Problem-Structure-Based Pruning for Cost-Optimal Planning

Amanda Coles and Andrew Coles
University of Strathclyde,
Glasgow, UK
firstname.lastname@cis.strath.ac.uk

Abstract

In this paper we focus on efficient methods for pruning the state space in cost-optimal planning. The use of heuristics to guide search and prune irrelevant branches has been widely and successfully explored. However, heuristic computation at every node in the search space is expensive, so reducing the number of nodes to be considered by up-front analysis has great potential. Our contributions are not concerned with heuristic guidance, rather completeness-preserving pruning techniques that reduce the number of states a planner must explore to find an optimal solution. We focus on search space reduction and aim to draw the attention of the optimal planning community towards pruning techniques based on problem analysis that are orthogonal to the use of heuristics. We present results showing that our techniques can improve upon state-of-the-art optimal planners, both when using blind search and importantly in conjunction with modern heuristics, thus improving their potential.

1 Introduction

In cost-optimal planning, the key challenge in scaling to larger problems is the sheer size of the search space that must be explored to find a provably optimal solution. In attempting to address this, a great deal of attention has been paid to the creation of heuristics. The function of heuristics can be thought of in two ways: guiding the planner towards exploring states that are most likely to lead to a goal; and pruning (or at least heavily deferring the exploration of) states that are never going to (cost-efficiently) lead to a goal state. These two problems are traditionally considered together as part of the heuristic evaluation step performed on a per-state basis.

As recently as 2008, the optimal track of the International Planning Competition alerted the community to the fact that heuristic computation can be very expensive: the Baseline planner, performing blind-search, solved more problems than any of the other then-state-of-the-art planners in the competition. More recently, new heuristics such as $h_{\text{LM-CUT}}$ (Helmert & Domshlak, 2009) have been produced, that (in combination with A* search) are capable of outperforming the Baseline planner. The observation remains, however, that the overhead of heuristic computation at each state is a comparatively expensive way of finding that a given state is of no interest, so there are gains to be made through minimizing the number of such states at which heuristic computation must be performed.

Also, theoretical work by Helmert & Röger (2008) demonstrates that for a number of planning benchmarks, even if a heuristic is near-perfect, an exponential number of nodes must be visited before a provably optimal solution is found. This further suggests that heuristics alone is unlikely to be a panacea, and other mechanisms of pruning the search space must be explored in order to scale to larger problems.

We propose an approach orthogonal, and complementary, to per-state heuristic computation: reducing the branching factor during search using information obtained from up-front static analyses. Such an approach shifts the burden of analysis away from search-time heuristic evaluation to preprocessing, thereby cutting per-node overheads to a minimum. We present a number of techniques to reduce the number of nodes that must be considered. The over-arching theme linking all the techniques is the pre-search analysis of SAS+ encodings, identifying parts of the search space that can be avoided — whilst preserving optimality. Our focus is on techniques leading to one of three observations: a given action is irrelevant, can be removed from the planning task; a given action should always be applied when certain conditions hold; and a given action should never be applied if certain conditions hold. Our analysis represents direct consideration of the second of the aforementioned requirements of a heuristic; rather than relying on this being tackled as a side-effect of reasoning about the first.

We note that our techniques are not limited to cost-optimal planning and can be applied in satisfycing planning. Our focus here is on the former, as a planner must not only find a valid solution; but also explore enough of the search space to ensure no better solution exists. We evaluate the use of our techniques in two contexts. The first is the Baseline planner, which has no per-node heuristic overheads, and hence can expand many nodes very efficiently; but tends to exhaust available memory. Second, in contrast to this, we consider the recent $h_{\text{LM-CUT}}$ heuristic: computationally expensive, but giving good heuristic estimates allowing the exploration of fewer nodes. Empirically, its weakness lies in the per-node heuristic computation overheads, meaning it is more likely to hit a time limit imposed when solving problems. We explore how our techniques enhance performance in each of these very different settings.

2 Background

In this paper we consider cost-optimal planning, using the SAS+ representation (Bäckström & Nebel, 1995). Defined formally, a cost-optimal planning problem is a tuple $\Pi = \langle V, s_0, s_\star, O, cost \rangle$ where:

- V is a set of variables, where each $v \in V$ can take a value from a corresponding finite domain D_v. A partial assignment is a function f assigning values to a set $V' \subseteq V$, with $f[v] \in D(v)$ for each V'. A state s is a partial assignment where $V' = V$, and is said to satisfy a partial assignment f over variables V' *iff* $\forall v \in V'\, s[v] = f[v]$.
- s_0 is the initial state of the planning problem.
- $s_\star$ is the goal, a partial assignment to variables $G \in V$. For a state s to be a goal state, $\forall v \in G\, s[v] = s_\star[v]$.
- O is a set of operators, each $\langle pre, \mathit{eff} \rangle$. pre is a partial assignment denoting the operator's preconditions, and each $\langle v, post \in D(v) \rangle \in \mathit{eff}$ denotes an effect $v = d$.

