g, f)-matching problem is* NP*-hard.*

Proof Clearly, if $g(x) > 0$ for every vertex x then we have the classical vertex-weighted problem, which is tractable. Otherwise, we make a transformation from the *minimum cover problem* [182],[131, Problem SP5]. We will prove that the problem is intractable if $g(x) = 0$ for all vertices on one side of the bipartite graph with bipartition (X, Y). Let C be the collection of subsets of a finite set S in the minimum cover problem. Let every element of the set S be represented by the vertices of X on one side of the bipartite graph, and let every element of the collection C be represented by the vertices of Y on the other side of the bipartite graph. There exists an edge between $x \in X$ and $y \in Y$ iff x is an element of the subset y. Additionally, for vertices of X we set $w(x) = 0$ and the degree conditions to $g(x) = f(x) = 1$. Further, for vertices of Y we set $w(y) = 1$, and the degree conditions to $g(y) = 0$ and $f(y) = d(y)$, where $d(y)$ denotes the degree of the vertex y (the cardinality of the subset $c \in C$; $|c| > 2$, since otherwise the problem is solvable in polynomial time). There is a one-to-one correspondence between the solution to the minimum cover problem and the solution to the minimum vertex-weighted (g, f)-matching. Thus, the last problem is NP-hard. □

From an interpretation point of view, this problem is equivalent to the weighted domination problem, where one has to select a set of vertices in order to control every vertex of a graph and the use of the same vertex once or several times does not change the cost of the dominating set.

However, for convex bipartite graphs[2] the problem is solvable in polynomial time. We will now explain how to compute a minimum vertex-weighted (g, f)-matching in a convex bipartite graph, which corresponds to a configuration of variables of which all domains are intervals. We adapt our method from the algorithm presented in [28]. If we assume that the weighted vertices are arranged in increasing order of their minimum weight, which can be realized in $\mathcal{O}(n \cdot \log n)$ computational time, then the algorithm for computing the minimum-weight (g, f)-matching problem in convex bipartite graphs has a linear complexity of $\mathcal{O}(n)$. This leads to an overall complexity of $\mathcal{O}(n \log n + hn)$ for the filtering algorithm with respect to M_{min}, where h denotes half of the number of vertices incident with the forbidden edges.

In order to prove the next result, we need first to define the *interval subset sum* problem. Let finite set A, size $s(a) \in \mathbb{Z}^+$ for each $a \in A$, and positive integers L and U are given. The question, which appears, is whether there exists a subset $A' \subseteq A$ such that the sum of the sizes of the elements in A' lies within the interval from L to U. We have the following lemma:

Lemma 7.2.2 *The interval subset sum problem is* NP*-complete.*

Proof Direct reduction from SUBSET SUM [182],[131, Problem SP13] by letting $L = B = U$, where B is a positive integer from an arbitrary instance of SUBSET SUM. □

Theorem 7.2.3 *The problem to determine whether G has a vertex-weighted (g, f)-matching with cost lying between given bounds L and U is* NP*-complete.*

Proof In order to show that our problem is NP-complete we must show that it belongs to NP and that all NP-problems are polynomial reducible to it. The first part is easy: we need only to check whether the cost of a vertex-weighted degree-matching lies within the given interval. To prove the second part we show that the *interval subset sum* problem is polynomial time reducible to our problem. We transform from the *interval subset sum* problem. Let finite set A, size $s(a) \in \mathbb{Z}^+$ for each $a \in A$, and bounds L and U are given in the *interval subset sum* problem. Let every element of A be represented by the vertices $X = \{x_1, \ldots, x_k\}$, whereby $k = |A|$, on one side, and by the vertices $Y = \{y_0, y_1, \ldots, y_k\}$ on the other side of the bipartite graph. For every $i = 1, \ldots, k$ there exists an edge between x_i and y_0 and between x_i and y_i. Additionally, for vertices of X we set degree conditions to $g(x) = f(x) = 1$ and weight $w(x)$ to 0; for vertices of Y we set $g(y_0) = 0$, $f(y_0) = k - 1$, $w(y_0) = 0$ for vertex y_0 and $g(y_i) = 0$, $f(y_i) = 1$ and $w(y_i) = s(y_i)$ for all the remaining vertices. Then it is straightforward to check that there is a one-to-one correspondence between the solution to the interval subset sum problem and the vertex-weighted degree-matching problem. In consequence, our problem is NP-complete. □

In the above proof we have assumed that the subset A' does not allow repetition of elements. When this subset is considered to be a multiset then the polynomial time reduction from the *interval subset sum* problem looks as follows. For every $i = 1, \ldots, k$ we create a

[2]A bipartite graph G with bipartition (V_1, V_2) is called *convex* on V_2 if there exists an ordering of V_2 so that for any $x \in V_1$ the set $\Gamma(x)$ forms an interval in the ordering.

star graph $S_{k+1}^{(i)}$ of order $k+1$ with one internal (center) vertex $x_0^{(i)}$ and $k+1$ leaves $y_0^{(i)}, \ldots, y_k^{(i)}$. We set degree conditions $g(v) = f(v) = 1$ for all vertices, and define weights as in the construction given in the proof of the above theorem. Then there is a one-to-one correspondence between the solution to the interval subset sum problem and the vertex-weighted matching problem. Since we can verify this solution in polynomial time, this proves again that our problem is NP-complete.

Weighted alternating depth-first search

Given an arbitrary bipartite graph with an initial vertex-weighted (g, f)-matching M, we wish to compute the partition of edges. In this subsection we present a procedure that allows us to classify the edges of several alternating paths or cycles with only one search phase and to simply derive the decomposition of the graph. We use a special kind of depth-first search called a *weighted alternating depth-first search.* The purpose of the weighted alternating depth-first search is to detect the forbidden edges in the vertex-weighted matching. The routine will traverse the bipartite graph in the following manner. If the level is even all adjacent matched edges leave the vertex; when the level is odd all edges (except matched ones) leave the vertex. Thus, an alternating depth-first search simulates the traversing on a directed graph and constructs layers that alternately use matched and free edges. For odd levels, the vertices are simply given by the mates of the vertices from the previous even level. For even levels, the next vertex is some unvisited neighbor of the vertex from the previous odd level. It should be obvious that a weighted alternating depth-first search maintains weighted alternating paths and cycles.

In order to develop an algorithm we first introduce some notations. In fact, our algorithm is adapted from one presented in Chapter 4, therefore, only differences and additional steps will be reported here.

The crucial steps are the labelings of vertices. We start with the computation of the reduced weights with respect to the current matching M. It holds that $w'(x) = 0$ if $d_M(x) > 1$ for x on the odd level or even root, and $w'(x) = 0$ if $d_M(x) \geq 1$ for x on the even level or odd root; otherwise $w'(x) = w(x)$. Moreover, the cost of an alternating path P with endpoints x and y equals

$$\mathit{cost}(P) = \begin{cases} w'(y) - w'(x), & \text{if } x \text{ and } y \text{ are on the even levels} \\ w'(x) - w'(y), & \text{if } x \text{ and } y \text{ are on the odd levels} \\ -w'(x) - w'(y), & \text{if } x \text{ is on the even level and } y \text{ is on the odd level} \\ w'(x) + w'(y), & \text{if } x \text{ is on the odd level and } y \text{ is on the even level} \end{cases} \tag{7.1}$$

Notice that these formulas are used by the function $\texttt{costPath}(x, y)$ in our algorithm to compute the cost of the alternating path starting at x and terminating at y.

A weighted alternating depth-first search of the graph G with an initial maximum vertex-weighted degree-matching M visits all vertices of G in order of non-decreasing weights. In the case of a matching with the minimum cost the visited vertices are sorted in non-increasing order.

If the alternating depth-first search discovers an alternating cycle (this is the case for back edges) then this cycle is contracted by replacing all its vertices with the vertex, called *core* of this cycle (analogy to the base of a blossom), whose level is minimum. When a nested cycle is detected (this is the case for non-tree edges leading to cycles with gray cores), then these cycles are merged into one, with the core closest to the root of the tree. Additionally, for every core we determine the minimum weight of all critical vertices (i.e. $0 = g(x) < f(x)$ and $d_M(x) = 0$ on the odd level or $d_M(x) = 1$ on the even level) contained in the shrunken cycles represented by this core. By these operations we guarantee that every alternating path leading to any alternating cycle will have the minimum possible cost.

If the alternating depth-first search discovers an alternating path (this is the case for tree edges), then it marks all edges on this path as allowed and vertices as belonging to the path. All detected alternating paths lead always from the current vertex t_i to the root r_i. We process with alternating paths in the same way as with alternating cycles: for every vertex on the alternating path we determine the minimum weight of all critical vertices contained in the path. By this operation we guarantee that every alternating path leading to another alternating path will have the minimum possible cost. If the search encounters a vertex belonging to any alternating path (this is the case for non-tree edges leading to black vertices), then, if the cost is admissible, a new alternating path will be discovered.

The computation of the partition of edges works as follows. In the first phase we look for even alternating paths that start at a saturated vertex and terminate at an exposed vertex. This situation can only occur in the subgraph G_U. We pick such a vertex, make it the root and grow a weighted alternating depth-first tree. Thus, the first phase performs the search on every connected subgraph of G_U starting from saturated vertices in B_2 and matched edges.

The second phase performs the search on every connected subgraph of G_W. Observe that it is not necessary to look for alternating cycles but only to examine all critical vertices (if they exist). Thus, this phase can consist of maximal two steps. The first step performs the weighted alternating depth-first search from critical vertices with $d_M(x) = 0$ and free edges, the second step starts from critical vertices with $d_M(x) = 1$ and matched edges.

For a maximum vertex-weighted matching we process the vertices in non-decreasing weight order. Thus, the weights of the roots of the weighted alternating depth-first trees can only increase during the phase of the search. This has the following consequence. When we grow a weighted alternating depth-first tree with root r without discovering an alternating path or cycle, we know that all vertices in the tree can only reach exposed vertices with weight at most $w(r)$. And hence, they do not have to be taken into account by later calls.

Let us summarize some important properties of the weighted alternating depth-first search:

- Each vertex on the even level is adjacent by a matched edge with a vertex on the odd level.

- Each vertex on the odd level is adjacent by a free edge with a vertex on the even level.

- Every tree has a negative (or a neutral) root on the even level, or an exposed (or a positive) root on the odd level.
- Alternating cycles are detected by back edges.
- Nested cycles are detected by cross (or forward) edges leading to a cycle with a gray core.
- Alternating paths are detected by tree edges leading either to a positive vertex on the even level, or to a negative vertex on the odd level, or to an exposed vertex on the even level.
- Alternating paths are detected by non-tree (forward or cross) edges leading to a vertex belonging to an alternating path.
- Admissible paths are detected by tree edges, or non-tree edges leading to an admissible alternating path or to an alternating cycle with critical vertices.
- Edges included in any alternating cycle or any admissible alternating path are allowed.
- Matched edges included in no alternating cycle nor admissible alternating path are mandatory.
- Free edges included in neither the alternating cycle nor in the admissible alternating path are forbidden.

The entire procedure is presented in Algorithm 18.

Let us show by an example how our algorithm works. The corresponding bipartite graph is depicted in the figure below. For the sake of simplicity, we define the weight of the vertex to be equal to its number. Since there are exposed vertices (in the subgraph G_U), we will grow trees with roots on the even level.

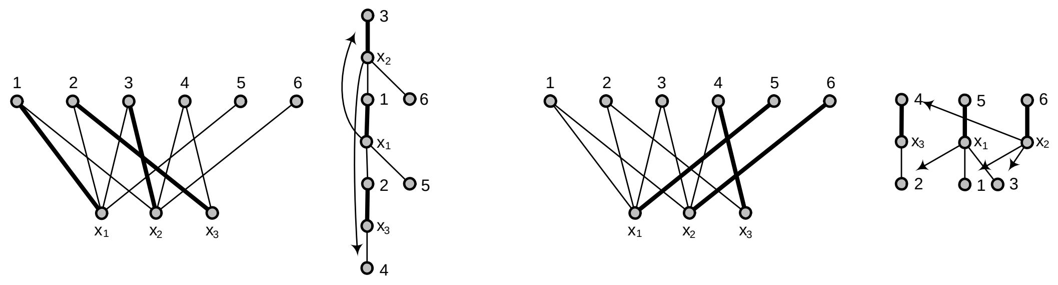


Figure 7.2: Weighted alternating depth-first search

On the left side of the figure the minimum vertex-weighted matching ($cost(M_{min}) = 6$) is shown. We grow a weighted alternating depth-first tree with a value 3 at the root, because it is the first value in decreasing order of weights of all saturated vertices. We find a weighted alternating path $3x_2 1x_1 2x_3 4$, as leading from saturated vertex 3 to exposed vertex 4, with cost $1 (= 4 - 3)$, then an alternating cycle $3x_2 1x_1 3$ is detected. Then the second alternating

Algorithm 18 Weighted Alternating Depth-First Search of G

Require: Bipartite graph G with a maximum vertex-weighted (g, f)-matching M_{max}

Ensure: Partition of edges into MANDATORY, ALLOWED and FORBIDDEN

```
procedure WeightedAlternatingDFS(r,s)
set $color[s] \leftarrow GRAY$
for every edge $\{s,t\} \in M$ if $level[s]$ is even or $\{s,t\} \notin M$ if $level[s]$ is odd do
  if $color[t] = WHITE$ then {tree edge}
    set $parent[t] \leftarrow s$
    set $level[t] \leftarrow level[s] + 1$
    perform WeightedAlternatingDFS(r,t)
  else {gray or black vertex}
    if $color[core(t)] = GRAY$ then {cycle detected}
      mark edge $\{s,t\}$ as ALLOWED
      perform ContractCycle(s,t) (see Algorithm 4)
    else if $path[core(t)] <> NIL$ then {forward or cross edge}
      let $c = costPath(r, path[core(t)])$
      if $c \geq w_{min}$ then
        mark edge $\{s,t\}$ as ALLOWED
        set $path[s] \leftarrow core(t)$
      end if
    end if
  end if
end for
set $color[s] \leftarrow BLACK$
if $path[s] = NIL$ and $pathDetected(r,s)$ then
  let $c = costPath(r,s)$
  if $c \geq w_{min}$ then
    set $path[s] \leftarrow s$
  end if
end if
if $path[s] <> NIL$ and $s \neq r$ then
  set $p \leftarrow parent[s]$
  mark edge $\{s,p\}$ as ALLOWED
  set $t \leftarrow p$
  if $path[p] <> NIL$ then
    set $t \leftarrow path[p]$
  end if
  if $pathDetected(r,t)$ and $costPath(r, path[s]) < costPath(r,t)$ then
    set $path[p] \leftarrow p$
  else
    set $path[p] \leftarrow path[s]$
  end if
end if
```

path $3x_21x_15$ with cost $2(= 5 - 3)$ will be found. The forward edge $\{x_2, 4\}$, as leading to a vertex belonging to an alternating path, discovers a new alternating path $3x_24$ with cost $1(= 4 - 3)$. The fourth alternating path $3x_26$ with cost $3(= 6 - 3)$ is detected. Since there are no unvisited vertices, our algorithm terminates.

Now consider the right side of the figure, where the same graph with the maximum vertex-weighted matching is shown ($cost(M_{max}) = 15$). We build a weighted alternating depth-first forest starting from saturated vertices in increasing order of their weights. First, we grow a tree with root 4 and we discover one alternating path $4x_32$ (detected by tree edge $\{x_3, 2\}$ leading to exposed vertex 2) with cost $-2(= 2 - 4)$. Next, we grow a tree with root 5 and we discover three alternating paths: $5x_12$ (detected by cross edge $\{x_1, 2\}$ leading to an alternating path) with cost $-3(= 2 - 5)$, $5x_11$ (detected by tree edge $\{x_1, 1\}$) with cost $-4(= 1 - 5)$, and $5x_13$ (detected by tree edge $\{x_1, 3\}$) with cost $-2(= 3 - 5)$. Finally, we grow a third tree with root 6 and we discover three alternating paths: $6x_24x_32$ (detected by cross edge $\{x_2, 4\}$) with cost $-4(= 2 - 6)$, $6x_21$ (detected by cross edge $\{x_2, 1\}$) with cost $-5(= 1 - 6)$, and $6x_23$ (detected by cross edge $\{x_2, 3\}$) with cost $-3(= 3 - 6)$. Since all the vertices have been visited, we may stop the algorithm.

We conclude this subsection with an analysis of the running time of the algorithm. If we suppose that the arithmetical operations such as additions and subtractions, as well as comparisons, can be done in constant time, then the algorithm is of linear complexity. We assume that the evaluation of boolean expressions is performed from left to right in a short-circuiting way. Because the search strategy guarantees that each edge is encountered at most three times (the first time when the edges are traversed, the second time when the cycles are being contracted, and the third time during the backtracking), the total time to perform the weighted alternating depth-first search is $\mathcal{O}(m + n)$. We had assumed that we have an ordering of the vertices according to their weights, because the algorithm for computing the vertex-weighted matching provides such a sorting (for details, see [288, Chapter 4]).

7.2.2 Graphs with edge-weighted matchings

In this subsection an algorithm for the determination of the partition of edges is developed. The algorithm itself is slightly involved, and the example given in the sequel will also be useful in understanding the algorithm.

In a similar way as in the case of the vertex-weighted matching, the interval subset sum problem can be polynomially reduced to the edge-weighted matching problem. This transformation leads to the following result being the direct consequence of Theorem 7.2.3:

Corollary 7.2.4 *The edge-weighted (g, f)-matching problem with lower and upper bounds on the cost is intractable in general.*

Proof Just as in the proof of Theorem 7.2.3 we transform from the interval subset sum problem. Let an arbitrary instance of the interval subset sum problem be given. In this problem we have set A of numbers $\{a_1, \ldots, a_k\}$, size $s(a_i) \in \mathbb{Z}^+$ for each $a_i \in A$, the lower bound L, and the upper bound U. We create a star graph S_k with an internal vertex x_0.

Let every element of the set A be represented by an edge $e_i = \{x_0, x_i\}$ for which we set cost $w(e_i) = s(a_i)$. Additionally, we set degree conditions as follows: for internal vertex x_0 we set $g(x_0) = 1$ and $f(x_0) = k$, whereas for all leaves x_i, $i = 1, \dots, k$, we set $g(x_i) = 0$ and $f(x_i) = 1$. The whole construction obviously requires only polynomial time. To prove that this reduction works, we can easily verify that there is a one-to-one correspondence between the solution to the interval subset sum problem and the solution to the edge-weighted (g, f)-matching problem. This completes the proof. □

We want to compute a maximum edge-weighted matching in G efficiently. There are well-known algorithms that compute the maximum edge-weighted matching in a bipartite graph, i.e. the popular Hungarian algorithm for the assignment problem. The running times of the best algorithms for the maximum-weight matching problem in bipartite graphs are usually stated as $\mathcal{O}(n \cdot (m + n \cdot \log n))$ [4, Chapter 12.4] and $\mathcal{O}(\sqrt{n} \cdot m \cdot \log(nW))$ [123] (here, W denotes the largest edge weight).

Table 7.2 summarizes known polynomial-time algorithms for the problem (cf. [264, Section 17.5a]). Time bounds are stated in terms of the number n of vertices, the number m of edges, and in some cases in terms of an upper bound W on the edge weights (assumed in these cases to be integers). The algorithms whose time bounds involve W assume integer weights, whereas others run on arbitrary rational or real weights.

Year	Author(s)	Complexity	Strategy/Remarks
1931	Egerváry [92]	-	theoretical
1946	Easterfield [87]	$\mathcal{O}(2^n \cdot n^2)$	transversals
1949	Robinson [263]		cycle canceling
1955	Kuhn [198,199]	$\mathcal{O}(n^4)$	Hungarian method
1957	Munkres [226]	$\mathcal{O}(n^4)$	Hungarian method
1960	Iri [172]	$\mathcal{O}(n^2 \cdot m)$	voltage configurations
1969	Dinits & Kronrod [79]	$\mathcal{O}(n^3)$	diagonal scaling
1970	Edmonds & Karp [91]	$\mathcal{O}(n^3)$	labeling method
1971	Tomizawa [289]	$\mathcal{O}(n^3)$	reduction and induction
1977	Johnson [175]	$\mathcal{O}(n \cdot m \cdot \log_d n)$, $d = \lceil \frac{m}{n} \rceil$	d-ary heaps [174]
1983	Gabow [117]	$\mathcal{O}(n^{\frac{3}{4}} \cdot m \cdot \log W)$	cost scaling
1984	Fredman & Tarjan [109]	$\mathcal{O}(n \cdot (m + n \cdot \log n))$	Fibonacci heaps
1988	Gabow & Tarjan [122,123]	$\mathcal{O}(\sqrt{n} \cdot m \cdot \log(nW))$	approximated scaling
1992	Orlin & Ahuja [239]	$\mathcal{O}(\sqrt{n} \cdot m \cdot \log(nW))$	improved scaling
1996	Cheriyan & Mehlhorn [62]	$\mathcal{O}(n^{\frac{5}{2}} \cdot \log(nW)(\frac{\log^{(2)} n}{\log n})^{\frac{1}{4}})$	bit compression
1999	Kao et al. [179]	$\mathcal{O}(\sqrt{n} \cdot m \cdot W)$	unfolded graphs
2001	Kao et al. [180]	$\mathcal{O}(\sqrt{n} \cdot m \cdot W \cdot \log_n \frac{n^2}{m})$	unfolded graphs
2012	Duan & Su [82]	$\mathcal{O}(\sqrt{n} \cdot m \cdot \log W)$	Balinski & Gomory [18]

Table 7.2: History of algorithms for the edge-weighted bipartite matching problem

The most efficient algorithms for weighted matching problems use abstract data structures, called *priority queues* (or *heaps*), consisting of a collection of elements, each with an associated priority. On a priority queue the following operations are possible [67, Section 6.5]:

insert(x,k) – inserts a new element with priority k into the queue (heap),

find_min() – finds and returns an element with the minimal priority,

decrease_key(x,k) – decreases the priority of the element in the queue,

extract_min() – removes and returns an element with the smallest priority.

