Exploiting Problem Symmetries in State-Based Planners

Nir Pochter
School of Eng. and Computer Science
The Hebrew University of Jerusalem
Jerusalem, Israel
nirp@cs.huji.ac.il

Aviv Zohar
Microsoft Research
Silicon Valley
Mountain View, CA
avivz@microsoft.com

Jeffrey S. Rosenschein
School of Eng. and Computer Science
The Hebrew University of Jerusalem
Jerusalem, Israel
jeff@cs.huji.ac.il

Abstract

Previous research in Artificial Intelligence has identified the possibility of simplifying planning problems via the identification and exploitation of symmetries. We advance the state of the art in algorithms that exploit symmetry in planning problems by generalizing previous approaches, and applying symmetry reductions to state-based planners. We suggest several algorithms for symmetry exploitation in state-based search, but also provide a comprehensive view through which additional algorithms can be developed and fine-tuned. We evaluate our approach to symmetry exploitation on instances from previous planning competitions, and demonstrate that our algorithms significantly improve the solution time of instances with symmetries.

1 Introduction

Classical planning problems are notorious for the exponentially large number of states that need to be navigated in order for a solution to be found. Such problems, however, often contain symmetries. In that case, many of the different states in the search space are in fact symmetric to one another, with respect to the task at hand, and the result of searching through one state will inevitably be equivalent to searches through all of its symmetric counterparts. Though detecting and exploiting such symmetries would shrink the search space, the actual exploitation of symmetries is made difficult by the fact that symmetry detection is potentially complex. All currently known algorithms for the highly related problem of finding graph automorphisms work in exponential time, in the worst case. The situation is worsened by the fact that the graph on which we wish to detect automorphisms is the huge search space that the planning problem defines; it does not typically fit into memory.

The idea of exploiting symmetries has appeared in model checking, e.g., in work by Emerson and Sistla (1996), and in the context of constraint satisfaction problems, beginning with a seminal paper by Puget (1993). The most common approach uses additional constraints to break existing symmetries (e.g., see (Walsh 2007)) and reduce the search space to a more manageable size. Planning problems can be reduced to various constraint satisfaction problems and to propositional logic formulas. Some have tried to use this approach to apply symmetry reduction in search (Miguel 2001; Rintanen 2003). Rintanen adds symmetry-breaking constraints to the propositional logic representation of transition sequences derived from planning problems to remove symmetries from different points in the sequence. However, state-of-the-art planners usually employ a state-based approach combined with heuristic search. It is unclear how to directly translate knowledge in symmetry exploitation in earlier planning systems and CSPs to improvements in today's most efficient planners.

A previous line of work by Fox and Long (1999; 2002) has shown how to prune symmetries in planning via modifications of their GraphPlan-based algorithm (which is not state-based). Another approach is to use symmetries to improve heuristics instead of pruning. Porteous, Long and Fox (2004) do so for "almost-symmetries" in the FF planner.

Our own work can be seen as a continuation of this line of research, and its extension to the more ubiquitous state-based planners. Among our contributions, we present a framework for the understanding of symmetry exploitation from the state-based point of view.

The essence of our approach to symmetry exploitation is made up of two main steps. First, we explain how knowledge of symmetries in the search space can be useful to prune some of the search done by state-based algorithms such as A*; for this, we employ techniques similar to the canonicalization used by Emerson (1996) as well as pruning of shallower symmetries similar in spirit to Rintanen (2003). We then explain how to deduce the existence of relevant symmetries in the search space from the smaller problem description (in a way similar to Cohen *et. al.* (2006) which have done so for CSPs). Since our approach works at the core of the search algorithm by pruning branches of the search, it is completely independent of any heuristic that may be applied to guide that search, and can assist whenever the planning problem contains some amount of symmetries.

1.1 Example

We begin by presenting an example that has appeared in the International Planning Competition (IPC), and has been examined in the context of symmetries by Fox and Long: the gripper domain. The gripper domain consists of two rooms, n numbered balls, and a robot with two arms, or grippers.

