Robust Solutions for Constraint Satisfaction and Optimisation under Uncertainty

by

Emmanuel Hebrard

Diplôme d'Etudes Approfondies, Universite Montpellier II,

2002

THE UNIVERSITY OF
NEW SOUTH WALES

SYDNEY • AUSTRALIA

A thesis submitted in fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

University of New South Wales

2011

This thesis entitled:
Robust Solutions for Constraint Satisfaction and Optimisation under Uncertainty
written by Emmanuel Hebrard
has been approved for the Department of Computer Science

Supervisor: Prof. Toby Walsh

Signature ____________________ Date ____________________

The final copy of this thesis has been examined by the signatory, and I find that both the content and the form meet acceptable presentation standards of scholarly work in the above mentioned discipline.

Declaration

I hereby declare that this submission is my own work and to the best of my knowledge it contains no materials previously published or written by another person, nor material which to a substantial extent has been accepted for the award of any other degree or diploma at UNSW or any other educational institution, except where due acknowledgement is made in the thesis. Any contribution made to the research by others, with whom I have worked at UNSW or elsewhere, is explicitly acknowledged in the thesis.

I also declare that the intellectual content of this thesis is the product of my own work, except to the extent that assistance from others in the project's design and conception or in style, presentation and linguistic expression is acknowledged. Parts of the dissertation have appeared in the following publications which have been subject to peer review.

1. Emmanuel Hebrard, Brahim Hnich and Toby Walsh. **Super Solutions in Constraint Programming**. *In Proceedings of the 6th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimisation Problems* (CP-AI-OR-2004).

2. Emmanuel Hebrard, Brahim Hnich and Toby Walsh. **Robust Solutions for Constraint Satisfaction and Optimization**. *In Proceedings of the 16th European Conference on Artificial Intlligence* (ECAI-2004).

3. Emmanuel Hebrard, Brahim Hnich and Toby Walsh. **Super CSPs**. *In workshop on Online Constraint Solving: Handling Change and Uncertainty, held alongside the 9th International Conference on Principles and Practice of Constraint Programming* (CP-2003).

Signature ____________________ Date ____________________

Abstract

We develop a framework for finding robust solutions of constraint programs. Our approach is based on the notion of fault tolerance. We formalise this concept within constraint programming, extend it in several dimensions and introduce some algorithms to find robust solutions efficiently.

When applying constraint programming to real world problems we often face uncertainty. Whilst reactive methods merely deal with the consequences of an unexpected change, taking a more proactive approach may guarantee a certain level of robustness. We propose to apply the fault tolerance framework, introduced in [Ginsberg 98], to constraint programming: A robust solution is one such that a small perturbation only requires a small response. We identify, define and classify a number of abstract problems related to stability within constraint satisfaction or optimisation. We propose some efficient and effective algorithms for solving these problems. We then extend this framework by allowing the repairs and perturbations themselves to be constrained. Finally, we assess the practicality of this framework on constraint satisfaction and scheduling problems.

Dedication

A ma filleule Julie,

A mes parents,

Acknowledgements

First and foremost, I would like to thank my supervisor, Toby Walsh, who directed my Ph.D. studies, guided and supported me, and from whom I learnt so much. I acquired from him a great deal of research methodology, learnt many aspects of computer science, (plus a significant amount of applied geography ;) and much more.

It is difficult to quantify how much I am indebted to my co-supervisor Brahim Hnich and D.E.A supervisor Christian Bessiere. The former for the countless discussions we had on the topic of this very thesis and the invaluable advice and support he gave me. The latter for having introduced constraint programming in such clear terms, for his guidance during my D.E.A., and for having introduced me to Toby.

I am also deeply thankful to Zeynep Kiziltan whose thesis was always opened for guidance during all the writing process of the current dissertation.

There would be too many names to cite, so I will simply thank everyone I worked with at 4C in Cork, and in the KRR group in Sydney. Both experiences were priceless.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The thesis defended in this dissertation is that:

> *The concept of fault tolerance can be extended in a number of directions to provide robust solutions to constraint satisfaction and optimisation problems. Finding fault tolerant solutions is significantly more difficult than finding regular solutions. However, efficient and effective algorithms can be designed for that purpose.*

Fault tolerance has been introduced and studied on Boolean satisfiability problem by Roy *et al.* [Ginsberg 98, Roy 98] as a framework for finding robust solutions of combinatorial problems. In this chapter, we introduce and motivate our approach to fault tolerance within constraint satisfaction and optimisation. We briefly introduce constraint satisfaction and optimisation problems in Section 1.1. We motivate our work, discuss the notion of uncertainty and review the previous work in this area in Section 1.2. We introduce our approach to tackling uncertainty in Section 1.3 and review the contributions of this dissertation and outline the subsequent chapters in Section 1.4.

1.1 Constraint Programming

Constraint Programming is a generic framework for solving hard combinatorial problems. The factor that makes CP stand out is its wider and richer language of constraints. In the **Propositional Satisfiability** framework, the problem may be decomposed into a conjunction of disjunctions of Boolean literals. In **Mathematical Integer Programming**, the atomic parts are linear equations. A constraint, on the other hand, is a broader, albeit rather ill defined, notion. Any n-ary relation over a set of values that can be checked in polynomial time can be qualified as constraint. For instance, $(X \leq Y)$, $(X_i \neq X_j \ \ \forall i \neq j)$ or $(|\{v \mid X_i = v \ \ \forall i\}| \leq N)$ are all constraints, however the first is a linear equation of arity 2, the second is not linear, and the third is NP-hard to reason with. In fact, the idea behind constraint programming is that a problem can be partitioned into smaller problems (constraints) with the only requirement that a **propagation** algorithm is readily available for each of them. Consequently, one may see constraint programming as a framework for integrating algorithms in such a way that they can interact efficiently for solving larger problems.

A Constraint Satisfaction Problem (CSP) involves a set of variables, a finite domain of values associated to every variable and a set of constraints, i.e., relations restricting the combinations of values for subsets of variables. A CSP is satisfiable if and only if there is a assignment of values to variables, such that all constraints are satisfied.

Example 1. *For instance, consider the* ***Jobshop Scheduling Problem*** *(*JSP*). In this problem, a set of n jobs need to be scheduled, using a set of m resources. A job is a sequence of m activities of given durations, along with a relation mapping each activity in the job with a resource. In Figure 1.1, we illustrate a* JSP *involving* 3 *jobs and* 3 *resources. Each row corresponds to a job, and an activity is represented as a rectangle whose length corresponds to its duration and whose colour corresponds to the resource it uses.*

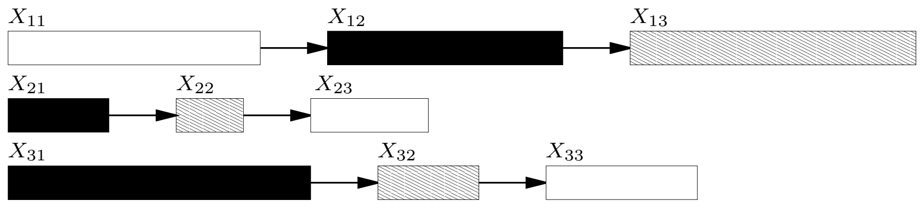


Figure 1.1: An instance of the Jobshop Scheduling Problem.

The activities are to be scheduled, that is, a time point needs to be allocated to every activity such that the following constraints hold:

- *Within a job, the sequence of activities must be respected.*

- *Two activities sharing a resource must not overlap in time.*

- *The schedule must fit into a given time windows (makespan).*

The instance of JSP *represented in Figure 1.1 can be modelled and solved through constraint programming. We can, for instance, associate a variable to each activity, that is, X_{ij} stands for the j^{th} activity of the i^{th} job. These variables will take their values into a discretised set of time points. To ensure that the sequence within each job is respected we introduce a set of temporal precedence constraints for any job i:*

$$\forall j \in [1..m-1],\ X_{ij} + \text{Duration}(X_{ij}) \leq X_{ij+1}$$

Next, for any pair X_{ij}, X_{kl} of activities sharing a resource, we have a disjunctive constraint ensuring that they do not overlap:

$$(X_{ij} + \text{Duration}(X_{ij}) \leq X_{kl}) \vee (X_{kl} + \text{Duration}(X_{kl}) \leq X_{ij})$$

Finally, we restrict the domains to ensure that the deadline D is met.

$$\forall i, j\ X_{ij} + \text{Duration}(X_{ij}) \leq D$$

We give an example of valid schedule in Figure 1.2. Notice that the makespan is minimal, that is, there is no shorter feasible schedule for these jobs.

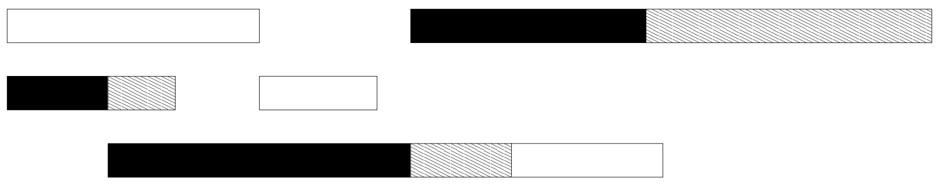


Figure 1.2: A solution of the problem illustrated in Figure 1.1.

1.2 Uncertainty

When dealing with real world problems, efficiency and optimality may not be the only concerns. For instance, the most tightly optimised schedule may not be the best one if a single activity being delayed causes all other activities to stand idle. Moreover, in many cases it is difficult or even impossible to obtain a perfect matching between the real situation being modelled and the mathematical model. The real problem may simply be too complex, and no perfectly accurate model can be produced; some data may be unavailable or erroneous; the environment may simply be inherently dynamic, hence the model cannot easily capture more than a given fixed state, etc. In this dissertation we focus on the latter situation, however all these cases, from the modelling and solving viewpoint, will result in a gap between the mathematical model and the actual state of the environment when the solution is deployed. In the problems we shall consider, the state of the environment is not known accurately while solving or is prone to change. Uncertainty is ubiquitous in constraint programming application domains. Scheduling and planning problem are, by essence, affected by such changes. In a job-shop scheduling problem, for instance, the duration of processing an activity, or the availability of a resource at a time t may change, because of an unexpected event. A temporary power shortage may result in a activity being delayed, or in a machine being unavailable for a period of time. In a design problem the actual specifications and feature of the object to design may evolve during the development process; in a packing problem, a new item may need to be packed, or its dimension may happen to be different than first specified; more generally, preferences might evolve over time.

A number of methods have been proposed to tackle uncertainty within constraint programming (see [Verfaillie 05] for a survey). A common approach is to look proactively for a **robust** solution. The concept of robustness can be given many characterisations. For instance, a solution is usually seen as robust if future changes are unlikely to affect it, that is, this solution is able to undergo perturbation without being invalidated. We shall refer to this type of robustness as **reliability**. However reliability is not the only valuable property for a solution to have. We explore a second viewpoint on the solution robustness, and we refer to this property as **stability**. For instance, in a trip planning problem, we do not want to change the plan dramatically because of an unexpected perturbation along the trip. If the plan is not stable. a train running late could entail a chain of delays, resulting in a missed plane and a complete replanning. Intuitively, a stable plan is resilient to such situations in the sense that a small perturbation, such as a delay, can be dealt with by a proportionally small change in the solution. The notion of **fault tolerance** has been introduced in the propositional satisfiability framework by

Roy, Ginsberg and Parkes [Ginsberg 98]. A solution is fault tolerant if a small change in the model is guaranteed not to result in a large perturbation of the solution.

1.2.1 Related Work

We now give a brief overview of past and current work on uncertainty in constraint programming and scheduling. We list and comment on some approaches to uncertainty and subsequently propose an informal classification into which we can locate the framework developed in this dissertation. Notice that this list is in no way exhaustive. We direct the interested reader to more detailed surveys by Verfaillie and Jussien [Verfaillie 05], and Davenport and Beck [Davenport 00]. The former addresses constraint programming techniques in the context of dynamic and uncertain environment whilst the latter is focused on scheduling problems.

Flexible Constraint Satisfaction Problems: There are several incentives for introducing degrees of violation of constraints, and in general, more continuous outcomes than the usual satisfiability/unsatisfiability. Partial CSP ([Freuder 89, Freuder 92]) much like the MaxSAT problem, deals with overconstrainedness by relaxing the constraints, the goal being to satisfy most of them. Subsequent frameworks, such as Probabilistic CSP [Schiex 92], fuzzy CSP [Dubois 93], valued CSP [Schiex 95] and semiring-based CSP [Bistarelli 95, Bistarelli 99] have been proposed to deal with preferences and overconstrainedness but can also been used to deal with uncertainty. Indeed, consider a problem subject to uncertainty, that is, where contingent external events might invalidate a solution. Such an event can be seen as a constraint that may or may not hold. The idea is thus to express uncertain constraints as soft constraints. They do not need to be satisfied, but satisfying them is an extra security. A solution that satisfies many such soft constraints is therefore more robust to changes.

Example 2. *Consider for instance a* JSP *where the durations of activities are not known with certainty. We can model this situation using a valued* CSP. *In this framework, each constraint is associated with a totally ordered set of valuations that provides a gradual notion of consistency. The valuations are aggregated using an operation such as max, $\sum$ or $\prod$, for instance. Let $p_i(d)$ be the probability that the duration of an* *activity X_i exceeds d. We can replace the following precedence constraint:*

$$X_i + \text{Duration}(X_i) \leq X_j$$

with a valuation τ over the combinations of values (v_i, v_j) for X_i and X_j:

$$\tau(v_i, v_j) = p_i(v_j - v_i)$$

Similarly the disjunctive constraints may be replaced with the following valuation σ:

$$\sigma(v_i, v_j) = min(p_i(v_j - v_i), p_j(v_i - v_j))$$

The valuations shall thus be aggregated using the product operation, so that the total cost of a solution represents the probability that two activities overlap in the schedule. An optimal solution therefore minimises this probability, hence it is robust.

Dynamic Constraint Satisfaction Problem: The concept of Dynamic Constraint Satisfaction has been introduced in [Dechter 88] to model belief revision in a Belief Maintenance System (BMS). A Dynamic Constraint Network (DCN) is defined by Dechter as follows:

> *A Dynamic Constraint Network (DCN) is a sequence of static CNs each resulting from a change to the preceding one, representing new facts about the environment being modelled.[...]*
> page 3, [Dechter 88].

We can partition the past work on dynamic constraint satisfaction into the following three lines of research:

- Solution and nogood reuse: The idea here is that given a sequence $\{\mathcal{P}_1, \mathcal{P}_2, \ldots \mathcal{P}_k\}$ of constraint networks, a solution for $\mathcal{P}_i$ can be used to help search for a solution to $\mathcal{P}_{i+1}$. Indeed the assumption is that since the change between two constraint networks is relatively small, a solution of $\mathcal{P}_{i+1}$ should be relatively close to a solution of $\mathcal{P}_i$. Similarly, when a part of $\mathcal{P}_i$ is found to be inconsistent, we know that the same hold for cn_{i+1}, providing that the change between cn_i and cn_{i+1} was an addition of constraints. Examples of such reasoning can be found in [Verfaillie 94] and [Schiex 94].

- Incremental arc consistency: Here the situation is a little different as the assumption is that a change, i.e., a jump from $\mathcal{P}_i$ to $\mathcal{P}_{i+1}$ can happen at any time **while** solving $\mathcal{P}_i$. Therefore search algorithms need to be adapted to accommodate such changes. For instance, in [Bessiere 91] the well known algorithm Maintain Arc Consistency (MAC) is modified in such a way that the inference step does not need to be restarted from scratch when a change arises.

- Minimal perturbation: Here the idea is that the solution for $\mathcal{P}_{i+1}$ should be as close as possible to the solution for $\mathcal{P}_i$. The minimal perturbation problem has been studied in a scheduling setting [Sakkout 98, Sakkout 00] as well as in constraint programming [Ran 02, Barták 03]. This is motivated by the same goal as this dissertation: achieving stability. However, this viewpoint is purely reactive whilst ours is purely proactive.

The framework we develop in this dissertation can be seen as proactively addressing the problem of reducing the perturbation caused by an external event, that is, the transition between two consecutive constraint networks in a DCN.

Observe that representations of uncertain problems as flexible CSPs or as dynamic CSPs are not mutually exclusive. In [Miguel 01] and [Miguel 03], it is shown that both frameworks can be applied in conjunction and an integrated algorithm that handles soft constraints while reusing past reasoning in a dynamic setting is proposed.

Stochastic Constraint Satisfaction Problem: In classical constraint programming, variables correspond to decisions. However, when modelling problem involving uncertainty, we may need variables that correspond to the possible states of objects that we cannot control. In the **Mixed Constraint Satisfaction problem** [Fargier 93, Fargier 96], some of the variables are controllable and other are not controllable. The latter models the environment and its uncertainties whilst the former are classical decision variables. The goal is to find a solution maximising the probability of satisfying the environmental constraints. Similar frameworks have been introduced for Boolean satisfiability [Littman 01] and constraint satisfaction [Walsh 02, Manandhar 03]. These frameworks also involve decision and state variables, however the probabilities are linked to the values of state variables.

In the **Branching Constraint Satisfaction Problem** [Fowler 00, Fowler 03], the future is only partially known. One must therefore make as robust as possible decisions with respect to future events. Similarly, in the **Open Constraint Satisfaction Problem** [Lamma 99, Faltings 02], the constraint network is not completely known in advance. However, the motivation is different. Whereas, in a branching constraint satisfaction problem we want to maximise the chance of eventually solving the CSP when the events unfold, in an open problem some partially known aspects, such as values, may relax the current problem, while some other, like constraints, may tighten it.

1.2.2 A Classification of Robustness

We have seen that the notion of robustness can be given many interpretations. Moreover, within the context defined above there might be restrictions on how robustness may be achieved. For instance, repairs may not be allowed and once a solution is computed, it must be executed without change. In this case, only a proactive approach can reduce the impact of the uncertainty. On the other hand, there are situations where nothing is known about what is prone to change or break, and therefore a reactive approach is likely to perform better. In order to put our approach for tackling uncertainty into context with respect to previous work, we consider three important dimensions across which such frameworks can be characterised. This view is certainly not comprehensive and we leave many dimensions uncovered.

Stability vs. Reliability: Within the framework described earlier there are mainly two ways for evaluating the robustness of a solution to a CSP. We will refer to the first and maybe most intuitive definition as **reliability**, that is the ability to remain a solution as often as possible, even in the event of a change in the model.

> A solution is more reliable than another if and only if it has a greater probability to remain a solution after a change.

The second notion, although widespread is perhaps less intuitive. The **stability** of a solution is its ability to be affected as little as possible by a change. This gives way to a more ambiguous definition as one should first define what does “affected” mean. One accepted definition, which we shall use in the subsequent chapters, is the discrepancy between the solution and its replacement after the change. However, there is again a number of definitions that can be better adapted to the problem dealt with, such as the loss in optimality, the CPU-time required to compute the alternative, and the number of new resources consumed. We simply define the stability of a solution as its proximity to an alternative, leaving the definition of the distance open for now.

> A solution f is more stable than another solution g if and only if, in the event of a change, a closer alternative to f than to g exists.

As observed in [Verfaillie 05], these two properties are not mutually exclusive, and a sensible approach could be to combine both views.

Example 3. *We illustrate the concepts or **reliability** and **stability** using the instance of* JSP *in Figure 1.1.*

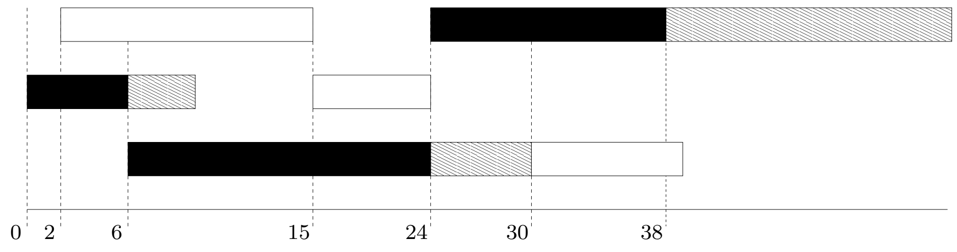


Figure 1.3: A feasible schedule (f) for the JSP illustrated in Figure 1.1.

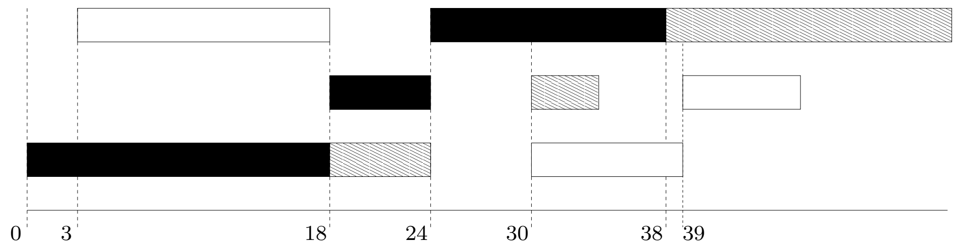


Figure 1.4: A feasible schedule (g) for the JSP illustrated in Figure 1.1.

We suppose now that the activities have uncertain release dates and that the probability of an event precluding the release of an activity at time t *is a linear increasing function of* t*. Notice that we assume these unexpected events to be independent, that is, a perturbation of* t *does not entail a perturbation of* $t-1$*. We give in Figures 1.3 and 1.4 two solutions with equal (optimal) makespan. Clearly, since the mean start time of the activities in the first solution (Schedule* f*, Figure 1.3) is less than in the second* *solution (Schedule g, Figure 1.4), the probability that Schedule f will be executed exactly as planned is larger than for Schedule g. In this context, one may conclude that Schedule f is more **reliable** than Schedule g.*

However, suppose now that an unexpected event occurs at time 0, and as a result, the activity being processed at that time must be postponed (the other release dates remain unaffected). In Schedule f, the only way to respond to this event is to postpone the next activity, resulting in a chain of postponed activities, as illustrated in Figure 1.5. However, consider Schedule g. There is enough space to swap this activity with the

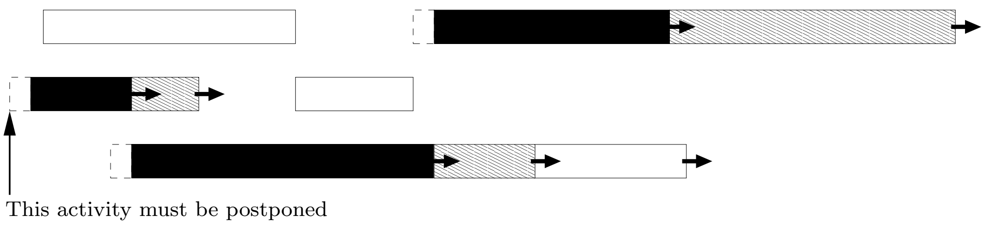


Figure 1.5: The consequences of postponing an activity in Schedule f.

next one using the same resource (that is, the next activity with the same colour). A repaired schedule where the resulting perturbation is bounded (only two other activities moved) is illustrated in Figure 1.6. Notice that in fact, for any small postponing of

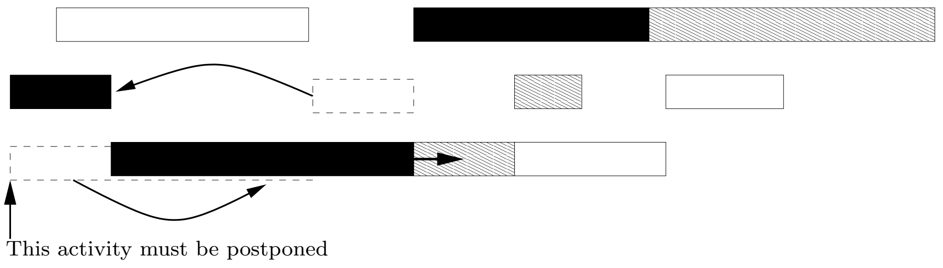


Figure 1.6: The consequences of postponing an activity in Schedule g.

*a single activity in Schedule g, no more than two activities need to be rescheduled in response (the makespan will increase in only three cases). One can therefore conclude that Schedule g is more **stable** than Schedule f.*

Proactivity vs. Reactivity: Methods for dealing with changes are traditionally labelled as proactive or reactive. Proactive methods are preferred choices when the solving phase is off-line. In this case, a certain amount of time can be used to find a solution with intrinsic robustness. The aim is to reduce the amount of work during the repair phase, or even avoid it altogether. Reactive methods, on the other hand are preferred when the changes are completely unknown, when their consequences are too extreme to be dealt with proactively or when not enough computational time is available during the solving phase. Here again, both approaches are clearly not fighting each other, but rather act in synergy. Indeed an intrinsically robust solution should require less work during the repair phase, and a good reactive method can make up for a weakness that was not covered off-line.

Knowledge Intensity: One important criterion to take into account when considering a method for tackling uncertainty in a problem is its reliance upon knowledge about the environment or expertise in the domain. Ideally, no additional knowledge over the data used to build the classical constraint network is required and no more expertise than for solving the problem without taking uncertainty into account. For instance, some approaches might require a list of the possible changes and an associated probability distribution. Other methods might be specific to an application domain and useless outside this domain. Most often a finite universe of possible changes must be provided. However, a fully reactive approach, for instance, may not require any specific knowledge. For instance, the JSP with uncertain release dates, described in Example 3 can be modelled as a valued CSP where time points are attached a weight corresponding to the probability of failure. However, in order to do so, one needs to know the probability distribution at the modelling stage.

Other Dimensions: Clearly we do not cover all attributes. One may partition methods on their ability to deal with a single change as opposed to a set of changes in parallel, or a sequence of small changes. Alternatively, one may insist on the difference between uncertainty coming from incorrect or partially known data, as opposed to external events. Another dimension is the ability to deal with optimisation, and the tradeoff between robustness and optimality thereof.

1.3 Fault Tolerance

We can observe that certain of these attributes promote or sometimes hinder other attributes. For instance, it seems difficult for such a framework to be at the same time

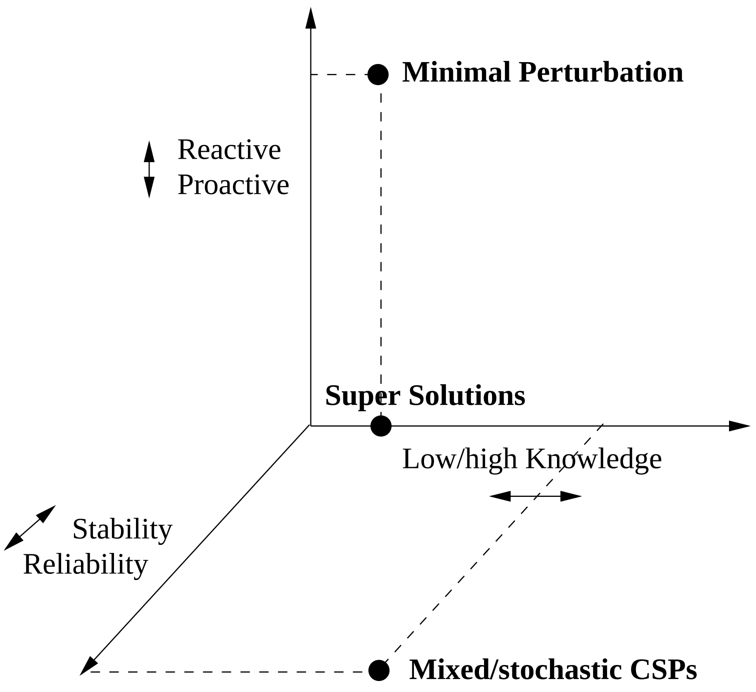


Figure 1.7: Three approaches to robustness and their respective position in the space of possible methods.

proactive and to require little knowledge about the environment. On the other hand, a purely reactive method can only improve the stability i.e., minimise the effect of an unexpected event but cannot improve the reliability which is intrinsically an attribute of the initial solution. The notion of **fault tolerance** developed in this dissertation is proactive whilst requiring very little or no additional knowledge. This notion does not improve directly the reliability, but the stability of a solution, by guaranteeing the existence of nearby alternative solutions in case of a small change. For instance, consider a warehouse allocation problem where a number of shops need to be supplied by a set of warehouses. Certain warehouses do not contain the necessary goods for some shops and the cost of supplying one shop depends on the geographical location of the warehouse and of the shop outlet, on the transportation system and so on. The problem is to find an allocation of warehouses to shops, respecting the capacity of the warehouses such that the cost stays within a given limit. When deploying a solution computed off-line, it may be the case that a warehouse is temporarily unable to supply one or several shops. That may happen because the good has not been produced on time, or perhaps the transportation is at fault. Suppose that the solution computed during the

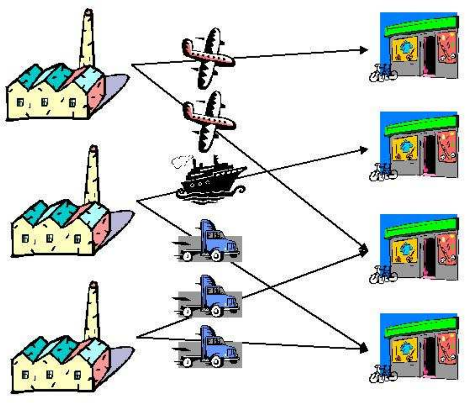


Figure 1.8: The Warehouse Allocation Problem.

solving phase involves a warehouse-to-shop assignment that is no longer valid. Another solution satisfying the new constraint needs thus to be computed online to replace the old solution. However, one might want this new solution to be as close as possible to the old solution. Indeed it might be impractical to change too many routes on short notice. A solution where a small perturbation, such as one warehouse unable to deliver a good to a shop, can be repaired with a suitably small further alteration, such as swapping a few assignments, is said to be fault tolerant. The notion of *super*-models [Ginsberg 98] or δmodels [Roy 98] has first been defined for Boolean satisfiability problems by Roy, Ginsberg and Parkes to exploit this idea. We quote Roy's Ph.D. thesis:

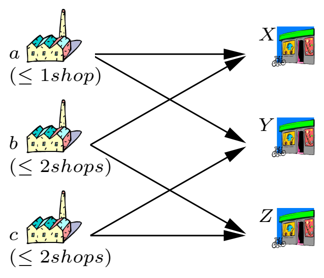
> *To model this concept of fault tolerance we introduce the notion of δmodels: these are satisfying assignments of Boolean formula for which any small alteration, such as a single bit flip, can be repaired by another small alteration, yielding a nearby satisfying assignment.*
> Preamble, page 5, [Roy 98].

We generalise this definition to constraint satisfaction on non-Boolean domains. There are several definitions suitable for integer domains that collapse to Roy's definition on Boolean variables. We shall formally define and study two of these definitions in Chapter 3. We give here a less formal definition and illustrate it through examples.

Definition 1. *A (a, b)-super-solution is a solution in which any set of* a *or fewer variables can be assigned different values, providing that at most* b *other variables are also reassigned.*

This definition is very close to Roy's. However, observe that in our case, when a variable loses the value it is assigned, the *existence* of an alternative (along with the repairs) is required. Another way would be to require *all* values in the domain to be a valid alternative (here again with extra repairs).

Example 4. *Consider the instance of the warehouse allocation problem illustrated in Figure 1.9a, This instance involves 3 shops X, Y, Z and three warehouses a, b, c. We suppose that all costs are identical, and that warehouse b and c can supply at most two shops whilst warehouse a cannot supply more than one shop. Moreover, the goods required by the shop X are not supplied by warehouse c, and the same is true for the pairs Y-b and Z-a. We list all solutions satisfying the constraints in Figure 1.9b.*



(a) Instance

$$\langle X = a, Y = c, Z = b\rangle$$
$$\langle X = a, Y = c, Z = c\rangle$$
$$\langle X = b, Y = a, Z = b\rangle$$
$$\langle X = b, Y = a, Z = c\rangle$$
$$\langle \underline{X = b}, \underline{Y = c}, \underline{Z = b}\rangle$$
$$\langle X = b, Y = c, Z = c\rangle$$

(b) Solutions

Figure 1.9: An instance of the Warehouse Allocation Problem.

Consider the solution $f = \langle X = a, Y = c, Z = b\rangle$, and suppose that, for some reason, warehouse c cannot supply shop Y anymore. The only alternative for supplying shop Y is warehouse a. However, warehouse a already supplies the shop X and can only supply one shop. Therefore we not only need to replace the supplier for Y, but also for X. A valid alternative would then be $\langle X = b, Y = a, Z = b\rangle$. On the other hand, the solution $g = \langle X = b, Y = c, Z = b\rangle$ is more stable. Indeed $X = b$ can be replaced with $X = a$, $Y = c$ can be replaced with $Y = a$ and finally $Z = b$ can be replaced with $Z = c$, and this without any further change. The latter therefore is a $(1, 0)$-super-solution whilst the former is only a $(1, 1)$-super-solution.

Therefore, by choosing a $(1, 0)$-*super*-solution over other solutions, it is possible at the outset to guarantee that we can deal with any perturbation on a single route without stepping on other assignments, which is often a valuable property.

1.4 Contributions and Outline

This dissertation contributes toward better understanding, modelling and solving problems subject to uncertainty.

- We extend the definition of *super*-model to the constraint satisfaction and optimisation framework and define a number of problems related to this definition. Moreover, we further extend the definition to cover a larger range of problems with a better accuracy for modelling breakages and repairs.
- We analyse the complexity of a number of decision and optimisation problems related to finding *super*-solutions of constraint networks. We show that finding *super*-solutions is NP-hard on several tractable classes of constraint networks.
- We introduce the first algorithms for finding *super*-solutions and for finding solution with optimal stability.

This dissertation is organised as follows:

Chapter 2: Formal Background We introduce the technical background necessary to the development of the subsequent chapters. We give some basics of computational complexity theory. We briefly present the constraint satisfaction and optimisation problems as well as backtracking algorithms and the concepts of search and consistency. Finally we introduce some notations and analysis methods used later in this dissertation.

Chapter 3: Definitions and Complexity We formally define the concept of *super*-solution as well as a set of related problems. Then we analyse the computational complexity of these problems. Finally we study tractable classes of CSPs and the complexity of finding *super*-solutions on these problems.

Chapter 4: Full Fault Tolerant Solutions We study a particular case of *super*-solutions where no repair at all is allowed in response to a breakage. We introduce three new methods and recall a former approach for this problem. One of these methods is a reformulation whilst the other two rely on local consistency properties. We also investigate the design of propagation algorithms for enforcing this novel type of local consistency on some global constraints. Then we theoretically compare the filtering power and computational complexity of these four algorithms.

Chapter 5: Weak Fault Tolerant Solutions We introduce an algorithm for the general case, where the size of the breakages and repairs are not fixed. We first present a naive algorithm which dynamically creates sub-problems for checking that breakages are repairable. We subsequently show how we can make inference on the resolution of a sub-problem in order to reduce the overall search space.

Chapter 6: Maximally Fault Tolerant Solutions We study some ways of maximising the repairability of a solution. We introduce a set of Branch & Bound type algorithms for the problem of minimising the maximum size of a repair, and for the problem of maximising the number of breakages admitting a repair. These algorithms are extensions of the algorithms introduced for the satisfaction case.

Chapter 7: Extensions to the Framework We show that the algorithm introduced for the general case can handle more complex definitions of breakages and repairs. We introduce the notion of a constraint for controlling the breakages and another constraint for controlling the repairs. Finally we analyse the concept of symmetry breaking within the *super*-solution framework. We show that symmetry breaking can be used. However, it must be used in a slightly different way than for regular CSPs.

Chapter 8: Applications and Experimental Results We empirically investigate three main issues related to *super*-solutions. First we measure the increase in computational complexity when searching *super*-solutions rather than regular solutions. Second we compare the algorithms introduced in preceding chapters and assess the significance of the inference methods introduced. Finally we study the more practical aspect of this framework: We use the Jobshop Scheduling Problem to assess the tradeoff between computational effort and robustness as well as solution quality against robustness.

Chapter 9: Conclusion and Future Work We conclude the dissertation and discuss limitations and future work.

Chapter 2

Formal Background

2.1 Introduction

In this chapter we recall the formal background and introduce the notations used in this dissertation. We first recall some basic notions of complexity in Section 2.2. The problems we tackle belong either to P, NP or to the NP optimisation complexity class (P^{NP}). In Section 2.3 we introduce the constraint satisfaction and optimisation problems. Then in Section 2.4 we define some concepts central to constraint programming such as search and consistency processing that we use extensively in subsequent chapters. Finally, we introduce some conventions and notations used throughout the paper.

2.2 Worst Case Complexity

A problem is in the class **P**, standing for **Polynomial-time**, if there exists a deterministic Turing machine deciding this problem in a number of steps polynomial in the input size. A problem is in the class **NP**, standing for **Non-deterministic Polynomial-time**, if it is verifiable in polynomial time by a non-deterministic Turing machine. A non-deterministic Turing machine is a "parallel" Turing machine that can take many independent computational paths simultaneously.

A problem is NP-hard if it is at least as hard than all problems in NP. To prove hardness we use polynomial-time reductions. Given two problems P and Q, a **Turing reduction** from P to Q is a program computable by an **oracle machine**, i.e., a Turing machine with an oracle for deciding Q in one step, that can compute P. Intuitively, if

Q is easy then we can easily solve P, hence Q is at least as hard P. Turing reductions are often subject to additional restrictions, for example that the oracle machine runs in polynomial time (polynomial Turing reduction, or Cook reduction). Moreover, since the class NP is not closed under Turing reduction, **many-one** reductions are often used instead, where the additional restriction is that the oracle can only be called once, and at the end of the computation. By transitivity, a problem is NP-hard if it is at least as hard than another NP-hard problem. A problem which is both in NP and NP-hard is called NP-complete.

The problems tackled in this dissertation will either be in P, NP or in P^{NP}. The class P^{NP} (also written $\Delta_2\text{P}$) contains all problems P polynomially Turing reducible to a problem Q in NP. We also classify problems in $\text{P}^{\text{NP}[log(n)]}$ where the Turing reduction uses an oracle machine that runs in logarithmic time. As these problems are often functional, the actual complexity classes for optimisation problems used in this dissertation are FP^{NP} and $\text{FP}^{\text{NP}[log(n)]}$.

2.3 Constraint Satisfaction and Optimisation

2.3.1 Constraint Satisfaction Problem

We first define the notion of **constraint network** and we define the **constraint satisfaction problem** (CSP) as deciding if a constraint network accepts a solution.

Definition 2. *A constraint network is a triplet $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ whose components are:*

- *A totally ordered set of **variables** $\mathcal{X} = \{X_1, \dots X_n\}$.*
- *A mapping $\mathcal{D} : \mathcal{X} \mapsto 2^{\Lambda}$ associating each variable X_i to a finite **domain** $\mathcal{D}(X_i) \subseteq \Lambda$, where Λ is a set of **values**. Typically Λ is a subset of $\mathbb{Z}$.*
- *A set of **constraints** $\mathcal{C} = \{C_1(V_1), \dots C_m(V_m)\}$. A constraint $C(V) \in \mathcal{C}$ is a subset of mappings from a set of variables V to Λ: $C(V) \subseteq \{f \mid f : V \mapsto \Lambda\}$.*

Constraints can be defined either in intention, that is, as a predicate or in extension, that is as a list of **tuples**. Given a constraint $C(X_{i_1}, \dots X_{i_k})$, a **tuple** $\tau \in \Lambda^{|V|}$ is identified to the function mapping $\tau : X_{i_j} \rightarrow \tau[j]$. The tuple τ **satisfies** a constraint $C(V)$, if and only if τ is an element of the associated relation, written $\tau \in C(V)$. We impose the same restrictions as defined in [Bessiere 03] on the predicates used to define constraints in intention:

- Given a tuple τ, the proposition "$\tau \in C(V)$" can be decided by an algorithm polynomial in the size of τ.
- The relation $\tau \in C(V)$ is defined over $\Lambda^{|V|}$ independently of the actual domains of the variables in V.

By convention, we will use upper case for variables and constraints, lower case for values, and we will index variables from 1 to n, constraints from 1 to m and values from 1 to d when possible. Given a subset of variables $E \subseteq \mathcal{X}$, an **assignment** of E is a mapping $f : E \mapsto \Lambda$ such that, for any variable $X \in E$, $f(X) \in \mathcal{D}(X)$. A unary assignment $f : X \rightarrow v$ is often denoted $X = v$. The restriction of an assignment $f : E \mapsto \Lambda$ to a subset F of E, denoted $f|_F$, is equal to f on F ($f(F) = f|_F(F)$), and is not defined otherwise. Assignments on disjoint domains can be composed by a simple union. Let $F, G \subseteq \mathcal{X}$ be two disjoint sets of variables, and $f : F \mapsto \Lambda$ and $g : G \mapsto \Lambda$ be two assignments for these sets. We define the union of f and g as the mapping $f \cup g : F \cup G \mapsto \Lambda$ such that $(f \cup g)|_F = f$ and $(f \cup g)|_G = g$. An assignment $f : E \mapsto \Lambda$ is **consistent** if and only if it satisfies all the constraints over E, i.e., for all $C(V) \in \mathcal{C}$, if $V \subseteq E$ then $f|_V \in C(V)$. A **solution** is a consistent assignment whose domain (in the analytical sense) is $\mathcal{X}$, and the set containing all solutions of a constraint network $\mathcal{P}$ is referred to as *sol*($\mathcal{P}$). A **partial solution** is defined as a mapping from variables to subsets of values included in their domains:

$$\varphi : \mathcal{X} \mapsto 2^{\Lambda} \text{ s.t. } \varphi(X) \subseteq \mathcal{D}(X)\ \forall X \in \mathcal{X}$$

Such a partial solution defines an assignment $f : \mathcal{A} \mapsto \Lambda$, where $X \in \mathcal{A}$ if and only if $\varphi(X) = \{f(X)\}$. We shall often identify the partial solution and the corresponding assignment. The problem of deciding if a constraint network has a solution is known as the **Constraint Satisfaction Problem** (CSP):

CSP
Instance. A constraint network $\mathcal{P}$.
Question. Does $\mathcal{P}$ accept a solution?

The graph $G = (\mathcal{X}, \{V \mid C(V) \in \mathcal{C}\})$ is referred to as the **constraint graph**. Notice that it is an hypergraph unless all constraints are binary i.e., $\forall C(V) \in \mathcal{C},\ |V| = 2$. We introduce the notion of **distance** between assignments. The distance $\Delta_A(f, g)$ between two assignments over a set of variables A is defined as the number of discrepancies or Hamming distance extended to non-Boolean domains:

$$\Delta_A(f, g) = |\{i \mid X_i \in A \ \wedge \ f(X_i) \neq g(X_i)\}|$$

We shall often use $\Delta(f, g)$ instead of $\Delta_{\mathcal{X}}(f, g)$.

2.3.2 Propositional Satisfiability

For sake of simplicity, we shall consider the propositional satisfiability problem (SAT) as a CSP on a Boolean constraint network. A solution of a SAT formula, referred to as a **model** in this particular case, is a mapping $\alpha : \mathcal{X} \mapsto \{0, 1\}$. Moreover, a SAT formula is often given in **Conjunctive Normal Form** (CNF), i.e., a conjunction of clauses. A clause is a disjunction of literals; it therefore forbids exactly one combination of assignments for the involved variables.

2.3.3 Constraint Satisfaction and Optimisation Problem

We now define the **constraint satisfaction and optimisation problem** (CSOP). Notice that we choose here to define this problem as a CSP augmented with a single objective function to minimise. Other frameworks, such as Partial CSP [Freuder 89], Valued CSP [Schiex 92] or Semi-Ring CSP [Bistarelli 95] may be preferred. However they are all essentially amenable to a CSP with a single objective function (for instance by merging all soft constraints). We shall focus on this simple definition throughout this dissertation. Given a constraint network $\mathcal{P}$ and an objective function Φ, i.e., a mapping $\Phi : sol(\mathcal{P}) \mapsto \mathbb{N}$, we denote $\Phi(\mathcal{P})$ the minimum value of $\Phi(f)$ for any solution $f \in sol(\mathcal{P})$.

CSOP
Instance. A constraint network $\mathcal{P}$ and an objective function Φ
Question. What is the value of $\Phi(\mathcal{P})$?

In practice, the objective function Φ must take its value in a finite interval. When a binary search on the values of Φ is possible, we only need to impose an exponential bound in the size of an encoding, i.e., $max(\Phi) - min(\Phi) \leq k^{p(n)}$ where $p(n)$ is a polynomial function, k a constant and n the size of the encoding notwithstanding Φ. Within this restriction, the problem of computing the minimal value of $\Phi(f)$ for any solution f of a constraint problem is in the complexity class P^{NP}. Indeed, in order to solve $minimise(\Phi)$ subject to $\mathcal{P}$, one can proceed by dichotomy, alternatively solving the satisfaction problem $\mathcal{P}$ augmented with the following constraint:

$$\Phi \leq \frac{ub - lb}{2}$$

and changing lb and ub accordingly. The number of calls to an NP oracle is therefore bounded by $\log_2(ub - lb)$, that is, $p(n)log_2(k)$, hence this construction is a polynomial Turing reduction to a problem in NP.

2.4 Consistency and Search

The algorithms for solving constraint satisfaction problems can be partitioned into three main classes.

Local search algorithms (e.g., `Taboo Search`, `Simulated Annealing`...) start from a complete assignment f and iteratively apply local moves either randomly or in a deterministic way. The process stops when f satisfies all constraints, i.e., is a solution, or when a limit of moves or time is reached.

Algebraic algorithms (e.g., `Adaptive Consistency`, `Bucket Elimination`...) perform operations on the constraint relations until either the problem is transformed into a unique relation or is globally consistent.

Backtracking algorithms (e.g., `Backtrack`, `MAC`...) alternate search phases (decision making) and inference phases where the consequences of these decisions are evaluated. When a decision, no matter what subsequent choices are made, leads to an inconsistency, then this decision is withdrawn and the complementary possibilities are explored.

In this dissertation we shall focus on the third type of algorithm. More precisely we shall often refer to the **Maintain Arc Consistency** (`MAC`) algorithm ([Gaschnig 74, Gaschnig 79]). Most current constraint toolkits are based upon `MAC`. The concept of **Arc Consistency**, used as inference step by this algorithm, was first introduced by Waltz

[Waltz 75], and subsequently studied by Mackworth and Freuder in [Mackworth 77] and [Mackworth 85]. Although first defined on binary constraints, the extension to non-binary constraints is not difficult. We shall define and use the extended notion of **Generalised Arc Consistency** (GAC) throughout this dissertation, independently of the constraint arity. Notice that the following definition corresponds to the notion of **Relational Arc Consistency** [Dechter 96].

Definition 3. *Given a constraint $C(V)$, a* GAC ***support*** *for $X = v$ on $C(V)$ is an assignment σ such that $\sigma(X) = v$ and $\sigma \in C(V)$.*

Definition 4. *A value $v \in \mathcal{D}(X)$ is generalised arc consistent with respect to a constraint $C(V)$ if and only if there exists a support for $X = v$ on $C(V)$. A variable $X \in \mathcal{X}$ is* GAC *with respect to $C(V)$ if and only if every value $v \in \mathcal{D}(X)$ is* GAC *with respect to $C(V)$. A constraint $C(V)$ is* GAC *if and only if each variable in V is* GAC *with respect to $C(V)$. Finally, a constraint network is* GAC *if and only if all constraints are* GAC.

2.4.1 Maintain Arc Consistency

As stated above, this algorithm alternates inference with search, that is, decision making and withdrawing. The inference step consists of computing the generalised arc consistent closure of the constraint network. We first describe the `AC3` algorithm, introduced by Mackworth and Freuder [Mackworth 85]. This procedure is often used for its simplicity and good performance in practice. Then we introduce the `MAC` algorithm that uses this inference method within a backtracking search.

Generalised Arc Consistent Closure: Let $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ be a constraint network. We say that a domain relation $\mathcal{D}'$ is a subset of the domain relation $\mathcal{D}$ if and only if the following holds: $\forall X \in \mathcal{X},\ \mathcal{D}'(X) \subseteq \mathcal{D}(X)$.

Definition 5. *The closure of a constraint network $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ with respect to a local consistency ϕ is the constraint network $\mathcal{P}' = (\mathcal{X}, \mathcal{D}', \mathcal{C})$ consistent for ϕ and such that $\mathcal{D}'$ is included in $\mathcal{D}$, and any domain relation that is a strict subset of $\mathcal{D}$ and a strict super set of $\mathcal{D}'$ is not consistent for ϕ.*

The generalised arc consistent closure of a set of domains $\mathcal{D}$ over single constraint C, denoted GAC$(C, \mathcal{D})$, is obtained using either the propagation method defined with generic closure algorithm (`AC3` [Mackworth 85], `AC2001` [Bessiere 05], and

`GAC-schema` [Bessiere 97] for instance), or a propagation algorithm associated to a constraint (`AllDifferent` [Régin 94], `GlobalCardinality` [Régin 96]). Throughout this dissertation, we assume that the constraints are either binary or accepts a dedicated propagation algorithm. In other words, we do not cover the case of non-binary constraints defined extensionally, that is, as a list of tuples. Hence, we first define an algorithm similar to `AC3` for computing the arc consistent closure on binary constraint networks, then we discuss the slight modifications to accommodate larger arity (global) constraints for which a propagation algorithm is available. We denote this closure algorithm GAC.

Closure Algorithm: We give the pseudo code of a procedure to compute the GAC closure of a constraint network in Algorithm 2. The return value is `false` if the closure is empty and `true` otherwise. The closure is obtained by iterating over a stack Q of constraints (loop 2). Q initially contains all constraints (Line 1). Moreover, because the constraints are checked in only one direction, the dual arcs are added. At each iteration, a constraint is selected (Line 3), and the procedure `propagate` (Algorithm 3) is called (Line 4). A constraint is pushed on the stack whenever the domain of a variable in its scope is reduced (Line 6). The process terminates when a fixed point is reached, that is, when the stack is empty. The generic propagation method for binary constraints, `propagate`, revises the domain of one variable with respect to a constraint and returns `true` if and only if some values have been pruned. For non-binary constraints, we assume that a different method `propagate` is associated and called in place of the generic method. Notice that such a procedure is often not directed, hence the stack initially contains only one occurrence of any global constraint.[1]

Backtracking Procedure: We give the pseudo-code of the Maintain Arc Consistency (`MAC`) algorithm in Figure 2.1. `MAC` starts with a partial solution $\varphi = \mathcal{D}$ and either extends to a solution (Line 1) or fails, thus proving infeasibility. A variable is assigned if and only if its image under φ is a singleton. During the search phase, a unassigned variable is chosen (Line 2), and the problem is divided into as many branches as values for that variable (Line 4). In each branch, an unassigned variable X is reduced to a singleton $\varphi(X) \leftarrow \{v\}$ (Line 5). During the inference phases, the domains $\mathcal{D}$ are

[1] Actual implementations of `MAC` usually have far more complex management of this queue.

replaced by φ and the GAC closure is achieved (Line 6). If the closure is not empty, then the search continues recursively (Line 7) until all variables are assigned. If the closure is empty and if not all values have been explored for this variable, then the previous state is restored (Line 8) to the previous saved state (Line 3). Then, a new branch corresponding to a new value is searched. Alternatively, if none of these branches lead to a solution, the last decision is withdrawn, we say that the algorithm **backtracks** (Line 9).

2.4.2 Branch & Bound

In this section we recall the basic Branch & Bound procedure. As opposed to a backtracking procedure applied to decision problems, the Branch & Bound algorithm successively explores and **bounds** the search space in order to find the optimal outcome for the objective function. The search part is akin to a backtracking procedure. However, since the optimal outcome is not known until all possibilities have been exhausted, the search does not stop when a solution is found. Instead, when a solution f is found, its value for the objective function $\Phi(f)$ is computed (Line 1) and the upper bound ub is updated accordingly. Then, the constraints base is augmented with the bound $\Phi(\mathcal{P}) < ub$ (Line 2). This bound can be used as a constraint to reduce the search space in the procedure `Filtering` (Algorithm 5), queried at each node (Line 3). Notice that this procedure may differ from a simple `GAC` procedure. The algorithms introduced in Chapter 6 are such alternative `Filtering` procedures.

The inference method (procedure `Filtering`, Line 3) is called at each node of the search tree, and performs inference with respect to the constraints and also with respect to the objective function and the current upper bound ub. This procedure is used to **bound** the search tree, that is, to guide the search toward strictly improving solutions by pruning branches where all solutions can only be worse or equal to the upper bound with respect to the current upper bound. In fact, since the constraint $\Phi(\mathcal{P}) < ub$ is added to the constraint network when an improving solution is found, a classical GAC closure procedure can make inference with respect to the objective function. However, since the relation $\Phi(\mathcal{P}) < ub$ may not be a constraint in the strict sense of the term[2] ,

[2] For some problems tackled in this dissertation, checking the truth value of this relation is NP-complete.

Algorithm 1 MAC

```
   Data   : $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$, $\varphi$, $\mathcal{F}[= \mathcal{X}]$
   Result : Does $\mathcal{P}$ admit a solution
1  if $\mathcal{F} = \emptyset$ then return true;
2  choose $X \in \mathcal{F}$;
3  save $\varphi$;
4  foreach $v \in \varphi(X)$ do
5      $\varphi(X) \leftarrow \{v\}$;
6      if GAC($\mathcal{P}' = (\mathcal{F}, \varphi, \mathcal{C})$) then
7          if MAC($\mathcal{P}, \varphi, \mathcal{F} \setminus \{X\}$) then return true;
8      restore $\varphi$;
9  return false;
```

Algorithm 2 GAC

```
   Data   : $\mathcal{P} = (\mathcal{X}, \varphi, \mathcal{C})$
   Result : The GAC closure of $\mathcal{P}$
1  $Q \leftarrow \mathcal{C} \cup \{C(Y, X) \mid C(X, Y) \in \mathcal{C}\}$;
2  while $Q \neq \emptyset$ do
3      select and delete any $C(X_i, X_j)$ from $Q$;
4      $pruned \leftarrow$ propagate($C(X_i, X_j)$, $\varphi$);
5      if $\varphi(X_j) = \emptyset$ then return false;
6      if $pruned$ then $Q \leftarrow Q \cup \{C(X_j, X_k)\ \forall k\}$;
   return true;
```

Algorithm 3 propagate

```
   Data   : $C(X_i, X_j)$, $\varphi$
   Result : The GAC closure of $X_j$ with respect to $C(X_i, X_j)$
   $pruned \leftarrow$ false;
   foreach $w \in \varphi(X_j)$ do
       if $\nexists v \in \varphi(X_i)$ s.t. $\langle v, w \rangle \in C(X_i, X_j)$ then
           $\varphi(X_j) \leftarrow \varphi(X_j) \setminus \{w\}$;
           $pruned \leftarrow$ true;
   return $pruned$;
```

Figure 2.1: The Maintain Arc Consistency algorithm.

in the problems introduced in this dissertation, we explicitly pass the objective function and the upper bound to the procedure `Filtering`. Moreover, we do not describe the actual behaviour of this procedure. Notice that we assume that the objective function

Algorithm 4 `Branch&Bound`

```
   Data   : $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$, $\varphi$, $\Phi$, $\mathcal{F}[= \mathcal{X}]$, $ub[= 0]$
   Result : The maximum value of $\Phi$
   if $\mathcal{F} = \emptyset$ then
1      $ub \leftarrow \Phi(\mathcal{P})$;
2      $\mathcal{C} \leftarrow \mathcal{C} \cup \{\Phi(\mathcal{P}) < ub\}$;
       return;
   choose $X \in \mathcal{F}$;
   save $\varphi$;
   foreach $v \in \varphi(X)$ do
       $\varphi(X) \leftarrow \{v\}$;
3      if Filtering($(\mathcal{F}, \varphi, \mathcal{C}), \Phi, ub$) then
           Branch&Bound($\mathcal{P}, \varphi, \Phi, \mathcal{F} \setminus \{X\}, ub$);
       restore $\varphi$;
   return $ub$;
```

Algorithm 5 `Filtering`

```
   Data   : $\mathcal{P}$, $\Phi$, $ub$
   Result : $\mathcal{P} \leftarrow$ The closure of $\mathcal{P}$ for some filtering method
1  return GAC($\mathcal{P}$);
```

Figure 2.2: The Branch & Bound algorithm.

Φ is positive. Therefore, until a first complete solution is found, the upper bound is set to 0 and has no impact on the search.

2.5 Conventions and Notations

Throughout this dissertation, we shall study the theoretical properties of a number of algorithms. Most of these procedures are either closure or search algorithms using closures. We are interested in three properties:

Complexity: We shall study the space and time complexity of several algorithms and reformulations. We use the following notations as consistently as possible: Given a constraint network $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$, we denote n the number of variables ($|\mathcal{X}|$), m the number of constraints ($|\mathcal{C}|$) and d the domain size, usually assumed homogeneous over all variables. Finally the constraint arity, that is, the cardinality of a constraint scope is denoted c, notice that this value is also often assumed homogeneous across constraints. A constraint can be defined either in **extension**, as a list of tuples, or in **intention**, that is by a predicate that can be called to check if a tuple is satisfying or not. The space and time complexity can be affected since the size of an instance is different in both cases. Using these notations, the size of a constraint network is $\mathcal{O}(nd + md^c)$ in extensional form and $\mathcal{O}(nd + m)$ if constraints are given in intention. In Chapter 4, we consider by default the extensional cases, and sometimes restrict the result to binary networks. However, in Chapter 5 and 6, we take a more practical viewpoint and consider by default that a constraint is non-binary and defined in intention.

Soundness and Completeness: We shall prove the soundness and completeness of several reformulations, local consistencies and closure or search algorithms. To avoid repetition and to clarify the notion of soundness and completeness in each of these cases, we define these notions in their respective contexts. In general, an algorithm for a decision problem is **sound** if and only it never answers “NO” on a satisfiable instance and **complete** if and only if it never answers “YES” on an unsatisfiable instance.

A local consistency ϕ is sound if and only if any assignment that can participate in a solution is consistent for ϕ. In other words the consistency will not rule out any decision that can lead to a solution. Local consistencies are usually not complete in the sense that a problem may be consistent whilst not having a solution.

A closure algorithm for a local consistency is sound if and only if it does not prune consistent assignments and complete if it prunes all inconsistent assignments. In other words, the result $\mathcal{D}'$ of the closure of $\mathcal{D}$ for a local consistency ϕ is the domain relation such that for every variable X, $\mathcal{D}'(X)$ is the largest subset of $\mathcal{D}(X)$ for which ϕ holds.

A search algorithm is sound if and only if any returned solution is valid. A search algorithm is complete if and only if, when a solution exists, it is always eventually found. In other words, a proof of unsatisfiability is always sound. Since all algorithms studied in this dissertation are backtrack search algorithms, alternating decision and inference making, and since the backtrack procedure does not hinder soundness nor completeness, we only need to prove two properties:

- To guarantee soundness of the algorithm, since only the inference method can discriminate an assignment, we must make sure that the inference method applied to a full assignment fails if this assignment is not a solution.

- To guarantee completeness we must ensure that the inference method is sound, i.e., never prunes a valid branch of the search tree.

A reformulation is a one to one mapping over problems in NP. Given a reformulation R mapping an instance $\mathcal{P}$ to $\mathcal{P}'$, we say that R is sound (resp. complete) if and only if applying a sound (resp. complete) algorithm on $\mathcal{P}'$ is a sound (resp. complete) method to solve $\mathcal{P}$. In our cases, the instance being reformulated will often be a constraint network $\mathcal{P}$ for which we aim at finding a *super*-solution, and the target another constraint network $\mathcal{P}'$ where we ask for a regular solution. The sound and complete algorithm used in the proof will thus be `MAC` and `GAC` for respectively search algorithms and consistency properties or closures.

Tightness: Last, we shall compare the filtering power, or tightness, of different local consistencies. In line with [Debruyne 97], we say that a local consistency property Φ is as strong as Ψ (written $\Phi \succeq \Psi$), if and only if, given any domains, Φ holds implies that Ψ holds; we say that Φ is stronger than Ψ (written $\Phi \succ \Psi$) if and only if $\Phi \succeq \Psi$ but not $\Psi \succeq \Phi$; we say that Φ is equivalent to Ψ (written $\Phi \simeq \Psi$) iff $\Phi \succeq \Psi$ and $\Psi \succeq \Phi$; we say that they are incomparable otherwise (written $\Phi \bowtie \Psi$).

Chapter 3

Definitions and Complexity

3.1 Introduction

In this chapter we formally define the notion of *super*-solution and relate it to *super*-models for propositional satisfiability in Section 3.2. Then we define several related problems in Section 3.3 and discuss their complexity in Section 3.4. In particular it is interesting to note that finding *super*-solutions rather than regular solutions does not change the complexity class of the problem in general. However, we show, in Section 3.5 that for some tractable classes of CSP, restricted either on the macro-structure (constraint graph), or micro-structure (constraint relations) finding *super*-solutions may be NP-hard.

3.2 Fault Tolerant Solutions

3.2.1 Super Models

The notion of *super*-models has first been defined for Boolean satisfiability problems by Roy *et al.* in [Roy 98]. An (a, b)-*super*-model α is a model of a Boolean formula such that for any set of atoms A, if $|A| \leq a$, then there exists a disjoint set B such that $|B| \leq b$ and if we negate $\alpha(A \cup B)$ we obtain another model. We quote the definition of $(1, 1)$-*super*-models from Roy's Ph.D. thesis:

Definition 6. *A super-model of a Boolean formula F is a satisfying assignment α of F, $F(\alpha) = 1$, such that for every i, if we negate the ith bit of α, there is another bit $j \neq i$ of α which we can negate to get another satisfying assignment.*

Chapter 1, page 6, *[Roy 98]*.

3.2.2 Super Solutions

We shall see that the generalisation of supermodels from Boolean to finite domains is simple and does not entail extra computational complexity. Nevertheless, the term **negate** can be interpreted either as **existential** or **universal** when applied to non-Boolean variables. For instance, negating an assignment could either mean that the value used in assignment is no longer available and an alternative can be chosen freely, or that a distinct value is arbitrarily forced by external circumstances. We therefore propose two generalisations to finite domain constraint satisfaction problems. Both definitions collapse to Roy's definition when domains are Boolean.

Example 5. *We illustrate the various concepts we introduce through the constraint network depicted in Figure 3.1. We define variables, domains and constraints for a simple constraint network, and list all the solutions accepted by this network.*

Variables:	Constraints:	Solutions:
$X_1 \in \{0,1\}$	$X_1 \leq X_2$	$\langle 0,0,0,0 \rangle$
$X_2 \in \{0,1\}$	$X_3 \leq X_4$	$\langle 0,1,0,1 \rangle$
$X_3 \in \{0,1\}$	$X_1 = X_3$	$\langle 1,1,1,1 \rangle$
$X_4 \in \{0,1\}$	$X_2 = X_4$	

Figure 3.1: A simple constraint network and its solutions.

We first define the notion of **repairability** that will be reused in subsequent definitions. We use the notion of distance $\Delta_A(f,g)$ between two assignments f and g on a set A, defined in Section 2.3.

Definition 7. *A **breakage** A is a subset of variables ($A \subseteq \mathcal{X}$). A b-repair of a breakage A for a solution f is a solution g such that $\Delta_A(f,g) = |A|$ and $\Delta(f,g) \leq |A| + b$.*

*A **breakage** A is **existentially** b**-repairable** (or simply b-repairable) for a solution f if and only if there exists a b-repair of A for f.*

*Similarly, A is **universally** b**-repairable** iff for every assignment τ of A such that $\Delta_A(\tau, f) = |A|$, there exists a b-repair g of A for f such that $g|_A = \tau$.*

A *repair* of a breakage A for a solution f is an alternative solution with a distinct image on A and at most b discrepancies on $\mathcal{X} \setminus A$. For instance, in Figure 3.1, the solution $\langle 0, 0, 0, 0\rangle$ is a 1-*repair* of the breakage $\{X_2\}$ for the solution $\langle 0, 1, 0, 1\rangle$. A *super*-solution is then defined as a solution such that any subset of a or less variables (breakage) is repairable. A *super*-solution is existential if every breakage is extensionally repairable or universal if every possible ways for that subset to change is covered by a *repair*, i.e., it is universally repairable.

