Alastair F. Donaldson
Peter Gregory
(Eds.)

Almost-Symmetry in Search

SymNet Workshop
New Lanark, Scotland, 10–11 January 2005
Proceedings

Supported by SymNet, an EPSRC Funded Network of Excellence

Department of Computing Science
University of Glasgow
Glasgow G12 8QQ
Scotland

Technical Report: TR-2005-201
May 2005

Preface

This volume contains the proceedings of the SymNet Workshop on Approximate Symmetry in Search, held during January 10–11, 2005, in New Lanark, Scotland, UK. The aim of the workshop was to allow a small number of reserachers interested in symmetry and search problems to congregate in an informal setting, for discussion and presentation of reserach ideas. We feel that the workshop succeeded in this aim, with a series of talks and discussions involving contributions from the constraint satisfaction, model checking, and planning communities.

The workshop was organised jointly by the department of Computing Scince, University of Glasgow, and the department of Computer and Information Sciences, University of Strathclyde, as part of SymNet, and EPSRC funded network. We would like to thank the attendees of this workshop for making the event the great success it was. We would also like to thank Ian Gent for his encouragement, the EPSRC for their financial support, the members of SymNet for their interest in reading these proceedings, and Jon Ritchie for arranging for the proceedings to be printed.

An electronic version of this document is available from the SymNet website, located at `http://symnet.dcs.st-and.ac.uk/`.

May 2005

Alastair Donaldson
Peter Gregory
(Editors)

Table of Contents

Concrete Applications of Almost-Symmetry

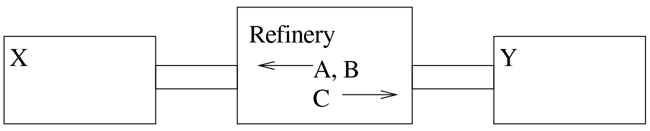
Peter Gregory[1] and Alastair Donaldson[2]

[1] University of Strathclyde
Glasgow, UK
[2] University of Glasgow
Glasgow, UK

Abstract. It seems intuitive that approximate symmetries occur in many real-world problems. Many of the examples that can be thought of quickly could be modelled in such a way as to reveal all of the symmetry. The more interesting examples occur when the aspect of the problem that has to be abstracted out is relevant to the solution.
Are these interesting examples pervasive throughout different search domains? The aim of this section is to describe real-world problems that contain approximate symmetries, and to discuss whether or not these problems could be abstracted to reveal symmetries without compromising solution correctness.

1 Chemical Plant Operation

In the 2004 International Planning Competition, one of the problem domains was the PipesWorld domain [1]. This domain models an oil-refinery that has to pipe certain materials to different places. There is a range of different oil-derived chemicals to pipe between different locations. It is sometimes possible to transport more than one type of chemical down the same pipe, given that the chemicals are compatible with each other (i.e., do not mix or react).



	A	B	C
A	✓	✓	✓
B	✓	✓	✗
C	✓	✗	✓

Fig. 1. A chemical refinery pumping three different types of chemicals (A, B and C) to two different locations (X and Y). The arrows on the refinery refer to the locations that the chemicals need to be pumped to. The grid shows which chemicals may be placed in the same pipe safely.

Each type of chemical could be considered symmetric if they each had the same compatibility relations to the other chemicals. Typically, they do not, their compatibility relations are often just similar. Consider the situation in Figure 1. The refinery must pump chemicals A and B to location X and chemical C to location Y. Chemical A is compatible with C, whilst B is not. Other than this fact, A and B are indistinguishable. In this situation, it is clear that this piece of information (compatibility with Chemical C) can be abstracted out. A and B are almost-symmetric with respect to the abstraction.

In general, the compatibilities of the chemicals could be modified to increase the symmetry of the problem, as could properties of pipes and locations. The amount of abstraction performed would clearly affect the usefulness of the abstraction in terms of solving the original problem.

2 Concurrent System with Priorities

Model checking [5] is a popular technique for the verification of concurrent systems. To verify a system by model checking, the system must first be converted (usually by hand) into a finite state model. Properties of the system are then verified by exhaustive search of this model. The application of model checking is limited due to the state-space explosion problem—as the number of components in a concurrent system increases, the size of the state-space of a model associated with this system grows combinatorially, quickly becoming too large to feasibly check. A lot of research in model checking concentrates on techniques to alleviate this problem. A popular technique is *symmetry reduction*. This involves exploiting the replicated structure of a concurrent system. Replication of *identical* processes in the system results in replicated portions of the state-space associated with this system. If known before search, this symmetry in the state-space can be exploited, and a smaller *quotient* state space can be searched instead, saving both time and memory.

In practice, concurrent systems may be *almost* symmetric, but not fully symmetric. For example, a system may be comprised of a set of processes competing for access to a shared resource [8]. These processes are identical, except that each process has an integer priority level. Access to the resource will be granted to a process with the highest priority level if several processes request access simultaneously.

The state graph of such a system will have a smaller group of symmetries than that of a system without priority levels. Thus the savings available through standard symmetry reduction techniques may be modest. However, when verifying a general property of the system, such as deadlock freedom, or the mutual exclusion property (the resource is always accessed by at most one process), the priority levels do not affect the truth or falsity of the property. In such cases it may be possible to abstract away from process priority levels and assume that there *is* full symmetry between components, in order to verify such properties over a small quotient structure with respect to a larger group of symmetries.

3 Constrained Latin Square

A Latin Square is an $n \times n$ grid in which n tiles of n different colours are placed such that no colour appears more than once in each row and column. These mathematical artifacts have application in experimental design. Rather than *any* Latin Square, often a Latin Square with distinct properties is required. For example, Figure 2(a) shows a 4×4 Latin Square. If the numbers (which represent colours) in the Latin Square correspond

1	2	3	4
2	3	4	1
3	4	1	2
4	1	2	3

(a) Unconstrained Latin Square

1	3	2	4
3	1	4	2
2	4	1	3
4	2	3	1

(b) Latin Square in which 3 and 4 must never be placed together on any row.

Fig. 2. Two Latin Squares.

to different drugs in a medical trial, the columns refer to weeks, and the rows to test subjects, then we can see that Patient 1 takes drugs 1, 2, 3 and 4 in their respective weeks.

However, the people at our ethical drug-trial lab know that taking drugs 3 and 4 in consecutive weeks can be dangerous, and so to find a schedule of trials, extra constraints must be placed on the Latin Square. In this case that 3,4 or 4,3 must not occur in any row (An example is shown in Figure 2(b)). This new problem has less symmetry than the original Latin Square problem. Thus symmetry breaking techniques for arbitrary Latin Squares may not be applicable when searching for constrained Latin Squares.

Intuitively, the constrained Latin Square 'almost' has the same symmetries as the unconstrained one. This relationship between unconstrained and constrained Latin Squares is discussed later in these proceedings [2]. For more information on the Latin Square and Design Theory in general see [3] on the web.

4 Balanced Academic Curriculum Problem

The difficulty of University timetabling problems is well-known (cite). The Balanced Academic Curriculum Problem (BACP) [4] schedules a set of courses across a certain number of periods for a complete degree programme. Each class requires a different amount of effort and therefore to distribute the classes uniformly would lead to variations in the effort required for individual periods. The problem is to balance this effort optimally between the periods. The problem is made more difficult by the fact that some courses have prerequisites. The problem can be defined by the following factors:

– **Courses** The list of courses available. Every course must be taken (at some point) by each student;
– **Number of Periods** The number of distinct time-periods in which to study. For example, in a 3-year degree with two semesters per year, the number of periods would be six;

– **Academic Load** The effort required, or academic credits achievable, for taking each course;
– **Prerequisites** The courses that must be studied in order to study each course;
– **Min. Academic Load** The minimum academic credits that each period should yield;
– **Max. Academic Load** The maximum realistic academic credits achievable by any one student;
– **Min. and Max. Number of Courses** The minimum and maximum number of courses required in each period.

The problem is to assign each course a period, such that the periods are optimally balanced.

Symmetry in this problem arises when different courses are equivalent to one another. However, the prerequisites of the courses will typically break this symmetry. Two courses with different prerequisites are now asymmetric, even if those prerequisites are non-interfering. Clearly these courses are still equivalent at some abstraction of the problem, but we cannot abstract the prerequisites out of the model as they are relevant constraints of the problem. Thus the BACP exhibits almost-symmetry.

There are two things we can do to increase the symmetry in this model. The first is to loosen the constraints by removing prerequisites. The second is to tighten the constraints by adding prerequisites to courses (to make them equivalent to more constrained courses). The almost-symmetry in the problem can be revealed by either of these approaches. Solving the problem using the first abstraction would lead to infeasible solutions and any solver would have to keep track of the true constraints to preserve soundness. Using the second abstraction may yield a sub-optimal solution since the added constraints will compromise completeness, again the solver will have to compensate for this.

Without solvers that can deal with almost-symmetry, it is difficult to say which alternative approach is best.

5 Automated Manufacture

Martin and Weihe [6], with industrial partners, have studied a problem involving the design of schedules for circuit-board assembly. The application is described in further detail in these proceedings [7].

The problem deals with a conveyor-belt with PCBs and several robotic arms that can place components. These arms can only operate in small windows on the belt and can only have access to a small number of different components. The problem is to assign components to different arms to optimise a schedule.

The arms are clearly not symmetric, since their location and viewing-window are important to the solution and cannot be ignored. However, interchanging two arms will affect only the quality of the solution, not its feasibility.

References

1. Edelkamp, S., Hoffmann, J.: International planning competition. `http://ls5-www.cs.uni-dortmund.de/~edelkamp/ipc-4/` (2004)

2. Harvey, W.: Symmetric relaxation techniques for constraint programming. In: SymNet Workshop on Almost-Symmetry in Search, New Lanark. (2005)
3. Queen Mary, U.o.L.: Design theory. `http://designtheory.org/`
4. Castro, C., Manzano, S.: Variable and value ordering when solving balanced academic curriculum problems. In: Proceedings of 6th Workshop of the ERCIM WG on Constraints. (2001)
5. Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. The MIT Press, Cambridge, Masachusetts, 1999.
6. Martin, R., Weihe, K.: Breaking weak symmetries. In: Proceedings of the 4th International Workshop on Symmetry and Constraint Satisfaction Problems. (2004)
7. Martin, R.: Approaches to symmetry breaking for weak symmetries. In: SymNet Workshop on Almost-Symmetry in Search, New Lanark. (2005)
8. A. Prasad Sistla and Patrice Godefroid. Symmetry and reduced symmetry in model checking. *ACM Transactions on Programming Languages and Systems*, 25(4):702–734, July 2004.

Restoring Symmetries in Almost Symmetric Graph Structures

Derek Long and Maria Fox[⋆]

University of Strathclyde, Glasgow, UK

Abstract. The concept of symmetries in graphs is well understood: a symmetry of a graph is simply an automorphism of the graph. There are well-known techniques for finding graph automorphisms [2], based on partitioning of nodes according to their out-degrees. It is not uncommon for a graph to exhibit structure that is *almost* symmetric. That is, the graph would contain high degrees of symmetry were it not for minor blemishes in the structure: missing edges or excess edges. In this paper, we propose and briefly examine techniques for identifying the edges that lead to the breakdown in symmetries. The work described in this paper is still at an early stage of development, so we outline the directions in which we intend to progress our exploration.

1 Introduction

Symmetries have been an important subject of research for decades, forming the basis for the development of group theory. They arise in many areas of science, providing an elegant theoretical tool for interpreting many phenomena. Computer science is no exception, offering a fertile ground for exploitation of symmetries in a wide range of problems. In a similar way, graphs have proved a powerful abstract formalism for representing a huge range of problems in computer science. It is therefore no surprise that symmetries and graphs have been studied in conjunction. Symmetries in graphs are graph automorphisms and they are an abstraction of a very powerful idea that arises in many areas of computer science. Finding graph automorphisms is a problem that sits on the edge of tractability: efficient algorithms are known, although none has been proved polynomial. NAUTY [2] is one of the best known implementations of an algorithm for finding graph automorphisms.

Many graphs arise in contexts in which symmetries would be useful, but the graphs exhibit little or no useful symmetry. This can often be the consequence of comparatively minor blemishes in the structure of the graphs, since a single missing or excess edge can break a very large collection of potential symmetries of a graph. For example, consider a 5 clique, which has 120 automorphisms: if a single edge is removed then this drops to just 12 automorphisms! In this paper we consider the problem of finding the best possible modifications to a graph that restore (or create) symmetries.

⋆ The authors also with to acknowledge the contributions of Julie Porteous in discussions leading to the work reported here.

2 Graph Automorphisms

It is helpful to briefly review the most common algorithmic process by which graph automorphisms are identified. The algorithm begins by partitioning the nodes of the graph according to their outdegrees (the numbers of edges connecting to each of the nodes). Then, in an iterative process, these partitions are further sub-partitioned by considering the outdegrees of the nodes within one partition when restricted to edges that link the nodes to nodes in another partition. Once no partition further partitions according to this criterion, then the partitions represent the seeds of a group of automorphisms. An exhaustive search procedure can be applied in which each of the partitions is split, in all possible ways, by the removal of one node from the partition. The new partitioning of the nodes created in this way is then examined, as before, for further implied partitionings, until it is stable. The search is pursued depth-first until all partitions are singletons. On backtracking through the search tree and expanding alternative choices, the algorithm will generate different orderings of the nodes in the partitioned graph and each such different ordering is automorphic with the original ordering.

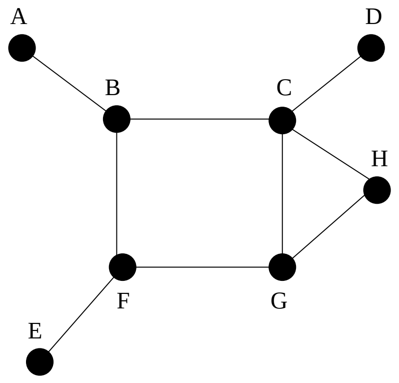


Fig. 1. Simple graph example.

An example will help to clarify this procedure. Consider the graph show in figure 1. This partitions into the collection: $\{ADE|H|BFG|C\}$, in ascending order of outdegrees, from 1 to 4. Now, considering the first and fourth partitions, we see that D has outdegree 1 (the edge to C), while A and E have 0 (no edges to C). Hence, we refine the partition collection into: $\{AE|D|H|BFG|C\}$. Similarly, if we consider the partition containing B against that containing C, we find that B and G have outdegree 1, while F has 0. This leads to the partitions: $\{AE|D|H|F|BG|C\}$. Comparing A and E with $\{BG\}$ we further divide into: $\{E|A|D|H|F|BG|C\}$ and finally, comparing B and G with A, we obtain: $\{E|A|D|H|F|G|B\}$. Since no partition ends up with more than one element, this graph has no non-trivial automorphisms.

A key observation to be made about this process is that potential symmetries of the graph are lost whenever a collection of vertices is split into two or more partitions: the splitting necessarily prevents any symmetry between pairs of vertices each of which appears in a separate partition.

3 Almost Symmetries of Graphs

We now attempt to formalise the definition of an almost symmetry of a graph. This idea is closely related to the concept of almost symmetries in planning problems, introduced in [1]. The intuition is that symmetries arise as a consequence of abstractions. For example, to claim that two solutions of the n-queens problem are symmetrical, we will usually need to abstract out the underlying checkerboard pattern of colours and the physical locations of the queens (as opposed to the relative locations on the board). Thus, to increase the degree of symmetry in a structure we must apply some abstraction to it. The abstraction will remove the sources of differentiation between elements of the structure and allow more opportunities for symmetry to arise. This intuition motivates the following definitions:

Definition 1. *A* graph abstraction relation *is any binary relation on graphs, $\triangleright$, such that $G \triangleright H$ only if G and H have the same set of nodes.*

Definition 2. *Given a graph G, a graph abstraction relation $\triangleright$, then for any graph H such that $G \triangleright H$, any automorphism of H is an* almost symmetry *of G, with respect to $\triangleright$.*

Where the abstraction we are using is clear from the context, we will not make it explicit in referring to almost symmetries of a graph. It is clear that these definitions allow a very broad form for almost symmetries of a graph. The crucial constraint is from the graph abstraction relation we consider. In general, we will only be interested in graph abstractions that maintain a close relationship between the two graphs. If we were to apply an abstraction that simply throws away all the edges of the original graph then we would end up with a very high degree of almost symmetry (all the vertices then become symmetric with each other, so there will be a complete permutation symmetry on them). Of course, this abstraction is unlikely to be of much interest, since it throws away too much of the structure of the original graph. Instead, we will want to exploit abstractions that eliminate small parts of a graph, while retaining most of what makes the graph interesting. The abstractions we are most interested in take the following basic form:

Definition 3. *Given a positive integer, d, the subgraph abstraction to distance d, $\triangleright_d$ is the relation such that $G \triangleright_d H$ iff H is a subgraph of G containing all the nodes of G and in which G has at most d more edges than H. H is called a* subgraph abstraction of G at a distance d from G.

Candidate subgraph abstractions of a graph at a given distance, d, away are easy to construct, since they simply involve removing subsets of d edges from the graph. In practice, we are not interested in removing large sets of edges and d will be restricted to a small number. How small is not yet clear: it is possible that the number of edges we could consider removing should be measured as a proportion of the size of the graph being abstracted, but the complexity of a naive search for suitable subgraph abstractions is exponential in *d* and this is likely to force *d* to remain modest.

An alternative abstraction involves adding edges to a graph, *G*, but this is easily handled using subgraph abstraction by taking the complement of *G* and then applying subgraph abstraction to it, since the edges removed from the complement correspond to edges added to the original graph. More extensive abstractions, such as addition and removal of edges, remain outside the scope of the current work.