The robot can pick up a single ball in each of its grippers, move between rooms, and put balls down. A problem instance is usually set up in the following manner: all balls and the robot begin in one room, and the goal is to move the balls to the other room.

The problem seems trivial to solve (at least for humans), but is surprisingly difficult for planners. The difficulty stems from the fact that the balls in the problem are effectively interchangeable, but are not considered as such by conventional planning algorithms. The balls may be picked up in any of $n!$ possible orders, each of which will lead to a different plan, and planners may potentially try all such orders. The reason for this was well-explained by Helmert and Röger (2008), where it is shown that any A* search with an imperfect heuristic is bound to explore states on multiple shortest paths to the goal. Our objective is to identify during search that some states (and therefore the plans going through them) are symmetric to one another, and can thus be pruned; e.g., the state in which the robot holds one ball in its right gripper (and the left is free) is symmetric to the state in which the ball is held in the left gripper (with the right one free). Only one of the two states needs to be explored in order to find an optimal plan. This means that on domains in which symmetries occur, the heuristic will no longer have to be perfect in order to avoid opening most of the states.

2 Preliminaries

We consider planning in the SAS$^+$ formalism (Bäckström and Nebel 1995). Tasks that are represented in STRIPS or PDDL can be automatically converted to SAS$^+$(Helmert 2006), which is sufficiently general, and is used in Fast-Downward—the planner used in our implementation. The algorithms we present do not take into account action costs and conditional operators, but these, and other additional structures in the planning language, can be accounted for with only minor modifications.

2.1 Definitions for Planning Problems

Definition 1 *A planning task is a tuple $(V, O, s_0, S_\star)$:*

- *$V = \{v_1, ..., v_n\}$ is a finite set of state variables, each with a finite domain D_v.*[1] *A fact is a pair $\langle v, d\rangle$ where $v \in V$ and $d \in D_v$. A partial variable assignment is a set of facts, each with a different variable. A state is a partial variable assignment defined over all variables V.*
- *O is a set of operators o specified via $\langle pre(o), eff(o)\rangle$, both being partial variable assignments to V.*
- *s_0 is the initial state.*
- *$S_\star$ is a partial variable assignment which is the goal.* [2]

An operator $o \in O$ is applicable in a state s iff $pre(o) \subseteq s$.

The states of a problem naturally define a search space in the form of a state transition graph—a directed multigraph (S, E), where S is the set of all states, and E, the set of edges, contains a directed edge from state s to state s' exactly for every operator that is enabled at s and leads to s'. [3]

An instance of a planning problem is then fully specified by designating an initial state s_0 and goals $S_\star$ with the objective of finding the shortest possible path (i.e., a plan) between s_0 and some $s_\star \in S_\star$.[4]

2.2 Graph Automorphisms, Permutation Groups

The symmetries in the problem are manifested as automorphisms in the state transition graph. We briefly present some relevant definitions from Group Theory.

An automorphism of a graph (S, E) is a permutation $\sigma : S \to S$ of the vertices of the graph that maintains the structure; i.e., for every two vertices $s_1, s_2 \in S$, we have that $(s_1, s_2) \in E$ iff $(\sigma(s_1), \sigma(s_2)) \in E$. The automorphisms of (S, E), with the composition action, constitute a *group* denoted $Aut(S, E)$. Unless otherwise specified, we will denote by G some subgroup of $Aut(S, E)$ (and refrain from using it to denote graphs).

Definition 2 *$\Sigma = \{\sigma_1, \ldots, \sigma_k\}$ is said to* generate *a finite group G, if G is exactly all the permutations that are obtained by repeatedly composing elements of Σ.*

Finding a generating set for the automorphisms of a graph is harder than solving the graph isomorphism problem (for which no polynomial-time algorithm is currently known). See the paper by Luks (1993) for an overview of complexity results relating to permutation groups. Generating sets are usually found using brute force backtracking search with some specialized pruning techniques that allow the set to be found surprisingly quickly in practice.