Clearly, there are n `insert` operations, n `extract_min` operations and at most $m-n+1$ `decrease_key` operations. An implementation of a priority queue is said to be efficient if each operation takes $\mathcal{O}(\log n)$ amortized time. Running times of several implementations of priority queues are shown in the following table:

Year	data structure	amortized running time	references
1964	binary heap	$\mathcal{O}(m \cdot \log n)$	Williams [306]
1972	leftlist heap	$\mathcal{O}(m \cdot \log n)$	Crane [69]
1975	d-ary heap	$\mathcal{O}(m \cdot \log_d n)$, $d = \lceil \frac{m}{n} \rceil$	Johnson [174]
1977	mergeable heap	$\mathcal{O}(m \cdot \log \log C)$	van Emde Boas et al. [298]
1978	binomial heap	$\mathcal{O}(m \cdot \log n)$	Vuillemin [305], [52]
1984	Fibonacci heap	$\mathcal{O}(m + n \cdot \log n)$	Fredman & Tarjan [109]
1985	skew heap	–	Sleator & Tarjan [273]
1986	min-max heap	–	Atkinson et al. [16]
1986	pairing heap	$\mathcal{O}(m + n \cdot \log n)$	Fredman et al. [108], [275]
1987	deap	–	Carlsson [55],[56]
1988	relaxed heap	$\mathcal{O}(m + n \cdot \log n)$	Driscoll et al. [81]
1990	one-level radix heap	$\mathcal{O}(m + n \cdot \log C)$	Ahuja et al. [5]
1990	two-level radix heap	$\mathcal{O}(m + n \cdot \log C / \log \log C)$	Ahuja et al. [5]
1990	Fibonacci radix heap	$\mathcal{O}(m + n \cdot \sqrt{\log C})$	Ahuja et al. [5]
1993	diamond deque	–	Chang & Du [61]
1997	monotone heap	$\mathcal{O}(m + n \cdot (\log C)^{\frac{1}{3}+\epsilon})$	Cherkassky et al. [63,64]
2000	trinomial heap	$\mathcal{O}(m + n \cdot \log n)$	Takaoka [278]
2003	2-3 heap	$\mathcal{O}(m + n \cdot \log n)$	Takaoka [279]
2004	layered heap	$\mathcal{O}(m + n \cdot \log \log n)$	Elmasry [93]
2008	thin heap	$\mathcal{O}(m + n \cdot \log n)$	Kaplan & Tarjan [181]
2008	two-tier relaxed heap	$\mathcal{O}(m + n \cdot \log n)$	Elmasry et al. [95]
2009	quake heap	$\mathcal{O}(m + n \cdot \log n)$	Chan [60]
2009	rank-pairing heap	$\mathcal{O}(m + n \cdot \log n)$	Haeupler et al. [146,147]
2010	violation heap	$\mathcal{O}(m + n \cdot \log n)$	Elmasry [94]

Table 7.3: History of algorithms for priority queues

We give amortized running time as a sequence of m `decrease_key` and n `extract_min` operations ($n \leq m$). Symbol $-$ denotes that a priority queue is not suitable for our approach, because of the lacking of decrease operation. These implementations are given here only for didactic purposes.

Priority queues derive great importance from their use in solving a wide range of combinatorial problems, including shortest path, minimum spanning tree, weighted bipartite matching and minimum cost flow problem; see [4] or [67] for a discussion.

Recall that an edge is forbidden in any maximum matching iff it belongs to no alternating cycle (in the subgraph G_W) starting from a saturated vertex, or to no alternating path of even length (in the subgraphs G_U or G_O) starting from an exposed vertex. The search for all forbidden and mandatory edges can be done in linear time by applying the so-called alternating breadth- and alternating depth-first traversals (see Section 4.3).

However, a maximum edge-weighted matching problem is more complex than a maximum matching problem, because in the latter we need only to know whether there is an alternating cycle (or an alternating path) containing a given edge or not. For an edge-weighted problem we need to identify whether there is a particular alternating cycle (or a particular alternating path) with an optimal cost, containing a given edge, because there are weights on edges and the global bounds must be fulfilled. In the following we denote by C^e, P^e, M^e_{min} and M^e_{max}, respectively, the lightest alternating cycle, the lightest alternating path, the minimum-weight matching and the maximum-weight matching involving the edge e.

Now we can make the following connection between partition of edges and weighted matchings.

Theorem 7.2.5 *Given an edge-weighted graph G with a minimum edge-weighted matching M_{min} and a maximum edge-weighted matching M_{max}. The graph G has no admissible edge-weighted matching if $cost(M_{min}) > c_{max}$ or $cost(M_{max}) < c_{min}$. Analogously, an edge e does not belong to any admissible edge-weighted matching if $cost(M^e_{min}) > c_{max}$ or $cost(M^e_{max}) < c_{min}$.*

Proof We only need to restrict our attention to the minimum edge-weighted matching since the proof for the other case is similar. By definition, if $c_{max} < cost(M_{min})$ then G cannot have any admissible edge-weighted matching. Further, if there exists a matching containing edge e such that the cost of the matching lies between the given bounds, then this edge is admissible. On the other hand, the edge e is forbidden if $c_{max} < cost(M^e_{min})$. □

From this result we can define a simple, brute-force pruning algorithm for computing the partition of edges. First, we compute the minimum and the maximum-weight matching M_{min} and M_{max} in an edge-weighted graph G. If $cost(M_{min}) > c_{max}$ or $cost(M_{max}) < c_{min}$ then we know that no admissible edge-weighted matching exists. Otherwise, four possible scenarios are to be considered (cf. Figure 7.4). If $cost(M_{min}) < c_{min} \leq cost(M_{max}) \leq c_{max}$ we choose an arbitrary free edge $e = \{v_i, v_j\} \notin M_{max}$ and compute a maximum-weight matching M'_{max} in the graph $G' = G - v_i - v_j$. These operations require time $\mathcal{O}(m + n \cdot \log n)$ when starting in graph G' from the maximum-weight matching $M_{max} - \{v_i, mate(v_i)\} -$

$\{v_j, mate(v_j)\}$ with exposed vertices $mate(v_i)$ and $mate(v_j)$ [118]. Finally, we remove the edge e from the graph G and mark it as forbidden if a matching in G' with the same deficiency as in G does not exist or $cost(M^e_{max}) = cost(M'_{max}) + w(e) < c_{min}$.

In order to find all mandatory edges we choose an arbitrary edge $e = \{v_i, v_j\} \in M_{min} \cap M_{max}$ and recompute a maximum-weight matching M''_{max} in the graph $G'' = G - e$. We mark the edge e in the graph G as mandatory if the matching in G'' with the same deficiency as in G does not exist or $cost(M''_{max}) < c_{min}$. Obviously, this entire procedure runs in time $\mathcal{O}(m \cdot (m + n \cdot \log n))$. The other scenarios can be handled similarly.

At first glance, it does not seem easy to improve this algorithm because for each edge e the distances are computed in $G - e$. However, if for each vertex v we compute the lightest alternating path from v to every vertex in G, we will be able to know which incident edges are forbidden. Therefore, since there are n vertices, we can compute the partition of edges in time $\mathcal{O}(n \cdot (m + n \cdot \log n))$, which improves the previous complexity. But the partition can be determined more quickly.

Weighted alternating breadth-first search

The purpose of the weighted alternating breadth-first search is to detect the forbidden edges in the edge-weighted matching due to the fact that the edge can increase (or decrease) the cost of the weighted matching to the value that is not allowed. We use the breadth-first search as strategy, because it finds the shortest paths and cycles. We will recall some properties of breadth-first search which are important for our algorithm.

Similarly, as for the vertex-weighted graphs, we need to traverse the edges of the graph with respect to an initial edge-weighted matching. The weighted alternating breadth-first search starts from a given vertex r on the prescribed level l (0-even or 1-odd), corresponding to the type of the vertex (negative, neutral, or positive). In the first step, we traverse all (matched or free) edges incident with the vertex r. When we visit the vertices adjacent to the start vertex r then these vertices are placed into next level $l + 1$. In the second step, we scan all the new vertices we can reach from the (free or matched) edges, respectively. These new vertices, which are adjacent to vertices on the level $l + 1$ and assigned to no level, are located in level $l + 2$, and so on. Thus, a weighted alternating breadth-first search, similar to a weighted alternating depth-first search, simulates the traversing on a directed graph and constructs layers that alternately use matched and free edges. The weighted alternating breadth-first search terminates when every vertex has been visited and every edge has been traversed.

The weighted alternating breadth-first search uses a priority queue Q to store processed vertices. There are several possible ways for implementing the priority queue needed by the weighted alternating breadth-first search. We can implement the priority queue with a simple bucket data structure, which is an array of entries indexed by the priorities, or with a radix heap data structure. We can also use a Fibonacci heap in order to obtain a strongly polynomial algorithm $\mathcal{O}(m+n\cdot\log n)$ [4, Appendix A.4], [67, Chapter 20]. This immediately yields an overall $\mathcal{O}(m + n \cdot \log n)$ time complexity to classify the edges.

We determine back and cross edges in order to detect lightest alternating cycles and paths. The algorithm designates tree and cross edges belonging to any optimal alternating path and back edges belonging to any optimal alternating cycle as traversed. The remaining edges are not marked as traversed unless they have been already traversed in one of the previous phase of the algorithm. The algorithm terminates when all the edges have been traversed (become tree edges in any breadth-first forest). Additionally, if there exists an edge that does not occur in any of the lightest alternating paths or cycles generated by our algorithm then this edge is forbidden if the edge is free, and mandatory if the edge is matched.

Recall that our task consists of finding the edges that cannot be a part of an admissible weighted degree-matching. Therefore, detecting edges that cannot be a part of a lightest alternating cycle or a lightest alternating path would be sufficient.

For an edge-weighted matching problem we assume that G is with non-negative weights and at least one minimum/maximum-weight matching has been found. Thus, throughout our algorithm there is an initial weighted matching at hand, the cost of which is improved iteratively.

To find a partition of edges, the algorithm operates in phases. During each phase, the algorithm finds a partition of edges in one of the connected (canonical) components. Since we have to consider every vertex separately we split our algorithm into four phases.

The first phase of the algorithm processes the subgraph G_U and looks for even alternating paths and alternating cycles in it. The next phases of the algorithm process the subgraph G_W. In the second phase we maintain alternating paths that start from a positive vertex and a free edge. In the third phase we look for alternating paths that start from a negative vertex and a matched edge. Clearly, during the second phase the first edge of the alternating path changes its status from free to matched but in the third phase the first edge of the alternating path changes its status from matched to free. These two phases can only occur in (g, f)-matchings. In the last phase we compute alternating cycles that start from a neutral vertex and a matched edge. This situation appears only in f-matchings.

We have made the partition of edges and from the above algorithm we have obtained forbidden, allowed and mandatory edges. The canonical decomposition and thus the partition of edges is easy to perform.

Let us summarize some important properties of the weighted alternating breadth-first search:

- Every matched edge connects a vertex on the even level with a vertex on the odd level
- Every free edge connects a vertex on the odd level with a vertex on the even level.
- The lightest alternating cycles are detected by back edges.
- The lightest alternating paths can be determined by tree edges or cross edges.
- Every tree path from an arbitrary vertex to any of its descendants is the lightest path.
- A tree with non-tree edges contains only admissible alternating cycles and paths.

- Edges included in any admissible alternating cycle or any admissible alternating path are allowed.
- Matched edges included in no admissible alternating cycle nor admissible alternating path are mandatory.
- Free edges included neither in an admissible alternating cycle nor in an admissible alternating path are forbidden.

Observe that the weighted alternating breadth-first search is a modified version of Dijkstra's algorithm [67, Section 24.3]. However, for the alternating paths, the adapted routine significantly reduces the number of queue operations and the running time of the pruning algorithm, since fewer insert and decrease priority operations need to be performed. The improvement maintains an upper bound for the cost variable and performs only queue operations with values smaller than the bound.

The weighted alternating breadth-first search algorithm can also be viewed as a method which successively looks for lightest alternating cycles and paths with respect to a given weighted matching M and a predetermined start vertex or edge. In order to avoid multiple consideration of the same edge the algorithm traverses only these edges which have not yet been traversed in the actual breadth-first search. For this purpose we use the disjoint set union data structure of Gabow & Tarjan [121] (see also Table 4.2).

The pseudocode for the weighted alternating breadth-first search is shown in Algorithm 19.

Let us describe our procedure in more detail. Firstly, we initialize the data structure for each vertex. We use `parent` to maintain predecessor information in the tree. The array `cost` represents the shortest distances from the root r to the vertices in the tree. Each vertex is initially labeled with ∞. The procedure maintains a priority queue Q for the vertices of G. For each vertex v it computes a weight $cost[v]$ of a lightest path from the root r to v (which corresponds to the path with the minimum cost). Initially, it marks r as root, sets $cost[r]$ to 0 and inserts the item $\langle r, 0\rangle$ into the priority queue. In the main loop our procedure deletes from the priority queue a vertex v with the minimal value of cost. On the even level the procedure takes the matched edges; otherwise, it takes all outgoing free edges of v. The routine checks whether the distance function does not satisfy the triangle inequality with respect to edge $\{v, w\}$. Whenever such an edge is found we use it to reduce $cost[w]$ to $cost[v] + w'(v, w)$. Consider an edge $\{v, w\}$ and let $c = cost[v] + w'(v, w)$. In case of alternating paths our procedure checks whether c is smaller than the current maximal admissible cost. If so, three cases will be distinguished. If $parent[w] = NIL$ (or $cost[w]$ equals ∞) the routine inserts an item $\langle w, c\rangle$ into Q (this is the case when a white vertex indicates a tree edge). Otherwise, if $c < cost[w]$, the procedure decreases the priority of w in Q to c and updates $cost[w]$ to c (this is the case when a gray vertex indicates a tree edge). In the last case the edge $\{v, w\}$ is marked as a non-tree edge (this is the case for gray or black vertices).

Algorithm 19 Weighted Alternating Breadth-First Search of G

Require: Bipartite graph G with a minimum edge-weighted (g,f)-matching M_{min}

Ensure: Partition of edges into MANDATORY, ALLOWED and FORBIDDEN

```
procedure WeightedAlternatingBFS(r)
Q.insert(r,0)
set $color[r] \leftarrow GRAY$
while $Q \neq \emptyset$ do
  set $v \leftarrow$ Q.extract_min()
  for every edge $\{v,w\} \in M$ if $level[v]$ is even or $\{v,w\} \notin M$ if $level[v]$ is odd do
    set $c \leftarrow cost[v] + w'(v,w)$
    if $NEUTRAL \in sign[r]$ or $c \leq w_{max}$ then
      if $c < cost[w]$ then {tree edge}
        set $cost[w] \leftarrow c$
        set $parent[w] \leftarrow v$
        set $level[w] \leftarrow level[v] + 1$
        if $color[w] = WHITE$ then {white vertex}
          Q.insert(w,c)
          set $color[w] \leftarrow GRAY$
        else {gray vertex}
          Q.decrease_key(w,c)
        end if
      end if
      set $color[v] \leftarrow BLACK$
      add edge $\{v, parent[v]\}$ to the tree $T$ and mark it as TRAVERSED
      if $pathDetected(r,v)$ and $cost[v] + costPath(r,v) \leq w_{max}$ then
        mark all edges on the path from $v$ to $r$ as ALLOWED
      end if
    end if
  end for
end while
perform DFS on the tree $T$ in order to classify the non-tree edges (see Theorem 3.2.1)
for each back edge $\{v,w\}$ do
  mark back edge $\{v,w\}$ as TRAVERSED
  if $cost[v] - cost[w] + w'(v,w) \leq w_{max}$ then {cost of the cycle}
    mark back edge $\{v,w\}$ as ALLOWED
    mark all edges on the path from $v$ to $w$ as ALLOWED
  end if
end for
for each cross edge $\{v,w\}$ do {cost of the path}
  if $pathDetected(r,w)$ and $cost[v] + w'(v,w) + costPath(r,w) \leq w_{max}$ then
    mark cross edge $\{v,w\}$ as TRAVERSED and ALLOWED
    mark all edges on the path from $v$ to $r$ as ALLOWED
  end if
end for
```

The decision, as to whether a vertex is a critical vertex (an endpoint of some alternating path), is made when a vertex is permanently labeled (is colored black). When a vertex v is identified as a critical vertex, there exists an admissible lightest alternating path with respect to the maximum/minimum edge-weighted matching from v to the root r. The reason is that we cannot guarantee that the computed alternating path for gray labeled vertices would be the shortest path, since the path for a gray colored vertex could be changed at any time. All legal neighbor information is maintained by the data structure `parent`. Note that the graph we construct is a digraph, with weights on vertices (by potential function π) and edges (by reduced weights w'), even though the input graph was only an edge-weighted undirected one.

Our algorithm is of a hybrid nature. It consists of a multiple breadth-first traversal, where we always explore from the vertex of highest degree with respect to untraversed edges. First, the algorithm determines the vertex incident with the maximal number of free edges with respect to an initial matching M. Next, the algorithm generates a tree with a weighted alternating breadth-first fashion (as described in the text above) and decreases the degree of the visited vertices by the number of traversed edges. The algorithm then considers a new vertex of highest degree to explore from, thus building a new tree. The edges examined in this way form a new forest, becoming part of the weighted spanning tree (tree edges) or forming an alternating cycle (back edges) or path (cross edges) of admissible cost. The algorithm terminates when all edges in G have been traversed.

The correctness of our algorithm follows immediately from the following results:

Theorem 7.2.6 *Lightest alternating cycles are discovered by back edges. Moreover, the cost of the cycle equals $cost[v] - cost[w] + w'(v, w)$, where $w'(v, w)$ is a reduced weight of the back edge $\{v, w\}$.*

Proof If $\{v, w\}$ is a back edge, then there must be a path in the breadth-first tree from v to w. This path together with edge $\{v, w\}$ forms a cycle. Since there are only cycles of even length in bipartite graphs (Theorem 2.5.1), and any breadth-first tree contains only shortest paths, this defines an even alternating cycle of admissible cost. Since there exists the lightest alternating path of weight $cost[w]$ from root r to vertex w, and the lightest alternating path of weight $cost[v]$ from r to v, then the lightest alternating path from v to w has the weight $cost[v] - cost[w]$ (since any subpath of the lightest path must also be the lightest path). Thus, the cost of the alternating cycle is equal to $cost[v] - cost[w] + w'(v, w)$. □

Theorem 7.2.7 *Lightest alternating paths can be determined by tree edges or cross edges. Moreover, the cost of the (edge-weighted) path equals $cost[v] + w'(v, w) + costPath(r, w)$, where $w'(v, w)$ is a reduced weight of the edge $\{v, w\}$ and $costPath(r, w)$ is a cost of the (vertex-weighted) alternating path from r to w with respect to the potential function π.*

Proof This follows from the fact that there exists an admissible alternating path from an appropriate vertex v of the breadth-first tree to the root r. If the endpoints satisfy the conditions needed for an alternating path then the lightest alternating path is detected.

According to Lemma 7.1.2 and Formula (7.1) the cost of any alternating path with endpoints x and y equals $\sum_{e\in P} w'(e) + costPath(x, y)$.

If $\{v, w\}$ is a tree edge, then the tree edges from the root r to the vertex v plus the edge $\{v, w\}$ form an alternating path of weight $cost[v] + w'(v, w) + costPath(r, w)$.

If $\{v, w\}$ is a cross edge, two distinct alternating paths from the root r to the vertex w can be identified. One of these follows tree edges directly to w and has weight $cost[w] + costPath(r, w)$. The other alternating path follows tree edges to v and then uses cross edge $\{v, w\}$, giving a cost of $cost[v] + w'(v, w) + costPath(r, w)$. □

The time complexity associated with the weighted breadth-first search is $\mathcal{O}(m+n\cdot\log n)$. There are $n-1$ tree edges and $m-n+1$ non-tree edges; hence the category of non-tree edges (back edge or cross edge) must be determined at most $m - n$ times (some back edges, as leading to the root of the tree, or some cross edges, as leading to gray vertices, can be found immediately). Observe that the number of back edges leading to the root r is $d(r) - d_M(r)$. Thus, the complexity of carrying out $m - n + 1 - d(r) + d_M(r)$ edges turns out to be harder to realize since more breadth-first traversals must be performed. Therefore, it is difficult to characterize the overall complexity of the algorithm but it is $\mathcal{O}(\max\{1, \tau(G)\}\cdot(m+n\cdot\log n))$ in the worst case (when more weighted forests must be examined), where $\tau(G)$ denotes the cycle cover number of G; the average case running time is much better (since we can find a few weighted alternating cycles in one search phase). The best case running time of the algorithm is $\mathcal{O}(m + n \cdot \log n)$ (when all the non-tree edges can be classified in linear time). Observe that when all matched edges have reduced weight 0 and there only exist even alternating cycles in the bipartite graph with bipartition (V_1, V_2), we can improve the best case running time to $\mathcal{O}(m + k \cdot \log k)$, where $k = \min\{|V_1|, |V_2|\}$ (this can only happen in the case of the ordinary weighted matchings). Using suitable data structures, the algorithm requires $\mathcal{O}(m + n)$ space.

We now give an example which demonstrates how our algorithm works. The corresponding bipartite graph is depicted in the below figure: on the left side we have a minimum edge-weighted matching, on the right side a maximum edge-weighted matching is shown. The numbers near the edges denote their weights, whereas the reduced weights are given in the parentheses. The potential of each vertex is also given in both cases. It is easy to check that the bold edges form a minimum edge-weighted matching, which has weight $cost(M_{min}) = 15$ and a maximum edge-weighted matching with a total weight $cost(M_{max}) = 32$. Since there are exposed vertices (in the subgraph G_U), we will grow trees with roots on the even level.