4 Finding Almost Symmetries in Graphs

Now that we have defined what we are looking for, we consider how we can find it. The simplest algorithm is to take the original graph, systematically remove edges from it (incrementing the number of edges to be removed as the set of abstracted graphs at the current distance is exhausted), applying NAUTY to each candidate subgraph. This algorithm is obviously naive and impossibly expensive for finding good abstractions at any significant distance from the original graph. In addition, it will simply enumerate all possible almost symmetries without any discrimination. In general, we will be able to identify better or worse kinds of almost symmetry according to the way in which they relate different vertices in the original graph. We are still examining ways to evaluate candidates, but we have considered the following heuristics: sets of vertices that we would be happy to see made symmetric can be assigned reward values — the more vertices of such a set that are drawn into a symmetry, the better the symmetry. Similarly, edges can be assigned costs — the more edges we remove the more structure we lose from the original graph and often some edges are of much more importance in preserving the integrity of the original graph than others. Therefore, the cost of an abstraction can be measured according to the sum of the costs of the edges removed, while the benefit can be measured according to the sum of the rewards for vertices that are drawn into the almost symmetries the abstraction generates.

Another factor appears to be important: in certain contexts it is important to identify almost symmetries that relate the largest possible substructures of the original graph. For example, in figure 2 the almost symmetry between the two sub-structures indicated could be more interesting than the more local symmetries created by the alternative abstraction proposed. In other contexts it might be of more use to find the local symmetries, since local symmetries are more robust to subsequent operations on the graph. This means that if, for instance, a search is being carried out across structures represented by the graph, then as choices are made that affect the roles of the vertices in the search space, more of the local symmetries will remain active and might be exploited in reducing subsequent branches of the search than would be the case for large-scale symmetries. It is possible to heuristically estimate the potential degree of additional symmetry in an abstraction. If we consider the first level of partitioning performed by the partition-based automorphism identification algorithm described in section 2, then a heuristic measure of the potential number of automorphisms is the product of the factorials of the sizes of each of the partition sizes. This is based on the possibility that all vertices in a partition could be symmetric with one another, independently of the symmetries on the vertices in other partitions. This is therefore a maximum possible

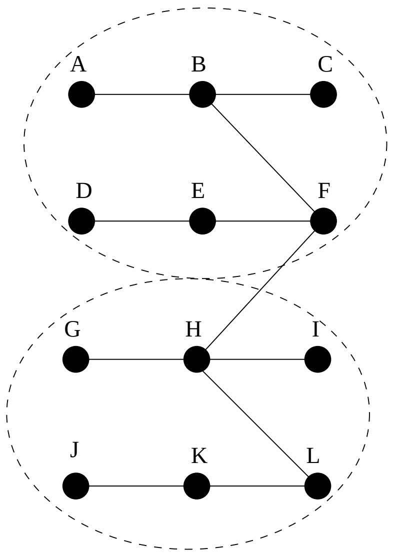


Fig. 2. Almost symmetries in a graph: if the edge FH is removed then the two identified structures are symmetric. On the other hand, if BF is removed then there is more local symmetry, since the vertices A and C are symmetric, while DEF and JKL are symmetric and G and I are symmetric, giving three independent 2-fold symmetries.

number of automorphisms, but it gives a guide to which candidate partitions should be favoured when considering which edges to remove in a subgraph abstraction. The other measurement of interest is the sum of the sizes of the partitions, each taken modulo *k*, for some integer *k*. If this sum is multiplied by *k* then it indicates the maximum number of nodes that might be involved in a structural symmetry of order *k*.

Consider the graph shown in figure 3: the sizes of the initial partitions are shown in table 1, along with the sizes of the partitions that are possible after the removal of one edge. Alongside these we show the factorial products for the corresponding potential abstractions and the sum of partition sizes modulo *k* for different values of *k*. It is of considerable interest to note that in this case the 3-fold symmetry offers the best scope for large-scale symmetry, with removal of an edge in partition $\{B,E,F,H,M\}$, while the removal of an edge in the partition $\{C,G,L\}$ offers the best scope for local symmetries. The former case can be achieved by removal of edge EF (and it is the only choice), which does indeed restore full 3-fold symmetry to the resulting graph. The latter case cannot be achieved for this example, so removal of the same edge, EF, offers next best scope for local symmetries. After that, removal of an edge between partition $\{A,D,I,J,K,N,O\}$ and partition $\{C,G,L\}$ (line labelled 101 in the table) offers best scope. This option leads to discovery that KL or LO would offer separate 2-fold symmetries for the two resulting graph components.

The next question is how to determine which edges might be removed. When an edge is removed, it affects the outdegrees of both ends. As a consequence, we are

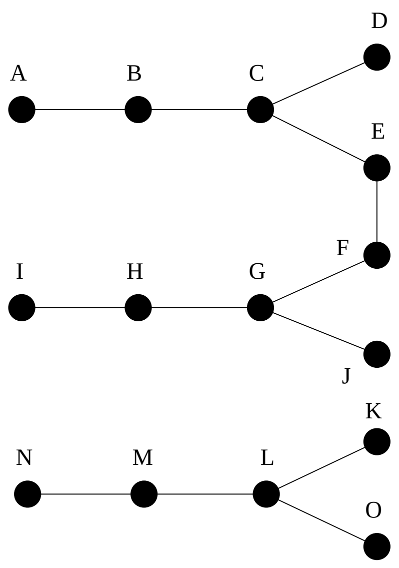


Fig. 3. Example of a graph with 3-fold almost symmetry (discovered by removal of edge EF).

Graph	Outdegree				Factorial	$k\times$ Sum modulo k		
	0	1	2	3	product	2	3	4
Original	0	7	5	3	3628800	12	12	8
200	2	5	5	3	172800	12	9	8
110	1	7	4	3	725760	12	12	8
101	1	6	6	2	1036800	14	12	8
020	0	9	3	3	10363680	12	15	8
011	0	8	5	2	976800	14	9	12
002	0	7	7	1	25401600	12	12	8

Table 1. Table of heuristic symmetry measurements for the graph in figure 3. The initial partitions of the original graph are: $\{A,D,I,J,K,N,O\}$, $\{B,E,F,H,M\}$ and $\{C,G,L\}$. The entries at the left indicate where vertices move out of a partition (for outdegrees 1, 2 and 3 respectively) and one position left. They sum to 2 because one edge is being considered for removal here.

looking for pairs of vertices that should both move into different partitions in order to improve the symmetry of the abstracted graph. This process can be modelled as a search for a matching between certain pairs of vertices. Because a single vertex might lose more than one edge in the abstraction, we must construct a new graph in which vertices are duplicated according to the number of possible edges we are considering removing and then we search for maximal matchings in this new graph. The new graph will contain only those vertices in the partitions that have been identified as candidates for edge-removal. So, for example, in the graph in figure 2, having identified the possibility of removing one edge in partition $\{B, E, F, H, M\}$, we need only build a graph containing these vertices and edges between them in the original graph. If we want to weight the edges to represent relative costs for their removal we will then be looking for a minimum cost matching of size d in the subgraph. This problem is a version of well-known matching problems for graphs and algorithms exist to perform it efficiently, particularly when we are restricting the size of the matching to small values of d. There are complications when d is larger than 1, since there is no constraint that edges that are removed should each link distinct pairs of vertices. For this reason, we construct a separate graph in which to identify candidate matchings. This graph, $S(G)$ contains only the candidate vertices and edges between them, from the original graph G. However, each vertex appears in $S(G)$ as many times as we are prepared to consider reducing its outdegree. For example, if we are searching with $d = 2$ and are examining a single partition, $\{A, B, C, D\}$, say, then each vertex in the partition may have its outdegree reduced by 2 (assuming that we do not allow self-looping edges). Thus, we consider two copies of each vertex $\{A_1, A_2, B_1, B_2, C_1, C_2, D_1, D_2\}$ and create edges between pairs of instances of each vertex if there is an edge between the corresponding vertices in G. This allows us to find matchings that use a given vertex multiply often, but it also allows us to find matchings that use single edges multiply often, which is not useful to us. We are still examining ways to overcome this problem.

Once the matching is identified and removed from the graph, the resulting subgraph must be tested for automorphisms. We believe that there should be opportunities to capture information from the subsequent partitioning process to enable us to refine the decision about which matching is likely to offer better almost symmetries. In particular, when a partition is bound to further subdivide, regardless of removal of a small number of edges, then there is no point in attempting to improve the symmetries associated with that partition.

5 Future Work and Conclusions

The work described in this paper is still at an early stage. We are in the process of exploring alternative search strategies for the identification of the best edges to remove in order to identify good abstractions leading to almost symmetries. There remain, too, important questions about the exploitation of almost symmetries. We have already shown that almost symmetries can play a useful role in planning [1]. It seems plausible that similar possibilities can arise in other search problems. In addition, we believe that the abstract forms of symmetry and almost symmetry that arise in considering graph structures could have a much wider application. For example, symmetry of molecules is an important area of research in chemistry [4] and the concept of approximate molecular symmetries has already been considered as part of the Pluto tool [3].

6 Acknowledgements

The authors wish to thank colleagues in the Symmetry in Search EPSRC Network of Excellence for interesting discussions on symmetries, almost symmetries and automorphisms in graphs, all of which have contributed to the work described here. Thanks are also due to Peter Gregory and Alastair Donaldson for organising a very inspiring workshop and forcing us to think enough about these ideas to write them down.

References

1. M. Fox, D. Long, and J. Porteous. Abstraction-based action ordering in planning. In *Proceedings of IJCAI'05*, 2005.
2. B.D. McKay. *Nauty* user's guide 1.5. Technical Report TR-CS-90-02, ANU, Canberra, 1990.
3. W. D. Samuel Motherwell, Gregory P. Shields, and Frank H. Allen. Visualization and characterization of non-covalent networks in molecul ar crystals: automated assignment of graph-set descriptors for asymmetric molecu les. *Acta Crystallographica Section B*, 55(6):1044–1056, Dec 1999.
4. Jing Wen Yao, Jason C. Cole, Elna Pidcock, Frank H. Allen, Judith A. K. Howard, and W. D. Samuel Motherwell. *CSDSymmetry*: the definitive database of point-group and space-gr oup symmetry relationships in small-molecule crystal structures. *Acta Crystallographica Section B*, 58(4):640–646, Aug 2002.

Almost-Symmetry Research in Planning: A Review

Peter Gregory

University of Strathclyde
Glasgow, UK

Abstract. How can the almost-symmetry in planning problems be exploited? One method is to assume a more symmetric state than is actually true then attempt to fix the resulting 'solution'. This approach is interesting but as yet unimplemented. Recent results show that almost-symmetry can be exploited in forward-search planners, specifically the Fast Forward (FF) planning system.
This paper surveys the progress made in planning relating to almost-symmetries.

1 Introduction

Two approaches to exploiting almost-symmetry in planning are discussed here. The first is introduced in [1], this approach assumes the initial state is actually more symmetric than it is. The hope is that after a solution is found, a prefix can be added to the plan that will restore the validity of the solution. This approach has not been exploited in practice.

A different way of exploiting the almost-symmetry in problems (by way of a property abstraction) is described in [2, 3]. Modifications are made to the FF planner [4] (a heuristic forward-search planner) to prefer choices symmetric to those chosen earlier in the plan. Results have shown this approach to be statistically beneficial in a number of planning domains.

2 Plan Prefixes

Space exploration is a very active research area in the planning community. Imagine a situation where several planetary rovers are exploring in an on-line environment. The rovers could be performing many different tasks; rock/soil sampling, atmospheric testing, taking photos, etc. In this complex environment, it is likely that soon after the start of the mission there will be a general asymmetry between the rovers; for example, their instruments will be calibrated differently, they will be exploring different locations.

If the environment is very complex then it may be infeasible to explore the asymmetric problem. If the rovers were symmetric, however, the problem may again become solvable. The problem is now split into two sub-problems. Firstly, find a highly symmetric state that is reachable for only a small cost and secondly, solve the new symmetric problem.

Planning to a symmetric state explicitly could be difficult, not least because the goal would be difficult to specify. So, the supposed best current idea is to assume a symmetric state and then attempt to create a prefix later. This gives more flexibility to the planner for the second stage. In our planetary rovers example we might assume that all of the instruments start in the same calibration configuration, as it should be quite inexpensive to turn off all of the experimentation instruments for example; thus increasing the symmetry in the problem.

This approach to handling almost-symmetry has not been explored in current planners. This is probably because the benchmark instances are not currently large enough to warrant such an approach. With small plan makespans, the negative aspects of this technique are much more obvious (i.e. the plan is rarely optimal because of the introduction of a plan prefix.

3 Forward-Search Property Abstraction

3.1 The Fast-Forward Planning System

The FF planner [4] has been one of the most competitive planners of recent years. It works with a simple relaxation of planning domains, combined with Enforced Hill-Climbing local search. FF's heuristic works by computing a relaxed plan-graph by ignoring the delete effects of actions. The length of the extracted relaxed plan forms the heuristic to guide the search.

FF will only consider actions that came from the first level of the relaxed plan, these are called helpful actions.

3.2 The Property Abstraction

One way to abstract a planning problem is, for each object, only consider the type of relations an object has and not actually to which other objects it is bound by those relations. Thus, a rover at some location with a soil sample is symmetric to any other rovers at locations with a soil sample, even though the locations and soil samples have different identities.

3.3 Using the Property Abstraction

This abstraction can be used to further inform the search of FF [2]. When FF chooses which action to apply from its useful action set, the choice is arbitrary. If the useful action set becomes large then the choice becomes less informed. The property abstraction gives us more information about the action choices.

If there was an action applied earlier in the plan that is almost-symmetric to one we are considering in the useful action set, then we can assume it best to apply that action now. This is the only change made to FF's search, to prefer symmetric choices in the search. From the domains studied, for two metrics (time and states visited), this approach is shown to out-perform FF, to a statistical significance of $> 95\%$.

4 Conclusion

Two interesting ways of exploiting the almost-symmetry in planning problems are to introduce symmetry explicitly in the solution, and to use abstractions to implicitly describe the symmetry in the problem. When introducing symmetry explicitly, the solutions will have the overhead of that introduction, this will be significant in some cases, not in others. When using abstractions, there must be a clear reason to believe that the abstraction will contribute useful information to the problem and not just add to the complexity of the problem. One way to ensure this is by only considering actions that would have been considered anyway (as is done in [2]).

The interested reader is encouraged to follow all of the references in this brief review of the almost-symmetry research in planning.

References

1. Fox, M., Long, D.: Symmetries in planning problems. In: Proceedings of the 3rd International Workshop on Symmetry and Constraint Satisfaction Problems. (2003)
2. Fox, M., Long, D., Porteous, J.: Abstaction-based action ordering in planning. In: International Joint Conference on AI (IJCAI). (2005)
3. Porteous, J., Long, D., Fox, M.: The identification and exploitation of almost symmetry in planning problems. In Brown, K., ed.: Proceedings of the 23rd UK Planning and Scheduling SIG. (2004)
4. Hoffmann, J., Nebel, B.: The FF planning system: Fast plan generation through heuristic search. Journal of Artificial Intelligence Research **14** (2001) 253–302

Partial Symmetry in Model Checking

Alastair Donaldson

University of Glasgow,
Glasgow, UK

Abstract. Symmetry reduction techniques have been shown to be successful in combatting the state-space explosion problem for model checking. We provide a brief survey of techniques which extend the application of symmetry reduction to *partially* symmetric systems.

1 Introduction

Model checking [3] is an increasingly popular technique for the formal verification of concurrent systems. The application of model checking is limited due to the state-space explosion problem—as the number of components represented by a model increases, the size of the associated state-space grows exponentially. As such, models of realistic systems are often too large to feasibly check. Symmetry reduction techniques [2, 4, 13] can be used to combat this problem for models of systems with many replicated components. Symmetry in a system can result in portions of the state-space of a model of the system being *equivalent* up to rearrangement of component ids. If symmetry is known to be present in a model then model checking of certain properties can be performed over a quotient state-space, which is generally smaller than the full state-space of the model.

In practice, a concurrent system may consist of many similar, but not identical, processes. For example, processes in a system may be distinguished by a set of priority levels. In this case the state-space underlying a model of the system will not exhibit full symmetry. However, in certain cases, it is safe to assume that processes *are* identical, and perform model checking over a reduced state-space with respect to this *partial* symmetry. In this paper we overview the theory of symmetry in model checking, then provide a brief survey of techniques for handling partially symmetric systems.

2 Symmetry in Model Checking

Model checking involves checking the correctness of a temporal logic formula ϕ over a Kripke structure $\mathcal{M} = (S,R,L)$ and a set of atomic propositions AP, where S is a finite set of states, $R \subseteq S \times S$ is a total transition relation, and $L : S \to 2^{AP}$ labels each state with the propositions that are true at the state. The Kripke structure $\mathcal{M}$ represents a model of a concurrent system. In practice $\mathcal{M}$ is obtained from a high level specification written in a language such as Promela [12].

