Breaking Weak Symmetries in Constraint Programming

Vom Fachbereich Informatik der Technischen Universität Darmstadt genehmigte
Dissertationsschrift
zur Erreichung des akademischen Grades
Doctor rerum naturalium (Dr. rer. nat.)
von

Dipl. Math. Roland Martin
Geburtsort: Singen am Hohentwiel

Technische Universität Darmstadt
Fachbereich Informatik, Fachgruppe Algorithmik

Referent: Prof. Dr. Karsten Weihe
Prof. Dr. Meinolf Sellmann

Einreichungsdatum / Disputation: 14.02.2007 / 11.05.2007

Hochschulkennziffer: D17

Darmstadt 2007

Contents

List of Figures

Acknowledgement

When strength leaves you, when all that is right seems wrong, when all hope is gone and you feel that oblivion in nigh, there is one thing, one feeling that lets you carry on and guides you with its blessing light through the darkness ...

friendship.

Writing a thesis consumes some of the best years of your life. Therefore, when finishing the thesis the question remains: *"Was it really worth it?"*. In my case I can answer this question with a satisfied: Yes! Although a thesis is the achievement of a single individual (and hopefully in all cases it is the writer itself :-) many other people contributed to what is finally the finished thesis. I my case many people from different fields and countries enriched my thesis with their comments or just with things like cheering me up, giving me faith and hope or just be there when I dearest needed them!

There are so many people I would like to thank and although I will try to give them all the honours they deserve although I'm pretty sure I will forget some people.

I like to thank my supervisor Prof. Dr. Karsten Weihe. Not only for giving me the chance to proof that I have what it takes do to research. But also for supporting me when I was down on my knees and lost faith. He always put trust in my talents when I was in doubt about me.

I like to thank my Prof. Dr. Meinolf Sellmann who was willing to be my second supervisor. It meant a lot to me that he was willing to supervise my thesis although we just know each other from conferences and did not have the chance to do research together.

I like to thank the people from SymNet in the UK. SymNet supported me financially several times such that I could visit workshops and conferences I wouldn't be able to visit, otherwise. Also the discussions with the SymNet members were very fruitful for my research and many of the results in this thesis were inspired by talks and discussions held on events I was financed by SymNet.

There are also a lot of people from the CP scene I would like to thank for discussions, inspiration and friendship. First of all Barbara Smith. Barbara gave me invaluable advise every now and then. She was my first contact to the constraint programming community and offered me a lot of help such during all the time.

I would like to take many other people from the constraint programming community: Ian Gent, Karen Petrie, Warwick Harvey, Peter Nightingale, Peter Gregory, Allastor Donaldson, Chris Chefferson, Chris Unsworth, David Bourke, Zeynep Kiziltan, Bernadette Hernandez, Michela Milano, and some more that I forgot for sure. You

people helped me, were open to questions and discussions and offered me friendship during the conferences we met.

When you are not writing or working on your thesis your hobbies help you to relax and regain will and energy to carry on. I like to thank these people that shared a hobby with me. I thank the parties of the Jade Dragons especially the parties of the Lakes Island for good company in Eressea. It is always fun to play with you.

Friends. What would my life be without them! All of you gave me so much energy and strength and encouraged me or just where there when I need someone to laugh with and forget about all the mess that troubled my mind.

Rico, we had so much fun and we both have seen hard times. But we always carried on and tried to make the best of it. Every time the going got tough I could rely on you as a friend that watched my 6 and gave me "close fire support". Thanks for all! Achim, Sven and Andi. Although we couldn't see us much during the last years it was always a pleasure to spend time with you. I remember so much events we had fun and good friendship. And I look forward to a lot more of events where we can celebrate the reunion. Our babies will hopefully form the next generation of our "band of brothers" and have the same joy we had all the time.

Martin, Stephan, Steffen, Marco, Michael. Thanks for all the AD&D gaming and all the friendship and support with my thesis. Life would not be the same without you guys. Special thanks to Martin for mastering our AD&D sessions and bringing so many good plots on the table.

Stephan, thanks for all your help with my thesis. You had always time and ideas how to improve this or that. It is comforting to have you as a friend.

Maik. It's really funny. From a professional relation: You as my dealer of games and I as your customer we developed a real and deep friendship I would miss if it wouldn't be anymore. We both had hard times where we tried to cheer each other up and support the other as good as possible. I'm glad we could cut down the regional distance between us such that we can see each other more often.

My dear friends. My life is enriched by you. I'm so blessed to have so many and so close friends.

But not only existing persons have an influence on my life. For me several non-existing entities gave me strength and courage during the time I wrote this thesis.

I like to thank Stahl Donnerklinge, Paladin of the Order of the Flaming Blade. My alter ego in Dungeons and Dragons role-play. With his identity I could bring order and righteousness to Toril.

I like to thank Judge Dredd, Judge of Mega-City One. His unbreakable belief in the law often gave me strength and courage.

I like to thank Guileman, Primarch of the mighty and most honourable Chapter of the Ultramarines of the undying and divine Emperor. Your codex and the outstanding righteousness of the Ultramarines where always a guiding light for me like the emperor guides the ships safely through the tainted warp space.

I like to thank my family for supporting me all the time. Without their support and embracing love I wouldn't be the person I am today. Thank you Mum and Dad for giving me life. You invested so much love, effort and dedication in me. I know now by myself how much work it is to raise a child. But I also know now by myself how

much it enriches your life and how much joy it is. You always supported me on my path of life and made so much possible for me. I love you so much! Also I like to thank my sister Beatrice and her husband Ralf,my niece Ida, and my nephews Erik and Nils for being there for me when I needed them. It was so adventurous to see the kids grow up and I love you all and hope we can spend a lot of time together.

And my last thanks are contributed to the most important people in my life. My wife Peggy and my son Ares Raidho. They give my life and my every breath a meaning. Peggy, you always encouraged me and trusted in me, when I faltered. Your embracing love cherished me and let me stand up when I was brought down to my knees. Our love is love like it is meant to be: This is everlasting love! I love you so much for giving life to our little son Ares. Ares. You turned my life upside down. When you were finally born to this world and I held you the first time in my arms I felt so much love and happiness I have never before in my whole life. Every smile, every touch, every look you give me gives my life meaning and happiness. May the strength of Ares be in you. Always listen to the voice of your natural Raidho. It will keep you in balance and show you the way to justice, righteousness and happiness.

This thesis is dedicated to you Peggy and Ares Raidho.

Zusammenfassung

Viele Probleme aus Industrie und Wirtschaft sind kombinatorische Probleme mit einer sehr hohen Komplexität. D. h. bisher ist kein Algorithmus bekannt, welcher diese Art von Problemen (oder auch nur eines davon) effizient löst. Constraint Programming bietet Techniken, welche es ermöglichen schwere kombinatorische Probleme beweisbar zu lösen, ohne tatsächlich jede Lösungskombination explizit zu berechnen. Das Prinzip der Beweisbarkeit beruht auf der Tatsache, daß mögliche Lösungskombinationen von der Suche ausgeschlossen werden, falls sie nicht zulässig sind. Dies geschieht über Inferenz-Verfahren, welche unzulässige Variablenbelegungen erkennen, noch bevor sie instantiiert werden. Dadurch können frühzeitig ganze Teilbereiche des Lösungsraums von der Suche ausgeschlossen werden, so daß sich die Anzahl von tatsächlich zu berechnenden Lösungen drastisch verkleinert. Wenn alle diese verbliebenden Lösungenskombination berechnet sind, ist das Problem exhaustiv betrachtet und im Falle der Optimierung ist eine optimale Lösung gefunden. Im Falle eines Entscheidungsproblemes wurde entweder eine Lösung während des Suchprozesses gefunden oder durch die exhaustive Suche bewiesen, daß keine Lösung existiert. Symmetrien in Problemen bilden die kombinatorischen Möglichkeiten eines Problemes auf sich selbst ab. Symmetriebehandlung (auch Symmetriebrechung genannt) reduziert das Problem auf seinen kombinatorischen Kern. Besteht ein Problem aus der Belegung von n Variablen und sind diese Variablen symmetrisch, dann existieren $n!$ viele Permutationen für eine beliebige Lösung. Ein simples Beispiel ist eine Nebenbedingung der Form: $x_1+x_2+x_3 = 6$. Eine Lösung für diese Nebenbedingung ist z.B. $x_1 = 1, x_2 = 2, x_3 = 3$. Jede Permutation der Werte ist aber ebenso eine Lösung, weil ($1+2+3 = 1+3+2 = 2+1+3 = 2+3+1 = 3+1+2 = 3+2+1 = 6$). Es existieren also sechs Lösungen, welche genau die gleiche Lösung (nämlich die Werte 1,2,3 zu benuten) beschreiben. Symmetrien bilden eine mathematische Gruppe und daher kann die Gruppentheorie benutzt werden, um zu zeigen, daß das Auschließen von symmetrischen Lösungen eine korrekte Methode ist, den Lösungsraum zu verkleinern, ohne Information zu verlieren.

In dieser Dissertation klassifizieren wir eine neue Art von Symmetrie, welche die Eigenschaft hat, daß sie nur auf einem Teilproblem P_1 des ursprünglichen Problems P symmetrisch ist. D. h. daß es Variablen und/oder Nebenbedingungen in P gibt, denen die Symmetriefunktion nicht genügt. Diese Symmetrie nennen wir *schwache Symmetrie (weak symmetry)*. Schwache Symmetrien können nicht auf dem Problem P gebrochen werden, weil eine schwach symmetrische Variablenbelegung, welche dem Problem P_1 genügt (d.h. das Nebenbedingungssystem erfüllt) nicht automatisch auch P genügt. Insbesondere bedeutet dies, daß nicht alle schwach symmetrischen Equivalente einer Lösung, welche P_1 erfüllt zu den gleichen Lösungen für P führen. Symmetriebrechung von schwachen Symmetrien auf dem Problem P führt daher zu einem irreversiblen Verlust von Lösungen, so daß ein Problem nicht mehr

beweisbar gelöst werden kann. Wir führen in dieser Dissertation eine Technik ein, welche es uns erlaubt, schwache Symmetrien zu brechen, ohne Lösungen zu verlieren. Dazu führen wir zusätzliche Variablen ein, welche es erlauben, die durch die Symmetriebrechung augeschlossenen Lösungen während der Suche zu rekonstruieren. Dies ermöglicht es, Symmetriebrechung anzuwenden und damit den Lösungsraum zu verkleinern, ohne Lösungen zu verlieren. Es werden also nur solche Variablenbelegungen ausgeschlossen, welche tatsächlich keine neue Lösung liefern. Diese Technik ist unabhängig vom eingesetzten Solver und benötigt keinen zusätzlichen Programmcode, um angewandt zu werden. Dadurch ist unsere vorgestellte Technik nicht an eine bestimmte Software gebunden und steht jedem zur Verfügung, der sich mit Constraint Programming beschäftigt. Das Anwenden erfordert zudem kein Wissen in Gruppentheorie, sondern nur grundlegende Modellierungsfähigkeiten. Dies erweitert den Anwenderkreis weiterhin. Die meisten Probleme aus der Industrie und Wirtschaft bestehen aus mehreren Teilproblemen, welche nicht unabhängig voneinander gelöst werden könen. Gerade diese Problem beinhalten meist keine Symmetrien sondern nur schwache Symmetrien. Unsere Technik erlaubt es nun also neue Anwendungsfelder für Constraint Programming im allgemeinen und die Symmetriebrechung im speziellen zu erschließen.

Wissenschaftlicher Werdegang

1996–2002	Studiengang Mathematik mit dem Abschluß Diplom-Mathematiker an der Universität Konstanz
2002–2007	Promotion im Fachbereich Informatik, TU Darmstadt

Declaration

Parts of the dissertation have appeared in the following publications which have been subject to peer review.

1. Roland Martin and Karsten Weihe. *Breaking weak symmetries.* In Warwick Harvey and Zeynep Kiziltan, editors, *Proceeding of the Fourth International Workshop on Symmetry and Constraint Satisfaction Problems*, pages 48–54, Toronto, Canada, 2004.

2. Roland Martin. *Approaches to Symmetry Breaking for Weak Symmetries.* In Alastair Donaldson and Peter Gregory, editors, *Proceedings of the Workshop on Almost-Symmetry in Search*, SymNet Workshop on Almost-Symmetry in Search, New Lanark, 2005

3. Roland Martin and Karsten Weihe. *Solving the Magic Square Problem by Using Weak Symmetries.* In M. Carlsson, F. Fages, B. Hnich, and F. Rossi, editors, *Proceedings of the Joint ERCIM/CoLogNet International Workshop on Constraint Solving and Constraint Logic Programming – CSCLP 2005*, CSCLP 2005, Uppsala, Sweden, 2005

4. Roland Martin. *The challenge of exploiting weak symmetries.* In Brahim Hnich, Mats Carlsson, Francois Fages, and Francesca Rossi, editors, *Recent Advances in Constraints, LNCS 3978*, volume LNCS 3978, pages 149–163, 2005.

5. Roland Martin. *Speeding up Weak Symmetry Exploitation for Separable Objectives.* In Alastair Donaldson, Peter Gregory, and Karen Petrie, editors, *Proceeding of the Workshop on Symmetry and Constraint Satisfaction Problems*, SymCon 2006, Nantes, France, 2006

6. Roland Martin and David A. Burke. *An Approach to Symmetry Breaking in Distributed CSP.* In K. Petrie, editor, *Proceeding of the International Symmetry Conference*, International Symmetry Conference, Edinburgh, 2007 (To appear in the Annals of AI and Maths)

Abstract

Constraint programming (CP) is a powerful solving paradigm that is based on inference and search control algorithms and suitable for arbitrary/various $\mathcal{NP}$-hard combinatorial problems beyond linearity. The flexibility of constraints – the working machines of a CP solver – allow a wide range of problems to be solved by constraint programming solvers. Constraint propagation and search control are to two main concepts that make CP an efficient solving strategy. The former identifies infeasible regions of the search space and prunes them. The latter allows to state search heuristics that guide the search in more promising regions of the search space. A problem is passed to a constraint solver by a model using variables and constraints. The flexibility of modelling a problem for a CP solver allows rapid prototyping and solving whereas problem-tailored algorithms need a long development time. Small changes in the problem description can be compensated by just altering the model while problem-tailored algorithms may be not applicable to the new situation anymore. This makes constraint programming very robust in terms of modelling.

Another very powerful approach in constraint programming is *symmetry breaking*. A symmetry in constraint programming can be seen as a function mapping several solutions to each other. A symmetry preserves the feasibility state of a solution. Therefore, solutions that are symmetric are either all feasible or all infeasible and build a solution class. Symmetry breaking reduces the problem to its combinatorial core by excluding all but one symmetric solution of each solution class. The search space is significantly reduced by excluding equivalent solutions from the search such that only unique regions of the search space are investigated during the search. A condition for symmetry breaking to be sound is that no unique solution is excluded in the process of symmetry breaking. All symmetry breaking methods are based upon this necessary criteria. There are several symmetry breaking methods proposed during the last years and the success of these methods was proved in several publications on conferences and workshops [1, 3, 4, 2]

In this thesis we investigate a kind of symmetry we call *weak symmetry*. Weak symmetries have the special property that the weak symmetric equivalents are only equivalent for a subset of the variables and constraints of a problem, i.e. some variables and constraints are not respected by the symmetry. Solutions that are weak symmetric are only full symmetric with respect to the subset of the variables and constraints of the problem. This means that in the context of the full problem weak symmetric solutions can have different feasibility states. Excluding all but one solution, as it is done by symmetry breaking, does exclude solutions that cannot be retrieved afterwards. Therefore, a weak symmetry cannot be broken by standard symmetry methods without losing solutions. Weak symmetries are interesting from an academical view on how to break them in order to again reduce the problem to

its combinatorial core. But moreover, although only recently discovered weak symmetries have already a large application range. Weak symmetries arise in classical areas as optimisation and satisfaction as well as in real-world scenarios, distributed constraint satisfaction programmes soft constraints, planning, scheduling, and model checking. In the first field – real-world problems – weak symmetries often arise by an objective function or just by the fact that the problem investigated consists of several interlinked problems which cannot be solved individually without conflicting with the other subproblems. In the former three fields – real-world problems, soft constraints and distributed constraint satisfaction programmes – often there are more weak symmetries than standard symmetries. With weak symmetry breaking new problem classes can be handled more efficiently by constraint programming. Therefore, weak symmetry breaking introduces a large potential of improvement for symmetry breaking in particular and constraint programming in general.

In this thesis we classify weak symmetries and introduce an approach to weak symmetry breaking that is based on pure modelling.

The advantages of this approach are:

- **Universality:** Every solver uses a modelling language to state problems that are passed to the solver. Our approach does not need additional implementations so we are not limited to a specific solver.
- **Ease of use:** A person familiar with modelling can adept the approach easily and is capable of remodelling a problem to break weak symmetries. Although modelling needs some expertise the principles of modelling weak symmetries are very easy to understand such that even inexperienced constraint programmer can use the technique immediately.
- **No background knowledge required:** Symmetries are based on groups and therefore the theory of symmetries and symmetry breaking is group theory. It is possible to use weak symmetry breaking without specific knowledge of group theory.
- **Readiness:** Since no extra code has to be written, incorporated or adjusted, our approach can be used instantly. Existing models can be easily upgraded with weak symmetry breaking with just a few changes to the constraints.
- **Interoperability:** When a model is revised, the weak symmetry can be handled using standard symmetry breaking methods such that the approach also profits from research in this field. Any symmetry breaking method can be used once the model is revised in the way we propose.
- **Concurrent Symmetry Breaking:** Problems may contain standard and weak symmetries. Both can be handled concurrently since weak symmetry breaking actually transforms a weak symmetry into a standard symmetry from a certain viewpoint.
- **Robustness:** Since no method specific code has to be added to the existing solver, the chance of producing errors is minimal and reduced to the validity of the model. But there is no problem with memory management, exception handling, etc.

- **Openness to Refinement:** The basic approach can be extended in several ways to suit different applications. For example, it is easy to adopt the approach for *partial symmetries*. Although based on modelling, it is also possible to extend the approach by incorporating code to the constraint solver. This way the approach can be adopted to various applications and scenarios in order to maximise efficiency.

Our approach is based on introducing new variables to a model called *SymVars*. The set of SymVars represent symmetric equivalents of solutions to a symmetric subproblem P_1 of the regarded *constraint satisfaction problem* P, whereby P is not symmetric. The search space of the SymVars represents the whole equivalence class of a solution that is weakly symmetric on P. Since all symmetric equivalents of a solution to P_1 are reflected by the SymVars, it is sufficient to find just one solution of each equivalence class in P_1. That is exactly what symmetry breaking does. Therefore, by introducing SymVars, we are able to break the weak symmetry on the problem by standard symmetry breaking methods. If the search encounters a feasible variable assignment (with respect to the symmetric part P_1 of the problem) its symmetric equivalents are investigated to see whether one of them also satisfies the asymmetric part of the problem. If so, a solution is found and if not a different variable assignment is sought that satisfies the constraints of P_1. For this variable assignment again the symmetric equivalents are investigated and so on.

Depending on the problem, we do not have to search the whole equivalence class of a solution. Infeasibility can be determined for partial SymVar instantiations such that backtracking can be performed excluding parts of the search space. It is also possible to state a variable and value ordering on the subproblem of instantiating the SymVars which can help to investigate the equivalence class faster.

We also demonstrate some techniques to speed up our approach. Some of them are based on modelling like stating conditional constraints to annihilate search on the stabiliser of a variable assignment. Other techniques require writing code and incorporate this in the solver. One of these techniques is based on exploiting special behaviour for a class of problems. In this class, an objective function evaluates the each solution and the function is separable. This means that certain parts of a solution contribute a specific amount to the objective value and this amount is independent from the rest of the solution. In these problems, we can impose a domination criteria on partial solutions such that we can (by storing partial information) prune large parts of the search space in addition to the savings provided by weak symmetry breaking.

Chapter 1

Introduction

Finding a good solution that respects some restrictions is ubiquitous in every day life. Consider for example a business woman that has to schedule several events. The following events have to fit in the schedule:

- She has to send a report to her superiors by the end of the day. Writing the report will take about 2 hours of total working time but can be interrupted by other tasks.
- She has a meeting with other colleagues of her department. The meeting has to start between ten and twelve in the morning and takes about one hour. The meeting can only be scheduled when all attendees have got time.
- A meeting at a company is arranged in the afternoon. The meeting will take about 2 hours and travelling time is about thirty to fifty minutes (depending on traffic).
- The children have to be picked up after school at six o'clock in the evening
- A candlelight dinner is arranged with her partner at eight o'clock in the evening.

The question is whether there is a schedule such that all tasks can be performed. If not, which events could be postponed and what are the consequences?

When solving problems of any kind there are basically two main solving approaches. The first is to use problem-tailored algorithms that fit more or less perfectly for the problem. The second is to use existing solving architectures or special problem solvers like linear solvers [46] or SAT solvers [57] for example.

The gain in using problem-tailored algorithms is that they can tackle the problem without distorting the problem and specific ideas can be used to solve exactly this problem. The drawback is that mostly the algorithms have to be developed and implemented anew to suit the special needs of the problems or to achieve optimal performance. In addition to this, changes in the problem description may cause the algorithms to fail and not seldom new different algorithms have to be used because the changes make the problem unfit for the algorithm used so far.

The gain in using existing solvers is that the solving architecture is already at hand. Therefore, development time is relatively short. Changes in the problem description can be incorporated easily by adding new constraints or changing the existing

constraints. This is done on the level of modelling. Also, the solving algorithms are often highly sophisticated, optimised and efficient. The main drawback is that these solvers are mostly specialised for certain scenarios like linear programming which limits the usability. Therefore, the problem often has to be relaxed such that it fulfills the restrictions of the solver. In many cases, these relaxations do not fully reflect the true problem and the solutions are subject to interpretation.

Somewhere between these two approaches lies the technique of *Constraint Programming (CP)*. CP is based on a tree traversal algorithm to enumerate the search space as a core solving strategy. Several algorithms based upon branch and bound (in case of optimisation) are incorporated to speed up the search.

CP also gives the user the possibility to direct the search through the search tree. Thereby, heuristics and special characteristics of a solution can be implemented.

Basically, CP is used for $\mathcal{NP}$-hard combinatorial problems. No polynomial algorithm is known to solve such a problem and it is unknown if there exists one, either.

CP has become very popular and is used successively in many application fields like scheduling [8], bio-informatics (sequence alignment) [36], [6] operation research (optimisation) [88], and many more real-life applications [89], [76].

The roots of CP can be found in the fields of artificial intelligence back in the 1960s [88]. But CP has applications in various fields of computer science like operations research, computational logic, combinatorial algorithms, discrete mathematics, and programming languages [54], [76].

Symmetries are ubiquitous in constraint programming models and enlarge the search space of a problem. Many problems like the social golfer problem [41] contain a super-exponential number of symmetries. Therefore, breaking these symmetries does greatly reduce the solution space yielding the combinatorial core of the problem. Therefore, it is most desirable to exclude symmetric equivalents from the search space in order to speed up the search. The process of excluding symmetric equivalents is called symmetry breaking. Several methods have been investigated to break symmetries. And they all have the same modus operandi: exclude all but one solution of each solution class. Symmetry breaking is a sound operation since all solutions excluded are symmetric to the one solution (called the *representant* of the solution class that is not excluded. Therefore, no information is lost. Symmetry breaking has proved to be very effective and efficient in several problems (see [7, 18, 22, 29, 32, 31, 40] for example) in reducing the search time and/or reducing the search space. There are a lot of problems with symmetries and a lot of symmetries with many, sometimes a super-exponential number of symmetries to investigate. But most problems investigated originate from the fields of puzzle problems or have limited use in the real world. Also real world problems are often more complex and interlinked such that they do not correspond fully to the problems investigated in academy. Academic problems are very suitable to investigate the core of a problem and project these "lessons learned" to larger problems. Often a part of a problem is extracted and investigated for the best solving method. But an optimal solution to this partial problem does not necessarily yield an optimal solution (or even a feasible solution) for the global problem. So, if there is no one-to-one correspondence between these problems the results have limited significance. Therefore, it would be desirable to have results on symmetry breaking that do correspond more exactly with the real world.

Weak symmetries are a kind of symmetry that act only on subproblem of the original problem. Therefore, weak symmetric solutions are only symmetric with respect to the variables and constraints of the subproblem they act on. Indeed, there are a lot of problems, especially real-world problems contain no standard symmetry but a lot of weak symmetries [59]. In these problems symmetry breaking could not be performed since that would have led to a loss in solutions. Also a lot of already investigated problems in constraint programming contain weak symmetries along with standard symmetries. Up to now only the standard symmetries could be tackled. Therefore, there is a large application field where symmetry breaking could not be applied to and also a lot of problems where symmetry breaking could be used more efficiently than already done.

The key flaw in trying to breaking weak symmetries is that solutions are lost in the process of symmetry breaking. Therefore, standard symmetry breaking is not sound anymore. We introduce a modelling technique that not only allow us to break weak symmetries without losing solutions, moreover, using the technique we can use any symmetry breaking method that exists, if desired. Therefore, we can prosper from the experience and work invested in symmetry breaking methods. Even more important: all future symmetry breaking methods can be used as long as they are designed to respect the main principle of preserving at least one solution of each solution class.

Weak symmetries (and especially weak symmetry breaking) gives new potential to investigate in symmetry breaking. More application fields and especially real world problems can now be in the focus of research in symmetry breaking. Also these results can be used directly for the real world problems since the whole problem can be considered and not only a partial problem.

Our technique is based on pure modelling such that is independent of the used solver, self-written code and extra knowledge in specific fields like group theory. Still, it is flexible in the way that it can be combined with self-written code to enhance the power or make use of special properties of a problem. An example for this is investigated in Section 5. The most efficient symmetry breaking methods are based on algorithms that have to be implemented and adjusted for the specific symmetry to be broken. If the code or the knowledge how to use this code is not at hand there are symmetry breaking methods based on modelling. Therefore, our approach is usable and fruitful even if no sophisticated algorithms for the symmetry breaking are used but pure modelling. So, our modelling technique can be not only used for a wide range of applications and problems, moreover, it can be used by experienced and inexperienced constraint programmers independent of the solver used.

1.1 Organisation of the Dissertation

This thesis is organised as follows:

Chapter 1, Introduction: Here we give a brief introduction in the roots of constraint programming. We briefly show that symmetries are very common in constraint programming models and that symmetry breaking is the key to overcome the flaw of symmetries in a problem. Also we introduce our research and the possibilities that arise with that.

Chapter 2, Fundamentals of Constraint Programming: Here we provide an overview on constraint programming and the techniques used in constraint programming.

Chapter 3, Symmetry: We introduce in symmetries in constraint programming and the mathematical concept behind symmetries – Group theory. Also we give an overview of symmetry breaking methods.

Chapter 4, Weak Symmetry: Here we present part of our work. Weak symmetries are formally introduced and several problems and problem fields are investigated for weak symmetries. Also we introduce the modelling approach to break weak symmetries and investigate related work.

Chapter 5, Efficient Symmetry Group Investigation for Separable Objectives: In this chapter we present a way how to investigate the symmetry group of weak symmetries in problems with separable objectives. The resulting algorithm can be encapsulated in a global constraint. This shows that our modelling approach is open to specialisation and can be adapted to suit problems with special features.

Chapter 6, Computational Results: Here we investigate different instance sets of three problem containing weak symmetries and present the results.

Chapter 7, Conclusions and Future Work : In this final chapter we summarize our work and contribution. Also we give some directions which are very interesting and where research can be carried on.

Appendix: Here, for the sake of completeness, we present additional plots for the results of Chapter 6 that were not directly used for the analyses.

Chapter 2

Fundamentals of Constraint Programming

A brief History of Constraint Programming: Some of the earliest ideas on which CP is based may be found in the artificial intelligence area of constraint satisfaction back to the 1960s [88]. Early application areas for constraints were for example circuit modeling [84], scene labeling [67], as well as interactive graphics [14]. For example the Sketchpad, developed in the early 1960s by Ivan Sutherland [85], comprised some of the vital concepts of constraint programming [88]:

1. constraints as a declarative relation
2. local propagation constraint solvers
3. multiple cooperating solvers

"The main step towards constraint programming was achieved when it was noted that logic programming was just a particular kind of constraint programming. Logic programming is based on a declarative computational paradigm in which a program is a logic theory and each computation step solves a system of term equations via the unification algorithm"[88].

Constraints have proven to be even more than just knowledge-representations. They turned out to be useful for guiding computations and enable pruning of uninteresting branches during the search.

Following were languages that incorporated and used constraints like CLP(R)[50], Prolog III [12], and CHIP [16]. These languages used the early ideas of propagation and constraint processing.

Constraints: A constraint can be thought of as a restriction of the space of feasible solutions. Constraints are ubiquitous in every-day life and in most areas of human endeavor. Simple facts and nature laws can be expressed using constraints:

- The three angles of a triangle must sum up to 180 degrees.
- The sinus of an angle in a rectangular triangle is defined by the quotient of the opposite leg and the hypotenuse.

- The four bases DNA strands are made of can only combine in certain pairs;
- An object is only visible if there is a line of sight to it (i.e. if no other object blocks the way).

Also business rules can be states using constraints:

- The office of the head and the secretariat must be adjacent.
- ...

These are only a few examples of constraints common in science and economy. Even celebrated conjectures of mathematics like Fermat's Last Theorem may be viewed as the question whether certain constrains are satisfiable [88].

The idea of constraints is to state *what* has to be satisfied and not *how* this is achieved. Constraints are the conceptual basis of constraint satisfaction problems.

Constraint Satisfaction Problem (CSP):

For a single constraint it is easy to determine whether it can be satisfied (i.e. there exists a solution such that the constraint is fulfilled). But for a set of constraints the task is in general very hard. Informally a CSP consists of a set of decision variables and a set of constraints stated over these variables.[1] The question is whether there is a solution (a value assignment to the variables) that satisfies all constraints simultaneously.

Many decision problems can be formulated as CSPs. For example, consider the n-queens problem [31]. In this problem n queens have to be placed on a $n \times n$ chessboard such that no two queens can attack each other. Or consider graph colouring [51]: Given is a planar graph consisting of nodes and edges between pairs of nodes. There are various questions possible. For example, can the nodes of the graph be coloured using three colours such that no two nodes connected by an edge have the same colour.

In general, there is no known polynomial algorithm that solves CSPs. Solving CSPs is computationally intractable [56] and is $\mathcal{NP}$-hard.

Constraint Programming:

"Constraint programming (CP) is the study of computational systems based on constraints"[88].

Constraint Programming provides a solving architecture for CSPs. Although intractable, CP has been successfully applied to various applications like scheduling [77], planning [77], and many more real-life applications [89], [76]. For some application fields like scheduling, CP remains the most promising solving architecture (without regarding problem-tailored algorithms). Especially if there is not much knowledge about the structure of the solution space, CP can be an efficient and easy to use alternative to problem-tailored algorithms.

Unlike many problem solvers like those based on the simplex algorithm or SAT solvers, constraint programming can handle more general problems since the problem

[1] A CSP is formally defined in Section 2.1.

specification is not very restricted. The only limitation that applies is that the problem can be stated in the underlying constraint language of the specific constraint programming solver.

But since constraint programming is capable of various problem specifications, there is no specialised solving algorithm that can profit from certain characteristics of the structure of the solution space. In linear problems, for example, the form of the solution space can be exploited very efficiently by the simplex algorithm.

A constraint solver acts like an enumerator on the set of all possible assignments but with two crucial ingredients:

- constraint propagation and
- search control

Propagation is capable of detecting inconsistencies of a partial solution (i.e. an assignment to a subset of all variables) early in the search and also before the inconsistency basically occurs explicitly. Therefore, large regions of the search tree may be pruned without investigation. Search control enables the solver to steer the search in the search tree to investigate interesting or more promising regions first. This is done using a heuristic or exploiting randomness. Since every possible solution is either ruled out implicitly or investigated explicitly, the search is complete.

We give a brief introduction in the fundamentals of constraint programming. First we state the basic definitions in Section 2.2, give a brief overview of the solving process in Section 2.3 and introduce the concept of consistency and propagation in Section 2.4. In Section 2.5 we show how search control works by ordering heuristics. Finally, in Section 2.6 we introduce into modelling and show some common modelling techniques.

2.1 Problem Solving by CP

CP attracts more and more attention from industry [73]. One reason is that CP solvers get more efficient and compatible. A second and more important reason is that more and more problems are modelled and solved using CP technology. Many problems are very complex and no general solving approach is known that solves the problem efficiently. In such a case CP can be used gainfully.

A complex example was presented by ILOG [73]. A large real world problem was successively solved using a mixture of solving architectures. The problem was split in three parts: allocation/planning, batching and scheduling. The allocation subproblem was solved using mixed integer programming (ILOG CPlex MIP [46]). The scheduling subproblem was solved using scheduling algorithms (ILOG Scheduler [48]). These subproblems are solved efficient using these approaches. For batching there is no solving approach that fit all needs of the problem. Therefore, CP was used (ILOG Solver [49] to solve the batching subproblem successfully.

As mentioned before there are ubiquitous problems that are of interest in constraint programming. A few examples are:

- Drawing up a timetable for a conference

- Choosing frequencies for a mobile-phone network
- Checking the satisfiability of a logical formula
- Fitting a protein structure to measurements
- Laying out components on a circuit board
- Scheduling a construction project

All these problems involve searching for a solution that satisfies a set of constraints.

For many problems or classes of problems there exist efficient solvers that solve any instance of a problem in polynomial time. These problem classes are said to be tractable. Examples are Gauss-Jordan elimination for systems of linear equations, calculus for graph discussions or the simplex algorithm for fractional linear programs.

2.1.1 Complexity of CSPs

Whether a CSP is tractable or not depends on the constraint language that has to be used to model the problem, where a constraint language is defined as a set of relation types over a finite set D [77]. A constraint language induces a class of problems. Namely all problems that can be modelled using exactly this constraint language (not a subset of the relations). If a constraint language is proven to be tractable then all problems in its problem class are tractable. And analogously for intractable constraint languages. For example affine relations corresponds to simultaneous linear equations which is tractable while the inequality relation $\neq$ corresponds to graph colouring and is $\mathcal{NP}$-hard.

Although mathematics was able to classify many constraint languages as tractable there are also constraint languages that are proved to be intractable and also many constraint languages that are not classified yet. That means that no efficient solving algorithm is known but no proof , either, that the constraint language is intractable. For problems of the later two cases constraint programming is a good solving strategy. In general the CSPs that are solved by constraint programming are intractable [56].

2.2 Terminology

In this section, we introduce the concepts and notations used throughout the dissertation.

define the prerequisites upon which the concepts of constraint programming is based on.

There are two kinds of problems that are regarded in Constraint Programming . These are satisfaction and optimisation problems. While in the first scenario it is sufficient to find just one solution in the later scenario the task is to find an optimal solution.

Definition 2.1 (Constraint Satisfaction Problem, CSP) *A Constraint Satisfaction Problem is characterised by $P = (X, D, C)$, where*

- $X = \{x_1, \ldots, x_n\}$ *is the set of variables;*
- $D = \{D_{x_1}, \ldots, D_{x_n}\}$ *is the set of domains for the variables in* X*;*
- $C = \{c_1, \ldots, c_m\}$ *is the set of constraints, where each constraint states a relation over a subset of variables.*

There is also a scenario where all solutions to a CSP are sought after. In research this is used to determine the efficiency of different solving heuristics or strategies. In real-world applications this can be used to classify problems (whether they are *tight-fit* i.e. there exist only very few solutions or whether they are *loose* i.e. there exist many solutions). Also in terms of robustness finding all solutions is useful. Sometimes some solutions may be hard to realize (although perfectly valid) which cannot be expressed in the constraint system. In this case alternative solutions can be considered from the list of solutions. Consider for example buying a house. In this case the customer would like to see all houses that satisfy the specified constraints and not just the first found by the system.

Optimisation can be regarded as a variation of satisfaction where the difference is that each solution is also assigned an objective value which leads to a ranking of the solutions.

Definition 2.2 (Constraint Satisfaction Optimisation Problem, CSOP)

A Constraint Satisfaction Optimisation Problem $P_{opt} = (X, D, C, f)$ *consists of a CSP* $P = (X, D, C)$ *defined as in Definition 2.1 and an objective function* $f : X \to \mathbb{R}$ *that evaluates each solution by assigning an objective value to it. The goal is to find a solution with minimal or maximal objective value.*

Of interest is only a solution with minimal/maximal value i.eản optimal solution. Therefore, only the best solution is returned at termination of the solving process. In practice the underlying CSP is solved. When it returns a solution a constraint is added that states that only solutions with an objective value better than the last found solution is feasible. When the whole search space is traversed the last solution found is returned as the optimal solution. Therefore, optimisation can be regarded as repetitive solving the underlying CSP.

Example 2.3 *Sudoku Puzzle*

Consider the sudoku puzzle for an example of an CSP. ***Sudoku****, also known as* ***Number Place****, is a logic-based placement puzzle. The aim of the puzzle is to enter a numerical digit from* 1 *through* 9 *in each cell of a* 9×9 *grid made up of* 3×3 *subgrids (called regions), starting with various digits given in some cells (the givens); each row, column, and region must contain only one instance of each digit [92]. (See Figure 2.1 for a Sudoku puzzle and its solution).*

The problem can be modelled in the following way: Each cell of the 9×9 *grid is a variable* x_{ij} *and the domain of each variable is* $D_{x_{ij}} = \{1, \ldots, 9\}$*. The constraints are that all cells in each row and column of the grid and any cells in each subgrid contain the numbers* 1 *to* 9*. Or in other words, no two cells of each column, each row and each subgrid may have the same number assigned. This can also be expressed by the sum of these cells which must be* 45*.*

The formal description of the problem is:

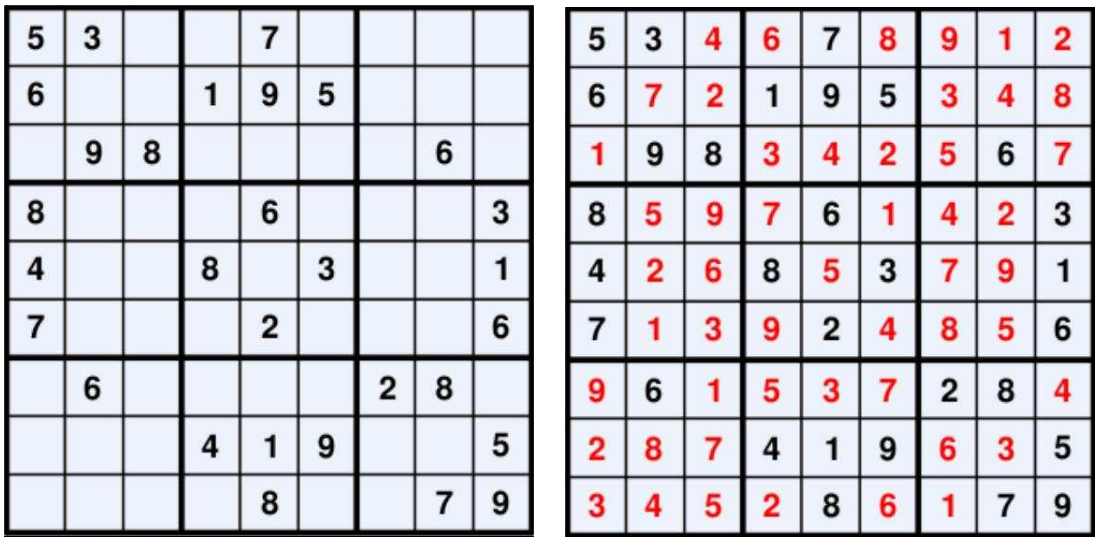


Figure 2.1: A Sudoku puzzle and its solution

$X = \{x_{i,j}\}, i, j \in \{1, \ldots, 9\}$
$D = \{D_{x_{ij}} = \{1, \ldots, 9\}\}$
$C =$
$\forall i \in \{1, \ldots, 9\} : \sum_{j \in \{1,\ldots,9\}} x_{ij} = 45$ *(rows of the grid)*
$\forall j \in \{1, \ldots, 9\} : \sum_{i \in \{1,\ldots,9\}} x_{ij} = 45$ *(columns of the grid)*
$\forall k, l \in \{0, 1, 2\} : \sum_{i,j \in \{1,\ldots,3\}} x_{(3 \cdot k+i, 3 \cdot l+j)} = 45$ *(subgrids)*
$\forall i \in \{1, \ldots, 9\} : x_{ij_1} \neq x_{ij_2}, j_1, j_2 \in \{1, \ldots, 9\}, j_1 \neq j_2$ *(rows all different)*
$\forall j \in \{1, \ldots, 9\} : x_{i_1 j} \neq x_{i_2 j}, i_1, i_2 \in \{1, \ldots, 9\}, i_1 \neq i_2$ *(columns all different)*
$\forall k, l \in \{0, 1, 2\} : x_{(3 \cdot k+i_1, 3 \cdot l+j_1)} \neq x_{(3 \cdot k+i_2, 3 \cdot l+j_2)}, i_1, i_2, j_1, j_2 \in \{1, 2, 3\}, i_1 \neq i_2, j_1 \neq j_2$ *(subgrids all different)*

When a variable is assigned a value out of its domain the variable is said to be *instantiated*. Each domain of each variable states which values the variable can take. The solution space of a CSP is the Cartesian product of the domains of all its variables.

Definition 2.4 ((Partial) Variable Assignment)
Consider a CSP $P = (X, D, C)$. In a **variable assignment** *each variable $x_i \in X$ of P is assigned a value $v_i \in D_i$. In a* **partial variable assignment** *only a subset of the variables $x_i \in X$ of P is assigned a value $v_i \in d_i$.*

Every variable assignment to all the variables in X is either evaluated feasible or infeasible. To state which values are feasible with each other we use constraints. The constraints state the possibilities of coexisting values for variables. A feasible variable assignment is also called a solution of the CSP.

Definition 2.5 (Solution)
Consider a CSP $P = (X, D, C)$.
If a variable assignment is consistent with all the constraints in C it is **feasible** *and called a* **solution** *to P.*

Normally a constraint does not act on all the variables of the problem but on a subset. The number of involved variables is called the *arity* of the constraint.

Definition 2.6 (Arity of a Constraint)

The number of variables over that a constraint is stated is called the ***arity*** *of the constraint and is denoted by* $|c|$.

A constraint is said to be *satisfied* if all its variables are instantiated such that its logical declaration is true and said to be *violated* otherwise.

Search in constraint programming is done usually by assigning variables consecutively. In each step a partial variable assignment is extended by an additional assignment of one variable.

Definition 2.7 (Partial Solution)

Given a CSP $P = (X, D, C)$ *and a partial variable assignment to the variables* $X' \subset X$.

If a partial variable assignment is consistent with all constraints in C' *it is* **feasible** *and called a* **partial solution**. C' *is the projection of* C *to variables in* X'*, such that all constraints in* C' *contain only variables of* X'.

Variables that are not instantiated are called *future variables*.

The search space of a CSP can be represented by a tree. Solving a CSP corresponds to visiting nodes of the tree until a leave that satisfies C is found. Each layer of the tree represents a concrete variable and a node in the search tree represents a partial variable assignment to all the variables up to this layer (beginning from the root). The leaves of the search tree represent all possible variable assignments of the CSP amongst are the solutions.

Example 2.8 *A small search tree*

Consider a CSP with three boolean variables x_1, x_2, x_3. *The full search tree can be seen in Figure 2.2.*

The internal nodes with the labels 2 *to* 7 *represent partial assignments while the nodes with the label* 8 *to* 15 *represent full assignments to all three variables. Each layer under the root represents a variable that is assigned. The path from the root to an internal node marks the partial assignment. The path from the root to a leaf represents a full assignment. We assume that a left child of a node represents the assignment of* 0 *to the variable in this layer and a right child the assignment* 1. *For example the assignment in the node* 5 *would be* $(x_1 = 0, x_2 = 1)$ *and the assignment in the node* 12 *would be* $(x_1 = 1, x_2 = 0, x_3 = 0)$.

2.3 Overview of the Solving Process

In constraint programming a partial variable is extended to a full assignment. Thereby the search space is enumerated. But unlike a brute force approach (where all possible assignments are instantiated) constraint programming makes use of two concepts:

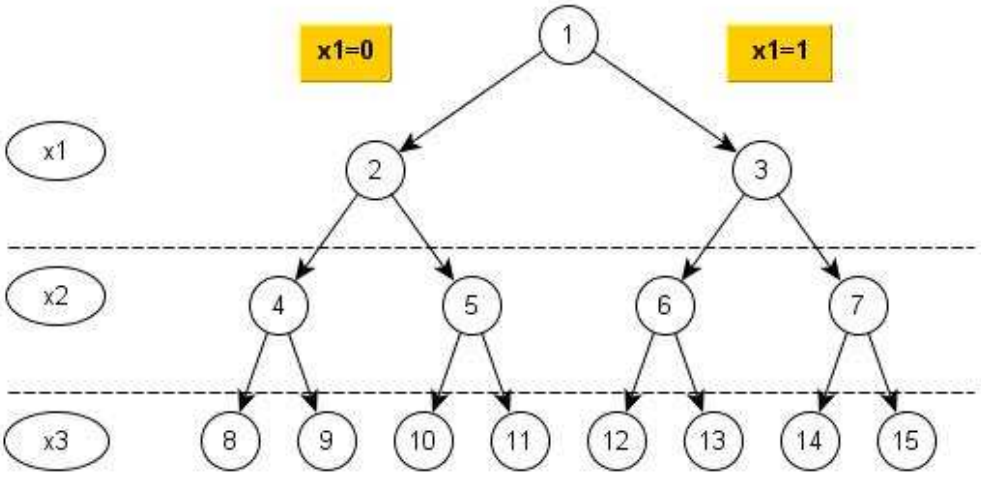


Figure 2.2: A full search tree with three boolean variables.

- Constraint Propagation: Pruning infeasible sub-trees by reasoning on the constraints (See Section 2.4)
- Controlling the search: Variable and value ordering heuristics (See Section 2.5)

Constraint programming infers necessary conditions over the constraints' variables [76].

Constraint propagation can be performed in a pre-processing step as well as during the search. When applied during the search values are removed that cannot co-exist with the partial solution found so far by the search. By doing so the number of possible variable assignments to be investigated is reduced. Early pruning leads to considerable reduction of the search space of the problem. This is due to the fact that each sub-tree of a node represented by the pruned value does not have to be considered since it will be infeasible.

After each variable instantiation, propagation algorithms remove infeasible values if possible. Depending on the propagation algorithm used a different *consistency state* is reached. More sophisticated propagation algorithms are more time consuming but result in a higher level of consistency. The stronger the propagation algorithms are the more infeasible values they can remove. Basically, the strength of a propagation algorithm lies in the ability to detect infeasible values early. We will see in Section 2.4 the connection between consistency states and constraint propagation in detail.

Controlling the search is another useful and crucial concept of constraint programming. This is realised by two forms of ordering decisions. The first is to determine the order in which the variables are assigned and the second is in which order the

values for each variable are assigned. This will be discussed in depth in Section 2.5.2.

2.3.1 The Solving Process

When searching for a solution the variables are instantiated consecutively. After each instantiation, infeasible values of all future variables are removed from their domain when they are detected by a propagation algorithm. This corresponds with pruning sub-trees in the search tree.

Example 2.9 *A pruned search tree*

Consider the CSP from Example 2.8. In the search tree in Figure triangle represent infeasible variable assignments. The nodes with the label 4 *and* 6 *are do not have children. That means that these nodes are not explored since their parent node was detected infeasible.*

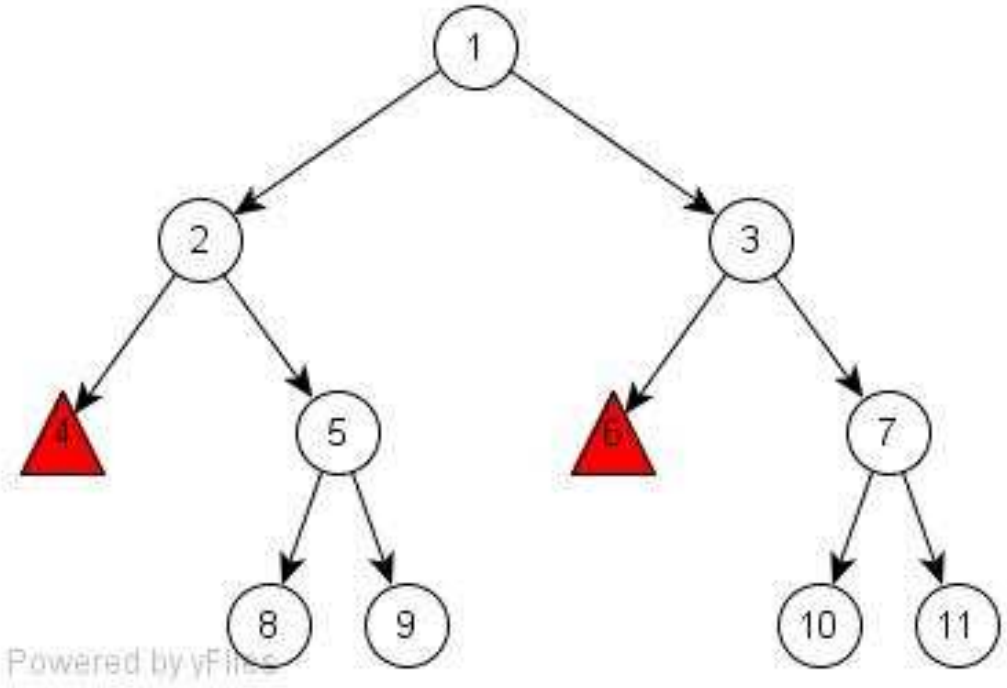


Figure 2.3: A search tree with pruned sub-trees

If a partial variable assignment is detected to be infeasible it is abandoned and the search performs *backtracking*. When backtracking the instantiation of the last considered variable is undone and the value is removed from its domain. This corresponds with stepping back one level in the search tree and delete the considered edge from the tree. When this is done the variable is instantiated again with a different value. If there is no other value left backtracking is performed again. By

doing so the whole search tree is traversed recursively(although not entirely since sub-trees are pruned for infeasible values in the domains).

Removing an edge from the search tree is done in the model by adding a new constraint to the constraint store. This constraint is an unary inequality constraint that states that this variable cannot take this value.

Also new constraints are added that prohibit the detected infeasible values. The constraints C and the domains D are altered according to the last instantiation:

- The instantiated variables in the constraints are replaced by their assigned value
- From a certain point of view the original CSP is replaced by a new CSP with each instantiation. The difference between the two is that the new CSP has tighter constraints on the variables

When the CSP is insoluble the corresponding partial assignment is infeasible and backtracking takes us to a previous CSP (modified by the constraint that the last assignment is infeasible).

Example 2.10 *A pruned search tree with backtracking*

Consider the CSP from Example 2.8. In the search tree in Figure the labels of the nodes correspond to the order in which the nodes are investigated. Triangles mark pruned subtrees and squares mark investigated infeasible assignments. Backtracking takes place in all nodes but these with the labels 3 *and* 13*. The former node is not investigated since it was pruned. The latter is the last assignment checked and search terminates. The nodes with the labels* $6, 8, 9$ *and* 10 *are marked infeasible. While* $6, 9$ *and* 10 *are leaves this is detected by evaluating them. the node labeled* 8 *is evaluated infeasible on backtracking since it cannot be extended to a solution. In contrast node* 4 *is labeled feasible since it can be extended to a solution (node* 5*).*

2.4 Consistency States and Constraint Propagation

In the last section we stated that infeasible values are removed from their corresponding domains. This process is called *constraint propagation*. Depending on the level of constraint propagation applied, a problem reaches different *consistency states*.

Constraint Propagation helps to reduce the search effort by pruning infeasible values from the search tree. Although pruning is vital it does not come for free. The different constraint propagation algorithms achieve different states of consistency. Basically the higher the level of consistency the more time and space expensive is the algorithm. Therefore, there is a trade-off between applying higher levels of consistency and the invested time to do so.

2.4.1 Consistency States

There are different levels of consistency for a CSP and the higher the level of consistency the tighter are the constraints of the CSP and the more infeasible variable

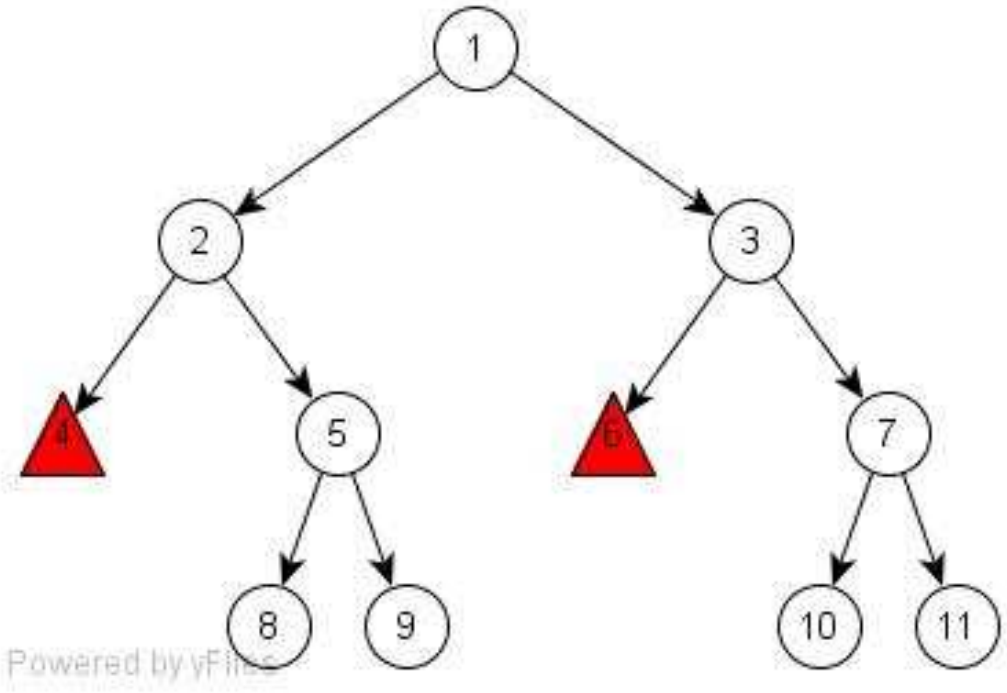


Figure 2.4: A search tree with pruned sub-trees

assignments are excluded. Basically the goal is to prune as many inconsistent values from the domains of the variables as possible. On a high level of consistency more values can be pruned since they can be detected infeasible, which is not always possible for lower levels of consistency.

The levels express the highest arity i of constraints that can be checked for consistency. Therefore, 3-consistency means that all constraints $c \in C$ with $|c| \leq 3$ can be made consistent. The higher the level of consistency that has to be achieved the higher are the costs in terms of time and space. The time and space cost for enforcing i-consistency is exponential in i [14]. The most regarded consistency states are Node, Arc and Path consistency (which corresponds to 1-, 2- and 3-consistency). In practice mostly arc consistency is applied for binary constraint systems [54].

2.4.1.1 Binary Constraint Systems

For the further definitions we consider the constraints in C to be binary at most. That means each constraint $c \in C$ is of the form $|c| = 1$ (unary) or $|c| = 2$ (binary).

Binary constraint systems can be represented as a graph where the nodes are the variables and the edges are the constraints connecting the two variables they are stated over.

Example 2.11 *A constraint Graph*

Consider a CSP with four variables $x_1, \ldots, x_4$ and the following constraints:

$$\begin{array}{lccc} c_1: & x_1 & < & x_2 \\ c_2: & x_1 + x_3 & = & 7 \\ c_3: & x_1 & > & x_4 \\ c_4: & x_2 \cdot x_4 & > & 8 \\ c_5: & x_3 & < & x_4 \end{array}$$

Figure 2.5 visualises the constraint graph of this CSP.

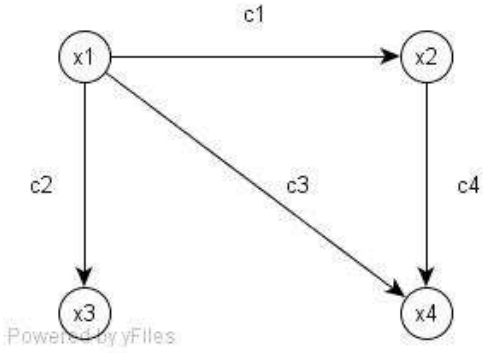


Figure 2.5: A constraint graph over 4 variables

A constraint graph is represented undirected since a constraint can be evaluated in both directions: A constraint $x_1 < x_2$ can also be seen as $x_2 > x_1$.

2.4.1.2 Node Consistency

Node consistency is the lowest state of consistency and is very cheap to achieve in terms of time.

Definition 2.12 (Node Consistency)

*A variable x is called **node consistent** if no value of its domain D_x violates any unary constraint $c_i \in C$, where c_i constrains x.*

If all variables are node consistent the constraint system is also node consistent.

Unary constraints are of the form $x\{=, \neq, <, \leq, >, \geq\}v$. Domain reduction therefore, is very simple. All values that violate these constraints are removed from their corresponding domain.

2.4.1.3 Arc Consistency

If the constraints of a problem are all binary the problem can be represented by a *constraint graph*. The nodes of the graph are the variables denoted by their domains. The edges are the constraints connecting the two variables of the constraint.

Definition 2.13 (Arc Consistency)

A variable x_i is called ***arc consistent*** *if for every edge in the constraint graph $\{x_i, x_j\}$ there exists for each value $v_i \in D_{x_i}$ at least one value $v_j \in Dx_j$ such that the constraint is satisfied.*

If for an edge $e = \{x_i, x_j\}$ the variables x_i and x_j are arc consistent the constraint representing e is arc consistent.

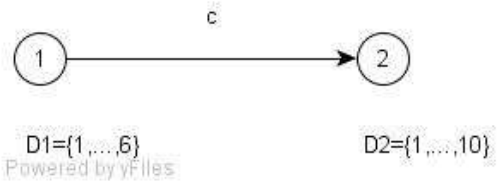
If all variables of a CSP are arc consistent, the constraint system is arc consistent.

The constraint graph representation in Figure 2.5 is undirected (since constraints are undirected). But both direction of an arc have to be checked in order to make the problem arc consistent.

Consider a constraint c_1 on the variables x_1 and x_2:

$$c_1 : x_1 + x_2 = 8$$

The domains of the variables are initially $D_1 = D_2 = \{1, \ldots, 10\}$. In the Figure 2.6 the arc representing c_1 is arc consistent in the direction $x_1 \rightarrow x_2$ since D_1 contains only feasible values. Since D_2 contains infeasible values the constraint is not consistent in the other direction.



Example 2.14

Figure 2.6: A constraint arc consistent in only one direction

In Figure 2.7 both domains contain only feasible values. The arc therefore is arc consistent in both directions.

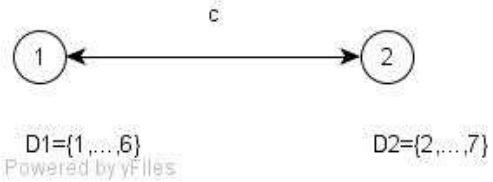


Figure 2.7: A constraint arc consistent in both direction

2.4.1.4 Path Consistency

At first glance it seems that arc consistency is powerful enough to removes all infeasible values in a binary constraint system. However, this is not the case.

Consider the following example:

Although all arcs are arc consistent it is obvious that this constraint system has no solution. But arc consistency would not detect the inconsistency. A higher level of consistency is needed to detect this inconsistency.

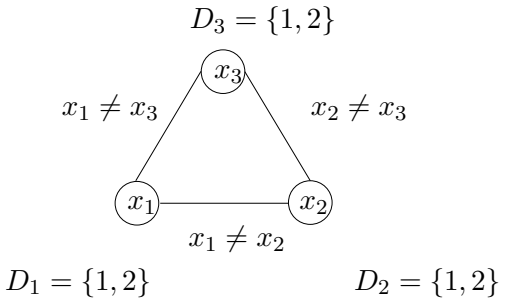


Figure 2.8: A binary constraint graph where arc consistency cannot detect infeasibility

Definition 2.15 (Path Consistency)

Consider a CSP with binary constraints.

A pair of two variables$\{x_i, x_j\}$ is path consistent relative to a variable x_k iff for every consistent assignment $(x_i = v_i, x_j = v_j)$ there exists a value $v_k \in D_k$ such that the assignments $(x_i = v_i, x_k = v_k)$ and $(x_k = v_k, x_j = v_j)$ are consistent.

Path consistency in the constraint graph can be seen as following a path from the node corresponding with the variable x_i to the node corresponding with the variable x_j.

2.4.1.5 Generalised Constraint Systems

For more general CSPs with n-ary constraints consistency is defined to suit these kind of constraints. There is a generalisation of node and arc consistency called *i-consistency*. Also there is a more generalised version of arc consistency itself called *generalised arc consistency*. Since reaching higher states of consistency is very expensive in time and space (even exponential) mostly i-consistency is not performed for $i > 3$. A weaker consistency but not that expensive is *bounds consistency*. This is often applied when the domains of the variables are large sets of integers. Propagation then is only performed on the bounds.

Since arc consistency is the most used consistency state that is propagated we show the generalisation of arc consistency.

Definition 2.16 (Generalised Arc Consistency)

Consider a constraint $c \in C$ that constrains the variables $v_1, \ldots, v_m$.

*A variable $x_i \in \{1, \ldots m\}$, is **generalised arc consistent** relative to c if and only if for every value $v_i \in D_i$ there exists a tuple of values $(v_1, \ldots, v_{i-1}, v_i, v_{i+1}, \ldots, v_n)$, such that c is satisfied.*

Definition 2.17 (i-Consistency)

Consider a constraint system C with an arity of at most i and a constraint $c \in C$ with $|c| = i$.

If all variables that are constrained by c are generalised arc consistent, the constraint is **i-consistent**.

If all constraints in C are i-consistent the constraint system is **i-consistent**.

Especially for large integer domains it is often not possible to maintain arc consistency for example since this would be too space exhaustive. In such a case bounds consistency is a good choice because it just checks the bound values of a domain for consistency.

Definition 2.18 (Bounds Consistency)

Given a constraint $c \in C$ and a variable $x_i \in X$ that is constrained by c.

If x_i is generalised arc consistent for the bound values $v_{i_{\min}}, v_{i_{\max}}$ of its domain D_i then x_i is **bounds consistent** *relative to c.*

If all variables are bound consistent with respect to all relevant constraint the constraint system is **bounds consistent**.

2.4.2 Constraint Propagation Algorithms

Consistency is achieved by propagation algorithms. There are different algorithms that aim at different levels of consistency. From a certain point of view propagation algorithms replace the original CSP P with a new equivalent CSP P' that is easier to solve [14]. That means that the number of decisions – and thereby the search space – is reduced from P to P'. Often it is possible to deduce a solution directly from P' or detect infeasibility of P'. A solution to P' is also a solution to P since both CSPs are equivalent. The same holds for infeasibility.

In practice very expensive propagation algorithms are not applied since often the time invested for the additional pruning does not pay-off. That means that the amount of time saved by the additional pruning is smaller than by investigating these nodes explicitly.

The algorithms applied most often are for node consistency (which is very simple and fast), arc consistency/generalised arc consistency and bounds consistency.

There are also propagation algorithms (called *global constraints*) that can be used for problems with special properties. These problems are often subproblems of larger problems and do occur very often. A global constraint encapsulates an efficient algorithm that takes advantage of the special problem properties. Thereby, more pruning is achieved or pruning takes just a fraction of the time compared to standard propagation algorithms. But a global constraint can only be applied for problems with this special properties. Global constraints are investigated in Section 2.4.2.4

We will concentrate in this chapter on the algorithms for binary constraint systems since they are the ones with the most attention in research. Also during search the ary of constraints decreases temporarily. This is due to the fact that instantiated variables in a constraint act like a constant. Therfore a n-ary constraint becomes a $(n-1)$-ary constraint if one of its variables x_i is instantiated. This effect is temporarily since it is reversed on backtracking from x_i.

Example 2.19 *Consider variables $x, y \in \{1, \ldots, 5\}$ and a constraint $c : x + y < 8$. If x is instantiated to $x = 3$ for example, c changes to $c' : y < 5$.*

Notation 2.1 *To check feasibility of a constraint c and a partial assignment to the variables $x_1, \ldots, x_k$ we use the following notation:*
$c(x_1 = v_1, \ldots, x_k = v_k) = true$ if the partial variable assignment is feasible with the constraint.

2.4.2.1 Node Consistency Algorithm

Node consistency is established by checking the domains of all variables that are constrained by a unary constraint. Values that are forbidden by a unary constraint are removed from the domain of the variable. When all unary constraints are evaluated in this way they can be removed from the constraint system since they will have no further effect on the feasibility of a variable assignment.

Node consistency algorithms can be applied in a preprocessing step before search. They also can be again applied to n-ary constraints that decreased their arity to unary during the search.

Node Consistency
Input: Variable x
Constraint c_i
Domain D_x
Output: Domain D'_x
begin
  $D'_x = \emptyset$;
  for $d \in D_x$ **do**
    if $(c_i(x = d))$ **true then**
      $D'_x = D'_x \cup d$;
    end
  end
  return D'_x
end

Algorithm 1: Node Consistency Algorithm

The idea is to check for each value in the domain, whether it satisfies the constraint. If so it is in the new domain of feasible values D'_x and not otherwise.

This procedure is applied for all unary constraints in the constraint system. The worst-case runtime to achieve node consistency for the CSP is $\mathcal{O}(|C| \times |D_{max}|)$, where D_{max} is the largest domain of all variables that are unary constrained.

This runtime only applies if the constraint system consists only of unary constraints. In this case the remaining values in the domains are all feasible and any combination of these values is feasible.

2.4.2.2 Arc Consistency Algorithm

The most crucial change from node to arc consistency is that constraints have to be regarded multiple times in general. When the domain of a variable is altered all constraints that involve this variable have to be checked anew since this could allow further pruning.

There are different AC algorithms which differ by their worst-case runtime. While the first on AC-1 has a time-complexity of $\mathcal{O}(|X| \cdot |C| \cdot |D^3_{max}|)$ AC-4 has a time and space-complexity of $\mathcal{O}(|C| \cdot |D^2_{max}|)$ [14]. We will show here a very straight forward algorithm for simplicity.

In contrary to the node consistency algorithm a change in a domain of one variable can have impact on the domain of the other variable. Therefore, the procedure

Arc Consistency Algorithm
Input: Variables x, y
Constraints $C = \{c_1, \ldots, c_k\}$
Domains D_x, D_y
Output: Domains D'_x, D'_y
begin
  $D'_x = D_x$;
  $D'_y = D_y$;
  repeat
    $\tilde{D}_x = D'_x$;
    $\tilde{D}_y = D'_y$;
    for $c \in C$ **do**
      $D'_x = \text{directedArc}(x, y, D'_x, D'_y)$;
      $D'_y = \text{directedArc}(y, x, D'_y, D'_x)$;
    end
  until $(D'_x = \tilde{D}_x \vee D'_y = \tilde{D}_y)$;
  return D'_x, D'_y
end

procedure directedArc(l, r, D_l, D_r)
begin
  $D'_l = \emptyset$;
  for $d_l \in D_l$ **do**
    for $d_r \in D_r$ **do**
      if $(c(l = d_l r = d_r))$ **true then**
        $D'_l = D_l \cup d_l$;
      end
    end
  end
  return D'_l
end

Algorithm 2: Arc consistency algorithm

directedArc() has to be called (for both directions of the edge) until no more propagation took place.

The worst-case runtime is $\mathcal{O}(|C| \cdot |D_{max}|^3)$. The procedure directedArc() has a runtime of $\mathcal{O}(|D_{max}|^2)$. It is called for each constraint in the constraint system which can be $|C|$ times. In each such a loop only one value may be removed from a domain. Therefore there could be $2 \cdot |D_{max}|$ such loops.