An operator $\langle pre, \mathit{eff} \rangle$ can be applied to a state s *iff* s satisfies pre, and applying it yields a state $s' = s$, modulo for each effect $v = d$, $s'[v] = d$. For convenience, if an operator has a precondition $v = p$ and an effect $\langle v, q \rangle$ we refer to it having a pre_post condition $\langle v, p, q \rangle$.

- $cost$ is a cost function over operators. We assume each $cost(o) \in \mathbb{R}_0^+$. Setting all costs to 1 is equivalent to seeking a shortest plan.

Defined thus, a solution to a planning problem is a lowest-cost ordered sequence of operators $o \in O$ that when applied to s_0 produce a state satisfying $s_\star$.

We make use of two structures that can be derived from SAS+. First, the causal graph, CG (Helmert, 2006). This defines the relationships between the task variables, comprising a vertex $CG(v)$ for each $v \in V$, and an edge $CG(v') \rightarrow CG(v)$ iff v' is mentioned in a precondition or effect of an operator with an effect on v. Second, the domain transition graph for each variable, $DTG(v)$, denotes the transitions between its values. It contains a node for each of D_v, and an edge $n \rightarrow n'$ (labelled o) for each $o \in O$ with an effect $v = n'$ and either a precondition $v = n$, or no precondition on v (in which case the effect $v = n'$ can be attained irrespective of the prior value of v).

In this work we consider state-pruning for cost-optimal planning as forwards search. We build on two contrasting planners using this approach:

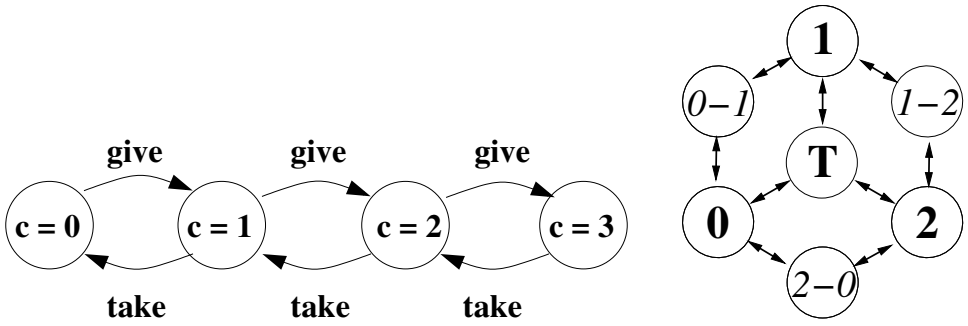
- 'Baseline' from the optimal track of IPC 2008, i.e. A* blind search — this exhibited the best performance in the competition;
- A* search using $h_{\text{LM-CUT}}$ (Helmert & Domshlak, 2009), a recent state-of-the-art heuristic for optimal planning, and one of the few shown to comprehensively outperform blind search.

In satisfycing planning, work on abstraction approaches (Haslum, 2007) and macro-actions (Botea, Müller, & Schaeffer, 2005) is related, as they can be seen as finding ways to transform a problem into one for which a solution is easier to find. However, in optimal planning, any transformations must preserve optimality, which these do not, and it is this challenge we focus on.

3 Pruning Irrelevant Actions

One well-established technique for reducing the number of ground actions to consider when planning is to perform a reachability analysis backwards from the goals: any action not reached during this has no effects relevant to the task goals, and can hence be discarded. The power of this approach, however, is limited in domains with undirected search spaces. Consider, for instance, a small problem from the Driverlog domain, with one truck, driver and package (variables t, d and p); and a single goal $p = g$, where g is some goal location. Regression from the goal will reach all load and unload actions:

- unloading the package at its goal location means the package was in the truck;
- thus, it was loaded into the truck at some location;
- thus, it could have been unloaded at that location first.



(a) Resource Map (b) DTG for a Driver

Figure 1: Domain Transition Graphs

Even options which are clearly suboptimal are kept: we know no optimal trajectory will return p to its initial location, nor will it move p out of its goal location once there. The actions may lie on a path to the goal, but with respect to optimal solutions, they are irrelevant. We can generalize this notion of pruning irrelevant actions, and prune actions as follows:

Definition 1 — Simple Irrelevant Action
An action is a simple irrelevant action *iff*:

1. It has a single effect $\langle v, \mathit{aft}\rangle$ and, optionally, a precondition $v = \mathit{bef}$;
2. The causal graph contains no edges out of $CG(v)$;
3. Either $s_0[v] = \mathit{aft}$, or $s_\star[v] = \mathit{bef}$.