Definition 3 *The* orbit *of a vertex s with respect to some subgroup G is denoted by $G(s)$ and is simply the set of vertices to which elements in G map s. $G(s) = \{\sigma(s) \mid \sigma \in G\}$.*

Given a generating set for G, the orbit of vertices can be computed in polynomial time. We will be especially interested in subgroups of $Aut(S, E)$ that fix a certain vertex or a set of vertices. These are defined below:

Definition 4 *The* point-stabilizer *of a vertex s with respect to G, denoted G_s is a subgroup of G that contains all the permutations that fix s. $G_s = \{\sigma \mid \sigma \in G, \sigma(s) = s\}$.*

Definition 5 *The set-stabilizer of a set $X \subset S$ with respect to G is the subgroup of G that maps elements of X only to X. $G_X = \{\sigma \in G \mid \forall x \in X \;\; \sigma(x) \in X\}$.*

Finding generators for the stabilizer of a single vertex can be done in polynomial time. In contrast to that, finding generators for the set-stabilizer group is a more difficult problem, that is known to be harder than graph isomorphism.

[1] To simplify definitions, we assume unique names for the values, so given a value it is known to which variable it corresponds.

[2] We will slightly abuse notation and use $S_\star$ to denote the set of goal states.

[3] Equivalent operators may exist, and so there can be more than one edge between two states.

[4] We often refer to states in the planning problem as vertices in the transition graph, and vice versa.

3 Symmetries in the Search Space

Our goal in this section is to show how the symmetries in the transition graph can be used to prune the search. The first observation we make is that stabilizing vertices can help when automorphisms are applied to paths.

Observation 1 *Let (S, E) be a graph with a group G of automorphisms, and let $(s_0, s_1, \ldots, s_\star)$ be a sequence of vertices that forms a path from s_0 to $s_\star$. If we simply apply some $\sigma \in G$ to the path, we will have another path of equal length: $(\sigma(s_0), \sigma(s_1), \ldots, \sigma(s_\star))$. This path, however, may not begin at s_0, and thus may be of little value. Instead, we may choose to fix a given vertex s_i along the path via the point-stabilizer subgroup G_{s_i}, and apply the transformation only to some part of the path. From the application of any $\sigma \in G_{s_i}$ to the suffix $s_i, \ldots, s_\star$, we get a path $(s_0, s_1, \ldots, s_{i-1}, \sigma(s_i), \sigma(s_{i+1}), \ldots, \sigma(s_\star))$ which is connected because $s_i = \sigma(s_i)$. This path does begin at s_0.*

This rather basic observation will form the basis of our modification of state-based planning algorithms, and specifically ones that are based on A*. To use it, we will need to be able to perform three basic tasks: to find $G \subset Aut(S, E)$, to compute the stabilizer G_s of a state s, and to determine for two states s_1, s_2 if $s_1 \in G(s_2)$.

We discuss how to perform these three tasks in the next section, and for now present how they may be used in a planning problem. As we are constrained for space, we assume the reader is intimate with the workings of the A* algorithm, and do not specify it. In addition, we do not present full proofs of the correctness of our algorithms, which follow along similar lines to the proofs of A*'s correctness.

3.1 Searching in Orbit Space

Observation 1 suggests that in order to ensure that symmetric paths always connect to s_0, we may stabilize any vertex along a path that starts there. One vertex will surely be along any path from s_0, and that is s_0 itself. A simple approach would then be to restrict ourselves to the group G_{s_0}.

Our goal is to find a shortest path from s_0 to any of the nodes in $G_{s_0}(S_\star)$. Once such a path is found to some node $\sigma(s_\star)$, we can translate it to a path from start to goal by simply applying σ^{-1} on the entire path. To gain the most benefit from our search procedure, we do not conduct the procedure on the original search space, but rather consider orbits of the search space as nodes in our search, in a manner similar to that of Emerson and Sistla (1996). For this purpose we define a graph with vertex set S_{orb}, the set of orbits of vertices in the transition graph with respect to the group G_{s_0}. The edge set that we will use is E_{orb} which will contain an edge from one orbit to another iff there exists an edge between nodes from these orbits in the original graph. $S_{orb} = \{G_{s_0}(s) | s \in S\}$; $E_{orb} = \{(o_1, o_2) \in S_{orb} \times S_{orb} \,|s_1 \in o_1,\ s_2 \in o_2,\ (s_1, s_2) \in E\}$