Definition 8. *An existential-(a, b)-super-solution f is a solution of $\mathcal{P}$ such that for any set $A \subseteq \mathcal{X}$, if $A \leq a$, then A is b-repairable for f.*

Definition 9. *A universal-(a, b)-super-solution f is a solution of $\mathcal{P}$ such that for any set $A \subseteq \mathcal{X}$, if $A \leq a$, then A is universally b-repairable for f.*

In the constraint network illustrated in Figure 3.1, $\langle 0, 1, 0, 1\rangle$ is an existential-$(1, 1)$-*super*-solution since $\langle 0, 0, 0, 0\rangle$ is a 1-*repair* for the breakages $\{X_2\}$ and $\{X_4\}$ whilst $\langle 1, 1, 1, 1\rangle$ is a 1-*repair* for the breakages $\{X_1\}$ and $\{X_3\}$. The solution $\langle 0, 1, 0, 1\rangle$ is both an existential-$(1, 1)$-*super*-solution and a universal-$(1, 1)$-*super*-solution since domains are Boolean. In fact it is also a $(1, 1)$-*super*-model of the equivalent Boolean formula. Notice that there is a number of ways one can generalise the definition to non-Boolean domains, whilst being consistent with the original definition. We cover these two natural definitions in the present chapter. Then in subsequent chapters, we shall focus on the former, existential-(a, b)-*super*-solution, which we shall refer to as (a, b)-*super*-solution or simply *super*-solution when there are no ambiguities. Finally, we shall see in Chapter 7 that the methods introduced in this dissertation can handle variations of the classical definition.

The main reason for focusing on existential-(a, b)-*super*-solutions is that the universal condition is much too strong to be practically useful. For instance, if there exists a unary assignment $X = v$ that does not participate in any solution, then there is no universal-(a, b)-*super*-solution since the breakage $\{X\}$ is not repairable. There are similar, though weaker, conditions for existential *super*-solutions. For instance, a SAT formula or a constraint satisfaction problem with a **backbone variable** [Schneider 96] cannot have any existential-(a, b)-*super*-solution. Indeed, a variable belongs to the backbone if and only if it takes the same value in all solutions. Therefore, there is no alternative for this variable, hence a breakage involving this variable is not repairable for any repair size. A last example is the case of **permutation** problems. In a permutation problem a set of n variables must be assigned a set of n values such that no two variables share the same value. These problems arise frequently in practice. However there cannot be any $(1,0)$-*super*-solutions, either existential or universal, for permutation problems, since if a variable was to be reassigned, the value it was previously assigned must be taken by some other variable, hence at least one repair is necessary. For instance the constraint network illustrated in Figure 3.1 does not admit any $(1,0)$-*super*-solutions, since it contains two equality constraints.

Another important observation is that we cannot find super-solutions merely by adding to the constraint network a given (global) constraint satisfying the conditions defined in Section 2.3. More precisely, there is no constraint C satisfying the definition in [Bessiere 03] and such that for any network $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$, a solution of $\mathcal{P}' = (\mathcal{X}, \mathcal{D}, \mathcal{C} \cup \{C\})$ is a *super*-solution of $\mathcal{P}$. Indeed, constraints are defined as polynomial time checkable relations over a set of values, and should not be affected by the actual domains $\mathcal{D}$. We show through an example that the semantics of such a constraint relation is not independent of the domains.

Example 6. *For instance, consider the following constraint network:*

$$X_1 \leq X_2 \leq X_3 \quad \mathcal{D}(X_i) = \{1,2,3\}\ \forall i$$

The mapping $f : X_i \rightarrow i$ is a existential-$(1,0)$-super-solution. However, if the domains are changed to:

$$\mathcal{D}(X_i) = \{1,2,3\}\ \forall i \in [1..2] \quad \mathcal{D}(X_3) = \{1,3\}$$

then f, although still a valid assignment, is no longer an existential-$(1,0)$-super-solution, as there is no repair if X_3 breaks.

A constraint relation can be defined as either accepting the tuple $\langle 1,2,3 \rangle$ or rejecting it. However it cannot be conditional on the current state of the domains.

3.2.3 Repairability

We have seen that some properties of problems and models do not allow *super*-solutions. Alternatively, a constraint network can simply be too tightly constrained to accept a *super*-solution. In such cases some solutions may still be preferred as they are "closer" to a *super*-solution than others. To this end, we define the notion of partial **repairability** of a solution to be the number of breakages that can be repaired.

Definition 10. *The existential-(a,b)-repairability of a solution f is equal to the number of subsets A of $\mathcal{X}$ with cardinality less or equal than a that are b-repairable for f.*

Definition 11. *The universal-(a,b)-repairability of a solution f is equal to the number of subsets A of $\mathcal{X}$ with cardinality less or equal than a that are universally b-repairable for f.*

For instance, in example 3.1, the solution $\langle 0, 1, 0, 1\rangle$ has a $(1, 1)$-repairability of 4, whilst $\langle 0, 0, 0, 0\rangle$ and $\langle 1, 1, 1, 1\rangle$ have both a $(1, 1)$-repairability of 2. The repairability therefore defines a relaxed notion of stability. We consider in this chapter the problem of finding a solution with maximal repairability. There are other measures of stability, related to the notion of *super*-solution, that could be useful in certain situations. We also consider, in this dissertation the relaxation of the repair size b. In this case, we want to find the solution for which b is minimum. However, many other possibilities exist, for instance we could maximise the size of the repairable breakages (a), or we could aggregate these criteria in some way.

3.3 Problem Definition

We define some problems related to the notion of *super*-solution. We consider two classes of extensions besides the standard problem of deciding the existence of an universal or existential *super*-solution for a constraint network. In the second class we seek *super*-solutions with maximum objective value on constraint satisfaction and optimisation problems. Finally, we define partial problems, that can be seen as analogous to MaxCSP or MaxSAT in the context of *super*-solutions: The solution returned is not necessarily a *super*-solution but as close as possible to be one.

Here again we covered the problems that seemed "natural". However many other problems related to *super*-solutions could be considered.

3.3.1 Satisfaction Problems

We first define the problem of the existence of *super*-solutions, either universal or existential.

$\exists(a,b)$-SuperCSP
Instance. A constraint network $\mathcal{P}$.
Question. Does there exist an existential-(a,b)-*super*-solution of $\mathcal{P}$?

$\forall(a,b)$-SuperCSP
Instance. A constraint network $\mathcal{P}$.
Question. Does there exist a universal-(a,b)-*super*-solution of $\mathcal{P}$?

3.3.2 Optimisation Problems

Next we define the problem of finding an optimal *super*-solution for some objective function. Here again the *super*-solution may be universal or existential. Moreover, we may ask the repairs to be themselves optimal or near optimal or even not restrict them at all with respect to the objective function.

$\exists(a,b)$-SuperCSOP
Instance. A constraint network $\mathcal{P}$, an exponentially bounded objective function Φ.
Question. What is the minimum value of $\Phi(f)$ where f is an existential-(a,b)-*super*-solution of $\mathcal{P}$?

$\forall(a,b)$-SuperCSOP
Instance. A constraint network $\mathcal{P}$, an exponentially bounded objective function Φ.
Question. What is the minimum value of $\Phi(f)$ where f is a universal-(a,b)-*super*-solution of $\mathcal{P}$?

$\exists(a,b)$-SuperCSOP*
Instance. A constraint network $\mathcal{P}$, an exponentially bounded objective function Φ.
Question. What is the minimum value σ for which an existential-(a,b)-*super*-solution and a complete set of *repair*s have an image under Φ less than or equal to σ?

$\forall(a,b)$-SuperCSOP*
Instance. A constraint network $\mathcal{P}$, an exponentially bounded objective function Φ.
Question. What is the minimum value σ for which an universal-(a,b)-*super*-solution and a complete set of *repair*s have an image under by Φ less than or equal to σ?

3.3.3 Partial Problems

Finally, we define the problem of finding the solution with best repairability, or minimal maximum repair size. These problems are "partial" in the sense that the robustness condition is relaxed, i.e., partially enforced, as in the partial constraint satisfaction problem [Freuder 89].

$\exists a$-MinBCSP
Instance. A constraint network $\mathcal{P}$.
Question. What is the minimum value of b for which there exists an existential-(a, b)-*super*-solution of $\mathcal{P}$?

$\forall a$-MinBCSP
Instance. A constraint network $\mathcal{P}$.
Question. What is the minimum value of b for which there exists a universal-(a, b)-*super*-solution of $\mathcal{P}$?

$\exists(a, b)$-MaxRepairCSP
Instance. A constraint network $\mathcal{P}$.
Question. What is the maximum value of existential-(a,b)-repairability(f) for any solution f of $\mathcal{P}$?

$\forall(a, b)$-MaxRepairCSP
Instance. A constraint network $\mathcal{P}$.
Question. What is the maximum value of universal-(a,b)-repairability(f) for any solution f of $\mathcal{P}$?

3.4 Complexity

3.4.1 Decision Problems

Deciding if a SAT problem has an (a, b)-*super*-model is NP-complete [Ginsberg 98]. Since *super*-solutions collapse to *super*-models on Boolean domains, and SAT problems can be seen as Boolean CSPs, the NP-hardness result lifts immediately. Throughout this section, n will denote the number of variables ($|\mathcal{X}|$) of a constraint network $\mathcal{P}$, and d the domain size $|\mathcal{D}(X_i)|$ which we consider uniform across all variables unless stated otherwise. The parameters a and b are considered part of the definition of the problem. Moreover, a needs to be a constant for these proofs to be correct. However, there is no such restriction on b. Notice that since we are not considering a and b as "data", an

NP-completeness result for a given value of a or b does not necessarily extend to larger values.

Theorem 1. $\exists(a,b)$-SuperCSP *is NP-complete if a is constant.*

Proof. $\exists(a,b)$**-SuperCSP is in NP:** The polynomial witness is the *super*-solution itself and a set of *repair*s. The number of *repair*s is $\sum_{k=1}^{k=a}\binom{n}{k}$. This a polynomial number ($\leq n^a$) of solutions, and checking each of them can be done in polynomially bounded time.

$\exists(a,b)$**-SuperCSP is NP-hard:** In the particular case where the constraint network is a SAT formula F, $\exists(a,b)$-SuperCSP is equivalent to deciding if F has an (a,b)-*super*-model (NP-complete from [Ginsberg 98]). $\square$

Theorem 2. $\forall(a,b)$-SuperCSP *is NP-complete if a is constant.*

Proof. $\forall(a,b)$**-SuperCSP is in NP:** The polynomial witness is the *super*-solution itself and a set of *repair*s. The number of *repair*s is $\sum_{k=1}^{k=a}\binom{n}{k}(d-1)^a$. This a polynomial number ($\leq n^a d^a$) of solutions, and checking each of them can be done in polynomially bounded time.

$\forall(a,b)$**-SuperCSP is NP-hard:** In the particular case where the constraint network is a SAT formula F, $\exists(a,b)$-SuperCSP is equivalent to deciding if F has an (a,b)-*super*-model (NP-complete from [Ginsberg 98]). $\square$

We nevertheless give a reduction of CSP into $\exists(a,b)$-SuperCSP that shall be used in some subsequent proofs. Our reduction constructs a constraint network which, if it has any solution, has an existential-(a,b)-*super*-solution for any $a+b\leq n$. An example illustrating this construction is given in Figure 3.2.

Construction 1. *Given a constraint network $\mathcal{P}=(\mathcal{X},\mathcal{D},\mathcal{C})$ we first assume, without loss of generality, that only strictly positive values are used ($\Lambda=\mathbb{Z}^+$). We construct $\mathcal{P}'=(\mathcal{X},\mathcal{D}',\mathcal{C}')$ as follow s. The domains of each variable is concatenated to its negation: $\forall X\in\mathcal{X},\ \mathcal{D}'(X)=\mathcal{D}(X)\cup\{-v\mid v\in\mathcal{D}(X)\}$. Then we modify the constraints so that they behave equivalently on the new values. For instance, we can replace every occurrence of a variable X by its absolute value $abs(X)$. Clearly, this constraint network has a solution if and only if the original network also has. In addition, any breakage*

$A \subseteq \mathcal{X}$ variables can be repaired by replacing the corresponding values with their negated values and any number of reassignments (negations). Therefore, for any given $a, b \leq n$, $\mathcal{P}'$ has an existential-(a,b)-super-solution if and only if $\mathcal{P}$ is satisfiable.

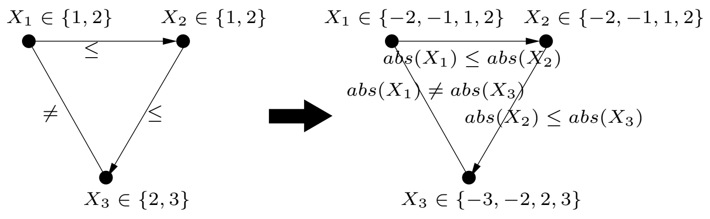


Figure 3.2: The reduction of an instance of CSP to (a,b)-*super*-SuperCSP.

Similarly, we give a reduction of CSP to $\forall(a,b)$-SuperCSP.

Construction 2. *Given a constraint network $\mathcal{P}$, we build $\mathcal{P}'' = (\mathcal{X}, \mathcal{D}, \mathcal{C}'')$ by replacing the set of constraints $\mathcal{C}$ by a singleton $\mathcal{C}'' = \{C''(\mathcal{X})\}$ with $C''(\mathcal{X})$ defined as follows:*

$$\tau \in C''(\mathcal{X}) \Leftrightarrow \exists \rho \mid \Delta(\tau, \rho) \leq a \ \wedge \ \forall C(V) \in \mathcal{C}, \ \rho_{|_V} \in C(V)$$

First, observe that this constraint is polynomial to check, since, given a tuple τ, the number of tuples at Hamming distance a or less is $\sum_{k \in [0..a]}(\binom{n}{k}d^k)$, which is a polynomial in a. Now we show that any solution f of $\mathcal{P}$ is a universal-(a,b)-super-solution of $\mathcal{P}''$. Indeed, consider any breakage $A \subseteq \mathcal{X}$ such that $|A| \leq a$. Any assignment of A extended so as to match f on $\mathcal{X} \setminus A$, is by definition at a Hamming distance $|A| \leq a$ of f and thus is a solution. Finally it is easy to see that if $\mathcal{P}$ has no solution, then neither has $\mathcal{P}''$ since $C''(\mathcal{X})$ would then be empty.

We saw that the problem (a,b)-SuperCSP is in NP. Moreover, it is NP-complete to decide if a given solution is an $(1,b)$-*super*-solution.

Theorem 3. *Deciding if a solution is an existential-$(1,b)$-super-solution is NP-complete.*

Proof. **Deciding if a solution is an $(1,b)$-*super*-solution is in NP:** The polynomial witness is the set of n *repair*s. Checking each of them can be done in polynomially bounded time.

Deciding if a solution is an $(1,b)$-*super*-solution is NP-hard: We reduce the problem of deciding the existence of a clique of size at least K in a graph to our problem. Given a graph $G = (V, E)$, we introduce a constraint network $\mathcal{P}$ with $n = |V|$ Boolean variables $X_1, \dots X_n$ standing for the nodes of the graph, and one extra Boolean variable X_0. Then, for every pair of nodes $v_i, v_j \in V$ such that $(v_i, v_j) \notin E$, we introduce the constraint $C(X_0, X_i, X_j) = (X_0 = 0 \Rightarrow (X_i + X_j \leq 1))$. Finally the "query" solution will be $f : X_i \rightarrow 1$, and the parameters a and b will be set respectively to 1 and $n - K$. We show that there exists a clique of size at least K in G if and only if f is an $(1, b)$-*super*-solution of $\mathcal{P}$.

$\Rightarrow$: Suppose that $Cl \subseteq V$ is such that $\forall v_i, v_j \in Cl, (v_i, v_j) \in E \ \wedge \ |Cl| \geq K$. notice that as long as X_0 is assigned to 1, no constraint can be violated, hence any breakage $\{X_i\}$ such that $1 \leq i \leq n$ admits a 0-*repair*. Now consider the breakage $\{X_0\}$. Given a clique $Cl \subseteq V$ of cardinality K, we define the solution g as follows:

$$g(X_i) = \begin{cases} 1 & if \ v_i \in Cl \\ 0 & otherwise \end{cases}$$

The solution g is a b-*repair* for the breakage $\{X_0\}$. Consider, without loss of generality, a constraint $C(X_0, X_i, X_j)$, by definition, $(v_i, v_j) \notin E$, therefore v_i and v_j cannot both belong to Cl, hence $(g(X_i) + g(X_j)) \leq 1$. Consequently, $C(X_0, X_i, X_j)$ is satisfied. Moreover, since $|Cl| = K$, exactly K variables are assigned to the value 1, hence $\Delta(f, g) = n - K + 1$.

$\Leftarrow$: Suppose that f is an $(1, b)$-*super*-solution. It follows that the breakage $\{X_0\}$ must admit a b-*repair*. The only one alternative assignment for X_0 is 0, therefore the "conditional" constraints now ensure that two variables corresponding to nodes that do not share an edge in the graph cannot be simultaneously assigned to 1. Hence the restriction of any solution g to $X_1, \dots X_n$ corresponds to a clique in G. Moreover, since the number of discrepancies with f $(\Delta(f, g))$ is at most $n - K + 1$, at least K variables are assigned 1. Therefore the corresponding clique has cardinality K. □

3.4.2 Optimisation Problems

The proofs for optimisation problems are straightforward as they use the Constructions 1 and 2. The only difficulty is to make sure that the objective value of a *super*-solution corresponds to that of the corresponding solution and, more critically, that the same property applies to *repair*s when needed.

Theorem 4. $\exists(a,b)$-SUPERCSOP *is* P^{NP}*-complete if* a *is constant.*

Proof. **$\exists(a,b)$-SUPERCSOP is in $\mathbf{P}^{\mathrm{NP}}$:** We need to show that a polynomial number of calls to an NP oracle is sufficient to solve this problem. We use the following oracle: "does there exist an existential-(a,b)-*super*-solution f for the constraint network $\mathcal{P}$ such that $\Phi(f) \leq k$?". This problem is in NP, the polynomial witness is f plus the *repair*s, and checking that $\Phi(f) \leq k$ can be done in polynomial time. We can therefore proceed as usual by dichotomy, and only $log(ub - lb)$ calls are needed (hence $O(n)$).

$\exists(a,b)$-SUPERCSOP is $\mathbf{P}^{\mathrm{NP}}$-hard: We reduce CSOP $(\mathcal{P},\Phi)$ to $\exists(a,b)$-SUPERCSOP $(\mathcal{P}',\Phi')$. The constraint network $\mathcal{P}'$ is obtained by Construction 1 from $\mathcal{P}$. We know that any existential-(a,b)-*super*-solution f of $\mathcal{P}'$ is such that $abs(f)$ is a solution of $\mathcal{P}$. Furthermore, let Φ' be the objective function defined as Φ whereall occurrences of a variable X are replaced by $abs(X)$. It follows that the objective value for Φ' of any existential-(a,b)-*super*-solution of $\mathcal{P}'$ is equal to the value by Φ of the corresponding solution $abs(f)$ of $\mathcal{P}$. Therefore f is an optimal existential-(a,b)-*super*-solution of $(\mathcal{P}',\Phi)$ if and only if $abs(f)$ is an optimal solution of $(\mathcal{P},\Phi)$. □

Theorem 5. $\exists(a,b)$-SUPERCSOP* *is* P^{NP}*-complete if* a *is constant.*

Proof. **$\exists(a,b)$-SUPERCSOP* is in $\mathbf{P}^{\mathrm{NP}}$:** We need to show that a polynomial number of calls to an NP oracle is sufficient to solve this problem. We use the decision version as oracle: "does there exist an (a,b)-*super*-solution f for the constraint network $\mathcal{P}$ augmented with $\Phi \leq k$?". This problem is in NP (see Theorem 1). We can therefore proceed as usual by dichotomy, and only $log(ub - lb)$ calls are needed (hence $O(n)$).

$\exists(a,b)$-SUPERCSOP* is $\mathbf{P}^{\mathrm{NP}}$-hard: We can reuse the same reduction as for $\exists(a,b)$-SUPERCSOP. Indeed, the objective value by Φ' of any *repair* obtained by negating a (set of) variable(s) is the same as that of the existential-(a,b)-*super*-solution. Therefore f is an optimal existential-(a,b)-*super*-solution of $(\mathcal{P}',\Phi')$ if and only if $abs(f)$ is an optimal solution of $(\mathcal{P},\Phi)$. □

Theorem 6. $\forall(a,b)$-SUPERCSOP *is* P^{NP}*-complete if* a *is constant.*

Proof. **$\forall(a,b)$-SUPERCSOP is in $\mathbf{P}^{\mathrm{NP}}$:** We need to show that a polynomial number of calls to an NP oracle is sufficient to solve this problem. We use the following oracle:

"does there exist an universal-(a,b)-*super*-solution f for the constraint network $\mathcal{P}$ such that $\Phi(f) \leq k$?". This problem is in NP, the polynomial witness is f plus the *repair*s, and checking that $\Phi(f) \leq k$ can be done in polynomial time. We can therefore proceed as usual by dichotomy, and only $log(ub - lb)$ calls are needed (hence $O(n)$).

$\forall(a,b)$-SuperCSOP **is $\mathbf{P}^{\mathrm{NP}}$-hard:** We reduce a CSOP $(\mathcal{P}, \Phi)$ to a $\forall(a,b)$-SuperCSOP $(\mathcal{P}'', \Phi)$. The constraint network $\mathcal{P}''$ is obtained by Construction 2 from $\mathcal{P}$. We know that any solution f is a universal-(a,b)-*super*-solution of $\mathcal{P}''$ if and only if it is a solution of $\mathcal{P}$. Moreover, the value of f by the objective function Φ does not change. Therefore f is an optimal universal-(a,b)-*super*-solution of $(\mathcal{P}'', \Phi)$ if and only if it is an optimal solution of $(\mathcal{P}, \Phi)$. □

Theorem 7. $\forall(a,b)$-SuperCSOP* *is P^{NP}-complete if a is constant.*

Proof. $\forall(a,b)$-SuperCSOP* **is in $\mathbf{P}^{\mathrm{NP}}$:** We need to show that a polynomial number of calls to an NP oracle is sufficient to solve this problem. We use the decision version as oracle: "does there exist an (a,b)-*super*-solution f for the constraint network $\mathcal{P}$ augmented with $\Phi \leq k$?". This problem is in NP, see Theorem 2. We can therefore proceed as usual by dichotomy, and only $log(ub - lb)$ calls are needed (hence $O(n)$).

$\forall(a,b)$-SuperCSOP **is $\mathbf{P}^{\mathrm{NP}}$-hard:** reduce a CSOP $(\mathcal{P}, \Phi)$ to a $\forall(a,b)$-SuperCSOP $(\mathcal{P}'', \Phi'')$. The constraint network $\mathcal{P}''$ is obtained by Construction 2 from $\mathcal{P}$. The objective function is defined as follows:

$$\Phi''(f) = \begin{cases} \Phi(f) & if\ f \in sol(\mathcal{P}) \\ max(\Phi(g),\ \forall g \mid \Delta(f,g) \leq a) & otherwise \end{cases}$$

It follows that any *repair* g of f is such that $\Phi''(g) \geq \Phi''(f)$. Therefore f is an optimal universal-(a,b)-*super*-solution of $(\mathcal{P}'', \Phi'')$ if and only if it is an optimal solution of $(\mathcal{P}, \Phi)$. □

3.4.3 Partial Problems

In the subsequent reductions, we will use the problem of computing the clique of maximum size of a graph (MaxClique) which is $\mathrm{P}^{\mathrm{NP}[log(n)]}$-complete ([Papadimitriou 94]).

MaxClique
Instance. A Graph $G = (V, E)$.

> **Question.** What is the maximum cardinality of $C \subseteq V$ such that $\forall x, y \in C,\ (x, y) \in E$.

We denote $\exists$1-MinBCSP (resp. $\forall$1-MinBCSP) the problem of finding minimum value of b for which there exists a existential-$(1, b)$-*super*-solution (resp. universal-$(1, b)$-*super*-solution).

Theorem 8. $\exists$1-MinBCSP *is* $P^{\mathrm{NP}[log(n)]}$*-complete.*

Proof. **$\exists$1-MinBCSP is in $\mathbf{P}^{\mathrm{NP}[log(n)]}$:** Consider the problem of deciding if a constraint network $\mathcal{P}$ has a existential-(a, b)-*super*-solution for $b \leq k$. This problem is in NP as f and the *repair*s form a valid polynomial witness. Moreover, b ranges in $[0 \dots n]$, therefore the number of calls to this NP oracle is bounded by $log_2(n)$.

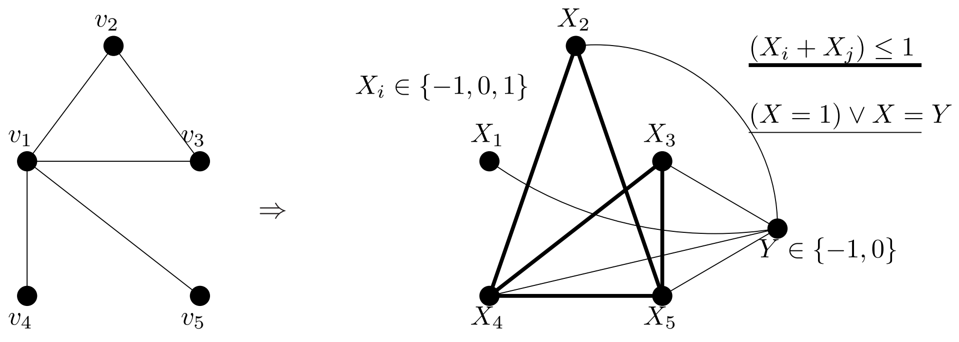
$\exists$1-MinBCSP is $\mathbf{P}^{\mathrm{NP}[log(n)]}$-hard: We will reduce MaxClique to $\exists$1-MinBCSP. We illustrate this reduction in Figure 3.3. Given a graph $G = (V, E)$, we construct a CSP $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ with $n + 1$ variables: $\mathcal{X} = \{X_i \mid v_i \in V\} \cup \{Y\}$. All X_i's take value in $\{-1, 0, 1\}$ and Y takes value in $\{-1, 0\}$. For every pair $(v_i, v_j) \notin E$, we introduce a constraint $C(X_i, X_j) = (X_i + X_j \leq 1)$ to forbid both variables corresponding to a "non-edge" to be simultaneously assigned to 1. Therefore, the variables assigned to 1 must correspond to a clique of G. For all i, we introduce the constraint $C(Y, X_i) = (X_i = 1 \vee Y = X_i)$. These constraints ensure that all X_i that are not assigned to 1 are altogether assigned to either 0 or -1 along with Y. Notice that both situations (all 0s or all -1s) are symmetric. Therefore we consider, without loss of generality, the case where they are all assigned to 0. Now let K be the set of variables assigned to 1. We know that K must correspond to a clique, moreover we will show that the least b for which there exists a $(1, b)$-*super*-solution is $n - |K|$. There are 3 types of breakage. For the two first we show that no more than $n - |K|$ changes are required in repair, and for the last we show that exactly $n - |K|$ changes are required:

- A breakage on X assigned to 1: X can be reassigned to 0 and no further change is needed.
- A breakage on X assigned to 0: X can be reassigned to -1 along with all other variables currently assigned to 0, hence we need at most $n - |K|$ changes.

- A breakage on Y: Y has only one alternative, -1. Therefore we must similarly reassign all other variables currently assigned to 0, hence exactly $n-|K|$ changes.

Therefore there exists a existential-$(1, n-|K|)$-*super*-solution but no existential-$(1,b)$-*super*-solution for $b < n - |K|$. It follows that the existential-$(1,b)$-*super*-solution with smallest b corresponds to the clique of maximum cardinality. □

Example 7. *In Figure 3.3 we give an example of a graph G and its reformulation as a constraint network $\mathcal{P}$ such that the optimal value for the* MinBCSP *on $\mathcal{P}$ is equal to the size of a maximal clique in G. The constraints standing for the "non-edges" are represented with bold lines, whilst additional constraints linking variables standing for nodes to the extra variable Y are represented with thin lines.*



(a) G (b) $\mathcal{P}$

Figure 3.3: The reduction of an instance of MaxClique to ∃1-MinBCSP.

Theorem 9. ∀1-MinBCSP *is $P^{\mathrm{NP}[log(n)]}$-complete.*

Proof. ∀1-MinBCSP **is in $\mathbf{P}^{\mathrm{NP}[log(n)]}$:** We can reuse the same polynomial Turing reduction as for ∃1-MinBCSP, substituting existential with universal.

∀1-MinBCSP **is $\mathbf{P}^{\mathrm{NP}[log(n)]}$-hard:** We use the same many-one reduction as for ∃1-MinBCSP with the following changes: Prior to the construction described in the earlier proof, we add n unconnected vertices to G. This does not essentially change the problem as the maximum clique size is preserved. However, the maximum clique size is n, that is, half the number of variables. Now we consider again all types of breakage:

- A breakage on X, assigned to 1 and forced to $X = 0$: No further change is needed.
- A breakage on X, assigned to 1 and forced to $X = -1$: We need to reassign all variables currently assigned to 0 to -1, hence $2n - |K| + 1$ changes.
- A breakage on X, assigned to 0 and forced to $X = -1$: We need to reassign all variables currently assigned to 0 to -1, hence $2n - |K|$ changes.
- A breakage on X, assigned to 0 and forced to $X = 1$: We reassign the $|K|$ variables currently assigned to 1 to 0, hence at most K changes ($K \leq n$).
- A breakage on Y: Y has only one alternative, -1. Therefore we must similarly reassign all other variables currently assigned to 0, hence exactly $2n - |K|$ changes.

Therefore there exists a universal-$(1, 2n - |K| + 1)$-*super*-solution but no universal-$(1, b)$-*super*-solution for any $b \leq 2n - |K|$. It follows that the universal-$(1, b)$-*super*-solution with smallest b corresponds to the clique with maximum cardinality. □

Theorem 10. $\exists(1, b)$-MAXREPAIRCSP *is* $P^{\mathrm{NP}[log(n)]}$*-complete for any* $b \geq 0$.

Proof. **$\exists(1, b)$-MAXREPAIRCSP is in $\mathbf{P}^{\mathrm{NP}[log(n)]}$:** Consider the problem of deciding if a constraint network $\mathcal{P}$ has a solution f such that there exist k distinct variables $X_{i1}, \dots X_{ik} \in \mathcal{X}$ accepting a *b-repair*. This problem is in NP as f and the k *repair*s form a valid polynomial witness. Moreover, the maximum number of *repair*s is n.Therefore the number of calls to this NP oracle is bounded by $log_2(n)$.

$\exists(1, b)$-MAXREPAIRCSP is $\mathbf{P}^{\mathrm{NP}[log(n)]}$-hard: We first prove that $\exists(1, 0)$-MAXREPAIRCSP is $\mathrm{P}^{\mathrm{NP}[log(n)]}$-hard by reducing MAXCLIQUE to this problem. Given a graph $G = (V, E)$ we construct a constraint network $\mathcal{P}$ as follows: We introduce a Boolean variable X_i for each vertex $v_i \in V$. For every pair $(v_i, v_j) \notin E$, we introduce a constraint $C(X_i, X_j) = (X_i + X_j \leq 1)$ to forbid both variables to be simultaneously assigned to 1. Therefore, the variables assigned to 1 must correspond to a clique of G. We add n^2 variables and constraints to correlate the repairability with the number of variables X_i assigned to 1. For every variable X_i we add $\{Y_{i1}, \dots Y_{in}\}$ and n constraint $C(X_i, Y_{ik}) = (Y_{ik} \leq X_i)\ \forall k \in [1..n]$. This gadget, shown in Figure 3.4, ensures that for every variable X_i, the n Y_{ik} can be reassigned without further changes only if X_i is assigned to 1. Indeed, if $X_i = 0$, then $C(X_i, Y_{ik})$ forces $Y_{ik} = 0$ and no alternative is

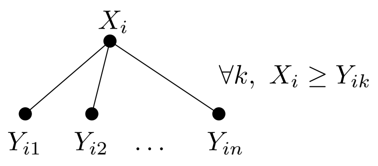

X_i
$\forall k,\ X_i \geq Y_{ik}$
Y_{i1} Y_{i2} $\ldots$ Y_{in}

Figure 3.4: The gadget for correlating repairability with clique size.

possible. Moreover, if $X_i = 1$ then it is $(1,0)$-repairable if and only if $Y_{ik} = 0$ for any k. Let K be a clique of maximum cardinality and consider the solution f such that $\forall i, k,\ f(Y_{ik}) = 0$ and $\forall i,\ f(X_i) = 1$ iff $v_i \in K$. The repairability of f is $(n+1)|K|$ since for each X_i such that $f(X_i) = 1$, we know that $X_i, Y_{i1}, \ldots Y_{in}$ are repairable. Now consider any solution g that assigns a strictly smaller set of X_i to 1, that is, $\sum_i(g(X_i)) < |K|$. Then at least $n(n - |K| + 1)$ variables (the corresponding Y_{ik}) are not repairable. And therefore the number of repairable variables is bounded by the complement: $n|K|$. The solution f has therefore maximum repairability.

Now we show that a similar reduction can be made for every $0 \leq b \leq n$. For every variable Y_{ik} we add b variables $Z_{ik_1}, \ldots Z_{ik_b}$ and b constraints:

$$Y_{ik} = Z_{ik_1} = Z_{ik_2} = \ldots Z_{ik_b}$$

For every "vertex" variable X_i, we count the number of repairs according to the value taken. If X_i is assigned to 1 all Y_{ik}, Z_{ik_j} assigned to 0 then X_i is repairable as it can be reassigned to 0. Moreover, Y_{ik} and Z_{ik_j} for any $1 \leq j \leq b$ can be reassigned to 1 provided that all of them (b more) are. Now suppose that X_i is assigned to 0, then all Y_{ik}, Z_{ik_j} must be assigned to 0. Moreover, if one breaks, then $b+1$ repairs are required (all other Y_{ik}'s, Z_{ik_j}'s and also X_i). Therefore a solution, corresponding to a clique K, such that a number $|K|$ of node variables (X_i's) are assigned to 1 has $|K|(bn+1)$ repairs. □

Theorem 11. $\forall(1,b)$-MaxRepairCSP *and is* $P^{\mathrm{NP}[log(n)]}$*-complete for any* $b \geq 0$.

Proof. $\forall(1,b)$-MaxRepairCSP **is in** $\mathbf{P}^{\mathrm{NP}[log(n)]}$**:** The same polynomial Turing reduction as for the existential version may be used,

$\forall(1,b)$-MaxRepairCSP **is $\mathbf{P}^{\mathrm{NP}[log(n)]}$-hard:** The same many-one reduction as for the existential version may be used as it involves only Boolean domains. □

We summarise these complexity results in Table 3.1.

	$(1,0)$	$(1,b)$	(a,b)
∃SuperCSP	NP-complete	NP-complete	NP-complete
∃SuperCSOP	P^{NP}-complete	P^{NP}-complete	P^{NP}-complete
∃SuperCSOP*	P^{NP}-complete	P^{NP}-complete	P^{NP}-complete
∃MinBCSP	-	$\mathrm{P}^{\mathrm{NP}[log(n)]}$-complete	$\in \mathrm{P}^{\mathrm{NP}[log(n)]}$
∃MaxRepairCSP	$\mathrm{P}^{\mathrm{NP}[log(n)]}$-complete	$\mathrm{P}^{\mathrm{NP}[log(n)]}$-complete	$\in \mathrm{P}^{\mathrm{NP}[log(n)]}$

	$(1,0)$	$(1,b)$	(a,b)
∀SuperCSP	NP-complete	NP-complete	NP-complete
∀SuperCSOP	P^{NP}-complete	P^{NP}-complete	P^{NP}-complete
∀SuperCSOP*	P^{NP}-complete	P^{NP}-complete	P^{NP}-complete
∀MinBCSP	-	$\mathrm{P}^{\mathrm{NP}[log(n)]}$-complete	$\in \mathrm{P}^{\mathrm{NP}[log(n)]}$
∀MaxRepairCSP	$\mathrm{P}^{\mathrm{NP}[log(n)]}$-complete	$\mathrm{P}^{\mathrm{NP}[log(n)]}$-complete	$\in \mathrm{P}^{\mathrm{NP}[log(n)]}$

Table 3.1: The complexity of finding *super*-solutions.

3.5 Polynomial Classes

The complexity of finding *super*-models of Boolean formulae has been studied in depth in [Roy 98]. Tractable classes for generalised satisfiability (Boolean CSPs) are well known, and Schaefer's theorem [Schaefer 78] indicates that any Boolean problem is either isomorphic to 0-valid-SAT, 1-valid-SAT, 2SAT, Horn-SAT, dual Horn-SAT, affine-SAT or is NP-complete. The complexity of finding *super*-models for these classes is summarised in Table 3.2 (from [Roy 98]).

	0-valid-SAT	1-valid-SAT	2SAT
$(1,1)$-*super*-SAT	NP-complete	NP-complete	P
$(1,2)$-*super*-SAT	NP-complete	NP-complete	NP-complete

	Horn-SAT	dual Horn-SAT	affine-SAT
$(1,1)$-*super*-SAT	NP-complete	NP-complete	P
$(1,2)$-*super*-SAT	NP-complete	NP-complete	P

Table 3.2: The complexity of finding *super*-models for SAT tractable classes ([Roy 98]).

The situation is slightly less clear for non-Boolean CSP. Indeed, a similar dichotomy theorem exists, however, it is limited to ternary domains [Bulatov 02]. The question remains open for domains of larger cardinality. Tractable classes of CSP can be partitioned into two categories. The tractability may result either from properties of the network macro-structure, that is, the constraint graph, or properties of its micro-structure, that is, the constraint relations. Examples of the first approach can be found in [Freuder 82, Freuder 85, Gyssens 94, Montanari 91] and examples of the second approach can be found in [van Beek 92, Cooper 94, van Hentenryck 92, Jeavons 97, Karousis 93]. In this section we show that the archetypal tractable class defined over the macro-structure, TreeCSP, remains NP-hard for *super*-solution existence. We also show that arguably the simplest class of CSP which tractability comes from the constraint relations (class-0) is also NP-hard for *super*-solutions. Finally, we show that the converse may also be true, i.e., for some NP-hard instances of CSP, deciding if a *super*-solution exists may be done in polynomial time, although we are not aware of any useful instance of this class. Notice that since all the problems analysed in this section are particular cases of the problem of finding *super*-solutions on general constraint networks, the membership to the class NP is trivial. We therefore only prove NP-hardness, hence proving NP-completeness.

3.5.1 Tractability due to the Constraint Graph

We call TreeCSP the problem of deciding if a binary constraint network whose constraint graph is a tree has a solution. This problem can be solved in polynomial time, as generalised arc-consistency is equivalent to global consistency on such CSPs. Moreover, a generalisation of TreeCSP, where the constraint graph has a bounded treewidth have been shown to be polynomial to solve in [Freuder 85]. We show in this section that the problem of deciding if a TreeCSP has a universal-$(1, b)$-*super*-solution (resp. existential-$(1, b)$-*super*-solution) is NP-complete for $b \geq 1$ (resp. $b \geq 2$). The membership to the class NP is clear since the same witness as in a general constraint network can be used. We therefore first prove NP-hardness for $b = 1$ (resp. $b = 2$) and then we give a reduction from the case $b = k$ to $b = k + 1$ that preserves the tree structure, hence proving NP-completeness for any $b \geq 1$ (resp. $b \geq 2$).

Theorem 12. $\forall(1,1)$-SuperTreeCSP *is NP-complete.*

Proof. We reduce 3-Colouring to the problem of deciding if a CSP such that the constraint graph is a tree has a universal-$(1,1)$-*super*-solution. Given a graph $G = (V, E)$ we construct the CSP $\mathcal{P}$ as follows: We introduce $n = |V|$ variables $X_1, \dots X_n$ taking values in $\{red, blue, green\}$. Then we introduce a variable Y whose domain consist of all triplets $\langle i, j, c\rangle$ such that $(i, j) \in E$ and $c \in \{red, blue, green\}$, plus a triplet $\langle 0, 0, 0\rangle$. Now, for all i in $[1..n]$ we introduce a constraint $C(Y, X_i)$ with the following relation:

$$Y = \langle 0,0,0\rangle \ \vee \ (Y[1] \neq i \ \wedge \ Y[2] \neq i) \ \vee \ Y[3] \neq X_i$$

Clearly, this constraint network is a tree, since the constraints overlap only on the variable Y.

First, we show that if the 3-Colouring problem has a solution then $\mathcal{P}$ has a universal-$(1,1)$-*super*-solution. Consider a colouring of G, and let f be the solution of the CSP such that $f(X_i) = c$ if and only if the vertex v_i takes colour c and $f(Y) = \langle 0, 0, 0\rangle$. Obviously, it is a solution as all constraints are satisfied when $Y = \langle 0, 0, 0\rangle$. Now, without loss of generality, consider a breakage on X_i. Any value (colour) is consistent, therefore this breakage can be repaired without any extra reassignment. Finally, we must ensure that Y can take any of the values in its domain, and reassigning only one variable will be enough to get a solution. Without loss of generality, consider a value $\langle i, j, c\rangle$. Any constraint $C(Y, X_k)$ such that $k \neq i$ and $k \neq j$ is satisfied. Thus, only $C(Y, X_i)$ and $C(Y, X_j)$ are possibly violated. However, since there is an edge $(i, j) \in E$, we know that the vertices v_i and v_j do not take the same colour, and consequently $X_i \neq X_j$. We therefore can have $X_i = c$, or $X_j = c$ (or neither) but not both. Suppose that we have $X_i = c$, we reassign X_i to any value in $\{red, blue, green\} \setminus \{c\}$, this is a solution since $C(Y, X_i)$ will be satisfied.

Now we show that if the 3-Colouring problem has no solution then $\mathcal{P}$ has no universal-$(1,1)$-*super*-solution. There is a one-to-one correspondence between the colourings (satisfying or not) of G and the assignments of $\{X_1, \dots X_n\}$. Suppose that there is no colouring such that every edge connect vertices of different colours. Then for any mapping, consider one edge (ij) such that v_i and v_j share the same colour. The corresponding assignment f in $\mathcal{P}$ is such that $f(X_i) = f(X_j) = c$. Moreover, since

$(i, j) \in E$, Y contains the value $\langle i, j, c\rangle$ where $c = X_i = X_j$. If Y is assigned the value $\langle i, j, c\rangle$ then we have to change both X_i and X_j. Hence there is no universal-$(1, 1)$-*super*-solution. □

Theorem 13. $\exists(1, 2)$-SuperTreeCSP *is NP-complete.*

Proof. We reduce 3-Colouring to $\exists(1, 2)$-SuperTreeCSP. We use the same construction as in the previous proof to which we add $m = 3.|E|$ variables, one for each edge $(i, j) \in E$, and colour $c \in \{red, blue, green\}$ such that $Z_{ijc} \in \{\langle 0, 0, 0\rangle, \langle i, j, c\rangle\}$. Then we add m constraints $C(Y, Z_{ijc})$ with the following relation:

$$Z_{ijc} = \langle 0, 0, 0\rangle \ \vee \ Y = Z_{ijc}$$

Here again, this constraint network is a tree, since the constraints overlap only on the variable Y.

First, we show that if the 3-Colouring problem has a solution then $\mathcal{P}$ has a existential-$(1, 1)$-*super*-solution. Let define f as in the previous proof, and let extend it to the extra variables as follows: $f(Z_{ijc}) = \langle 0, 0, 0\rangle$. Any variable X_i for $i \in [1..n]$ can take any value without violating the constraints. We have seen that Y can take any of its value entailing at most 1 X_i to be reassigned. Moreover, changing Y will not violate $C(Y, Z_{ijc})$ for any i, j, c. Now we consider the breakage of a variable Z_{ijc}. The only alternative value for Z_{ijc} is $\langle i, j, c\rangle$, and the only one constraint that is violated by this reassignment is $C(Y, Z_{ijc})$. We therefore reassign Y to $\langle i, j, c\rangle$, and from the last proof, we know that at most one further reassignment will be required, for a total of 2.

Now we show that if the 3-Colouring problem has no solution then the $\mathcal{P}$ has no universal-$(1, 1)$-*super*-solution, using the same argument as in the previous proof. Consider one edge (ij) such that v_i and v_j share the same colour. The corresponding assignment f in $\mathcal{P}$ is such that $f(X_i) = f(X_j) = c$. Consider now the breakage of Z_{ijc}. The only alternative is $\langle i, j, c\rangle$ and the only way to satisfy $C(Y, Z_{ijc})$ is to assign Y to $\langle i, j, c\rangle$. Moreover, to satisfy respectively $C(Y, X_i)$ and $C(Y, X_j)$ is to reassign X_i and X_j to some other colour. Hence we need at least 3 reassignments. □

Example 8. *In Figures 3.5 and 3.6 we give an example of colouring problem and the instances obtained by reduction to* $\exists(1, 2)$-SuperTreeCSP *and to* $\forall(1, 1)$-SuperTreeCSP*.*

The original instance of 3-Colouring *is represented in Figure 3.5a. Its reduction to an instance of* $\forall(1,1)$-SuperTreeCSP *is shown in Figure 3.5b and to an instance of* $\exists(1,2)$-SuperTreeCSP *in Figure 3.6. Observe that the domains given aside the graphs are sets of tuples.*

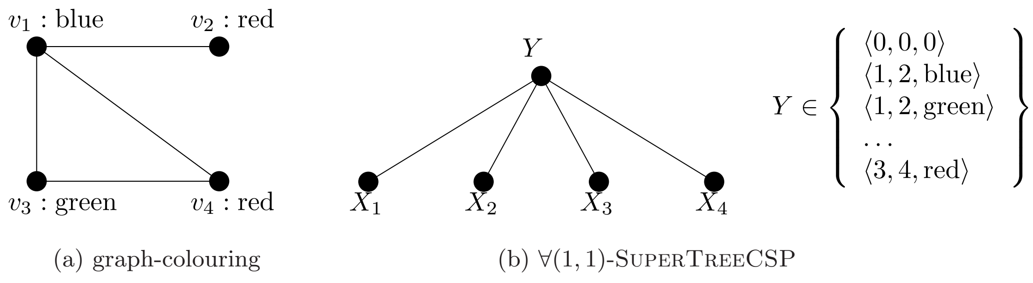

v_1 : blue
v_2 : red
v_3 : green
v_4 : red
(a) graph-colouring
Y
X_1
X_2
X_3
X_4
$Y \in \left\{ \begin{array}{l} \langle 0,0,0\rangle \\ \langle 1,2,\text{blue}\rangle \\ \langle 1,2,\text{green}\rangle \\ \dots \\ \langle 3,4,\text{red}\rangle \end{array} \right\}$
(b) $\forall(1,1)$-SuperTreeCSP

Figure 3.5: The reduction of an instance of GraphColouring to $\forall(1,1)$-SuperTreeCSP.

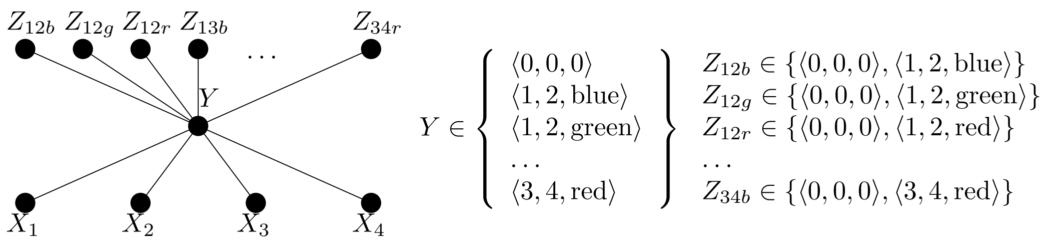

Z_{12b} Z_{12g} Z_{12r} Z_{13b} … Z_{34r}
Y
X_1 X_2 X_3 X_4
$Y \in \left\{ \begin{array}{l} \langle 0,0,0\rangle \\ \langle 1,2,\text{blue}\rangle \\ \langle 1,2,\text{green}\rangle \\ \dots \\ \langle 3,4,\text{red}\rangle \end{array} \right\}$
$Z_{12b} \in \{\langle 0,0,0\rangle, \langle 1,2,\text{blue}\rangle\}$
$Z_{12g} \in \{\langle 0,0,0\rangle, \langle 1,2,\text{green}\rangle\}$
$Z_{12r} \in \{\langle 0,0,0\rangle, \langle 1,2,\text{red}\rangle\}$
…
$Z_{34b} \in \{\langle 0,0,0\rangle, \langle 3,4,\text{red}\rangle\}$

Figure 3.6: The reduction of an instance of GraphColouring to $\exists(1,2)$-SuperTreeCSP.

We now show that $\exists/\forall(1,b)$-SuperCSP is easier than (can be reduced to) $\exists/\forall(1,b+1)$-SuperCSP. We cannot use Construction 1 since it shows that finding *super*-solutions is harder than finding solutions whilst, in this case, finding solutions is polynomial. The following construction preserves the tree structure of the constraint graph. We can therefore conclude that finding existential (resp. universal) $(1,b)$-*super*-solutions on tree-CSP is NP-hard for any $b > 1$.

Theorem 14. *For any $b > 0$, there exists a many-one reduction from* $\forall(1,b)$-SuperCSP *to* $\forall(1,b{+}1)$-SuperCSP *and another one from* $\exists(1,b)$-SuperCSP *to* $\exists(1,b{+}1)$-SuperCSP*, and this reduction preserves the tree structure of the constraint network.*

Proof. We reduce $\exists(1,b)$-SUPERCSP to $\exists(1,b+1)$-SUPERCSP. Without loss of generality, we assume that the constraint network $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ involves only positive values. We construct $\mathcal{P}' = (\mathcal{X}', \mathcal{D}', \mathcal{C}')$ defined as follows:

- We add one variable Y_{iv} for every $X_i \in \mathcal{X}$ and every $v \in \mathcal{D}(X_i)$.
  $\mathcal{X}' = \mathcal{X} \cup \{Y_{iv} \mid X_i \in \mathcal{X}, v \in \mathcal{D}(X_i)\}$
- The domain of an extra variable Y_{iv} contains the value v and -1.
  $\forall X_i \in \mathcal{X},\ v \in \mathcal{D}(\mathcal{X}),\ \mathcal{D}'(X_i) = \mathcal{D}(X_i)\ \wedge\ \mathcal{D}'(Y_{iv}) = \{-1, v\}$
- We add one constraint for each variable Y_{iv} that forces Y_{iv} to take a different value than X_i.
  $\mathcal{C}' = \mathcal{C} \cup \{C(X_i, Y_{iv}) \mid Y_{iv} \in \mathcal{X}'\}$ where $C(X_i, Y_{iv}) = \{\tau \mid \tau[1] \neq \tau[2]\}$

First, we show that for any $(1,b)$-*super*-solution of $\mathcal{P}$ we can construct a $(1,b+1)$-*super*-solution of $\mathcal{P}'$. Given f a $(1,b)$-*super*-solution of $\mathcal{P}$, we define f' as follows:

$$f'(X) = \begin{cases} f(X) & if\ X \in \mathcal{X} \\ -1 & otherwise \end{cases}$$

We consider every possible breakage of f'. We distinguish 3 cases:

1. If $X_i \in \mathcal{X}$ breaks: We know that there exists a b-*repair* of $\{X_i\}$ for f therefore at most b reassignments are required in $\mathcal{X}$. Moreover, any constraint $C(X_i, Y_{iv})$ is still satisfied as $f'(Y_{iv}) = -1$.

2. $Y_{iv} \in \mathcal{X}' \setminus \mathcal{X}$ breaks, and $f'(X_i) \neq v$: No more reassignments are required as only $C(X_i, Y_{iv})$ constrains Y_{iv} and is still satisfied.

3. $Y_{iv} \in \mathcal{X}' \setminus \mathcal{X}$ breaks, and $f'(X_i) = v$: The only alternative is v, thus we must change the value assigned to X_i. At this point, the situation is similar to case 1, and we know that at most b other reassignments are required, hence a total of $b+1$ reassignments.

Now we show that for any $(1,b+1)$-*super*-solution f' of $\mathcal{P}'$, its restriction to $\mathcal{X}$ is a $(1,b)$-*super*-solution of $\mathcal{P}$. We need to make sure that the breakage of any variable $X_i \in \mathcal{X}$ can be repaired with at most b reassignments of variables in $\mathcal{X}$. Consider a variable X_i, and suppose that $f'(X_i) = v$. We have $Y_{iv} = -1$ otherwise $C(X_i, Y_{iv})$

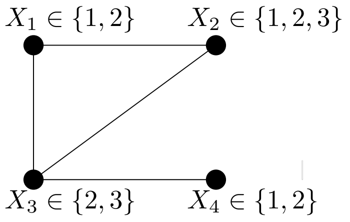
would not be satisfied. Now consider a breakage of f' on Y_{iv}. Since f' is a $(1, b+1)$-*super*-solution we know that there exists g such that $g(Y_{iv}) = v$ and $\Delta_{\mathcal{X}'}(f', g) \leq b + 2$. Moreover, we know that $g(X_i) \neq v$, therefore $f'(X_i) \neq g(X_i)$. Now, since $Y_{iv} \notin \mathcal{X}$, we have $\Delta_{\mathcal{X}}(f', g) \leq b + 1$ hence g is a b-*repair* of $\{X_i\}$ for f' restricted to $\mathcal{X}$.

Notice that the same construction works both for universal and existential *super*-solutions. Indeed, the domain of any variable in $\mathcal{X}' \backslash \mathcal{X}$ is binary, therefore, an "universal" breakage is actually equivalent to an "existential" breakage. $\square$

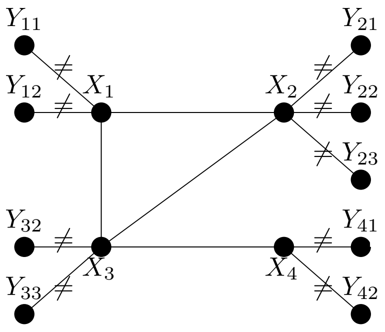
Corollary 1. *For any given $b > 0$, $\forall(1, b)$-SuperTreeCSP and $\exists(1, b{+}1)$-SuperTreeCSP are NP-complete.*

It is an open problem if $\exists(1,1)$-SuperTreeCSP is polynomial. We would either need to give a polynomial time decision procedure (based perhaps on enforcing some level of local consistency) or come up with a new reduction.

Example 9. *In Figure 3.7 we give an instance of a $\forall/\exists(1, b)$-SuperCSP and its reduction to an instance of $\forall/\exists(1, b + 1)$-SuperCSP. For every variable in the instance depicted in Figure 3.7a we add as many variables as values in the domain in the instance shown in Figure 3.7b. Observe that the tree width of the constraint graph is not affected since all the constraints introduced are binary relation between a variable and its "own" extra variables.*



(a) $\forall/\exists(1, b)$-SuperCSP



(b) $\forall/\exists(1, b + 1)$-SuperCSP

Figure 3.7: The reduction of an instance of $\forall/\exists(1, b)$-SuperCSP to $\forall/\exists(1, b + 1)$-SuperCSP.

3.5.2 Tractability due to the Constraint Relations

We show that one of the simplest class of CSP such that a property of the constraint relations entails tractability is NP-hard for *super*-solutions. It is shown in [Jeavons 97] that the closure of the constraint relations for some types of operations is a sufficient condition for tractability. We recall the formalism used in [Jeavons 97] study some of the tractability classes investigated by Jeavons and Cohen in our specific context.

Definition 12. *Let $\otimes : \Lambda^k \mapsto \Lambda$ be a k-ary operation on Λ, and let C be a n-ary relation over Λ. For any collection of k tuples $\tau_1, \dots \tau_k \in C$, the n-tuple $\otimes(\tau_1, \dots \tau_k)$ is defined as follows:*

$$\otimes(\tau_1, \dots, \tau_k) = (\otimes(\tau_1[1], \dots \tau_k[1]), \dots \otimes (\tau_1[n], \dots, \tau_k[n]))$$

Finally the closure $\otimes(C)$ of C under $\otimes$ is defined as the n-ary relation:

$$\{\otimes(\tau_1, \dots \tau_k) \mid \tau_1, \dots \tau_k \in C\}$$

We take as example a **constant** operation that associates the value c to any tuple: $\otimes(v_1, \dots v_k) = c$. Following the definition, it means that C is closed under $\otimes$ if and only if $\langle c, c, \dots, c\rangle \in C$ or $C = \emptyset$. The resulting class of CSP (called CLASS-0) is trivially polynomial as $f : X_i \rightarrow c$ is a solution unless there is an empty constraint, in which case the CSP is unsatisfiable. [1]

Theorem 15. *Deciding if a* CSP *in* CLASS-0 *has a existential-(a, b)-super-solution is NP-complete.*

Proof. We reduce CSP to this problem. Given a constraint network $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ we construct $\mathcal{P}' = (\mathcal{X}, \mathcal{D}', \mathcal{C}')$ defined as follows: The domains $\forall X \in \mathcal{X}\ \mathcal{D}'(X) = (\mathcal{D}(X) \cup \{v' \mid v \in \mathcal{D}(X)\} \cup \{c\})$ and $\mathcal{C}' = \{C(V) \cup \{c^{|V|}\} \mid C(V) \in \mathcal{C}\}$. Notice that we chose c so that it does not appear in any original domain.

Suppose we modify a CSP by adding an extra value to each variable, and relaxing the constraints to permit variables to take this new value. The new CSP has all the solutions of the old, plus the solution that assigns the new value to each variable. As

[1] Notice that the domains are here considered as unary constraints and must therefore contain c

it always has at least one solution, CSP is polynomial. However, (a, b)-SuperCSP is NP-hard as it requires finding the *super*-solutions of the original CSP. □

As a second example we consider the language of constraints closed under a majority operation. As defined in [Jeavons 97], the majority operation $\otimes$ is ternary and has the following semantic

$$\otimes(v_1, v_2, v_3) = \begin{cases} v_2 & if\ v_2 = v_3 \\ v_1 & otherwise \end{cases}$$

The class of CSP whose constraints are all closed under this majority operation is tractable ([Jeavons 97]). However this is not the case with SuperCSP:

Theorem 16. *Deciding if a* CSP *has a existential-*$(1, b)$*-super-solution is NP-hard even if all constraints are closed under a majority operation.*

Proof. Deciding if a 2-SAT formula admits a $(1, b)$-*super*-solution is NP-complete for $b > 1$ [Roy 06]. 2-clauses are trivially closed under the majority operation. Therefore the closure for the majority operation is not a sufficient condition for tractability of SuperCSP. □

We now summarise the results on the complexity of finding *super*-solutions for some tractable classes of CSPs. We report the complexity results for the class of constraint networks that are tractable because the language of constraints is closed under a constant operation in Table 3.3 and a majority operation in Table 3.4. Finally the complexity results for the problem of finding *super*-solution when the constraint graph is a tree is summarised in Table 3.5.

	$\exists$ (closure: constant)	$\forall$ (closure: constant)
$\forall b$ $(1, b)$-*super*-CSP	NP-complete	NP-complete

Table 3.3: The complexity of finding *super*-solutions on languages of constraint closed under a constant operation.

3.6 Tractable *super*-CSP

In this section we state that $(1, 0)$-*super*-solutions TreeCSP (both existential and universal) can be computed in polynomial time, however since it requires a reformulation

	$\exists$ (closure: majority)	$\forall$ (closure: majority)
$(1, \leq 1)$-*super*-CSP	?	?
$(1, > 1)$-*super*-CSP	NP-complete	NP-complete

Table 3.4: The complexity of finding *super*-solutions on languages of constraint closed under a majority operation.

introduced and detailed in Chapter 4, we refer the reader to section 4.2.2 for a proof.

Theorem 17. $\exists(1,0)$-SuperTreeCSP *is in P*

Proof. See section 4.2.2. □

Theorem 18. *On binary constraint networks,* $\forall(1,0)$-SuperCSP *is in P*

Proof. Consider a tuple $\langle v, w\rangle$ that does not satisfy a constraint $C(X,Y)$ (i.e., a nogood, $\langle v, w\rangle \notin C(X,Y)$). The assignment $X = v$ (resp. $Y = w$) cannot participate in a universal-$(1,0)$-*super*-solution since assigning Y to w (resp. X to v) would violate $C(X,Y)$. Therefore, only non constrained values can be part of a universal-$(1,0)$-*super*-solution. Checking which values are not constrained can be done in polynomial time, by listing all tuples. Clearly, if at least one such value exists for each variable, then a universal-$(1,0)$-*super*-solution exists. □

So far, we have seen that several tractable classes of CSP are no longer tractable in the *super*-solution framework. However, tractable classes for SuperCSP are not necessarily tractable classes for CSP. We give a counter example.

Theorem 19. *There exist classes of CSPs for which* CSP *is NP-complete but* (a,b)-SuperCSP *is polynomial.*

Proof. Suppose we add to any CSP a variable with a singleton domain and no constraints on it. CSP is still NP-complete. However, any breakage involving this variable is unrepairable. Such a problem has no (a,b)-*super*-solution for any a or b. Thus (a,b)-SuperCSP is trivially polynomial. □

	∃Tree	∀Tree
$(1,0)$-*super*-CSP	P	P
$(1,1)$-*super*-CSP	?	NP-complete
$(1,2)$-*super*-CSP	NP-complete	NP-complete
$(1,b)$-*super*-CSP	NP-complete	NP-complete

Table 3.5: The complexity of finding *super*-solutions of CSPs whose constraint graph is a tree.

3.7 Summary and Limitations

In this chapter we defined the notion of *super*-solution as well as some related decision, optimisation and partial problems. The subsequent complexity analysis showed that computing *super*-solutions rather than regular solutions does not change the complexity class of the problem in general. However, as it was observed in [Roy 98] for propositional satisfiability, polynomial classes of CSP often become NP-hard when computing *super*-solutions. This seems to indicate that whilst the type of robustness proposed in this dissertation tends to increase the complexity of a problem, it does not dramatically change it. Several questions remain open. For instance, the complexity of the $(1,1)$-TreeCSP problem, or the complexity of finding $(1,0)$-*super*-solutions and $(1,1)$-*super*-solutions on constraint languages closed under a majority operation, are not known. We did not find an example of a non-trivial class of constraint network that would be tractable for the SuperCSP problem whilst NP-hard for the CSP problem. Moreover, throughout this chapter, we gave several complexity results assuming that the parameter a was equal to 1. This is obviously a particular case, hence in these cases the more general version of the same problem where a may be set to any value between 1 and some constant k, is harder, however this may not be the case for some particular values of a. For instance, using Construction 1 we proved that the (a,b)-SuperCSP is NP-complete for any pair of value $a \in [1..k]$, $b \in [0..n]$, where k is a constant, and the same was possible for other problems. However, for the problem of: deciding if a solution is an (a,b)-*super*-solution; deciding if a constraint network whose constraint graph is a tree admits an (a,b)-*super*-solution; finding the minimum value of b such that there exists an (a,b)-*super*-solution; finding the solution with maximal (a,b)-repairability; there may exist some values of a, in the interval $[2..n]$, for which a polynomial algorithms exists. In fact, this is unlikely, however we did not prove the opposite.