This pruning relies on there being no dependencies between the affected variable $v \in V$ and any other, to ensure that there is no reason to retrace variable transitions within $DTG(v)$ for the benefit of other variables. Whilst sufficient in Driverlog, restricting the maximum number of packages each truck can carry at once will break the definition: each load/unload action also modifies the value of the relevant capacity variable. The intuition remains unaffected, however: there is no benefit of using capacity in its own right. We now present an approach to recognizing such cases.

3.1 Resource Analysis

A resource generic type (Long & Fox, 2000) captures the PDDL idiom of propositionally encoding a resource level using a stack of propositions; for instance, the number of free cells, or the remaining capacity of a truck. An abstraction of such a structure is presented in Figure 1a, and within a SAS+ encoding, resources correspond to variables whose DTGs are of the form presented, and of which certain conditions hold. Our concern here is resources that are taken (and, optionally, given back) in the process of changing the values of other state variables. We define resources as follows:

Definition 2 — Resource
A variable r is a resource *iff*: it has no value specified in the goal state; no operator has a precondition on r without also having an effect on r (i.e. observing the value of the resource but not changing it); and we can map the values from its domain D_r to an ordered sequence of levels $[l_r(0)..l_r(n)]$ where:

1. All 'take' actions have:

- a pre_post condition $\langle r, l_r(i+1), l_r(i)\rangle$ (i.e. the action decreases r from $l_r(i+1)$ to $l_r(i)$);
- an effect $\langle u, RUV(u,r)\rangle$, such that $RUV(u,r)$ is a common effect across all the 'take' actions on r that have an effect on u. u is then said to be a resource-using variable, and $RUV(u,r)$ is the value it adopts when using r.

2. Further, for every take action, for each $j \in [1..n]$ there is an action that is identical, modulo its pre_post condition on r being $\langle r, l_r(j), l_r(j-1)\rangle$.
3. All 'give' actions have at least two pre_post conditions:
   - one $\langle r, l_r(i), l_r(i+1)\rangle$ (i.e. increasing r);
   - another $\langle u, RUV(u,r), e\rangle$, where $e \neq RUV(u,r)$, for some resource-using variable u.
4. Further, for every give action, for each $j \in [0..(n-1)]$ there is an action that is identical, modulo its pre_post condition on r being $\langle r, l_r(j), l_r(j+1)\rangle$.
5. All actions with an effect on r can be characterized as gives or takes;
6. All actions with an effect on a resource-using variable u also have a precondition on u; all actions with an effect $\langle u, RUV(u,r)\rangle$ have another taking r; and all actions with a pre_post condition $\langle u, RUV(u,r), e\rangle$ where $e \neq RUV(u,r)$, also give r.
7. $(s_0[r] = l_r(n-i)) \Leftrightarrow (i = |\{u \in V \mid s_0[u] = RUV(u,r)\}|)$

The key consequence of this definition is that the level of a resource r is never important for its own ends. Intuitively, by insisting it cannot be used in a prevail condition or have a goal value, then the only case in which r can be referred to or changed is by actions with a take or give pre_post condition acting upon it. Then, from Point 2 we know that effectively equivalent take actions are available at any point where there is a non-zero value of r; and from Point 4 we get a similar guarantee on give actions.

We can now identify irrelevant resources as follows:

Definition 3 — Irrelevant Resource
A variable r is irrelevant *iff* it is a resource under Definition 2 and:

1. Its domain D_r contains $n+1$ values;
2. There are at most n resource-using variables that could use r.

Simply, if the capacity of the resource is at least as great as the number of variables that could want to use it, then in any state S where there is a resource-using variable v and $S[v] \neq RUV(v,r)$, there is always at least one unit of the resource still available, to be used by v if needs be.

Finally, we revisit the definition of an irrelevant action as follows:

Definition 4 — Irrelevant Action
An action A is an irrelevant action *iff*:

1. It has an effect $\langle v, \mathit{aft}\rangle$ and, optionally, a precondition $v = \mathit{bef}$;
2. Either $s_0[v] = \mathit{aft}$, or $s_\star[v] = \mathit{bef}$.

3. All edges out of $CG(v)$ are to resource variables, and hence any other effects of A are pre_post conditions on resources.

4. For each non-irrelevant resource r, if $s_*[v] = \mathit{bef}$ then $\mathit{bef} \neq RUV(v,r)$); or if $s_0[v] = \mathit{aft}$ then $\mathit{aft} \neq RUV(v,r)$.

---

4 Inferring Inevitable Actions

In this section we focus on identifying conditions under which certain actions can be applied automatically, without loss of optimality. In doing so, the intermediate states can be discarded: at any state S in which the conditions for automatically applying an operator o are satisfied, we can replace S with the state reached after applying o, and disregard the other successors. In doing so, we can both avoid their heuristic evaluation, and — when performing A* search — avoid having to record the intermediate states on the closed list.