Let G be a graph with the minimum edge-weighted matching M_{min} shown on the left side of the figure. Starting from (saturated) vertex y_1, we construct the weighted alternating tree T_{min} displayed close to the graph G. Since the level is even only matched edge $\{y_1, x_2\}$ leaves the root. On the odd level we examine (three) free edges, which are ordered according to their reduced weights and use increasing order for the priority queue: $\{x_2, y_4\}$ with reduced weight 0, $\{x_2, y_2\}$ with reduced weight 3, and $\{x_2, y_3\}$ with reduced weight 9.

When the search is continued, edges $\{y_4, x_4\}$, $\{x_4, y_5\}$ and $\{y_5, x_3\}$ are chosen and inserted into tree T_{min}. Next, the back edge $\{x_3, y_4\}$ is examined and a weighted alternating

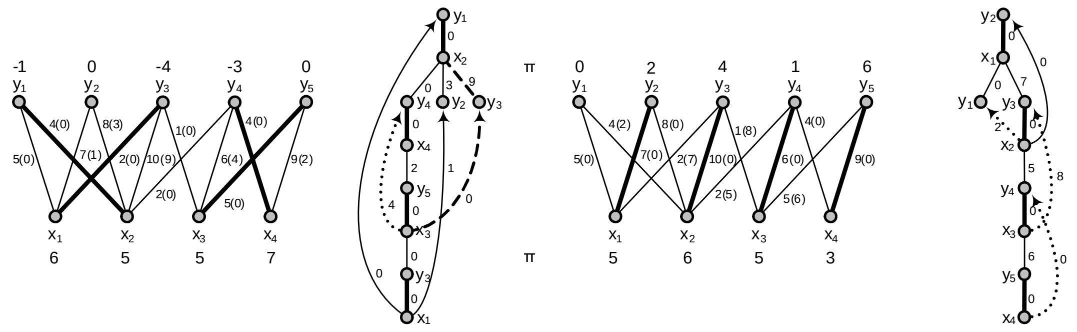


Figure 7.3: Weighted alternating breadth-first search

cycle $y_4x_4y_5x_3y_4$ of cost 6 is detected. The remaining two non-tree edges $\{x_1, y_1\}$ and $\{x_1, y_2\}$ can be easily determined. The first one, as leading to the root y_1, is the back edge that detects a new weighted alternating cycle $y_1x_2y_4x_4y_5x_3y_3x_1y_1$ of cost 2. The second one, as leading to the gray vertex y_2, is the cross edge that discovers a weighted alternating path $y_1x_2y_4x_4y_5x_3y_3x_1y_2$ of cost 4.

When free edge $\{x_2, y_2\}$ is examined we find an adjacent exposed vertex y_2, which yields a new alternating path $y_1x_2y_2$ in G of cost 4. We want to use this path to determine allowed edges. For this purpose, we trace this path backwards from its endpoint y_2 to the root y_1 of T_{min}.

Note that edge $\{x_2, y_3\}$ is ignored when cross edge $\{x_3, y_3\}$, as leading to the gray vertex y_3, is traversed and the vertex y_3 is examined.

Since all vertices have been visited but not all edges have been traversed, further steps must be performed in order to examine all of them. Thus, the algorithm will begin the search from the vertex y_3. We do not give its steps here, since it is a good exercise for the reader to continue the example.

Let G be a graph with an initial maximum edge-weighted matching M_{max} shown on the right side in the above figure. Then a call of a weighted alternating breadth-first search yields the result drawn on the right side of the figure, where choices are made according to the reduced weights of edges. We begin by constructing a weighted alternating tree T_{max} with root y_2, as described for the minimum-weight matching in the previous case. Whenever we encounter a back edge closing an alternating cycle, we mark all involved edges as allowed. The resulting weighted tree T_{max} constructed by a weighted alternating breadth-first search is shown on the right side of the figure. The only cross edge is $\{x_2, y_1\}$; the three back edges are $\{x_2, y_2\}$, $\{x_3, y_3\}$ and $\{x_4, y_4\}$.

In this example we have only demonstrated the weighted alternating breadth-first search starting from the (first in order) saturated vertices y_1 and y_2 in B_2. The reader is invited to analyze the search starting from the vertex y_3 or y_4 with the highest degree in B_2.

7.3 Application to Optimization Constraints

In this section we demonstrate how our algorithms can be used for checking the feasibility and establishing hyper-arc consistency for some of the most frequently used global constraints. Various examples are given that illustrate the effectiveness of the approach.

Since we deal with weighted matchings, we consider the undirected weighted bipartite graph G, called a *weighted value graph*. On one side we have a vertex for every variable and on the other side we have a vertex for each value that occurs in the domains of variables. There is an edge $\{x_i, d_j\}$ in G iff $d_j \in D_{x_i}$. We assign weights to vertices or edges and degree conditions to vertices. Additionally, graph G is associated with lower and upper bounds, c_{min} and c_{max}, on the cost. Initially $c_{min} = \min(D_{cost})$ and $c_{max} = \max(D_{cost})$.

Optimization constraints representable by weighted matchings are defined on a vertex- or an edge-weighted graph G and a cost variable COST. They state that G admits an appropriate weighted matching with cost at least $\min(D_{cost})$ and at most $\max(D_{cost})$. More formally,

$$\underset{e}{\forall}\ \underset{M}{\exists}(e \in M \Rightarrow \min(D_{cost}) \leq \mathit{cost}(M) \leq \max(D_{cost})) \ .$$

Note that by the above definition we cannot restrict all values of D_{cost} to belong to a solution. This would, however, be the case if we had defined the consequent $\mathit{cost}(M^e) \in D_{cost}$. The reason for omitting this additional restriction on the cost variable COST is that it makes the task of establishing hyper-arc consistency intractable. This follows from an easy transformation from SUBSET SUM [182],[131, Problem SP13]. Therefore, we assume that the domain of the COST variable consists of one single interval of consecutive values. Note that this is actually the case for many real-world instances. Hence, with respect to the COST variable we will establish bounds consistency. Thus, to enforce bounds consistency on the cost variable, we increase a lower bound and decrease an upper bound of D_{cost} to the cost of the minimum and maximum-weight matching in the weighted value graph.

We now develop a propagation algorithm for the optimization constraints, so that the solution is representable by an appropriate weighted matching. Our routine works in the following manner. First, we compute the minimum and the maximum cost that can be achieved. These values can be used to detect inconsistency and to narrow the bounds of the COST variable. If $\mathit{cost}(M_{max}) < \min(D_{cost})$ or $\mathit{cost}(M_{min}) > \max(D_{cost})$ then the constraint is not consistent. Otherwise, if $\{\mathit{cost}(M_{min}), \mathit{cost}(M_{max})\} \cap D_{cost} \neq \emptyset$, the constraint has a solution. On the other hand, if $\mathit{cost}(M_{min}) < \min(D_{cost})$ and $\max(D_{cost}) < \mathit{cost}(M_{max})$, the constraint may not be consistent (see Figure 7.4).

Next, we filter the inconsistent edges and corresponding domain values in G. We determine the partition of edges with respect to the corresponding weighted matching and remove the forbidden edges from the graph. If an edge in the matching is deleted we recompute this matching and repeat these steps until no edge deletion occurs any more.

We compute for every variable x_i and every value $d_j \in D_{x_i}$ the minimum $M^{x_i d_j}_{min}$ and the maximum $M^{x_i d_j}_{max}$ weighted matching, which can be achieved, if x_i is fixed to d_j and all other domains remain the same. This can be used for pruning the domains of x_i: if

$cost(M_{min}^{x_i d_j}) > max(D_{cost})$ or $cost(M_{max}^{x_i d_j}) < min(D_{cost})$, we can remove the value d_j from the domain of the variable x_i.

Finally, we narrow the lower bound of the cost variable to $\max\{\min(D_{cost}), cost(M_{min})\}$ and the upper bound to $\min\{\max(D_{cost}), cost(M_{max})\}$. If any domain becomes empty then the constraint is inconsistent.

We can further use the particular structure of G. In the following we show how to solve the problem by means of decomposition theory. In order to establish the partition of edges algorithmically, we first compute an optimal (g, f)-matching in the bipartite graph G. This can be realized in $\mathcal{O}(\sqrt{g(V)} \cdot m)$ operations by an algorithm due to Gabow [115]. In this way we get the canonical decomposition $\langle A_1, B_1, C_1\rangle$ and $\langle A_2, B_2, C_2\rangle$. Using these sets of partition, we can construct three, possible empty, induced subgraphs of G: G_U, G_O and G_W. For optimization constraints representable by weighted matchings, in a similar way as for hard global constraints, it must hold that $G_O = \emptyset$.

Using the standard approach of the Dulmage-Mendelsohn Decomposition we can easily determine the partition of edges. We remove all mandatory edges from the graph G and adjust the allowed bounds on the cost accordingly. Additionally, we shrink the bounds of degree conditions, which is equivalent to enforcing bounds consistency on cardinality variable domains associated with the bipartite graph (see Section 4.3.4).

We may assume that the canonical decomposition of G has been applied, since otherwise we know that we can determine the canonical subgraphs of G in linear time (see Theorem 4.3.3). We can then try to solve the problem for each connected component separately. But for the vertex-weighted matching problem it suffices to consider only the connected components of the subgraph G_U since every alternating cycle in G_W does not change the cost of the vertex-weighted matching (unless $g(x) = 0 \wedge f(x) > 2$ for some vertices x).

In our algorithm we have considered the following six exclusive scenarios:

condition	*interpretation*	*description*
1) $cost(M_{max}) < c_{min}$	I I | |	constraint inconsistent
2) $cost(M_{min}) < c_{min} \leq cost(M_{max}) \leq c_{max}$	I | I |	pruning with respect to M_{max}
3) $cost(M_{min}) < c_{min} \leq c_{max} < cost(M_{max})$	I | | I	intractable in general
4) $c_{min} \leq cost(M_{min}) \leq cost(M_{max}) \leq c_{max}$	| I I |	pruning with respect to M
5) $c_{min} \leq cost(M_{min}) \leq c_{max} < cost(M_{max})$	| I | I	pruning with respect to M_{min}
6) $c_{max} < cost(M_{min})$	| | I I	constraint inconsistent

Figure 7.4: Six scenarios for the weighted matching problem with bounds on the cost

The general procedure is presented in Algorithm 20. For all tractable scenarios, we achieve hyper-arc consistency for the assignment variables $x_1, \ldots, x_n$. Additionally, for the cost variable COST, we achieve in scenario 4 – bounds consistency, in scenario 2 – upper bound consistency, and in scenario 5 – lower bound consistency. Therefore, in the following we will write under these assumptions that we achieve hyper-arc consistency for the assignment variables and bounds consistency for the cost variable.

Theorem 7.3.1 *An optimization constraint whose solution is representable by a weighted matching is consistent iff there exists a weighted matching with the cost lying between the lower and the upper bound of the cost variable* COST.

Proof Let G be the weighted value graph associated with the global constraint. Assume that there exists a weighted matching of G with the cost within a given interval. From this weighted matching we can build a solution of our constraint. On the other hand, if the constraint is consistent then there exists a feasible weighted matching which corresponds to the solution of the constraint. □

It is now time to demonstrate our idea on concrete examples. For every global constraint we first give its formal definition, then we derive a transformation to the weighted matching problem, and finally we discuss some related global constraints. All definitions are taken from the Global Constraint Catalog [27] (in some cases we changed the definition in order to adapt it for our purposes).

For some examples we present our idea with figures. On the left side of the figure the domains are given. On the right side of the figure the domains after pruning are shown. In the middle of the figure the graph associated with the constraint and the decomposition of edges is depicted. The edges of a matching M are shown in bold. The potential π of each vertex is shown near the vertex in the auxiliary graph. Edge weights w and reduced weights w' are indicated next to each edge in the auxiliary graph as $w(w')$.

WEIGHTED_PARTIAL_ALLDIFF This constraint is abbreviated as WPA. It has the form WEIGHTED_PARTIAL_ALLDIFF(VARIABLES, UNDEF, VALUES, COST). From an interpretation point of view this constraint is a generalization of a constraint called ALLDIFFERENT_EXCEPT_0: all variables must have pairwise distinct values except those which are assigned value UNDEF. Additionally, with every value that occurs in the domains we associate the weight which is defined by the VALUES collection. It is assumed that the weight of value UNDEF is always equal to 0. The constraint states that the sum of the weights of the values that are assigned to the variables must be equal to variable COST. More formally,

$$\begin{aligned}
&\text{WEIGHTED\_PARTIAL\_ALLDIFF}(\langle x_1, \ldots, x_n \rangle, undef, weights, cost) = \\
&\qquad \{(d_1, \ldots, d_n) \in D_{x_1} \times \cdots \times D_{x_n} \mid \underset{\substack{i,j\\ i<j}}{\forall} (d_i \neq d_j \vee d_i = d_j = undef) \wedge \\
&\qquad\qquad min(D_{cost}) \leq \sum_{i=1}^{n} weight(d_i) \leq max(D_{cost})\}.
\end{aligned}$$

Algorithm 20 Cost-based propagation routine for optimization constraints

```
Require: Global constraint representable by a weighted bipartite graph $G$ with bounds
Ensure: Hyper-arc consistency, constraint inconsistent, or unknown
  Compute the maximum matching $M$
  Compute the Dulmage-Mendelsohn Canonical Decomposition (see Algorithm 3)
  if $G_O \neq \emptyset$ then
    return FALSE {problem over-constrained}
  end if
  Determine the partition of edges (see Algorithm 4 and 5)
  Identify mandatory edges and use them to decrease the bounds accordingly
  Shrink the bounds of the degree conditions $g$ and $f$ (see Theorem 4.3.14)
  Remove forbidden and mandatory edges from the graph
  Compute the minimum-weight matching $M_{min}$
  if $c_{max} < cost(M_{min})$ then {scenario 6}
    return FALSE {constraint inconsistent}
  end if
  Compute the maximum-weight matching $M_{max}$
  if $c_{min} > cost(M_{max})$ then {scenario 1}
    return FALSE {constraint inconsistent}
  end if
  if $c_{min} \leq cost(M_{min})$ and $c_{max} \geq cost(M_{max})$ then {scenario 4}
    set $c_{min} \leftarrow cost(M_{min})$
    set $c_{max} \leftarrow cost(M_{max})$
    return TRUE
  end if
  repeat
    if $c_{max} \neq \infty$ then {scenarios 3 and 5}
      Determine the partition of edges with respect to $M_{min}$ (see Algorithm 18 and 19)
      Remove forbidden edges from the graph
      if $c_{min} \neq -\infty$ then
        Recompute the maximum-weight matching $M_{max}$
        set $c_{max} \leftarrow \min(c_{max}, cost(M_{max}))$
      end if
    end if
    if $c_{min} \neq -\infty$ then {scenarios 2 and 3}
      Determine the partition of edges with respect to $M_{max}$ (see Algorithm 18 and 19)
      Remove forbidden edges from the graph
      if $c_{max} \neq \infty$ then
        Recompute the minimum-weight matching $M_{min}$
        set $c_{min} \leftarrow \max(c_{min}, cost(M_{min}))$
      end if
    end if
  until no more forbidden edges detected
  if $c_{min} > cost(M_{min})$ and $c_{max} < cost(M_{max})$ then {scenario 3}
    return MAYBE {constraint may be not consistent}
  end if
  return TRUE
```

The following two models can be used to establish hyper-arc consistency for this constraint. We build a vertex-weighted value graph, remove from it the vertex representing value UNDEF and define the degree conditions as follows. For each vertex representing a variable we set $g(x) = f(x) = 1$ if UNDEF $\notin D_x$ or $g(x) = 0$ and $f(x) = 1$, otherwise. For each vertex representing a value we set $g(x) = 0$ and $f(x) = 1$. Further, we assign a weight to every vertex. Each vertex representing a variable gets weight 0, and the weights of vertices representing values are given by the VALUES collection. Then it is easy to prove that any vertex-weighted perfect (0,1)-matching corresponds to the solution of the constraint and establishing hyper-arc consistency is equivalent to remove all forbidden edges from the vertex-weighted graph defined as above.

The second model can be defined in a similar way. We create the vertex-weighted value graph with the following degree conditions. For each vertex representing a variable we set $g(x) = f(x) = 1$. For each vertex representing the defined value we set $g(x) = f(x) = 1$, whereas for the vertex representing value UNDEF we set $g(x) = 0$ and $f(x) = d(x)$. Analogously, as in the first model, we assign a weight to every vertex. Each vertex representing a variable gets weight 0, the weights of vertices representing defined values are given by the VALUES collection; the weight of value UNDEF is defined to be 0. Then it is straightforward to check that there is a one-to-one correspondence between any admissible vertex-weighted complete (g, f)-matching and the solution of the constraint. Therefore, establishing hyper-arc consistency for this constraint can be done easily after computing the partition of edges.

We believe that our filtering algorithm is simpler and more intuitive than that presented in [288, Chapter 4]. Our approach is indeed similar and more general, but our effort is comparable to that of Sven Thiel. In contrast to his method, our propagation routine performs only one depth-first search and we do not have to compute regrets explicitly, but we are able to determine the partition of edges with the help of the weighted alternating depth-first search. The partition of edges can be determined by starting from matched edges and weighted vertices sorted in increasing order of their weights.

NVALUE The NUMBER_OF_DISTINCT_VALUES constraint (abbreviated as NVALUE) was introduced in [240] under the name of CARDINALITY_ON_ATTRIBUTE_VALUES. The constraint has the form NVALUE(NVAL,VARIABLES), where NVAL is a domain variable and VARIABLES is a collection of variables. The constraint states that the number of distinct values taken by the variables of the collection VARIABLES lies within the domain of the variable NVAL. More formally, NVALUE(N,$\langle x_1, \ldots, x_n \rangle$) holds iff

$$\left| \bigcup_{1 \le i \le n} \{x_i\} \right| = N.$$

Achieving hyper-arc consistency for the NVALUE constraint is, in general, NP-hard [42], but incomplete polynomial-time filtering algorithms have been proposed [26],[41].

The purpose of this example is to show how to solve this constraint by means of the weighted matching. We build a vertex-weighted value graph with the following weights and degree conditions. For each vertex representing a variable we set $w(x) = 0$, $g(x) = f(x) = 1$

and for each vertex representing a value we set $w(x) = 1$, $g(x) = 0$ and $f(x) = d(x)$, where $d(x)$ denotes the degree of the vertex x. Then it is easy to prove that any admissible vertex-weighted perfect (g, f)-matching represents the solution of the constraint. When all the domains of the variables are intervals, we know that the constraint can be made hyper-arc consistent. Otherwise, establishing hyper-arc consistency is NP-hard (see Theorem 7.2.1).

In Figure 7.5 we present an example from [26] and we show how our method works.

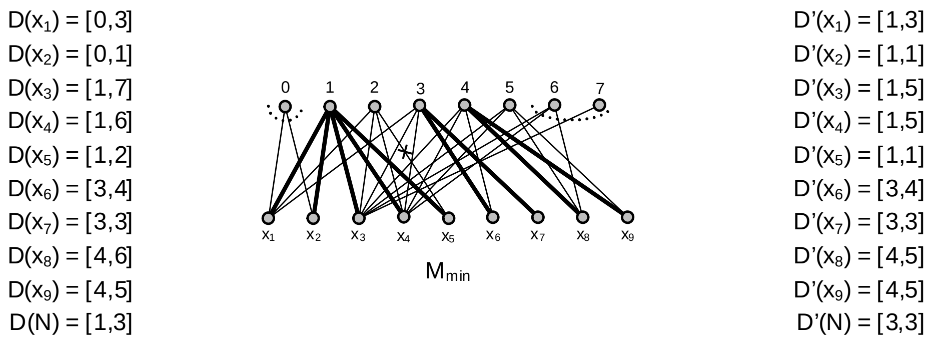


Figure 7.5: Pruning of the NVALUE constraint

The constraint generalizes the constraints ALLDIFFERENT (by setting $D_{nval} = \{n\}$), NOT_ALL_EQUAL (by $D_{nval} = \{2, \dots, n\}$) and ALL_EQUAL (by $D_{nval} = \{1\}$), but one should use the filtering algorithms specified for these three constraints, because they achieve a more efficient pruning.

SUM_OF_WEIGHTS_OF_DISTINCT_VALUES This global constraint (abbreviated as SWDV) is a weighted version of the NVALUE constraint. The constraint was introduced in [28], where the filtering algorithm has also been given. The constraint has the form SWDV(VARIABLES,VALUES,COST). All variables take the values in the VALUES collection. In addition, COST is the sum of the weights associated to the distinct values taken by the variables. The formal definition of this constraint looks as follows:

$$\begin{aligned}&\text{SUM\_OF\_WEIGHTS\_OF\_DISTINCT\_VALUES}(\langle x_1, \dots, x_n\rangle, weights, cost) = \\ &\qquad \{(d_1, \dots, d_n) \in D_{x_1} \times \cdots \times D_{x_n} \mid S = \bigcup_{1 \le i \le n} \{d_i\}, \\ &\qquad\qquad min(D_{cost}) \le \sum_{d_i \in S} weight(d_i) \le max(D_{cost})\}.\end{aligned}$$

In a similar way to the previous example, we build a vertex-weighted value bipartite graph with the following degree conditions. We set $g(x) = f(x) = 1$ for each vertex representing a variable and $g(x) = 0$ and $f(x) = d(x)$ for each vertex representing a value. Then it is easy to see that the solution of this constraint corresponds to the vertex-weighted matching problem. The minimum/maximum cost of a variable assignment is equal, respectively, to the cost of the minimum/maximum vertex-weighted perfect (g, f)-matching in the vertex-weighted value bipartite graph defined as above.

According to Theorem 7.2.1 the constraint is intractable in general. However, we know that when the vertex-weighted bipartite graph is convex (which corresponds to the situation that each variable has a domain consisting of one single interval of consecutive values) this leads to a polynomial algorithm establishing hyper-arc consistency.

Our filtering algorithm is basically equivalent to that presented in [28] but considerably more intuitive since it uses the language of matching theory. Sven Thiel states [288, page 109] that the "lower side" of SWDV is quite different from the "upper side" and seems to be more difficult to handle. However, as this example shows, there exists a certain connection between them and both sides can be handled with the same method: vertex-weighted matching.