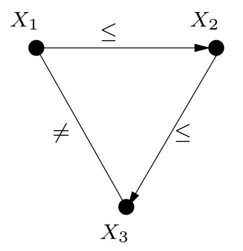
Chapter 4

Full Fault Tolerance

4.1 Introduction

In this chapter we extensively study a particular case, the existential-$(1, 0)$-*super*-solutions. The first reference to this particular type of *super*-solutions appears in [Weigel 98], where it is referred to as **fault tolerant solutions**. Any variable of a $(1, 0)$-*super*-solution can be assigned an alternative value without any further change. This is therefore an important case, useful when no extra repair is allowed in case of a breakage. The notion of **substitutability** and **interchangeability** [Freuder 91] are closely related to full fault tolerance. Given two values $v, w \in \mathcal{D}(X)$ we say that w is substitutable for v if and only if for any solution f such that $f(X) = v$, there exists a solution g such that $g(X) = w$ and $g(\mathcal{X} \setminus \{X\}) = f(\mathcal{X} \setminus \{X\})$. Furthermore, v and w are interchangeable if and only if v is substitutable for w and vice versa. Notice that interchangeable values are equivalent alternative choices. In fact, a solution f such that for any variable X, there exists $v \in \mathcal{D}(X)$ substitutable for $f(X)$ is a $(1, 0)$-*super*-solution. However the relation is in one direction only since the alternative value need not be interchangeable, or even substitutable for the original assignment. The reason for this is that, for $(1, 0)$-*super*-solutions, the substitution need to be consistent with respect to a unique solution whilst interchangeable or substitutable values are so for all solutions.

Example 10. *We give an example of* $(1, 0)$*-super-solution in Figure 10. Among all solutions, only one, namely* $\langle 1, 2, 3\rangle$ *is a* $(1, 0)$*-super-solution. Indeed for each breakage of size 1, there exists an alternative solution where all other variables stay unchanged. All solutions are listed in Figure 4.1b and the* 0*-repairs for all breakages of* $\langle 1, 2, 3\rangle$ *are* *listed in Figure 4.1c.*



(a) Constraint network

$\langle 1,1,2\rangle$	$\langle 1,1,3\rangle$
$\langle 1,2,2\rangle$	$\langle \mathbf{1,2,3}\rangle$
$\langle 1,3,3\rangle$	$\langle 2,2,3\rangle$
$\langle 2,3,3\rangle$	$\langle 3,3,3\rangle$

(b) Solutions

$\{\}:$	$\langle 1,2,3\rangle$
$\{X_1\}:$	$\langle 2,2,3\rangle$
$\{X_2\}:$	$\langle 1,3,3\rangle$
$\{X_3\}:$	$\langle 1,2,2\rangle$

(c) Breakages/repairs

Figure 4.1: A constraint network, its solutions, and the repairs of its unique $(1,0)$-*super*-solution.

We shall see that in the full fault tolerance case, the concept of *super*-solution can be given a **local** characterisation, while for any $a, b \geq 1$ an (a,b)-*super*-solution is essentially a **global** concept. The intuitive idea is that we can enforce a local property which guarantees that any solution found is a $(1,0)$-*super*-solution. By analogy to local consistency, we will refer to this property as *super*-GAC. This principle cannot easily be extended to (a,b)-*super*-solution because the discrepancies between a b-repair and a (a,b)-*super*-solution may not be local, i.e., may be arbitrarily distributed across the network. Therefore it is difficult to assert that a *repair* exists without having a global view. In this chapter we introduce some methods for taking advantage of this local reasoning. We shall see that this approach is closely related to the notion of local consistency for classical CSPs. Moreover we shall see that these consistency methods may be more or less difficult to adapt to non-binary and/or global constraints. Whenever the approach we are presenting applies only to binary constraint network, we state it explicitly.

In Section 4.2 we recall a reformulation approach for finding $(1,0)$-*super*-solutions from [Weigel 98], then we introduce a new reformulation approach. In Section 4.3 we introduce two methods using this property of locality, and we define the notion of *super*-GAC. We define the respective closures for these local consistencies and introduce some algorithms to enforce them in Section 4.4. In Section 4.5 we compare formally the different methods introduced as well as the reformulation method from [Weigel 98].

Finally, in Section 4.6, we give the complete search algorithm using the consistency property defined earlier.

4.2 Reformulation Methods

4.2.1 Previous Method: $\mathcal{P}+\mathcal{P}$ Reformulation

We first review the previous method described in [Weigel 98] to find $(1,0)$-*super*-solutions that we shall denote $\mathcal{P}+\mathcal{P}$. Notice that in [Weigel 98] this reformulation is given for binary constraint networks. In fact, as we show in this section, the same construction can be made for network with constraints of arbitrary large arity. Given a constraint network $\mathcal{P}=(\mathcal{X},\mathcal{D},\mathcal{C})$, $\mathcal{P}+\mathcal{P}=(\mathcal{X}^+,\mathcal{D}^+,\mathcal{C}^+)$ is defined as follows:

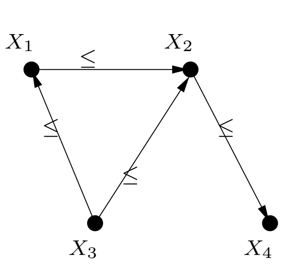
Definition 13. *The reformulation $\mathcal{P}+\mathcal{P}$ of the constraint network $\mathcal{P}$ is a triplet $(\mathcal{X}^+,\mathcal{D}^+,\mathcal{C}^+)$ such that: $\mathcal{X}^+$ contains two copies X_i and X_i^+ of every variable $X_i \in \mathcal{X}$. The domains of original and duplicate variables are equal and unchanged. $\mathcal{C}^+$ contains all constraints in $\mathcal{C}$ plus a constraint $X_i \neq X_i^+$ for each variable $X_i \in \mathcal{X}$. Moreover, for every constraint $C(V) \in \mathcal{C}$, and every variable $X_i \in V$, a constraint $C(V \cup \{X_i^+\} \setminus \{X_i\})$ is added to $\mathcal{C}^+$.*

	$\mathcal{P}$			$\mathcal{P}+\mathcal{P}$
Variables:	$\mathcal{X}$	$\mathcal{X}^+$	$=$	$\mathcal{X} \cup \{X_i^+ \mid X_i \in \mathcal{X}\}$
Domains:	$\mathcal{D}$	$\mathcal{D}^+(X_i)$	$=$	$\mathcal{D}(X_i)\ \ \forall X_i \in \mathcal{X}$
		$\mathcal{D}^+(X_i^+)$	$=$	$\mathcal{D}(X_i)\ \ \forall X_i \in \mathcal{X}$
Constraints:	$\mathcal{C}$	$\mathcal{C}^+$	$=$	$\mathcal{C} \cup \{X_i \neq X_i^+ \mid X_i \in \mathcal{X}\}$
				$\cup\{C(V \setminus \{X_i\} \cup \{X_i^+\}) \mid C(V) \in \mathcal{C}\}$

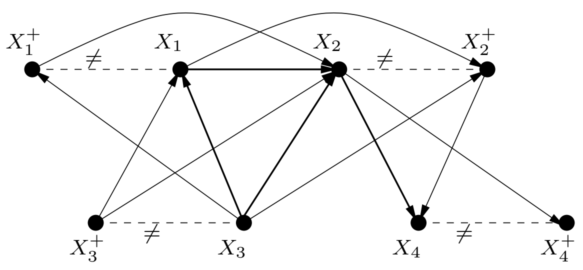
Table 4.1: The $\mathcal{P}+\mathcal{P}$ reformulation summary.

Example 11. *We give an example of the $\mathcal{P}+\mathcal{P}$ reformulation in Figure 4.2. A simple constraint network $\mathcal{P}$ is shown in Figure 4.2a and its reformulation is given in Figure 4.2b. The bold arcs stand for the original constraints, the thin arcs stand for the extra constraints (one for each variable of each original constraint) and finally, the dashed arcs stand for the disequality constraints between original and duplicated variables.*

Theorem 20. *The reformulation $\mathcal{P}+\mathcal{P}$ is sound and complete and has a space complexity in $\mathcal{O}(nd + cmd^c)$.*



(a) A constraint network $\mathcal{P}$



(b) The reformulation $\mathcal{P}+\mathcal{P}$ of $\mathcal{P}$

Figure 4.2: A constraint network $\mathcal{P}$ and its reformulation $\mathcal{P}+\mathcal{P}$.

Proof. We show that the solutions of $\mathcal{P}+\mathcal{P}$ restricted to $\mathcal{X}$ are exactly $(1,0)$-*super*-solutions of $\mathcal{P}$.

Soundness: Any solution $f : \mathcal{X}^+ \mapsto \Lambda$ of the reformulation $\mathcal{P}+\mathcal{P}$ is such that its restriction $f|_{\mathcal{X}}$ to the variables of $\mathcal{P}$ is a $(1,0)$-*super*-solution of $\mathcal{P}$. Let f be solution of $\mathcal{P}+\mathcal{P}$, for any X_i we can substitute the image given to X_i with the image given to X_i^+, which must be different, whilst keeping all constraints in $\mathcal{C}$ satisfied. Indeed for any constraint $C(V)$ such that $X_i \in V$, we know that $C(S \setminus \{X_i\} \cup \{X_i^+\})$ is satisfied. Therefore, for any breakage on a variable X_i in $\mathcal{P}$, the value assigned to X_i^+ is a valid alternative, that requires no reassignment. In other words, the solution g equal to f on $\mathcal{X} \setminus \{X_i\}$ and such that $g(X_i) = f(X_i^+)$ is a 0-*repair* of X_i for f.

Completeness: Conversely, if f is a $(1,0)$-*super*-solution of $\mathcal{P}$, then for any variable X_i there exists a 0-*repair* of X_i for f. Now consider the assignment g of $\mathcal{P}+\mathcal{P}$ such that for any i, $g(X_i) = f(X_i)$ and $g(X_i^+)$ is equal to the image of X_i by the 0-repair of X_i for f. We have $g(X_i^+) \neq g(X_i)$, and moreover any duplicated constraint is satisfied as substituting the $f(X_i)$ with its alternative does not violate any constraint in $\mathcal{P}$.

Space Complexity: The number of variables is $2n$, the number of constraints $(c+1)m$: To the original constraint $C(V)$ is added one constraint for each element of V. The domains and constraint relation do not change. The space complexity therefore is in $\mathcal{O}(nd + cmd^c)$, i.e. exactly $2nd + 3md^2$ for binary constraint networks. □

4.2.2 New Method: $\mathcal{P} \times \mathcal{P}$ Reformulation

We now present a second and new reformulation approach. This reformulation increases the amount of pruning a local consistency like GAC would achieve as well as the space complexity. Notice also that this reformulation is restricted to binary constraint networks. Indeed, this construction relies on the observation that the restriction of a super solution to a pair of variables always follows the same pattern. An assignment $\langle X = v_1, Y = w_1 \rangle$ is a $(1,0)$-*super*-solution on the network restricted to X, Y if and only if there exists $v_2 \in \mathcal{D}(X)$ and $w_2 \in \mathcal{D}(Y)$ such that $\langle X = v_1, Y = w_1 \rangle$, $\langle X = v_1, Y = w_2 \rangle$ and $\langle X = v_2, Y = w_1 \rangle$ are all consistent. We illustrate this pattern in Figure 4.3.

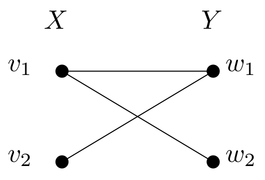


Figure 4.3: A $(1,0)$-*super*-solution over two variables.

For non-binary constraints with bounded arity, it is possible to isolate the same pattern. For a 3-tuple to be a $(1,0)$-*super*-solution over 3 variables, 4 tuples need to be consistent: the tuple itself plus three *repair*s sharing all but one value. We illustrate this pattern in Figure 4.4.

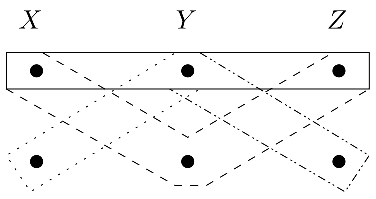


Figure 4.4: A $(1,0)$-*super*-solution over three variables.

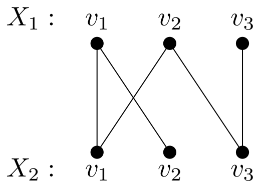
However this approach relies on enumerating tuples, therefore it is likely to scale badly when the arity of constraints grows. Moreover, the question of whether for some global constraints this reasoning can be done without enumerating tuples, is beyond the scope of this dissertation. We therefore limit our study of this method to binary constraint networks. Given a binary constraint network $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ we define $\mathcal{P} \times \mathcal{P} = (\mathcal{X}^\times, \mathcal{D}^\times, \mathcal{C}^\times)$ as follows:

Definition 14. *The reformulation $\mathcal{P} \times \mathcal{P}$ of the constraint network $\mathcal{P}$ is a triplet $(\mathcal{X}^\times, \mathcal{D}^\times, \mathcal{C}^\times)$ such that: The variables are unchanged $\mathcal{X}^\times = \mathcal{X}$. The domains $\mathcal{D}^\times$ are cross products of the original domains $\mathcal{D}$, minus all pairs of identical values. $\mathcal{C}^\times$ is such that the constraint graph is equal to that of $\mathcal{C}$, however the constraint relations are changed to accept a tuple $\langle\langle v_i, v_j\rangle, \langle v_k, v_l\rangle\rangle$ iff $\langle v_i, v_k\rangle, \langle v_i, v_l\rangle$ and $\langle v_j, v_k\rangle$ are all allowed tuples in the corresponding constraint in $\mathcal{C}$.*

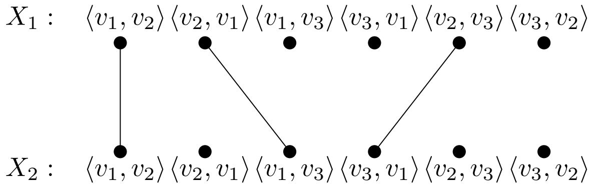
	$\mathcal{P}$	$\mathcal{P} \times \mathcal{P}$		
Variables:	$\mathcal{X}$	$\mathcal{X}^\times$	$=$	$\mathcal{X}$
Domains:	$\mathcal{D}$	$\mathcal{D}^\times(X_i)$	$=$	$\mathcal{D}(X_i)^2 \setminus \{\langle v, v\rangle \mid v \in \mathcal{D}(X_i)\}$
Constraints:	$\mathcal{C}$	$\mathcal{C}^\times$	$=$	$\{C^\times(V) \mid C(V)\}$ s.t. $\langle\langle v_i, v_j\rangle, \langle v_k, v_l\rangle\rangle \in C^\times(V) \Leftrightarrow$ $\langle v_i, v_k\rangle \in C(V) \wedge \langle v_i, v_l\rangle \in C(V) \wedge \langle v_j, v_k\rangle \in C(V)$

Table 4.2: The $\mathcal{P} \times \mathcal{P}$ reformulation summary.

Example 12. *We give an example of the $\mathcal{P} \times \mathcal{P}$ reformulation in Figure 4.5. The first figure (Figure 4.5a) shows the microstructure of a single constraint $C(X_1, X_2)$ between two variables X_1 and X_2. Every arc corresponds to a tuple allowed by the constraint, i.e., there is an arc between v_i and v_j if and only if the tuple $\langle v_i, v_j\rangle$ belongs to $C(X_1, X_2)$. The same constraint, after reformulation into $\mathcal{P} \times \mathcal{P}$ is shown in Figure 4.5b. The variables domains are changed and the arcs are changed accordingly.*



(a) A constraint in $\mathcal{P}$



X_1 :
$\langle v_1, v_2\rangle$ $\langle v_2, v_1\rangle$ $\langle v_1, v_3\rangle$ $\langle v_3, v_1\rangle$ $\langle v_2, v_3\rangle$ $\langle v_3, v_2\rangle$
X_2 :
$\langle v_1, v_2\rangle$ $\langle v_2, v_1\rangle$ $\langle v_1, v_3\rangle$ $\langle v_3, v_1\rangle$ $\langle v_2, v_3\rangle$ $\langle v_3, v_2\rangle$

(b) The reformulation of this constraint in $\mathcal{P} \times \mathcal{P}$

Figure 4.5: A constraint $C(X_1, X_2)$ in $\mathcal{P}$ and its reformulation $C^\times(X_1, X_2)$ in $\mathcal{P} \times \mathcal{P}$.

Theorem 21. *The reformulation $\mathcal{P} \times \mathcal{P}$ is sound and complete and has a space complexity in $\mathcal{O}(nd^2 + cmd^{2c})$.*

Proof. We show that the solutions of $\mathcal{P} \times \mathcal{P}$ projected onto $\mathcal{X}$ are exactly $(1, 0)$-*super*-solutions of $\mathcal{P}$.

Soundness: Consider a solution $f : \mathcal{X} \mapsto \Lambda^2$ of the reformulation $\mathcal{P} \times \mathcal{P}$, and let $g : \mathcal{X} \mapsto \Lambda$ be the solution of $\mathcal{P}$ such that a variable is assigned the first element of the tuple assigned by f, i.e., $g(X_i) = f(X_i)[1]$. We show that g is a $(1, 0)$-*super*-solution. Clearly, g is solution as $\langle g(X_i), g(X_j)\rangle \in C^\times(X_i, X_j) \Rightarrow \langle g(X_i)[1], g(X_j)[1]\rangle \in C(X_i, X_j)$. Now we need to show that for every variable there exists an alternative assignment that satisfies all constraints. Consider a variable X_i and all constraints $C(X_i, Y)$ over this variable on $\mathcal{P}$. By definition of $\mathcal{P} \times \mathcal{P}$, we have $\langle f(X_i)[2], f(Y)[1]\rangle \in C(X_i, Y)$, moreover, we have $f(X_i)[1] \neq f(X_i)[2]$. Since we assumed that $f(Y)[1] = g(Y)$, the second element of the tuple assigned to X_i in $\mathcal{P} \times \mathcal{P}$ is thus a valid alternative for $g(X_i)$ in $\mathcal{P}$.

Completeness: Let g be a $(1, 0)$-*super*-solution of $\mathcal{P}$, for every variable X_i there is an alternative value to the assignment $g(X_i)$ (say v_i). We show that $f : X_i \rightarrow \langle g(X_i), v_i\rangle$ is a solution of $\mathcal{P} \times \mathcal{P}$. Consider a constraint $C^\times(X_i, X_j)$ of $\mathcal{P} \times \mathcal{P}$, we have $\langle g(X_i), g(X_j)\rangle \in C(X_i, X_j)$. Moreover, since v_i (resp. v_j) can be substituted to $g(X_i)$ (resp. $g(X_j)$), we also have $\langle v_i, g(X_j)\rangle \in C(X_i, X_j)$ (resp. $\langle g(X_i), v_j\rangle \in C(X_i, X_j)$). Therefore, by definition of $C^\times(X_i, X_j)$ we have $\langle f(X_i), f(X_j)\rangle \in C^\times(X_i, X_j)$.

Space Complexity: The number of variables and the number of constraints do not change, whilst the domains size are squared (d^2). Therefore the space complexity of this reformulation is in $\mathcal{O}(nd^2 + md^{2c})$. □

We are now in a position to prove Theorem 17, from Chapter 3, using this reformulation.

(Proof of Theorem 17). Using this construction we can prove that deciding if a constraint network whose constraint graph is a tree accepts a existential-$(1, 0)$-*super*-solution can be done in polynomial time. Indeed, given a constraint network $\mathcal{P}$, the $\mathcal{P} \times \mathcal{P}$ reformulation does not change the constraint graph, hence is a tree if and only if the constraint graph of $\mathcal{P}$ is a tree. Therefore enforcing GAC is enough to solve this problem, and this can be done in $\mathcal{O}(md^4)$, since the GAC closure on $\mathcal{P}$ can be achieved in $\mathcal{O}(md^2)$. □

4.3 Local Consistencies for Robustness

In this section we introduce two local consistencies (GAC+ and *super*-GAC) that can be used in MAC-like algorithms to efficiently find $(1,0)$-*super*-solutions. We relate these two consistencies to the notion of **multiconsistency** introduced in [Elbassioni 05].

4.3.1 Arc Consistency Extended: GAC+

If a constraint network accepts a $(1,0)$-*super*-solution, then by definition, every variable has a valid alternative assignment, i.e., at least two values in its domain are consistent with the rest of the *super*-solution. Therefore, given a constraint network $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ closed under generalised arc consistency, if there exists a variable $X \in \mathcal{X}$ such that $|\mathcal{D}(X)| \leq 1$ then $\mathcal{P}$ has no $(1,0)$-*super*-solution. We define GAC+ for a constraint network as follows:

Definition 15. *A constraint network* $cn = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ *is* GAC+ *if and only if it is* GAC *and there is no variable* $X \in \mathcal{X}$ *whose domain's* GAC *closure is a singleton.*

Notice that this consistency method is not limited to binary constraint networks in any way. Indeed, the definition is based on generalised arc consistency and does not make any assumption about the constraint arity.

Theorem 22. GAC+ *is sound.*

Proof. We show that if a constraint network $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ admits a $(1,0)$-*super*-solution f, then there exists a domain relation $\mathcal{D}'$ included in $\mathcal{D}$, i.e., $\forall X \in \mathcal{X},\ \mathcal{D}'(X) \subseteq \mathcal{D}(X)$, such that $\mathcal{P}' = (\mathcal{X}, \mathcal{D}', \mathcal{C})$ is GAC+ and f is a solution of $\mathcal{P}'$.

Let f be a $(1,0)$-*super*-solution of $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$. It therefore is a solution, hence any assignment $X = f(X)$ is GAC. Moreover, since f is a $(1,0)$-*super*-solution, we know that for every variable $X \in \mathcal{X}$, there exists another solution g such that $f(X) \neq g(X)$. Since g is also a solution, the generalised arc consistent domain of X contains both $f(X)$ and $g(X)$. The same reasoning can be done for all variables in $\mathcal{X}$. As a consequence, we can construct a domain relation $\mathcal{D}'$ containing, for a variable X_i, both $f(X_i)$ and $g_i(X_i)$, where g_i is a 0-*repair* for the breakage $\{X_i\}$. Observe that f is a $(1,0)$-*super*-solution of $(\mathcal{X}, \mathcal{D}', \mathcal{C})$. □

Example 13. *Consider the constraint $X < Y$ and suppose that X and Y both take value in $\{0, 1\}$. The* GAC *closure of this constraint grounds X to 0 and Y to 1, hence it is not* GAC+.

4.3.2 Super Arc Consistency: *super*-GAC

We now define the concept of *super*-GAC. Given a constraint $C(V)$, an assignment $X = v$ is *super*-consistent with respect to a constraint $C(V)$, if and only if it can be extended to a $(1, 0)$-*super*-solution of $C(V)$. As for GAC we first define the notion for an assignment and lift this notion to variables, constraints and finally constraint networks. Notice that this process is not straightforward since a variable can be *super*-GAC whilst not every value in its domain is so. When computing the closure of a variable, a difference needs to be made between the values that are *super*-GAC and those that are not.

Definition 16. *Given a constraint network $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$, and a constraint $C(V) \in \mathcal{C}$, where $X \in V$, an assignment $X = v$ is super-*GAC *if and only if there exists an assignment σ of V such that $\sigma(X) = v$ and σ is a $(1, 0)$-super-solution of the constraint network defined by $(V, \mathcal{D}, \{C(V)\})$.*

*A variable is super-*GAC *if and only if all values in its domain are either super-*GAC *or participate in a 0-repair of super-*GAC *support. A constraint is super-*GAC *if and only if all variables in its scope are super-*GAC*. A constraint network is super-*GAC *if and only if all its constraints are super-*GAC*.*

As for GAC+, the definition does not make any assumption about the constraint arity. However, we shall see that the closure algorithm we propose for this local consistency is restricted to binary constraint networks.

Theorem 23. *super*-GAC *is sound.*

Proof. We show that if a constraint network $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ admits a $(1, 0)$-*super*-solution f, then there exists a domain relation $\mathcal{D}'$ included in $\mathcal{D}$, i.e., $\forall X \in \mathcal{X},\ \mathcal{D}'(X) \subseteq \mathcal{D}(X)$, such that $\mathcal{P}' = (\mathcal{X}, \mathcal{D}', \mathcal{C})$ is *super*-GAC and f is a solution of $\mathcal{P}'$.

Let f be a $(1, 0)$-*super*-solution of $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$. We construct the same domain as in the soundness proof for GAC+: For every variable $X_i \in \mathcal{X}$, $\mathcal{D}'(X_i)$ will contain $f(X_i)$ and $g_i(X_i)$, where g_i is a 0-*repair* for the breakage $\{X_i\}$.

Now, for every constraint $C(V)$, if we restrict the network to $(V, \mathcal{D}', \{C(V)\})$, the corresponding restriction $f|_V$ of f is still a $(1,0)$-*super*-solution. Indeed, for any variable X_i the restriction $g_i|_V$ of g_i is a solution of $(V, \mathcal{D}', \{C(V)\})$. As a consequence, any value of any variable either participates in a *super*-GAC support or a 0-*repair* of such a support. $\square$

Example 14. *Consider the constraint $X \neq Y$ and suppose that X takes a value in $\{0,1\}$ whilst Y takes a value in $\{0,1,2\}$. There are two $(1,0)$-super-solutions for this constraint: $\langle 0,2\rangle$ and $\langle 1,2\rangle$. Therefore, the assignments $Y = 0$ and $Y = 1$ are not super-*GAC*. However, consider the first $(1,0)$-super-solution, $\langle 0,2\rangle$: A breakage on Y is repaired by the 0-repair $\langle 0,1\rangle$. Similarly, a breakage of $\langle 1,2\rangle$ on Y can be repaired by $\langle 1,0\rangle$. Consequently, although some assignments are not super-*GAC*, the variables X and Y are super-*GAC*, as well as the constraint $X \neq Y$.*

To take this subtlety into account, we introduce, when defining a closure algorithm for this consistency, two domain relations to replace the classical domain $\mathcal{D}$.

4.3.3 k-multiconsistency

In [Elbassioni 05] the authors introduce k-multiconsistency, a generalisation of generalised arc consistency. Given a constraint $C(V)$, an assignment $X = v$ is k-multiconsistent if and only if it has k distinct supports. This notion may be used as a value ordering heuristic, as choosing a value with multiple supports may lead to a solution more quickly. Moreover, it is linked to the notion of robustness as defined in this dissertation since the extra supports are in fact alternative assignments. We recall the definition from [Elbassioni 05]:

Definition 17. *A unary assignment $X = v$ is* k-multiconsistent *on a constraint $C(V)$ iff there exist k distinct supports for $X = v$ on $C(V)$.*

The notion of k-multiconsistency and $(1,0)$-*super*-solutions are closely related. However, as pointed out in [Elbassioni 05], an assignment such that all values are 2-multiconsistent is not necessarily a $(1,0)$-*super*-solution.

Example 15. *For instance consider the variables $X_1 \in \{1,2\}$, $X_2 \in \{1,2\}$, $X_3 \in \{3,4\}$ and $X_4 \in \{3,4\}$ subject to an* AllDifferent *constraint, X_1, X_2, X_3 and X_4 must be* *assigned* 4 *distinct values. The complete list of solutions is as follows:*

$$\langle 1,2,3,4\rangle \ \langle 1,2,4,3\rangle \ \langle 2,1,3,4\rangle \ \langle 2,1,4,3\rangle$$

All unary assignments have two distinct supports hence any solution is 2*-multiconsistent, yet there is no* $(1,0)$*-super-solution.*

Multiconsistency therefore is weaker than *super*-GAC. Moreover, in the previous example, any of the GAC tuples are 2-multiconsistent, therefore in a search algorithm using multiconsistency, we must check that the solution is indeed a $(1,0)$-*super*-solution at each leaf of the search tree. In this dissertation we do not study an algorithm based on this method, however such algorithm may be considered for future work.

4.4 Algorithm for Achieving Closure

4.4.1 Differences with Classical Closures

Local consistencies are usually defined over values or assignments, and then lifted to variables, constraints and a constraint network by computing the closure. For instance, a variable X_i is GAC iff all values in X_i are GAC a constraint is GAC iff all its constrained variables are GAC and finally a constraint network is GAC iff all constraints are GAC. When a value is arc inconsistent, a constraint forbidding it $(X \neq v)$ can be added to the constraint network without removing any solution. This constraint will result in pruning in the context of a search algorithm. However, in the case of *super*-GAC or multiconsistency values that are essential for providing support may not themselves be *super*-GAC or multiconsistent. Therefore, posting such a constraint for an assignment that is not multiconsistent or *super*-GAC may remove multiconsistent solutions or $(1,0)$-*super*-solutions. In [Elbassioni 05] the notion of multiconsistency is defined only for values, and the authors study the problem of finding an assignment such that all values are k-multiconsistent. However, this notion is not explored in the context of search and therefore no closure property is given.

Example 16. *We give an example to show that one cannot compute the* 2-multiconsistent *closure in a similar way as for* GAC, *i.e., by iteratively pruning inconsistent values.*

Indeed, such a closure over a constraint network $\mathcal{P}$ can be empty even though a 2-multiconsistent *assignment exists. Consider the following problem $\mathcal{P}$:*

$$X, Y \in \{1, 2\} \; X \leq Y$$

The tuple $\langle 1, 2\rangle$ is 2*-multiconsistent since $X = 1$ (resp. $Y = 2$) has two supports on Y (resp. on X). However, if we compute the closure, $X = 2$ and $Y = 1$ are first pruned as they have only one support. Then, after these removals, $X = 1$ and $Y = 2$ are no longer* 2*-multiconsistent, therefore the closure is empty. Similarly, even though there exists a* $(1, 0)$*-super-solution ($\langle 1, 2\rangle$) the super-*GAC *closure is empty: Since $X = 2$ does not participate in a* $(1, 0)$*-super-solution one may want to add the constraint $X \neq 2$, however, there is no* $(1, 0)$*-super-solution to the resulting constraint network.*

We shall see that *super*-GAC can be enforced by computing not one but two inter-related closures. One will contain all values that may be part of a $(1, 0)$-*super*-solution whilst the second may also contain values contributing only to the repair of a $(1, 0)$-*super*-solution. These two closures are defined with respect to each other and can thus only be computed together. The closure algorithm that we introduce in this section are restricted to binary constraint networks. Indeed, though the GAC+ closure algorithm could easily be extended to non-binary constraints, this is not the case for the *super*-GAC algorithm. The reason is that we use the same observation made in Section 4.2.2, i.e., for a tuple of values to be a $(1, 0)$-*super*-solution over two variables, they must each have two distinct supports, and support each other. The situation is more complex, although similar, for non-binary constraints. Since we aim at comparing these approaches, we restrict our study to the binary case for all consistency and reformulation methods, even though GAC+ and to a lesser extent GAC($\mathcal{P} + \mathcal{P}$) can easily deal with non-binary and global constraints.

4.4.2 GAC+ **Closure**

The GAC+ closure is very similar to the GAC closure.

Definition 18. *A domain relation $\mathcal{D}$ is closed under* GAC+ *if and only if it is closed under* GAC *and for any variable X, the cardinality of $\mathcal{D}(X)$ is at least 2.*

Defining a closure algorithm is therefore straightforward as it only requires to check the extra condition stating that no domain can be singleton. However, devising a closure algorithm that can be used within a backtrack search is slightly more tricky. Indeed, we defined the Maintain Arc Consistency algorithm (MAC, Algorithm 1) as a procedure iteratively reducing the current domains, or partial solution, φ and computing the GAC closure. As a grounded variable has, by definition, a singleton domain, the closure method used during search cannot be as simple as the stand alone algorithm suggested above. To solve this apparent contradiction, when computing the GAC+ closure during search, it is convenient to consider not one, but two partial solutions i.e., set of domains. We thus use two mappings $s\mathcal{D}(X)$ and $r\mathcal{D}(X)$, standing for *super*-domain and *repair*-domain. The former is in every respect equivalent to the partial solution, usually denoted φ. The latter, $r\mathcal{D}$, is used to keep track, for grounded variables, of valid alternative values. The GAC+ closure within a backtracking procedure, can thus be defined using $s\mathcal{D}$ and $r\mathcal{D}$ and the following constraints, for any variable $X \in \mathcal{X}$:

$$\text{GAC}(s\mathcal{D}) \tag{4.1}$$

$$v \in r\mathcal{D}(X) \Leftrightarrow (\forall Y \in \mathcal{X},\ \exists w \in s\mathcal{D}(Y)\ s.t.\ \langle v, w\rangle \in C(X,Y)) \tag{4.2}$$

$$|r\mathcal{D}(X)| \geq 2 \tag{4.3}$$

The first and third conditions enforce the closure defined in this section, whilst the second condition makes sure that $r\mathcal{D}$ contain all values consistent with the current search assignment.

Example 17. *For instance consider the following constraint network:*

$$\mathcal{X} = \{X, Y\}, \mathcal{D}(X) = \{1,2,3\}, \mathcal{D}(Y) = \{1,2,3\}, \mathcal{C} = \{(X + Y > 2)\}$$

This constraint network is generalised arc consistent. Now consider the search for a solution. In the first branch, the `MAC` *algorithm might commit to the assignment $X = 1$, hence reduce the partial solution as follows $\varphi(X) = \{1\}$, and compute the* GAC*closure leading to:*

$$\varphi(X) = \{1\},\ \varphi(Y) = \{2,3\}$$

Whilst the corresponding GAC+ *closure, using $s\mathcal{D}$ and $r\mathcal{D}$, is:*

$$s\mathcal{D}(X) = \{1\},\ s\mathcal{D}(Y) = \{2,3\},\ r\mathcal{D}(X) = \{1,2,3\},\ r\mathcal{D}(Y) = \{2,3\}$$

We give the pseudo-code for a GAC+ closure algorithm in Figure 4.6. This algorithm is very similar to the classical GAC algorithm. The main difference is that instead of the usual partial domain relation φ, this algorithm computes the closure on two domain relations $s\mathcal{D}$ and $r\mathcal{D}$. A second difference is that a failure occurs if a domain becomes singleton after propagation (Line 1). The procedure `propagate` (Algorithm 7) is slightly modified so as to perform the same pruning on both relations (lines 1 and 2). However, observe that $s\mathcal{D}$ and $r\mathcal{D}$ may differ since they are not reduced in the same way during the search step of a backtrack algorithm. When used statically (that is, outside of a search process) Algorithm 6 does nothing more than GAC apart from checking for singleton domains (and consequently failing).

Algorithm 6 GAC+

```
    Data   : $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$, $s\mathcal{D}[= \mathcal{D}]$, $r\mathcal{D}[= \mathcal{D}]$
    Result : $s\mathcal{D}, r\mathcal{D}$ (the GAC+ closure of $\mathcal{P}$)
    $Q \leftarrow \mathcal{C} \cup \{C(Y, X) \mid C(X, Y) \in \mathcal{C}\}$;
    while $Q \neq \emptyset$ do
        select and delete any $C(X_i, X_j)$ from $Q$;
        $pruned \leftarrow$ propagate+($C(X_i, X_j)$, $s\mathcal{D}$, $r\mathcal{D}$);
1       if $s\mathcal{D}(X_j) = \emptyset \vee |r\mathcal{D}(X_j)| \leq 1$ then return false;
        if $pruned$ then $Q \leftarrow Q \cup \{C(X_j, X_k) \mid C(X_j, X_k) \in \mathcal{C}\}$;
    return true;
```

Algorithm 7 `propagate+`

```
    Data   : $C(X_i, X_j)$, $s\mathcal{D}$, $r\mathcal{D}$
    Result : The GAC+ closure of $X_j$ with respect to $C(X_i, X_j)$
    $pruned \leftarrow$ false;
    foreach $w \in r\mathcal{D}(X_j)$ do
        if $\nexists v \in s\mathcal{D}(X_i)$ $s.t.$ $\langle v, w \rangle \in C(X_i, X_j)$ then
1           $s\mathcal{D}(X_j) \leftarrow s\mathcal{D}(X_j) \setminus \{w\}$;
2           $r\mathcal{D}(X_j) \leftarrow r\mathcal{D}(X_j) \setminus \{w\}$;
            $pruned \leftarrow$ true;
    return $pruned$;
```

Figure 4.6: An algorithm for computing the GAC+ closure of a constraint network based on `AC3`.

4.4.3 *super*-GAC **Closure**

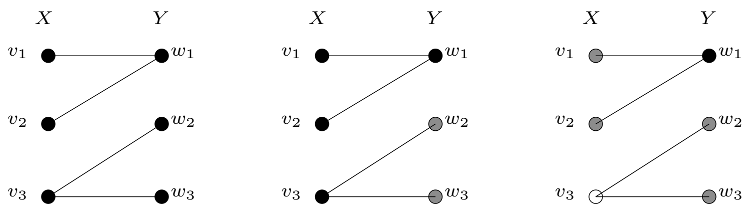
In order to define a closed form that preserves all $(1,0)$-*super*-solutions, we introduce an intermediate degree of consistency. The usual domains $\mathcal{D}$ will here again be replaced by two mappings $s\mathcal{D}(X)$ and $r\mathcal{D}(X)$. The *super*-GAC domain $s\mathcal{D}(X)$ will contain all *super*-GAC values of X whilst $r\mathcal{D}(X)$ will contain all values that participate in a 0-*repair* of a $(1,0)$-*super*-solution but not necessarily to the *super*-solution itself. As for generalised arc consistency, computing this closure for constraints with arbitrary large arity may be difficult. However we shall see that for binary constraints, this closure can be achieved in the same time complexity as for regular arc consistency. Initially $s\mathcal{D}, r\mathcal{D}$ and $\mathcal{D}$ are equal, then we enforce the following rules until a fixed point is reached:

1. A value v is in $s\mathcal{D}(X)$ iff for every $C(X,Y) \in \mathcal{C}$ there exists $w \in s\mathcal{D}(Y)$ and $r \in r\mathcal{D}(Y)$, $r \neq w$ such that $\langle v, w\rangle$ and $\langle v, r\rangle$ are allowed tuples.
2. A value v is in $r\mathcal{D}(X)$ iff for every $C(X,Y) \in \mathcal{C}$ there exists $w \in s\mathcal{D}(Y)$ such that $\langle v, w\rangle$ is an allowed tuple.

Once $r\mathcal{D}(X)$ and $s\mathcal{D}(X)$ can no longer be reduced by rule 1 or 2, then $s\mathcal{D}$ maps variables to *super*-GAC assignments.

We introduce an algorithm for computing the *super*-GAC closure of a constraint network. Algorithm 8 is inspired by `AC3` hence has a non-optimal time complexity. We therefore give another version of the same algorithm (Algorithm 10) based on `AC4`. This algorithm is the first optimal arc consistency closure procedure proposed in the literature by Mohr and Henderson ([Mohr 86]). We detail the complexity of these algorithms in Section 4.5.5.2. The procedure managing the constraint queue is identical to the GAC closure procedure (Algorithm 2), described in Chapter 2. The only difference is again the pair of domain relations replacing the classical domain. The procedure `propagate-sup` (Algorithm 9), on the other hand, is very different. This procedure revises the super and repair domains of the variable X_j with respect to the constraint $C(X_i, X_j)$. For every value w in the repair domain of X_j, we compute the subset of $r\mathcal{D}(X_i)$ participating to a GAC support of $X_j = w$ (loop 1). Then, if this set of support has a cardinality less than 2, we deduce that this value cannot participate in a $(1,0)$-*super*-solution, hence we remove it from $s\mathcal{D}(X_j)$ (Line 3). In the next step, we check if there is a supporting value for $X = w$ in $s\mathcal{D}(X_i)$. If not then the assignment $X = w$ is removed from $r\mathcal{D}(X_j)$ (Line 5). Finally, since the relation $s\mathcal{D}(X_j) \subseteq r\mathcal{D}(X_j)$ should always hold, we set $s\mathcal{D}(X_j)$ to the intersection of both domains (Line 6).

Example 18. *We give an example of the closure computed with super-`AC` in Figure 4.7. Consider the constraint network involving two variables X and Y, with respectively domains $\{v_1, v_2, v_3\}$ and $\{w_1, w_2, w_3\}$. The allowed tuples correspond to edges in Figure 4.7a. The super and repair domains are initially equal to the original domains. In the first iteration (Figure 4.7b), the constraint $C(X, Y)$ is propagated. The assignments $Y = w_2$ and $Y = w_3$ have only one support in $r\mathcal{D}(X)$ and $s\mathcal{D}(X)$, therefore they are removed from $s\mathcal{D}(Y)$. We represent values that belong to both super and repair domain in black, values that belong only to the repair domain in grey and values that do not belong to either domain in white.*



(a) Initial state (b) Iteration 1 (c) Iteration 2

Figure 4.7: A constraint over two variables.

In the second iteration (Figure 4.7c), the constraint $C(Y, X)$ is propagated. The assignments $X = v_1$ and $X = v_2$ have only one support in $r\mathcal{D}(Y)$ and $s\mathcal{D}(Y)$, therefore they are removed from $s\mathcal{D}(Y)$. Moreover, the assignment $X = v_3$ has no support in $s\mathcal{D}(Y)$, therefore it is removed from both $r\mathcal{D}(X)$ and $s\mathcal{D}(X)$. Hence, $s\mathcal{D}(X)$ is empty and the algorithm fails.

We briefly explain the `AC4` version of the *super*-GAC closure algorithm (*super*-GAC4). First, the internal structures used to remember supporting values on neighbouring variables are initialised in Algorithm 11. The structure $S[i, v]$ is initialised to the set of pairs (j, w) such that the tuple $\langle X_i = v, X_j = w\rangle$ is consistent ($\langle v, w\rangle \in C(X_i, X_j)$), exactly as in the `AC4` algorithm. Furthermore, the structure $count_{sup}[(i, j), v]$ (resp.

Algorithm 8 *super*-`AC`

```
Data   : $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$, $s\mathcal{D}[= \mathcal{D}]$, $r\mathcal{D}[= \mathcal{D}]$
Result : $s\mathcal{D}, r\mathcal{D}$ (the super-GAC closure of $\mathcal{P}$)
$Q \leftarrow \mathcal{C} \cup \{C(Y, X) \mid C(X, Y) \in \mathcal{C}\}$;
while $Q \neq \emptyset$ do
    select and delete any $C(X_i, X_j)$ from $Q$;
    $pruned \leftarrow$ propagate-sup($C(X_i, X_j)$, $s\mathcal{D}$, $r\mathcal{D}$);
    if $s\mathcal{D}(X_j) = \emptyset$ then return false;
    if $pruned$ then $Q \leftarrow Q \cup \{C(X_j, X_k) \mid C(X_j, X_k) \in \mathcal{C}\}$;
return true;
```

Algorithm 9 `propagate-sup`

```
    Data   : $C(X_i, X_j)$, $s\mathcal{D}$, $r\mathcal{D}$
    Result : The super-GAC closure of $X_j$ with respect to $C(X_i, X_j)$
    $pruned \leftarrow$ false;
    foreach $w \in r\mathcal{D}(X_j)$ do
        $sup \leftarrow \emptyset$;
1       foreach $v \in r\mathcal{D}(X_i)$ do
            if $\langle v, w \rangle \in C(X_i, X_j)$ then $sup \leftarrow sup \cup \{v\}$;
2       if $|sup| < 2$ then
3           $s\mathcal{D}(X_j) \leftarrow s\mathcal{D}(X_j) \setminus \{w\}$;
            $pruned \leftarrow$ true;
4       if $sup \cap s\mathcal{D}(X_i) = \emptyset$ then
5           $r\mathcal{D}(X_j) \leftarrow r\mathcal{D}(X_j) \setminus \{w\}$;
            $pruned \leftarrow$ true;
6   $s\mathcal{D}(X_j) \leftarrow s\mathcal{D}(X_j) \cap r\mathcal{D}(X_j)$;
    return $pruned$;
```

Figure 4.8: An algorithm for computing the *super*-GAC closure of a constraint network based on `AC3`.

$count_{rep}[(i,j),v]$) stores the number of supporting values for $X_i = v$ in the super domain (resp. repair domain) of X_j. Then the counters are checked, and there are two types of outcome:

1. $count_{sup}[(i,j),v] = 0$ (Algorithm 11 line 1 and 2): The assignment $X_i = v$ has no GAC support in $s\mathcal{D}(X_j)$. It means that $X_i = v$ can neither participate in a $(1,0)$-*super*-solution nor in a *repair*. Therefore it is removed from both super and repair domains.

2. $count_{rep}[(i,j),v] < 2$ (Algorithm 11 Line 2): The assignment $X_i = v$ has only one GAC support in $s\mathcal{D}(X_j)$ and $r\mathcal{D}(X_j)$. It means that $X_i = v$ cannot in a $(1,0)$-*super*-solution, whilst it can still possibly participate in a *repair*.

Now, to propagate these events, we use two queues Q_{rep} and Q_{sup}. The first, Q_{rep}, contains the values that have been removed from their repair domain, whilst Q_{sup} contains the values that have been removed from their super domain. In Algorithm 10 the events in the queues Q_{rep} and Q_{sup} are recursively propagated. In the first case (Line 1), v was removed from $r\mathcal{D}(X_i)$, hence only values in $s\mathcal{D}(X_j)$ can be affected, as values in $r\mathcal{D}(X_j)$ only require a support in $s\mathcal{D}(X_i)$. In the second case (Line 2), v was removed from $s\mathcal{D}(X_i)$. If a value has no GAC support in $s\mathcal{D}(X_i)$ it must be removed from both $s\mathcal{D}(X_j)$ and $r\mathcal{D}(X_j)$.

4.5 Theoretical Properties:

Theorem 24. *The closure algorithm* GAC+ *is sound and complete and runs in* $\mathcal{O}(md^2)$ *on binary constraint networks.*

Proof. **Soundness:** We show that if a value is GAC+, then the GAC+ closure algorithm does not prune it. A value v is GAC+ only if it is GAC and there is no singleton domains in the GAC closure. Since the procedure GAC+ prunes a value only if it does not have a GAC support for a given constraint, the first condition holds. Moreover the second condition is clearly satisfied since the only other condition for failure is precisely a domain becoming singleton.

Completeness: We show that if a value is not GAC+, then the GAC+ closure algorithm prunes it. A value v may not be GAC+ either because it is not GAC or

Algorithm 10 *super*-GAC4

```
    Data   : $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$, $s\mathcal{D}[= \mathcal{D}]$, $r\mathcal{D}[= \mathcal{D}]$
    Result : $s\mathcal{D}, r\mathcal{D}$ (the super-GAC closure of $\mathcal{P}$)
    $\langle Q_{sup}, Q_{rep} \rangle \leftarrow$ initialise($\mathcal{P}$);
    while $|Q_{sup}| + |Q_{rep}| > 0$ do
        if $|Q_{rep}| > 0$ then
1           select and delete a pair $(i, v)$ from $Q_{rep}$;
            foreach $(j, w) \in S[i, v]$ do
                $count_{rep}[(j, i), w] \leftarrow count_{rep}[(j, i), w] - 1$;
                if $count_{rep}[(i, j), v] < 2$ then
                    $s\mathcal{D}(X_j) \leftarrow s\mathcal{D}(X_j) \setminus \{w\}$;
                    $Q_{sup} \leftarrow Q_{sup} \cup \{(j, w)\}$;
        else
2           select and delete a pair $(i, v)$ from $Q_{sup}$;
            $count_{sup}[(j, i), w] \leftarrow count_{sup}[(j, i), w] - 1$;
            if $count_{sup}[(j, i), w] = 0$ then
                $r\mathcal{D}(X_j) \leftarrow r\mathcal{D}(X_j) \setminus \{w\}$;
                $Q_{rep} \leftarrow Q_{rep} \cup \{(j, w)\}$;
                if $w \in s\mathcal{D}(X_j)$ then
                    $s\mathcal{D}(X_j) \leftarrow s\mathcal{D}(X_j) \setminus \{w\}$;
                    $Q_{sup} \leftarrow Q_{sup} \cup \{(j, w)\}$;
```

Algorithm 11 `initialise`($\mathcal{P}$)

```
    $\forall i, v\ S[i, v] \leftarrow 0$;
    $\forall i, j, v\ count_{sup}[(i, j), v] \leftarrow count_{rep}[(i, j), v] \leftarrow 0$;
    $Q_{sup} \leftarrow Q_{rep} \leftarrow \emptyset$;
    foreach $C(X_i, X_j) \in \mathcal{C}$ do
        foreach $\langle v, w \rangle \in C(X_i, X_j)$ do
            $S[i, v] \leftarrow S[i, v] \cup \{(j, w)\}$;
            $S[j, w] \leftarrow S[j, w] \cup \{(i, v)\}$;
            $count_{sup}[(i, j), v] \leftarrow count_{sup}[(i, j), v] + 1$;
            $count_{sup}[(j, i), w] \leftarrow count_{sup}[(j, i), w] + 1$;
            $count_{rep}[(i, j), v] \leftarrow count_{rep}[(i, j), v] + 1$;
            $count_{rep}[(j, i), w] \leftarrow count_{rep}[(j, i), w] + 1$;
1       if $count_{sup}[(i, j), v] = 0$ then
            $r\mathcal{D}(X_i) \leftarrow r\mathcal{D}(X_i) \setminus \{v\}$;
            $Q_{rep} \leftarrow Q_{rep} \cup \{(i, v)\}$;
2       if $count_{rep}[(i, j), v] < 2 \ \vee\ count_{sup}[(i, j), v] = 0$ then
            $s\mathcal{D}(X_i) \leftarrow s\mathcal{D}(X_i) \setminus \{v\}$;
            $Q_{sup} \leftarrow Q_{sup} \cup \{(i, v)\}$;
    return $\langle Q_{sup}, Q_{rep} \rangle$;
```

Figure 4.9: An algorithm for computing the *super*-GAC closure of a constraint network based on `AC4`.

because the GAC closure is such that the domain of a variable is reduced to a single value. If the value is not GAC then it will be pruned since GAC+ checks values for GAC supports. If the GAC closure is such that the domain of a variable is reduced to a singleton, this will be discovered when checking the domain size after pruning, in Line 1 of Algorithm 6.

Complexity: The worst case time complexity is equal to that of enforcing GAC. The further check on the domain size is in constant time, and can only happen when a value is pruned, hence $\mathcal{O}(nd)$ times. The worst case time complexity thus is equal to the GAC closure algorithm that we are using. Optimal closure algorithms for GAC run in $\mathcal{O}(md^2)$ on binary constraint networks. □

Theorem 25. *The closure algorithms super*-GAC *and super*-GAC4 *are sound and complete and super*-GAC4 *runs in* $\mathcal{O}(md^2)$ *on binary constraint networks.*

Proof. Notice that we restrict our proof to the propagation of one constraint. The iterative process over multiple constraints is essentially the same as in GAC closure algorithms, that is, a stack containing the changes that have not yet been taken into account.

Soundness: We prove that if a value belongs to a $(1,0)$-*super*-solution of the restriction of a constraint network to a single constraint, applying the closure algorithm *super*-GAC does not prune this value from the super domain. Consider a constraint $C(X_i, X_j) \in \mathcal{C}$, and let v, w be two values such that $\langle v, w\rangle$ is a $(1,0)$-*super*-solution of the constraint network $\mathcal{P} = (\{X_i, X_j\}, \mathcal{D}, \{C(X_i, X_j)\})$. By definition of $(1,0)$-*super*-solution, the three following conditions must hold:

- $\langle v, w\rangle \in C(X_i, X_j)$
- $\exists w' \in \mathcal{D}(X_j) \setminus \{w\},\ \text{s.t.} \langle v, w'\rangle \in C(X_i, X_j)$
- $\exists v' \in \mathcal{D}(X_j) \setminus \{v\},\ \text{s.t.} \langle v', w\rangle \in C(X_i, X_j)$

We consider the values v and w, and show that the AC3-based version of *super*-GAC does not remove them from $s\mathcal{D}(X_i)$ and $s\mathcal{D}(X_j)$. Since initially $s\mathcal{D} = r\mathcal{D} = \mathcal{D}$, the values v' and w' will not be removed from respectively $r\mathcal{D}(X_i)$ and $r\mathcal{D}(X_j)$, unless v and w are removed from $s\mathcal{D}(X_i)$ and $s\mathcal{D}(X_j)$. We consider the value w. The set *sup*, computed in Line 1 of Algorithm 9 contains at least v and v' so neither Condition 2 nor Condition 4 is triggered. Then when checking the constraint in the opposite direction, the same reasoning can be done for v. Hence v and w are not pruned from $s\mathcal{D}(X_i)$ and $s\mathcal{D}(X_j)$. The same property holds for the `AC4`-based version of *super*-GAC. Indeed, the counter $count_{rep}[(i,j),v']$ is greater than or equal to 1 since $\langle v',w\rangle \in C(X_i,X_j)$. Similarly the counter $count_{rep}[(i,j),w]$ is initialised to a value greater than equal to 2, and will stay positive as long as v' is not removed from $r\mathcal{D}(X_i)$.

Completeness: We prove that given a constraint $C(X_i,X_j)$, if a value v is not pruned from $s\mathcal{D}(X_i)$, then it participates in a $(1,0)$-*super*-solution of the constraint network $\mathcal{P} = (\{X_i,X_j\},\mathcal{D},\{C(X_i,X_j)\})$.

We first consider the `AC3`-based version of *super*-GAC. Since, after the algorithm reached a fixed point, we still have $v \in s\mathcal{D}(X_i)$, it implies (condition 2) that v has at least two GAC supporting values, say w and w', in $r\mathcal{D}(X_j)$. Moreover, at least one of these supporting values, say w, is in $s\mathcal{D}(X_j)$ (condition 4). Therefore by the same reasoning w also has at least two supports in $r\mathcal{D}(X_i)$, one of them is v, and let v' be another one. Clearly, the tuple $\langle v,w\rangle$ is a $(1,0)$-*super*-solution of $\mathcal{P} = (\{X_i,X_j\},\mathcal{D},\{C(X_i,X_j)\})$, and $\langle v,w'\rangle$ and $\langle v',w\rangle$ are the 0-*repair*s.

Now we consider the `AC4`-based version of *super*-GAC. After the initialisation phase, the number of supports are counted and since v is not pruned, we have $count_{rep}[(i,j),v] \geq 2$ and $count_{sup}[(i,j),v] \geq 1$. Now the counters are decreased if and only if a GAC supporting value of $X_i = v$ is removed from respectively $r\mathcal{D}(X_j)$ or $s\mathcal{D}(X_j)$. Since after a fixed point is reached, we assumed that $v \in s\mathcal{D}(X_i)$ still holds, it means that we still have $count_{rep}[(i,j),v] \geq 2$ and $count_{sup}[(i,j),v] \geq 1$. Hence v has at least two supporting values in $r\mathcal{D}(X_j)$ and at least one in $s\mathcal{D}(X_j)$. The same reasoning can be done to show that this last value also has two supports in $r\mathcal{D}(X_i)$. Hence we can construct the same $(1,0)$-*super*-solution as in the `AC3` case.

Complexity: Since the worst case time complexity of the `AC4`-based version is better, we restrict our result to this case. As in the classical `AC4`, all structures and counters are monotonically decreasing during the iterative process, and the updates are done in constant time. There are two sets of counters and lists, therefore the worst case time complexity is twice as large. In conclusion this algorithm runs in $\mathcal{O}(md^2)$, hence is optimal. $\square$

4.5.1 Notations for consistency comparison

We now formally study the local consistency properties introduced in Section 4.3, along with arc consistency on the two reformulations described earlier, applied to binary constraint networks. We first compare the filtering level and then the complexity of achieving the corresponding closure, using the framework described in Section 2.5.

We consider the following consistencies:

- GAC+ as defined in Section 4.3.1. The generalised arc consistent closure is computed. GAC+($\mathcal{P}$) holds iff the GAC closure is such that all domains contain at least 2 values.
- *super*-GAC as defined in Section 4.3.2. The closure over the whole constraint network is computed using Algorithm 8. *super*-GAC($\mathcal{P}$) holds iff for every variable $X \in \mathcal{X}$, we have $s\mathcal{D}(X) \neq \emptyset$.
- GAC($\mathcal{P}$+$\mathcal{P}$): We compute the GAC closure on $\mathcal{P}$+$\mathcal{P}$, obtained by reformulation of the constrain network $\mathcal{P}$. GAC($\mathcal{P} + \mathcal{P}$) holds iff this closure is not empty.
- GAC($\mathcal{P}\times\mathcal{P}$): We compute the GAC closure on $\mathcal{P}\times\mathcal{P}$, obtained by reformulation of the constrain network $\mathcal{P}$. GAC($\mathcal{P} \times \mathcal{P}$) holds iff this closure is not empty.

Notice that in the proofs below we always consider constraint networks where all variables are constrained ($X \in \mathcal{X} \Rightarrow \exists Y\ C(X, Y) \in \mathcal{C}$). Although this is required for the proof to be correct, it could easily be worked around by considering domains as unary constraints and adapting the consistency and closure definitions accordingly.

4.5.2 Static vs. Dynamic context

Local consistencies can have rather different behaviour when seen statically, i.e., as stand alone procedures, or dynamically, i.e., as part of a backtrack search algorithm. For instance certain local consistency algorithms may have amortised time complexity along one branch of the search tree, whilst other are essentially repeating all the work done in the previous step. With the local consistencies introduced in this chapter, the difference is even more acute. In fact, the critical point is that decisions, or assignments made during search are not exactly akin to domain reduction in the strict sense. We work with two domain relations, $s\mathcal{D}$ and $r\mathcal{D}$ instead of a unique partial solution φ, and whilst filtering algorithms manipulate both domains, decisions are always reduction of $s\mathcal{D}$, and never affect directly $r\mathcal{D}$. For instance, as seen in Section 4.4.2, the GAC+ closure algorithm never prune $s\mathcal{D}$ and $r\mathcal{D}$ differently, yet these relation can possibly be different during search, since the decisions are made on $s\mathcal{D}$ and not on $r\mathcal{D}$. We first compare the filtering level of the various local consistencies in the classical, static context, then we show that the relation between GAC+ and the GAC on the $\mathcal{P}+\mathcal{P}$ reformulation changes if used within or outside a search algorithm.

4.5.3 Filtering Level: static context

Using the notation defined in Section 2.5, the relation between the different consistency properties introduced in Section 4.3 is as follows:

Theorem 26 (Static context)**.**

$$\text{GAC+} \simeq \text{GAC}(\mathcal{P}+\mathcal{P}) \tag{4.4}$$

$$\textit{super}\text{-GAC} \succ \text{GAC+} \tag{4.5}$$

$$\text{GAC}(\mathcal{P}\times\mathcal{P}) \succ \textit{super}\text{-GAC} \tag{4.6}$$

Proof. Let $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ be a constraint network, we prove the three propositions above:

Proposition 4.4:

(GAC+ $\succeq$ GAC($\mathcal{P}+\mathcal{P}$)) Suppose that GAC+($\mathcal{P}$) holds, $\mathcal{D}$ is closed under GAC, and $\mathcal{D}$ is such that $|\mathcal{D}(X)| > 1\ \forall X \in \mathcal{X}$. Now consider the reformulation $\mathcal{P}+\mathcal{P}$. The original constraints are GAC since $\mathcal{P}$ is GAC. The duplicated constraints are GAC since they are identical to the original ones, and the domains of a variable X and its duplicate X^+ are identical. The not-equals constraints between original and duplicated variables are GAC since any variable has at least 2 values.

(GAC($\mathcal{P}+\mathcal{P}$) $\succeq$ GAC+) Suppose that $\mathcal{P}$ is not GAC+, then in the generalised arc consistent closure of $\mathcal{P}$, there exists at least one variable X_i such that $|\mathcal{D}(X_i)| \leq 1$. Since $\mathcal{P}+\mathcal{P}$ contains $\mathcal{P}$, we have $|\mathcal{D}(X_i)| \leq 1$ as well when computing the GAC closure.

We suppose, without loss of generality, that $\mathcal{D}(X_i) = \{v\}$, and we consider the duplicate X_i^+ of X_i. Since X_i^+ and X_i are linked to the same neighbouring variables with the same constrains, any value that is not GAC for X_i cannot be so for X_i^+. Now consider the assignment $X_i^+ = v$, it has no support on X_i since $\langle v, v \rangle$ does not satisfy $X_i \neq X_i^+$. Therefore, the GAC closure is empty.

Proposition 4.5:

(*super*-GAC $\succeq$ GAC+) Suppose that GAC+ does not hold. Then there exists a variable X whose domain contains at most one arc consistent value. Since the second rule for enforcing *super*-GAC requires a GAC support in $s\mathcal{D}$ for any value in $r\mathcal{D}$, all arc inconsistent values will be pruned from $r\mathcal{D}(X)$. Now consider a constraint $C(X,Y)$, any value in $s\mathcal{D}(Y)$ has at most one support in $s\mathcal{D}(X)$ for $C(X,Y)$, hence can be removed from $s\mathcal{D}(Y)$. As a consequence, *super*-GAC does not hold.

(GAC+ $\not\succeq$ *super*-GAC) See counter-example in Figure 4.10. The first graph shows the micro-structure of a simple constraint network with two variables, each with a ternary domain. A link represents an allowed combination. An intermediate step as well as the final *super*-GAC closure are represented through colours. For a variable X, a "black" value belongs to $s\mathcal{D}(X)$, a "grey" value to $s\mathcal{D}(X)$ and a "white" value is pruned. $\mathcal{P}$ is GAC+ since the network is arc consistent and every domain contains 3 values. However, $\mathcal{P}$ is not *super*-GAC since the greyed values (in the second graph) are not in $s\mathcal{D}$, they have only one support. In the second step, the whitened variables (in the third graph) are also removed from both $r\mathcal{D}$ and $s\mathcal{D}$ since they do not have a support in a $s\mathcal{D}$.

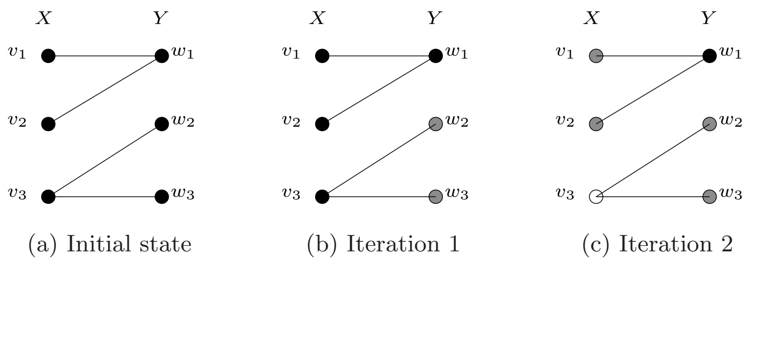
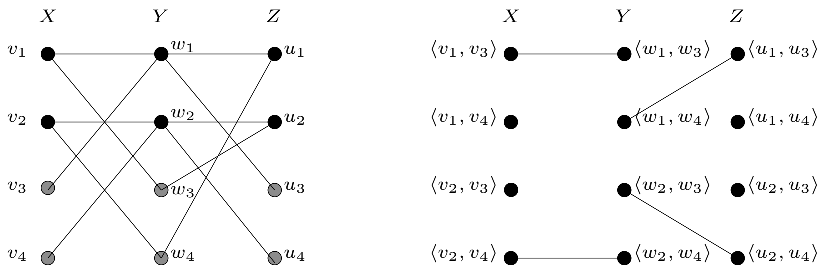


Figure 4.10: A counter example for GAC+ $\succeq$ *super*-GAC.

Proposition 4.6:

(GAC($\mathcal{P} \times \mathcal{P}$) $\succeq$ *super*-GAC) Suppose that GAC($\mathcal{P} \times \mathcal{P}$) holds, then for any two variables $X, Y \in \mathcal{X}$ there exist two pairs $\langle v1, r1 \rangle \in \mathcal{D}^{\times}(X), \langle v2, r2 \rangle \in \mathcal{D}^{\times}(Y)$, such that $\langle v1, r2 \rangle, \langle r1, v2 \rangle$ and $\langle v1, v2 \rangle$ are allowed tuples for $C(X, Y)$. Therefore $v1$ belongs to $s\mathcal{D}(X)$ and $v1$ and $r1$ belong to $r\mathcal{D}(X)$. Thus, we have $s\mathcal{D}(X) \neq \emptyset$ and $|r\mathcal{D}(X)| > 1$ and similarly $s\mathcal{D}(Y) \neq \emptyset$ and $|r\mathcal{D}(Y)| > 1$. Therefore, $\mathcal{P}$ is *super*-GAC.

(*super*-GAC $\not\succeq$ GAC($\mathcal{P} \times \mathcal{P}$)) See counter-example in Figure 4.11. The first graph shows the micro-structure of a simple constraint network with three variables, each with a quaternary domain. A link represents an allowed combination and the *super*-GAC closure is represented through colours. For a variable X, a "black" value belongs to $s\mathcal{D}(X)$ and a "grey" value to $s\mathcal{D}(X)$. $\mathcal{P}$ is *super*-GAC since every variable X is such that $s\mathcal{D}(X) \neq \emptyset$ and $|r\mathcal{D}(X)| > 1$. The second graph shows $\mathcal{P} \times \mathcal{P}$, which is not GAC.