We can conduct a search in (S_{orb}, E_{orb}) and the resulting path will translate to a plan in the original transition graph. While we wish to run a search on the graph of orbits, we really only know how to search in the original search space. Our heuristic function may also only apply to states and not to orbits. Therefore, the practical search algorithm simulates a search on the orbits through a search in the original space, by using a single state from each orbit as its representative:

1. Before the search, find generators for the group G_{s_0}.
2. Whenever generating a node s_2, search for a previously generated node s_1 such that $s_2 \in G_{s_0}(s_1)$. If such a node was found, treat s_2 exactly as if it was s_1, otherwise, s_2 represents a newly found orbit.
3. Stop the search when expanding a node from $G_{s_0}(s_\star)$.

Alas, the stopping condition of the algorithm we present is not guaranteed to work with all heuristic functions. In fact, since we stop the search at some node from $G_{s_0}(s_\star)$ that is not necessarily a goal state, our algorithm must assure us that when this node is expanded, its f-value is minimal. For general admissible heuristics it is not guaranteed that $h(\sigma(s_\star)) = 0$, but an admissible heuristic that will give all nodes in $G_{s_0}(s_\star)$ the same h-value of 0, will work.

Notice that the above algorithm will never expand the children of a node that is not from a new orbit. It will be matched to a previously generated node, and the A* algorithm will treat it as that node (it may update its f-value if it has a lower one than the previously-seen representative for the orbit, but it does not need to be expanded).

While many heuristics that will give all goal-symmetric states the same value do exist, we would like an algorithm that would function with admissible heuristics that are not necessarily symmetric. In addition, as we will later see, the need to check the halting condition of the algorithm for each node (trying to see if it is in the orbit of the goal) requires a great deal of computation. Our next modification (which is also the one used in our experiments) will allow us to simplify this check, and to use heuristics that are just admissible, by using a somewhat restricted set of symmetries.

3.2 Stabilizing the Goal

Our previous algorithm was not guaranteed to reach a goal state, which caused it to require complicated checks for the terminating condition and to use only goal-symmetric heuristics. To solve this issue, we work with the symmetry group $G_{s_0,S_\star}$ that stabilizes both the start, and the partial assignment $S_\star$. The modified algorithm is as follows:

1. Before the search, find generators for the group $G_{s_0,S_\star}$.
2. Whenever generating a state s_2, search for a previously generated state s_1 such that $s_2 \in G_{s_0,S_\star}(s_1)$. If such a state was found, treat s_2 exactly as if it was s_1, otherwise, s_2 represents a newly found orbit.
3. Stop the search when a goal state is expanded.

This algorithm is also effectively a search in the orbit-space of a smaller group, but the orbit of goal states contains only goal states, and so our halting condition is simplified. An additional benefit is that we can relax the search for matching states in step 2. If we fail to match a state to another one in its orbit, it will simply be added as a node in the search but will not change the minimal length of the path we end up finding. We can therefore do this matching in a heuristic manner (but we must make sure that no false positive matches are made). Our previous approach required exact matching in order to identify the goal state correctly.

3.3 Shallow Pruning of Symmetric States

Both of the previous algorithms have stabilized the start state and have thus limited the symmetries that were used. To show that other approaches may be possible, we will briefly exhibit another modification that can be implemented alone or combined with search in orbit-space that will work through somewhat different symmetry groups.

When expanding a certain state s during search in highly symmetric problems, we often find that we are generating a large number of child states, many of which are symmetric to one another. We then waste effort matching each of them to the orbits of previously generated states, and pruning them then. It would be ideal if we could use a shallow pruning technique to avoid generating some of these immediate children.

1. For each state s that is being expanded, compute the pointwise stabilizer of the goal and s itself: $G_{s,S_\star}$.
2. Choose one operator from each orbit in $G_{s,S_\star}$ to apply in s (given that it is enabled), and avoid applying the others.[5]
3. Stop the search once a goal state has been expanded.

