Extending the Exploitation of Symmetries in Planning

Maria Fox and **Derek Long**
Dept. Computer Science
Science Laboratories
University of Durham, UK
maria.fox@durham.ac.uk, d.p.long@durham.ac.uk

Abstract

Highly symmetric problems result in redundant search effort which can render apparently simple problems intractable. Whilst the potential benefits of symmetry-breaking have been explored in the broader search community there has been relatively little interest in the exploitation of this potential in planning. An initial exploration of the benefits of symmetry-breaking in a Graphplan framework, by Fox and Long in 1999 (Fox & Long 1999) yielded promising results but failed to take into account the importance of identifying and exploiting new symmetries that arise during the search process. In this paper we extend the symmetry exploitation ideas described in (Fox & Long 1999) to handle new symmetries and report results obtained from a range of planning problems.

Introduction

Problems in which there are many objects capable of the same behaviours often exhibit a highly symmetrical structure. This symmetry might be geometric, as in puzzles like n-queens, indicative of functional identity, as in problems in which there are many uninterestingly different objects, or arising as a consequence of redundancy in a physical system. In each case, symmetries result in wasted search effort because of the proliferation of effectively identical search branches. Many authors, for example (Ip & Dill 1996) in the model-checking community and (Gent & Smith 2000), (Roy & Pachet 1998) and (Crawford *et al.* 1996) in the CSP community, have considered ways of breaking symmetry and experimented with the effect of this on search efficiency. Symmetry breaking has not been much studied in the planning community, although techniques for recognising and exploiting some of the symmetry present in planning problems have been described (Fox & Long 1999).

In (Fox & Long 1999) it was shown that symmetries detected in a planning problem could be broken during the backward search process of a Graphplan-style planner and that this could yield exponential improvement in performance. The strategy involved removing objects from their initial symmetry groups as they were selected to play specific roles in a plan, and returning them to their symmetry groups on backtracking. When a choice leads to failure no other objects in the symmetry group need to be tried, because their symmetry with the failed choice means that they too will lead to failed search effort. The experiments presented in that paper demonstrate that exponential improvements could be obtained only when objects, having been removed by selection from their initial symmetry groups, never entered new symmetry groups, not present in the initial state of the problem, during the course of the search. In fact, in general, new symmetry groups do appear during search as objects move into new relationships with one another, and the extent to which this happens has an inverse effect on the benefits to be obtained from the symmetry-breaking strategy described in that paper. This can be seen from the Gripper experiments presented in that paper, where the exploitation of symmetry yields an order of magnitude improvement in performance but the trend is still exponential. This is because as balls move from $room1$ to $room2$ they enter a new symmetry group with respect to $room2$, but the new symmetry group is not detected so the resulting symmetry cannot be exploited by the planner. By contrast, in the simple TSP experiment exponential improvements were obtained because no new symmetry groups arise during search.

In fact, the strategy described in (Fox & Long 1999) can be modified so that new symmetry groups can be identified and exploited during the search process. This paper describes the necessary extensions and the effects on the performance of the planner. The work described in this paper was implemented by extending Stan version 3, and we refer to the resulting system as SymmetricStan. In SymmetricStan the initial symmetries are found automatically by examination of the initial state and the operator descriptions, as in Stan

version 3. The hypothesis is that objects that share identical initial states, and can make only identical transitions, are uninterestingly different (functionally identical). This analysis gives SymmetricStan access to some, but not all, of the symmetry available for exploitation in a planning domain. It might be possible to identify more of the available symmetry by automatic analysis, but we have not yet considered this potential in detail. The problem of automatically identifying all symmetries is computationally hard and is not addressed here. Furthermore, the completeness of the planner relies on the identified symmetries being solution preserving. Again, it is too hard to guarantee this automatically so the guarantee must be provided offline (it is straightforward to prove that our notion of functional identity is solution preserving).

This paper is structured in the following way. We first describe the technical problems of expressing and exploiting symmetries in a general search context. We consider symmetry exploitation in the contexts of planning, solving CSPs and constructing reachability analyses. We then describe the technical aspects of implementing a symmetry exploitation mechanism that is able to identify and exploit newly arising symmetries during the search process, and explain how this differs from the approach described in (Fox & Long 1999). We then present results demonstrating that our approach gives us exponential improvements in performance in highly symmetric domains, and we demonstrate the improvements obtained over the results reported in (Fox & Long 1999). Finally, we consider scope for future developments in symmetry exploitation in planning.