X
Y
Z
v_1
v_2
v_3
v_4
w_1
w_2
w_3
w_4
u_1
u_2
u_3
u_4
$\langle v_1, v_3 \rangle$
$\langle v_1, v_4 \rangle$
$\langle v_2, v_3 \rangle$
$\langle v_2, v_4 \rangle$
$\langle w_1, w_3 \rangle$
$\langle w_1, w_4 \rangle$
$\langle w_2, w_3 \rangle$
$\langle w_2, w_4 \rangle$
$\langle u_1, u_3 \rangle$
$\langle u_1, u_4 \rangle$
$\langle u_2, u_3 \rangle$
$\langle u_2, u_4 \rangle$

(a) *super*-GAC closure (b) $\mathcal{P} \times \mathcal{P}$ reformulation

Figure 4.11: A counter example for *super*-GAC $\not\succeq$ GAC($\mathcal{P} \times \mathcal{P}$).

□

4.5.4 Filtering Level: dynamic context

The essential difference with the static case is that when making a decision, the current "state", represented by the domain relations $s\mathcal{D}$ and $r\mathcal{D}$, can be changed in a way that is not possible for the consistency closure algorithm. In fact, only GAC+ has this property: whilst $s\mathcal{D}$ and $r\mathcal{D}$ are filtered exactly in the same way in the closure procedure, a decision during search reduces $s\mathcal{D}$ and leaves $r\mathcal{D}$ unchanged. However, within GAC+, this difference between $s\mathcal{D}$ and $r\mathcal{D}$ does matter. On the other hand, with *super*-GAC there is not such a difference between the static and dynamic case since $s\mathcal{D}$ and $r\mathcal{D}$ are not filtered in the same way. In particular, a reduction of $s\mathcal{D}$ without any change in $r\mathcal{D}$ can be simulated in a static constraint network. Indeed suppose that we want to simulate the fact that the decision $X = v$ has been made by the search algorithm. We introduce an extra variable X' and extend the domain relations $s\mathcal{D}$ and $r\mathcal{D}$ as follows:

$$s\mathcal{D}(X') = \{s\},\ r\mathcal{D}(X') = \{s, r\}$$

Then we introduce the constraint $C(X, X')$, defined as follows:

$$\forall w \in r\mathcal{D}(X),\ \langle w, s\rangle \in C(X, X') \ \wedge\ \langle v, r\rangle \in C(X, X')$$

Any value except v will be removed from $s\mathcal{D}(X)$ when propagating *super*-GAC and this new variable will not have any more impact. We therefore revise the static relations stated for GAC+ in Section 4.5.3 using the same framework, except that the initial that we also provide the initial values of $s\mathcal{D}$ and $r\mathcal{D}$. The corresponding decisions applied to the domain of the reformulation method need be applied on $\mathcal{P} + \mathcal{P}$ as follows:

$$\mathcal{D}^+(X) = s\mathcal{D}(X),\ \mathcal{D}^+(X^+) = r\mathcal{D}(X) \tag{4.7}$$

$$\tag{4.8}$$

The relations between consistencies change slightly when observed within a dynamic context.

Theorem 27 (Dynamic context)**.**

$$\text{GAC}(\mathcal{P} + \mathcal{P}) \succ \text{GAC+} \tag{4.9}$$

$$\textit{super}\text{-GAC} \succ \text{GAC}(\mathcal{P} + \mathcal{P}) \tag{4.10}$$

$$\text{GAC}(\mathcal{P} \times \mathcal{P}) \succ \textit{super}\text{-GAC} \tag{4.11}$$

Proof. Let $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ be a constraint network, and let $s\mathcal{D}$, $r\mathcal{D}$ represent the current state of the search procedure, we prove the three propositions above:

Proposition 4.4:

(GAC($\mathcal{P} + \mathcal{P}$) $\succeq$ GAC+) Suppose that $\mathcal{P}$ is not GAC+. Then, once GAC has been propagated on $s\mathcal{D}$ and each domain $r\mathcal{D}(X)$ has been made arc consistent with

$s\mathcal{D}$, there exists a variable X_i such that $|r\mathcal{D}(X_i)| \leq 1$. Now, within $\mathcal{P} + \mathcal{P}$, consider the sub-problems defined respectively over the set of variables $\mathcal{X}$ and $\mathcal{X} \setminus \{X_i\} \cup \{X_i^+\}$. Both are equivalent to the original constraint network $\mathcal{P}$, except that the domain of X_i and X_i^+ are respectively equal to $s\mathcal{D}(X_i)$ and $r\mathcal{D}(X_i)$. Therefore, in both cases, the same propagation of GAC will lead to the same result, $\mathcal{D}^+(X_i)$ and $\mathcal{D}^+(X_i^+)$ will both be reduced to the same singleton. Consequently the disequality constraint between X_i and X_i^+ is generalised arc inconsistent.

(GAC+ $\not\succeq$ GAC($\mathcal{P} + \mathcal{P}$)) Consider the constraint network $\mathcal{P}$ and its $\mathcal{P} + \mathcal{P}$ reformulation as shown in Figure 4.13. The domains are initially all equal to $\{1, 2\}$. Both $\mathcal{P}$ and $\mathcal{P}+\mathcal{P}$ are GAC, moreover, all domains contain at least two values, hence $\mathcal{P}$ is GAC+. Now we assume the following domain relations for $\mathcal{P}$ and $\mathcal{P}+\mathcal{P}$ before applying respectively GAC+ and GAC. These domains correspond to the decision $X = 1$.

$$\begin{array}{lcl@{\qquad\qquad}lcl}
s\mathcal{D}(X) & = & \{1\} & \mathcal{D}^+(X) & = & \{1\} \\
s\mathcal{D}(Y) & = & \{1,2\} & \mathcal{D}^+(Y) & = & \{1,2\} \\
s\mathcal{D}(Z) & = & \{1,2\} & \mathcal{D}^+(Z) & = & \{1,2\} \\
r\mathcal{D}(X) & = & \{1,2\} & \mathcal{D}^+(X^+) & = & \{1,2\} \\
r\mathcal{D}(Y) & = & \{1,2\} & \mathcal{D}^+(Y^+) & = & \{1,2\} \\
r\mathcal{D}(Z) & = & \{1,2\} & \mathcal{D}^+(Z^+) & = & \{1,2\}
\end{array}$$

Figure 4.12: A counter example for (GAC($\mathcal{P} + \mathcal{P}$) $\not\succeq$ GAC+) (within search).

We apply GAC+ on $s\mathcal{D}$ and $r\mathcal{D}$. First, $s\mathcal{D}$ is made generalised arc consistent, no value is removed. Since the value 2 for X has one support on Y (1) and another on Z (1), it is not removed from $r\mathcal{D}$. Since $s\mathcal{D} \subseteq r\mathcal{D}$, no more value can be pruned. Now, if GAC is enforced on $\mathcal{P} + \mathcal{P}$, the domain relation $\mathcal{D}^+$ is wiped out. Indeed, we have $X \neq X^+$, hence $X^+ \neq 1$:

$$\mathcal{D}^+(X^+) = \{2\}$$

Then propagating the constraints $X^+ + Y \leq 3$ and $X^+ + Z \leq 3$ leads to:

$$\mathcal{D}^+(Y) = \{1\},\ \mathcal{D}^+(Z) = \{1\}$$

In turn, the constraints $Y \neq Y^+$ and $Z \neq Z^+$ are triggered:

$$\mathcal{D}^+(Y^+) = \{2\},\ \mathcal{D}^+(Z^+) = \{2\}$$

However, both the constraints $Y = Z^+$ and $Z = Y^+$ are now unsatisfiable, hence a failure.

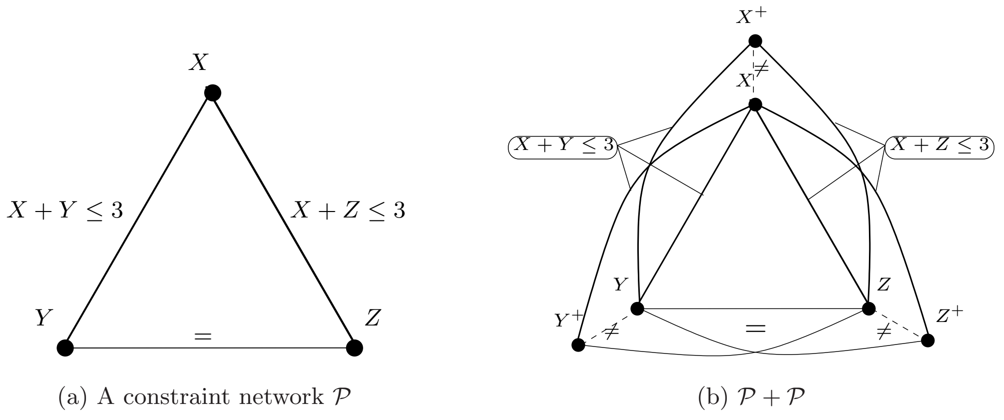


Figure 4.13: An example of the consequences of the same decision for `MAC+` and `MAC` on $\mathcal{P}+\mathcal{P}$.

Proposition 4.5:

(*super*-GAC $\succeq$ GAC($\mathcal{P}+\mathcal{P}$)) As shown in the preamble of this proof, any domain reduction made during search can also happen during the propagation phase, hence the proof of the static case holds.

(GAC+ $\not\succeq$ *super*-GAC) We can reuse the same counter-example (see Figure 4.10).

Proposition 4.6:

Here again, the proof of the static case is still valid.

(*super*-GAC $\not\succeq$ GAC($\mathcal{P}\times\mathcal{P}$)) We can reuse the same counter-example (see Figure 4.11). $\square$

4.5.5 Complexity

4.5.5.1 Complexity of *super*-GAC on Global Constraints

We study the complexity of enforcing *super*-GAC on three examples of global constraints: AllDifferent, Among and Sum.

Definition 19. *The constraint* AllDifferent(V) *holds if and only if all variables in V are assigned to distinct values.*

$$\tau \in \text{AllDifferent}(V) \Leftrightarrow |\{\tau(X) \mid X \in V\}| = |V|$$

Definition 20. *The constraint* Among(V, N, S) *holds if and only if exactly N variables in V are assigned to values represented in the set S.*

$\tau \in \textsc{Among}(V, N, S) \Leftrightarrow |\{X \mid X \in V \ \wedge \ \tau(X) \in S\}| = \tau(N)$

Definition 21. *The constraint* $\textsc{Sum}(V, N)$ *holds if and only if the values assigned to variables in* V *add up to* N*.*

$\tau \in \textsc{Sum}(V, N) \Leftrightarrow \sum_{X \in V} \tau(X) = \tau(N)$

The complexity of computing a *super*-GAC support is incomparable to the complexity of finding a generalised arc consistent support. In [Elbassioni 05] the authors show that computing a *super*-GAC support on an AllDifferent constraint is NP-hard. On the other hand, consider for instance a Sum constraint. Such constraints do not have *super*-GAC support as if we modify one variable the sum must change, hence enforcing *super*-GAC is trivial. However it is NP-hard to enforce GAC on a Sum constraint. Finally there exists constraints for which *super*-GAC and generalised arc consistency are both polynomial to compute. For instance consider the Among constraint. For a set S of values, the solution f satisfies $\textsc{Among}(X_1, \dots X_n, S, N)$ if and only if there are exactly $f(N)$ variables X_i such that $f(X_i) \in S$. Computing a GAC support can be done in polynomial time, moreover, computing a *super*-consistent support can also be done in polynomial time. First we observe that, for any variable X, if a value v is the only one element of S in $\mathcal{D}(X)$, then it cannot participate in any $(1, 0)$-*super*-solution. Indeed, consider a solution f such that $f(X) = v$, there is no alternative for the assignment $X = v$, as changing it to any other value in $\mathcal{D}(X)$ would require to change the value of N. Similarly, if v is the only one value not in S, then again it does not participate in any $(1, 0)$-*super*-solution. On the other hand, suppose that for any variable X, and any value v there exists another value $w \neq v$ in $\mathcal{D}(X)$ such that $v \in S \Leftrightarrow w \in S$. now consider any solution f satisfying $\textsc{Among}(X_1, \dots X_n, S, N)$. To any assignment $X = v$, we can substitute the assignment $X = w$ and the constraint remains satisfied, hence f is a $(1, 0)$-*super*-solution. Now, assuming that the algorithm `among` returns a GAC assignment (or fails) in polynomial time, we can devise *super*-`among` (Algorithm 12) for *super*-GAC. We summarise these complexity results in Table 4.3.

4.5.5.2 Binary Constraint Network

Now we give the complexity of the different approaches developed in this chapter on binary constraint networks. Throughout this section, m stands for the number of

Algorithm 12 *super*-`among`

```
Data   : $X_1, \dots X_n$, $S$, $N$
Result : Is there a $(1,0)$-super-solution $f$ satisfying $\textsc{Among}(X_1, \dots X_n, S, N)$
foreach $i \in [1..n]$ do
    if $|S \cap \mathcal{D}(X_i)| = 1$ then $\mathcal{D}(X_i) \leftarrow \mathcal{D}(X_i) \setminus S$;
    if $|S \cap \mathcal{D}(X_i)| = |\mathcal{D}(X_i)| - 1$ then $\mathcal{D}(X_i) \leftarrow \mathcal{D}(X_i) \cap S$;
return among$(X_1, \dots X_n, S, N)$;
```

Figure 4.14: An algorithm for finding $(1,0)$-*super*-solutions of an Among constraint.

consistency	AllDifferent	Among	Sum
GAC	P	P	NP-complete
super-GAC	NP-complete	P	$O(1)$

Table 4.3: The complexity of computing $(1,0)$-*super*-solutions for some global constraints.

constraints ($m = |\mathcal{C}|$) and d for the domain size, which we shall consider uniform across all variables ($d = |\mathcal{D}(X)| \; \forall X \in \mathcal{X}$). We shall also consider that the number of variables ($n = |\mathcal{X}|$) is always smaller than the number of constraints ($n < m$).

GAC+**:** Computing the GAC closure can be done in $\mathcal{O}(md^2)$ which dominates the complexity of checking that every domain contains at least two values ($\mathcal{O}(n)$). Therefore the GAC+ closure can be computed in $\mathcal{O}(md^2)$.

GAC($\mathcal{P} + \mathcal{P}$)**:** The number of constraints in $\mathcal{P} + \mathcal{P}$ is $3m + n$, the domains are unchanged. Therefore, computing the GAC closure on $\mathcal{P} + \mathcal{P}$ can be done in $\mathcal{O}(md^2)$.

super-GAC**:** Algorithm 8 is based on **AC3**. The same arguments used for AC3 can be adapted. It is easy to see that `propagate` runs in $\mathcal{O}(d^2)$. Moreover, a constraint is revised only if the mappings $s\mathcal{D}$ and $r\mathcal{D}$ have changed for one of the constrained variables, i.e, a value is removed from $s\mathcal{D}(X)$ or $r\mathcal{D}(X)$. This can happen at most $2d$ times for each variable, hence $4d$ times. The complexity of *super*-`AC` is therefore $\mathcal{O}(md^3)$. However, this closure can be computed with an **AC4** based procedure, hence with an optimal worst time complexity of $\mathcal{O}(md^2)$. Algorithm 10 is such an optimal algorithm for computing the *super*-consistent closure. The support counters (one for every value of both variables of every constraint) are duplicated. One will stand for the number of supports currently in $s\mathcal{D}$ whilst the second will stand for the number of supports in $r\mathcal{D}$. Notice that at each step in the process, at least one of these counters is decreased, there are $4md$ counters whose initial values are at most d, hence the complexity in $\mathcal{O}(md^2)$.

GAC($\mathcal{P}\times\mathcal{P}$)**:** The number of constraints in $\mathcal{P}\times\mathcal{P}$ is unchanged, and the domains are squared. Therefore, computing the GAC closure on $\mathcal{P}\times\mathcal{P}$ can be done in $\mathcal{O}(md^4)$.

Figure 4.15 illustrates the relations stated in Theorem 26, whilst Table 4.4 shows the worst case complexity of computing the closure for these consistencies.

$$\begin{tikzcd}
 & \text{GAC}(P\times P) \arrow[d, "\succ"] & \\
 & \textit{super}\text{-GAC}(P) \arrow[dl, "\succ"'] \arrow[dr, "\succ"] & \\
\text{GAC}(P+P) \arrow[rr, leftrightarrow, "\simeq"'] & & \text{GAC+}(P)
\end{tikzcd}$$

(a) Static context

$$\begin{tikzcd}
\text{GAC}(P\times P) \arrow[d, "\succ"] \\
\textit{super}\text{-GAC}(P) \arrow[d, "\succ"] \\
\text{GAC}(P+P) \arrow[d, "\succ"] \\
\text{GAC+}(P)
\end{tikzcd}$$

(b) Dynamic context

Figure 4.15: The relation between consistencies (reads if **tail** holds then **head** holds).

	GAC+($\mathcal{P}$)	GAC($\mathcal{P}+\mathcal{P}$)	*super*-GAC($\mathcal{P}$)	GAC($\mathcal{P}\times\mathcal{P}$)
complexity:	$\mathcal{O}(md^2)$	$\mathcal{O}(md^2)$	$\mathcal{O}(md^2)$	$\mathcal{O}(md^4)$

Table 4.4: The complexity of computing some local consistency closures for finding $(1,0)$-*super*-solutions.

4.6 Search Algorithms

We now present two new search algorithms: `MAC+` and *super*-`MAC`, maintaining respectively the GAC+ and *super*-GAC closure during search. One difference to keep in mind between `MAC` on one hand and `MAC+` or *super*-`MAC` on the other hand, is that the notion of decision, or choice point, is different. For `MAC`, a decision of the form

$X = v$ translates to a reduction of the corresponding domain in the constraint network ($\mathcal{D}(X) = \{v\}$). For `MAC+` or *super*-`MAC`, the same decision $X = v$, will translate to $s\mathcal{D}(X) = \{v\}$, whilst $r\mathcal{D}(X)$ is unchanged. The super domain $s\mathcal{D}$ plays the same role as φ in the regular `MAC` algorithm, whilst the repair domain $r\mathcal{D}$ is not reduced when a decision is taken.

4.6.1 Maintain GAC+ (`MAC+`)

This algorithm establishes GAC+ at each node. That is, it maintains GAC and backtracks if a domain wipes out or becomes singleton. The domains are managed as follows:

- For any variable X, if X is assigned, that is, a decision $X = v$ has been made, then $s\mathcal{D}(X) = \{v\}$.
- For any variable X, if X is not assigned, then $s\mathcal{D}(X) = r\mathcal{D}(X)$.
- For any variable X, we have $w \in r\mathcal{D}(X)$ if and only if $X = w$ has a GAC support for any constraint $C(X, Y) \in \mathcal{C}$.

In `MAC`, only future variables are pruned, since the values assigned to past variables are guaranteed to have a support in each future variable. Here, this property holds for values in $s\mathcal{D}$, but not for values in $r\mathcal{D}$. Therefore a domain of a variable X that is already assigned may become a singleton because of an assignment further down in the search tree. Therefore, GAC is established on $r\mathcal{D}$, and on the whole network (Line 1), and not only on the future variables. Algorithm 13 implements `MAC+`.

4.6.2 Super Maintain Arc Consistency (*super*-`MAC`)

The pseudo code for *super*-`MAC` (Algorithm 14) is similar to `MAC+`, decisions are here again reductions of $s\mathcal{D}$. Notice that here again, the consistency closure is computed on the whole constraint network (Line 1) instead of its restriction to future variables.

4.6.3 Soundness and Completeness

We have established an ordering relation on the different filtering methods. Since `MAC`($\mathcal{P} \times \mathcal{P}$) always backtracks when one of the other algorithms does, whilst `MAC+`

Algorithm 13 `MAC+`

```
Data   : $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$, $s\mathcal{D}[= \mathcal{D}]$, $r\mathcal{D}[= \mathcal{D}]$, $\mathcal{F}[= \mathcal{X}]$
Result : Does $\mathcal{P}$ admit a $(1,0)$-super-solution
if $\mathcal{F} = \emptyset$ then return true;
choose $X \in \mathcal{F}$;
save $s\mathcal{D}$ and $r\mathcal{D}$;
foreach $v \in s\mathcal{D}(X)$ do
    $s\mathcal{D}(X) \leftarrow \{v\}$;
1   if GAC+($\mathcal{P}, s\mathcal{D}, r\mathcal{D}$) then
        if MAC+($\mathcal{P}, s\mathcal{D}, r\mathcal{D}, \mathcal{F} \setminus \{X\}$) then return true;
    restore $s\mathcal{D}$ and $r\mathcal{D}$;
return false;
```

Figure 4.16: An algorithm for finding $(1,0)$-*super*-solution using GAC+.

Algorithm 14 *super*-`MAC`

```
Data   : $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$, $s\mathcal{D}[= \mathcal{D}]$, $r\mathcal{D}[= \mathcal{D}]$, $\mathcal{F}[= \mathcal{X}]$
Result : Does $\mathcal{P}$ admit a $(1,0)$-super-solution
if $\mathcal{F} = \emptyset$ then return true;
choose $X \in \mathcal{F}$;
save $s\mathcal{D}$ and $r\mathcal{D}$;
foreach $v \in s\mathcal{D}(X)$ do
    $s\mathcal{D}(X) \leftarrow \{v\}$;
1   if super-AC($\mathcal{P}, s\mathcal{D}, r\mathcal{D}$) then
        if super-MAC($\mathcal{P}, s\mathcal{D}, r\mathcal{D}, \mathcal{F} \setminus \{X\}$) then return true;
    restore $s\mathcal{D}$ and $r\mathcal{D}$;
return false;
```

Figure 4.17: An algorithm for finding $(1,0)$-*super*-solution using *super*-GAC.

never backtracks unless all the other algorithms do. Therefore any solution found by `MAC`($\mathcal{P} \times \mathcal{P}$) will eventually be found by the others, and `MAC+` will only find solutions found by one of the other algorithms. We prove that `MAC+` is correct and MAC($\mathcal{P} \times \mathcal{P}$) is complete. Hence all four algorithms are correct and complete.

Theorem 28. *For any given CSP $\mathcal{P}$, the sets of solutions of* `MAC+`*($\mathcal{P}$), of super-MAC($\mathcal{P}$), of* `MAC`*($\mathcal{P} \times \mathcal{P}$), and of* `MAC`*($\mathcal{P} + \mathcal{P}$) are identical and equal to the* $(1,0)$*-super-solutions of $\mathcal{P}$.*

Proof. Since there is a total ordering in filtering power between all algorithms (`MAC(`$\mathcal{P} \times \mathcal{P}$`)` $\succeq$ *super*-`MAC` $\succeq$ `MAC(`$\mathcal{P} + \mathcal{P}$`)` $\succeq$ `MAC+`), we only need to prove that:

- The strongest closure algorithm is sound, hence the strongest search algorithm is complete (finds a solution if any).
- The weakest closure algorithm applied to a full assignment decides this assignment, hence the weakest search algorithm is sound (does not return an inconsistent solution).

`MAC+` is sound: We prove that given a full assignment f, GAC+ succeeds only if f is a $(1,0)$-*super*-solution. Suppose that f is not a $(1,0)$-*super*-solution, then there exists a variable X such that $f(X) = v$ and $\forall w \in \mathcal{D}(X), v \neq w$, $f_{|X=w}$ is not a solution. Therefore when all the variables are assigned, and so there remain in the domains only the values that are GAC, $\mathcal{D}(X) = \{v\}$ and thus f is not returned by `MAC+`.

`MAC(`$\mathcal{P} \times \mathcal{P}$`)` is complete: We prove that applying GAC to $\mathcal{P} \times \mathcal{P}$ does not remove values participating in a $(1,0)$-*super*-solution. In other words, it does not remove a tuple $\langle v_1, r_1 \rangle$ such that v_1 participates in a $(1,0)$-*super*-solution and r_1 in the corresponding *repair*. Let f be a $(1,0)$-*super*-solution, for any variables X, Y, let $f(X) = v_1$ and r_1 one of its possible repairs. Similarly v_2 is the value assigned to Y and r_2 its repair. It is easy to see that the pairs $\langle v_1, r_1 \rangle$ and $\langle v_2, r_2 \rangle$ are GAC, i.e, $\langle v_1, v_2 \rangle$, $\langle v_1, r_2 \rangle$ and $\langle r_1, v_2 \rangle$ are allowed tuples. □

4.7 Summary and Limitations

In this chapter we introduced three new methods for finding $(1,0)$-*super*-solutions. In the $\mathcal{P} \times \mathcal{P}$ reformulation, we change all domains to be a cross-product of themselves and change the constraints so that the only valid tuples are those corresponding to the restriction of a $(1,0)$-*super*-solution over two variables. The two other methods are local consistency properties. The first, GAC+, is a slight extension over GAC whilst the second, *super*-GAC, allows stronger inferences by using the same type of reasoning as used in the $\mathcal{P} \times \mathcal{P}$ reformulation. An earlier reformulation method, $\mathcal{P} + \mathcal{P}$ as well as a local consistency technique closely related to GAC+ and *super*-GAC are also discussed.

We introduced two closure algorithms, for GAC+ and *super*-GAC. We proved that a backtracking algorithm using these procedures as inference methods is a sound and complete method for finding $(1, 0)$-*super*-solutions. Moreover we show that their computational complexity does not exceed that of closure algorithms for GAC, and hence is optimal.

We then compared the three new methods, along with the $\mathcal{P} + \mathcal{P}$ reformulation. A classical `MAC` algorithms searching the $\mathcal{P} \times \mathcal{P}$ reformulation is shown to explore the smallest search tree, although this comes at the cost of a greater time complexity at each node. The second strongest method is *super*-GAC followed by `MAC` on $\mathcal{P} + \mathcal{P}$ and finally GAC+ is showed to be the weakest approach. However, as we shall see in the experimental results in Chapter 8, the approach that seems to offer the best tradeoff is *super*-GAC.

There are numerous open and unexplored questions about $(1, 0)$-*super*-solutions. For instance, we restricted our analysis to binary constraint networks. Even though $\mathcal{P} + \mathcal{P}$, *super*-GAC, and GAC+ have been defined on unrestricted networks, the closure algorithms that we introduced are restricted to binary constraints. This limitation is easy to handle with GAC+ since this method is a slight extension over GAC. However, extending *super*-GAC whilst keeping a reasonable computational complexity seems challenging. As a first attempt in this direction we showed that specific propagation algorithms achieving *super*-GAC on some global constraints can sometimes be easier (Sum) or just as hard (Among) as propagating GAC on the same global constraints. However, in [Elbassioni 05], Elbassioni and Katriel showed that propagating *super*-GAC on the AllDifferent is NP-hard.

Chapter 5

Weak Fault Tolerance

5.1 Introduction

Fault tolerant solutions are a very restricted type of robustness. When repairing a solution, we are only allowed to change the variable that breaks. In this chapter we consider how to drop this restriction by developing a backtracking algorithm for computing existential-(a, b)-*super*-solutions for $a, b \geq 1$. In such solutions, a set of variables of cardinality b can be reassigned in order to repair a breakage of size at most a. As opposed to $(1, 0)$-*super*-solutions, the consequences of a breakage are not restricted to the "broken" variables, any variable in the network can potentially be reassigned in response to a breakage. There can be $a + b$ discrepancies between a *super*-solution and its *repair*s, and their exact location is not known. Consequently, it is difficult to use similar local consistency properties, such as GAC+ and *super*-GAC used in Chapter 4. We therefore take a different approach to devise an effective algorithm for finding (a, b)-*super*-solution with arbitrary a and b. Instead of checking each constraint separately, we check the breakages, i.e., combinations of $k \leq a$ variables, to assert repairability. This algorithm relies on a decomposition of the problem into a **master-problem**, on which the main procedure develops a search tree, and a set of dynamically created **sub-problems**, each one witnessing the **repairability** of a breakage with respect to the current partial solution. The numbers of breakages may be large and solving each sub-problem is not a priori an easy task. However, we introduce several ways of making inferences in order to:

- Solve less sub-problems by identifying feasibility without search in some cases.

- Solve sub-problems more efficiently.
- Infer inconsistency in the master-problem whilst solving sub-problems.

When solving a sub-problem in order to find a *repair* of a partial solution, two types of decisions are to be made. The first type of decision is whether or not a variable should be reassigned, and the second is the value assigned to such a variable. We shall see that some inference can be made on the first type, that is, we may be able to decide, before solving a sub-problem that a variable cannot be reassigned. This kind of inference, i.e., **equality constraints** is very valuable, because it allows both to reduce the sub-problem, and to infer inconsistencies in the master-problem itself. An important notion for the point above is the **neighbourhood** of a breakage. We define the distance between two variables (or two sets of variables) as the shortest path that links these variables (or sets) in the constraint graph. We shall see that for an (a, b)-*super*-solution, the changes required to repair a breakage A must be at distance at most b of A. Given such a neighbourhood, we know that all variables outside cannot be reassigned for repair. This is a very weak form of "locality" since the constraint network may be dense and thus the neighbourhood of a variable may be arbitrarily large. However we shall see that even for dense constraint networks, some additional inference rules may be used for pruning the master problem thus greatly reducing search.

In Section 5.2, we describe a very simple and basic version of this decomposition algorithm. Then, in Section 5.3 we introduce two ways of discovering equality constraints, one based on the notion of neighbourhood, and the second on a simple consistency preprocessing. In Section 5.4, 5.5 and 5.6 respectively, we use this last notion to achieve the three goals announced earlier, i.e., solving fewer sub-problems, solving them faster, and taking advantage of them to infer inconsistencies in the master-problem. Then we give the pseudo code of the algorithm taking the above improvements into account and we discuss its implementation. Finally, in Section 5.7 we theoretically compare this algorithm, restricted to $(1, 0)$-*super*-solutions with the specific algorithm introduced in Chapter 4.

5.2 The Decomposition Algorithm

In this section, we introduce an algorithm for computing (a, b)-*super*-solutions of a constraint network. An (a, b)-*super*-solution is defined as a solution such that any breakage, i.e., subset of variables with arity a or less, has a *b-repair*. The number of breakages grows exponentially with their size. The exact number of breakages of size $k \leq a$, over n variables is:

$$\sum_{k=1}^{k=a} \binom{n}{k}$$

Notice that this number has no closed form. However, it is clearly bounded by a polynomial of degree a.

Let first consider a naive generalisation of the $\mathcal{P} + \mathcal{P}$ reformulation approach discussed in Chapter 4. Informally this approach duplicated the variables, so that the copies will take as value the alternative in case of breakage. Now, in the general case, a repair is not limited to a single variable, but can potentially affect any variable in the network. Following the same line, not only one variable, but the entire network is therefore duplicated for every breakage. Moreover, every duplicated network is constrained to have a completely disjoint assignment for the variables contained in the breakage and to share at least $n - (a + b)$ assignments otherwise.

Example 19. *In Figure 5.1 we illustrate such an encoding. For each breakage (two breakages of size two are given in the example) the whole network needs to be duplicated. For each break A, a constraint is posted on the set of variable A and on the corresponding set of duplicated variables, to ensure that the duplicates are assigned different values. Finally a constraint whose scope contains all original variables and their duplicates ensures that no more than $|A| + b$ variables are assigned differently.*

The size of such an encoding would be prohibitive. An algorithm somewhat emulating this encoding has been introduced in [Hebrard 04]. Notice that this algorithm is restricted to $(1, b)$-*super*-solutions and hence to a linear number of breakages. The idea is to keep as many partial solutions as breakages, and extend all of them in parallel. Each of these partial solutions is consistent with respect to the constraints on a corresponding implicit duplicate network. One solution stands for the *super*-solution, whilst the n other stand for *b-repair*s, and are extended using a backtrack procedure. During search, if any

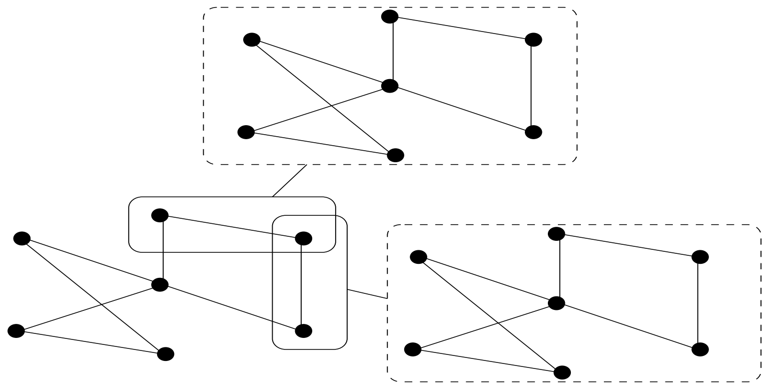


Figure 5.1: A naive reformulation approach for solving an (a, b)-SuperCSP.

of these solutions cannot be extended to the next variable, then a backtrack occurs on all of them, including the *super*-solution. This method avoids explicitly representing every duplicate network. However, it introduces many partial solutions. Since the number of breakage can be exponentially large, this method is again not viable for large values of a.

We therefore avoid the use of any data-structure that would grow with the number of breakages. The approach that we develop in this chapter follows the same principle, but does not store a constraint network nor a solution for each breakage and thus remain polynomial in size, for any a and b. It is essentially equivalent to the naive reformulation, except that the duplicate networks are created on the fly. For each breakage, we dynamically create the corresponding sub-problem, process it and “forget” it. This solves the space complexity issue, but obviously not the time complexity issue. This next step, i.e. improving the scalability and efficiency, is the focus of the rest of this chapter.

5.2.1 Decomposition Approach

We now introduce an algorithm implementing the decomposition approach. The basis is a backtracking algorithm that augments generalised arc consistency processing with a further check of **repairability**. At each node in the search tree, i.e., after each decision, a copy of the problem being solved is created for every breakage $A \subseteq \mathcal{X}$ such that all variables in A are assigned. The domain of any variable in A is reduced so that the value currently assigned in the master problem cannot be chosen. Moreover, a

Similarity constraint is posted, ensuring that the extended Hamming distance to the current partial solution is small enough. Given a partial solution φ and an integer N, this constraint ensures that the number of discrepancies with respect to φ is bounded by N. As the solution of this sub-problem is thus a partial *b-repair*, N is set to $|A| + b$. Notice that we may pass a complete solution f as parameter of the Similarity constraint as well, which will be treated as the corresponding partial solution φ where $\varphi(X) = \{f(X)\}$ for all X.

Definition 22. *The constraint* Similarity$(\mathcal{X}, \varphi, A, b)$ *holds if and only if at most* $|A|+b$ *variables in* $\mathcal{X}$ *are assigned to values not represented in* φ*.*

$$\tau \in \textsc{Similarity}(\mathcal{X}, \varphi, A, b) \Leftrightarrow |\{X_i \mid X_i \in \mathcal{X} \ \wedge \ \tau(X_i) \notin \varphi(X_i)\}| \leq |A| + b$$

Now, let consider a constraint network $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$, a partial solution φ over $k \leq |\mathcal{X}|$ variables and a breakage $A \subseteq \mathcal{X}$. We define the sub-problem $\mathcal{P}^{\varphi,A}$ as follows:

Definition 23. *(see Table 5.1) Let* $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ *be a constraint network, and* φ *a partial solution of* $\mathcal{P}$, $\mathcal{P}^{\varphi,A}$ *is a triplet* $(\mathcal{X}^{\varphi,A}, \mathcal{D}^{\varphi,A}, \mathcal{C}^{\varphi,A})$ *such that:* $\mathcal{X}^{\varphi,A} = \{X^{\varphi,A} \mid X \in \mathcal{X}\}$, $\mathcal{D}^{\varphi,A} = \mathcal{D}$, *and* $\mathcal{C}^{\varphi,A}$ *is equal to* $\mathcal{C}$ *augmented with the constraints* $\forall X \in A \ (X \neq X^{\varphi,A})$ *and* Similarity$(\mathcal{X}^{\varphi,A}, \varphi, A, b)$.

	$\mathcal{P}$	$\mathcal{P}^{\varphi,A}$		
Variables:	$\mathcal{X}$	$\mathcal{X}^{\varphi,A}$	$=$	$\{X^{\varphi,A} \mid X \in \mathcal{X}\}$
Domains:	$\mathcal{D}$	$\mathcal{D}^{\varphi,A}(X^{\varphi,A})$	$=$	$\mathcal{D}(X)$
Constraints:	$\mathcal{C}$	$\mathcal{C}^{\varphi,A}$	$=$	$\mathcal{C} \cup \{\forall X \in A \ (X \neq X^{\varphi,A})\}$ $\cup\{\textsc{Similarity}(\Gamma_b(A), \varphi, A, b)\}$

Table 5.1: The sub-problem $\mathcal{P}^{\varphi,A}$ (Brute-force approach).

Given a sub-problem $\mathcal{P}^{\varphi,A}$ one can check for the existence of a partial solution of the same length as in the master problem in polynomial time providing that a and b are constant. The number of assignments that satisfy the Similarity constraint is bounded by $n^{a+b}d^{a+b}$ where d is the maximum domain size. Indeed, there are less than n^a breakages, and for each, there are at most n^b subset of variables used as *repair.* Hence less than n^{a+b} sets of variables assigned differently, each containing at most $a+b$ variables and thus d^{a+b} possible assignments. Therefore it is possible to generate and test all such assignments in polynomial time. During the search phase, the partial solution φ is reduced, exactly as in the `MAC` algorithm. During the inference phase we first compute the GAC closure of $\mathcal{P} = (\mathcal{F}, \varphi, \mathcal{C})$, where $\mathcal{F} \subseteq \mathcal{X}$ contains all unassigned variables plus the variable involved in the last decision. Then for each breakage whose variables are all assigned, i.e., for any subset $A \subseteq (\mathcal{X} \setminus \mathcal{F})$ such that $|A| \leq a$, we create $\mathcal{P}^{\varphi,A}$. If $\mathcal{P}^{\varphi,A}$ is unsatisfiable, then the breakage A cannot be repaired. We do not actually compute a solution of $\mathcal{P}^{\varphi,A}$, since searching the variables that are not yet assigned in the master-problem can be a costly procedure. Instead, we solve $\mathcal{P}^{\varphi,A}$ up to the same level in the search tree as we currently are in the master-problem. In other words, we use as witness of satisfiability a partial solution ψ, closed under generalised arc consistency and such that all variables assigned in φ are also assigned in ψ, i.e., $X \in (\mathcal{X} \setminus \mathcal{F}) \Rightarrow |\psi(X)| = 1$. If there is no such partial solution then we fail, withdraw the last decision and backtrack in the master-problem. Notice that we create a sub-problem $\mathcal{P}^{\varphi,A}$ only for the breakages A involving only assigned variables. We do so because for a breakage A such that $A \subseteq \mathcal{F}$, the resolution of the corresponding sub-problem cannot fail. In fact, the constraints $X \neq X^{\varphi,A}$ are GAC when $|\varphi(X)| > 1$. Therefore, a breakage A such that $X \in A$ is not assigned yields a sub-problem $\mathcal{P}^{\varphi,A}$ whose solutions are a strict super set of the sub-problem $\mathcal{P}^{\varphi,A\setminus\{X\}}$.

5.2.2 Explanation of the Algorithm

Main Backtracker: Algorithm 15 (`decomposition-backtrack`) implements the decomposition approach described in Section 5.2.1. This procedure develops a search tree on the master-problem and the procedure `repairability` is called at each node (Line 1), creating and solving sub-problems.

Enforcing repairability: Algorithm 16 (`repairability`) ensures that each breakage of the partial solution φ has a *b-repair*. For every set A of $k \leq a$ assigned variables ($A \subseteq (\mathcal{X} \setminus \mathcal{F})$ and $|A| = k$), the sub-problem $\mathcal{P}^{\varphi,A}$ is created. Then the restriction of $\mathcal{P}^{\varphi,A}$ to the variables assigned in the master-problem $(\mathcal{X} \setminus \mathcal{F})$ is solved (Line 1). The procedure `repairability` returns `true` if all sub-problems are satisfiable, and `false` otherwise. The procedure `solve` may be any algorithm deciding constraint satisfaction problems. We assume that `MAC` is used. The notation `solve`$(\mathcal{P}|_S)$ means that the solving method is only required to assign variables in $S \subseteq \mathcal{X}$. For instance,

```
Algorithm 15 decomposition-backtrack
    Data   : $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$, $\varphi$, $\mathcal{F}[= \mathcal{X}]$, $a$, $b$
    Result : Does $\mathcal{P}$ admit an $(a, b)$-super-solution
    if $\mathcal{F} = \emptyset$ then return true;
    choose $X \in \mathcal{F}$;
    save $\varphi$;
    foreach $v \in \varphi(X)$ do
        $\varphi(X) \leftarrow \{v\}$;
1       if GAC($\mathcal{P}' = (\mathcal{F}, \varphi, \mathcal{C})$) & repairability($\mathcal{P}, \varphi, \mathcal{F} \setminus \{X\}, a, b$) then
            if decomposition-backtrack($\mathcal{P}, \varphi, \mathcal{F} \setminus \{X\}, a, b$) then return true;
        restore $\varphi$;
    return false;
```

Figure 5.2: A backtracking algorithm for finding (a, b)-*super*-solutions.

when using MAC, the partial solution φ will be closed under generalised arc consistency, and such that for any variable $X \in S$ $|\varphi(X)| = 1$.

```
Algorithm 16 repairability
    Data   : $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$, $\varphi$, $\mathcal{F}[= \mathcal{X}]$, $a$, $b$
    Result : Check the repairability of grounded breakages
    foreach $A \subseteq (\mathcal{X} \setminus \mathcal{F})$ such that $|A| \leq a$ do
1       if $\neg$solve($\mathcal{P}^{\varphi,A}|_{\mathcal{X} \setminus \mathcal{F}}$) then return false;
    return true;
```

Figure 5.3: An algorithm for checking the repairability of a partial solution.

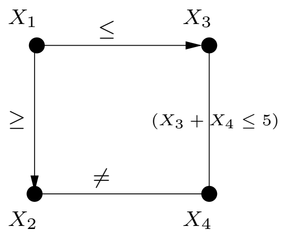
5.3 Repair Localisation

In Chapter 4, we showed that local reasoning was sufficient for finding $(1, 0)$-*super*-solutions. In this section we show that it is still possible in certain cases to restrict the computation involved in checking a *repair* to a smaller part of the constraint network. However, the size of the sub-problem increases with a, b and the density of the network, and therefore cannot be bounded in general. The observation that we make is that there must be at least $n - (a + b)$ variables assigned equally in the master problem and any sub-problem, where $n = |\mathcal{X}|$. Therefore, being able to know beforehand (that is, before starting a search process on a sub-problem) whether a given variable is within the set of $n - (a + b)$ equally assigned variables is very valuable information. In this section we show how one can deduce such equalities, then in Section 5.4.2 and 5.6, we explore two ways of taking advantage of them. Notice that all the inference methods introduced in this chapter are not restricted to binary or even bounded arity constraints. However some equalities are deduced by reasoning on the topology of the network, therefore the existence of constraints with large arity is likely to affect these methods.

Example 20. *We instantiate this concept through an example. Suppose that we are looking for a* $(1, 0)$*-super-solution* f *for a constraint network* $\mathcal{P}$*. We know that for any breakage* $\{X\}$*, a solution* g *of* $\mathcal{P}^{f,\{X\}}$ *must be so that* $n-1$ *variables are assigned equally in both solutions, i.e.,* $\Delta(f, g) = 1$*. Moreover, we know that* X *cannot be assigned to the same value. Therefore, given a partial solution* φ *of* $\mathcal{P}$ *that can be extended to* f*, the* Similarity *constraint reduces the domains of* $\mathcal{P}^{f,\{X\}}$ *as follows:*

$$\mathcal{D}^{f,\{X\}}(\mathcal{X} \setminus \{X\}) = \varphi((\mathcal{X} \setminus \{X\})$$

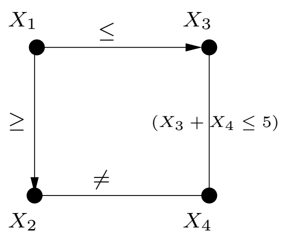
Consider the constraint network in Figure 5.4. The domains of X_1, X_2 *and* X_3 *are all equal to* $\{1, 2, 3\}$ *and the domain of* X_4 *is* $\{1, 2, 4\}$*. Suppose that we begin the search by making the decision* $X_1 = 1$*. We reduce* $\varphi(X_1)$ *to* $\{1\}$ *and compute the generalised arc consistent closure of* φ *(see Figure 5.4).*



$$\begin{array}{lcl}
\mathcal{D}(X_1) & = & \{1,2,3\} \\
\mathcal{D}(X_2) & = & \{1,2,3\} \\
\mathcal{D}(X_3) & = & \{1,2,3\} \\
\mathcal{D}(X_4) & = & \{1,2,4\}
\end{array}
\qquad
\begin{array}{lcl}
\varphi(X_1) & = & \{1\} \\
\varphi(X_2) & = & \{1\} \\
\varphi(X_3) & = & \{1,2,3\} \\
\varphi(X_4) & = & \{2,4\}
\end{array}$$

Figure 5.4: An example of neighbourhood-based inference making.

We then check the breakage $\{X_1\}$*. We show the initial domains as created following the definition, as well as the partial solution* ψ *equal to the* GAC *closure, with the domain reduction due to the* Similarity *constraint (see Figure 5.5). In* $\mathcal{P}^{\varphi,A}$*, the domain of* X_1 *(*$\psi(X_1)$*) is changed to* $\{2, 3\}$*. Then when propagating* GAC*, the value* 1 *is removed from* $\psi(X_3)$ *hence* 4 *is removed from* X_4*.*



$$\begin{array}{rclrcl}
\mathcal{D}^{\varphi,\{X_1\}}(X_1) & = & \{2,3\} & \psi(X_1) & = & \{2,3\} \\
\mathcal{D}^{\varphi,\{X_1\}}(X_2) & = & \{1,2,3\} & \psi(X_2) & = & \{1\} \\
\mathcal{D}^{\varphi,\{X_1\}}(X_3) & = & \{1,2,3\} & \psi(X_3) & = & \{2,3\} \\
\mathcal{D}^{\varphi,\{X_1\}}(X_4) & = & \{1,2,4\} & \psi(X_4) & = & \{2\}
\end{array}$$

Figure 5.5: The resolution of the sub-problem $\mathcal{P}^{\varphi,\{X_1\}}$ for the partial solution φ and the breakage $\{X_1\}$.

Making such an inference can greatly reduce the sub-problems, but more importantly, the same reasoning can be done in the reverse direction. For instance here, for any solution f such that $f(X_1) = 1$ we have $f(X_2) \in \{1\}$, $f(X_3) \in \{2,3\}$ and $f(X_4) \in \{2\}$. Therefore we can prune $X_3 = 1$ and $X_4 = 4$ in the master-problem. However, the situation is more complex when the number of allowed changes is not null ($b > 0$). In this case, it is no longer possible to intersect the domains of variables that are not involved in a break with the current partial solution. We still have the property that $n - (a + b)$ variables must be assigned with the same value for the *super*-solution and a *repair*, but identifying these variables is more difficult. Indeed, since one or more changes are allowed, any variable can, at the outset, be assigned differently than in the master problem. In Section 5.3.1 and 5.3.2 we investigate two inferences rules to deduce such equality constraints between a variable in the master-problem and its homologue in a sub-problem. We shall denote $Eq^{\varphi,A}$ the set of variables that must be assigned equally in the master-problem $\mathcal{P}$ and in $\mathcal{P}^{\varphi,A}$. Once we have identified a set Eq of equality constraints, the following pruning rule can thus be enforced:

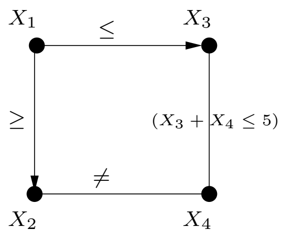
$$\varphi(Eq) = \mathcal{D}^{\varphi,A}(Eq).$$

We now explore two ways for deducing the membership of a variable X to $Eq^{\varphi,A}$. The first one takes advantage of the fact that the *repair*s for a breakage A cannot be topologically too distant from the variables in A in the constraint network. The second is a simple deduction on the states of the domains after enforcing a consistency property on the sub-problem. We shall see that both methods can be combined to achieve more than the sum of the individual pruning.

5.3.1 Breakage Neighbourhood

The first idea is that, intuitively, a change must be **close** to the breakage in the constraint graph. For instance, in a $(1,1)$-*super*-solution, any "repaired" variable must share a constraint with the "broken" variable (say X). Indeed if it was not the case, then it would mean that all the constraints involving X are satisfied by the solution f for both $X = f[X]$ and $X = v$ for a value $v \neq f[X]$. Moreover, we know that f satisfies all constraints, since it is a solution, therefore, the breakage $\{X\}$ need no repair at all, a valid alternative is $X = v$.

Example 21. *For instance consider the constraint network in Figure 5.6, and suppose now that we are looking for a* $(1,1)$-*super-solution. We make the same decision in the master-problem, namely:* $X_1 = 1$. *The same domains* $\mathcal{D}^{\varphi,\{X_1\}}$ *are created, however, since there is a constraint* $C_{\geq}(X_1, X_2)$ *and another* $C_{\leq}(X_1, X_3)$ *we cannot make the intersection with* $\varphi(X_2)$ *nor* $\varphi(X_3)$. *However, we can still intersect* $\mathcal{D}^{\varphi,\{X_1\}}(X_4)$ *with* $\varphi(X_4)$*:*



$$\begin{array}{lcl}
\mathcal{D}^{\varphi,\{X_1\}}(X_1) & = & \{2,3\} \\
\mathcal{D}^{\varphi,\{X_1\}}(X_2) & = & \{1,2,3\} \\
\mathcal{D}^{\varphi,\{X_1\}}(X_3) & = & \{1,2,3\} \\
\mathcal{D}^{\varphi,\{X_1\}}(X_4) & = & \{1,2,4\}
\end{array}
\qquad
\begin{array}{lcl}
\psi(X_1) & = & \{2,3\} \\
\psi(X_2) & = & \{1,2,3\} \\
\psi(X_3) & = & \{2,3\} \\
\psi(X_4) & = & \{2\}
\end{array}$$

Figure 5.6: A second example of neighbourhood-based inference making.

Let us introduce some necessary notation in order to formalise this reasoning to any a, b. The notion of **path** between two variables is defined as the corresponding graph concept in the constraint graph.

Definition 24. *A path linking* $X, Y \in \mathcal{X}$ *is a sequence of constraints* $C(V_1), \ldots C(V_k)$ *such that* $i = j + 1 \ \Rightarrow\ V_i \cap V_j \neq \emptyset$ *and* $X \in V_1$ *and* $Y \in V_k$.

The length of a path is equal to the cardinality of the sequence of constraints. The **distance** between two variables $\delta(X, Y)$ is the length of the shortest path between these variables. The **neighbourhood** up to a distance d of X, denoted $\Gamma_d(X)$, is defined as the set of variables linked to X by a path of length d or less.

Definition 25. $\Gamma_d(X) = \{Y \mid \delta(X,Y) \leq d\}$.

Similarly, we define the neighbourhood $\Gamma_d(A)$ of a set of variables A as the union of the neighbourhoods; $\Gamma_d(A) = \bigcup_{X \in A} \Gamma_d(X)$. Moreover, we define the **orbit** at distance d of a set $A \subseteq \mathcal{X}$, denoted $\Omega_d(A)$, as the set of variables such that the shortest path with any element of A is exactly d.

Definition 26. $\Omega_d(A) = \Gamma_d(A) - \Gamma_{d-1}(A)$.

Example 22. *In Figure 5.7, we illustrate the neighbourhood of a variable. The neighbourhood of X_1 at distance 0 (solid line) is $\{X_1\}$, the neighbourhood at distance 1 (dashed line) is the set $\{X_1, X_4, X_6\}$ and the neighbourhood at distance 2 (dash-point line) contains $\{X_1, X_2, X_4, X_5, X_6, X_7\}$.*

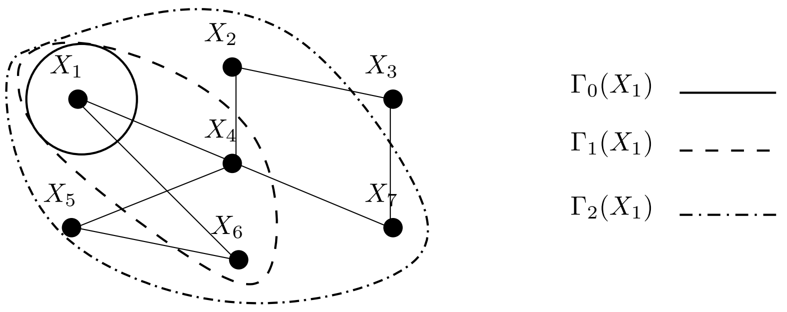


Figure 5.7: The neighbourhood of a variable.

We now state the theorem central to the subsequent use of neighbourhood in the decomposition algorithm. Informally, it shows that if there exists a *b-repair* for a particular breakage A, then all **necessary** reassignments are within the neighbourhood of A up to a distance b, but first we prove the following Lemma:

Lemma 1. *Let $A \subseteq \mathcal{X}$ be a set of variables, $f : \Gamma_d(A) \mapsto \Lambda$ and $g : \mathcal{X} \setminus \Gamma_{d-1}(A) \mapsto \Lambda$ two consistent assignments such that $f(\Omega_d(A)) = g(\Omega_d(A))$. Then any assignment h constructed from f and g such that $\forall X \; (h(X) = f(X) \;\vee\; h(X) = g(X))$ is consistent.*

Proof. Let $g : \Gamma_d(A) \mapsto \Lambda$ and $f : \mathcal{X} \setminus \Gamma_{d-1}(A) \mapsto \Lambda$ be two consistent assignments such that $f(\Omega_d(A)) = g(\Omega_d(A))$. Moreover, let h be an assignment such that $\forall X \; (h(X) = f(X) \;\vee\; h(X) = g(X))$. Without loss of generality, consider any constraint $C(V)$ on a set of variables V. There is a path of length one, $(C(V))$, between any two variables in V. Therefore, the variables in V belongs to at most two orbits $\Omega_{d_1}(A)$ and $\Omega_{d_2}(A)$ such that $d_1 \leq d_2$ are consecutive or equal. We consider the three possible cases:

1. $d_1 < d$ and $d_2 < d$: $h(V) = g(V)$ and $g(V) \in C(V)$ hence $h(V) \in C(V)$.

2. $d_1 \geq d$ and $d_2 \geq d$: $h(V) = f(V)$ and $f(V) \in C(V)$ hence $h(V) \in C(V)$.

3. $d_1 = d - 1$ and $d_2 = d$: $h(V) = g(V)$ and $g(V) \in C(V)$ hence $h(V) \in C(V)$.

□

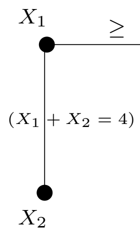
Theorem 29. *Let f be a solution of a constraint network $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ and $A \subseteq \mathcal{X}$ a set of variables. If g is a solution such that $\Delta_A(f,g) = |A|$ and $\Delta(f,g) \leq |A| + b$ then there exists h, such that $\Delta_A(f,h) = |A|$, $\Delta(f,h) \leq |A| + b$ and $\Delta_{\Gamma_b(A)}(f,h) = \Delta(f,h)$.*

Proof. Let f and g be two solutions satisfying the premise, that is, there exists a set of variables A, such that $\Delta_A(f,g) = |A|$ and $\Delta(f,g) \leq |A| + b$. We construct another solution h that satisfies the conclusion, that is, all changes with respect to f are within the set $\Gamma_b(A)$. Let d be the smallest integer such that $f(\Omega_{d+1}(A)) = g(\Omega_{d+1}(A))$. Since all orbits are disjoint, there must be at least $|A| + d$ discrepancies between $f(\Gamma_d(A))$ and $g(\Gamma_d(A))$. Therefore, as the total number of discrepancies is less or equal than $|A| + b$, we have $d \leq b$. Now consider the restriction of f to $\mathcal{X} \setminus \Gamma_d(A)$ and g to $\Gamma_{d+1}(A)$. We define h to be equal to g on $\Gamma_d(A)$ and equal to f on the complement $\mathcal{X} \setminus \Gamma_d(A)$. By definition we know that $f(\Omega_{d+1}(A)) = g(\Omega_{d+1}(A))$ therefore by Lemma 1, h is a solution. Since h is defined with respect to g and f we clearly have $\Delta(h,f) \leq \Delta(g,f)$. Moreover, by definition, all the discrepancies are located within $\Gamma_b(A)$. □

5.3.2 Preprocessing of the Similarity Constraint

The second idea is simply to preprocess $\mathcal{P}^{\varphi,A}$, by enforcing a local consistency. As a result of this preprocessing, the domains may be reduced so that $|A| + b$ variables have their domain disjoint with their image by the partial solution φ. If this is the case, then all other variables must be assigned as in the master-problem.

Example 23. *For instance consider the constraint network in Figure 5.8. Here again, we start the search by making the decision $\varphi(X_1) = \{1\}$. We show the initial domains,* *the* GAC *closure of the partial solution φ as well as the domains of $\mathcal{P}^{\varphi,A}$ after enforcing generalised arc consistency as a preprocessing. Since $\mathcal{D}^{\varphi,\{X_1\}}(X_2)$ and $\varphi(X_2)$ are dis-*



$$\begin{array}{lll} \mathcal{D}(X_1) = \{1,2,3\} & \varphi(X_1) = \{1\} & \mathcal{D}^{\varphi,\{X_1\}}(X_1) = \{2,3\} \\ \mathcal{D}(X_2) = \{1,2,3\} & \varphi(X_2) = \{3\} & \mathcal{D}^{\varphi,\{X_1\}}(X_2) = \{1,2\} \\ \mathcal{D}(X_3) = \{1,2,3\} & \varphi(X_3) = \{1\} & \mathcal{D}^{\varphi,\{X_1\}}(X_3) = \{1,2,3\} \end{array}$$

Figure 5.8: An example of preprocessing-based inference making.

joint, we can deduce that X_3 should be assigned the same value in a $(1,1)$-super-solution f of $\mathcal{P}$ and in a solution of $\mathcal{P}^{f,\{X_1\}}$. Therefore we can prune $X_3 = 2$ and $X_3 = 3$, hence $\mathcal{D}^{\varphi,\{X_1\}}(X_3) = \{1\}$. Let φ be a partial solution, closed under generalised arc consistency, and $\mathcal{P}^{\varphi,A}$ the sub-problem for this partial solution and a breakage A. If the set $Dff = \{X \mid (\varphi(X) \cap \mathcal{D}^{\varphi,A}(X)) = \emptyset\}$ is such that $|Dff| = |A| + b$ then we can safely set $Eq^{\varphi,A}$ to $\mathcal{X} \setminus Dff$.

Notice that stronger local consistencies properties may be used as preprocessing. For instance, **Singleton Arc Consistency** [Debruyne 97], [Prosser 00] is a good candidate to replace GAC in this preprocessing.

The notion of neighbourhood, introduced in Section 5.3.1, can be used to further exploit the inference due to a preprocessing of the Similarity constraint. Suppose that the set of discrepancies *Dff* is such that $|A| < |Dff| < |A| + b$. We normally cannot infer any equality constraint. However, let $k = |Dff| - |A|$, we show that this situation is in fact equivalent to searching a $b-1$-*repair* for the breakage *Dff*, hence the set $\mathcal{X} \setminus \Gamma_{b-k}(Dff)$ can be added to $Eq^{\varphi,A}$.

Theorem 30. *If* Similarity$(X_1, \dots X_k, \varphi, A, b)$ *accepts a solution f, and if there is a set Dff of variables such that $X \in Dff \Rightarrow \varphi(X) \cap \mathcal{D}(X) = \emptyset$ and $A \subset Dff$, then it also accepts a solution g such that all variables in $\Gamma_{b-(|Dff|-|A|)}(Dff)$ take their values in φ.*

Proof. This is an almost immediate application of Theorem 29. If we assume that the domains are made generalised arc consistent, or any other sound local consistency, then all variables in the set *Dff* must be assigned values outside φ. Moreover, if there exists a solution, then the number of discrepancies with φ is less than or equal to $|A|+b$, hence $|\mathit{Dff}| - |\mathit{Dff}| + |A| + b$. Let f be a solution extending φ, and suppose that there exists a solution g such that the number of discrepancies with f is less than or equal to $|A| + b$. By applying Theorem 29, using Dff as the set of variables with domains disjoint to φ, then we know that there exists a third solution h, such that all discrepancies are within the set $\Gamma_{b-(|\mathit{Dff}|-|A|)}(\mathit{Dff})$. $\square$

5.4 Avoiding Unnecessary Checks

5.4.1 Multidirectionality

Multidirectionality is a concept used for implementing general purpose algorithms for enforcing generalised arc consistency. The idea is that when a tuple supporting a value is found, the same tuple can be used to support other values it involves. In the same way as a tuple $\sigma \in C(V)$ is a support for any value involved in σ, a *b-repair* g of a breakage A is actually a *b-repair* for several subsets of variables taken from A and the variables reassigned in g.

Example 24. *For instance, suppose that we look for a* $(2,2)$*-super-solution and suppose that* g *is a* 2*-repair of the breakage* $\{X, Y\}$ *that requires reassigning the variables* $\{V, W\}$*, that is,* $f(V) \neq g(V)$ *and* $f(W) \neq g(W)$*. Then* g *is also a* 2*-repair of* $\{X, V\}, \{X, W\}, \{Y, V\}, \{Y, W\}$ *and* $\{V, W\}$*. We therefore need not to look for repair for these breakages.*

Theorem 31. *If* g *is a b-repair of* A *for* f *and* $B \subseteq \mathcal{X}$ *is such that* $\Delta_{\mathcal{X}\setminus B}(f, g) = 0$*, then for all* $A' \subseteq B \ \wedge \ |A'| \geq |B| - b$*,* g *is also a b-repair of* A'*.*

Proof. Let g be a *b-repair* of A for f, and let B be the subset of $\mathcal{X}$ such that $f(X) \neq g(X) \Leftrightarrow X \in B$. Moreover, let $A' \subseteq B$. Recall that a *b-repair* of a breakage A' for a solution f is a solution h such that $\Delta_{A'}(f, h) = |A'|$ and $\Delta(f, h) \leq |A'| + b$. We show that g is *b-repair* of a A' for f. By definition of A' we have $\Delta_{A'}(f, g) = |A'|$. Now $\Delta(f, g) = |B|$, therefore g is a valid *repair* if and only if $|B| \leq |A'| + b$, hence $|A'| \geq |B| - b$. $\square$

5.4.2 Ground Neighbourhood

Here we show that it is possible to avoid checking some breakages because the necessary checks have been done earlier in the search tree.

Theorem 32. *Let φ, ψ be two partial solutions of $\mathcal{P}$, both closed under GAC, such that $\forall X\ \psi(X) \subseteq \varphi(X)$. If $\Gamma_{b+1}(A) \subseteq \mathcal{G}$ where $\mathcal{G} = \{X \mid |\varphi(X)| = 1\}$ then $\mathcal{P}^{\varphi,A}$ is satisfiable if and only if $\mathcal{P}^{\psi,A}$ is satisfiable.*

Proof. Let f be a solution of $\mathcal{P}$ extending ψ, that is, $\forall X \in \mathcal{X}\ f(X) \in \psi(X)$ and g a solution of $\mathcal{P}^{\varphi,A}$. Let $\mathcal{G}$ be the set of variables assigned in φ, i.e., $\mathcal{G} = \{X \mid |\varphi(X)| = 1\}$, now, by definition of $\mathcal{P}^{\varphi,A}$, we have $\Delta_{\mathcal{G}}(f, g) \leq |A| + b$. Therefore, if we apply Theorem 29 to the restrictions of f and g to $\mathcal{G}$, we know that there exists $h : \mathcal{G} \mapsto \Lambda$ such that $\Delta_{\mathcal{G}}(f, h) = \Delta_{\Gamma_b(A)}(f, h) \leq |A| + b$. Moreover, using Lemma 1 we can extend h to be equal to f on $\mathcal{X} \setminus \mathcal{G}$. Clearly, h is then a solution of $\mathcal{P}^{\psi,A}$ since it is consistent with the constraints of $\mathcal{P}$, $\Delta(f, h) \leq |A| + b$ and $\Delta_A(f, h) = \Delta_A(f, g) = |A|$. $\square$

Using this Theorem, we can simply avoid solving a sub-problem $\mathcal{P}^{\psi,A}$, if ψ extends φ and the neighbourhood of A up to a distance $b+1$ is entirely assigned in the master-problem. Indeed we know that this repair will hold in any subtree, hence we do not need to check it unless we backtrack beyond this point.