The intuition behind the modification is as follows. If s is not itself on the shortest path to the goal, then failing to generate some of its children will not affect the path found in the search (if one of its children is on the shortest path, it will be a child of another node as well). If it is on the shortest path, then two symmetric actions will only lead to symmetric states (with respect to $G_{s,S_\star}$) and a path through one such state to the goal can be turned into a path through the other, as is demonstrated in Observation 1.

This form of symmetry pruning is somewhat costly as it requires many computations of the state stabilizer. It is very similar in spirit to the approach of Fox and Long (2002) which they term dynamic symmetries. In fact, from the state-based perspective, it is not the symmetry group that changes, but rather the subgroup of it that we exploit at any given point. The shallow pruning algorithm fails to remove many symmetric states from the graph when applied alone (symmetries between states that do not have a common parent will not be detected), but it can be combined with our second version of search in orbit space.

4 Symmetries in the Problem Description

Above, we have shown how to exploit symmetries to speed up search. We now explain how to detect such symmetries, and how to perform the basic computations we needed.

Our approach to symmetry detection is similar to the ones used in previous research. We detect symmetries in the description of the problem and infer through these about symmetries in the transition space, i.e., we identify places where variables, values and operators can be renamed (permuted somehow) so as to leave us with exactly the same problem as before. These syntactic symmetries in the problem description, which is much smaller than the search space, imply the existence of symmetries in the transition graph.[6] To find syntactic symmetries, we construct a graph that encodes the problem's structure, and use conventional group-theoretic algorithms to find generators for its automorphism group. While the algorithms to find the generators for the automorphism group run in worst-case exponential time, they are quite manageable for graphs of hundreds of thousands of vertices. These suffice to describe typical planning problems that challenge current planners.

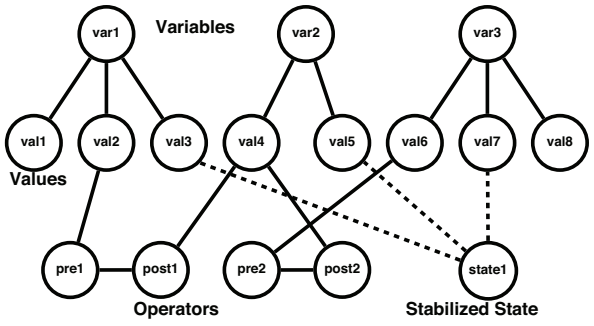

Figure 1: A problem description graph (PDG) with a stabilized state.

Formally, we define an undirected graph for a planning problem, that we call the *Problem Description Graph* (PDG). The PDG has four types of vertices: there is a vertex for every variable, every value, and two vertices for each operator—one that represents the preconditions and one that represents the effects. The vertices are connected by edges as follows. We connect every variable vertex to the values that belong to it, every precondition and effect belonging to the same operator, every precondition vertex to the values in its precondition list, and finally, each effect vertex to the values in its effects list. Figure 1 illustrates such a graph.

We are specifically interested in the group G^{PDG} of automorphisms of the PDG that are restricted to map vertices of each type (variables, values, preconditions, and effects) only to vertices of the same type. Several tools solve this problem of colored-graph automorphism quickly. Our own implementation uses Bliss (Junttila and Kaski 2007).

Observation 2 *Every permutation $\pi \in G^{PDG}$ can be interpreted as a permutation operator on states. We denote this operation by $\hat{\pi} : S \to S$. It is defined in the following manner: $\hat{\pi}(s) = \{\langle \pi(v), \pi(d) \rangle \mid \langle v, d \rangle \in s\}$*

That is, if the value d is assigned to variable v in state s then the value $\pi(d)$ holds in variable $\pi(v)$ in state $\hat{\pi}(s)$. The edges between variables and their values in the PDG assure us that $\hat{\pi}(s)$ will be a legal state (all variables will be assigned exactly one value), as π is thus restricted to preserve the link between a variable and its values.