Exploiting Symmetries in Search

The simplest way of understanding the concept of symmetry is in a CSP context in which the problem is expressed as a set of variables each associated with a domain of values. Search can be seen to take place in the space of partial assignments, and the choices associated with a node correspond to the values available for assignment to a particular unassigned variable. In this context we can say that if two values in the domain of V, v_i and v_j, are symmetric then failure of the assignment of v_i to V implies failure of the assignment of v_j to V. This observation justifies the pruning of the branch corresponding to the choice of v_j as soon as the choice of v_i fails.

When the assignment of some v_i is made, to variable V, v_i can no longer be seen as symmetric to the values it was symmetric to prior to its selection. This symmetry can only be regained when the choice is undone on backtracking. Furthermore, subsequent choices are made in the context of the assignment of v_i to V, so the only symmetries that can be exploited when these choices are backtracked over are the ones that are active in the context of the selection of v_i. The n-queens example presented in Figure 1 shows how symmetries change between contexts. Before any queens are placed the board has horizontal, vertical and diagonal symmetries (also some rotational symmetries, not depicted). When the first queen is placed, in the origin, only the symmetry corresponding to the leading diagonal is active in the resulting board position. The second queen is placed on the board in this context. Now – suppose the resulting board position cannot be developed to a solution. When the second choice is backtracked over only the search branch corresponding to the position reflected over the leading diagonal can be pruned. When the first choice is backtracked over all of the original symmetries can be exploited to prune search. As can be seen in the figure, this results in the removal of four choices for this first queen – the bottom right hand corner is excluded by composition of the horizontal and vertical symmetries.

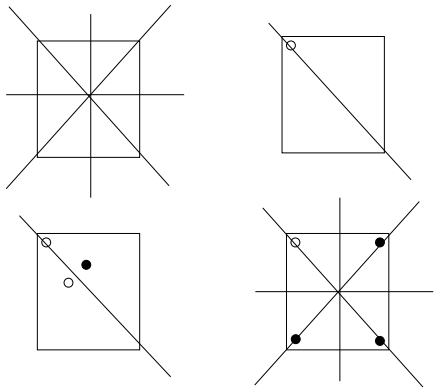

Figure 1: The process by which geometric symmetries can be exploited in backtracking over queen positions in an n-queens puzzle (black dots are choices excluded by symmetries on backtracking).

Gent and Smith (Gent & Smith 2000) discuss the n-queens problem as an example of one where symmetry breaking can yield orders of magnitude gains in performance in the production of all solutions given a specific n. They explain how symmetries, hand-coded by the domain modeller, can be exploited to construct just one solution from the equivalence class of solutions defined by the symmetries, and the other solutions extracted by application of the symmetries to that one solution. One might expect an exponential improvement in performance to be demonstrated, but in fact this is not obtained because of failure of the strategy to

maximise the potential for symmetry-breaking in the search process. In the n-queens situation new symmetries not present in the initial board do not arise because the empty board already contains all available symmetries in the problem. However, the order in which variables are considered for assignment does affect the effectiveness with which these symmetries can be exploited. For example, when the first queen is placed on the board all symmetres will be lost, unless the queen is placed on an axis (as in the figure), and never regained unless that choice is backtracked over. However, the placement of the second queen can allow some symmetry to be recovered. For example, if the first queen is placed at position (r2,c1) and the second at position (r3,c4) then rotational symmetry returns to the board (rotation through 180 degrees). However, if the second queen is placed in a different column then no symmetry will arise. Gent and Smith observe that variable ordering can be significant in determining how much symmetry can be exploited in the search for a CSP solution.

Planning domains typically reveal less geometric structure than puzzles of this nature, but the processes by which symmetries are broken by the selection of choices and then reinstated on backtracking are identical. Fox and Long present the results obtained from the Gripper domain, depicted in Figure 2, in which the balls are functionally identical (they all begin in $room1$, end in $room2$ and are capable of being picked up and dropped) and the two grippers are also functionally identical (they both start and end empty and are capable of holding and of being empty). Stan version 3 is capable of using these symmetries to prune alternative ball and gripper selections during the search process (on the grounds that if a plan to move all the balls to $room2$ cannot be found, by time-point k, by picking up $ball1$, then no plan will be found, by time point k, by picking up $ball2$). In this domain symmetries that are not present in the initial state *do* arise during the search process. Figure 3 depicts a symmetry that arises during the search process that cannot be exploited by Stan version 3.