5.5 Sub-problems Solving

Each sub-problem is a regular constraint satisfaction problem. In this section we first introduce a propagation algorithm for the SIMILARITY constraint. Then we show how one can obtain a better propagation of the SIMILARITY constraint by using neighbourhood based inference. Finally we show how can we can take advantage of the same reasoning and propagate it back to the master problem.

5.5.1 Propagation Algorithm for the SIMILARITY Constraint

First, notice that using Theorem 29, given a sub-problem $\mathcal{P}^{\varphi,A}$, we can infer equality constraints on the set of variables equal to $\mathcal{X} \setminus \Gamma_b(A)$. Therefore, the domains outside $\Gamma_b(A)$ can be made equal, in the sub-problem, to the partial solution φ when creating $\mathcal{P}^{\varphi,A}$, i.e.,

$$\forall i,\ X_i \notin \Gamma_b(A) \Rightarrow \mathcal{D}^{\varphi,A}(X_i) = \varphi(X_i)$$

Moreover, the SIMILARITY constraint can be posted on $\Gamma_b(A)$ instead of $\mathcal{X}$, as the set $\{i \mid X_i \in (\mathcal{X} \setminus \Gamma_b(A)) \ \wedge \ \mathcal{D}(X_i) \cap \varphi(X_i) = \emptyset\}$ is empty. As a result we can change the definition of a sub-problem $\mathcal{P}^{\varphi,A}$ in the following way:

	$\mathcal{P}$	$\mathcal{P}^{\varphi,A}$		
Variables:	$\mathcal{X}$	$\mathcal{X}^{\varphi,A}$	$=$	$\{X^{\varphi,A} \mid X \in \mathcal{X}\}$
Domains:	$\mathcal{D}$	$\mathcal{D}^{\varphi,A}(X^{\varphi,A})$	$=$	$\varphi(X)$ if $X \notin \Gamma_b(A)$
				$\mathcal{D}(X)$ otherwise
Constraints:	$\mathcal{C}$	$\mathcal{C}^{\varphi,A}$	$=$	$\mathcal{C} \cup \ \{\forall X \in A \ (X \neq X^{\varphi,A})\}$
				$\cup\{\text{SIMILARITY}(\Gamma_b(A), \varphi, A, b)\}$

Table 5.2: The sub-problem $\mathcal{P}^{\varphi,A}$ (Neighbourhood-based inference).

Algorithm 17 (`Similarity-propagate`) enforces generalised arc consistency on the variables of a SIMILARITY constraint. This algorithm first computes the smallest expected set *Dff* of discrepancies to φ (Line 1):

$$Dff = \{X \mid \mathcal{D}(X) \cap \varphi(X) = \emptyset\}$$

We then have three cases:

1. If $|Dff| > |A| + b$ then the constraint cannot be satisfied (Line 2). In that case, the domains are wiped out so that the overall closure algorithm will fail in Line 3.

2. If $|Dff| = |A| + b$ then we can set the domain of any variable X_i such that $\mathcal{D}(X_i) \cap \varphi(X_i) \neq \emptyset$ to $\mathcal{D}(X_i) \cap \varphi(X_i)$.

3. If $|Dff| < |A| + b$ then the constraint is GAC as every variable can be assigned any value providing that all X_i not in *Dff* take a value included in $\varphi(X_i)$, and we will still have $|Dff| \leq |A| + b$, therefore nothing happens. However, if A is a strict subset of *Dff*, then we can infer equality constraints as explained in Section 5.3.2. We show how to soundly perform the corresponding pruning in Algorithm 18.

Algorithm 17 `Similarity-propagate`

```
   Data    : $\mathcal{X} = \{X_1, \dots X_n\}, \varphi, A, b$
   Result  : The GAC closure of $\mathcal{X}$ with respect to SIMILARITY
   $pruned \leftarrow$ false;
1  $Dff = \{X_i \mid \mathcal{D}(X_i) \cap \varphi(X_i) = \emptyset\}$;
2  if $|Dff| > |A| + b$ then
       $pruned \leftarrow$ true;
3      $\mathcal{D} \leftarrow (\mathcal{D} : X_i \rightarrow \emptyset)$;
   else
4      if $|Dff| = |A| + b$ then
           foreach $X_i \in (\mathcal{X} \setminus Dff)$ do
               if $\mathcal{D}(X_i) \not\subseteq \varphi(X_i)$ then
                   $pruned \leftarrow$ true;
                   $\mathcal{D}(X_i) \leftarrow (\mathcal{D}(X_i) \cap \varphi(X_i))$;
   return $pruned$;
```

Figure 5.9: An algorithm for computing the GAC closure of a SIMILARITY constraint.

5.5.2 Neighbourhood Inference for the SIMILARITY Constraint

In Figure 5.10, we give another version of the same algorithm. The procedure `Similarity-propagate-`Γ also uses the neighbourhood based inference to deduce equality constraints and perform the corresponding pruning. In Line 1, the neighbourhood of *Dff* at a distance $b-(|Dff|-|A|)$ is computed. Then the value of *Dff*, which previously stood for the variables **necessarily** different, is set to this neighbourhood, that is, the set of variables **possibly** different. Notice that if $|Dff| = |A| + b$ then the condition in Line 4 of Algorithm 17 would be satisfied, and the same pruning would take place. Otherwise, either $|Dff| = |A|$, and then nothing will happen, since the value of *Dff* will be set to the whole scope of the constraint, or $|Dff| > |A|$ and then some equality constraints may be inferred and immediately propagated (Line 3).

5.6 Master-problems Solving

It has been showed through an example in Section 5.3 that the notion of equality between a variable in $\mathcal{P}$ and its homologue in $\mathcal{P}^{\varphi,A}$ can be used for pruning both $\mathcal{P}^{\varphi,A}$

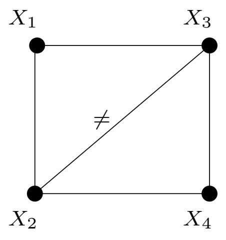
Algorithm 18 `Similarity-propagate-Γ`

```
    Data   : $\mathcal{X} = \{X_1, \dots X_n\}, \varphi, A, b$
    Result : The GAC closure of $\mathcal{X}$ with respect to SIMILARITY
    $pruned \leftarrow$ false;
    $Dff = \{X_i \mid \mathcal{D}(X_i) \cap \varphi(X_i) = \emptyset\}$;
    if $|Dff| > |A| + b$ then
        $pruned \leftarrow$ true;
        $\mathcal{D} \leftarrow (\mathcal{D} : X_i \rightarrow \emptyset)$;
    else
1       $Dff \leftarrow \Gamma_{|A|+b-|Dff|}(Dff)$;
2       foreach $X_i \in (\mathcal{X} \setminus Dff)$ do
            if $\mathcal{D}(X_i) \not\subseteq \varphi(X_i)$ then
                $pruned \leftarrow$ true;
3               $\mathcal{D}(X_i) \leftarrow (\mathcal{D}(X_i) \cap \varphi(X_i))$;
    return $pruned$;
```

Figure 5.10: An algorithm for computing the GACclosure of a SIMILARITY constraint using neighbourhood-based inference.

and $\mathcal{P}$. In this section we give an example of pruning on the master-problem that dramatically reduces the search, then we discuss how to handle this inference method in the `repairability` algorithm.

Example 25. *Consider the following constraint network and suppose that we are looking*



$$3 \leq X_1 + X_2 \leq 4$$
$$3 \leq X_1 + X_3 \leq 4$$
$$C(X_{2/3}, X_4) = \begin{cases} \langle 1,1 \rangle \\ \langle 1,2 \rangle \\ \langle 2,2 \rangle \\ \langle 3,3 \rangle \end{cases}$$

$$\begin{aligned} \mathcal{D}(X_1) &= \{1,2,3\} \\ \mathcal{D}(X_2) &= \{1,2,3\} \\ \mathcal{D}(X_3) &= \{1,2,3\} \\ \mathcal{D}(X_4) &= \{1,2,3\} \end{aligned}$$

Figure 5.11: An example of inference making from a sub-problem to the master-problem.

for a $(1,1)$*-super-solution, the first decision is to assign* $X_1 = 1$*, in the next Figure, we show the partial solution* φ *equal to the generalised arc consistent closure of this decision, as well as the* GAC *closure of* $\mathcal{P}^{\varphi,A}$*. We then apply the inference rule described in Section 5.3 and thus intersect* φ *with* $\mathcal{D}$ *on* $Eq^{\varphi,A} = \mathcal{X} \setminus \Gamma_1(A)$*.*

$$\begin{array}{rclrclrcl}
\varphi(X_1) & = & \{1\} & \mathcal{D}^{\varphi,\{X_1\}}(X_1) & = & \{2,3\} & \varphi(X_1) & = & \{1\} \\
\varphi(X_2) & = & \{2,3\} & \mathcal{D}^{\varphi,\{X_1\}}(X_2) & = & \{1,2\} & \varphi(X_2) & = & \{2,3\} \\
\varphi(X_3) & = & \{2,3\} & \mathcal{D}^{\varphi,\{X_1\}}(X_3) & = & \{1,2\} & \varphi(X_3) & = & \{2,3\} \\
\varphi(X_4) & = & \{2,3\} & \mathcal{D}^{\varphi,\{X_1\}}(X_4) & = & \{1,2\} & \varphi(X_4) \cap \mathcal{D}^{\varphi,\{X_1\}}(X_4) & = & \{2\}
\end{array}$$

Figure 5.12: Continuation of figure 5.11.

We have seen that the equality constraints can be inferred while preprocessing a sub-problem. We can therefore modify `repairability` by taking into account the previous observations. We use a set Eq in which variables participating to equality constraints discovered while preprocessing the SIMILARITY constraint. Moreover, their domains at that time are stored as well using an array $\mathcal{D}_{Eq}$. If the resolution of the sub-problem succeed, then for any variable $X \in Eq$, can have its domain intersected to $\mathcal{D}_{Eq}(X)$. Algorithm 19 (`repairability-`Γ) implements these modifications.

First the set *covered* is initialised as the empty set in Line 1. This set will be used to store the indices of the breakages that are known to be repairable by multidirectionality. The set $\mathcal{G}$ contains all assigned variables (Line 2). We checked only the breakages which are not known to be repairable by multidirectionality (Line 3) and such that their neighbourhood up to a distance $b+1$ is not entirely contained into $\mathcal{G}$ (Line 4). Then every breakage A satisfying these two preconditions are checked as follows:

- The sub-problem $\mathcal{P}^{\varphi,A}$ is created as shown in Table 5.2.
- The generalised arc consistent closure of $\mathcal{P}^{\varphi,A}$ is computed, if this closure is empty, then the algorithm fails (Line 5).
- All variables that must be equal in $\mathcal{P}$ and $\mathcal{P}^{\varphi,A}$ are stored in the set Eq and have their domain stored in the array $\mathcal{D}_{Eq}$ (line 6, 7 and 8).
- A solution of $\mathcal{P}^{\varphi,A}$, restricted to variables assigned in the main solution φ is computed, if no such solution exists, the algorithm fails (Line 9).
- The variables of $\mathcal{P}$ have their domain intersected with the domains stored in Eq (Line 10).

- If some values have been removed, the main partial solution φ is made generalised arc consistent (Line 11).

Algorithm 19 `repairability-Γ`

```
   Data   : $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$, $\varphi$, $\mathcal{F}[= \mathcal{X}]$, $a$, $b$
   Result : Check the repairability of grounded breakages
1  $covered \leftarrow \emptyset$;
2  $\mathcal{G} \leftarrow \{X \mid |\varphi(X)| = 1\}$;
   foreach $A \subseteq (\mathcal{X} \setminus \mathcal{F})$ such that $|A| \leq a$ do
3      if $A \notin covered$ then
4          if $\Gamma_{b+1}(A) \not\subseteq \mathcal{G}$ then
5              if ¬preprocess($\mathcal{P}^{\varphi,A}$) then return false;
6              $Dff \leftarrow \{X_i \mid \mathcal{D}^{\varphi,A}(X_i) \cap \varphi(X_i) = \emptyset\}$;
7              $Eq \leftarrow (\mathcal{X} \setminus \Gamma_{|A|-|Dff|+b}(Dff))$;
8              foreach $X \in Eq$ do $\mathcal{D}_{Eq}(X) \leftarrow \mathcal{D}^{\varphi,A}(X)$;
9              if ¬solve($\mathcal{P}^{\varphi,A}|_{\mathcal{X}\setminus\mathcal{F}}$) then return false;
               $pruned \leftarrow$ false;
10             foreach $X \in Eq$ do
                   if $\varphi(X) \not\subseteq \mathcal{D}_{Eq}(X)$ then
                       $pruned \leftarrow$ true;;
                       $\varphi(X) \leftarrow \mathcal{D}_{Eq}(X)$;
11             if $pruned$ & ¬GAC($\mathcal{P}, \varphi, \mathcal{F}$) then return false;
12             foreach $A' \subseteq (A \cup B)\ \wedge\ |A'| \geq |A \cup B| - b$ do
                   $covered \leftarrow (covered \cup \{A'\})$;
   return true;
```

Figure 5.13: An algorithm for checking the repairability of a partial solution using neighbourhood-based inference.

5.6.1 Implementation

We now discuss the implementation and complexity of the improvements due to the techniques described in Sections 5.4, 5.5 and 5.6 over the `repairability` algorithm.

Neighbourhood: For every variable $X \in \mathcal{X}$, the neighbourhoods $\Gamma_k(X)$ for all values of k between 1 and $b+1$ included are computed and stored. This is done as a preprocessing step by a simple breadth first search on the constraint graph and $n(b+1)$ sets of size at most n are required to store the result. We need to run $n(b+1)$ times a breadth first search algorithm ($\mathcal{O}(n^2)$) on the constraint graph. The complexity of the preprocessing procedure thus is $\mathcal{O}(bn^3)$ where $n = |\mathcal{X}|$ is the number of variables, i.e., nodes in the constraint graph. However, this cost is amortised over the possibly exponential sized search tree so is often negligible. The neighbourhood $\Gamma_b(A)$ of a breakage A is computed dynamically (during search) by performing a union operation over the neighbourhood of the elements in A.

Multidirectionality: The notion of multidirectionality (Section 5.4.1) is implemented using a set *covered* (Algorithm 19, Line 1). This set contains a reference to any breakage that is covered through multidirectionality (Algorithm 19, Line 12) breakages in this set are not checked (Algorithm 19, Line 3). We used a simple algorithm introduced by Knuth [Knuth 04] to generate all breakages, i.e., combinations of $k \leq a$ variables. This algorithm generates the combinations in lexicographic order and therefore constitutes an ordering on these combinations. Moreover, given one combination in input, one can compute the rank of this combination in the ordering in linear time on the size of the tuple. For instance given the combination $c = (x_1, x_2, \ldots x_k)$ of k elements amongst n, the rank of c in the lexicographical ordering is:

$$rank(c) = \sum_{i=1}^{i=k} \binom{x_i}{i}$$

The size of the tuple is in our case a small constant, we thus have an efficient way of knowing if the breakage that we currently consider is covered by an earlier repair. Each time a new repair is computed, all breakages it covers are added to *covered*, then when we generate a combination, we simply check that its index is not in this set otherwise we do not need to find a repair for it. Notice that this set is potentially large ($\sum_{k=1}^{k=a} \binom{n}{k}$), however, each element is a simple integer and this proved manageable in practice. Moreover, we can bound this set and yet keep soundness and completeness of the main algorithm, although this means that we may solve unnecessarily some sub-problems.

Assigned neighbourhood: The sub-problems corresponding to breakages such that their neighbourhood up to a distance $b+1$ is entirely assigned are not checked as well (Algorithm 19, Line 4, as described in Section 5.4.2). This test is a simple set inclusion operation.

Equality Constraints: Following Section 5.3.2, some equalities (set Eq) are either deduced from the preprocessing of the sub-problem, using the neighbourhood notion as described in Section 5.3 (Algorithm 19, Line 7).

Pruning on the Master-problem: Finally, the partial solution on the master-problem is reduced accordingly to Section 5.6. After preprocessing the sub-problem (Algorithm 19, Line 5) the partial solution on variables linked with "equality" constraints are reduced to match the domains of the sub-problem (Line 10).

The search procedure `decomposition-backtrack-`Γ (Algorithm 20) is basically unchanged with respect to Algorithm 15, the call to the inference method `repairability` is simply replaced by `repairability-`Γ.

Algorithm 20 `decomposition-backtrack-`Γ

```
Data   : $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$, $\varphi$, $\mathcal{F}[= \mathcal{X}]$, $a$, $b$
Result : Does $\mathcal{P}$ admit an $(a, b)$-super-solution
if $\mathcal{F} = \emptyset$ then return true;
choose $X \in \mathcal{F}$;
save $\varphi$;
foreach $v \in \varphi(X)$ do
    $\varphi(X) \leftarrow \{v\}$;
    if GAC($\mathcal{P}' = (\mathcal{F}, \varphi, \mathcal{C})$) & repairability-$\Gamma$($\mathcal{P}, \varphi, \mathcal{F} \setminus \{X\}, a, b$) then
        if decomposition-backtrack-$\Gamma$($\mathcal{P}, \varphi, \mathcal{F} \setminus \{X\}, a, b$) then return true;
    restore $\varphi$;
return false;
```

Figure 5.14: A backtracking algorithm for finding (a, b)-*super*-solutions.

5.7 Theoretical Properties

5.7.1 Soundness and Completeness

We did not explicitly define a level of consistency for which `repairability-`Γ would be the closure algorithm. Moreover, since we do not restrict the "classical" consistency used in conjunction with this procedure, the exact level of consistency is not properly defined. Therefore we restrict our proofs to the soundness of the closure algorithm, i.e., this algorithm does not prune values that can participate to (a, b)-*super*-solutions and to the soundness and completeness of the `decomposition-backtrack-`Γ algorithm.

Theorem 33. `repairability-`Γ *is a sound algorithm for filtering domains for the* (a, b)-SuperCSP *problem.*

Proof. Applying GAC is clearly sound since (a, b)-*super*-solutions are solutions. We show that if the current partial solution φ is extendable to an (a, b)-*super*-solution f of $\mathcal{P}$, then `repairability-`Γ does not fail. Consider the (a, b)-*super*-solution f to which φ can be extended. Without loss of generality, let $A \subseteq \mathcal{X}$ be a breakage ($|A| \leq a$). We show that is there exists g a *b-repair* of A for f, then the sub-problem $\mathcal{P}^{f,A}$ is satisfiable:

- g is a solution of $\mathcal{P}$ and therefore satisfies all constraints in $\mathcal{C}$.
- g is such that $\Delta_A(f, g) = |A|$ therefore $\forall X \in A\ f(X) \neq g(X)$.
- The constraint Similarity$(\mathcal{X}^{\varphi,A}, f, A, b)$: is satisfied, since $\Delta(g, f) \leq |A| + b$.

It follows that g is a solution of $\mathcal{P}^{f,A}$. Therefore, if there exists an (a, b)-*super*-solution then `repairability-`Γ does not fail. □

Theorem 34. `decomposition-backtrack-`Γ *is a sound and complete algorithm for solving the* (a, b)-SuperCSP *problem.*

Proof. **Soundness:** We show that if $\mathcal{F} = \emptyset$ then the partial solution φ actually is an (a, b)-*super*-solution. For every breakage A, the sub-problem $\mathcal{P}^{\varphi,A}$ has been solved in the previous level of the search tree. Now consider the solution f defined by φ, that is, $f(X_i) = v \Leftrightarrow \varphi(X_i) = \{v\}$ and a solution g of $\mathcal{P}^{\varphi,A}$.

- g is clearly a solution of $\mathcal{P}$ as $\mathcal{P}^{\varphi,A}$ is strictly tighter than $\mathcal{P}$, indeed we have: $\forall X \in \mathcal{X}\ \mathcal{D}^{\varphi.A}(X) \subseteq \mathcal{D}(X)$ and $\mathcal{C} \subseteq \mathcal{C}^{\varphi.A}$.
- $\Delta_A(f, g) = |A|$ since $\forall X \in A\ f(X) \notin \mathcal{D}^{\varphi.A}(X)$.
- $\Delta(g, f) \leq |A| + b$ since g satisfies the constraint Similarity$(\mathcal{X}^{\varphi.A}, \varphi, A, b)$.

Completeness: We show that if there exists an (a, b)-*super*-solution then the algorithm `decomposition-backtrack-`Γ will succeed. The procedure `repairability-`Γ is sound, therefore no decision leading to a solution can be pruned. Therefore, if there exists an (a, b)-*super*-solution then `decomposition-backtrack-`Γ will eventually find it. □

5.7.2 Complexity

We study the complexity of the procedure `repairability-Γ`. In Figure 5.15 we show a simplified form of the procedure `repairability-Γ`, where unessential parts that do not contribute to the worst case complexity are omitted.

```
Algorithm 21 repairability-Γ
   Data   : $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$, $\varphi$, $\mathcal{F}[= \mathcal{X}]$, $a$, $b$
   Result : Check the repairability of grounded breakages
1  $covered \leftarrow \emptyset$;
2  foreach $A \subseteq (\mathcal{X} \setminus \mathcal{F})$ such that $|A| \leq a$ do
       if $A \notin covered$ then
3          if $\neg$solve($\mathcal{P}^{\varphi,A}|_{\mathcal{X}\setminus\mathcal{F}}$) then return false;
   return true;
```

Figure 5.15: An abstracted version of the `repairability-Γ` procedure.

Worst Case Time Complexity

This procedure cycles potentially through all breakages in loop 2. Since the number of breakages has no closed form, we define the function $T(n, k)$ as the number of subsets of cardinality k or less in a set of cardinality n. This number is equal to the sum of the binomial coefficients, $\binom{n}{i}$ for $1 \leq i \leq n$.

$$T(n,k) = \sum_{i=1}^{i=k} \binom{n}{i}$$

This number will also conveniently been used to express the maximum number of repairs. Given a constraint network $\mathcal{P}$ and a partial solution φ, let l be the number of grounded variables in φ, n the total number of variables $n = |\mathcal{X}|$, d the domain size and m the number of constraints $m = |\mathcal{C}|$. [1]

The GAC closure of φ is computed, the worst case complexity of an optimal GAC algorithm on binary constraints is $\mathcal{O}(md^2)$. Breakages are checked in turn. The number of breakages is, as shown in Section 5.2, is $T(l, a)$, hence loop 2 cycles at most $T(l, a)$ times. The procedure `solve` is typically carried out by a backtracking algorithm such as `MAC`, hence its worst case complexity is in general exponential. However, in

[1] We assume here that all domains have the same cardinality

this case, the Similarity constraint restricts the search space to a polynomial in b. Moreover the depth of the search tree is limited to l. Consider a breakage A and the corresponding sub-problem $\mathcal{P}^{\varphi,A}$. As a consequence of the Similarity constraint, at most b variables, besides those in A, can be assigned differently than in φ. There are less than $T(l,b)$ such sets of variables, and at most d^{a+b} possible assignments for the $a+b$ "reassigned" variables. Therefore the total number of solutions of $\mathcal{P}^{\varphi,A}$ is bounded above by $T(l,b)d^{a+b}$. The overall worst case complexity of the inference method introduced in this chapter therefore is $\mathcal{O}(T(l,a)T(l,b)d^b)$. Since $T(n,k)$ is itself bounded by n^k, the complexity can be seen as $\mathcal{O}(l^{a+b}d^{a+b})$.

Space complexity

The procedure `repairability-Γ` sequentially solves a number of sub-problems. These sub-problems have the same space complexity as the master problem since the Similarity is defined in intention. In fact, since they are solved sequentially, and since the constraints are shared with the master problem, only an additional domain relation need to be stored on memory ($\mathcal{O}(nd)$). Moreover, `repairability-Γ` uses a few data structures to store necessary information:

- The future breakages that are covered by multidirectionality are stored in the set *covered*. The number of breakages can in principle be $T(n,a) = \sum_{k=1}^{k=a} \binom{n}{k}$ which is in contradiction with our condition to keep a space complexity polynomial even when a grows. However, in practice this set needs relatively little space since only the indices of the breakages are stored. Moreover, the set *covered* is not necessary for soundness or completeness, therefore we can bound its space complexity arbitrarily.

- The indices of assigned variables are stored in the set $\mathcal{G}$, hence a space complexity in $\mathcal{O}(n)$.

- The indices of variables that are assigned differently (*Dff*) and that are involved in an equality constraint (*Eq*) are also bounded by the total number of variables $\mathcal{O}(n)$.

- The array $\mathcal{D}_{Eq}$, storing the domains of variables participating in an equality constraint has a $\mathcal{O}(nd)$ space complexity in the worst case.

Therefore, notwithstanding the set *covered*, the space complexity of the procedure `repairability-`Γ is not larger than that of the main backtracking procedure.

5.7.3 Comparison with Full Fault Tolerance Algorithms

In this section we compare the decomposition algorithm introduced in this chapter with the algorithms dedicated to full fault tolerance in Chapter 4. Here again we differentiate the static (closure as stand alone procedure) from the dynamic (closure within search) context, since, similarly to GAC+, applying the *repairable* procedure has different behaviour according to the context. We define the consistency *repairable* as the consistency resulting from the application of the procedure `repairability-`Γ in conjunction with generalised arc consistency.

Definition 27. *A constraint network $cn = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ is said repairable if and only if it is* GAC *and the closure of $\mathcal{P}$ by* `repairability-`Γ *is $\mathcal{P}$.*

Theorem 35 (Static viewpoint)**.** *repairable* $\simeq$ GAC+

Proof. Let $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ be a constraint network,

(*repairable* $\succeq$ GAC+) Suppose that GAC+ does not hold. Then either $\mathcal{P}$ is not GAC in which case *repairable*($\mathcal{P}$) clearly does not hold, or there exists a domain $\mathcal{D}(X)$ such that its GAC closure is a singleton $\{v\}$. In this case, the sub-problem $\mathcal{P}^{\mathcal{D},\{v\}}$ has no solution since the constraint $X \neq v$ cannot be satisfied whilst all other constraints stay the same.

(GAC+ $\succeq$ *repairable*) Suppose that GAC+ holds. Then the constraint network is GAC and any domain contains at least 2 values The procedure `repairability-`Γ checks only breakages for which all variables are ground. Therefore, after achieving the GAC closure, if no domain is reduced to a singleton, no breakage will be checked, hence *repairable* holds. □

Theorem 36 (Dynamic viewpoint)**.** *repairable* $\simeq$ GAC($\mathcal{P} + \mathcal{P}$)

Proof. Let $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ be a constraint network,

(*repairable* $\succeq$ GAC($\mathcal{P} + \mathcal{P}$)) Suppose that for some partial solution φ on $\mathcal{P}$ *repairable* holds. We consider the same partial solution extended to $\mathcal{P} + \mathcal{P}$. The constraints in $\mathcal{P}$ are GAC since φ is GAC. Without loss of generality, let X be a variable, and consider the sub-problem $\mathcal{P}^{\varphi,X}$. There exists a partial solution ψ such that $\psi(\mathcal{X} \setminus X) = \varphi(\mathcal{X} \setminus X)$ and $\psi(X) \neq \varphi(X)$ since *repairable* holds. Therefore, if GAC is enforced on $\mathcal{P} + \mathcal{P}$, the domain of X^+ will contain at least the value assigned to X in ψ: $(\varphi(X) \cup \psi(X)) \subseteq \mathcal{D}^+(X^+)$. The duplicate constraints are GAC since φ is GAC and the disequality constraints are GAC since $\varphi(X) \neq \psi(X)$.

(GAC$(\mathcal{P}+\mathcal{P}) \succeq$ *repairable*) suppose that for some partial solution φ on $(\mathcal{P}+\mathcal{P})$, GAC$(\mathcal{P}+\mathcal{P})$ holds.

First, we show that all breakages checked by `repairability-`Γ produces satisfiable sub-problems. We consider, without loss of generality, a breakage $\{X\}$ on φ restricted to the variables in $\mathcal{P}$. If $|\varphi(X)| \neq 1$, this breakage is not checked, hence we can assume that the image of X by φ is a single value v. Now, since GAC$(\mathcal{P}+\mathcal{P})$ holds for the partial solution φ, there exists a value $w \neq v$ such that $w \in \mathcal{D}^+(X^+)$. Moreover, consider the partial solution ψ, equal to φ on any variable $Y \neq X$, and mapping X to $\{w\}$. First, observe that $(\mathcal{X}, \psi, \mathcal{C})$ is GAC since $(\mathcal{X} \setminus \{X\} \cup \{X^+\}, \varphi, \mathcal{C})$ is GAC. Moreover, all variables that are assigned in φ are assigned in ψ, and since φ and ψ agree on all variables but X, ψ is a solution for $\mathcal{P}^{\varphi,\{X\}}$.

Second we show that if the algorithm `repairability-`Γ was to prune a value, this value would not be GAC in $\mathcal{P}+\mathcal{P}$ with respect to φ. In fact, for a breakage $\{X\}$, there is an equality constraint on all variables in $\mathcal{X} \setminus \{X\}$. Therefore, a value needs to be pruned if it has no GAC support on $\mathcal{D}(X) \setminus \varphi(X)$ after the GAC closure is computed. However the GAC closure of $\mathcal{D}(X) \setminus \varphi(X)$ is identical to the GAC closure of $\mathcal{D}^+(X^+)$, since the same constraints apply, with the exception of $X \neq X^+$ which is already propagated in $\mathcal{D}(X) \setminus \varphi(X)$. □

5.8 Summary and Limitations

We have introduced a new algorithm for finding *super*-solutions. This procedure is, to our knowledge, the only available method for finding (a, b)-*super*-solutions for unrestricted values of a and b. We first introduced a brute force backtracking procedure, `decomposition-backtrack`. This procedure dynamically creates, at each node in the search tree, a sub-problem for every breakage such that a solution of this sub-problem

$$\begin{tikzcd}
 & \text{GAC}(P \times P) \arrow[d, "\succ"] & \\
 & \textit{super}\text{-GAC}(P) \arrow[dl, "\succ"'] \arrow[d, "\succ"] \arrow[dr, "\succ"] & \\
\text{GAC}(P + P) \arrow[r, leftrightarrow, "\simeq"'] & \text{GAC+}(P) \arrow[r, leftrightarrow, "\simeq"'] & \textit{repairable}
\end{tikzcd}$$

(a) Static context

$$\begin{tikzcd}
 & \text{GAC}(P \times P) \arrow[d, "\succ"] & \\
 & \textit{super}\text{-GAC}(P) \arrow[dl, "\succ"'] \arrow[dr, "\succ"] & \\
\text{GAC}(P + P) \arrow[rr, leftrightarrow, "\simeq"'] \arrow[dr, "\succ"'] & & \textit{repairable} \arrow[dl, "\succ"] \\
 & \text{GAC+}(P) &
\end{tikzcd}$$

(b) Dynamic context

Figure 5.16: The relation between consistencies (reads if **tail** holds then **head** holds).

is a repair of the main solution. We showed that it is possible to take advantage of the resolution of sub-problems in order to reduce the overall search space. We can deduce that some variables must be equal in the master problem and in a given sub-problem. In order to do so, we proposed a propagation algorithm for the SIMILARITY constraint. The idea is that when a variable need to be reassigned as a response to a breakage, there must be an uninterrupted chain of changes to the breakage, that is, a path in the constraint graph. This reasoning, in conjunction with a classical consistency processing, can be used to infer a number of equality constraints. When, for a variable involved in an equality constraint, some pruning occurs in the sub-problem while preprocessing, the same pruning can be done in the master problem. This, in consequence, reduces the search space of the master problem, hence the number of sub-problems that need be solved.

We then analysed the theoretical properties of this algorithms. The worst case time complexity for an inference step is in $\mathcal{O}(n^{a+b}d^{a+b})$. It is thus very dependent on the size of the parameters a and b. We compared this algorithm with the methods for finding $(1, 0)$-*super*-solutions introduced in Chapter 4.

We believe that this algorithm could be further developed in several dimensions. For instance, for large values of the parameters a and b we may need a radically different approach. Another drawback is that if the method has some "look-ahead" aspects it relies too heavily on a “look-back” reasoning which is often not as efficient. Indeed, part of the reasoning applies to the future, i.e., not already assigned, variables, such as the GAC processing and the pruning due to equality constraints, however the resolution itself of sub-problems is essentially a “look-back” technique since it checks already assigned variables. We believe that the difficulty of designing a purely “look-ahead” inference method is linked to the fact that the (a, b)-repairability is not a local property, hence requires global reasoning.

Chapter 6

Partial Fault Tolerance

6.1 Introduction

In this chapter we introduce a number of algorithms for optimising solution robustness. The algorithms introduced in Chapters 4 and 5 have several drawbacks. First and most importantly there is not always an (a, b)-*super*-solution for some given a and b. It also appears that finding (a, b)-*super*-solution is significantly more difficult than finding solution on problems of comparable sizes. It was shown, in Chapter 3 that several tractable classes of constraint satisfaction problems become NP-hard when searching *super*-solutions. Moreover it was shown in Chapter 5 that local consistency properties, largely responsible for the effectiveness of constraint programming in general, cannot be applied directly to ensure the repairability of solutions.

The situation is very close to that faced with **over-constrained** problems. A constraint network is said to be over-constrained if it admits no solution. Notice that in practice, a problem may be considered over-constrained because no solution can be found with the available methods, whilst one or more solutions could possibly exist. To answer this problem, a number of frameworks have been introduced, for instance **Partial Constraint Satisfaction**, [Freuder 89, Freuder 92], or **Weighted Constraint Satisfaction**, to deal with such problems. The idea behind all these frameworks is to relax the constraints, and to find a solution satisfying as much as possible the relaxed constraints. The same idea can be applied to *super*-solutions. We consider two ways of relaxing the robustness condition, both introduced and analysed in Section 3.3.3. These approaches solve most of the drawbacks mentioned earlier. Indeed the resulting Branch

& Bound algorithms will typically be "anytime". We typically start from a solution found via the best classical method for solving the problem, and then only tighten the robustness condition in a Branch & Bound process. Therefore, even if no *super*-solutions exist, a solution can be returned any time after the base algorithm provided one. The more time we spend on searching, the more robust the solution eventually returned will be.

In the first partial problem, MinBCSP, we want to find a solution minimising the value of b for which it is an (a, b)-*super*-solution. In fact, this problem can be solved as a sequence of regular SuperCSP problem, using the algorithm introduced in Chapter 5. We discuss in Section 6.2 the respective merits of using a "top-down" strategy where the value of b is decreased until no existential-(a, b)-*super*-solution can be found, or a "bottom-up" strategy where instead the value of b is increased until the first existential-(a, b)-*super*-solution is found.

The second partial problem, MaxRepairCSP, has several advantages over the first approach. The idea here is to find the solution with greatest repairability. The existential-(a, b)-repairability of a solution is defined in Section 3.2.3 (def. 10), as the number of breakages that can be repaired. We introduce three Branch & Bound algorithms for maximising the solution robustness. The first and second, are restricted to $(1, 0)$-*super*-solutions. They are described in Section 6.3, and respectively use the closure algorithms introduced in Section 4.4.2 and 4.4.3, extending the corresponding satisfaction algorithms. The third algorithm maximises the existential-(a, b)-repairability of a solution, with unrestricted a and b, and extends the satisfaction algorithm introduced in Chapter 5. The inference mechanisms, introduced in the same chapter cannot, however, be used exactly as stated and needs to be adapted to this algorithm. We show, in Section 6.4, that the same type of inference can still be achieved, however, using a more complex system of counters.

In Section 6.2, we introduce an algorithm for minimising the value of b for which there exists a $(1, b)$-*super*-solution and discuss the complexity of this problem. In Section 6.3, we introduce two Branch & Bound algorithms for maximising the $(1, 0)$-repairability of a solution, extending respectively `MAC+` and *super*-`MAC`. Finally, in Section 6.4, we introduce a Branch & Bound algorithm for maximising the (a, b)-

repairability of a solution extending the algorithm `decompose-backtrack-`Γ.

6.2 Minimising the Repair Size

This relaxation of SUPERCSP has been introduced in Chapter 3. We still insist that the solution should be an (a, b)-*super*-solution, however the value b is relaxed. An optimal solution for this problem is an (a, b)-*super*-solution with minimal value of b.

6.2.1 Objective Function

Given a constraint network $\mathcal{P} = \{\mathcal{X}, \mathcal{D}, \mathcal{C}\}$, the objective function Φ that we want to minimise is the value of b for which $\mathcal{P}$ admits an (a, b)-*super*-solution.

$$\Phi(f) = min\{b \mid f \text{ is an } (a, b)\text{-}\textit{super}\text{-solution of } \mathcal{P}\}$$

The problem of computing $\Phi(f)$, even when provided with a solution f is NP-hard. This problem has been introduced in Section 3.3.3.

6.2.2 Inference Method: "b^{min}-*repairable*"

The search is controlled by the following relation that bounds the search space toward improving solutions:

$$ub > min\{b \mid \exists f \text{ an } (a, b)\text{-}\textit{super}\text{-solution of } \mathcal{P}\}$$

This is the regular condition that we add to a CSP for finding (a, b)-*super*-solutions. In fact, we show that we can bound the search using the regular inference method for SUPERCSP, i.e., the procedure `repairability-`Γ introduced in Chapter 5 (Algorithm 19). We denote b^{min}-*repairable* this inference method within a Branch & Bound context.

6.2.3 Closure Algorithm for "b^{min}-*repairable*"

The procedure `repairability-`Γ can be used without modification as inference method for this objective function. We can therefore solve MINBCSP with a Branch & Bound algorithm using the `Filtering` procedure shown in algorithm 22. However, there is a notable difference when using this algorithm in this dynamic context. In the previous chapter, the value of b was fixed and often assumed to be a small constant.

This is no longer the case here since we typically start the search with b set to an upper bound and progressively decrease its value. Propagating the consequences of the relation $\Phi(\varphi) < ub$ on a partial solution φ, or even computing the value of $\Phi(f)$ when a new solution f is found is considerably harder because the value of b is not bounded. We showed in Theorem 3 that given a constraint network $\mathcal{P}$, a solution f of $\mathcal{P}$ and an integer b, the problem of deciding if f is indeed an (a, b)-*super*-solution of $\mathcal{P}$ is NP-complete. Algorithm 23 computes this value using the `repairability-`Γ procedure and a binary search, whilst algorithm 22 implements the filtering method for MinBCSP using the procedure `repairability-`Γ introduced in Section 5.6 (Algorithm 19).

6.2.3.1 Commented Pseudo Code

Algorithm 22 b^{min}*-repairable*

```
    Data   : $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$, $\varphi$, $\Phi$, $\mathcal{F}$, $ub$
    Result : Is the GAC closure of $\mathcal{P}$ $ub$-repairable
    return( GAC($\mathcal{P}' = (\mathcal{F}, \varphi, \mathcal{C})$) & repairability-$\Gamma(\mathcal{P}, \varphi, \mathcal{F} \setminus \{X\}, a, ub - 1)$ );
```

Figure 6.1: An inference method for MinBCSP.

Algorithm 23 `min-B`

```
    Data   : $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$, $f$, $a$
    Result : $\Phi(f) = min\{b \mid f$ is an $(a, b)$-super-solution of $\mathcal{P}\}$
1   $ub \leftarrow n + 1$;
2   $lb \leftarrow 0$;
    while $ub > lb$ do
3       if repairability-$\Gamma(\mathcal{P}, f, a, \lfloor \frac{ub+lb}{2} \rfloor)$ then
4           $ub \leftarrow \lfloor \frac{ub+lb}{2} \rfloor$;
        else
5           $lb \leftarrow \lfloor \frac{ub+lb}{2} \rfloor + 1$;
    return $ub$;
```

Figure 6.2: An algorithm for computing the minimum maximum repair size of a solution.

Both algorithms are straightforward implementations using the procedure `repairability-`Γ.

In algorithm 22, given an upper bound ub, apply exactly the filtering done by the satisfaction algorithm described in the previous chapter. In algorithm 23 the value of $\Phi(f)$ for a solution f is computed in a binary search. At any given time, ub is greater or equal than $\Phi(f)$ whilst lb is less than or equal to $\Phi(f)$. Indeed when the condition in Line 3 succeeds then ub is set to $\lfloor\frac{ub+lb}{2}\rfloor$, however f is $\lfloor\frac{ub+lb}{2}\rfloor$-*repairable*. Similarly, when it fails, then lb is set to $\lfloor\frac{ub+lb}{2}\rfloor + 1$, however f is not $\lfloor\frac{ub+lb}{2}\rfloor$-*repairable*.

6.2.4 Theoretical Properties

Theorem 37. *b^{min}–repairable is a sound inference method for the* MinBCSP *problem.*

Proof. We show that no value that can participate in an improving solution is pruned when applying the procedure b^{min}–*repairable*. Let ub be the current upper bound, i.e., ub is greater then or equal to $\Phi(f)$ for any solution f found so far. An improving solution therefore is $(ub-1)$-repairable, hence since the procedure `repairability-`Γ is sound, applying it with b set to $ub-1$ does not remove any value participating in an $(a, ub-1)$-*super*-solution. □

Theorem 38. *A Branch & Bound algorithm using b^{min}–repairable is a sound and complete method for solving* MinBCSP*.*

Proof. **Soundness:** We show that the solution returned is optimal. Let ub be the optimal value of the objective function as found by a Branch & Bound algorithm using b^{min}–*repairable* as inference method. The algorithm stops when the constraint network $\mathcal{P}$ augmented with $\{\Phi(\mathcal{P}) < ub\}$ is exhausted and proved unsatisfiable. Therefore, $\mathcal{P}$ does not admit an $(a, ub-1)$-*super*-solution.

Completeness: This method is complete since the Branch & Bound search is sound and complete and the procedure `repairability-`Γ is sound. □

This problem (MinBCSP) has been classified into the NP optimisation complexity class ($\mathrm{P}^{\mathrm{NP}[log(n)]}$). However, whilst, in most optimisation problems, computing the objective value of a solution and propagating with respect to this value can typically be achieved in polynomial time, this task is itself $\mathrm{P}^{\mathrm{NP}[log(n)]}$-complete here. Therefore the algorithm we are introducing for this problem is unlikely to be very efficient.

Theorem 39. *Computing the minimum value of b for which a solution is an (a, b)-super-solution of a constraint network is $P^{\mathrm{NP}[log(n)]}$-complete*

Proof. **Computing the minimum value of b is in $\mathbf{P}^{\mathrm{NP}[log(n)]}$:** We need to show that a polynomial number of calls to a NP oracle is sufficient to solve this problem. We use the following oracle: "does there exists a b-*repair* for each breakage?". This problem is in NP, the polynomial witness is the set of *repair*s. We can proceed by dichotomy on the value of b. Since b is bounded by n, only $log(n)$ calls are needed.

Computing the minimum value of b is $\mathbf{P}^{\mathrm{NP}[log(n)]}$-hard: We reuse the construction described in the proof of Theorem 3 to reduce MaxClique to the problem of computing the minimum value of b for which a solution f is an (a, b)-*super*-solution. Given a graph $G = (V, E)$, we introduce a constraint network $\mathcal{P}$ with $n = |V|$ Boolean variables $X_1, \ldots X_n$ standing for the nodes of the graph, and one extra Boolean variable X_0. Then, for every pair of nodes $v_i, v_j \in V$ such that $(v_i v_j) \notin E$, we introduce the constraint $C(X_0, X_i, X_j) = (X_0 = 0 \Rightarrow (X_i + X_j \leq 1))$. Finally the "query" solution will be $f : X_i \rightarrow 1$, and the parameters a and b will be set respectively to 1 and $n - K$. As shown in the NP-completeness proof, there exists a b-*repair* of $\{X_0\}$ for f if and only if there exists a clique of size $n - b$ in G. Therefore, since all other breakages admit a 0-*repair*, computing the minimum value of b is equivalent to computing the size of the maximum clique. □

Consequently, the inference method based on `repairability-Γ` may be prohibitive. Indeed it requires one to solve this NP-hard problem when a solution is found and then performs inference with the `repairability-Γ` procedure, which has exponential time complexity when b is not bounded.

6.2.5 Alternative Approaches

The classical Branch & Bound algorithm can be seen as a **top-down** approach since we start with a large, pessimistic value for the objective function, and gradually decrease it until no more solutions can be found. The advantage of this approach is that the search tree is explored only once. On the other hand if we start with an optimistic value for the objective function and increase it until a solution is found, a sequence of search trees need to be exhausted. Moreover, the algorithm is no longer anytime since the optimal solution is found first. It may however be worthwhile using such a **bottom-up** approach for the MinBCSP problem since the complexity of the inference method is exponential in the value of the objective function. Alternatively, we may want to use a binary search procedure.

6.3 Maximising Full Fault Tolerance

In this section, we study the problem of finding partial $(1,0)$-*super*-solutions and extend two satisfaction algorithms introduced in Chapter 4 to find maximally $(1,0)$-repairable solutions.

6.3.1 Objective Function

The objective function Φ_0 that we want to maximise is the $(1,0)$-repairability (see Definition 10, Section 3.2.3). Grounded to the case where $a = 1$ and $b = 0$, yields the following formula, where f is a solution of the constraint network $\mathcal{P} = \{\mathcal{X}, \mathcal{D}, \mathcal{C}\}$.

$$\Phi_0(f) = |\{X | X \in \mathcal{X}, \exists g \in sol(\mathcal{P}) \ s.t. \ g(X) \neq f(X) \ \wedge \ \forall Y \neq X \ \ g(Y) = f(Y)\}|$$

We shall instead minimise the complement of Φ_0 to the total number of breakages:

$$\Phi(f) = \sum_{k=1}^{k=a} \binom{n}{k} - \Phi_0(f)$$

6.3.2 Inference Method: “GAC+^{max}”

Here, contrary to the situation in Section 6.2, the inference method used in the satisfaction algorithm cannot be used without modification. However, we shall see that the principles used for reasoning about fault tolerant solutions can be adapted to optimise $(1,0)$-repairability instead. We first describe a simple filtering method based on `GAC+`. The idea is, after the current partial domains have been made generalised arc consistent, to count the number of singleton domains. Any variable whose domain is singleton cannot be repaired. For any solution f achievable with the current domain (sD), the following inequality holds:

$$|\{X \mid (1 = |r\mathcal{D}(X)|)\}| \leq \Phi(f)$$

The inference method GAC+^{max} therefore prunes any branch of the search tree where the number of singleton *repair*-domains is greater than or equal to *ub*.

6.3.3 Closure Algorithm for "GAC+^{max}"

As in Chapter 4, the search algorithm (Branch & Bound in this case) uses two partial solutions $s\mathcal{D}$ and $r\mathcal{D}$ instead of one. This algorithm is similar to algorithm 13 though it backtracks only when the number of singleton domains becomes larger than the current upper bound. In other words, the following relation is used to bound search:

$$|\{X \mid (1 = |r\mathcal{D}(X)|)\}| < ub$$

6.3.3.1 Commented Pseudo Code

Algorithm 24 `GAC+`^{max}

```
  Data   : $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$, $s\mathcal{D}[= \mathcal{D}]$, $r\mathcal{D}[= \mathcal{D}]$, $ub$
  Result : $s\mathcal{D}, r\mathcal{D}$, the GAC+$^{max}$ closure of $\mathcal{P}$
  $Q \leftarrow \mathcal{C} \cup \{C(Y, X) \mid C(X, Y) \in \mathcal{C}\}$;
  while $Q \neq \emptyset$ do
      select and delete any $C(X_i, X_j)$ from $Q$;
      $pruned \leftarrow$ propagate+$^{max}(C(X_i, X_j), s\mathcal{D}, r\mathcal{D})$;
      if $s\mathcal{D}(X_j) = \emptyset$ then return false;
      if $pruned$ then $Q \leftarrow Q \cup \{C(X_j, X_k)\ \forall k\}$;
1 return $(|\{X \mid |r\mathcal{D}(X)| = 1\}| < ub)$;
```

Algorithm 25 `propagate+`^{max}

```
  Data   : $C(X_i, X_j)$, $s\mathcal{D}$, $r\mathcal{D}$
  Result : The GAC+ closure of $X_j$ with respect to $C(X_i, X_j)$
  $pruned \leftarrow$ false;
  foreach $w \in r\mathcal{D}(X_j)$ do
      if $\nexists v \in s\mathcal{D}(X_i)$ s.t. $\langle v, w \rangle \in C(X_i, X_j)$ then
          $s\mathcal{D}(X_j) \leftarrow s\mathcal{D}(X_j) \setminus \{w\}$;
          $r\mathcal{D}(X_j) \leftarrow r\mathcal{D}(X_j) \setminus \{w\}$;
          $pruned \leftarrow$ true;
  return $pruned$;
```

Figure 6.3: An algorithm for computing the GAC+^{max} closure of a constraint network.

The propagation algorithm, `propagate+`^{max} is identical to `propagate+`. Both procedures simply achieve GAC on two domain relations ($s\mathcal{D}$ and $r\mathcal{D}$), and with respect to the decisions made on the first domain relation ($s\mathcal{D}$). The only difference with the satisfaction version of this inference method (`GAC+`) is that a failure occurs if at least ub *repair*-domains are reduced to a singleton, instead of only one for `GAC+`. This is checked in Line 1 of algorithm 24.

6.3.4 Theoretical Properties

Theorem 40. `GAC+`^{max} *is a sound inference method and runs in* $\mathcal{O}(md^2)$ *on binary networks.*

Proof. **Soundness:** Since the filtering is restricted to the GAC closure, all we have to prove is that the lower bound computed at each node is valid, i.e., that at least this number of breakages cannot be repaired in any sub-solution.

Consider a partial solution $s\mathcal{D}$, and a variable $X \in \mathcal{X}$ such that there is only one possible value $v \in r\mathcal{D}(X)$ consistent with $s\mathcal{D}$. We showed in the soundness proof for GAC+ that the breakage $\{X\}$ does not admit a 0-*repair*. Hence the total number of repairable breakages cannot exceed the number of variables such that their domain under the relation $r\mathcal{D}$ has size 2 or more.

Complexity: As for GAC+, the complexity is dominated by the GAC closure, hence $\mathcal{O}(md^2)$ for a binary constraint network. □

Theorem 41. *A Branch & Bound algorithm using* `GAC+`^{max} *is a sound and complete method for solving* $(1,0)$-MaxRepairCSP.

Proof. **Soundness:** We show that the solution returned is optimal. Let ub be the optimal value of the objective function as found by a Branch & Bound algorithm using `GAC+`^{max} as inference method. The algorithm stops when the constraint network $\mathcal{P}$ augmented with $\{\Phi(\mathcal{P}) < ub\}$ is exhausted and proved unsatisfiable. Therefore, $\mathcal{P}$ does no admit a solution whose repairability is less than $\sum_{k \leq a} \binom{n}{k} - ub$.

Completeness: This method is complete since the Branch & Bound search is sound and complete and the procedure `GAC+`^{max} is sound. □

6.3.5 Inference Method: "*super*-GAC^{max}"

The second inference method, *super*-GAC, introduced in Section 4.3.2, can also be adapted for maximising the $(1, 0)$-repairability. However, we need to significantly change the algorithm since assignments that are not *super*-GAC can still be used in an optimal solution, since it is not necessarily a $(1, 0)$-*super*-solution. Since this inference method is essentially an extension of the *super*-`AC` algorithm, it is restricted to binary constraint networks. Here again, extending this algorithm to the non-binary case is not straightforward. Global constraints can be developed using the same type of inference as presented in this section, however we do not give example of such "soft-*super*-global constraint" in this dissertation.

The basic principle of *super*-GAC is to partition generalised arc consistent values into two categories:

- sD contains values that have two GAC supports, one of them being itself contained in $s\mathcal{D}(X)$ for any neighbouring variable X.
- rD contains values that have at least one GAC support in sD(X) for any neighbouring variable X.

Any value that does not fit in one of these two categories can safely be removed. Clearly this method cannot be used in a Branch & Bound framework, since some breakages can be left unrepaired. However, some inference can still be made. The idea is, whenever a value v for a variable X violates the first rule, to store the variables for which the second support does not exist. Now, if we commit to this value, then all the variables stored in this way will cease to be repairable, since the assignment $X = v$ is consistent with one and only one assignment for each of them.

6.3.6 Closure Algorithm for "*super*-GAC^{max}"

We adapt the algorithm *super*-`AC`, introduced in Section 4.4.3. Two extra data-structures, *unrep* and *PList* are introduced to store respectively the variables known to be unrepairable, and that would cease to be repairable if a certain assignment was made. The first, *unrep*, is a set of variables that cannot be repaired, given the decisions made so far. This set is initialised as the set of variables with *repair*-domains reduced to a singleton, since, by using the same reasoning as for GAC+, these variables have no alternative:

$$unrep = \{X \mid (1 = |r\mathcal{D}(X)|)\}$$

The second data-structure, *PList*, is an array of sets, mapping the values of each variable to a set of variables that would cease to be repairable if this value was used in assignment. If $\mathcal{X}$ is the set of variables and Λ the set of values in the constraint network, then *Plist* maps tuples $\mathcal{X} \times \Lambda$ to a subset of $\mathcal{X}$:

$$PList : \mathcal{X} \times \Lambda \mapsto 2^{\mathcal{X}}$$

This data structure is initially empty. Then, when a value $v \in s\mathcal{D}(X)$ is such that only one value support it in the *repair*-domain of some variable Y, the variable Y is added to the set $PList[X, v]$. Indeed is the search algorithm commits to the assignment $X = v$, then no alternative is possible for Y, since only one value is consistent with $X = v$. When revising the domain of a variable X, with respect to an arc $C(X, Y)$, we update $PList$ with the following rule:

$$\begin{gathered} Y \in PList[X, w] \\ \Leftrightarrow \\ |\{v \mid v \in r\mathcal{D}(Y) \;\wedge\; \langle w, v\rangle \in C(X, Y)\}| = 1 \end{gathered}$$

Moreover, the set *unrep* is updated after each domain revision. Indeed any variable in the intersection of the sets $Plist[X, v]$ for all v in $s\mathcal{D}(X)$ cannot be repaired since any choice of assignment for X would render them unrepairable. Therefore we enforce the following relation:

$$\forall X \in \mathcal{X},\ (\bigcap_{v \in s\mathcal{D}(X)} PList[X, v]) \subseteq unrep$$

6.3.6.1 Commented Pseudo Code

We describe algorithms 26 and and compare them to their satisfaction equivalent introduced in Section 4.4.3 (Algorithm 8 and 9). In the top-level closure algorithm (Algorithm 26) the two data-structures, *unrep* and *Plist* are initialised in lines 1 and 2 respectively. The only difference, in the top level closure algorithm is that when a

Algorithm 26 *super*-$\texttt{AC}^{max}$

```
    Data   : $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$, $s\mathcal{D}[= \mathcal{D}]$, $r\mathcal{D}[= \mathcal{D}]$, $ub[= n]$
    Result : $s\mathcal{D}$, $r\mathcal{D}$, the super-GAC$^{max}$ closure of $\mathcal{P}$
1   $unrep \leftarrow \{X \mid X \in \mathcal{X} \ \wedge \ |r\mathcal{D}(X)| = 1\}$;
2   foreach $X \in \mathcal{X}$ do
        foreach $v \in s\mathcal{D}(X)$ do  $PList[X, v] \leftarrow \emptyset$;
    $Q \leftarrow \mathcal{C} \cup \{C(Y, X) \mid C(X, Y) \in \mathcal{C}\}$;
    while $Q \neq \emptyset$ do
        select and delete any $C(X_i, X_j)$ from $Q$;
        $size = |unrep|$;
        $pruned \leftarrow$ propagate-sup$^{max}(X_i, X_j, unrep, ub, List)$;
        if $size < |unrep|$ then
3           $Q \leftarrow \mathcal{C} \cup \{C(Y, X) \mid C(X, Y) \in \mathcal{C}\}$;
        else
            if $pruned$ then  $Q \leftarrow Q \cup \{C(X_j, X_k)\ \forall k\}$;
        if $s\mathcal{D}(X_j) = \emptyset$ then return false;
    return true;
```

Algorithm 27 $\texttt{propagate-sup}^{max}$

```
    Data   : $C(X_i, X_j)$, $s\mathcal{D}$, $r\mathcal{D}$, $PList$, $DList$, $ub$
    Result : The super-GAC$^{max}$ closure of $X_j$ with respect to $C(X_i, X_j)$
    $pruned \leftarrow$ false;
    foreach $w \in r\mathcal{D}(X_j)$ do
        $sup \leftarrow \emptyset$;
1       foreach $v \in r\mathcal{D}(X_i)$ do
            if $\langle v, w\rangle \in C(X_i, X_j)$ then  $sup \leftarrow sup \cup \{v\}$;
        if $sup \cap s\mathcal{D}(X_i) = \emptyset$ then
2           $r\mathcal{D}(X_j) \leftarrow r\mathcal{D}(X_j) \setminus \{w\}$;
            $pruned \leftarrow$ true;
        else
3           if $w \in s\mathcal{D}(X_j) \ \wedge \ |sup| < 2$ then
4               $PList[X_j, w] \leftarrow PList[X_j, w] \cup \{X_i\}$;
                if $|PList[X_j, w] \cup unrep| \geq ub$ then
                    $s\mathcal{D}(X_j) \leftarrow s\mathcal{D}(X_j) \setminus \{w\}$;
                    $pruned \leftarrow$ true;
    $s\mathcal{D}(X_j) \leftarrow s\mathcal{D}(X_j) \cap r\mathcal{D}(X_j)$;
    $unrep \leftarrow unrep \cup (\bigcap_{w \in s\mathcal{D}(X_j)} PList[X_j, w])$;
    return $pruned$;
```

Figure 6.4: An algorithm for computing the *super*-GAC^{max} closure of a constraint network.

new element is inserted into the set *unrep* all arcs are added to the queue (Line 3). Indeed, when any value that was consistent in the previous state of the set *unrep* can potentially be made inconsistent by this insertion.

As in the satisfaction algorithm, when revising the domain of a variable X_j against a recently modified variable X_i, the number of supports for each value is stored in the set *sup* (Line 1). When a value $v \in r\mathcal{D}(X_j)$ has no support in the *super*-domain of a variable X_i, v is pruned as in the satisfaction algorithm. Indeed it means that this value is inconsistent with all the possible sub-solutions. On the other hand, the condition triggering the removal of a value v from $s\mathcal{D}(X_j)$, whilst remaining in $r\mathcal{D}(X_j)$ is slightly different. The main difference here is that lacking an alternative support for a given constraint is not a sufficient condition as a certain number of unrepairable variables can be tolerated. However, a value can be inferred to participate only in sub-optimal solutions. Indeed the cardinality of *unrep* is a lower bound on the number of unrepairable variables. Moreover, the data-structure $PList$ associates to a tuple $\langle X, v\rangle$ a set of variables for which no alternative would be possible if the decision $X = v$ was taken. Therefore, if, for a given value $v \in s\mathcal{D}(X)$, the cardinality of the set $PList[X, v]$ added to the value of *unrep* is greater than or equal to the current upper bound *ub*, then v can be pruned from $s\mathcal{D}(X)$, however not from $r\mathcal{D}(X)$.

The corresponding test, in line 3 of algorithm covers this case. The first conjunct checks if the considered value is indeed in $s\mathcal{D}(X_j)$, since elements of $r\mathcal{D} \setminus s\mathcal{D}$ do not follow the same rule. The second conjunct states that w has only one support in the domain of X_i, hence the corresponding decision would make X_i not repairable, since this support would not have any alternative. When this conditions hold then X_i is added to the list of variables that would become unrepairable if the decision $X_j = v$ is taken ($PList[X_j, v]$). Clearly if the maximum number of variable tolerated to be unrepairable, *ub*, is less than or equal to $|unrep \cup PList[X_j, v]|$ then v is not a valid assignment for X_j and cannot be part of a solution with a repairability less than $n - ub$. Consequently v can be removed from $s\mathcal{D}(X_j)$, however, it can still possibly be an alternative for another assignment and should not be removed from $r\mathcal{D}(X_j)$.

Example 26. *Consider the constraint network shown in Figure 6.5. We trace the execution of the algorithm super-*$\texttt{AC}^{max}$ *on this instance.*

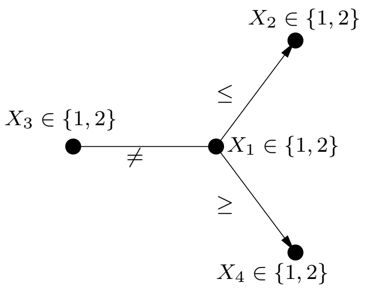


Figure 6.5: A constraint network.

In the first iteration, the constraint $(X_4 \leq X_1)$ *is propagated. The assignment* $X_4 = 2$ *has only one support on* X_1 *(*$X_1 = 2$*), therefore,* X_1 *is added to* $PList[X_4, 2]$*. The set unrep is left unchanged since* $PList[X_4, 1]$ *is empty. No constraint need be added to the queue* Q*. The state of the internal data-structures is shown in Figure 6.6:*

ub:	3			
$unrep$:	$\emptyset$			
	$PList[*,1]$	$PList[*,2]$	$s\mathcal{D}$	$r\mathcal{D}$
X_1	$\emptyset$	$\emptyset$	$\{1,2\}$	$\{1,2\}$
X_2	$\emptyset$	$\emptyset$	$\{1,2\}$	$\{1,2\}$
X_3	$\emptyset$	$\emptyset$	$\{1,2\}$	$\{1,2\}$
X_4	$\emptyset$	$\{X_1\}$	$\{1,2\}$	$\{1,2\}$

Q: $\{(X_1 \leq X_2)$
$(X_1 \neq X_3)$
$(X_1 \geq X_4)$
$(X_2 \geq X_1)$
$(X_3 \neq X_1)\}$

Figure 6.6: The data structures after iteration 1 of *super*-$\texttt{AC}^{max}$.

In the second iteration, the constraint $(X_3 \neq X_1)$ *is propagated. The assignment* $X_3 = 1$ *has only one support on* X_1 *(*$X_1 = 2$*) and similarly,* $X_3 = 2$ *has only one support* $X_1 = 1$*. Therefore,* X_1 *is added to* $PList[X_3, 1]$ *and to* $PList[X_3, 2]$*. Then* X_1 *is added to the set unrep since it belongs to both* $PList[X_3, 1]$ *and* $PList[X_3, 2]$*. All constraints but* $(X_3 \neq X_1)$ *are added to the queue* Q*. The state of the internal data-structures is shown in Figure 6.7:*

In the third iteration, the constraint $(X_4 \leq X_1)$ *is propagated, however, nothing is inferred. Then, in the fourth iteration, the constraint* $(X_2 \geq X_1)$ *is propagated. The assignment* $X_2 = 1$ *has only one support on* X_1 *(*$X_1 = 1$*), therefore,* X_1 *is added to* $PList[X_2, 1]$*. The set unrep is left unchanged since* $PList[X_2, 2]$ *is empty. No* *ub*: 3

unrep: $\{X_1\}$

	$PList[*,1]$	$PList[*,2]$	$s\mathcal{D}$	$r\mathcal{D}$
X_1	$\emptyset$	$\emptyset$	$\{1,2\}$	$\{1,2\}$
X_2	$\emptyset$	$\emptyset$	$\{1,2\}$	$\{1,2\}$
X_3	$\{X_1\}$	$\{X_1\}$	$\{1,2\}$	$\{1,2\}$
X_4	$\emptyset$	$\{X_1\}$	$\{1,2\}$	$\{1,2\}$

Q: $\{(X_1 \leq X_2)$ $(X_1 \neq X_3)$ $(X_1 \geq X_4)$ $(X_2 \geq X_1)$ $(X_4 \leq X_1)\}$

Figure 6.7: The data structures after iteration 2 of *super*-AC^{max}.

constraint need be added to the queue Q. The state of the internal data-structures is shown in Figure 6.8:

ub: 3

unrep: $\{X_1\}$

	$PList[*,1]$	$PList[*,2]$	$s\mathcal{D}$	$r\mathcal{D}$
X_1	$\emptyset$	$\emptyset$	$\{1,2\}$	$\{1,2\}$
X_2	$\{X_1\}$	$\emptyset$	$\{1,2\}$	$\{1,2\}$
X_3	$\{X_1\}$	$\{X_1\}$	$\{1,2\}$	$\{1,2\}$
X_4	$\emptyset$	$\{X_1\}$	$\{1,2\}$	$\{1,2\}$

Q: $\{(X_1 \leq X_2)$ $(X_1 \neq X_3)$ $(X_1 \geq X_4)\}$

Figure 6.8: The data structures after iteration 4 of *super*-AC^{max}.