Proposition 1 *The operator $\hat{\pi}$ (as defined above in Observation 2 for any corresponding $\pi \in G^{PDG}$) is a proper permutation on states. Furthermore, $\hat{\pi} \in Aut(S, E)$.*

[5] The orbits of an edge in the transition graph is the set of edges to which it is mapped by the group action.

[6] Fox and Long (1999) advocate using "object symmetries" which use more restricted subgroups than those we detect, but may be found faster in practice. This alternative approach is compatible with our methods of exploitation.

The proposition holds because the edges in the PDG encode the entire structure of the planning problem and any (colored) automorphism of the PDG is equivalent to renaming the operators, variables and values in a way that preserves the semantics. A similar observation about the relation between syntactic symmetry and semantic symmetry has been made in the context of CSPs (Cohen et al. 2006). We do not provide the proof due to lack of space.

4.1 Stabilizing a Given State

We now show how to look for subgroups that stabilize a given state, using the PDG representation. Given a state s, we are interested in all $\pi \in G^{PDG}$ for which $\hat{\pi}(s) = s$.

Observation 3 *Using the definition of $\hat{\pi}$'s operation on states, we can see that in order for a state to remain unchanged under $\hat{\pi}$, it must be that π maps all values that hold in s onto themselves: $G_s = \{\hat{\pi} | \langle v, d\rangle \in s \rightarrow \langle \pi(d), \pi(v)\rangle \in s\}$. This means that in order to find G_s we just need to compute the set stabilizer (within G^{PDG}) of all values that hold in s.*[7]

We ensure that assigned values are mapped only into each other by adding a vertex to the PDG, connecting it only to assigned values (as depicted in Figure 1), and then computing generators for the automorphisms of this modified graph.

4.2 Locating States from the Same Orbit

Given a group G, and a state s, we wish to know if our search has already encountered a state in $G(s)$. One naive (and time-consuming) approach would be to iterate over all previously encountered states, and to attempt to compute $\sigma \in G$ that will perform the match. Each such comparison is in itself a computational problem (that is harder than Graph-Iso). We shall instead use a method that is commonly employed in computational group theory: the canonical form. We map each state s into another state $C(s) \in G(S)$ that acts as a representative for the orbit. $C_G()$ is a mapping such that: $C_G(s) = C_G(s')$ iff $\exists \sigma \in G : s = \sigma(s')$. One possible way to pick a representative state is to look for the one that has a lexicographically minimal representation. Using the canonical form, we are able to rapidly check if a new state is symmetric to one that has been previously encountered by keeping the canonical form of all states we encounter in some form of a hash-set (or any other data structure that lets us test membership in O(1) time). Since A* keeps such data sets of generated states anyway, we simply use the canonical form as the key instead of the state itself. Unfortunately, is is NP-hard to find the lex-minimal state (Luks 1993).

Approximate canonical forms If we wish to speed up the canonical-form computation for states, we can use a heuristic approach. Instead of finding the lexicographically smallest state in $G(s)$, we greedily search for one that is relatively small (by applying various permutations) and keep the best state we encounter. We may often find that instead of mapping all members of $G(s)$ to a single state, we map them to several ones that are local minima in our greedy search. In this case we do not reduce the search space as effectively, but we do it much faster.

[7] Partial assignments are stabilized in the same manner.

5 Experimental Evaluation

Symmetry exploitation has the potential to slow search if the savings it provides in the number of explored states are negligible in comparison to the additional running time it requires per state. We therefore conducted our experiments with the aim of checking if the pruning algorithm fulfills its purpose, i.e., if it makes the search faster on problems that contain enough symmetries, while not severely hurting performance in problems that do not have any.

Since shallow pruning requires calculating automorphisms that stabilize each state, it is too slow to use if the canonical state evaluations that it saves are very efficient. Our search in orbit space that was based on greedy canonicalization attempts proved so fast that we did not gain from additional shallow pruning (but slower approaches, e.g., exact matching, benefit a great deal).