The newly arising symmetries can be automatically detected by, at each state visited in the search process, examining the configurations of objects that have lost their initial symmetric relationships. Any objects in identical configurations in a given state will form a new symmetric group within that state. In the next section the precise means by which this analysis is performed in SymmetricStan will be described. Before progressing to that description it is necessary to provide a more formal definition of what is meant by a symmetry, and of how symmetries can be identified from functional

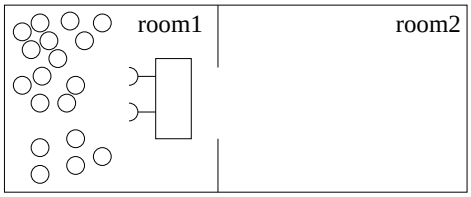

Figure 2: The initial state of a trivial instance of the Gripper domain, in which all the balls and both grippers are respectively symmetric. The robot can move between the rooms and the balls can be picked and dropped.

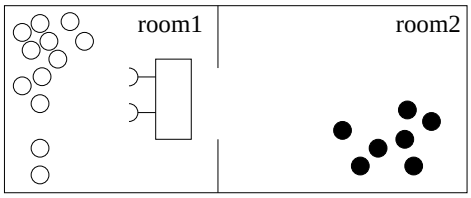

Figure 3: A Gripper state during search, in which some of the balls have been moved to $room2$. The black balls form a symmetric collection not present in the initial state.

identities in the initial state of a problem.

It is convenient to define a symmetry as a composition of permutations on values and on variables (one of which may be the identity permutation). A simple encoding of the n-queens problem as a CSP can be constructed in which the variables correspond to column numbers and the values correspond to row numbers. The variables all share the same domain and the assignment of a value to a variable corresponds to specifying the row for the position of the queen in that column. Under this encoding horizontal symmetry can be expressed as the composition of:

$$r_i \leftrightarrow r_{n-i+1}$$

with the identity permutation on variables. Vertical symmetry can be expressed as the composition of:

$$c_i \leftrightarrow c_{n-i+1}$$

with the identity permutation on values. Gent and Smith use a more flexible interpretation of a symmetry (any one-to-one mapping that is solution-preserving) but this is highly permissive and prevents the exploitation potential of taking a group-theoretic view of symmetries. Indeed, using the definition of symmetries as permutations closed under composition it is possible

to generate the full collection of n-queens symmetries from just one rotation and one reflection, whereas using Gent and Smith's definition it is necessary to specify all of the symmetries by hand. The permutations-based definition also emphasises the relationship between symmetries in search and symmetry groups in the group-theoretic sense.

It is not a trivial step to see how the symmetries obtained from functional identity in a planning problem can be expressed as permutations in this way. In fact, the permutations exist only implicitly in Stan version 3 and in SymmetricStan. They can be explicitly obtained by first identifying the collections of functionally identical objects in the initial state. These collections can then be used to generate all of the symmetric propositions and symmetric actions defined by the domain description. Taking again a CSP view, the propositions (subscripted by time identifiers to indicate the points at which they would be considered for assignment during the search process) can be taken to be the variables in the problem and the actions (again subscripted by time to indicate when they could be assigned) can be taken to be the values. Then permutations on the values can be formed by pairwise swaps, throughout the current assignment, of time-subscripted actions based on the symmetries between the objects they manipulate. For example, if $ball1$ and $ball2$ are functionally identical, and $left$ and $right$ are functionally identical, then

$$pickup_k(ball1, left, room1)$$

can be swapped with

$$pickup_k(ball2, left, room1)$$

and

$$pickup_k(ball2, left, room1)$$

can be swapped with

$$pickup_k(ball2, right, room1)$$

(where k is the time step at which the action is applied). By composing these permutations it can be seen that

$$pickup_k(ball1, left, room1)$$

can be swapped with

$$pickup_k(ball2, right, room1)$$

(the rooms are not functionally identical because they start and end in different states). These pairwise swaps result in the symmetric assignments that need not be considered during search if a particular value selection leads to failure.