Let $\mathcal{M} = (S,R,L)$ be a Kripke structure. An *automorphism* of $\mathcal{M}$ is a bijection $\alpha : S \to S$ which satisfies the following condition:

– $\forall s,t \in S,\ (s,t) \in R \Rightarrow (\alpha(s),\alpha(t)) \in R$,

In a model of a concurrent system with many replicated processes, Kripke structure automorphisms usually involve the permutation of process identifiers of identical processes throughout all states of a model. The set of all automorphisms of the Kripke structure $\mathcal{M}$ forms a group under composition of mappings. This group is denoted $Aut(\mathcal{M})$. A subgroup G of $Aut(\mathcal{M})$ induces an equivalence relation $\equiv_G$ on the states of $\mathcal{M}$ thus: $s \equiv_G t \Leftrightarrow s = \alpha(t)$ for some $\alpha \in G$. The equivalence class under $\equiv_G$ of a state $s \in S$, denoted $[s]$, is called the *orbit* of s under the action of G. The orbits can be used to construct a *quotient* Kripke structure $\mathcal{M}_G$ as follows:

Definition 1. *The quotient Kripke structure $\mathcal{M}_G$ of $\mathcal{M}$ with respect to G is a tuple $\mathcal{M}_G = (S_G, R_G, L_G)$ where:*

– $S_G = \{[s] : s \in S\}$ *(the set of orbits of S under the action of G),*
– $R_G = \{([s],[t]) : (s,t) \in R\}$,
– $L_G([s]) = L(rep([s]))$ *(where $rep([s])$ is a unique representative of $[s]$).*

In general $\mathcal{M}_G$ is a smaller structure than $\mathcal{M}$, but $\mathcal{M}_G$ and $\mathcal{M}$ are equivalent in the sense that they satisfy the same set of logic properties which are *invariant* under the group G (that is, properties which are "symmetric" with respect to G). For a proof of the following theorem, together with details of the temporal logic CTL^*, see [3].

Theorem 1. *Let $\mathcal{M}$ be a Kripke structure, G a subgroup of $Aut(\mathcal{M})$ and ϕ a CTL^* formula. If ϕ is invariant under the group G then*

$$\mathcal{M}, s \models \phi \Leftrightarrow \mathcal{M}_G, [s] \models \phi$$

Thus by choosing a suitable symmetry group G, model checking can be performed over $\mathcal{M}_G$ instead of $\mathcal{M}$, often resulting in considerable savings in memory and verification time [2, 4].

3 Virtual Symmetry

Virtual symmetry [9] is a general condition for a model $\mathcal{M}$ and a group G, which, if satisfied, means that model checking of symmetric properties can be performed over the quotient model $\mathcal{M}_G$, even if G is not a group of automorphisms of $\mathcal{M}$. The intuition behind virtual symmetry is as follows. If G is a group which permutes the components of the model $\mathcal{M}$, then although G is not necessarily a symmetry group for $\mathcal{M}$, if $\mathcal{M}$ is *virtually symmetric* with respect to G then there is an abstraction $\mathcal{M}^G$ of $\mathcal{M}$ such that G *is* a symmetry group for $\mathcal{M}^G$. Essentially, the model $\mathcal{M}^G$ is obtained by adding edges to $\mathcal{M}$ in such a way that G preserves the resulting transition relation.

Before giving the definition of virtual symmetry, for the sake of completeness we give two previous notions of partial symmetry—*near automorphisms* and *rough symmetry* [8]. Although virtual symmetry subsumes these notions, they are helpful in understanding the definition of virtual symmetry, which is quite abstract.

Near automorphisms: Suppose $\mathcal{M}$ is a model of a system, and $\mathcal{I}$ the set of process identifiers associated with $\mathcal{M}$. A permutation $\theta \in Sym\ mathcalI$ acts on a state s of $\mathcal{M}$ by permuting the components of s. A permutation $\theta \in Sym\ \mathcal{I}$ is said to be a near automorphism of $\mathcal{M}$ if, for every transition $s \to t$ of $\mathcal{M}$, either $\theta(s) \to \theta(t)$ is a transition of $\mathcal{M}$ or s is totally symmetric with respect to $Aut(\mathcal{M})$. (That is, s is invariant under $Aut(\mathcal{M})$.) The model $\mathcal{M}$ is said to be nearly symmetric with respect to G if G is a group of near automorphisms for $\mathcal{M}$.

Rough symmetry: If, on the other hand, G is a subgroup of $Sym\ \mathcal{I}$ then $\mathcal{M}$ is roughly symmetric with respect to G if for every pair of states s and s' where $s \sim_G s'$, any transition from s is matched by a transition from s' provided the associated local transition (from s') would involve a process with the highest priority.

If $\mathcal{M}$ is a nearly (roughly) symmetric model with respect to group G then, despite the lack of complete symmetry, the quotient model $\mathcal{M}_G$ is bisimilar to the original model $\mathcal{M}$. It follows that symmetry reduction preserves all symmetric CTL^* properties, thus a symmetric CTL^* property can be safely checked over $\mathcal{M}_G$.

Virtual symmetry: The notions of near and rough symmetry [8] are subsumed by the notion of *virtual* symmetry [9]. The symmetrization R^G of a transition relation R by a group G is defined by

$$R^G = \{\alpha(s) \to \alpha(t) : \alpha \in G \text{ and } s \to t \in R\}.$$

Intuitively, symmetrizing a transition relation can be thought of as the process of adding transitions which are missing due to asymmetry in the system.

Definition 2. *A structure $\mathcal{M}$ is virtually symmetric with respect to a group G acting on S if for any $s \to t \in R^G$, there exists $\alpha \in G$ such that $s \to \alpha(t) \in R$.*

If a Kripke structure $\mathcal{M}$ is virtually symmetric with respect to a group G, then $\mathcal{M}$ is bisimilar to the quotient model $\mathcal{M}_G$, and model checking of symmetric properties can be performed over $\mathcal{M}_G$. A method of demonstrating the virtual symmetry of a structure by counting missing arcs of the structure has been proposed [9]. However, it is unclear how virtual symmetry can be detected from the source text of a model.

The results on near-automorphisms, rough symmetry and virtual symmetry [8, 9] are proved, for simplicity, in the case where models do not involve shared variables or channels.

4 Guarded Annotated Quotient Structures

The problem of applying symmetry reduction to systems with little or no symmetry is also considered in [14]. The notion of an annotated quotient structure [10, 7, 11] is extended to a *guarded annotated quotient structure*. Suppose $\mathcal{M}$ is the Kripke structure of a system, and $\mathcal{M}' \supseteq \mathcal{M}$ is obtained from $\mathcal{M}$ by adding transitions (in a similar manner to the process of symmetrization described above [9]), so that $\mathcal{M}'$ has more symmetry than $\mathcal{M}$. Then a guarded annotated quotient structure for $\mathcal{M}$ can be viewed as an annotated quotient structure for $\mathcal{M}'$, with edges labelled with guards to indicate which processes can make each transition, so that the original edges of $\mathcal{M}$ can be recovered from the representation of $\mathcal{M}'$ [14]. A temporal formula f can be checked over the guarded annotated quotient structure by unwinding the structure, even if f is not symmetric with respect to the automorphisms used for reduction. This approach potentially allows large factors of reduction to be obtained since a larger group of automorphisms than would usually be possible using standard quotient structure reduction can be employed. Encouraging experimental results using the SMC model checker [15] are reported. Once again, no indication is given as to how the kind of asymmetry handled by this approach can be detected from the source text of a program.

5 Other Approaches

Ajami et al. [1] show that standard approaches to symmetry reduction in CTL^* model checking [4, 5, 10], which simultaneously exploit symmetries of both the system and the property, fail to capture symmetries in LTL path subformulae. They investigate an approach to symmetry reduction using a quotient structure for the synchronous product of the Büchi automaton of a LTL formula and the global state transition graph. The approach exploits local symmetries of the Büchi automaton. They present algorithms showing that model checking can be efficiently performed over this quotient structure, and claim to have implemented these algorithms, but do not provide any experimental results.

6 Conclusions

We have surveyed various approaches to alleviating the state-space explosion problems for systems which are partially symmetric. An interesting problem for future research will be automatically detecting partial symmetries from system descriptions. Current approaches assume that information about partial symmetries is known *a priori*. Perhaps existing techniques for symmetry detection [6] could be extended to handle partially symmetric systems.

References

1. K. Ajami, S. Haddad, and J. Ilie. Exploiting symmetry in linear time temporal logic model check ing: One step beyond. In B. Steffen, editor, *Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS '98)*, volume 1384 of *Lecture Notes in Computer Science*, pages 52–67, Lisbon, Portugal, March/April 1998. Springer-Verlag.
2. D. Bosnacki, D. Dams, and L. Holenderski. Symmetric Spin. *International Journal on Software Tools for Technology Transfer*, 4(1):65–80, 2002.
3. E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. The MIT Press, Cambridge, Masachusetts, 1999.
4. E. Clarke, R. Enders, T. Filkhorn, and S. Jha. Exploiting symmetry in temporal logic model checking. *Formal Methods in System Design*, 9(1–2):77–104, 1996.

5. E. Clarke, T. Filkorn, and S. Jha. Exploiting symmetry in temporal logic model checking. In C. Courcoubetis, editor, *Proceedings of the Fifth International Conference on Computer Aided Verification (CAV '93)*, volume 697 of *Lecture Notes in Computer Science*, pages 450–461, Elounda,Greece, June/July 1993. Springer-Verlag.
6. A. F. Donaldson and A. Miller. Automatic symmetry detection for model checking using computational group theory. To appear in *Proceedings of the 13th International Symposium on Formal Methods (FM'05)*, *Lecture Notes in Computer Science*, Newcastle Upon Tyne, UK, July 2005. Springer-Verlag.
7. E. A. Emerson and A. P. Sistla. Utilizing symmetry when model-checking under fairness assumptions: An automata-theoretic approach. *ACM Trans. on Programming Languages and Systems*, 19(4):617–638, July 1997.
8. E. A. Emerson and R. J. Trefler. From asymmetry to full symmetry: New techniques for symmetry reduction in model checking. In L. Pierre and T. Kropf, editors, *Proceedings of the 10th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME '99)*, volume 1703 of *Lecture Notes in Computer Science*, pages 142–156, Bad Herrenalp, Germany, September 1999. Springer-Verlag.
9. E. A. Emerson, J. W. Havlicek, and R. J. Trefler. Virtual symmetry reduction. In *Proceedings of the fifteenth Annual IEEE Symposium on Logic in Computer Science*, pages 121–131, Santa Barbara, California, USA, June 2000. IEEE Computer Society Press.
10. E. A. Emerson and A. P. Sistla. Symmetry and model checking. *Formal Methods in System Design*, 9(1–2):105–131, August 1996.
11. V. Gyuris and A. Sistla. On-the-fly model checking under fairness that exploits symmetry. *Formal Methods in System Design*, 15(3):217–238, November 1999.
12. G. J. Holzmann. *The SPIN model checker: primer and reference manual*. Addison Wesley, Boston, 2003.
13. C. N. Ip and D. Dill. Better verification through symmetry. *Formal Methods in System Design*, 9:41–75, 1996.
14. A. P. Sistla and P. Godefroid. Symmetry and reduced symmetry in model checking. *ACM Transactions on Programming Languages and Systems*, 25(4):702–734, July 2004.
15. A. P. Sistla, V. Gyuris, and E. A. Emerson. SMC: A symmetry-based model checker for verification of safety and liveness properties. *ACM Transactions on Software Engineering and Methodology*, 9:133–166, 2000.

Modelling and Dynamic Symmetry Breaking in Constraint Programming

Karen E. Petrie

Cork Constraint Computation Center
University College Cork
Cork, Ireland
k.petrie@4c.ucc.ie

Abstract. Symmetry in constraint satisfaction problems can give rise to redundant search. The aim in symmetry breaking is to avoid such redundancy by excluding all but one example of each equivalence class of solutions. Two methods that have been developed to do this dynamically are Symmetry Breaking During Search and Symmetry Breaking via Dominance Detection. Modelling in CP means to move from a natural language specification of a problem, to a CSP formulation. This paper presents two case studies on the interaction between dynamic symmetry breaking and modelling.

1 Introduction

Combinatorial search is arguably the most fundamental aspect of Artificial Intelligence (AI) [2]. It is an extremely active research area, and has become very important commercially, through Constraint Programming (CP). Software packages such as ECL^{i}PS^e from IC-Parc [4] and ILOG Solver [17] are widely used on problems such as work force management at BT, resulting in savings of many millions for the companies concerned.

A Constraint Satisfaction Problem (CSP) consists of a set of variables each of which has a domain of values, and a set of constraints on the variables and values: a solution is an allocation of values to variables consistent with the constraints. A constraint solver *searches* for this solution by alternating phases of *branching* and *inference* to find an assignment of values to a set of variables which satisfies the constraints. The branching phase selects a variable and a possible value for it and seeks a solution in which it has that value. If no solution is found, then another value is tried. Branching thus causes the system to explore a tree of possible partial assignments, seeking one that can be completed. In the Inference phase, the solver attempts to deduce consequences of the constraint and the current partial assignment.

Modelling in CP means to move from a natural language specification of a problem, into a CSP instance consisting only of variables, values and constraints. It may be possible to find more than one model of a problem, in which case a model is sought that can efficiently lead to a solution through CSP solving techniques. This is where variable and value ordering heuristics fit into modelling process. This paper concentrates on the interaction of modelling and search with symmetry.

Constraint Satisfaction Problems (CSPs) are often highly symmetric. Symmetries may be inherent in the problem, as in placing queens on a chess board that may be rotated and reflected. Additionally the modelling of a real problem as a CSP can introduce extra symmetry: problem entities which are indistinguishable may in the CSP be represented by separate variables leading to $n!$ symmetries between n variables.

> **Definition of Symmetry** *Given a CSP L, with a set of constraints C, a symmetry of L is a bijective function f which maps a representation of a search state α to another search state, so that the following holds:*
> 1. *If α satisfies the constraints C, then so does $f(\alpha)$.*
> 2. *Similarly, if α is a no-good, then so too is $f(\alpha)$. [18]*

Symmetries can give rise to redundant search, while searching for solutions a partial assignment may be considered which is symmetric to one previously examined. If a partial assignment does not lead to a solution, neither will any symmetric assignment, and if it does lead to a solution, the new solution is symmetrically equivalent to one already found. To avoid this redundant search constraint programmers try to exclude all but one in each equivalence class of solutions. Many methods have been developed for this purpose. These symmetry exclusion methods can be divided into two classes: *static* and *dynamic*. Static symmetry breaking methods operate before search commences, and dynamic symmetry breaking methods operate during search.

In some classes of problems, the symmetry can be removed by remodelling the problem. For example, the golfers problem is: *32 golfers want to play in 8 groups of 4 each week, in such a way that any two golfers play in the same group at most once. How many weeks can they do this for?* This problem is highly symmetric. A possible model for this problem decides which group each player is assigned to in each week: the groups and the weeks (as well as the players) can be interchanged. By remodelling this problem using set variables, much of the symmetry can be removed [21].

Another static symmetry breaking method, involves adding constraints to the basic model. For instance, many problems (including the golfers problem above), have symmetry due to indistinguishable variables. Often, this symmetry can be removed by adding constraints that the value of these variables must be in ascending order. Crawford, Ginsberg, Luks and Roy developed a technique for constructing symmetry breaking ordering constraints for more general symmetries. It involves listing all possible permutations for each symmetry, then creating appropriate ordering constraints which allow only the first permutation to remain [5]. This technique affects the CP model both by the addition of constraints, and by fixing the variable ordering to be used during search.

In more recent years, Flener *et al.* have concentrated on symmetry constraints for *matrix models*; where "a matrix model is a constraint program that contains one or more matrices of decision variables" [7]. For example the golfers problem can be modelled as a 3-d boolean matrix whose dimensions correspond to weeks, players and groups. A variable $x_{ijk} = 1$ iff in week i, player j plays in group k [21]. The orderings constraints which are proposed deal with *row* and *column* symmetries, where a *row (column)* symmetry of a 2-d matrix is a bijection between the variables of two of its rows (columns) that preserve solutions and non-solutions. Two rows(columns) are *indistinguishable* if their variables are pairwise indistinguishable due to a row (column) symmetry. A matrix model has *row (column) symmetry* iff all the rows (columns) of one of its matrices are indistinguishable. In the above matrix model of the golfers problem, the groups, weeks and the players are all indistinguishable, this results in row (column) symmetries.

In contrast to static symmetry breaking methods, dynamic symmetry breaking methods operate during the search process. The two dynamic symmetry breaking methods we will concentrate on in this paper are, symmetry breaking during search [1, 13], and symmetry breaking via dominance detection [6, 8]. More recently, computational group theoretic versions of these methods have been devised, namely GAP-SBDS [12] and GAP-SBDD [14].

Symmetry breaking during search (SBDS), was developed by Gent and Smith [13], having been introduced by Backofen and Will [1]. The search tree is built from decision points, where a decision point has two possible choices; either assign a value to a variable, or do not assign that value to that variable. When a decision point is first reached during search a value is assigned to a variable; if at a later stage in search the decision point is revisited then a constraint is imposed that the variable should not have the previously assigned value. SBDS operates by taking a list of symmetry functions (provided by the user) and placing related constraints when backtracking to a decision point and taking the second branch.

A feature of SBDS is that it only breaks symmetries which are not already broken in the current partial assignment: this avoids placing unnecessary constraints. A symmetry is broken when the symmetric equivalent of the current partial assignment is not consistent with that assignment. The following expression explains how SBDS works:

$$A \ \& \ g(A) \ \& \ var \neq val \ \Rightarrow g(var \neq val)$$

where A is the partial assignment made so far during search, $g(A)$ is the symmetric equivalent of A and $g(var \neq val)$ is the symmetrical equivalent to this failed assignment. If A is the current partial assignment and it has been established that $var \neq val$, it needs to be ensured that an unbroken symmetry is being dealt with, so a check is undertaken that $g(A)$ still holds. Then to ensure that the symmetrically equivalent subtree to the current subtree will not be explored, the constraint $g(var \neq val)$ is placed. An SBDS library is now available in the ECL^{i}PS^e constraint programming system [4]. As previously mentioned, SBDS requires a function for each symmetry in the problem describing its effect on the assignment of a value to a variable. If these symmetry functions are correct and complete, all the symmetry will be broken; as a result of this only non-isomorphic solutions will be produced. Although SBDS has been successfully used with a few thousand symmetry functions, many problems have too many symmetries to allow a separate function for each.