COST_ALLDIFFERENT The COST_ALLDIFFERENT constraint is the conjunction of the well-known ALLDIFFERENT constraint and the SUM constraint. The constraint has the form COST_ALLDIFFERENT(VARIABLES,MATRIX,COST). Just as for the common ALLDIFFERENT constraint, all variables should take distinct values. In addition, the domain variable COST is equal to the sum of the weights associated to the fact that we assign value d_j to variable x_i. These weights are given by the MATRIX collection. More formally,

$$\begin{aligned}&\textsc{cost\_alldifferent}(\langle x_1,\ldots,x_n\rangle, weights, cost) = \\ &\qquad\{(d_1,\ldots,d_n)\in D_{x_1}\times\cdots\times D_{x_n} \mid \underset{\substack{i,j\\ i\neq j}}{\forall} (d_i\neq d_j)\wedge \\ &\qquad\qquad min(D_{cost}) \leq \sum_{i=1}^{n} weight(x_i,d_i) \leq max(D_{cost})\}.\end{aligned}$$

The addition of costs to an ALLDIFFERENT constraint has been studied by Yves Caseau and François Laburthe [58]. The authors do not propose any filtering algorithm but only show an interest in computing the consistency of such a global constraint. Our effort shows that the feasibility of this constraint can be checked by searching for an admissible edge-weighted complete matching in the corresponding edge-weighted value graph and hyper-arc consistency can be established in polynomial time.

The following model is used to establish hyper-arc consistency of the domain variables and bounds consistency of the cost variable. We create an edge-weighted bipartite value graph. Then it is easy to check that there is a one-to-one correspondence between the solution of this constraint and computing an admissible edge-weighted complete matching.

According to Theorem 7.2.5 the value d_j of the variable x_i is not consistent with the COST_ALLDIFFERENT constraint iff there exists no admissible edge-weighted matching which contains an edge $e = \{x_i, d_j\}$. So, if e is not contained in any admissible alternating cycle or any admissible alternating path then the value d_j of the variable x_i is not supported and can be deleted from the domain D_{x_i}.

Our constraint is strongly related with the constraint WEIGHTED_PARTIAL_ALLDIFF, which we previously discussed. It should not be difficult to observe that the vertex-weighted matching problem can be regarded as a particular case of the edge-weighted matching problem. We can reduce the former to the latter by turning the vertex-weight function into an edge weight-function. The weight of every edge would simply be the sum of the weights of its incident vertices $w(x_i, d_j) = w(x_i) + w(d_j)$ for every edge $\{x_i, d_j\}$ (in our case $w(x_i) = 0$ for all i). This simple transformation from the vertex-weighted matching problem to the edge-weighted matching problem shows that the same filtering algorithm can be used.

MINIMUM_WEIGHT_ALLDIFFERENT This constraint (abbreviated as MWA) is a minimization version of the ALLDIFFERENT constraint with costs. It was introduced in [58] (under the name MinWeightAlldiff) and in [105] (under the name IlcAlldiffCost). The filtering algorithm is described in [267]. With the help of COST_ALLDIFFERENT this constraint can be modeled as follows.

We first create an edge-weighted value graph associated with the constraint and set $c_{min} = -\infty$. In order to check the feasibility we compute the minimum edge-weighted perfect matching M_{min}. Using the Hungarian algorithm it can be realized in time $\mathcal{O}(n \cdot (m + n \cdot \log n))$. If $cost(M_{min}) > \max(D_{cost})$ we know that the constraint has no solution.

To achieve hyper-arc consistency we need to remove all edges $e \notin M_{min}$ that cannot be a part of any lightest alternating cycle C^e with $cost(M_{min}) + cost(C^e) \leq max(D_{cost})$. The constraint can be made hyper-arc consistent in the worst-case computational time of $\mathcal{O}(\tau(G) \cdot (m + n \cdot \log n))$.

Observe that a very important property of our algorithm is that we can run it without explicitly transforming the graph G into a directed corresponding network N^M, as described in [267]. This can be realized by means of the weighted alternating breadth-first search. In addition, it is not necessary to compute all-pairs shortest paths as proposed in the mentioned paper.

Meinolf Sellmann [267] posed an open problem whether his algorithm can be implemented to run incrementally faster. This example gave the affirmative answer to his question.

SOFT_INVERSE_VAR The SOFT_INVERSE_VAR constraint is the soft version of the INVERSE constraint with the variable-based violation measure. The constraint is denoted by SOFT_INVERSE_VAR(NODES,NODES,COST). With the algorithm we propose, we are able for the first time to achieve hyper-arc consistency for this constraint. First, recall the definition of the hard version of the constraint:

$$\textsc{inverse}(\langle x_1, \dots, x_n \rangle, \langle y_1, \dots, y_n \rangle) =$$
$$\{(d_1, \dots, d_n) \in D_{x_1} \times \cdots \times D_{x_n}, (d'_1, \dots, d'_n) \in D_{y_1} \times \cdots \times D_{y_n} \mid \underset{i \neq j}{\forall} (d_i = j \Leftrightarrow d'_j = i)\}.$$

We can make the soft constraint hyper-arc consistent in the following way. This constraint can be expressed by a complete edge-weighted bipartite graph, called a *weighted variable graph*, in which vertices correspond to the variables and the weight function is defined as follows. There is an edge between two vertices x and y with weight $w(x, y) = 0$ if $x \in D_y \wedge y \in D_x$. There is an edge with weight $w(x, y) = 1$ if only one of these two conditions holds (i.e. $x \in D_y \wedge y \notin D_x \vee x \notin D_y \wedge y \in D_x$). All the remaining edges become the weight $w(x, y) = 2$. It is now straightforward to see that there is a one-to-one correspondence between the minimum edge-weighted perfect matching in the corresponding graph and the solution of the constraint.

COST_GLOBAL_CARDINALITY This constraint was introduced in [258] under the name GLOBAL_CARDINALITY_CONSTRAINT WITH COSTS, where also the algorithm achieving hyper-arc consistency, based on the minimum cost flow algorithm, has been given. The constraint has the form COST_GCC(VARIABLES,VALUES,MATRIX,COST). It states that the number of variables in the VARIABLES collection which assume value from the VALUES collection is within the given interval. Additionally, the assigned COST is equal to the sum of the weights associated to the edges. These weights are given by the MATRIX collection. More precisely, for a given set of variables, this constraint is the conjunction of the SUM constraint and the GCC constraint.

A constraint is specified in terms of a set of VARIABLES which take their values in a subset of VALUES. It constrains the number of times a value $v_i \in$ VALUES is assigned variables in VARIABLES to be within a given interval $[l_{v_i}, u_{v_i}]$. The minimal and the maximal number of occurrences of each value can be different from the others. This motivates the following definition:

$$\begin{aligned}\text{COST\_GLOBAL\_CARDINALITY}(\langle x_1,\ldots,x_n\rangle, \langle l_{v_1},\ldots,l_{v_k}\rangle, \langle u_{v_1},\ldots,u_{v_k}\rangle, \mathit{weights}, \mathit{cost}) = \\ \{(d_1,\ldots,d_n) \in D_{x_1}\times\cdots\times D_{x_n} \mid \\ \forall_j \left(l_{v_j} \le \left| \bigcup_{\substack{1\le i\le n \\ v_j = d_i}} \{i\} \right| \le u_{v_j} \right) \wedge \\ \mathit{min}(D_{\mathit{cost}}) \le \sum_{i=1}^{n} \mathit{weight}(x_i, d_i) \le \mathit{max}(D_{\mathit{cost}})\}.\end{aligned}$$

The following model can be used to establish hyper-arc consistency for this constraint. We construct an edge-weighted bipartite value graph with degree conditions g and f. The vertices on one side correspond to the variables and vertices on the other side correspond to the values. For each variable x_i we define $g(x_i) = f(x_i) = 1$ and for each value v_j we set $g(v_j) = l_{v_j}$ and $f(v_j) = u_{v_j}$. Then it is easy to show that there exists a one-to-one correspondence between the admissible edge-weighted perfect (g, f)-matching and the solution of the constraint (Figure 7.6).

In order to check the feasibility we first compute a minimum edge-weighted (g, f)-matching M_{min} in an edge-weighted value graph G defined as above. To achieve hyper-arc consistency we need to remove all edges $e \notin M_{min}$ that cannot be a part of any perfect (g, f)-matching with cost less than $\max(D_{cost})$. This simply corresponds to edges e that are not contained in any lightest alternating cycle C^e with $\mathit{cost}(M_{min}) + \mathit{cost}(C^e) \le \mathit{max}(D_{cost})$ or in any lightest alternating path P^e of even length with $\mathit{cost}(M_{min}) + \mathit{cost}(P^e) \le \mathit{max}(D_{cost})$.

In order to compute the partition of edges we begin the weighted alternating breadth-first search from positive (or negative) vertices. If not all edges have been traversed we continue the search from neutral vertices.

Our version of the constraint allows us to solve the minimization, maximization or optimization problem, although in the original paper due to Jean-Charles Régin [258] only a minimization version of the constraint was considered.

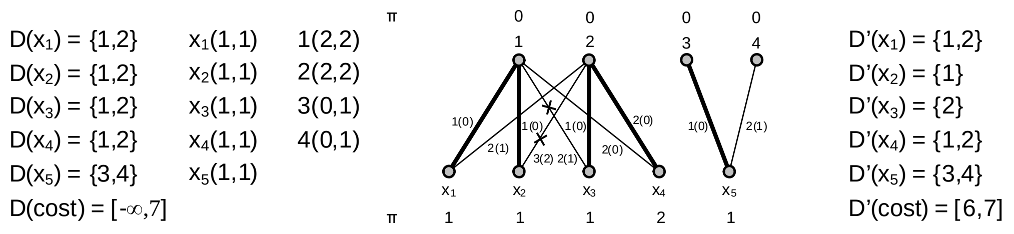


Figure 7.6: Pruning of the COST_GCC constraint

COST_SYMMETRIC_CARDINALITY This global constraint extends the familiar SYMMETRIC_CARDINALITY constraint with a weight function associated with each individual assignment. It is a conjunction of the latter and the SUM constraint. The constraint was introduced in [190], where also the filtering algorithm based on flow theory has been proposed. The constraint is defined as follows:

$$\begin{aligned}
&\text{COST\_SYMMETRIC\_CARDINALITY}(\langle x_1,\dots,x_n\rangle, bounds, weights, cost) = \\
&\qquad \{((d_1^1,\dots,d_1^{k_1}),\dots,(d_n^1,\dots,d_n^{k_n})) \in D_{x_1}^{k_1}\times\cdots\times D_{x_n}^{k_n} \mid \\
&\qquad\qquad \forall_i (l_{x_i} \le k_i \le u_{x_i}) \wedge \\
&\qquad\qquad \forall_j \left(l_{v_j} \le \left| \bigcup_{\substack{1\le i\le n\\ v_j = d_i}} \{i\} \right| \le u_{v_j} \right) \wedge \\
&\qquad\qquad min(D_{cost}) \le \sum_{i=1}^{n}\sum_{j=1}^{k_i} weight(x_i, d_i^j) \le max(D_{cost})\}.
\end{aligned}$$

In a similar way to the previous example, this constraint can be modeled with the help of the edge-weighted bipartite graph. We define the degree conditions g and f as follows. For each variable x_i we set $g(x_i) = l_{x_i}$ and $f(x_i) = u_{x_i}$ and for each value v_j we set $g(v_j) = l_{v_j}$ and $f(v_j) = u_{v_j}$. Then it is easy to show that every admissible edge-weighted perfect (g, f)-matching represents the solution of the constraint (Figure 7.7). Therefore, establishing hyper-arc consistency for this constraint is equivalent to the search for forbidden edges in the edge-weighted value graph defined as above. The consistency of the constraint can be checked in $\mathcal{O}(f(V)\cdot(m+n\cdot\log n))$ and hyper-arc consistency can be computed in the complexity of $\mathcal{O}(\tau(G)\cdot(m+n\cdot\log n))$ providing that an optimal edge-weighted perfect (g, f)-matching has been found.

The constraint is consistent if there exists an edge-weighted perfect (g, f)-matching which satisfies lower and upper bounds of the cost variable. Recall that perfect (g, f)-matchings can have the different number of edges. This property gives a way of computing the consistency of the constraint by determining a minimum/maximum perfect (g, f)-matching M with a maximum number of edges and checking if $min(D_{cost}) \le cost(M) \le max(D_{cost})$. If this inequality does not hold we would need to verify whether there exists another perfect (g, f)-matching M' such that $cost(M') < cost(M)$ satisfying bound conditions of the cost variable.

For a COST_GLOBAL_CARDINALITY constraint there is an equivalent constraint SYMMETRIC_CARDINALITY with costs where the cardinality of each variable is restricted to $[1,1]$. Similarly, for a COST_ALLDIFFERENT constraint there is a COST_SYMMETRIC_CARDINALITY constraint where the cardinalities of each variable are restricted to $[1,1]$ and the cardinalities of each value are restricted to be within the interval $[0,1]$.

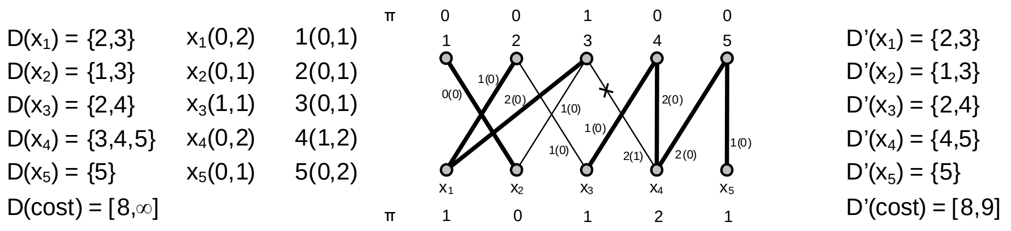


Figure 7.7: Pruning of the COST_SYMMETRIC_GCC constraint

COST_SYMMETRIC_ALLDIFFERENT In this example we discuss a constraint SYMMETRIC_ALLDIFFERENT associated with costs. We will present a filtering algorithm for this constraint, which is based on standard algorithms of combinatorial optimization for the computation of weighted matchings in general graphs. To the best of our knowledge the COST_SYMMETRIC_ALLDIFFERENT constraint itself has not been treated before.

The constraint has the form COST_SYMMETRIC_ALLDIFFERENT(NODES,MATRIX,COST). With every value d_j that occurs in the domain of some variable x_i we associate a non-negative weight $w(x_i, d_j)$ which is defined by MATRIX. Clearly, $w(x_i, d_j) = w(x_j, d_i)$. The constraint states that COST must be equal to the sum of the weights of the values that are assigned to the variables. The parameter COST is a cost variable. Its domain must be an interval.

The COST_SYMMETRIC_ALLDIFFERENT constraint is a generalization of the SYMMETRIC_ALLDIFFERENT constraint in which a cost is associated with every value of each variable. Then, each solution of the underlying SYMMETRIC_ALLDIFFERENT constraint is associated with a global cost equal to the sum of the costs associated with the assigned values of the solution. The SYMMETRIC_ALLDIFFERENT constraint with costs restricts the global cost to be within a lower and upper bound of the COST variable.

The constraint is defined more formally as follows:

$$\begin{aligned}&\text{COST\_SYMMETRIC\_ALLDIFFERENT}(\langle x_1,\dots,x_n\rangle, \mathit{weights}, \mathit{cost}) =\\ &\qquad \{(d_1,\dots,d_n) \in D_{x_1} \times \cdots \times D_{x_n} \mid \underset{i,j}{\forall}(d_i \neq i \wedge d_i = j \Leftrightarrow d_j = i) \wedge \\ &\qquad\qquad min(D_{cost}) \leq \sum_{\substack{i=1\\ i<d_i}}^{n} \mathit{weight}(x_i, x_{d_i}) \leq max(D_{cost})\}.\end{aligned}$$

In Section 5.4 (Example 1) about the common SYMMETRIC_ALLDIFFERENT constraint we have seen that solutions of this global constraint correspond in a natural way to perfect matchings in a so-called *value graph*. Since our new problem deals with weighted assignments, we consider the *weighted value graph* G. We construct graph $G = (V, E)$ with the vertex set $V = \{x_1, \dots, x_n\}$ and edge set $E = \{\{x_i, x_j\} \mid i \in D_{x_j} \wedge j \in D_{x_i}\}$. We extend the value graph G by applying a weight function to its edges. The weight of edge $\{x_i, x_j\}$ is $w(x_i, x_j)$ for all $1 \leq i < j \leq n$ and $j \in D_{x_i}$.

Let COST_SYMMETRIC_ALLDIFFERENT(NODES,MATRIX,COST) be the constraint under consideration in this example and let G be its edge-weighted value graph. It is easy to see that there is a tight correlation between this constraint and the edge-weighted perfect matching problem. Therefore, there is also a one-to-one correspondence between the optimal cost of variable assignments and admissible edge-weighted perfect matchings in G.

We have transformed our constraint into an edge-weighted matching problem. We want now to compute an admissible edge-weighted perfect matching in G efficiently. For the latter problem, a series of well-known algorithms has been developed. A maximum edge-weighted perfect matching can be found from scratch in time $\mathcal{O}(n \cdot (m + n \cdot \log n))$ using the algorithm by Gabow [118] or in time $\mathcal{O}(m \cdot \log(nW) \cdot \sqrt{n \cdot \alpha(m, n) \cdot \log n})$ by applying the sophisticated algorithm by Gabow & Tarjan [124] (here, W is the largest magnitude of an edge weight and α is an inverse of Ackermann's function, which is very slow growing).

The following table summarizes known polynomial-time algorithms for solving the problem; here, $d = \max\{m/n, 2\}$ is the density of the graph (cf. [264, Section 26.3a]):

Year	Author(s)	Complexity	Strategy
1965	Edmonds [89]	$\mathcal{O}(n^4)$	primal-dual
1973	Gabow [113]	$\mathcal{O}(n^3)$	labeling
1976	Lawler [201]	$\mathcal{O}(n^3)$	labeling
1982	Galil et al. [126]	$\mathcal{O}(n \cdot m \cdot \log n)$	priority queue
1984	Gabow et al. [120]	$\mathcal{O}(n \cdot (m \cdot \log_d^{(3)} n + n \cdot \log n))$	contraction, packets
1985	Gabow [116]	$\mathcal{O}(n^{3/4} \cdot m \cdot \log W)$	cost scaling, shells
1990	Gabow [118]	$\mathcal{O}(n \cdot (m + n \cdot \log n))$	Fibonacci heaps
1991	Gabow & Tarjan [124]	$\mathcal{O}(\sqrt{n \cdot \alpha(m, n) \cdot \log n} \cdot m \cdot \log(nW))$	ϵ-optimality
1999	Blum [47]	$\mathcal{O}(n \cdot m \cdot \log n)$	reachability
2012	Huang & Kavitha [169]	$\mathcal{O}(\sqrt{n} \cdot m \cdot W \cdot \log_n \frac{n^2}{m})$	augmenting

Table 7.4: History of algorithms for the edge-weighted non-bipartite matching problem

However, the matching can be updated more quickly. For instance, if arbitrary changes are made to the edges incident to one vertex, a new maximum edge-weighted perfect matching can be constructed by finding one weighted augmenting path [20],[116]. This can be done in time $\mathcal{O}(m + n \cdot \log n)$ [118].

The filtering algorithm for the SYMMETRIC_ALLDIFFERENT constraint with costs is an extension of the filtering algorithm of the SYMMETRIC_ALLDIFFERENT without costs and is based on finding an admissible weighted perfect matching of G.

In the following, we describe an algorithm that achieves hyper-arc consistency in the same worst-case running time as is needed to compute a minimum edge-weighted perfect matching when using the above mentioned methods or algorithms.

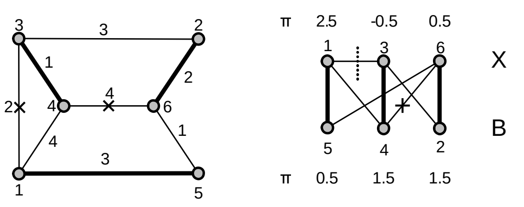
To achieve hyper-arc consistency of the COST_SYMMETRIC_ALLDIFFERENT constraint, we need to remove all values from variable domains that cannot be a part of any feasible assignment of values to variables with associated cost lying within a given interval. That is, in the graph interpretation of the problem, we need to compute and remove the set of edges that cannot be a part of any perfect matching with cost both greater than or equal to minimal and less than or equal to maximal value of the cost variable COST.

From the above results we can define the following filtering algorithm for computing the partition of edges. First, we compute the minimum and the maximum-weight perfect matching M_{min} and M_{max} in the weighted value graph G. If the perfect matching does not exist, $w(M_{min}) > \max(D_{\text{COST}})$ or $w(M_{max}) < \min(D_{\text{COST}})$ then we know that the constraint has no solution. Otherwise, we detect the forbidden edges in G and corresponding domain values in the constraint. Using the standard approach of the Gallai-Edmonds Decomposition presented in Chapter 5, we can easily determine the set of admissible and forbidden edges.

In order to achieve this, we must find an extreme set. This can be done by means of the algorithm given in Chapter 5. We choose an arbitrary (not yet visited) vertex x and compute the Gallai-Edmonds Decomposition $\langle A, B, C, D\rangle$ in $G-x$. We know that the set $X = A \cup \{x\}$ is extreme in G. Then, for every free edge e connecting X to D_i we recompute the minimum edge-weighted perfect matching in $G[D_i] \cup \{e\}$ and set $w(e)$ to an obtained weight decreased by the cost of the matching in the blossom $G[D_i]$. It takes only one iteration to complete the optimal matching due to the incrementality of the algorithm. Then, in order to find the forbidden edges, we perform the weighted alternating breadth-first search on the bipartite multigraph G_0 with bipartition $(X, B \cup base(D_i))$, where $base(D_i)$ denotes the base of the blossom D_i. Next, we mark all vertices in $X \cup B$ as visited and repeat the whole routine until all vertices are marked as visited (Figure 7.8).