We implemented our algorithm on the Fast Downward system (Helmert 2006), and compared between the running times of the planner with and without our modifications. Our tests were conducted using various heuristics and were done on domains from recent planning competitions (IPCs). For symmetry detection, we used Bliss (Junttila and Kaski 2007), a tool that finds generators for automorphism groups. All experiments were conducted on Linux computers with 3.06GHz Intel Xeon CPUs, using 30-minute timeouts. For all experiments we set a 2GB memory limit.

We used three heuristics for testing: first, LM-Cut (Helmert and Domshlak 2009), which is one of the best heuristic function known today; second, Merge-and-Shrink (Helmert, Haslum, and Hoffmann 2007), which is less informative, but has a different behavior as it spends more time in pre-processing and less during the heuristic calculation for each state; finally, we also tested using h^0 (blind search).

Table 1 shows the number of tasks solved by each heuristic, with and without the symmetry-pruning algorithm.[8] The biggest advantage was, as expected, in the gripper domain, where all heuristics failed to solve more than 7 tasks without our modification, and all of them solved all 20 tasks with it. Even though other domains contained relatively few symmetries, using symmetries for pruning still improved the results in some of those domains.

Table 2 shows the number of node expansions and search time for each heuristic in selected tasks, both with and without symmetry detection. Empty cells correspond to runs that exceeded the time and memory constraints. We chose to display a sample of instances from the gripper domain (where symmetries were abundant). Other domains had fewer symmetries, and we picked the mprime domain, where we did not solve additional instances, but did improve the running time of some tasks, as a typical example (we display the

[8] Because of space limitations, we did not present these numbers for domains in which the symmetry detection algorithm did not change the number of tasks solved, which are: airport, blocks, depot, drivelog, freecell, grid, logistics00, mystery, mprime, openstacks, pathway-noneg, rover and trucks.

	h^0		$h^{m\&s}$		h^{LM-cut}	
domain	reg	sym	reg	sym	reg	sym
gripper(20)	7	**20**	7	**20**	7	**20**
logistics98(35)	2	2	4	5	**6**	**6**
miconic-STRIPS(150)	50	50	53	56	**141**	**141**
pipesworld-notankage(50)	14	16	21	**22**	17	18
pipesworld-tankage(50)	9	13	14	**16**	10	13
psr-small(50)	48	49	**50**	**50**	49	49
satellite(36)	4	5	6	6	7	**9**
tpp(30)	5	6	6	6	6	**7**
zenotravel(20)	7	8	11	11	12	**13**
Total	273	296	316	336	434	**455**

Table 1: The number of solved tasks per heuristic

	h^0				$h^{m\&s}$				h^{LM-cut}			
	expanded		**time(sec)**		**expanded**		**time(sec)**		**expanded**		**time(sec)**	
inst	reg	sym	reg	sym	reg	sym	reg	sym	reg	sym	reg	sym
gripper												
7	10.1M	1.72K	90.0	0.16	10.1M	937	126.5	2.83	10.1M	561	1320	0.14
8	—	2.01K	—	0.25	—	1.35K	—	3.62	—	740	—	0.24
20	—	60.5K	—	66.32	—	16.0K	—	41.07	—	3.14K	—	10.5
mprime												
5	—	—	—	—	1.70M	565K	161.2	147.6	46.2K	22.6K	525	224.1
12	108K	77K	5.92	6.03	35.0K	29.8K	8.78	9.18	122	87	1.57	0.97
19	—	—	—	—	150K	150K	208.3	212.4	—	—	—	—
21	1.5M	438K	186	360	—	—	—	—	900	504	229	97
logistics-9-0-extra-trucks												
2	—	—	—	—	1.07M	325K	43.3	22.3	74.5K	22.9K	130	30.7
3	—	—	—	—	4.12M	657K	188	57.6	183K	30.7K	425	54.2
4	—	—	—	—	—	—	—	—	286K	31.4K	946	71.3

Table 2: Expanded nodes and search time in select instances

four solved instances with the maximal number of expanded nodes). While some cases show improvements in both node expansions and running time (instances with enough symmetries), others exhibit improvements only in node expansion or none at all (few or no symmetries). Still, the loss in running time is never severe. The improvement in the number of node expansions did not lead to the same effect in search time in all heuristics. $h^{m\&s}$ and h^0 spend very little time per state evaluation (and so symmetry pruning is less effective), while h^{LM-cut}, which is currently the best heuristic known, takes more time to calculate per state.