2.4.2.3 Path Consistency Algorithm

Although arc consistency performs very good on propagation it cannot detect all infeasibilities as already seen in the Example 2.8.

This problem is not soluble since there are only two different values for three variables that have to take all different values. Although this is obvious arc consistency is not able to detect the infeasibility. This is due to the fact, that each arc is satisfied since there exists a value for each variable such that the constraint that represents the arc is satisfied.

To detect this infeasibility we need a path consistency algorithm. While in arc consistency only one arc is checked, a path (a connection of arcs) is checked simultaneously. That means that on a path $x_1 - \ldots - x_n$ for a value $d_1 \in D_1$ there must exist values $d_2, \ldots, d_n, d_i \in D_i$, such that the value assignments $(x_i = d_i, i \in \{1, \ldots n\})$ satisfy all the constraints that imply the path $x_1 - \ldots - x_n$.

If we would check the path $x_1 - x_2 - x_3$ in the Example 2.8 it had been clear that there is no value assignment that satisfies the constraint system.

2.4.2.4 Global Constraints

A global constraints is a more complex constraint using a special purpose algorithm for propagation. Global constraints are available for very common subproblems that are present in various problems and applications. The idea behind global constraints is that a well known problem (or subproblem) can be solved more efficiently using a more sophisticated and specialised algorithm than performing standard propagation algorithms.

The drawback in global constraints is that they can just handle a specific subproblem and also have to be implemented (unless they are part of the solver distribution which holds for the most popular ones).

Global constraints do not enrich the modelling features of a modelling language. They are designed for faster pruning. It is possible to model a global constraint using standard constraints (which is shown on Page 49) but pruning with standard propagation algorithms is poorer.

Propagation by Global Constraints: In general as described in Section 2.3.1 after each variable instantiation propagation algorithms are called that try to reduce the domain of the future variables. When a global constraint is used, this is different. The propagation algorithm is encapsulated and triggered by the use of a global constraint instead of the standard propagation algorithms [5]. After a variable – on that a global constraint is stated – is instantiated instead of calling the propagation algorithm the algorithm encapsulated in the global constraint is called first. The

algorithm then evaluates which values of future variables are inconsistent and deletes them from the corresponding domains. The difference between the behaviour of a standard propagation algorithm and a global constraint is that the global constraint uses other more problem-tailored methods to check for inconsistent values. That speeds up the propagation process sometimes considerably.

Examples for global constraints are *alldifferent*, *atleast*, *distribution*, *cumulative* etc [10].

Global constrains are developed for common subproblems in CSPs. For example many problems like the TSP or the Rehearsal Problem [82]. contain permutation of some variables or all the values of the variables must be pairwise different (alldifferent). The idea is to use an efficient algorithm and re-use it in several problems and applications such that the effort pays off.

Basically it is possible to use an algorithm for every subproblem. But not for all such problems there is an efficient algorithm known and there is no use in writing sophisticated algorithms for a subproblem that is only present in a very specialised kind of problems. One strength of global constraints is the re-usability in different applications.

2.4.2.5 Example: *alldifferent*

The alldifferent global constraints `alldifferent`$(x_1, \ldots, x_n)$ is defined over a set of variables $X = \{x_1, \ldots, x_n\}$. The constraint is satisfied if and only if all variables in X are assigned pairwise different values.

This can also be modelled using basic constraints:

$\forall i, j \in \{1, \ldots, n\} : x_i \neq x_j$

But due to the mass of constraints that are posted propagation would take a long time.

One implementation for the `alldifferent` constraint is using a matching algorithm [77].

We have a sets T_1 containing the variables X and a set T_2 containing the domain of the variables $V = \{v_1, \ldots, v_m\}$. (In the case that $m = n$ this is a permutation such that the `alldifferent` constraint is also applicable for permutations). There are no edges between vertex' in T_1 and also not in T_2. But every vertex in T_1 is connected to every vertex in T_2 by an edge. Whenever a variable x_i is assigned a value v_j all the edges $(x_k, v_j), k \in \{1, \ldots, n\}, k \neq i$ are removed from the graph. This corresponds with removing the value v_j from the domain of all other variables.

The gain now is that the propagation of the `alldifferent` constraint can also be used to propagate the domains of the variables even further by the used "standard"propagation algorithms.

2.5 Search Strategies and Ordering Heuristics

Although the concept of constraint propagation leading to several layers of consistency and powerful pruning algorithms is one of the reasons for the success of constraint programming often applying consistency algorithms alone is not efficient

enough to solve a problem. Even when all inconsistent values are removed often there still remains an overwhelming number of potential solutions in the search space. This *richness* of solutions is often deceptive since especially in scenarios like optimisation or time-limited search the task is to find a *good* solution soon. Therefore, just traveling through the search space could be too inefficient. Instead the search must be guided to promising parts of the search space.

This is done by ordering heuristics which decide the order of the variables and the order of the values that are considered for each variable.

There are some general heuristics that can be applied but also crucial knowledge about the features of a solution – if at hand –can be used as a heuristic. Using this knowledge it is often possible to assign some variables with certain values that are most likely in a solution while the rest of the variables are considered following a standard heuristic.

Therefore, it is desirable that the user have some control on the search process. And that's exactly what's the second benefit of constraint programming. It is possible to control the search for a solution.

This is very contrary to the concept of other solving approaches. But it is very dearly needed since other solving approaches are based on powerful, often problem-tailored algorithms that already exploit all implicitly known knowledge of the problem. But since constraint programming is capable of various problems the solving algorithms are very limited. Therefore, interaction from the user is needed to close the gap of efficiency with other solving approaches.

Basically, controlling the search is done by using a heuristic that investigates more promising regions of the search space first. Although specifying the search heuristic is done by the user there are several techniques used implicitly by the solver to propagate the constraints. Therefore, searching in a CSP consists of two techniques:

- search strategy
- search heuristic

The search strategies are implemented in the solver and several different strategies can be chosen. The efficiency of a strategy is not influenced by the problem because the strategy just defines the actions done at each node in the search tree. Although more sophisticated strategies achieve mostly better performance in terms of pruning the search tree they often have a higher complexity and The main concepts of search strategies are backtracking, forward checking, back jumps and incomplete stochastic search.

The search heuristic is specified by the user. The idea is to use implicit knowledge of the problem and try to steer the search in the most promising direction of the search tree. Unlike the search strategy a heuristic most be adjusted to the problem. Often every problem needs a new problem–tailored heuristic (although more general heuristics exist and can be applied) to achieve best results. The heuristic basically defines at each node in the search tree which node is considered next. Basically there are two kinds of meta-heuristics that can be combined. One is the variable ordering which states which variable is considered next (this is not available in static search trees, where the order of the variables is already fixed). The other is value ordering which states which value of the variable next considered is chosen. Each of these two meta-heuristics have several heuristics that are shortly explained in this section. The techniques presented in this section can be found in [87] for example.

2.5.1 Search Strategies

As mentioned before the search strategy states the behaviour in each node of the search tree. We will start with the most basic strategy used.

2.5.1.1 Backtracking (BT)

Backtracking is the simplest technique and the backbone of a constraint solver. The idea is that whenever a partial assignment is detected infeasible the actual partial solution is abandoned (since it cannot be extended to form a feasible solution) and the last partial solution that is feasible is considered. Since feasibility is checked after each instantiation that means that the search backtracks from this node to its father node.

Consider a partial solution $(x_1 = v_1, \ldots, x_k = v_{k-1})$ that is feasible. When the next variable x_k is instantiated the partial solution $(x_1 = v_1, \ldots, x_{k-1} = v_{k-1}, x_k = v_k)$ is checked for feasibility. To detect infeasibility only constraint containing variables of the partial solution are considered, i.e. all variables in these constraints are assigned. If the partial solution is infeasible the last assignment $x_k = v_k$ is undone and the constraint $x_k \neq v_k$ is imposed and the next value $v_{k'}$ for x_{k+1} is considered. If there is no such value $v_{k'}$ the domain of the variable x_k is empty and the partial solution $(x_1 = v_1, \ldots, x_k = v_{k-1})$ is evaluated infeasible and the search backtracks.

This way the whole search tree can be investigated.

2.5.1.2 Forward Checking (FC)

The problem with simple backtracking is that an inconsistency is only detected *after* the instantiation of a variable. It would be helpful if that inconsistency would have already been discovered *before* the variable is instantiated. This is exactly what forward checking does.

Forward checking does not only reason on constraints containing the variables of the already assigned variables but also on constraints containing not instantiated variables (called *future variables*). Thereby inconsistencies can be detected before the instantiation. By doing so, constraints have to be considered that include variables of the partial solution. Therefore, it is checked whether the partial solution is feasible with all other future variables. If infeasibility on one or more of the future variables is detected the search backtracks as described in the backtracking section above.

The following example illustrates the difference between simple backtracking and forward checking.

2.5.1.3 Partial Look Ahead

A technique that goes further is the partial look ahead technique. In this technique it is not only checked whether the partial solution is feasible with the future variables. Moreover it is checked consecutively for all future variables x_i whether their values are consistent with the future variables on the lower levels $x_j, j > i$ of the search tree. If not the inconsistent values are removed from the domain of the future variable x_i. In this technique the search tree must be static since each variable is only checked with variables in layers below that layer (never above).

By doing so we are able to exclude values of future variables that would result only in infeasible solutions.

Consider the left chessboard in the Figure 2.9 for an example.

2.5.1.4 Maintaining Arc Consistency (MAC)

MAC now is even more generalised and also called *full look ahead*. In MAC the future variables are checked for inconsistent values. But each future variable is checked against each other for inconsistent values. In MAC the search tree may be dynamic since the order of the variables does not matter anymore for using MAC

Example 2.20 *Consider the n-queens problem and a partial variable assignment as can be seen in the Figure 2.9. On the left chessboard only the fields that are attacked by the queen can be pruned from the future variables (Forward Checking). On the chessboard in the middle also the field* 3 *can be pruned from the variable* x_d *since a queen on this position would attack all remaining fields in the row* e*. This is detected since the variable* x_e *is a future variable below the layer of* x_d *(Look Ahead). On the chessboard on the right also the field* 4 *of the variable* x_c *can be removed, since a queen on this field would attack all remaining filed in the row* b*. This is detected since all future variables* $(x_b, \ldots, x_e)$ *are checked against each other future variable (MAC).*

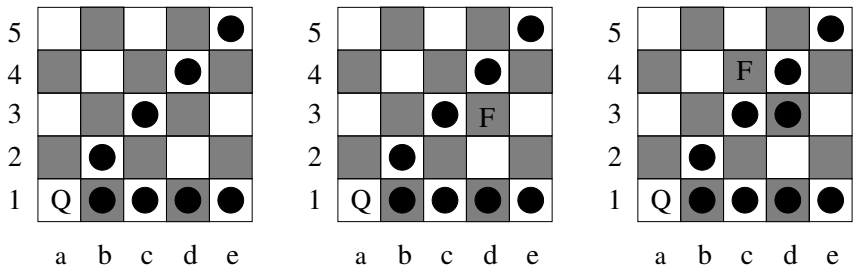


Figure 2.9: Left: Simple backtracking Middle: Forward Checking Right: Full Look Ahead

2.5.1.5 Complexity of Search Strategies

Although the more sophisticated search strategies like partial or full look ahead are more successful in terms of domain reduction they imply higher costs on the runtime.

Consider a problem with n variables and the first variable is instantiated. We investigate how many variable checks have to be done in the worst case for the individual propagation techniques.[2]

For backtracking there are no checks to do. Forward checking requires up to $n-1$ variable checks. Partial look ahead requires up to $n-1+n-2+\ldots+1 = (\frac{n\cdot(n-1)}{2})$ variable checks. MAC requires up to $(n-1)^2$ variable checks. As one can see the number of variable checks rise considerable with the different techniques. But the

[2] A variable check is a check between two variables x_i and x_j where it is checked whether values of x_i can be pruned. We do not take into account how many values are left in the domain since this cannot be predicted due to propagation success.

hope is that the achieved pruning of the techniques save more time than is invested by the search.

The gain in the more expensive propagation techniques is that inconsistencies are detected earlier in the search tree. Therefore, the overall success (of saving time by using these strategies) depends on the achieved reduction of the search tree.

2.5.1.6 Stochastic Search

When using stochastic search, the search is incomplete. In most scenarios the task is to find an optimal solution or prove that there is no solution to the problem. In theses scenarios stochastic search is not applicable since the results cannot be proven correct in all cases. (Solutions may be not optimal or there is a solution although the search did not find it). Nonetheless, there are scenarios that profit from this search strategy. Consider optimisation within a given time limit. The task here is to find a solutions within a given time limit and choose the best found so far. By using stochastic search strategies the search space may be investigated in a way that whenever the current partial solution looks *promising* it is further instantiated and if not it is abandoned. Promising means for example that the partial solution respects the given search heuristic to some extend or that only a certain number of backtracks are allowed on a specific level, etc.

2.5.2 Ordering Heuristics

Controlling the search is vital in CP since there is no sophisticated solving algorithm like simplex for linear problems. A CP solver relies on the power of pruning and good search heuristics. Therefore, one task of the modeller is to find a search heuristic such that pruning can efficiently decrease the search.[3] Without specified ordering heuristics default heuristics are considered. Which heuristics these are depends on the used solver.

As mentioned before heuristics as well as additional constraints are useful to apply implicit knowledge of the problem. That means that a solution is most likely to respect certain characteristics. In this case the heuristic would be used to build partial variable assignments that are more likely part of a solutions rather than assigning variables and values blindly. But even if no such knowledge is at hand heuristics can help to speed up the search for a solution. They aim on achieving propagation in an early stage of the search and reducing the search tree thereby further. It has to be mentioned that the heuristic itself does *no* propagation. But the idea is to consider a variable or a value that either fits best or is likely to achieve further propagation.

Ordering heuristics are defined for variables and their values. Mostly they are strongly connected such that the one cannot be considered fruitfully without the other. We will first state the meaning of variable and value heuristics and then show some prominent search heuristics that work efficiently.

[3]Note hat maximal pruning is not the main target but reducing the time to find a (optimal) solution or prove that there is none.

2.5.2.1 Variable Ordering

The variable ordering is concerned with the question which variable is the next in the search tree. In static orderings the variable ordering in determined before the search starts and does not change during search. In a static search tree each layer represents a specific variable. This is not necessarily the case for dynamic orderings where the order of considered variables may change during the search.

Fail First Principle

"To succeed, try first where you are most likely to fail [39]."

The idea is to consider among a set of decisions to make the one decision which is most difficult. This corresponds with solving the difficult part of the problem first before dealing with the easy part. Since decisions in the difficult part may be wrong the search has to backtrack. But this is done at an early state in the search. For wrong decisions large subtrees can be pruned. If the easy decisions are done in the beginning the search ends up with always backtracking at the same conflicts for the difficult part of the problem. This may happen in *every* subtree because the decisions done at the beginning of the search are infeasible for the difficult part which cannot be detected.

There are several heuristics that respect the fail first principle. We describe the smallest domain first heuristic here but there are several others [83, 9]

Smallest Domain First: This heuristic considers a variable with the smallest domain next. The idea is that instantiations on such a variable has more impact on the search. If a variable has just a view values in its domain left the chance is higher that the domain becomes empty by propagation of other variable assignments. Therefore, a lot of backtracking is needed. Even if the assigned value is not feasible the hope is that this is detected early and the search can prune early without searching the whole subtree.

A different heuristic which respects the fail first principle is the most constrained variable first heuristic.

Most constrained variable: The idea is mainly the same as in the smallest domain first heuristic. A variable that is very constrained is assigned first such that either other variable assignments may not lead to an empty domain of this variable or the instantiation may result in more propagation. Since the variable is part of many constraints chances are good that an assignment either achieves a lot of propagation reducing the search tree further or detect the infeasibility of this variable (which means that the partial assignment is infeasible).

2.5.2.2 Value Ordering

When the decision which variable to considered is done the next question is what value to assign to this variable. The idea is to order the values in a way such that a branch that is more likely to contain a solution is searched before all other. Note that when all solutions are sought for a CSP then the value ordering is immaterial since the whole search tree has to be investigated.

In the succeed first principle a value is assigned that may most likely result in a feasible solution. There are ideas how to combine these ideas with forward checking to decide which is the best value to consider[28, 53].

Since ordering strategies are just heuristics there are counterexamples for each heuristic. Even for different instances of the same problem the same heuristic may perform different. Nonetheless the use of heuristics is vital to find solutions early or prune the search tree such that it can be traversed in acceptable time.

2.6 Modelling

The representation of a problem is a serious issue in constraint programming. A CSP is represented by a model. This model is fed into the constraint solver to be solved. There are different more or less equivalent models for a problem. All these models are equivalent in terms of the input and output they receive and produce, i.e. the problem they solve. But they may use different variables and constraints with different semantics. The model used has a drastic effect on how easy it is to find a solution for the problem [77]. Although modelling is solver-independent some techniques present in one solver may not be supported by all solvers. An example is the concept of a set variable [35] that can take several values instead of one and represent therefore, a set rather than a single value. This feature has evolved during time and proved useful such that it is now used in most solvers.

Modelling is not just the way to represent a model. There are several different models describing the same model and often some are more efficient than others. Therefore, modelling is a crucial part of the process to find a solution.

2.6.1 Representation of a Problem and Different Viewpoints

As mentioned before a problem can be modelled in different ways. The diversity for different models originate from different viewpoints on the problem mostly. The term *viewpoint* is informally introduced by Geelen [28] and subsequently adopted and formally defined by Law and Lee [55, 77].

A viewpoint is a pair (X, D), where X is a set of variables and D is the set of domains for these variables (analogously to the definition of a CSP). An assignment to a variable $x \in X$ has a special meaning for the problem in terms of the viewpoint (X, D). Constraints on X ensure that every solution of the problem is also a solution to the model expressed by the viewpoint (X, D) and the resulting constraints. Different viewpoints now lead to fundamentally different models of the problem.

Example 2.21 *n-Queens Problem*

First Viewpoint: Variables for each row, value represents the column

Consider the following two viewpoints for the n-queens problem. In the first model the viewpoint $v_1 = (X_1, D_1)$ is expressed by $X_1 = D_1 = \{1, \ldots, 8\}$. Each variable represent a row of the chessboard and each value in the domain represent a column of the chessboard. So the assignment $x_2 = 4$ means that a queen is placed in the second row and the fourth column of the chessboard. The constraints now have to ensure that only one queen is in each column (no value is assigned more than once in X) and on each diagonal each only one queen is placed.

Second Viewpoint: Variables for each field, value represents a queen (or no queen)

In the second model the viewpoint $v_2 = (X_2, D_2)$ is expressed by $X_2 = \{1, \ldots, 64\}$ and $D_1 = \{0, 1\}$. Here each field of the chessboard has its own variable and the

assignment $x_4 = 1$ means that a queen is placed on the field represented by x_4. In addition to the constraints induced by v_1 there must also be a constraint that states that in each row only one queen is placed. Also the number of queens has to be restricted to 8. All constraints can be represented by sums over the according variables. For example the constraint for the first row is $\sum_{1 \leq i \leq 8} x_1 = 1$.

As one can see the two viewpoints v_1 and v_2 induce totally different constraints. While the set of constraints for v_1 is a subset of the set of constraints for v_2 the constraints expressing the same information have to be represented different.

In the first viewpoint X and D could also be permuted, such that X represents the columns and D the rows. But due to the symmetry of the chessboard (See Section 3.1.3.2) they yield the same constraints and therefore these viewpoints are identical.

Sometimes also a viewpoint imply different constraints. As we have seen in the example above the viewpoint v_1 can be seen row-wise or columnwise (variables represent rows or columns). Although the constraints in this case are identical the semantic is different. Still beyond symmetry of viewpoints a model induced by a viewpoint can be changed without changing the underlying problem. This is the case when *implied constraints* are added to the set of constraints (See Section 2.6.2.2) or global constraints are used instead of basic constraints (See Section 2.4.2.4).

2.6.2 Auxiliary Variables, Implied Constraints, and Channeling Constraints of Dual Models

Additional variables and constraints are added to the model without changing the meaning, i.e. the set of solutions of the problem. This can be done by adding variables and/or constraints that express a different aspect of the problem and are used to help making decisions for the original variables. There are a lot of successful approaches where adding variables and/or constraints to the model did improve the search process drastically [81].

The intention of adding data to the model is generally to improve the search process. Although more variables and constraints lead to a delay in the process of the search (there are more variables to instantiate and propagation takes longer since there are more constraints to check) this is more than compensated by the additional pruning these data achieves.

2.6.2.1 Auxiliary Variables

Auxiliary variables are introduced to a model because it is difficult to express the constraints on the existing variables or make it possible that the constraints can be expressed in a way that they propagate better [81]. For a successful application consider [15].

2.6.2.2 Implied Constraints

Implied constraints can be deduced from the existing set of constraints. They do not change the set of solutions and therefore are logically redundant. That is why they are also called *redundant* constraints. Implied constraints are added to make

implicit knowledge explicit to the solver such that propagation is more effective. But not all implied constraints are useful or do help propagation. For example consider the constraints: $x_1 < x_2$ and $x_2 < x_3$ the constraint $x_1 < x_3$ would be an implied constraint. But it will not help propagation because all pruning would also be achieved by the arc consistency algorithm. Therefore, the task is to find implied constraints that reveal structure of the problem and pass it to the solver. Again consider [15] for a successful application.

2.6.2.3 Channeling Constraints

For some problems it is useful to use two different viewpoints at the same time. That means that roughly two models are represented in one where each alone is a valid description of the problem. These two viewpoints are not connected at first. To achieve a connection between them channeling constraints are used. These constraints are used to propagate knowledge from one viewpoint to the other. That means that for example propagation is the first model can also be used for propagation in the second viewpoint. The advantage of this approach is the extra propagation that comes from the models. Also this offers a richer modelling opportunity. Often it is very inconvenient (or costly in terms of time and propagation) to state all constraints for a certain viewpoint. These constraints could perhaps be stated more efficient in different viewpoint. Therefore, these viewpoints are combined in a model and channeling constraints are used for linking both viewpoints to achieve a better propagation.

2.6.3 Choosing a Model

Choosing a good model for a problem is vital for the solving process. As we have seen in this section there are many modelling decisions to make and it is by far not known in the beginning which decisions will be helpful and which not. There are rules of thumb for modelling from experienced experts like Simonis *et al.* [78] and Smith [81]. But Smith also points out that these guidelines are worth discussing and should not be taken literally.

Still modelling is one of the key features that determines success or failure of a constraint programming approach for a problem. Also researchers of other areas such as operations research [86, 93] state that modelling is a crucial part of finding a solution to a problem.

Chapter 3

Symmetry

Symmetries are motivated by geometric reflections and rotations of objects. The feature of a symmetry thereby is that the reflected object is indistinguishable from the original object. The reflection of a square is still a square for example. In general a symmetry describes a transformation of an object in another object with the property that both objects are indistinguishable. Symmetry is a relation between objects. Two objects are symmetric if they are indistinguishable.

For example two cars are symmetric if they are of the same model. In production items that are produced of often symmetric or the machines used to manufacture items are identic.

We are concerned with symmetry as a transformation of an object that does not change the relevant properties of the object. While symmetry is beautiful in nature it is troublesome in solving CSPs. The reason for the trouble with symmetries in a CSP is that they slow down the search dramatically. For the search process two variable assignments with different assignments are different from each other. But we will see in this chapter that two symmetric variable assignments can be transformed in each other. That means although they have different values assigned to the variables they express the same state and there is a function that transforms them in each other. Therefore, if we have investigated one variable assignment we already know the status (feasible or infeasible) for the symmetric pendant(s) and we are not interested in investigating it again explicitly by the search.

The problem with symmetries in CSPs is that equivalent search states are visited over and over again by the search [34]. This leads to a waste of time. Fortunately it is possible to remove symmetry from a CSP without losing information. This process is called *symmetry breaking*. It is done by exploiting the symmetry in a way that ideally only one of all the equivalent states is visited and all the others are excluded from the search. There are several different techniques how to deal with symmetry and its exploitation. The background for the term symmetry breaking dates back to the most historically used technique of adding constraints to the original model that prohibit (ideally all) symmetric solutions. The new model with the so called symmetry breaking constraints added have a smaller number of symmetries or none at all.

Symmetry breaking gives the enormous potential benefit of reducing the search space to its combinatorial core and reduces it thereby. This helps overcome one of the crucial problems of constraint programming: solving large problems in reasonable time.

In this chapter we will investigate symmetries in CSPs in Section3.1, where we define symmetry and state some problems that comprise symmetries. We will also investigate the group theoretical background of symmetry in Section 3.2 which gives us the justification and proof that symmetry breaking is a sound technique. Furthermore we will see in Section 3.3 some techniques of symmetry breaking used.

3.1 Symmetry in CSPs

3.1.1 Symmetry Definition

A symmetry can be thought of as a transformation of an object which preserves the relevant properties of the object. That means before and after the transformation the object has the same features and obeys the same constraints.

In geometry for example consider the rotation of a chessboard on 180 degree. The rotated chessboard is indistinguishable from the original board. In production it does not matter in which order identical objects are are produced. So any permutation of a sequence is indistinguishable.

For a long time there have been several definitions for symmetry in the CP community and unfortunately they were not equivalent.

There are two basic types for symmetry definitions in CSPs: those that see symmetry as a property of the solutions and those that see symmetry as a property that can be identified in the statement of the problem, without solving it [11].

For the purpose of this thesis we choose the definitions of the chapter symmetry in constraint programming from [77].

Definition 3.1 (Solution Symmetry)

A solution symmetry is a permutation of the set of ⟨variable, value⟩ pairs which preserves the set of solutions.

Definition 3.2 (Problem Symmetry)

A problem symmetry is a permutation of the set of ⟨variable, value⟩ pairs which preserves the set of constraints.

Cohen *et al.* [11] give a more rigorous definition of the two forms of symmetry and furthermore show the difference between the two definitions in practice.

Throughout this thesis a symmetry basically is a function that maps a variable assignment to a different but equivalent one. This mapping is basically a permutation and the feature is that the state of the variable assignment (feasible or infeasible) is not changed by the transformation.

3.1.2 Modelling

The design of the model for a CSP has a substantial impact on the solving efficiency. An appropriate reformulation of a model can turn an insoluble problem into a soluble one in practical terms [81]. Also different models can have different symmetries and it is possible that the symmetry in one formulation can be handled easier than in a

different one. Also the number of symmetries may be reduced by a different model. It is also possible to prevent symmetry at all by reformulation (using a different model).

Example 3.3 *General Subset Problem*

Consider a problem where you have to choose k among n pairwise different items due to some restrictions. One possible model would be to have a variables $x_1, \ldots, x_k$ each with a domain $D_i = \{1, \ldots, n\}$. In this model there is a symmetry in the sequence of the values chosen. Therefore, if $(2, 1, 4)$ is a solution then also $(4, 2, 1)$ is a solution since both just denote that the first, the second and the fourth item is chosen.

A different model would be to introduce boolean variables $x_1, \ldots, x_n$ each with a domain $D_i = \{0, 1\}$. This way for each of the item i it is denoted whether it is chosen ($x_i = 1$) or not ($x_i = 0$).

Using the second model the permutation of a solution is broken.

Again, modelling is one of the key features of constraint programming and proves its usefulness also in symmetry breaking.

3.1.3 CSPs with Symmetries

In this section we will investigate some well-studied academic problems that contain symmetries. Some of these problems are not only academical but have strong relations to real-life problems. Depending on the representation of the problem there may be different symmetries. Although we are not interested in all possible models or the most efficient representation of the problems we will discuss different models for a problem if appropriate.

3.1.3.1 Matrix Models

A *matrix model* is a representation of a CSP with one or more matrices of decision variables [54]. The structure of the matrix (columns and rows) is transfered to the variables in the matrix. Therefore, operations on a column affect all variables in this column and likewise for rows.

Example 3.4 *Loading Problem Consider that m packages have to be loaded in n trucks. A possible model of the decision variables is to have a boolean decision matrix $A^{m \times n}$ where each column represents a specific truck. Each row stands for a specific package. A 1 at the position a_{ij} means that the package i is transported by truck j. There are constraints that restrict the possibilities of loading the trucks which we are not concerned any further for this example.*

Consider the following solution for an instance with six packages $p_1, \ldots, p_6$ and three trucks t_1, t_2, t_3.

$$A = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \end{pmatrix}$$

This represents the solution that the packages number two and three are loaded in the first truck, the first, the fifth and the sixth are loaded in the second truck and the fourth package is loaded in the third truck.

Many matrix models comprise row and/or column symmetries. That is a permutation of the columns or the rows or a combination of them. In matrix models with column or row symmetries the number of symmetric solutions are enormous. A matrix with a column symmetry and n columns comprises $n!$ symmetric solutions. In combination with a row symmetry and m rows this number increases to $m! \times n!$ symmetric solutions.

Example 3.5 *Symmetries in the Loading Problem*

A permutation of the column represents that the trucks change place in the presentation. This results in a different loading of the trucks. If for example the first and the last truck are permuted then the first truck will be loaded just by the fourth package (the original loading of the last truck) and vice versa. But since the trucks are identic this doesn't matter.

A permutation of the row represents a rearrangement of the packages.

But the crucial property of the solution – that the same number of packages are loaded together in a truck – is not lost by a permutation of the columns and/or rows. Therefore, any permutation still represents the same solution.

A lot of problems can be modelled as a matrix model. In this section most of the problems can be modelled using a 1 or 2 dimensional decision variable matrix. From a certain point of view nearly every model is a matrix model. In the extreme case the matrices are just vectors such that there is only row *or* column symmetry.

3.1.3.2 n-Queens Problem

In the n-queens problem the task is to place n queens on a $n \times n$ chessboard, such that no two queens can attack each other. See Figure 3.1.

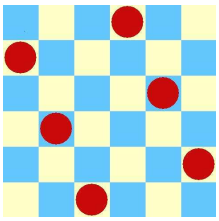


Figure 3.1: A feasible placement of 6 queens on a 6×6 chessboard.

This problem has the eight symmetries of the chessboard (including the identity). It can be turned on 90, 180, 270, 360 degree. Also we can reflect the chessboard about the horizontal axis, the vertical axis and both of the diagonal axis. See Figure 3.2.

3.1.3.3 Magic Square Problem

In the magic square problem, the numbers $1, \ldots, n^2$ have to be assigned to a $n \times n$ square such that the sum of the numbers in each row, in each column and in both

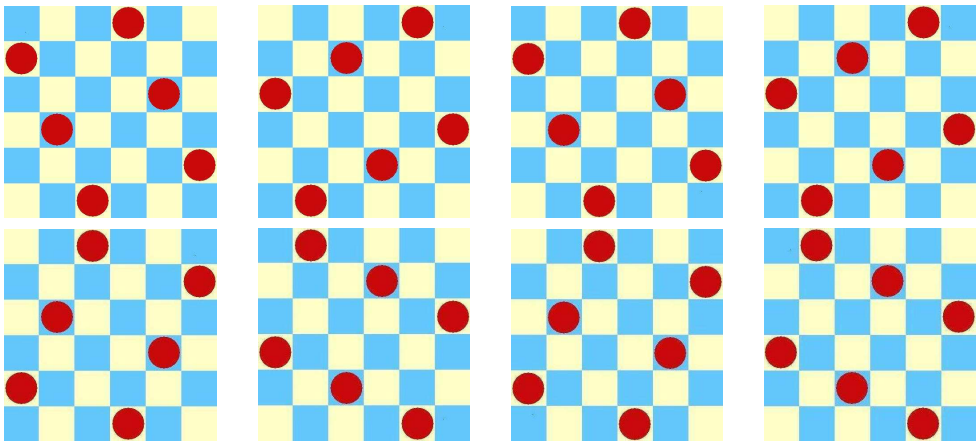


Figure 3.2: The Symmetry Group of the Chessboard: In the first line: Id, 90, 180 and 270 degree turn. In the second line the above line is mirrored on the x-axis

main diagonals are equal. The value m for this sum is called *magic number* and necessarily satisfies $m = \frac{n^3+n}{2}$. A magic square of size 4 is presented in Figure 3.3.

4	5	14	11
7	16	9	2
10	1	8	15
13	12	3	6

Figure 3.3: A feasible magic square of the size 4 with the magic number 34

Again in this problem the symmetries are the chessboard symmetries as described above in 3.1.3.2

3.1.3.4 Golomb Ruler

An n-mark Golomb ruler is a set of n distinct nonnegative integers $(a_1, \ldots, a_n)$, called *marks*, such that the positive differences $|a_i - a_j|$, computed over all possible pairs of different integers $i, j = 1, \ldots, n$, with $i \neq j$ are distinct.

Let a_n be the largest integer in an n-mark golomb ruler g. The golomb ruler $g = (a_1, \ldots, a_n)$ is optimal if there exists no other n-mark ruler $g' = (a'_1, \ldots, a'_n)$ with $a'_n < a_n$. In such a case, $|a_n|$ is called the *length* of the optimal n-mark ruler g.

Example 3.6 *An optimal golomb ruler with* 5 *marks*

The golomb ruler $g = (0, 1, 4, 9, 11)$ *is optimal with the length* 11 *and the differences* $(1, 3, 5, 2)$ *on the neighboured marks.*

The symmetry in golomb ruler is reversing the ruler, i.e. reversing the sequence of differences $|a_i - a_{i+1}|, 1 \leq i < n$.

Example 3.7 *For ruler* $g = (0, 1, 4, 9, 11)$ *of the example above is optimal with the differences* $(1, 3, 5, 2)$ *on the neighboured marks. Reversing these differences to* $(2, 5, 3, 1)$ *leads to the following ruler* $g' = (0, 2, 7, 10, 11)$

3.1.3.5 Knapsack Problem

Given is a container with a capacity restriction c and n different items each with a weight w_i. The task is to select a subset of the items to put into the container such that the sum of these selected weights w_{total} is less or equal c and the difference$c - w_{total}$ is minimal. We denote a solution by a boolean vector in the length of the items to indicate whether a an item is packed (1) or not (0).

Example 3.8 *Knapsack Instance containing* 8 *items with optimal solution*

Item weights: $(1, 1, 3, 5, 3, 5, 1, 4)$

Container capacity: 12

Optimal solution: $s_1 = (0, 1, 1, 1, 1, 0, 0, 0)$

Symmetry arise between items with the same weights. If one item t_i with weight w_i is part of the solution it can be exchanged with any other object t_j with weight w_j if $w_i = w_j$. For a solution just the weights are summed up. It does not matter to which concrete item they belong.

Example 3.9 *In the optimal solution above we added the weights* $(1, 3, 3, 5)$. *We only have two items with a weight of* 3 *and both are in the solution. But there are three items with the weight* 1 *and two items with the weight* 5. *Any combination choosing one from each weight class* 3 *and* 5 *is a symmetric solution such that there are six solutions:*

$$s_1 = (0, 1, 1, 1, 1, 0, 0, 0)$$
$$s_2 = (1, 0, 1, 1, 1, 0, 0, 0)$$
$$s_3 = (0, 0, 1, 1, 1, 0, 1, 0)$$
$$s_4 = (0, 1, 1, 0, 1, 1, 0, 0)$$
$$s_5 = (1, 0, 1, 0, 1, 1, 0, 0)$$
$$s_6 = (0, 0, 1, 0, 1, 1, 1, 0)$$

3.1.3.6 Traveling Salesperson Problem (TSP)

Consider n cities and between each pair of cities the distance $d(i, j)$ between these cities. Sought is a tour connecting all cities to a cycle with minimal length d_{total}.

Example 3.10 *TSP instance*

Consider five cities with the following matrix of distances $D = d_{ij}, 1 \leq i, j \leq 5$ *where* d_{ij} *denotes the distance from city* i *to city* j.

$$D = \begin{pmatrix} \infty & 1 & 5 & 3 & 2 \\ 1 & \infty & 1 & 4 & 3 \\ 5 & 1 & \infty & 1 & 4 \\ 3 & 4 & 1 & \infty & 1 \\ 2 & 3 & 4 & 1 & \infty \end{pmatrix}$$

A optimal solution cycle $s_1 = 1 \rightarrow 2 \rightarrow \ldots \rightarrow n \rightarrow 1$ *with* $d_{total} = 6$.

The TSP comprises two different symmetries. Since the tour is a cycle every point can be chosen as a starting point leading to n different tours.

Example 3.11

$1 \rightarrow 2 \rightarrow \ldots \rightarrow n \rightarrow 1$

$2 \rightarrow 3 \rightarrow \ldots \rightarrow 1 \rightarrow 2$

...

$n \rightarrow 1 \rightarrow \ldots \rightarrow (n-1) \rightarrow n$

Every of these tours can be reversed which means to travel the cycle in the other direction.

So there are $2n$ *symmetric cycles for each unique solution, all with the same distance.*

3.1.3.7 Social Golfers Problem

The social golfers problem (problem 10 in CSPLib [44]) is a well-studied problem in the CP community. In the social golfers problem n golfers want to play in k groups of $\frac{n}{k}$ players each week. But any two golfers may only play once together in the same group. The question is how many weeks they can do so. This problem is very interesting for the community because it has a super-exponential number of symmetries. A lot of research has been done on the social golfers problem (for example [40, 41, 18, 54]).

Example 3.12 32 *golfers in* 8 *groups of* 4 *players.*

With this configuration the golfers can play for 9 *weeks [44].*

For 32 golfers, 8 groups and a group size of 4 there are $32!10!8!^{10}4!^{80}$ symmetries! These are:

- permutation of the 32 golfers (32! symmetries)
- permutation of the 10 weeks (10! symmetries)
- in each week the groups can be separately permuted ($8!^{10}$ symmetries)
- within each group the four players can be separately permuted ($4!^{80}$ symmetries)

That means more than 10^{198} symmetries for each unique solution. But in Section 3.3.2 a way to present the model with far less symmetries is stated.

3.1.3.8 The Rehearsal Problem [82]

A concert consists of n pieces of music of different durations each involving a different combination of the m ensemble members. Players can arrive at rehearsals immediately before the first piece in which they are involved and depart immediately after the last piece in which they are involved. The problem is to devise an order in which the pieces can be rehearsed so as to minimise the total time that players are waiting to play, i.e. the total time when players are present but not currently playing.

In the rehearsal problem the rehearsal order can be reversed which does not change the total waiting time. The symmetry is just to reverse the rehearsal order.

3.2 Group Theoretical Background

Symmetries have a very valuable feature: symmetries form a mathematical group. Since groups are well-studied in mathematics, we are in the fortunate situation that all the results in group theory also hold for symmetries. Groups are well-structured and exactly this structure is the key element of breaking symmetries without losing information. Symmetric solutions form an equivalence class as we will see and the benefit is that with only one representative of each equivalence class all other solutions can be retrieved. This is the key feature needed to apply symmetry breaking. We can restrict the search to find *only one* solution of each equivalence class and we are not restricted to *which particular* solution since all are equivalent so we do not lose any information.

So by applying symmetry breaking we reduce the search space drastically without losing information. Also every solution excluded from the search can be found afterwards by computing the equivalence class of each solution if desired.

In this section we will see the concept of group theory that can be applied fruitfully in constraint programming.

We will first introduce permutation as an example for symmetry in Section 3.2.1. Permutations are wide-spread in CSPs containing symmetries. Then we will introduce groups in Section 3.2.2 and give examples motivated by permutations. Finally we investigate operations of a group on a set in Section 3.2.3 and learn how this can be used for symmetry breaking.

3.2.1 Permutations – The Symmetric Group

The symmetric group describes the permutation of elements of a sequence.

Definition 3.13 (Permutation)

A **permutation** *of a sequence of n elements is a different ordering of the $\{1, \ldots, n\}$ elements of the sequence. The set of elements in the sequence is called the* **permutation set**. *The set of all permutations over n elements is called the* **symmetric group** *and is denoted by S_n.*

Lemma 3.14 *S_n has $n!$ elements (permutations).*

The simplest notation of a permutation is to write the sequence in a row and write in a second row the position where the corresponding element is permuted to. This notation is called the *Cauchy form*. Another form more frequently used in group theory is the *cycle form*, where a permutation partitions the permutation set in disjunctive subsets which is explained later in this section more detailed.

Permutations are ubiquitous in every-days life and also in CSPs.

Example 3.15 *Consider six persons queuing up for the bus at the bus station. The first person in the queue has the first position, the second person the second position and so on. If for example the first and second person change their positions the first person is on the second position and vice versa. Therefore, the initial bus queue is the initial ordering.*

The Cauchy form consists of two rows of numbers. In the first row the initial order is stated and in the second row it is stated to which position in the order this element is permuted.

Consider the permutation of the bus queue from Example 3.15 in the way that the first person changes position with the third and the second with the fifth. The corresponding Cauchy form for that permutation would be:

$$p_1 = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 3 & 5 & 1 & 4 & 2 & 6 \end{pmatrix}$$

Formally permutations are bijective functions that map a set to itself. For p_1 the function is $p_1(1,2,3,4,5,6) = (3,5,1,4,2,6)$. If only a result for a subset of the order is desired the argument of p_1 is just that subset. The permutation p_1 for the elements $(2,4,6)$ would be $p_1(2,4,6) = (5,4,6)$. If it is clear what the permutation set is the permutation can just be denoted by the second row: $p_1 = (3,5,1,4,2,6)$.

Consider a different permutation where the first person changes with the sixth and the fourth with the fifth. The corresponding permutation is:

$$p_2 = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 6 & 2 & 3 & 5 & 4 & 1 \end{pmatrix}$$

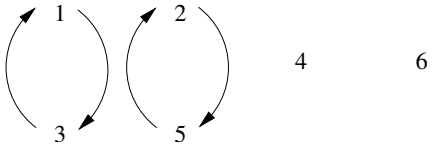
It is also possible to combine permutations $p_3 = p_1 \circ p_2$. This is done by applying the permutations successively from left to right. Note that this is contrary to the way functions are evaluated in calculus. In the example above the combination of $p_1 \circ p_2$ would result in the following permutation:

$$p_3 = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 3 & 4 & 6 & 5 & 2 & 1 \end{pmatrix}$$

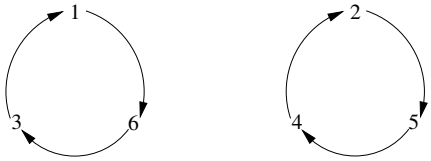
For the persons that would mean for example that the first person changes to the third seat ($p_1(1) = 3$) and and stays there ($p_2(3) = 3$). The second person changes to the fifth seat ($p_1(2) = 5$) and from there to the fourth seat ($p_2(5) = 4$).

p_3 is actually a combination of two other permutations: $p_3 = p_1 \circ p_2$. That means that p_3 is generated by the other two permutations.

In the cyclic form a permutation is denoted in disjunctive subsets of the permutation set. Each subset contain the elements that maps to each other For example the cyclic form of p_1 is: $p_1 = (1,3),(2,5),(4),(6)$



while the cyclic form of p_3 is: $p_3 = (1, 6, 3), (2, 5, 4)$



In p_1 1 is mapped to 3 and 3 is mapped to 1. These elements form a cycle. 4 is mapped to itself and forms a *one-cycle*. One-cycles can be omitted when writing the cycle form of a permutation since it is implicitly clear that the element is mapped to itself. In p_3 1 is mapped to 6, 6 is mapped to 3 and 3 is mapped to 1.

Since permutations can be combined it is also possible to combine a permutation with itself which means to perform the same permutation repeatedly. If a permutation is applied often enough the result is the identity. The number how often a permutation has to be applied to form the identity depends on the length of the cycles and is the lowest common multiple of the cycles of the permutation. For example p_1 has to be applied twice to form the identity and p_3 three times.

Example 3.16 *We will explicitly show this for p_3 with its cycle form $p_3 = (1, 6, 3)(2, 5, 4)$ but regard only the first element of each cycle of p_3.*

$$\begin{aligned} p_3(1,2) &= (6,5) \\ p_3(6,5) &= (3,4) \\ p_3(3,4) &= (1,2) \\ {p_3}^3 &= p_3 \end{aligned}$$

Definition 3.17 (Fixed Points of a permutation)

The elements that are mapped to identity by a permutation p are called the fixed points of p denoted by $fix(g)$.

Definition 3.18 (Support of a permutation)

The elements that are not mapped to identity by a permutation p are called the support of p denoted by $supp(g)$.

Examining p_1 the support of p_1 is $supp(p_1) = (1, 2, 3, 5)$ and the fixed points are $fix(p_1) = (4, 6)$.

The support and the fixed points are always disjunctive sets of a permutation and the union of both sets results in the permutation set.

In the following we will see some definitions of special forms of permutations.

Definition 3.19 (k-cycle)

A permutation p that contains only one cycle with k elements and one-cycles otherwise in the cycle form is called a k-cycle.

Definition 3.20 (Transposition)

k-cycle with $k = 2$ is called a transposition.

Definition 3.21 (Cyclic Permutation)

A permutation on n elements that forms a k-cycle is called cyclic.

The permutations p_1, p_2 and p_3 are not cyclic since they contain more than one cycle (one-cycles not counted).

Lemma 3.22 *If two permutations p_i and p_j operate on different supports they are commutative and it holds $p_i \circ p_j = p_j \circ p_i$.*

Two permutations are commutative on the same support if one is the inverse of the other such that the composition is the identity.

The inverse of a permutation can be easily determined by just swapping the two rows.

$$p_3 = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 6 & 5 & 1 & 2 & 4 & 3 \end{pmatrix}$$

$$p_3^{-1} = \begin{pmatrix} 6 & 5 & 1 & 2 & 4 & 3 \\ 1 & 2 & 3 & 4 & 5 & 6 \end{pmatrix}$$

and if ordered

$$p_3^{-1} = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 3 & 4 & 6 & 5 & 2 & 1 \end{pmatrix}$$

As we have seen permutations have many features like combining them to new permutations, build the inverse permutation and the identity does not change a permutation and the same permutation can be applied to itself several times until the result is the identity. In general the investigation of *one* permutation is rather uninteresting. More interesting is to investigate sets of permutations with special features. These special sets are the mathematical groups in group theory.

3.2.2 Groups

Operations on sets form the basic property of a group so we start giving the definition for operations and work our way from the basics to the concepts of group theory that can be applied in constraint programming. In the following we will undermine the mathematical definitions with examples and describe how this translates in a constraint programming consensus. This is done using permutations but this is no limitation. All results hold for symmetries in general. In fact every symmetry is a permutation. We start with defining an operation on a set of elements. For our purpose the set will form a group as we will see and the elements we are regarding are search states of the search tree.

Definition 3.23 (Operation on a set)

Let M be a set and $a, b, c \in M$. A **operation $\top$ on a set** *is a mapping*

$$M \times M \to M$$
$$(a, b) \to c$$
$$a \top b = c$$

An operation is called **associative** *, if*

$$(a \top b) \top c = a \top (b \top c)$$

and **commutative**, *if*

$$a \top b = b \top a$$

In the section about permutations we have already seen that two permutations can be combined to form a new permutation. So this defines an operation on permutations and the result is also a permutation.

Definition 3.24 (Group)

A group consists of a set G with an associative operation on G. Furthermore there is a neutral element with respect to the operation and for each element there exists an inverse.

- $\forall g_1, g_2 \in G : g_1 \top g_2 \in G$ *(operation)*
- $\forall g_1, g_2 \in G : (g_1 \top g_2) \top g_3 = g_1 \top (g_2 \top g_3)$ *(associative)*
- $\exists 1 \in G : \forall g \in G : g \top 1 = g$ *(neutral element)*
- $\forall g \in G : \exists g^{-1} : g \top g^{-1} = 1$ *(inverse element)*

We will investigate a puzzle game, called *Rubik's Cube* named after its inventor Rubik [91]. Then we will show that the set of permutations form a group.

Example 3.25 *Rubik's Cube*

Rubik's Cube is a cube of the dimensions $(3 \times 3 \times 3)$*. We do not regard the* 27 *subcubes but the* 54 *tiles that are visible from the subcubes. Each tile is coloured in one of six colours and there are exactly* 9 *tiles in each colour. The task is to arrange the tiles from an arbitrary position in a way such that each facet of the cube is uni-coloured.*

Arranging the tiles is done by twisting a level of subcubes by 90 *degree, which is called an elementary twist. Levels can be twisted horizontally as well as vertically. Elementary twists can be combined to form a twist sequence.*

The Rubik's Cube together with the twist operation forms a group. The group elements are all possible twist sequences that could be performed on the cube:

- *Closedness under Combination: Combining two twists leads to a twist sequence*
- *Associative: Since we can perform twists only one after the other associativity is preserved*
- *Neutral element: Performing no twist is the neutral element*

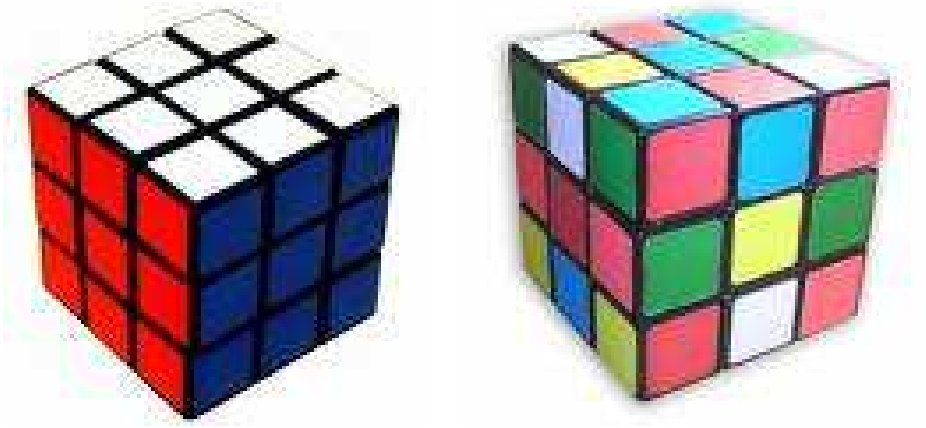


Figure 3.4: An ordered Rubik's Cube on the left and an unordered Rubik's Cube on the right

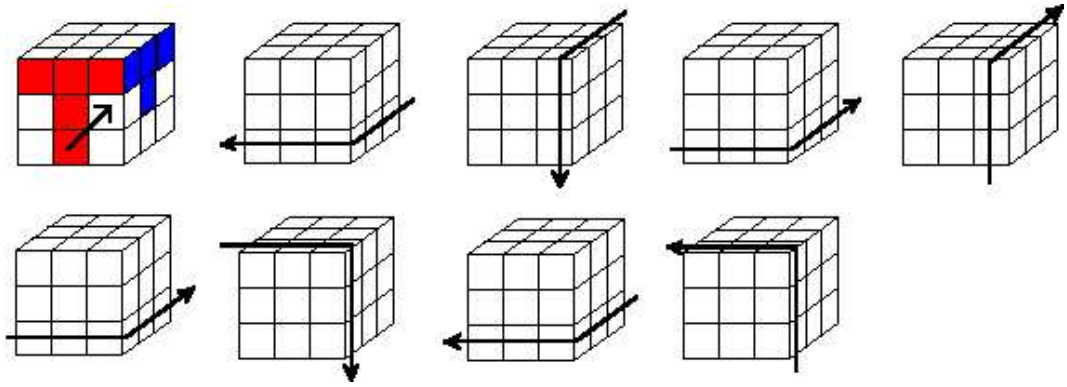


Figure 3.5: Some possible twists on the Rubic's Cube

- *Inverse element: A twist can be neutralised by performing the twist in the opposite direction*

Example 3.26 *Permutations*

The set S_n with the operation of combining permutations forms a group:

- *Closedness under Combination: Combining two permutations states an other permutation*
- *Associative: This can be seen be consequently writing down all permutations*
- *Neutral element: The identity leaves a permutation unchanged and therefore is the neutral element*
- *Inverse element: For each permutation there exists an inverse such that combining them results in the identity*

Definition 3.27 (Order of a Group)

The order of a group G is the number of elements in G and is denoted by $|G|$.

A group can be represented either by listing all elements or – more convenient – listing its generators.

Definition 3.28 (Generators of a Group)

Let G be a group and $S \subset G$ a set. If all elements of G can be represented by a combination of the elements of S by the group operation and also all combinations of elements of S are in G then S is a set of generators of the group G, denoted by $G = \langle S \rangle$.

Example 3.29 *Consider the group $G = \{(1,2,3),(2,3,1),(3,1,2)\}$. A generator of G is $S = \{(2,3,1)\}$. S could also be chosen as $S = \{(3,1,2)\}$ or $S = \{(2,3,1),(3,1,2)\}$. It holds that $G = \langle(2,3,1)\rangle = \langle(3,1,2)\rangle = \langle(2,3,1),(3,1,2)\rangle$.*

In general not every subset of a group $S \subset G$ generates G but if all generators of the group are in S it does not matter how many other elements of G are also in S.

In group theory one is not interested in every set that generates a group but in a set with minimal number of elements. In the example above the minimal number of generators is one. A group that consists just of one generator is called cyclic. In the example above we could choose two different elements as generator. For representing the group it does not matter which one is chosen. They are both equivalent since both generate the whole group.

Definition 3.30 (Minimum Cardinality of a Set of Generators)

Let G be a group. The minimum cardinality of a set of generators of G is denoted by $d(G)$.

When generating a group it is sufficient to know only a minimal cardinality set of the generators.

Definition 3.31 (Subgroup)

A subset $H \subset G$ is a subgroup of G, if H itself forms a group with the same operation defined on G.

Simple examples for subgroups are the Group G itself and $\{id\}$.

Example 3.32 *A simple Subgroup*

Consider the the group $G = \{(1,2,3),(2,3,1),(3,1,2),(1,3,2),(2,1,3),(3,2,1)\}$ generated by $S = \langle(2,3,1),(1,3,2)\rangle$. By definition any combination of the elements of S is an element of G. Consider now the group G_1 that is generated by $S_1 = \langle(2,3,1)\rangle$. Every element of G_1 is also an element of G since it is generated by a combination of elements of S_1 which is a subgroup of S. By definition G_1 is a group but also a subset of G and therefore a subgroup of G.

Given a group G and a subgroup H one can construct *cosets*. A coset is constructed by choosing any element $g \in G$ and combine it with every element $h \in H$. The resulting set is called the (right) coset gH.

Definition 3.33 (Coset)

The coset of a group is denoted by

$gH = \{a = gh | h \in H\}$

Definition 3.34 (Index of H in G)

The number of cosets of a subgroup H is called **index** *of H in G and is denoted by* $[G : H]$

Every coset aH has as many elements as H. Therefore, the following holds:

$|G| = |H| \cdot [G : H]$.

Theorem 3.35 (Lagrange) *Let G be a finite group and H a subgroup of G. The order of H is a factor of the order of G.*

Example 3.36 *n-Queens Problem*

The n-queens problem has eight symmetries: Rotate the chessboard stepwise by 90 degree and flipping the board (and rotate again). Flipping the chessboard is a subgroup H of the group G of embedding of the chessboard. There are two possible positions such that the $|H| = 2$. The index of H in G is therefore $[G : H] = 4$. And the the order of H multiplied by the index of H in G is the order of G.

This helps us in classifying the solution space. If only partial symmetry breaking is performed (i.e. not all symmetries are broken) this may help identifying which subgroup of symmetries should be broken.

Two cosets that are generated by different elements $g_1, g_2 \in G, g_1 \neq g_2$ are either disjoint or the same: $g_1H \cap g_2H \neq \emptyset \Rightarrow g_1H = g_2H$. It also holds that all cosets of H have size $|H|$.

A set of generators for G can also be found by taking one element of each coset of H.

Since all cosets of a subgroup H are disjoint or the same the set of cosets forms a *partition* of G.

Definition 3.37 (Partition)

Let M be a set. A **Partition** *P of M is a fragmentation of M in non-overlapping subsets $T_i, i = 1, \ldots, n$:*

$$T_i \cap T_j = \emptyset, i, j \in \{1, \ldots, n\}, i \neq j \tag{3.1}$$

$$T_1 \cup \ldots \cup T_n = M \tag{3.2}$$

A partition can also be induced by circumstances other then cosets. In general a partition is induced by an *equivalence relation*. This relation states for two elements whether they are identic (with respect to the relation) or not.

Definition 3.38 (Equivalence Relation)

Let M be a set and $a, b, c \in M$. A relation on a set M is an **equivalence relation** *if the relation is transitive, symmetric and reflexive*

- $a \sim b$ *und* $b \sim c \Rightarrow a \sim c$ *(transitive)*
- $a \sim b \Leftrightarrow b \sim a$ *(symmetric)*
- $a \sim a$ *(reflexive)*

Consider for example $\mathbb{Z}$ and the relation ≥ 0. This relation partitions $\mathbb{Z}$ in two subsets. One where all elements are greater or equal zero and one with all negative numbers.

An equivalence relation partitions a set (or a group in our case) in several subsets. In each subset all elements have the same characteristics relative to the equivalence relation. The elements in a subset are equivalent. They form an *equivalence class*.

Definition 3.39 (Equivalence Class K_a)

The equivalence class K_a of an element $a \in M$ consists of elements $b \in M$ with $b \sim a$:

$$K_a = \{b \in M | a \sim b\}$$

Example 3.40 *Consider the solution space M to be an assignment of n variables, such that elements $a, b \in M$ can be $a = (1, 4, 2)$, $b = (1, 2, 4)$. The corresponding equivalence relation is the set of all permutations of n elements S_n. Since b is a permutation of a they are both in the same equivalence class. Therefore, it holds that $a \sim b$ and $b \in K_a$.*

A different example is a placement in the n-queens problem. Any placement is equivalent to the placements that result by rotating the chessboard by 90 degree (and also 180 and 270 degree).

The special feature of an equivalence class is that once an element of the class is found (no matter which one) all others can be explicitely generated by applying the equivalence relation to that element. In the example above once an assignment is

found to the n variables the whole equivalence class can be generated by applying all $n!$ permutations.

This is the reason why symmetries can be broken. If a solution is found by the search all symmetric solutions can be generated using the symmetry function. Since these solutions are equivalent they are also equivalent in terms of feasibility. Therefore, it is sufficient to find only *one* solution per equivalence class by the search process and exclude all other symmetric solutions from the search. This way the solution space reduces drastically. This does not only apply to full solutions but also to partial solutions. If a partial solution is infeasible all symmetric equivalents will be as well (and the same holds for feasibility). Since we have exactly one representant per equivalence class we do not lose solutions by reducing the search space by symmetry breaking.

We can not only decide whether two solutions are equivalent respective to an equivalence relation. Since we do know the corresponding function of the equivalence relation we can generate all equivalent solutions. In the example above we can compute the whole equivalence class K_a by applying all permutations to a. Therefore, we can safely exclude symmetric solutions from search.

A different very important feature of groups is that equivalence classes build a partition of the group. This way each element occurs exactly once in a partition. We have seen that no solution is lost, if symmetry breaking is applied the way that exactly one representant of each equivalence class is found by the search. Therefore, symmetry breaking is a complete method. In terms of efficiency the partition of the solution space ensures that no element is found twice since each element is part of exactly one equivalence class. Therefore, the solution space is reduced by symmetry breaking to its most restricted part (which means that only unique solutions are found).

3.2.3 Group Homomorphisms as Equivalence Relations

We will investigate how a group acts on a set. The group in our case will be the group of the symmetries in a problem and the set are the (partial) variable assignments. By applying a symmetry to an assignment, it is mapped to a different assignment. In mathematical terms this is just a group acting on a set. This insight lead the way to orbits and stabilizers which are used to characterise the actions of a group on a set. In dynamic symmetry breaking especially stabilizers can be used to indicate which symmetries are left unbroken after the assignment of a value to a variable [77].

Definition 3.41 (Group Homomorphism)

Given two groups G_1 and G_2. The operation on G_1 is $\top$ and the operation on G_2 is $\oplus$.

A function $f : G_1 \mapsto G_2$ is a **group homomorphism** *if for all elements $a, b \in G_1$ the following holds:*

$$f(a \top b) = f(a) \oplus f(b)$$

A group homomorphism induces an equivalence relation.

Example 3.42 *The map $\varphi : \mathbb{C}^\times \to \mathbb{R}^\times$ with $\varphi(a) = |a|$ and $\mathbb{C}^\times = \mathbb{C} \setminus \{0\}$ is a group homomorphism and the induced equivalence relation on $\mathbb{C}$ is: $a \sim b$ if $|a| = |b|$.*

Definition 3.43 (G-set)

Given a group G and a set M. An operation of G on M is a mapping $G \times M \to M$ which maps each pair (g, x), $g \in G$ and $x \in M$ to an element $gx \in M$. Written $(g, x) \mapsto gx$.

The mapping obeys the following axioms:

- $\forall x \in X : 1x = x$, *1 is the neutral element of G*
- $\forall g, g' \in G : (gg')x = g(g'x)$, $x \in X$ *(associative)*

A set M with an operation of G is called a $G-$set.

Example 3.44 *Consider a solution $a = (2, 1, 3, 3, 5, 3)$ and the two permutations (transpositions)*

$$p_1 = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 1 & 2 & 3 & 6 & 5 & 4 \end{pmatrix} \text{ and } p_2 = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 1 & 2 & 4 & 3 & 5 & 6 \end{pmatrix}$$

Both produce the same result $p_1(a) = p_2(a) = (2, 1, 3, 3, 5, 3)$. Even more: $p_1(a) = p_2(a) = a$.

In the above example we have seen that different permutations can result in the same solution even more they leave the initial solution unchanged. The set of solutions that leave a solution a unchanged is called the *stabilizer* of a.

Definition 3.45 (Stabilizer)

*The **Stabilizer** of an element $x \in M$ is the set of elements of G, which leave x unchanged. $G_x = \{g \in G | gx = x\}$.*

The stabilizer G_x is a subgroup of G. In terms of symmetry breaking if a stabilizer for an element x is known it is sufficient to apply only one element of G_x to x. All other elements would show the same result. Therefore, by identifying a stabilizer the number of symmetries to apply can be reduced. We will see in section 3.3 that this has a crucial effect on the time and space efficiency of symmetry breaking since a lesser number of symmetry breaking constraints have to be stated.

An other interesting subset of a group is called *orbit*.

Definition 3.46 (Orbit)

The orbit $O_x(G)$ under G of an element $x \in M$ is the image of x under G:

$O_x(G) = \{gx | g \in G\}$.

The orbit of an element $x \in M$ is the subset of elements of M that are equivalent to x with respect to all group elements. In terms of symmetry breaking the orbit of an element (an assignment) is the equivalence class of elements that are symmetric to the original assignment.

As mentioned before the symmetries of a problem form a group. In symmetry breaking normally for each symmetry a constraint has to be stated. Orbits are

Figure 3.6: On the left an open Rubik's Cube and on the right the Backbone of a Rubik's Cube

Figure 3.7: The tiles on the left are the corner subcubes with three different colours on it while the tiles on the right are the edge subcubes with two different colours

interesting in the way that they tell to which elements an assignment is mapped and the corresponding constraints can be stated. In order not to state the *same* constraints more often the stabilizer can be used. Since the elements of the stabilizer map an assignment to the same element it is not needed to state all these constraints. All these constraints forbid the same symmetric equivalent. Therefore, it is sufficient to state just one constraint. This helps keeping the constraint store small which is more efficient.

Example 3.47 *Rubik's Cube Revisited*

As seen before in the Example 3.25 the possible positions of the Rubik's cube form a group. There are $\frac{8!\cdot 3^8\cdot 12!\cdot 2^{12}}{3\cdot 2\cdot 2}$ *about* $4,3\times 10^{19}$ *different positions of the cube [91].*

In fact, there are even more positions if you take in account that the cube can be shattered and rearranged by hand (See Figures 3.6 and 3.7). In this case there are about $5,1\times 10^{20}$ *positions. But it is not possible to reach them all by a combination of twists. It depends on the way the cube is arranged after shattering.*

Therefore, these $5,1\times 10^{20}$ *positions are partitioned into the twelve possible orbits. Taking the definition of orbits that means that* G *is the group of all possible twists and* M *is the set of all possible arrangements of the cube. The* G*-set then corresponds*

to all the positions $5,1 \times 10^{20}$ *positions which partitions into the twelve orbits with* $4,3 \times 10^{19}$ *elements each.*

In fact the twist operation on the cube is a permutation. The colours of the tiles are permuted by each twist.

Example 3.48 *Permutations:*

In a permutation problem where G is the group of all permutations the orbit of an element is the set of all permutations of this element. Consider a solution $x_1 = (2,1,3)$ *and the permutation group G.*

$O_{x_1}(G) = \{(1,2,3),(1,3,2),(2,1,3),(2,3,1),(3,1,2),(3,2,1)\}$. *Consider a different solution* $x_2 = (2,1,1)$. *Then the orbit contains fewer elements since more group elements result in the same permutation:* $O_{x_2} = \{(1,1,2),(1,2,1),(2,1,1)\}$.

n-Queens problem:

In the n-queens problem all eight possible embeddings by flipping and rotation of the board are in the orbit. See Figure 3.2.

Also here it is possible that some permutations result in the solution. For the following solution of the 4*-queens problem all permutations leave the solution unchanged.*

If the stabilizer for each subgroup is just the identity the orbit for each element x has the same number of elements as the group G: $\forall x \in X : |O_x| = |G|$. In general it holds $|G| = |O_x| + |G_x| - 1$ since the identity counts for the stabilizer and the orbit.

3.3 Symmetry Handling/Breaking

Speeding up the search process is a very desirable goal for a constraint programmer.

Symmetries in a problem formulation slow down the search since equivalent parts of the solution space have to be investigated over and over although the structure and the solutions of these parts are already implicitly known. Therefore, breaking symmetries efficiently – and thereby reducing the search space – is a strategic goal in constraint programming and has draw attention from constraint programmers form all over the world (see for example [7, 13, 62, 22, 18, 31, 40, 54, 74]). They all agree that efficient symmetry breaking is very vital for the success of constraint programming.

The underlying structure of symmetry – the group – enables us to exclude symmetric equivalents without losing information. Symmetry breaking is the process of communicating this structure to the CP solver. There are several techniques of breaking symmetries which will be investigated in this section. In general symmetry breaking is done by remodelling of the problem (Section 3.3.2), static symmetry breaking (Section 3.3.3) or breaking constraints dynamically (Section3.3.4).

Symmetries cause serious problems in CSPs since symmetric solutions do not provide new informations. If the symmetry function is known all symmetric solutions of an equivalence class can be computed if just one solution is known. Speaking in terms of group theory it is sufficient to find just one element of each orbit.

This computation can be done apart from the tree search process much faster. Therefore, it is too time-consuming to find symmetric solutions by searching the search

tree. In addition not only solutions are symmetric but also no-goods. That means that also symmetric failures in the search would be rediscovered over and over again which leads to a waste of time.

Symmetric solutions are excluded from the search tree. Which means each node investigated by the search is distinguishable from all others. Solutions are not lost since they can be derived after the search is done by applying the symmetry function exhaustively. Therefore, excluding symmetries (which is also known as *symmetry breaking*) – which means excluding sub-trees of the search tree – preserves all solutions of the problem such that it is a sound operation.

The gain in reducing the search space by symmetry breaking can be huge. If the symmetry group just has two elements the search space is reduced by the factor 2 when applying symmetry breaking. Many problems involve permutations such that the symmetry group has $n!$ elements. In this case the search space is reduced to $\frac{1}{n!}$ of its original size by applying symmetry breaking. This is an enormous reduction. Still there is a flaw that is introduced with symmetry breaking. Although this search space is reduced the process of symmetry breaking may be costly such that the gain is not that enormous and depending on the symmetry breaking method it is not possible to apply it to very large problems or with its full reduction power. Therefore, also experience is needed when choosing the best symmetry breaking method for a problem. It must be said that symmetry breaking must be performed as carefully as modelling the problem. Especially if constraints are stated to break the symmetry. If the constraints are not stated fully correctly then solutions (which means in this case whole equivalence classes) could be excluded from search. This loss is irrecoverable such that the solver may return a wrong solution (i.e. a non-optimal solution in optimisation or the answer that the problem has no solution although there are some).