Finally, to enforce bounds consistency on the cost variable COST, we narrow the lower and the upper bound of D_{COST} to the cost of the minimum and the maximum-weight perfect matching in the weighted value graph. If any domain becomes empty then the constraint is not consistent.

$D(x_1) = \{3,4,5\}$
$D(x_2) = \{3,6\}$
$D(x_3) = \{1,2,4\}$
$D(x_4) = \{1,3,6\}$
$D(x_5) = \{1,2,6\}$
$D(x_6) = \{2,4,5\}$
$D(cost) = [2,9]$



$D'(x_1) = \{4,5\}$
$D'(x_2) = \{3,6\}$
$D'(x_3) = \{2,4\}$
$D'(x_4) = \{1,3\}$
$D'(x_5) = \{1,6\}$
$D'(x_6) = \{2,5\}$
$D'(cost) = [6,8]$

Figure 7.8: Pruning of the COST_SYMMETRIC_ALLDIFFERENT constraint

COST_TOUR The global constraint COST_TOUR is an optimization version of the TOUR constraint. This constraint can be used to model the *Traveling Salesperson Problem* (abbreviated as TSP). Recall that the TSP is the problem of finding a Hamiltonian cycle visiting a set of n cities only once and minimizing the travel distance. The constraint has the form COST_TOUR(NODES,MATRIX,COST) where NODES is a collection of n variables whose domains are subsets of $\{1,\ldots,n\}$, MATRIX is an array containing the costs (distances or capacities) of connecting each pair of vertices and COST is an objective function. The constraint is satisfied when NODES form one cycle involving all variables with an optimal cost. More formally,

$$\begin{aligned}
&\textsc{cost\_tour}(\langle x_1,\ldots,x_n\rangle, \mathit{weights}, \mathit{cost}) = \\
&\quad \{((d_1^1,d_1^2),\ldots,(d_n^1,d_n^2)) \in D_{x_1}\times\cdots\times D_{x_n} \mid \underset{i,j}{\forall}(j\in\{d_i^1,d_i^2\} \Leftrightarrow i\in\{d_j^1,d_j^2\})\wedge \\
&\qquad \underset{S\subset\{1,\ldots,n\}}{\forall}\left(\bigcup_{i\in S}\{d_i^1,d_i^2\}\neq S\right)\wedge \\
&\qquad\quad \mathit{min}(D_{cost}) \le \sum_{\substack{i=1\\ i<d_i}}^{n} \mathit{weight}(x_i,x_{d_i}) \le \mathit{max}(D_{cost})\}.
\end{aligned}$$

The second condition in the above definition is the so-called *subtour elimination constraint*. It excludes cycles between arbitrary proper subset S of vertices of G by requiring that at least one edge connects a vertex outside S to one within S. This is the so-called NOCYCLE constraint [57],[247].

Clearly, the TSP is a special (i.e. connected) minimum edge-weighted perfect 2-matching. Since it is a difficult NP-complete problem [131, Problem ND22], no complete solution can be expected. But our results discussed above shed some light on this connection. They also yield a nice partial pruning method for this constraint.

A necessary condition for the satisfiability of this constraint is to have no more than one single connected component. The second necessary condition is that the graph must be 2-connected. Other necessary conditions are given in Chapter 5 (Section 5.4).

In this example we demonstrate a partial filtering method not presented before. Our idea is based on identifying perfect matchings in an incremental graph corresponding to the underlying graph associated with the constraint. This follows from the fact that every Hamiltonian cycle is a connected perfect 2-matching. Our routine looks for a perfect matching in the incremental graph and any edge not belonging to it will be removed from the associated graph. This will result in deleting some edges from the associated graph, which cannot be a part of any Hamiltonian cycle.

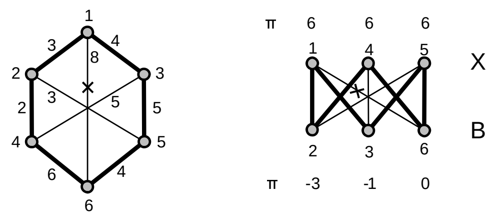
Let us first try to count how many vertices and edges will have the incremental graph. Let $G=(V,E)$ be a graph associated with the COST_TOUR constraint. Clearly, the number of vertices is equal to the number of variables, and the number of edges equals half of the sum of domain cardinalities. Thus, $n=|X|$ and $m=\frac{1}{2}\sum|D_{x_i}|$ for all $x_i\in X$.

For $f(x)=2$ the gadget, which is the complete bipartite subgraph $K_{d(x),d(x)-2}$, with weight $w(e)=0$ for every internal edge e in the gadget, obviously has $2d(x)-2$ vertices and $d(x)(d(x)-2)$ edges for every $x\in V$ and $d(x)\ge 2$. Thus, the incremental graph G^* will have $4m-2n\in\mathcal{O}(m)$ vertices and $\sum_{x\in V} d(x)^2-3m\in\Omega(m\cdot n)$ edges and the complexity of finding the maximum matching in incremental graph would be $\mathcal{O}(m^2\cdot(n+\log m))$ by using the algorithm due to Gabow [118] or $\mathcal{O}(m\cdot n\cdot\log(mW)\cdot\sqrt{m\cdot\alpha(m\cdot n,m)\cdot\log m})$ by applying the sophisticated algorithm due to Gabow & Tarjan [124].

Note that for $d(x) > 4$ the reduction procedure which allows us to transform the perfect 2-matching problem in G onto the perfect matching problem in G^* is very inefficient, because it highly increases the number of vertices and edges of the incremental graph. The computation of the minimum edge-weighted matching may be prohibitively expensive and can hinder this method from being systematically used during the search for a solution to a perfect 2-matching problem. Therefore, one way to reduce the time complexity would be to construct a graph G^* with the number of edges estimated by $\mathcal{O}(m)$. We define, for every $x \in V$, a more suitable gadget as the complete bipartite subgraph $K_{d(x),2}$ with $d(x)$ pendant edges attached at external vertices (see [220]). It is now easy to see that a perfect matching in G^* corresponds to a perfect f-matching in G in which matched edges form a perfect 2-matching. Since the resulting graph will have $4m + 2n$ vertices and $7m \in \mathcal{O}(m)$ edges, as required, the edge-weighted matching algorithm with sufficiently complexity will be maintained.

The COST_TOUR constraint is called a WEIGHTED_CIRCUIT constraint in [37]. The authors propose the filtering algorithm based on the 1-tree relaxation due to Michael Held and Richard M. Karp [156,157]. We demonstrate the behavior of the filtering method by the sample graph given in Figure 7.9. The graph has been taken from [37] as a running example. We examine this graph again.

D(x_1) = {2,3,6}
D(x_2) = {1,4,5}
D(x_3) = {1,4,5}
D(x_4) = {2,3,6}
D(x_5) = {2,3,6}
D(x_6) = {1,4,5}
D(cost) = [0,25]



D'(x_1) = {2,3}
D'(x_2) = {1,4,5}
D'(x_3) = {1,4,5}
D'(x_4) = {2,3,6}
D'(x_5) = {2,3,6}
D'(x_6) = {4,5}
D'(cost) = [24,25]

Figure 7.9: Pruning of the COST_TOUR constraint

COST_PATH Shortest path problems play a central role in both the design and use of communication networks. In this example we consider a constraint for the shortest path problem.

The global constraint COST_PATH is defined on an edge-weighted graph G. The constraint is satisfied if some edges form an optimal path in G between two distinguished vertices s and t. This constraint can be formally defined as follows:

$$\begin{aligned}
&\text{COST\_PATH}(\langle x_1, \dots, x_n\rangle, weights, cost) = \\
&\{((d_1^1, d_1^2), \dots, d_s, \dots, d_t, \dots, (d_n^1, d_n^2)) \in D_{x_1} \times \cdots \times D_{x_n} \mid \underset{i,j}{\forall} (j \in \{d_i^1, d_i^2\} \Leftrightarrow i \in \{d_j^1, d_j^2\}) \wedge \\
&\qquad \underset{S \subseteq \{1,\dots,n\} \setminus \{s,t\}}{\forall} \left(\bigcup_{i \in S} \{d_i^1, d_i^2\} \neq S \right) \wedge \\
&\qquad\qquad min(D_{cost}) \leq \sum_{\substack{i=1 \\ i<d_i}}^{n} weight(x_i, x_{d_i}) \leq max(D_{cost})\}.
\end{aligned}$$

The COST_PATH constraint was introduced in [268] under the name SHORTER_PATH, where also an incomplete cost-based filtering was given. In this example we show how to solve the problem of optimal path between a designated pair of vertices by solving an edge-weighted parity (g, f)-matching problem. We transform the COST_PATH constraint into an edge-weighted parity (g, f)-matching problem as follows. Suppose that we want to solve the optimal path problem from the source vertex s to the target vertex t. We construct an auxiliary edge-weighted graph G. We set $f(s) = f(t) = 1$ and $g(x) = 0$, $f(x) = 2$ for all the remaining vertices. It is easy to see that any path from s to t represents a parity (g, f)-matching M in the corresponding edge-weighted graph G (i.e. $d_M(s) = d_M(t) = 1$ and $d_M(x) \in \{0, 2\}$ for all other vertices x).

In order to transform the parity (g, f)-matching problem into an f-matching problem we add a loop of weight $w(x, x) = 0$ to each vertex x, except vertices s and t. Now we could transform the f-matching problem to the perfect matching problem as demonstrated in the previous example, but we will present another transformation. The transformation that we give is taken from [4, Chapter 12.7]. It consists of two steps. In the first step, we insert two additional vertices, x'_e and x''_e, in the middle of each edge $e = \{x_i, x_j\}$ for which $x_i \neq x_j$ and $f(x_i) = f(x_j) = 2$. Further, we set $w(x_i, x'_e) = w(x_j, x''_e) = \frac{1}{2}w(x_i, x_j)$ and $w(x'_e, x''_e) = 0$. In the second step of the transformation we split each vertex x_i such that $f(x_i) = 2$ into two vertices x'_i and x''_i. For each edge $\{x_i, x_j\}$ we introduce two edges $\{x'_i, x_j\}$ and $\{x''_i, x_j\}$ with the same weight as the edge $\{x_i, x_j\}$. It is easy to establish an equivalence between the edge-weighted perfect f-matching in G and the edge-weighted perfect matching in the resulting graph. Therefore, we can obtain an optimal path with the same cost by solving the admissible edge-weighted perfect matching problem by any polynomial time algorithm.

It is easy to calculate that the resulting graph will have at most $2m + 2n$ vertices and at most $5m$ edges. Hence, the size of the resulting graph is polynomial in the size of the original graph. Furthermore, the cost of a perfect f-matching in G is equal to the cost of the corresponding perfect matching M in the resulting graph G^* (Figure 7.10).

The problem of finding an optimal path connecting two given vertices of an undirected connected graph is trivial since there is only one such path. To decide whether there exists an optimal path that visits a certain set of edges is NP-hard. Consequently the problem of achieving hyper-arc consistency for the COST_PATH constraint is also NP-hard.

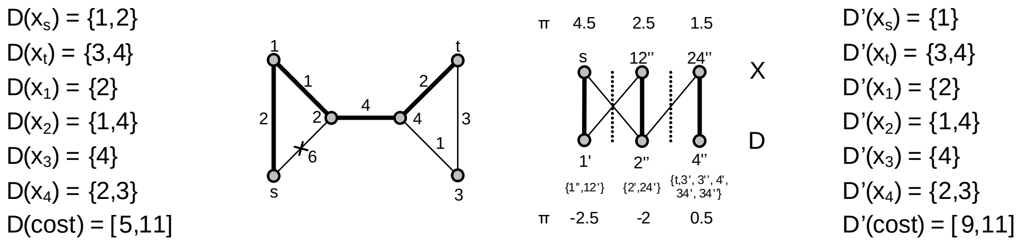


Figure 7.10: Pruning of the COST_PATH constraint

7.4 Summary

In this chapter we have proposed an efficient way of implementing a filtering algorithm for the optimization constraints. The method is based on the modified version of the alternating depth-first search (for vertex-weighted matchings) and alternating-breadth-first search (for edge-weighted matchings). We have demonstrated that weighted matching is a powerful tool for cost-based modeling several global constraints. We have shown that hyper-arc consistent filtering algorithm is NP-hard if an optimal solution should be found as lying within the given interval of bounds. It remains open whether the technique we have used can be generalized to more optimization constraints.

The following table summarizes all the results for optimization constraints that were discussed in this chapter. The constraints designated with an asterisk are NP-hard. Here n denotes the total number of vertices in the graph G associated with the constraint, k is the minimum of the number of values (the cardinality of the union of all variable domains) or the number of variables, m is the number of edges (the sum of the cardinalities of the variable domains), h denotes half of the number of vertices incident with the forbidden edges (the number of pruned variable-value pairs), p denotes the number of maximal extreme sets in G, and $t = \max\{1, \tau(G)\}$. By $S(n,m)$ we denote the time required to solve a shortest path problem in a graph with n vertices and m edges, e.g. $S(n,m) \in \mathcal{O}(m + n \cdot \log n)$.

optimization constraint	weighted matching	checking feasibility	hyper-arc consistency	reference
WPA	vertex	$\mathcal{O}(n \cdot m)$	$\mathcal{O}(h \cdot m)$	[288]
NVALUE(*)	vertex	-	-	[26],[41]
SWDV(*)	vertex	-	-	[28]
COST_ALLDIFFERENT	edge	$\mathcal{O}(n \cdot S(n,m))$	$\mathcal{O}(t \cdot S(k,m))$	here
MWA	edge	$\mathcal{O}(n \cdot S(n,m))$	$\mathcal{O}(t \cdot S(k,m))$	[267]
SOFT_INVERSE_VAR	edge	$\mathcal{O}(n \cdot S(n,n^2))$	$\mathcal{O}(t \cdot S(n,n^2))$	here
COST_GCC	edge	$\mathcal{O}(n \cdot S(n,m))$	$\mathcal{O}(t \cdot S(n,m))$	[258]
COST_SYMMETRIC _CARDINALITY	edge	$\mathcal{O}(f(V) \cdot S(n,m))$	$\mathcal{O}(t \cdot S(n,m))$	[190]
COST_SYMMETRIC _ALLDIFFERENT	edge	$\mathcal{O}(n \cdot S(n,m))$	$\mathcal{O}(t \cdot p \cdot S(n,m))$	here
COST_TOUR(*)	edge	-	-	here
COST_PATH(*)	edge	-	-	[268]

Table 7.5: Summary of results for weighted graphs

Chapter 8

Conclusion

In the course of this thesis we have studied an application of matching theory within a constraint programming framework. The presented work answered some open questions in the context of global constraints.

We hope that our research will be a powerful tool in the constraint programming community. In fact, a generic filtering algorithm for global constraints whose solution is represented by a matching in a particular graph can be obtained from our results in a quite natural way.

In the final chapter of this thesis we summarize the preceding chapters, highlight open problems and present some ideas on future work.

Summary of Results

In Chapter 2 we surveyed some of the fundamental definitions relating to the area of complexity theory, graph theory, matching theory, and constraint programming.

In Chapter 3 we discussed algorithms based on searching a graph using either a depth-first search or a breadth-first search. Some applications of these two traversals were given. The contribution of this chapter was Gray-Path Theorem for a depth-first search and the algorithm for the classification of edges in a breadth-first search. Until now no such algorithm was known. Later in this thesis it had been shown how these results can be combined to create a generic filtering algorithm.

In Chapter 4 we presented decomposition theory for bipartite graphs and proposed filtering algorithms for global constraints representable as a matching problem in a bipartite graph. The algorithm is based on the modified version of the graph traversals which we have named an alternating breadth-first search and an alternating depth-first search. The additional contribution of this chapter was the Dulmage-Mendelsohn Canonical Decomposition for degree-matchings and the theorem that the degrees of bipartite degree-factors are consecutive.

In Chapter 5 we presented decomposition theory for general graphs and proposed filtering algorithms for global constraints representable as a matching problem in a general graph. The contribution of this chapter was the decomposition into matching covered subgraphs for f-matchings. We have formally defined extreme sets with respect to f-matchings and provided some of their properties.

In Chapter 6 we presented decomposition theory for directed graphs. We have discovered an algorithm for computing strongly connected components based on matching theory. We introduced a useful filtering technique based on a directed matching and showed how it is possible to transform any directed matching into a bipartite matching. The method described in Chapter 4 allowed us to solve some graph partitioning constraints efficiently. We also presented efficient methods for incomplete pruning of the variable domains. Our result can be applied to other constraints with a similar graph representation and structure.

In Chapter 7 we dealt with the weighted matching problems. We have presented a filtering framework based on weighted matching that uses decomposition techniques. The contribution of this chapter was the cost-based propagation algorithm for optimization constraints. We have provided a clear and complete description of the use of decomposition theory to prune domains of variables in some global optimization constraints. We have proved, by transformation from the interval subset sum problem, that a general weighted degree-matching problem with lower and upper bounds on the cost is NP-complete. The partition of edges in the vertex-weighted bipartite graph can be obtained in linear time by using a modified depth-first search called a weighted alternating depth-first search. The partition of edges in the edge-weighted graph associated with the optimization constraint can be determined by forming an alternating breadth-first forest by using a modified breadth-first search called a weighted alternating breadth-first search. The method may be regarded as a variant of Dijkstra's algorithm.

Open Problems

We have some open problems that could not be solved in this thesis.

1. The classification of edges in a breadth-first search in only one (traversal) phase.
2. The existence of a linear algorithm for the detection and handling of extreme sets.
3. The definition of extreme sets with respect to perfect (g, f)-matchings.
4. A hyper-arc consistency algorithm for the SOFT_2-CYCLE_VAR constraint with the complexity of $\mathcal{O}(\sqrt{n} \cdot m)$.
5. Can a pruning according to dominators be realized with the help of matching theory?
6. How many solutions does a BINARY_TREE constraint with domains $[1, n]$ have?
7. Can a (weighted) spanning tree constraint [80, Chapter 5],[247],[260],[262] be represented as a (weighted) matching problem?
8. Can an edge-weighted perfect (g, f)-matching problem with arbitrary costs be transformed into an edge-weighted perfect (g, f)-matching problem with non-negative costs?

We conclude by listing all the global constraints that were discussed in this thesis.

global constraint	abbreviation	discussed in
ALLDIFFERENT ALLDIFFERENT_EXCEPT_0		Chapter 4

BINARY_TREE(*)
CIRCUIT(*)
		Chapter 6

CLIQUE(*)		Chapter 5
CORRESPONDENCE		Chapter 4
COST_ALLDIFFERENT COST_GLOBAL_CARDINALITY COST_PATH(*) COST_SYMMETRIC_ALLDIFFERENT COST_SYMMETRIC_CARDINALITY COST_TOUR(*)	 COST_GCC COST_SYMMETRIC_GCC 	Chapter 7
CYCLE(*) DERANGEMENT		Chapter 6
GLOBAL_CARDINALITY INVERSE	GCC	Chapter 4
MAP(*)		Chapter 6
MINIMUM_WEIGHT_ALLDIFFERENT NVALUE(*)	MWA	Chapter 7
PATH(*)		Chapter 6
PROPER_FOREST		Chapter 5
SAME SOFT_ALLDIFFERENT_VAR		Chapter 4
SOFT_DERANGEMENT_VAR		Chapter 6
SOFT_GLOBAL_CARDINALITY_VAR	SOFT_GCC_VAR	Chapter 4
SOFT_INVERSE_VAR		Chapter 7
SOFT_SAME_VAR		Chapter 4
SOFT_SYMMETRIC_ALLDIFFERENT_VAR		Chapter 5
SOFT_SYMMETRIC_CARDINALITY_VAR SOFT_USED_BY_VAR SORT SORT_PERMUTATION		Chapter 4
SUM_OF_WEIGHTS_OF_DISTINCT_VALUES	SWDV(*)	Chapter 7
SYMMETRIC_ALLDIFFERENT SYMMETRIC_ALLDIFFERENT_EXCEPT_0 SYMMETRIC_ALLDIFFERENT_LOOP		Chapter 5
SYMMETRIC_CARDINALITY	SYMMETRIC_GCC	Chapter 4
TOUR(*)		Chapter 5
TREE		Chapter 6
UNDIRECTED_PATH(*)		Chapter 5
USED_BY		Chapter 4
WEIGHTED_PARTIAL_ALLDIFF	WPA	Chapter 7

Table 8.1: List of global constraints discussed in this thesis

Future Work

In this thesis preliminary work on applications of matching theory in constraint programming is investigated. As a part of our future work we plan to continue and extend our research, improve the current implementation and develop new algorithms. Some specific goals for the immediate future include:

Empirical evaluation. All algorithms we developed here are practicable and easy to implement. Thus, we expect this work to be relevant for many applications and practical approaches in the subject of constraint programming. We consider an empirical evaluation of our algorithms to be an interesting question that deserves further study.

Implementation. All methods and algorithms presented in this thesis have been implemented in C using LEDA library. The code has been developed in Borland C++ Builder. The code can also be compiled using GCC. An important piece of work would be an implementation of our pruning technique into some constraint solver libraries such as, for example, Gecode, JaCoP, Choco, SICStus and other.

Component matchings. In this thesis we have only investigated degree-matchings. Another interesting kind of matchings is component matchings. A component matching is defined in terms of spanning subgraphs having specified (connected) components. From this point of view, ordinary 1-factors are just the same as K_2-factors, 2-factors are the same as cycle factors, and acyclic (1,2)-factors are exactly path factors. Although we have considered cycle and path factors we did not give more details about the necessary or sufficient conditions for a graph to have a component factor.

Principle of duality. The principle of duality is an important concept. We have outlined and provided some examples thereof. It would be very interesting to give more results where duality theory can be applied.

Continuity property in general graphs. We have proven that bipartite degree-factors have a continuity property (see Theorem 4.3.15). We know that this result does not hold for general graphs. But an interesting question is whether we can determine vertices in a general graph whose degrees are consecutive in degree-factors.