4.1 Slip-Streaming Goal Actions

The first type of operators we consider are goal-achieving operators. The intuition behind our interest in these is as follows. Suppose we have a variable v, with no incoming edges into $CG(v)$ in the causal graph, and with goal value $s_*[v]$. If in a state s (where $s[v] \neq s_*[v]$) there is an applicable operator o with effect $\langle v, s_*[v]\rangle$, and no other operator o' with this effect carries a lower cost, then o must be applied. Simply, such an operator is side-effect free: it only changes v to its goal value, and from the causal graph we guarantee a specific value of v is never needed for an operator changing any other variable.

Generalizing this to include the cases where v interacts with resource variables gives us the following:

Definition 5 — Slip-Stream Action
An operator o is a Slip-Stream action in the state s *iff* the following conditions hold:

1. o is applicable in s, has an effect $\langle v, s_\star[v]\rangle$, and $s[v] \neq s_\star[v]$;

2. There does not exist an operator o' with $cost(o') < cost(o)$ and an effect $\langle v, s_\star[v]\rangle$.

3. Any edge out of $CG(v)$ is to $CG(r)$, where r is a resource variable (Definition 2);

4. $s_\star[v] \neq RUV(v,r)$ for any non-irrelevant resource r (Definition 3).

The provisi for resource variables still follow the intuition that o must be side-effect free, but allow operators whose side-effects that can only ever be beneficial: after o, we guarantee that v is not using any resource (other that ones which are known to be irrelevant).

4.2 Tunnel Macros

Tunnel Macros were first introduced as part of a Sokoban solver (Junghanns & Schaeffer, 2001). Consider a grid-based maze: here, 'tunnels' arise when a column of empty squares has a column of blocked squares on each side. Upon entering such a tunnel, assuming the points along it bear no significance, the only optimal choice is to carry on until the end:

walking backwards would introduce a cycle. As such, we can skip consideration of the immediate states.

To generalize this idea to planning, we identify analogs to tunnels within DTGs. Consider by way of example Figure 1b, the DTG for a Driverlog driver. In the benchmark Driverlog problems, there exist major locations (here designated 0,1,2) between which trucks can drive. Between these, are path locations $((0\text{-}1), (1\text{-}2), (2\text{-}0))$, through which drivers can walk. This results in the DTG shown: a driver can be at a major location, a path location, or in a truck (T).

Now consider a state s where $s[d] = 1$ for a driver d. If we apply the operator (`walk d 1 0-1`), we reach s' where $s'[d] =$ (0-1). $(d =$ (0-1)) is a precondition of only two actions: walking back to $d = 1$, which would be cyclical; or, walking to $d = 0$. Hence, the only reasonable option is the latter, and it becomes apparent that path locations are analogs to tunnels: once entered, the only reasonable option is to walk on. These principles can be generalized to SAS+ operators, subject to certain criteria:

Definition 6 — Tunnel Macros
In whichever state s we can and do apply an operator o, with preconditions pre and a single effect $\langle v, p \rangle$, a state s' is reached in which we know the following partial assignment holds:

$$s_{min} = \{(v, p)\} \cup \{(a, b) \in pre \mid a \neq v\}$$

We can then tunnel if:

1. $v = p$ is not a goal;
2. Any operator with a precondition $v = p$ has an effect on v;
3. For every operator o' with a pre_post condition $\langle v, p, p' \rangle$:
   - the preconditions of o' are satisfied by s_{min};
   - any other effect of o' is on an irrelevant resource (see Definition 3).

   We call the set of all such operators $tunnels$.

If $tunnels$, is empty the process terminates. Otherwise, each $o' \in tunnels$ can be applied to s', yielding a set of states, each s''. The tunnel macro procedure can then be applied recursively, to each s'', and s' can then be discarded.

In the Driverlog example, there is only one possible tunnel outcome: $o' =$ (`walk d 0-1 0`), leading to $s[d] = 0$. In general, applying o can lead to many options o'. As stated in the definition, we handle these cases by considering each $o' \in tunnels$ and applying tunnel-macro detection recursively, finding multiple macros stemming from o. Note that, because tunnel macro detection relies only on the partial assignment s_{min} guaranteed to hold after o, we can detect tunnel-macros for each operator prior to search. If there are tunnel macros for o, and o is applied at some state during search to yield s', we know s' satisfies s_{min}, hence the detected tunnel outcomes can be applied (leading to one or more replacements for s').

5 Symmetry Detection

Fox and Long (Fox & Long, 1999) introduced the notion of *functional symmetry* in PDDL planning. Here we contribute a mechanism for identifying symmetry in a SAS+ setting.