To allow SBDS to be used in situations where there are too many symmetries to allow a function to be created for each, Gent *et. al.* [12] have linked SBDS in ECL^{i}PS^e with GAP (Groups, Algorithms and Programming) [10], a system for computational algebra and in particular *computational group theory* (CGT). Group theory is the mathematical study of symmetry. GAP-SBDS allows the symmetry group rather than its individual elements to be described. GAP is used when a value is assigned to a variable, at a decision point, to find the *stabiliser* of the current partial assignment, i.e. the subgroup which leaves it unchanged. Then if the decision point is revisited on backtracking, the constraints are dynamically calculated from the stabiliser and placed accordingly. GAP-SBDS allows the symmetry to be handled more efficiently than in SBDS; the elements of the group are not explicitly created which is akin to what the symmetry functions represent in SBDS. However, there is an overhead in communication necessitated between GAP and ECL^{i}PS^e.

Symmetry Breaking via Dominance Detection (SBDD) [6, 8] performs a check at every node in the search tree to see if it is dominated by a symmetrically equivalent subtree already explored, and if so prunes this branch. In SBDD, the dominance detection function is based on the problem symmetry and is hard-coded for each problem. This means in practice SBDD can be difficult to implement, as the design of the dominance detection function may be complicated; the user has to ensure that all the symmetry of the problem is incorporated within the function to enforce full symmetry breaking.

Gent *et. al.* [14] have recently developed GAP-SBDD, a generic version of SBDD that uses the symmetry group of each problem rather than an individual dominance detection function and links SBDD (in ECL^{i}PS^e) with GAP. At each node in the search tree, ECL^{i}PS^e communicates the details of that node to GAP, and GAP returns false if dominance has been detected and that branch can be pruned, or true otherwise. Occasionally full dominance is not detected but there are variable/value pairs which are easily detected as being eligible for domain deletion; at which point GAP returns true followed by a list of variable/value pairs for which this is the case. ECL^{i}PS^e removes these values from the corresponding variables domains before search continues.

It is clear that static symmetry breaking methods affect the choice of model for a CSP. This situation is less clear for dynamic symmetry breaking methods. In general, dynamic symmetry breaking methods do not fix the CSP model, the only proviso is that the symmetry should be definable in terms of the search variables. This paper presents two cases studies which show how dynamic symmetry breaking and modelling techniques can interact. The first study shows that by considering both the model of the problem and the chosen symmetry breaking method an efficient method can be derived. The second study shows how the model chosen for a given problem can affect the choice of most efficient dynamic symmetry breaking method.

2 Case Study: SBDS and ‘Peaceable Armies of Queens’

Robert Bosch introduced the “Peaceably Coexisting Armies of Queens” problem in his column in Optima in 1999 [3]. It is a variant of a class of problems requiring pieces to be placed on a chessboard, with requirements on the number of squares that they attack: Martin Gardner [11] discusses more examples of this class. In the “Armies of Queens” problem, we are required to place two equal-sized armies of black and white queens on a chessboard so that the white queens do not attack the black queens (and necessarily v.v.) and to find the maximum size of two such armies. Bosch asked for an integer programming formulation of the problem and how many optimal solutions there would be for a standard 8×8 chessboard.

A straightforward model of the problem has a variable s_{ij} to represent a square on row i, column j of the board:

$$\begin{aligned} s_{ij} &= 1 \text{ if there is a white queen on square } (i,j) \\ &= 2 \text{ if there is a black queen on square } (i,j) \\ &= 0 \text{ otherwise} \end{aligned}$$

If M is the region that may be attacked by a given square, then we can express the 'non-attacking' constraints as:

$$\begin{aligned} s_{i_1 j_1} &= 1 \Rightarrow s_{i_2 j_2} \neq 2 \\ \text{and } s_{i_1 j_1} &= 2 \Rightarrow s_{i_2 j_2} \neq 1 \text{ for all } ((i_1, j_1), (i_2, j_2)) \in M \end{aligned}$$

or more compactly as:

$$s_{i_1 j_1} + s_{i_2 j_2} \neq 3 \text{ for all } ((i_1, j_1), (i_2, j_2)) \in M$$

Tests in ECL^{i}PS^e show that, the single constraint gives the same number of backtracks as the two implication constraints, but is faster.

Constrained variables w, b count the number of white and black queens respectively (using the counting constraint: occurrences, provided in ECL^{i}PS^e). The last constraint is $w = b$, and the objective is to maximise w. This is achieved by adding a lower bound on w whenever a solution is found, so that future solutions must have a larger value of w; when there are no more solutions, the last one found has been proved optimal.

The model has n^2 search variables and approximately $4n^3$ binary constraints, as well as the counting constraints which have arity n^2, where n is the number of rows in the board.

	Finding Optimal				Finding All Optimal Solutions		
n	No. of Backtracks to find first optimal solution	Total Number of Backtracks	Optimal Number of Queens	Time (secs)	Number of Backtracks	Number of Solutions	Time (secs)
2	0	1	0	0.0	1	1	0.0
3	1	2	1	0.0	17	16	0.0
4	4	28	2	0.01	149	112	0.02
5	190	265	4	0.16	383	18	0.20
6	1344	4998	5	3.63	9623	560	5.24
7	21882	93532	7	87.95	189013	304	132.99
8	802255	2716158	9	3215.2	-	-	-

Table 1. Results: Basic Model with no Symmetry Breaking

Table 1 gives results for finding the optimal number of queens and proving that it is optimal, as well as for finding all optimal solutions. These experiments were run with a simple static variable ordering heuristic which searches the board: top row, left to right, then second row, left to right, and so on. The value ordering heuristic is the standard ECL^{i}PS^e one, which assigns values in numerical order starting with the smallest. The result for finding all solutions when $n = 8$ are missing as this result was not obtainable within the cut-off imposed of 1 hour.

2.1 SBDS in 'Armies of Queens'

The 'Armies of Queens' problem has the usual symmetry of the chessboard (reflection in the horizontal, vertical and both diagonal axes, and rotations through 90°, 180° and 270° and the identity); in addition, in any solution we can swap all the white queens for all the black queens, and we can combine these two kinds of symmetry. Hence the problem has 16 symmetries. SBDS is ideal for problems such as this since it only requires a simple function to describe the effect of each symmetry (other than the identity) on the assignment of a value to a variable. Hence, in this case, just 15 such functions are required.

The seven chessboard symmetry functions are labelled x, y, $d1$, $d2$, $r90$, $r180$, $r270$. The function which interchanges black and white is labelled BW; and the functions which combine the chessboard symmetries with interchanging black and white, are labelled as the board symmetries prefixed with BW. The symmetry functions take a variable, s_{ij} and a possible value for this variable, v before returning the symmetric variable and the symmetric value as:

$$\begin{aligned}
x &: s_{ij}, v \rightarrow s_{i,n+1-j}, v \\
y &: s_{ij}, v \rightarrow s_{n+1-i,j}, v \\
d1 &: s_{ij}, v \rightarrow s_{j,i}, v \\
d2 &: s_{ij}, v \rightarrow s_{n+1-j,n+1-i}, v \\
r90 &: s_{ij}, v \rightarrow s_{j,n+1-i}, v \\
r180 &: s_{ij}, v \rightarrow s_{n+1-i,n+1-j}, v \\
r270 &: s_{ij}, v \rightarrow s_{n+1-j,i}, v \\
bw &: s_{ij}, v \rightarrow s_{i,j}, [if\ v = 0\ then\ 0\ else\ 3 - v] \\
bwx &: s_{ij}, v \rightarrow s_{i,n+1-j}, [if\ v = 0\ then\ 0\ else\ 3 - v] \\
bwy &: s_{ij}, v \rightarrow s_{n+1-i,j}, [if\ v = 0\ then\ 0\ else\ 3 - v] \\
bwd1 &: s_{ij}, v \rightarrow s_{j,i}, [if\ v = 0\ then\ 0\ else\ 3 - v] \\
bwd2 &: s_{ij}, v \rightarrow s_{n+1-j,n+1-i}, [if\ v = 0\ then\ 0\ else\ 3 - v] \\
bwr90 &: s_{ij}, v \rightarrow s_{j,n+1-i}, [if\ v = 0\ then\ 0\ else\ 3 - v] \\
bwr180 &: s_{ij}, v \rightarrow s_{n+1-i,n+1-j}, [if\ v = 0\ then\ 0\ else\ 3 - v] \\
bwr270 &: s_{ij}, v \rightarrow s_{n+1-j,i}, [if\ v = 0\ then\ 0\ else\ 3 - v]
\end{aligned}$$

Suppose that $n = 8$ and the first assignment places a white queen in the top left corner: $s_{1,1} = 1$. The symmetric assignments are: $x : s_{1,8} = 1$, $y : s_{8,8} = 1$, $d1 : s_{1,1} = 1$,

$d2 : s_{8,1} = 1$, $r90 : s_{1,8} = 1$, $r180 : s_{8,8} = 1$, $r270 : s_{8,1} = 1$, $bw : s_{1,1} = 2$, $bwx : s_{1,8} = 2$, $bwy : s_{8,1} = 2$, $bwd1 : s_{1,1} = 2$, $bwd2 : s_{8,8} = 2$, $bwr90 : s_{1,8} = 2$, $bwr180 : s_{8,8} = 2$, $bwr270 : s_{8,1} = 2$. All the symmetries which swap black and white, apart from *bw* are inconsistent with $s_{1,1} = 1$, because the symmetrically equivalent assignment would place a black queen in one of the corners where it could be attacked by the first assignment, so these symmetries are no longer considered on this branch. On backtracking to the first choice point, where $s_{1,1} = 1$ is set, and taking the alternative branch of $s_{1,1} \neq 1$, the symmetry functions are used to calculate the symmetric variables (*SymVar*) and values (*SymVal*). Lastly constraints of the form $SymVar \neq SymVal$ are placed in order to stop the subtree symmetric to this from ever being explored. This process ensures that if a white queen can not be placed in the top corner, then a queen is never placed in any of the corners.

	Finding Optimal				All Solutions		
n	No. of Backtracks to find first optimal solution	Total Number of Backtracks	Optimal Number of Queens	Time (secs)	Number of Backtracks	Number of Solutions	Time (secs)
2	0	1	0	0.1	1	1	0.0
3	1	2	1	0.03	2	1	0.03
4	4	9	2	0.10	16	10	0.10
5	68	70	4	0.60	64	3	0.52
6	462	886	5	7.30	1286	35	9.19
7	6994	15538	7	138.16	24106	19	181.310
8	298235	473141	9	4454.45	-	-	-

Table 2. Results: Basic Model with SBDS

Table 2 shows the empirical results when SBDS is integrated into the simple CP model outlined in Section 2. Comparing this with Table 1 shows that SBDS gives a factor greater than 5 improvement in number of backtracks for the $n = 8$ case. However, the runtime increases when SBDS is used. This is because the first value chosen by the value ordering heuristic represents an empty square on the chessboard. The symmetry breaking constraints placed by SBDS when backtracking from these assignments, will forbid placing an empty square in a symmetrically equivalent position. These constraints occur an overhead and are not useful in steering search towards improved solutions. In fact as better, solutions with more queens on the board are found they become redundant. Later on in search, when leaving empty squares has been tried, values 1 then 2 will be allocated, which relate to placing white and black queens respectively. When SBDS is triggered through backtracking past failed cases of these assignments more useful constraints are returned. These constraints are the ones that operate to reduce the number of backtracks so significantly. In general, when trying to anticipate the effect of SBDS on a given model, it is worth considering the variable and value ordering heuristics. If these heuristics will lead to placing constraints early in search which, will have little effect at the time, then become vacuous at a later stage of search, it is worth considering if a better heuristic can be found.

2.2 Value Ordering and SBDS

The value ordering heuristic which places empty squares first can also hinder the optimisation process. The first solution to be found has 0 allocated to every square, which is equivalent to an empty board. This gives a lower bound of 0 for the maximum number of white queens which can be placed on the board. A constraint is then posted which says that the next number of white queens must be greater than this lower bound which in this case would be > 0. The process continues by increasing the lower bound in integer increments until the optimum number (m) is found. At this point, the program searches for a solution with maximum number of white queens $m+1$; on failing to find one it has proven that m is indeed the optimum. If instead of allocating empty squares in the initial stages, queens are placed on squares first, the earliest solution found gives, a better lower bound for the optimum. In this case the program commences by placing as many white queens as possible then as many black queens as possible, only allocating empty squares when no queens can be placed. The lower bound then becomes the number of black or white queens (there is a constraint to ensure they are equal) on the board (p). Optimisation continues as before, by setting a constraint which states that the next value found must be greater than p. This value ordering heuristic is also potentially a good heuristic with respect to SBDS. The first decisions made relate to placing queens on the board, if these decisions are backtracked past at a later stage, than SBDS can place constraints which state that a queen should not be placed in the given square. These constraints are useful in directing search. In optimisation problems, by considering the best heuristic for a problem through knowledge of the optimisation process, a good heuristic for SBDS may also be derived, as the extra information given to the optimisation process can relate to SBDS placing more informative symmetry breaking constraints. In general, by considering the best heuristics for a given problem, a good heuristic will also be found with respect to SBDS, as the heuristic chosen will build a search tree which starts by trying the mostly likely value for a variable, this relates to the scope of constraints that SBDS can place.

It is possible to implement this new strategy as a value ordering heuristic which tries 1 before 2, before 0; hence it implements allocating queens to squares on the board before leaving them empty. However, this heuristic does have a time overhead as a decision process has to be undertaken at each search variable to see which value should be allocated. A less complex approach is to reassign the values so that $0 = \textit{white queen}$, $1 = \textit{black queen}$ and $2 = \textit{empty square}$. Then allocate 0 before 1, before 2 as before. In SBDS this approach does necessitate a minor change to the symmetry functions which interchange black and white queens.

2.3 Variable Ordering and SBDS

In the previous experiments in Section 2.1, a static variable ordering heuristic was used which assigned the top row of the board from left to right followed by the second row from left to right until all the variables were assigned. If constraints were being used to break the symmetry this static ordering may be mandatory, as often the variable order must be defined before search commences, in order to ensure these methods are complete and no solutions are lost. If SBDS is the symmetry breaking method chosen, this information is not needed before search commences, so the use of dynamic variable ordering is permitted, and can be easily integrated with the SBDS library. A dynamic variable ordering chooses the next variable to be allocated during search, according to the search decisions and the resulting propagation to that point. A common and well proven heuristic is smallest domain first (SDF), which allocates the next variable to be assigned a value to be the one with the smallest number of entries in its domain.

2.4 Experimental Results of Combining Variable and Value Ordering Heuristics with SBDS

Table 3 contains the results of combining the value ordering heuristic outlined in Section 2.2 and SDF variable ordering as discussed in Section 2.3 with SBDS.

	Finding Optimal					All Solutions	
n	Lower bound on optimum	Optimum No. of Queens	No. of Bt. to find first optimal solution	Total Number of Bt.	Time (secs)	Number of Bt.	Time (secs)
2	0	0	0	1	0.0	1	0.0
3	1	1	0	2	0.02	2	0.0
4	2	2	0	4	0.05	12	0.05
5	3	4	3	12	0.17	23	0.22
6	4	5	1	153	1.79	405	3.16
7	5	7	9	2231	21.23	5186	47.90
8	6	6	266	46894	406.39	106940	752.11

Table 3. Results: SBDS with SDF Variable Ordering & Value Ordering heuristic

Comparing the previous results for the basic model with SBDS shown in Table 2 with the more advanced model results shown in Table 3, shows a large reduction in time for all cases. For $N = 8$ the reduction in time is 10 fold to find the optimal number of queens that can be placed on the board. The reduction in the total number of backtracks for $N = 8$ is equally impressive at 10 fold again, but the most impressive reduction comes in the number of backtracks to find the first optimal solution which is reduced by a factor greater than 1000 for the $N = 8$ case. This means that good lower bounds for the optimum are being found early in search. In the $N = 8$ case it can be seen that the lower bound is 6 for the new value ordering, whereas it was 0 in the original case, the actual optimal value is 9 so 6 is a good approximation.

Turning to finding all the solutions to the problem it can be seen that there is a great reduction in both backtracks and time, between the original model and the current model. It is possible to prove that there are 71 non-isomorphic results for $N = 8$, with the approved variable and value ordering heuristic, this is a new result.

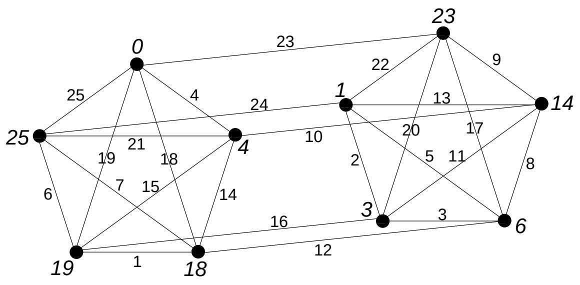


Fig. 1. Graceful labelings of $K_5 \times P_2$ and the Double Wheel DW_5

Looking back at the results without SBDS (Table 1) it can be seen that these new results outperform those, both in terms of time and backtracks, in all cases. This shows that the combination of modelling techniques and the SBDS library can be very powerful in efficiently solving problems.