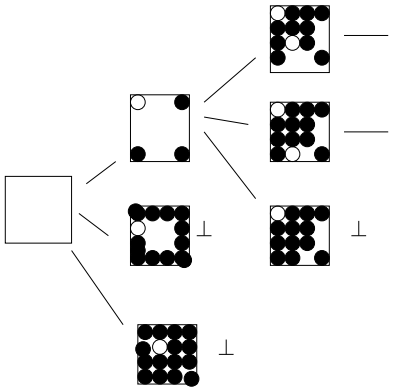

Figure 4: The first two moves in the 4-queens problem, eliminating symmetric alternatives (and alternatives ruled out by the rules of the game) in a forward-chaining search. As soon as a column cannot be filled the branch fails.

It is important to observe that the exploitation of symmetry as described here is not dependent on the use of backtracking search. The processes described here can be implemented successfully in a forward-chaining reachability analysis simply by using a record of which symmetric alternatives have been tried to eliminate choices at each level in the forward search. Figure 4 shows how alternatives can be quickly eliminated in the 4-queens problem.

Recovering New Symmetries in SymmetricStan

The potential for new symmetry states to arise during the search process in the solution of many planning problems means that the exploitation strategy described in (Fox & Long 1999) has to be extended to gain the full benefits of symmetry-breaking. In Stan version 3 the standard GraphPlan-style (Blum & Furst 1995) backward search through the plan graph was modified by the addition of two data structures, one level-independent, called *brokenSym*, which recorded whether objects were in or out of their initial symmetry groups and one level-dependent, called *triedGroups*, which recorded which collections of symmetric alternatives had effectively been tried once a choice had been tried and had failed. BrokenSym could be level-independent because every object that started in a symmetry group in the initial state could, from then on, only be either *in* that group or *out* of it. Because of this the brokenSym record could be implemented as a boolean vector with one entry per domain ob-

ject. The triedGroups structure was level-dependent because failure of a particular action choice at layer k does not obviate the need to consider a symmetric equivalent at layer $k+1$ (hence the need to subscript actions with level numbers in the representation of the symmetries).

The behaviour displayed by Stan version 3 is that, as choices are made in the search process, a decreasing amount of symmetry remains to be exploited. Since no symmetries not present in the initial state are discovered during the search process this tends towards an exponentially worsening trend in performance. SymmetricStan extends the behaviour of Stan version 3 by making it possible to identify the important new symmetries as they arise during search. This is achieved as described in the following paragraphs. The description given here is presented in terms of the modifications that need to be made to a Graphplan-style backward search of a plan graph. However, the underlying strategy for recording and exploiting the symmetry status of objects throughout the search process of a planner is identical in other planning architectures and can be implemented in a way similar to that described here.

In SymmetricStan, the data structure recording the membership of objects in symmetry groups is no longer level-independent. Objects are no longer either in or out of their initial groups, they might move between several different groups throughout the search process. This membership pattern is level-dependent because objects make transitions between groups as a consequence of making transitions between states. We require brokenSym to record, for each object, whether it is in or out of the symmetry group that it belonged to before entering the current layer. Thus, if *ball*1 was in the symmetry group of balls in *room*1 at level k, and the action of picking up *ball*1 is performed at level k, then *ball*1 can be recorded as having left its symmetry group at level $k-1$. A separate data structure, symRecords, maintains the symmetry group associated with each domain object. This can be consulted at any time in order to discover what symmetry group a given object belongs to at a given layer in the graph.

SymRecords and brokenSym are in a close relationship with one another. At each layer, l_i, in the graph brokenSym is consulted to ascertain which objects have left a symmetry group in the transition to l_i. For each such object, o, the collection of active propositions at l_i is scanned to identify the propositions that refer to o. These propositions, with references to o removed, are referred to as *hulls* for o. For example, the proposition *(at ball1 room2)* produces the hull *(at * room2)* for the object *ball*1. The collection of hulls associated with o can be used as an index into an associative array containing the existing symmetry groups and o can then be correctly associated with its current symmetry group. This array can be maintained in a level-independent way. The hull-construction phase, and the identification of symmetry group membership for each object, is done when all actions at l_i have been selected.