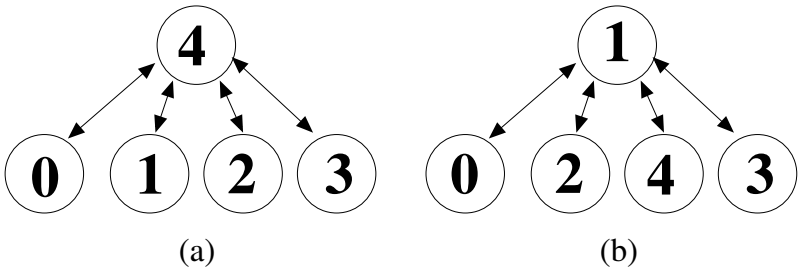


Figure 2: Package Domain Transition Graphs

According to Fox and Long, two entities are be defined to be functionally symmetrical as follows:

Definition 7 — PDDL Functional Symmetry
Two entities a, b are functionally symmetrical in a state s *iff*:

1. a and b are of the same type;
2. Neither a nor b is a constant appearing in the precondition or effects of any action schema;
3. a and b are placed in equivalent propositions in the current state. Taking a proposition to be defined by a tuple $\langle name, \{e_0..e_m\}\rangle$, where $e_0..e_m$ is a list of parameters:

$$\begin{aligned} &\{\langle name, \quad \{e_0..e_n, a, e_{n+2}..e_m\}\rangle \in s\} \\ \equiv &\{\langle name, \quad \{e_0..e_n, b, e_{n+2}..e_m\}\rangle \in s\} \end{aligned}$$

4. Similarly, a and b are placed in equivalent goal propositions.

With this definition one can build *symmetry groups* — groups of pairwise functionally symmetrical entities — and use these as a basis for pruning effectively equivalent action choices. For full details on how to do this in a PDDL setting, we refer the reader to (Fox & Long, 1999).

At the core of Definition 7 is the notion of entities and their function, inferred from types, facts and actions. In a SAS+ formalism, however, we have none of these, so we will consider symmetries between variables. A variable's DTG can be thought of characterizing the behavior of some component of the problem. For example, within a simple logistics problem, a variable denotes the status of each package: either at a location, or in a named truck. Its DTG has a characteristic topology: no transitions between 'at location' values; and no transitions between 'in truck' values.

Ignoring labels on edges for the moment, the problem of finding structure-preserving mappings between two DTGs is that of graph isomorphism (McKay, 1981). If an isomorphism can be found, a mapping is defined from the vertices in one DTG onto the vertices in another. Let us consider, by way of example, the two DTGs A and B in Figure 2 (two packages from a logistics problem with a single truck)[1]. Each vertex in $DTG(a)$ and $DTG(b)$ corresponds to a value in D_a or D_b respectively. If we denote their domain values as $[a_0..a_n]$ and $[b_0..b_n]$, an isomorphism is a structure-preserving mapping $map_{B,A}$

[1]Note here that in an automatic translation to SAS+, such as (Helmert, 2009), the numerical domain values (and hence DTG vertex labels) are arbitrary, and do not correspond to the names of the original locations.

from B to A, where $map_{B,A}(i)$ defines the vertex a_j that corresponds to $b_i \in B$. As the vertices correspond to variable values, the mapping can then be used to map values of b onto values of a, with $map_{B,A}(i)$ dictating the value of A symmetric to $b = i$.

Whilst graph isomorphism detection can narrow down the potential isomorphisms from the values B to A, considering topology alone is not sufficient: we must also compare operators relevant to B and/or A. For instance, in Figure 2, $map_{B,A}(1) = 4$, but it appears b_0 could map to any $a_0..a_3$: however, these correspond to named locations, and hence are not interchangeable. For a variable v, the relevant operators we must consider, $ro(v)$, are:

$$ro(v) = \{o \in O \mid \begin{array}{lll} (v = k) & \in pre(o) \\ \vee \quad \langle v, q \rangle & \in \mathit{eff}(o) \end{array} \}$$

With a candidate isomorphism, we can morph the preconditions and effects of each operator $o_B \in ro(B)$. For a given o_B, we define $map_{B,A}(o_B)$ as:

- replace each precondition $(b = k)$ with $(a = map_{B,A}(k))$;
- replace each effect $\langle b, q \rangle$ with $\langle a, map_{B,A}(q) \rangle$;

As an extension of this:

$$map_{B,A}(ro(B)) = \{map_{B,A}(o_B) \mid o_B \in ro(B)\}$$

Then, if $map_{B,A}(ro(B))$ is equivalent to $ro(A)$, we have shown that $map_{B,A}$ defines an isomorphism from variable B to A. For clarity, equivalence between operators and operator sets is defined as:

- two operators are considered equivalent if their costs, preconditions and effects are the same;
- two operator sets a, b are considered equivalent if for each $o \in a$, there is an equivalent operator in $o' \in b$, and vice-versa.