In Section 3.3.1 we will investigate symmetry detection since symmetries are often not obvious in a problem such that detection is needed. Section 3.3.2 is concerned with symmetry breaking by simple modelling which means mainly to reformulate the model. Section 3.3.3 and 3.3.4 are concerned with static and dynamic symmetry breaking respectively. In static symmetry breaking constraints are stated *before* the search starts and in dynamic symmetry breaking constraints are stated *during* the search. In Section 3.3.5 we investigate symmetry breaking by a global constraint and state in Section 3.3.6 briefly auxiliary methods of symmetry breaking to give an idea what other methods could be used. We also take a look on incomplete symmetry breaking methods in Section 3.3.7 and briefly state some attempts to combine symmetry breaking methods in Section 3.3.8. The last Section 3.3.9 is concerned with some applications where symmetry breaking was applied successfully.

3.3.1 Symmetry Detection

Before a symmetry can be excluded it must be identified in the problem.

There are two ways of identifying symmetries. Either by hand or automatic. While the first way was the most common the later get more and more interesting and investigated in research.

When given a problem description or a CSP directly it is not obvious, whether or not the problem has symmetries and which they are. First the problem description

has to be investigated manually. That means symmetry detection is often subject to experience and ingenuity of the modeller. Some kinds of symmetries are more or less obvious others are more catchy to find.

When identifying symmetries by the user there is a method independently studied by McDonald [30] and Harvey [43]. The idea is to make it easier for the modeller to write down all the symmetries. This is achieved by creating a system which produces the symmetry group needed for symmetry breaking without the user having knowledge in group theory [77]. One of the main features of this system is also to allow stating symmetries in a (for the modeller) simple form. Unfortunately, this technique is up to date limited to the most common forms of symmetry and do not allow expressing arbitrary symmetry groups [77]. Although this is no approach for general symmetries it is helpful in problems with common symmetries. Still the approach is based on incorporating the system that produces the symmetry group. This is not included in standard solvers and have to be included or written in the case it does not exist for the desired solver.

There are methods that can be used for automatic symmetry detection [26, 63]. In general the structure of the problem is investigated and checked whether it yields symmetries. A technique to find symmetries is to investigate the constraint graph of a problem and look for graph automorphisms, i.e. a remapping of the variables and edges such that the graph is unchanged after the transformation. Determining the automorphism group the constraint graph delivers the symmetries of a problem.

One way of detecting symmetries is done by an automatic modelling component like Conjure [26]. The problem is stated in a high level language like Essence [25] and a model is returned in which the symmetry is broken. In fact Essence and Conjure do even more. They look for the most efficient model for the problem. This way modeller with less experience can state a problem and be sure that the problem is efficiently represented to the solver. This technique is still evaluated such that it is not fully usable at the moment. But when it can be used this could help spreading constraint programming to more users.

3.3.2 Symmetry Breaking by Modelling/Reformulation

Some CSPs contain symmetries by the nature of the specific problem. In the n-queens problem the chessboard is symmetric. But different models may introduce new symmetries or reduce the number of symmetries. In general the modeller is interested in a model with as few symmetries as possible.

A very good example for the impact of remodelling is the social golfer problem stated in Section 3.1.3.7. It is possible by introducing variables to reduce the symmetries from $32!10!8!^{10}4!^{80}$ to $32!10!$ [79]. This is still a huge number but far less than in the original model. The reduction is achieved by introducing a variable for each pair of players that indicate in which weak they play together. This way only the permutation of the players and the weeks remains in the model.

There are several other problems where symmetry breaking was achieved by reformulation. One idea is to introduce classes for objects that are identic [15]. This corresponds to building equivalence classes. In the model instead of assigning an object, an *object class* is assigned. Constraints ensure that each class is assigned as often as it contains elements.

3.3.3 Static Symmetry Breaking

Once a symmetry is detected it is possible to break it by stating symmetry breaking constraints before search. These constraints exclude symmetric equivalents such that only one representative of each class of solutions is feasible. This is called static symmetry breaking in contrast to dynamic symmetry breaking (see Section 3.3.4) where constraints are posted during search.

This form of symmetry breaking is the most historically among all techniques. This is due to the fact that stating constraints is a key feature of a constraint solver and therefore possible in every solver. Other symmetry breaking techniques involve third-party products like GAP [38] or self-written code that is not provided in the standard solver. During time more and more concepts like these of global constraints are incorporated in solvers but most techniques require the production of code to benefit from the ideas.

Puget [71] proved that every CSP comprising a symmetry can be represented in a way such that the symmetry is eliminated.

Smith [81] points out that symmetry breaking constraints often allow to state implied constraints that could not be derived otherwise. This side-effect was observed in the template design problem [70] by Proll and Smith. This offers new potential to state problems more efficiently.

3.3.3.1 Ordering Constraints

One kind of symmetry breaking constraints are ordering constraints. These constraints induce an order on the assignments such that only one permutation out of each solution class is feasible. Namely the lexicographically least by default. There is a technique called *lex-leader* that constructs symmetry breaking ordering constraints for variable constraints [13]. Roughly speaking a set of variables is interpreted as a string and a variable assignment to the variables can be considered a word. And only the lexicographically least word is feasible. This approach can only be applied for a static variable ordering since the order of the variables induces an ordering of the full solutions.

It is also possible to state simple lexicographical ordering constraints that can also be used with a dynamic variable ordering. These constraints are called *Crawford Constraints* [13].

There is a major drawback in using symmetry breaking constraints. Often these constraints conflict heavily with the search heuristic. The problem is that the symmetry breaking constraints state clearly *which* equivalent of each solution class is feasible (the lexicographically least for example. But it is not guaranteed that the search heuristic will find this solution first. It is also possible that this solution is found very late in the search with dramatic impact on the search time [33] Search wastes a lot of time in equivalent branches which are marked infeasible by the symmetry breaking constraints before finding the one feasible branch.

Consider for example symmetry breaking constraints that order a set of variables $x_1, \ldots, x_n$ lexicographically increasing. The value assignment heuristic assigns the smallest value in the domain of the variables but the variable assignment heuristic considers the variables due to a different criteria which orders x_n to be assigned first.

Depending on the domains of the variables it may take a lot of time until search finds out that x_n is assigned infeasible.

Dynamic symmetry breaking methods are designed to overcome this weakness of the static approach.

3.3.4 Dynamic Symmetry Breaking

In dynamic symmetry breaking symmetries are broken during search. That means that there are no constraints at the beginning of the search that state the symmetry breaking. The key idea is to perform the search without restricting the used heuristic. But – depending on the strategy used – at specific events during search symmetry breaking is performed. In SBDS the event is a backtracking move and symmetry breaking constraints are added to the constraint store. In SBDD the event is the instantiation of a variable and the action is to check whether a symmetric equivalent has been investigated before.

3.3.4.1 SBDS (Symmetry Breaking During Search)

In SBDS [31] on backtracking from an infeasible partial assignment constraints are posted that prohibit every symmetric equivalent to this no-good. SBDS is based on a method proposed by Backofen and Will [7].

Consider that there is a branching decision at a choice point $var = val$ as opposed to $var \neq val$ at a partial assignment A. Consider further that there is a particular solution symmetry g that maps assignments (full or partial) to other assignments. We assume that the symmetry g can be expressed by a constraint and acts piecewise. This means that an for an extension of A to $A+(var = val)$ it holds that $g(A+(var = val) = g(A) + g(var = val)$.

The constraint that is posted is :

$$A \ \& \ g(A) \ \& \ var \neq val \Rightarrow g(var \neq val) \tag{3.3}$$

That means if it holds that A is a feasible assignment and also $g(A)$ and the variable assignment $var = val$ infeasible is, then the symmetric version of this assignment $g(var = val)$ is also infeasible and the constraint $g(var \neq val)$ is posted that prohibits that the symmetric version of var is assigned with the value val.

Example 3.49 *Consider the* 8*-queens problem and the* 180 *degree rotation of the board as the symmetry function* g*. The variables of the problem are* Q_i *where* i *represents the* i*-th row and the value assigned represents the column the queen is placed.*

Consider the partial assignment $A : (Q_1 = 2, Q_2 = 4)$*. Performing search it becomes clear that this partial assignment cannot be extended to a solution and search backtracks in* Q_2*. That means that the constraint* $Q_2 \neq 4$ *is posted. Since we now that* A *does not lead to a solution we also know that* $g(A) = (Q_8 = 7, Q_7 = 5)$ *cannot be extended to a feasible solution.*

Therefore, the constraint that is posted is:

$$Q_1 = 2 \;\&\; Q_2 \neq 4 \;\&\; Q_8 = 7 \Rightarrow Q_7 \neq 5 \tag{3.4}$$

This constraint ensures that when the following assignments are done: $Q_1 = 2$, $Q_2 \neq 4$ *and* $Q_8 = 7$ *then the assignment* $Q_7 = 5$ *is forbidden.*

In short SBDS ensures that on backtracking the symmetric equivalents to the infeasible partial assignment are excluded from search by posting a constraint. In the example of the 8-queens problem seven symmetry functions like the one above have to be stated in order to break all symmetries. And at each no-good seven new constraints are posted (whereby it is possible that these seven constraints are not pairwise different which means. This happens if two or more symmetry functions g_1 and g_2 map a partial assignment A to the same assignment: $g_1(A) = g_2(A)$.).

It has to be mentioned that on backtracking all constraints posted in the corresponding subtree due to backtracking are removed from the constraint store in order not to slow down the system. These constraints can be removed without losing symmetry breaking power.

Example 3.50 *Consider the Example 3.49 and a partial assignment* $B = (Q_1 = 2, Q_2 = 4, Q_3 = 6)$. *Since* A *cannot be extended to a feasible solution a partial assignment containing* A *also cannot be extended to a feasible solution. Therefore, at some point the search backtracks with the decision* $Q_3 \neq 6$ *and the appropriate constraints are posted by the SBDS functions. When search now backtracks in* A *we do not need the constraints posted at* B *by the SBDS functions any more since they are implicitly stated in the constraints posted at* A. *Therefore, the constraint store does not contain obsolete constraints.*

Still it is possible that identic constraints are posted by the SBDS functions as mentioned above. But an extension of SBDS, GAP-SBDS [34] manages this. It should be said that GAP-SBDS is only available in the ECL^iPS^e Solver such that the use is limited to this solver.

For each symmetry a function has to be written that handles posting the constraint that prohibits exactly this symmetric assignment. As one can see the problem with SBDS is, that in problems with many symmetries many functions have to be implemented in order to break all symmetries. For larger permutations it is not possible to state all symmetry functions due to the enormous number of symmetries. GAP-SBDS handles this by describing the symmetry group rather can each individual symmetry [77]. In the SBDS equation :

$$A \;\&\; g(A) \;\&\; var \neq val \Rightarrow g(var \neq val) \tag{3.5}$$

only $g(A)$, the computation of the orbit of the assignment A, is handled efficiently by GAP. The rest of the components are handled by the solver. With GAP-SBDS the SBDS implementation is that effective that it can handle groups with billions of elements instead of thousands of elements for the implementations of SBDS before [77].

Still there remains a problem which limits the use of SBDS. On problems with large symmetry groups like permutations the enormous number of constraints that are to be stated at each no-good cause space problems. This may lead to a slow down of the search since too many constraints have to be checked or a breakdown in the worst. The search visits less nodes but the time spend in each node is much higher such that this outweighs the gain of the reduced number of search states to visit.

3.3.4.2 SBDD (Symmetry Breaking via Dominance Detection)

In SBDD [18] or dominance detection [24] each partial variable assignment is investigated and it is decided whether a symmetric equivalent has been visited before. If so the search backtracks since the equivalent sub-tree has been investigated before.

The idea is to check at each choice point in the search, whether it is equivalent to or dominated by a node that has been visited earlier.

SDBS and SBDD are equivalent up to implementation from a certain viewpoint which Harvey showed in [42]. The difference between the both approaches is mainly that in SBDS constraints are stated to *prevent* running into equivalent states while SBDD checks backward whether the actual search state is equivalent to anyone visited before.

Note that is does not count for an investigated search state A that is dominated by a previously investigated search state B, whether B can be extended to a feasible solution or not. If B cannot be extended to a feasible solution, neither A can, because they are symmetric equivalent. If B can be extended to feasible solutions, so can also A. But these solutions are equivalent to the ones found as extensions of B. Therefore, no unique solution is lost using the technique.

Fahle, Schamberger and Sellmann state in [18] that using SBDD Symmetries can be used to prune values from future variables. But whether this pays off depends on the application.

The approach needs a database T to store information of the search space already explored. Also a problem specific function $f : (A, B) \rightarrow \{true, false\}$ is needed that states, if two search states are equivalent. At each search state A it is checked whether it is dominated by a search space $B \in T$.

A problem of the approach are the dominance checks. If T consists of many search states then it is very costly to check equivalence. An method to overcome these problems is to perform dominance checks not at every choice point but only for a subset. This reduces the number of checks performed but may also detect dominated search states later which causes a loss of efficiency.

The main drawback to SBDD is that the entire investigated tree has to be stored, which is of exponential size. This has to be done in order to check at a node whether it is symmetric to a previously investigated node. But this weakness can be overcome using the following idea. Instead of storing *all* nodes investigated only nodes with fully investigated subtrees are stored. In the dominance checks now it is not checked whether a node is symmetric to a previously investigated one but if it is an extension of a node with fully investigated subtree. This is possible since a node with fully investigated subtree either cannot be extended to a solution (such that the actual node also cannot be extended to a solution) or is leads to solutions in which case the solutions of the actual node are symmetric to the ones found before and can be pruned.

3.3.5 Symmetry Breaking by Global Constraints

Techniques like SBDS and SBDD require to model the symmetries problem specific anew for different problems. In SBDS the symmetry functions have to be stated and in SBDD the function that states dominance of two search states have to be written. Also these approaches have difficulties in dealing with super-exponential number of symmetries. In SBDS a huge number of constraints would have to be posted which slows the system down let alone implementing a function for each symmetry function. And in SBDD also the dominance checks whether two search states are equivalent would be very costly.

A different approach is to use global constraints for breaking symmetries. The gain is that the global constraint does not have to be adjusted and is therefore problem independent. Still as any global constraint the usability is limited to the scope of the global constraint.

Permutations for example are a class of symmetries that arise often in CSPs. Focusing on a class of symmetries like permutations has the advantage that specialised algorithms for propagation can be found that achieve generalised arc consistency for the given set of constraints. Work mostly has concentrated on symmetry breaking in matrix models [54, 22, 20, 21] (see Chapter 3.1.3.1 about matrix models.).

Kiziltan introduced a specialised algorithm for lexicographically ordering rows and columns in matrix models [54]. The algorithm prunes values from future variables that represent a symmetric search state of the one investigated thereby breaking symmetries. In the case of only row or column symmetries all symmetries can be broken. If row and column symmetries occur then lexicographic ordering does not break all combinations of the row and column symmetries.

3.3.6 Auxiliary Techniques

There are some other techniques to break symmetries, which are here just briefly investigated to provide an overview of the existing symmetry breaking methods.

3.3.6.1 Conditional Symmetry Breaking

Conditional symmetry arise during search. They are not present from the beginning. They occur in search when the remaining subproblem is a symmetric. Conditional symmetries have to be identified upfront mostly. Symmetry breaking can be done by stating conditional constraints that are only part of the constraint store if some conditions hold. The work of Gent *et al*[29] state that it is also possible to handle conditional symmetries by reformulating the problem or use a variant of SBDD.

3.3.6.2 GE-Trees

A GE-tree is defined as a tree which contains only one solution of each equivalence class under the symmetry group of the problem [75]. That means that search does not explore the full tree but the GE-tree in which are no symmetric equivalents by definition. The abbreviation stands for *Group Equivalence*.

A GE-tree is defined as a tree where no two nodes are symmetrically equivalent and for each solution class of the problem one node is in the tree. The latter

condition is needed in order not to lose solutions to the problem. GE-trees are are also intended to be viewed as a conceptual paradigm capable of classifying and comparing symmetry breaking techniques.

The idea is that any technique that constructs a GE-tree does automatically break all symmetries. Therefore, GE-trees are useful to refine existing techniques and develop new ones.

3.3.6.3 Symmetry Breaking Using Stabilizers (STAB)

In this technique stabilizers are used to reduce the number of symmetric equivalents. Like SBDS STAB adds symmetry breaking constraints during search. But unlike SBDS only constraints are posted that leave the actual partial assignment A unchanged. The posted constraint are expressed by lexicographic ordering constraints. The stabilizer of an element is defined as

$$G_A = \{g \in G | g(A) = A\}$$

The stabilizer G_A is a subgroup of G and therefore $|G_A|$ is a divisor of $|G|$ (see Definition 3.34). STAB posts an lexicographic ordering constraint for each element of the stabilizer:

$$A \leq_{Lex} g(A), \forall g \in G_A$$

The number of constraints posted this way is much smaller than the number of symmetries in the symmetry group. Therefore, symmetries remain in the problem and STAB is an incomplete symmetry breaking method.

3.3.6.4 Symmetry Excluding Heuristics

Meseguer and Torras [64] proposed an approach in which symmetries are used to guide the search. The idea is to lead the search in parts of the search space with a high degree of non-symmetric solutions. They investigated several heuristics and partially combined them with no-good recording. The latter approach was rather disappointing while the former showed good results. Excluding symmetries by variable and value heuristics is not much investigated and exploited by the constraint community up to date [77] and there might still be a lot of potential.

3.3.7 Incomplete and Partial Symmetry Breaking

Some of the techniques introduced before are incomplete symmetry breaking methods like the STAB method introduced in Section 3.3.6.3. But it is always possible to perform partial symmetry breaking instead of complete symmetry breaking. Often the problem is that symmetry breaking does not come for free. Static symmetry breaking constraints may inflict with the search, SBDS posts an enormous number of constraints, SBDD has to maintain a large database and perform checks. For problems with a large number of symmetries it may not be possible to break all the symmetries. Therefore, the idea is to break just some of them. In partial symmetry breaking the question arise *which* symmetries should be broken and which not.

There is no general answer to that. In some problems partial symmetry breaking may outperform complete symmetry breaking [62]. Still it is not predictable in which cases partial symmetry breaking does outperform complete symmetry breaking.

3.3.8 Combining Symmetry Breaking Methods

The symmetry breaking methods introduced have different features. It would be desirable to combine several techniques such that the advantages of all these approaches could be used. Unfortunately it is very hard to combine different symmetry breaking methods correctly. The reason is that each symmetry breaking methods excludes all but one element of each equivalence class. When symmetry breaking methods are combined they have to respect the *same* element in each equivalence class. If this is not the case solutions will be lost.

There are some approaches where existing symmetry breaking methods were combined and modified such that they could be applied together. The results were rather disappointing. One approach was to combine SBDS and SBDD by Harvey [42] and in an other approach GAP-SBDD and GE-trees were combined [52]. In the latter case the approach was less efficient then GAP-SBDD alone.

Still there are two successful approaches. One by Puget who combined SBDD with the incomplete method STAB which outperformed SBDD alone. The second approach was done by Petrie [69] and combined GAP-SBDS and GAP-SBDD. In this approach these methods were combined in a way such that one method was applied at the top of the search tree and the other on the bottom of the search tree. Results sometimes outperformed the single approaches but in no case the approach was less efficient than the least efficient of the two single approaches.

3.3.9 Successful Applications of Symmetry Breaking

In the handbook of constraint programming [77] Gent *et al* point to a few successful application areas of constraint programming. Although the list is far from being complete. Still most of the announced applications are either academic problems, combinatorial puzzles or problems that occur mostly in combination with other problems in real life. Therefore, the success of symmetry breaking in real life problems is either not documented or not present.

The problem is that most real life problems are very complex and consist of several interlinked subproblems. Each of these subproblems cannot be solved individually without side effects on the rest of the problem. That means an optimal solution for a subproblem may only lead to a suboptimal solution for the whole problem. The key flaw is that these side effects of one subproblem to the others often destroys the symmetry of a subproblem.

The next chapters will introduce in the key contribution of this thesis: The definition of weak symmetries and a modelling approach that enables us to break weak symmetries.

Chapter 4

Weak Symmetry

As stated in the last chapter, real world problems are particularly complex. Often they consist of several subproblems. These subproblems are interlinked such that the symmetry of a subproblem is nullified by these links. Also we stated in the last chapter that the search space of a problem with symmetries is much larger than the search space of the same problem with the symmetries broken. Therefore, we should be glad when there is no symmetry in a problem, such that the search space consists of unique solutions. If an isolated subproblem comprises a symmetry that the global problem does not have, all solutions are unique, although the symmetry in the subproblem is not broken. Symmetry breaking would reduce the search space on these subproblems which would lead to an overall reduction of the search space of the global problem and again would make symmetry breaking an outstanding method for solving real world problems.

However, applying symmetry breaking to subproblems where the global problem is not symmetric leads to a loss of solutions. If just a subproblem contains a symmetry the solutions of a specific equivalent class of the subproblem induces *different* solutions to the global problem. Therefore, it is not possible to exclude symmetric solutions of a subproblem. *All* solutions have to be considered for the further solving process. This is not possible using standard symmetry breaking such that a great potential is not applicable.

In this chapter we investigate closely the special symmetry – that we call *weak symmetry* which acts only on a subproblem of a global problem. Also we give a definition that is very general in the sense that it is open to extensions since weak symmetries are newly discovered and may act in a larger context than considered here.

Identifying weak symmetries together with the modelling approach to handle weak symmetries described in Section 4.3 gives us a new dimension in symmetry breaking. First of all we can break symmetries in problems where it was not possible to do so before. Second, we can use any standard symmetry breaking method described in Section 3.3. And last all future symmetry breaking methods can be used, since they all rely on the same fact: excluding all but one element of each equivalence class.

We start by giving the formal definition of weak symmetry in Section 4.1. In Section 4.2 we show that weak symmetries are widespread and do even occur in already investigated problems without being discovered. Section 4.3 is concerned with the modelling technique that enables weak symmetry breaking. In Section 4.4 we present

related work in the field of weak symmetries and conclude in Section 4.5 with the benefits and limitations of the approach.

4.1 Weak Symmetry Definition

In contrast to a proper symmetry, a weak symmetry cannot be simply broken. This is due to the modus operandi of symmetry breaking techniques. Symmetry breaking techniques exclude all but one representative of each equivalence class of solutions from the search tree. Since all elements in an equivalence class have the same features it is safe to exclude them from the search. No solution (with different features) will be missed this way.

A weak symmetry in contrast to a standard symmetry induces also classes of variable assignments but the elements in each class are only symmetric with respect to a subset of the variables and/or the constraints of the CSP. This implies that it is not sufficient to consider only one representative of each class and exclude the rest from the search as is done in standard symmetry breaking. In fact all solutions of a class have to be considered in order not to lose solutions. Therefore, breaking a weak symmetry with standard techniques leads to a loss of solutions.

We will formally define the structure of a weak symmetry in the following. Also, we will use the magic square problem introduced in Section 3.1.3.3 as a running example in the following.

4.1.1 Running Example: Magic Square Problem

As stated on Page 62, the numbers $1, \ldots, n^2$ have to be assigned to a $n \times n$ square such that the sum of the numbers in each row, in each column and in both main diagonals are equal. The value m for this sum necessarily satisfies $m = \frac{n^3+n}{2}$. A standard model without symmetry breaking constraints is presented in the following:

Variables X:

$\text{square}_{ij}, i, j \in N, N = \{1, \ldots n\}$ (the magic square)

Domains D:

$\forall i, j \in N : \text{square}_{ij} \in \{1, \ldots, n^2\}$ (all possible numbers)

Constraints C:

$\forall i \in N : \sum_{j \in N} \text{square}_{ij}$	$=$	m	(each row sums up to m)
$\forall j \in N : \sum_{i \in N} \text{square}_{ij}$	$=$	m	(each column sums up to m)
$\sum_{i \in N} \text{square}_{ii}$	$=$	m	(diagonal sums up to m)
$\sum_{i \in N} \text{square}_{i,n+1-i}$	$=$	m	(anti diagonal sums up to m)
alldifferent(square)			(all numbers are assigned)

4.1.2 Weak Symmetry Definition

Weak symmetries act on problems with special properties. To characterize weak symmetries we first define weakly decomposable problems. In a weak decomposition of a problem all variables and constraints that are respected by the weak symmetry are gathered in one subproblem.

Definition 4.1 (Weakly Decomposable Problem)

*A problem $P = (X, D, C)$ is **weakly decomposable** if it decomposes into two subproblems $P_1 = (X_1, D_1, C_1)$ and $P_2 = (X_2, D_2, C_2)$ with the following properties:*

$$X_1 \cap X_2 \neq \emptyset \tag{4.1}$$
$$X_1 \cup X_2 = X \tag{4.2}$$
$$C_1 \cup C_2 = C \tag{4.3}$$
$$C_1 \cap C_2 = \emptyset \tag{4.4}$$
$$C_2 \neq \emptyset \tag{4.5}$$
$$D_1 = pr_1(D) \tag{4.6}$$
$$D_2 = pr_2(D) \tag{4.7}$$

where pr_i denotes the projection to the subspace defined by the subset X_i of the variables in P.

The first condition states that P_1 and P_2 contain a subset of shared variables (namely $X_1 \cap X_2$). These variables have to assume the same values in both subproblems to deliver a feasible solution to P. Therefore, they link both problems. Without that restriction the problem would be properly decomposable. The second and third condition states that none of the variables and constraints of the original problem P are lost. Furthermore the third and fourth condition state that C_1 and C_2 is a partition of C. Basically this is not necessary for feasibility. A constraint could be in both subsets (if defined on $X_1 \cap X_2$ only) but would be redundant for one of the problems because the solution to the other subproblem would already satisfy this constraint. Therefore, this is just a question of efficiency. The fifth condition states that P_2 is not allowed to be unconstrained. However, note that this restriction does not hold for P_1. This is since we want to group the symmetric data in P_1 and a problem without constraints is perfectly symmetric. Every CSP is weakly decomposable, and usually there will be multiple weak decompositions. However, we concentrate on weak decompositions where the weak symmetry acts as a proper symmetry on P_1. In fact, we are searching for a weak decomposition that introduces new symmetries in P_1.

The weak decomposition implies a solving order. First P_1 is solved and then P_2 is solved. Solving P_2 can be regarded as extending the solution found in P_1 such that it also respects the constraints in P_2. A solution to P_2 then represents a solution to P.

From a certain point of view a weak decomposition is a relaxation of constraints and variables which are put in a different subproblem (P_2) while the remaining variables and constraints form a new subproblem (P_1). This is to some extend the terminology and viewpoint of Harvey [45] who calls this *symmetric relaxation*.

We will see three weak decompositions of the magic square problem in the following and shortly discuss them.

Example 4.2 *Weak Decomposition of the Magic Square Problem (Relaxing Diagonal Constraints*

Subproblem P_1 (assigning the numbers with respect to the row and column constraints):

Variables X_1 :

$\text{square}_{ij}, i, j \in N, N = \{1, \ldots n\}$ *(the magic square)*

Domains D_1 :

$\forall i, j \in N : \text{square}_{ij} \in \{1, \ldots, n^2\}$ *(all possible numbers)*

Constraints C_1*:*

$\forall i \in N : \sum_{j \in N} \text{square}_{ij} = m$ *(each row sums up to m)*
$\forall j \in N : \sum_{i \in N} \text{square}_{ij} = m$ *(each column sums up to m)*
$\text{alldifferent}(\text{square})$ *(all numbers are assigned)*

Subproblem P_2 *(check whether the solution of* P_1 *also respects the diagonal constraints):*

Variables X_2 :

$\text{square}_{ii}, i \in N$ *(variables of the diagonal)*
$\text{square}_{i,n+1-i}, i \in N$ *(variables of the anti diagonal)*

Domains D_2 :

$\forall i \in N : \text{square}_{ii} \in N$ *(all possible numbers)*
$\forall i \in N : \text{square}_{i,n+1-i} \in N$ *(all possible numbers)*

Constraints C_2*:*

$\sum_{i \in N} \text{square}_{ii} = m$ *(diagonal sums up to m)*
$\sum_{i \in N} \text{square}_{i,n+1-i} = m$ *(anti diagonal sums up to m)*

Note that when the whole problem P has to be solved the variables $X_1 \cap X_2$ *(in this case the variables forming the diagonal and the anti-diagonal) have to be assigned the same values.*

This weak decomposition introduces new symmetries to P_1. The row and column constraints in the model allow permutations of the rows and columns.

Example 4.3 *Weak Decomposition of the Magic Square Problem (Relaxing Column Constraints)*

Subproblem P_1 *(assigning the numbers with respect to the row and diagonal constraints):*

Variables X_1 :

$\text{square}_{ij}, i, j \in N, N = \{1, \ldots n\}$ *(the magic square)*

Domains D_1 :

$\forall i, j \in N : \text{square}_{ij} \in \{1, \ldots, n^2\}$ *(all possible numbers)*

Constraints C_1*:*

$\forall i \in N : \sum_{j \in N} \text{square}_{ij} = m$ *(each row sums up to m)*
$\sum_{i \in N} \text{square}_{ii} = m$ *(diagonal sums up to m)*
$\sum_{i \in N} \text{square}_{i,n+1-i} = m$ *(anti diagonal sums up to m)*
$\text{alldifferent}(\text{square})$ *(all numbers are assigned)*

Subproblem P_2 *(check whether the solution of* P_1 *also respects the column constraints):*

Variables X_2 :

$\text{square}_{ij}, i, j \in N, N = \{1, \ldots n\}$ *(the magic square)*

Domains D_2 :

$\forall i, j \in N : \text{square}_{ij} \in \{1, \ldots, n^2\}$ *(all possible numbers)*

Constraints C_2*:*

$\forall j \in N : \sum_{i \in N} \text{square}_{ij} = m$ *(each column sums up to m)*

Note that when the whole problem P has to be solved the variables $X_1 \cap X_2$ *(in this case all variables) have to be assigned the same values.*

This weak decomposition does not introduce new symmetries to P_1. Although the relaxation of the column constraints in P_1 would allows column permutation this is prohibited by the diagonal constraints. Therefore, this weakly decomposition seems not futile for our purposes.

This demonstrates that not every weak decomposition introduces symmetries. The constraints and variables that have to be relaxed in P_1 have to be chosen carefully. The next example shows that introducing symmetry can also be disadvantageous.

Example 4.4 *Weakly Decomposition of the Magic Square Problem (Relaxing the Diagonal and the Alldifferent Constraint)*

Subproblem P_1 *(assigning the numbers with respect to the row and column constraints):*

Variables X_1 :

$\text{square}_{ij}, i, j \in N, N = \{1, \ldots n\}$ *(the magic square)*

Domains D_1 :

$\forall i, j \in N : \text{square}_{ij} \in \{1, \ldots, n^2\}$ *(all possible numbers)*

Constraints C_1*:*

$\forall i \in N : \sum_{j \in N} \text{square}_{ij} = m$ *(each row sums up to m)*
$\forall j \in N : \sum_{i \in N} \text{square}_{ij} = m$ *(each column sums up to m)*

Subproblem P_2 *(check whether variables of* P_1 *are all different and the diagonal constraints are respected):*

Variables X_2 :

$\text{square}_{ij}, i, j \in N, N = \{1, \ldots n\}$ *(the magic square)*

Domains D_2 :

$\forall i, j \in N : \text{square}_{ij} \in \{1, \ldots, n^2\}$ *(all possible numbers)*

Constraints C_2*:*

$\sum_{i \in N} \text{square}_{ii} = m$ *(diagonal sums up to m)*
$\sum_{i \in N} \text{square}_{i,n+1-i} = m$ *(anti diagonal sums up to m)*
$\text{alldifferent}(\text{square})$ *(all numbers are assigned)*

This weakly decomposition introduces also row and column symmetries like the weakly decomposition in Example 4.2. But this weakly decomposition allow variable assignments that are feasible for P_1 which cannot form a solution to P since not all numbers are used and others are used several times.

The task in identifying weak symmetries is to find a weakly decomposition that introduces new symmetries in P_1. Also as much variables and constraints should be put in P_1 such that P_2 only contains the asymmetric variables and constraints.

Definition 4.5 (Weak Symmetry)

Consider a weakly decomposable problem P with a decomposition (P_1, P_2).

A symmetry $f : X_1 \to X_1$ on P_1 is called a **weak symmetry on** *P with respect to the decomposition (P_1,P_2) if it cannot be extended from X_1 to a symmetry on X.*

The intention of the decomposition of the problem is that X_1 contains all symmetric variables (and only those) and X_2 contains the rest of the variables. The gain is that we get a subproblem where the weak symmetry affects all variables and all constraints (P_1) and one subproblem that is not affected by the weak symmetry (P_2). On P_1 the symmetry is actually a proper symmetry.

In the Example 4.4 the alldifferent constraint also respects the induced weak symmetries of the row and column permutations. Therefore, it is better not to relax it in P_1. This tightens P_1 such that infeasible assignments can be ruled out early in the search. We therefore look for a weak decomposition that introduces more symmetries in the first subproblem but that relaxes as few constraints and variables in P_1 as possible.

4.1.3 Regarding Local Weak Symmetries

Finding a weak decomposable representation of a problem is a theoretic view on the problem that induces a variable ordering. First the variables of P_1 have to be instantiated, then the variables of P_2. Also it reveals whether a problem has weak symmetries. Even more, the weak symmetry affects the whole subproblem P_1, which means it is a standard symmetry on P_1.

Still we cannot break the symmetry simply on P_1 since we could lose solutions on P. We need to find a way how to pass *all* solutions of an equivalence class in P_1 to P_2. Each solution is checked whether it can be extended to a full solution that satisfies P. The question is how to pass the solutions to P_2. Again, this is where group theory helps us. As we have seen in Theorem 3.46 applying the symmetry group to a solution generates the whole equivalence class of this solution. In fact, we need to apply the symmetry group (of the weak symmetry) to the solution of P_1. This generates the whole equivalence class of this solution. If we can do that we can break the symmetry in P_1 using any symmetry breaking method.

Example 4.6 *Magic Square of size* 4

When we choose the weakly decomposition represented in Example 4.2 we find a magic square that does not necessarily satisfy the diagonal constraints like the one presented in Figure 4.1.

This solution does not satisfy the diagonal constraints and therefore no solution to P. But the question is, whether an equivalent solution does satisfy P_2. Instead of

4	5	11	14
7	16	2	9
10	1	15	8
13	12	6	3

Figure 4.1: A magic square that does not satisfy the diagonal constraints

dismissing the solution we check whether a symmetric equivalent to this solution does satisfy P_2. In this case a row and/or column permutation could satisfy the diagonal constraints. Checking the column permutations of this solution this indeed yields a solution that also satisfies the diagonal constraints and therefore P_2. This can be achieved by permuting the third and fourth column as shown in Figure 4.2.

4	5	11	14
7	16	2	9
10	1	15	8
13	12	6	3

4	5	14	11
7	16	9	2
10	1	8	15
13	12	3	6

Figure 4.2: The right magic square is a column permutation of the left one

When we check the whole equivalence class for each solution of P_1 we can break exactly these symmetries in P_1 that constructs this equivalence class. In this case the row and column permutations.

In Section 4.3 we will present our approach how to break weak symmetries. In the following we will investigate some CSPs that contain weak symmetries.

4.2 CSPs Containing Weak Symmetries

As seen in the example of the magic square problem also standard problems may comprise weak symmetries considering the correct weak decomposition of it. But especially in real world problems many symmetries are weak. As stated in Section 3.3.9 many real world problems are complex and consist of several interlinked problems where a symmetry can be found on a subproblem but not on the whole problem. The weak decomposition reflects exactly the structure of these problems. The subproblems cannot be solved individually. There are also two research fields where weak symmetries arise. These are soft constraints and distributed constraint satisfaction problems (DisCSPs). These two fields are investigated in this section as well.

4.2.1 Construction of Weak Symmetries

A weak symmetry is always a standard symmetry on a subproblem of the full problem considered. From a certain point of view every symmetry is a weak symmetry (and vice versa). It just depends whether you look at a full problem or a subproblem. With this insight it is easy to see that weak symmetries can be found in two ways:

1. relaxing constraints on a problem such that the relaxed problem contains new symmetries
2. add constraints on a symmetric problem such that the tightened problem is less symmetric

4.2.2 Puzzle CSPs with Weak Symmetries

The magic square problem is a nice example to demonstrate that academic problems comprise weak symmetries in a natural way. It is also possible to alter some problems in a way that they comprise weak symmetries. This can be done by adding constraints. The unchanged problem has proper symmetries which become weak due to the new added constraints.

4.2.2.1 Diagonal Latin Square [45]

The standard problem is the latin square problem. A latin square of order n in an $n \times n$ array where each row and column is a permutation of $1, \ldots, n$ [45].

A *diagonal* latin square is a latin square where also the diagonal and anti-diagonal are a permutation of $1, \ldots, n$.

The latin square comprise row and column symmetries while the diagonal latin square does not comprise them.

It's quite obvious that in the weak decomposition the added constraints together with the variables they are stated on form P_2, while the original problem forms P_1. The weak symmetries are the row and column permutations.

Example 4.7 *A (diagonal) latin square of size* 6

1	2	3	4	5	6
2	3	1	6	4	5
5	4	6	1	3	2
4	1	2	5	6	3
6	5	4	3	2	1
3	6	5	2	1	4

Weak Symmetries:

Relaxing the diagonal constraints introduces column and row permutations.

4.2.2.2 The Rehearsal Problem

The rehearsal problem is introduced in Section 3.1.3.8.

Weak Symmetry:

A weak symmetry can be found by considering the order to be a tour. That means that after the n-th number in the sequence the first follows. The weak symmetry is at which position the order starts. If the order starts at the i-th position the last play in the rehearsal order is the $(n - i + 1)$-th position.

4.2.2.3 Asymmetric Traveling Salesperson Problem

The traveling salesperson problem we introduced in Section 3.1.3.6 is symmetric. That means that for each pair of cities (i, j) the distance $d(i, j) = d(j, i)$. In the asymmetric version of the problem this is not the case. There may be cities where $d(i, j) \neq d(j, i)$. This happens for example if there are only one-way routes from and to a city. In this case different streets have to be used. The same may happen if there is a distance in altitude between cities. Going up is more expensive than going down. In this case the weights d represent costs rather than distances.

Weak Symmetry:

In the asymmetric version of the problem a tour cannot be reversed because this introduces different distances. Therefore, reversing a tour is a weak symmetry. The constraint that makes the symmetry weak is the sum constraint over the weights. More specifically the asymmetric weights cause the *weakness* of the symmetry.

4.2.2.4 Weighted Magic Square [61, 59]

In the weighted version of the problem we search for a magic square that does not respect the diagonal constraints. Instead every field in the matrix has a weight associated. A valid magic square is evaluated by the sum of the field weights multiplied by the assigned number. Sought is a magic square with maximal weight.

Example 4.8 *Weighted Magic Square of Size* 4

Consider the following weights for the columns:

$$weights = \begin{pmatrix} 1 & 0 & 2 & 3 \\ 0 & 2 & 1 & 1 \\ 2 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 \end{pmatrix}$$

Now consider the following two solutions:

$$MS_1 = \begin{array}{|c|c|c|c|} \hline 4 & 5 & 11 & 14 \\ \hline 7 & 16 & 2 & 9 \\ \hline 10 & 1 & 15 & 8 \\ \hline 13 & 12 & 6 & 3 \\ \hline \end{array}$$

$$MS_2 = \begin{array}{|c|c|c|c|}\hline 14 & 5 & 11 & 4 \\ \hline 9 & 16 & 2 & 7 \\ \hline 8 & 1 & 15 & 10 \\ \hline 3 & 12 & 6 & 13 \\ \hline \end{array}$$

MS_2 is just a column permutation of MS_1 where the first and last columns are permuted with each other. The objective value of MS_1 is 162*, while that of MS_2 is just* 136*.*

The magic square in this problem has row and column permutations (since there are no diagonal constraints). But the optimisation function prohibits permutations since this could lead to a different weight of the magic square. Therefore, the optimisation constraint and optimisation variable form P_2 while the square variables and constraints form P_1. Also the chessboard symmetries are weak symmetries. But in combination with the row and column permutations not all eight chessboard symmetries are needed. The flip around the y-axis is for example the same as assigning the first column to the last, the second to the last but one and so on. The only remaining symmetry is the swap around the diagonal. This symmetry transforms rows into columns and vice versa. This can be modelled using a single boolean SymVar that indicates whether the square is flipped or not.

4.2.3 Real World Problems with Weak Symmetries

The examples presented here do not have to exist exactly in the presented way. They just show that some problems in real life are more complex and may involve different problems.

4.2.3.1 Open Pit Surface Mine Excavation [58]

In this problem a mine has to be exploited to get the ore that is hidden in the mine. The mine is assumed to be a cuboid. The mine is divided into 3 dimensional blocks. Due to drill and sonic examination the ore concentration in each block can be estimated. There are mining capacities such that each week only a certain amount of blocks can be mined that are considered to be mined simultaneously. The blocks can only be mined layer by layer to prevent collapses of the blocks. A block can also only be mined if at least four of its neighbouring blocks in the same layer are mined. Therefore, mining must start at the corner of a layer. Also a block must be mined if at least six neighbouring blocks are already mined. The price for ore differs through the year but is assumed to be known roughly. Ore that is excavated is assumed to be sold at the price at the next week. Sought is an excavation schedule of the mine such that the profit in selling ore is maximised. See Figure 4.3 for two examples of open surface mines.

Weak Symmetry:

The corner blocks are symmetric such that a mining order can start at each of the four corners. But the ore concentration of all blocks is different such that different amounts of ore can be mined.

The problem even comprise more weak symmetries. Not only the corner blocks are weakly symmetric. Also at each time all blocks that can be mined are weakly

Figure 4.3: Two examples of an open surface mine

symmetric. The underlying symmetry in this case is a conditional symmetry: two blocks are symmetric if they can be mined. This shows that weak symmetries can also arise during search like conditional symmetries.

4.2.3.2 Assignment of Shooting Ranges on a Military Training Ground

Military Training Grounds are highly used by troops performing shooting exercises. This is not restricted to small arms but also grenade throwing, tank cannons, artillery fire, missile fire and bombing is trained simultaneously on some training grounds. All of these weapons have different security rules and ranges of forbidden trespassing. Therefore, the simultaneous shooting of different weapon systems has to be coordinated carefully such that troops are not caught in friendly fire. Most restrictions that apply are of geometric nature like the safety zones. Since shooting ranges are booked for a certain time also the geometric restrictions impose temporal constraints.

A military training ground exists of several shooting ranges, target ranges and spotting points to observe the target ranges. Furthermore a street system consists that connects the different ranges and the training ground with the outside world.

We will give a closer description of the facilities of a military training ground in the following. See also Figure 4.4 for a visualisation:

- **shooting range:** A shooting range is an area that is prepared in a way such that different weapon systems can be deployed and shoot in a designated target range. Some shooting ranges are especially prepared for small arms and tank cannons, others are especially prepared for artillery fire like shells and rockets. Depending on which kind of fire is performed in the shooting range different safety zones have to be respected. For example the safety zone for some ammunition is very small while for other ammunition the distance from the shooting range to target ground is a safety zone. A shooting range may only be used by one troop unit at a time using just one kind of fire system.

- **target range:** A target range is a designated area on the military training ground that is only used to shoot in. Trespassing is not allowed due to safety reasons. Normally the target ground is a plane such that the impact of the fire can be seen and evaluated. The target range is surrounded by a safety

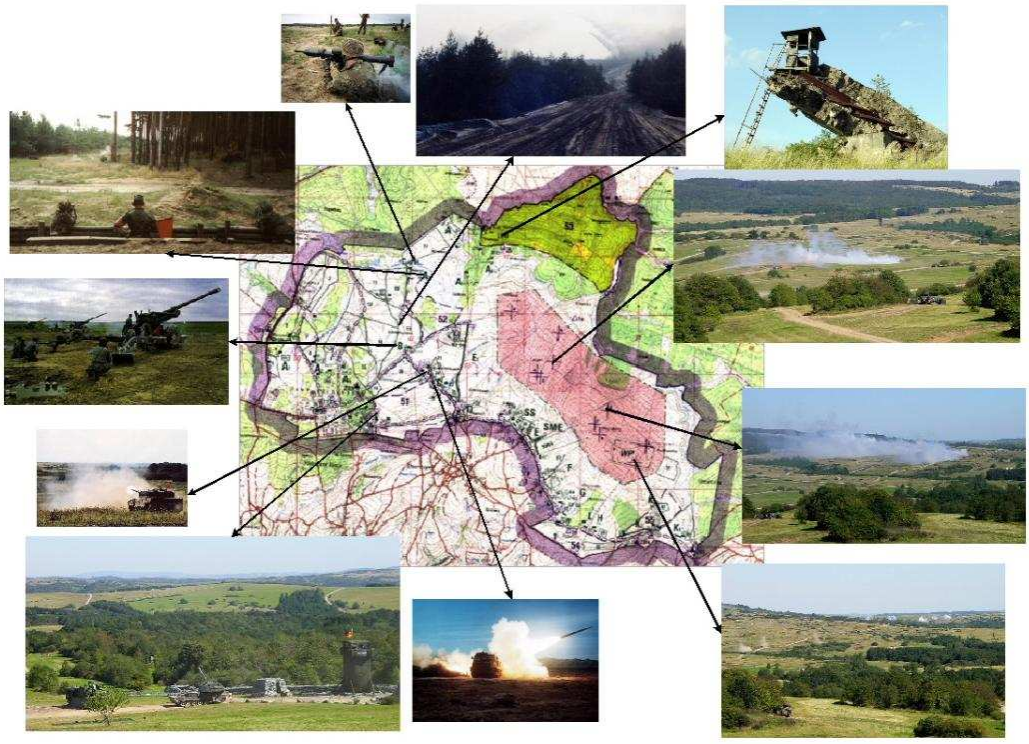


Figure 4.4: A Military Training ground utilised by several troop units.

zone that is also not to be trespassed during shooting. The target range can be used by several troop units at the same time.

- **spotting point:** From a spotting point one or more target ranges can be observed. This is crucial to safety to decide especially for indirect fire whether the target is hit or at least spotted. If a fire strike is not spotted it could be impacted anywhere and the fire training has to be stopped immediately. There are also spotting points that are very close to the target range in a bunker. But for several ammunition sorts or several weapon systems these spotting points cannot be used. A spotting point can be used by up to three troop units at the same time.

When shooting ranges are assigned to troops the safety zones of the shooting range may not overlap in a way such that a shooting range is inside the safety zone of any other shooting range in use. Also at least one route to the shooting range must be outside any safety zone. This is crucial to allow the medic to reach the shooting range in case of emergency.

Each troop unit shoots with a specified weapon system and specified ammunition (if several different ammunition sorts are used the one with the highest safety restrictions is assumed to be used at any time). Each troop unit specifies a desired amount of time that it wants to train. Also possibly the desired time interval for arrival and/or the desired time interval for leaving the training ground may be specified.

Weak Symmetries:

The shooting ranges are symmetric in terms of assigning troops to the ranges. But asymmetric in the different safety ranges imposed on the type of weapon fired from the shooting range. That means that an assignment of troops to shooting ranges may be infeasible but a permutation of the troops on the ranges is feasible since the different units shoot with different weapon systems and therefore the safety ranges change in a way such that all constraints are respected.

A different weak symmetry can be seen in the temporal constraints. A permutation of the troop units may be feasible where the initial distribution is infeasible since streets are blocked by the safety zones.

A third weak symmetry is the change of the fighting style. Different fighting styles and different ammunition may alter the safety zones such that different neighbouring units may not exercise the same fighting style at the same type.

4.2.3.3 Machine Composition

Consider a complex machine that consists of several components. The task is to determine all its components from a list of items for each component such that the performance of the machine is maximised. There are explicite constraints that state which specific components can be combined due to several restrictions like space, voltage etc. The performance of the machine is mainly determined by the components chosen. Implicitly known is that the performance does also depend on the sequence of the components.

Weak Symmetry:

P_1 would be to determine a assignment of components. P_{Sym} permutes the components and the oracle in P_2 determines the performance of the machine.

4.2.3.4 Gate Scheduling of Planes

At an airport planes are to be assigned to gates to load and unload passengers. All planes have to use the same way to the starting field. The planes are to be scheduled to the gates such that the waiting time for planes to get to the starting field is minimised. A plane is considered waiting if it is ready boarded but cannot leave the gate since it could block the way of a plane already on the way to the starting field.

Weak Symmetry:

Since the gates are in a line a plain on the last gate has the longest way to the landing field (see Figure 4.5). In P_1 an assignment of planes to gates is determined while in P_{sym} a permutation of the assignment is considered and P_2 determines the waiting times for all the planes.

4.2.3.5 PC Board Manufacturing [60, 59]

We present here a simplified version of the original problem in order to concentrate on the crucial parts for weak symmetries. For a formal problem description of the original problem see [27].

In the problem certain components must be mounted on PC boards by a mounting machine consisting of several mounting devices. The task is to maximise the

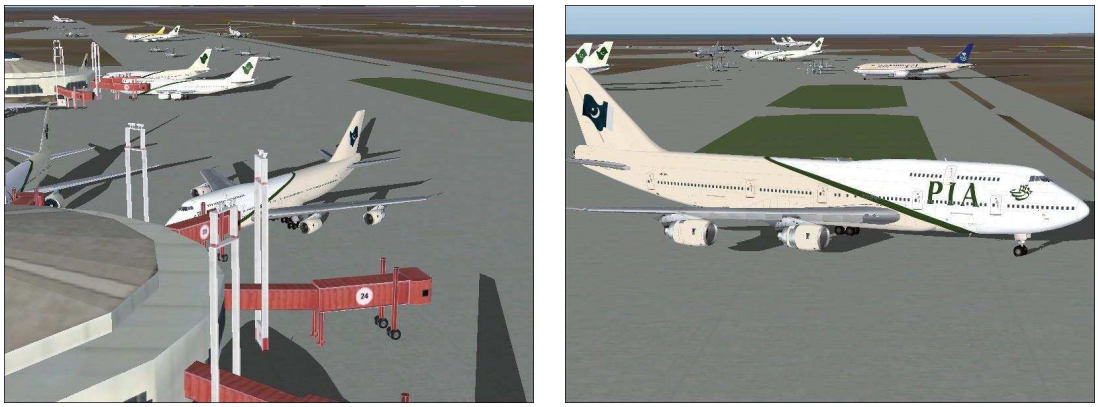


Figure 4.5: Gates assignment and way to starting field

workload of the whole machine. We concentrate only on a subproblem of the whole solving process. That is to find a setup of component types for the individual mounting devices to maximise the potential workload.[1]

The machine consists of several mounting devices. Each mounting device has access to a set of component types (called setup) that are to be mounted on the PC boards. In addition each mounting machine has only access to a part of the PC board layout and can therefore only mount components inside this visibility area (see Figure 4.6). The PC board layout is specified by a list of mounting tasks. A mounting task is specified by a component type and a position, where to mount this component type.

The problem is modelled as follows: The machine is represented by an $m \times n$ variable matrix $A^{m \times n}$ where m is the number of different component types that can be assigned to a mounting device and n is the number of mounting devices on the machine. The domain of variables $a_{ij} \in A$ is the set of component types. An assignment $a_{ij} = k$ means that a component of type k is placed on the mounting device j in the ith slot.

The constraints:

- No component type may be assigned more than once to a column
- Certain component types may not be assigned together in a column
- Each component type achieves a certain workload when assigned to a column. The workload differs from column to column. This represents the visibility of the mounting device.

The PC board consists of a list of mounting tasks specified by a component type and position where to place this component type. Each mounting device has access to a subset of the mounting tasks that have to be performed (called visibility further on). But a mounting device has to be assigned the component type of a mounting task to actually perform it (called placeability further on). This specifies what part of the board can be accessed by the mounting device (see Figure 4.6). The visibility of each mounting device is represented by an array indexed over the set

[1]The actual workload assigned to the devices is a subset of the workload determined in this subproblem. But the higher the possible workload the higher the degree of freedom for the concrete assigning problem not considered here.

of component types indicating how many mounting tasks – using this component type – are visible for this mounting device. So the placeability of each machine is simply the sum of the mounting tasks that are placeable with the assigned setup of component types. Some component types may not be present simultaneously in a setup. This is expressed by constraints stating which component types are feasible with each other in a setup. The task now is to find a feasible setup (an assignment of component types to the mounting devices) such that the overall placeability (the sum of the placeability over all mounting devices) is maximised.

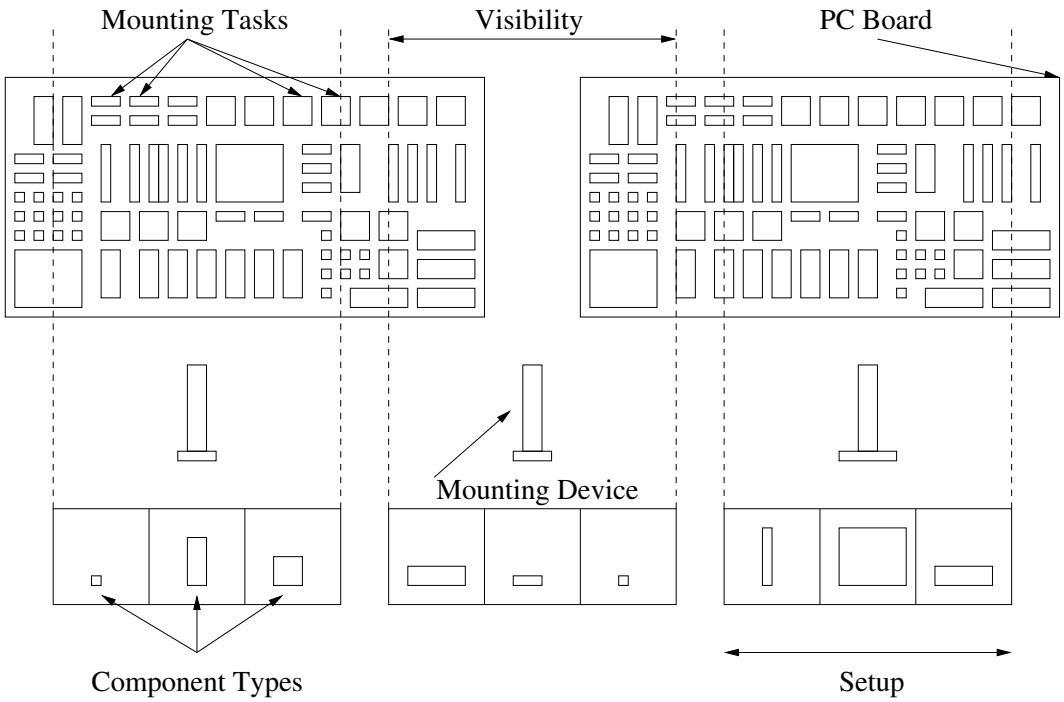


Figure 4.6: A machine consisting of three mounting devices each with a setup of three component types.

Weak Symmetry:

The weak symmetry of the problem is that the mounting devices are symmetric in terms of assigning a setup. So a feasible setup is feasible independent from the mounting device it is assigned to. But each mounting device has a different visibility of the board. So certain mounting tasks cannot be seen (and therefore not performed) by certain mounting devices. That means a setup achieves different workloads depending to which mounting device it is assigned. Therefore, the permutation of the setups on the machine is the weak symmetry.

This is modelled by introducing a SymVar place_i for each setup i. An assignment $\text{place}_i = j$ means that the ith setup is assigned to the jth mounting device on the machine. In the model that means that the column i is permuted to the column j in the matrix A representing the machine setup. To ensure that the assignment of the SymVars is a permutation an all different constraint is stated on them. The weak symmetry is broken by stating column ordering constraints in P_1.

4.2.3.6 Partial Knowledge of Constraints - Using Oracles

Sometimes not all information of a problem is at hand. For example some constraints are not given explicitly. Instead a variable assignment can be passed to an oracle that can decide whether it is feasible, i.e. a solution or not. It may seem odd and not be used in real life. But there are several good reasons for providing not all knowledge. Often very sophisticated knowledge is a trade secret and not given outside a company or even to other divisions. In this case an oracle is provided that encapsulates the critical data and decides whether a variable assignment is feasible. A different example is personal information. Personal information is often not provided to prohibit data mining or scrutinising people. Features like the age, sex, nationality, religion, level of expertise etc. of a person may not be provided. For some objects technical data is not available to prevent reverse engineering. In these cases it also cannot be decided fully whether a full variable assignment is a solution or not.

4.2.3.7 Soft Constraints

Some CSPs are *over-constrained* which means that there exists no solution to the problem [65]. There are two ways to handle it, both working by relaxation. First relax some constraints totally and search for a solution that does not respect these constraints. Second introduce weights on each constraint that states the degree of violation of the constraint and search for a solution where the total violation is minimised (Lagrange Relaxation). The first method has the drawback, that a solution does not respect the relaxed constraints at all. Some solutions may respect the constraints to a certain extent (but not fully) but it is not possible to decide during search which solution would be better. In the second method it is possible to rank the solutions by the degree of the violation of constraints. Therefore, the best solution is sought with the least violation.

Constraints with a weight function are called *soft constraints*.

There is also a scenario where soft constraints are applied other then over-constrained CSPs. That is to rank identical solutions. Consider for example graph colouring. Each colour is assigned a value and the task is to find a colouring of the graph with the highest ranked colouring.

A soft constraint is defined by a discrete cost function and a symbol $\top$ (top). $\top$ means that this assignment is infeasible. This indicates that the constraint is "hard"(a normal constraint) for this assignment and not satisfied. A problem containing soft constraint is called a *SoftCSP*.

4.2.3.8 Weak Symmetries in Soft Constraints

Nearly all SoftCSPs have weak symmetries if the "hard"version of the CSP have a symmetry. The feature that creates the weak symmetries is the cost function that is applied to the constraint. If the function evaluates symmetric solutions with different values then these solutions are not symmetric anymore and the former symmetry is a weak symmetry in the SoftCSP.

Example 4.9 *Soft Graph Colouring*

Consider the graph colouring problem. Here a map must be coloured such that no two adjacent countries are assigned the same colour. The problem can be modelled as a graph, whereby each country is a node and there is an edge between two nodes if the two countries are neighboured. See Figure 4.7 for example.

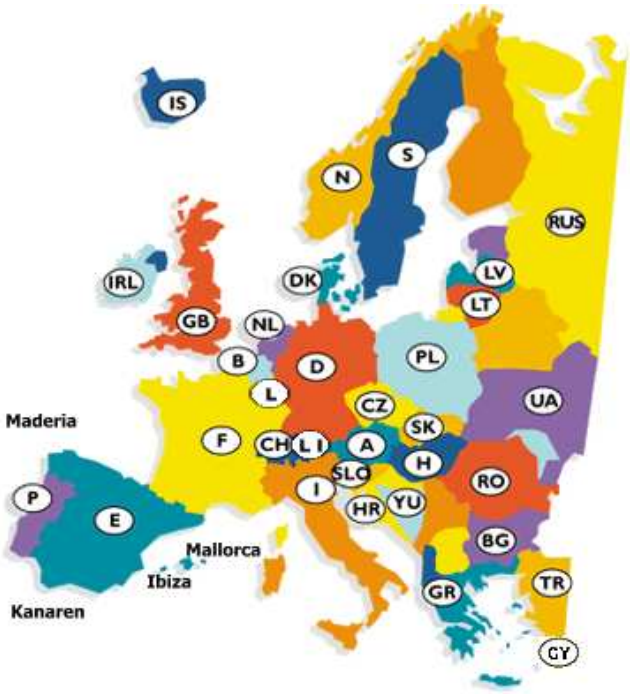


Figure 4.7: A coloured map of Europe

The following colours are available with the following weights:

- *blue :* 1
- *red :* 2
- *orange :* 3
- *green :* 4

The task is to maximise the weight of the colouring of the graph.

Weak Symmetry:

In the graph colouring problem a permutation of the colours of a solution is still a solution and therefore a symmetry. In the soft graph colouring problem permuting colours is not allowed since this does change the weight of the colouring. Therefore, the colour permutation is a weak symmetry.

The decomposition would be that P_1 consists of the graph colouring problem without weights. P_{sym} would permute the colours to the nodes and P_2 evaluates the weight of the graph colouring.

4.2.4 Weak Symmetries in Distributed CSPs (DisCSPs)

Often CSPs involve multiple participants [19]. A nice example is the problem *meeting scheduling* [66, 68]. In this problem a set of meetings for several participants have to be scheduled such that no participant has overlapping meetings. Also ordering and deadline constraints have to be respected.

The description of the weakly decomposable problem naturally fits into the description of DisCSPs.

A Distributed Constraint Satisfaction Problem consists of a set of *agents*, $A = \{a_1, a_2, ..., a_n\}$, and for each agent a_i, a set $X_i = \{x_{i1}, x_{i2}, \ldots, x_{im_i}\}$ of *variables* it controls, such that $\forall i \neq j \; X_i \cap X_j = \emptyset$. Each variable x_{ij} has a corresponding domain D_{ij}. $X = \bigcup X_i$ is the set of all variables in the problem. $C = \{c_1, c_2, \ldots, c_t\}$ is a set of *constraints*. Each c_k has a *scope* $s(c_k) \subseteq X$. The *agent scope*, $a(c_k)$, of c_k is the set of agents that c_k acts upon: $a(c_k) = \{a_i : X_i \cap s(c_k) \neq \emptyset\}$. An agent a_i is a *neighbour* of an agent a_j if $\exists c_k : a_i, a_j \in a(c_k)$. For each agent a_i, $p_i = \{x_{ij} : \forall c \; x_{ij} \in s(c) \rightarrow s(c) \subseteq X_i\}$ is its *private* variables – variables which are not directly constrained by other agents' variables – and $e_i = X_i \setminus p_i$ is its *public* variables – variables that do have direct constraints with other agents. A *global assignment*, g, is the selection of one value for each variable in the problem: $g \in \prod_{ij} D_{ij}$. A *local assignment*, l_i, to an agent a_i, is an element of $\prod_j D_{ij}$.

A solution to a DisCSP is an assignment to each variable a value from its domain such that no constraints are violated. The solution process, however, is restricted: each agent is responsible for the assignment of its own variables, and thus agents must communicate with each other in order to find a global solution.

4.2.4.1 Symmetry Handling in DisCSPs

To search for a consistent local assignment to an agent's subproblem, a centralised Constraint Programming (CP) solver can be used. This allows the local solving process to benefit from specialised CP techniques such as arc consistency and global constraints. Another CP technique, symmetry breaking, is more difficult to apply to DisCSP because:

- solutions that are symmetric for one agent's subproblem but that contain different assignments to the agent's public variables may not be equivalent with regards to the global problem; and
- no single agent has a complete view of all variables and all constraints in the global problem.

Since only an agents public variables affect the other agents in the problem, symmetry breaking can be used as long as all solutions in an equivalence set have identical assignments to the agent's public variables. However, if this is not the case it is not possible to break the symmetry without risking losing solutions. To date, the issue of symmetry breaking in DisCSP has not been addressed.

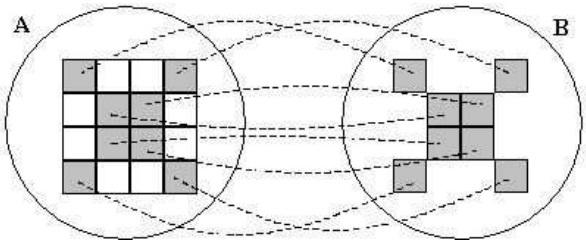


Figure 4.8: The diagonal variables of the agent A are public variables that are shared with the variables of the second agent B.

4.2.4.2 Weak Symmetries in DisCSPs

Every symmetry in DisCSP is potentially a weak symmetry. If the problem of one agent contains a symmetry it cannot be said whether the other agents do also have that symmetry. As long as this is not sure every symmetry has to be regarded to be a weak symmetry.

To demonstrate how we can break weak symmetries in DisCSP, we artificially construct a distributed magic square problem.

In our distributed scenario agent A owns all variables that make up a magic square but for its particular local subproblem it is only interested in enforcing row and column constraints. Agent B, holds variables that are constrained to be equal to the diagonal and anti-diagonal variables of agent A. Furthermore agent B will enforce the diagonal and anti-diagonal constraints on its variables. This can be seen as an instance of a Multi-agent agreement problem where agents need to agree on the values of public variables that are bound by equality constraints, considering their own private internal variables and constraints. This can be seen in Figure 4.8.

The resulting problem is similar to that of a weakly decomposable centralised problem and so we can apply weak symmetry breaking techniques. As an example, consider the column symmetries of agent A's subproblem. Any solution found that satisfies the row and column constraints can be permuted by changing the order of the columns to produce $n!$ symmetrical solutions. This symmetry can be broken by adding a constraint that orders the columns such that the first element in a column is less than the first element of the following column - thus, only one solution of each equivalence set will be found. However, in the DisCSP case, we can not afford to lose these equivalent solutions. It is possible that one valid solution found for agent A can not be extended to a valid solution in agent B, while an equivalent solution can be. See Figure 4.9 for an example.

Weak Symmetry:

Again the row and column permutations are the weak symmetries.

4.3 Breaking Weak Symmetries

Up to now we have seen that some problems comprise weak symmetries naturally and some can be extended such that the symmetry of the problem becomes weak. We proposed to weakly decompose a problem such that all the variables and constraints

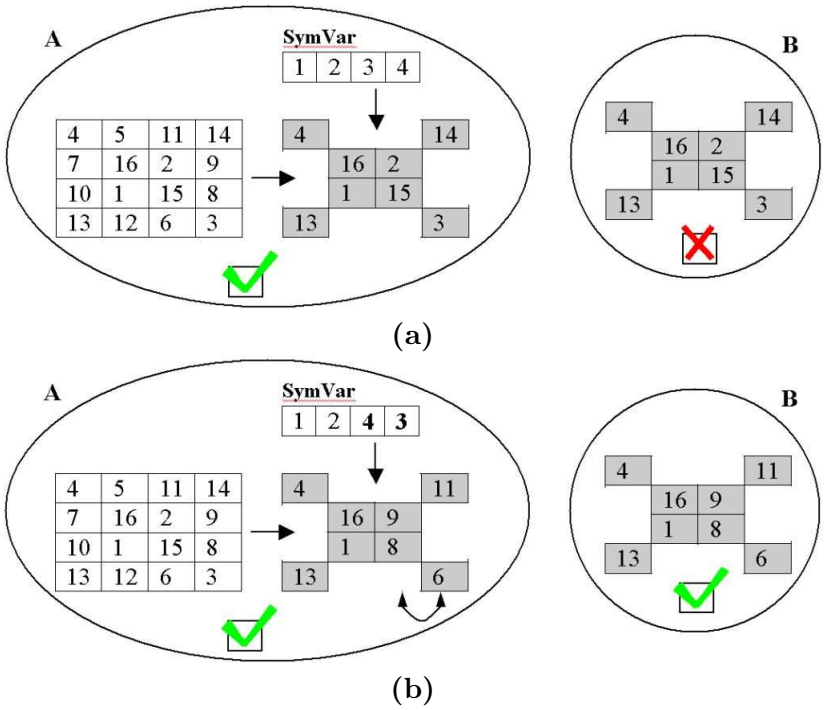


Figure 4.9: Magic square: $n = 4, m = 34$. By introducing a SymVar agent A can break the column symmetry of its original subproblem (a). Equivalent solutions can then be generated using the SymVar to find a permutation that is also valid for agent B's constraints (b).

that make the symmetry weak are arranged in a new subproblem (P_2) such that the rest of the problem (P_1) is symmetric. The solving order is implied by the weak decomposition which is simply solve P_1 first and then P_2.