A subtlety arises when concurrent activities refer to the same object. The first concurrent action selected will cause the object to leave its symmetry group. The second concurrent action selected will not affect whether the object is in or out of a symmetry group because the object will have already had its symmetry broken by the first action. If the second action is undone, by backtracking, the symmetries of the object will not be restored – indeed the object symmetries will only be restored if the first of the concurrent actions is backtracked over. In this way, a collection of concurrent actions is handled as a single package of actions on the object. The symmetry of the object is affected by selection and de-selection of the whole package, although the order in which the actions are selected determines which action has the symmetry-breaking effect of the package on that object.

The final data structure to consider is the triedGroups structure. In Stan version 3 triedGroups was implemented as an array of the outer shells of matrices, one for each action symmetry group, each one having dimensions corresponding to the number of potentially symmetric arguments of the actions in that symmetry group. So, in the Gripper example, since balls and grippers both form object symmetry groups in the initial state, but rooms do not, the action symmetry group to which *pickup(ball1,left,room1)* belongs yields a two-dimensional matrix with balls along one axis and grippers along the other. There are as many columns and rows as there are balls and grippers respectively, with each column label corresponding to a single ball and each row label corresponding to a single gripper. An extra column and row, labelled $(*,*)$ is supplied to represent the symmetry groups for balls and for grippers, respectively. In the matrix entry for the pickup action group at layer k, a mark in the cell corresponding to $(left,*)$ means that all balls have effectively been picked up in the left gripper (that is: the attempt to pickup *ball*1 in the left gripper failed to reach a solution by layer k, so no other pickup actions using the left gripper need be considered at this layer). A mark in $(*,ball1)$ means that all grippers have effectively been tried on *ball*1, at layer k, and a mark in $(*,*)$ means that no other pickup actions need be tried at layer k. In Stan version 3 the inner cells in the matrices were of no interest, since they corresponded

to specific instantiations of the associated action. For example, a mark in $(left, ball1)$ would simply assert that the attempt to pickup $ball1$ in the left gripper had been tried. This yields no useful pruning information so these cells were not stored.

In SymmetricStan the triedGroups structures have a more sophisticated interpretation. Each layer in the graph still maintains an array of matrices corresponding to the action symmetry groups, and each matrix has dimensions determined by the number of potentially symmetric arguments. However, in SymmetricStan the columns and rows of the matrices are labelled by representatives of symmetry groups which need not be singletons as in the matrices in Stan version 3. Because the column and row labels now refer to symmetry groups the inner cells of the matrices become important, because a mark in $(left, ball1)$, for example, means that all balls symmetric to $ball1$ have effectively been tried in all grippers symmetric to the left gripper. Figure 5 depicts this situation. It can be observed that the triedGroups structure is very similar in organisation to that in Stan version 3, but that the way it is interpreted is different and its correct use relies upon the correct maintenance of the associative array associating objects with symmetry groups throughout the search process.

Figure 5 should be read in the following way: the object $b1$ loses its symmetry as the search enters layer k of the graph. Its new symmetry status is identified by the construction of its hulls and this process identifies $b1$ as belonging to a symmetry group also containing $b3$ and $b5$. The ball $b1$ is taken to be the representative for this group. There is no significance in its selection – any member ball can play this role. It can now be seen that a mark in the cell corresponding to $(left, b1)$, in the triedGroups matrix for the pickup actions, is made when the action *pickup(b1,left,room1)* is tried and fails. Since $b1$ is the representative for the group also containing $b3$ and $b5$, the same cell would be marked if *pickup(b3,left,room1)* had been tried instead, or even if the pickup had been made using the right gripper (since left is the representative for the group containing both the left and right grippers). The matrices in triedGroups contain entries for all potentially symmetric objects, even though many of these entries will not be used. This is because the matrices are constructed before it is known which objects will be representative of the symmetry groups available at the corresponding layers.

Results

We have conducted a range of experiments to investigate the effects on planner performance of the extensions described in the previous sections. We compared Stan version 3 with SymmetricStan in order to discover the impact of exploiting symmetry groups that are not present in the initial state of a problem but arise during the search process. In order to dispel the concern that symmetries might cause less significant problems for forward searching planners we experimented with FF (Hoffmann & Nebel 2001) to see to what extent its behaviour is sensitive to problem symmetries. All experiments were performed under Linux on a PC with 1Gb memory and a 1.4MHz Intel processor. We used FF version 2.2. All times for SymmetricStan and Stan version 3 include the preprocessing time spent on identifying functional identities.