Finally, for the two to be considered symmetrical in search, either neither has a specified goal value, or $s_*[A] = map_{B,A}(s_*[B])$. Symmetry detection in this way will not identify as much symmetry as the approach taken by Fox & Long, as entities previously recognizable as symmetric may be compiled to more than one SAS+ variable. However, as we shall now discuss, it can be used in search in a manner that introduces minimal overheads. It is worth noting that graph isomorphism in general is non-trivial; we will discuss in our evaluation the trade-off between the costs of symmetry detection and planner performance.

5.1 Symmetry Breaking

Having now identified variables that are symmetrical, we will use the information to reduce the number of states visited during search. The technique we use is adding Symmetry Breaking constraints (Crawford, Ginsberg, Luks, & Roy, 1996) (SBCs). Where previously during search there would a range of symmetric choices available (in planning, operators to apply), the SBCs render only one option available, thus pruning the effectively equivalent alternatives.

Returning to the concept of a symmetry group, with the discussed notion of symmetry between SAS+ variables we can define a symmetry group:

Definition 8 — SAS+ Variable Symmetry Group
A SAS+ variable symmetry group M consists of an ordered list of at least two variables $[m_0..m_n]$, such that:

- For each variable $m_i \in [m_1..m_n]$, m_i is symmetric with m_0, through a map_{m_i,m_0}.

With the (arbitrary) order imposed on the list of members of M, we can now break symmetry, by adding constraints to allow only one of each effectively equivalent operator to remain applicable in each state. First, for each value $p \in D_{m_i}$ of each $m_i \in M$ we forbid the equivalent values of the variables earlier in the list M, denoted $SBV(m_i, p)$. These are the basis of the symmetry-breaking constraints relevant to using that value.

$$SBV(m_i, p) = \bigcup_{j \in [0..(i-1)]} \{(m_j \neq map^{-1}_{m_j,m_0}(\, map_{m_i,m_0}(p)\,))\}$$

Then, with these constraints on variables' values, we add constraints to operators, based on their preconditions. The symmetry-breaking constraints on an operator are defined as follows:

Definition 9 — Symmetry-Breaking Constraints
For an operator o, with preconditions pre, the symmetry breaking constraints $SBC(o)$ are:

$$SBC(o) = \{ \quad v \neq w \mid \nexists i \in D_v.(v = i) \in pre \\ \wedge \exists (p = q) \in pre \,.\, (v \neq w) \in SBV(p, q)\}$$

If a state s contradicts one of the conditions in $SBC(o)$, and o is applicable, then we know there is at least one other applicable action that: (i) is identical to o, modulo referring to the equivalent settings of variables earlier in the list of one or more symmetry groups' members; and (ii) would lead to an effectively equivalent successor state. Thus, for o to be applied in s, we insist that no such equivalent action is available, i.e.:

$$\forall (v \neq w) \in SBC(o).s[v] \neq w$$

6 Evaluation

We evaluate our techniques under A* cost-optimal search, with two contrasting heuristics, to give an insight into the benefits of the techniques in different settings. The first is blind search, exploring many nodes, but keeping per-node overheads to a minimum. The second is a powerful but expensive heuristic, $h_{\text{LM-CUT}}$. Each test was ran on a 3.4GHz Pentium IV machine and limited to 30 minutes and 1.5GB of memory. We evaluated on the 20 STRIPS domains taken from the last three International Planning Competitions. For pre-2008 domains, no action costs were specified, so they were each taken to be 1. Also, when evaluating $h_{\text{LM-CUT}}$, all costs were set to 1, due to the implementation available.

Our first observations, relevant to both sets of results, are the generality of our techniques. Of the 20 domains used, our analyses recognized some feature in 16, and were able to make use of that feature to improve performance in 12 of these (features identified in these 12 are shown in the final column of Table 1). (We also note that in many domains, more than one feature was identified and gave rise to performance enhancements.) This demonstrates our techniques have a good level of generality: the domains were a diverse

	Coverage					Time Scores				Node Scores				Features
Domain	baseline	all	-tun	-sym	-pru	all	-tun	-sym	-pru	all	-tun	-sym	-pru	Used
driverlog	7	**10**	9	10	7	0.02	0.07	0.02	0.16	0.02	0.06	0.02	0.15	IRGTS
rovers	5	**7**	7	7	5	0.04	0.04	0.04	1	0.04	0.04	0.04	1	–GT-
zeno	7	**8**	8	8	8	0.2	0.2	0.22	0.56	0.19	0.19	0.2	0.59	I-G-S
pathways	4	4	4	4	4	0.23	0.23	0.23	1	0.26	0.26	0.26	1	–G–
satellite	4	**5**	4	5	5	0.25	0.74	0.24	0.33	0.32	0.71	0.32	0.47	–GTS
mystery	**15**	14	14	15	14	0.38	0.38	0.4	0.81	0.35	0.35	0.38	0.78	IRG-S
pipestankage	7	**10**	10	7	10	0.57	0.58	1.01	0.57	0.6	0.6	1	0.6	-R–S
transport-strips	11	11	11	11	11	0.58	0.59	0.62	0.95	0.52	0.52	0.57	0.93	IRG-S
tpp	5	5	5	5	5	0.58	0.97	0.61	0.58	0.48	1	0.48	0.48	-R-T-
elevators-strips	9	**10**	10	10	10	0.68	0.68	0.68	1.01	0.61	0.61	0.61	1	IRG-S
psrsmall	48	**49**	49	49	48	0.9	0.9	0.9	1.01	0.85	0.85	0.85	1	IRG-S
openstacks-strips	17	17	17	17	17	0.91	0.91	1	0.91	0.91	0.91	1	0.91	—-S
sokoban-strips	16	16	16	16	16	0.99	1	0.99	0.99	0.96	1	0.96	0.96	-R-T-