Still the weak symmetry is not broken by this weak decomposition alone. But several facts are implied by the decomposition and the solving order:

- a solution that satisfies P_2 does also satisfy P_1 by definition (since a solution of P_2 is an extension of a solution to P_1), while the reverse is not the case
- a nogood to P_1 will never be checked in P_2
- the weak symmetry of P is a standard symmetry on P_1

The last point implies that in P_1 symmetry breaking could be performed. When applying symmetry breaking each solution s_1 that is found in P_1 represents a class of solutions. But since not all solutions to P_1 does also satisfy P_2, following the first point, we have to consider the whole solution class of s_1 for P_2. This can be done by applying the symmetry group on s_1 before extending s_1 to a solution of P_2. This means that each solution s_1' of the solution class is considered consecutively for P_2. If s_1' does not satisfy P_2 a different solution s_1'' is considered from the solution class.

Weak symmetry breaking therefore consists of three parts:

1. find a weak decomposition (P_1, P_2) of P

2. break the symmetry on P_1
3. apply the symmetry group to each solution of P_1 before checking P_2

Any method of symmetry breaking can be used in P_1 as long as it satisfies the necessary condition that of each solution class at least one solution is preserved by the search. Note that for solution completeness, i.e. no solutions are lost, it is not necessary that exactly one solution per solution class is found. But if more solutions of a class will be found then each time the same solution class will be checked in P_2. For efficiency it is desirable to choose a symmetry breaking method that is complete, i.e. all symmetries are broken.

As we will see in Section 4.4 there are different ways to apply the symmetry group to a solution of P_1. We introduce a modelling approach to do so that is unique and does not require additional code or knowledge other than modelling.

4.3.1 Theoretical Idea

We apply the symmetry group to a solution s_1 via a newly introduced variable π called *Symmetry Variable – SymVar*. Each value assigned to π represents a different element of the symmetry group. The combination of s_1 and π then states a specific solution of the solution class of s_1.

Definition 4.10 (Symmetry Variable)

Consider a CSP $P = (X, D, C)$ with a weak decomposition (P_1, P_2) and a weak symmetry f on P.

A **symmetry variable (SymVar)** *$\pi \in S[X_1]$ represents the group of symmetric solutions of f in P_1. Its domain is the symmetric group on X_1, denoted by $S[X_1]$.*

If the SymVar is the identity then the solution passed to P_2 is s_1. In any other case the permuted solution of P_1 (which is equivalent with respect to the weak symmetry) is passed to P_2. The solution of P_1 together with the assignment of the SymVar represents a partial variable assignment to P_2 and P. It is checked whether it also satisfies the constraints of P_2 and if so all variables in $X_2 \backslash X_1$ are assigned for finding a solution to P_2. If the partial assignment does not satisfy P_2 a different element of the equivalence class is considered by a different value for the SymVar. If none of the elements satisfy P_2, a new solution to P_1 is sought. This way the whole problem is investigated and no solution is lost. Note that only for solutions of P_1 the SymVar is instantiated. A nogood to P_1 is backtracked and will never be checked in P_2 as stated in Section 4.3. Therefore, infeasible variable assignment classes are excluded from search by symmetry breaking as usual.

Theorem 4.11 (Solution Preservation)

The solution space of P is totally reflected by the decomposition (P_1, P_2) and a SymVar π such that every solution of P can be uniquely represented by a solution to P_1, an assignment to the SymVar and a solution to P_2.

Proof. A solution of P yields a solution to P_1 and P_2 directly. π can be chosen as the identity. A solution to P_1, a SymVar assignment $\pi \in S[X_1]$ and a solution to P_2 can be transformed into a solution of P by assigning the permutation under π of

the solution to P_1. P_2 commits all variables $x_i \in \{X_2 \backslash X_1\}$ to P. Since π restores all solutions of P_1 that are excluded by the weak symmetry breaking no solution is lost and the solution space of P is totally reflected by the decomposition (P_1, P_2) and π.

4.3.2 Modelling Approach

In practice this concept of a single SymVar as a representative is not supported in constraint programming solvers on the level of modelling. In the theoretic concept the symmetry is mostly a function in several variables. In Section 3.2.1 we have seen that a permutation of n elements is stated by a function over n variables like stated here $p(1, 2, \ldots, n) = (n, n-1, \ldots, 1)$. Therefore, instead of one variable we use a set of variables, each variable representing an element of the symmetry (for example a column in column permutation). A feasible variable assignment to these variables then represents a specific element of the equivalence class. We will also see that choosing a set of variables has some advantages in terms of pruning over choosing one variable to represent the symmetry group.

Definition 4.12 (Applying the Symmetry Group P_{sym})

Consider a CSP $P = (X, D, C)$ with a weak decomposition (P_1, P_2) and a weak symmetry f on P.

$P_{sym} = (X_{sym}, D_{sym}, C_{sym})$ is a subproblem of P that models applying the symmetry group to a solution of P_1. X_{sym} is the set of SymVars representing the variables of P_1. D_{sym} is the domain for all SymVars and C_{sym} is the set of constraints that model the symmetric group induced by f. A solution of P_{sym} represents an element of the symmetric group induced by f.

Example 4.13 *In the Magic Square Problem:*

Consider the magic square example from the Example 4.2. In this example the diagonal constraints and the corresponding variables form P_2 which introduces row and column permutations in P_1. For the purpose of clarity we restrict ourself in this example to just regarding the column permutation.

In P_1 we can break the column permutation by any symmetry breaking method desired. For every solution s_1 of P_1 we have to check all column permutations of this solution whether they satisfy also P_2 or not.

We introduce n SymVars $symCol$, one for each column of the square with the domain $\{1, \ldots, n\}$. An assignment $symCol_i = j$ represents the permutation of the i-th column of s to the j-th column of the solution s'.

To guarantee that the solutions found by P_{sym} are valid permutations an alldifferent constraint is stated on the SymVars.

Subproblem P_{sym} (Perform the column permutations):

Variables X_{sym} :

$\text{symCol}_i, i \in N, N = \{1, \ldots n\}$ *(one SymVar for each column)*

Domains D_{sym} :

$\forall i \in N : \text{symCol}_i \in \{1, \ldots, n\}$ *(all columns of the square)*

Constraints C_{sym}:

$alldifferent(symVar)$ *(each column in the permuted solution is assigned)*

In the example from Page 99 the according SymVar assignments would be $symCol = (1, 2, 4, 3)$.

4	5	11	14
7	16	2	9
10	1	15	8
13	12	6	3

4	5	14	11
7	16	9	2
10	1	8	15
13	12	3	6

Figure 4.10: Left is a solution s_1 of P_1 and right a symmetric equivalent s_1' via a column permutation that also satisfies P_2.

To incorporate P_{sym} in the model we need to reformulate P_2 slightly. The modification applies to the constraints in C_2. This has to be done since the original constraints are stated on the variables in P_1. But we do not consider the variables of P_1 directly but the symmetric equivalents determined by the SymVars. Therefore, we incorporate the SymVars in the constraints.

In the following we state the complete magic square problem using SymVars. We do not state symmetry breaking constraints because we want to concentrate on the weak symmetry part.

Example 4.14 *The Complete Magic Square Problem Model Using SymVars*

Note that for simplicity only the column permutation is considered to be weak. Therefore, only SymVars for the columns are introduced.

P:

Variables X: $\text{square}_{ij}, i, j \in N$ *(the magic square)*
$\text{symCol}_i, i \in N$ *(the SymVars for the columns)*

Domains D:

$\forall i, j \in N : \text{square}_{ij} \in \{1, \ldots, n^2\}$
$\forall i \in N : \text{symCol}_i \in N$

Constraints C:

$$C_1 : \begin{cases} \forall i \in N : \sum_{j \in N} \text{square}_{ij} = m & \textit{(each row sums up to } m\textit{)} \\ \forall j \in N : \sum_{i \in N} \text{square}_{ij} = m & \textit{(each column sums up to } m\textit{)} \\ \text{alldifferent(square)} & \textit{(all numbers are assigned)} \end{cases}$$

$$C_{sym} : \{ \text{ alldifferent(symCol)} \quad \textit{(a permutation of the columns)}$$

$$C_2 : \begin{cases} \sum_{i \in N} \text{square}_{i,\text{symCol}_i} = m & \textit{(diagonal sums up to } m\textit{)} \\ \sum_{i \in N} \text{square}_{i,n+1-\text{symCol}_i} = m & \textit{(anti diagonal sums up to } m\textit{)} \end{cases}$$

4.3.3 Solving Ordering of the Subproblems

The reformulation of P using P_{sym} induces the following solving order:

1. search a solution for P_1
2. instantiate the SymVars in P_{sym} to determine a symmetric equivalent
3. check whether the symmetric solution of P_1 determined by the SymVar assignment satisfies P_2 and if so extend the solution to a full solution

When a solution to P_1 is found the whole solution class is investigated by P_{sym} and passed consecutively to P_2. When all elements are checked or ruled out P_{sym} is exhaustively investigated and backtracking will free variables of P_1 such that a new solution to P_1 is found. This process is repeated until P_1 is exhaustively investigated.

4.3.3.1 Ordering in P_{sym}

Since P_{sym} consists of a set of variables this implies that a variable and value ordering is possible to perform. The SymVars are also instantiated in a tree search. That means that the symmetric solution is constructed bit by bit with each SymVar instantiation. The order in which the SymVars are instantiated is not necessarily fixed such that it is possible to apply an ordering heuristic. Also the value ordering is not fixed and can be done by an ordering heuristic. This way it is possible to investigate promising permutations first. Also it is possible to prune infeasible values from the future SymVars. Since the constraints of P_2 can be used for pruning purposes as soon as the first SymVar is instantiated. Therefore, applying the symmetry group via P_{sym} gives us more possibilities in investigating the group. Also this approach enables us to rule out whole subgroups of the symmetry group which is done by propagation and backtracking in P_{sym}.

4.3.3.2 Combining Weak and Standard Symmetry Breaking

Weak and standard symmetries may occur at the same time in a problem. We have seen in Section 3.3.8 that combining symmetry breaking methods in standard symmetry breaking causes problems. This is since they exclude different representatives of the solution classes such that solutions could be lost.

Since the weak symmetry is transformed to a standard symmetry on a subproblem it is possible to break both symmetries concurrently (but still with the same symmetry breaking method). The only difference is that for the weak symmetry P_{sym} has to be incorporated such that no solution is lost. Weak symmetry breaking by this approach does not inflict with standard symmetry breaking and combining both makes the symmetry breaking effort more powerful.

Theorem 4.15 *Weak symmetry breaking via SymVars does not conflict with standard symmetry breaking and vice versa.*

Proof.

Since weak symmetry breaking does not exclude solutions of P no solution is lost. Proper symmetry breaking does only exclude solutions that are symmetric on P.

We just have to show that proper symmetry breaking does not exclude a weakly symmetric solution. But since weakly symmetric solutions cannot be mapped on each other (only with respect to the variables in X_1) they are not excluded by proper symmetry breaking by definition.

To solve P we consider the partial solution $s_{P_{sym}}$. When a solution is found, the search backtracks and reconsiders values for the SymVars to determine a new solution. All these solutions are symmetric equivalents to the solution s_{P_1}. Only when the search backtracks and variables in X_1 are reconsidered, a solution for a different equivalence class can be found.

By using SymVars we can break the symmetry in P_1 but do not lose any symmetric solution in an equivalence class.

4.4 Related Work

Weak symmetries recently were also discovered in different fields of research. Since research developed independently there are different definitions for symmetries that act only on a subproblem.

4.4.1 Weak Symmetry Definitions

In [17] the different definitions and fields where weak symmetries arise are discussed. Weak Symmetries arise in Model Checking and Planing and are called *almost symmetry* in these fields.

A different viewpoint on weak symmetries comes from Harvey [45]. He extends the definition of relaxation such that it covers symmetries.

Definition 4.16 (Relaxation of a Problem [45])

A relaxation R of a problem P is a weakening of the constraints of P such that any solution of P is a solution of R.

Definition 4.17 (Symmetric Relaxation)

A symmetric relaxation SR of a problem P is a relaxation of P such that SR has more symmetry than P.

This definition corresponds to weakly decomposable problems where SR is P_1. But the definition is not general enough for our purposes. In symmetric relaxations only constraints can be relaxed. In optimisation also the variable representing the objective value has to be relaxed for example.

4.4.2 Alternative Approaches and Strategies for Breaking Weak Symmetries

Harvey [45] regards the problem of weak symmetries from a group theoretical point of view. Harvey calls the technique he uses *symmetric relaxation*. The approaches of symmetric relaxation and weak symmetry breaking were developed independently.

In this approach a relaxed problem SR of the original problem P is solved. A solution s to SR represents the set of solutions generated by the symmetry group G of SR [45]. Further he states that searching a solution to P decomposes into two related problems: finding a solution s to SR and finding an element of the symmetry group G that maps s to a solution of P.

Although this is also the idea behind weak symmetry breaking it slightly differs from the weak symmetry approach. Sometimes it is not enough to just determine a symmetric solution to P_1 to solve also P. If P_2 contains also variables these variables have to be assigned as well to find a solution to P.

The techniques introduced in [45] to find a symmetric equivalent of a solution s are different to those used in weak symmetry breaking. In the latter case we use modelling to investigate the equivalence class of a solution while Harvey uses group theory via an external package called GAP [38].

In [45] several approaches how to investigate the symmetry group are introduced. We will shortly introduce them in the following.

The Two-Phase Method:

The idea behind the two-phase method is to solve the two subproblems (solving the relaxed problem SR and find an element of the symmetry group that map the solution of SR to a feasible solution of P) sequentially. This corresponds to the approach we use for weak symmetry breaking. But the second subproblem is solved here by calling GAP for an element and if it does not map the solution of SR to a solution of P a different element is called.

The Switching Method:

In the switching method both subproblems are solved simultaneously. In practice a solution to SR is sought and an element g of the symmetry group that maps the solution of SR to a solution of P. It is checked at every time whether the partial assignment p of SR together with g satisfies P. By default the identity is chosen for g and search is performed in SR until a no-good is discovered. In this case a different element g' is sought such that $g'(p)$ satisfies P. If such an element g' is found search continues in SR. If there is no such element then search backtracks in SR.

4.4.2.1 Comparing the Approaches

Both approaches the one of Harvey [45] and the one introduced in this thesis basically are based on the same facts.

- a part of the original problem P_1 is (more) symmetric while the whole problem P is not
- a solution s for the subproblem is sought and a corresponding group element g of the symmetry group such that gs satisfies P.

In both approaches there are no restrictions in the methods used to find a solution for the first subproblem. The difference in the approaches lies in the techniques used to investigate the symmetry group (solving P_{sym} in the context of this thesis).

The methods using GAP provide more freedom when the symmetry group is investigated. The switching method demonstrates that it is possible to interrupt the search in P_1 at a nogood and check whether a symmetric equivalent is feasible. In this case search in P_1 continues and backtracks otherwise. This is not possible in our modelling method. On the other hand we can impose an ordering heuristic on the SymVars such that we can control the order in which the symmetry group elements are investigated. An other important fact is that in P_{sym} pruning and backtracking can occur such that many elements of the symmetry group are pruned and do not have to be tested. Using GAP the elements are just generated and passed to the solver.

There are no computational results on symmetric relaxation techniques such that there is no comparison of the efficiency of the two approaches. The approach of Harvey is more flexible which is demonstrated by the switching method. Also it is possible to use all operations of GAP which might lead to an efficient investigation of the symmetry group. Still the main disadvantage of this approach is that GAP has to be called as an external program. This costs a lot of time such that the efficiency of using GAP may be dominated by the communication time between the solver and GAP. Including GAP in a solver, i.e. rewrite the GAP code in the solver code would reduce the communication overhead. But GAP is a complex software package that involved the work of many leading researchers in computational group theory such that rewriting the code is not possible for non-professionals.

An other disadvantage from our point of view is that the user must be confident with group theory and using GAP. Also the methods shortly introduced must be implemented by the user in order to benefit from them. The solver used must be capable of support of calling external programs or there must be a GAP interface which may not be the case for every constraint solver.

4.5 Benefits, Limitations and Upgrades of the Approach

Mostly an advantage in one field is bought by a disadvantage in another field. The same applies to our approach. Here we discuss the benefits and limitations of the approach. Also we discuss some possibilities to upgrade weak symmetry breaking to reduce redundant search if there is any.

4.5.1 Benefits

4.5.1.1 More Symmetry Breaking

The benefit of the approach is clearly that the weak symmetry of the problem can be broken und a lot of search effort can be saved in P_1. This was not possible before since standard symmetry breaking would have excluded non-symmetric solutions. Each solution of P_1 represents the whole equivalence class of this solution. But these solutions are not equivalent for the remaining subproblem P_2. Therefore, *all* these solutions have to be checked explicitly which is done in P_{sym}. Although this takes a lot of time it means that a lot of potential solutions are checked successively. Therefore, the ratio of solutions investigated in a certain amount of time is high once a solution to P_1 is found.

4.5.1.2 Easy Usability and Availability

Every CSP has to be modelled in order to solve it with a CP solver. Weak symmetry breaking is completely based on modelling. Therefore, every constraint programmer can use it without much knowledge in group theory or other techniques. No code has to be written and incorporated to the solver. Weak symmetries can be broken as soon as they are identified.

4.5.1.3 Control of Investigation Order of the Symmetry Group in P_{sym}

The investigation of the symmetry group is modelled simply with variables and constraints. It is also possible to state variable and value ordering heuristics. This way it can be controlled how the symmetry group is investigated. Also pruning leads to a reduction of the search space. This has a benefit over simply checking the elements of the symmetry group without control and the possibility to prune infeasible elements.

This also allows to apply heuristics that investigate different regions of the search space. In scenarios where the solving time is bounded it is likely that incomplete search has to be chosen. If the symmetry group is investigated only partially it could be desirable to explore different regions of the search space such that completely different permutations are considered. That means to investigate permutations that share a small subpath. Two permutations π_1 and π_2 have a common subpath if they are identic from the root of the search tree up to a certain level i. That means up to the level i the SymVars $sv_1, \ldots, sv_i$ have the same assignments and it holds $\pi_1(k) = \pi_2(k), 1 \leq k \leq i$.

4.5.2 Limitations

The weakly decomposition of a problem P has the drawback that no pruning on the constraints of P_2 is possible. The constraints of P_2 are stated also over the SymVars. Since the SymVars are not instantiated when P_1 is solved the constraints of P_2 cannot be used for pruning in P_1. In worst case that means that a partial solution is extended that is infeasible for P_2. But even if reasoning on the constraints of P_2 would be possible it is not possible to simply backtrack. Since each (partial) assignments in P_1 represents the whole equivalence class all solutions would have to be ruled out before backtracking in this assignment could take place. This corresponds to the switching method of Harvey [45].

4.5.3 Importance of Modelling

Modelling stands at the beginning of every CSP. With a model even for $\mathcal{NP}$-hard problems it can often be decided whether it is fruitful to solve the problem using CP technology or not. Also modelling can be used to characterise instances of a problem. For example, if a solution is found to an instance within a given time limit the instance is easy to solve and hard otherwise. Also it can be used to determine whether a certain quality of solution exists for the problem. In this case a solution has to be found with an objective value higher than a specified bound. This is absolutely realistic since in real-life often online-optimisation is performed

and one is looking for a good solution or just whether there is a solution instead of an optimal solution. That means that the problem is just partially investigated instead of exhaustively like it is the case in most academic studies.

Therefore, models enable the user to evaluate a problem before spending much time for implementing a new algorithm let alone find a suitable one. Still modelling is not straight forward and also much effort has to be put in the task to find a good model. But the developing times are faster and the approach is more flexible the constructing algorithms. So modelling is the first choice in rapid prototyping solving ideas and classifying a problem.

Since modelling is one of the foundations of solving CSPs with constraint programming all efforts should be made to investigate the possibilities of modelling. This thesis is actually a tribute to this effort since we introduce a technique that enable us to break weak symmetries just be modelling. That means that a new class of problems can now be investigated by pure modelling with a gain in efficiency.

4.5.4 Extensions

It is possible to extend and adjust the method of weak symmetry breaking for several applications. This may involve programming such that the approach then is not purely on modelling. In Chapter 5 we will discuss an algorithm that can handle a specific class of problems very efficiently. This algorithm works in conjunction with our weak symmetry breaking method.

4.5.4.1 Avoid Stabilizers of Weak Symmetries

By using conditional constraints, it is possible to upgrade the approach for some kind of problems. The specific feature of these problems is that the columns (or rows, depending on the objects the weak symmetry acts on) cannot be ordered strictly lexicographically. In this case some columns may have assigned the same values and are therefore identic. The SymVars of these columns represent the same values and permuting them leads to the same solution.

What we want to do is identify the stabilizer of this solution to P_1 and exclude it from the search.

Example 4.18 *Consider a solution to P_1 consisting of a matrix $A^{m\times n}$. Consider further that the two columns i and j of the matrix have identical assignments: $a_{.i} = a_{.j}$. That means that the corresponding SymVars sv_i, sv_j represent the same values. An SymVar assignment $sv_i = j, sv_j = i$ results in the same solution as assigning the identity.*

To identity the stabilizer (with respect to the weak symmetry) means to identify identic columns of the solution. Therefore, the columns are checked whether they are pairwise different. For each pair that is not different a constraint is stated that orders them.

Example 4.19 *Consider the example above. In this case a constraint is stated that orders the SymVars sv_i, sv_j: $sv_i < sv_j$.*

These constraints that are only stated in the case of a non-empty stabilizer are part of P_{sym} since they prohibit assignments on the SymVars.

If the matrix in P_1 is lexicographically ordered only $n-1$ checks have to be performed. If the matrix is not ordered $\frac{n^2-n}{2}$ checks have to be performed.

For some problems like the magic square problem the stabilizer is always empty since the values to be assigned are all different.

4.5.4.2 Partial Weak Symmetries

In some applications partial symmetries arise. That is for example in column permutation that not all columns can be permuted but just a subset. The rest is fixed. If this symmetry is weak then we have a partial weak symmetry. This can easily be modelled in P_{sym} by assigning the identity to each of these SymVars.

Example 4.20 *Partial Permutations*

Consider a matrix $A^{m \times n}$. Further all but the first four columns can be permuted and that permutation is weak.

P_{sym} is then modelled as follows:

Variables:

$SymVar_i, i \in \{1, \ldots, n\} = N$

Domains:

$\forall i \in N : SymVar_i \in N$

Constraints:

$alldifferent(SymVars)$

$\forall j \in 1, \ldots, 4 : SymVar_j \quad = \quad j$ *(fixing the first four columns)*

This way the first four columns are fixed in the matrix and only the last $n-4$ columns are permuted.

4.5.4.3 Using the Minimal Set of Generators for the SymVars

The chessboard symmetries comprise a nice structure. The minimal set of generators of the symmetry group consists of two elements (turning the board by 90 degree and turning the board around the y-axis). Applying an element of the symmetry group preserves the structure of the chessboard. Fields that were neighboured before are also neighboured after applying the symmetry. We can use this structure to reduce the number of SymVars we need. Instead of one SymVar for each queen we just need 2 SymVars. One for the 90 degree turn (with a domain of 0 to 3) and one for the y-axis turn (with a domain of 0 to 1). These two variables are used in the constraints that induce the weak symmetry.

4.5.5 Conclusion

Weak symmetries are formally identified and introduced in this chapter. Also it is shown that weak symmetries act on a subproblem as a normal symmetry. Standard symmetry breaking *can* be performed on this subproblem *if* the solutions excluded

by the symmetry breaking method are not lost. This is achieved by our approach using SymVars to model the search in the symmetric group.

In research often subproblems of larger problems are considered and evaluated. Unfortunately a real world problem consists of subproblems that cannot be fully decomposed. Therefore, an optimal solution to a subproblem may not be part of an optimal solution for the global problem. Symmetries in these problems are often weak and could not be identified or broken. Using our technique it is now possible to investigate these larger problems more efficiently exploiting the weak symmetries and reducing the search space. Therefore, our approach helps enlarging the fields in which CP can be applied successively.

Chapter 5

Efficient Symmetry Group Investigation for Separable Objectives

We consider a method that reduces the number of elements of the symmetry group to consider for weakly symmetric problems with special properties. The approach can also be used for permutation problems in general such that this approach is not limited to weak symmetry breaking. The idea is to store partial permutations during the solving process and re-use them for future solutions. This reduces the number of weakly symmetric equivalents to consider for these solutions. The idea can be applied to problems where the weak symmetry is introduced by a *separable* objective function. We present the theoretical soundness of the idea and a prototype algorithm that could be incorporated as a global constraint to the constraint solver. This approach is not limited to weak symmetry breaking via SymVars. It can be combined with other approaches like the one proposed by Harvey [45]. In Section 5.1 we introduce in a special class of optimisation problems. Those with a separable objective. This is carried on and further explained in Section 5.2. A problem that has a separable objective is shortly investigated in 5.4. Section 5.5 shows how the approach proposed for separable objectives could be further investigated. In Section 5.3 we introduce a more general version that just relies on the fact that the objective function is separable. In Section 5.7 we introduce a version that can be used for problems with special properties for the remaining subproblem at a search state that is based on the ideas of [80].

5.1 Introduction

In optimisation problems often the objective function is separable in the columns/rows of a variable matrix. For each column/row a partial objective value can be computed that is independent from all other variable instantiations of the matrix. The objective value then is an aggregation (for example the sum) of these partial values. The objective function f of a solution S decomposes to: $f(S) = f_1(s_1) \oplus f_2(s_2) \ldots, \oplus f_k(s_k)$, where it holds:

$s_i \subset S, \forall 1 \leq i \leq k$

$s_i \cap s_j = \emptyset, \forall 1 \leq i < j \leq k$.

The subsets s_i are for example columns or rows of the variable matrix.

The operator $\oplus$ can be $+, -$, logical AND (for feasibility), etc. In case the operation is addition then the values of the parts of the solution are just added. In case the operation is feasibility each subset of the solution can be evaluated for feasibility. That means if one subset is infeasible the search can backtrack.

As the operation *logical AND* indicates this concept of separability can also be applied to constraints in general and not only to the objective function (which is a special case of a constraint). We will concentrate in this chapter on separable objective functions only and do not consider logical AND for the operator $\oplus$.

If a separable objective function induces a weak symmetry on a problem then our approach can be used to save time in investigating the equivalence class for weakly symmetric solutions. The idea is to store partial results from investigating the equivalence class of certain solutions and re-use them for the investigation of so-called neighbouring solutions. Thereby, the number of solutions to check explicitely is reduced.

5.2 Separable Objectives

5.2.1 Prerequisites

For this chapter we consider the following problem structure: The CSP $P = (X, D, C)$ is an optimisation problem, i.e. there exists an objective function f that assigns each solution S of P an objective value $f(S) = v$ which leads to a ranking of all solutions. P weakly decomposes into P_1 and P_2 where the weak symmetry in P_1 is a column permutation of a search variable matrix $\chi^{m \times n} \subset X$. P_2 just consists of the objective function as a constraint on χ: $P_2 = (\chi, D_\chi, f)$. A solution S to P consists of the permutation s_π of the solution s to P_1 and the objective value v associated to s_π via the function f. The column permutation symmetry in P_1 is broken by a symmetry breaking method. P_{sym} consists of investigating the symmetric equivalents using n SymVars sv_i representing the columns of χ. An assignment to all SymVars sv_i therefore models s_π which is a solution to P_{sym}.

Convention: Small capital variables s are solutions to the subproblem P_1 while large capital variables S are solutions to the whole problem P.

We also define a *partial permutation of order i* in the following. For our purpose a partial permutation of order i is a permutation of just some consecutive variables while the rest of the variables is not assigned yet. This represents a search state for P_{sym} where some SymVars are assigned already and others are still unassigned.

Definition 5.1 (Partial Permutation of i variables)

*Consider a permutation π of n variables. A **partial permutation of i variables**, π_i is an assignment of the first i variables on the domain $\{1, \ldots, n\}$ with $\pi(j) \neq \pi(k)$ for $1 \leq j, k \leq i$.*

A partial permutation i variables, π_i implies that i values have been assigned to the first i variables (in our case the SymVars) and $n - i$ values have not been assigned.

5.2.2 Separable Objectives

Separable objectives have the special feature that the objective function f itself can be broken down to independent sub-functions $f_1 \ldots, f_n$, each defined over a variable set $\chi_i \subset \chi$. The variable sets of the sub-functions form a partition of χ.

For our purpose we regard only functions where the subsets χ_i form a structure like the columns or the rows of χ.

Definition 5.2 (Separable Objective)

*An objective function stated over a search variable matrix χ is **separable** in the disjunctive subsets $s_1, \ldots, s_k \subset S$ of a solution S if*

- *the contribution of an assigned subset s_i to the objective value is independent from the assignments of all other subsets $s_j, j \neq i$*
- *the objective value can be computed from the separate contributions of all subsets s_i*

For our purpose the subsets s_1 are always the rows or the columns of a variable matrix. Substituting the subsets with columns the definition is:

An objective function stated over a search variable matrix χ is separable in the columns if

- the contribution of an assigned column to the objective value is independent from the assignments of all other columns
- the objective value can be computed from the separate contributions of all columns

We choose the subsets to be rows or columns because in this case we can regard row or column permutations. Note that especially in real-world optimisation there are a lot of problems that introduce weak symmetries. They are often separable in the desired way since the optimisation function itself introduces the weak symmetry.

5.2.3 Separable Objectives and Weak Symmetries

If the variable matrix χ comprises for example a column permutation symmetry on P_1 but not on P the symmetry is weak on P. For the sake of simplicity we consider the weak symmetry to be a column permutation from now on. If the objective function is also separable in the columns we can store the partial permutations and the achieved partial objective value. These partial permutations of a solution s can be re-used for a solution s' if s and s' are *neighbouring* solutions.

Definition 5.3 (Neighbouring Solutions)

Given two solutions s and s' to a problem P. Each solution consists of a search variable matrix $\chi^{m \times n}$.

*s and s' are **neighbouring solutions** if the following holds:*

$\exists i \in \{1, \ldots, n\} \forall j \in \{1, \ldots i\} : s_j = s'_j$, where s_j is the j-th column of the solution s and analogously for s'.

Definition 5.4 (Neighbourhood Degree)

Given two neighbouring solutions s and s' to a problem P. Each solution consists of a search variable matrix $\chi^{m \times n}$.

The highest index i for which s and s' are neighbouring is called the ***neighbourhood degree*** *of s and s':*

$nbhDeg(s, s') = \max_{i \in \{1,\ldots,n\}} \forall j \in \{1, \ldots i\} : s_j = s'_j$

Note that we define neighbourhood as a successional feature. If two solutions are neighbouring for a certain index i than they are also neighbouring for all $j < i$. The reason for that is to achieve a tradeoff between the efficiency and the space complexity of the proposed method. It is without loss of generality possible to define the neighbourhood relation just for single and not for successional indices. But this would result in a super-exponential space consumption such that the method would not be applicable in practice.

In our scenario s and s' are solutions to P_1 and they are subject to column permutation to determine the solution for the CSP P. That means that the whole equivalence class for all solutions to P_1 have to be checked explicitly. In fact for each solution $n!$ permutations have to be considered.

Consider now that s and s' are neighbouring with the degree k. In this case the permutation of the first i columns is part of both solutions s and s'. Without loss of generality s is found before s' in the search. The idea is to re-use the results of the partial permutations of the first k columns from s for the computation of the permutations to s'. By doing so the number of permutations that have to be checked explicitely for s' reduces from $n!$ to $\frac{n!}{k!}$ (See Section 5.3).

Therefore, we store the partial permutations as well as the achieved objective value when checking all permutations π of s. In fact, we do not need to store *all* partial permutations but only *dominating* partial permutations.

Definition 5.5 (Dominating Permutation)

A permutation π dominates an other permutation π' with respect to s if

$f(s_\pi) \geq f(s_{\pi'})$*, where $f()$ is the objective function.*

If π dominates all other permutations with respect to s, π is a ***dominating permutation****.*

Definition 5.6 (Dominating Partial Permutation of i variables)

Consider π_i and π'_i to be partial permutations of i variables.

π_i dominates π'_i with respect to s if

- *s_{π_i} and $s_{\pi'_i}$ have assigned the same set of values (but to different variables)*
- *$f(s_{\pi_i}) \geq f(s_{\pi'_i})$, where $f()$ is the objective function*

If π_i dominates all other partial permutations $\bar{\pi}_i$ that have assigned the same set of values, π_i is a ***dominating partial permutation with respect to*** *s.*

Caveat: In the following we will speak of partial permutations instead of partial permutations of i variables if the index i is not explicitly needed.

In fact we store for each subset of the set of columns one dominating partial permutation. We can do this because we are only interested in a partial permutation that achieves the best objective value for each subset of the columns. The number of dominating partial permutations is considerably smaller than that of all partial permutations.

Storing the partial permutations can be done during the search process of P_{sym}. After each instantiation of a SymVar it is checked whether this partial permutation dominates a previously found partial permutation on the set of assigned columns. If so the new partial permutation is stored.

If all permutations π have been considered for s, search backtracks to find a new solution s' to P_1. The neighbourhood degree k of s and s' is determined and the permutation of s' is started. Since $nbhDeg(s, s') = k$ the first k columns of s and s' are identical. Due to the separable objective function f that means that a partial permutation of the first k columns achieves the same partial objective value for s and s'. Since we already performed these partial permutations on s we do not want to perform them again but use the stored dominated partial permutations.

Therefore, we instantiate only the last $n - k$ SymVars, instead of all n SymVars. When all these SymVars are assigned there are k remaining values that were not assigned. For these values we recall the stored dominating partial permutation which is applied to the remaining k unassigned SymVars. Together this forms a permutation of all columns. When the objective value is determined the search backtracks and performs search on the last $n - k$ SymVars. For each permutation of the last $n - k$ SymVars the dominating partial permutation for the not assigned columns are recalled and the permutation problem is reduced to $\frac{n!}{k!}$ permutations to check.

The gain in the method is not only the reduction for *one* other solution. But it holds for each solution $\bar{s}$ that is neighbouring with s. Therefore, the stored partial solutions can be used for each $\bar{s}$. Depending on $nbhDeg(s, s')$ more or less permutations can be omitted for $\bar{s}$.

All solutions with a neighbourhood degree of 0 to any previous solution are called *cardinal solutions*. If $nbhDeg(s, \hat{s}) = 0$ then the first column in both solutions is different: $s_1 \neq \hat{s}_1$.

Definition 5.7 (Cardinal Solution) *A solution s that has a neighbourhood degree of 0 with all previous found solutions is called a* ***cardinal solution****.*

Only for cardinal solutions partial permutations are stored whereby memory is erased with each new cardinal solution.

5.3 Approach for Separable Objectives

As outlined before the idea of the approach is to store and re-use partial information from solutions investigated before. We will describe the approach in this section more detailed. It consists of two phases. The first is storing partial solutions during solving P_{sym} for cardinal solutions. The second is calling these stored data for

solving non-cardinal solutions. We introduce a variable ordering on P_1 and P_{sym}. This has to be done in order to find the solutions in the desired order.

5.3.1 Variable Ordering for the Approach

5.3.1.1 Ordering on P_1

For our approach we consider a fixed variable ordering for the variables in χ. The matrix is assigned columnwise beginning with the smallest index 1 up to n.

The reason for that is that we want the backtracking in a way such that

- as many columns as possible keep fully instantiated
- backtrack first occurs in the column with the highest index that still has variables assigned

By doing so we achieve the following:

1. all solutions neighbouring to a cardinal solution s are found consecutively
2. the neighbourhood degree between s and any new solution is decreasing (which means that the highest neighbourhood degree is found first)
3. once a solution $\hat{s}$ is found that is not neighbouring with s (i.e. the neighbourhood degree between s and $\hat{s}$ is 0) no further solution is neighbouring with s.

This way the assignment of the columns change from "right to left"during the search which produces the desired feature of decreasing neighbourhood degree.

Theorem 5.8 (Neighbourhood Decrease)

Consider the variable ordering in P_1 for χ to be performed columnwise increasingly. Then the solutions are found in a way such that the neighbourhood degree of a cardinal solution s and any solution s' found before the next cardinal solution $\hat{s}$ will never increase.

Proof. If the variables are assigned in the proposed order then the assignments in the columns will change from right-to-left changing first columns with higher indices. Consider for the cardinal solution s and a solution s' that the neighbourhood degree is k. Any solution s'' found after s' has at least one of the columns with an index lesser than or equal k changed in comparison to s. Therefore, the neighbourhood degree cannot increase for the solutions between two cardinal solutions.

As soon as the first column is reconsidered the neighbourhood degree to all previously found solutions is 0. This trivially holds for the first solution as well.

5.3.1.2 Ordering on P_{sym}

We consider here fixed but different variable orderings depending on the kind of a solution (cardinal or non-cardinal). For cardinal solutions s we assign the SymVars increasingly from sv_1 to sv_n. For a non-cardinal solution s' with $nbhDeg(s, s') = k$ we assign the SymVars decreasingly from sv_n to sv_{k+1}.

Note that the variable ordering for cardinal solutions is no limitation. All permutations of s have to be considered anyway in order not to lose solutions. We save i-elementary subsets of the set of columns with the property that the first i SymVars are instantiated during the search. This is done to save storing capacity. By doing so for each $2 \leq i \leq n-1$ we store all i-elementary subsets of the set of columns. The i-elementary subsets stored represent all partial solutions of a permutation of the first i SymVars on the set of columns.

When P_{sym} is exhaustively investigated for each such a subset a dominating partial permutation is stored.

Theorem 5.9 (Dominating Permutation)

After all permutations are performed for a cardinal solution s a dominating partial permutation is stored for each subset of the columns.

Proof. For a partial permutation $\pi_c(s)$ the set of assigned columns c is determined. For this subset it is checked whether $\pi_c(s)$ achieves a better objective value than the best found partial permutation $\pi'_c(s)$. If so, $\pi_c(s)$ is stored since it dominates $\pi'_c(s)$. When all permutations are performed for each subset of the columns a partial permutation is stored. Since only dominating permutations are stored and the search is exhaustive the last stored permutation in each subset is dominating.

For non-cardinal solutions we profit from the stored partial permutations. We only have to assign SymVars with index $i > k$. The assignment of the first k SymVars is determined by the partial permutation of k variables that assign exactly these k values that where not assigned to the SymVars with index $i > k$.

Example 5.10 *Consider a CSOP $P = (X, D, C, f)$ with a variable matrix with 8 columns. The objective function f is separable in the columns which induces the weak symmetry of column permutation. Therefore, a weak decomposition (P_1, P_2) is chosen. Consider also a cardinal solution s and a non-cardinal solution s' with $nbhDeg(s, s') = 5$ of P_1. s and s' are identical in the first 5 columns. Therefore, a partial permutation of 5 variables achieves the same contribution to the objective value. Both solutions differ at least in the 6th column. P_{sym} for s' is now just to assign the SymVars representing column 6to8.*

Assume the following assignment for these SymVars: $sv_6 = 3, sv_7 = 5, sv_8 = 1$. The values $\{2, 4, 5, 6, 7\}$ are not assigned yet. These values have to be assigned to the SymVars $sv_1, \ldots, sv_5$. But when P_{sym} for s was investigated these values were already assigned to exactly these SymVars. These SymVars represent the same columns in s and s' and a partial permutation of k variables for s achieves exactly the same results as a partial permutation of k variables for s'. The subproblem of a partial permutation of k variables is identical for both solutions. Therefore, we can recall the domination partial permutation of k variables for the values $\{2, 4, 5, 6, 7\}$ and assign it to the SymVars $sv_1, \ldots, sv_5$.

5.3.2 Storing Partial Permutation

Partial permutations are just stored for cardinal solutions. This is in particular the first solution s found for P_1. The next cardinal solution $\hat{s}$ is the first found that is not neighbouring with s (i.e. $nbhDeg(s, \hat{s}) = 0$). The next cardinal solution $\hat{\hat{s}}$ is the first found that is not neighbouring with $\hat{s}$ and so on. All solutions found between two cardinal solutions are neighbouring with the first found of these two. For a cardinal solution s all permutations of s have to be considered in order not to lose solutions. For all other solutions only partial permutations have to be considered since the rest of the permutation is taken from the cardinal solution. The number of partial permutations to consider for a solution depends on the neighbourhood degree of this solution and its cardinal solution.

5.3.2.1 Process of Storing

Consider a cardinal solution s to P_1, without loss of generality the first found solution. To find a solution to P, s has to be permuted. Therefore, the symmetry variables are assigned. The assignment is done such that the columns of the matrix χ, represented by the SymVars, are assigned form the lowest index to the highest. After each instantiation of a SymVar the partial permutation represented by this partial assignment is stored if it dominates all previously found solutions on the set of assigned values. More specifically the objective value and the concrete assignment of the SymVars is stored. Since the objective is separable the objective value can be obtained.

Example 5.11 *Consider the following partial SymVar assignment:* $sv_1 = 3, sv_2 = 1, sv_3 = 4$. *This means the first column of* χ *is permuted to the third column and so on. Consider that the objective value for this partial assignment is* 34. *Then the data* $\langle(3, 1, 4), 34\rangle$ *is stored for the partial permutation. If the partial assignment is extended by* $sv_4 = 6$ *achieving a objective value* 42, *then* $\langle(3, 1, 4, 6), 42\rangle$ *is stored for the partial permutation.*

If a different partial permutation achieves a higher objective value than the old one it is overwritten by the better one.

Consider the example above and a new partial SymVar assignment: $sv_1 = 4, sv_2 = 3, sv_3 = 1, sv_4 = 6$ *achieving an objective value of* 50. *The old partial assignment* $\langle(3, 1, 4, 6), 42\rangle$ *is overwritten by* $\langle(4, 3, 1, 6), 50\rangle$.

Although there are $n!$ permutations the number of dominating partial permutations to store is smaller.

Theorem 5.12 (Highest neighbourhood degree k)

For a cardinal solution s with n columns it is sufficient to regard only a neighbourhood degree of $2 \leq k \leq n - 1$.

Proof. A neighbourhood degree of 1 does not have to be regarded since in this case there is only one search variable left in P_2 and there exists only one value for this variable due to the permutation. That means that a solver does automatically assign the value and compute the objective value. Therefore, there is no use in storing one-elementary subsets. A neighbourhood degree of n is not possible because this would

mean that both solutions s and s' have only identical columns which means that $s = s'$. This is impossible since a constraint solver can't find the identical solution again.

That implies that only dominating partial permutations for all 2 to $n-1$-elementary subsets are to be stored. For each subset one dominating partial permutation is stored.

Theorem 5.13 (Storing Capacity)

The size to store all dominating partial permutations is $2^n - n - 2$.

Proof. For each subset of the columns one dominating permutation has to be stored. There are $\binom{n}{i}$ subsets of the size i. Theorem 5.12 implies that we do not need the subsets of size $0, 1$ and n. Therefore, there are $\sum_{2 \leq i \leq n-1} \binom{n}{i}$ subsets which equals $2^n - n - 2$.

In the applications we are investigating n is bound to be 20 at most. The storing capacity is a practical amount and the approach is not only theoretically but can be applied. But the approach cannot be performed for very large instances without restrictions due to memory restrictions. Still Section 5.5.3 introduces an idea such that the approach can be performed also for larger instances but with a lesser efficiency.

5.3.3 Applying Stored Partial Permutations

The stored partial permutations for a cardinal solution s can be used for each solution s' with $nbhDeg(s, s') > 0$. When s' is found and $nbhDeg(s, s') = k$ the first k columns do not have to be permuted anew.

First a permutation of the last $k + 1, \ldots, n$ SymVars is sought. Then it is determined which values of the columns are not assigned to these SymVars. For these values a dominating partial permutation of the first k SymVars is re-called from the stored data. Since the stored partial permutation for these values is dominating it represents an optimal solution for this subproblem.

Therefore, the problem P_{sym} for non-cardinal solutions reduces to investigate only $\frac{n!}{k!}$ permutations instead of $n!$.

Theorem 5.14 (Reduction for Non-cardinal Solutions)

Given a cardinal solution s and a solution s' with $nbhDeg(s, s') = k$.

The number of permutations that have to be explicitly investigated for s' reduces from $n!$ to $\frac{n!}{k!}$.

Proof. Due to $nbhDeg(s, s') = k$ the first k columns of both solutions are identical. Therefore, only the last $n - k$ columns of s' have to be permuted on the n columns of the matrix. For the remaining k free columns the stored dominating permutation can be taken. Therefore, the number of explicitly investigated solutions is $\frac{n!}{k!}$.

5.4 Briefly Investigate a Problem with Separable Objectives

We use the problem from the field of automated manufacturing from Section 4.2.3.5 to demonstrate our ideas. For the sake of clarity we just repeat the problem description we are concerned with in the following.

5.4.1 Problem Description

In the problem certain components must be mounted on PC boards by a mounting machine consisting of several mounting devices. The task is to maximise the workload of the whole machine. We concentrate only on a subproblem of the whole solving process. That is to find a setup of component types for the individual mounting devices to maximise the potential workload.[1]

The machine consists of several mounting devices. Each mounting device has access to a set of component types (called setup) that are to be mounted on the PC boards. In addition each mounting machine has only access to a part of the PC board layout and can therefore only mount components inside this visibility area. The PC board layout is specified by a list of mounting tasks. A mounting task is specified by a component type and a position where to mount this component type.

The problem is modelled as follows: The machine is represented by an $m \times n$ variable matrix $A^{m \times n}$ where m is the number of different component types that can be assigned to a mounting device and n is the number of mounting devices on the machine. The domain of variables $a_{ij} \in A$ is the set of component types. An assignment $x_{ij} = k$ means that a component of type k is placed on the mounting device j in the ith slot.

The constraints:

- No component type may be assigned more than once to a column
- Certain component types may not be assigned together in a column
- Each component type achieves a certain workload when assigned to a column. The workload differs from column to column. This represents the visibility of the mounting device.

In the real-world the matrix χ would have about 10 rows and 6 to 20 columns. Where the most common case is a matrix with 12 columns.

5.4.2 Weak Symmetry, Neighbourhood and the Separable Objective of the Problem

The columns of the matrix χ can be permuted which does not change feasibility. But the assigned component types achieve a different potential workload. Therefore, the column permutation is a weak symmetry.

[1] The actual workload assigned to the devices is a subset of the workload determined in this subproblem. But the higher the possible workload the higher the degree of freedom for the concrete assigning problem not considered here.

Solutions to P_1 (finding a setup for the machine) have a rather high degree of neighbourhood. This is due to the fact that only few changes in the setup constitutes a new solution. This way the neighbourhood degree decreases slowly such that the time spent for storing solutions for a cardinal solution is outperformed by the saving of performing permutations.

The objective function is separable in the columns since the potential workload can be determined for each column separately.[2]

A drawback in the problem is that pruning due to objective value bounding for P_{sym} is not very effective. The reason is that we maximise the objective value and the contributions of each assigned column is strictly positive. That means that mostly the majority of the SymVars have to be instantiated before pruning can be performed. Therefore, applying our technique could save an enormous amount of work.

5.4.3 Efficiency of Applying the Approach

This discussion is held from a theoretical point since the technique has not been applied to the problem yet. Still due to the investigations in [59] we have a lot of knowledge about the structure of the solution space. As mentioned before the neighbourhood degree is very high in the problem. In practice a lot of solutions just differ by two or three columns. In a standard instance with 12 columns that would mean a reduction from 12! to $12^2 - 12 = 132$ permutations for several solutions. The number of solutions is rather high in the problem. This means that even for instances with 12 columns it is not possible to solve the problem exhaustively within reasonable time. Using our method for weak symmetry exploitation a much larger number of solutions can be investigated or the problem could be solved exhaustively for smaller instances which is a large improvement for the problem.

5.4.4 Related Work

We use a domination criterion to reduce the size of solutions to store. The domination criterion can be used since the objective function is separable. There are other approaches that use dominance to speed up the search in different ways. SBDD (Symmetry breaking by dominance detection) [18] for example checks whether a current search state is dominated by a previously found search state. Focacci and Shaw [23] prune search branches that are dominated by other using local search. Smith [80] uses no-good recording to detect whether current search states lead to the same remaining subproblem as previously investigated search states. So far we do not use domination to reduce the number of permutations to consider. But Smith' approach [80] could be incorporated to do exactly so. In the following we explain the approach of Smith.

Normally nogoods are not recorded to avoid them in the future since search will never visit them again. But in some scenarios an assignment can occur that is equivalent to an already visited nogood. In this case if the nogood is recorded to check the equivalence it is possible to backtrack in the actual variable assignment

[2]The objective function is even separable in the rows but since the row permutation is not a weak symmetry this does not help here.

since it will also fail. The equivalence of two assignments is in the sense that both assignments leave the search in the same state. Therefore, if one assignment fails the other will also.

In general two assignments are considered equivalent if they leave the search in the same state.

Example 5.15 *Equivalent assignments*

Consider a string of six variables each with the domain of the digits $\{0, \ldots, 9\}$ *and the values have to be all different. The two search states* $(0, 1, 2)$ *and* $(2, 0, 1)$ *leave the search in the same subproblem. In each search state the future variables* x_4, x_5, x_6 *have the remaining domains* $\{3, \ldots, 9\}$.

If the assignment $(0, 1, 2)$ *cannot be extended to a feasible solution (where the conflict is not within the first three variables) neither can* $(2, 0, 1)$.

Smith discussed in [80] an approach of caching previously visited search states for permutation problems. The idea is to check for an assignment whether it is equivalent to a previously visited search state in the way that the resulting subproblem is the same (i.e. in both assignments the same values have been assigned to the instantiated variables, where also in both assignments the same variables have been already assigned (which is given in a static variable ordering).

The key feature behind this idea is that a search state does not have to be investigated if it is equivalent to a previously visited one *and* there are also no constraints that impose an asymmetry on the instantiated subproblem. We will investigate the last property more closely following in 5.4.5. The problems investigated by Smith in the paper do have that property.

Smith investigates two different problems. A satisfaction problem, the game Black Hole invented by David Parlett [80] and an optimisation problem, talent scheduling [80, 82]. In Black Hole the goal is to produce a feasible sequence of cards from a deck of cards with the following constraints:

- the Ace of space is the first card in the sequence and put aside to build the pile
- all other 51 cards of the deck are laid out in 17 column each with 3 cards, whereby only the top card in each column is visible (if a card is removed the card below is visible on this column)
- the cards are put on the pile sequentially
- a card can be chosen to put on the pile if it is visible
- a card can only be put on the pile if it is one less or greater then the top card of the pile (which is initially the Ace of spades)
- all cards must be placed on the pile

The talent scheduling problem arises in film production. There are a certain number of scenes to be filmed and each scene needs a subset of the actors to participate in it. To minimise costs the actors are just hired from the first scene they participate

in until the last scene they participate in (even if they do not work in between). The task is to minimise the total cost for the film production by scheduling the scenes in a way that the total actors costs are minimal.

Caching the search states is done using an array which length is encoded by a binary integer. In the black hole problem the length is 51 bit since there are 51 variables. A 1 on the k-th bit of the integer indicates that the k-th card is part of the search state. When a search state is to be explored it is checked first whether the actual search state plus the new assignment has already been explored before. If so the search backtracks and if not caches this search state.

5.4.5 Classifying applicable scenarios

Smith [80] states that caching could not be applied in all permutation problems. For example in Langford's problem (prob024 in CSPLib [90] the order of the values assigned is crucial for feasibility. In the Langford's Number problem $L(k, n)$, k sets of numbers from 1 to n have to be arranged successively such that each number $j \in 1, \ldots n$ is exactly j positions form the last number j away.

But Smith does not discuss the classification when a permutation problem can be solved using the caching technique. This is where we are interested in in this section.

As stated above we mention that not only search states have to be equivalent but also the already solved subproblem has to be symmetric. In general that means that all permutations of the instantiated subproblem are fully exchangeable and imply the identical subproblem (not only the equivalent remaining subproblem (i.e. the same domains for the future variables in each equivalent search state). If this is not the case then caching cannot be used without the risk of loosing solutions.

Example 5.16 *Caching Search States*

Consider a simple problem where the variables $x_1, \ldots, x_n$ have to be instantiated and the only constraint is the all different constraint on these variables.

For the sake of simplicity we assume a static variable ordering. Consider further two search states t_1 and t_2 that have the first k variables instantiated. When t_2 is discovered to be equivalent to t_1 it is backtracked since t_1 (and the underlying subtree) has already been investigated.

Variation 1:

Consider now to add a constraint $c_1 : x_1 + x_n < \frac{n}{2}$. In this case t_1 and t_2 are equivalent since the same set of values have been assigned to the first k variables but if the value for x_1 is different in both search states, then c_1 may or may not be satisfied. That means that it cannot be backtracked in t_2 automatically and caching cannot be used.

Variation 2:

Consider now to add a constraint $c_2 : x_1 + \ldots + x_k > 3n$ instead of c_1. In this case t_1 and t_2 are equivalent and the remaining subproblem is identic since the variables $x_1, \ldots, x_k$ are fully exchangeable. Here Caching can be used fruitfully.

When regarding the variation 1 and 2 of the problem one property is crucial: the instantiated subproblem must be symmetric. That means any permutation of the

variable value assignment must leave the search in the same state. Therefore, it is not sufficient to cache just the search state. To each search state the domains of all future variables would have to be stored. This would be very space expensive.

For separable objectives the remaining subproblem is not equivalent such that the approach cannot be applied to our problem.

5.5 Extensions of the Approach

Here we consider some extensions and variations that could be used for the approach. We discuss the advantages and disadvantages of each idea.

5.5.1 Neighbourhood as a Discrete Feature

We limit ourself to regard neighbourhood as a successional feature. This is done to make the approach applicable. The memory consumption is growing super-exponential otherwise which would allow only very small instances to be solved.

5.5.1.1 Advantages

If the neighbourhood is defined discrete, i.e. columns do not have to be successional to count for the neighbourhood degree we do not have to impose a variable ordering on P_1 that is that strict. Columns do not have to be considered increasingly but can be assigned arbitrarily as long as all variables of a column are assigned successively.

5.5.1.2 Disadvantages

The memory consumption is much higher for saving all dominating partial permutations. This is due to the fact that in the successional neighbourhood the values for each k-elementary subset are only assigned to the SymVars $sv_1 \ldots, sv_k$. For a discrete neighbourhood these values could be assigned to any k SymVars. Also there are more saving operations which consume time during the search.

5.5.2 Imposing a Lower Bound k_{min} for the Neighbourhood Degree

We limit ourself to store data only for cardinal solutions. But since the neighbourhood degree is decreasing during search more and more efficiency is lost. This happens for solutions s, s', s'' with $nbhDeg(s, s'') < nbhDeg(s', s'')$, where s is the cardinal solution. In this case the permutation reduction would be better if s' was the cardinal solution. It is possible to impose a lower bound for the neighbourhood degree k_{min} such that the saving for further solutions is higher. That would mean that a solution s' with $nbhDeg(s, s') < k_{min}$ to a cardinal solution s is announced a new cardinal solution and partial permutations for s' have to be stored.

5.5.2.1 Advantages

Imposing a lower bound on the neighbourhood degree k_{min} would guarantee that for each non-cardinal solution at most $\frac{n!}{k_{min}!}$ permutations have to be performed.

5.5.2.2 Disadvantages

For the newly announced cardinal solutions storing operations have to be performed which cost time. Fortunately the storing capacity has not to be extended. Moreover, if for a solution s' and a cardinal solution s it holds $nbhDeg(s, s') = k < k_{min}$, the k-elementary subsets do not have to be computed again. Only subsets with more than k elements have to be stored anew. But for s' $n!$ permutations have to be performed. It is not possible to use the stored k-elementary partial solutions and extend them to an optimal permutation. Only *one* dominating partial permutation is stored for each subset. A different one (even a non-dominating partial permutation) may be extended to a solution with a better objective value that the stored one achieves. Therefore, solutions could be lost by extending the stored partial permutations. We will see in Section 5.7 that for certain problems the approach could be altered in the spirit of [80]. This way even cardinal solutions could be investigated with less work.

5.5.3 Imposing an Upper Bound k_{max} for the Neighbourhood Degree

On the other hand we do not restrict the maximal degree of neighbourhood k_{max}. Since the memory consumption is exponential in the neighbourhood degree it may be necessary to impose an upper bound. That means that only neighbourhood degrees up to k are respected. Clearly this is a loss of efficiency for the method but it makes it applicable.

5.5.3.1 Advantages

The clearest advantage is that the extra memory consumption is under control which makes the approach applicable. Although this clearly limits the theoretically achievable efficiency it also offers us a chance of pruning in P_{sym}. Only permutations of the first k_{max} variables are stored. We only have to investigate the assignment of these variables exhaustively since we do not have to keep track of the optimal partial assignments. That gives us the freedom to prune partial permutations beyond an assignment of k_{max} variables.

5.5.3.2 Disadvantages

Clearly not full efficiency could be achieved since the solutions of the problem may have neighbourhood degrees greater than k_{max} such that theoretically more permutations could be avoided to perform.

5.6 Algorithm for the Exploitation of Weak Symmetry Permutations

The algorithm consists of two core methods. One method is used for storing partial permutations and the second is used for calling the stored solutions. Depending on the kind of solution investigated (cardinal or not) the appropriate method is called. We also explain the data structures used in the approach.

5.6.1 Data Structures

These data structures used can be implemented in several ways. For our purpose we are not regarding the effectiveness of the data structures here but focus on simplicity.

- `InstSymVars`: InstSymVars is a linked list. Each new value is linked at the end of the list. It marks the assignments for the SymVars where the i-th SymVar sv_i is represented by the i-th element in the list and the assigned value to sv_i is the data at this position.
- `assignedCols`: assignedCols is an integer in binary representation. It marks the subset of assigned values for the SymVars. A 1 at the i-th position marks that the value i has been assigned to a SymVar.
- `partObjVal`: partObjVal is an integer that represents the partial objective value for the current partial permutation.
- `PartialPerm`: PartialPerm is an array containing two entries at each index. The first entry is the linked list `InstSymVars` and the second is the integer `partObjVal`. The array is indexed from 0 to 2^n represented by the integer `assignedCols`. The array therefore contains all data of the partial permutations.
- `CardSol`: CardSol is an two-dimensional array that represents the actual cardinal solution.
- `nbhDeg`: nbhDeg is an integer that marks the neighbourhood degree of two solutions.

5.6.2 Algorithm Sketch for Weak Symmetry Exploitation

The algorithm interfaces with the constraint solver at certain events. For example on the instantiation of a variable. We assume for simplicity of code that we can interface with the solver at any time and retrieve all needed information by simply calling it from the solver. These calls are marked in the code with *solver.method*.

Methods are:

- onSolutionP1(s): Code is executed if a solution s to P_1 is returned.
- onInstantion(Var): Code is executed if the variable in the argument is instantiated.
- variableOrdering(Vars): The variables are instantiated in the order they appear in the list.
- onBacktracking(Var): Code is executed if the search backtracks in the variable in the argument.
- assignValues(Values, Vars): The values of the list Values are assigned to the corresponding variable in the list Vars.

The algorithm (see Algorithm 3) basically works by determining the neighbourhood degree. If it is a cardianl solution then all SymVars have to be instantiated and the partial solutions are stored at the according position (see Algorithm 4). If it is not a cardinal solution then only a reduced problem has to be solved and the full solution is completed by the stored partial solutions (see Algorithm 5).

```
Main Procedure
begin
    while solver.onSolutionP1(s) do
        nbhDeg = determineNbhDeg(CardSol,s);
        if (nbhDeg=0) then
            CardSol = s;
            solveCardinalPermutations;
            else
                solveReducedPermutations(nbhDeg);
            end
        end
    end
end
```

Algorithm 3: Main Procedure

```
Procedure solveCardianlPermutations
begin
    solver.variableOrdering(sv_1, ..., sv_n);
    if solver.onInstantion(sv_i) then
        InstSymVars.add(sv_i);
        assignedCols + = sv_i;
        partObjVal = updatePartObjVal(sv_i);
        updatePartialPerm(assignedCols, InstSymVars, partObjVal);
    end
    if (solver.onBacktracking(sv_i)) then
        InstSymVars.remove(sv_i);
        updatePartObjVal(sv_i);
        assignedCols − = sv_i;
    end
end
```

Algorithm 4: Procedure solveCardinalPermutations

5.7 A Modified Version for Permutation Problems with Special Properties

The major drawback in the approach is still the full investigation of P_{sym} for cardinal solutions. In large instances this may consume a lot of time such that not enough time could be spend for neighbouring solutions where the efficiency of the approach comes from. Still it is a large improvement comparing to investigate P_{sym} fully for all solutions of P_1. Still it would be desirable to reduce the number of partial assignments to check for P_{sym}. Indeed there is a technique to do exactly that. Smith [80] caches search states in a permutation problem to prevent investigating the same remaining subproblems over and over.

```
Procedure solveReducedPermutations
begin
    solver.variableOrdering(sv_n, ..., sv_{k+1});
    if solver.onInstantiation(sv_(k + 1)) then
        assignedCols = determineNotAssignedCols;
        solver.assignValues((sv_1, ..., sv_k),(PartPerm[assignedCols]);
    end
end
```

Algorithm 5: Procedure solveReducedPermutations

5.7.1 Caching Search States

The key idea of the idea is that a partial assignment A can be dominated by a partial assignment B, if in both assignments the same set of variables are instantiated and the same set of values is assigned to the set of variables and the search has discovered that B cannot be extended to a feasible solution. In this case also A cannot be extended to a feasible solution since the remaining problem to solve is the same in both cases. But this does only hold if the remaining subproblems of A and B are *identical*.

In the case of optimisation the idea works analogously. Even more the technique induces a domination criteria: A search state A is dominated by a search state B (again the same set of variables and values are assigned) if the following holds for the objective function $f()$: $f(B) \geq f(A)$. In this case A cannot be extended to a solution with a better objective value than B achieves since the remaining subproblem for both search state is identic.

Example 5.17 *Consider for simplicity a permutation of the numbers* $1, \ldots, 4$ *and an objective function that assigns each number a objective value depending on the position in the order. Therefore, the following value matrix is given:*

$$D = \begin{pmatrix} 6 & 5 & 3 & 4 \\ 4 & 2 & 8 & 3 \\ 2 & 3 & 2 & 1 \\ 3 & 1 & 1 & 5 \end{pmatrix}$$

An entry d_{ij} *means that if the number* i *is the* j*-th element in the order it achieves the value* d_{ij}*.*

Consider now two search states $A = (1, 2)$ *and* $B = (2, 1)$*. Both leave the search in the same state: the numbers* 3 *and* 4 *can only take the third and fourth place in the order. The partial objective values for the search states are* $f(A) = 8$ *and* $f(B) = 9$*. Looking at the remaining subproblem the partial solution* $(., ., 3, 4)$ *achieves the best value with* 7*. Therefore, extending* A *and* B *to full solutions can best be achieved by this partial solution. The search state* B *had the best partial objective value and will have it after the extension since the same value is added in both cases. Therefore, the search state* B *dominates the search state* A*. When the domination is discovered the dominated assignment can be abandoned.*

There are problems where the remaining subproblem is not necessarily identic.

Example 5.18 *Consider the Example 5.17 with the partial assignments A and B. Consider further a constraint that states that the sum of the numbers in the first and fourth position must be uneven.*

Still A and B achieve the same partial values: $f(A) = 8$ and $f(B) = 9$. But the remaining subproblems in A and B are not symmetric. The best assignment in the last example $(2, 1, 3, 4)$ which achieved the best objective value is not feasible anymore. The only full solution for A is $(1, 2, 4, 3)$ with an objective value of 15 and for B is $(2, 1, 3, 4)$ with an objective value of 11. Although A is dominated by B, A can be extended to a solution better than B can be. In this example the search in A could not be abandoned.

If the remaining subproblem is identic for all search states then the domination criteria can be used to prune dominated search states. In this case cardinal solutions do not have to be investigated exhaustively which saves a lot of time. Also the approach can be used for non-cardinal solutions. The remaining subproblem of instantiating $k-n$ SymVars (if the neighbourhood degree of this solution is k to the cardinal solution) can be solved by dominance pruning. Search and solution storing is performed in a different way in this case.

5.7.2 Altering the search

Since domination can only be detected on the same level in the search tree (for a fixed variable ordering) a depth first search would be rather inefficient since many search states on different depths of the search tree would be investigated. It would be preferable to consider search states of the same level consecutively such that domination could be detected early and dominated search states are excluded from search. This can be achieved by performing a breath-first search. All nodes of one level are investigated before moving to the next level. This represents the following search: On each level k all k-elementary subsets of the SymVars with the domain $\{1, \ldots, k\}$ are permuted. Only dominating permutations are stored with their objective value and also only these search states are further pursued in search. All other search states are dominated by these and can be pruned therefore. We will see in Section 5.7.4 how many search states have to be visited in this case. This way the search tree is investigated level by level and only one dominating partial permutation is stored for each subset of SymVars.

Still for each search state the partial permutation is stored if it dominates a previously found partial permutation with the same set of variables and values assigned. But if the actual partial permutation is dominated by the stored partial permutation search backtracks at this time since every extension of this search state is dominated by the same extension of the stored partial permutation.

Still for each subset of values one dominating permutation has to be stored so there is no reduction in the storage capacity but there is a gain in pruning. Whenever domination is detected the search state can be abandoned which saves a lot of work.

5.7.3 Stored Data

The storage capacity in this approach is exactly the same as introduced and proved in Theorem 5.13. For each k-elementary subset one dominating permutation has to be stored.

5.7.4 Counting the Search States to be Visited

In the k-th layer of the search graph $\frac{n}{k}$ dominating permutations are stored and only these search states are further investigated and expanded in the next layer. For the next layer $k+1$ each of the remaining search states is expanded. For each search state there are $n-k$ values to choose for instantiation. All these values have to be checked to determine a dominating permutation. Therefore, in each layer $k+1$ the number of investigated search states is $\binom{n}{k} \cdot (n-k)$. In total there are $\sum_{k=0}^{n-1} \binom{n}{k} \cdot (n-k)$ search states that have to be investigated.

Example 5.19 *Consider the problem from Example 5.17. There are four numbers to permute. We will have a closer look at the second layer of the search tree. In this for each two-elementary subsets of* 1 *to* 4 *one dominating permutation has to be stored.*

$\{1,2\}$*:*			$\{1,3\}$*:*			$\{1,4\}$*:*		
$1,2$	$=$	8	$1,3$	$=$	14	$1,4$	$=$	9
$2,1$	$=$	9	$3,1$	$=$	7	$4,1$	$=$	8
$\{2,3\}$*:*			$\{2,4\}$*:*			$\{3,4\}$*:*		
$2,3$	$=$	13	$2,4$	$=$	8	$3,4$	$=$	6
$3,2$	$=$	5	$4,2$	$=$	6	$4,3$	$=$	12

There are $\binom{4}{2} = 12$ *possible assignments. For each subset there are two. Only one per subset is dominating and this one is stored. Therefore, there are only* $\binom{4}{2} \cdot \frac{1}{2!} = 6$ *remaining search states.*

5.7.5 Disadvantage of the approach

Although using the domination criterion to reduce the overall search effort for P_{sym} it may take longer to find the first solution since the search is arranged in breath-first manner. This is not a problem for exhaustive investigation but in online-optimisation in larger instances it might just take too long to investigate the search tree in this way. Also the approach is only applicable if the for all search states the corresponding remaining subproblems are identic.

5.7.6 Different Propagators

Each of the two approaches introduced before have their advantages and disadvantages depending on the scenario they are applied to. In online-optimisation it is often not possible to investigate a problem exhaustively. Instead of an optimal solution a good solution is sought. Therefore, it is desirable to find a first solution fast. Also some permutation problems do not have the feature that the remaining subproblems for all corresponding search states is identic. In such a scenario the approach introduced in Section 5.3 here would be better since it is in a depth-first-search manner. It would be desirable to write the global constraint in a way such that depending on the scenario (satisfaction, exhaustive search, online-optimisation, etc) the global constraint chooses the best algorithm to propagate.