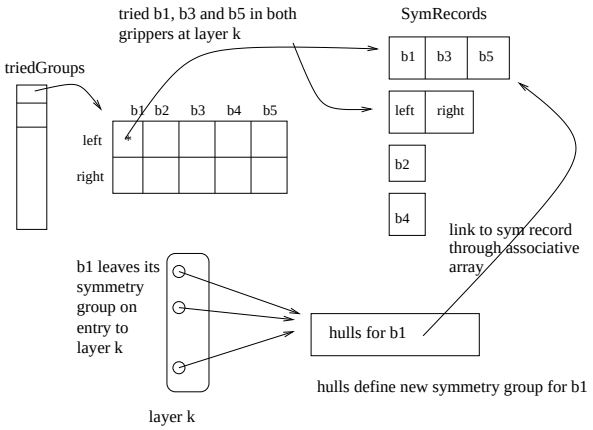

Figure 5: The triedGroups matrix for pickup actions at layer k, indicating that all balls symmetric to $b1$ have been tried in all grippers symmetric to the left gripper.

Our first experiment compared Stan version 3 and SymmetricStan on a range of problems from the Gripper domain. These results are shown in Figure 7. As can be seen, Stan version 3 can solve the first five problems, the largest of which contains 12 balls. Performance is degrading by a factor of about 20, leading us to conclude that Stan version 3 would require in the order of 64 days to solve the 14 ball problem. SymmetricStan solves the whole problem set, to optimality, exhibiting only a quadratically growing increase in requirements (in the log-scaled graph the downward slope of the SymmetricStan curve indicates a polynomial rate of increase).

Stan version 3 already exhibits significant performance improvements, in domains containing symmetry, over raw Graphplan, indicating that the Graphplan search strategy, which attempts to find a plan at every level from the fix-point layer to the solution layer is sensitive to symmetry and performs badly when there are many redundant alternatives to explore and

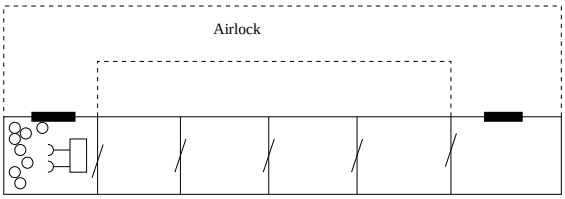

Figure 6: The Airlock domain. Opening a door seals the previously opened door permanently closed.

the fix-point is many layers away from the solution layer (this is the case for the Gripper domain). By contrast, heuristic forward searching planners, such as FF, are unaffected by symmetries in problems where the heuristic value of a state is close to its actual value. In solving these problems FF never has to backtrack into a state from which it would be forced to perform a breadth-first search of a large plateau. However, Hoffmann (Hoffmann 2001) has shown that the heuristic function used by FF produces good estimates only in relatively simply structured problems. As Hoffmann shows, the Gripper domain is simply structured and FF is able to solve arbitrarily large Gripper problems with linear increase in performance requirements.

This does not in any way undermine the importance of symmetry exploitation, as our next experiment demonstrates. In problems where the heuristic function used by FF is not able to provide good estimates of value, the ability to handle symmetries could significantly reduce the sizes of the plateaux that would need to be explored on backtracking.

We constructed the Airlock domain as an example of a highly symmetric domain that FF is unable to solve for the reasons described above. Airlock is an extended version of the Gripper domain in which there is a chain of rooms connected by doors. Opening a door causes the last door to be irrevocably sealed (so the doors are one-way). The last room is connected by a door to an air-lock connected to the first room. The problem is to transport all the cargo stored in the first room into the last room. FF cannot easily solve this problem because it is fooled, by the presence of the airlock, into concluding that the problem remains solvable if the first item of cargo is moved into the last room before any other items are moved. In fact, the only solution is to move all of the cargo into the adjacent room and then keep moving the cargo room by room until the last room is reached (the airlock is never used). To find this solution FF is forced into performing a breadth first search of all of the orders in which the cargo items could have been transported. The Airlock domain gives rise to many new symmetries during search - every one of the

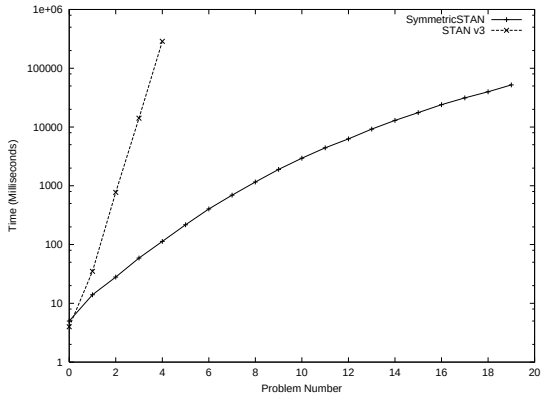

Figure 7: Comparison of Stan version 3 and SymmetricStan on Gripper problems (log-scaled).

rooms in the chain provides the opportunity for a new symmetric collection of cargos. FF cannot, of course, exploit this feature, but SymmetricStan can.