Table 1: Results for adding features to the Baseline planner. To remove noise, result problems solved in < 1s are excluded (except in coverage). Node and Time scores are relative to Baseline. Features key: I: irrelevant actions, R: resource analysis, G: slip-streaming goal actions, T: tunnel macros, S: symmetry.

collection (all compatible domains from recent competitions), and were not specially selected for features. The ability to match features in over 3/4 of these domains, and to improve performance in over 1/2 of them, is a pleasing result in itself. We have only included results in the table for the 12 domains in which performance was improved. For any of the other 8, the data is uninteresting, other than showing our analyses never added more than 2% to the runtime of the planner, and coverage was always identical.

Considering what our analyses recognized, we shall highlight a few results. As one might expect, in domains with payloads (passengers or people), both irrelevant and slip-stream actions were found (e.g. packages should never be returned to initial locations; can be automatically unloaded at their goals; and then should never be subsequently loaded). In PSR and Pathways, an interesting case of slip-streaming actions arose: the domains were compiled from ADL to STRIPS, and the dummy actions introduced to mark that disjunctive goals had been achieved could be slip-streamed. Thus, some of the search overheads incurred in using the compiled domain are eliminated. Tunnel macros were found in five domains, including Sokoban, where the detection was facilitated through recognizing irrelevant resources: each square is a resource (available, or occupied with a stone/the player), but resources for squares only reachable by the player are irrelevant, and hence can be tunneled through. Symmetry detection found the intuitive symmetries between payloads with identical goals, as well as items appearing in identical orders in Openstacks, and the tanks at the storage locations in Pipestankage.

Let us now turn our attention to the performance results, considering first using the analyses with the baseline planner. Data for this is shown in Table 1, where we consider using all of our analyses (all), along with a configuration disabling each of the features, to allow assessment of which features provide benefits in which domains. We take the approach of subtracting each feature from the whole system, rather than adding each to the baseline individually, in order to allow for synergy between different techniques. All time and node scores are relative to the baseline, based only on problems that were

	Coverage					Time Scores				Node Scores				
Domain	$h^{\text{LM-CUT}}$	all	-tun	-sym	-pru	all	-tun	-sym	-pru	all	-tun	-sym	-pru	Features
driverlog	13	**14**	14	14	14	0.24	0.42	0.33	0.39	0.15	0.25	0.24	0.3	IRGTS
rovers	7	**9**	9	9	7	0.25	0.25	0.25	1.01	0.21	0.21	0.21	1	–GT-
zeno	12	12	12	12	12	0.52	0.51	0.57	0.86	0.5	0.5	0.57	0.85	I-G-S
pathways	5	5	5	5	5	0.52	0.51	0.52	1	0.48	0.48	0.48	1	–G–
satellite	8	**9**	8	9	9	0.4	0.97	0.4	0.42	0.38	0.93	0.38	0.42	–GTS
mystery	**17**	16	16	17	16	0.86	0.86	1	0.86	0.95	0.95	1	0.95	IRG-S
pipestankage	9	**10**	10	9	10	0.28	0.28	1	0.28	0.3	0.3	1	0.3	-R–S
transport-strips	12	12	12	12	12	0.82	0.82	0.86	0.95	0.8	0.8	0.84	0.94	IRG-S
tpp	6	6	6	6	6	1	1.01	1.01	1	1	1	1	1	-R-T-
elevators-strips	19	**20**	20	19	20	0.78	0.78	0.79	0.97	0.77	0.77	0.78	0.98	IRG-S
psrsmall	48	48	48	48	48	1.79	1.81	1.91	0.99	0.99	0.99	1	0.99	IRG-S
openstacks-strips	6	**7**	7	6	7	0.59	0.58	1	0.59	0.58	0.58	1	0.58	—-S
sokoban-strips	20	20	20	20	20	0.91	1	0.92	0.92	0.97	1	0.97	0.97	-R-T-