3 Case Study: SBDS versus SBDD and 'Graceful Graphs'

There is limited past work comparing GAP-SBDS and GAP-SBDD. Harvey [15] studied the algorithms theoretically and concluded that SBDS and SBDD are closely related, the difference being where in the search tree, and how, symmetry breaking is enforced. Gent *et al.* [14] applied GAP-SBDS and GAP-SBDD to instances of the balanced incomplete block design (BIBD) problem and showed that GAP-SBDD could solve much larger problems, and was faster than GAP-SBDS on the smaller problems which both could solve. They surmised that this was due to the communication overhead between GAP and ECL^{i}PS^e, since the overhead in GAP-SBDD, which usually returns only a Boolean answer, is less than in GAP-SBDS, where a set of constraints is returned.

On the other hand, Petrie and Smith [19] found that in *Graceful Graphs* problems, GAP-SBDS outperformed GAP-SBDD on all instances studied. In the next section, the reason for this difference in performance is identified.

3.1 Graceful Graphs

A labeling f of the vertices of a graph with q edges is *graceful* if f assigns to each vertex a unique label from $\{0,1,...,q\}$ and, when each edge xy is labeled with $|f(x)-f(y)|$, the edge labels are all different [9]. (Hence, the edge labels are a permutation of $1,2,...,q$.) Figure 1 shows an example.

Lustig and Puget [16] give a constraint model for finding a graceful labeling of a graph. A basic CSP model has a variable for each node $x_1,x_2,...,x_n$, each with domain $\{0,1,...,q\}$ and a variable for each edge $d_1,d_2,...,d_q$, each with domain $\{1,2,...,q\}$. The constraints of the problem are: if edge k joins nodes i and j then $d_k = |x_i - x_j|$; $x_1,x_2,...,x_n$ are all different; and $d_1,d_2,...,d_q$ are all different.

ECL^{i}PS^e provides two different levels of propagation for the *alldifferent* constraint. It can either be treated as a clique of binary $\neq$ constraints or as a *global alldifferent*

		BT	ECL^{i}PS^e time	GAP time	Total time
GAP-SBDD	$K_3 \times P_2$	13	0.23	0.50	0.73
	$K_4 \times P_2$	173	7.18	2.72	9.90
	$K_5 \times P_2$	4402	337.69	88.20	426.89
GAP-SBDS	$K_3 \times P_2$	9	0.20	0.33	0.53
	$K_4 \times P_2$	165	7.15	1.35	8.50
	$K_5 \times P_2$	4390	352.10	36.61	388.71

Table 4. Comparison of GAP-SBDS and GAP-SBDD showing backtracks (bt) and the time (in seconds) for finding all graceful labelings of $K_3 \times P_2$, $K_4 \times P_2$, $K_5 \times P_2$.

which does more propagation. We use the *global alldifferent* on the edge variables and the binary $\neq$ version on the node variables. They are treated differently because the values assigned to the edge variables form a permutation and hence give more scope for domain pruning than the node variables, which have more possible values than variables. The node variables are used as the search variables. More information on the modeling of this problem and the symmetry group is given by [19].

The graph $K_5 \times P_2$, shown in Figure 1, consists of two copies of K_5, with corresponding vertices in the two cliques forming the vertices of a path P_2. The symmetries of $K_5 \times P_2$ are: first, any permutation of the 5-cliques which act on both in the same way. Second, inter-clique symmetry: all the node labels in the first clique can be interchanged with the labels of the adjacent nodes in the second. Third, complement symmetry: every vertex label x_i can be replaced by its complement $q - x_i$. The graph symmetries and the complement symmetry can be combined with each other. Hence, the size of the symmetry group is $5! \times 2 \times 2$. In general, $K_m \times P_2$ graphs have a symmetry group of size $m! \times 2 \times 2$. This study concentrates on symmetry breaking in 3 such graphs, with $m = 3$, 4 and 5. The results of finding all graceful labelings of these graphs using either GAP-SBDS or GAP-SBDD can be found in Table 1. (All experiments in the paper were run on a 1.6GHz Pentium 4 processor with 512MB of memory, using ECL^{i}PS^e version 5.7 and GAP version 4.2.) From Table 4, it can be seen that GAP-SBDD is slower than GAP-SBDS for all instances. This is also true for other graphs, as shown by [20].

3.2 Analysis

To explain why GAP-SBDS is faster than GAP-SBDD for finding graceful labelings of graphs with symmetry, we have analysed the behaviour of GAP-SBDS and GAP-SBDD for the three graphs $K_3 \times P_2$, $K_4 \times P_2$ and $K_5 \times P_2$. The reasons for the differences in search are consistent, but for simplicity only the results for $K_3 \times P_2$ are presented here.

It should be noted that Table 1 gives the number of *deep backtracks*. We use the term deep backtrack when the search has progressed beyond a decision point, but then later has to revisit it. A *shallow backtrack* occurs when propagating the constraint $var = val$ on the left branch of a decision point causes a failure, and the $var \neq val$ branch is taken instead. Most constraint programming systems count the number of deep backtracks, but in this case it does not accurately reflect differences in search. In GAP-SBDS, symmetry-breaking constraints can be added whenever the left ($var \neq val$) branch is followed, including after a shallow backtrack.

Figure 3 shows the search trees created by GAP-SBDS and GAP-SBDD in finding all graceful labelings of $K_3 \times P_2$, from the point where the first difference occurs, which is after the first two solutions (from 4 in total) have been found. The variable names A to E in Figure 3 correspond to the nodes shown in Figure 2; the edges and corresponding edge variables are named by a pair of letters corresponding to the nodes defining the edge.

After assigning $C = 5$, GAP-SBDS immediately reverses from this decision to follow the $C \neq 5$ branch (a shallow backtrack), whereas GAP-SBDD continues, setting $E = 1$, before returning to take the $C \neq 5$ branch later in search (a deep backtrack).

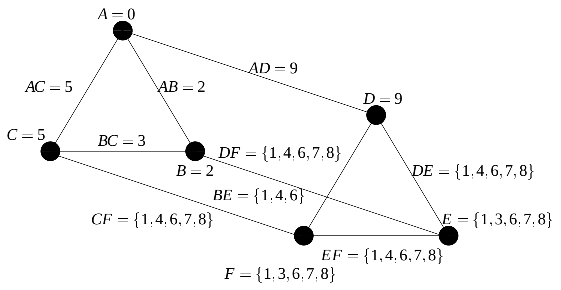


Fig. 2. The domains of the node and the edge variables after propagating $C = 5$, using GAP-SBDD

The difference in the search trees is due to differences in constraint propagation. GAP-SBDD arrives at the search state shown in Figure 2. One of the edges must be labeled 9 (the number of edges in the graph) and the adjacent nodes must be labeled 0 and 9. At this stage $A = 0$ and B and C are labeled with values other than 9; hence D, the only other node adjacent to A, must take the value 9, and this inference is made by constraint propagation. Figure 2 shows the variable domains at this point. Because there are already edges labeled 2 (AB) and 3 (BC), the edges DE and DF cannot have those values, and hence E and F cannot have the values 6 or 7. Using GAP-SBDS, the domains of the variables are also reduced by symmetry-breaking constraints previously added on this branch. Those that are relevant in this case are symmetric equivalents of $B \neq 1$, namely $E \neq 1$, $F \neq 1$, $E \neq 8$ and $F \neq 8$. (Because of the graph symmetry, nodes E and F are symmetric to node B, and the value 8 is symmetric to the value 1 because of the complement symmetry.) The only remaining value in the domains of both E and F is 3, and since these variables must have different values, this branch fails.

Most of this propagation cannot occur in GAP-SBDD. GAP just returns a boolean to indicate whether the current node is dominated or not, and possibly a list of values to prune from the domains of specific search variables. In the current implementation, a variable/value pair is returned for domain pruning if its assignment would cause dominance to be detected. In this case $E/1$, $F/1$, $E/8$ and $F/8$ are not returned. Although

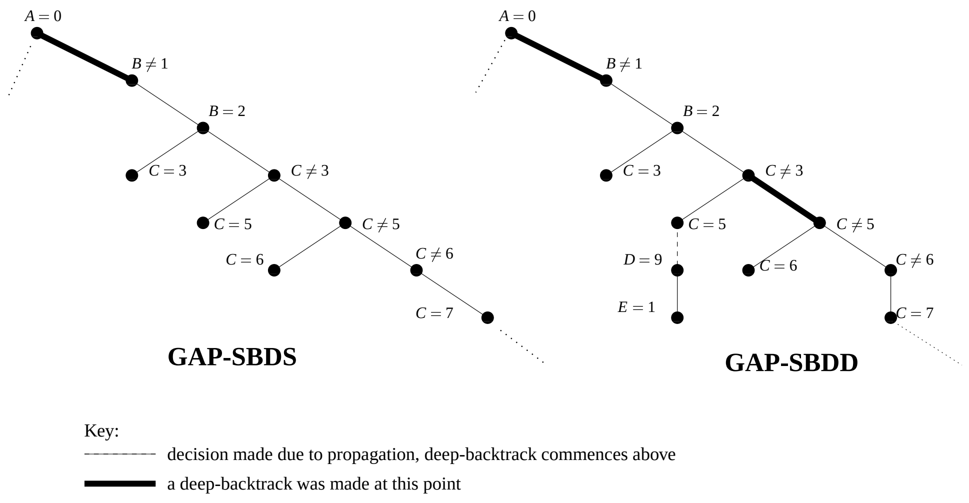


Fig. 3. The search tree branch where GAP-SBDS and GAP-SBDD differ

GAP-SBDD successfully breaks the symmetry (in this case by detecting dominance when the assignment $E = 1$ is tried) posting SBDS constraints at an earlier stage can clearly lead to earlier pruning.

The reason this difference between GAP-SBDS and GAP-SBDD is highlighted by experimentation on this problem, as opose to on the other problems consider by Gent *et al.* [14], relates directly to the model of the problem; specifically to the fact that the search variables are not the most constrained variables with the model. GAP-SBDS breaks symmetry by placing constraints, these constraints can propogate with all the variables within the model. GAP-SBDD provides no information which could be related to the variables not directly involved in search.

4 Conclusion

Symmetry exclusion methods can be divided into two classes: static and dynamic. Static symmetry breaking methods operate before search commences, whereas dynamic symmetry breaking methods operate during search. Static symmetry breaking methods generally require a fixed model with static variable and value ordering heuristics. Dynamic symmetry breaking methods leave the CP practitioner with more freedom as to which model to chose; the only proviso is that it must be possible to define the symmetry in terms of the search variables.

In this paper, through the use of two case studies, we have shown how the CP model can interact with dynamic symmetry breaking methods. The first case study illustrated how dynamic symmetry breaking and modelling can interact to provide an efficient method for solving a problem. The second case study shows how the model chosen for a given problem, can affect the choice of most efficient dynamic symmetry breaking method.

This paper represents a preliminary study, showing that, by combining modelling techniques and dynamic symmetry breaking, more efficient solving techniques can be derived than by considering either of these aspects individually. Further work in this area is needed if the exact relationship between symmetry breaking and modelling is to be fully understood.

Acknowledgements

The author would like to thank Barabara Smith, Ian Gent, Tom Kelsey and Steve Linton for their continuing encouragement and interest. I am also very grateful to Warwick Harvey for his technical assistance with ECL^{i}PS^e, and Alex Ferguson for offering his expert Latex knowledge. This work was supported partially by EPSRC grant GR/R29673, and Science Foundation Ireland grant 00/PI.1/C075.

References

1. R. Backofen and S.Will. Excluding symmetries in constraint-based search. In Joxan Jaffar, editor, *Proc. of CP'99*, LNCS 1713, pages 73–87. Springer, 1999.
2. Roman Bartak. Guide to Constraint Programming. Technical report, Charles University Prague, 1998.
3. Robert A. Bosch. Peaceably coexisting armies of queens. *Optima (Newsletter of the Mathematical Programming Society)*, 62:6–9, 1999.
4. A. M. Cheadle, W. Harvey, A. J. Sadler, J. Schimpf, K. Shen, and M. G. Wallace. ECLiPSe: An introduction. Technical Report IC-Parc-03-1, IC-Parc, 2003. `www.icparc.ic.ac.uk/eclipse/`.
5. James Crawford, Matthew L. Ginsberg, Eugene Luks, and Amitabha Roy. Symmetry-breaking predicates for search problems. In Luigia Carlucci Aiello, Jon Doyle, and Stuart Shapiro, editors, *KR'96: Principles of Knowledge Representation and Reasoning*, pages 148–159. Morgan Kaufmann, San Francisco, California, 1996.
6. Torsten Fahle, Stefan Schamberger, and Meinolf Sellmann. Symmetry breaking. In Toby Walsh, editor, *Proc. of CP'01*, LNCS 2239, pages 93–107. Springer, 2001.
7. P. Flener, A.M. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, J. Pearson, and T. Walsh. Breaking row and column symmetries in matrix models. In P. Van Hentenryck, editor, *Proc. of CP'02*, LNCS 2470, pages 462–476. Springer, 2002.
8. Filippo Focacci and Michela Milano. Global cut framework for removing symmetries. In Toby Walsh, editor, *Proc of CP'01*, LNCS 2239, pages 77–92. Springer, 2001.
9. J.A. Gallian. A Dynamic Survey of Graceful Labeling. In *The Electronic Journal of Combinatronics*, 2002. (`http://www.combinatorics.org/Surveys`).
10. The GAP Group. *GAP – Groups, Algorithms, and Programming, Version 4.2*, 2000. (`http://www.gap-system.org`).
11. Martin Gardner. Chess queens and maximum unattacked cells. *Math Horizon*, pages 12–16, November 1999.
12. I. P. Gent, W. Harvey, and T. Kelsey. Groups and constraints: Symmetry breaking during search. In P. Van Hentenryck, editor, *Proc. of CP'02*, LNCS 2470, pages 415–430. Springer, 2002.

13. I. P. Gent and B. M. Smith. Symmetry breaking in constraint programming. In *Proc. of ECAI-2002*, pages 599–603. IOS Press, 2000.
14. Ian P. Gent, Warwick Harvey, Tom Kelsey, and Steve Linton. Generic SBDD Using Computational Group Theory. In Francesca Rossi, editor, *Proc. of CP'03*, LNCS 2833, pages 333–347. Springer, 2003.
15. Warwick Harvey. Symmetry Breaking and the Social Golfer Problem. In *Proc. SymCon-01: Symmetry in Constraints*, pages 9–16, 2001.
16. I.J.Lustig and J.-F. Puget. Program Does Not Equal Program: Constraint Programming and Its Relationship to Mathematical Programming. In *INTERFACES*, volume 31(6), pages 29–53, 2001.
17. ILOG. *ILOG Solver 5.0 User's Manual*, 2001.
18. I. McDonald and B. M. Smith. Partial symmetry breaking. In *Proc. of CP'02*, LNCS 2470, pages 431–445. Springer, 2002.
19. K. E. Petrie and B. M. Smith. Symmetry breaking in graceful graphs. In *Proc. of CP'03*, LNCS 2833, pages 930–934. Springer, 2003.
20. Karen Petrie. Why SBDD can be worse than SBDS. In *Proc. SymCon-03: Symmetry in Constraints*, pages 168–176, 2003.
21. Barbara M. Smith. Reducing Symmetry in a Combinatorial Design Problem. Technical report, School of Computer Studies, University of Leeds, January 2001.

Approaches to Symmetry Breaking for Weak Symmetries

Roland Martin

Algorithmics Group
Darmstadt University of Technology
64283 Darmstadt, Germany
`martin@algo.informatik.tu-darmstadt.de`

Abstract. In this paper we consider a kind of symmetry, which we call *weak symmetry*.
Weak symmetries occur in different application fields like planning, scheduling and model checking as well as in extensions of classical problems.
In contrast to a proper symmetry, a weak symmetry of a constraint satisfaction problem acts only on a subset of the variables and preserves the feasibility state only with respect to a subset of the constraints.
We discuss a reformulation concept where we use additional variables which we call *SymVar (Symmetry Variable)*. These variables enable us to exploit weak symmetries and achieve symmetry breaking on the symmetric variables of the problem without losing solutions.
Roughly speaking by using SymVars we rearrange the search tree in a way such that all symmetric solutions of an equivalence class are arranged under a specific node.
We also present results for a relaxed real-world problem from the automated manufacturing. Therefore, we compared our approach to a standard approach for the problem.

1 Introduction

Symmetries of a constraint satisfaction problem transform a (partial) solution into a symmetric (partial) solution and preserve the state of feasibility: no-goods are tranformed into symmetric no-goods while feasible solutions are transformed into symmetric feasible solutions. Therefore, symmetries decompose the search space into classes of symmetric solutions, whereby each class either contains feasible solutions only or infeasible solutions only.

When searching for all solutions to a problem it is sufficient to find only one solution in each class of solutions. The symmetric equivalents can be derived by applying the symmetry function exhaustively to each class after the search process. Therefore symmetries should be excluded from the search space to speed up the search.

[1] In cooperation with Philips/Assembléon,Netherlands

Various techniques have been proposed for symmetry handling. In general it is done by reformulation of the model, excluding the symmetry up-front via constraints, breaking it during the search or by a combination thereof.

Weak symmetries act only on a subset of the variables and respect only a subset of the constraints of the problem. Therefore, weak symmetries preserve the state of feasibility only with respect to the subset of variables they act on and only for the constraints they respect. That means that if two solutions are symmetric under the weak symmetry they yield different full solutions. As a consequence weak symmetries do not decompose the search space into classes of symmetric solutions.

But weak symmetries cannot be simply be excluded, since this would result in a loss of solutions that cannot be derived afterwards. Nonetheless we will present a technique that enables us to deal with weak symmetries such that they can be broken without losing solutions.

The above mentioned kinds of symmetry handling are also possible for weak symmetries once they are dealt with accordingly.