5.8 Conclusions

We proposed a new algorithm that exploits weak symmetries for separable objectives in a way such that the number of permutations to perform can be reduced for certain solutions. By spending a manageable amount of memory partial permutations are stored for so called cardinal solutions. These data is used to save performing permutations for non-cardinal solutions. We introduced the definitions for separable objectives and the neighbourhood between solutions. Also we stated the theoretical ideas of the algorithm and showed correctness. The algorithm is presented in pseudo-code but could be implemented in many solvers as a global constraint which we do not present here.

Already outlined is a generalisation of the algorithm such that the stored data can be updated from time to time if desired. Up to now there are no experimental results for the approach. So the next step is clearly to test it in a constraint solver environment and determine the outcome of applying this technique. Due to recent tests with weak symmetries we are very confident that this algorithm could considerably reduce the search.

Not investigated yet is the possibility to reduce the search effort for cardinal solutions by using a dominance criterion on the permutations to consider following the idea of Smith [80]. But using the idea we would have to change the way the permutations are investigated. This could mean that the approach may be faster in exhaustive search but might not be suitable for online optimisation as for example the application of the automated manufacturing where a first solution has to be found early. We discuss this in the following.

Chapter 6

Computational Results

In this chapter we investigate some problems with weak symmetries computationally. For each problem we compare two models. One usually called *standard*, is a standard approach to the problem while the other, called *weak* handles the weak symmetry. We will compare the results of these two models in terms of the runtime and in terms of the quality of the found solution (for optimisation). Since weak symmetries are not investigated, there are no benchmark sets available. Other benchmarks that are available are not suitable since they do refer to problems without weak symmetries. Therefore, we chose some problems with weak symmetries and generated sets of random instances. We used ILOG OPL Studio 3.5 [47] on a 800 MHz Laptop with 256 MB RAM. The decision to use OPL was motivated by the claim that our technique is based on modelling and does not need extra code to be implemented. OPL as a modelling language therefore is suits our needs exactly. We investigate three different problems in this chapter. In Section 6.2 the weighted magic square problem is considered. In Section 6.3 a problem from the field of automated manufacturing is considered and in Section 6.4 weighted graph colouring is considered.

6.1 Statistical Investigation Methods

In the problems we compare the same key performance indicators (KPI). Therefore, we state in this section how we investigate the results.

To compare the results of the two models we provided the following comparisons:

- Quality of first solution found
- Time of the first solution found
- Quality of the best solution found
- Time of the best solution found
- Speed-up vs. value increase using Weak
- Overview of the solution process using median and quantiles

The plots we are using always show the result of the weak approach in comparison to the standard approach. All numbers used to show speed-ups etc in tables are percentage numbers that show the performance of the weak approach in comparison to the standard approach. We are not interested in the actual value achieved by objective function but rather in the factor of outperforming. Therefore, if a value is 0 it means that both approaches achieve the same performance. A value of 100 means that the weak approach outperforms the standard approach by 100 %.

Most plots are presented in the following general way: The result of each instance is represented by a circle. The x coordinate of the solution represents the investigated attribute of the solution found using the standard approach. The y coordinate represents the investigated attribute of the solution found using the weak approach. To distinguish the regions where the individual approach outperforms a line is plotted that separate these regions. We will call the region between the y-axis and the bisector the *upper region* and the region between the bisector and the x-axis the *lower region*. The performance difference can be measured by the the minimal distance from the circle to each point of the line (the orthogonal distance from the bisector to the circle). The further away a point is from this line the larger is the performance difference.

In the following we will describe the meaning of each comparison using some plot figures as examples. In the investigated problems not all possible plots are presented. In the Appendix we included the relevant plots such that our conclusion can be followed fully.

6.1.1 Quality of first solution found

This compares the values of the first solution found by each approach. Compare Figure 6.1. The x coordinate represents the value of the standard approach and the y coordinate that of the weak approach. The weak approach outperforms the standard one in the upper region while the standard approach outperforms the weak one in the lower region.

6.1.2 Time of first solution found

This compares the time needed to find the first solution. Compare Figure 6.2. The x coordinate represents the time of the standard approach and the y coordinate that of the weak approach. The weak approach outperforms the standard one in the lower region.

6.1.3 Quality of the best solution found

This compares the values of the best solution found by each approach (in the time interval considered). Compare Figure 6.3. The x coordinate represents the value of the standard approach and the y coordinate that of the weak approach. The weak approach outperforms dito in the upper region.

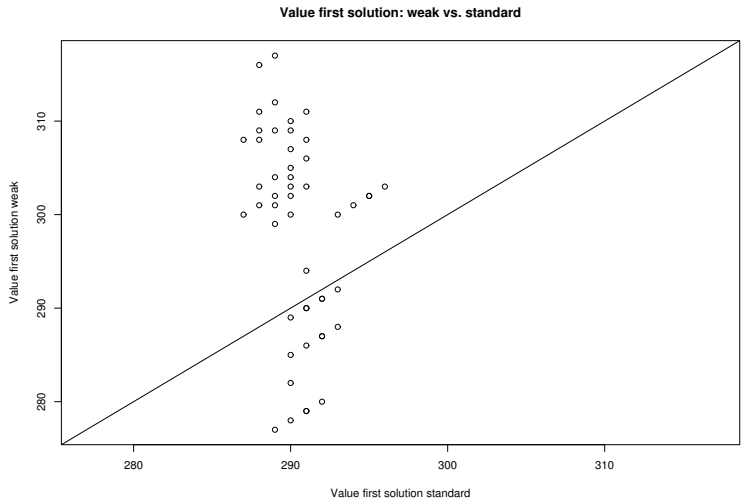


Figure 6.1: Value for first solution plot

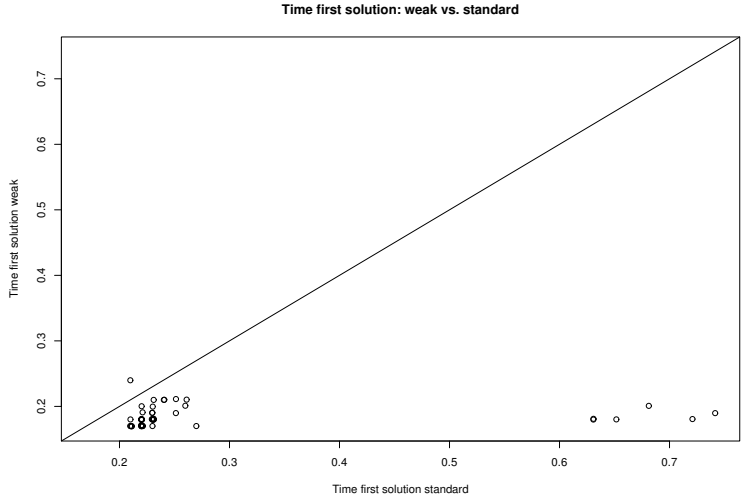


Figure 6.2: Time for first solution plot

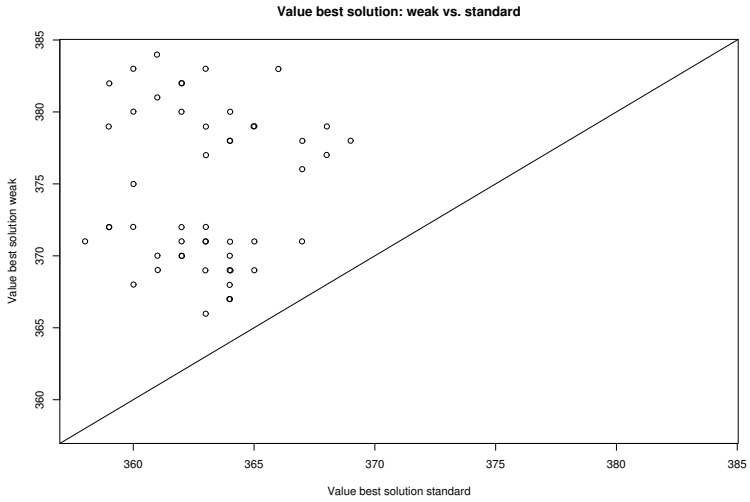


Figure 6.3: Value for best solution

6.1.4 Time of the best solution found

This compares the time needed to find the best solution (in the time interval). The x coordinate represents the time of the standard approach and the y coordinate that of the weak approach. The weak approach outperforms on the lower region.

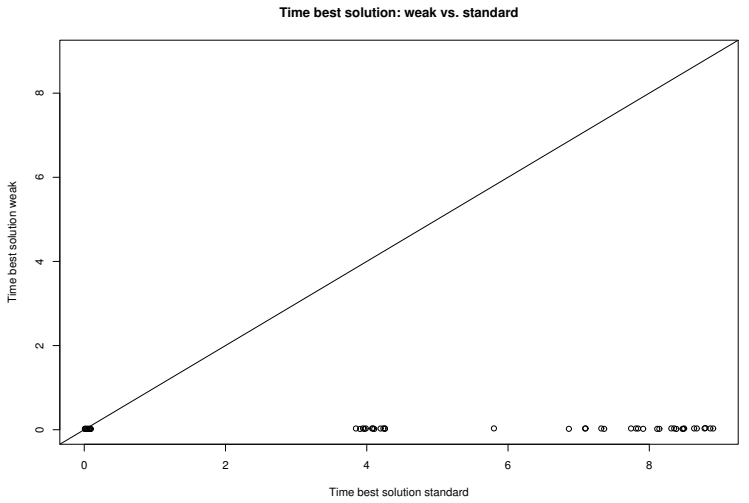


Figure 6.4: Time for best solution

6.1.5 Speed-up in Exhaustive Search

In case of exhaustive search we can derive a speed-up for the time to find the optimum as well as the speed-up for the exhaustive exploration of the search space. Both is presented in a plot that just states the speed-up for each instance. See Figure 6.5

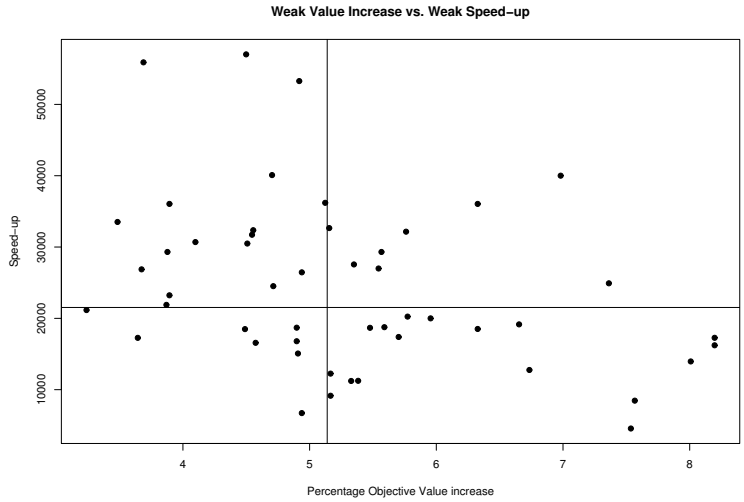


Figure 6.5: 8×12 instance set: Value increase and speed-up for the weak approach

for example. Note that there is no correlation in the x-axis such that the sequence of the points in the plot does not give any information.

6.1.6 Speed-up vs. Value Increase using Weak

This plot compares the speed-up and the value increase relative to the standard approach for the best found solution. The x-axis represents the increase in the objective value using the weak approach instead of the standard one.

The line parallel to the x-axis represents the median of the instance set. The y-axis represents the percentage in speed-up relative to the standard optimum using the weak approach. The line parallel to the y-axis represents the median of the instance set. Since this plot is a bit more complex we give an example in Figure 6.6.

Consider an instance represented by the point $(3, 1000)$. This means that the objective value of weak optimum is 3% higher then that of the standard optimum. This is represented by the x coordinate. Also the weak approach found a solution that was better than the standard optimum in the tenth of the time it took the standard approach. Therefore, the weak approach outperforms by 1000 %. Which is represented by the y coordinate.

6.1.7 Overview of the solution process using median and quantiles

This plot shows the solution process of the whole instance set. For an example see Figure 6.8. The x-axis represents the solving time. The y-axis represents the achieved objective value percentage relative to the optimum found by the standard approach after the maximum processing time allowed. In the standard approach this is clearly limited to 100 while it can be higher for the weak approach correspondingly performance in the opt plot in Figure 6.8 (the standard approach) converges to 100 as time elapses. The lines in the plot indicate the 10, 25, 50, 75 and 90 percent quantiles of the instances, respectively. The region between the 25 and 75 percent

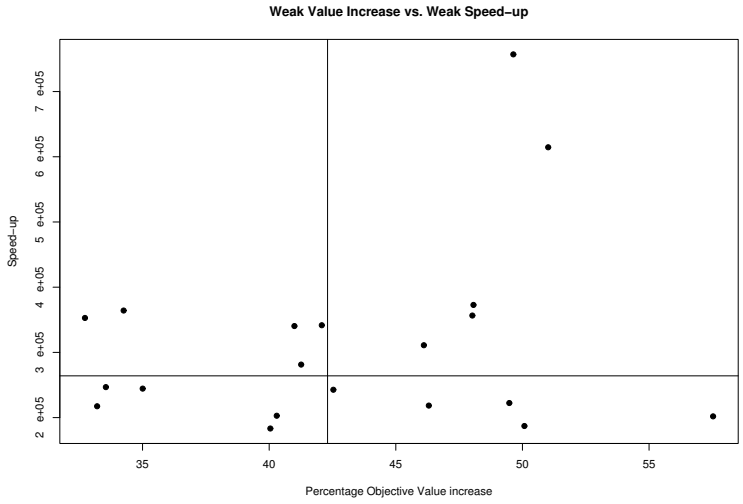


Figure 6.6: Value increase and speed-up for the weak approach

quantile is presented in a darker grey while the regions between the 10 and 25 percent quantile and the 75 and 90 percent quantile is presented in a lighter grey. The line in the darker grey region is the median. There is also a dashed line in the plot which represents the mean of the instance set. In this plot all instances are regarded simultaneously and individual information cannot be factored out. Therefore, we use the quantiles to provide deeper insights in the results.

Consider for example that at the point 100 seconds in the plot the 10 % quantile has a value of 99, the 50 % quantile has a value of 98 and the 90 % quantile has a value of 96. That means that 10 % of the instances reached an objective value that is greater or equals 98, i. e. they have achieved 98 % of the solution value found by the standard approach. 50 % of the instances achieved an objective value of 98 and 90 % of all instances achieved a value of 96.

At a certain time point the distance between the achieved objective value for quantiles can also give information about how large the difference is for the individual instances. If the distance is rather small then all instances have about the same results. If the distance between the 10 % and 90 % quantile is rather large that indicates that there are a few instances that perform very well (about 10 % of the instances) and some that perform rather bad (also about 10 % of the instances). So a smaller difference indicates that all instances can be solved with more or less the same efficiency. So quantiles are very good to investigate whether the solving method has problems in solving some of the instances (which would result in a large difference of the 10 % and 90 % quantile) or is equally effective for most of the instances.

6.2 Weighted Magic Square

The weighted magic square is a good example for investigating weak symmetries. The weak decomposition is straightforward and the weak symmetries are then very obvious.

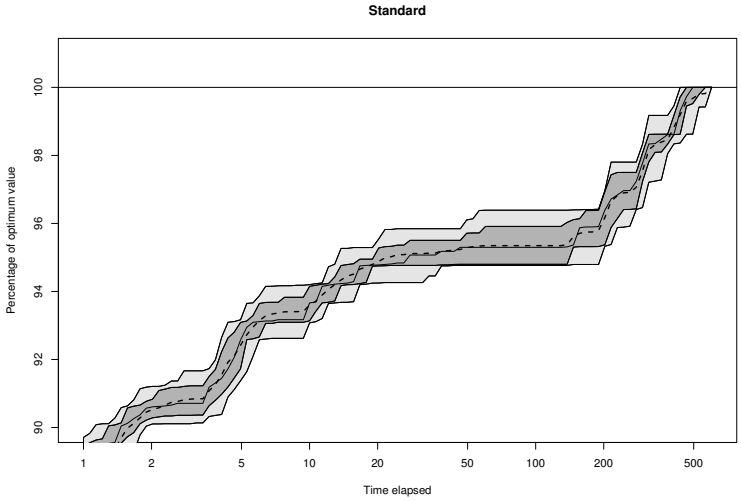


Figure 6.7: Convergence behaviour towards optimality: Standard approach

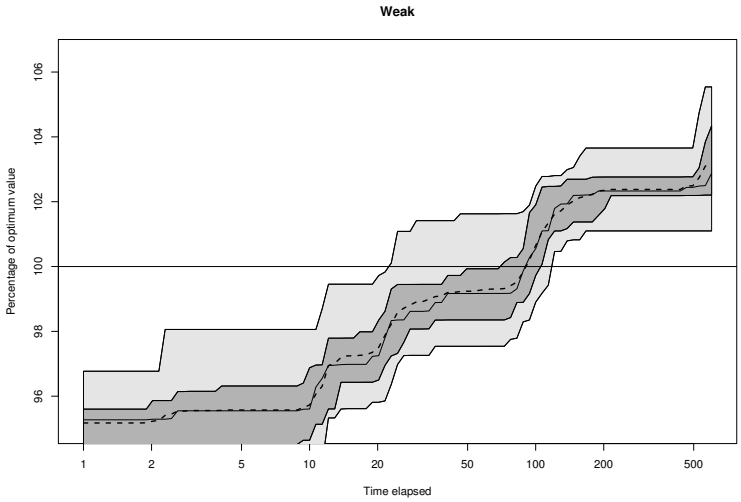


Figure 6.8: Convergence behaviour towards optimality: Weak approach

6.2.1 Problem Definition

Consider the problem definition from Section 4.2.2.4. We introduce weights on the columns (or the individual fields of the square depending on the scenario we investigate). P consists of finding a magic square where the columns and rows (but not the diagonals) sum up to the number m such that the objective value is optimised. The objective function is explained later since it is different for the individual scenarios investigated.

We consider the following weak decomposition $P = (P_1, P_{sym}, P_2)$ of the problem: In P_1 a feasible variable assignment is sought such that each column and row sums up to m. P_{sym} is concerned with permuting the columns and in P_2 the objective value is evaluated.

We investigate the following scenarios:

- Weights on columns (weights are pairwise different)
- Weights on columns (weights are not pairwise different)
- Weights on fields of the square

Each scenario is investigated with the objective to minimise and to maximise the objective value. This is done because of two reasons. First, depending on the distribution of the weights some instance sets may be easier to solve for one objective than for the other. Therefore, if a instance set is hard to optimise for example in maximisation it is easier for minimisation. Since the weak approach does permute the variables it is not affected by this flaw. Even an assignment that does not lead to a good first solution can be improved by reassigning the SymVars. The standard approach has to backtrack a long way until the first solution is changed fundamentally. Second, since the weights are all non-negative the objective value does not decrease with a variable assignment. Pruning on the objective value is easier in minimisation in this case since it can be easily determined, whether the actual objective value already exceeds the best found solution. We want to investigate whether this has an effect on solving the instances.

6.2.1.1 Weights on the Columns (pairwise different)

The objective is to maximise/minimise the sum of products of each column multiplied by the weight of the corresponding column:

$$\min \mathit{or} \max \sum_{col \in \{1,\ldots,n\}} weight[col] \cdot \prod_{row \in \{1,\ldots,n\}} square[row, col]$$

The column permutation is weak while the row permutation is a standard symmetry. The chessboard symmetries which cannot be stated as a combination of row and column permutations are also weak. In fact this is just the flip on the diagonal. This symmetry turns columns into rows and vice versa.

Actually the objective function allows a greedy approach to find the best permutation automatically. The columns must be assigned by decreasing value to the

weights decreasingly such that the column with the largest column product is assigned to the highest weight. This refers to maximisation and the reversed is the case in minimisation. Therefore, it would be sufficient to assign the SymVars using this heuristic. But we consider this scenario not in terms of solving it most efficiently but in terms of comparing the two techniques. Therefore, we do not use this implicit knowledge. Still, using it, we could dramatically speed up the search where a corresponding heuristic for the standard approach cannot be applied as efficiently since it interferes with solving the problem. The column permutation is the weak symmetry group and consists of $n!$ elements.

6.2.1.2 Weights on the Columns (not pairwise different)

The difference to the scenario above is that more symmetries arise: If two columns c_1, c_2 are assigned to corresponding weights w_1, w_2 with $w_1 = w_2$ then the same objective value is achieved when c_1 and c_2 are permuted. This means that the stabiliser for the column permutation is not trivial. Therefore, we can pose ordering constraints such that this permutation is prohibited. In the standard approach a conditional constraint is posted such that the first elements of the corresponding columns are ordered. In the example above the constraint would be $c_{11} < c_{12}$. In the weak symmetry approach the conditional constraint is posted on the corresponding SymVars. In the example above the constraint would be $SymVar_1 < SymVar_2$. In both approaches the conditional constraints are generic. That means identic weights are not known beforehand and no matter which weights are identical the correct constraints are stated. The weak symmetry group consists of $\frac{n!}{k_1! \cdot \ldots \cdot k_j!}$ elements, where j is the number of different values and each k_i marks the multiplicity of each element i in the sequence.

6.2.1.3 Weights on the Cells

The objective is to maximise/minimise the sum of the cells multiplied by the corresponding weight:

$$\min \mathit{or} \max \sum_{col,row \in \{1,\ldots,n\}} weight[row, col] \cdot square[row, col]$$

In this scenario all symmetries are weak. Therefore, no symmetry is broken in the standard scenario. In the weak symmetry approach SymVars are introduced for the columns as well as the rows. Therefore, the weak symmetry group consists of $n! \cdot n!$ elements.

6.2.2 Instance Sets and Generation

For the instances we generate the weights on the columns (on the cells respectively) randomly ranging from zero to nine. We want the weights to be pairwise different (except for the scenario not pairwise different) such that the symmetry group is full for the weak symmetries. This represents the worst case that can happen for weak symmetries.

For example in the case of 4 columns with a total weight of 15 the following weight assignment is feasible: $(2, 3, 4, 6)$. Note that the weights do not have to be ordered. Therefore, also $(4, 3, 2, 6)$ is a feasible weight assignment.

In the scenario of weights on the columns we want the weights to be pairwise different in each row and each column. Therefore, we chose latin squares [45] as weights (i.e. for a square of dimensions $n \times n$ each row and column consists of a permutation of the numbers $1, \ldots, n$).

We investigated several weighted magic squares from size 4 to 7. For each size we generated several sets of instances. Each set has the same total weight number. From each set 50 instances (20 for the larger squares) are solved with a computational time limit of 600 seconds. Each instance set is investigated for maximising and minimising.

We concentrated deliberately on small instances. The reason is that in small examples the weak symmetry group is smaller such that more individual solutions can be investigated for P_1 within the time limit. Also already for squares of size 5 the solution space is so large that even in several hours it cannot be investigated exhaustively (independently from the approach).

6.2.3 The Models

6.2.3.1 Weights on the Columns

We do not distinguish between pairwise different weights and identic weights. The model is the same, only the previously introduced conditional constraint – that orders permutations with the same column weight – is included in the second case.

Standard Model: Weights on the columns turn the column permutation in a weak symmetry which is not broken in the standard model. The row permutation is still a standard symmetry and can be broken in both models regardless. Also we break the symmetry of flipping the square around the diagonal in both models. This symmetry maps the rows to columns and vice versa. In order not to lose solutions we change the objective function such that for each solution also the flipped square is evaluated (see Example 6.1). The objective function therefore is altered to:

$$\begin{aligned}&\text{minimise } \min\{obj_1, obj_2\}\\ &obj_1 = \textstyle\sum_{col\in\{1,\ldots,n\}} weight[col] \cdot \prod_{row\in\{1,\ldots,n\}} square[row, col]\\ &obj_2 = \textstyle\sum_{col\in\{1,\ldots,n\}} weight[col] \cdot \prod_{row\in\{1,\ldots,n\}} square[col, row]\end{aligned}$$

Note: Although this is also a way of breaking weak symmetries we included this in the standard model since the technique of altering the objective function is not what we want to investigate. We focus on the effects of introducing SymVars.

Example 6.1 *Consider the following assignment:*

4	5	14	11
7	16	9	2
10	1	8	15
13	12	3	6

This solution is evaluated by the variable obj_1. When flipped around the diagonal axis the assignment changes to:

4	7	10	13
5	16	1	12
14	9	8	3
11	2	15	6

This solution is evaluated by the variable obj_2.

In the standard model we break the row permutation by ordering the first column. The flip on the diagonal axis is broken by ordering the cells $square_{12} < square_{21}$.

Weak Model: The standard model is extended in terms of symmetry breaking. Also the first row is ordered. This breaks the column permutation which is weak. Therefore, a SymVar $SymCol_i$ for each column i is introduced and an all different constraint is stated on the SymVars. To break the remaining chessboard symmetry of the anti-diagonal flip we order the upper leftmost and the lower rightmost cell of the square. Also we fix the upper leftmost entry to equal 1. This can be done since the row permutation is a standard symmetry and the column permutation is performed via the SymVars. The objective function is adjusted to:

$$\begin{aligned}&\text{minimise } \min\{obj_1, obj_2\}\\ &obj_1 = \textstyle\sum_{col\in\{1,\dots,n\}} weight[col] \cdot \prod_{row\in\{1,\dots,n\}} square[row, SymCol[col]]\\ &obj_2 = \textstyle\sum_{col\in\{1,\dots,n\}} weight[col] \cdot \prod_{row\in\{1,\dots,n\}} square[SymCol[col], row]\end{aligned}$$

6.2.3.2 Weights on the Cells

Standard Model: Weights on the cells turn all symmetries of the problem in weak symmetries. Therefore, no symmetry can be broken in the standard model. But still we apply the same technique as stated above and alter the objective function to:

$$\begin{aligned}&\text{minimise } \min\{obj_1, obj_2\}\\ &obj_1 = \textstyle\sum_{col,row\in\{1,\dots,n\}} weight[row, col] \cdot square[row, col]\\ &obj_2 = \textstyle\sum_{col,row\in\{1,\dots,n\}} weight[row, col] \cdot square[col, row]\end{aligned}$$

Weak Model: Again the standard model is extended in terms of symmetry breaking. In the weak model we introduce a SymVar $SymCol_i$ for each columns i and a SymVar $SymRow_j$ for each row j. On both sets of SymVars an all different constraint is stated. The objective function is adjusted to:

$$\begin{aligned}&\text{minimise } \min\{obj_1, obj_2\}\\ &obj_1 = \textstyle\sum_{col,row\in\{1,\dots,n\}} weight[row, col] \cdot square[SymRow[row], SymCol[col]]\\ &obj_2 = \textstyle\sum_{col,row\in\{1,\dots,n\}} weight[row, col] \cdot square[SymCol[col], SymRow[row]]\end{aligned}$$

6.2.3.3 Search Heuristics

We use the generate heuristic of OPL that assigns the variables column-wise in both models. In the weak model we first generate the magic square and then generate the SymVars. Although there would be other options for variable and value ordering we wanted to keep the model simple such that the results reflect the difference in breaking weak symmetry breaking in comparison to not break them.

6.2.4 Results for Scenario: Weights on Columns (pairwise different)

In the following we will compare and investigate the results with respect to the following cases:

- standard vs. weak: on all instance sets
- min vs. max: on all instance sets
- for each size n of the square individually: Differences in instance sets with different total column weights
- for each size n of the square: changes when moving to higher values for n

In the Tables 6.1 and 6.2 we give an overview of the key performance indicators (KPI) for all the instance sets and scenarios for maximisation and minimisation, respectively. The tables always show the results of the weak approach in comparison with the standard approach. Therefore, no absolute numbers for the objective value for example are given but the percentage of outperforming. That means if for example an entry at a position speed-up is 20, then the weak approach outperforms the standard approach by 20 %. We always state the min, max and median value for each KPI. Min and max state the best and worst performance. The median state the performance on the whole instance set. This is critical to measure the effectiveness since there is no use if one instance is solved extraordinarily good but all others perform very bad. Also the other way around one instance may be solved very badly but all others perform very well.

We present in the tables the value increase (First Sol Val Impr) and the speed-up (Speed-up First Sol) for the first solution found. Also the value increase for the best found solution (Best Sol Val Impr) in case of non-exhaustive search is stated. Since in exhaustive search both approaches find the same optimum there cannot be a value increase. The speed-up for finding a solution with at least the same quality of the best solution that the standard approach (Speed-up Best Sol) is also given. In case of exhaustive search this gives us the speed-up for finding the optimum. Last, in the case of exhaustive search we state the speed-up of the whole search process in comparison with the time the standard approach takes for the investigation (Speed-up Exh).

These results are interpreted also in the following by the above stated cases.

6.2.4.1 Standard vs. Weak

Quality of First Solution: In all instance sets the value for the first solution was the same. No approach could outperform the other.

Time for First Solution:

Mostly the first solution is found faster by the standard approach. Still there are some single instances were the weak approach found the first solution earlier than the standard approach.

It is possible that it takes the weak approach longer due to the symmetry breaking constraints. Since there are more constraints to check evaluating is more costly in

	4 Columns				5 Columns				6 Columns	
Total weight	7	8	15	60	12	15	20	60	20	30
Min First Sol Val Impr	0	0	0	0	0	0	0	0	0	0
Max First Sol Val Impr	0	0	0	0	0	0	0	0	0	0
Mean First Sol Val Impr	0	0	0	0	0	0	0	0	0	0
Median First Sol Val Impr	0	0	0	0	0	0	0	0	0	0
Min Speed-Up First Sol	-100	- 100	- 88	-77	- 50	- 50	- 50	-89	- 74	-71
Max Speed-Up First Sol	10	100	300	50	700	735	701	-3	-71	-69
Mean Speed-Up First Sol	-22	- 11	- 6	-17	32	37	30	- 35	-73	-70
Median Speed-Up First Sol	0	0	-25	-2	- 33	- 33	- 33	-38	-73	-70
Min Best Sol Val Impr	–	–	–	–	7	-0.5	0.6	-0.7	-3	15
Max Best Sol Val Impr	–	–	–	–	102	48	23	2	30	26
Mean Best Sol Val Impr	–	–	–	–	37	14	9	0.2	11	20
Median Best Sol Val Impr	–	–	–	–	20	9	5	0.1	11	20
Min Speed-Up Best Sol	- 50	- 50	- 33	-40	8725	-100	-61	-100	-100	4839
Max Speed-Up Best Sol	44510	42310	42160	17164	845716	972766	895788	346	15649	15859
Mean Speed-Up Best Sol	23715	19307	15105	4124	516389	367308	238415	24	7055	11084
Median Speed-Up Best Sol	24776	20298	13553	200	626329	374979	629	-28	5005	14288
Min Speed-Up Exh	336	362	357	391	–	–	–	–	–	–
Max Speed-Up Exh	449	449	420	418	–	–	–	–	–	–
Mean Speed-Up Exh	401	398	395	409	–	–	–	–	–	–
Median Speed-Up Exh	398	391	403	410	–	–	–	–	–	–

Table 6.1: Results for the objective maximisation in percentage relative to standard approach

	4 Columns				5 Columns				6 Columns	
Total weight	7	8	15	60	12	15	20	60	20	30
Min First Sol Val Impr	0	0	0	0	0	0	0	0	0	0
Max First Sol Val Impr	0	0	0	0	0	0	0	0	0	0
Mean First Sol Val Impr	0	0	0	0	0	0	0	0	0	0
Median First Sol Val Impr	0	0	0	0	0	0	0	0	0	0
Min Speed-Up First Sol	-100	-100	-75	- 52	-50	-42	-43	-89	- 75	- 71
Max Speed-Up First Sol	0	300	100	50	-31	- 33	-16	-16	- 60	-58
Mean Speed-Up First Sol	-20	-12	- 17	-10	-36	-36	-34	-41	-71	-68
Median Speed-Up First Sol	0	0	-4	0	-33	- 33	- 33	-42	-73	-70
Min Best Sol Val Impr	–	–	–	–	0	27	32	42	40.1	60
Max Best Sol Val Impr	–	–	–	–	42	58	57	46	60.5	68
Mean Best Sol Val Impr	–	–	–	–	28	45	43	44	51	65
Median Best Sol Val Impr	–	–	–	–	33	47	42	44	52.0	64
Min Speed-Up Best Sol	-26	-35	-41	207	-77	99026	183277	28603	5475	5760
Max Speed-Up Best Sol	4603	4965	3550	4406	982696	953321	757021	353294	9359	8745
Mean Speed-Up Best Sol	1085	1225	680	2086	498428	355031	312976	263445	7993	7615
Median Speed-Up Best Sol	655	845	432	1742	535322	350640	312976	272901	8229	8331
Min Speed-Up Exh	248	206	248	413	–	–	–	–	–	–
Max Speed-Up Exh	523	536	1112	431	–	–	–	–	–	–
Mean Speed-Up Exh	376	369	412	423	–	–	–	–	–	–
Median Speed-Up Exh	375	376	410	424	–	–	–	–	–	–

Table 6.2: Results for the objective minimisation in percentage relative to standard approach

the weak approach. Looking at the choice points and number fails investigated they differ by less than 10. Therefore, it seems that the symmetry breaking constraints are very costly to investigate.

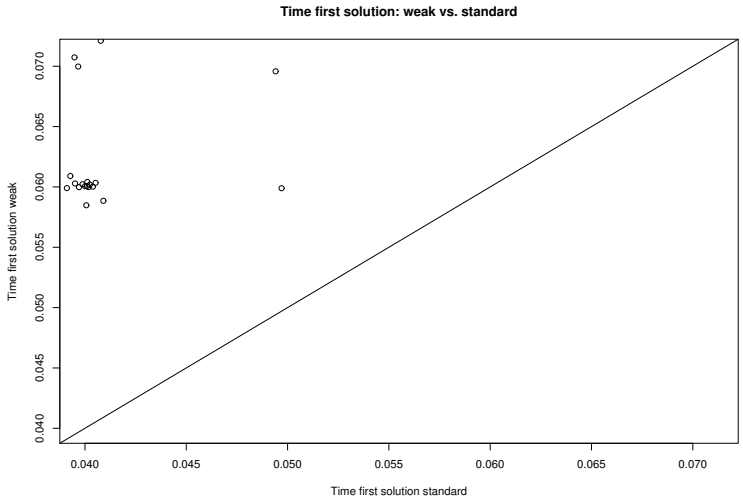


Figure 6.9: Time for first solution in minimisation on the instance set of 5 columns with weight 20

Quality of Best Solution:

In all instance set (that were not investigated exhaustively) the median quality of the solution could be improved. Medians reaching from 0.1 % up to 20 % in maximisation and 33 % to 64 % in minimisation. See Figure 6.10 for example. Still there are some instances where the standard approach found the best solution. But these are just some single instances in the sum of all instances. The worst result is a worsening of 3 % for a single instance. The others have a worsening less than 1 %.

The question why the standard approach can outperform the weak approach on the best solution is straightforward. One would guess that a solution found by the standard approach should also be found by the weak approach since the ordering heuristics are basically identic. Although this is true there are two other facts that influence the search and may take place in these cases:

1. The symmetry breaking constraints may forbid this particular assignment in P_1: The solution found by the standard approach is not feasible in P_1 since the columns are not ordered. The weak approach would find this solution eventually but only by assigning the SymVars. Still, the representant of this particular solution is not found within the given time limit.

2. The weak approach has to investigate the weak symmetry group for each solution found in P_1. Therefore, it is possible that the solution found by the standard approach would also be found by P_1 but was just not within the given time limit.

Time for Best Solution:

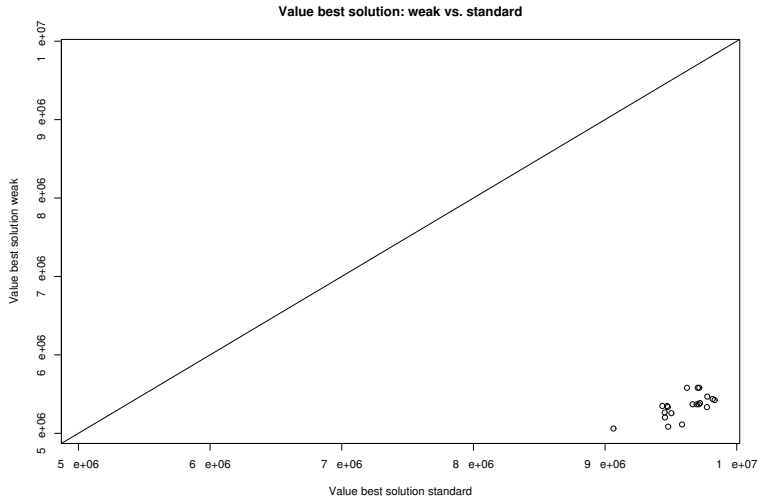


Figure 6.10: Value for the best solution in minimisation on the instance set 5 columns with weight 60

Except for one instance set the weak approach achieves a speed-up in median. There are some single instances where the best solution is found by the standard approach. In the instance set where standard outperforms the best speed-up for the standard approach is 28 % in median. In the other instance sets a speed-up of 2 times up to over 6000 times was achieved. See Figure 6.11 for example. The speed-up refers to the time it took the weak approach to find a solution with at least the same quality as the best found solution of the standard approach. Although investigating a choice point in the search tree is more costly for the weak approach (as stated in the investigation for the time for the first solution) weak outperforms the standard approach since in the following search the weak approach investigated fewer choice points (due to symmetry breaking) and the investigation of P_{sym} delivers $n!$ solutions with the potential to have a better objective value than the best already found. In the solution, where the weak approach could not outperform the standard approach the reverse is the case. Looking at the different solutions the weak and standard approach find, it can be observed that the standard approach backtracks further in the search tree. Therefore, finding different solutions which have more potential and in the end achieve a good objective value. The weak approach finds also exactly these solutions but very late since for each feasible solution of P_1 P_{sym} has to be investigated. Therefore, weak finds the solutions with a better objective value late in the search without finding the same solution found by the standard approach within the time limit.

Time for Exhaustive Search:

Only the instances of size 4 could be solved exhaustively within the given time limit. In all instances the search time was improved such that the weak approach solved every single instance faster than the standard approach. The weak approach was 3 to 4 times faster in exploring the search space in median. See figure 6.12 for example. This shows basically how weak symmetry breaking shrinks the search space which cannot be observed on the other instance sets of larger size since the time limit is much too short to investigate the whole search space. And although the exploration

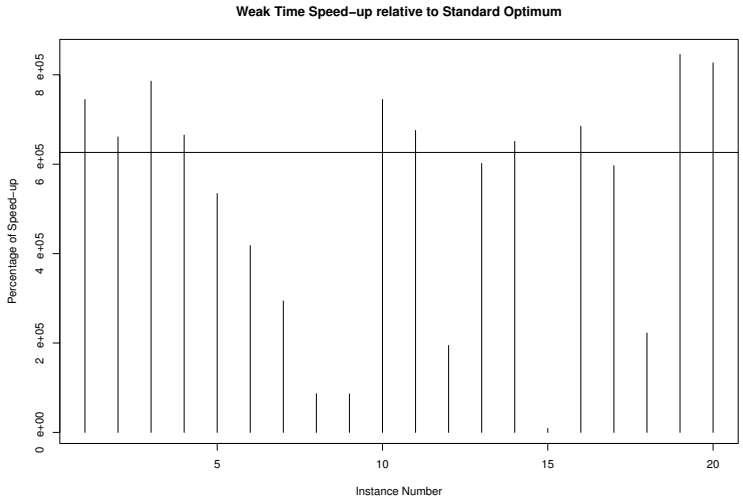


Figure 6.11: Speed-Up for finding the best solution in maximisation on the instance set 5 columns with weight 12

in each choice point in the search tree is more expensive this is outperformed by the total saving in choice points to investigate.

6.2.4.2 Min vs. Max

The weak approach achieves far better outperforming results on the minimisation scenario. This is not because the approach handles minimisation considerable well. The main reason is that the standard approach handles this scenario worse than optimisation for the investigated instance sets. In the instance sets where there is a big difference in the speed-up between minimisation and maximisation the standard approach achieved different performances. In minimisation the standard approach finds solutions late and with a poorer quality. Since the solutions of both approaches are compared weak compares with bad results and achieves an extremely high performance. Considering the Figures 6.13 and 6.14 it can be seen that the weak approach finds a solution that is better than the best of standard within seconds. The same happens in the minimisation scenario just that the found objective value is smaller than the best found by the standard approach.

6.2.4.3 Different Total Column Weights

For maximisation it looks as if the performance of the weak approach decreases with increasing total column weight. But this cannot be observed in the minimisation scenario. Again, in the instance sets where the speed-up and best solution improvement are worse than the other instance sets the standard approach finds good solutions early in the search. The speed-up is rather small since the weak approach finds solutions with just a marginal better objective value.

It may be that the standard approach can use pruning better if the total weight is increased. Compare Figures 6.15 and 6.16 to see the difference in the quality of the solutions which explains the performance differences.

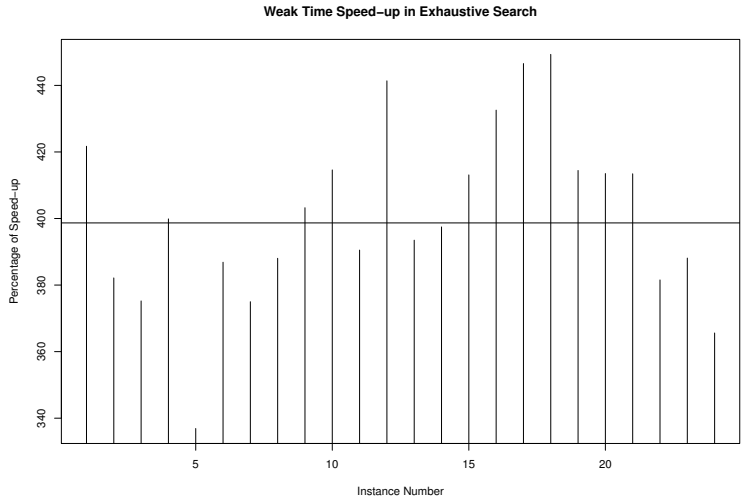


Figure 6.12: Time for the exhaustive exploration of the search space in maximisation on the instance set 4 columns with weight 7

6.2.4.4 Increasing Dimension

With increasing dimensions symmetry breaking can reduce the search space further such that it can be expected that the weak approach performs better with higher dimensions on the exhaustive search. This cannot be observed in the small amount of time we perform search. For improving the best solution or the speed-up is it also likely that the weak approach shows better results. In the instance sets investigated the speed-up decreases but is still 50 to 140 times faster in median than the standard approach for the largest dimension tried of the magic square.

6.2.4.5 Relation between Speed-up and Value Increase

Interesting is also whether an instance with value increase does achieves also a high speed-up or whether the one is bought for the other. The results show that there seems to be no relation between these two categories. Sometimes a solution with a high speed-up also has a high value increase. As Figures 6.17 and 6.18 for example show that the points representing the performance are distributed in the plot. Therefore, a high performance in both categories can be achieved or the contrary or anything between. Still, as pointed out earlier, if the weak approach finds solutions with a better much better objective value than the solutions found by the standard approach the chance is high that the speed-up is also high.

6.2.5 Results for Scenario: Weights on Columns (Not Pairwise Different)

In this scenario we want to investigate whether weak can outperform the standard approach for identic weights. Identic weights reduce the number of permutations to consider for P_{sym}. The conditional symmetry breaking constraints for identic weights are included in both approaches. Therefore, the comparison aims at investigating

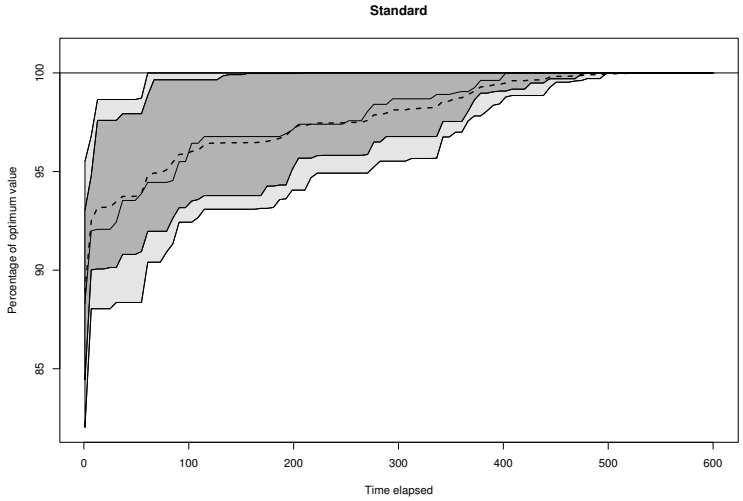


Figure 6.13: Search towards optimality in maximisation on the instance set 5 columns with weight 12

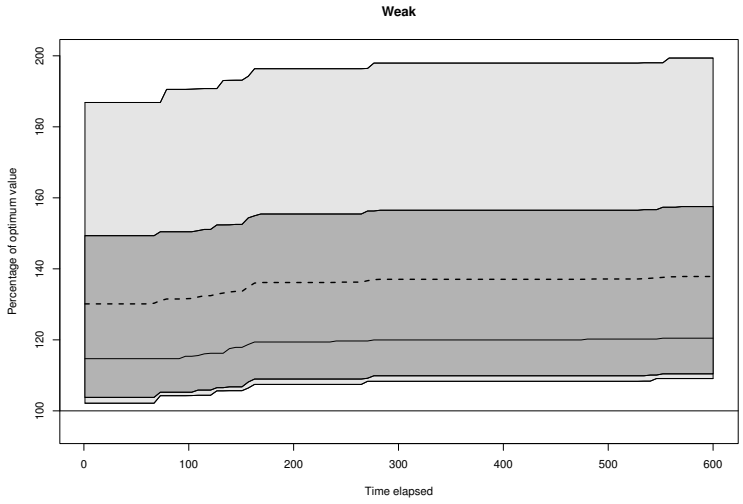


Figure 6.14: Search towards optimality in maximisation on the instance set 5 columns with weight 12

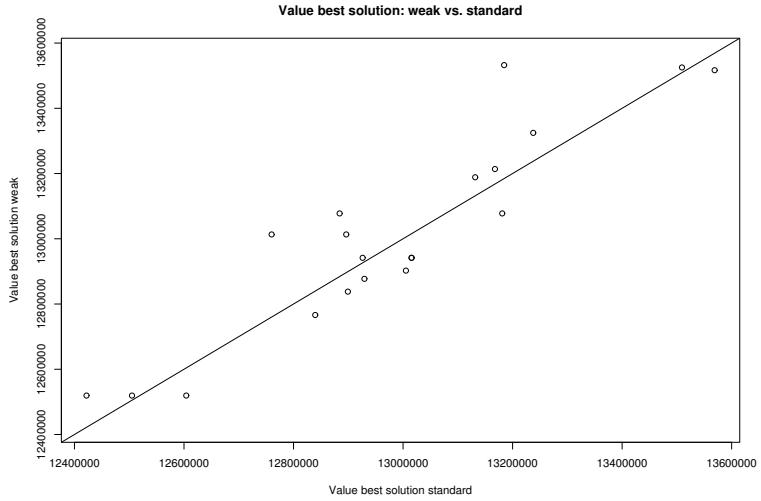


Figure 6.15: Comparing the best solutions in maximisation on the instance set 5 columns with weight 60

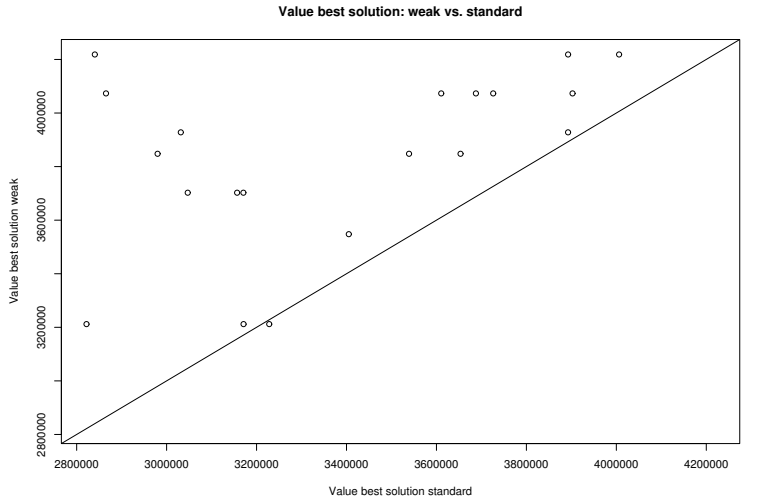


Figure 6.16: Comparing the best solutions in maximisation on the instance set 5 columns with weight 15

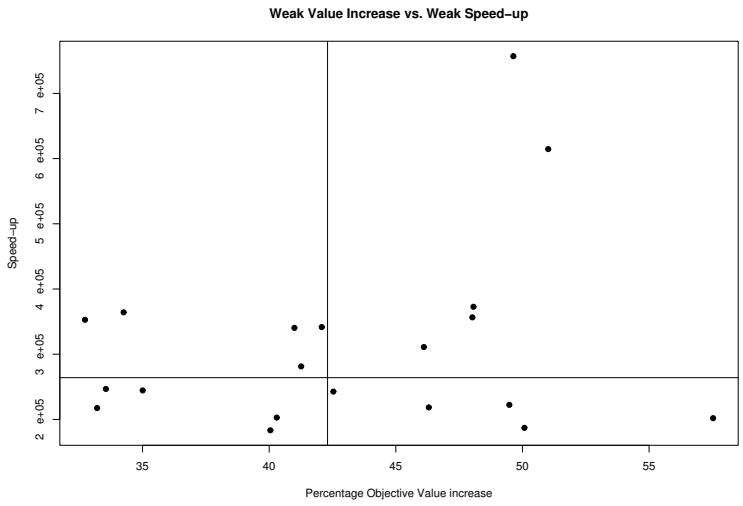


Figure 6.17: The value improvement and speed-up for the instance set 5 columns with weight 20 in minimisation

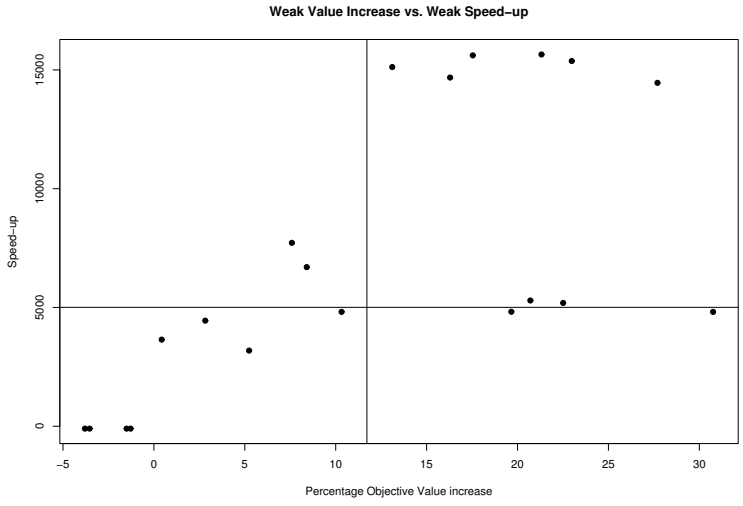


Figure 6.18: The value improvement and speed-up for the instance set 6 columns with weight 20 in maximisation

whether the weak approach profits more than the standard approach. The same KPIs are represented as in the Table 6.2. We did not try different total weights. So the total weight in the two instance sets is different for each instance. The results are presented in Table 6.3.

	4 Columns		5 Columns	
Total weight	Min	Max	Min	Max
Min First Sol Val Impr	-67	-41	-10	- 46
Max First Sol Val Impr	21	7	32	13
Mean First Sol Val Impr	0	16	7	- 9
Median First Sol Val Impr	0	14	0	0
Min Speed-Up First Sol	-100	0.0	-60	-60
Max Speed-Up First Sol	0	0.0	50	46
Mean Speed-Up First Sol	-21	0.0	-40	-36
Median Speed-Up First Sol	0	0.0	-50	-50
Min Best Sol Val Impr	–	–	21	0
Max Best Sol Val Impr	–	–	51	13
Mean Best Sol Val Impr	–	–	23	4
Median Best Sol Val Impr	–	–	21	3
Min Speed-Up Best Val	-62	-50	-59	- 94
Max Speed-Up Best Val	31595	160840	1227115	1109345
Mean Speed-Up Best Val	6431	12387	522124	179341
Median Speed-Up Best Val	3694	275	537693	1122
Min Speed-Up Exh	3465	3405	–	–
Max Speed-Up Exh	10768	10077	–	–
Mean Speed-Up Exh	6584	6461	–	–
Median Speed-Up Exh	6403	6521	–	–

Table 6.3: Results for not pairwise different weights in percentage

Indeed the weak approach performs very good proving optimality within one second. The search time for the standard approach is even higher than the solving times for the instances without symmetry breaking. It seems that the symmetry breaking constraints do interfere with the search such that the total search time is not reduced but increased. In instance sets of size 5 the best solution could be improved between 3 and 21 % which is not very different from the results of the non-identic weights scenario. The same holds for the speed-up of about 11 and 5300 times in median. Still there is a large difference in minimising and maximising. In minimisation the weak approach achieves the better results in each category. The same could be observed in the non-identic weights scenario for this instance set size. The reason is again, that the standard approach finds better solutions faster in the maximisation scenario. In minimisation it is the other way around such that the speed-up is rather high. See for example Figure 6.19 and Figure 6.20.

6.2.6 Results for Scenario: Weights on Cells

In this scenario all symmetries are weak such that no symmetry breaking is performed in the standard approach. Also in the weak approach we have two kinds of SymVar one for the columns and one for the rows. Permuting both rows and columns means to invest a lot of time to investigate the weak symmetry group. On the other hand the potential of finding good objective values of the weak symme-

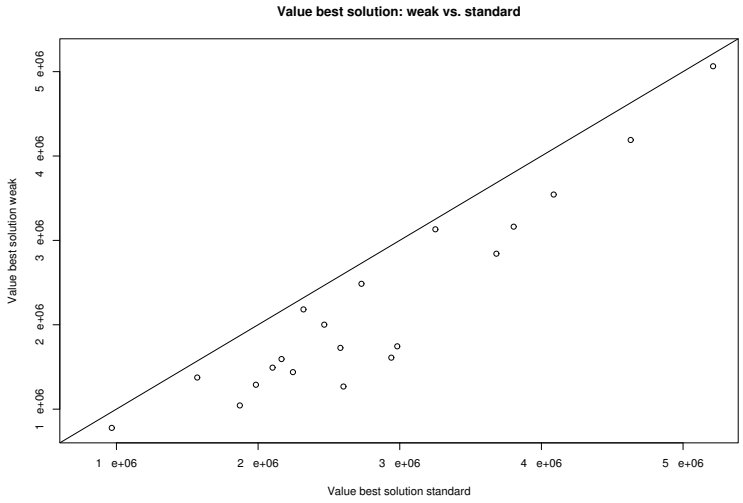


Figure 6.19: Comparison of the best solutions of both approaches in the instance set 5 Columns in minimisation

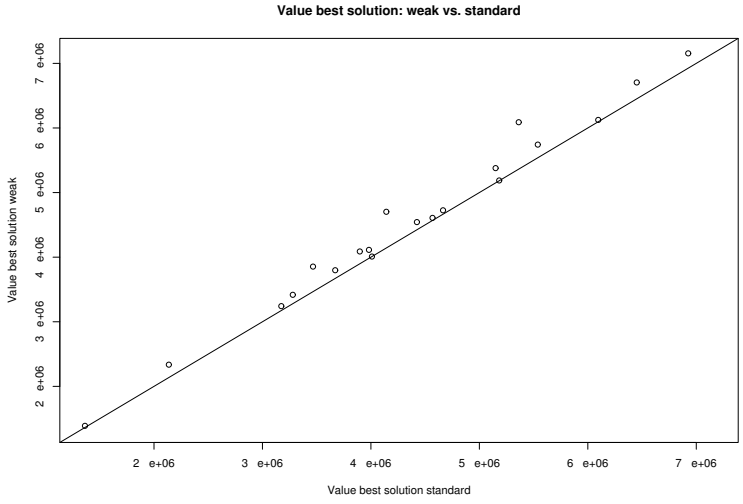


Figure 6.20: Comparison of the best solutions of both approaches in the instance set 5 Columns in maximisation

try group (i.e. the range of objective values achieved by the solutions) is very high. Still the advantage in using SymVars is that with each instantiation several variables move their place in the matrix and can achieve completely different objective values. In standard an instantiation of one variable just changes one value in the sum. In this scenario we are again interested whether the weak approach can outperform the standard approach. A drawback for the weak approach is that the symmetry breaking constraints inflict with the search. Since the standard approach do not have any symmetry breaking constraints it will find especially the first solution more likely faster. The question is whether his has an effect on the performance of the weak approach in comparison with the standard approach. In Table 6.4 we present the results for this scenario.

	4 Columns		5 Columns		6 Columns		7 Columns	
Total weight	Min	Max	Min	Max	Min	Max	Min	Max
Min First Sol Val Impr	-2	1	0.04	-0.18	- 0.1	-0.2	0.0	0.0
Max First Sol Val Impr	-0.7	1	0.26	0.04	0.1	0.05	0.0	0.0
Mean First Sol Val Impr	-1	1	0.1	-0.07	0.01	- 0.1	0.0	0.0
Median First Sol Val Impr	- 1	1	0.1	-0.09	0.02	-0.1	0.0	0.0
Min Speed-Up First Sol	- 97	-90	-70	-25	99	162	-39	- 37
Max Speed-Up First Sol	2055	2055	1370	1403	230	234	-36	-32
Mean Speed-Up First Sol	99	57	53	55	186	189	-36	-36
Median Speed-Up First Sol	0	-19	0	0	188	190	-36	-36
Min Best Sol Val Impr	–	–	0.02	8	2	2	2	0.6
Max Best Sol Val Impr	–	–	0.33	15	2	3	1	1
Mean Best Sol Val Impr	–	–	0.03	11	2	3	1	1
Median Best Sol Val Impr	–	–	0.02	11	2	3	1	1
Min Speed-Up Best Sol	272	-76	-37	874136	23265	50827	5542	- 38
Max Speed-Up Best Sol	195815	16423	1260680	1499680	34609	55105	5977	3519
Mean Speed-Up Best Sol	19680	1657	385069	1390256	31864	54392	5812	1663
Median Speed-Up Best Sol	875	112	1044	1488403	33936	54488	5833	2377
Min Speed-Up Exh	18	15	–	–	–	–	–	–
Max Speed-Up Exh	47	50	–	–	–	–	–	–
Mean Speed-Up Exh	40	34	–	–	–	–	–	–
Median Speed-Up Exh	42	36	–	–	–	–	–	–

Table 6.4: Results for weights on cells in percentage

The first solution is found not very late in comparison with the standard approach. In the instance set of size 6 the weak approach even achieves a positive speed-up for finding the first solution. In this instance set the symmetry breaking seems to ease the search while it does not help in the other instance sets. Small improvements on the quality of the best found solution (between 0 and 11 %) could be achieved. See for example Figure 6.21 and Figure 6.22. This is less than in the scenario where weights are only introduced on columns. The speed-up in finding a solution with at least the same quality as the best solution of the standard approach is larger than in the other scenarios but seems to decrease with increasing size of the instance sets. In the exhaustive search only speed-ups of about 30 % could be achieved. This is considerably less than in all the other scenarios. A reason is that the weak approach has to investigate too many assignments in P_{sym} before pruning is possible. Since the objective value can only be determined (even only partially) when one kind of SymVar is fully instantiated and the second is about to be instantiated, since the objective function is stated on both kinds of SymVars. If for example the SymVars for the columns are instantiated first, all $n!$ permutations have to be performed without the chance of pruning. For each of these column permutations the row permutations are checked and only here pruning can be done. It seems that this causes many search states to consider that would not be necessary. But this cannot be decided before actually visiting this search state. Also, not many different solutions of P_1 can be investigated in the time interval since a lot of time is spent in P_{sym}.

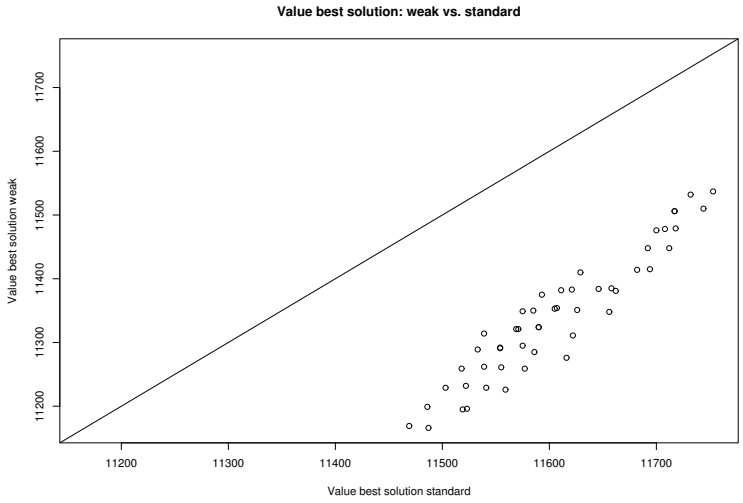


Figure 6.21: Comparison of the best solutions of both approaches in the instance set 6 Columns in minimisation

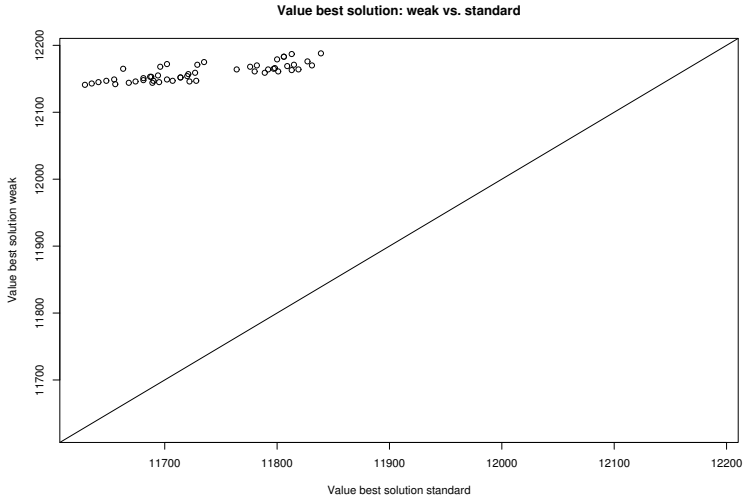


Figure 6.22: Comparison of the best solutions of both approaches in the instance set 6 Columns in maximisation

6.3 Automated Manufacturing

6.3.1 Problem Description

Consider the problem of Section 4.2.3.5. The machine is modelled by a matrix $A^{m \times n}$, where m represents the number of component types that can be assigned to a setup and n represents the number of setups on the machine. The objective is to maximise the overall placeability of the machine setup:

$$maximise \sum_{r \in \{1,\ldots,m\}} \sum_{c \in \{1,\ldots,n\}} placeability[c, A[r, c]]$$

6.3.2 Instance Sets and Generation

We generated several instance sets with different machine dimensions.
For following data is fixed for each instance set:

- the number of columns and rows of the matrix (the number of workstations and component types per workstation, respectively)
- the number of different component types
- the compatibility class for each component type (component types can only be assigned to the same column if they have the same compatibility class)
- the feasibility matrix (for each tuple ⟨component type × column⟩ how often this component type can be mounted when assigned to this column). The value range is from zero to ten and randomly determined.

The number of each component type available is randomly generated. Each instance set consists of 50 instances constructed as explained above.

6.3.3 Symmetry Breaking in the Models

Standard Model: In the standard approach the columns are ordered to break the row permutation which is a standard symmetry.
Weak Model: The standard model is extended in terms of symmetry breaking. The column permutation is also broken by Crawford constraints [13]. A SymVar $SymCol$ for each column is introduced and an all different constraint is stated on the set of SymVars. The objective is altered to:

$$\max \sum_{r \in \{1,\ldots,m\}} \sum_{c \in \{1,\ldots,n\}} placeability[c, A[r, SymCol[c]]]$$

6.3.4 Ordering Heuristics

In this scenario we used a variable and value ordering for the models. We tried several heuristics and chose the best for each individual model. That means that the heuristics are different but it would be not realistic to agree on one heuristic that has obvious disadvantages for one of the models. Therefore, it is a best-to-best comparison.

6.3.4.1 Ordering Heuristic for the Standard Model

We start with generating the first row of the matrix. This way the compatibility classes are implicitly assigned to the columns and after assigning the first row this constraint is satisfied for the rest of the search. If the matrix were generated column-wise without the first row already assigned then values could be assigned that conflicts with the compatibility classes which is not detected and causes a lot of backtracking.

The rest of the matrix is instantiated row-wise where for each cell a value is chosen that achieves the best placeability value. This way the setups are optimised during completion.

6.3.4.2 Ordering Heuristic for the Weak Model

The whole matrix is generated column-wise with no other modifications. This is due to the fact that we are just searching for a feasible matrix assignment. We cannot optimise the assignment of the individual setups at this time as in the standard model since we do not know yet which values the assignments achieve. When assigning the SymVars we chose for each SymVar the value such that the represented setup achieves the highest placeability. Therefore, we greedily assign the SymVars in the hope that the first solution achieves a good objective and pruning in P_{sym} can take place.

6.3.5 Results

We chose a time limit of 600 seconds and compare the quality of the solutions found in this time interval. All instance sets include 50 instances randomly constructed by the above introduced method.

None of the instances can be solved exhaustively in this scenario. Therefore, the best found solution is called the optimum (of standard or weak) in the following. We visibility reasons we perturbed the scatter plots such that identical solutions are not completely covered by each other. The variance is minimal such that the results are not really influenced. Identic solutions may still overlap but do not completely cover each other.

For the first solution we investigate the time and the quality of the solution. For the best solution we investigate just the quality of the solution. It is not possible to compare the times since both solutions are not optimal and have different values. Further, we investigate the solving process using quantile plots. Also the speed-up and the value increase is stated.

We will closely investigate the instance set 6×10. The other instance sets (6×12, 8×12 and 8×20) are then compared to the results of the first instance set. This way the latter three instance sets are investigated only qualitative with the goal to work out a trend for larger instances. The KPI for all the instance sets can be found in Table 6.5.

In this part we do not include all plots but only some that show interesting criteria. All plots for the instance sets can be found in the Appendix.

	6×10	6×12	8×12	8×20
Min First Sol Val Impr	-24	-4	-3	-10
Max First Sol Val Impr	1	9	7	0
Mean First Sol Val Impr	-14	2	1	-4
Median First Sol Val Impr	-15	3	1	-5
Min Speed-Up First Sol	-8	-12	-42	-34
Max Speed-Up First Sol	270672	298	264	299
Mean First Sol Val Impr	20136	52	35	56
Median Speed-Up First Sol	57	25	18	38
Min Best Sol Val Impr	-0.8	0.8	3	1
Max Best Sol Val Impr	15	6	8	3
Mean First Sol Val Impr	7	3	5	2
Median Best Sol Val Impr	6	2	5	2
Min Speed-Up Best Sol	-100	220	4554	-63
Max Speed-Up Best Sol	60041	4576	57014	21419
Mean First Sol Val Impr	6159	859	24396	2826
Median Speed-Up Best Sol	2371	459	21527	714

Table 6.5: Results for automated manufacturing instance sets in percentage

6.3.5.1 Instances of Size 6×10

First solution quality:

The standard approach found in all but one instances a solution that was better than that of the weak approach. Only a few were better for the standard approach only slightly. See Figure 6.23 for details. Since the search heuristic of the standard approach is more designed to find a good solution first this is not surprising. The search heuristic for the weak approach are designed to find a feasible solution. Since the SymVars are not instantiated it is not possible to assign the variables in P_1 in a way such that the first solution has guaranteed a good solution. But this is compensated by assigning the SymVars greedily.