In the fifth iteration, the constraint $(X_1 \geq X_4)$ is propagated. The assignment $X_1 = 1$ has only one support on X_4 ($X_4 = 1$), therefore, X_4 is added to $PList[X_1, 1]$. The set unrep is left unchanged since $PList[X_1, 2]$ is empty. No constraint need be added to the queue Q. The state of the internal data-structures is shown in Figure 6.9:

ub: 3

unrep: $\{X_1\}$

	$PList[*,1]$	$PList[*,2]$	$s\mathcal{D}$	$r\mathcal{D}$
X_1	$\{X_4\}$	$\emptyset$	$\{1,2\}$	$\{1,2\}$
X_2	$\{X_1\}$	$\emptyset$	$\{1,2\}$	$\{1,2\}$
X_3	$\{X_1\}$	$\{X_1\}$	$\{1,2\}$	$\{1,2\}$
X_4	$\emptyset$	$\{X_1\}$	$\{1,2\}$	$\{1,2\}$

Q: $\{(X_1 \leq X_2)$ $(X_1 \neq X_3)\}$

Figure 6.9: The data structures after iteration 5 of *super*-AC^{max}.

In the sixth iteration, the constraint $(X_1 \neq X_3)$ is propagated. The assignment

$X_1 = 1$ has only one support on X_3 ($X_3 = 2$) and similarly, $X_1 = 2$ has for only support $X_3 = 1$. Therefore, X_3 is added to $PList[X_1, 1]$ and to $PList[X_1, 2]$. Now the union of unrep $= \{X_1\}$ and $PList[X_1, 1] = \{X_3, X_4\}$ has a cardinality of 3, i.e. equal to ub, hence the value 1 is removed from $s\mathcal{D}(X_1)$. Finally, X_3 is added to the set unrep since it belongs to both $PList[X_1, 1]$ and $PList[X_1, 2]$. All constraints but $(X_1 \neq X_3)$ are added to the queue. The state of the internal data-structures is shown in Figure 6.10:

ub: 3
unrep: $\{X_1, X_3\}$

	$PList[*,1]$	$PList[*,2]$	$s\mathcal{D}$	$r\mathcal{D}$
X_1	$\{X_3, X_4\}$	$\{X_3\}$	$\{2\}$	$\{1,2\}$
X_2	$\{X_1\}$	$\emptyset$	$\{1,2\}$	$\{1,2\}$
X_3	$\{X_1\}$	$\{X_1\}$	$\{1,2\}$	$\{1,2\}$
X_4	$\emptyset$	$\{X_1\}$	$\{1,2\}$	$\{1,2\}$

Q: $\{(X_1 \leq X_2)$
$(X_1 \geq X_4)$
$(X_2 \geq X_1)$
$(X_3 \neq X_1)$
$(X_4 \leq X_1)\}$

Figure 6.10: The data structures after iteration 6 of *super*-AC^{max}.

In the seventh iteration, the constraint $(X_4 \leq X_1)$ is propagated, however, no pruning is performed. Then, in the eighth iteration, the domain of X_3 is revised with respect to the constraint $(X_3 \neq X_1)$. The value 2 has no support in $s\mathcal{D}(X_1)$ and thus is removed from both $r\mathcal{D}(X_3)$ and $s\mathcal{D}(X_3)$. The constraint $(X_1 \neq X_3)$ is added to the queue, however no pruning is performed. The state of the internal data-structures is shown in Figure 6.11:

ub: 3
unrep: $\{X_1, X_3\}$

	$PList[*,1]$	$PList[*,2]$	$s\mathcal{D}$	$r\mathcal{D}$
X_1	$\{X_3, X_4\}$	$\{X_3\}$	$\{2\}$	$\{1,2\}$
X_2	$\{X_1\}$	$\emptyset$	$\{1,2\}$	$\{1,2\}$
X_3	$\{X_1\}$	$\{X_1\}$	$\{1\}$	$\{1\}$
X_4	$\emptyset$	$\{X_1\}$	$\{1,2\}$	$\{1,2\}$

Q: $\{(X_1 \leq X_2)\}$
$(X_2 \geq X_1)\}$

Figure 6.11: The data structures after iteration 8 of *super*-AC^{max}.

In the next (ninth) iteration, the constraint $(X_2 \geq X_1)$ is propagated. The assignment $X_2 = 1$ has no support in $s\mathcal{D}(X_1)$ and thus the value 1 is removed from both $r\mathcal{D}(X_2)$ and $s\mathcal{D}(X_2)$. The constraint $(X_1 \leq X_2)$ is already in the queue and is propagated next. The assignment $X_1 = 2$ has only one support on X_2 ($X_2 = 2$).

Therefore, X_2 is added to $PList[X_1, 2]$. Since the union of unrep $= \{X_1, X_3\}$ and $PList[X_1, 2] = \{X_2, X_3\}$ has a cardinality of 3, i.e., greater than ub, the value 2 is removed from $s\mathcal{D}(X_1)$. The super-domain of X_1 is thus emptied and as a consequence, the algorithm fails. There is no solution with repairability 2 or less.

6.3.7 Theoretical Properties

Lemma 2. *If $Y \in PList[X, w]$, for any sub-solution f, if $f(X) = w$ then the breakage $\{Y\}$ is not 0-repairable.*

Proof. Suppose that $Y \in PList[X, w]$, then $X = w$ has only one GAC support, say v, in Y. Now consider a solution f, derived from the current domains, and involving the assignment $X = w$. The value assigned to Y must be v since it is the only support for $X = w$. There is thus no alternative for $Y = v$, hence the breakage $\{Y\}$ is not 0-repairable. $\square$

Theorem 42. *super*-GAC^{max} *is a sound inference method and runs in $\mathcal{O}(md(n+d)^2)$ on binary constraint networks.*

Proof. **Soundness:** We prove that the pruning achieved with `propagate-sup`^{max} is sound. The algorithm performs two types of inference. First, given a variable X, a value v with no support in the *super*-domain of some variable sharing a constraint with X, is removed from both $s\mathcal{D}$ and $r\mathcal{D}$ (Line 2). This inference is sound since v is arc inconsistent with respect to $s\mathcal{D}$. Therefore, there is no solution, derived from the current *repair*-domain that involves $X = v$, hence no $(1, 0)$-*super*-solution nor 0-*repair*.

The second type of inference is when a value is removed from the *super*-domain but not from the *repair*-domain because committing to this value would lead to a sub-optimal solution. We thus show that if such a decision was made the number of unrepairable variables in any sub-solution would be greater than or equal to the upper bound *ub*. By Lemma 2 we know that if the decision $X = v$ is made, then the variables in $PList[X, v]$ would not be 0-repairable in any subsequent solution. Moreover, the variables initially in the set *unrep* have no alternative, and therefore cannot be repaired. It follows from Lemma 2 that any variable Y added to *unrep* cannot be repaired as well. Indeed, it means that for all values of a variable X assigning this value would make Y unrepairable and since at least one value need to be assigned, Y is necessarily unrepairable. Therefore, in all solutions involving the assignment $X = v$, all variables in $unrep \cup P$ are unrepairable. Hence if $|unrep \cup P|$ is greater than ub, then all these solutions are sub-optimal.

Complexity: We first look at the procedure `propagate-sup`^{max}. This algorithm performs two tasks, first it explores all tuples in the constraint to find support. As for GAC or *super*-GAC processing, this is done in $\mathcal{O}(d^2)$. However this algorithm also updates the data-structures $PList$ and *unrep*. This may result in at most d insertions for $PList$, and at most d set intersections for *unrep*. Each of these intersections can be computed in $\mathcal{O}(n)$. Moreover, the size of the set $unrep \cup Plist[X_j, w]$ is checked at most d times, and each check can be done in $\mathcal{O}(n)$. Therefore, the total complexity for the procedure `propagate-sup`^{max} thus is $\mathcal{O}(d(n + d))$.

Now we count the potential number of calls to this procedure before reaching a fix point. The idea behind the proof of complexity for `AC3` in [Mackworth 85] is to count how many time an arc $C(X_i, X_j)$ enters the queue, hence how many times the method `propagate` is called. In [Mackworth 85] Mackworth and Freuder observe that to enter the queue, a value in the domain of X_i need be removed in a previous call, therefore each arc can only enter the queue d times. In the case of *super*-`AC`^{max}, an arc $C(X_i, X_j)$ may be added to the queue for three reasons. The first one is when a value is removed from $s\mathcal{D}(X_i)$, and can happen d times. The second is when a value is removed from $r\mathcal{D}(X_i)$, and can happen d times. Finally, the third reason is when the set *unrep* is modified, and can happen n times. Since there are twice as many arcs as constraints (one for each direction) the total number of calls to `propagate-sup`^{max} is in $\mathcal{O}(m(n + d))$. The worst case time complexity of the procedure *super*-`AC`^{max} therefore is $\mathcal{O}(md(n + d)^2)$. □

Theorem 43. *A Branch & Bound algorithm using* super-GAC^{max} *is a sound and complete method for solving* $(1, 0)$-MaxRepairCSP.

Proof. **Soundness:** We show that the solution returned is optimal. Let ub be the optimal value of the objective function as found by a Branch & Bound algorithm using *super*-GAC^{max} as inference method. The algorithm stops when the constraint network $\mathcal{P}$ augmented with $\{\Phi(\mathcal{P}) < ub\}$ is exhausted and proved unsatisfiable. Therefore, $\mathcal{P}$ does not admit a solution whose repairability is less than $\sum_{k \leq a} \binom{n}{k} - ub$.

Completeness: This method is complete since the Branch & Bound search is sound and complete and the procedure *super*-GAC^{max} is sound. □

6.4 Maximising Weak Fault Tolerance

6.4.1 Objective Function

The objective function Φ that we consider here is the (a,b)-repairability (see Definition 10, Section 3.2.3), yielding to the following formula, where f is a solution of the constraint network $\mathcal{P} = \{\mathcal{X}, \mathcal{D}, \mathcal{C}\}$.

$$\Phi(f) = |\{A \mid A \subseteq \mathcal{X} \ \wedge \ |A| \leq a, \exists g \in sol(\mathcal{P}) \ s.t. \ \Delta_A(g,f) = |A| \ \wedge \ \Delta_{\mathcal{X} \setminus A}(f,g) \leq b\}|$$

6.4.2 Inference Method: "*b-repairable*^{max}"

We adapt the inference method used for (a,b)-SuperCSP in much the same way that we adapted the local consistency *super*-GAC to deal with $(1,0)$-MaxRepairCSP. The same difficulties arise, i.e., the pruning inferred with the `repairability-`Γ procedure cannot be used without some modification, since there is some tolerance on the number of unrepairable breakages. However, here the situation is made a little bit easier by the fact that the filtering method introduced for (a,b)-SuperCSP has not a local but a global view. Indeed the algorithm does not process arcs or constraints. Instead, the focus is on breakages, and since each one is checked at most once, we do not need to keep a set of contingently unrepairable breakages, we can keep a simple count on them.

6.4.3 Closure Algorithm for "*b-repairable*^{max}"

The `repairability` procedure is called during each propagation phase to ensure that all the breakages on assigned variables are repairable. In algorithm 19, the breakages are checked in turn, and the algorithm fails as soon as one breakage is found unrepairable. As a filtering method for (a,b)-MaxRepairCSP, we can count the number of breakages and fail only if this number matches the current upper bound. However, we want to take advantage of the reasoning made while preprocessing the Similarity constraint. Similarly to the reasoning made for $(1,0)$-MaxRepairCSP, committing to a value pruned because of an equality constraint inferred in this way would make the breakage currently checked unrepairable. However, as opposed to algorithm 26, this procedure goes through the breakages in turn. We do not thus need to remember which breakage would be made unrepairable by a given assignment, as they are never visited twice in the same pass. We therefore introduce a simple data-structure *Count* that maps an integer $Count[X, v]$ to every value v of each variable X:

$$Count : \mathcal{X} \times \Lambda \mapsto \mathbb{N}$$

6.4.3.1 Commented Pseudo Code

We describe algorithm 28 and compare it to its satisfaction equivalent introduced in Section 5.6 (Algorithm 19). First, the network $\mathcal{P}$ is made generalised arc consistent in Line 1 and we fail if any domain is emptied. Then, the counters *Count* are all initialised to 0 in Line 2, and the total count on the number of unrepairable breakages *unrep* is set to $|S_{unrep}|$ in Line 3.

Then we loop over breakages that satisfy the two following constraints: Their neighbourhood up to distance $b + 1$ is not included into the set of assigned variables; They are not member of S_{unrep}. At each iteration, we create the sub-problem $\mathcal{P}^{\varphi,A}$ and do the following:

- The sub-problem $\mathcal{P}^{\varphi,A}$ is preprocessed in Line 4 using an arc consistency method, and using the algorithm 17 to propagate the Similarity constraint. On failure (line 5), i.e. domain wipe-out, the value of *unrep* is incremented and the index of the current breakage is inserted into S_{unrep}, and we jump to the next breakage.

- The equality constraint between $\mathcal{P}$ and $\mathcal{P}^{\varphi,A}$ are inferred in Line 7, though not directly posted. These equality constraints are computed as described in Section 5.5.2: the set I of variables necessarily taking different values in φ and in a solution of $\mathcal{P}^{\varphi.A}$ is first computed in Line 6. Then using the neighbourhood inference a set of equality constraints may be deduced.

- The sub-problem $\mathcal{P}^{\varphi,A}$ is solved in Line 8. On failure (Line 9), i.e. unsatisfiability, the value of *unrep* is incremented and the index of the current breakage is inserted into S_{unrep}, and we jump to the next breakage.

Algorithm 28 `repairability`^{max}-$\Gamma(\mathcal{P}, \varphi, \mathcal{F}, a, b)$

```
   Data   : $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$, $\varphi$, $ub$
   Result : $\varphi$
 1 if $\neg$GAC($\mathcal{P}' = (\mathcal{F}, \varphi, \mathcal{C})$) then return false;
 2 foreach $X \in \mathcal{F},\ v \in \varphi(X)$ do $Count[X, v] \leftarrow 0$;
 3 $unrep \leftarrow 0$;
   $\mathcal{G} \leftarrow \{X \mid |\varphi(X)| = 1\}$;
   foreach $A \subseteq (\mathcal{X} \setminus \mathcal{F})$ such that $|A| \leq a$ do
       if $\Gamma_{b+1}(A) \not\subseteq \mathcal{G}$ then
 4         if $\neg$preprocess($\mathcal{P}^{\varphi,A}$) then
 5             $unrep \leftarrow unrep + 1$;
           else
 6             $I \leftarrow \{X_i \mid \mathcal{D}^{\varphi,A}(X_i) \cap \varphi(X_i) = \emptyset\}$;
 7             $Eq \leftarrow (\mathcal{X} \setminus \Gamma_{|A|-|I|+b}(I))$;
 8             if $\neg$solve($\mathcal{P}^{\varphi,A}|_{\mathcal{X}\setminus\mathcal{F}}$) then
 9                 $unrep \leftarrow unrep + 1$;
               else
                   foreach $X \in Eq$ do
                       $Vals \leftarrow \mathcal{D}^{\varphi,A}(X) \setminus \varphi(X)$;
                       foreach $v \in Vals$ do
10                         $Count[X, v] \leftarrow Count[X, v] + 1$;
11                         if $unrep + Count[X, v] \leq ub$ then
12                             $\varphi(X) \leftarrow \varphi(X) \setminus \{v\}$;
13                             if $\varphi(X) = \emptyset$ then return false;
   if $unrep \leq ub$ then
       return false;
   else
       return true;
```

Figure 6.12: A filtering algorithm for MaxRepairCSP.

- If the equality constraints were to be applied, the main partial solution φ could be pruned. Any value $v \in \varphi(X)$ that would be removed by such equality constraint has its counter $Count[X, v]$ incremented in Line 10. Whenever, for a variable X and a value v, we have $T - (Count[X, v] + unrep) \leq ub$ (if statement in Line 11) then we prune $X = v$ in Line 12, and fail if $\varphi(X)$ is emptied (Line 13).

6.4.4 Theoretical Properties

Theorem 44. `repairability`^{max}-Γ *is a sound inference method and runs in* $\mathcal{O}(l^{a+b}d^{a+b})$.

Proof. **Soundness:** We prove that the pruning achieved with `repairability`^{max}-Γ is sound. The algorithm first achieves the GAC closure. This is sound in this context since only the robustness condition is relaxed. We therefore only need to prove that the pruning performed in Line 12 is sound. A value $v \in \varphi(X)$ is pruned when the condition in Line 11 is triggered. We show that if it is the case, then any sub-solution involving the assignment $X = v$ would be sub-optimal. Without loss of generality, let v be a value in $s\mathcal{D}(X)$, and suppose that the condition in Line 11 holds. We consider the case where $\varphi(X_j) = \{v\}$ and count unrepairable breakages. All breakages in S_{unrep} are not repairable since the corresponding sub-problem has been proven unsatisfiable in an earlier inference step. Moreover, consider a breakage A for which $Count[X, v]$ has been incremented. Since the satisfaction version of the algorithm (Algorithm 19) is sound, and since this algorithm would prune this value when checking A, this value cannot participate in a solution where A is b-repairable. Since breakages in S_{unrep} are not checked for repairability, any solution involving $X = v$ will have at least $|S_{unrep}| + Count[X, v]$ unrepairable breakages.

Complexity: The time complexity is unchanged with respect to the `repairability-`Γ algorithm as shown in Figure 5.13 □

Theorem 45. *A Branch & Bound algorithm using* `repairability`^{max}-Γ *is a sound and complete method for solving* $(1, 0)$-MaxRepairCSP.

Proof. **Soundness:** We show that the solution returned is optimal. Let ub be the optimal value of the objective function as found by a Branch & Bound algorithm using `repairability`^{max}-Γ as inference method. The algorithm stops when the constraint network $\mathcal{P}$ augmented with $\{\Phi(\mathcal{P}) < ub\}$ is exhausted and proved unsatisfiable. Therefore, $\mathcal{P}$ does no admit a solution whose repairability is less than $\sum_{k \leq a} \binom{n}{k} - ub$.

Completeness: This method is complete since the Branch & Bound search does not hinder completeness, and the procedure `repairability`^{max}-Γ is sound. $\square$

6.5 Summary and Limitations

In this chapter we addressed two "partial" problems related to *super*-solution. In the first problem, MinBCSP, we relax the value of the parameter b, and search for an (a, b)-*super*-solution with minimal b. In the second problem, MaxRepairCSP, we relax the condition that all breakages need be repaired, and search for a solution with maximal (a, b)-repairability.

In both cases, we showed that the methods introduced in previous chapters can be extended to solve these partial problems. However, whereas the Branch & Bound algorithms introduced to solve the latter problem (MaxRepairCSP) retain some good properties of the satisfaction version, this is not the case for MinBCSP. Indeed, in this case, since the value of the parameter b is not bounded, therefore the computational complexity of the inference method (`repairability-`Γ) is exponential. This is not surprising since we showed that even computing the minimum value of b for which a given solution is an (a, b)-*super*-solution is NP-hard.

Our study of partial *super*-satisfiability is in no way exhaustive. This analysis is focused on the above-mentioned two ways of relaxing the robustness condition. Another, unexplored though similar, direction would be to maximise the value of a for which there exists an (a, b)-*super*-solution. Alternatively, we could interleave the classical soft constraint / partial CSP framework for robustness. For instance we could investigate the problem of finding *super*-solutions such that repairs are not solutions but violate only a few constraints, or a minimal number of constraints. Moreover, as was the case in Chapter 4, the algorithms we introduce for finding solutions with maximal $(1, 0)$-repairability are restricted to binary constraint networks. We did not extend the reformulation methods to the partial *super*-satisfiability context. Notice, in particular, that the $\mathcal{P} + \mathcal{P}$ reformulation could be easily adapted, since we would simply need to maximise the number of satisfied disequality constraints in the reformulated network.

A last interesting and unexplored research direction is the use of global constraints for maximising repairability. As regular global constraints can be extended to soft global constraints, the propagations algorithms discussed in Chapter 4 could be adapted to this optimisation setting.

Chapter 7

Extensions to the Framework

7.1 Introduction

In this chapter, we extend the concept of *super*-solution. We have argued that *super*-solutions require little or even no knowledge of the environment. In fact, following the definition of (a, b)-*super*-solutions in Section 3.2, the only parameters that may need to be set before solving are the values of a and b. The value of a, i.e., the cardinality of a breakage, may be adapted according to the likeliness of multiple variables breaking simultaneously. Similarly, the value of b may change according to what is considered an acceptable repair in the problem we are solving. However in some cases, we may want to take advantage of some properties of the problem, and we might want to express some related though different notions of robustness or stability. In this chapter we introduce the notion of robustness model. The model tackled in previous chapters, that is, existential-(a, b)-*super*-solutions, is only one among many possibilities. In this robustness model, a breakage corresponds to "loosing" the values currently assigned to a set of variables whilst a valid repair is a set of further reassignments of bounded cardinality. We significantly extend the range of robustness model expressible and solvable using the algorithms introduced in Chapter 5 and 6. We also tackle, in this chapter, the notion of symmetry and symmetry breaking while searching *super*-solutions.

We motivate the need for a richer modelling language and show that our algorithm only needs a slight adaptation to be able to solve the resulting extended problems in Section 7.2. Finally we study the concept of symmetry and symmetry breaking within the fault tolerance framework in Section 7.3.

7.2 Extended Modelling

In practice, we may have restrictions on how the problem is likely to break, or how we may repair it, and we may want to take into account some facts we know about the environment which are not properly handled by the regular definition. We give some examples of such restrictions, partitioned into 3 classes:

Breakage and repair set restrictions: The first class contains the restrictions on the breakage and repair sets. We may want to decide, for every variable whether it is prone to change, and whether it can be reassigned when repairing.

For instance, a job shop problem may involve some resources that are reliable and other that are not. If a variable stands for a machine, then we can limit breaks to a subset of the variables. In the mixed constraint satisfaction and stochastic constraint satisfaction frameworks ([Fargier 96], [Manandhar 03]), the variables are either controllable, if they are regular decision variables, or uncontrollable, if they correspond to properties of the environment. In such a case, the breakages could be restricted to uncontrollable variables and repairs to controllable variables. Similarly, whilst some variables may be reassigned, others may have to take the value they are originally assigned. For instance consider the trip planning problem. We might want to limit the size of the repairs, however we may want to make sure that a repair does not involve changing something expensive and ordered prior to the trip, such as a plane ticket, because of a perturbation on a train or bus transit.

Breakage restrictions: The second class restricts how the alternative values for a variable involved in a breakage can be chosen. Breakages may need to be more complex restrictions than a single value removal.

For example, when the values represent time, an alternative value might have to be larger than the broken value. In that case if the value v for a variable X "breaks", then any value w such that $w < v$ should not be available for reassigning X. Another example is when several values in a domain refer to the same real object. For instance, suppose that we have to fit k items in containers, and that each item can either be put in width ways or length ways. We have one variable for each container, and $2k$ values in each domain. Assigning the value $2v$ to container X means that we put item k in width ways, whereas assigning $2v - 1$ means that we put item k in length ways. If an item becomes, for some reason, unfit for a given container, then the pair of values corresponding to this single item cannot repair each other. Moreover, it often happens that certain values are simply not brittle and so cannot break. For instance if a value represents a resource that cannot be depleted.

Repair restrictions: The third class restricts how the variables outside a breakage can be reassigned to repair this breakage. The notion of "acceptable repair" may vary, and is not always as simple as a bound on the number of discrepancies with the solution being repaired.

For instance, consider again the situation where values represent time points. We might want to ensure that variables assigned to values less than v should not be reassigned in order to repair a breakage involving the value v. Indeed, values less than v represent events that already occurred at the time the breakage occurs. Furthermore, and for the same reason, all repair values should have larger values, after the repair, than v. Alternatively consider a situation where you want to allocate tasks to a set of contractors, with some extra precedence and non-overlapping constraints. In such a scenario, a valuable property to have is that a change in the schedule for one contractor does not have any consequences on the other contractors, but the number of rescheduled tasks for the faulty contractor may not matter. Therefore the repairs should be limited in the number of contractors affected by the changes rather than the cardinality of the changes. Third, in the dual viewpoint, we may have variables for time points and values for tasks. In this case we may want to ensure that all b repair variables are later in some ordering than the smallest of the a broken variables.

We propose a framework to deal with these classes of restrictions and show that the algorithm `decomposition-backtrack` can be very easily adapted to deal with this more expressive framework. Notice that since the algorithms for MinBCSP and MaxRepairCSP are extensions of `decomposition-backtrack`, the same restrictions are easily adaptable to these cases as well. We then briefly show in Section 7.2.4 how the same extensions can or cannot be handled by other algorithms such that `MAC+` and *super*-`AC`. We recall the definition of a sub-problem $\mathcal{P}^{\varphi,A}$ created for checking the repairability of a breakage A. This definition is then extended in the subsequent sections to model the notions of **breakage and repair set restrictions**, **breakage restric-**

tions and **repair restrictions**.

	$\mathcal{P}$	$\mathcal{P}^{\varphi,A}$		
Variables:	$\mathcal{X}$	$\mathcal{X}^{\varphi,A}$	$=$	$\{X^{\varphi,A} \mid X \in \mathcal{X}\}$
Domains:	$\mathcal{D}$	$\mathcal{D}^{\varphi,A}(X^{\varphi,A})$	$=$	$\varphi(X)$ if $X \notin \Gamma_b(A)$ $\mathcal{D}(X)$ otherwise
Constraints:	$\mathcal{C}$	$\mathcal{C}^{\varphi,A}$	$=$	$\mathcal{C} \cup \{\forall X \in A\ (X \neq X^{\varphi,A})\}$ $\cup\{\textsc{Similarity}(\Gamma_b(A), \varphi, A, b)\}$

Table 7.1: The sub-problem $\mathcal{P}^{\varphi,A}$ (Reminder).

7.2.1 Breakage, Repair and Free Sets

In the classical definition, the breakages are simply all subsets of at most a variables. However, notice that the algorithm `decomposition-backtrack` can soundly handle an arbitrary set of breakages. Restricting the number of variables that can be reassigned in a repair is also easy. Any variable which cannot be part of a repair corresponds to an equality constraint on the occurrence of this variable in the master-problem and its homologue in all sub-problems. This concept of equality constraint and how one can handle it in the search process is explained in Chapter 5. Therefore, in all generality, one can provide any list, defined either extensionally or intentionally of breakages. However, we introduce a description using three sets, one for the variables that can possibly break, one for the variable that can possibly be used for repair, and a third set for the variables that can be reassigned freely, i.e., without any penalty.

Breakage Set: The breakage set $\mathcal{BS}$ contains all variables that can be involved in a breakage. The set of breakage to consider therefore is: $\{A \mid A \in \mathcal{BS} \ \wedge \ |A| \leq a\}$

Repair Set: The repair set $\mathcal{RS}$ contains all variables that can be used for repair. Therefore the set of equality constraints Eq is always a superset of $\mathcal{X} \setminus \mathcal{RS}$.

Free Set: The free set $\mathcal{FS}$ contains all variables which reassignment does not matter. Therefore the Similarity constraint never constrains any variable in $\mathcal{FS}$.

The breakage set $\mathcal{BS}$ does not affect the construction of sub-problems, however it dictates how many such sub-problems need be solved. We adapt the definition of a sub-problem $\mathcal{P}^{\varphi,A}$, taking repair sets and free sets ($\mathcal{RS}$ and $\mathcal{FS}$) into account:

The equality constraints on $\mathcal{X} \setminus \mathcal{RS}$ are taken into account when setting the domains of the corresponding variables. Only the domains of variable that can be

	$\mathcal{P}$	$\mathcal{P}^{\varphi,A}$		
Variables:	$\mathcal{X}$	$\mathcal{X}^{\varphi,A}$	$=$	$\{X^{\varphi,A} \mid X \in \mathcal{X}\}$
Domains:	$\mathcal{D}$	$\mathcal{D}^{\varphi,A}(X^{\varphi,A})$	$=$	$\varphi(X)$ if $X \notin (\Gamma_b(A) \cap \mathcal{RS})$ $\mathcal{D}(X)$ otherwise
Constraints:	$\mathcal{C}$	$\mathcal{C}^{\varphi,A}$	$=$	$\mathcal{C} \cup\ \forall X \in A\ (X \neq X^{\varphi,A})$ $\cup\{\textsc{Similarity}(\Gamma_b(A) \setminus \mathcal{FS}, \varphi, A, b)\}$

Table 7.2: The sub-problem $\mathcal{P}^{\varphi,A}$ (Breakage and repair set restriction).

repaired are set to the full original domain $\mathcal{D}$, otherwise they are set to the current partial solution φ. Notice that the equality constraint may also be taken into account in the opposite direction. Furthermore, the variables in the free set $\mathcal{FS}$ are not constrained by the Similarity constraint.

7.2.2 Constraints on the Breakages

According to Definition 8, the alternative for a breakage is any **different** value. This is enforced in a sub-problem $\mathcal{P}^{\varphi,A}$ by the following set of constraints:

$$\forall X \in A\ (X \neq X^{\varphi,A})$$

However, the algorithm `decomposition-backtrack` can handle other types of constraints to control the alternative values of a breakage. For instance consider the situation mentioned in introduction where values represent time points. This set of constraints can be replaced with:

$$\forall X \in A\ (X + d \leq X^{\varphi,A})$$

The value of d is the expected maximal length of a delay. In fact, any relation on (subsets of) $A \cup X^{\varphi,A} \mid X \in A$ can be given instead of the simple pairwise disequality constraint. Observe that as long as this relation is more restrictive than the set of pairwise disequalities, then all the inference rules discussed in chapter 5 and 6 can still be used soundly. Moreover, if the new constraint is more restrictive then checking breakages containing non assigned variables can lead to better filtering. We have seen in Chapter 5 that only breakages on variables that are assigned in the main partial solution are checked because no extra pruning can be obtained. However this is in fact a property of the disequality constraints used to model a breakage. If C_{break} is stronger than the set of disequalities, then it is possible to achieve extra inference by creating sub-problems and preprocessing them on "future" breakages.

Theorem 46. *If $X \in A$ then $sol(\mathcal{P}^{\varphi,A\setminus\{X\}}) \subseteq sol(\mathcal{P}^{\varphi,A})$.*

Proof. Let f be a solution of $\mathcal{P}^{\varphi,A\setminus\{X\}}$. We show that f satisfies all constraints in $\mathcal{P}^{\varphi,A}$.

- **Constraints in $\mathcal{P}$:** Clearly, the constraints of the original constraint network are satisfied.

- **Equality constraints:** In $\mathcal{P}^{\varphi,A}$ we impose a equality constraint on X, i.e., set $\mathcal{D}^{\varphi,A}(X)$ to $\varphi(X)$ if and only if $X \notin \Gamma_b(A)$. However, the set $\Gamma_b(A \setminus \{X\})$ is a supper set, hence there are at least as many equality constraints in $\mathcal{P}^{\varphi,A\setminus\{X\}}$.

- **Similarity constraint:** The Similarity constraint in $\mathcal{P}^{\varphi,A\setminus\{X\}}$ constrains a smaller set of variable ($\Gamma_b(A \setminus \{X\})$ instead of $\Gamma_b(A)$) to have a smaller discrepancy with φ ($|A| + b - 1$ instead of $|A| + b$), therefore if f satisfies the former, it also satisfies the latter.

- **Disequality constraints:** The constraint $X \neq X^{\varphi,A}$ is the only disequality constraint in $\mathcal{P}^{\varphi,A}$ that is not in $\mathcal{P}^{\varphi,A\setminus\{X\}}$. However, since $\varphi(X)$ is not a singleton, this disequality is GAC for any value of $f(X)$.

□

Notice also that the constraints controlling the breakages can be modified for efficiency reasons. Indeed, the original constraints within the set A can sometimes combine with these breakage constraints into a global constraint for which a stronger propagation is possible. For instance, if the constraint network $\mathcal{P}$ contain the constraint AllDifferent($\mathcal{X}$), then the constraint controlling a breakage A could be AllDifferent($A \cup X^{\varphi,A} \mid X \in A$). Therefore we shall denote the constraint controlling how a breakage A can be given alternative values $C_{break}(A)$ and change the definition of $\mathcal{P}^{\varphi,A}$ as follows:

We denote C^0_{break} the classical constraint for controlling breakages, i.e., $\forall X \in A\ (X \neq X^{\varphi,A})$. We say that a constraint $C_1(V)$ is tighter than another constraint $C_2(V)$ if and only if $C_1 \subseteq C_2$.

	$\mathcal{P}$	$\mathcal{P}^{\varphi,A}$		
Variables:	$\mathcal{X}$	$\mathcal{X}^{\varphi,A}$	$=$	$\{X^{\varphi,A} \mid X \in \mathcal{X}\}$
Domains:	$\mathcal{D}$	$\mathcal{D}^{\varphi,A}(X^{\varphi,A})$	$=$	$\varphi(X)$ if $X \notin (\Gamma_b(A) \cap \mathcal{RS})$ $\mathcal{D}(X)$ otherwise
Constraints:	$\mathcal{C}$	$\mathcal{C}^{\varphi,A}$	$=$	$\mathcal{C} \cup \{C_{break}(A)\}$ $\cup\{\textsc{Similarity}(\Gamma_b(A) \setminus \mathcal{FS}, \varphi, A, b)\}$

Table 7.3: The sub-problem $\mathcal{P}^{\varphi,A}$ (Breakage constraint).

Theorem 47. *If C_{break} is tighter than C^0_{break} then the procedure* `repairability` *using C_{break} instead of C^0_{break} is a sound inference method for the (a,b)-*SuperCSP *problem.*

Proof. Let C_{break} be a constraint strictly tighter than C^0_{break} (i.e., $C_{break} \subseteq C^0_{break}$). We show that Theorem 30 applies in this broader context. Since C_{break} is tighter than the classical set of disequality constraints, the premise of Theorem 30 holds. Now we have to make sure that this new solution satisfies the constraint C_{break}. In the construction used to prove Theorem 29 the third solution h is in fact equivalent to f on A. Therefore the existence of a solution g such that $\Delta_A(f,g) \leq |A| + b$ ensures not only that there exists another repair h with all discrepancies within $\Gamma_b(A)$, but also that this repair has the same image than g on A. Since we suppose that g satisfies C_{break}, so does h, hence the inference method `repairability` can be used. □

7.2.3 Constraints on the Repairs

Similarly, repairs are controlled through a constraint, namely Similarity. Here again, this constraint can be changed and the algorithm `decomposition-backtrack` can still be used. We change the definition of the sub-problems accordingly:

	$\mathcal{P}$	$\mathcal{P}^{\varphi,A}$		
Variables:	$\mathcal{X}$	$\mathcal{X}^{\varphi,A}$	$=$	$\{X^{\varphi,A} \mid X \in \mathcal{X}\}$
Domains:	$\mathcal{D}$	$\mathcal{D}^{\varphi,A}(X^{\varphi,A})$	$=$	$\varphi(X)$ if $X \notin (\Gamma_b(A) \cap \mathcal{RS})$ $\mathcal{D}(X)$ otherwise
Constraints:	$\mathcal{C}$	$\mathcal{C}^{\varphi,A}$	$=$	$\mathcal{C} \cup \{C_{break}(A), C_{repair}(\Gamma_b(A) \setminus \mathcal{FS})\}$

Table 7.4: The sub-problem $\mathcal{P}^{\varphi,A}$ (Repair constraint).

In the previous section, we have seen that the inference method based on equality constraints can be used if the constraint controlling the breakages is tighter than the disequality constraint. This is not the case for the constraint controlling the repairs.

Example 27. *For instance consider the following constraint network:*

$$X \geq Y \leq Z,\ \mathcal{D}(X) = \mathcal{D}(Y) = \mathcal{D}(Z) = \{1, 2\}$$

And now suppose that we want the repairs to be classical 1*-repairs,* ***and*** *we insist that the sum of all variables should have the same parity in a repair and in the main solution:*

$$C_{repair}(V) = \textsc{Similarity}(V, \varphi, A, 1)\ \wedge\ (\textstyle\sum_{X\in\mathcal{X}} X^{\varphi,A}\ mod\ 2) = (\sum_{X\in\mathcal{X}} X\ mod\ 2)$$

The tuple $\langle 1, 1, 1\rangle$ for X, Y, Z is a solution, and $\langle 2, 1, 2\rangle$ is a valid repair for a the breakage $\{X\}$ since only one variable is reassigned beside X and the sum is odd. However although $\langle 2, 1, 1\rangle$ is a valid 1*-repair in the classical sense, it is not here since it violates the parity constraint. Moreover, the tuple $\langle 2, 2, 1\rangle$ is not consistent. Therefore this is a counter example of Theorem 30 in the case of a more restrictive constraint.*

One may therefore consider the possibility of using the inference method introduced in Chapter 5 for each "repair control" constraint case by case. We show, for example, that the constraint we are using in the experimental section for the job-shop scheduling problem can benefit from this reasoning. The constraint we are using is defined as follows:

$$\begin{gathered}\textsc{Similarity-JSP}(X_1, \dots X_n, \varphi, A, b) = \\ \textsc{Similarity}(X_1, \dots X_n, \varphi, A, b)\ \wedge\ \forall i \in [1..n],\ (X_i = X_i^{\varphi,A} \vee X_i^{\varphi,A} \geq min(A))\end{gathered}$$

Theorem 48. *The procedure* `repairability` *using* Similarity-JSP *instead of* Similarity *is a sound inference method for the (a, b)-*SuperJSP *problem.*

Proof. The constraint Similarity-JSP ensures that whenever a variable is reassigned in a *repair*, it is reassigned to a value greater than or equal to the value of the least broken value. Now we show that the proof of Theorem 29 is still valid with this restriction. Consider a solution f and a *repair*, g, valid for Similarity-JSP. A solution h such that all discrepancies are restricted to $\Gamma_b(A)$ can be constructed in the same way, and is a valid solution by Lemma 1. Moreover, in this construction, all variables are either assigned as in f or as in g. Therefore, h satisfies the extra constraint $\forall i \in [1..n],\ (X_i = X_i^{\varphi,A} \vee X_i^{\varphi,A} \geq min(A))$. □

7.2.4 Solving Extended Problems

We showed that the algorithm introduced in Chapter 5 can be extended to accommodate more complex definitions of *super*-solutions. We give the pseudo-code for an algorithm taking into account all these extensions to the framework. This algorithm takes as input:

- **A constraint network** $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$.
- **A set** $\mathcal{BS} \subseteq \mathcal{X}$ standing for the breakage set.
- **A integer** a standing for the maximum size of a breakage.
- **A set** $\mathcal{RS} \subseteq \mathcal{X}$ standing for the repair set.
- **A set** $\mathcal{FS} \subseteq \mathcal{X}$ standing for the free set.
- **A constraint** C_{break} for controlling breakages.
- **A constraint** C_{repair} for controlling repairs.

The main backtracking procedure is basically unchanged, apart from the extra parameters explained above. The filtering procedure, on the other hand, is slightly changed to handle this more expressive framework. The construction of the sub-problems is detailed in Algorithm from Line 1 to Line 2.

7.3 Symmetry

Many real world problems contain symmetry (see [Crawford 96, Emerson 93, Flener 02, Fox 99] for examples of problems exhibiting symmetries). In a job shop scheduling problem, for instance, some of the machines or some of the jobs may be identical. Hence, swapping the corresponding assignments does not affect satisfiability. When a problem involves symmetries, this fact can be exploited to dramatically reduce the search effort. Indeed we can avoid exploring certain branches, if symmetrically equivalent branches have been or will subsequently be explored. This reasoning on symmetrically equivalent objects in the context of search algorithm has first been exploited in [Brown 88]. Since then, several methods have been developed to effectively take advantage of symmetry in constraint programming. Symmetry Breaking During Search

Algorithm 29 `dec-backtrack-ext`

```
Data   : $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$, $\varphi$, $\mathcal{F}[= \mathcal{X}]$, $\mathcal{BS}$, $a$, $\mathcal{RS}$, $\mathcal{FS}$, $C_{break}$, $C_{repair}$
Result : Does $\mathcal{P}$ admit a super-solution
if $\mathcal{F} = \emptyset$ then return true;
choose $X \in \mathcal{F}$;
foreach $v \in \varphi(X)$ do
    $\varphi(X) \leftarrow \{v\}$;
    if GAC($\mathcal{P}' = (\mathcal{F}, \varphi, \mathcal{C})$) then
        if repairability-ext($\mathcal{P}, \varphi, \mathcal{F} \setminus \{X\}, \mathcal{BS}, a, \mathcal{RS}, \mathcal{FS}, C_{break}, C_{repair}$) then
            if dec-backtrack-ext($\mathcal{P}, \varphi, \mathcal{F} \setminus \{X\}, a, \mathcal{RS}, \mathcal{FS}, C_{break}, C_{repair}$) then
                return true;
    restore $\varphi$;
return false;
```

Algorithm 30 `repairability-ext`

```
  Data   : $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$, $\varphi$, $\mathcal{F}[= \mathcal{X}]$, $a$, $b$
  Result : Check the repairability of grounded breakages
  foreach $A \subseteq \mathcal{BS}$ such that $|A| \leq a$ do
1     foreach $X \in \mathcal{X}$ do
          if $X \notin (\Gamma_b(A) \cap \mathcal{RS})$ then
              $\mathcal{D}^{\varphi,A}(X^{\varphi,A}) \leftarrow \varphi(X)$;
          else
              $\mathcal{D}^{\varphi,A}(X^{\varphi,A}) \leftarrow \mathcal{D}(X)$;
2     $\mathcal{C}^{\varphi,A} \leftarrow \mathcal{C} \cup \{C_{break}(A), C_{repair}(\Gamma_b(A) \setminus \mathcal{FS})\}$;
      if $\neg$solve($\mathcal{P}^{\varphi,A}|_{\mathcal{X} \setminus \mathcal{F}}$) then return false;
  return true;
```

Figure 7.1: Extended backtracking Algorithm for finding (a, b)-*super*-solutions.

(SBDS) [Backofen 99, Gent 02], Symmetry Breaking by Dominance Detection (SBDD) [Fable 01, Focacci 01], and Symmetry Breaking Constraints [Kiziltan 04, Puget 93] are all examples of such symmetry breaking methods.

A symmetry is a property of a mathematical object to remain invariant under some transformations. In [Cohen 05], Cohen *et al.* review several definitions of symmetry for constraint satisfaction problems. They distinguish two main categories, a symmetry may be defined either on (partial) solutions or on constraints. In this section, we shall define the notion of symmetry as a bijective mapping on solutions. We do not cover alternative definitions of symmetries as it is not the primary concern of this dissertation. Our aim is to study the concept of symmetry within the *super*-solution framework. In particular we are interested in understanding when and why symmetry breaking tools can be used when searching *super*-solutions. Indeed, if such methods cannot be used, then it is very unlikely that *super*-solutions might ever be found for certain large and highly symmetric problems. We first give a very general definition of symmetry over solutions, and show that the image of a *super*-solution by such a symmetry is not always a *super*-solution. As a result, there exist symmetries for which we cannot use the usual symmetry breaking methods. However, we give a sufficient condition for guaranteeing that the symmetric image of a *super*-solution is itself a *super*-solution. The class of symmetries satisfying this condition is in fact broad and contains most of the symmetries that can be handled by symmetry breaking tools.

7.3.1 Symmetry and *super*-solutions

A constraint network $\mathcal{P}$ is symmetric if there exists a bijection γ, different from the identity relation, such that the set of solutions of $\mathcal{P}$ remains unchanged under γ. However, within this definition, it may happen that the symmetric image of a (a,b)-*super*-solution is not itself a (a,b)-*super*-solution. We now formally define the concept of symmetry of a constraint network, and then give an example of symmetry that does not preserve repairability.

An automorphism is a bijective mapping from a set onto itself. Let $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ be a constraint network. We assume that the domains are uniform, that is, the domain $\mathcal{D}(X)$ of any variable is equal to a common set Λ.

Definition 28. *A symmetry γ of a constraint network $\mathcal{P}$ is an automorphism on $sol(\mathcal{P})$.*

Example 28. *Consider a the following constraint network:*

$$\mathcal{P}: \quad X + Y + Z \leq 4,\ \mathcal{D}(X) = \mathcal{D}(Y) = \mathcal{D}(Z) = \{1, 2\}$$

We list the set $sol(\mathcal{P})$ of solutions of this constraint network, and define a bijective mapping γ from $sol(\mathcal{P})$ onto itself, i.e., a symmetry of $\mathcal{P}$, in Figure 7.2:

$$\begin{array}{lcrcl}
\langle 1,1,1\rangle & \qquad & \gamma : \langle 1,1,1\rangle & \rightarrow & \langle 2,1,1\rangle \\
\langle 2,1,1\rangle & & \gamma : \langle 2,1,1\rangle & \rightarrow & \langle 1,2,1\rangle \\
\langle 1,2,1\rangle & & \gamma : \langle 1,2,1\rangle & \rightarrow & \langle 1,1,2\rangle \\
\langle 1,1,2\rangle & & \gamma : \langle 1,1,2\rangle & \rightarrow & \langle 1,1,1\rangle
\end{array}$$

Figure 7.2: The set of solutions *sol*($\mathcal{P}$) and a symmetry mapping γ on this set.

Whereas $\langle 1,1,1\rangle$ is a $(1,0)$-super-solution, its symmetric image $\langle 2,1,1\rangle$ is not. If we somehow break this symmetry we might loose the tuple $\langle 1,1,1\rangle$, i.e., the only one $(1,0)$-super-solution for this constraint network.

We have seen in this example that not all symmetries preserve repairability, that is, the symmetric image of an (a,b)-*super*-solution is not necessarily an (a,b)-*super*-solution. However, in practice, symmetries are often also defined on partial assignments. Such symmetries are typically **distributive**, that is, given two disjoint assignments, the image of their composition is equal to the composition of their images. Although not all symmetries defined on partial assignments have this property, this is a very common characteristic in practice. We define formally this notion of distributivity, and subsequently show that this is a sufficient condition to ensure that repairability is preserved.

We defined the union operation on assignments in Section 2.3. Since the image of an assignment by a symmetry γ is itself an assignment, the union operation can still be used with the same semantics.

Definition 29. *A symmetry γ is distributive if and only if, for any pair of assignments f, g, defined respectively on $F \in \mathcal{X}$ and $G \in \mathcal{X}$, such that $G \cap F = \emptyset$, the following identity holds:*

$$\gamma(f) \cup \gamma(g) = \gamma((f \cup g))$$

Notice that, as a direct consequence of this definition, a distributive symmetry can be defined as a mapping on unary assignments. Indeed for any assignment f defined on a set of variables A, if γ is a distributive symmetry, then:

$$\gamma(f) = \bigcup_{X \in A} \gamma(f|_X)$$

From now on, we therefore consider a distributive symmetry as an automorphism on $\mathcal{X} \times \Lambda$:

$$\gamma : \mathcal{X} \times \Lambda \mapsto \mathcal{X} \times \Lambda$$

We shall use the notation $\gamma(f|_X)$ to denote the symmetric image by γ of the unary assignment $X = f(X)$. Moreover, for a given set of variables A, $\gamma(f|_A)$ denotes $\bigcup_{X \in A} \gamma(f|_X)$. Notice that it follows directly from Definition 29 that the symmetric image $\gamma(f|_A)$ has the same cardinality as $f|_A$, that is, $|A|$.

To help proving that distributivity is a sufficient condition for guaranteeing that the repairability of a solution is preserved through a symmetry, we first define the projection of a symmetry on variables with respect to a solution, and then prove an intermediate lemma. Given a solution f, and a distributive symmetry γ, we define the projection of γ on variables, denoted θ_f, as the function mapping a variable X to the variable on which $\gamma(f|_X)$ is defined. In other words, if f maps X to v and γ maps $f|_X$ to $\gamma(f|_X) : Y \rightarrow w$, then θ_f maps X to Y.

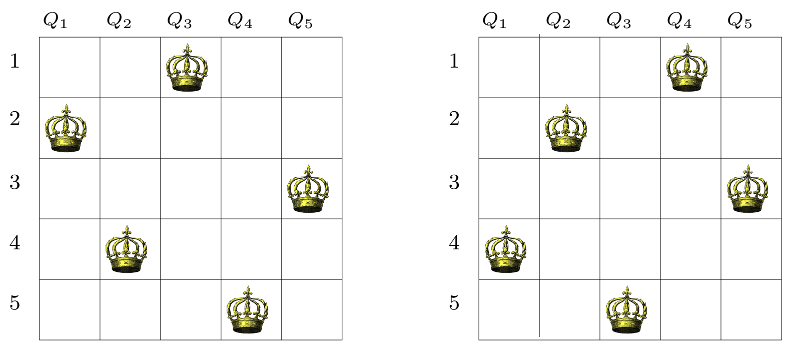
Definition 30. *Given a distributive symmetry γ and a solution f of a constraint network $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$, the projection of γ on variables with respect to f, is the function $\theta_f : \mathcal{X} \mapsto \mathcal{X}$ such that $\theta_f(X) = Y$ if and only if γ maps the assignment $X = f(X)$ to the assignment $Y = v$ where v can be any value in $\mathcal{D}(Y)$.*

Given a distributive symmetry γ and a set of variable $A \subseteq \mathcal{X}$, $\theta_f(A)$ is thus the set of variables on which γ maps the restriction of f to A.

Example 29. *The N-Queens problem is to put N Queens on a chessboard so that no two queens can attack each other. This is usually modelled with N integer variables standing for the row occupied by each queen. A solution and its symmetric by a $90°$ rotational symmetry are shown in Figure 7.3.*

We can define the $90°$ rotational symmetry as follows:

$$\gamma : \langle X_i : j \rangle \rightarrow \langle X_j : N - i \rangle$$



(a) A solution f

(b) The symmetric of f by a rotation of 90°

Figure 7.3: A symmetry of the 5-Queens problem.

Observe that γ is distributive. Indeed it is defined on unary assignments and the image of larger assignments can be obtained by composition. For instance, The image of the solution $f = \langle Q_1 : 2, Q_2 : 4, Q_3 : 1, Q_4 : 5, Q_5 : 3\rangle$ by the symmetry γ is $\gamma(f) = \langle Q_1 : 3\rangle \cup \langle Q_2 : 5\rangle \cup \langle Q_3 : 1\rangle \cup \langle Q_4 : 4\rangle \cup \langle Q_5 : 2\rangle$, that is, $\gamma(f) = \langle Q_1 : 3, Q_2 : 5, Q_3 : 1, Q_4 : 4, Q_5 : 2\rangle$. Given the solution f, the projection θ_f of γ on variables is the function mapping a variable whose image by f is v to the variable X_{N-v}, that is:

$$\theta_f : X_i \to X_{N-f(X_i)}$$

For instance, the value of $\theta_f(\{X_1, X_2, X_3\})$ is $\{X_1, X_2, X_4\}$.

Theorem 49. *Given a solution f of a constraint network $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ and a distributive symmetry γ, then the projection $\theta_{\mathcal{X}}$ of γ on $\mathcal{X}$ with respect to f is an automorphism on $\mathcal{X}$.*

Proof. Suppose that $\theta_{\mathcal{X}}$ is not injective. Then there exist two variables X, Y such that $\theta_{\mathcal{X}}(X) = \theta_{\mathcal{X}}(Y)$. Hence the image to which γ maps $f|_{X,Y}$ is not an assignment, which contradicts the hypothesis.

Suppose that $\theta_{\mathcal{X}}$ is not surjective. Then there exists a variable X with no antecedent by $\theta_{\mathcal{X}}$, hence γ does not map f to a full solution, which contradicts the hypothesis. $\square$

Lemma 3. *Given a constraint network $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$, a distributive symmetry γ, two solutions f, g of $\mathcal{P}$ and a set $A \subseteq \mathcal{X}$, the Hamming distance between f and g on A is* *equal to the Hamming distance between $\gamma(f)$ and $\gamma(g)$ on $\theta_f(A)$.*

$$\Delta_A(f,g) = \Delta_{\theta_f(A)}(\gamma(f), \gamma(g))$$

Proof. Let f, g be two solutions of a constraint network $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$, γ a distributive symmetry and A a subset of $\mathcal{X}$.

Consider any variable $X \in A$, and suppose that $f(X) = g(X)$. Then $f|_X = g|_X$, hence $\gamma(f|_X) = \gamma(g|_X)$. By definition of θ_f, $\gamma(f|_X)$ and $\gamma(g|_X)$ are thus both defined on $\theta_f(X)$. Therefore, for each variable X in A such that $f(X)$ and $g(X)$ are equal, so are $\gamma(f)(\theta_f(X))$ and $\gamma(g)(\theta_f(X))$ on the variable $\theta_f(X)$. Since $\theta_f(X)$ is included in $\theta_f(A)$, the following inequality holds:

$$\Delta_A(f,g) \geq \Delta_{\theta_f(A)}(\gamma(f), \gamma(g)) \tag{7.1}$$

On the other hand since γ is bijective, it is invertible. We consider the distributive symmetry $\gamma' = \gamma^{-1}$, two solutions $f' = \gamma(f), g' = \gamma(g)$ and the set $A' = \theta_f(A)$. We can make exactly the same reasoning, hence we have the following inequality:

$$\Delta_{A'}(f', g') \geq \Delta_{\theta_{f'}(A')}(\gamma^{-1}(f'), \gamma^{-1}(g'))$$

We go through the terms of the right hand side of the inequality:

- $\theta_{f'}(A')$: This is the set of variables on which is defined the image of $\gamma(f)|_{\theta_f(A)}$ by γ^{-1}. The function $\theta_{f'}$ is the projection of γ^{-1} on variables with respect to $\gamma(f)$. Therefore it is in fact the inverse of θ_f, hence it maps $\theta_f(A)$ to A.
- $\gamma^{-1}(f')$: This is in fact $\gamma^{-1}(\gamma(f))$, hence it is equal to f.
- $\gamma^{-1}(g')$: This is in fact $\gamma^{-1}(\gamma(g))$, hence it is equal to g.

Therefore, the dual inequality holds:

$$\Delta_{\theta_f(A)}(\gamma(f), \gamma(g)) \geq \Delta_A(f,g) \tag{7.2}$$

Hence the conjunction of Inequations 7.1 and 7.2 proves the lemma. □

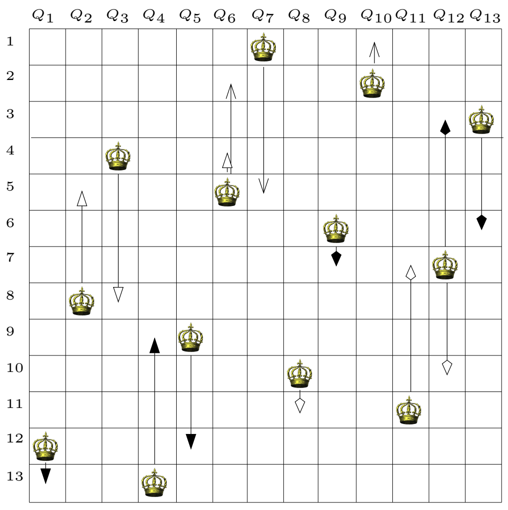
Theorem 50. *Given a CSP with a distributive symmetry γ, $\gamma(f)$ is an (a,b)-super-solution if and only if f is an (a,b)-super-solution.*

Proof. Let f be an (a,b)-*super*-solution of $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ and γ a distributive symmetry. We show that $\gamma(f)$ is an (a,b)-*super*-solution. Since $\gamma(f)$ is, by definition of γ, a solution, we only need to show that every breakage of $\gamma(f)$ accepts a b-*repair*. Consider a b-*repair* g of f for a breakage $A \subseteq \mathcal{X}$. By direct application of Lemma 3, $\gamma(g)$ is b-*repair* of f for the breakage $\theta_f(A)$. We show that all breakages are covered. Consider the set $\mathcal{A}_k = \{A \mid A \subseteq \mathcal{X} \wedge |A| = k\}$ and its image by θ_f, $\mathcal{A'}_k = \{\theta_f(A) \mid A \in \mathcal{A}_k\}$. The function θ_f is bijective and is therefore a permutation on $\mathcal{X}$. Moreover, since γ is distributive, θ_f is also a permutation on subsets of a given size. Since $\mathcal{A}_k$ contains all breakages of size k, so does its its image by θ_f, hence we have $\mathcal{A}_k = \mathcal{A'}_k$ for all $1 \leq k \leq a$. Consequently, all breakages are covered, and $\gamma(f)$ is an (a,b)-*super*-solution.

Conversely, we show that if $\gamma(f)$ is an (a,b)-*super*-solution then f is an (a,b)-*super*-solution. This is in fact a direct consequence of the fact that γ is bijective. $\square$

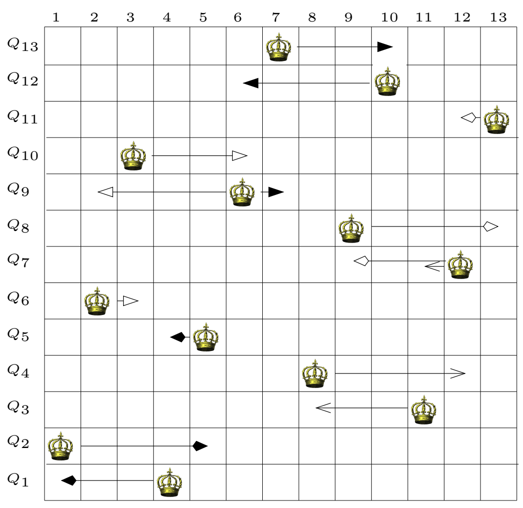
Example 30. *We give an example of* $(1,2)$*-super-solution of a* 13-Queens *problem and its symmetric image by a* $90°$ *rotation. In Figure 7.4 the first chessboard (Figure 7.4a) is filled so that no queen threatens another. Moreover it is a* $(1,2)$*-super-solution, therefore, for any queen, if we must move this queen to another row, then only two other queens need to move to obtain another solution. The repairs are represented as arrows of different shapes indicating an alternative rows for sets of* 3 *queens. If all the moves indicated with similar arrows are made, the resulting chessboard position satisfies the* 13-Queens *problem.*

On the second chessboard (Figure 7.4b), we rotate the solution by $90°$*, and then rotate the chessboard itself by* $-90°$*. In other words, the positive rotation is a symmetric transformation, whilst the negative rotation is in fact equivalent to physically rotating the figure itself. The resulting position is thus identical, however, the queen-variables correspond to rows instead of columns, since only the positive rotation was applied to the constraint model. By a direct application of Theorem 50 we know that the symmetric solution represented in Figure 7.4b is a* $(1,2)$*-super-solution. In other words, a consequence of Theorem 50 is that a solution of the* N-Queens *problem is repairable along rows if and only if it is repairable along columns.*



Q_1 Q_2 Q_3 Q_4 Q_5 Q_6 Q_7 Q_8 Q_9 Q_{10} Q_{11} Q_{12} Q_{13}
1 2 3 4 5 6 7 8 9 10 11 12 13

(a) The repairs of a $(1, 2)$-*super*-solution



1 2 3 4 5 6 7 8 9 10 11 12 13
Q_{13} Q_{12} Q_{11} Q_{10} Q_9 Q_8 Q_7 Q_6 Q_5 Q_4 Q_3 Q_2 Q_1

(b) The repairs of the same solution after a $90°$ rotation

Figure 7.4: A $(1, 2)$-*super*-solution of the 13-Queens problem and its symmetric image by a $90°$ rotation.

7.3.2 Symmetry Breaking and *super*-solutions

In the previous section, we have seen that if a symmetry is distributive, then the symmetric image of an (a, b)-*super*-solution is still an (a, b)-*super*-solution. This result is important since it shows that if we break a symmetry γ, i.e., if we eliminate either of f or $\gamma(f)$ for any assignment f, then we are guaranteed that one (a, b)-*super*-solution will remain, provided that one exists. However, symmetry breaking tools, such as Symmetry Breaking During Search (SBDS) [Backofen 99, Gent 02], Symmetry Breaking by Dominance Detection (SBDD) [Focacci 01, Fable 01], or Symmetry Breaking Constraints [Puget 93, Kiziltan 04] cannot be directly used in the algorithms introduced in this dissertation. Indeed, most of the methods introduced in previous chapters rely on finding solutions witnessing repairs for the (a, b)-*super*-solution. Whereas removing symmetric solutions does not remove all (a, b)-*super*-solutions, it may remove a *repair*, critical for assessing the repairability of the main solution. Although it depends to some extent on the various methods introduced in this dissertation, there exists a simple common answer to this problem: *Symmetries can be broken on the master problem, but not on the sub-problems.* We highlight this limitation through an example, then we detail how to best use symmetry breaking tools for the various SuperCSP algorithms.

Example 31. *Consider a CSP with two variables $X, Y \in \{1, 2\}$, and the following allowed tuples: $\{\langle 2,2\rangle \langle 1,2\rangle \langle 2,1\rangle\}$. This problem has 3 solutions, but only one $(1,0)$-super-solution, $\langle 2,2\rangle$. Now observe that X and Y are symmetric. We can break this symmetry either during search or statically. We might add an ordering constraint between X and Y, for instance $X \leq Y$. By doing so, we eliminate the solution $\langle 2,1\rangle$ because it is symmetric to $\langle 1,2\rangle$. However, we also lose a* 0*-repair, $\langle 2,1\rangle$, which is crucial for proving that $\langle 2,2\rangle$ is a* $(1,0)$*-super-solution. Thus, by breaking the symmetry in a* SuperCSP*, we may loose some super-solutions.*

$\mathcal{P}+\mathcal{P}$: In this reformulation, the network is duplicated and the restriction of a solution to the original part corresponds to the $(1,0)$-*super*-solution whilst the restriction of the same solution to the duplicate corresponds to the *repair*s. The symmetry breaking methods need thus be restricted to the variables in $\mathcal{X}$, and should not apply to the duplicated variables, i.e., $\mathcal{X}^+$.

$\mathcal{P}\times\mathcal{P}$: The situation is trickier here since the part corresponding to the *super*-solution itself is closely entangledwith the part corresponding to repairs. We give an example of how it can be done in the simple case of a symmetry breaking constraint.