Weak symmetries occur in many fields of applications and are already discovered and identified in planning, scheduling and model checking. (See [1], [2] and [3], [4]). Also extensions to classical problems like the rack configuration problem (see [5] and [6]) contain weak symmetries. To our best knowledge up to now there is no practical approach that tackles the problem and shows results.

This article introduces a reformulation strategy that addresses the problem. The benefit in solving the problem by reformulation is that it can be applied without changing or adjusting the used solver for the problem or write further code to tackle the problem. Furthermore one is not bound to a specific solver when using the reformulation approach. And as mentioned before all other kinds of symmetry handling are possible in conjunction with this approach.

Throughout the paper we will consider the *n*-queens problem as a small example to explain our ideas. In the *n*-queens problem the task is to place *n* chess queens on an $n \times n$ chessboard such that no two queens can attack another.

The results presented are obtained from two scenarios of a relaxed real-world problem from the fields of automated manufacturing.

Section 2 states the definitions and consequences of weak symmetry and SymVars. The benefit and tradeoff of using SymVars for weak symmetry breaking are discussed in Section 3. Section 4 states the problem description and results of the scenario we use to investigate weak symmetries. Also the results are discussed in this section. Section 5 concludes with an outlook to further work.

2 Weak Symmetry Description

2.1 Prerequisites

We characterize a satisfaction problem by $P = (X,C)$, whereby $X = \{x_1,\dots,x_n\}$ is the set of variables and $C = \{c_1,\dots,c_m\}$ is the set of constraints.

For an optimisation problem we just extend this formulation to $P = (X,C,f)$, whereby X and C are defined like above and $f = f(x_1,\dots,x_n)$ is the objective function. [2]

A solution to P is denoted by $s_P = (X)$. This means that each variable in X is assigned a variable of its corresponding domain.

As mentioned before, we will consider a variation of the n-queens problem. The variation is that each field of the chessboard yields a certain weight. Wanted is an n-queen placement such that the sum of the weights achieved by this placement is maximised.

Variables and additional data:

- $w_{ij}, (i,j \in \{1,\dots,n\})$ weights on the board
- $q_1,\dots,q_n$ (variables for the queens)
- obj (variable for the objective value)

Constraints:

- $alldifferent(q_1,\dots,q_n)$ (no two queens in the same column)
- $q_i + i \neq q_j + j$ (for any anti-diagonal: no two queens can attack each other)
- $q_i - i \neq q_j - j$ (for any diagonal: no two queens can attack each other)
- $obj = \sum_{i \in \{1..n\}} w_{i,q_i}$ (the objective function)

A simple example would be to evaluate all white fields with 0 and the black fields with 1. This would imply to maximise the number of queens on black fields.

2.2 Weak Symmetry Definition

Weak symmetries act on problems with special properties. To characterize weak symmetries we first define weakly decomposable problems.

Definition 1 (Weakly Decomposable Problem)
A problem $P = (X,C)$ is **weakly decomposable** *if it decomposes into two subproblems $P_1 = (X_1,C_1)$ and $P_2 = (X_2,C_2)$ with the following properties:*

$$X_1 \cap X_2 \neq \emptyset \tag{1}$$

$$X_1 \cup X_2 = X \tag{2}$$

$$C_1 \cup C_2 = C \tag{3}$$

$$C_1 \cap C_2 = \emptyset \tag{4}$$

$$C_2 \neq \emptyset \tag{5}$$

[2] Note that f can be also represented as a constraint and the objective value can be modelled as a variable.

The first property 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 property states that none of the variables and constraints of the original problem P are lost. Furthermore (3) and (4) 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 last property states that P_2 is not allowed to be unconstrained. But note that this restriction does not hold for P_1.

An example (besides the weighted n-queens problem) for a weakly decomposable problem is also the magic knight tour. (See [7] and [8]). In this problem a knight tour on a chessboard is sought for where the numbers of the moves constitutes a magic square. The weakly decomposition is that P_1 consists of the magic square problem and P_2 constitutes that when following the numbers 1 to n^2 this is a knight tour.

In the weighted n-queens problem:
The weighted n-queens problem is weakly decomposable.
P_1 (placement of the queens):

- $X_1 = \{q_1, \dots, q_n\}$
- $C_1 = \{alldifferent(q_1, \dots, q_n); q_i + i \neq q_j + j; q_i - i \neq q_j - j;\}$

P_2 (determining the objective value):

- $X_2 = \{obj, q_1, \dots, q_n\}$
- $C_2 = \{obj = \sum_{i \in \{1..n\}} w_{iq_i}\}$

A symmetry that acts on the subproblem P_1 (but not on P_2) is considered a weak symmetry.

Definition 2 (Weak Symmetry)
Given a weakly decomposable problem P with a decomposition (P_1, P_2).
A symmetry S is called a **weak symmetry** *on P with respect to the decomposition (P_1, P_2) iff S acts on P_1 but not on P_2.*

The intention of the decomposition of the problem is that X_1 contains all symmetric variables (and only these) and X_2 contains all variables.

The gain is that we get a subproblem that is not affected by the weak symmetry (P_2) and a subproblem where the weak symmetry affects all variables and all constraints (P_1). Therefore, the weak symmetry acts like a common symmetry on P_1.

In the weighted n-queens problem:
The weak symmetries acting on the problem are the 8 symmetries of the chessboard. For convenience we just consider one symmetry exemplarily: the flip on the anti-diagonal.

A feasible placement of the queens is also feasible if we flip the board as proposed. Therefore the symmetry acts on P_1.

But since the weights of the fields are different and not symmetric the two solutions lead to different objective values. Therefore, the symmetry does not act on P_2 and the symmetry is weak on P with the respect to the decomposition (P_1,P_2).

2.3 Weak Symmetry Breaking

Since the weak symmetry does not affect the whole problem it cannot be broken on the whole problem. But it can be broken in P_1 (where it acts as a proper symmetry). That means that in the search tree equivalent solutions of P_1 are identified with each other. On the other hand we would lose the symmetric solutions by breaking the symmetry. Therefore we need a way to represent these solutions explicitly because they are needed in order to solve P_2 which delivers a full solution for P.

Note that the solving order just reflects the variable ordering. The problem is not explicitely split. The weakly decomposable problem property just yields that a symmetry is weak and helps us to separate symmetric from asymmetric variables.

In order to represent these symmetric solutions we introduce additional variables called *SymVars*. These variables state the symmetric equivalents of a solution.

Notation 1 *Let P be a weakly decompoasable problem with a decomposition (P_1,P_2), $P_1 = (X_1,C_1), P_2 = (X_2,C_2)$. Let $X_{sym} = \{y_1,\dots,y_\ell\}$ be a set of SymVars that consitute the variables of the subproblem P_{sym}.*

A solution to P_1 is denoted by $s_{P_1} = (X_1)$.

A solution to P_{sym} is denoted by $s_{P_{sym}} = (X_1,X_{sym}) = (s_{P_1},X_{sym})$

A solution to P_2 is denoted by $s_{P_2} = (X_1,X_{sym},X_2) = (s_{P_{sym}},X_2)$.

A solution to P_2 is automatically a solution to P.

Let $s_{P_1} = (v_1,\dots,v_n)$ be a solution to P_1, whereby v_i is a value of the corresponding domain of x_i, $i \in \{1,\dots,n\}$.

A solution $s_{P_{sym}} = (s_{P_1},v'_1,\dots,v'_\ell)$ is a symmetric solution to s_{P_1}, whereby v'_j is a value of the corresponding domain of y_j, $j \in \{1,\dots,\ell\}$

The solving order now is to search a solution s_{P_1} to P_1, determine a symmetric equivalent $s_{P_{sym}}$ in P_{sym} and use this solution to determine a solution to P_2 which already states a solution to P.

Consequences:

– Every feasible value assignment to the SymVars constitutes a symmetric solution to s_{P_1}
– None of the values in X_1 have to be reconsidered to receive a symmetric equivalent
– The symmetry can be broken in P_1 because all symmetric solutions to P_1 are expressed by $s_{P_{sym}}$

In the weighted *n*-queens problem:

For each queen a SymVar is introduced. Each SymVar represents a symmetric value for its corresponding queen. In the case of an assignment $q_i = j$ and the symmetry of the anti-diagonal flip the corresponding SymVar assumes the values $sym_{q_j} = i$ and $sym_{q_i} = j$ (the identity).

P_{sym} (considering a symmetric placement):

- $X_{sym} = \{X_1, sym_{q_1}, \ldots, sym_{q_n}\}$
- $C_{sym} = \Big\{ \big(\forall i \in \{1,\ldots,n\} : q_i = j \Rightarrow sym_{q_i} = j\big) \vee$
  $\big(\forall i \in \{1,\ldots,n\} : q_j = i \Rightarrow sym_{q_i} = j\big) \Big\}$

Note that there is not necessarily one SymVar for each variable in X_1. Often it holds that $|X_{sym}| < |X_1|$.

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 reconsiders variables in X_1 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.

In the following we will only consider optimisation problems where P_1 is the basic problem, and P_2 imposes additional constraints for optimisation. That is $X_2 \backslash X_1$ is just the optimisation variable and C_2 just contains the optimisation constraint (i.e. the optimisation function). The weighted n-queens problem is such a problem.

Then the search tree looks like in Figure 1.

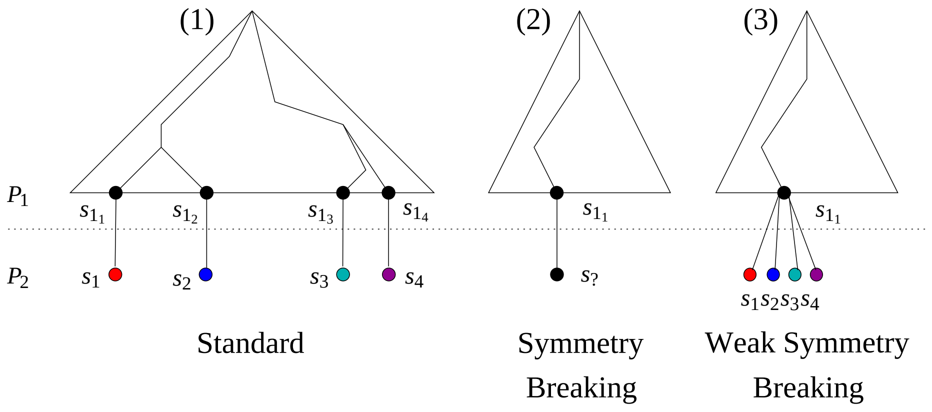


Fig. 1. Optimisation Problem: s_{1_1} to s_{1_4} represent symmetric solutions to P_1 while s_1 to s_4 represent the objective value of these solutions. The numeration yields just the order in which the solutions are found. (1) shows a class of solutions in the search tree without symmetry breaking, whereby the solutions have different objective values. Standard symmetry breaking would lead to a the search tree (2), where solutions with different objective values are identified in s_{1_i} which is the solution not excluded in this class. In (3) the symmetry is broken but the different solutions are preserved by using SymVars for s_{1_1}.

2.4 Applying SymVars

Figure 1 gives insights to the search tree. In (3) the idea of using SymVars can be seen. In practice the path from the choice point s_{1_i} to the solutions $s_1,\ldots,s_4$ can consist of more than one decision. This depends on the representation of the symmetry and the number of SymVars. For geometrical symmetries (like in the *n*-queens problem) it is possible to determine values for all SymVars at the same time. For a permutation the values are determined successively and therefore assigning the SymVars is represented by an own search subtree. For the scenario we investigate throughout this article that means that the leaves of this search tree are the solutions to P_2 (in general these leaves are only the roots for the subproblem P_2).

The SymVars cannot attain any values. Since they represent the symmetry of the problem there are certain constraints which state the symmetry.

The complete weighted *n*-queens problem modelling P:

– $X = \{q_1,\ldots,q_n,sym_{q_1},\ldots,sym_{q_n},obj\}$
– $C = \{$
  - $alldifferent(q_1,\ldots,q_n); q_i + i \neq q_j + j; q_i - i \neq q_j - j;$
  - $\Big\{ \big(\forall i \in \{1,\ldots,n\} : q_i = j \Rightarrow sym_{q_i} = j\big) \vee$
    $\big(\forall i \in \{1,\ldots,n\} : q_j = i \Rightarrow sym_{q_i} = j\big) \Big\}$
  - $obj = \sum_{i \in \{1..n\}} w_{i,sym_{q_i}}$

– Objective: $\max obj$

3 Benefit and Tradeoff

Breaking weak symmetries does not automatically lead to a speed-up in search. Basically it depends on the scenario as well as the instance that is considered.

3.1 General Observations

Although the symmetry can be broken in P_1 which reduces the breadth of the search tree we spend more variables which extends the search tree. Therefore we cannot tell up-front whether we speed up the search.

There are several facts that have to be taken into account:

– gain of symmetry breaking
– additional search for the SymVars
– propagation during the search

Basically these facts determine whether this approach is useful or not. While the first leads to a speed up the latter, two basically means an increase in the search time. For the second fact this can be seen immediately. The drawback with the propagation is, that during the search in P_1 there is no possibility to propagate on the current partial solution. This is due to the fact that P_2 can only compute an objective value when the

SymVars are determined as well. That means that no propagation on the objective value is done in P_1. But when determining a symmetric equivalent in P_{sym} propagation can be used. Therefore, in most cases not all solutions in an equivalence class have to be considered.

Again, it is not possible to tell whether the symmetry breaking outweighs the additional variables and the lack of propagation. This has to be determined empirically.

3.2 Scenario-driven Observations

Much of the efficiency in breaking weak symmetries depends also on the scenario that is considered.

We identified these facts that have to be taken into account:

– the ratio of $\frac{|X_{sym}|}{|X_1|}$
– the boundedness of the problem

Basically breaking weak symmetries is more efficient when there is just a small number of SymVars in contrast to a larger number of variables in P_1. This is due to the fact that with just a view assignments a new symmetric equivalent can be found and the subproblem P_{sym} is relatively small compared to P_1.

The boundedness of the problem is another indicator. Consider a problem where every variable assignment is a solution. In this case the time for considering P_{sym} could be too costly. On the other hand consider a problem that is really tight-fit which means there are just very few feasible solutions in contrast to a huge number of infeasible solutions. Consider further that these solutions are all belonging to one class of solutions (but that is not known by the user). As soon as the first solution is found the SymVars find all solutions.

4 Results

We will use a relaxation of a real world problem from the automated manufacturing of PC Boards. (See [9] and [10]).

The scenario we are regarding is optimisation within a given time interval. This is due to the fact that solving the problem entirely would take too much time (days or even weeks of computing for a single instance).

4.1 Problem Description

The original task (in short) is to optimize the throughput rate of a mounting machine (which consists of several workstations) that mounts PC boards. In the relaxed version we maximise the possibility to place components on the PC boards. Basically a setup has to be determined for the machine (more specifically a setup to each workstation) and this setup states which and how many components could be mounted. In more detail the combination workstation and assigned setup determines the feasibility for mounting a component.

Basically the problem can be modelled as a matrix problem. (See [11] – [14] for matrix problems). The task is to assign given items of specific types to the cells of the matrix such that certain constraints are fulfilled. There is a profit for each tuple (column i,item j) that is achieved when an item of type j is located in the column i. The task is to search a distribution of the items to the matrix such that the resulting sum of profits is maximised. More formally:

Given:

- $m, n \in \mathbb{N}$, (the dimensions of the variable matrix)
- $K = \{1, \dots, k\}$, (the set of different item types)
- $t_1, \dots, t_k \in \mathbb{N}$, $\sum_{i=1}^{k} t_i = m \cdot n$, (the item quantity per type)
- $s_1, \dots, s_c \subset K$, $s_1 \dot{\cup} \dots \dot{\cup} s_c = K$, (the sets of compatible item types)
- $V^{n \times k}, v_{ij} \in \mathbb{N}, i \in \{1, \dots, n\}, j \in \{1, \dots, k\}$, (the profit achieved when an item of type j is placed in column i)

Wanted:

- $A^{m \times n}$, $a_{ij} \in K, i \in \{1, \dots, m\}, j \in \{1, \dots, n\}$, (a distribution of the items)
- $\max obj$, (maximise the achieved profits)

Constraints:

- $\forall j \in \{1, \dots, n\} \exists d : \{a_{1j}, \dots, a_{mj}\} \subseteq s_d$, (all items in a row must be in the same compatibility set)
- $\forall k \in K : \sum_{i \in \{1,\dots,m\}, j \in \{1,\dots,n\}} (a_{ij} = k) = t_k$, (each item is assigned)
- $obj = \sum_{i \in \{1,\dots,m\}, j \in \{1,\dots,n\}} v_{i(a_{ij})}$, (the sum of all profits achieved)

Weakly Decomposable Problem:

This problem is weakly decomposable since the distribution of the items forms P_1 while considering the objective value forms P_2.

Weak Symmetry:

Since it doesn't matter for the objective value in which row an item is located, the row symmetry is a normal symmetry and can be broken without consequences. But identical item types achieve different weights for different columns. Therefore the column symmetry is weak.

Weak Symmetry Breaking:

We introduce for each column of the matrix a SymVar. The domain of the SymVars is the number of columns. An assignment $SymVar[i] = j$ means that the $i-th$ column is permuted to the $j-th$ column. Therefore the numbers assigned to column i by P_1 are considered to be assigned on column j for determining the objective value.

The Decomposition:

The problem is decomposed and solved as follows:
$P_1 = (X_1, C_1)$, with $X_1 = \{A^{m \times n}\}$
$P_{sym} = (X_{sym}, C_{sym}$ is now defined by $X_{sym} = \{pos_1, \ldots, pos_n\}$ and
$C_{sym} = \{alldifferent(pos_1, \ldots, pos_n)\}$
$P_2 = (X_2, C_2)$, with $X_2 = \{X_1, X_{sym}, obj\}$
The solving order (in terms of variable ordering) is $P_1 - P_{sym} - P_2$.