Figure 8 presents the data we collected for the Airlock problem. It can be seen that, although FF performs well (in terms of time consumption – plan quality is quite poor) on small Airlock problems it scales very badly. The graph is log-scaled, and the upward trend of the FF curve therefore indicates that FF's performance requirements grow super-exponentially with the increase in instance size (measured as the number of cargos to be transported). FF was terminated on the penultimate problem (9 cargos) having exhausted available memory. By contrast, SymmetricStan solved all of the problems presented, to optimality, requiring 1.5Mb for solving the problem that defeated FF. The SymmetricStan curve follows more or less a straight line, indicating that there is still exponential growth. We are investigating the reasons for this.

We performed a final experiment to determine consistency of performance of SymmetricStan. We used a range of Ferry instances containing up to 50 cars. SymmetricStan was able to solve all problems with polynomial increases in demands as instance sizes increase. Stan version 3, which itself provides an exponential improvement in performance over raw Graphplan (Fox & Long 1999), exhibits exponential growth and was able to solve instances containing only up to 20 cars.

The ShaPer (Guéré & Alami 2001) system presents an alternative view of symmetries by exploiting symmetry in the domain structure rather than at search time. By pre-planning analysis, some replication in the domain structure can be identified and removed. The underlying structure in both Gripper and Ferry exhibits considerable replication, so we compared SymmetricStan with ShaPer on these domains. Interest-

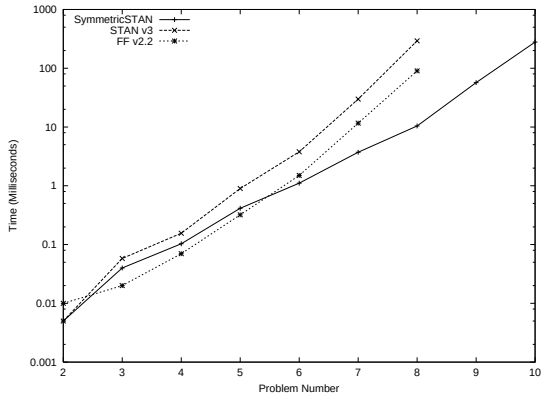

Figure 8: Comparison of time requirements of Stan version 3, SymmetricStan and FF on Airlock problems (log scaled).

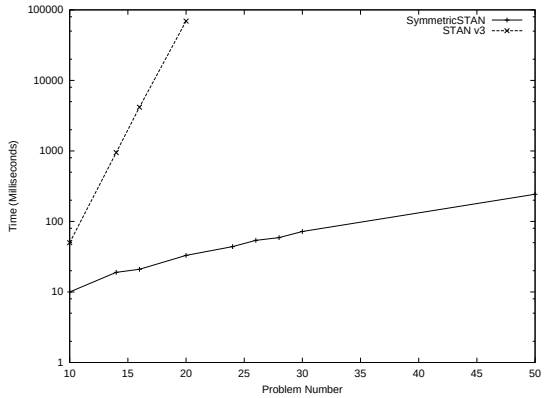

Figure 9: Comparison of time requirements of SymmetricStan and Stan version 3 on Ferry problems (log-scaled).

ingly, ShaPer is faster than SymmetricStan on the Gripper data set, solving the largest problem in about 10 seconds (including pre-planning time), but in the Ferry domain SymmetricStan is faster. The 50 car Ferry problem is solved by SymmetricStan in 120 milliseconds, and by ShaPer in 5.56 seconds (including the pre-planning time). Although the ShaPer approach can be powerful when there is much replicating structure the SymmetricStan approach has the advantage that it can be exploited in any domain that contains any amount of functional identity. The overheads involved in managing the data structures for symmetry exploitation are small so no significant penalty is paid when a domain contains no usable symmetries. This was demonstrated to be the case for Stan version 3 and it remains the case for SymmetricStan.