First solution time:

The search time was mostly faster for the weak approach. Only one instance was solved slightly slower by the weak approach. The speed-up for the weak approach is 35.6% in median. See Figure 6.23 for details. Weak symmetry breaking seems to pay-off in this instance set. Although the first solution is worse than that of standard it was mostly found earlier. And by the time the standard approach found its first solution, weak had already found new solutions that mostly outperformed the standard approach.

Best solution quality:

In all but one instance weak outperforms the standard approach by a median of 6.9% and about 15% in the best case. One instance did not find a solution with a better objective value than standard did, but was only 0.8% worse. See Figure 6.25 for details.

Search towards Optimality Analysis:

The 90 % quantile of the weak approach reaches the standard optimum within the first few seconds and also the 90% quantile reaches the 100% level within 50 seconds.

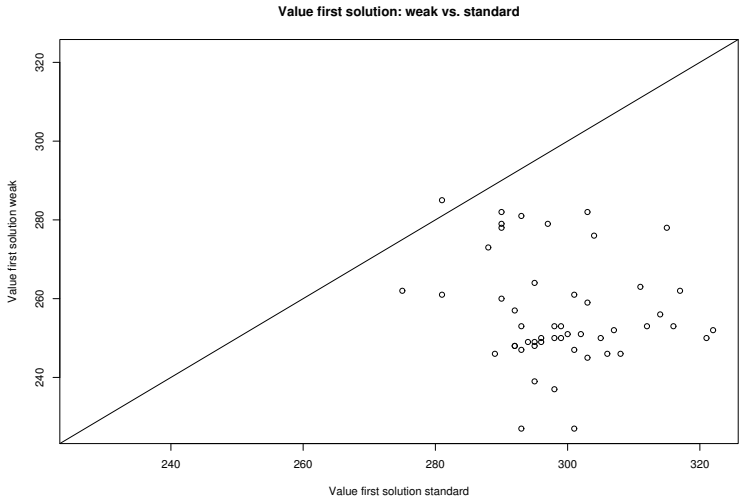


Figure 6.23: 6 × 10 instance set: Quality of first solution

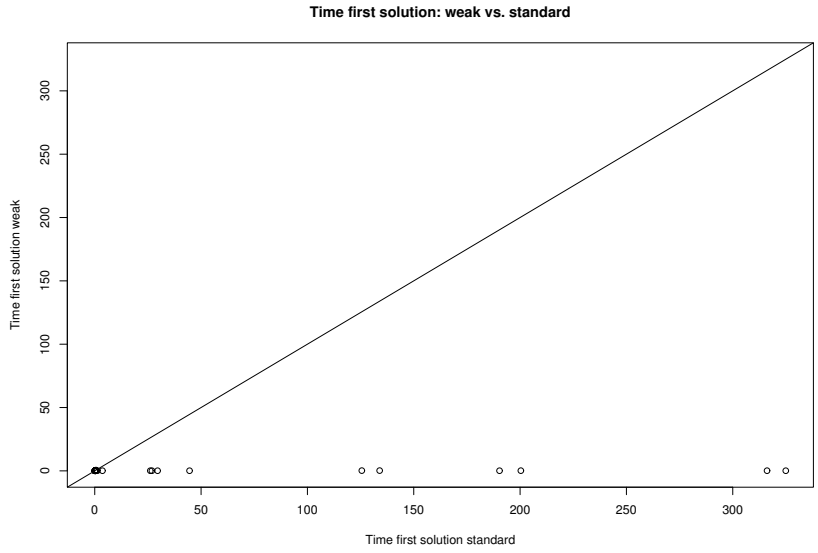


Figure 6.24: 6 × 10 instance set: Time for first solution

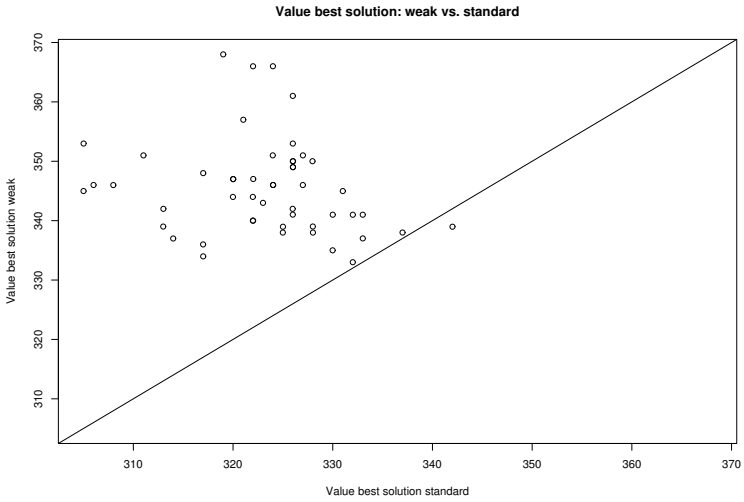


Figure 6.25: 6×10 instance set: Quality of best solution

The 10% quantile reaches the top level of 107% within 20 seconds and 25% within about 75 seconds. Since not all solutions achieve this objective value the quantiles keep their level and do not merge like in the standard plot. See Figure 6.26 and Figure 6.27 for details. Since there are some instances in the set where the standard approach cannot find a solution for a long time the quantiles of this plot reach higher levels rather late in the search. Also the mean (dashed line) lies below the 10% quantile for some time. This indicates that the objective value of some instances were rather bad in comparison with the rest of the instances of the set. The 90% quantile reaches the 96% mark only after about 130 seconds.

Speed-up for the best solution:

Since the solutions that both approach find are different (with different objective values) we cannot compare the time needed to find the best solution. Therefore, we compare the time it took the standard approach to find the best solution with the time it took the weak approach to find a solution that has at least the same or a better objective value. This gives us the speed-up of the weak approach over the standard approach.

For this instance set only one solution did not find a solution of the same quality than the standard approach. Therefore, the minimum speed-up is -100% of this instance set. In all other instances the weak approach outperforms the standard approach clearly. In median the weak approach finds a solution more than 23 times faster. The maximum is at about 600 times for the instances that did not find a solution early in the search. See Figure 6.28 for details. Investigating the total performance gain (i.e. the speed-up and the value increase) for each instance it can be seen, that there is not general trend. There are some solutions with a rather small total performance (no large improvement and also no large speed-up) but also some with a high total performance.

6.3.5.2 Instance sets of the size 6×12, 8×12 and 8×20

First solution time:

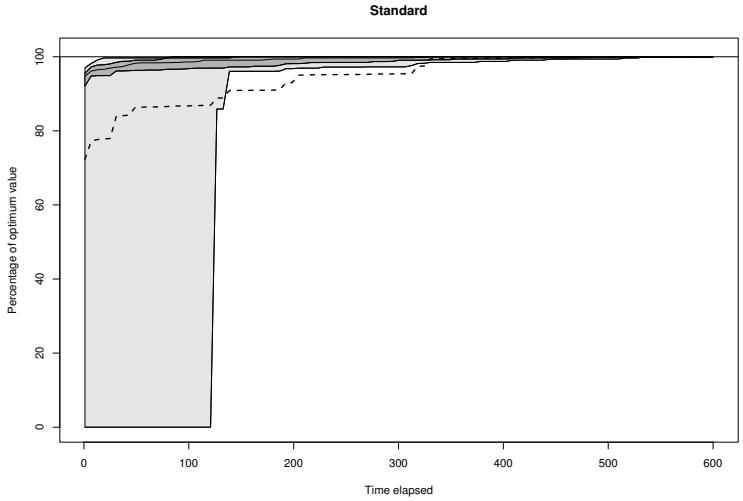


Figure 6.26: 6×12 instance set: Time course of standard

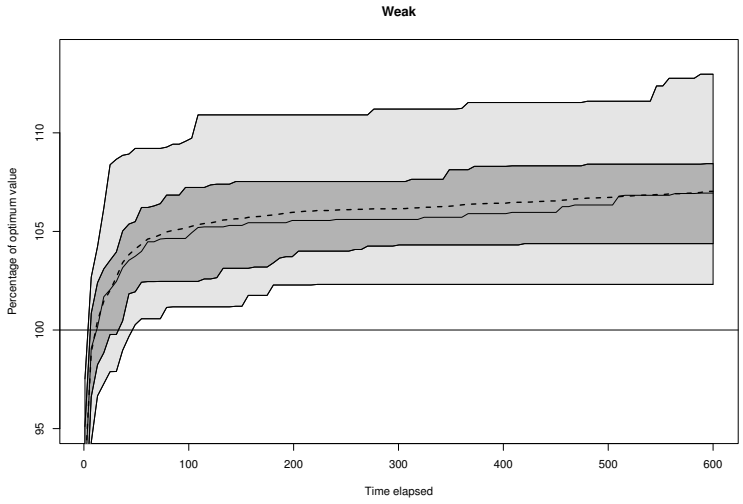


Figure 6.27: 6×12 instance set: Time course of weak

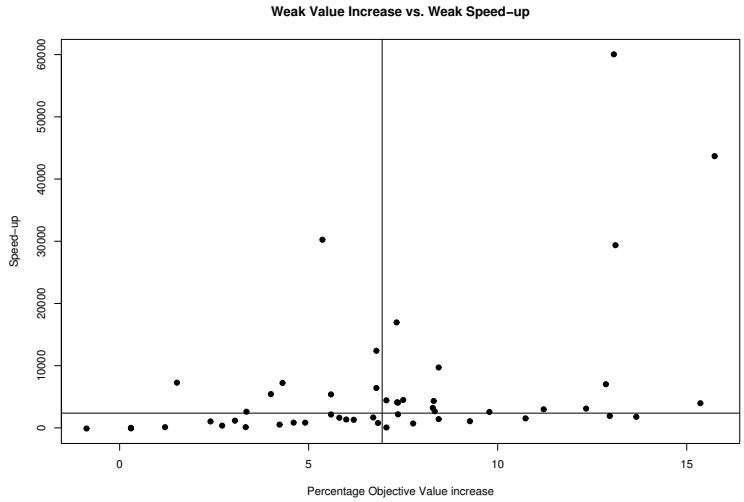


Figure 6.28: 6×10 instance set: Value increase and speed-up for the weak approach

The speedup for the first solution is positive in all instance sets and best in the 8×20 instance set with about 38 %. But also in all sets there were some instances where the standard approach outperforms the weak approach. See Figure 6.29 for example. In most cases the symmetry breaking helps to find a solution in P_1 early although more time in each choice point has to be invested due to the additional symmetry breaking constraints. Therefore, the weak approach finds the first solution mostly faster than the standard approach.

First solution quality:

In the instance sets 6×12, 8×12 the best first solutions were mostly found by the weak approach with a median increase of 3.7% and 1.6% respectively. For the instance set 8×20 the median value increase is negative. Also in this set no first solution of the weak approach had a better objective value than the first solutions of the standard approach. But since the weak approach increases the quality of the solution rapidly once the first solution is found this flaw is compensated within the first few permutations investigated in P_{sym}. Together with the observation that the weak approach finds the first solution faster in most instances the following happens: by the time the standard approach finds the first solution the objective value of the so far best solution of the weak approach outperforms this solution. See Figure 6.30 for example.

Best solution quality:

The best solution is found in all instances of each instance set by the weak approach. It seems that the gain increases with a higher number of rows (see Figures 6.31 and 6.32). Since each SymVar instantiation permutes a whole column, more variables are re-assigned if there are more rows. Therefore, each SymVar instantiation has the potential of altering more values. This could explain the better performance. Increasing the number of columns does mean that more symmetries are broken but also that more permutations have to be considered for each equivalence class. Therefore, it takes the weak approach longer until it can investigate a different equivalence class. Since the problem cannot be exhaustively investigated we cannot

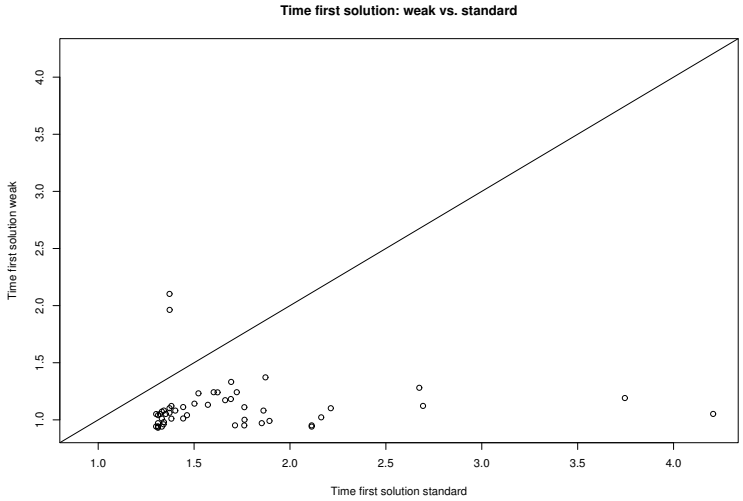


Figure 6.29: 8×20 instance set: Time for first solution

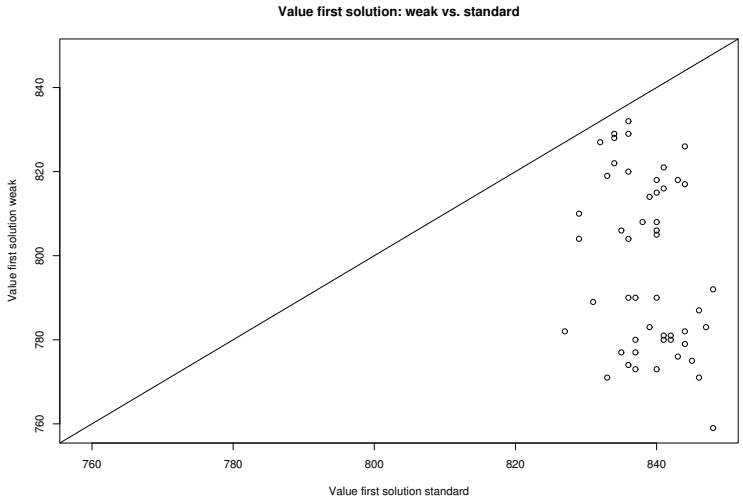


Figure 6.30: 8×20 instance set: Quality of first solution

see the full effect of the gain in symmetry breaking. Although the greedy approach does not find the optimal permutation at first (as the results on the quality of the first solution shows) the value increases rapidly. Therefore, it could be more efficient to investigate each equivalence class only partially such that more different equivalence classes could be investigated.

Search towards optimality Analysis:

The standard approach finds its best solution earlier in the larger instances. The 50% quantile reaches the 100% mark at about 450 seconds in the 6×12 and in the 8×12 scenario. But in the 8×20 scenario this is achieved at about 120 seconds. Also the other quantiles reach a high value earlier. After 100 seconds all quantiles of the 8×20 scenario reach a objective value of more than 99% of the best solution. In the 8×12 scenario the quantiles reach from 97.5 % up to 99 % and in the 6×12 scenario the quantiles reach only to 95-96.5%. But this does not mean that the standard approach improves on larger instances. It rather means that the solutions that are considered are so similar that they do not improve the objective value. Since the weak approach finds better solutions in all instances standard did not reach the optimum. It is most likely that in the standard approach it will take a considerable amount of time until a solution improves the objective value since a lot of variables have to be freed until significant changes in the solutions happen.

The weak approach outperforms the standard approach rather fast where better results are achieved on the larger instances. While the 100 % mark is reached between 20 and 110 seconds (from the 10 % quantile to the 90% quantile) this is achieved in the 8×12 scenario between 5 and 7.5 seconds and in the 8×20 scenario between 2 and 50 seconds. This shows how much potential is in permuting whole sets of variables. By permuting the columns via the SymVars all variables change their values during the investigation of P_{sym}. In the standard approach a lot of variables are not reassigned in the whole search interval. See Figures 6.33, 6.34, 6.35, and 6.35 for example.

Outperforming:

As already stated before the weak approach outperforms in all scenarios in terms of the solving time to reach the standard best solution and in terms of improving this solution. But also interesting is whether a high speed-up comes along with a small value increase or not.

In all scenarios no instance achieves a maximum in both categories. But there is no general trend and also the instances are rather distributed than clustered. Therefore, no general statement can be given on the correlation of speed-up and value increase. See Figure 6.37 and Figure 6.38 for example.

6.4 Weighted Graph Colouring

6.4.1 Problem Description

Consider the problem of Section 4.2.3.8. We want to colour of a graph consisting of n nodes with m colours, such that two nodes connected by an edge have different colours. Each node i has a ranking for each colour j $ColRanking_{ij}$ expressed by a number. So a colour with rank one fits best and the higher the number the worse the colour fits. Sought is a labeling $colouring_i$ for each node i where adjacent nodes

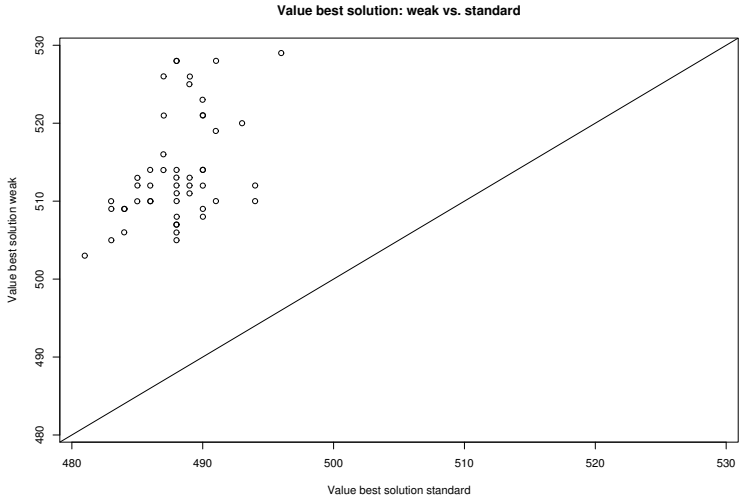


Figure 6.31: 8×12 instance set: Value of the best solution

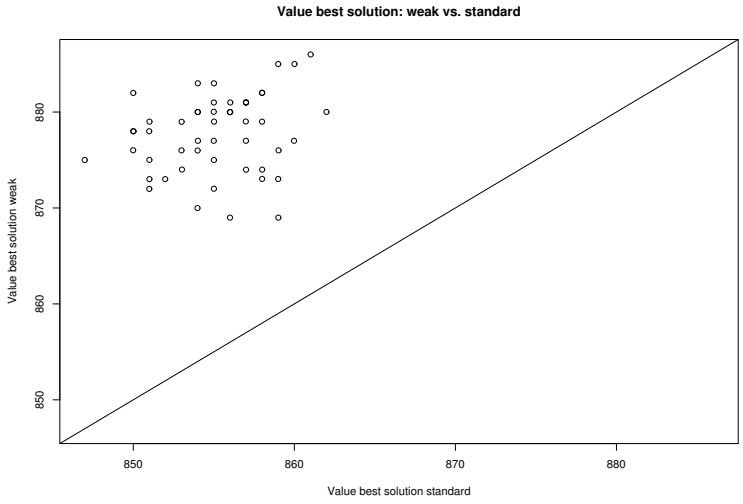


Figure 6.32: 8×20 instance set: Value of the best solution

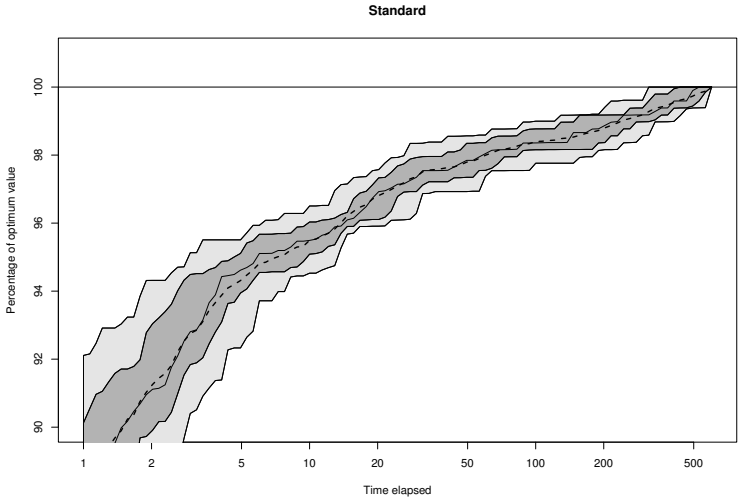


Figure 6.33: 8×12 instance set: Convergence toward optimum

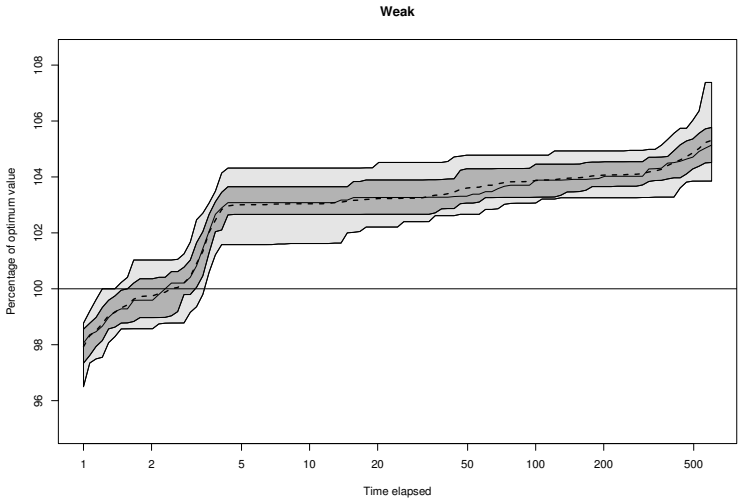


Figure 6.34: 8×12 instance set: Convergence toward optimum

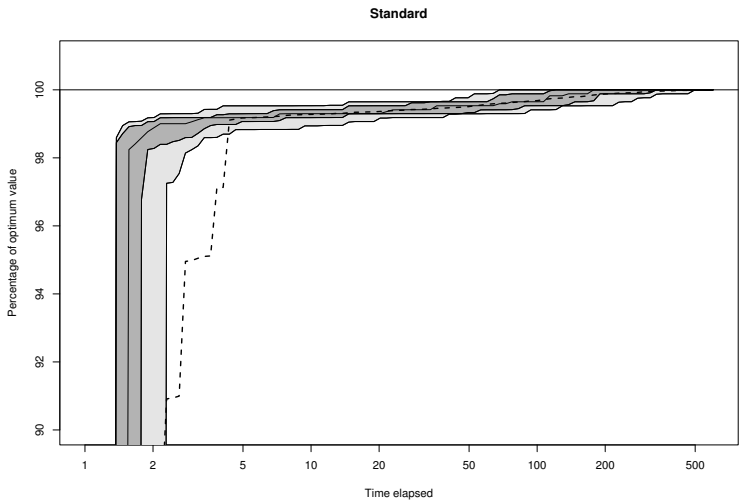


Figure 6.35: 8×20 instance set: Convergence toward optimum (logarithmic x-scale)

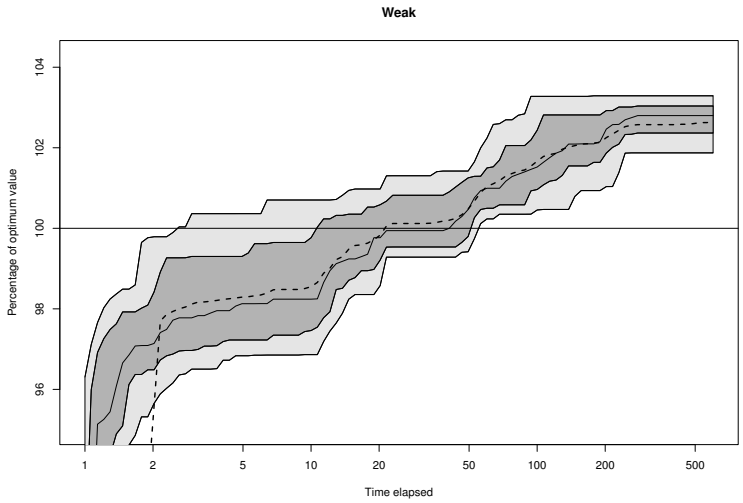


Figure 6.36: 8×20 instance set: Convergence toward optimum (logarithmic x-scale)

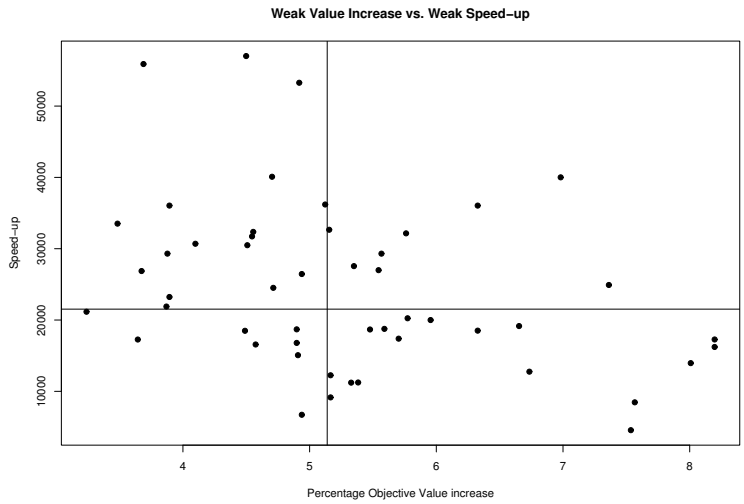


Figure 6.37: 8×12 instance set: Value increase and speed-up for the weak approach

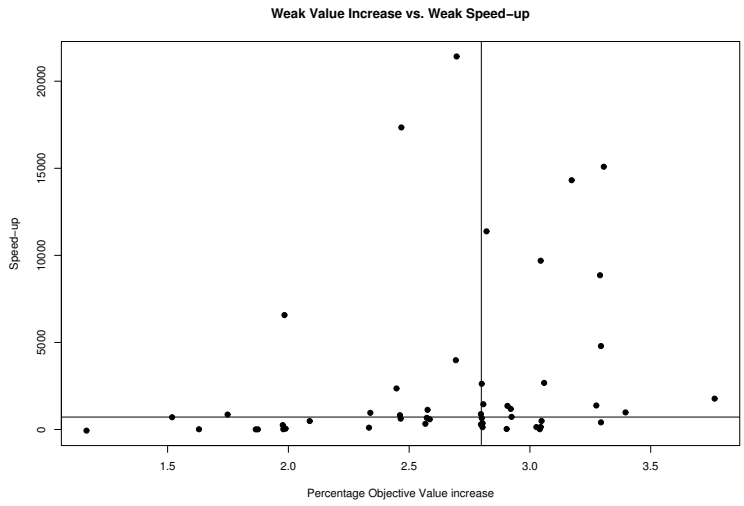


Figure 6.38: 8×20 instance set: Value increase and speed-up for the weak approach

do have different colours and the overall ranking of the assigned colours is best. This is expressed by the objective function:

$$\text{minimise } \sum_{i \in \{1,\ldots,n\}} ColRanking[i, colouring[i]]$$

6.4.2 Models

6.4.3 Standard Model

The weight of the colours make the colour permutation a weak symmetry. Therefore, no symmetry breaking is possible in the standard model.

6.4.4 Weak Model

In the weak model a SymVar $colourPerm_i$ for each colour i is introduced. Also an all different constraint is stated on the set of SymVars. The column permutation is broken by introducing a constraint that states that each colour may only be as often assigned as the previous colour in the colour list.

Consider for example three colours. If the first colour in the list is assigned five times the second and third colour can only be assigned five times as well.

6.4.5 Results

The first test indicated that the standard approach outperforms the weak approach clearly. Where a instance is solved exhaustively in seconds by the standard approach the weak approach took minutes. This was the case in all instances regarded. There is one main reason why the weak model is not efficient in this problem. In the weak approach it is not possible for a partial assignment of P_1 to indicate the solution quality (i.e. the objective value) since computing the objective value is done in P_2. Therefore, a partial assignment in P_1 has to be completed and only with the instantiation of the SymVars the objective value can be computed. In the standard approach a partial assignment can be abandoned whenever the partial solution considered cannot be extended to form a better solution.

In this scenario the saving in pruning partial solutions by the objective value clearly dominates the saving of symmetry breaking and the additional effort of investigating P_{sym} in the models.

This problem shows that weak symmetry breaking is not successfully applicable for all problems or that other techniques are needed to overcome this flaw. A pure modeling approach as investigated in this problem was not fruitful.

Chapter 7

Conclusions and Future Work

Weak symmetries are only recently found and investigated. There is still a lot of potential in this field and a lot of research directions are open to find more applications of weak symmetries. We will point out some of these directions in this section and state our ideas.

Especially in DisCSPs (see Section 4.2.4) we have showed that symmetries happen to be weak symmetries in general. Therefore, it seems to be fruitfully to incorporate a weak symmetry breaking technique directly in a solver for DisCSPs. Also in the fields of symmetry breaking especially automatic symmetry detection is an interesting field where weak symmetries detection could profit from. Our main approach is based on pure modelling which allows a wide field of applications since no extra code or special expertise is needed to apply the technique. Chapter 5 shows that it is possible to increase efficiency of the approach by implementation. Therefore, it would be interesting to incorporate these ideas and others in a solver to form a global constraint for weak symmetry breaking. In the Section 7.1 to Section 7.9 we describe possible future research directions and in Section 7.10 we give the conclusions of this thesis.

7.1 Multiple Weakly Decomposable Problems

There is no limitation of the weak decomposition to exactly two subproblems. It is possible that a problem splits weakly to several subproblems and that between each a weak symmetry is present. Such a problem then would split in $P_1 - P_{sym_1} - P_2 - P_{sym_2} - \ldots - P_n$.

The Definition 4.1 would then change to

Definition 7.1 *Multiple Weakly Decomposable Problem A problem $P = (X, D, C)$ is **multiple weakly decomposable** if it decomposed into $n > 2$ subproblems $P_i = (X_i, D_i, C_i), 1 \leq i \leq n$ with the following properties:*

$$
\begin{aligned}
X_1 \cap \ldots \cap X_n &\neq \emptyset && (7.1)\\
X_1 \cup \ldots \cup X_n &= X && (7.2)\\
C_1 \cup \ldots \cup C_n &= C && (7.3)\\
C_i \cap C_j &= \emptyset, i \neq j && (7.4)\\
C_i &\neq \emptyset, i > 1 && (7.5)\\
D_i &= pr_i(D) && (7.6)\\
& && (7.7)
\end{aligned}
$$

An example where this could happen is a problem that consists of two variable matrices A_1 and A_2. The constraints induce a weak symmetries on each of the matrices itself. Also A_2 could only be determined if A_1 is assigned due to some constraints connecting these two matrices. In this case A_1 would be instantiated first, a symmetric equivalent would be assigned. For that A_2 is instantiated and a symmetric equivalent is determined.

7.2 Weak Symmetry Detection

Automatic symmetry detection is a field with much interest in. Since expertise and experience in symmetry is needed to detect symmetries the desire is to free the user from the need to have this expertise. Weak symmetry detection is somehow different from symmetry detection. The problem is that a normal test, whether a problem is symmetric or not would fail since the problem is *not* symmetric. We therefore propose a different method. We rely on the fact that there is an oracle that can tell us, whether a problem is symmetric or not and tells us which are the symmetries. We are aware that this oracle may not exist in such a way but research in this field is very active. Approaches like determining the graph automorphism group [72] are able to detect symmetric structures already.

7.2.1 Proposal for Automatic Detection of Weak Symmetry

We assume that our problem is asymmetric in the following. We will regard symmetric problems in the Section 7.2.2 We first start with a test, whether a problem P is symmetric. Therefore, we use our symmetry oracle. The test fails since the problem is asymmetric by assumption. We then choose P_1 and P_2 to be empty and $P' = P$. P_1 is symmetric by definition. We then consecutively choose a constraint c_α and assign it and all corresponding variables of the constraint to P_1 such that $C_1 = \{c_\alpha\}$ and $X_1 = \{x_i \in c_\alpha\}$. We check whether P_1 is still symmetric via our symmetry oracle. If so the constraint and the variables stay in P_1. If not, this constraint have to be in P_2. Variables are removed from X_1 if they have no supporting constraint in C_1. This is repeated for all constraints until P' is empty. By doing so the symmetric part of the problem is accumulated in P_1 as desired and the asymmetric part in P_2.

If P_1 is empty at the end then the problem is not weak symmetric (and also not symmetric since the initial test showed this). If P is weakly decomposed by this algorithm then by construction of the decomposition the problem has weak symmetries that can be indicated by the symmetry oracle.

7.2.2 Proposal for Automatic Detection of Weak Symmetry in the Present of Proper Symmetries

There are problems that have proper *and* weak symmetries. Consider the problem from automated manufacturing introduced in Section 4.2.3.5. In this problem the row permutation is a normal symmetry while the column permutation is weak.

An initial test of symmetry would reveal that the row permutation is a symmetry. We reformulate the problem in the way that we introduce symmetry breaking constraints that break the row permutation. Checking again, the problem is determined asymmetric. We proceed as described in Section 7.2.1. The only difference is that we check the newly introduced symmetry breaking constraints at last. Up to this point the row permutation is indicated to be a symmetry on P_1. When checking with the symmetry breaking constraints the row permutation is not indicated a symmetry anymore. If no other symmetry is indicated, then the problem does not contain weak symmetries. In the case of the problem from automated manufacturing the column permutation would remain.

7.3 Parallelise the Weak Symmetry Approach

The weak symmetry approach by introducing SymVars can be parallelised if several processors are available. This is a very important feature since parallel processors are becoming more popular even in standard computers. Therefore, the need for parallel programmes rise. Constraint solvers can work in parallel and so it is very desirable to find models that can be parallelised.

Since the problem is not fully decomposable but weakly we cannot solve P_1 and P_2 fully independently.

Consider that we have n processors p_i and on each a constraint solver can run and want to solve the problem P with the weakly decomposition P_1, P_{sym}, P_2. P_1 is passed to the processor p_1. Whenever a solution s_i is found, it is passed to any processor p_i as long as there are free processors. Each processor p_i solves P_{sym} and P_2 (in case P_2 is just the objective function) for the passed solution s_i. Therefore, up to $n-1$ solutions of P_1 can be permuted simultaneously. The best objective value is stored in p_1 and each processor uses this objective value for pruning issues. That means that the individual processors do not only prune corresponding to their local best solution but on the global best solution.

If P_2 is not just the objective function then a solution from a processor solving P_{sym} passes its solution to a processor that solves P_2.

A drawback is that not all processors can work in parallel from the beginning since first a solution to P_1 has to be found. Still the problem P_1 is parallisable in the way that different processors start with different search strategies. But the topic of parallelise CSPs is beyond this work such that we do not have expert knowledge on all the possibilities. Another drawback is that if all processors are busy no other solution of P_1 can be processed. This could be compensated by putting solutions in a queue and recalled them if a processor becomes idle.

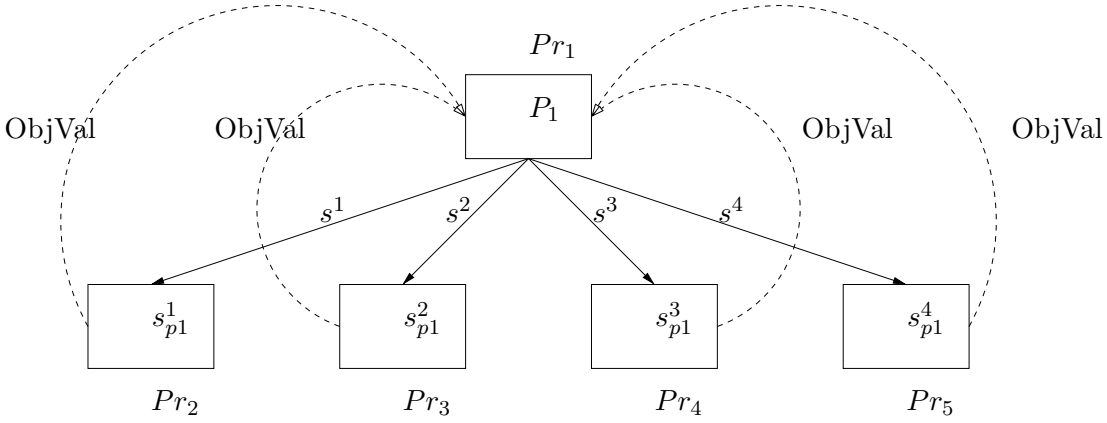


Figure 7.1: Processor Pr_1 passing solutions to the other processors which solve P_{sym} for this solution

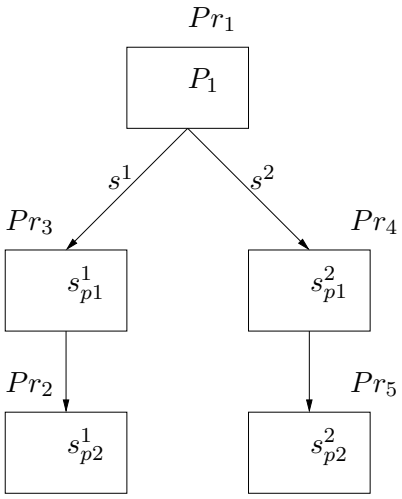


Figure 7.2: Processor Pr_1 passing solutions to the processors Pr_2 and Pr_3 which solve P_{sym} for this solution. Each of them in turn pass their solutions to a processor with solves P_2.

7.3.1 Parallelise for Separable Objectives

The approach is also combinable with the approach for separable objectives. The neighbourhood degree is determined for a solution in P_1. Instead of just passing the solution to a different processor also the neighbourhood degree is passed which determines automatically the SymVars that have to be instantiated. If the solution passed is a cardinal solution the partial permutations have to be stored. This data could either be passed to the processor solving P_1 or stored locally. As soon as these variables are instantiated the corresponding partial solution is called to complete the solution.

There is a problem with changing neighbourhoods. Since several solutions are computed in parallel it is likely that the cardinal solution changes for some of the solutions. In this case a copy of the results from the old cardinal solution has to be stored until all solutions referring to this approach are fully investigated. This means that the memory is not longer bounded as shown in Theorem 5.13. But since each processor has its own memory it is possible that the information is just passed to the individual processors. This could either be done incrementally or a processor is determined that stores the complete data for a specific cardinal solution. All other processors solving solutions neighboured to this cardinal solution get their data from this processor.

It is not sure whether the approach introduced in Section 5.7 based on [80] can also be used for parallelisation fruitfully. Since the search is performed more in breadth-first manner it could be that solutions are found to late such that many processors stay idle until solutions are passed to them. This may only be a problem if P_2 is not just the objective function.

7.4 Distributed CSPs

The main feature of distributed CSPs is that not all knowledge is available for one solver. Some constraints and/or variables may be only available to other solvers. This scenario happens often when different companies have to work together and trade secrets have to be kept save. In this case the company would not reveal their constraints. Instead a partial assignment is passed to them and their solver decide whether this assignment does satisfy their constraints. This is not restricted to constraints. Also such a solver could return additional variable assignments and this data is passed to a different solver which hides some data on its own and so forth.

A scenario that is also a topic of DisCSP is also security [94, 37]. Related to this topic is the task to *guess* the constraints of a different agent B. In our scenario we would like to know whether a symmetry of an agent A is a weak symmetry of a proper symmetry. In the latter case we could break the symmetry without using SymVars to search the symmetry group for identic solutions. This would speed up the search considerably.

The idea is to pass symmetric solutions from agent A to agent B and evaluate the results returned. If it is very likely that the symmetry is a proper symmetry then the search is altered in a way such that only unique solutions are passed to the agent B. Since it can never be said with absolute guaranty that the symmetry is proper the

symmetric solutions cannot be excluded from search. This way the method would be incomplete. But it could be used as a search heuristic as mentioned above.

7.5 Advanced Global Constraint for Separable Objectives

As already stated in Section 5.7.6 if the approach for separable objectives is implemented there are two possible methods. First the approach discussed in Section 5.3 and second the approach discussed in Section 5.7. The first one is more general while the second one relies on the fact that for corresponding partial variable assignments the remaining subproblem is identic. Both have their advantages and disadvantages so it would be a good idea to implement both and use them for the appropriate scenario. Even better would be to incorporate both algorithms in one global constraint. This global constraint then analyses the problem and the scenario and uses the most appropriate method.

Since the approaches are also interesting for standard permutation problems and not only for weak symmetry breaking this would be interesting beyond the fields of symmetry breaking.

7.6 Restarts for P_{sym}

In scenarios where the problem is not exhaustively investigated (online optimisation, least quality evaluation, satisfaction without proof for the existence of a solution) we do not need to investigate each subproblem exhaustively as well. The question is where it is more profitable to apply more search. In P_1 we cannot use information of P_2 for pruning. Therefore, we do not know which criteria we should follow to abandon the search in P_1.

In P_{sym} we can determine (in optimisation) the objective value. Therefore, we could abandon search in a tree that seems not very promising. To investigate different regions of the search space restarts would be a good choice. A restart means to abandon the search completely and return to the root restarting search in a different branch of of the search tree. This way more different regions can be investigated and total different permutations can be investigated instead of several solutions that share a common path in the search tree.

7.7 Alternative Models

As already mentioned in Section 4.5.4.3 weak symmetries do not have to be handled by SymVars. It is also possible especially for small symmetry groups to handle the weak symmetries different.

7.7.1 Weighted Magic Square

Consider the weighted magic square problem from Section 4.2.2.4 with the diagonal constraints. As in the normal version of the problem we just have the chessboard

symmetries in the problem but these are weak (because of the weights). We need two SymVars, one for each generator of the group. These elements are rotation and reflection. The SymVar for rotation has a domain of $\{1, \ldots, 4\}$ because of the four possibilities to rotate the board by 90 degree and the SymVar for reflection has two values indicating identity or reflection.

7.7.2 Small Symmetry Groups

If the group of a symmetry is rather small like the group of the chessboard symmetries it could be profitable to model weak symmetries in a different way. Instead of introducing SymVars we model the weak symmetry in the objective function. This way we regard all the weak symmetric (partial) assignments simultaneously. We backtrack if none of the weak symmetric assignments leads to a solution.

Since all weak symmetric equivalents have to be evaluated simultaneously this can only be done fruitfully for small groups. Therefore, the chessboard symmetries seem to be the largest possible group to investigate in this way. But this may vary from solver to solver. The gain in efficiency is still not very clear. It may be that an approach with SymVars outperforms this method for some problems and vice versa. This may mostly depend on the gain in pruning from the objective value. If it is possible to achieve more pruning by the objective value it seems fruitful to use the approach via simultaneously regarding weak symmetric assignments.

Example 7.2 *Asymmetric TSP*

In the asymmetric TSP introduced in Section 4.2.2.3 the distance between two nodes i and j is different: $d(i,j) \neq d(j,i)$. Therefore, reversing a tour is weakly symmetric. It is possible to model the weak symmetry without a SymVar. The weak condition is then expressed directly in the objective function. Consider that the length of a tour is stored in a variable ℓ. The objective is then $\min \ell$*. We can also store the length of a tour in reversed order in a variable ℓ_r.*

The objective now is: minimize $\min\{\ell, \ell_r\}$*. Therefore, when constructing a tour both lengths are computed incrementally and only if both are higher then the best found solution the search backtracks. Note that ℓ_r is not a SymVar but an objective variable that just sums up the length to state the objective value more convenient.*

If we use this objective function we can break the symmetry of reversed order of a tour. The drawback is that we have to compute two values at a time but we can use the objective value to achieve pruning which would not be possible if we would have split the problem and introduced a binary SymVar.

7.8 Partial Symmetry Breaking

In some problems not all symmetry can be broken or is wanted to be broken. Since we have seen in Chapter 6 symmetry breaking does not also shrink the search space but it also means more work for the constraint solver in each node of the search tree. Sometimes this additional work does not pay-off. Therefore, one strategy is to break only a subset of the symmetries and hope this is more efficient. One crucial questions arise in this approach which is not broached here: Which symmetries to break. In the context of weak symmetries partial symmetry breaking has an other

delicate problem. The weak symmetry group has to be adjusted to the symmetries that are broken in order to not find identic solutions again in the search process. Consider for example a permutation of four elements and only the permutation of the last two is prohibited by symmetry breaking constraints. If we still allow all permutations in P_{sym} then we will also permute the first two elements as well in P_{sym} although they are investigated explicitely in P_1. Therefore, we have to restrict the weak symmetry group to only contain the elements that are excluded in P_1 by symmetry breaking. In this case it is rather simple. For each element of a permutation that is not restricted by symmetry breaking constraints (in the above example the first and second element) we state a constraint in P_{sym} that prohibits these elements to be permuted. For each such an element the SymVar is fixed to its identity: $SymVar_i = i$. This way partial symmetry breaking can be performed in P_1 but no solution will be investigated twice in the search. This again is an example how flexible the weak symmetry approach is and what potential it has.

7.9 No Symmetry Breaking at all

It may be profitable to do no symmetry breaking at all in P_1 for a weak symmetry but still perform search in the weak symmetry group in P_{sym}. Clearly this means to find identical solutions over and over. But there are some scenarios where this could be profitable. Note that this will not pay-off for exhaustively investigate a problem since the search space is enlarged by this. But in satisfaction or especially online optimisation (within a given time limit) there seem to be scenarios where this can be applied fruitfully. If for example the search in P_1 will find solutions in an order such that they are of different equivalence classes mostly this means that up to a certain point all solutions considered will be unique. Investigating the equivalence classes of these solutions does deliver a lot of solutions such that more solutions could be investigated in the same time as would be possible in the same time in a standard approach. In satisfaction this increases the chance to find a feasible solution while especially optimisation profits from this since the objective value may be increased early and in the end a better solution is found. It is also possible to direct the search in P_1 such that solutions of different equivalence classes are found. This way the negative effects of symmetry breaking can be avoided but still profit from the richness of solutions found by the SymVars in P_{sym}. This would not be possible otherwise.

7.10 Conclusions

Weak symmetries are very common in CSPs and especially in real world problems. Often there are more weak symmetries than standard symmetries in a problem. While weak symmetries cannot be broken by standard symmetry breaking methods we introduce a modelling approach that enables us to do exactly this. By applying our technique a weak symmetry can be broken by any standard symmetry breaking method. Therefore, besides introducing the first computationally proven successful approach for handling weak symmetries, we provide new fields where standard symmetry breaking can be used. Therefore, in future also more complex and interesting problems can be investigated using symmetry breaking.

Again we state the advantages of our approach that we showed in this thesis:

- **Universality:** Every solver uses a modelling language to state problems that are passed to the solver. Our approach does not need additional implementations so we are not limited to a specific solver.
- **Ease of use:** A person familiar with modelling can adept the approach easily and is capable of remodelling a problem to break weak symmetries. Although modelling needs some expertise the principles of modelling weak symmetries are very easy to understand such that even inexperienced constraint programmer can use the technique immediately.
- **No background knowledge required:** Symmetries are based on groups and therefore the theory of symmetries and symmetry breaking is group theory. It is possible to use weak symmetry breaking without specific knowledge of group theory.
- **Readiness:** Since no extra code has to be written, incorporated or adjusted, our approach can be used instantly. Existing models can be easily upgraded with weak symmetry breaking with just a few changes to the constraints.
- **Interoperability:** When a model is revised, the weak symmetry can be handled using standard symmetry breaking methods such that the approach also profits from research in this field. Any symmetry breaking method can be used once the model is revised in the way we propose.
- **Concurrent Symmetry Breaking:** Problems may contain standard and weak symmetries. Both can be handled concurrently since weak symmetry breaking actually transforms a weak symmetry into a standard symmetry from a certain viewpoint.
- **Robustness:** Since no method specific code has to be added to the existing solver, the chance of producing errors is minimal and reduced to the validity of the model. But there is no problem with memory management, exception handling, etc.
- **Openness to Refinement:** The basic approach can be extended in several ways to suit different applications. For example, it is easy to adopt the approach for *partial symmetries*. Although based on modelling, it is also possible to extend the approach by incorporating code to the constraint solver. This way the approach can be adopted to various applications and scenarios in order to maximise efficiency.

We showed that our modelling approach is easy to apply for constraint programmers with any level in expertise and also independent of the constraint solver used or additional code. The approach can easily be extended and also incorporated in a global constraint in order to be more efficient for problems with special properties. Also the weak symmetry approach can be combined with every symmetry breaking method and therefore profits from research in the fields of standard symmetry breaking as well. We also showed in two scenarios that the approach outperforms a standard approach in terms of the search time and the quality of the solution. Our approach is the first approach that handles weak symmetries based on pure modelling successfully.

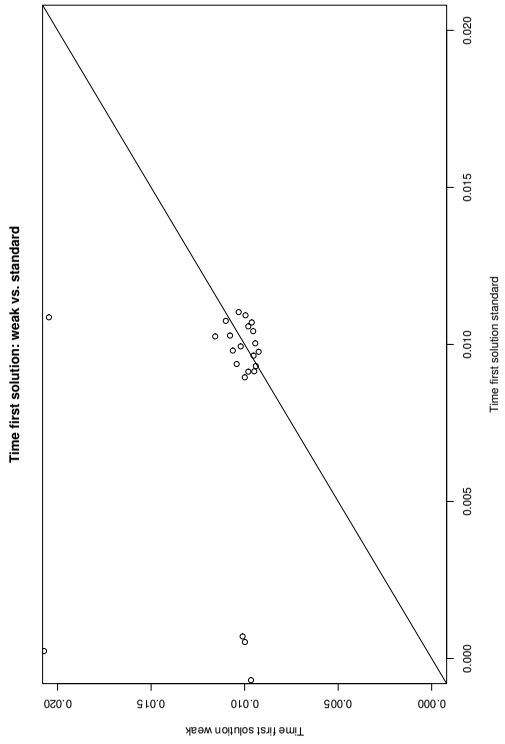
Appendix

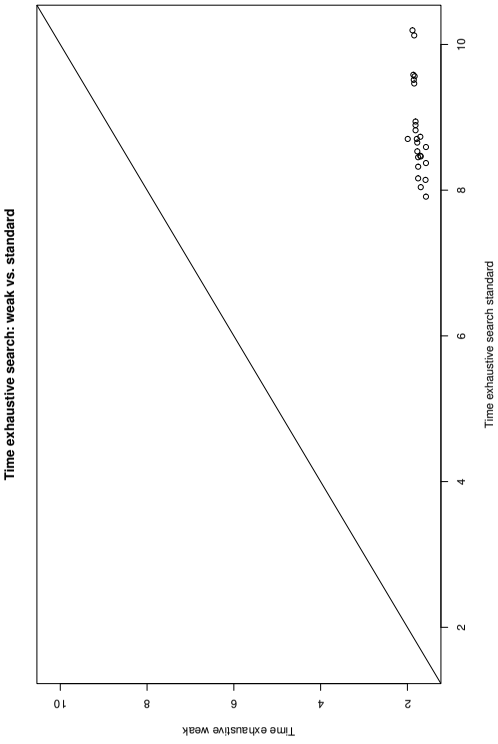
In Chapter 6 we only showed a few plots of the instance sets that show a specific behaviour. For the sake of completeness we present here the all plots such that the results can be observed.

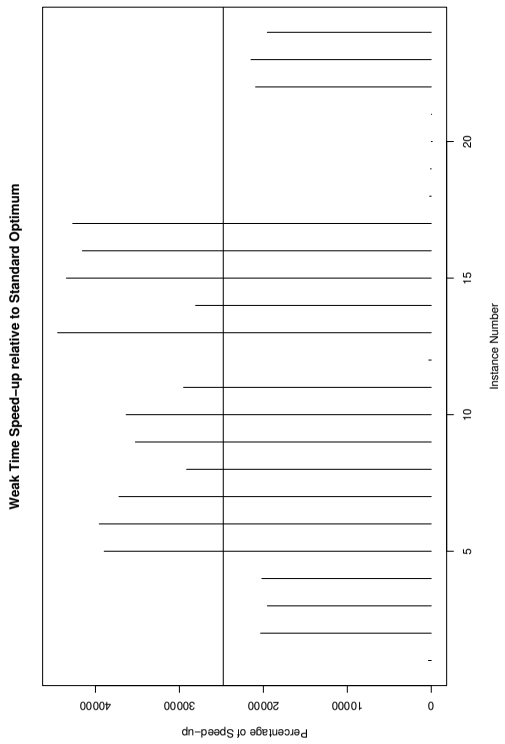
Weighted Magic Square

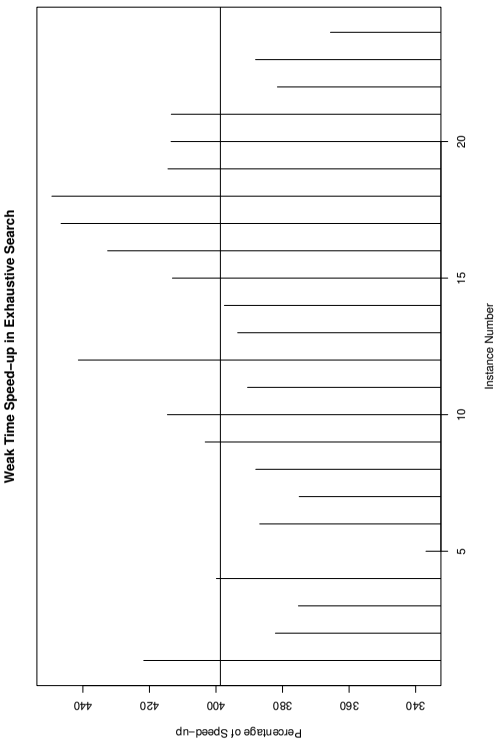
Scenario Weights on Columns (pairwise different)

Instance Set 4 Columns Total Weight 7: Maximisation

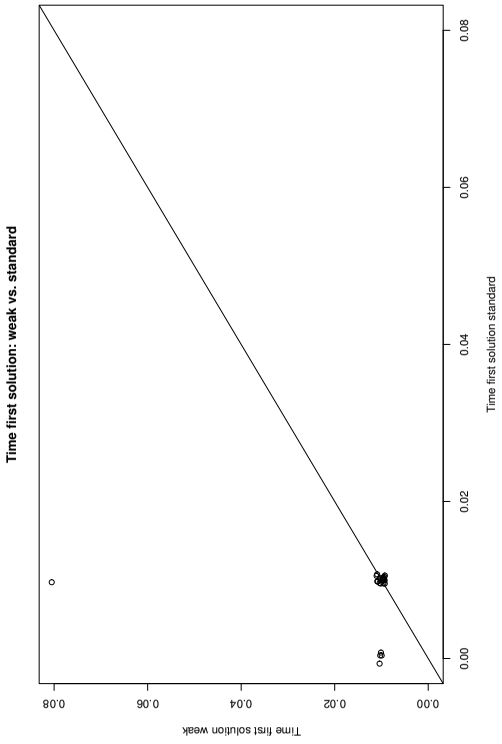


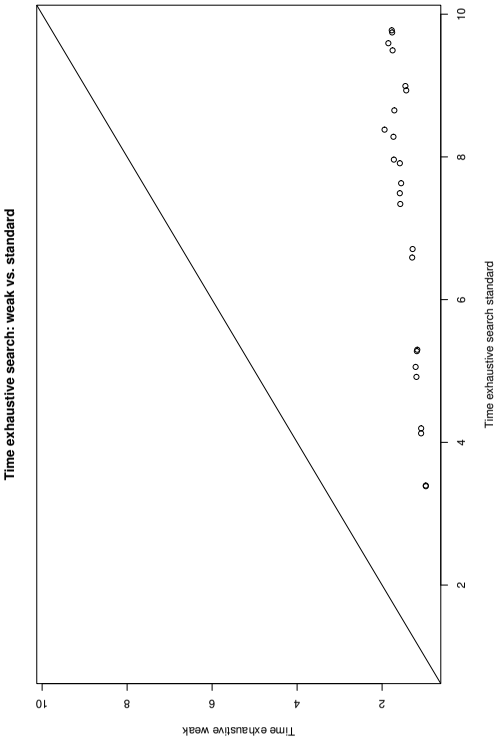


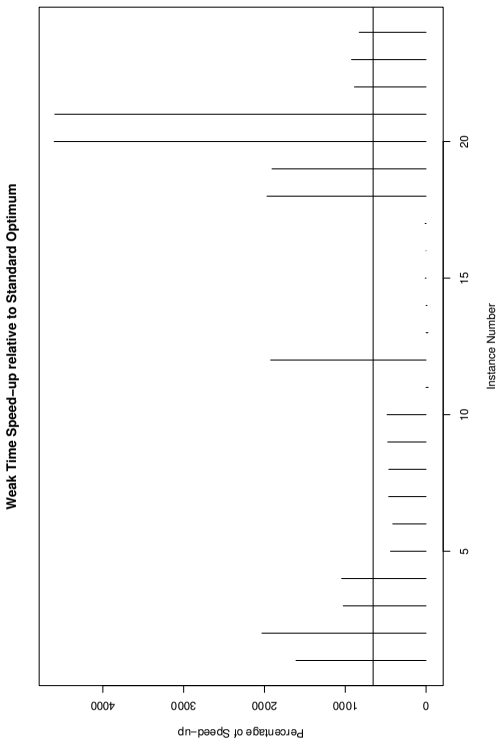


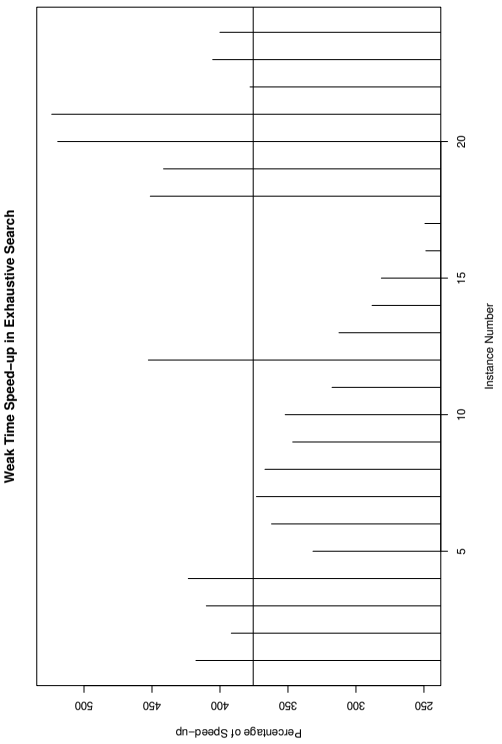


Instance Set 4 Columns Total Weight 7: Minimisation

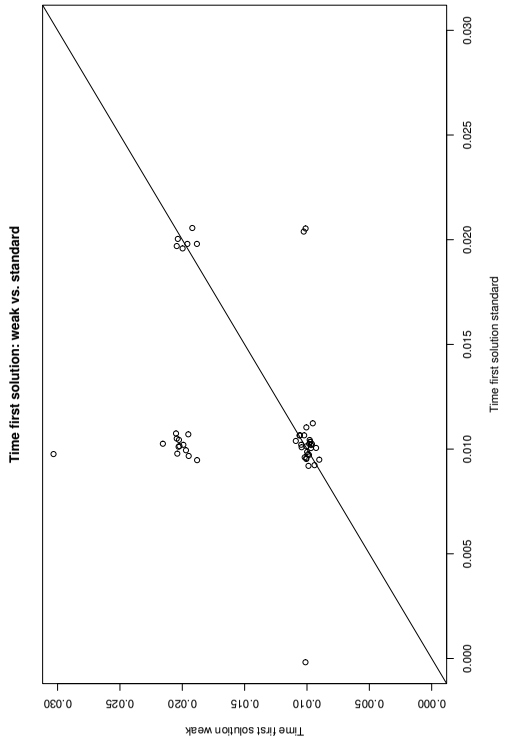


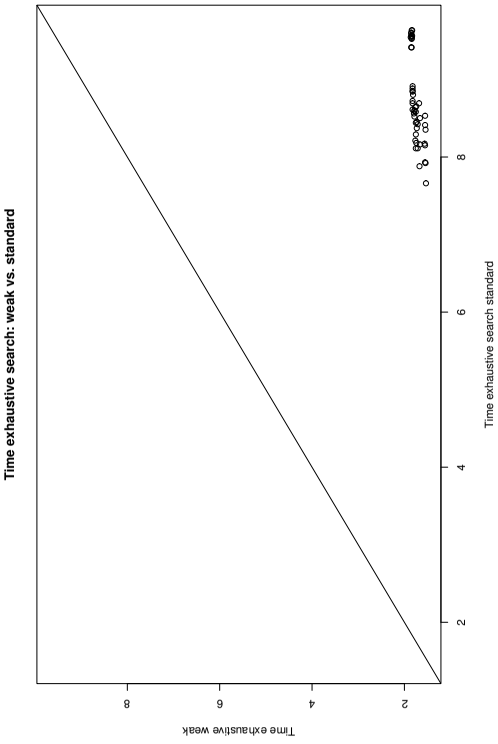


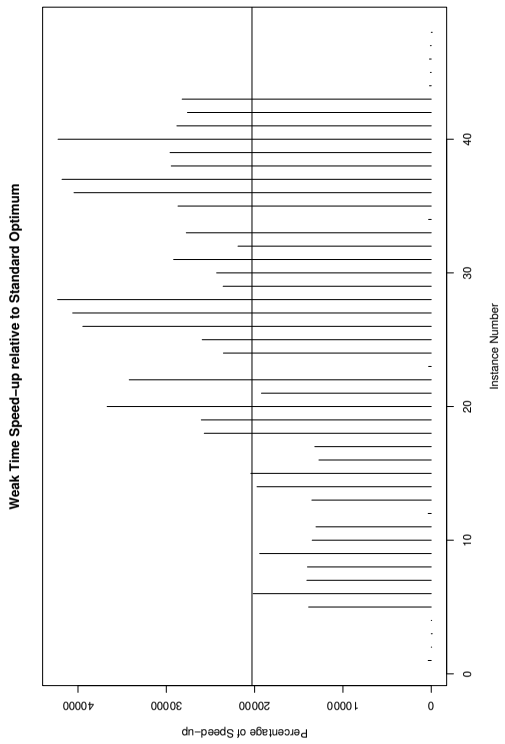


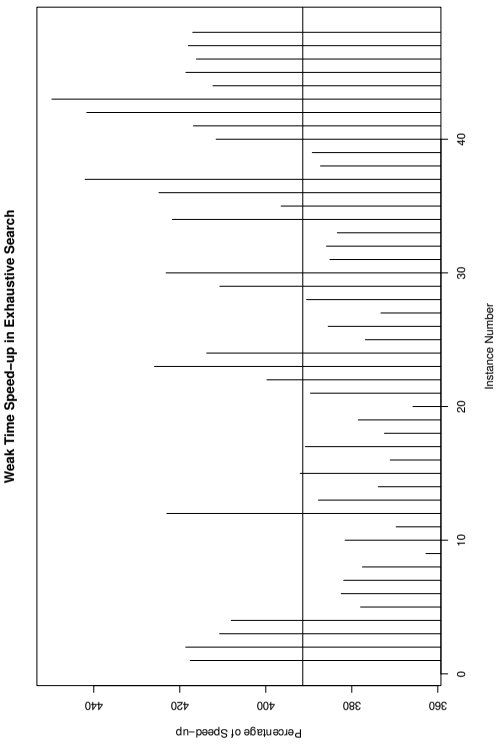


Instance Set 4 Columns Total Weight 8: Maximisation

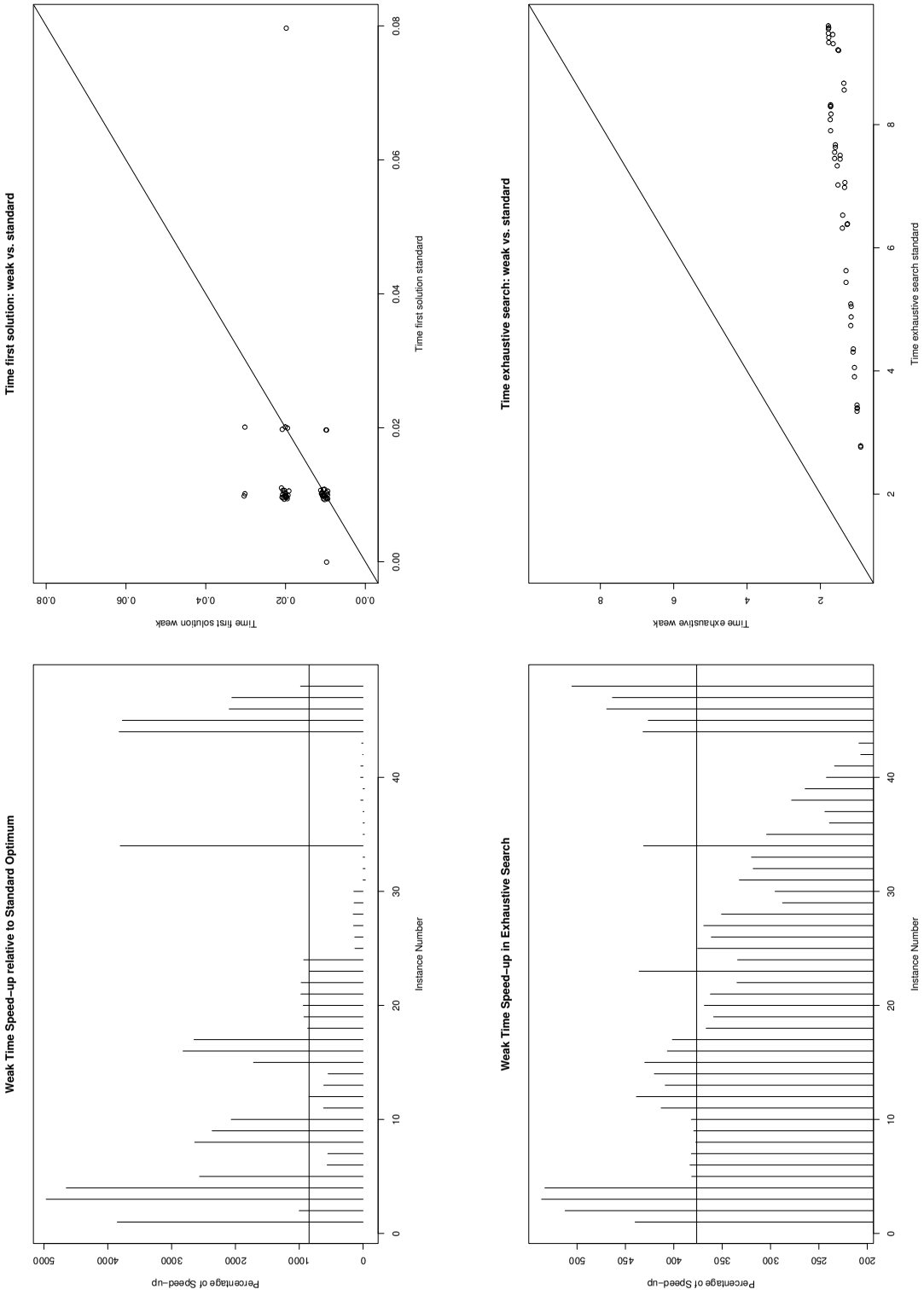




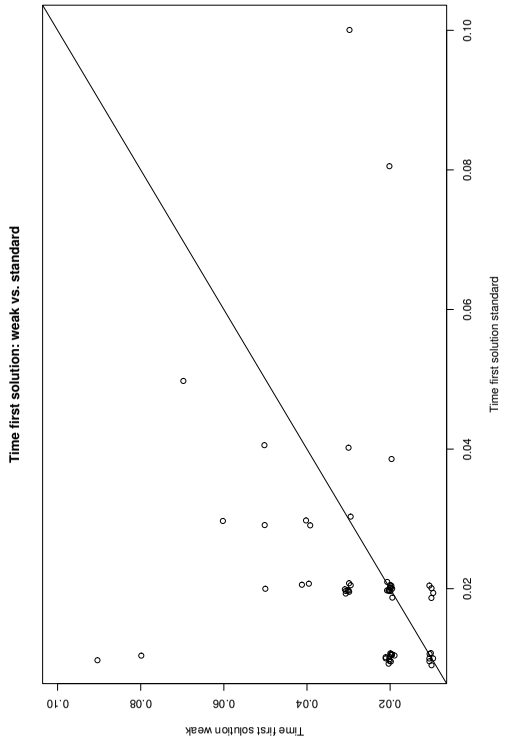


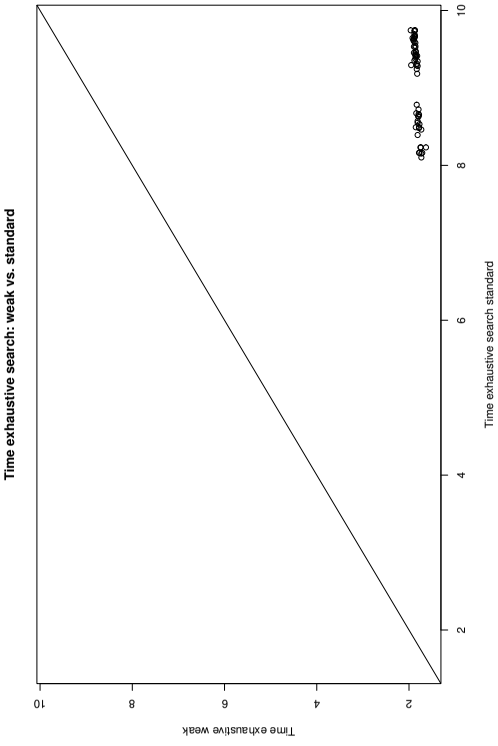


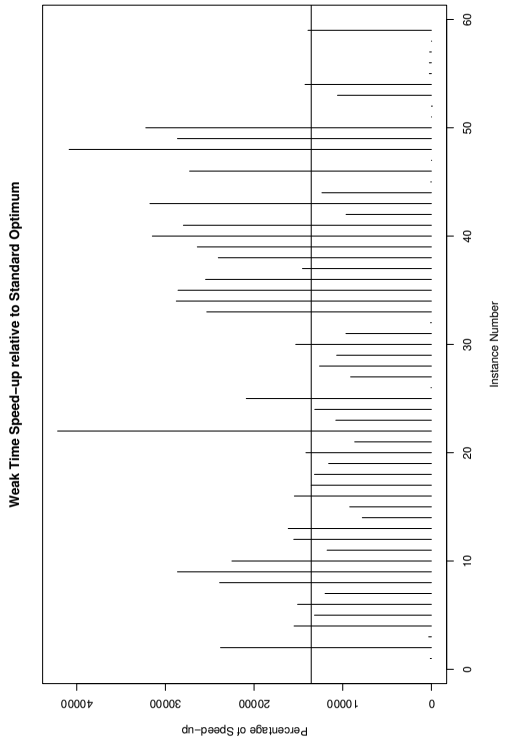
Instance Set 4 Columns Total Weight 8: Minimisation