Balanced constraints. One of the interesting questions for future work would be to apply our approach to balanced constraints such as BALANCE_CYCLE, BALANCE_PATH and BALANCE_TREE [27]. These constraints are characterized by the additional parameter measuring the difference between the number of vertices in the smallest pattern and the number of vertices in the largest pattern.

Weighted directed matchings. We have given the application of matching theory to the constraints representable by bipartite, general, directed and weighted graphs. Another natural extension of the matching problem arises when considering weighted directed matchings in digraphs.

Degree-matchings and linear programming. The weighted matching [89] and the weighted f-matching [304] (see also [24]) can be successfully formulated in terms of theory of linear programming. An interesting research area for future work would be to describe a linear programming formulation for all degree-matchings, especially weighted (g, f)-matchings.

Soft optimization constraints. For optimization constraints, the typical goal is to find the value of the objective function at the optimal solution to a hard combinatorial problem. Soft optimization constraints would be useful for addressing optimization problems that might be over-constrained. Until now, such constraints have not been investigated.

Hypergraphs. We would like to apply our research to hypergraphs. There are some sufficient conditions known for the existence of perfect matchings in hypergraphs. Some of these sufficient conditions are not computable in polynomial time. Let us remark that the matching problem of 3-uniform hypergraph[1] is equivalent to the 3-dimensional matching problem, which is NP-complete [182],[131, Problem SP1]. The reduction from the 3-dimensional matching problem to a global constraint problem allows us to prove that the problem is intractable.

Further constraints. Since not all the matching-based constraints have been discussed in this thesis, it could also be beneficial and valuable to find and have a look at more constraints.

Set variables. Finally, it would be interesting if the presented framework of this thesis could also cope with set variables by generalizing the obtained results for integer variables[2].

The last two areas of future work are not explicitly connected with matching theory.

Bipartite crossing. In the course of our research, before we dealt with the area of this thesis, we investigated the crossing number problem in bipartite graphs. The crossing number is the minimum possible number of crossings with which the graph can be drawn. A graph with crossing number 0 is a planar graph. This problem is NP-complete [132] but can be solved in linear time for caterpillars[3]. We wish to continue this research.

Planar CIRCUIT constraint. In this thesis we have given a partial filtering for the TOUR constraint. Another interesting problem is to find a Hamiltonian cycle in a planar graph. In this case we can use for pruning a necessary condition discovered by the Latvian mathematician Emanuel Grinberg [145] in 1968 (see also [270],[312]). The Hamiltonian cycle problem is NP-complete even for planar graphs [133].

[1]A hypergraph is called *r-uniform* if every edge contains precisely r vertices. Thus, the 2-uniform hypergraphs are exactly the ordinary undirected graphs without loops.

[2]An integer variable takes a value from a given integer set or a given integer interval, while a set variable takes a value from a given set of integer sets.

[3]A *caterpillar* (or a *comb*) is a tree in which every vertex is on a central stalk or only one edge away from the stalk. In other words, a caterpillar is a tree such that deleting all the leaves produces a (possibly empty) path. We refer to the remaining path as the spine of the caterpillar. The edges of a caterpillar can be partitioned into two sets: the spine edges, and the leaf edges.

Bibliography

[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[2] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data Structures and Algorithms.* Addison-Wesley, 1983.

[3] Alfred V. Aho and Jeffrey D. Ullman. *Principles of Compiler Design.* Addison-Wesley, Reading, Massachusetts, 1977.

[4] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows: Theory, Algorithms and Applications.* Prentice-Hall, Upper Saddle River, New Jersey, 1993.

[5] Ravindra K. Ahuja, Kurt Mehlhorn, James B. Orlin, and Robert E. Tarjan. Faster algorithms for the shortest path problem. *Journal of the Association for Computing Machinery*, 37(2):213–223, 1990.

[6] Ravindra K. Ahuja and James B. Orlin. A faster algorithm for the inverse spanning tree problem. *Journal of Algorithms*, 34(1):177–193, 2000.

[7] Martin Aigner. *Combinatorial Theory.* Springer, New York, 1979.

[8] Jin Akiyama and Mikio Kano. *Factors and Factorizations of Graphs*, volume 2031 of *Lecture Notes in Mathematics.* Springer, 2011.

[9] Stephen Alstrup, Dov Harel, Peter W. Lauridsen, and Mikkel Thorup. Dominators in linear time. *SIAM Journal on Computing*, 28(6):2117–2132, 1999.

[10] Helmut Alt, Norbert Blum, Kurt Mehlhorn, and Markus Paul. Computing a maximum cardinality matching in a bipartite graph in time $\mathcal{O}(n^{1.5}\sqrt{m/\log n})$. *Information Processing Letters*, 37(4):237–240, 1991.

[11] Ernst Althaus, Alexander Bockmayr, Matthias Elf, Thomas Kasper, Michael Jünger, and Kurt Mehlhorn. SCIL – Symbolic Constraints in Integer Linear Programming. In Rolf H. Möhring and Rajeev Raman, editors, *Algorithms – ESA 2002, 10th Annual European Symposium, Rome, Italy, September 17-21, 2002, Proceedings*, volume 2461 of *Lecture Notes in Computer Science*, pages 75–87. Springer, 2002.

[12] Richard P. Anstee. An algorithmic proof of Tutte's f-factor theorem. *Journal of Algorithms*, 6(1):112–131, 1985.

[13] Krzysztof R. Apt. *Principles of Constraint Programming.* Cambridge University Press, New York, 2003.

[14] Bruce W. Arden, Bernard A. Galler, and Robert M. Graham. An algorithm for equivalence declarations. *Communications of the ACM*, 4(7):310–314, 1961.

[15] Armen S. Asratian, Tristan M. J. Denley, and Roland Häggkvist. *Bipartite Graphs and their Applications.* Cambridge University Press, Cambridge, 1998.

[16] Mike D. Atkinson, Jörg-Rüdiger W. Sack, Nicola Santoro, and Thomas Strothotte. Min-max heaps and generalized priority queues. *Communications of the ACM*, 29(10):996–1000, 1986.

[17] Michel L. Balinski. Labelling to obtain a maximum matching. *Combinatorial Mathematics and its Applications*, pages 585–602, 1969.

[18] Michel L. Balinski and Ralph E. Gomory. A primal method for the assignment and transportation problems. *Management Science*, 10(3):578–593, 1964.

[19] Michel L. Balinski and Jaime Gonzalez. Maximum matchings in bipartite graphs via strong spanning trees. *Networks*, 21(2):165–179, 1991.

[20] Michael O. Ball and Ulrich Derigs. An analysis of alternative strategies for implementing matching algorithms. *Networks*, 13(4):517–549, 1983.

[21] Jørgen Bang-Jensen and Gregory Gutin. *Digraphs: Theory, Algorithms and Applications.* Springer, second edition, 2009. Previous edition 2000.

[22] Jørgen Bang-Jensen, Gregory Gutin, and Hao Li. Sufficient conditions for a digraph to be Hamiltonian. *Journal of Graph Theory*, 22(2):181–187, 1996.

[23] Holger Bast, Kurt Mehlhorn, Guido Schäfer, and Hisao Tamaki. Matching algorithms are fast in sparse random graphs. In Volker Diekert and Michel Habib, editors, *STACS 2004, 21st Annual Symposium on Theoretical Aspects of Computer Science, Montpellier, France, March 25-27, 2004, Proceedings*, volume 2996 of *Lecture Notes in Computer Science*, pages 81–92. Springer, 2004.

[24] Mohsen Bayati, Christian Borgs, Jennifer Chayes, and Riccardo Zecchina. Belief-propagation for weighted b-matchings on arbitrary graphs and its relation to linear programs with integer solutions. *SIAM Journal on Discrete Mathematics*, 25(2):989–1011, 2011.

[25] Lowell W. Beineke and Michael D. Plummer. On the 1-factors of a non-separable graph. *Journal of Combinatorial Theory*, 2(3):285–289, 1967.

[26] Nicolas Beldiceanu. Pruning for the minimum constraint family and for the number of distinct values constraint family. In Toby Walsh, editor, *Principles and Practice of Constraint Programming – CP 2001, 7th International Conference, CP 2001, Paphos, Cyprus, November 26 - December 1, 2001, Proceedings*, volume 2239 of *Lecture Notes in Computer Science*, pages 211–224. Springer, 2001.

[27] Nicolas Beldiceanu, Mats Carlsson, and Jean-Xavier Rampon. Global Constraint Catalog. Technical Report T2012-03, Swedish Institute of Computer Science, 2012.

[28] Nicolas Beldiceanu, Mats Carlsson, and Sven Thiel. Cost-filtering algorithms for the two sides of the sum of weights of distinct values constraint. Technical Report T2002-14, Swedish Institute of Computer Science, 2002.

[29] Nicolas Beldiceanu and Evelyne Contejean. Introducing global constraints in CHIP. *Mathematical and Computer Modelling*, 20(12):97–123, 1994.

[30] Nicolas Beldiceanu, Pierre Flener, and Xavier Lorca. The tree constraint. In Roman Barták and Michela Milano, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, 2nd International Conference, CPAIOR 2005, Prague, Czech Republic, May 30 - June 1, 2005, Proceedings*, volume 3524 of *Lecture Notes in Computer Science*, pages 64–78. Springer, 2005.

[31] Nicolas Beldiceanu, Fabien Hermenier, Xavier Lorca, and Thierry Petit. The increasing nvalue constraint. In Andrea Lodi, Michela Milano, and Paolo Toth, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, 7th International Conference, CPAIOR 2010, Bologna, Italy, June 14-18, 2010. Proceedings*, volume 6140 of *Lecture Notes in Computer Science*, pages 25–39. Springer, 2010.

[32] Nicolas Beldiceanu, Irit Katriel, and Xavier Lorca. Undirected forest constraints. In J. Christopher Beck and Barbara M. Smith, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, 3rd International Conference, CPAIOR 2006, Cork, Ireland, May 31 - June 2, 2006, Proceedings*, volume 3990 of *Lecture Notes in Computer Science*, pages 29–43. Springer, 2006.

[33] Nicolas Beldiceanu, Irit Katriel, and Sven Thiel. Filtering algorithms for the same constraint. In Jean-Charles Régin and Michel Rueher, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, First International Conference, CPAIOR 2004, Nice, France, April 20-22, 2004, Proceedings*, volume 3011 of *Lecture Notes in Computer Science*, pages 65–79. Springer, 2004.

[34] Nicolas Beldiceanu and Xavier Lorca. Necessary condition for path partitioning constraints. In Pascal Van Hentenryck and Laurence A. Wolsey, editors, *Integration of AI*

and OR Techniques in Constraint Programming for Combinatorial Optimization Problems: 4th International Conference, CPAIOR 2007, Brussels, Belgium, May 23-26, 2007. Proceedings, volume 4510 of *Lecture Notes in Computer Science*, pages 141–154. Springer, 2007.

[35] Nicolas Beldiceanu and Thierry Petit. Cost evaluation of soft global constraints. In Jean-Charles Régin and Michel Rueher, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, First International Conference, CPAIOR 2004, Nice, France, April 20-22, 2004, Proceedings*, volume 3011 of *Lecture Notes in Computer Science*, pages 80–95. Springer, 2004.

[36] Nicolas Beldiceanu and Helmut Simonis. A model seeker. Description and detailed results. Update of Technical Report 4C-2012-01. Cork Constraint Computation Centre, University College Cork, 19 May 2012.

[37] Pascal Benchimol, Willem-Jan van Hoeve, Jean-Charles Régin, Louis-Martin Rousseau, and Michel Rueher. Improved filtering for weighted circuit constraints. *Constraints*, 17(3):205–233, 2012.

[38] Claude Berge. Two theorems in graph theory. In *Proceedings of the National Academy of Sciences of the United States of America*, volume 43, pages 842–844, 1957.

[39] Claude Berge. *The Theory of Graphs and its Applications.* John Wiley & Sons, New York, 1962.

[40] Claude Berge. *Graphs and Hypergraphs.* North-Holland, Amsterdam, 1973.

[41] Christian Bessiere, Emmanuel Hebrard, Brahim Hnich, Zeynep Kiziltan, and Toby Walsh. Filtering algorithms for the nvalue constraint. In Roman Barták and Michela Milano, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, 2nd International Conference, CPAIOR 2005, Prague, Czech Republic, May 30 - June 1, 2005, Proceedings*, volume 3524 of *Lecture Notes in Computer Science*, pages 79–93. Springer, 2005.

[42] Christian Bessiere, Emmanuel Hebrard, Brahim Hnich, and Toby Walsh. The complexity of global constraints. In Deborah L. McGuinness and George Ferguson, editors, *Proceedings of the 19th National Conference of Artificial Intelligence (AAAI)*, pages 112–117, San Jose, California, 2004.

[43] Christian Bessiere and Pascal Van Hentenryck. To be or not to be ... a global constraint. In Francesca Rossi, editor, *Principles and Practice of Constraint Programming – CP 2003, 9th International Conference, CP 2003, Kinsale, Ireland, September 29 - October 3, 2003, Proceedings*, volume 2833 of *Lecture Notes in Computer Science*, pages 789–794. Springer, 2003.

[44] Noëlle Bleuzen-Guernalec and Alain Colmerauer. Narrowing a 2n-block of sortings in $\mathcal{O}(n \log n)$. In Gert Smolka, editor, *Principles and Practice of Constraint Programming*

– CP97, Third International Conference, CP97, Linz, Austria, October 29 - November 1, 1997, Proceedings, volume 1330 of *Lecture Notes in Computer Science*, pages 2–16. Springer, 1997.

[45] Norbert Blum. On the single-operation worst-case time complexity of the disjoint set union problem. *SIAM Journal on Computing*, 15(4):1021–1024, 1986.

[46] Norbert Blum. A new approach to maximum matching in general graphs. In Mike Paterson, editor, *Automata, Languages and Programming, 17th International Colloquium, ICALP90, Warwick University, England, July 16-20, 1990, Proceedings*, volume 443 of *Lecture Notes in Computer Science*, pages 586–597. Springer, 1990.

[47] Norbert Blum. Maximum matching in general graphs without explicit consideration of blossoms. Research Report, 26 October 1999. Universität Bonn.

[48] Norbert Blum. A simplified realization of the Hopcroft-Karp approach to maximum matching in general graphs. Research Report, 26 October 1999. Universität Bonn.

[49] Béla Bollobás. *Extremal Graph Theory.* Academic Press, London, 1978.

[50] Kellogg S. Booth and George S. Lueker. Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms. *Journal of Computer and System Sciences*, 13(3):335–379, 1976.

[51] Eric Bourreau. *Traitement de contraintes sur les graphes en programmation par contraintes [Processing constraints on graphs in the framework of Constraint Programming].* PhD thesis, University Paris, France, 1999. In French.

[52] Mark R. Brown. Implementation and analysis of binomial queue algorithms. *SIAM Journal on Computing*, 7(3):298–319, 1978.

[53] Adam L. Buchsbaum, Haim Kaplan, Anne Rogers, and Jeffery R. Westbrook. A new, simpler linear-time dominators algorithm. *ACM Transactions on Programming Languages and Systems*, 20(6):1265–1296, 1998.

[54] Adam L. Buchsbaum, Haim Kaplan, Anne Rogers, and Jeffery R. Westbrook. Corrigendum: A new, simpler linear-time dominators algorithm. *ACM Transactions on Programming Languages and Systems*, 27(3):383–387, 2005.

[55] Svante Carlsson. The deap – a double-ended heap to implement double-ended priority queues. *Information Processing Letters*, 26(1):33–36, 1987.

[56] Svante Carlsson, Jingsen Chen, and Thomas Strothotte. A note on the construction of the data structure "deap". *Information Processing Letters*, 31(6):315–317, 1989.

[57] Yves Caseau and François Laburthe. Solving small TSPs with constraints. In Lee Naish, editor, *Proceedings of the 14th International Conference on Logic Programming (ICLP), Leuven, Belgium, July 8-11, 1997*, pages 316–330, 1997.

[58] Yves Caseau and François Laburthe. Solving various weighted matching problems with constraints. *Constraints*, 5(1):141–160, 2000.

[59] Arthur Cayley. A theorem on trees. *Quarterly Journal of Pure and Applied Mathematics*, 23:376–378, 1889.

[60] Timothy M. Chan. Quake heaps: A simple alternative to Fibonacci heaps. In Andrej Brodnik, Alejandro López-Ortiz, Venkatesh Raman, and Alfredo Viola, editors, *Space-Efficient Data Structures, Streams, and Algorithms – Papers in Honor of Munro, J. Ian on the Occasion of His 66th Birthday*, volume 8066 of *Lecture Notes in Computer Science*, pages 27–32. Springer, 2013.

[61] Sung C. Chang and Min W. Du. Diamond deque: A simple data structure for priority deques. *Information Processing Letters*, 46(5):231–237, 1993.

[62] Joseph Cheriyan and Kurt Mehlhorn. Algorithms for dense graphs and networks on the random access computer. *Algorithmica*, 15(6):521–549, 1996.

[63] Boris V. Cherkassky, Andrew V. Goldberg, and Craig Silverstein. Buckets, heaps, lists and monotone priority queues. In *Proceedings of the 8th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 83–92, New Orleans, Louisiana, 1997.

[64] Boris V. Cherkassky, Andrew V. Goldberg, and Craig Silverstein. Buckets, heaps, lists and monotone priority queues. *SIAM Journal on Computing*, 28(4):1326–1346, 1999.

[65] Nicos Christofides. *Graph Theory: An Algorithmic Approach.* Academic Press, London, 1975.

[66] Keith D. Cooper, Timothy J. Harvey, and Ken Kennedy. A simple, fast dominance algorithm. *Software-Practice and Experience*, 4:1–28, 2001.

[67] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms.* The MIT Press, Cambridge, Massachusetts, second edition, 2001. Previous edition 1990.

[68] Gérard Cornuéjols. General factors of graphs. *Journal of Combinatorial Theory, Series B*, 45(2):185–198, 1988.

[69] Clark A. Crane. *Linear lists and priority queues as balanced binary trees.* Doctoral dissertation, Stanford University. Department of Computer Science, Stanford, California, USA, 1972.

[70] Radosław Cymer. Dulmage-Mendelsohn canonical decomposition as a generic pruning technique. *Constraints*, 17(3):234–272, 2012.

[71] Radosław Cymer. Weighted matching as a generic pruning technique applied to optimization constraints. *Annals of Operations Research*, 217(1):165–211, 2014.

[72] Radosław Cymer. Gallai-Edmonds decomposition as a pruning technique. *Central European Journal of Operations Research*, 23(1):149–185, 2015.

[73] Radosław Cymer. Propagation rules for graph partitioning constraints. *Journal of Graph Algorithms and Applications*, 20(2):363–410, 2016.

[74] Rina Dechter. *Constraint Processing.* Morgan Kaufmann Publishers, San Francisco, California, 2003.

[75] Pablo Diaz-Gutierrez and Meenakshisundaram Gopi. Quadrilateral and tetrahedral mesh stripification using 2-factor partitioning of the dual graph. *The Visual Computer*, 21(8-10):689–697, 2005.

[76] Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[77] Edsger W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, New Jersey, 1976.

[78] Efim A. Dinits. Algorithm for solution of a problem of maximum flow in a network with power estimation. *Soviet Mathematics Doklady*, 11(5):1277–1280, 1970.

[79] Efim A. Dinits and Mikhail A. Kronrod. An algorithm for the solution of the assignment problem. *Soviet Mathematics Doklady*, 10(6):1324–1326, 1969.

[80] Grégoire Dooms. *The CP(Graph) Computation Domain in Constraint Programming.* PhD thesis, Faculté des Sciences Appliquées, Université Catholique de Louvain, Louvain-La-Neuve, Belgium, 2006.

[81] James R. Driscoll, Harold N. Gabow, Ruth Shrairman, and Robert E. Tarjan. Relaxed heaps: An alternative to Fibonacci heaps with applications to parallel computation. *Communications of the ACM*, 31(11):1343–1354, 1988.

[82] Ran Duan and Hsin-Hao Su. A scaling algorithm for maximum weight matching in bipartite graphs. In *Proceedings of the 23rd Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1413–1424, Kyoto, Japan, 2012.

[83] Andrew L. Dulmage and Nathan S. Mendelsohn. Coverings of bipartite graphs. *Canadian Journal of Mathematics*, 10:517–534, 1958.

[84] Andrew L. Dulmage and Nathan S. Mendelsohn. A structure theory of bipartite graphs of finite exterior dimension. *Transactions of the Royal Society of Canada, Series III*, 53:1–13, 1959.

[85] Andrew L. Dulmage and Nathan S. Mendelsohn. On the inversion of sparse matrices. *Mathematics of Computation*, 16:494–496, 1962.

[86] Andrew L. Dulmage and Nathan S. Mendelsohn. Two algorithms for bipartite graphs. *Journal of the Society for Industrial and Applied Mathematics*, 11(1):183–194, 1963.

[87] Thomas E. Easterfield. A combinatorial algorithm. *The Journal of the London Mathematical Society*, 21(3):219–226, 1946.

[88] Jürgen Ebert. A note on odd and even factors of undirected graphs. *Information Processing Letters*, 11(2):70–72, 1980.

[89] Jack Edmonds. Maximum matching and a polyhedron with 0,1-vertices. *Journal of Research of the National Bureau of Standards*, 69B(1-2):125–130, 1965.

[90] Jack Edmonds. Paths, trees and flowers. *Canadian Journal of Mathematics*, 17:449–467, 1965.

[91] Jack Edmonds and Richard M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the Association for Computing Machinery*, 19(2):248–264, 1972.

[92] Jenő Egerváry. Matrixok kombinatorius tulajdonságairól [On combinatorial properties of matrices]. *Matematikai és Fizikai Lapok*, 38:16–28, 1931. In Hungarian.