Table 2: Results for adding features to A*+$h^{\text{LM-CUT}}$. To remove noise, results on problems solved in $< 1s$ are excluded (except in coverage). Node and Time scores are relative to A*+$h^{\text{LM-CUT}}$. All results used $h^{\text{LM-CUT}}$ release 3304; differences between results here and those in (Helmert & Domshlak, 2009) are due to differences in the machines used, and non-determinism in the PDDL-to-SAS+ translator (NB we use the same SAS+ encoding of each problem for each configuration.)

mutually solved. For a configuration c, the time score shown for domain d is:

$$\sum_p time(c, d, p) / \sum_p time(baseline, d, p)$$

where $time(c, d, p)$ is the time taken by c to solve problem p in that domain (we exclude problems that either configuration solves in under a second to remove noise). As can be seen, coverage is improved in 7 out of the 12 domains; and otherwise unaffected, other than in the Mystery domain. Here, on one problem (otherwise solvable in 38 seconds), the time spent detecting symmetry exceeds 30 minutes as, unusually, the space of possible isomorphisms our symmetry analysis must consider is large. This occurred in no other problem instance we tested on, and could be circumvented by imposing a cut-off on symmetry detection time.

Sometimes the improvements in coverage are small, due to large increases in difficulty between problem instances. As such, we also consider time taken to solve problems and nodes evaluated. With reference to the former, where one or more of our analyses succeeds, performance is always improved beyond that of the baseline. Nodes evaluated is closely aligned with time taken, indicating that our approach is reducing solution time through avoiding the exploration of unnecessary parts of the search space.

When planning with a good heuristic rather than blind search, one might expect that the scope for enhancing performance is lower; after all, the heuristic is already attempting to reduce search space exploration by heavily dissuading search from exploring some branches. However, pleasingly, the results for A* with $h^{\text{LM-CUT}}$, shown in Table 2, still indicate performance improvements, confirming that the techniques do indeed remain helpful in this setting. Coverage is improved on a number of domains, although not quite so much as with the baseline; though, again, this is in part due to jumps in difficulty between larger problems. The time taken to solve mutually solved problems is again reduced, substantially in many cases. Considering nodes evaluated, as might be expected, the ratio

between the number of nodes expanded is more favorable in the case of the baseline planner, as $h_{\text{LM-CUT}}$ will already be avoiding some useless states. However, pleasingly, the number of nodes evaluated is still reduced (in places, dramatically), showing our analyses offer something complementary to the capabilities of current heuristics.

7 Conclusions

We have shown that pre-processing techniques based on analysis can improve the performance of cost-optimal planners, with minimal overheads. These techniques prune irrelevant branches of the search space or allow skipping the evaluation of states that are never in themselves interesting. Our techniques can be used in isolation or in combination with any heuristic, complementing its use. The techniques we have introduced by no means exhaust the possibilities available for SAS+ preprocessing, so the idea in general has even wider potential. We also note that the competition domains, used in our evaluation, are mostly written by planning experts who are aware of the workings of planning technology and can write models that minimize redundancy in the representation; in the future as the users of planners become less specialized, the user base is likely to be less aware of modeling problems in the best possible way for solution, and this sort of analysis will become even more important.

References

Bäckström, C., & Nebel, B. (1995). Complexity results for SAS+ planning. *Computational Intelligence*, *11*(4), 625–655.

Botea, A., Müller, M., & Schaeffer, J. (2005). Macro-FF: Improving AI planning with automatically learned macro-operators. *JAIR*, *24*, 581–621.

Crawford, J., Ginsberg, M., Luks, E., & Roy, A. (1996). Symmetry-breaking predicates for search problems. In *Proc. KR*, pp. 148–159.

Fox, M., & Long, D. (1999). The Detection and Exploitation of Symmetry in Planning Problems. In *Proc. IJCAI*, pp. 956–961.

Haslum, P. (2007). Reducing accidental complexity in planning problems. In *Proc. IJCAI*, pp. 1898–1903.

Helmert, M. (2006). The Fast Downward Planning System. *JAIR*, *26*, 191–246.

Helmert, M. (2009). Concise finite-domain representations for PDDL planning tasks.. *AIJ*, *173*(5-6), 503–535.

Helmert, M., & Domshlak, C. (2009). Landmarks, critical paths and abstractions: What's the difference anyway?. In *Proc. ICAPS*, pp. 162–169.

Helmert, M., & Röger, G. (2008). How good is almost perfect?. In *Proc. of AAAI*, pp. 944–949.

Junghanns, A., & Schaeffer, J. (2001). Sokoban: enhancing general single-agent search methods using domain knowledge. *AIJ*, *129*(1-2), 219–251.

Long, D., & Fox, M. (2000). Automatic Synthesis and Use of Generic Types in Planning. In *Proc. AIPS*, pp. 196–205.

McKay, B. D. (1981). Practical Graph Isomorphism. *Congressus Numerantium*, *30*, 45–87.