Example 32. *Recall that a tuple $\langle\langle v_1, r_1\rangle, \langle v_2, r_2\rangle\rangle$ is consistent in the $\mathcal{P}\times\mathcal{P}$ reformulation if and only if all three tuples $\langle v_1, v_2\rangle$, $\langle v_1, r_2\rangle$ and $\langle r_1, v_2\rangle$ are consistent. Consider two variables X and Y with the same ternary domain ($\{1,2,3\}$), and the following constraint:*

$$X + Y \neq 4$$

We list the corresponding tuples in the reformulation as well as the corresponding sets of 3 *tuples in the original network.*

Now suppose that these two variables are symmetric hence we decide to post an ordering constraint $X \leq Y$. If the constraint was posted on the original constraint network, the tuples in bold font in Figure 7.5 would be forbidden. In every set of 3 *tuples in the rightmost part of Figure 7.5, there is a tuple such that $X > Y$, therefore the $\mathcal{P}\times\mathcal{P}$ reformulation would be empty.*

In fact the symmetry breaking constraint should be taken into account only for the tuples corresponding to the $(1,0)$*-super-solution itself, i.e., the first column of the*

reformulated tuples		original tuples
$\langle\langle 1,2\rangle,\langle 1,2\rangle\rangle$	$\Rightarrow$	$\langle 1,1\rangle$, $\langle 1,2\rangle$, $\langle\mathbf{2,1}\rangle$
$\langle\langle 1,3\rangle,\langle 2,1\rangle\rangle$	$\Rightarrow$	$\langle 1,2\rangle$, $\langle 1,1\rangle$, $\langle\mathbf{3,2}\rangle$
$\langle\langle 2,1\rangle,\langle 1,3\rangle\rangle$	$\Rightarrow$	$\langle\mathbf{2,1}\rangle$, $\langle 2,3\rangle$, $\langle 1,1\rangle$
$\langle\langle 2,3\rangle,\langle 3,1\rangle\rangle$	$\Rightarrow$	$\langle 2,3\rangle$, $\langle\mathbf{2,1}\rangle$, $\langle\mathbf{3,1}\rangle$
$\langle\langle 3,1\rangle,\langle 2,3\rangle\rangle$	$\Rightarrow$	$\langle\mathbf{3,2}\rangle$, $\langle 3,3\rangle$, $\langle 1,2\rangle$
$\langle\langle 3,2\rangle,\langle 3,2\rangle\rangle$	$\Rightarrow$	$\langle 3,3\rangle$, $\langle\mathbf{3,2}\rangle$, $\langle 2,3\rangle$

Figure 7.5: The $\mathcal{P} \times \mathcal{P}$ reformulation, before symmetry breaking.

rightmost part. The reformulation with the constraint $X \leq Y$ used as a symmetry breaking constraint should be as shown in Figure 7.6:

reformulated tuples		original tuples
$\langle\langle 1,2\rangle,\langle 1,2\rangle\rangle$	$\Rightarrow$	$\langle 1,1\rangle$, $\langle 1,2\rangle$, $\langle\mathbf{2,1}\rangle$
$\langle\langle 1,3\rangle,\langle 2,1\rangle\rangle$	$\Rightarrow$	$\langle 1,2\rangle$, $\langle 1,1\rangle$, $\langle\mathbf{3,2}\rangle$
$\langle\langle 2,3\rangle,\langle 3,1\rangle\rangle$	$\Rightarrow$	$\langle 2,3\rangle$, $\langle\mathbf{2,1}\rangle$, $\langle\mathbf{3,1}\rangle$
$\langle\langle 3,2\rangle,\langle 3,2\rangle\rangle$	$\Rightarrow$	$\langle 3,3\rangle$, $\langle\mathbf{3,2}\rangle$, $\langle 2,3\rangle$

Figure 7.6: The $\mathcal{P} \times \mathcal{P}$ reformulation, after symmetry breaking.

`MAC+`: This algorithm uses two sets of domains, $s\mathcal{D}$ and $r\mathcal{D}$. Whereas the former stands for the $(1,0)$-*super*-solution, the latter stands for the *repair*s. Therefore, symmetry breaking methods can only be used to prune or make inference on the the *super*-domain $s\mathcal{D}$ and not on the *repair*-domain $r\mathcal{D}$.

***super*-`MAC`:** The situation is identical as for `MAC+`. The search for a $(1,0)$-*super*-solution is done on $s\mathcal{D}$ whilst the search for *repair*s witnessing repairability is done on $r\mathcal{D}$. Therefore the answer is also identical: We shall restrict symmetry breaking methods to $s\mathcal{D}$.

`decomposition-backtrack-`Γ: Here again there is a clear distinction between the search for the (a,b)-*super*-solution and the search for b-*repair*s. Therefore, symmetry breaking methods can only be used to prune or make inference on the master problem ($\mathcal{P}$) and not on a sub-problem $\mathcal{P}^{\varphi,A}$ created for a partial solution φ and a breakage A.

Other Algorithms All other algorithms, for MaxRepairability or Min-BCSP are all based on the procedure `decomposition-backtrack`, hence the same restriction applies.

7.4 Summary and Limitations

In this chapter we extended the *super*-solution framework by allowing more complex definitions of breakages and repairs. We showed that the algorithm `decompose-backtrack`, introduced in Chapter 5, as well as the derived Branch & Bound algorithms to maximise repairability of minimise the size of the repairs (Chapter 6), can handle these extended definitions.

We defined the notion of robustness model, defined by three components:

- We first defined three sets, the breakage set $\mathcal{BS}$, the repair set $\mathcal{RS}$ and the free variables set $\mathcal{FS}$, to represent respectively which variables can break, which value can be reassigned in a restricted way in response to a breakage and which variables can be reassigned in an unrestricted way.
- Then we defined the concept of a constraint to control the breakages, that is, to restrict which values can or cannot break, and which alternative values are acceptable in reassignment.
- finally, we defined the concept of a constraint to control the repairs, that is restrict which combinations represent valid repairs in response to a breakage.

We subsequently showed that provided that the constraint controlling the breakages is tighter than the classical set of disequality constraint, the inference method based on equality constraints can be used soundly. However, we do not give any simple precondition about the constraint controlling the repairs, hence this question need to be solved on a case per case basis.

Finally we studied the concept of symmetry and symmetry breaking within the *super*-solution framework. We gave a sufficient property, namely, distributivity, to guarantee that *super*-solutions are preserved through a symmetric transformation. Then we showed that although symmetry breaking techniques, such as posting a constraint to exclude symmetrical assignments, cannot be used exactly as for regular CSPs, it is possible to break symmetries when searching for *super*-solutions. The idea is to apply the symmetry on distributive symmetries, and only on the part of the search corresponding to the *super*-solutions, as opposed to the search for repairs.

Chapter 8

Applications and Experimental Results

8.1 Introduction

Fault tolerance is a relatively recent concept. As such, the previous work, by Roy *et al.* on *super*-models, was almost purely theoretical. To our knowledge, the only mention of a method for finding *super*-models of a SAT formula is a reformulation method for $(1, 1)$-*super*-models [Ginsberg 98] and the $\mathcal{P} + \mathcal{P}$ reformulation for finding $(1, 0)$-*super*-solutions in [Weigel 98]. The experiments on the former reformulation were limited to the characterisation of a phase transition and no experiments were done with the $\mathcal{P} + \mathcal{P}$ reformulation. We therefore do not have a solid reference onto which a systematic comparison could be based. In this chapter, we aim to address three general questions related to the framework introduced earlier in this dissertation.

Phase Transition and Computational Complexity: Phase transition phenomena occur in many NP-complete problems. If a combinatorial problem is loosely constrained, many solutions are admissible. On the other hand, when a problem is tightly constrained, it is likely to be unsatisfiable. In randomly generated problems, by varying one of the parameters, the "constraindeness" can be controlled. It has been observed that for numerous problems, the transition between the satisfiable and unsatisfiable regions is sharp. Moreover, Cheeseman *et al.* [Cheeseman 91] showed that a peak in computational cost is often associated with this threshold between satisfiable and unsatisfiable regions. Typically, in the loosely constrained region, it is relatively easy to find a solution. Similarly, in the tightly constrained region, it is often easy to prove that no solution exist. Problems at the phase transition, on the other hand, cannot be easily proved either soluble or insoluble, hence typically have larger computational complexity. Phase transition phenomena have been characterised for several problems, such as Boolean Satisfiability [Cheeseman 91, Kirkpatrick 94, Mitchell 92], Graph colouring [Cheeseman 91], Constraint Satisfaction [Prosser 94, Smith 94], and Travelling Salesman [Gent 96b]. We investigate the phase transition phenomenon for the problem of the existence of *super*-solutions.

Another question related to computational complexity is whether finding *super*-solutions is harder, in practice, than finding regular solutions, and if so, how much harder. A straight comparison would be flawed since the problems solved are not equivalent. Often, the same instance of constraint satisfaction problem can be easily satisfiable when viewed as a regular CSP and more difficult because they are closer to the phase transition when viewed as a SuperCSP. On the other hand an instance can be respectively difficult because it is at the satisfiability phase transition whilst it cannot be easily proved that it has no *super*-solutions. However, even though the problem of finding solutions and *super*-solutions are different, it makes sense to compare their respective computational costs, since they are meant to apply to essentially the same applications.

Algorithms Comparison: We introduced three methods for finding $(1, 0)$-*super*-solutions in Chapter 4. We theoretically compared the worst case computational complexity and the filtering power of these algorithms, along with a fourth, earlier, reformulation method. In this chapter we extend this theoretical analysis with an empirical comparison.

In Chapter 5, we introduced a basic backtracking procedure with a brute-force filtering method. We then enhanced it using an inference method based on equality constraints deduced from the neighbourhood relation in the constraint graph. We empirically assess the effectiveness of this inference method by comparing the efficiency of the `decompose-backtrack` procedure with and without this inference step.

Optimisation and Application: Since finding *super*-solutions comes at a high computational cost and since a constraint network does not always admit any *super*-solution, the most practical use of the framework described in this dissertation seems to be the robustness optimisation algorithms introduced in Chapter 6. Since the problem MinBCSP was showed to have an inherent high computational complexity, we restrict our study to the problem MaxRepairCSP solved with a Branch & Bound procedure using Algorithm 28 (`repairability`^{max}) as filtering method.

The main question we want to address is to evaluate the extent of the tradeoff between computational complexity and robustness, and also between optimality and robustness. Indeed this method allows us to control how much computational effort and how large a decrease in optimality we are prepared to trade against robustness. The search effort can be controlled, since the first solution can be found with the best method available, and then our Branch & Bound algorithm progressively improves the repairability. The deviation in solution quality due to the increased robustness can be controlled as well. Once an optimal or near optimal solution has been found using a standard algorithm, the optimisation problem can be turned into a satisfaction problem where the discrepancy with respect to the optimal outcome is controlled by an extra constraint. We then look for the solution of the resulting problem with maximal repairability.

In Section 8.2 we describe the two benchmark problems we are using throughout this chapter. In subsequent sections, we address the questions stated above. In Section 8.3.1 we study the phase transition of SuperCSPs. Then in Section 8.3.2 we compare the search effort for finding $(1,0)$-*super*-solutions and (a,b)-*super*-solutions with respect to regular solutions. In Section 8.4.1, 8.4.2 and 8.5.1 we assess the significance of the inference methods proposed for respectively solving $(1,0)$-SuperCSPs, (a,b)-SuperCSPs and MaxRepairCSPs. Finally, in Section 8.5.2, we measure the tradeoff between optimality and robustness for the Jobshop Scheduling Problem.

All algorithms for solving $(1,0)$-SuperCSPs, i.e., *super*-`MAC`, `MAC+`, `MAC`$(\mathcal{P}+\mathcal{P})$ and `MAC`$(\mathcal{P}\times\mathcal{P})$ were written using van Beek's library of routines for solving binary constraint satisfaction problems [van Beek 94]. The experiments on these algorithms were all run on Pentium 3 processors, with 512 Mb of Ram under linux redhat 8.0.

The `decompose-backtrack` algorithm, and its Branch & Bound version for solving MaxRepairCSP are implemented using our constraint library [Hebrard 05]. The experiments on these algorithms were all run on Pentium 4 processors, 1024 Mb of ram under linux debian.

8.2 Benchmarks

We used two types of benchmarks to test our algorithms. For most of the experiments, we used the Jobshop Scheduling Problem since it is a well known and successful application of constraint programming, and tackling uncertainty is extremely relevant in this application domain. However, we also used Uniform Random Binary CSPs (URBCSP) for the algorithms that cannot handle non-binary constraints, or when we want to control the complexity of the instances with precision, for instance when studying the phase transition behaviour.

8.2.1 Jobshop Scheduling Problem (JSP)

In the **Jobshop Scheduling Problem** (JSP), a set $\mathcal{J} = \{J_1, \dots J_n\}$ of n jobs need to be scheduled on a set $\mathcal{M} = \{r_1, \dots r_m\}$ of m resources. A job $J_i = \{o_{i1}, \dots o_{im}\}$ is a set of m activities, along with a bijective relation $R_i : J_i \mapsto \mathcal{M}$. Every activity o_{ij} is associated with a duration d_{ij}, a release date s_{ij} and a due date e_{ij}. Moreover an activity o_{ij} must be executed on the resource $R_i(o_{ij})$.

The activities are to be scheduled, that is, a time point needs be allocated to every activity o_{ij}, between its release date s_{ij} and its due date e_{ij}. A schedule therefore associates to every activity o_{i_j} an effective start time es_{ij} and an effective end time ee_{ij} such that $s_{ij} \leq es_{ij} \leq e_{ij} - d_{ij}$ and $ee_{ij} = es_{ij} + d_{ij}$ and must satisfy the following constraints:

- Within a job J_i, the sequence given by activity indices must be respected, i.e. the effective end time ee_{ij} of an activity o_{ij} is lower or equal to the effective start time es_{ij+1} of the following activity o_{ij+1}, if any.
- Two activities sharing a resource must not overlap in time. In other words, for any pair of activities o_{ij} and o_{kl}, if $R_i(o_{ij}) = R_k(o_{kl})$, then the schedule is such that $ee_{ij} \leq es_{kl}$ or $ee_{kl} \leq es_{ij}$.

The objective function we consider in this chapter is to minimise the overall makespan, i.e., the difference between the largest effective end time and the lowest effective start time. Given a schedule f defining a mapping from activities to effective start and end time, the objective function Φ can be defined as follows:

$$\Phi(f) = max\{ee_{ij} \mid i \in [1..n], j \in [1..m]\} - min\{se_{ij} \mid i \in [1..n], j \in [1..m]\}$$

We generated random instances of JSP using Beck's generator based on the work by Watson *et al.* ([Watson 99]) itself a further development over Taillard's [Taillard 93]. In all instances solved in our experiments, the minimal and maximal length of the activities were fixed to 5 and 50 respectively. Therefore, a class of instances of JSP can be described as a pair $\langle \#jobs, \#machines \rangle$. The number of activities is equal to the product of the number of jobs by the number of machines.

8.2.1.1 Standard Model

This model involves nm variables $X_{11}, \dots X_{nm}$, where X_{ij} stands for the the effective start time of the activity o_{ij}. The domain of X_{ij} therefore is the interval bounded by the release date and the due date minus the duration of the corresponding activity, i.e., $[s_{ij}, \dots e_{ij} - d_{ij}]$. The sequence constraint for each job is modelled with a chain of Precedence constraints:

$$\forall J_i \in \mathcal{J}, \;\; \forall j \in [1..m-1], \; (X_{ij} + d_{ij} \leq X_{ij+1})$$

The Non-Overlap constraints are modelled as disjunctions of Precedence constraints:

$$\forall i, k \in [1..n], \; j, l \in [1..m], \;\; (R_i(o_{ij}) = R_k(o_{kl})) \Rightarrow (X_{ij} + d_{ij} \leq X_{kl} \; \vee \; X_{kl} + d_{kl} \leq X_{ij})$$

Example 33. *We illustrate an instance of* JSP *in Figure 8.1. The corresponding constraint network is shown in Figure 8.2. The constraints 8.1, 8.2 and 8.3 are the* Precedence *constraints for jobs J_1, J_2 and J_3 respectively. The constraints 8.4 to 8.6 stand for* Non-Overlap *constraints for resource r_1, whilst the constraints 8.7 to 8.9 and 8.10 to 8.12 correspond respectively to the resources r_2 and r_3.*

This model allows us to solve a "deadline JSP", i.e., the problem of the existence of a schedule with a given makespan. To minimise the makespan, we use a binary search as shown in Algorithm 31. First, an upper bound and a lower bound on the makespan are computed using respectively the procedure `get-upper-bound` and `get-lower-bound`.

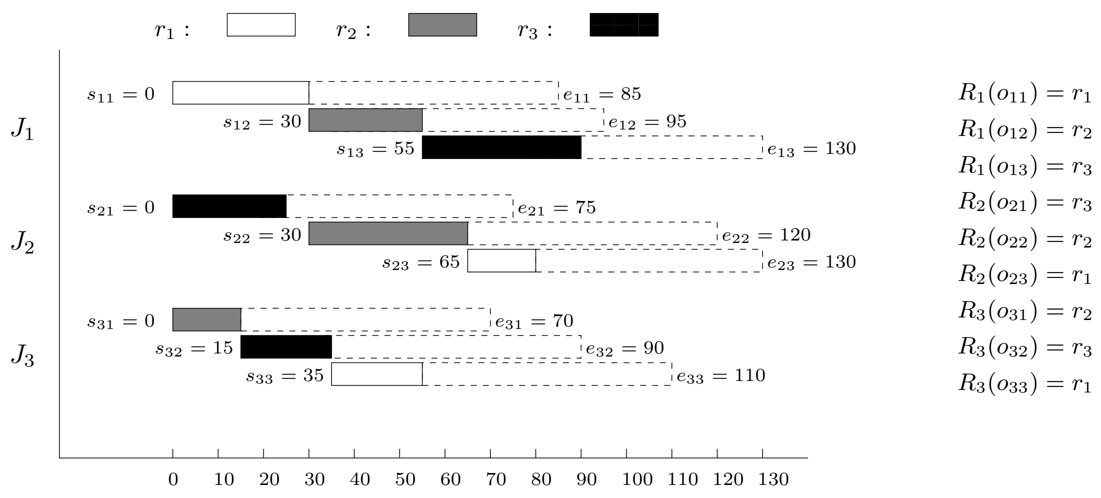


Figure 8.1: An example of Jobshop Scheduling Problem.

Then we start a binary search, where every step is a deadline JSP, and stop when the bounds collapse.

Procedure `get-upper-bound`: This procedure computes a sound upper bound on the minimal makespan for $\mathcal{P}$ (line 1 of Algorithm 31). We implemented it using a simple greedy algorithm. The activity with earliest due date is scheduled first, to the earliest start time that is consistent with earlier decisions. A consistent schedule is thus produced, and its makespan is used as upper bound.

Procedure `get-lower-bound`: This procedure (line 2 of Algorithm 31), computes the largest schedule for a single machine without slack. A schedule for a single machine r_k without slack is simply the sum of the durations of activities that must be executed on r_k. To this sum, we can add the durations of activities that must execute before the first or after the last activity because of the PRECEDENCE constraints on jobs.

8.2.1.2 Model Refinement

We refine our model with a number of techniques introduced for tackling regular scheduling problems. There are two reasons for doing so. First, larger instances can thus be solved, hence it makes the experimental results more significant. Moreover, it demonstrates that the procedure `decompose-backtrack` can accommodate complex

$$
\begin{aligned}
X_{11} + 30 \leq X_{12},\ X_{12} + 25 \leq X_{13} && (8.1)\\
X_{21} + 25 \leq X_{22},\ X_{12} + 35 \leq X_{23} && (8.2)\\
X_{31} + 15 \leq X_{32},\ X_{13} + 20 \leq X_{33} && (8.3)\\
X_{11} + 30 \leq X_{23} \ \vee\ X_{23} + 15 \leq X_{11} && (8.4)\\
X_{11} + 30 \leq X_{33} \ \vee\ X_{33} + 20 \leq X_{11} && (8.5)\\
X_{23} + 15 \leq X_{33} \ \vee\ X_{33} + 20 \leq X_{23} && (8.6)\\
X_{12} + 25 \leq X_{22} \ \vee\ X_{22} + 35 \leq X_{12} && (8.7)\\
X_{12} + 25 \leq X_{31} \ \vee\ X_{31} + 15 \leq X_{12} && (8.8)\\
X_{22} + 35 \leq X_{31} \ \vee\ X_{31} + 15 \leq X_{22} && (8.9)\\
X_{13} + 35 \leq X_{21} \ \vee\ X_{21} + 25 \leq X_{13} && (8.10)\\
X_{13} + 35 \leq X_{32} \ \vee\ X_{32} + 20 \leq X_{13} && (8.11)\\
X_{21} + 25 \leq X_{32} \ \vee\ X_{32} + 20 \leq X_{21} && (8.12)
\end{aligned}
$$

Figure 8.2: The constraint network modelling the JSP shown in Figure 8.1.

Algorithm 31 `minimise-makespan`

```
  Data   : A JSP $\mathcal{P}$
  Result : The minimum makespan for $\mathcal{P}$
1 $ub \leftarrow$ get-upper-bound($\mathcal{P}$);
2 $lb \leftarrow$ get-lower-bound($\mathcal{P}$);
  while $ub > lb$ do
3     if solve($\mathcal{P}, \lfloor \frac{ub+lb}{2} \rfloor$) then
          $ub \leftarrow \lfloor \frac{ub+lb}{2} \rfloor$;
      else
          $lb \leftarrow \lfloor \frac{ub+lb}{2} \rfloor + 1$;
  return $ub$;
```

Figure 8.3: An algorithm for computing the minimum makespan of a Jobshop Scheduling Problem.

propagation methods and variable ordering heuristics and take advantage of them. We briefly describe the three main improvements upon the simple model introduced in Section 8.2.1.1. Then we discuss about a fourth improvement that we eventually did not retain because it introduces new variables and therefore affects the *super*-solutions.

Edge Finder The set of NON-OVERLAP constraints can be modelled as a single global constraint (UNARYRESOURCE). A propagation algorithm for this constraint, `Edge-Finder`, has been introduced in [Carlier 94], [Nuijten 94b] and [Nuijten 94a] and further developed in [Vilim 04]. This algorithm has been very successfully applied to a range of scheduling problems. We do not describe this method as it is beyond the scope of this dissertation.

Shaving: We also used a consistency techniques known as **shaving** [Martin 96] in the scheduling community, and closely related to **Singleton Arc Consistency** [Debruyne 97] [Prosser 00], in the constraint programming community. In fact, shaving usually applies only on the domain bounds and requires ordered domains, whilst Singleton Arc Consistency (SAC) is defined on discrete domains. However both methods can be described as checking unary assignments by temporarily committing to them, and then applying a consistency technique to the whole network. Shaving uses bounds consistency whilst singleton arc consistency uses arc consistency. This procedure is repeated until a fixed point is reached, that is, all unary assignments are singleton consistent.

Operation Resource Reliance: Finally, we used the variable ordering heuristic described in [Sadeh 96]. This heuristic (**Operation Resource Reliance**, ORR) uses probabilistic reasoning to compute the criticality of resources and activities. Each possible start time for each activity is given equal probability, that is, $1/|\mathcal{D}(X_{ij})|$ for all values, or possible start times, of an activity o_{i_j}. Then for each resource r_k, we can compute the reliance on r_k for each time point as the sum of the probabilities that an activity will be in execution at this time. The variable whose corresponding activity participate to the largest, hence most critical, peak in reliance to a resource is chosen first.

Precedence Constraint Posting: The PRECEDENCE Constraint Posting model (PCP), has been first introduced by Smith and Cheng as a heuristic in [Smith 93] and [Cheng 94] and subsequently as a complete method in [Cheng 95]. In this alternative model, the decision variables represent precedence relations between activities sharing a resource. For every pair of activities $o_{ij},\ o_{kl}$ if $R_i(o_{ij}) = R_k(o_{kl})$ then a Boolean variable Y_{ijkl} is introduced. This variable indicates which of the precedence relation $o_{ij} \prec o_{kl}$ or $o_{kl} \prec o_{ij}$ is satisfied. For each variable Y_{ijkl} a conditional PRECEDENCE constraint is posted:

$$Y_{ijkl} \Rightarrow o_{ij} \prec o_{kl} \ \wedge\ \neg Y_{ijkl} \Rightarrow o_{kl} \prec o_{ij}$$

The variables standing for start times can be used to post conditional or unconditional PRECEDENCE constraints. We found that this model was the most efficient, particularly in conjunction with shaving and the `Edge-Finder` algorithm. However, this model complicates the search for *super*-solutions. Indeed, we need to decide if the new variables can or cannot break or be used as repairs. Using the notations introduced in Chapter 7, we can restrict the breakage and repair sets ($\mathcal{BS}$ and $\mathcal{RS}$) to the original variables X_{ij}, whilst the new variables Y_{ijkl} are put into the free set ($\mathcal{FS}$). Although, this modelling of breakages leads to a model equivalent to the classical definition on the simple model, it considerably reduces the performance of our algorithm. In fact, in the PCP model, only the new variables (Y_{ijkl}) are searched, since once these variable are instantiated, the resulting sub-problem is a simple temporal constraint network, and thus is satisfiable if and only if it is GAC. However, the breakages and repairs, in the model defined above, are defined only on the original variables. We believe that assigning last the variables that really matter for breakages and repairs is the reason for the inefficiency of this method. We therefore did not use this model. Observe, on the other hand, that breakages on the new variables Y_{ijkl} might in fact be interesting in some applications. For instance if we want to minimise the changes on the sequence of activity for each resource in the event of a breakage. In such cases, the PCP model would certainly be the best choice. This emphasises the fact that *super*-solutions are dependent on the modelling choices.

8.2.1.3 Uncertainty

The Jobshop Scheduling Problem is an example of an application where applying the strict definition of breakages and repairs is not sufficient to capture certain aspects of uncertainty that one may want to model. Indeed, since variables stand for activities and values for time points, if we assume that breakages are discovered while executing a schedule, certain repairs involving choices in the past are not possible. This situation was used as example in Chapter 7. We thus define a breakage constraint and a repair constraint specific to the Jobshop Scheduling Problem. A breakage is in fact a delay of duration d for the effective start time of an activity. If the time point t is not available, then neither is $t-1$ or earlier time points. Moreover, any value lower than $t+d$ is also unavailable.

$$C_{break}(A) = \forall X \in A \ (X + d \leq X^{\varphi,A})$$

For all instances in our experiments, the delay d is set to 2, meaning 2 units of time. For reference, the duration of an activity is at least 5 and at most 50 units of time. When repairing, the fact that values represent time points also has consequences. We assume that the delay is caused by a faulty machine or a temporary unavailability of a resource, and cannot be discovered before the expected start time of the delayed activity. Moreover, an activity cannot be assigned to a start time t in response to a delay of an activity initially scheduled to start at a start time $t' > t$. The constraint C_{repair} is based on the SIMILARITY, however, when a variable is reassigned it must be reassigned to a later start time than that of the earliest delayed activity. Moreover, the scope of this constraint is limited to activities whose start times are later that of the earliest delayed activity. We use the SIMILARITY-JSP defined in Section 7.2.3:

$$\begin{gathered} C_{repair}(V, \varphi, A, b) \\ \Leftrightarrow \\ \textsc{Similarity}(V, \varphi, A, b) \ \wedge \ \forall X \in V, \ (X = X^{\varphi,A} \vee X^{\varphi,A} \geq min(A)) \end{gathered}$$

Notice that there are several other refinements that could be added to the modelling of breakages and repairs. We chose this robustness model as it is close to the original definition and at the same time captures some “realistic” properties of a stable solution. By Theorem 47 and Theorem 48, we know that the inference methods based on neighbourhood and preprocessing can be used when using this model for the breakages and repairs. Moreover since the constraint controlling the breakages is strictly tighter than the classical disequality constraint, we can check breakages on variables that are not yet assigned.

We use in our experiments five algorithms for finding regular solutions or (a, b)-*super*-solutions of Jobshop Scheduling Problem:

- `JSPsolve`: We denote `JSPsolve` the algorithm for solving regular JSP and using `Edge-Finder`, shaving, and the ORR variable ordering heuristic.

- *super*-`JSPsolve`: The algorithm, using the same components as `JSPsolve`, however replacing the classical backtrack search with the `decompose-backtrack` procedure and **not** using the neighbourhood-based inference is referred to as *super*-`JSPsolve`.

- <u>*super*-`JSPsolve`-Γ</u>: The same procedure, augmented with the neighbourhood-based inference is denoted *super*-`JSPsolve`-Γ.
- <u>*super*-`JSPsolve`^{max}</u>: The Branch & Bound procedure for maximising the repairability of a schedule and using `repairability`^{max} as filtering method whilst **not** using neighbourhood-based inference is referred to as *super*-`JSPsolve`^{max}.
- <u>*super*-`JSPsolve`^{max}-Γ</u>: Finally, the Branch & Bound procedure for maximising the repairability of a schedule and using `repairability`^{max} as filtering method with neighbourhood-based inference is referred to as *super*-`JSPsolve`^{max}-Γ.

8.2.2 Uniform Random Binary Constraint Satisfaction Problem

We also use randomly generated problems (URBCSP) because the search effort necessary to solve an instance can be controlled with a good precision, by setting the parameters of the class of instances. Moreover, some of the algorithms we introduced, such as *super*-`AC` and `MAC`$(\mathcal{P}\times\mathcal{P})$ only apply to binary constraint networks and moreover, as implemented, some other algorithms, such as `MAC+` and `MAC`$(\mathcal{P}+\mathcal{P})$ cannot handle non-binary and global constraints. The algorithms for (a,b)-SuperCSP and (a,b)-MaxRepairCSP, on the other hand are integrated into constraint toolkits able to deal with complex constraints and propagation algorithms. However, even in that case, binary constraint networks were useful to answer some of the questions we want to address. The classical `domain/degree` [Bessiere 96] dynamic variable ordering was used in all experiments involving URBCSPs.

8.2.2.1 Model

The problem instances are generated according to model B in [Prosser 96], and an instance, that is, a constraint network $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ can be described with the following parameters:

- The number of variables $n = |\mathcal{X}|$.
- The uniform domain size $d = |\mathcal{D}(X)|\ \forall X \in \mathcal{X}$.

- The number of constraints $m = |\mathcal{C}|$.
- The uniform constraint tightness $t = 1 - \frac{|C(X,Y)|}{d^2}$.

We generated URBCSP instances using Frost and Bessiere's generator [Frost 96].

8.2.2.2 Uncertainty

Since these problems do not correspond to real world entities we used the standard definition of breakage and repair. We use three additional parameters to describe the type of *super*-solutions that are considered:

- The breakage size a.
- The maximum repair size b.
- The breakage set cardinality $k = |\mathcal{BS}|$

The repair and free set are always set equal to $\mathcal{X}$ and $\emptyset$ respectively. We can therefore define a class of instances as a tuple $\langle n, d, m, t, a, b, k \rangle$.

We use in our experiments four algorithms for finding $(1, 0)$-*super*-solutions on URBCSPs and two algorithms for finding (a, b)-*super*-solutions as well as `MAC` for finding regular solutions. The list of algorithms used in these experiments is as follows:

- `MAC`: The classical backtracking algorithm for solving CSPs, as described in Algorithm 1.
- `MAC`$(\mathcal{P} + \mathcal{P})$: The `MAC` algorithm applied to the $(\mathcal{P} + \mathcal{P})$ reformulation (def. 13) of a constraint network $\mathcal{P}$ and where a solution found can be interpreted as a $(1, 0)$-*super*-solution of $\mathcal{P}$.
- `MAC`$(\mathcal{P} \times \mathcal{P})$: The `MAC` algorithm applied to the $(\mathcal{P} \times \mathcal{P})$ reformulation (def. 14) of a constraint network $\mathcal{P}$ and where a solution found can be interpreted as a $(1, 0)$-*super*-solution of $\mathcal{P}$.
- `MAC+`: The backtracking algorithm using GAC+ as filtering method, as described in Algorithm 13.
- *super*-`MAC`: The backtracking algorithm using *super*-GAC as filtering method, as described in Algorithm 14.

- `decompose-backtrack`: The backtracking algorithm (Algorithm 15) using the brute-force version of the procedure `repairability` (Algorithm 16).
- `decompose-backtrack-Γ`: The backtracking algorithm (Algorithm 20), using the procedure `repairability-Γ` taking advantage of equality constraints and neighbourhood-based inference (Algorithm 19).

8.3 Phase Transition and Computational Complexity:

8.3.1 Phase Transition

Following previous work by Smith *et al.* and Williams *et al.* [Smith 96, Williams 94] on predicting the phase transition for constraint satisfaction, we locate the phase transition of finding *super*-solutions both experimentally and by an approximation based on Markov's inequality:

$$Pr(|sol(\mathcal{P})| \neq 0) < |sol(\mathcal{P})|$$

Where $Pr(|sol(\mathcal{P})| \neq 0)$ is the probability that the set $sol(\mathcal{P})$ has a non-null cardinality. So when $|sol(\mathcal{P})| < 1$, the probability that $\mathcal{P}$ is satisfiable is lower than 1. Hence, in [Smith 96, Williams 94] the phase transition is predicted around $|sol(\mathcal{P})| = 1$. We base our approach on the *κappa* framework [Gent 96a].

For a constraint network $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$, the expected number of solutions is:

$$|sol(\mathcal{P})| = (\prod_{X \in \mathcal{X}} |\mathcal{D}(X)|) \times (\prod_{C(V) \in \mathcal{C}} (1 - t_{C(V)})) \tag{8.13}$$

Where, given a constraint $C(V)$, the term $1 - t_{C(V)}$ is the complement to 1 of the tightness of $C(V)$, that is the ratio of allowed tuples in $C(V)$:

$$t_{C(V)} = 1 - \frac{|C(V)|}{\prod_{Y \in V} |\mathcal{D}(Y)|}$$

In our case, domain sizes and constraint tightness are uniform and all constraints are binary. Therefore, if d stand for the domain size and t for the constraint tightness, whilst n and m are respectively the number of variables and constraints, the formula can be simplified as follows:

$$1 = d^n(1-t)^m$$

We use the $\mathcal{P} \times \mathcal{P}$ reformulation to extend the formula to $(1, 0)$-*super*-solutions. The $\mathcal{P} \times \mathcal{P}$ reformulation has one solution if and only if $\mathcal{P}$ has a single *super*-solution. Therefore, we use the same formula, where n', m', d' and t' are the new values for respectively n, m, d and t. To approximate the value of this formula,we derive the values of the primed parameters (on $\mathcal{P} \times \mathcal{P}$) from the values of the corresponding parameters in the original constraint network ($\mathcal{P}$). The number of variables and constraints are not changed, hence $n' = n$ and $m' = m$. The domain size is known exactly:

$$d' =_{def} (d^2 - d)$$

Moreover, we can see $(1 - t')$ as the probability that a tuple of values on a pair of constrained variables satisfies the constraint. Let $C(X, Y) \in \mathcal{C}$ be a constraint of $\mathcal{P}$. The constraint $C^{\times}(X, Y)$ is the reformulated constraint corresponding to $C(X, Y)$ in $\mathcal{P} \times \mathcal{P}$. The tuple $\langle\langle v_1, v_2\rangle, \langle w_1, w_2\rangle\rangle$, satisfies $C^{\times}(X, Y)$ if and only if $\langle v_1, w_1\rangle$, $\langle v_1, w_2\rangle$ and $\langle v_2, w_1\rangle$ all satisfy $C(X, Y) \in \mathcal{C}$. Hence we can approximate the probability that an arbitrary tuple satisfies the reformulated constraint as the probability that 3 given tuples all satisfy the original constraint. Hence we approximate $(1 - t')$ as follows:

$$(1 - t') = (1 - t)^3$$

The formula thus becomes:

$$1 = (d^2 - d)^n (1 - t)^{3m} \tag{8.14}$$

8.3.1.1 Experimental Setting

The formula 8.13 ties the parameters of a CSP instance together so that when all but one of the arguments are fixed, the last takes the value corresponding to the phase transition. In this experiment we compare the observed and predicted values of the constraint tightness t at the phase transition, for a range of constraint density $\frac{2m}{n(n-1)}$ and fixed number of variables and domain size. We generate and solve a number of samples of 100 instances of URBCSP. In each sample, we fix the number of variables n to 40 and the uniform domain size d to 10. Moreover, the constraint tightness take a value between 0.1 and 0.9 and the constraint density between 0.1 and 0.32, i.e., between 75 and 255 constraints. The gap between each value of tightness is 0.02 whilst the density gap is 0.012, i.e., 10 constraints. This constitutes 760 samples (19 values of density and 40 values of tightness) denoted, as a whole, sample 8.15. We solve

$$\langle n = 40, d = 10, m = [75..255], t = [0.1..0.9] \rangle \tag{8.15}$$

these samples of 100 instances of URBCSP using `MAC` and *super*-`MAC`. Each value of $m \in \{75, 85, 95, \dots 255\}$ defines a sequence of 40 samples with increasing constraint tightness. For every sequence, we report the minimum value of constraint tightness for which more than half of the instance are unsatisfiable. Indeed, this point is usually accepted as the peak of difficulty at the phase transition. The same process was done once for the problem of finding regular solutions (CSP) and once for the problem of finding $(1,0)$-*super*-solutions ($(1,0)$-SuperCSP). We compare these data points to the expected values computed using formula 8.13 and our modified version (formula 8.14) in Figure 8.4.

8.3.1.2 Experimental Results

In figure 8.4, we plot, across the different values of density for which samples were generated and solved, the following values:

- `CSP threshold (prediction)`: The value of tightness for which we expect to encounter the threshold satisfiable/unsatisfiable according to κ.
- `CSP threshold (experimentally)`: The value of tightness for which the unsatisfiable instances outnumber the satisfiable instances within a sample.
- `super CSP threshold (prediction)`: The value of tightness for which we expect to encounter the threshold satisfiable/unsatisfiable according to our modified version of κ.
- `super CSP threshold (experimentally)`: The value of tightness for which the unsatisfiable instances, that is, such that no $(1,0)$-*super*-solutions exist, outnumber the satisfiable instances within a sample.

We observe that our modified version of κ approximates closely the empirical findings. Notice that the value of constraint tightness is slightly overestimated by κ

both for CSPs and SuperCSPs. This is in fact not surprising since the value predicted by the formula is the value of constraint tightness for which the probability of having exactly one solution is maximal whereas the observed data points correspond to the values where the number of unsatisfiable instances exceeds the number of satisfiable ones.

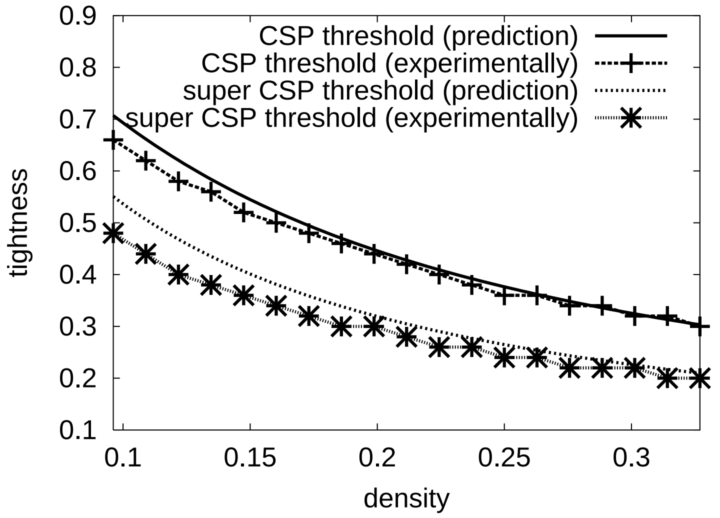


Figure 8.4: Phase transition expectation.

8.3.2 Comparison between CSP and SuperCSP

In this section we want to empirically quantify the increase in complexity when finding *super*-solutions rather than regular solution. In Chapter 3, we have seen that the problem of the existence of *super*-solutions is NP-complete, hence not harder than a regular CSP. However, in practice, finding *super*-solutions appears to be dramatically more difficult. Some theoretical results have suggested it might be harder. For instance, a number of tractable classes of CSP are not tractable anymore when searching *super*-solutions. Moreover, the complexity of the inference method used within the `decompose-backtrack`-Γ algorithm is much larger than that of `MAC`. In fact we shall see that even finding $(1,0)$-*super*-solutions is orders of magnitude more difficult than finding regular solutions. This result is more surprising as the complexity of *super*-`AC` is not larger than that of GAC and most of the tractability results lift to $(1,0)$-SuperCSP.

8.3.2.1 Experimental Setting

The difficulty we face here is that we aim at comparing two algorithms, `MAC` and *super*-`MAC` that are not solving the same problem. We solve a number of URBCSP instances with domain size and number of variables fixed whilst constraint tightness and density are distributed on a wide range of values. We record how hard, on average, are the instances in each sample with the same set of parameters, hence we can see the behaviour of both algorithms on a wide landscape of instances.

Full Fault tolerance (URBCSP**):** We first compare *super*-`MAC` to `MAC`, i.e., finding $(1, 0)$-*super*-solutions rather than solutions. We use the samples 8.15, as defined in Section 8.3.1.1.

Weak Fault tolerance (URBCSP**):** We then compare `decompose-backtrack-`Γ (for finding $(1, 1)$-*super*-solutions) to `MAC`. We use slightly smaller instances. The number of variables and domain size are fixed to respectively 30 and 8. Then, for a range of constraint tightness (0.1 to 0.9 with increment of 0.03) and density 0.11 to 0.3 with increment of 0.005 (that is, 50 to 130 constraints with increment of alternatively 2 and 3), we generate a sample of 100 instances for each combination:

$$\langle n = 30, d = 8, m = [50..130], t = [0.1..0.9]\rangle \tag{8.16}$$

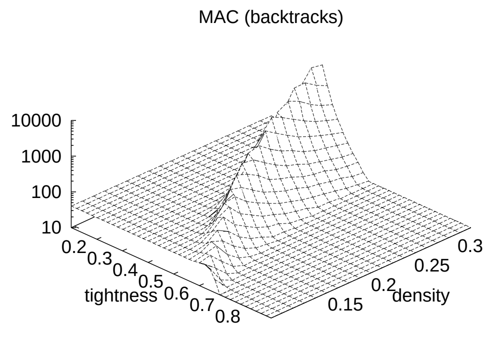
Weak Fault tolerance (JSP**):** Last, we compare the difficulty of solving regular Jobshop Scheduling Problems (JSP) as opposed to finding optimal $(1, 1)$-*super*-solutions for that same problem. We randomly generate a sample of 100 JSPs for problem sizes ranging from 4 jobs and 3 machines, i.e., 12 activities, to 8 jobs and 8 machines, i.e., 64 activities.

$$\langle X \text{ jobs},\ X \text{ machines}\rangle,\ X \in \{4, 5, \dots 8\} \tag{8.17}$$
$$\langle Y \text{ jobs},\ Y - 1 \text{ machines}\rangle,\ Y \in \{4, 5, \dots 8\} \tag{8.18}$$

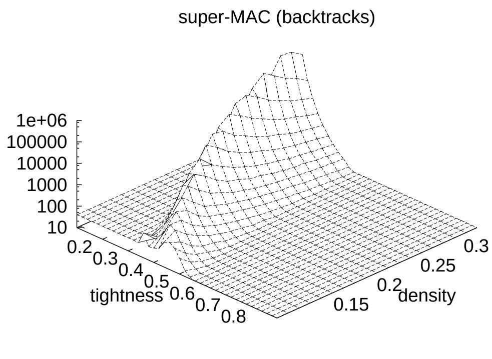
Then we solve these instances using either `JSPsolve` to find regular schedules, or *super*-`JSPsolve-`Γ to find $(1, 1)$-*super*-schedules satisfying the breakage and repair constraints described in Section 8.2.1.3. In order to solve an instance we used Algorithm 31, `minimise-makespan` in both cases. We put a time cutoff of 30 seconds on each "deadline" JSP. When an instance of deadline JSP is not solved within the time cutoff, it is considered as unsatisfiable for the algorithm `minimise-makespan`. Therefore, the makespan eventually found is not necessarily optimal.

8.3.2.2 Experimental Results

Full Fault tolerance (URBCSP**):** In Figure 8.5, the number of backtracks needed for `MAC` (fig. 8.5a) and *super*-`MAC` (fig. 8.5b) are plotted against the constraint tightness and density. We can observe the typical phase transition behaviour easy/hard/easy. Moreover, we observe that the threshold around which this transition occurs is shifted to lower values of tightness for *super*-`MAC`. This was expected since a constraint network accepts strictly less $(1,0)$-*super*-solutions than regular solutions. We can also observe that the hardest instances for *super*-`MAC` require several orders of magnitude more backtracks than the hardest CSP instances within this range of constraint networks.



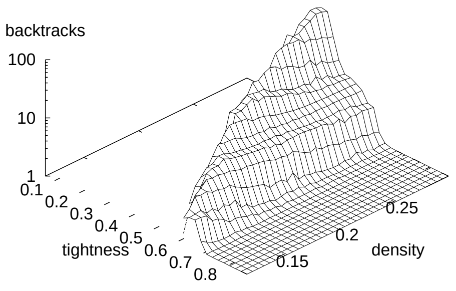
(a) Number of backtracks for `MAC`



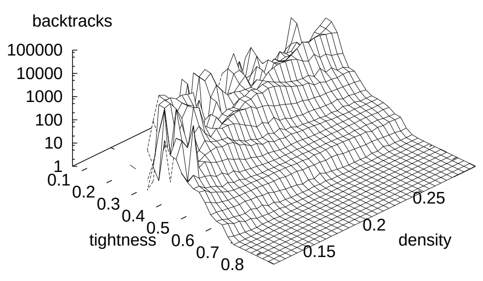
(b) Number of backtracks for *super*-`MAC`

Figure 8.5: Comparison between `MAC` and *super*-`MAC` on a range of URBCSP instances.

Weak Fault tolerance (URBCSP**):** Similarly, in Figure 8.6, the number of backtracks needed for `MAC` (fig. 8.6a) and `decompose-backtrack-`Γ (fig. 8.6b) are plotted against the constraint tightness and density. However, if `decompose-backtrack-`Γ indeed has a phase transition phenomenon, this is not exactly the easy/hard/easy landscape we expected. In fact in the underconstrained region (low tightness and density), some instance are exceptionally hard as observed for satisfiability problems in [Gent 94]. These instances are very loose, yet a wrong decision early in the search leads to an unsatisfiable branch that needs to be refuted. It seems that this situation can happen frequently on (a, b)-SuperCSPs, indeed, a wrong decision early in the search may prevent an otherwise loose problem to not be (a, b)-repairable. However, it is interesting to observe that the number of breakages checked by `decompose-backtrack`-Γ during search, shown in Figure 8.7, is at its peak at the tightness threshold where the number of unsatisfiable instances becomes larger than the number of satisfiable instances. Here again, we observe that the hardest instances for `decompose-backtrack`-Γ require several orders of magnitude more backtracks than the hardest CSP instances within this range of constraint networks.



(a) `MAC`: number of backtracks



(b) `decompose-backtrack`-Γ: number of backtracks

Figure 8.6: Comparison between `MAC` and `decompose-backtrack`-Γ on a range of UR-BCSP instances.

Weak Fault tolerance (JSP): We report the cpu-time in Figure 8.8, number of backtracks in Figure 8.9 and minimum makespan in Figure 8.22. Whereas `JSPsolve` is able to solve all instances to optimality, *super*-`JSPsolve`-Γ scales badly. On the largest instances (8 jobs, 8 machines), *super*-`JSPsolve`-Γ takes about 11 times more time to find an optimal $(1, 1)$-*super*-solution than `JSPsolve` for finding a solution. On instances with more jobs than machines, the difference is even greater, up to a factor 37. However, the discrepancy is not as marked as in random binary problems. In fact,

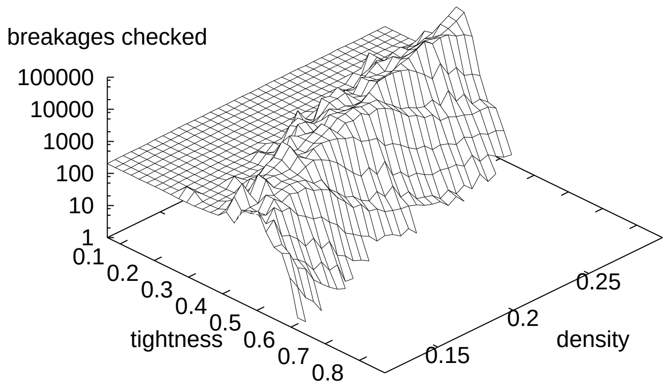


Figure 8.7: `decompose-backtrack-`Γ: number of breakages checked.

approximately half of the instances with 8 jobs and 8 machines were solved to optimality by *super*-`JSPsolve-`Γ within the 30 seconds cutoff, i.e., the solutions returned were the $(1,1)$-*super*-solutions with the shortest possible makespan. We believe that this relatively good result, compared to the random binary case, is due to the fact that breakages on variables that are not already assigned can be checked for repairability. This extra pruning, added to the fact that singleton consistency and the `Edge Finder` procedure is applied also when preprocessing a sub-problem, greatly reduces the search effort.

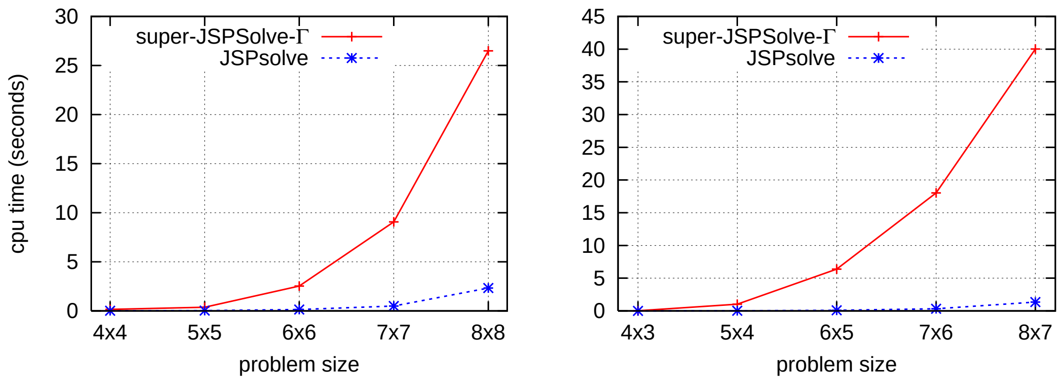


Figure 8.8: Comparison between JSP and $(1,1)$-SuperJSP (cpu-time).

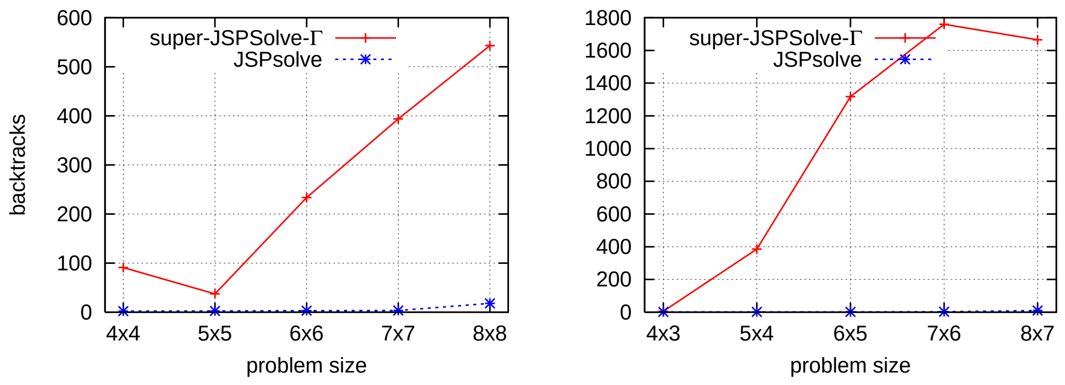


Figure 8.9: Comparison between JSP and (1, 1)-SuperJSP (backtracks).

8.4 Algorithms Comparison:

8.4.1 Algorithms for (1, 0)-SuperCSP

We compare the four methods described in Chapter 4 on random binary CSPs.

8.4.1.1 Experimental Setting

We compare MAC($\mathcal{P} \times \mathcal{P}$), *super*-MAC MAC($\mathcal{P} + \mathcal{P}$) and MAC+ on problem instances at the phase transition. The phase transition was located by iteratively solving samples with increasing constraint tightness and all other parameters fixed. When less than half of the instances within a sample admit a (1, 0)-*super*-solution, we fix the tightness to the current value. We used the following two classes of URBCSP:

$$\langle n = 50, d = 15, m = 100, t = 0.506, a = 1, b = 0, k = 50\rangle \qquad (8.19)$$
$$\langle n = 100, d = 6, m = 250, t = 0.277, a = 1, b = 0, k = 100\rangle \qquad (8.20)$$

Then. for each class, a sample of 100 instances is generated and solved with every algorithm. We use a time cutoff of 3000 seconds for solving a single instance.

8.4.1.2 Experimental Results

We report the average cpu-time and number of backtracks for solving the instances in samples 8.19 and 8.20 for every algorithm in table 8.1. Since for MAC+ and MAC($\mathcal{P}$ +

$\mathcal{P}$) not all instances could be solved within the time cutoff, we report the number of instances that could not be solved. The best results are in bold font.

	`MAC+`	`MAC`$(\mathcal{P}+\mathcal{P})$	`MAC`$(\mathcal{P}\times\mathcal{P})$	*super*-`MAC`
$\langle n=50, d=15, m=100, t=0.5, a=1, b=0, k=50\rangle$				
cpu-time (s)	788	43	53	**1.8**
backtracks	152601000	111836	**192**	2047
time out (3000 s)	12	**0**	**0**	**0**
$\langle n=100, d=6, m=250, t=0.27, a=1, b=0, k=100\rangle$				
cpu-time (s)	2257	430	3.5	**1.2**
backtracks	173134000	3786860	**619**	6487
time out (3000 s)	66	7	**0**	**0**

Table 8.1: Comparison of full fault tolerance algorithms.

We observe that the empirical results agree with the theoretical comparison on the filtering power of these four algorithms. The strongest algorithm, i.e., the algorithm requiring the smallest search tree, is `MAC`$(\mathcal{P}\times\mathcal{P})$ followed by *super*-`MAC`, `MAC`$(\mathcal{P}+\mathcal{P})$ and `MAC+` in this order. The results on cpu-time also are in line with the theoretical analysis, since `MAC`$(\mathcal{P}\times\mathcal{P})$ has a higher worst case time complexity than all other methods. In fact the most surprising result is how much `MAC`$(\mathcal{P}+\mathcal{P})$ outperforms `MAC+` in terms of backtracks, hence in cpu-time on relatively big instances. In conclusion, *super*-`MAC` outperforms all other algorithms as soon as the size of the problem increases.

8.4.2 Algorithms for (a,b)-SuperCSP

In this section we compare the naive version of `decompose-backtrack` to the version using the inference method based on neighbourhood described in Chapter 5 (`decompose-backtrack-`Γ).

8.4.2.1 Experimental Setting

We use both URBCSP and JSP instances to assess the improvement gained by using the inference method based on equality constraints and the neighbourhood relation in the constraint graph.

Uniform Random CSPs We compare the algorithms `decompose-backtrack` and `decompose-backtrack-`Γ for finding $(1,1)$-*super*-solutions on URBCSP instances. Since the peak of difficulty is less clearly centred on the satisfiable/unsatisfiable threshold for these algorithms, we solve instances on a range of constraint tightness values instead of only one class of instances at the phase transition. We used the following two classes of URBCSP:

$$\langle n = 50, d = 10, m = 75, t = [0.2..0.9], a = 1, b = 1, k = 20\rangle \tag{8.21}$$
$$\langle n = 50, d = 10, m = 125, t = [0.2..0.9], a = 1, b = 1, k = 20\rangle \tag{8.22}$$

Then. for each class, and for each value of tightness between 0.2 and 0.9 with an increment of 0.02, a sample of 100 instances is generated and solved with both algorithms. We use a time cutoff of 600 seconds.

JSP We use the same JSP the samples 8.17 and 8.18 used in Section 8.3.2.1. However we solve these instances using *super*-`JSPsolve`, i.e., without the inference method.

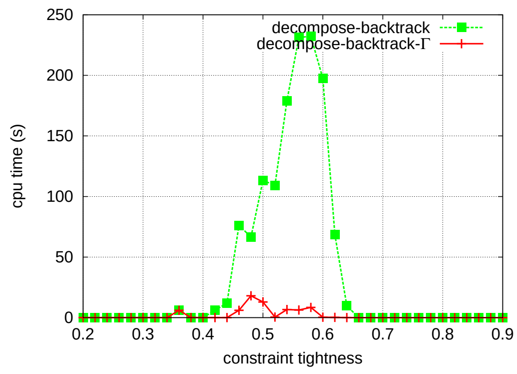
We also compare *super*-`JSPsolve` with *super*-`JSPsolve-`Γ for finding $(1,3)$-*super*-solutions, in order to evaluate the impact of the size of *repair*s on the performance of the algorithms. The instances generated for this case are slightly smaller. We do not show the results for the 8×7 and 8×8 instances since both methods almost always timed out within the 30 seconds cutoff.

8.4.2.2 Experimental Results

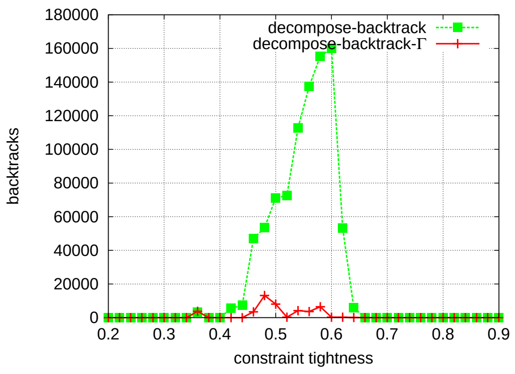
Uniform Random CSPs In figure 8.10 and 8.11 we plot, across the different values of constraint tightness for which samples were generated and solved, the following values:

- `decompose-backtrack`: The number of backtracks or the cpu-time needed to find a *super*-solution of the given class of constraint network without using the neighbourhood-based inference.

- `decompose-backtrack-`Γ: The number of backtracks or the cpu-time needed to find a *super*-solution of the given class of constraint network when using the neighbourhood-based inference.

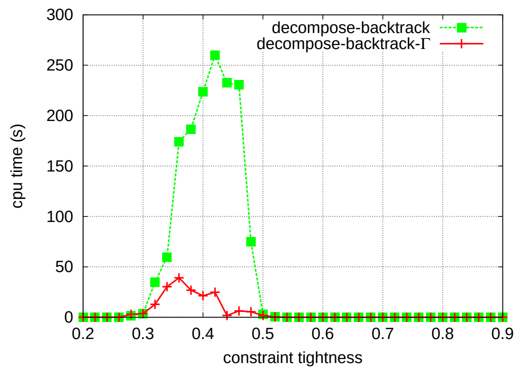


(a) cpu-time (seconds)

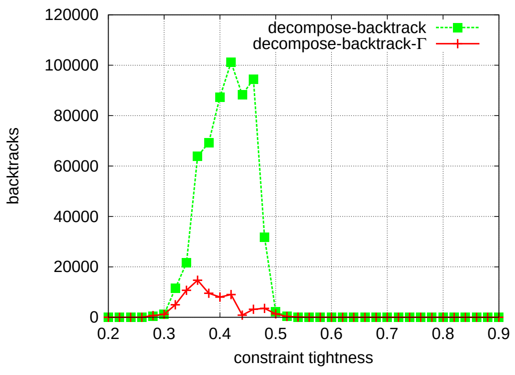


(b) backtracks

Figure 8.10: Neighbourhood-based inference: 75 constraints.



(a) cpu-time (seconds)



(b) backtracks

Figure 8.11: Neighbourhood-based inference: 125 constraints.

Clearly, `decompose-backtrack-`Γ dominates `decompose-backtrack` both in the size of the search tree and in cpu-time. We observe that the highest peak of computational complexity is not where the phase transition phenomenon usually takes place. The threshold between mostly unsatisfiable and mostly satisfiable instances is at respectively 0.62 and 0.48 of constraint tightness for the sample of 75 constraints and 125 instances. On the curve standing for the `decompose-backtrack-`Γ we can see that this

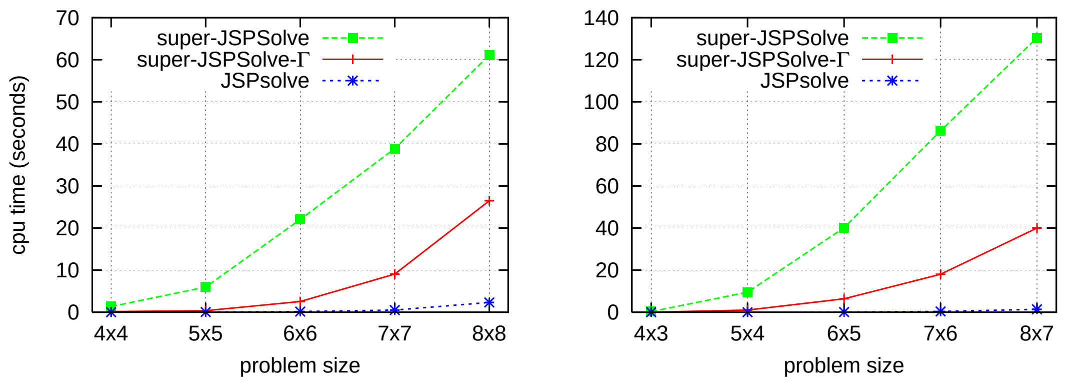


Figure 8.12: $(1,1)$-SuperJSP: CPU Time (seconds)

correspond to a second peak in computational complexity, slightly lower than the first peak, due to the "exceptionally hard instances" phenomenon. The first peak for the brute-force algorithm is comparatively not as important, however, this is due to the time cutoff that we imposed. Since the first peak is due to a few instances with very large search tree among numerous very easy instances, this phenomenon is not as visible as it should be because of the time cutoff. At the phase transition, however, the difficulty is more homogeneous, and even if the time cutoff also favours the brute-force method in these results, the curve is more indicative of the real gain in using the inference based on equality constraints and neighbourhood.

JSP In figure 8.12, 8.13 and 8.14 we respectively report the cpu-time, number of backtracks and minimum makespan found, across the different problem sizes for which samples were generated and solved for `JSPsolve`, *super*-`JSPsolve`-Γ and *super*-`JSPsolve`.