[93] Amr Elmasry. Layered heaps. In Torben Hagerup and Jyrki Katajainen, editors, *Algorithm Theory - SWAT 2004, 9th Scandinavian Workshop on Algorithm Theory, Humlebæk, Denmark, July 8-10, 2004, Proceedings*, volume 3111 of *Lecture Notes in Computer Science*, pages 212–222. Springer, 2004.

[94] Amr Elmasry. The violation heap: A relaxed Fibonacci-like heap. In My T. Thai and Sartaj Sahni, editors, *Computing and Combinatorics, 16th Annual International Conference, COCOON 2010, Nha Trang, Vietnam, July 19-21, 2010. Proceedings*, volume 6196 of *Lecture Notes in Computer Science*, pages 479–488. Springer, 2010.

[95] Amr Elmasry, Claus Jensen, and Jyrki Katajainen. Two-tier relaxed heaps. *Acta Informatica*, 45(3):193–210, 2008.

[96] Leonhard Euler. Solutio problematis ad geometriam situs pertinentis [The solution of a problem relating to the geometry of position]. *Commentarii Academiae Scientarium Imperialis Petropolitanae*, 8:128–140, 1736/1741. In Latin.

[97] Shimon Even and Oded Kariv. An $\mathcal{O}(n^{2.5})$ algorithm for maximum matching in general graphs. In *Proceedings of the 16th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 100–112, Berkeley, California, 1975.

[98] Shimon Even and Robert E. Tarjan. Computing an st-numbering. *Theoretical Computer Science*, 2(3):339–344, 1976.

[99] François Fages and Akash Lal. A global constraint for cutset problems. In *5th International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR)*, Montreal, Canada, 2003.

[100] Jean-Guillaume Fages and Xavier Lorca. Revisiting the tree constraint. In Jimmy Ho-Man Lee, editor, *Principles and Practice of Constraint Programming – CP 2011, 17th International Conference, CP 2011, Perugia, Italy, September 12-16, 2011. Proceedings*, volume 6876 of *Lecture Notes in Computer Science*, pages 271–285. Springer, 2011.

[101] Torsten Fahle. Cost based filtering vs. upper bounds for maximum clique. In *4th International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR)*, pages 93–107, Le Croisic, France, 2002.

[102] Tomás Feder and Rajeev Motwani. Clique partitions, graph compression and speeding-up algorithms. *Journal of Computer and System Sciences*, 51(2):261–272, 1995.

[103] Michael J. Fischer. Efficiency of equivalence algorithms. In Raymond E. Miller and James W. Thatcher, editors, *Complexity of Computer Computations*, pages 153–167. Plenum Press, New York, 1972.

[104] Philippe Flajolet and Andrew M. Odlyzko. Random mapping statistics. In Jean-Jacques Quisquater and Joos Vandewalle, editors, *Advances in Cryptology - EUROCRYPT '89, Workshop on the Theory and Application of Cryptographic Techniques, Houthalen, Belgium, April 10-13, 1989, Proceedings*, volume 434 of *Lecture Notes in Computer Science*, pages 329–354. Springer, 1990.

[105] Filippo Focacci, Andrea Lodi, and Michela Milano. Cost-based domain filtering. In Joxan Jaffar, editor, *Principles and Practice of Constraint Programming – CP'99, 5th International Conference, Alexandria, Virginia, USA, October 11-14, 1999, Proceedings*, volume 1713 of *Lecture Notes in Computer Science*, pages 189–203. Springer, 1999.

[106] Lester R. Ford, Jr. and Delbert R. Fulkerson. *Flows in Networks*. Princeton University Press, Princeton, New Jersey, USA, 1962.

[107] Pierre Fraisse, Pavol Hell, and David G. Kirkpatrick. A note on f-factors in directed and undirected multigraphs. *Graphs and Combinatorics*, 2(1):61–66, 1986.

[108] Michael L. Fredman, Robert Sedgewick, Daniel D. Sleator, and Robert E. Tarjan. The pairing-heap: A new form of self-adjusting heap. *Algorithmica*, 1(1-4):111–129, 1986.

[109] Michael L. Fredman and Robert E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the Association for Computing Machinery*, 34(3):596–615, 1987.

[110] Christian Fremuth-Paeger and Dieter Jungnickel. Balanced network flows. IV. Duality and structure theory. *Networks*, 37(4):194–201, 2001.

[111] Christian Fremuth-Paeger and Dieter Jungnickel. Balanced network flows. VIII. A revised theory of phase-ordered algorithms and the $\mathcal{O}(\sqrt{n}m \log(n^2/m)/\log n)$ bound for the nonbipartite cardinality matching problem. *Networks*, 41(3):137–142, 2003.

[112] Georg Frobenius. Über zerlegbare Determinanten [On decomposable determinants]. *Sitzungsbericht der Königlich Preussischen Akademie der Wissenschaften*, XVIII:274–277, 1917. In German.

[113] Harold N. Gabow. *Implementation of Algorithms for Maximum Matching on Nonbipartite Graphs.* PhD thesis, Department of Computer Science, Stanford University, Stanford, California, USA, 1973.

[114] Harold N. Gabow. An efficient implementation of Edmonds' algorithm for maximum matching on graphs. *Journal of the Association for Computing Machinery*, 23(2):221–234, 1976.

[115] Harold N. Gabow. An efficient reduction technique for degree-constrained subgraph and bidirected network flow problem. In *Proceedings of the 15th Annual ACM Symposium on Theory of Computing (STOC), 25-27 April, 1983, Boston, Massachusetts, USA*, pages 448–456, 1983.

[116] Harold N. Gabow. A scaling algorithm for weighted matching on general graphs. In *Proceedings of the 26th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 90–100, Portland, Oregon, 1985.

[117] Harold N. Gabow. Scaling algorithms for network problems. *Journal of Computer and System Sciences*, 31(2):148–168, 1985.

[118] Harold N. Gabow. Data structures for weighted matching and nearest common ancestors with linking. In *Proceedings of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 434–443, San Francisco, California, 1990.

[119] Harold N. Gabow. Path-based depth-first search for strong and biconnected components. *Information Processing Letters*, 74(3-4):107–114, 2000.

[120] Harold N. Gabow, Zvi Galil, and Thomas H. Spencer. Efficient implementation of graph algorithms using contraction. *Journal of the Association for Computing Machinery*, 36(3):540–572, 1989.

[121] Harold N. Gabow and Robert E. Tarjan. A linear-time algorithm for a special case of disjoint set union. *Journal of Computer and System Sciences*, 30(2):209–221, 1985.

[122] Harold N. Gabow and Robert E. Tarjan. Almost-optimum speed-ups of algorithms for bipartite matching and related problems. *The Association for Computing Machinery*, pages 514–527, 1988.

[123] Harold N. Gabow and Robert E. Tarjan. Faster scaling algorithms for network problems. *SIAM Journal on Computing*, 18(5):1013–1036, 1989.

[124] Harold N. Gabow and Robert E. Tarjan. Faster scaling algorithms for general graph matching problems. *Journal of the Association for Computing Machinery*, 38(4):815–853, 1991.

[125] Zvi Galil and Giuseppe F. Italiano. Data structures and algorithms for disjoint set union problems. *ACM Computing Surveys*, 23(3):319–344, 1991.

[126] Zvi Galil, Silvio Micali, and Harold N. Gabow. An $\mathcal{O}(EV \log V)$ algorithm for finding a maximal weighted matching in general graphs. *SIAM Journal on Computing*, 15(1):120–130, 1986.

[127] Tibor Gallai. Kritische Graphen II [Critical graphs]. *Magyar Tudományos Akadémia; Matematikai Kutató Intézetének Közleményei*, 8:373–395, 1963. In German.

[128] Tibor Gallai. Maximale Systeme unabhängiger Kanten [Maximal independent edge-systems]. *Magyar Tudományos Akadémia; Matematikai Kutató Intézetének Közleményei*, 9:401–413, 1964. In German.

[129] Bernard A. Galler and Michael J. Fischer. An improved equivalence algorithm. *Communications of the ACM*, 7(5):301–303, 1964.

[130] Gianluca Gallo. An $\mathcal{O}(n \log n)$ algorithm for the convex bipartite matching problem. *Operations Research Letters*, 3(1):31–34, 1984.

[131] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of* NP*-Completeness.* Freeman, San Francisco, California, 1979.

[132] Michael R. Garey and David S. Johnson. Crossing number is NP-complete. *SIAM Journal of Algebraic Discrete Methods*, 4(3):312–316, 1983.

[133] Michael R. Garey, David S. Johnson, and Robert E. Tarjan. The planar Hamiltonian circuit problem is NP-complete. *SIAM Journal on Computing*, 5(4):704–714, 1976.

[134] Ian P. Gent, Ian Miguel, and Peter Nightingale. Generalized arc consistency for the alldifferent constraint: An empirical survey. *Artificial Intelligence*, 172(18):1973–2000, 2008.

[135] Loukas Georgiadis. *Linear-Time Algorithms for Dominators and Related Problems.* PhD thesis, Princeton University, Princeton, USA, 2005.

[136] Loukas Georgiadis and Robert E. Tarjan. Finding dominators revisited: Extended abstract. In *Proceedings of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 869–878, New Orleans, Louisiana, 2004.

[137] Alain Ghouila-Houri. Une condition suffisante d'existence d'un circuit Hamiltonien [A sufficient condition for the existence of a Hamilton cycle]. *Comptes rendus de l'Académie des Sciences*, 25:495–497, 1960. In Latin.

[138] Fred Glover. Maximum matching in a convex bipartite graph. *Naval Research Logistics Quarterly*, 14(3):313–316, 1967.

[139] Andrew V. Goldberg and Alexander V. Karzanov. Maximum skew-symmetric flows. In Paul G. Spirakis, editor, *Algorithms – ESA '95, Third Annual European Symposium, Corfu, Greece, September 25-27, 1995, Proceedings*, volume 979 of *Lecture Notes in Computer Science*, pages 155–170. Springer, 1995.

[140] Andrew V. Goldberg and Alexander V. Karzanov. Maximum skew-symmetric flows and matchings. *Mathematical Programming, Series A*, 100(3):537–568, 2004.

[141] Andrew V. Goldberg and Robert Kennedy. Global price updates help. *SIAM Journal on Discrete Mathematics*, 10(4):551–572, 1997.

[142] Alan J. Goldman. Optimal matchings and degree-constrained subgraphs. *Journal of Research of the National Bureau of Standards*, 68B(1):27–29, 1964.

[143] Michel Gondran and Michel Minoux. *Graphs and Algorithms*. John Wiley & Sons, New York, 1984.

[144] Seymour E. Goodman, Stephen T. Hedetniemi, and Robert E. Tarjan. b-matchings in trees. *SIAM Journal on Computing*, 5(1):104–108, 1976.

[145] Emanuel J. Grinberg. Plane regular graphs of degree three without Hamiltonian circuits. *Latvian Mathematical Yearbook*, 4:51–58, 1968. In Russian. Translated by Dainis Zeps.

[146] Bernhard Haeupler, Siddhartha Sen, and Robert E. Tarjan. Rank-pairing heaps. In Amos Fiat and Peter Sanders, editors, *Algorithms – ESA 2009, 17th Annual European Symposium, Copenhagen, Denmark, September 7-9, 2009. Proceedings*, volume 5757 of *Lecture Notes in Computer Science*, pages 659–670. Springer, 2009.

[147] Bernhard Haeupler, Siddhartha Sen, and Robert E. Tarjan. Rank-pairing heaps. *SIAM Journal on Computing*, 40(6):1463–1485, 2011.

[148] Mahantesh Halappanavar. *Algorithms for vertex-weighted matching in graphs*. PhD thesis, Old Dominion University, Norfolk, Virginia Area, USA, 2009.

[149] Philip Hall. On representatives of subsets. *The Journal of the London Mathematical Society*, 10(1):26–30, 1935.

[150] Frank Harary. *Graph Theory*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1972.

[151] Frank Harary, Robert Z. Norman, and Dorwin Cartwright. *Structural Models: An Introduction to the Theory of Directed Graphs*. John Wiley & Sons, New York, 1965.

[152] Dov Harel. A linear time algorithm for finding dominators in flow graphs and related problems. In Robert Sedgewick, editor, *Proceedings of the 17th Annual ACM Symposium on Theory of Computing (STOC), May 6-8, 1985, Providence, Rhode Island, USA*, pages 185–194, 1985.

[153] David B. Hartvigsen. *Extensions of Matching Theory.* PhD thesis, Department of Mathematics, Carnegie-Mellon University, Pittsburgh, Pensylvania, USA, 1984.

[154] Nicholas J. A. Harvey. Algebraic algorithms for matching and matroid problems. *SIAM Journal on Computing*, 39(2):679–702, 2009.

[155] Katherine Heinrich, Pavol Hell, David G. Kirkpatrick, and Guizhen Liu. A simple existence criterion for $(g < f)$-factors. *Discrete Mathematics*, 85(3):313–317, 1990.

[156] Michael Held and Richard M. Karp. The traveling-salesman problem and minimum spanning trees. *Operations Research*, 18(6):1138–1162, 1970.

[157] Michael Held and Richard M. Karp. The traveling-salesman problem and minimum spanning trees: Part II. *Mathematical Programming*, 1(1):6–25, 1971.

[158] Pavol Hell and David G. Kirkpatrick. Algorithms for degree constrained graph factors of minimum deficiency. *Journal of Algorithms*, 14(1):115–138, 1993.

[159] Pascal van Hentenryck. *Constraint Satisfaction in Logic Programming.* The MIT Press, 1989.

[160] Pascal van Hentenryck and Jean-Philippe Carillon. Generality versus specificity: An experience with AI and OR techniques. In *Proceedings of the 7th National Conference on Artificial Intelligence (AAAI)*, pages 660–664, 1988.

[161] Martin Henz, Tobias Müller, and Sven Thiel. Global constraints for round robin tournament scheduling. *European Journal of Operations Research*, 153(1):92–101, 2004.

[162] Gábor Hetyei. 2 x 1-es téglalapokkal lefedhető idomokról [On rectangular configurations which can be covered by 2 x 1 rectangles]. *A Pécsi Tanárképző Főiskola Tudományos Kőzleményei Seria 6: Matematika*, 8:351–368, 1964. In Hungarian.

[163] Lu HongLiang and Yu QingLin. Constructive proof of deficiency theorem of (g, f)-factor. *Science China Mathematics*, 53(6):1657–1662, 2010.

[164] John E. Hopcroft and Richard M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal on Computing*, 2(4):225–231, 1973.

[165] John E. Hopcroft and Robert E. Tarjan. Algorithm 447: Efficient algorithms for graph manipulation. *Communications of the ACM*, 16(6):372–378, 1973.

[166] John E. Hopcroft and Robert E. Tarjan. Dividing a graph into triconnected components. *SIAM Journal on Computing*, 2(3):135–158, 1973.

[167] John E. Hopcroft and Robert E. Tarjan. Efficient planarity testing. *Journal of the Association for Computing Machinery*, 21(4):549–568, 1974.

[168] John E. Hopcroft and Jeffrey D. Ullman. Set merging algorithms. *SIAM Journal on Computing*, 2(4):294–303, 1973.

[169] Chien-Chung Huang and Telikepalli Kavitha. Efficient algorithms for maximum weight matchings in general graphs with small edge weights. In *Proceedings of the 23rd Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1400–1412, Kyoto, Japan, 2012.

[170] Oscar H. Ibarra and Shlomo Moran. Deterministic and probabilistic algorithms for maximum bipartite matching via fast matrix multiplication. *Information Processing Letters*, 13(1):12–15, 1981.

[171] Yannis Ioannidis, Raghu Ramakrishnan, and Linda Winger. Transitive closure algorithms based on graph traversal. *ACM Transactions on Database Systems (TODS)*, 18(3):512–576, 1993.

[172] Masao Iri. A new method of solving transportation-network problems. *Journal of the Operations Research Society of Japan*, 3:27–87, 1960.

[173] Diane M. Johnson, Andrew L. Dulmage, and Nathan S. Mendelsohn. Connectivity and reducibility of graphs. *Canadian Journal of Mathematics*, 14:529–539, 1962.

[174] Donald B. Johnson. Priority queues with update and finding minimum spanning trees. *Information Processing Letters*, 4(3):53–57, 1975.

[175] Donald B. Johnson. Efficient algorithms for shortest paths in sparse networks. *Journal of the Association for Computing Machinery*, 24(1):1–13, 1977.

[176] Tiko Kameda and J. Ian Munro. An $\mathcal{O}(|V| \cdot |E|)$ algorithm for maximum matching of graphs. *Computing*, 12:91–98, 1974.

[177] Mikio Kano and Gyula Y. Katona. Odd subgraphs and matchings. *Discrete Mathematics*, 250(1):265–272, 2002.

[178] Mikio Kano and Gyula Y. Katona. Structure theorem and algorithm on $(1, f)$-odd subgraphs. *Discrete Mathematics*, 307(11-12):1404–1417, 2007.

[179] Ming-Yang Kao, Tak-Wah Lam, Wing-Kin Sung, and Hing-Fung Ting. A decomposition theorem for maximum weight bipartite matchings with applications to evolutionary trees. In Jaroslav Nešetřil, editor, *Algorithms – ESA '99, 7th Annual European Symposium, Prague, Czech Republic, July 16-18, 1999, Proceedings*, volume 1643 of *Lecture Notes in Computer Science*, pages 438–449. Springer, 1999.

[180] Ming-Yang Kao, Tak-Wah Lam, Wing-Kin Sung, and Hing-Fung Ting. A decomposition theorem for maximum weight bipartite matchings. *SIAM Journal on Computing*, 31(1):18–26, 2001.

[181] Haim Kaplan and Robert E. Tarjan. Thin heaps, thick heaps. *ACM Transactions on Algorithms (TALG)*, 4(1):3:1–3:14, 2008.

[182] Richard M. Karp. Reducibility among combinatorial problems. In Raymond E. Miller and James W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, New York, 1972.

[183] Alexander V. Karzanov. On finding maximum flows in network with special structure and some applications. *Mathematical Problems for Production Control*, 5:81–94, 1973.

[184] Alexander V. Karzanov. Efficient realization of the Edmonds' algorithms for matching maximal power and maximal weight. *Studies in Discrete Optimization*, pages 306–327, 1976.

[185] Panagiotis Katerinis. Some results on the existence of $2n$-factors in terms of vertex-deleted subgraphs. *Ars Combinatoria*, 16-B:271–277, 1983.

[186] Irit Katriel. Matchings in node-weighted convex bipartite graphs. *INFORMS Journal on Computing*, 20(2):205–211, 2008.

[187] Irit Katriel and Sven Thiel. Complete bound consistency for the global cardinality constraint. *Constraints*, 10(3):191–217, 2005.

[188] Latife Genç Kaya and John N. Hooker. A filter for the circuit constraint. In Frédéric Benhamou, editor, *Principles and Practice of Constraint Programming – CP 2006, 12th International Conference, CP 2006, Nantes, France, September 25-29, 2006, Proceedings*, volume 4204 of *Lecture Notes in Computer Science*, pages 706–710. Springer, 2006.

[189] Waldemar Kocjan and Per Kreuger. Filtering methods for symmetric cardinality constraint. In Jean-Charles Régin and Michel Rueher, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, First International Conference, CPAIOR 2004, Nice, France, April 20-22, 2004, Proceedings*, volume 3011 of *Lecture Notes in Computer Science*, pages 200–208. Springer, 2004.

[190] Waldemar Kocjan, Per Kreuger, and Björn Lisper. Symmetric cardinality constraint with costs. Technical report, MRTC, Mälardalen University, 2004.

[191] Dénes König. Über Graphen und ihre Anwendung auf Determinantentheorie und Mengenlehre [On graphs and their applications in determinant theory and set theory]. *Mathematische Annalen*, 77:453–465, 1916. In German.

[192] Dénes König. Graphok és matrixok [Graphs and matrices]. *Matematikai és Fizikai Lapok*, 38:116–119, 1931. In Hungarian with German summary.

[193] Anton Kotzig. On the theory of finite graphs with a linear factor I. *Matematicko-Fyzikálny Časopis Slovenskej Akadémie Vied*, 9(2):73–91, 1959. In Slovak.

[194] Anton Kotzig. On the theory of finite graphs with a linear factor II. *Matematicko-Fyzikálny Časopis Slovenskej Akadémie Vied*, 9(2):136–159, 1959. In Slovak.

[195] Anton Kotzig. On the theory of finite graphs with a linear factor III. *Matematicko-Fyzikálny Časopis Slovenskej Akadémie Vied*, 10(4):205–215, 1960. In Slovak.

[196] Mekkia Kouider and Preben D. Vestergaard. Connected factors in graphs – a survey. *Graphs and Combinatorics*, 21(1):1–26, 2005.

[197] Joseph B. Kruskal. On the shortest spanning subtree and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7(1):48–50, 1956.

[198] Harold W. Kuhn. The Hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, 2:83–97, 1955.

[199] Harold W. Kuhn. Variants of the Hungarian method for assignment problems. *Naval Research Logistics Quarterly*, 3:253–258, 1956.

[200] Jean-Louis Laurière. A language and a program for stating and solving combinatorial problems. *Artificial Intelligence. An International Journal*, 10(1):29–127, 1978.

[201] Eugene L. Lawler. *Combinatorial Optimization: Networks and Matroids.* Holt, Rinehart & Winston, New York, 1976.

[202] Eugene L. Lawler, Jan K. Lenstra, Alexander H. G. Rinnooy Kan, and David B. Shmoys, editors. *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization.* John Wiley & Sons, New York, 1985.

[203] Michel Leconte. A bounds-based reduction scheme for difference constraints. In *Proceedings of the 2nd International Workshop on Constraint-based Reasoning (Constraint-96)*, Key West, Florida, 1996.

[204] Thomas Lengauer and Robert E. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems*, 1(1):121–141, 1979.

[205] Mordechai Lewin. On maximal circuits in directed graphs. *Journal of Combinatorial Theory, Series B*, 18(2):175–179, 1975.