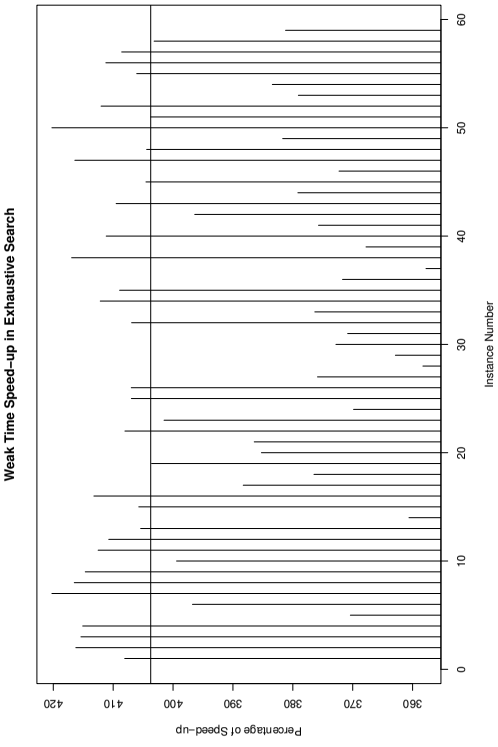


Instance Set 4 Columns Total Weight 15: Maximisation

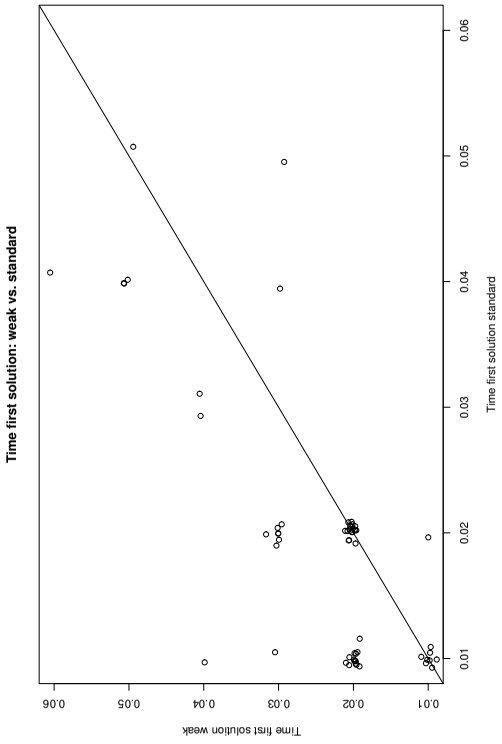


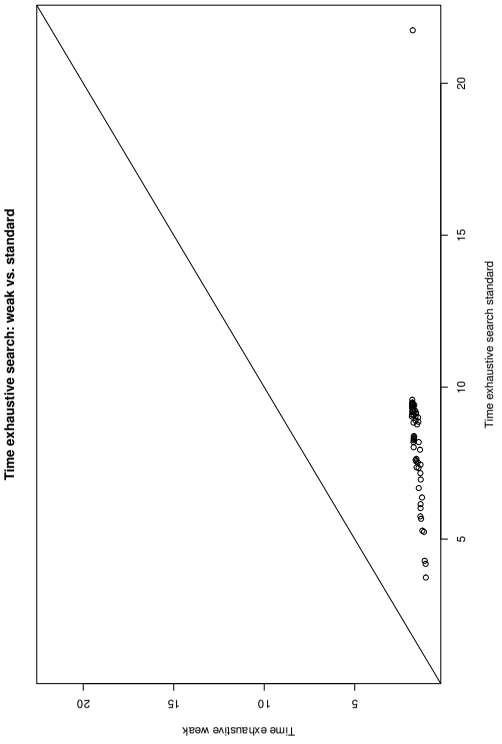


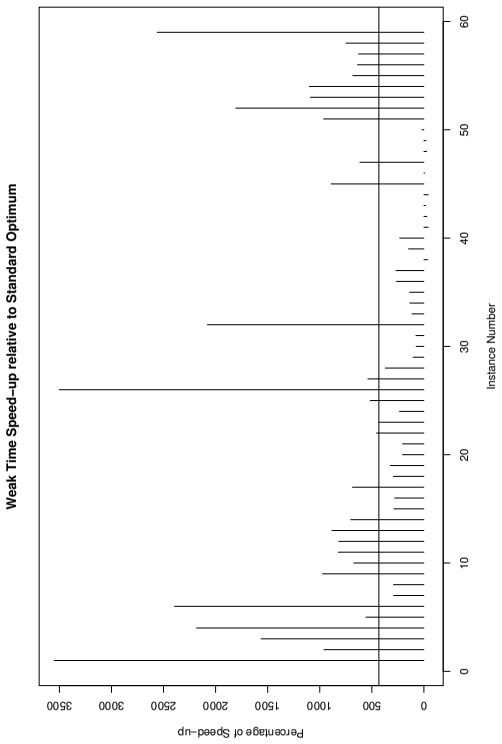


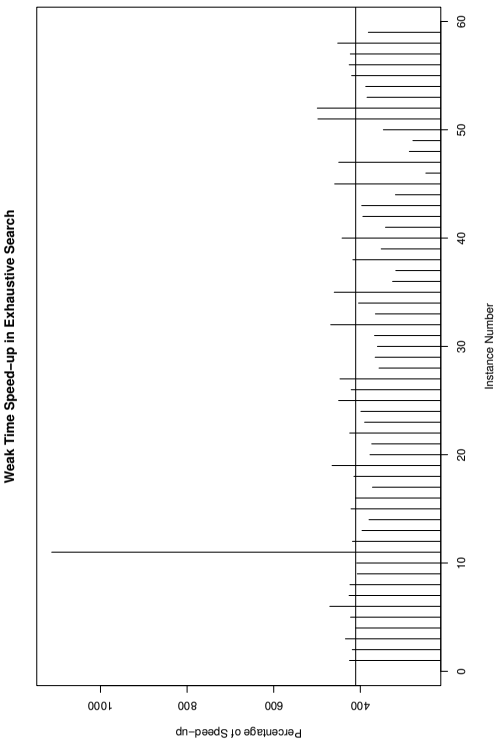


Instance Set 4 Columns Total Weight 15: Minimisation

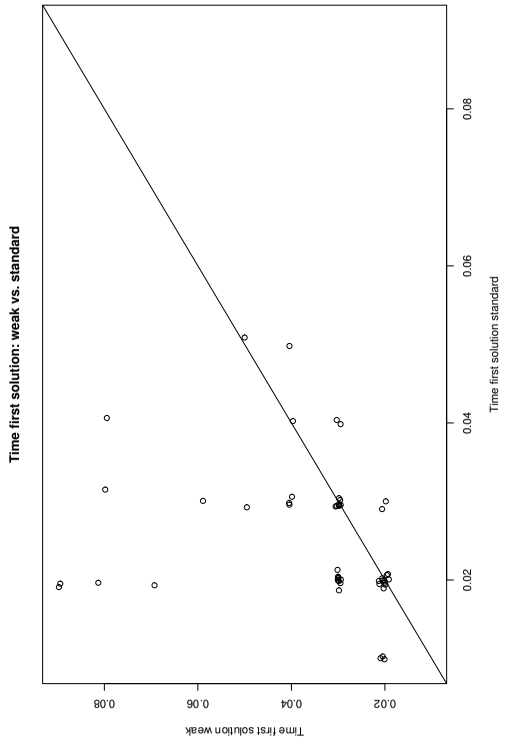


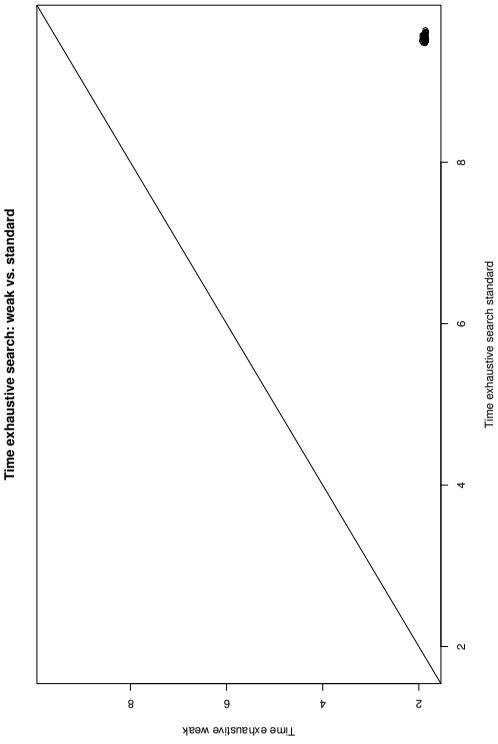


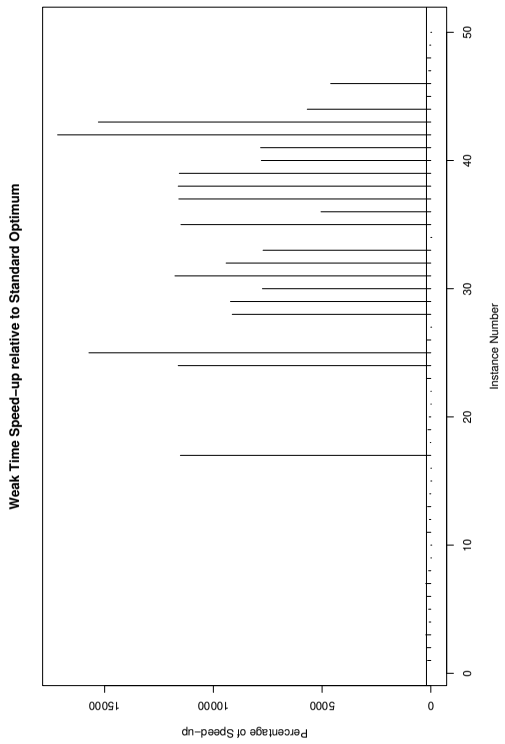


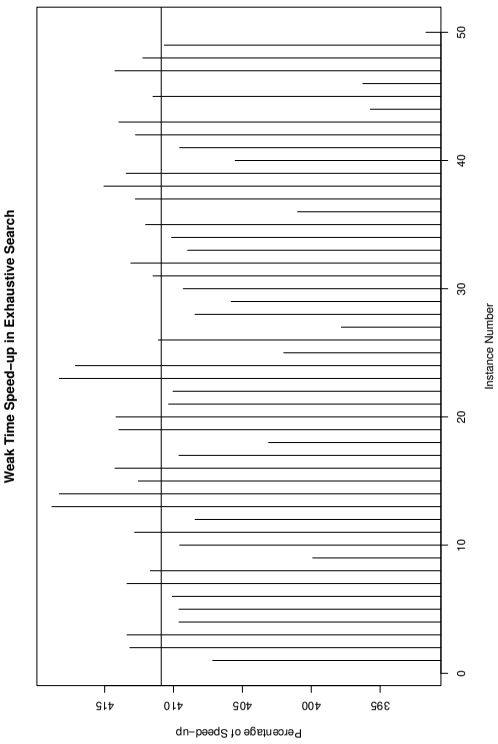


Instance Set 4 Columns Total Weight 60: Maximisation

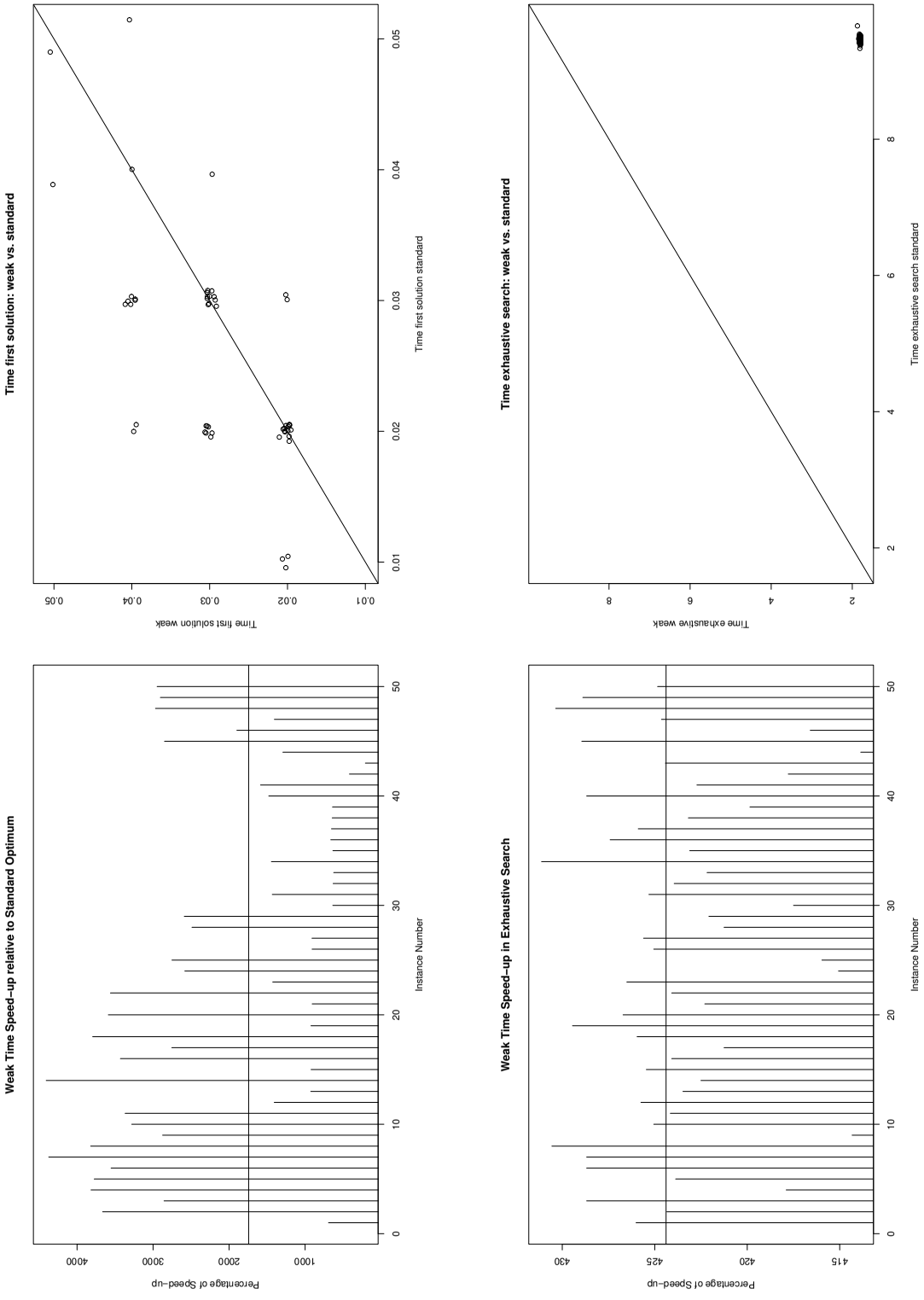




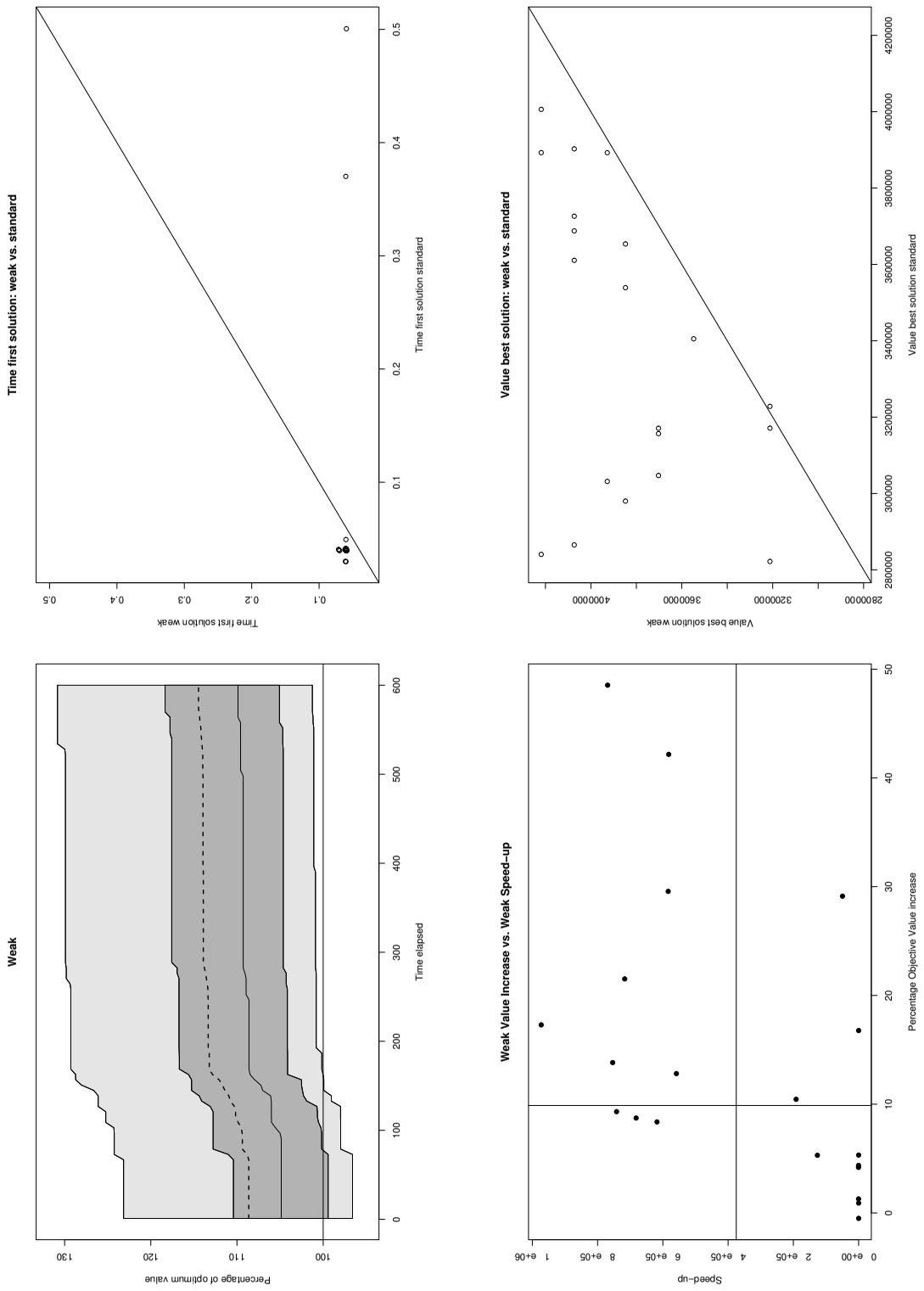




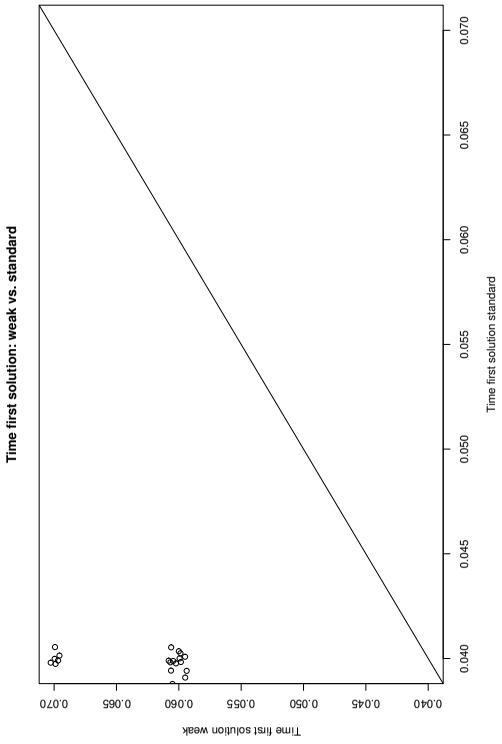
Instance Set 4 Columns Total Weight 60: Minimisation

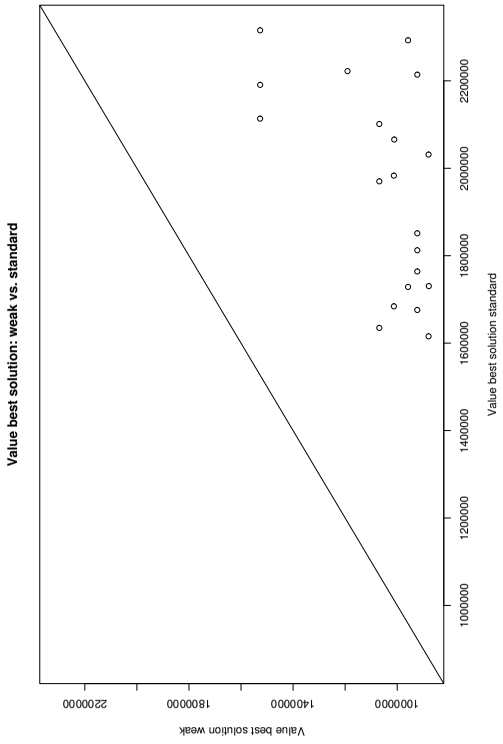


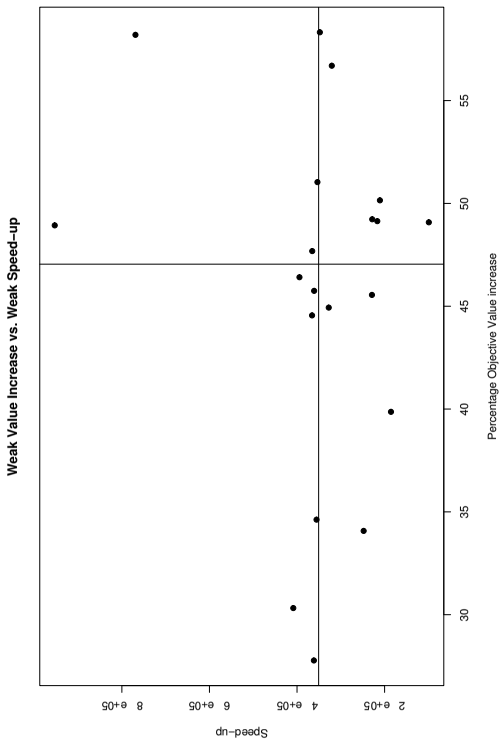
Instance Set 5 Columns Total Weight 15: Maximisation



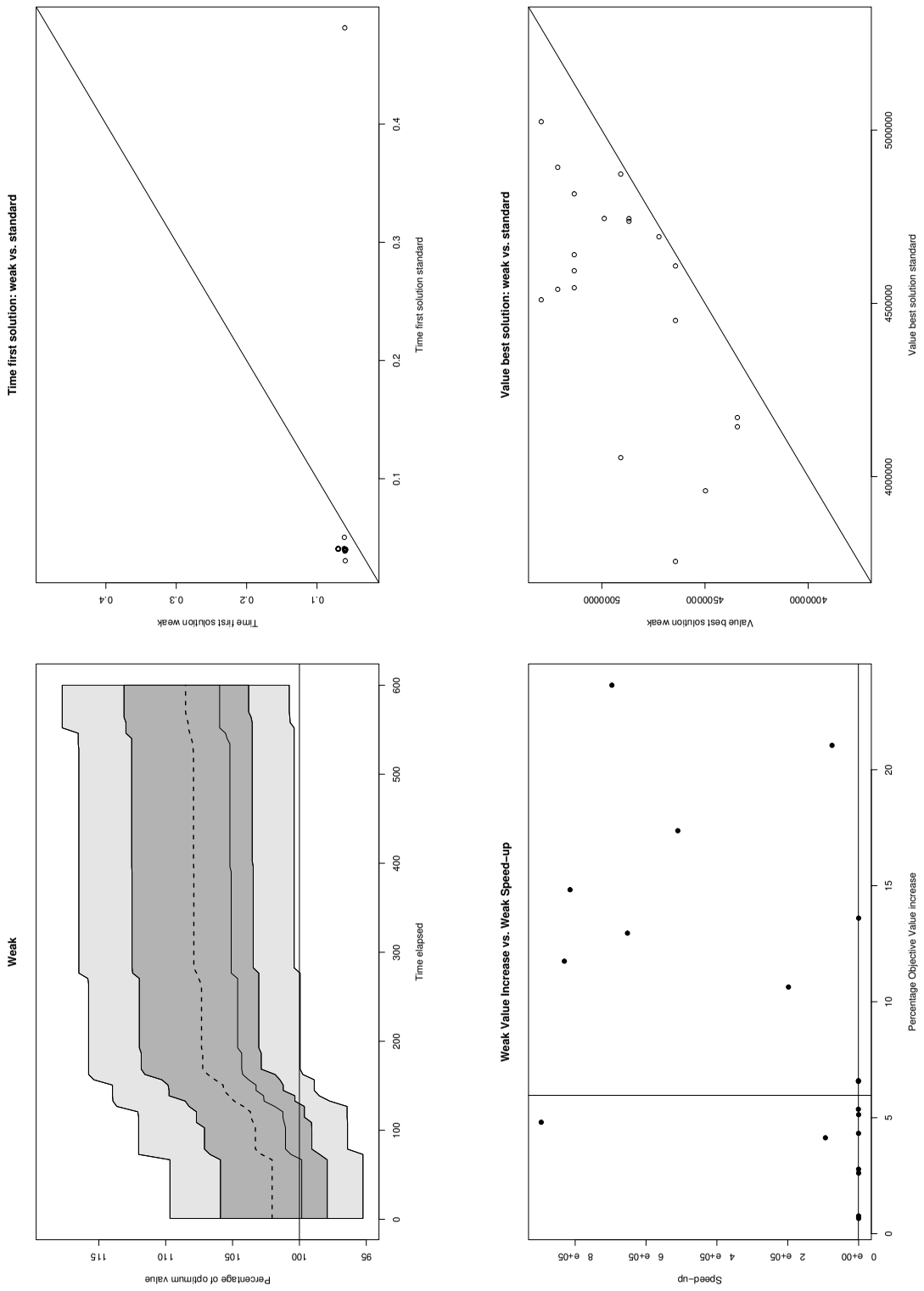
Instance Set 5 Columns Total Weight 15: Minimisation



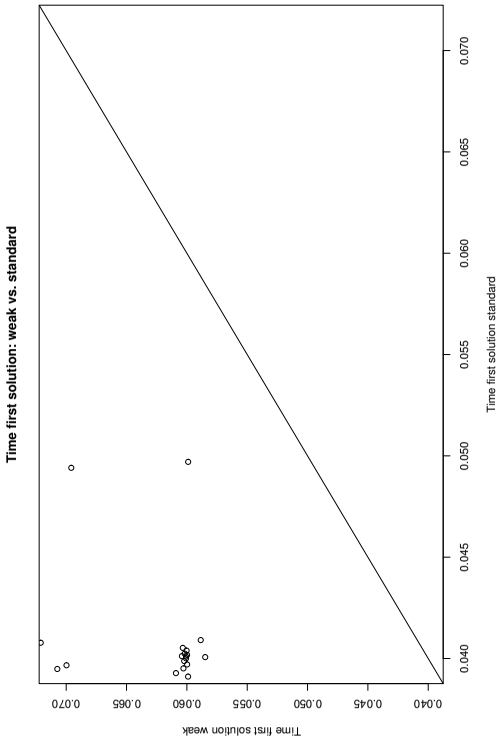


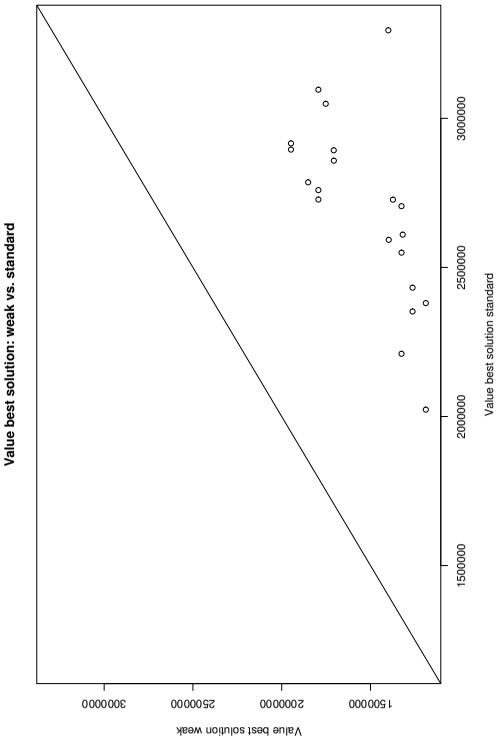


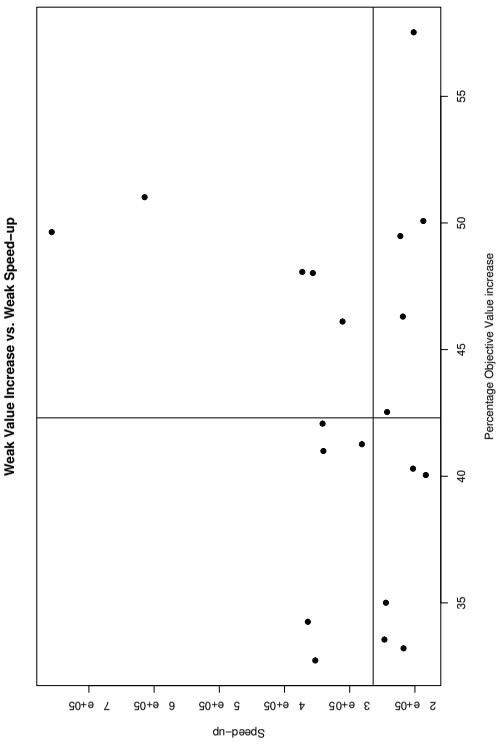
Instance Set 5 Columns Total Weight 20: Maximisation



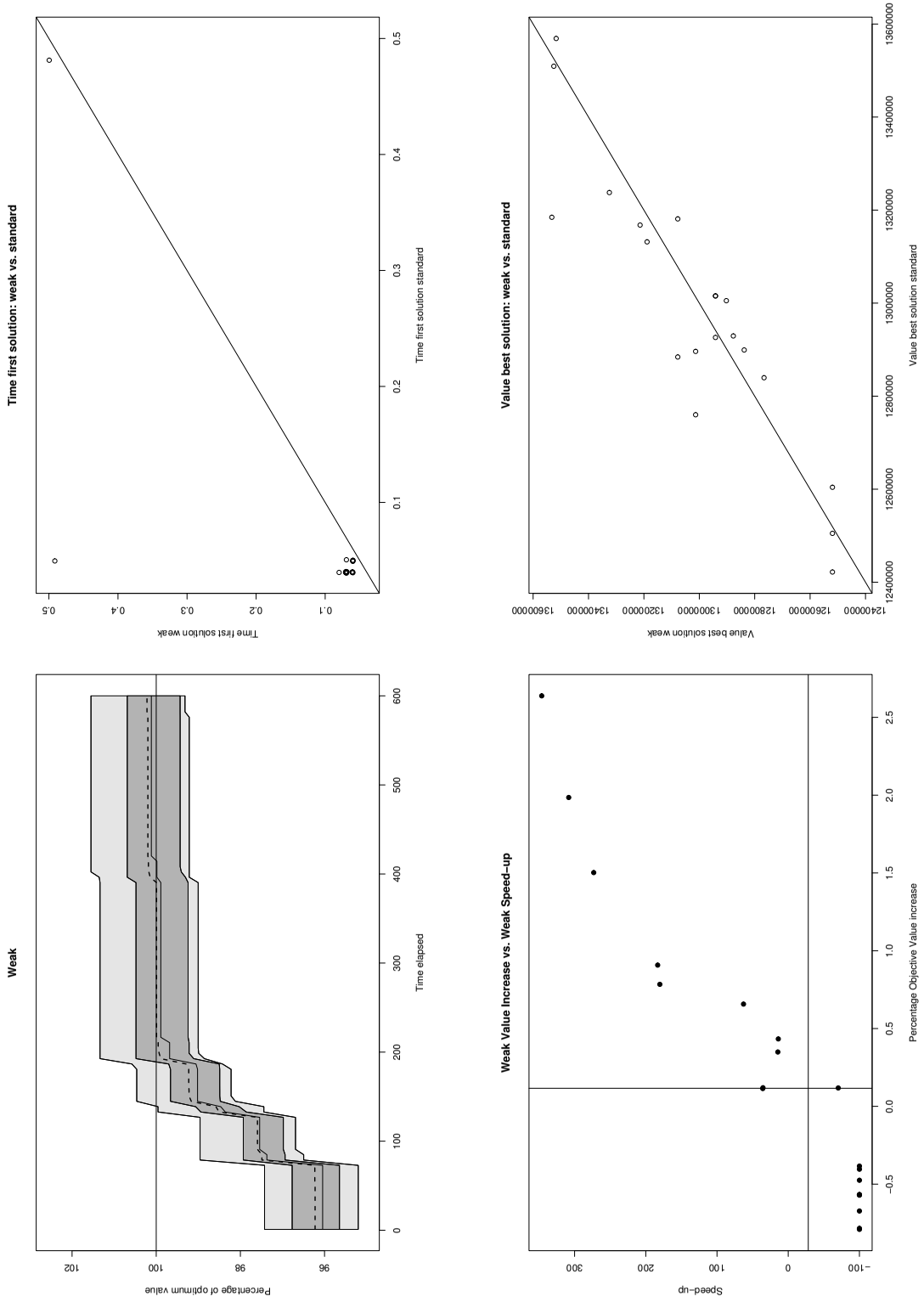
Instance Set 5 Columns Total Weight 20: Minimisation



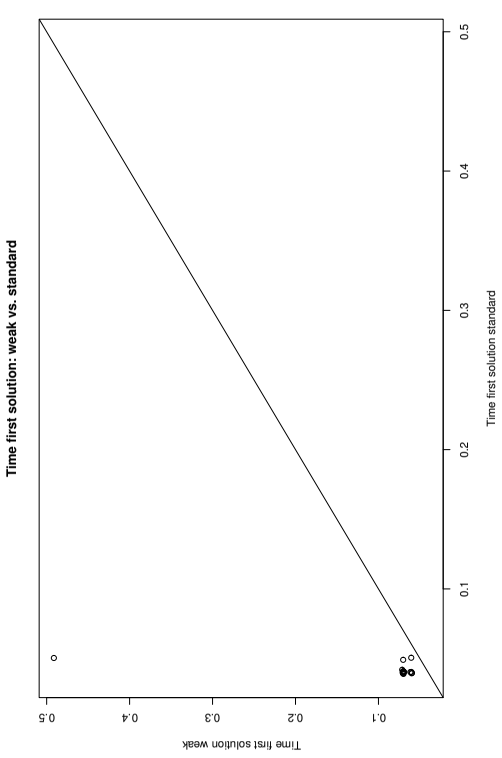


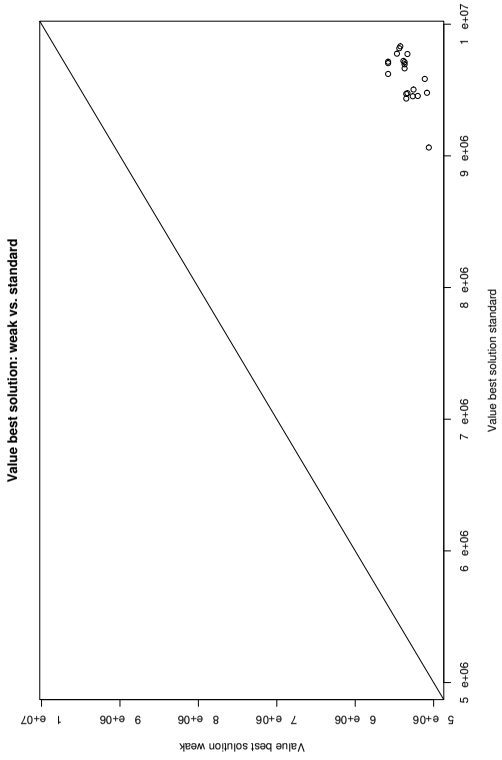


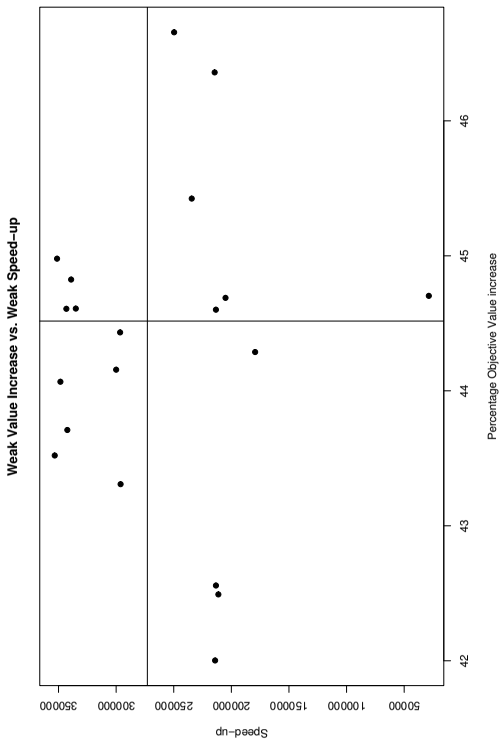
Instance Set 5 Columns Total Weight 60: Maximisation



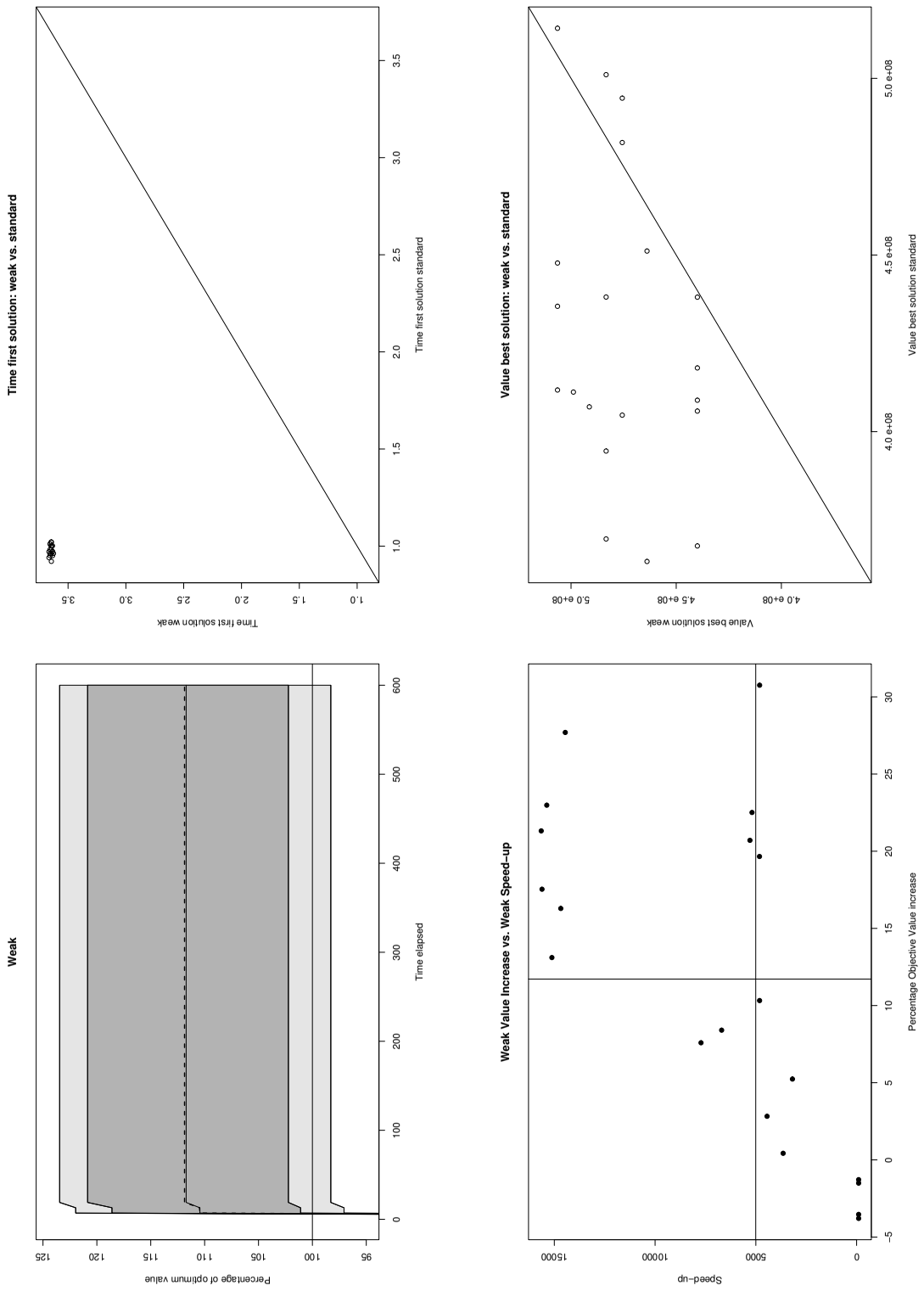
Instance Set 5 Columns Total Weight 60: Minimisation



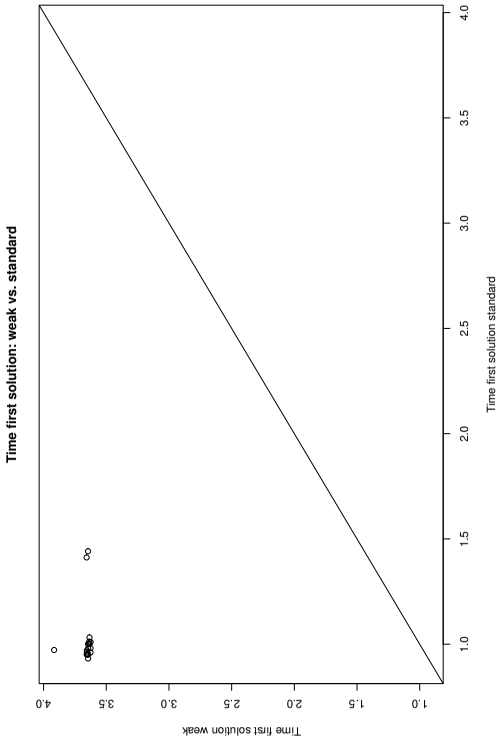


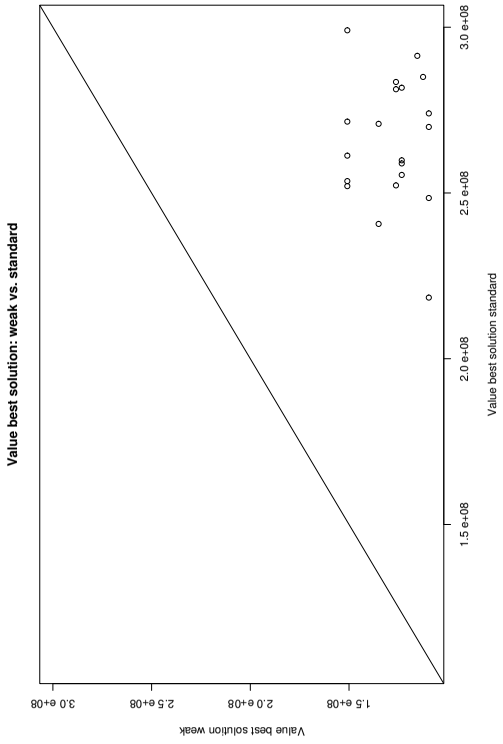


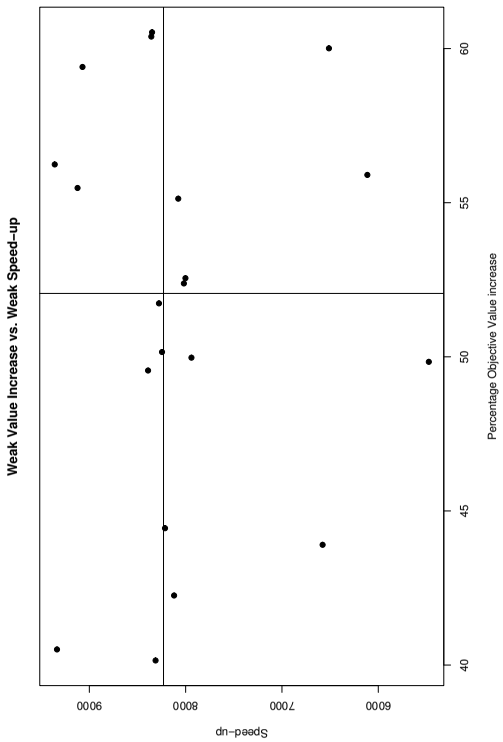
Instance Set 6 Columns Total Weight 20: Maximisation



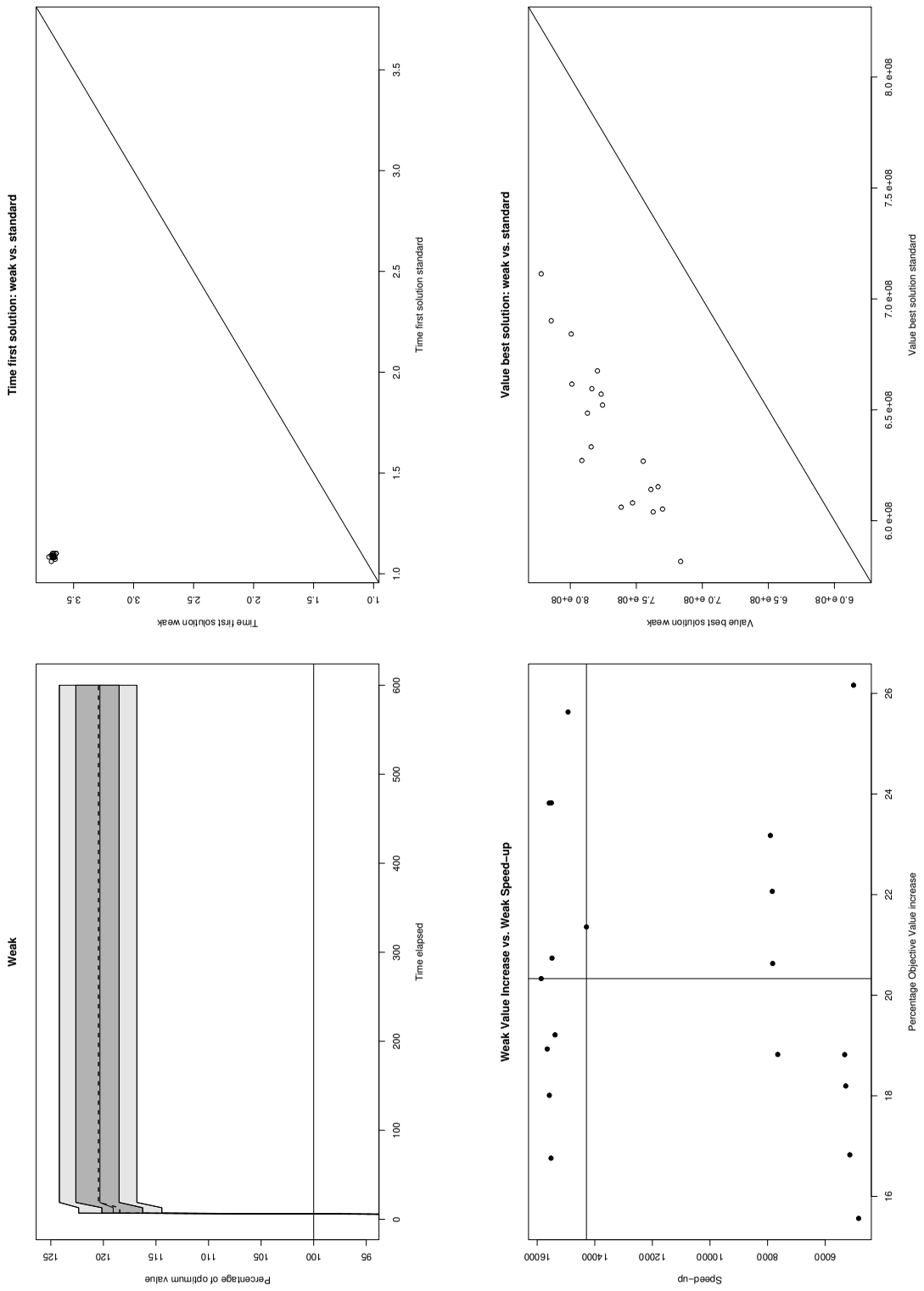
Instance Set 6 Columns Total Weight 20: Minimisation



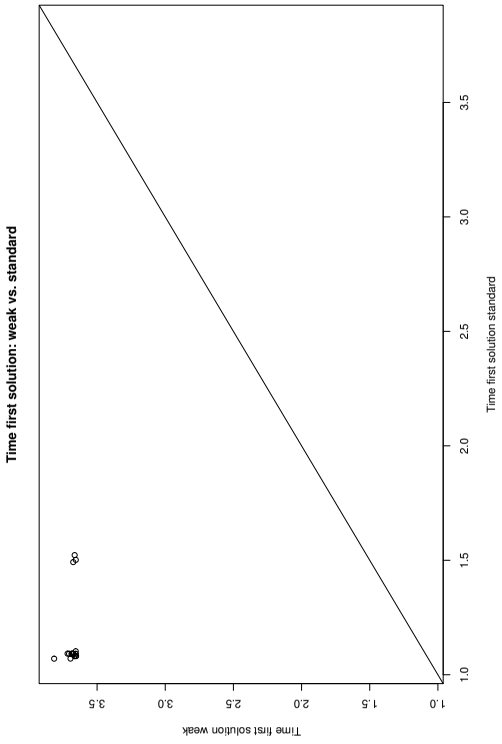


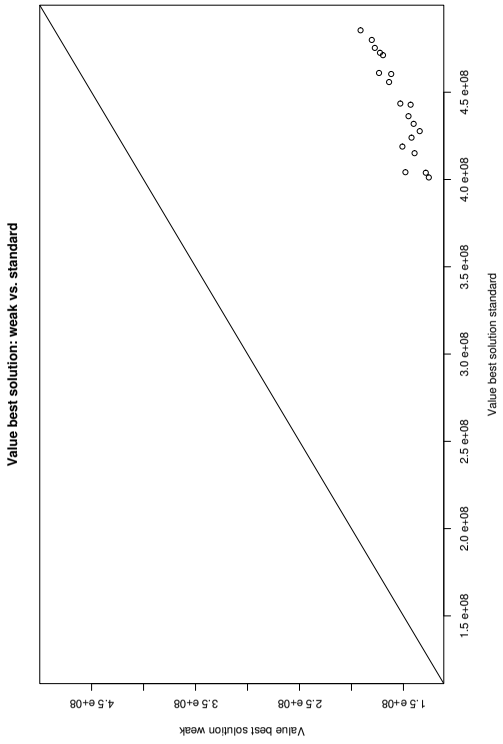


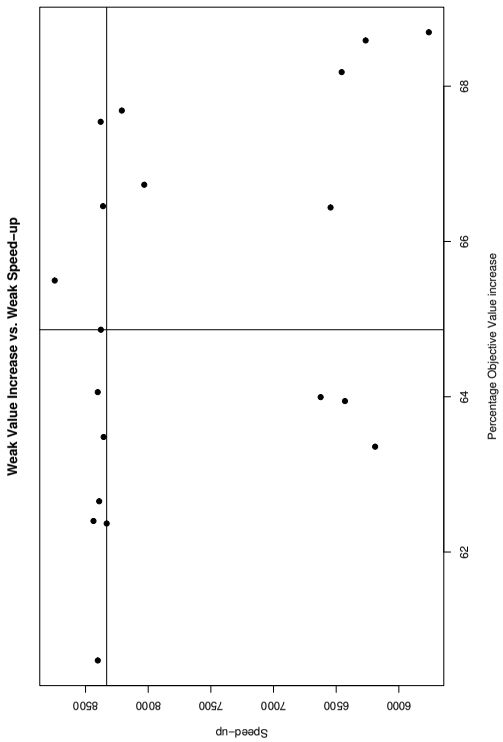
Instance Set 6 Columns Total Weight 30: Maximisation



Instance Set 6 Columns Total Weight 30: Minimisation

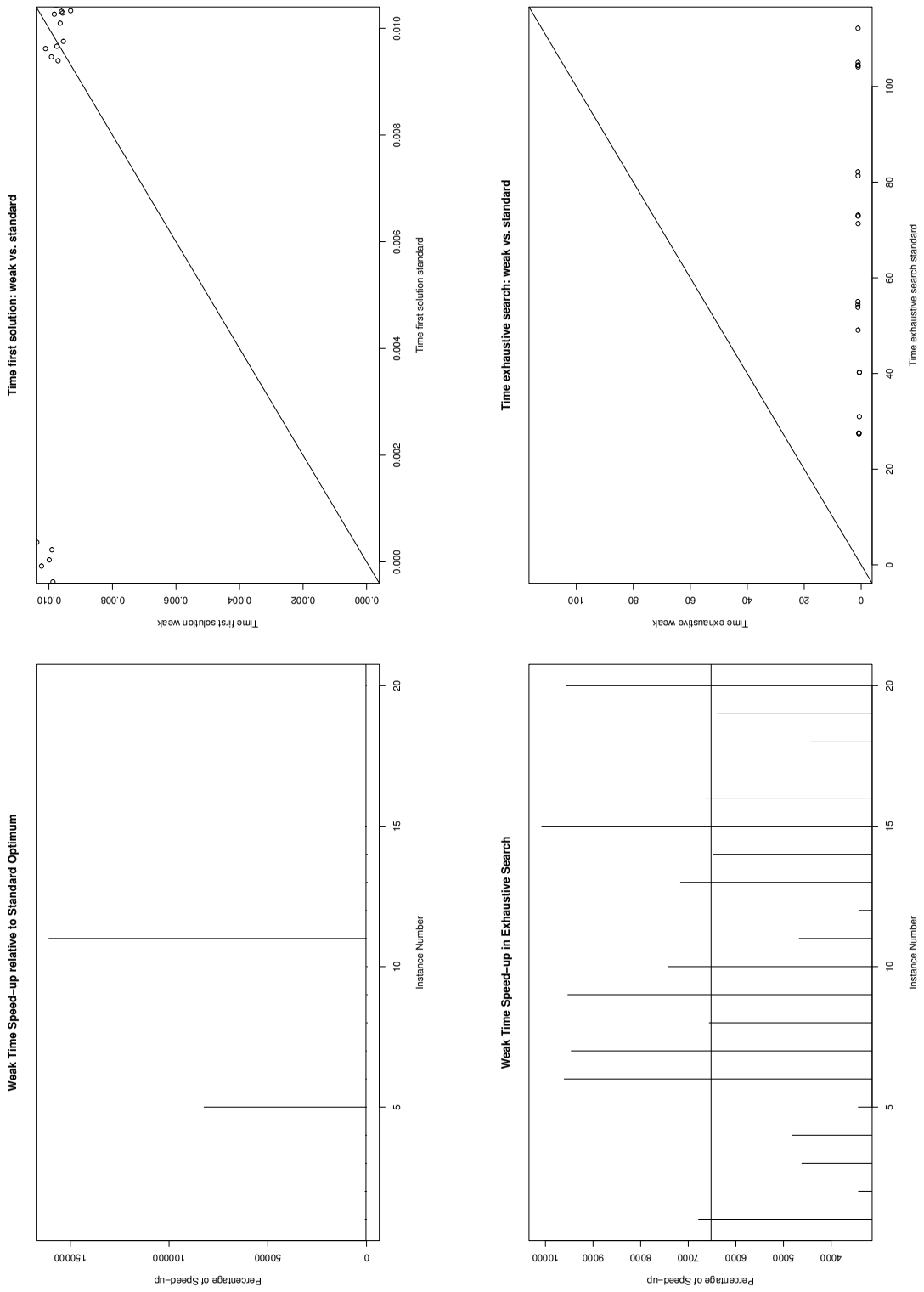




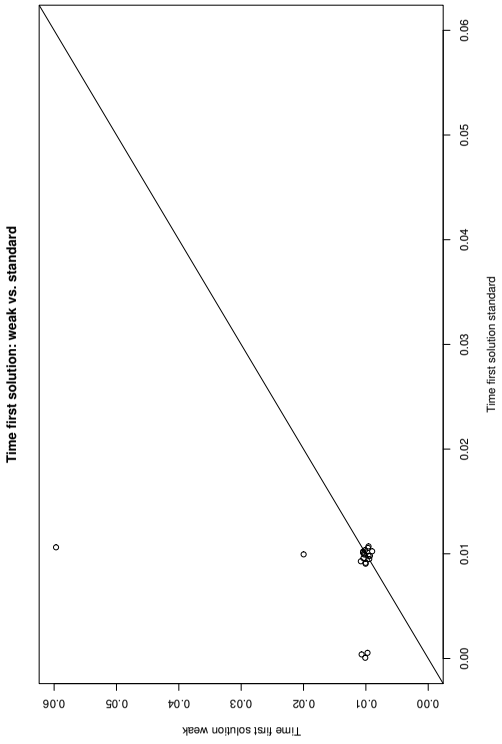


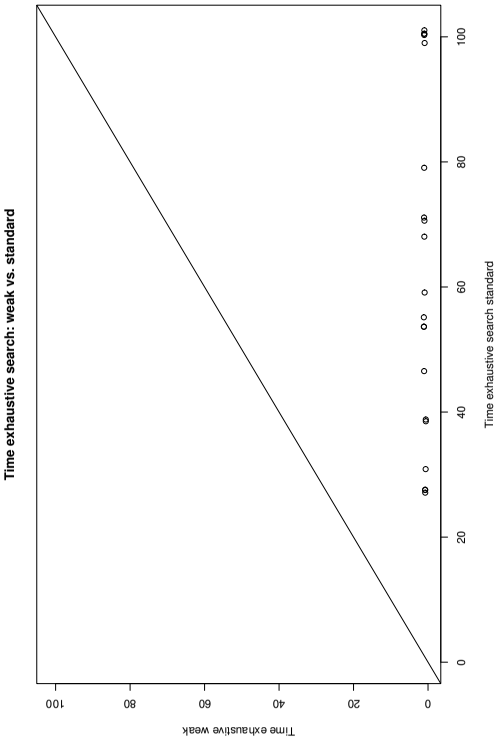
Scenario Weights on Columns (not pairwise different)