As the IPC problems are often constructed with few symmetries (perhaps in an attempt to focus on other difficulties), we sought to demonstrate that symmetries *can* occur naturally. The third set of instances we exhibit in Table 2 was created by taking a problem from the logistics domain and adding more trucks at the same locations as those of existing trucks.[9] The instance numbers described match the number of trucks that were added to the instance, and our algorithm does exploit emerging symmetries.

6 Conclusions

We presented a set of algorithms that we applied to state-based planners in order to help them deal with highly symmetric problems. Implementing our techniques shows that significant improvements in symmetric instances can be achieved without serious harm to performance elsewhere.

[9]The domain deals with delivery of packages using trucks.

Future work includes improving the approximation of the canonical state, and finding fast variants of shallow pruning that will make it more useful in practice. Also, we have not fully utilized all symmetries in the transition graph, and believe that other symmetry exploitation algorithms can be developed within the general framework we have outlined.

Acknowledgments This work was partially supported by Israel Science Foundation grant #898/05, and Israel Ministry of Science and Technology grant #3-6797.

References

Bäckström, C., and Nebel, B. 1995. Complexity results for SAS+ planning. *Computational Intelligence* 11:625–656.

Cohen, D.; Jeavons, P.; Jefferson, C.; Petrie, K.; and Smith, B. 2006. Symmetry definitions for constraint satisfaction problems. *Constraints* 11:115–137.

Emerson, E. A., and Sistla, A. P. 1996. Symmetry and model checking. *Formal Methods in System Design* 9(1/2):105–131.

Fox, M., and Long, D. 1999. The detection and exploitation of symmetry in planning problems. In *Proceedings of IJCAI'99*, 956–961.

Fox, M., and Long, D. 2002. Extending the exploitation of symmetries in planning. In *Proceedings of AIPS'02*, 83–91.

Helmert, M., and Domshlak, C. 2009. Landmarks, critical paths and abstractions: What's the difference anyway? In *Proceedings of ICAPS'09*, 162–169.

Helmert, M., and Röger, G. 2008. How good is almost perfect? In *Proceedings of AAAI'08*, 944–949. AAAI Press.

Helmert, M.; Haslum, P.; and Hoffmann, J. 2007. Flexible abstraction heuristics for optimal sequential planning. In *Proceedings of ICAPS'07*, 176–183.

Helmert, M. 2006. The fast downward planning system. *Journal of Artificial Intelligence Research* 26:191–246.

Junttila, T., and Kaski, P. 2007. Engineering an efficient canonical labeling tool for large and sparse graphs. In *Proceedings of ALENEX'07*, 135–149. SIAM.

Luks, E. M. 1993. Permutation groups and polynomial-time computation. *Groups and Computation, DIMACS series in Discrete Math and Theoretical Comp. Sci.* 11:139–175.

Miguel, I. 2001. Symmetry-breaking in planning: Schematic constraints. In *Proceedings of the CP'01 Workshop on Symmetry in Constraints*, 17–24.

Porteous, J.; Long, D.; and Fox, M. 2004. The identification and exploitation of almost symmetry in planning problems. In Brown, K., ed., *Proceedings of the UK PlanSIG'04*.

Puget, J.-F. 1993. On the satisfiability of symmetrical constrained satisfaction problems. In *Methodologies for Intelligent Systems*, volume 689 of *LNCS*. Springer. 350–361.

Rintanen, J. 2003. Symmetry reduction for SAT representations of transition systems. In *Proceedings of ICAPS'03*, 32–41.

Walsh, T. 2007. Breaking value symmetry. In *Principles and Practice of Constraint Programming (CP'07)*, volume 4741 of *LNCS*. Springer Berlin / Heidelberg. 880–887.