4.2 Models for Comparison

We compare two models. The first one doesn't break the weak symmetry while the second one uses SymVars to break the weak symmetry. The results are very promising.

We use the same search heuristic in both models to get comparable results.

In the standard model whenever a solution to P_1 is found the objective value is immediately computed and the search backtracks for a new solution. Propagation on the objective value for a partial assignment can take place.

In the weak symmetry model whenever a solution to P_1 is found a permutation is determined by P_{sym} and the objective value for the permuted solution is determined. That means for each solution in P_1 the whole equivalence class is considered before a new solution is determined by P_1. The gain is that as soon as P_1 delivers a solution $n!$ solutions can be evaluated. Basically that means that in this approach more solutions are considered in the same amount of time compared to the standard model.

The difference is even more drastic if the problem is really tight fit. That means there are very few feasible solutions.

4.3 Alternative Approaches

Since the problem has not been tackled before it is not possible to compare the approach with techniques other than standard approaches.

An intuitive and simple approach would be to split the problem into two models (P_1, P_2) and solve them successively. More specifically the symmetry is broken on P_1 and all solutions are stored. After P_1 is solved exhaustively for each solution in P_1 its equivalence class is computed. Successively for these solutions the objective value is determined (or in more general applications these solutions are input to P_2).

In this approach the symmetry is broken and there is no time to spend for additional variables. But also there will be no propagation on the objective value in this approach since it is computed after P_1 is solved exhaustively.

The two drawbacks in this approach are that all the solutions have to be stored to generate all symmetric solutions afterward and the whole problem has to be solved exhaustively before the first objective value is returned. The first drawback is only severe in problems with a huge number of solutions (which is very likely for real-world problems) and could in the worst-case lead to a crash of the computer. The second drawback is severe if solving the problem exhaustively takes too long (even with symmetry breaking). For real-world problems this is unfortunately ”state-of-the-art”. Mostly the time we can spend on searching a solution is just a fraction of the time it takes to solve the problem exhaustively.

So this approach cannot be chosen for comparison since we would not receive a solution in the given time interval.

4.4 Results

We generated various instances of the problem size 8×6 and 20×8. These instances of the first problem size are rather small but can be solved in a small amount of time. The larger instances correspond to a real-world job in terms of the variables to assign. There are 80 types of items and the weights range from one to nine. The instances differ from each other by the number of items per type that are to be assigned.

We chose a time-limit of 10 minutes (which is just a fraction of the solving time for the larger scenario). This is basically the amount of time that could be spend in a real-world scenario. We used ILOG OPL Studio 3.7 (see [15]) for the computation on a laptop computer equipped with a Pentium 4 with 3.2 GHz and 512 MB RAM.

For the problem size 8×6 we show the first solution and the corresponding time as well as the best solution and the corresponding time. We also indicate whether an optimum has been found and what time it took to solve the instance exhaustively. All time values are in seconds.

Instance	Strategy	First Sol	Solv Time	Best Sol	Solv Time	Opt found	Time to Prove
1	Standard	226	0.032	261	95.4	yes	148
	Weak	226	0.313	261	0.39	yes	93.9
2	Standard	214	0.047	260	307	yes	428
	Weak	221	0.531	260	0.6	yes	166
3	Standard	209	0.031	259	299	yes	419
	Weak	221	0.375	259	2.14	yes	162
4	Standard	214	0.047	261	305	yes	425
	Weak	226	0.359	261	2.12	yes	163
5	Standard	216	0.031	261	291	yes	407
	Weak	228	0.344	261	0.41	yes	160

For the problem size 20×8 we show the first solution and the corresponding time as well as the best solution and the corresponding time. Since none of the instances can be solved exhaustively we indicate at which time the weak symmetry approach outperforms the standard approach.

Instance	Strategy	First Sol	Solv Time	Best Sol	Solv Time	Weak outperforms
1	Standard	786	0.469	817	306	–
	Weak	758	1.188	841	210	1.250
2	Standard	766	0.859	825	299	–
	Weak	766	1.203	852	566	1.250
3	Standard	766	0.391	825	305	–
	Weak	766	1.171	844	200	1.937
4	Standard	767	0.359	826	301	–
	Weak	767	1.187	846	197	1.953
5	Standard	763	0.406	821	238	–
	Weak	763	1.235	838	211	1.344
6	Standard	764	0.422	817	306	–
	Weak	764	2.750	848	513	2.860

4.5 Conclusions

We showed just a small selection of the instances we generated since they all yield the same results.

In both scenarios the weak symmetry approach finds its first solution later than the the standard approach does. This is due to the fact that more variables and constraints have to be considered. But also in both scenarios weak symmetry very soon outperforms the standard approach. In the smaller scenario the weak symmetry approach finds an optimum within one or two seconds while in the standard approach an optimum is found within about one hundred and three hundred seconds. To prove the optimum took the weak symmetry less time than it took the standard approach to find an optimum in all cases.

In the larger scenario none of the solutions could find an optimum. Again the weak symmetry approach finds its first solution later than the standard approach does. But within one to three seconds the weak symmetry approach outperforms the standard approach. And the best solution within the time-limit also outperforms the result of the standard approach.

Although the weak symmetry approach has to spend more time in each choice point (due to the additional variables and constraints) it outperforms the standard approach clearly in these scenarios.

5 Outlook

We defined weak symmetries and introduced a modelling approach that enables us to break weak symmetries by using SymVars. Also we presented very encouraging results for several instances of a real-world problem.

Still there is much to investigate on weak symmetry breaking. One direction is the efficiency when the problem P_2 is more general. That means that there is a whole sub-problem that has to be solved.

The definition of weakly decomposable problems can iteratively be applied to de-compose P even further to $P = (P_1, \ldots, P_n)$ and on several of them act a different weak symmetry. It would be interesting to investigate such problems. Also the case that a problem yields two or more weak symmetries on the same subproblem seems very interesting. In our scenario this would be the case if the row symmetry would also be weak. In this case two different sets of SymVars would have to be used.

References

1. Peter Gregory *Almost–Symmetry in Planning* SymNet Workshop on Almost-Symmetry in Search, New Lanark, 2005
2. Alastair Donaldson *Partial Symmetry in Model Checking* SymNet Workshop on Almost-Symmetry in Search, New Lanark, 2005
3. Warwick Harvey *Symmetric Relaxation Techniques for Constraint Programming* SymNet Workshop on Almost-Symmetry in Search, New Lanark, 2005
4. Warwick Harvey *The Fully Social Golfer Problem* SymCon'03: Third International Workshop in Constraint Satisfaction Problems, Kinsale, Ireland, 2003
5. Z. Kiziltan, B. Hnich *Prob031: Rack Configuration Problem* http://4c.ucc.ie/ tw/csplib/prob031
6. I. Gent, T. Walsh, B. Selman *CSPLib: a problem library for constraints* http://4c.ucc.ie/ tw/csplib
7. Hosted by Guenter Stertenbrink *Computing Magic Knight Tours* http://magictour.free.fr
8. Compiled by George Jelliss *Knight's Tour Notes* http://www.ktn.freeuk.com
9. Rico Gaudlitz *Optimization Algorithms for Complex Mounting Machines in PC Board Manufacturing* Diploma Thesis, Darmstadt University of Technology, 2004
10. Siamak Tazari *Solving a core scheduling problem in modern assembly-line balancing* Technical Report, Darmstadt University of Technology, Oktober 2003
11. B. Smith *Modelling a Permutation Problem* In: Proceedings of ECAI 2000 Workshop on Modelling and Solving Problems with Constraints, 2000
12. B. Smith, I. Gent *Reducing Symmetry in Matrix Models: SBDS v. Constraints* Held on SymCon'01, 2001
13. P. Flener, A. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, T. Walsh *Matrix Modellling: Exploiting Common Patterns in Constraint Programming* In: Proceedings of the International Workshop on Reformulating Constraint Satisfaction Problems, 2002
14. P. Flener, A. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, J. Pearson, T.Walsh *Breaking Row and Column Symmetries in Matrix Models* In: 8th International Conference on Principles and Practices of Constraint Programming (CP-2002), Springer, 2002
15. V. Hentenryck, I. Lustig *The OPL Optimization Programming Language* The MIT Press
16. Krzysztof Apt *Principles of Constraint Programming* Cambridge University Press,2003
17. R. Backofen and S. Will *Excluding Symmetries in Constraint-Based Search* In: Principles and Practice of Constraint Programming, pp. 73-87,1999
18. I. Gent, B. Smith *Symmetry Breaking During Search in Constraint Programming* School of Computing Research Report 99.02, University of Leeds, 1999
19. I. Gent, B. Smith *Symmetry Breaking in Constraint Programming* In Horn, W., ed.: Proceedings of the 14th European Conference on Artificial Intelligence, pp. 599-603, 2000
20. J.-F. Puget *Symmetry Breaking Using Stabilizers* In Rossi, F., ed.: Proceedings of 9th International Conference on Principles and Practice of Constraint Programming (CP2003), Springer, 2003
21. Roland Martin, Karsten Weihe *Breaking Weak Symmetries* SymCon'04: 4th International Workshop of Symmetry and Constraint Satisfaction Problems, Toronto, Canada, 2004

Symmetric Relaxation Techniques for Constraint Programming

Warwick Harvey

IC-Parc, Imperial College London

Abstract. We present several techniques that can be used to allow the application of symmetry-breaking techniques in constraint programming to problems that are "not quite" symmetric. We do this through the concept of a *symmetric relaxation*.

1 Introduction

Significant attention has been devoted recently to techniques for dealing with constraint satisfaction problems that exhibit a high degree of symmetry; for example: [1, 2, 4–6, 8–10, 13–17]. A range of techniques have been developed, many of which have been quite successful. There are some problems however that are "not quite" symmetric, where some constraints — or perhaps an optimisation objective function — do not respect some or all of the symmetries. There is also a school of thought that "symmetries do not really occur in the real world" — e.g. even if an airline has a fleet of identical aircraft, those aircraft will have different maintenance histories, and thus are not really fully interchangeable in a schedule. For such problems, current approaches to handling symmetry cannot be applied, or can only be applied in a limited fashion.

We present several methods for applying the extensive body work that has been developed for handling fully symmetric problems to problems that are "not quite" symmetric. We do this through the use of a *symmetric relaxation*; that is, a relaxed version of the problem that is symmetric in the way that we want, and on which we can apply standard symmetry-breaking methods.

The aim is to broaden the set of problems to which symmetry-based methods can be applied beyond purely symmetric problems. Since there is no single symmetry-breaking technique that is best in all circumstances (they all have strengths and weaknesses and the technique of choice depends on the problem being solved) we focus on methods that are not tied to any one particular symmetry-breaking technique for the relaxed (symmetric) problem.

In the rest of this section we present some background material and further illustrate the concept of a symmetric relaxation. In Section 2 we present several techniques for exploiting symmetric relaxations, and in Section 3 we address the issue of implementation.

1.1 Symmetric Relaxations

The use of relaxations is a standard problem-solving technique.

Definition 1. *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.*

For example, two common relaxations are omitting integrality constraints and omitting non-linear constraints.

The relaxation R should be such that it is (generally) much easier to solve than P and solving R in some fashion helps in solving P.

Definition 2. *A* symmetric relaxation *SR of a problem P is a relaxation of P such that SR has more symmetry than P.*

Note that a solution S of SR represents a set of *possible* solutions of P, namely the set S^G where G is the symmetry group of SR. Solving P then decomposes into two related problems: finding a solution S of SR and finding an element of S^G that is a solution of P.

Determining what a good candidate SR is for any given P is an interesting problem in its own right, but beyond the scope of this paper. However, minimum criteria for it to be useful are:

– the extra symmetry of SR makes it easier to solve than P; and
– there is an efficient way to find any elements of S^G that are solutions of P.

In this paper we consider two ways to obtain SR from P: relaxing constraints that do not respect all the symmetry we want, and relaxing an optimisation objective function that does not respect all the symmetry we want.

1.2 Example: Diagonal Latin Squares

In this section we consider an example of a problem and corresponding symmetric relaxation.

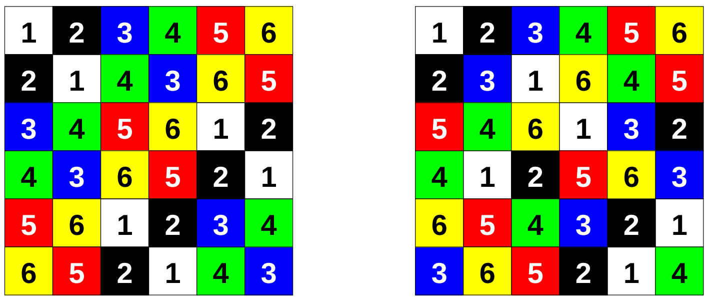
Definition 3. *A* latin square *of order n is an $n \times n$ array where each row or column is a permutation of $1 \ldots n$.*

See Figure 1(a) for an example. Let LS(n) be the problem of finding latin squares of order n. The symmetries of LS(n) are:

– permute the rows;
– permute the columns;
– permute the values; and
– permute the dimensions (rows, columns and values).

That is, applying any of the above operations to a solution of LS(n) yields another solution of LS(n).

Definition 4. *A* diagonal latin square *of order n is a latin square where the main leading and trailing diagonals are permutations of $1 \ldots n$.*



(a) latin square (non-diagonal)

(b) diagonal latin square

Fig. 1. Example latin squares of order 6

See Figure 1(b) for an example. Let DLS(n) be the problem of finding diagonal latin squares of order n. The symmetries of DLS(n) are:

- rotate or reflect the square as a whole;
- swap the first and last rows and the first and last columns (see Figure 2);
- cycle the first and last $\lfloor n/2 \rfloor$ rows and columns (see Figure 3); and
- permute the values.

Clearly, LS(n) is a symmetric relaxation of DLS(n): every solution of DLS(n) is a solution of LS(n), and LS(n) has more symmetry than DLS(n). Interestingly, while DLS(7) has 1832 unique solutions, LS(7) only has 147. That is, LS(7), with weaker constraints, has fewer solutions because it has more symmetry.

1.3 Related Work

Some of the ideas in this paper first appeared in [11].

Roland Martin has been pursuing a similar line of research [12], but from a somewhat different perspective.

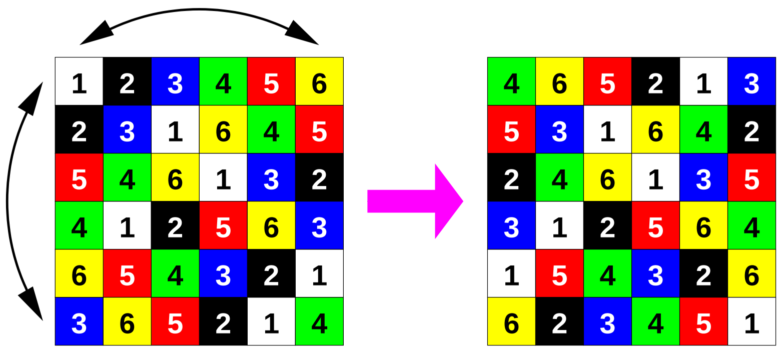


Fig. 2. Swapping first and last rows and columns

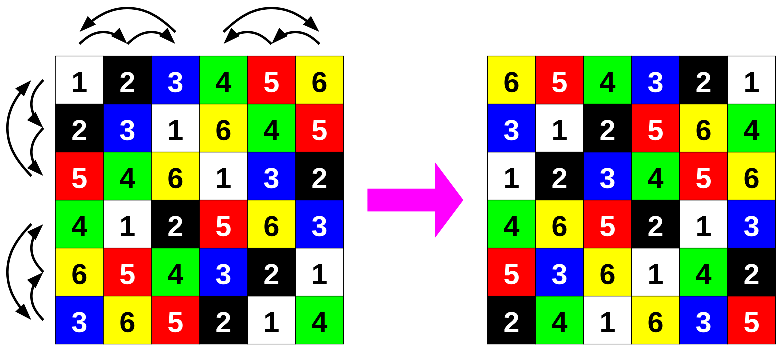


Fig. 3. Cycling first and last $\lfloor n/2 \rfloor$ rows and columns

There is also the obvious dual of symmetric relaxation: symmetric tightening. This is where the constraints are tightened in order to make the problem more symmetric; any solution of the tightening is a solution to the original problem, but some (hopefully not all!) solutions may be lost because they do not satisfy the tightened constraints.

Another technique, often used in work on combinatorial designs, is to constrain the *solutions* of a highly symmetric problem to have a certain set of symmetries. Exploiting such an assumed automorphism group allows solutions to be found to much larger problems than would be possible otherwise, but of course only solutions which possess the assumed symmetry are found. The technique is of course not restricted to combinatorial design problems; it has been used, for example, to find solutions to the maximum density still life problem [3].

2 Techniques for Exploiting Symmetric Relaxations

We now present our techniques for exploiting symmetric relaxations.

2.1 The Two-Phase Method

As noted earlier, when using a symmetric relaxation, solving a problem P decomposes into finding a solution S of SR and finding an element of S^G that is a solution of P. One obvious way of solving P is to solve these two problems sequentially; we call this the two-phase method.

If it is the constraints of the problem that we are relaxing, the algorithm looks like this:

1. Find a solution S of SR
2. Search for $g \in G$ such that S^g is a solution of P
   – Backtrack if no suitable g exists

Figure 4 shows a solution of LS(7), with an example of a set of permutations from LS(7)'s symmetry group that maps it to a solution of DLS(7).