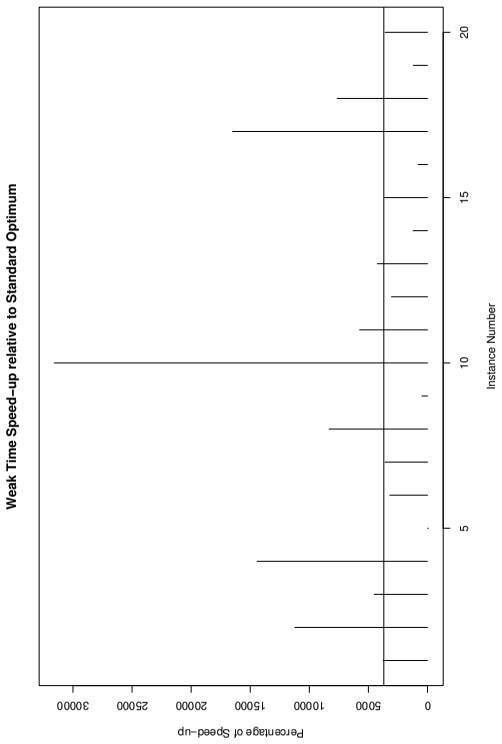
Instance Set 4 Columns: Maximisation

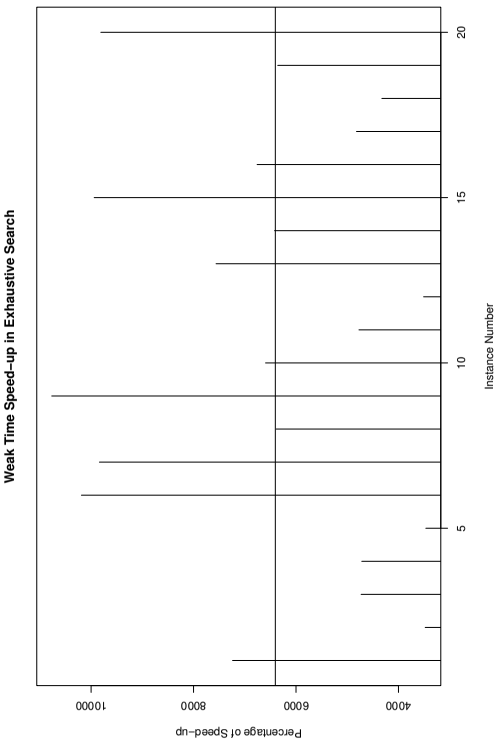


Instance Set 4 Columns: Minimisation

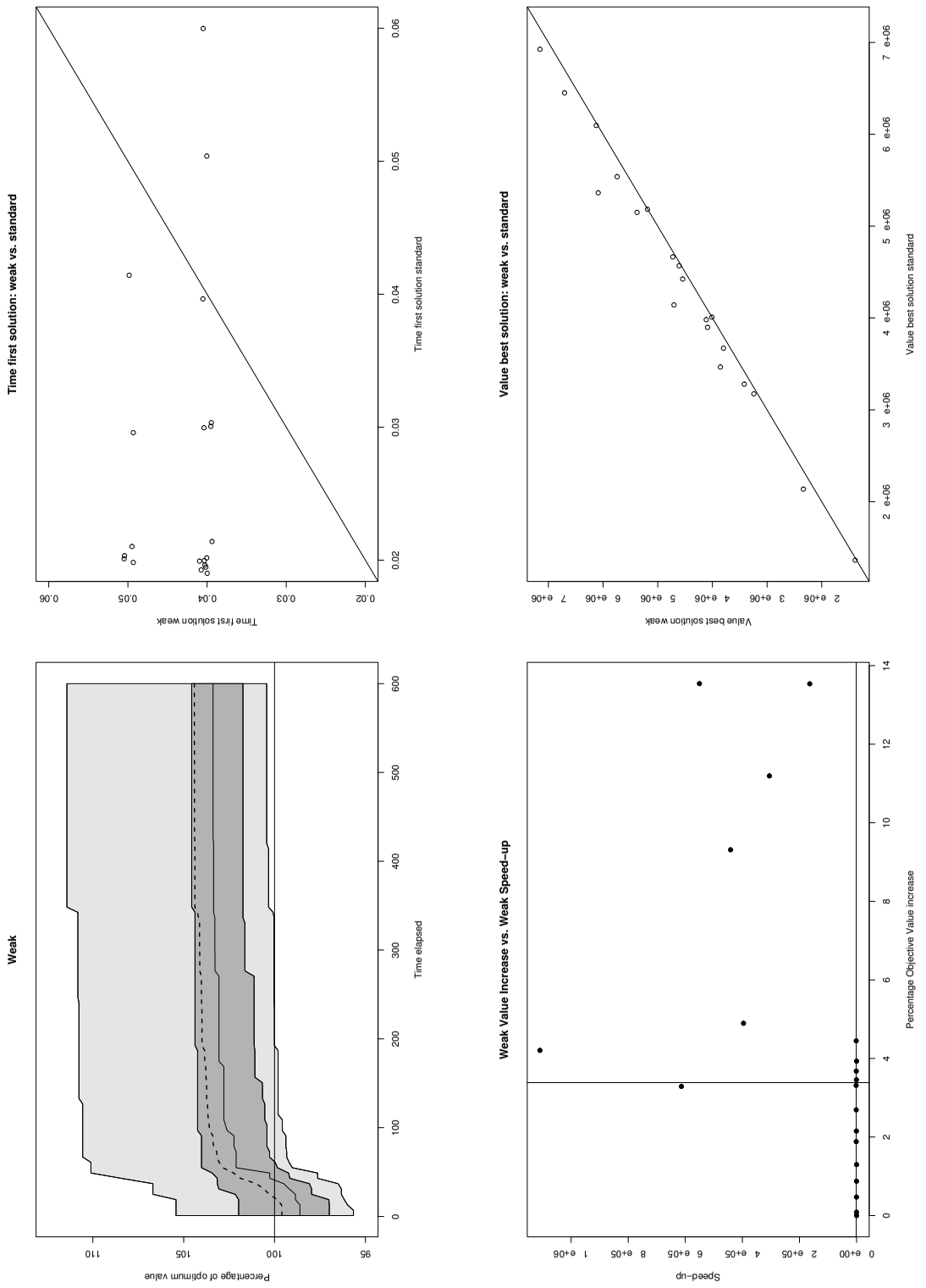




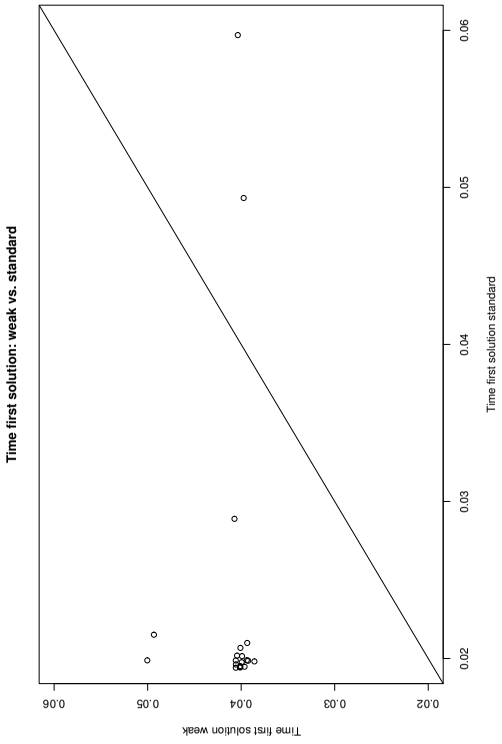


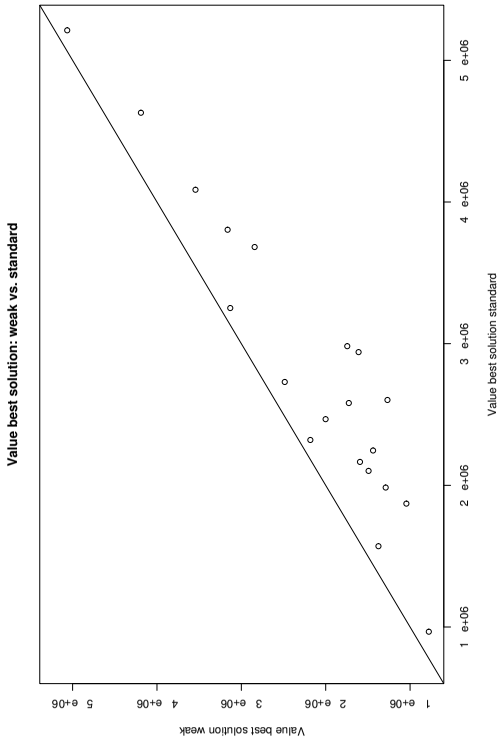


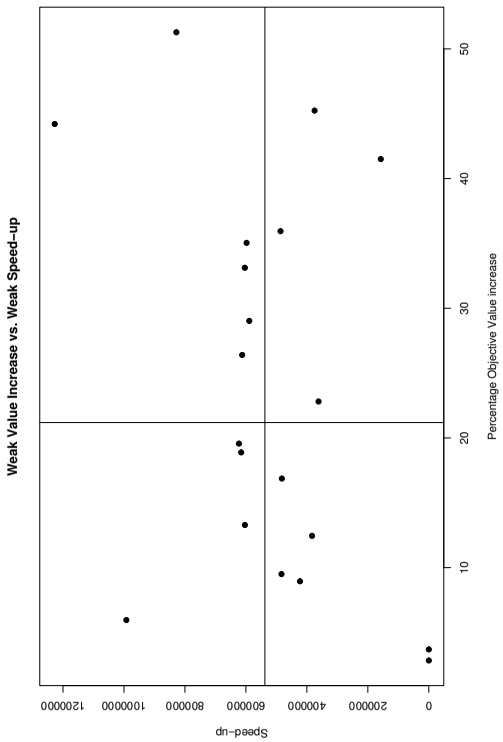
Instance Set 5 Columns: Maximisation



Instance Set 5 Columns: Minimisation

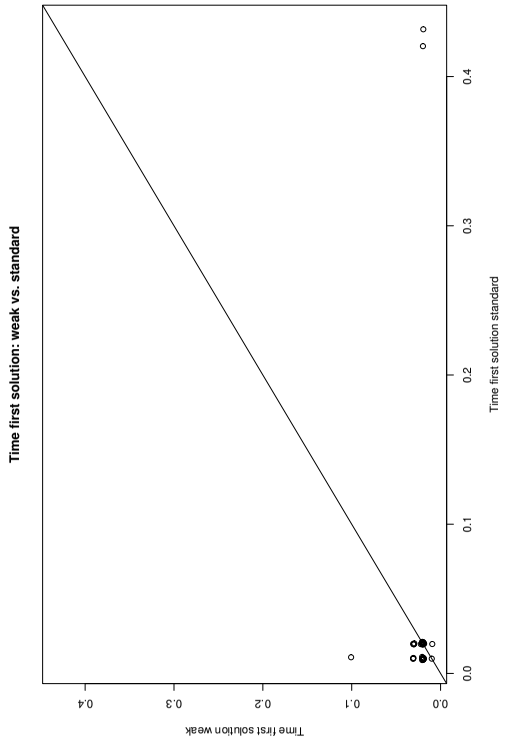


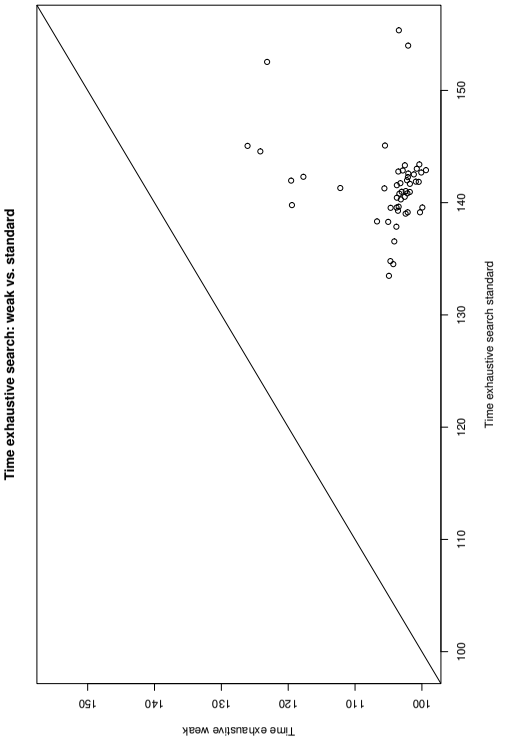


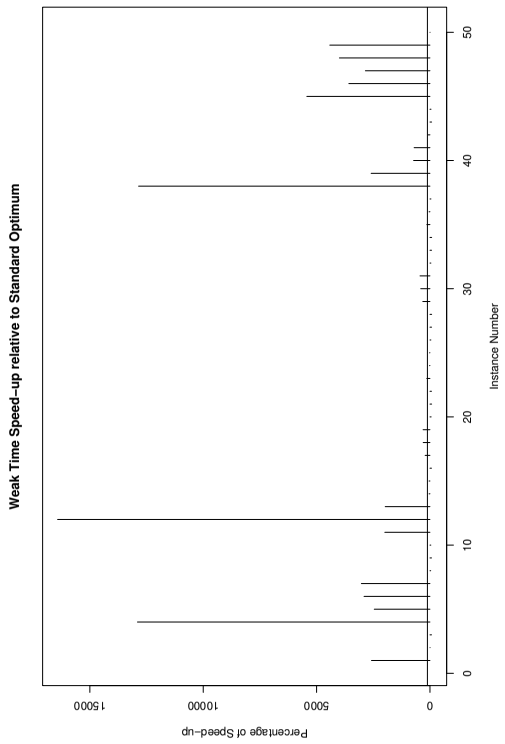


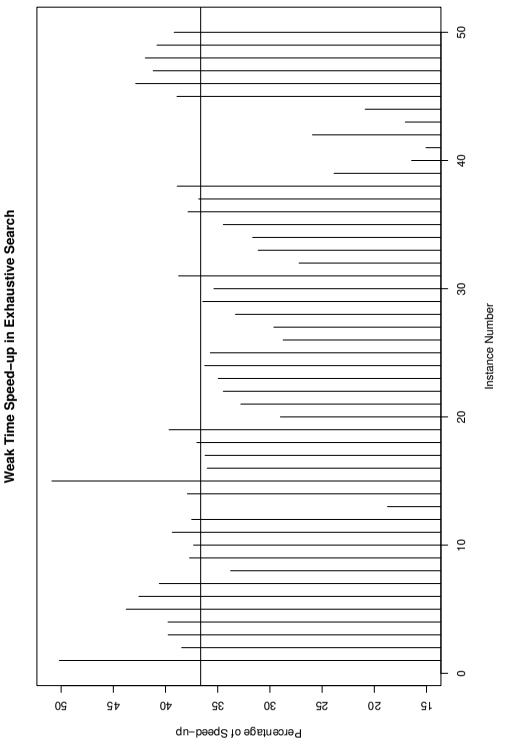
Scenario Weights on Cells

Instance Set 4 Columns: Maximisation

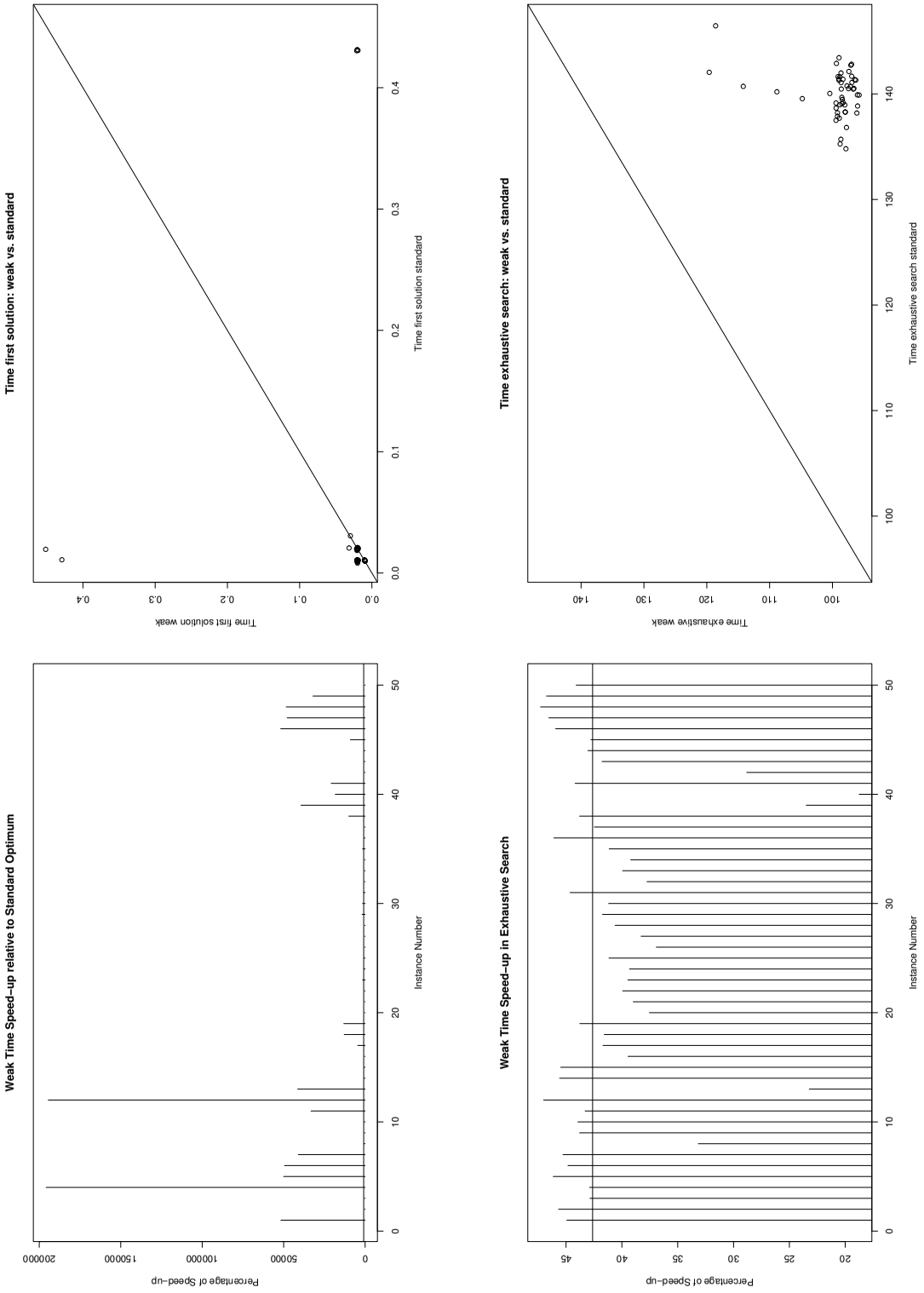




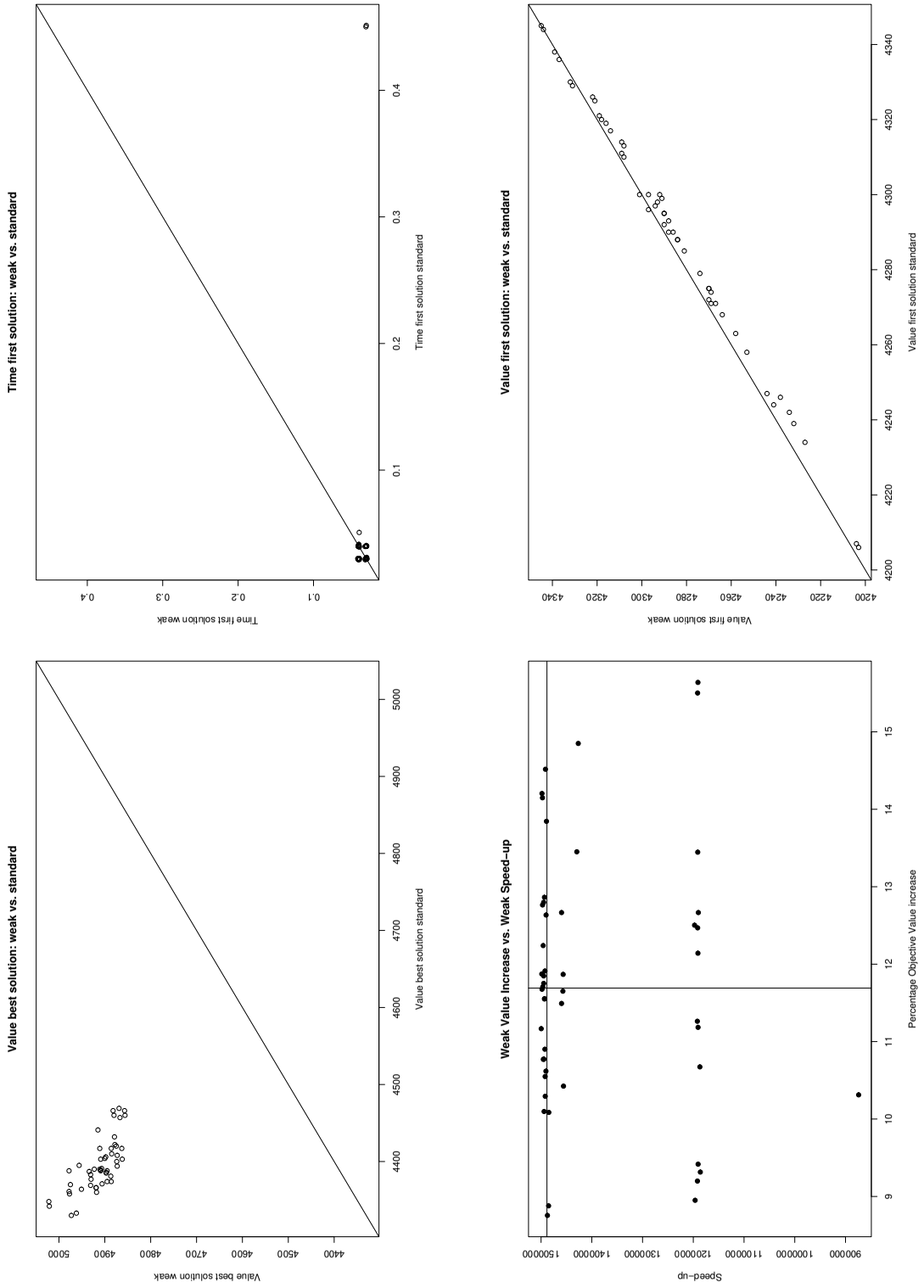




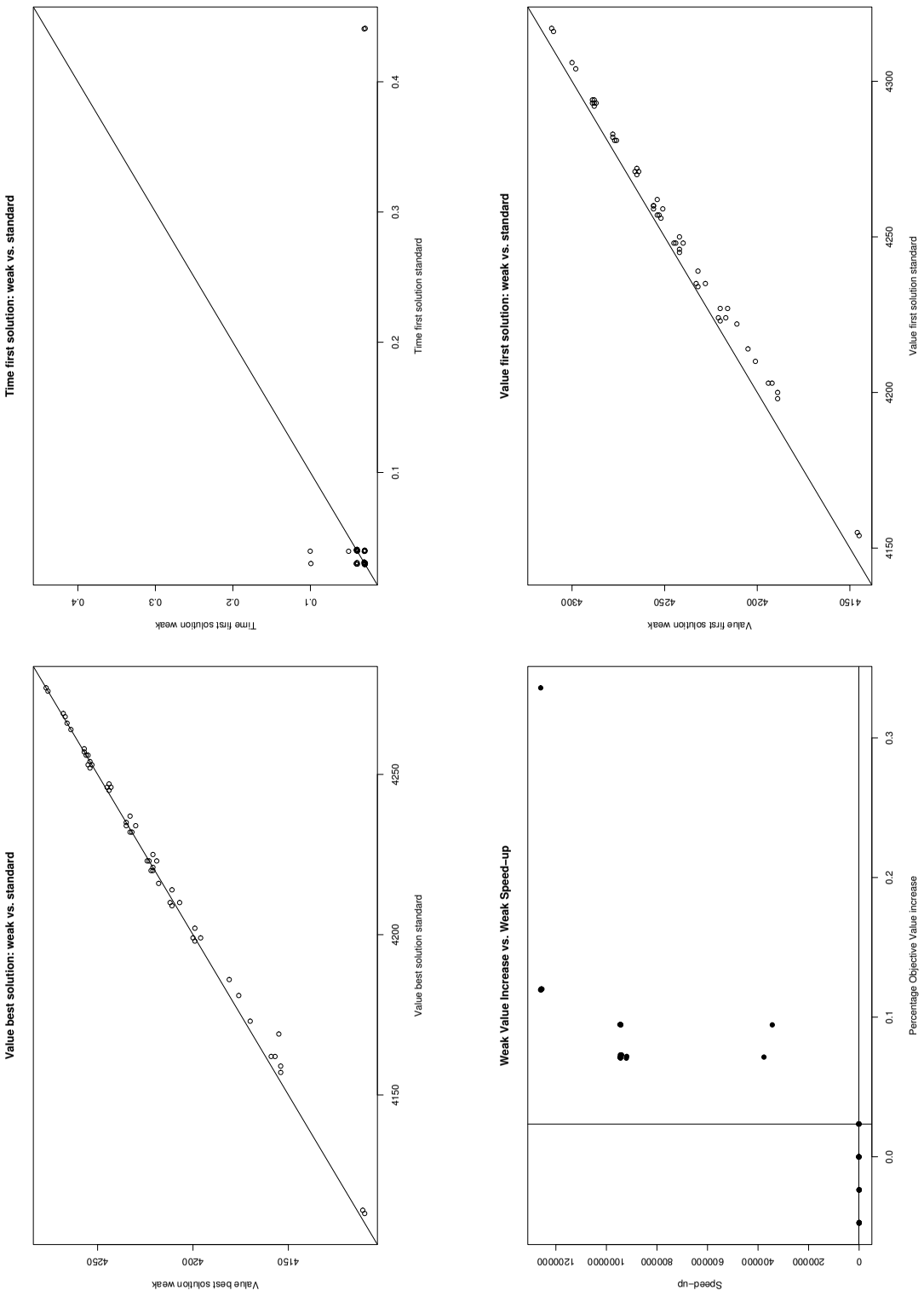
Instance Set 4 Columns Total Weight 7: Minimisation



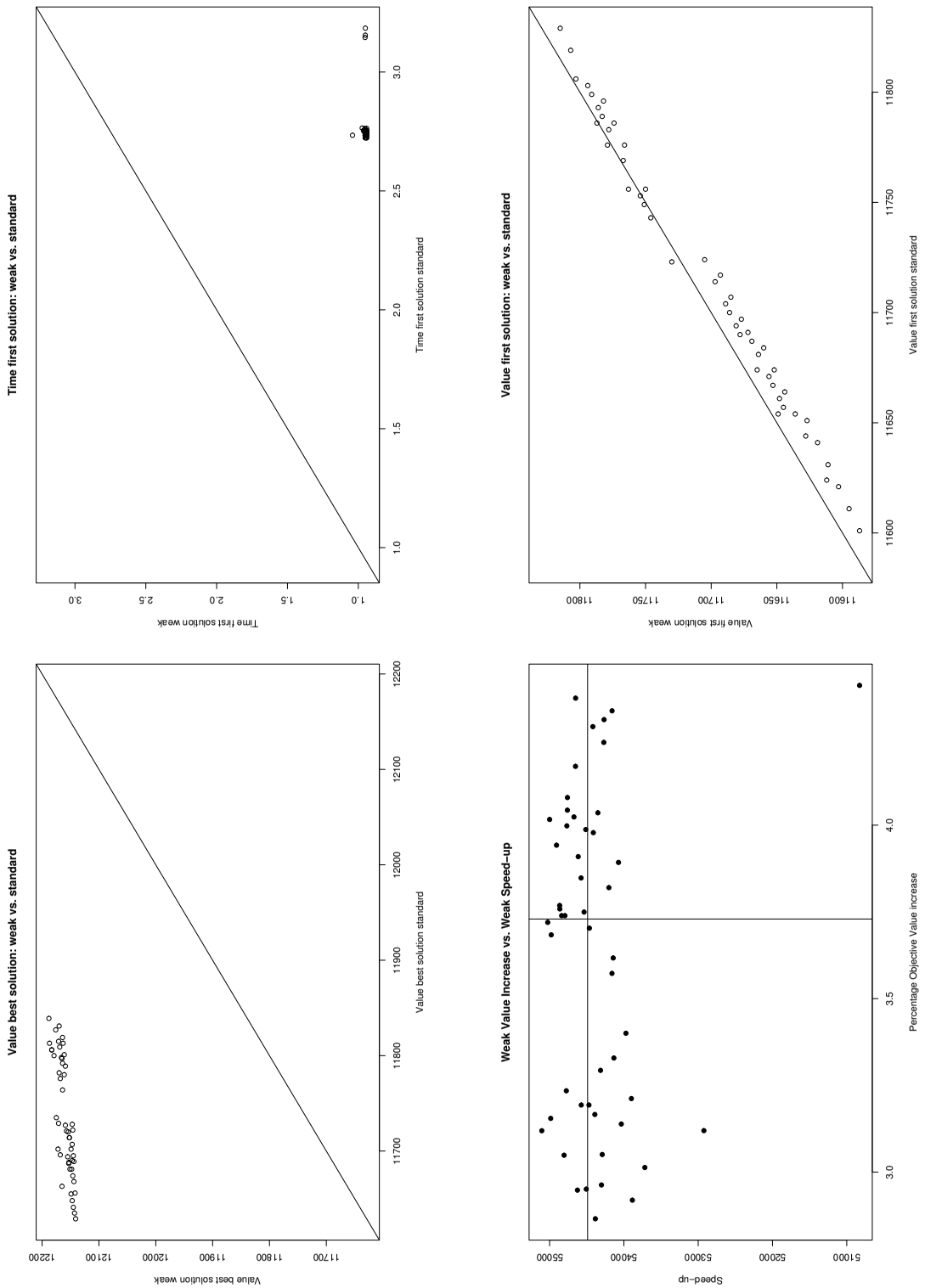
Instance Set 5 Columns: Maximisation



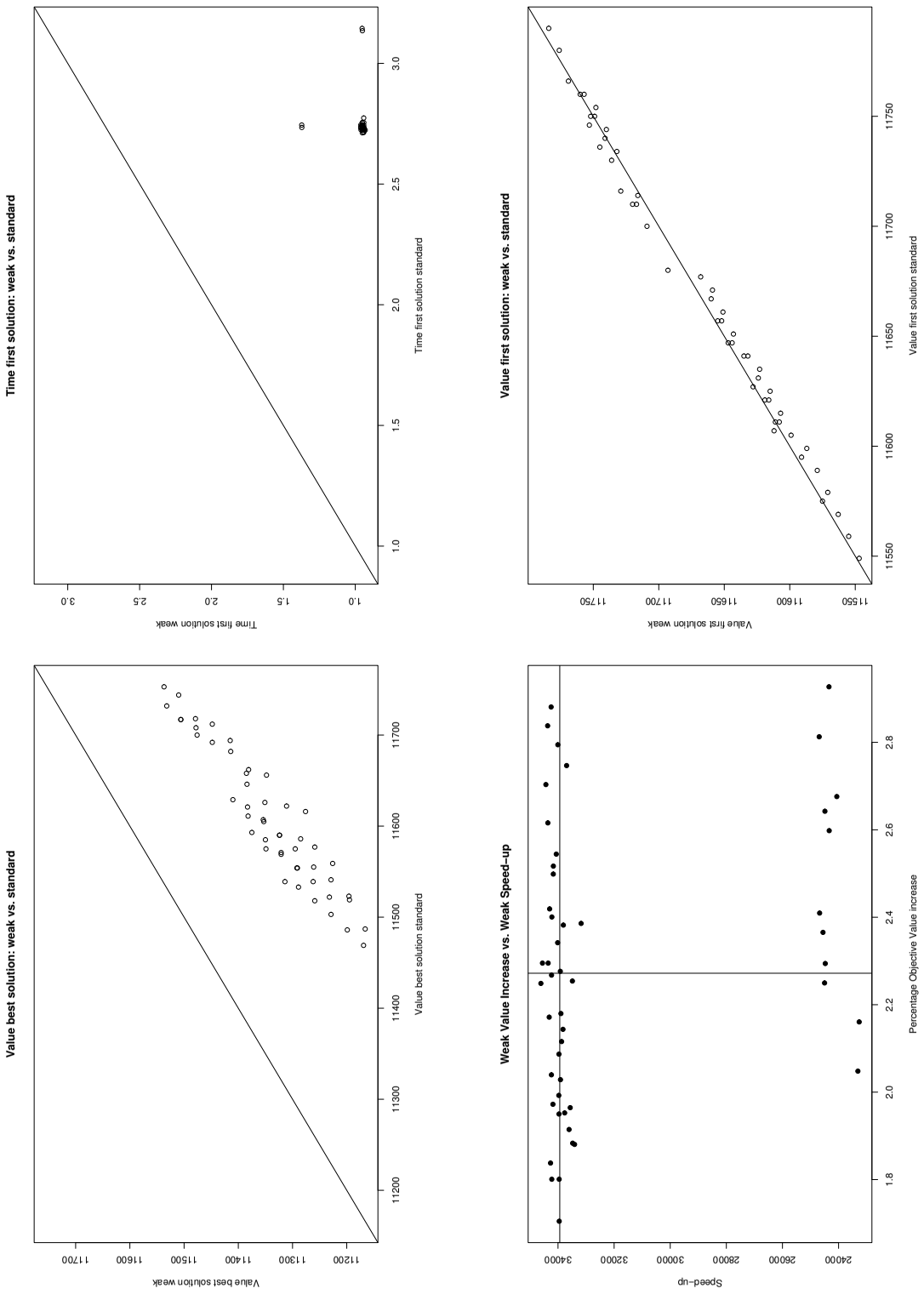
Instance Set 5 Columns: Minimisation



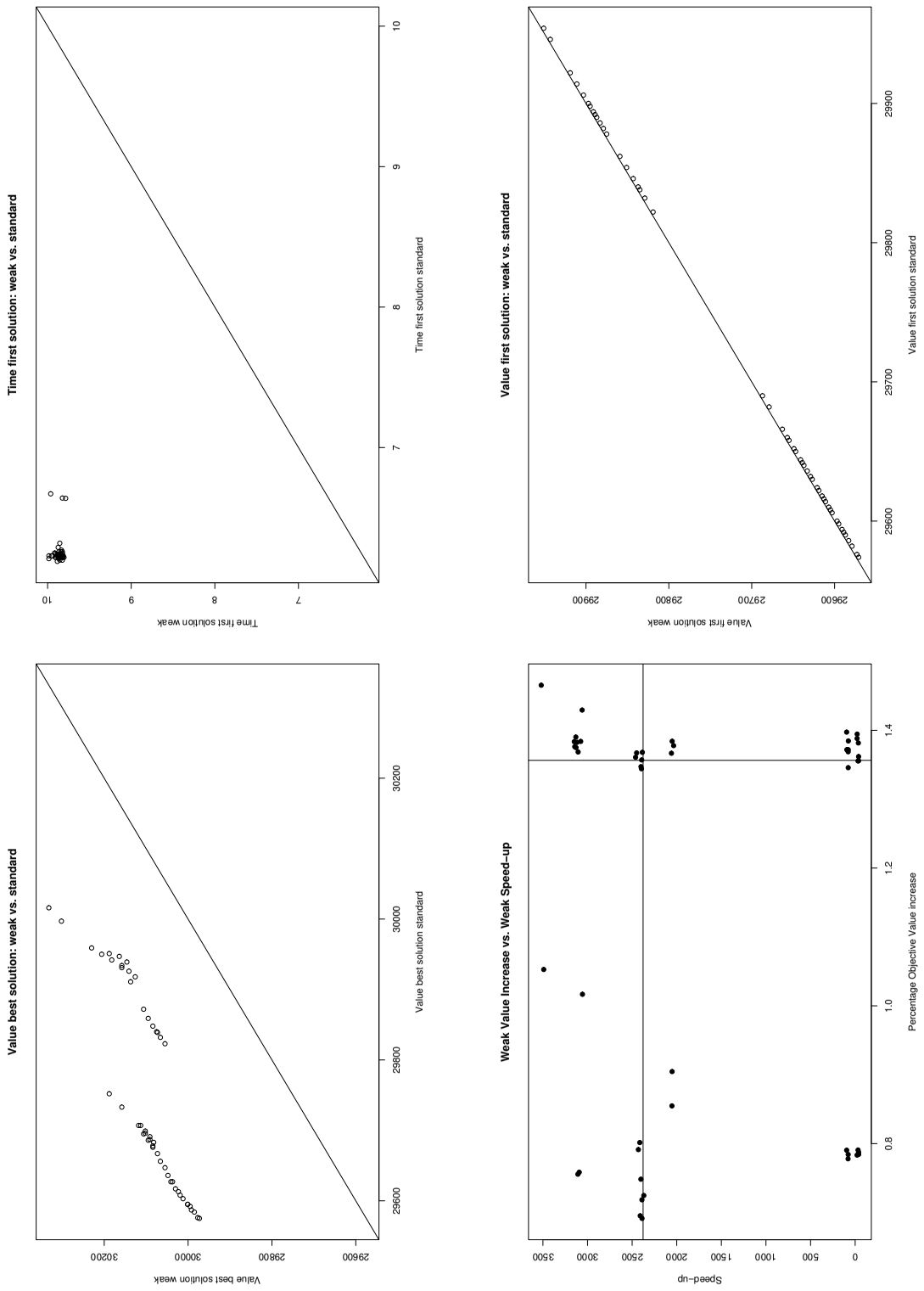
Instance Set 6 Columns: Maximisation



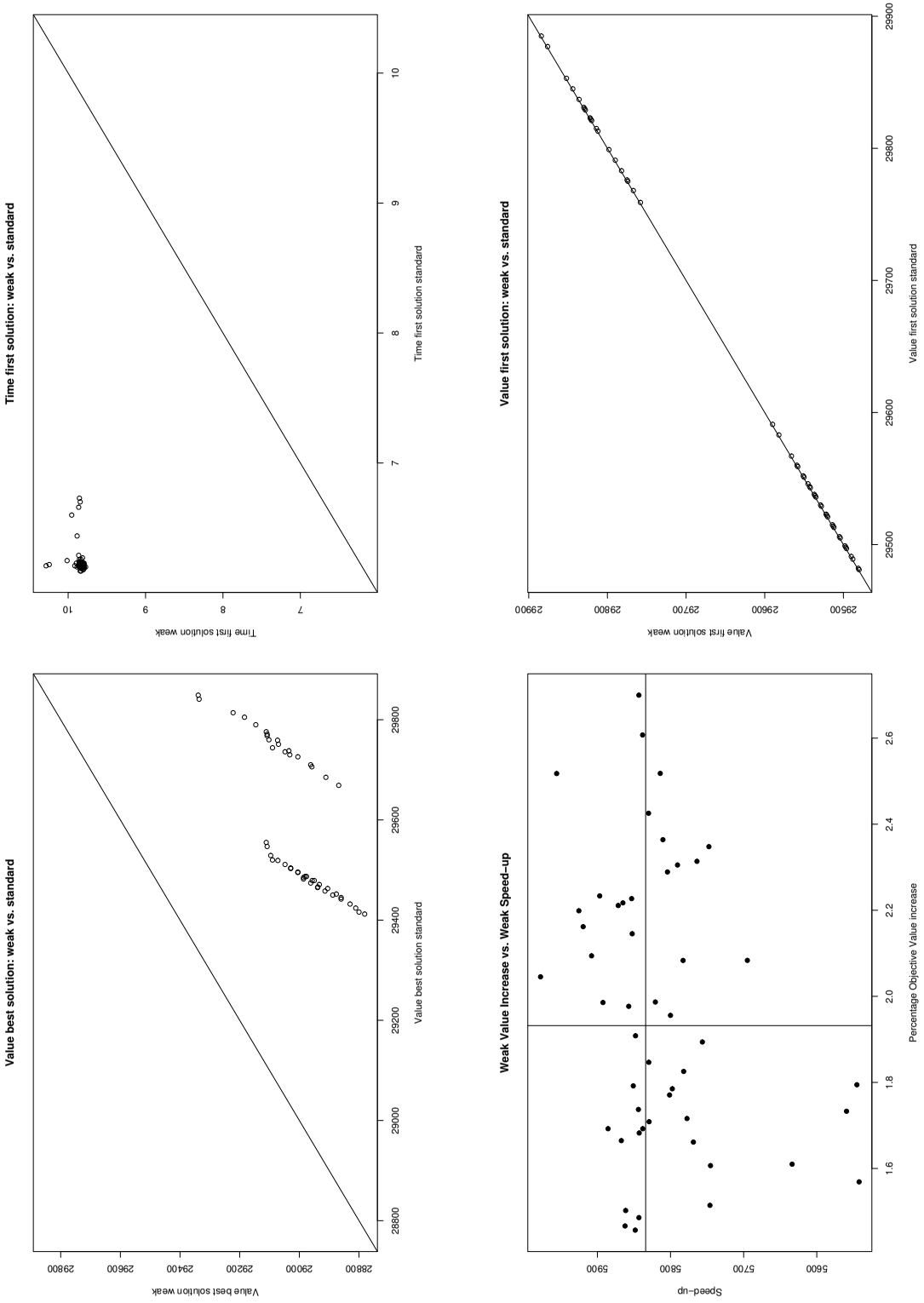
Instance Set 6 Columns: Minimisation



Instance Set 7 Columns: Maximisation

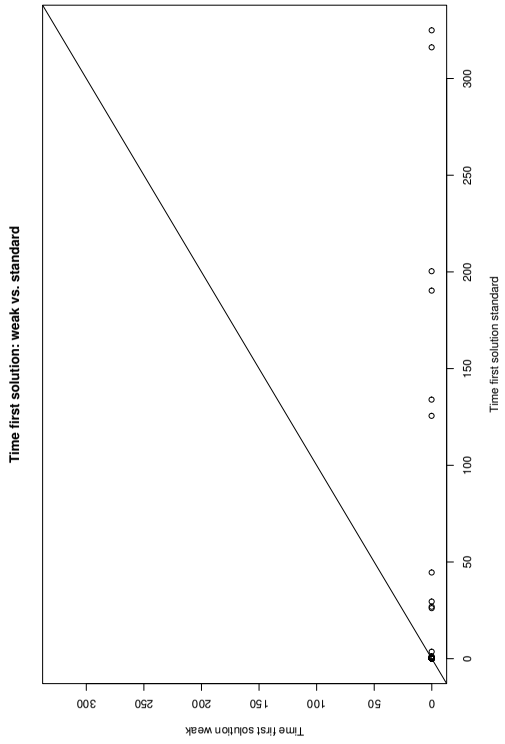


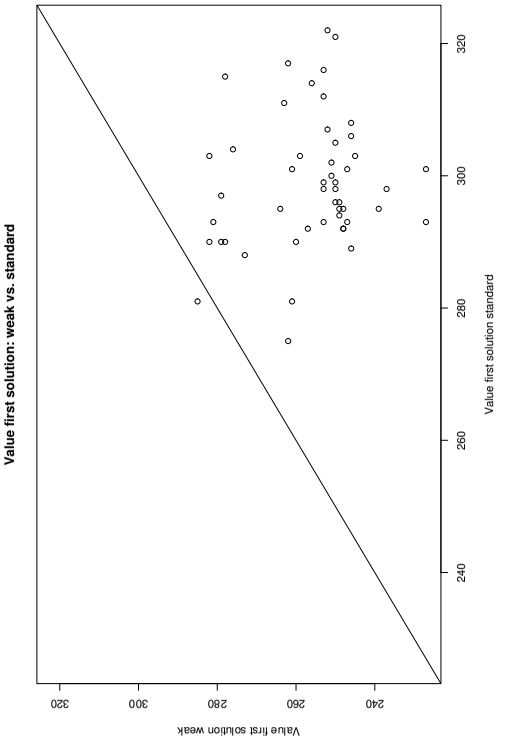
Instance Set 7 Columns: Minimisation

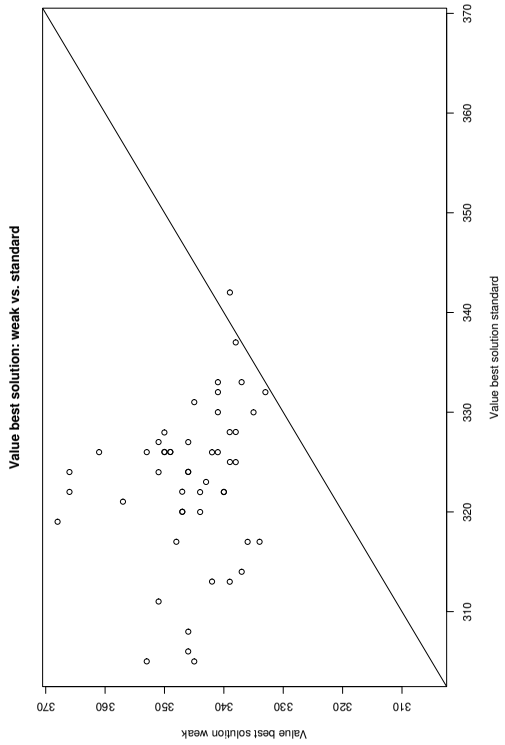


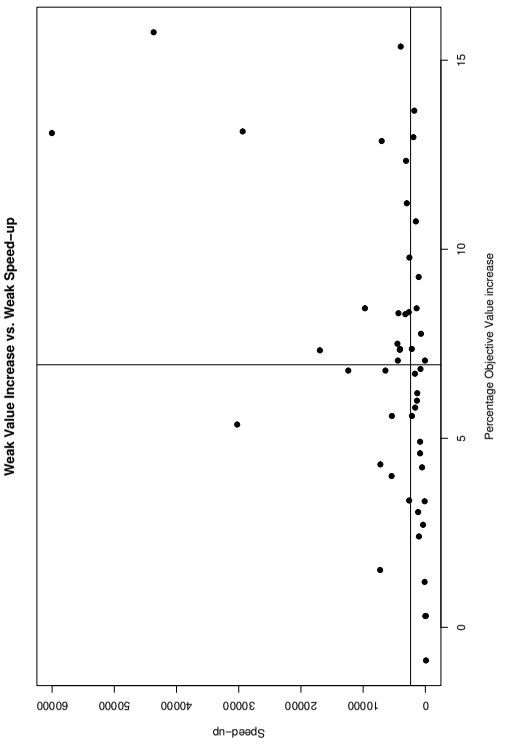
Automated Manufacturing

Instance Set 6×10

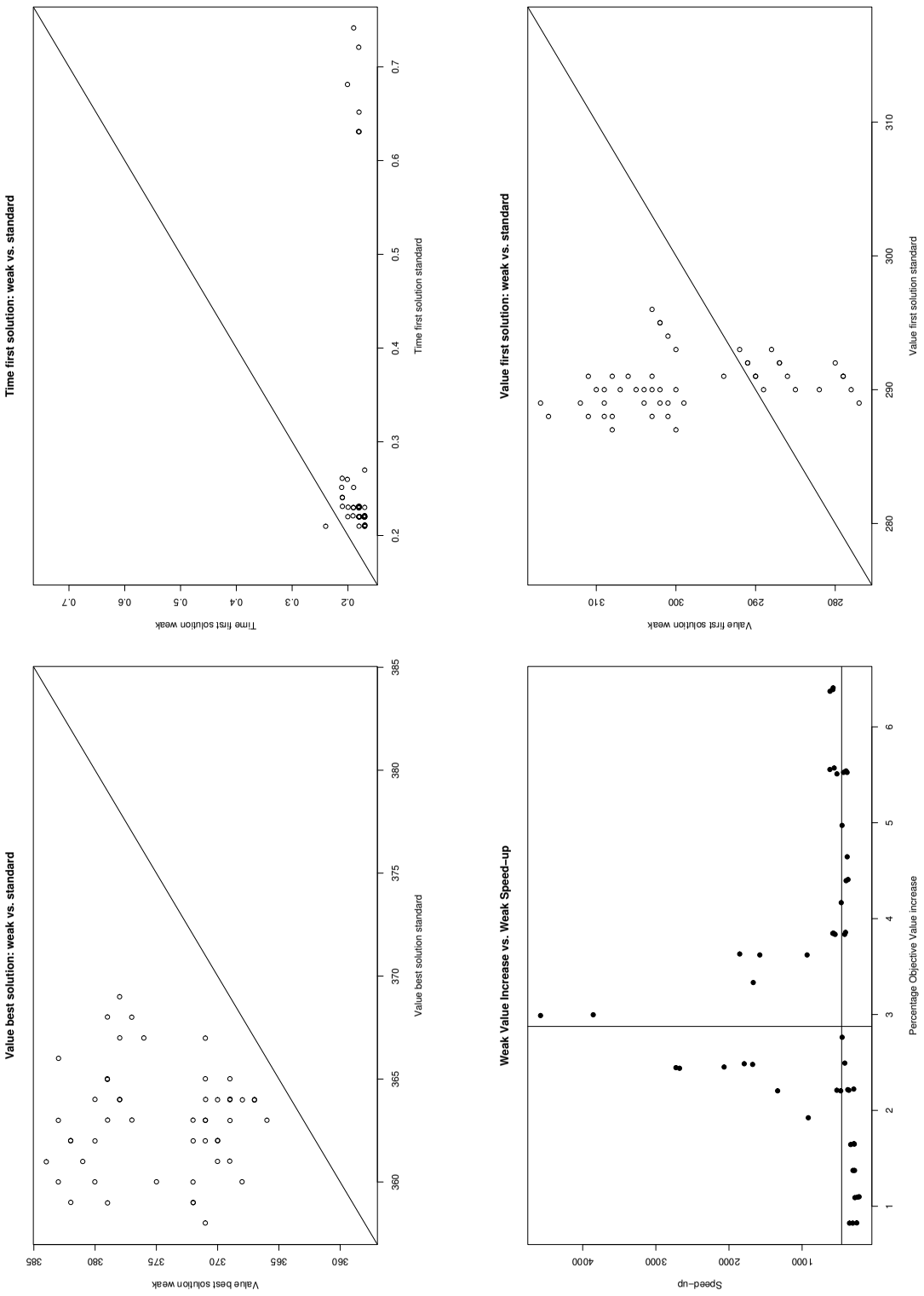




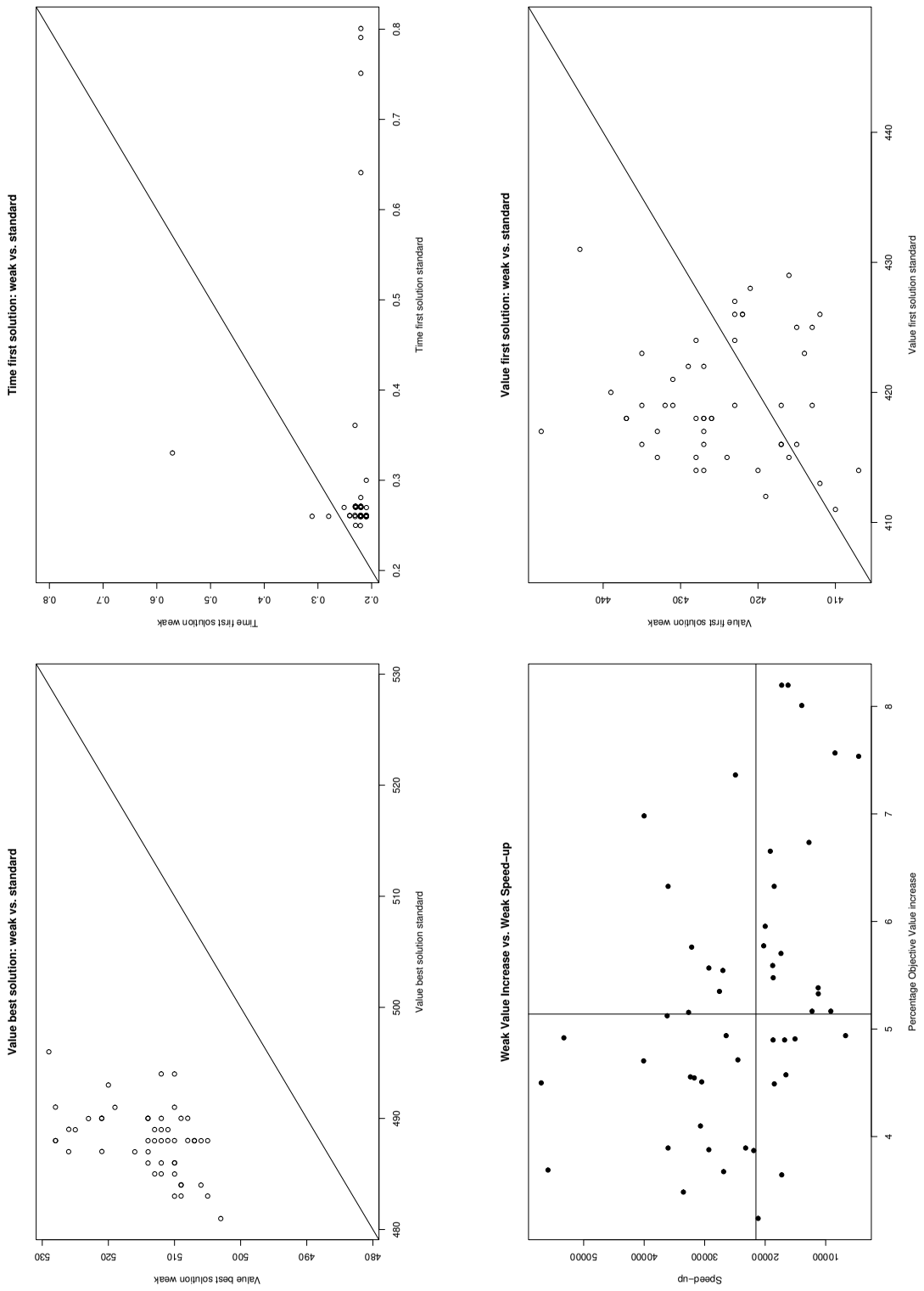




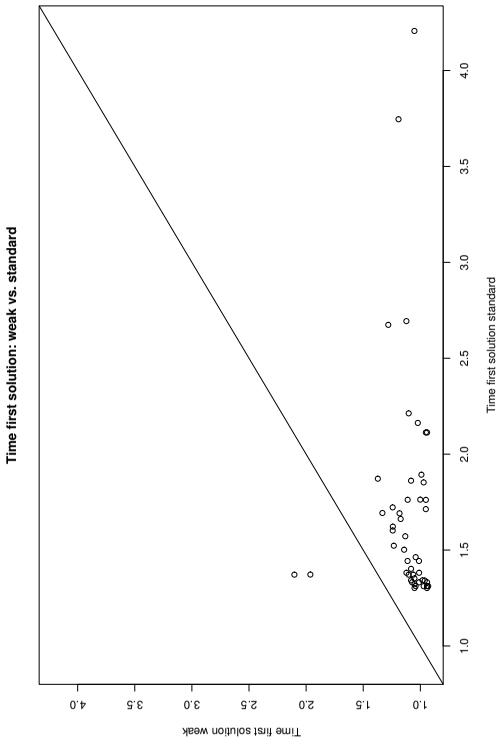
Instance Set 6×12

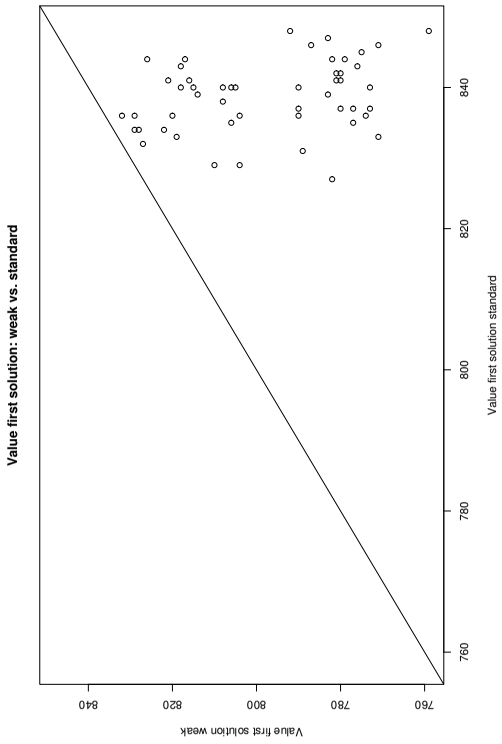


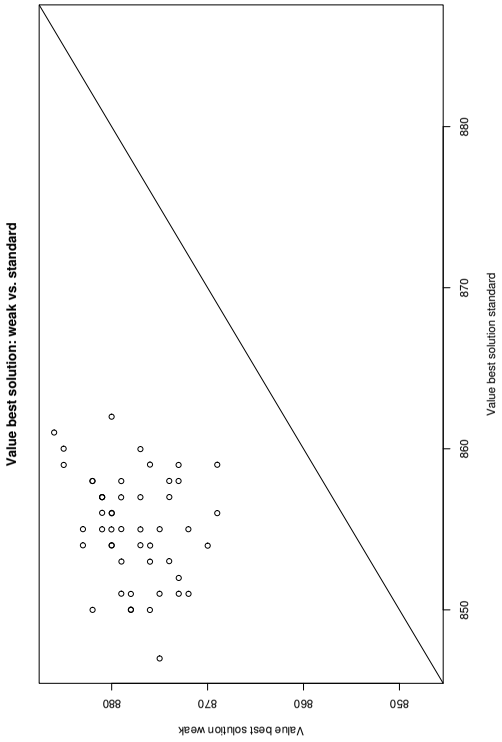
Instance Set 8×12

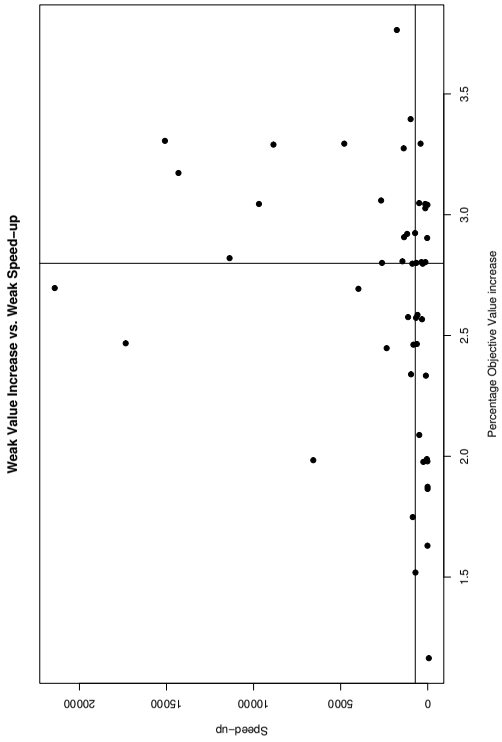


Instance Set 8×20









Bibliography

[1] SymNet Workshop on Almost-Symmetry in Search, 2005.

[2] Conference on Principles and Practice of Constraint Programming, 2006.

[3] International Workshop on Symmetry and Constraint Satisfaction Problems, 2006.

[4] International Symmetry Conference, 2007.

[5] K. Apt. *Principles of Constraint Programming.* Cambridge University Press, 2003.

[6] R. Backofen and D. Gilbert. Bioinformatics and Constraints. *Constraints*, 6:141–156, 2001.

[7] Rolf Backofen and Sebastian Will. Excluding Symmetries in Constraint-Based Search. In Joxan Jaffar, editor, *Proceedings of the 5th International Conference on Principles and Practice of Cosntraint Programming*, volume LNCS 1713, pages 73–87, 1999.

[8] P. Baptiste, C. LePape, and W. Nuijten. *Constraint-Based Scheduling. Applying Constraint Programming to Scheduling Problems.* 2001.

[9] J.C. Beck, P. Prosser, and R.J. Wallace. Trying Again to Fail-First. *CSCLP 2004: Joint Annual Workshop of ERCIM/CoLogNet on Constraint Solving and Constraint Logic Programming, Lausanne, Switzerland*, 2004.

[10] N. Beldiceanu, M. Carlsson, and J.-X. Rampon. Global Constraint Catalog. Technical report, SICS Technical Report T2005:08. Swedish Institute of Computer Science, Stockholm, Sweden., 2005.

[11] D. Cohen, P. Jeavons, C. Jefferson, K.E. Petrie, and B.M Smith. Symmetry Definitions for Constraint Satisfaction Problems. *Proceedings of Principles and Practice of Constraint Programming - CP 2005*, pages 17–31, 2005.

[12] A. Colmerauer. An introduction to Prolog-III. In *Communication of the ACM*, 1990.

[13] J. Crawford, M.L. Ginsberg, E. Luks, and A. Roy. Symmetry-breaking predicates for search problems. In *Principles of Knowledge Representation and Reasoning*, pages 148–159, 1996.

[14] Rina Dechter. *Constraint Processing.* Morgan Kaufmann, 2003.

[15] M. Dincbas, H. Simonis, and P. van Hentenryck. Solving the car-sequencing problem in constraint logic programming. In Y. Kodratoff, editor, *Proceedings ECAI-88*, pages 290–295, 1988.

[16] M. Dincbas, P. van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The constraint logic programming language CHIP. In *Proceedings of the International Conference on on Fifth Generation Computer Systems*, 1988.

[17] Alastair Donaldson and Peter Gregory, editors. *Proceedings of the SymNet Workshop on Almost-Symmetry in Search*, New Lanark, 2005.

[18] T. Fahle, S. Schamberger, and M. Sellmann. Symmetry Breaking. In Narendra Jussien and Francois Laburthe, editors, *Proceedings of the 7th International Conference on Principles and Practice of Cosntraint Programming*, volume LNCS 2239, pages 93–107, 2001.

[19] Boi Faltings. Distributed Constraint Programming. In F. Rossi, P. van Beek, and T. Walsh, editors, *Handbook of Constraint Programming*. Elsevier, 2006.

[20] P. Flener, A.M. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, J. Pearson, and T. Walsh. Matrix Modelling. Technical report, APES Reserach Group, 2001.

[21] P. Flener, A.M. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, J. Pearson, and T. Walsh. Symmetry in Matrix Models. Technical report, APES Reserach Group, 2001.

[22] Pierre Flener, Alan M. Frisch, Brahim Hnich, Zeynep Kiyiltan, Ian Miguel, Justin Pearson, and Toby Walsh. Breaking Row and Columns Symmetries in Matrix Models. In Pascal Van Hentenryck, editor, *Proceedings of the Eighth International Conference on Principles and Practice of Cosntraint Programming*, volume LNCS 2470, pages 462–476, 2002. FFHZMPW02.

[23] F. Focacci and P. Shaw. Pruning Sub-optimal Search Branches Using Local Search. In N. Jussien and Fl Laburthe, editors, *Proceedings of the Fourth International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimisation Problems*, pages 181–189, 2002.

[24] Filippo Focacci and Michaela Milano. Global Cut Framework for Removing Symmetries. In Toby Walsh, editor, *Proceedings of the 7th International Conference on Principles and Practice of Cosntraint Programming*, volume LNCS 2239, pages 77–92, 2001.

[25] A. M. Frisch, M. Grum, C. Jefferson, B. Martinez-Hermandez, and I. Miguel. The Design of ESSENCE: A Constraint Language for Specifying Combinatorial Problems. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI)*, 2007.

[26] A. M Frisch, C. Jefferson, B. Martinez-Hernandez, and I.Miguel. The Rules of Constraint Modelling. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI)*, 2005.

[27] Rico Gaudlitz. Optimization Algorithms for Complex Mounting Machines in PC Board Manufacturing. Master's thesis, Darmstadt University of Technology, 2004.

[28] P.A. Geelen. Dual Viewpoint Heuristics for Binary Constraint Satisfaction Problems. *Proceedings of ECAI'92*, pages 31–35, 1992.

[29] I. Gent, T. Kelsey, S.A. Linton, I. McDonald, I. Miguel, and B. M. Smith. Conditional Symmetry Breaking. In Peter van Beek, editor, *Principles and Practice of Constraint Programming - CP 2005*, volume LNCS 3709, pages 256–270, 2005.

[30] I. Gent and I. McDonald. NuSBDS: Symmetry Breaking made Easy. In B. Smith, I. Gent, and W. Harvey, editors, *Proceedings of the Third International Workshop on Symmetry in Constraint Satisfaction Problems*, pages 153–160, 2003.

[31] I. P. Gent and B. M. Smith. Symmetry Breaking During Search in Constraint Programming. *Proceedings ECAI 2000*, pages 599–603, 2000.

[32] Ian P. Gent, Warwick Harvey, Tom Kelsey, and Steve Linton. Gneric SBDD Using Computational Group Theory. In Francesca Rossi, editor, *Proceedings of the 9th International Conference on Principles and Practice of Constraint Programming*, volume LNCS 2833, pages 333–347, 2003.

[33] I.P. Gent, W. Harvey, and T. Kelsey. Groups and Constraints: Symmetry Breaking During Search. In *Proceedings of the Eighth International Conference on Principles and Practice of Cosntraint Programming*, 2002.

[34] I.P. Gent, K. E. Petrie, and J.-F. Puget. Symmetry in ConstraintProgramming. In F. Rossi, P. van Beek, and T. Walsh, editors, *Handbook of Constraint Programming*. Elsevier, 2006.

[35] C. Gervet. Constraints over Structured Domains. In F. Rossi, P. van Beek, and T. Walsh, editors, *Handbook of Constraint Programming*. Elsevier, 2006.

[36] D. Gilbert, R. Backofen, and R.H.C. Yap. Introduction to the special issue on bioinformatics. *Constraints*, 6:2–3, 2001.

[37] R. Greenstadt, J. P. Pearce, E. Bowring, and M. Tambe. Experimental analysis of privacy loss in DCOP algorithms. In *Proceedings of the fith international joint conference on Autonomous agents and multiagents*, pages 1424–1426, 2006.

[38] The GAP Group. GAP - Groups, Algorithms, and Programming, Version 4.3. http://www.gap-system.org, 2002.

[39] R.M. Haralick and G.L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–314, 1980.

[40] W. Harvey. Symmetry Breaking and the Social Golfer Problem. *Proceedings SymCon-01: Symmetry in Constraints*, pages 9–16, 2001.

[41] W. Harvey. The Fully Social Golfer Problem. *Proceedings SymCon 03: Symmetry in Constraints*, 2003.

[42] W. Harvey. A note on the compatibility of static symmetry breaking constraints and dynamic symmetry breaking methods. In *Proceedings SymCon-04: Symmetry and Constraint Satisfaction Problems*, 2004.

[43] W. Harvey, T. Kelsey, and K. Petrie. Symmetry Group Expressions for CSPs. In *Proceedings of the Third International Workshop on Symmetry in Constraint Satisfaction Problems*, pages 86–96, 2003.

[44] Warwick Harvey. CSP Lib prob010:The Social golfers problem.

[45] Warwick Harvey. Symmetric Relaxation Techniques for Constraint Programming. In *Proceedings of the SymNet Workshop on Almost-Symmetry in Search*, 2005.

[46] S.A. ILOG. *ILOG CPlex 6.0 Reference und User Manual*, 2002.

[47] S.A. ILOG. *ILOG OPL 3.5 User Manual*. ILOG, S.A., 2002.

[48] S.A. ILOG. *ILOG Scheduler Reference and User Manual*, 2002.

[49] S.A. ILOG. *ILOG Solver 5.3 Reference and User Manual*, 2002.

[50] J. Jaffar, S. Michaylov, P. J. Stuckey, and R. H. C. Yap. The clp(r) language and system. *ACM Transactions on Programming Languages and Systems*, 14:339–395, 1992.

[51] T. R. Jensen and B. Toft. *Graph Coloring Problems*. Wiley Interscience Series in Discrete Mathematics, 1995.

[52] T. Kelsey, S. Linton, and C.M. Roney-Dougal. New developments in symmetry breaking in search using computational group theory. *AISC*, 3249:199–210, 2004.

[53] N. Keng and D.Y. Yun. Planing/Scheduling Methodology for the Constrained Resource Problem. *Proceedings IJCAI'93*, pages 998–1003, 1993.

[54] Z. Kiziltan. *Symmetry Breaking Ordering Constraints*. PhD thesis, Uppsala University, 2004.

[55] Y.C. Law and J.H.M. Lee. Model Induction: A New Source of CSP Model Redundancy. *Proceedings of the 18th National Conference on Artificial Intelligence (AAAI-2002)*, pages 54–60, 2002.

[56] A.K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.

[57] J.P. Marques and K.A. Sakallah. GRASP: A Search Algorithm for Propositional Satisfiability. *IEEE Transactions on Computers*, 1999.

[58] Roland Martin. Open Mine Excarvation. Internal Presentation to Introduce in the Project with BHP Billiton, Australia, 2004.

[59] Roland Martin. The Challenge of Exploiting Weak Symmetries. In Brahim Hnich, Mats Carlsson, Francois Fages, and Freancesca Rossi, editors, *Recent Advances in Constraints, LNCS 3978*, volume LNCS 3978, pages 149–163, 2005.

[60] Roland Martin and Karsten Weihe. Breaking Weak Symmetries. In Warwick Harvey and Zeynep Kiziltan, editors, *Proceeding of the Fourth International Workshop on Symmetry and Constraint Satisfaction Problems*, pages 48–54, Toronto, Canada, 2004.

[61] Roland Martin and Karsten Weihe. Solving the Magic Square Problem by Using Weak Symmetries. In Brahim Hnich, editor, *Joint ERCIM/CoLogNet International Workshop in Constraint Solving and Constraint Logic Programming, CSCLP*, 2005.

[62] Iain McDonald and Barbara M. Smith. Partial Symmetry Breaking. In Pascal Van Hentenryck, editor, *Proceedings of the Eigth International Conference on Principles and Pracitse of Constraint Programming*, volume LNCS 2470, pages 431–445. Springer, 2002.

[63] B.D. McKay. *Nauty User's Guide (Version 2.2).*

[64] P. Meseguer and C. Torras. Exploiting Symmetries within Constraint Satisfaction Search. *Artificial Intelligence*, 129:133:163, 2001.

[65] Pedro Meseguer, Francesca Rossi, and Thomas Schiex. Soft Constraints. In F. Rossi, P. van Beek, and T. Walsh, editors, *Handbook of Constraint Programming*. Elsevier, 2006.

[66] P. J. Modi and M. Veloso. Multiagent Meeting Scheduling with Rescheduling. In *5th Workshop on Distributed Constraint Reasoning*, 2004.

[67] U. Montanari. Network of Constraints: Fundamental properties and applications to picture processing. *Information Science*, 7:95–132, 1974.

[68] A. Petcu and B. Faltings. An Efficient Constraint Optimization Method for Large Multiagent Systems. In *AAMAS Workshop on Large-scale Multi-agent Systems*, 2005.

[69] K.E. Petrie, B.M. Smith, and N. Yorke-Smith. Dynamic symmetry breaking in constraint programming and linear programming hybrids. In *Proceedings of the Second Starting AI Reserachers' Symposium – STAIRS 2004*, 2004.

[70] L.G. Proll and B. M. Smith. ILP and Constraint Programming Approaches to a Template Design Problem. *INFORMS Journal on Computing*, 10:265:275, 1998.

[71] J.-F. Puget. On the satisfiability of symmetrical constrained satisfaction problems. In *ISMIS, Vol 689 of LNCS*, 1993.

[72] J.-F. Puget. Automatic Detection of Variable and Value Symmetries. In P. van Beek, editor, *Principles and Practice of Constraint Programming - CP 2005*, volume LNCS 3709, pages 475–489, 2005.

[73] J.-F. Puget. Invited Talk at First International Day on Constraint Programming Tools (CP-Tools-06), Nantes, France. Nantes, France, September 2006.

[74] Jean-Francois Puget. Symmetry Breaking for Matrix Models Using Stabilizers. In Pascal Van Hentenryck, editor, *Proceedings of the Eighth International Conference on Principles and Practice of Constraint Programming*, volume LNCS 2470, pages 585–599, 2002.

[75] C.M. Roney-Dougal, I.P. Gent, T. Kelsey, and S. Linton. Tractable symmetry breaking using restricted search trees. In *Proceedings of the 16th European Conference on Artificial Intelligence, ECAI 2004*, 2004.

[76] F. Rossi. Constraint (Logic) Programming: A Survey on Research and Applications. *New Trends in Constraints*, 1865:40–74, 2000.

[77] F. Rossi, P. van Beek, and T. Walsh, editors. *Handbook of Constraint Programming*. Elsevier, 2006.

[78] H. Simonis, T. Cornelissens, V. Dumortier, G. Fabris, F. Nanni, and A. Tirabosco. Using Constraint Visualisation Tools. *Analysis and Visualization Tools for Constraint Programming*, LNCS 1870:321–356, 2000.

[79] B. M. Smith. Reducing Symmetry in a Combinatorial Design Problem. In *Proceedings of CP-AI-OR'01*, 2001.

[80] B. M. Smith. Caching Search States in Permutation Problems. In P. van Beek, editor, *Principles and Practice of Constraint Programming - CP 2005*, 2005.

[81] B. M. Smith. Modelling. In F. Rossi, P. van Beek, and T. Walsh, editors, *Handbook of Constraint Programming*. Elsevier, 2006.

[82] Barbara Smith. CSP Lib prob039: The Rehearsal Problem. http://www.csplib.org.

[83] B.M. Smith and S.A. Grant. Trying Harder to Fail First. *Proceedings ECAI'98*, pages 249–253, 1998.

[84] R. M. Stallmann and G. J. Sussmann. Forward Reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artificial Intelligence*, 9:135–196, 1977.

[85] I. Sutherland. Sketchpad: A man-machine graphical communication system. In *Proceedings of the IFIP Spring Joint Computer Conference*, 1963.

[86] M. Trick. Formulations and Reformulations in Integer Programming. *Proceedings of CPAIOR '05*, LNCS 3524:366–379, 2005.

[87] P. van Beck. Backtracking Search Algorithms. In F. Rossi, P. van Beek, and T. Walsh, editors, *Handbook of Constraint Programming*. Elsevier, 2006.

[88] P. van Hentenryck and V.A. Saraswat. Constraint programming: Strategic Directions in Constraint Programming. *Constraints*, 2:7–33, 1997.

[89] M. G. Wallace. Practical Applications of Constraint Programming. *Constraints*, 1:139–168, 1996.

[90] Toby Walsh. CSP Lib prob024: Langford's Number Problem. http://www.csplib.org.

[91] the free encyclopedia Wikipedia. Rubik's Cube: http://en.wikipedia.org/wiki/rubic%27s_cube.

[92] the free encyclopedia Wikipedia. Sudoku :http://en.wikipedia.org/wiki/sudoku.

[93] H. P. Williams. *Model Building in Mathematical Programming.* John Wiley and Sons, 2003.

[94] M. Yokoo, K. Suzuki, and K. Hirayama. Secure Distributed Constraint Satisfaction: Reaching Agreement Without Revealing Privat Information. In *Special Issue: Distributed Constraint Satisfaction*, pages 229–245, 2005.