[206] Witold Lipski and Franco P. Preparata. Efficient algorithms for finding maximum matchings in convex bipartite graphs and related problems. *Acta Informatica*, 15(4):329–346, 1981.

[207] Charles H. C. Little, Douglas D. Grant, and Derek A. Holton. On defect d-matchings in graphs. *Discrete Mathematics*, 13(1):41–54, 1975. Erratum: *Discrete Mathematics*, 14(2):203, 1976.

[208] Alejandro López-Ortiz, Claude-Guy Quimper, John Tromp, and Peter van Beek. A fast and simple algorithm for bounds consistency of the alldifferent constraint. In Georg Gottlob and Toby Walsh, editors, *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI), Acapulco, Mexico, August 9-15, 2003*, pages 245–250, 2003.

[209] Xavier Lorca. *Contraintes de Partitionnement de Graphe [Graph Partitioning Constraints]*. PhD thesis, Université de Nantes, Faculté des Sciences et des Techniques, Nantes, France, 2007. In French.

[210] László Lovász. Subgraphs with prescribed valencies. *Journal of Combinatorial Theory*, 8(4):391–416, 1970.

[211] László Lovász. Matching structure and the matching lattice. *Journal of Combinatorial Theory, Series B*, 43(2):187–222, 1987.

[212] László Lovász and Michael D. Plummer. *Matching Theory*. Annals of Discrete Mathematics (29). North-Holland, Amsterdam, 1986.

[213] Edward S. Lowry and Cleburne W. Medlock. Object code optimization. *Communications of the ACM*, 12(1):13–22, 1969.

[214] Yannis Manoussakis. Directed Hamiltonian graphs. *Journal of Graph Theory*, 16(1):51–59, 1992.

[215] Kim Marriott and Peter J. Stuckey. *Programming with Constraints: An Introduction*. The MIT Press, 1998.

[216] Nimrod Megiddo and Arie Tamir. An $\mathcal{O}(N \cdot \log N)$ algorithm for a class of matching problem. *SIAM Journal on Computing*, 7(2):154–157, 1978.

[217] Kurt Mehlhorn. *Data Structures and Algorithms 2: Graphs Algorithms and* NP*-Completeness*. Springer, 1984.

[218] Kurt Mehlhorn and Stefan Näher. *LEDA: A Platform for Combinatorial and Geometric Computing*. Cambridge University Press, Cambridge, 1999.

[219] Kurt Mehlhorn and Sven Thiel. Faster algorithms for bound-consistency of the sortedness and the alldifferent constraint. In Rina Dechter, editor, *Principles and Practice of Constraint Programming – CP 2000, 6th International Conference, CP 2000, Singapore, September 18-21, 2000, Proceedings*, volume 1894 of *Lecture Notes in Computer Science*, pages 306–319. Springer, 2000.

[220] Henk Meijer, Yurai Núñez-Rodríguez, and David Rappaport. An algorithm for computing simple k-factors. *Information Processing Letters*, 109(12):620–625, 2009.

[221] Silvio Micali and Vijay V. Vazirani. An $\mathcal{O}(\sqrt{|V|} \cdot |E|)$ algorithm for finding maximum matching in general graphs. In *Proceedings of the 21st Annual Symposium on Foundations of Computer Science (FOCS)*, pages 17–27, Syracuse, New York, 1980.

[222] Edward Minieka. *Optimization Algorithms for Networks and Graphs*. Marcel Dekker, New York and Basel, 1978.

[223] Rajeev Motwani. Average-case analysis of algorithms for matchings and related problems. *Journal of the Association for Computing Machinery*, 41(6):1329–1356, 1994.

[224] Marcin Mucha and Piotr Sankowski. Maximum matchings via Gaussian elimination. In *Proceedings of the 45th IEEE Symposium on Foundations of Computer Science*, volume 6, pages 248–255, 2004.

[225] Ketan Mulmuley, Umesh V. Vazirani, and Vijay V. Vazirani. Matching is as easy as matrix inversion. *Combinatorica*, 7(1):105–113, 1987.

[226] James Munkres. Algorithms for the assignment and transportation problems. *Journal of the Society for Industrial and Applied Mathematics*, 5(1):32–38, 1957.

[227] J. Ian Munro. Efficient determination of the transitive closure of a directed graph. *Information Processing Letters*, 1(2):56–58, 1971.

[228] Yunsun Nam. *Matching Theory: Subgraphs with Degree Constraints and other Properties*. PhD thesis, Department of Mathematics, University of British Columbia, Canada, 1994.

[229] Manoel Bezerra Campêlo Neto and Sulamita Klein. Maximum vertex-weighted matching in strongly chordal graphs. *Discrete Applied Mathematics*, 84(1-3):71–77, 1998.

[230] Peter Nightingale. The extended global cardinality constraint: An empirical survey. *Artificial Intelligence*, 175(2):586–614, 2008.

[231] Robert Z. Norman and Michael O. Rabin. An algorithm for a minimum cover of a graph. *Proceedings of the American Mathematical Society*, 10(2):315–319, 1959.

[232] Renata Ochranová. Finding dominators. In Marek Karpinski, editor, *Proceedings of the 4th Conference on Foundations of Computation Theory*, volume 158 of *Lecture Notes in Computer Science*, pages 328–334. Springer, 1983.

[233] William J. Older, Godfried M. Swinkels, and Maarten H. van Emden. Getting to the real problem: Experience with BNR prolog in OR. In *3rd International Conference on the Practical Application of Prolog (PAP)*, pages 465–478, Paris, France, 1995.

[234] Oystein Ore. Studies on directed graphs, I. *Annals of Mathematics*, 63(3):383–406, 1956.

[235] Oystein Ore. Studies on directed graphs, II. *Annals of Mathematics*, 64(1):142–153, 1956.

[236] Oystein Ore. Graphs and subgraphs. *Transactions of the American Mathematical Society*, 84:109–136, 1957.

[237] Oystein Ore. Studies on directed graphs, III. *Annals of Mathematics*, 68(3):526–549, 1958.

[238] Oystein Ore. Graphs and subgraphs, II. *Transactions of the American Mathematical Society*, 93:185–204, 1959.

[239] James B. Orlin and Ravindra K. Ahuja. New scaling algorithms for the assignment and minimum mean cycle problems. *Mathematical Programming*, 54(1):41–56, 1992.

[240] François Pachet and Pierre Roy. Automatic generation of music programs. In Joxan Jaffar, editor, *Principles and Practice of Constraint Programming – CP'99, 5th International Conference, Alexandria, Virginia, USA, October 11-14, 1999, Proceedings*, volume 1713 of *Lecture Notes in Computer Science*, pages 331–345. Springer, 1999.

[241] Christos H. Papadimitriou. *Computational Complexity.* Addison-Wesley, 1994.

[242] Christos H. Papadimitriou and Kenneth Steiglitz. *Combinatorial Optimization: Algorithms and Complexity.* Prentice-Hall, Englewood Cliffs, New Jersey, 1982.

[243] Julius Petersen. Die Theorie der regulären Graphs [The theory of regular graphs]. *Acta Mathematica*, 15:193–220, 1891. In German.

[244] Paul A. Peterson and Michael C. Loui. The general maximum matching algorithm of Micali and Vazirani. *Algorithmica*, 3(1-4):511–533, 1988.

[245] Thierry Petit, Jean-Charles Régin, and Christian Bessière. Specific filtering algorithms for over-constrained problems. In Toby Walsh, editor, *Principles and Practice of Constraint Programming – CP 2001, 7th International Conference, CP 2001, Paphos, Cyprus, November 26 - December 1, 2001, Proceedings*, volume 2239 of *Lecture Notes in Computer Science*, pages 451–463. Springer, 2001.

[246] Alex Pothen and Chin-Ju Fan. Computing the block triangular form of a sparse matrix. *ACM Transactions on Mathematical Software*, 16(4):303–324, 1990.

[247] Patrick Prosser and Chris Unsworth. Rooted tree and spanning tree constraints. In *17th ECAI Workshop on Modelling and Solving Problems with Constraints*, 2006.

[248] Jean-François Puget. A fast algorithm for the bound consistency of alldiff constraints. In *Proceedings of the 15th National Conference on Artificial Intelligence and 10th Innovative Applications of Artificial Intelligence Conference (AAAI / IAAI)*, pages 359–366, 1998.

[249] Paul W. Purdom. A transitive closure algorithm. *BIT Numerical Mathematics*, 10(1):76–94, 1970.

[250] Paul W. Purdom and Edward F. Moore. Algorithm 430: Immediate predominators in a directed graph. *Communications of the ACM*, 15(8):777–778, 1972.

[251] Luis Quesada. *Solving Constrained Graph Problems using Reachability Constraints based on Transitive Closure and Dominators.* PhD thesis, Université Catholique de Louvain, Belgium, 2006.

[252] Claude-Guy Quimper. *Efficient Propagators for Global Constraints.* PhD thesis, University of Waterloo, Waterloo, Ontario, Canada, 2006.

[253] Claude-Guy Quimper, Alejandro López-Ortiz, Peter van Beek, and Alexander Golynski. Improved algorithms for the global cardinality constraint. In Mark Wallace, editor, *Principles and Practice of Constraint Programming – CP 2004, 10th International Conference, CP 2004, Toronto, Canada, September 27 - October 1, 2004, Proceedings*, volume 3258 of *Lecture Notes in Computer Science*, pages 542–556. Springer, 2004.

[254] Claude-Guy Quimper, Peter van Beek, Alejandro López-Ortiz, Alexander Golynski, and Sayyed Bashir Sadjad. An efficient bounds consistency algorithm for the global cardinality constraint. *Constraints*, 10(2):115–135, 2005.

[255] Jean-Charles Régin. A filtering algorithm for constraints of difference in CSPs. In *Proceedings of the 12th National Conference on Artificial Intelligence (AAAI)*, pages 362–367, Seattle, Washington, 1994.

[256] Jean-Charles Régin. Generalized arc consistency for global cardinality constraint. In *Proceedings of the 13th National Conference on Artificial Intelligence (AAAI)*, pages 209–215, Portland, Oregon, 1996.

[257] Jean-Charles Régin. The symmetric alldiff constraint. In Thomas Dean, editor, *Proceedings of the 16th International Joint Conference on Artificial Intelligence, IJCAI 99, Stockholm, Sweden, July 31 - August 6, 1999*, pages 420–425, 1999.

[258] Jean-Charles Régin. Cost-based arc consistency for global cardinality constraints. *Constraints*, 7(3-4):387–405, 2002.

[259] Jean-Charles Régin. Using constraint programming to solve the maximum clique problem. In Francesca Rossi, editor, *Principles and Practice of Constraint Programming – CP 2003, 9th International Conference, CP 2003, Kinsale, Ireland, September 29 - October 3, 2003, Proceedings*, volume 2833 of *Lecture Notes in Computer Science*, pages 634–648. Springer, 2003.

[260] Jean-Charles Régin. Simpler and incremental consistency checking and arc consistency filtering algorithms for the weighted spanning tree constraint. In Laurent Perron and Michael A. Trick, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, 5th International Conference, CPAIOR 2008, Paris, France, May 20-23, 2008, Proceedings*, volume 5015 of *Lecture Notes in Computer Science*, pages 233–247. Springer, 2008.

[261] Jean-Charles Régin, Thierry Petit, Christian Bessiere, and Jean-François Puget. An original constraint based approach for solving over constrained problems. In Rina

Dechter, editor, *Principles and Practice of Constraint Programming – CP 2000, 6th International Conference, CP 2000, Singapore, September 18-21, 2000, Proceedings*, volume 1894 of *Lecture Notes in Computer Science*, pages 543–548. Springer, 2000.

[262] Jean-Charles Régin, Louis-Martin Rousseau, Michel Rueher, and Willem-Jan van Hoeve. The weighted spanning tree constraint revisited. In Andrea Lodi, Michela Milano, and Paolo Toth, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems: 7th International Conference, CPAIOR 2010, Bologna, Italy, June 14-18, 2010. Proceedings*, volume 6140 of *Lecture Notes in Computer Science*, pages 287–291. Springer, 2010.

[263] Julia B. Robinson. On the Hamiltonian game (A Traveling Salesman Problem). Research Memorandum RM-303, The RAND Corporation, Santa Monica, California, 5 December 1949.

[264] Alexander Schrijver. *Combinatorial Optimization. Polyhedra and Efficiency*, volume A, B, C. Springer, Berlin, 2003.

[265] Maria G. Scutellà and Gianluca Scevola. A modification of Lipski-Preparata's algorithm for the maximum matching problem on bipartite convex graphs. *Ricerca Operativa*, 46:63–77, 1988.

[266] Robert Sedgewick and Philippe Flajolet. *An Introduction to the Analysis of Algorithms.* Addison-Wesley, 1996.

[267] Meinolf Sellmann. An arc-consistency algorithm for the minimum weight all different constraint. In Pascal Van Hentenryck, editor, *Principles and Practice of Constraint Programming – CP 2002, 8th International Conference, CP 2002, Ithaca, New York, USA, September 9-13, 2002, Proceedings*, volume 2470 of *Lecture Notes in Computer Science*, pages 744–749. Springer, 2002.

[268] Meinolf Sellmann. Cost-based filtering for shorter path constraints. In Francesca Rossi, editor, *Principles and Practice of Constraint Programming – CP 2003, 9th International Conference, CP 2003, Kinsale, Ireland, September 29 - October 3, 2003, Proceedings*, volume 2833 of *Lecture Notes in Computer Science*, pages 694–708. Springer, 2003.

[269] Micha Sharir. A strong connectivity algorithm and its application in data flow analysis. *Computers & Mathematics with Applications*, 7(1):67–72, 1981.

[270] Yoshio Shimamoto. On an extension of the Grinberg theorem. *Journal of Combinatorial Theory, Series B*, 24(2):169–180, 1978.

[271] Jefferey A. Shufelt and Hans J. Berliner. Generating Hamiltonian circuits without backtracking from errors. *Theoretical Computer Science*, 132:347–375, 1994.

[272] Michael Sipser. *Introduction to the Theory of Computation.* PWS, Boston, 1997.

[273] Daniel D. Sleator and Robert E. Tarjan. Self-adjusting heaps. *SIAM Journal on Computing*, 15(1):52–69, 1986.

[274] Thomas H. Spencer and Ernst W. Mayr. Node weighted matching. In Jan Paredaens, editor, *11th International Colloquium on Automata, Languages and Programming, Antwerpen, Belgium, July 16-20, 1984 (EATCS sign). Proceedings*, volume 172 of *Lecture Notes in Computer Science*, pages 454–464.

[275] John T. Stasko and Jeffrey S. Vitter. Pairing heaps: Experiments and analysis. *Communications of the ACM*, 30(3):234–249, 1987.

[276] George Steiner and Julian S. Yeomans. A linear time algorithm for maximum matchings in convex, bipartite graphs. *Computers & Mathematics with Applications*, 31(12):91–96, 1996.

[277] Vahid Tabatabaee, Leonidas Georgiadis, and Leandros Tassiulas. QoS provisioning and tracking fluid policies in input queueing switches. *IEEE/ACM Transactions on Networking*, 9(5):605–617, 2001.

[278] Tadao Takaoka. Theory of trinomial heaps. In Ding-Zhu Du, Peter Eades, Vladimir Estivill-Castro, Xuemin Lin, and Arun Sharma, editors, *Computing and Combinatorics, 6th Annual International Conference, COCOON 2000, Sydney, Australia, July 26-28, 2000, Proceedings*, volume 1858 of *Lecture Notes in Computer Science*, pages 362–372. Springer, 2000.

[279] Tadao Takaoka. Theory of 2-3 heaps. *Discrete Applied Mathematics*, 126(1):115–128, 2003.

[280] Robert E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[281] Robert E. Tarjan. Finding dominators in directed graphs. *SIAM Journal on Computing*, 3(1):62–89, 1974.

[282] Robert E. Tarjan. Testing flow graph reducibility. *Journal of Computer and System Sciences*, 9(3):355–365, 1974.

[283] Robert E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the Association for Computing Machinery*, 22(2):212–225, 1975.

[284] Robert E. Tarjan. Edge-disjoint spanning trees and depth-first search. *Acta Informatica*, 6(2):171–185, 1976.

[285] Robert E. Tarjan. *Data Structures and Network Algorithms.* SIAM Press, Philadelphia, 1983.

[286] Robert E. Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic Discrete Methods*, 6(2):306–318, 1985.

[287] Robert E. Tarjan and Jan van Leeuwen. Worst-case analysis of set union algorithm. *Journal of the Association for Computing Machinery*, 31(2):245–281, 1984.

[288] Sven Thiel. *Efficient Algorithms for Constraint Propagation and for Processing Tree Description.* PhD thesis, Saarlandes University, Germany, 2004.

[289] Nobuaki Tomizawa. On some techniques useful for solution of transportation network problems. *Networks*, 1(2):173–194, 1971.

[290] Michael A. Trick. Integer and constraint programming approaches for round-robin tournament scheduling. In Edmund K. Burke and Patrick De Causmaecker, editors, *Practice and Theory of Automated Timetabling IV, 4th International Conference, PATAT 2002, Gent, Belgium, August 21-23, 2002, Selected Revised Papers*, volume 2740 of *Lecture Notes in Computer Science*, pages 63–77. Springer, 2003.

[291] Edward Tsang. *Foundations of Constraint Satisfaction.* Academic Press, 1993.

[292] William T. Tutte. The 1-factors of oriented graphs. *Proceedings of the American Mathematical Society*, 4(6):922–931, 1953.

[293] William T. Tutte. A short proof of the factor theorem for finite graphs. *Canadian Journal of Mathematics*, 6:347–352, 1954.

[294] William T. Tutte. Graph factors. *Combinatorica*, 1(1):79–97, 1981.

[295] Robert J. Urquhart. *Degree Constrained Subgraphs of Linear Graphs.* PhD thesis, Systems Engineering Laboratory, Department of Electrical Engineering, University of Michigan, Ann Arbor, Michigan, USA, 1967.

[296] Leslie G. Valiant. The complexity of computing the permanent. *Theoretical Computer Science*, 8(2):189–201, 1979.

[297] Leslie G. Valiant. The complexity of enumeration and reliability problems. *SIAM Journal on Computing*, 8(3):410–421, 1979.

[298] Peter van Emde Boas, Rob Kaas, and Erik Zijlstra. Design and implementation of an efficient priority queue. *Mathematical Systems Theory*, 10:99–127, 1977.

[299] Willem-Jan van Hoeve. *Operations Research Techniques in Constraint Programming.* PhD thesis, University Amsterdam, The Netherlands, 2005.

[300] Willem-Jan van Hoeve. The alldifferent constraint: a survey. In *Annual Workshop of the ERCIM Working Group on Constraints*, Prague, Czech Republik, June 2001.

[301] Jan van Leeuwen and Theo van der Weide. Alternative path compression techniques. Technical Report RUU-CS-77-3, Department of Computer Science, University of Utrecht, The Netherlands, 1977.

[302] Vijay V. Vazirani. A theory of alternating paths and blossoms for proving correctness of the $\mathcal{O}(\sqrt{|V|} \cdot |E|)$ general graph maximum matching algorithm. *Combinatorica*, 14(1):71–109, 1994.

[303] Vijay V. Vazirani. An improved definition of blossoms and a simpler proof of the MV matching algorithm. *CoRR*, abs/1210.4594, 2012.

[304] Walter Vogel. Bemerkungen zur Theorie der Matrizen aus Nullen und Einsen [Notes on the theory of matrices of zeros and ones]. *Archiv der Mathematik (Archives of Mathematics)*, 14:139–144, 1963. In German.

[305] Jean Vuillemin. A data structure for manipulating priority queues. *Communications of the ACM*, 21(4):309–315, 1978.

[306] John W. J. Williams. Algorithm 232: Heapsort. *Communications of the ACM*, 7(6):347–348, 1964.

[307] Christoph Witzgall and Charles T. Zahn. Modification of Edmonds' maximum matching algorithm. *Journal of Research of the National Bureau of Standards*, 69B(1-2):91–98, 1965.

[308] Douglas R. Woodall. Sufficient conditions for circuits in graphs. *Proceedings of the London Mathematical Society*, 24(4):739–755, 1972.

[309] Qinglin Roger Yu and Guizhen Liu. *Graph Factors and Matching Extensions.* Springer, 2009.

[310] Qinglin Roger Yu and Zhao Zhang. Extremal properties of $(1, f)$-odd factors in graphs. *Ars Combinatoria*, 84:161–170, 2007.

[311] Cui Yuting and Mikio Kano. Some results on odd factors of graphs. *Journal of Graph Theory*, 12(3):327–333, 1988.

[312] Joseph Zaks. Extending an extension of Grinberg's theorem. *Journal of Combinatorial Theory, Series B*, 32(1):95–98, 1982.

[313] Alessandro Zanarini. *Exploiting Global Constraints for Search and Propagation.* PhD thesis, École Polytechnique de Montréal, Canada, 2010.

[314] Alessandro Zanarini, Michela Milano, and Gilles Pesant. Improved algorithm for the soft global cardinality constraint. In J. Christopher Beck and Barbara M. Smith, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, 3rd International Conference, CPAIOR 2006, Cork, Ireland, May 31 - June 2, 2006, Proceedings*, volume 3990 of *Lecture Notes in Computer Science*, pages 288–299. Springer, 2006.

[315] Jianyang Zhou. A permutation-based approach for solving the job-shop problem. *Constraints*, 2(2):185–213, 1997.

Wissenschaftlicher Werdegang

Persönliche Daten

Name	Radosław Cymer
Geburt	18. Juni 1970 in Łódź (Polen)

Schulische Laufbahn

1977-1985	Grundschule, Łódź
1985-1991	Gymnasium, Łódź (Abitur)

Universitäre Laufbahn

1991-1996	Universität, Łódź (Informatikstudium, Magisterarbeit)
2006-2013	Universität, Hannover (Fern-Promotionsstudium)

Berufliche Laufbahn

1998-2000	Tutor für Studenten der Informatik (Hochschule)
1997-2013	Softwareentwickler, Datenbankadministrator (Wirtschaft)