If we are relaxing an objective function, the algorithm looks a little different:

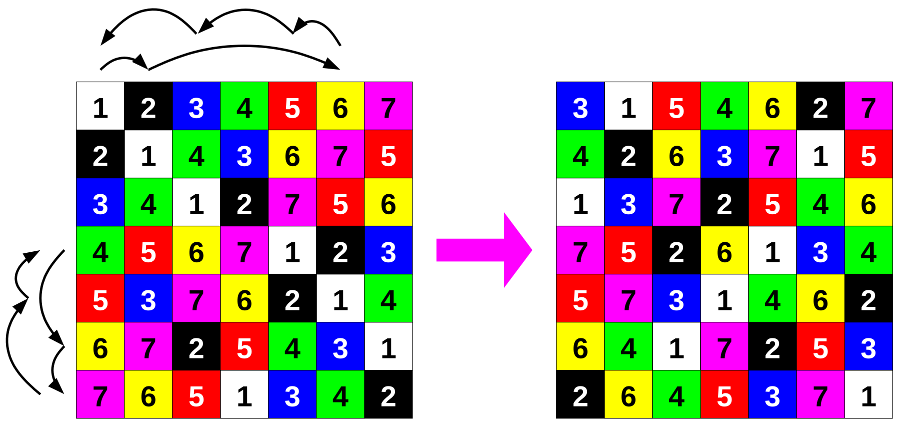


Fig. 4. Mapping a solution of LS(7) to a solution of DLS(7)

1. Find a solution S of SR
2. Search for $g \in G$ such that $f(S^g)$ is optimal

This must be repeated for all solutions of SR, with the final solution being an S^g that is globally optimal.

The two-phase method has obvious advantages: it is simple, has a clear separation of concerns, and does not impose any restrictions at all on the method(s) used to find solutions of SR. However, it also has an obvious disadvantage: the omitted/relaxed constraints (resp. relaxed objective function) cannot prune the search for solutions of SR, even when the current state cannot possibly lead to a solution (resp. optimal solution) of P. The two-phase method is expected to be suitable when:

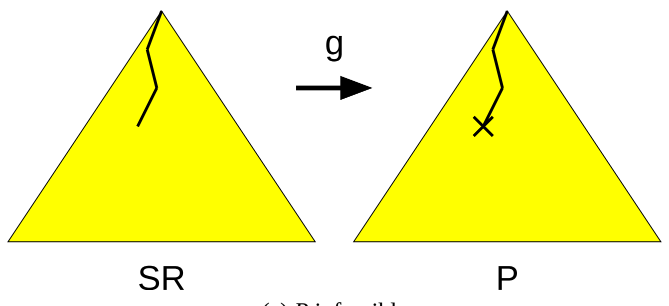
– SR has few solutions;
– it is likely that some S^g is a solution of P; or
– a bound on the objective function will not prune much anyway.

2.2 The Switching Method

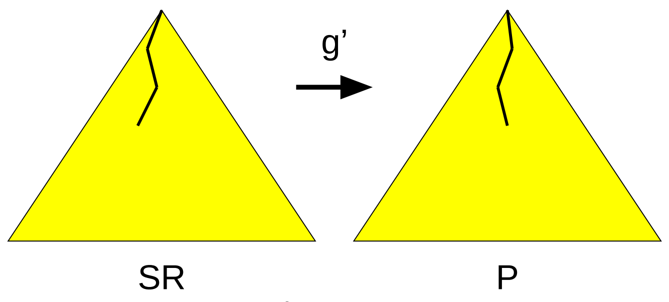
The second method we examine is called the switching method. Consider first the case where we are relaxing constraints.

Conceptually, we explore search trees for SR and P simultaneously, maintaining a mapping $g \in G$ such that P is not infeasible. Whenever infeasibility is detected for P, we try to switch to a different g that repairs the problem (see Figures 5 and 6). If no such g exists, we backtrack in SR.

There are several approaches possible for implementing such a scheme. One choice to be made is with respect to the variables used for P. If SR's extra symmetry with respect to P is just variable symmetry, then the variables of P can be the variables of SR, rearranged according to the current g. In this case, the relaxed constraints are simply checked for infeasibility against the rearranged variables.

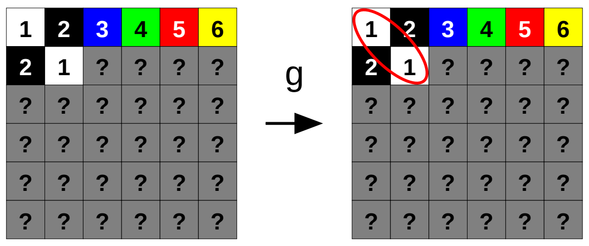


(a) *P* infeasible

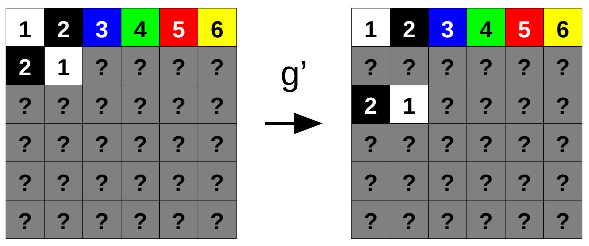


(b) *P*’s feasibility restored

Fig. 5. Change of *g* with the switching method’s concurrent search trees



(a) *P* infeasible



(b) *P*’s feasibility restored

Fig. 6. Illustration of the switching method when solving DLS(6)

Alternatively, one can have a separate copy of the variables for *P*. In this case, one can either use the full set of constraints for *P* (effectively mirroring the entire computation in both trees, with the choices made at each decision point in *SR* mapped to *P* using the current *g*), or one can use for *P* just those constraints that have been relaxed, with one-way propagation from the variables of *SR* to those of *P*. If *SR* has value symmetry not present in *P*, then the former option may be easier, since it is just the decisions that need to be mapped through a value symmetry, rather than the domains of all variables affected by propagation during the computation.

Obviously, these alternatives all have different implementation costs, and different pruning power; which one is best may depend on the problem.

For a relaxed objective function, the approach is similar, but extra work is needed when a solution is found (f is the objective function):

- Whenever the current state of *P* is infeasible or cannot lead to an improved optimal value, search for a new g
  - Backtrack if no suitable g exists
- Whenever we find a solution S of *SR*, search for $g \in G$ such that $f(S^g)$ is optimal
- Return best solution S^g found

The switching method has the advantage that the relaxed constraints or relaxed objective function can prune the search in *SR*. However, any inferences made in *P* cannot propagate back to *SR* (i.e. the state of *P* cannot cause domain reductions in *SR*, only the failure of the current branch). Also, using the objective function to guide the search (a common technique) may not be useful since the objective function we have is just one representative objective function, and a promising direction for this objective function may not coincide with what is good for the other possible objective functions. Finally, there needs to be some way to handle the non-monotonicity that occurs in *P* whenever we switch to a new g.

2.3 The Variable Mapping Method

Most of the drawbacks of the switching method can be avoided if we do not use a sequence of fixed gs, but instead use a single *variable* g throughout. As the search progresses, the domain of g can be reduced to exclude elements that cannot lead to feasible or optimal solutions of *P*. Since the variable g characterises all possible feasible mappings, it will also be possible to use it to propagate inferences from *P* back to *SR*. (We assume the use of separate variables for *SR* and *P*.)

This approach obviously has its own drawbacks. The main problem is implementing the variable g, and doing so efficiently and effectively. This is explored further in Section 3.

3 Implementing the Required Group Operations

In the previous section we have repeatedly said things along the lines of “Search for $g \in G$ such that ...” and even “use a variable g” — without giving any indication of how this might be done. These are not a trivial problems, and their efficiency is vital to the success of the methods described in this paper.

For the "Search for $g \in G$ such that ..." problems, it is quite possible to develop a bespoke solution based on the symmetry group using existing tools. Modelling a variable g using existing constraint solvers is harder, but one could use, for example, a vector of finite domain variables indicating how variables, values, or variable-value pairs are mapped from *SR* to *P*. This works as long as one can come up with suitable constraints to ensure that the only feasible assignments are those that correspond to elements of the appropriate symmetry group, and as long as one can infer and propagate useful information between *SR* and *P*.

Alternatively, one could use a constraint solver that directly supports variables with domains being elements of a group, which would mean that one implementation of the techniques presented in this paper could work for all symmetry groups. Unfortunately, to the best of our knowledge, a good and efficient full-featured Group Constraint Solver is still some way off. We have implemented a very basic prototype (using GAP [7] and ECL^{i}PS^e [18]), but it does not have all the features required to implement all the techniques presented here, and we expect it to need both more features and smarter algorithms in order to implement even the simplest of the problems here efficiently and effectively. A full discussion of the design and implementation of such a solver is, however, beyond the scope of this paper.

Note that whichever technique is used, there is likely to be a good deal of symmetry involved in the mapping problem. For example, for the two-phase method where we have a solution *S* of *SR* and we are trying to map this onto *P* such that all the constraints are satisfied, there are two potential sources of symmetry. First, *P* may have some symmetry, even if it is much less than that of *SR*. In this case there is no need to try any mappings that will result in something symmetrically equivalent (in *P*) to the result of a previously tried mapping. That is, we only need to try one mapping from each left coset of the symmetry group of *P*. Second, if *SR* is highly symmetric then there is a good chance that *S* has automorphisms. In this case, each element of a right coset of the automorphism group will map *S* to exactly the same candidate solution in *P*; thus we only need to try one element from each such coset.

4 Conclusions and Further Work

We presented the concept of a symmetry relaxation, and described several ways in which it could be used to solve problems that are "not quite" symmetric. None of the techniques presented are tied to any particular symmetry-breaking technique for solving the symmetric relaxation, which means that any of the many approaches to handling symmetry can be used for this part of the problem.

While a number of the presented techniques could be implemented now on a problem-by-problem basis, it seems that much work needs to be done before we can have an efficient, effective and general tool for solving problems in this way. The path to such a tool seems to be through the development of a Group Constraint Solver, an interesting and challenging task in its own right.

Beyond issues of implementation, there are several open questions:

– Which problems could benefit through the use of these techniques?
– How does one identify a good symmetric relaxation to use?
– How does one choose which technique to use for a given problem and relaxation?

5 Acknowledgments

The author would like to thank Steve Linton for his continuing guidance on computational group theory and the GAP system, and for discussions on how to implement a Group Constraint Solver. The author would also like to thank Joachim Schimpf for coining the expression "symmetric relaxation" to describe what we were trying to do. Finally, the author would like to thank the other participants of the SymNet workshop at New Lanark for many interesting discussions.

References

1. Rolf Backofen and Sebastian Will. Excluding symmetries in constraint-based search. In Joxan Jaffar, editor, *CP'99: Proceedings of the 5th International Conference on Principles and Practice of Constraint Programming*, LNCS 1713, pages 73–87. Springer, 1999.
2. Nicolas Barnier and Pascal Brisset. Solving the Kirkman's Schoolgirl Problem in a few seconds. In Pascal Van Hentenryck, editor, *CP 2002: Proceedings of the Eighth International Conference on Principles and Practice of Constraint Programming*, LNCS 2470, pages 477–491. Springer-Verlag, 2002.
3. Robert Bosch and Michael Trick. Constraint programming and hybrid formulations for three life designs. In Narendra Jussien and François Laburthe, editors, *Proceedings of the Fourth International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimisation Problems (CP-AI-OR'02)*, pages 77–91, 2002.
4. Torsten Fahle, Stefan Schamberger, and Meinolf Sellmann. Symmetry breaking. In Toby Walsh, editor, *CP 2001: Proceedings of the 7th International Conference on Principles and Practice of Constraint Programming*, LNCS 2239, pages 93–107, 2001.
5. Pierre Flener, Alan M. Frisch, Brahim Hnich, Zeynep Kızıltan, Ian Miguel, Justin Pearson, and Toby Walsh. Breaking row and column symmetries in matrix models. In Pascal Van Hentenryck, editor, *CP 2002: Proceedings of the Eighth International Conference on Principles and Practice of Constraint Programming*, LNCS 2470, pages 462–476. Springer-Verlag, 2002.
6. Filippo Focacci and Michaela Milano. Global cut framework for removing symmetries. In Toby Walsh, editor, *CP 2001: Proceedings of the 7th International Conference on Principles and Practice of Constraint Programming*, LNCS 2239, pages 77–92, 2001.
7. The GAP Group. *GAP – Groups, Algorithms, and Programming, Version 4.3*, 2002. (`http://www.gap-system.org`).
8. Ian P. Gent, Warwick Harvey, and Tom Kelsey. Groups and constraints: Symmetry breaking during search. In Pascal Van Hentenryck, editor, *CP 2002: Proceedings of the Eighth International Conference on Principles and Practice of Constraint Programming*, LNCS 2470, pages 415–430. Springer-Verlag, 2002.
9. Ian P. Gent, Warwick Harvey, Tom Kelsey, and Steve Linton. Generic SBDD using computational group theory. In Francesca Rossi, editor, *CP 2003: Proceedings of the 9th International Conference on Principles and Practice of Constraint Programming*, LNCS 2833, pages 333–347. Springer-Verlag, 2003.

10. Ian P. Gent and Barbara M. Smith. Symmetry breaking in constraint programming. In W. Horn, editor, *Proceedings of ECAI-2000*, pages 599–603. IOS Press, 2000.
11. Warwick Harvey. The fully social golfer problem. In Barbara M. Smith, Ian P. Gent, and Warwick Harvey, editors, *Proceedings of SymCon'03: The Third International Workshop on Symmetry in Constraint Satisfaction Problems*, pages 75–85, 2003.
12. Roland Martin. Approaches to symmetry breaking for weak symmetries. In Alastair Donaldson and Peter Gregory, editors, *Proceedings of the SymNet Workshop on Almost-Symmetry in Search, New Lanark*, 2005.
13. Iain McDonald and Barbara M. Smith. Partial symmetry breaking. In Pascal Van Hentenryck, editor, *CP 2002: Proceedings of the Eighth International Conference on Principles and Practice of Constraint Programming*, LNCS 2470, pages 431–445. Springer-Verlag, 2002.
14. Pedro Meseguer and Carme Torras. Exploiting symmetries within constraint satisfaction search. *Artificial Intelligence*, 129:133–163, 2001.
15. Ian Miguel, Chris Jefferson, and Alan Frisch. Constraints for breaking more row and column symmetries. In Francesca Rossi, editor, *CP 2003: Proceedings of the 9th International Conference on Principles and Practice of Constraint Programming*, LNCS 2833, pages 318–332. Springer-Verlag, 2003.
16. Jean-François Puget. Symmetry breaking revisited. In Pascal Van Hentenryck, editor, *CP 2002: Proceedings of the Eighth International Conference on Principles and Practice of Constraint Programming*, LNCS 2470, pages 446–461. Springer-Verlag, 2002.
17. Jean-François Puget. Symmetry breaking for matrix models using stabilizers. In Francesca Rossi, editor, *CP 2003: Proceedings of the 9th International Conference on Principles and Practice of Constraint Programming*, LNCS 2833, pages 585–599. Springer-Verlag, 2003.
18. Mark G. Wallace, Stefano Novello, and Joachim Schimpf. ECLiPSe : A platform for constraint logic programming. *ICL Systems Journal*, 12(1):159–200, May 1997.

Conclusions of the SymNet Workshop on Almost-Symmetry

Peter Gregory[1] and Alastair Donaldson[2]

[1] University of Strathclyde
Glasgow, UK
[2] University of Glasgow
Glasgow, UK

1 Defining Almost-Symmetry

One of the specific aims of this workshop, and of the EPSRC Network that supports it, is to find commonalities in how symmetry is researched and exploited throughout the spectrum of different search communities. The first step to achieving this is to use terminology in a consistent way. Having looked at how different people view almost-symmetry we provide a definition that is hopefully general enough to encompass all view-points, but specific enough to remain meaningful.

Definition 1. *Let P be a problem with symmetry group $Aut(P)$. Let P' be an abstraction of P such that $Aut(P) \subseteq Aut(P')$. Then P is almost-symmetric with respect to $Aut(P')$.*

Everybody's intuition about almost-symmetry reasoned about an abstracted problem that had more symmetry than the original problem. Two types of abstraction were discussed: the first being a new problem that was 'less constrained' than the original, the second being a new problem that is 'more constrained' than the original. There is a trade-off when defining the abstraction. Whilst the aim is to increase the symmetry in the model, removing too much information from the model renders the abstraction meaningless.

Definition 1 covers both types of abstraction where the 'constraints' on the problem are either loosened or tightened, because in both of these cases the aim is to increase the symmetry in the problem.

Note that when *constraints* are mentioned, we are talking generally about any problem, not just CP. Constraints in this context also refers to an optimisation function, since relaxing/tightening the optimisation function was noted as a useful abstraction.

2 Conclusion

Is almost-symmetry pervasive throughout search domains? Can we redefine our models to remove almost-symmetry? Examples throughout these proceedings demonstrate that almost-symmetry is pervasive. We certainly can't redefine our models in every circumstance to 'reveal' the underlying symmetry, the information that yields the asymmetry is often important to the solution.

Research into almost-symmetry is still in its infancy, these proceedings are suggestive of the benefits we will gain from further research. We have seen applications of almost-symmetry, interesting approaches for almost-symmetry detection and several suggestions for the exploitation of almost-symmetry.

The study of almost-symmetry is suggestive of a positive trend in symmetry research. All of the approaches introduce symmetry by way of an abstraction. The premise being, that current symmetry-breaking techniques can solve this highly-symmetric abstraction easily and quickly yield a true solution. The fact that we can realistically make this claim is very encouraging indeed.