We observe that here again, the inference based on neighbourhood and equality constraints represents a significant improvement over the brute-force method. The time cutoff that we put on the resolution of each deadline problem is quite low. Indeed, many deadline problems may need to be solved in order to optimise the makespan. Therefore the gain of the inference method is best seen in the makespan graph (Figure 8.14). With both *super*-`JSPsolve` and *super*-`JSPsolve`-Γ algorithms we search for $(1,1)$-*super*-schedules with minimal makespan. The length of the minimal makespan is thus the same in both cases, however, since *super*-`JSPsolve` is often not able to solve a given deadline problem within the 30 seconds limit, *super*-`JSPsolve`-Γ produces better

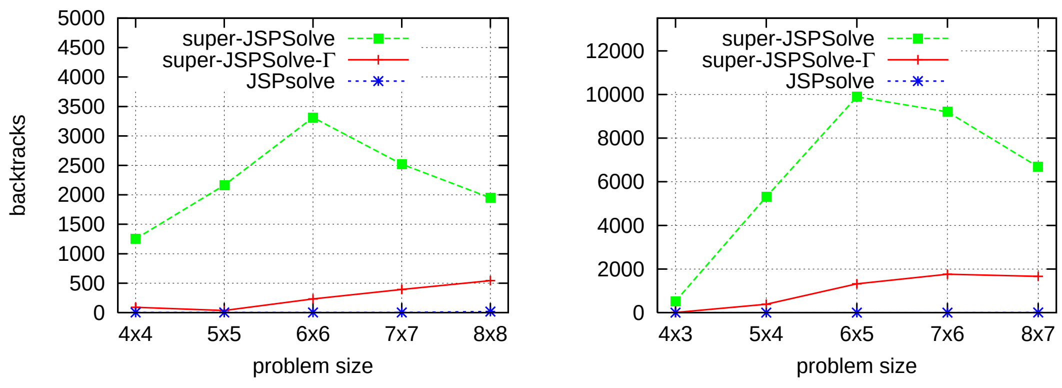


Figure 8.13: (1, 1)-SUPERJSP: Backtracks

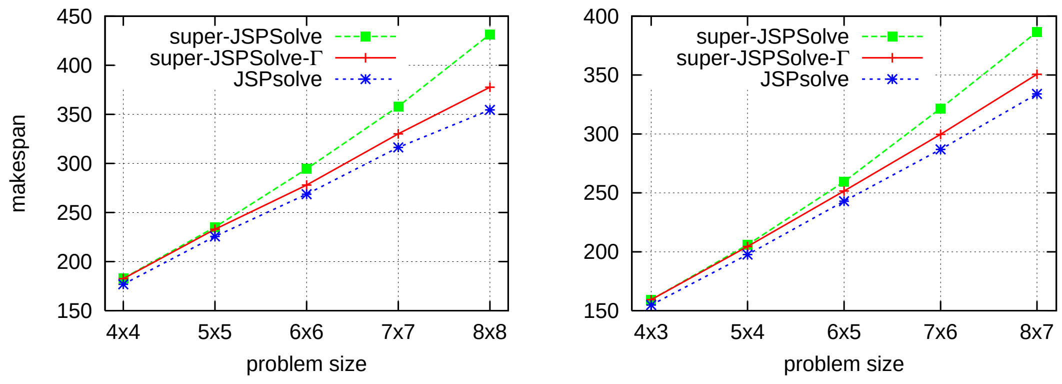


Figure 8.14: (1, 1)-SUPERJSP: Makespan Length

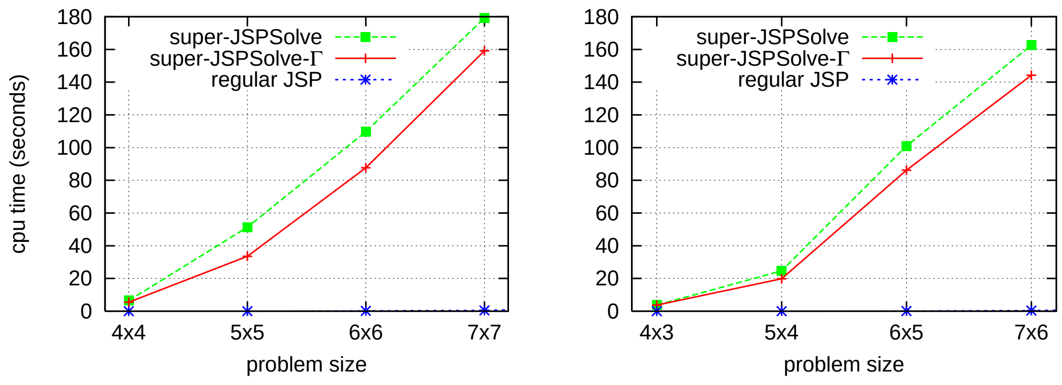


Figure 8.15: (1, 3)-SUPERJSP: Cpu-time (seconds)

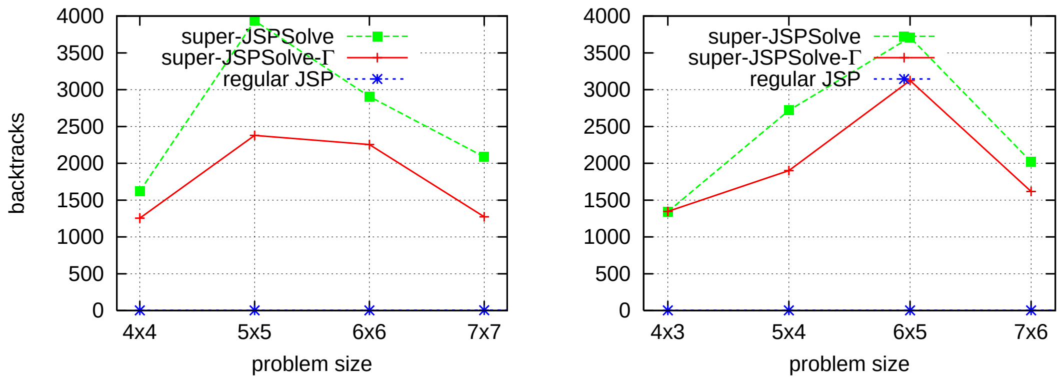


Figure 8.16: (1, 3)-SuperJSP: Backtracks

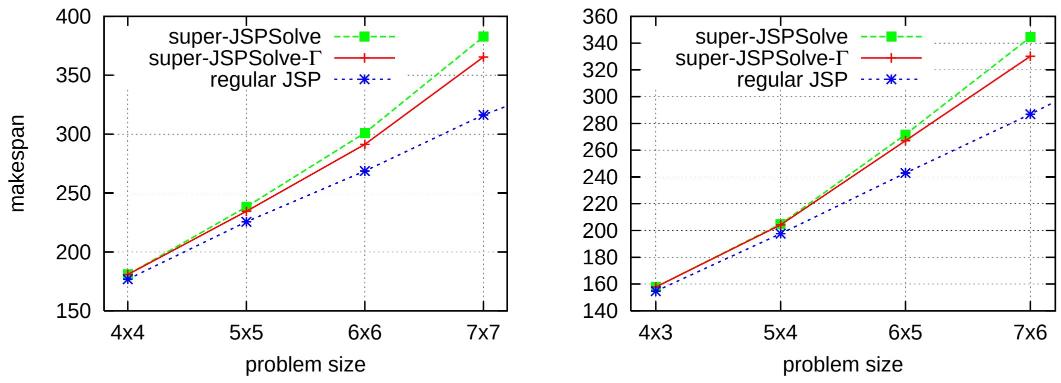


Figure 8.17: (1, 3)-SuperJSP: Makespan Length

schedules. The decrease in the number of backtracks (fig. 8.13) after a given size, is due to the time cutoff. The computational complexity of an inference step increases with the problem size, since the time cutoff is constant, the number of inference steps that can be done while solving a single problem decreases. Hence, as more and more resolutions go past the time cutoff, the average size of the search tree, hence the number of backtracks decreases.

The behaviour of *super*-`JSPsolve` and *super*-`JSPsolve`-Γ for finding $(1, 3)$-*super*-schedules with minimal makespan is similar, i.e., *super*-`JSPsolve`-Γ performs better. However, we observe that the difference is less marked. In fact since we look at a neighbourhood of size 3 instead of 1 it is not surprising that the inference is weaker.

8.5 Optimisation and Application:

Our purpose is twofold. First, we want to assess the gain from the inference method extended to the optimisation setting, i.e., that *super*-`JSPsolve`^{max}-Γ performs better than *super*-`JSPsolve`^{max}. Second, we want to quantify the tradeoff between time and robustness, that is, how fast can we improve the repairability of a solution. In the same line, we also want to evaluate the tradeoff between optimality and robustness, that is, how much decreasing the quality of a solution helps increasing its robustness.

8.5.1 Algorithms for (a, b)-MaxRepairCSP

We compare *super*-`JSPsolve`^{max}-Γ and *super*-`JSPsolve`^{max} on a range of jobshop scheduling problems.

8.5.1.1 Experimental Setting

We generated 4 samples of 250 instances of jobshob scheduling problems:

$$\langle 8 \text{ jobs}, \; 5 \text{ machines } (40 \text{ activities})\rangle \tag{8.23}$$
$$\langle 8 \text{ jobs}, \; 8 \text{ machines } (64 \text{ activities})\rangle \tag{8.24}$$
$$\langle 10 \text{ jobs}, \; 5 \text{ machines } (50 \text{ activities})\rangle \tag{8.25}$$
$$\langle 10 \text{ jobs}, 10 \text{ machines } (100 \text{ activities})\rangle \tag{8.26}$$

Every instance is first solved to optimality using `JSPsolve`. Then we create the deadline JSP by fixing the maximum makespan to 1.02 times the optimal makespan found by `JSPsolve`, i.e., the "tolerated" makespan is 2% longer than the minimum makespan. This deadline problem is passed to the procedures *super*-`JSPsolve`^{max} and *super*-`JSPsolve`^{max}-Γ for finding solutions with increasing $(1, 1)$-repairability. The time cutoff is set so that in most cases no improvement occurs after this limit. The sample 8.23 was solved with a cutoff of 30 seconds, samples 8.24 and 8.25 with a 60 seconds cutoff and finally sample 8.26 with a 120 seconds cutoff. The same instance is passed again as argument of *super*-`JSPsolve`^{max} and *super*-`JSPsolve`^{max}-Γ, however, to maximise $(1, 2)$-repairability. In this case the cutoff was set to 60 seconds for samples 8.23 and 8.24 and 120 seconds for samples 8.25 and 8.26.

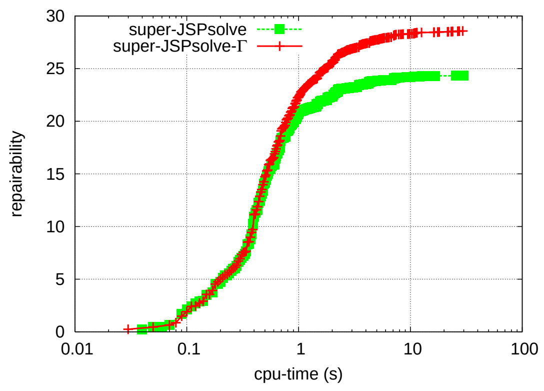
8.5.1.2 Experimental Results

In figure 8.18 to 8.21, we plot the average number of repairable variables over the instances of one class across cpu-time. More formally, for one or the other algorithm, let S_t be the set of solutions found at time t. This is by definition a sequence of solutions with increasing $(1, 1)$-repairability or $(1, 2)$-repairability. We define the monotonically increasing function $rep(t)$ that associates to every time point t, the (a, b)-repairability of the best (or last) solution found so far on a given instance, where $a = 1$ and $b \in \{1, 2\}$.

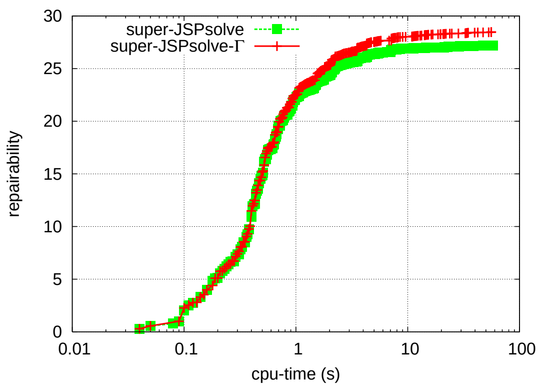
$$rep(t) = \begin{cases} 0 & if\ S_t = \emptyset \\ max\{(a,b)\text{−repairability}(f) \mid f \in S_t\} & otherwise \end{cases} \tag{8.27}$$

The curves plotted in Figure 8.18 stand for the average value of $rep(t)$ over all instances in sample 8.23. The left hand side graph is for $(1, 1)$-repairability whilst the right hand side is for $(1, 2)$-repairability. One curve is plotted for each algorithm, i.e., *super*-`JSPsolve`^{max} and *super*-`JSPsolve`^{max}-Γ. Similarly, Figure 8.19 shows the average of $rep(t)$ over sample 8.24, Figure 8.20 stands for sample 8.24 and Figure 8.21 for sample 8.26.

We observe that the algorithm using the neighbourhood-based inference, that is, *super*-`JSPsolve`^{max}-Γ, performs better than the brute-force algorithm *super*-`JSPsolve`^{max}. The procedure *super*-`JSPsolve`^{max} is clearly sub-optimal, that is, solutions with better repairability exist and *super*-`JSPsolve`^{max} does not converge quickly to these solutions

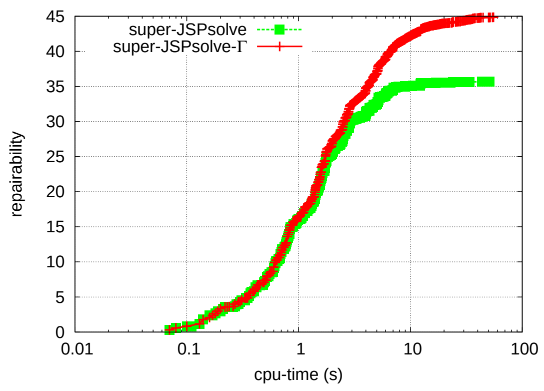


(a) 8 Jobs, 5 Machines, b = 1

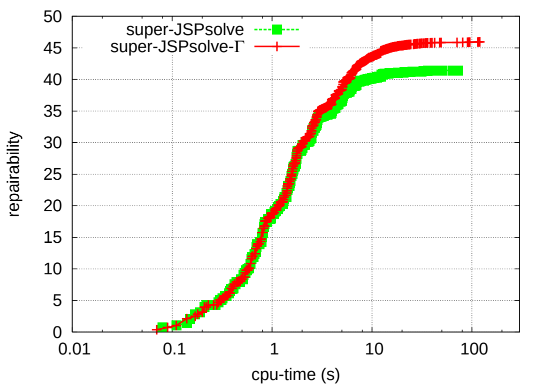


(b) 8 Jobs, 5 Machines, b = 2

Figure 8.18: The average repairability over time (8 jobs, 5 machines, 40 activities).

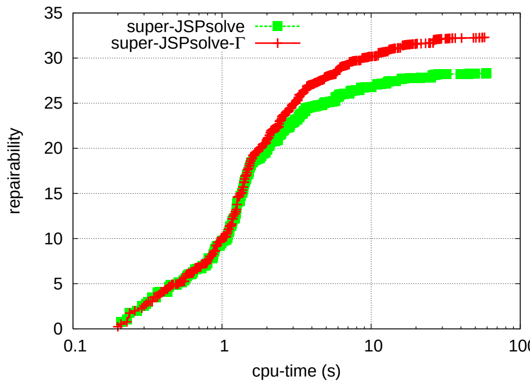


(a) 8 Jobs, 8 Machines, b = 1

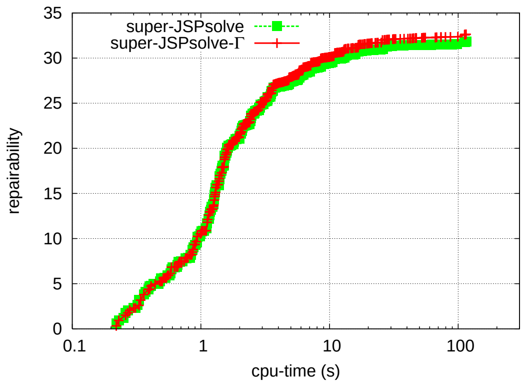


(b) 8 Jobs, 8 Machines, b = 2

Figure 8.19: The average repairability over time (8 jobs, 8 machines, 64 activities).

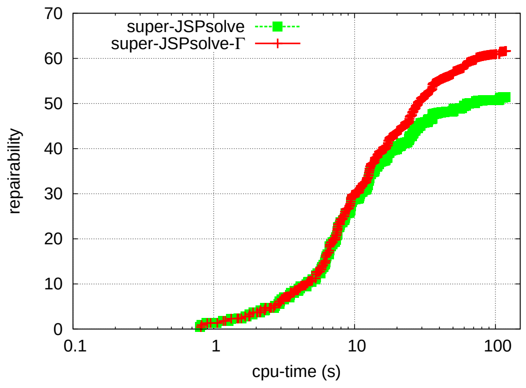


(a) 10 Jobs, 5 Machines, b = 1

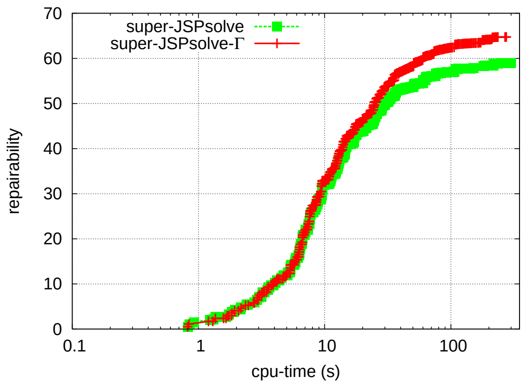


(b) 10 Jobs, 5 Machines, b = 2

Figure 8.20: The average repairability over time (10 jobs, 5 machines, 50 activities).



(a) 10 Jobs, 10 Machines, b = 1



(b) 10 Jobs, 10 Machines, b = 2

Figure 8.21: The average repairability over time (10 jobs, 10 machines, 100 activities).

with higher repairability. The same could be true for *super*-`JSPsolve`^{max}-Γ, however it is very difficult to assess as no other algorithm can be used for comparison. Nevertheless, these experiments show that the repairability of a solution can be increased by a significant amount over a reasonable period of time. This is a good result since this procedure does not affect the method used to produce the first solution, hence the computational effort until the first solution is found. Then we can run *super*-`JSPsolve`^{max}-Γ for as long as one is prepared to wait in order to increase repairability.

8.5.2 Tradeoff between Robustness and Optimality

We aim at quantifying the tradeoff between optimality and robustness. In order to do so, we first compare the makespan of optimal schedules and optimal $(1, 1)$-*super*-schedules. Then we compare the evolution of repairability over time when using *super*-`JSPsolve`^{max}-Γ with different values of tolerated makespan increase, from no increase at all to a 10% increase.

8.5.2.1 Experimental Setting

Trading Optimality for Repairability

We use the samples 8.17, defined in Section 8.3.2.1 for comparing `JSPsolve` to *super*-`JSPsolve`-Γ on JSP instances, with the same cutoff (30 seconds). The average makespan of schedules and $(1, 1)$-*super*-schedules are reported, and we look at the evolution of the discrepancy over increasing problem sizes.

Trading Optimality for Partial Repairability

We use a similar setting as in Section 8.5.1.1. We use only one set of 250 instances (sample 8.24 as defined in Section 8.5.1.1) and proceed in the same way. However, we use only the algorithm *super*-`JSPsolve`^{max}-Γ, and repeat the process for different values of makespan increase. Let mk be the optimal makespan, for a regular schedule, found with the algorithm `JSPsolve`. We search for the solution with maximal repairability of the instances in sample 8.24 with *super*-`JSPsolve`^{max}-Γ, where the the makespan is restricted to $mk + \Delta(mk)$. We use the following values for $\Delta(mk)$: 0%, 2%, 6% and 10%.

problem size	`JSPsolve` average makespan	*super*-`JSPsolve` average makespan	Δ makespan
4×4	176.848	182.566	0.032
5×5	225.561	232.898	0.032
6×6	268.7	277.99	0.034
7×7	316.23	330.18	0.044
8×8	354.45	377.61	0.065

Table 8.2: The makespan penalty for $(1, 1)$-*super*-schedules.

8.5.2.2 Experimental Results

Trading Optimality for Repairability

We give in table 8.2 and Figure 8.22 the average makespan found respectively by `JSPsolve` and *super*-`JSPsolve`-Γ of the instances in samples 8.17.

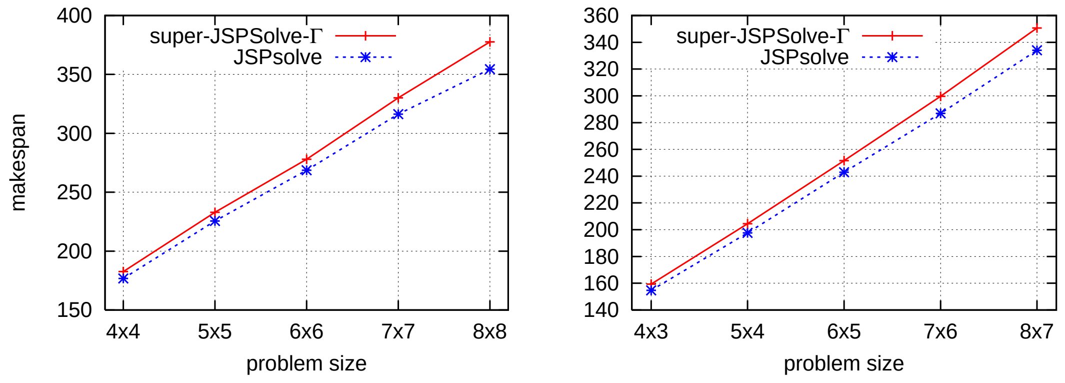


Figure 8.22: The makespan penalty for $(1, 1)$-*super*-schedules.

We observe that the makespan discrepancy between solutions and $(1, 1)$-*super*-solutions is constant ($\simeq 0.033$) across small problems and then increases for larger instances. This is in fact due to *super*-`JSPsolve`-Γ not being able to find the optimal $(1, 1)$-*super*-solution within the time cutoff.

Trading Optimality for Partial Repairability In figure 8.23, we plot the average number of repairable variables over the instances in sample 8.24 across cpu-time. Each curve stands for one value of tolerated makespan increase in $\{0\%, 2\%, 6\%, 10\%\}$.

As expected, the maximum repairability increases along with the tolerated makespan. However in some cases, better values of repairability are found faster for tighter makespan. In fact this is explained by the fact that when we increase the tolerance on the makespan,

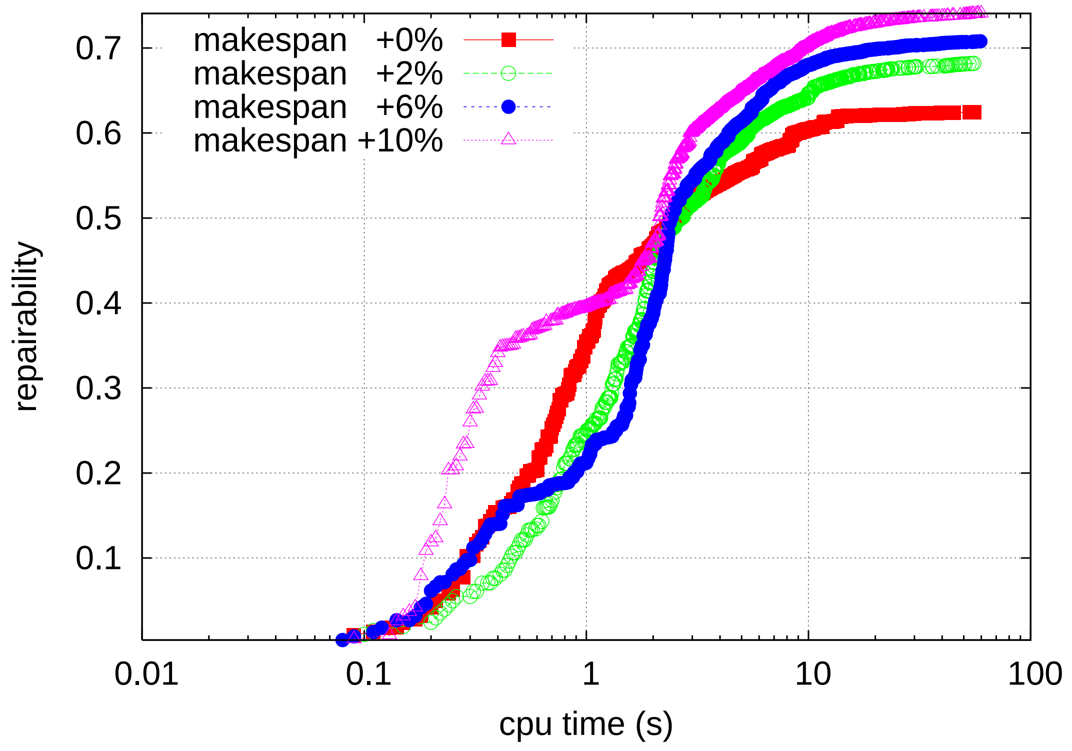


Figure 8.23: The tradeoff between $(1, 1)$-repairability and optimality.

the size of the search space increases dramatically. It is not rare, for 0 tolerance, i.e., when only solutions with optimal makespan are allowed, that *super*-`JSPsolve`^{max}-Γ was able to find the optimal and prove optimality within the time limit. This is never the case for large values of tolerance, since, intuitively, the problem is much more difficult. If we focus on the curve corresponding to a tolerance of 10%, we observe a "stair" shape. The first high gradient is due to the fact that the first solution and the surrounding first few improving solutions are found very quickly since the problem is loosely constrained. Then the next improvements of repairability are much harder to find as a large search space needs to be explored.

8.6 Summary and Limitations

In this section we addressed three issues:

Phase Transition and Computational Complexity: We were interested in empirically quantifying the computational complexity of finding *super*-solutions. We showed that, with current algorithms, finding *super*-solutions, for any size of breakage and repair, is considerably more difficult than finding regular solutions. This is true for random binary problems as well as for the jobshop scheduling problem, even though the difference is less significant for the latter.

We demonstrated that the problem of the existence of *super*-solutions has a phase transition phenomenon. Moreover, we can extend the κ formula to locate it on random binary CSPs. However, due to the relative inefficiency of the algorithms available to us, some exceptionally hard instances, in the loosely constrained instances dominate the computational complexity of this problem. The hardest instances, when b is strictly positive, are not at the cross-over point between underconstrained and overconstrained instances, but rather within the underconstrained region.

Algorithms Comparison: We were interested in evaluating the significance of the various inference methods introduced in this dissertation. The best algorithm for finding $(1, 0)$-*super*-solutions is *super*-`MAC`, as the theoretical comparison was suggesting. The inference method for the algorithm `decompose-backtrack`-Γ consisting in identifying equality constraints by using the neighbourhood relation and the preprocessing of the constraint controlling the repairs bring a significant gain in efficiency. This gain was clearly demonstrated for both random binary problems and the jobshop scheduling problem. However, this methods becomes less and less significant when the value of b increases.

Optimisation and Application: We demonstrated that *super*-`JSPsolve`^{max}-Γ is an effective method to increase the robustness of a solution. We showed that the repairability could be increased, for some jobshop scheduling instances, in a reasonable amount of time, and without sacrificing too much schedule quality, i.e., without significantly increasing its duration.

An empirical evaluation is by essence limited to the chosen testbenches, to the chosen experimental designs and to the implementation among other things. Moreover this set of experiments was limited by the fact that the framework is relatively new, and that almost no references exist for comparison. This set of experiments could be improved and extended in several dimensions. A richer variety of problems could be explored and we could have had a closer look at some aspects of the algorithms. However, there is one experimental question that we chose not to cover mainly because we believe we need a real world application for doing so. The question is the extent of the on-line benefit, with respect to the investment off-line when breakages occur while executing. The “benefit” can be either in computational cost to compute an alternative solution, or loss in optimality or in the amount of perturbation from the basic schedule and the alternative.

Chapter 9

Conclusion and Future Work

In this dissertation, we generalised to constraint satisfaction and optimisation the concept of fault tolerant solution, introduced by Roy *et al.* [Ginsberg 98, Roy 98] for Boolean satisfiability. The motivation behind this framework is that stability, that is, the property of having a close alternative in the case of a change, is a valuable property in a number of applications. Fault tolerant solutions are stable, hence robust, since they guarantee small repairs in response to small changes.

We analysed the computational complexity and proposed a number of algorithms for finding fault tolerant solutions. We extended the framework and made it more useful. Finally, we experimentally assessed the introduced algorithms. We showed that we can effectively and efficiently use these algorithms to increase solution robustness at the expense of a chosen increase in computational time and little, if any, decrease in solution quality. We recall the evidences supporting our thesis and summarise the contributions made in this dissertation in Section 9.1, then we discuss the limitations of our work in Section 9.2 before outlining some possible directions for future work in Section 9.3 and concluding in Section 9.4.

9.1 Contributions

The thesis defended in this dissertation includes three main points. We first claimed that the concept of fault tolerance could be extended in a number of directions in order to provide robust solutions to constraint satisfaction and optimisation problems. We then claimed that finding fault tolerant solutions was significantly more difficult than finding regular solutions. And finally, we claimed that efficient and effective algorithms could be designed for that purpose. We go through these claims and measure the progress made toward assessing them in this dissertation.

9.1.1 Extending the Framework

We extended the fault tolerance framework in a number of directions. First, we generalised the definition of fault tolerant solutions to constraint satisfaction and optimisation. Although both NP-complete, the constraint satisfaction problem is significantly richer than the Boolean satisfiability problem, since more complex relations and construct such as sets, multisets, or even functions and graphs can be expressed. We defined the notion of existential and universal *super*-solution. Whereas the former guarantees that any breakage of a given size accepts a repair, the latter ensures that all the alternative assignments, here again for breakages of given size, are valid.

We defined the notion of fault tolerant solution within optimisation problems. We extended the theoretical framework in this direction by studying the complexity of several problems related to finding fault tolerant solutions while optimising their objective value.

We extended the framework toward being more practically useful by introducing the concept of partial fault tolerance. In constraint satisfaction problems, when no solution can be found, the constraints are often relaxed, and the solution with least relaxation is searched. The same reasoning can be extended to maximising the robustness of a solution. Being able to relax the robustness condition is valuable since fault tolerant solutions do not always exist, and moreover, achieving robustness is often not the primary goal. The notion of repairability, that is, the number of breakages admitting a repair, seemed empirically particularly promising. We studied a Branch & Bound procedure for improving the repairability of a solution, which is arguably the most practical aspect of this framework. We used the Jobshop Scheduling Problem to assess the tradeoff between computational effort and robustness as well as solution quality against robustness.

Last, we extended the framework by allowing more complex definition of breakages and repairs. Often, on structured problems, a breakage can be more complex than the simple loss of the current value for a variable. Similarly, the notion of allowed repair may be more subtle than a simple count of discrepancies. We show that the general algorithm, and its extension to repairability maximisation, can handle far more complex models of robustness. In fact, any definition of breakage and repair that corresponds to a constraint can be dealt with. We therefore introduced the concept of a constraint for controlling the breakages, and a constraint for controlling the repairs. We showed that if the constraint controlling the breakages is stronger than the set of disequalities corresponding to the classical definition, and if the definition of a repair is unchanged, then the inference methods introduced in this dissertation to deal with the regular definition of *super*-solutions can be used soundly.

9.1.2 Computational Cost

Several pieces of evidence were put forward to show that the problem of finding fault tolerant solutions is significantly more difficult than finding regular solutions.

First, we analysed the computational complexity of several problems related to *super*-solutions. Besides the problem of the existence of existential or universal *super*-solutions, we also covered the case of optimising the objective value of a *super*-solution and optimising the repairability of a solution. When the parameter a is fixed, these problems are in the class NP (or NP optimisation) and therefore not harder, in the worst case, than regular CSPs. However, we proved that for several well known tractable classes of CSPs, finding fault tolerant solutions is NP-hard. We showed that the converse may be true, that is, finding fault tolerant solutions may be easier than finding regular solutions. However, no significant classes of constraint network have been shown to have this property.

We then empirically investigated the increase in computational cost of finding fault tolerant solutions over regular solutions on two classes of constraint problems. Although these experiments actually compare the behaviour of algorithms and not directly intrinsic properties of the problems themselves, we believe that the large gap in computational cost is a reflection of the difference in hardness to solve these problems in practice.

9.1.3 Designing Algorithms

We introduced three solution methods for finding full fault tolerant solutions, that is, solutions where no repair at all is allowed in response to a breakage. We showed that full fault tolerant solutions can be characterised through a local consistency property, in the same way that regular solutions can be defined as full assignments locally satisfying every constraint. We defined two such local consistencies, and developed algorithms upon them. We theoretically compared these algorithms with two reformulation methods, one of which was introduced in 98 by Weigel and Bliek [Weigel 98]. The strongest consistency method is to apply arc consistency on a novel reformulation. However, it comes at a higher worst case computational complexity. On the other hand, we introduced an optimal closure algorithm with the same complexity as arc consistency, whilst being stronger than all other methods. This theoretical analysis was then completed and confirmed by an experimental comparison.

In the general case where the allowed repair size is strictly positive, it is no longer easy to check the repairability of a solution locally. We introduced an algorithm for the general case, where the size of the breakages and repairs are not fixed. This algorithm backtracks in a classical search tree and the partial solution constructed in this "master problem" represents the *super*-solution. At each node in the search tree, and for each breakage, a "sub-problem" is created. This problem is such that a partial solution of same size is in fact a repair. The main idea for pruning the search tree is to identify variables that must be assigned equally in the master problem and in a given sub-problem. Since at most $a + b$ variables can differs in a repair, then at least $n - (a + b)$ must agree. Moreover, we consider the constraint graph, where nodes stand for variables and edges for constraints. We show that in the solutions with minimal change, the variables that are reassigned must form a connected sub-graph adjacent to the breakage. Therefore, any variable, such that the minimal path linking this variable to the breakage is longer than b, must be assigned equally. Then, we show that this reasoning can be refined while preprocessing the constraint controlling the discrepancy with the main solution. When, for a variable involved in an equality constraint, some pruning occurs in the sub-problem while preprocessing, the same pruning can be done in the master problem. This, in consequence, reduces the search space of the master problem, hence the number of sub-problems that need be solved. The same reasoning, based on equality constrains and the neighbourhood relation in the constraint graph, can be used, with a slight adaptation, when solving partial fault tolerance problems. This inference method was then empirically assessed.

Finally we showed that symmetry breaking can be used to help finding fault tolerant solutions. Symmetry is a concept both theoretically and practically very important. It is particularly significant for a number of real-world applications, since symmetries are common in these problems, and taking advantage of the symmetries of a problem may dramatically reduce the computational cost. However, not all symmetries preserve *super*-solutions. We showed that distributivity, that is, the fact that the conjunction of two symmetric parts is equal to the symmetric image of the conjunction, is a sufficient condition for preserving *super*-solutions.

9.2 Limitations

We discuss the limitations of our approach to uncertainty. A first significant limitation is in the fact that we chose to tackle this problem from the perspective of backtracking algorithms only. It may possibly be the case that local search algorithms or algebraic approaches such as bucket elimination are better suited for tackling this problem. The best approach is likely to depend on the application domain. We summarise the issues left open in this dissertation.

Complexity: Several questions remain open. For instance, the complexity of finding (a, b)-*super*-solutions when a is not fixed, the complexity of finding $(1, 1)$-*super*-solutions of constraint network with fixed treewidth, or the complexity of finding $(1, 0)$-*super*-solutions and $(1, 1)$-*super*-solutions on constraint languages closed under a majority operation are all open. Moreover, we proved NP-completeness for numerous problems when the parameter a is equal to 1. However the complexity of the same problems for higher values of a is not known.

Algorithms We restricted most of our analysis to algorithms for finding $(1, 0)$-*super*-solutions or partial $(1, 0)$-*super*-solutions of binary constraint networks. Some algorithms may be easily extended to non-binary constraints. However this is not straightforward in particular for the notion of *super*-GAC and the $\mathcal{P} \times \mathcal{P}$ reformulation. The main limitation of the general algorithm may be the fact that a large part of the inference method checks earlier decisions. We have seen that when tightening the definition of a breakage it is possible to check "future" breakages. Moreover, we have seen empirically, that in our model for finding *super*-solutions for the jobshop scheduling problem, where breakages were delays, that the ability to check future breakages seemed indeed to reduce the computational cost for finding *super*-solutions.

Experiments In our experiments we tackled three issues. However, this left many other legitimate questions unanswered. Arguably the most important question that we did not address is whether spending more time off-line to compute a *super*-solution really pays off when deploying it. This is in fact a very difficult question to answer since many assumption need to be made. For instance, what is a typical breakage for the chosen problem, and how frequently do they happen? How should the various parameters (a, b, $\mathcal{BS}$, $\mathcal{RS}$, $\mathcal{FS}$, C_{break}, C_{repair}, etc.) be set? How do we measure the benefit when deploying the solution?

9.3 Future Work

We identify two general directions of research that we did not follow in this dissertation and that could be addressed in future work.

Super Consistency for Large Arity Constraints: In Chapter 4, we briefly investigated propagation algorithms for *super*-GAC on global constraints. However, there are clearly many global constraints for which it would be interesting to come up with an algorithm or an NP-completeness proof in this context. Moreover, for $a, b \geq 1$, the inference based on equality constraints and neighbourhood relation can be seen as a general reasoning framework that could be applied to propagation algorithm for global constraints for this problem. There are in fact similarities with extending global constraints to soft global constraints.

Another direction of research is to extend *super*-consistencies (i.e., GAC+, *super*-GAC and multiconsistency) to constraints of larger arity. Alternatively, these consistencies could be extended in the same way (i, j)-consistency [Freuder 85] or relational consistencies [Dechter 96] extend arc consistency.

Variables and Values Ordering: We did not addressed the problem of search heuristics. In the experimental result chapter, we used respectively `domain/degree` and `Operation Resource Reliance` as dynamic variable ordering for random binary and jobshop scheduling instances. We used in both cases a lexicographic value ordering. In fact, it is likely that the algorithm we introduced, although closely related to the `MAC` algorithm, differs significantly enough so that a good heuristic for `MAC` is not necessarily good for `decompose-backtrack-`Γ. We observed, when using the PCP model for the jobshop scheduling problem, that `decompose-backtrack-`Γ was not responding well, and we conjectured that it may be because the variables that can break and repair are searched last. Branching on the most brittle variable first, or at least weighting the heuristics so that brittleness is taken into account may be a first step toward a variable ordering heuristic adapted to finding *super*-solutions.

9.4 Conclusion

The thesis defended in this dissertation is that:

> *The concept of fault tolerance can be extended in a number of directions to provide robust solutions to constraint satisfaction and optimisation problems. Finding fault tolerant solutions is significantly more difficult than finding regular solutions. However, efficient and effective algorithms can be designed for that purpose.*

We extended the fault tolerance framework to constraint satisfaction and optimisation. We also extended it by allowing richer definitions of breakages and repairs. Although the worst case time complexity is not in general higher than that of the regular constraint satisfaction problem our complexity analysis suggests that finding *super*-solutions is significantly more difficult. Indeed, we proved, for several classes of constraint networks, that deciding if a solution exists is tractable whilst deciding if a *super*-solution exists is NP-complete. However, we developed a number of inference methods for finding both full and weak fault tolerant solutions. Moreover, we extended these inference methods to an effective and efficient Branch & Bound procedure to improve the stability, or *repairability*, of a solution. This method empirically showed considerable promise. Since the search takes as starting point any ordinary solution, it can be added without overhead, and if we are prepared to wait longer, we can have a more robust solution.

Bibliography

[Backofen 99] R. Backofen & S. Will. Excluding Symmetries in Constraint-Based Search. In Joxan Jaffar, editor, *Proceedings of the 5th International Conference on Principles and Practice of Constraint Programming (CP-99)*, volume 1713 of Lecture Notes in Computer Science, pages 73–87, Alexandria, VA, USA, 1999. Springer-Verlag.

[Barták 03] R. Barták, T. Müller & H. Rudová. Minimal Perturbation Problem - A Formal View. In *Proceedings of the Joint Workshop of the ERCIM Working Group on Constraints and of the CologNet area on Constraint and Logic Programming*, Budapest, Hungary, 2003.

[Bessiere 91] C. Bessiere. Arc-Consistency in Dynamic Constraint Satisfaction Problems. In Thomas L. Dean & Kathleen McKrown, editors, *Proceedings of the 9th National Conference on Artificial Intelligence (AAAI-91)*, pages 221–226, Anaheim, CA, USA, 1991. AAAI Press / The MIT Press.

[Bessiere 96] C. Bessiere & J.C. Régin. MAC and Combined Heuristics: two reasons to forsake FC (and CBJ?) on hard problems. In Eugene C. Freuder, editor, *Proceedings of the 2nd International Conference on Principles and Practice of Constraint Programming (CP-96)*, Lecture Notes in Computer Science, pages 61–75, Cambridge, MA, USA, 1996. Springer-Verlag.

[Bessiere 97] C. Bessiere & J.C. Régin. Arc Consistency for General Constraint Networks: preliminary results. In Martha E. Pollack, editor, *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI-97)*, pages 398–404, Nagoya, Japan, 1997. Morgan Kaufmann.

[Bessiere 03] C. Bessiere & P. van Hentenryck. To be or not to be...a global constraint. In Francesca Rossi, editor, *Proceedings of the 9th International Conference on Principles and Practice of Constraint Programming (CP-03)*, volume 2833 of Lecture Notes in Computer Science, pages 789–794, Kinsale, Ireland, 2003. Springer-Verlag.

[Bessiere 05] C. Bessiere, J.C. Régin, R.H.C. Yap & Y. Zhang. An Optimal Coarse-grained Arc Consistency Algorithm. Artificial Intelligence, vol. 1656, no. 2, pages 165–185, 2005.

[Bistarelli 95] S. Bistarelli, U. Montanari & F. Rossi. Constraint Solving over Semirings. In Chris S. Mellish, editor, *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI-95)*, pages 624–630, Montral, Canada, 1995. Morgan Kaufmann.

[Bistarelli 99] S. Bistarelli, U. Montanari, F. Rossi, T. Schiex, G. Verfaillie & H. Fargier. Semiring-Based CSPs and Valued CSPs: Framework, Properties, and Comparison. Constraints, vol. 4, no. 3, pages 199–240, 1999.

[Brown 88] C.A. Brown, L. Finkelstein & P.W. Purdom Jr. Backtrack Searching in the Presence of Symmetry. In *Proceedings of the 6th International Conference on Applied Algebra, Algebraic Algorithms and Error-Correcting Codes (AAECC-88)*, pages 99–110, Rome, Italy, 1988.

[Bulatov 02] A.A. Bulatov. A Dichotomy Theorem for Constraints on a Three-element Set. In *Proceedings of 43rd IEEE Symposium on Foundations of Computer Science (FOCS-02)*, pages 649–658, Vancouver, Canada, 2002.

[Carlier 94] J. Carlier & E. Pinson. Adjustment of Heads and Tails for the Job-Shop Problem. European J ournal of Operational Research, vol. 78, pages 146–161, 1994.

[Cheeseman 91] P. Cheeseman, B. Kanefsky & B.M. Taylor. Where the Really Hard Problems Are. In John Mylopoulos & Raymond Reiter, editors, *Proceedings of the 12th International Joint Conference on Artificial Intelligence (IJCAI-91)*, pages 163–169, Sydney, Australia, 1991. Morgan Kaufmann.

[Cheng 94] C. Cheng & S.F. Smith. Generating Feasible Schedule under Complex Metric Constraints. In Barbara Hayes-Roth & Richard E. Korf, editors, *Proceedings of the 12th National Conference on Artificial Intelligence (AAAI-94)*, volume 2, pages 1086–1091, Seattle, WA, USA, 1994. AAAI Press.

[Cheng 95] C. Cheng & S.F. Smith. A Constraint-Posting Framework for Scheduling Under Complex Constraints. In *Proceedings of the Joint IEEE/INRIA Conference on Emerging Technologies for Factory Automation*, pages 269–280, Paris, France, 1995.

[Cohen 05] D. Cohen, P. Jeavons, C. Jefferson, K.E. Petrie & B.M. Smith. Symmetry Definitions for Constraint Satisfaction Problems. Constraints, vol. 11, no. 2–3, pages 17–31, 2005.

[Cooper 94] M.C. Cooper, D.A. Cohen & P.G. Jeavons. Characterizing Tractable Constraints. Artificial Intelligence, vol. 65, pages 347–361, 1994.

[Crawford 96] J. Crawford, M. Ginsberg, E. Luks & A. Roy. Symmetry Breaking Predicates for Search Problems. In *Proceedings of the 5th International Conference on the Principles of Knowledge Representation* *and Reasoning (KR-96)*, pages 148–159, Cambridge, MA, USA, 1996.

[Davenport 00] A.J. Davenport & C.J. Beck. A Survey of Techniques for Scheduling with Uncertainty. (*unpublished manuscript*, available at http://tidel.mie.utoronto.ca/publications.php), 2000.

[Debruyne 97] R. Debruyne & C. Bessiere. Some Practicable Filtering Techniques for the Constraint Satisfaction Problem. In Martha E. Pollack, editor, *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI-97)*, pages 412–417, Nagoya, Japan, 1997. Morgan Kaufmann.

[Dechter 88] A. Dechter & R. Dechter. Belief Maintenance in Dynamic Constraint Networks. In Tom Mitchell & Reid Smith, editors, *Proceedings of the 7th National Conference on Artificial Intelligence (AAAI-88)*, pages 37–42, St Paul, MN, USA, 1988. AAAI Press / The MIT Press.

[Dechter 96] R. Dechter & P. van Beek. Local and Global Relational Consistency. Journal of Theoretical Computer Science, vol. 173, no. 1, pages 283–308, 1996.

[Dubois 93] D. Dubois, H. Fargier & H. Prade. The Calculus of Fuzzy Restrictions as a Basis for Flexible Constraint Satisfaction. In *Proceedings of the 2nd IEEE International Conference on Fuzzy Systems (FUZZ-IEEE-93)*, pages 1131–1136, San Francisco, CA, USA, 1993. IEEE.

[Elbassioni 05] K. Elbassioni & I. Katriel. Multiconsistency and Robustness with Global Constraints. In Roman Barták & Michela Milano, editors, *Proceedings of the 7th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR-05)*, volume 3524 of Lecture Notes in Computer Science, pages 168–182, Prague, Czech Republic, 2005. Springer-Verlag.

[Emerson 93] E.A. Emerson & A.A. Sistla. Symmetry and Model Checking. In *Proceedings of the 5th International Conference on Computer Aided Verification (CAV-93)*, pages 463–478, Berlin, Heidelberg, 1993.

[Fable 01] T. Fable, S. Schamberger & M. Sellmann. Symmetry Breaking. In Toby Walsh, editor, *Proceedings of the 7th International Conference on Principles and Practice of Constraint Programming (CP-01)*, volume 2239 of Lecture Notes in Computer Science, pages 93–107, Paphos, Cyprus, 2001. Springer-Verlag.

[Faltings 02] B. Faltings & S. Macho-Gonzalez. Open Constraint Satisfaction. In Toby Walsh, editor, *Proceedings of the 8th International Conference on Principles and Practice of Constraint Programming (CP-02)*, volume 2470 of Lecture Notes in Computer Science, pages 356–370, Ithaca, NY, USA, 2002. Springer-Verlag.

[Fargier 93] H. Fargier & J. Lang. Uncertainty in Constraint Satisfaction Problems: a Probabilistic Approach. In *Proceedings of the 2nd European Conference on Symbolic and Quantitative Approaches to Reasoning and Uncertainty*, pages 97–104, Granada, Spain, 1993.

[Fargier 96] H. Fargier, J. Lang & T. Schiex. Mixed Constraint Satisfaction: a Framework for Decision Problems under Incomplete Knowledge. In William J. Clancey & Dan Weld, editors, *Proceedings of the 13th National Conference on Artificial Intelligence and the Eighth Conference on Innovative Applications of Artificial Intelligence (AAAI-96 / IAAI-96)*, pages 175–180, Portland, OR, USA, 1996. AAAI Press / The MIT Press.

[Flener 02] P. Flener, A. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, J. Pearson & T. Walsh. Breaking Row and Column Symmetries in Matrix Models. In Pascal van Hentenryck, editor, *Proceedings of the 8th International Conference on Principles and Practice of Constraint Programming (CP-02)*, volume 2470 of Lecture Notes in Computer Science, pages 462–476, Ithaca, NY, USA, 2002. Springer-Verlag.

[Focacci 01] F. Focacci & M. Milano. Global Cut Framework for Removing Symmetries. In Toby Walsh, editor, *Proceedings of the 7th International Conference on Principles and Practice of Constraint Programming (CP-01)*, volume 2239 of Lecture Notes in Computer Science, pages 77–92, Paphos, Cyprus, 2001. Springer-Verlag.

[Fowler 00] D.W. Fowler & K.N. Brown. Branching Constraint Satisfaction Problems for Solutions Robust under Likely Changes. In Rina Dechter, editor, *Proceedings of the 6th International Conference on Principles and Practice of Constraint Programming (CP-00)*, volume 1894 of Lecture Notes in Computer Science, pages 500–504, Singapore, 2000. Springer-Verlag.

[Fowler 03] D. Fowler & K. Brown. Branching Constraint Satisfaction Problems and Markov Decision Problems Compared. Annals of Operations Research, vol. 118, no. 1–4, pages 85–100, 2003.

[Fox 99] M. Fox & D. Long. The Detection and Exploitation of Symmetry in Planning Problems. In Thomas Dean, editor, *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI-99)*, pages 956–961, Stockholm, Sweden, 1999. Morgan Kaufmann.

[Freuder 82] E.C. Freuder. A Sufficient Condition for Backtrack-free Search. Journal of the ACM, vol. 29, pages 24–32, 1982.

[Freuder 85] E.C. Freuder. A Sufficient Condition for Backtrack-bounded Search. Journal of the ACM, vol. 32, pages 755–761, 1985.

[Freuder 89] E.C. Freuder. Partial Constraint Satisfaction. In N. S. Sridharan, editor, *Proceedings of the 11th International Joint Conference* *on Artificial Intelligence (IJCAI-89)*, pages 278–283, Detroit, MI, USA, 1989. Morgan Kaufmann.

[Freuder 91] E.C. Freuder. Eliminating Interchangeable Values in Constraint Satisfaction Problems. In Thomas L. Dean & Kathleen McKrown, editors, *Proceedings of the 9th National Conference on Artificial Intelligence (AAAI-91)*, pages 227–233, Anaheim, CA, USA, 1991. AAAI Press / The MIT Press.

[Freuder 92] E.C. Freuder & R.J. Wallace. Partial Constraint Satisfaction. Artificial Intelligence, vol. 58, no. 1–3, pages 21–70, 1992.

[Frost 96] D. Frost, C. Bessiere, R. Dechter & J.C. Régin. Random Uniform CSP Generator. URL: http://www.lirmm.fr/-bessiere/generator.html, 1996.

[Gaschnig 74] J. Gaschnig. A Constraint Satisfaction Method for Inference Making. In *Proceedings of the 12th Annual Allerton Conference on Circuit and System Theory*, University of Illinois, Urbana-Champaign, USA, 1974.

[Gaschnig 79] J. Gaschnig. Performance Measurement and Analysis of certain Search Algorithms. PhD thesis, Carnegie-Mellon University, 1979.

[Gent 94] I.P. Gent & T. Walsh. Easy Problems are Sometimes Hard. Artificial Intelligence, vol. 70, pages 335–345, 1994.

[Gent 96a] I.P. Gent, E. MacIntyre, P. Prosser & T. Walsh. The Constrainedness of Search. In William J. Clancey & Dan Weld, editors, *Proceedings of the 13th National Conference on Artificial Intelligence and the Eighth Conference on Innovative Applications of Artificial Intelligence (AAAI-96 / IAAI-96)*, pages 246–252, Portland, OR, USA, 1996. AAAI Press / The MIT Press.

[Gent 96b] I.P. Gent & T. Walsh. The TSP Phase Transition. Artificial Intelligence, vol. 88, no. 1–2, pages 349–358, 1996.

[Gent 02] I.P. Gent & B.M. Smith. Symmetry Breaking in Constraint Programming. In Frank van Harmelen, editor, *Proceedings of the 15th European Conference on Artificial Intelligence (ECAI-02)*, pages 599–603, Lyons, France, 2002. IOS Press.

[Ginsberg 98] M. Ginsberg, A. Parkes & A. Roy. Supermodels and Robustness. In Jack Mostow & Charles Rich, editors, *Proceedings of the 15th National Conference on Artificial Intelligence and the Tenth Conference on Innovative Applications of Artificial Intelligence (AAAI-98 / IAAI-98)*, pages 334–339, Madison, WI, USA, 1998. AAAI Press / The MIT Press.

[Gyssens 94] M. Gyssens, P.G. Jeavons & D.A. Cohen. Decomposing Constraint Satisfaction Problems Using Database Techniques. Artificial Intelligence, vol. 66, no. 1, pages 57–89, 1994.

[Hebrard 04] E. Hebrard, B. Hnich & T. Walsh. Robust Solutions for Constraint Satisfaction and Optimization. In Ramon López de Mntaras & Lorenza Saitta, editors, *Proceedings of the 16th European Conference on Artificial Intelligence (ECAI-04)*, pages 186–190, Valencia, Spain, 2004. IOS Press.

[Hebrard 05] E. Hebrard. halcsp, a simple and fast constraint solver. URL: http://www.cse.unsw.edu.au/-ehebrard/codef.htm, 2005.

[Jeavons 97] P. Jeavons, D. Cohen & M. Gyssens. Closure Properties of Constraints. Journal of the ACM, vol. 44, no. 4, pages 527–548, 1997.

[Karousis 93] L. Karousis. Fast Parallel Constraint Satisfaction. Artificial Intelligence, vol. 64, pages 147–160, 1993.

[Kirkpatrick 94] S. Kirkpatrick & B. Selman. Critical Behavior in the Satisfiability of Random Boolean Expressions. Science, vol. 264, pages 1297–1301, 1994.

[Kiziltan 04] Z. Kiziltan. Symmetry Breaking Ordering Constraints. PhD thesis, Uppsala University, 2004.

[Knuth 04] D. Knuth. The Art of Computer Programming: Pre-Fascicle 3a: Generating all Combinations. http://www-cs-faculty.stanford.edu/-knuth/fasc3a.ps.gz, 2004.

[Lamma 99] E. Lamma, P. Mello, M. Milano, R. Cucchiara, M. Gavanelli & M. Piccardi. Constraint Propagation and Value Acquisition: Why we should do it Interactively. In Thomas Dean, editor, *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI-99)*, pages 468–477, Stockholm, Sweden, 1999. Morgan Kaufmann.

[Littman 01] M. Littman, S. Majercik & T. Pitassi. Stochastic Boolean Satisfiability. Journal of Automated Reasoning, vol. 27, no. 3, pages 251–296, 2001.

[Mackworth 77] A.K. Mackworth. Consistency in Networks of Relations. Artificial Intelligence, vol. 8, no. 1, pages 99–118, 1977.

[Mackworth 85] A.K. Mackworth & E.C. Freuder. The Complexity of Some Polynomial Network Consistency Algorithms for Constraint Satisfaction Problems. Artificial Intelligence, vol. 25, no. 1, pages 65–74, 1985.

[Manandhar 03] S. Manandhar, A. Tarim & T. Walsh. Scenario-based Stochastic Constraint Programming. In Georg Gottlob & Toby Walsh, editors, *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI-03)*, pages 257–262. Morgan Kaufmann, 2003.

[Martin 96] D. Martin & P. Shmoys. A Time-based Approach to the Job-Shop Problem. In *Proceedings of the 5th International Conference on Integer Programming and Combinatorial Optimization (IPCO-96)*, Vancouver, Canada, 1996. Springer-Verlag.

[Miguel 01] I. Miguel. Dynamic Flexible Constraint Satisfaction and Its Application to AI Planning. PhD thesis, University of Edinburgh, 2001.

[Miguel 03] I. Miguel & Q. Shen. Fuzzy rrDFCSP and Planning. Artificial Intelligence, vol. 148, no. 1–2, pages 11–52, 2003.

[Mitchell 92] D. Mitchell, B. Selman & H. Lavesque. Hard and Easy Distribution of SAT Problems. In Paul Rosenbloom & Peter Szolovits, editors, *Proceedings of the 10th National Conference on Artificial Intelligence (AAAI-92)*, pages 459–465, Menlo Park, CA, USA, 1992. AAAI Press / The MIT Press.

[Mohr 86] R. Mohr & T.C. Henderson. Arc and Path Consistency Revisited. Artificial Intelligence, vol. 28, pages 225–233, 1986.

[Montanari 91] U. Montanari & F. Rossi. Constraint Relaxation may be Perfect. Artificial Intelligence, vol. 48, no. 2, pages 143–170, 1991.

[Nuijten 94a] W. Nuijten. Time and Resource Constraint Scheduling: A Constraint Satisfaction Approach. PhD thesis, Eindhoven University of Technology, 1994.

[Nuijten 94b] W.P.M. Nuijten & E.H.L. Aarts. Constraint Satisfaction for Multiple Capacitated Job Shop Scheduling. In Anthony G. Cohn, editor, *Proceedings of the 11th European Conference on Artificial Intelligence (ECAI-94)*, pages 635–639, Amsterdam, The Netherlands, 1994. John Wiley and Sons, Chichester.

[Papadimitriou 94] C.H. Papadimitriou. Computational Complexity. Addison-Wesley, 1994.

[Prosser 94] P. Prosser. Binary Constraint Satisfaction Problems: some are harder than others. In Anthony G. Cohn, editor, *Proceedings of the 11th European Conference on Artificial Intelligence (ECAI-94)*, pages 95–99, Amsterdam, The Netherlands, 1994. John Wiley and Sons, Chichester.

[Prosser 96] P. Prosser. An Empirical Study of the Phase Transition in Binary Constraint Satisfaction Problems. Artificial Intelligence, vol. 81, pages 81–109, 1996.

[Prosser 00] P. Prosser, K. Stergiou & T. Walsh. Singleton Consistencies. In Rina Dechter, editor, *Proceedings of the 6th International Conference on Principles and Practice of Constraint Programming (CP-00)*, volume 1894 of Lecture Notes in Computer Science, pages 353–368, Singapore, 2000. Springer-Verlag.

[Puget 93] J.F. Puget. On the Satisfiability of Symmetrical Constrained Satisfaction Problems. In *Proceedings of the 7th International Symposium on Methodologies for Intelligent Systems (ISMIS-93)*, pages 350–361, Trondheim, Norway, 1993.

[Ran 02] Y. Ran, N. Roos & J. van den Herik. Approaches to Find a Near-minimal Change Solution for Dynamic CSPs. In Narendra Jussien & François Laburthe, editors, *Proceedings of the 4th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR-02)*, Le Croisic, France, 2002.

[Régin 94] J.C. Régin. A Filtering Algorithm for Constraints of Difference in CSPs. In Barbara Hayes-Roth & Richard E. Korf, editors, *Proceedings of the 12th National Conference on Artificial Intelligence (AAAI-94)*, pages 362–367, Seattle, WA, USA, 1994. AAAI Press.

[Régin 96] J.C. Régin. Generalized Arc Consistency for Global Cardinality Constraint. In William J. Clancey & Dan Weld, editors, *Proceedings of the 13th National Conference on Artificial Intelligence and the Eighth Conference on Innovative Applications of Artificial Intelligence (AAAI-96 / IAAI-96)*, pages 209–215(2), Proceedings of the 13th National Conference on Artificial Intelligence and the Eighth Conference on Innovative Applications of Artificial Intelligence (AAAI-96 / IAAI-96), 1996. AAAI Press / The MIT Press.

[Roy 98] A. Roy. Symmetry Breaking and Fault Tolerance in Boolean Satisfiability. PhD thesis, University of Oregon, 1998.

[Roy 06] A. Roy. Fault Tolerant Boolean Satisfiability. to appear in the Journal of Artificial Intelligence Research, 2006.

[Sadeh 96] N. Sadeh & M.S. Fox. Variable and Value Ordering Heuristics for the Job-Shop Scheduling Constraint Satisfaction Problem. Artificial Intelligence, vol. 86, no. 1, pages 1–41, September 1996.

[Sakkout 98] H. El Sakkout, T. Richards & M. Wallace. Minimal Perturbance in Dynamic Scheduling. In Henri Prade, editor, *Proceedings of the 13th European Conference on Artificial Intelligence (ECAI-98)*, pages 504–508, Brighton, UK, 1998. John Wiley and Sons, Chichester.

[Sakkout 00] H. El Sakkout & M. Wallace. Probe Backtrack Search for Minimal Perturbation in Dynamic Scheduling. Constraints, vol. 5, no. 4, 2000.

[Schaefer 78] T.J. Schaefer. The Complexity of Satisfiability Problems. In *Conference Record of the 10th Annual ACM Symposium on Theory of Computing*, pages 213–226, San Diego, CA, 1978. ACM Press.

[Schiex 92] T. Schiex. Possibilistic Constraint Satisfaction Problems or "How to Handle Soft Constraints?". In *Proceedings of the Eighth Annual*

Conference on Uncertainty in Artificial Intelligence, pages 268–275, Stanford, CA, 1992.

[Schiex 94] T. Schiex & G. Verfaillie. Nogood Recording for Static and Dynamic Constraint Satisfaction Problems. IJAIT, vol. 3, no. 2, pages 187–207, 1994.

[Schiex 95] T. Schiex, H. Fargier & G. Verfaillie. Valued Constraint Satisfaction Problems : Hard and Easy Problems. In Chris S. Mellish, editor, *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI-95)*, pages 631–637, Montral, Canada, 1995. Morgan Kaufmann.

[Schneider 96] L. Schneider, J. Froschhammer, C. Morgenstern, T. Huslain & J.M. Singer. Searching for Backbones - an efficient parallel algorithm for the travelling salesman problem. Computer Physics Communications, vol. 96, pages 173–188, 1996.

[Smith 93] S.F. Smith & C. Cheng. Slack-Based Heuristics for Constraint Satisfaction Scheduling. In Richard Fikes & Wendy Lehnert, editors, *Proceedings of the 11th National Conference on Artificial Intelligence (AAAI-93)*, volume 2, pages 139–144, Washington D.C., USA, 1993. AAAI Press / The MIT Press.

[Smith 94] B. Smith. Phase Transition and the Mushy Region in Constraint Satisfaction Problems. In Anthony G. Cohn, editor, *Proceedings of the 11th European Conference on Artificial Intelligence (ECAI-94)*, pages 100–104, Amsterdam, The Netherlands, 1994. John Wiley and Sons, Chichester.

[Smith 96] B. Smith & S. Dyer. Locating the Phase Transition in Binary Constraint Satisfaction Problems. Artificial Intelligence, vol. 81, pages 155–181, 1996.

[Taillard 93] E. D. Taillard. Benchmarks for Basic Scheduling Problems. European Journal of Operational Research, vol. 64, pages 278–285, 1993.

[van Beek 92] P. van Beek. On the Minimality and Decomposability of Row-Convex Constraint Networks. In Paul Rosenbloom & Peter Szolovits, editors, *Proceedings of the 10th National Conference on Artificial Intelligence (AAAI-92)*, pages 447–452, Menlo Park, CA, USA, 1992. AAAI Press / The MIT Press.

[van Beek 94] P. van Beek. csplib, library of routines for solving binary constraint satisfaction problems. URL: http://ai.uwaterloo.ca/-vanbeek/software/software.html, 1994.

[van Hentenryck 92] P. van Hentenryck, P. Deville & Y. Teng. A Generic Arc-Consistency Algorithm and its Specializations. Artificial Intelligence, vol. 57, pages 291–321, 1992.

[Verfaillie 94] G. Verfaillie & T. Schiex. Solution Reuse in Dynamic Constraint Satisfaction Problems. In Barbara Hayes-Roth & Richard E. Korf, editors, *Proceedings of the 12th National Conference on Artificial Intelligence (AAAI-94)*, pages 307–312, Seattle, WA, USA, 1994. AAAI Press.

[Verfaillie 05] G. Verfaillie & N. Jussien. Constraint Solving in Uncertain and Dynamic Environments – a survey. Constraints, vol. 10, no. 3, pages 253–281, 2005.

[Vilim 04] P. Vilim. O(n log n) Filtering Algorithms for Unary Resource Constraint. In Jean-Charles Régin & Michel Rueher, editors, *Proceedings of the 6th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR-04)*, volume 3011 of Lecture Notes in Computer Science, pages 319–334, Nice, France, 2004. Springer-Verlag.

[Walsh 02] T. Walsh. Stochastic Constraint Programming. In Frank van Harmelen, editor, *Proceedings of the 15th European Conference on Artificial Intelligence (ECAI-02)*, pages 111–115. IOS Press, 2002.

[Waltz 75] D.L. Waltz. Understanding line drawing of scenes with shadows. In The Psychology of Computer Vision, ed P. Winston, pages 19–91, 1975.

[Watson 99] J.P. Watson, L. Barbulescu, A.E. Howe & L.D. Whitley. Algorithms Performance and Problem Structure for Flow-Shop Scheduling. In Jim Hendler & Devika Subramanian, editors, *Proceedings of the 16th National Conference on Artificial Intelligence and the Eleventh Conference on Innovative Applications of Artificial Intelligence (AAAI-99 / IAAI-99)*, pages 688–695, Orlando, FL, USA, 1999. AAAI Press / The MIT Press.

[Weigel 98] R. Weigel & C. Bliek. On Reformulation of Constraint Satisfaction Problems. In Henri Prade, editor, *Proceedings of the 13th European Conference on Artificial Intelligence (ECAI-98)*, pages 254–258, Brighton, UK, 1998. John Wiley and Sons, Chichester.

[Williams 94] C. Williams & T. Hogg. Exploiting the Deep Structure of Constraint Problems. Artificial Intelligence, vol. 70, pages 73–117, 1994.