In (Fox & Long 1999) Fox and Long describe a problem encountered with the use of symmetry-breaking in Stan version 3. When an action combination, $o_i \cdots o_k$ is tried at layer k, and fails to lead to a solution, symmetric action groups containing actions in this combination are marked as tried. No other combination containing any action, from the tried combination, belonging to a tried symmetry group can then be considered at layer k. The symmetric action can be applied, but not until layer $k+1$. This results in some sequentialisation of the plan, and the consequent loss of parallel optimality.

The same behaviour occurs in SymmetricStan with the consequence that we cannot claim to produce parallel optimal plans. When sequentialised, the plans produced by SymmetricStan are often sequentially optimal as in the Gripper and Airlock domains above. However, Graphplan plans are not guaranteed to be sequentially optimal so this side-effect of symmetry can result in loss of optimality for SymmetricStan. We believe the plans produced are close to optimal, but we are currently trying to understand better what guarantees can be given.

Conclusion

This paper presents the benefits to be obtained from proper handling of problem symmetries in the search behaviour of a planner. Initial work in this area, implemented in the Graphplan-based planner Stan version 3, failed to take into account the fact that, in many planning problems, symmetries not present in the initial state of the problem can emerge during the search process. If these new symmetries are not recognised and exploited much of the benefits of handling symmetries are lost. We have described what is needed to extend the symmetry-breaking mechanisms of Stan version 3 to take account of these new symmetries. The resulting system is SymmetricStan. The data presented here shows that SymmetricStan obtains exponential improvements in performance over raw Graphplan, FF and Stan version 3 itself.

Although our discussion has been largely in terms of Graphplan-style planning the basic exploitation strategy for handling symmetries is fully general. Extensions similar to those described here would be necessary to implement symmetry-breaking in any backtracking search process. In a forward reachability analysis symmetries can be exploited to prune alternatives layer by layer, and it is straightforward to consider using a data structure similar to triedGroups to store the information necessary to manage this pruning.

Several issues remain to be considered in future work. We have so far only considered the implementation of symmetry-breaking in a propositional planning

framework. The idea is easily extended to ADL planning, and the implementation we have described here could equally be achieved in an ADL planning context, for example in the Graphplan-based ADL planner IPP (Koehler *et al.* 1997). We are so far only exploiting symmetry that corresponds to what we call functional identity – there are other forms of symmetry that could bring benefits if we could characterise them reliably. For example, in many planning problems there are objects that are *almost* functionally identical, and there may be ways of determining whether the senses in which they are not identical are relevant or not to the solution of the problem.

The ability to recover symmetries, as is possible in SymmetricStan, relies on the automatic identification of symmetries during the search process. An important distinction between our work and the exploitation of symmetries in the CSP community, is that we can use our notion of functional identity to enable dynamic automatic detection of certain symmetries using inexpensive domain analysis techniques during the search process.

References

Blum, A., and Furst, M. 1995. Fast Planning through Plan-graph Analysis. In *Proceedings of 14th IJCAI.*

Crawford, J.; Ginsberg, M.; Luks, E.; and Roy, A. 1996. Symmetry breaking predicates for search problems. In *Proceedings of KR.*

Fox, M., and Long, D. 1999. The detection and exploitation of symmetry in planning problems. In *Proceedings of 16th IJCAI.*

Gent, I. P., and Smith, B. 2000. Symmetry breaking during search in constraint programming. In *Proceedings of ECAI.*

Guéré, E., and Alami, R. 2001. One action is enough to plan. In *Proceedings of 17th IJCAI*, 439–444.

Hoffmann, J., and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *JAIR* 14.

Hoffmann, J. 2001. Local search topology in planning benchmarks: an empirical analysis. In *Proceedings of 17th IJCAI.*

Ip, C. N., and Dill, D. L. 1996. Better verification through symmetry. *Formal Methods in System Design* 9.

Koehler, J.; Nebel, B.; Hoffmann, J.; and Dimopoulos, Y. 1997. Extending planning graphs to an ADL subset. In *Proc. ECP'97, Toulouse*, 273–285.

Roy, P., and Pachet, F. 1998. Using symmetry of global constraints to speed up the resolution of csps. In *Workshop on Non-binary Constraints, ECAI.*