Upwards: The Role of Analysis in Cost-Optimal SAS+ Planning

Andrew Coles and Amanda Smith
Department of Computer and Information Sciences,
University of Strathclyde, Glasgow, G1 1XH, UK
email: firstname.lastname@cis.strath.ac.uk

Abstract

In this paper, we will describe the planner UPWARDS, competing in the sixth international planning competition. Our primary focus will be on the novel contributions of the planner: in particular, the application of symmetry breaking, mobile analysis and tunnel macros in sequential cost-optimal SAS+ planning. A brief outline of UPWARDS itself is then provided.

1 Introduction

When performing optimal planning using state-space search, one of the key barriers to scalability is the number of states explored to find a provably optimal solution. The conventional approach to ameliorating this problem is to focus on the construction of an admissible heuristic: one which never over-estimates the distance from a state to the goal. Used in conjunction with a standard search algorithm (A*, IDA*, ...), this forms the basis of a planner. However, as discussed in (Helmert & Röger 2008), using a heuristic is not a panacæa: even with near-perfect heuristics, the number of states explored by standard heuristic search algorithms is a hindrance to scalability. Simply, a heuristic value alone cannot encapsulate all the guidance needed to plan efficiently.

In this paper, we pursue taking steps beyond heuristics when performing cost-optimal SAS+ planning, proposing several algorithm-level techniques. First, we consider how to infer and exploit symmetries, moving beyond PDDL entity symmetry (Fox & Long 1999) to SAS+ variable symmetry. Our approach carries negligible overheads, per-node, during search. Second, we consider the role of mobile generic types (Long & Fox 2000), and how such analysis can be exploited but without sacrificing optimality guarantees. Finally, we generalise the concept of 'Tunnel Macros' for use in domain-indepent planning, an idea previously used in a bespoke Sokoban solver (Junghanns & Schæffer 1997). Our competition planner, UPWARDS, employs these techniques in a regression search setting, but with minor modifications they are all suitable for use in forward-chaining search. Further, by being defined for use with SAS+, they are of potential use alongside state-of-the-art heuristics for optimal planning (Helmert, Haslum, & Hoffmann 2007).

2 Definitions

For a full definition of SAS+ planning, we refer the reader to the literature (Helmert 2006b; Bäckström & Nebel 1995). Briefly, a we define a SAS+ cost-optimal planning problem as a tuple $\Pi = \langle V, O, s_0, s_\star, cost \rangle$ where: V is a set of variables, each with corresponding domains D_v; O is a set of operators, each with a set of $prevail$ conditions and pre_post conditions; s_0 is the initial state; $s_\star$ is the goal (variable–value pairs which must hold true in any goal state); and $cost$ is a function denoting the cost, $cost(o)$, of each operator $o \in O$. We assume costs are fixed, non-negative values.

The task, in regression planning, is to find a sequence of operators through which $s_\star$ can be regressed in order to reach the initial state. A state s can be regressed through an operator o subject to:

$$\forall (v_n = c) \in prevail(o)\ s'[n] \in \{c \cup undef\}$$
$$\forall (v_n = (p \rightarrow q)) \in pre\_post(o)\ s'[n] \in \{q \cup undef\}$$

If s is regressed through o, a state s' is reached, where:

$$s' = (s \setminus \{v_n = q \mid v_n = (p \rightarrow q) \in pre\_post(o)\}) \cup prevail(o) \cup \{v_n = p \mid v_n = (p \rightarrow q) \in pre\_post(o)\}$$

The problem, then, is to find a sequence of operators through which $s_\star$ can be regressed to lead to a state $s \subseteq s_0$, and reversing this sequence forms a solution plan.

3 Symmetry Breaking

Symmetry breaking is a powerful tool within combinatorial search; it preserves completeness and optimality, whilst pruning provably equivalent areas of the search space. In problems where symmetry can be discovered effectively, and at reasonable cost, symmetry breaking can lead to substantial improvements in performance.

3.1 Background

In (Fox & Long 1999), Fox and Long introduced the notion of *functional symmetry* in PDDL planning (Fox & Long 2003). In PDDL, the definition of a problem is split into a domain and a problem file: the former captures the structure of the problem; the latter the instance at hand. The domain specifies entity types, abstract action schemata, predicates and constant entities. The problem specifies named entities of each type, propositions denoting the initial state (following the predicates defined in the domain), and the goal propositions. Under these semantics, two entities can be defined to be functionally symmetrical as follows:

Definition 3.1 — PDDL Functional Symmetry

2 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 $\langle name, <\text{parameter list}>\rangle$:
   $\{\langle name, < e_{0..n} >, a, < e_{n+1..m} >\rangle \in s\}$
   $\equiv \{\langle name, < e_{0..n} >, b, < e_{n+1..m} >\rangle \in s\}$
4. Similarly, a and b are placed in equivalent propositions in the goal state.

With this definition one can build *symmetry groups*, groups of pairwise functionally symmetrical entities. Within such groups, labels on actions are essentially arbitrary—they are not usefully distinguishable. Hence, by characterising this notion of symmetry formally, we can use it as a basis for pruning effectively equivalent action choices. For full details, we refer the reader to (Fox & Long 1999), but to summarise: when expanding s, with applicable operators Ω, we reach successors, one for each $o \in \Omega$. Given the effects of operators upon entities, we can prune redundant members of Ω: those whose outcomes are effectively equivalent. For instance, if ten balls are symmetrical, we need only keep the operators in Ω corresponding to manipulating a single ball.

3.2 Symmetry in SAS+

Our first contribution is a symmetry detection mechanism for SAS+. At the core of the symmetry definitions discussed so far is the notion of entities, and their presence in propositions. In SAS+, we do not have these: we have only variables, and their values, and it is therefore non-trivial to see how existing symmetry can be exploited in SAS+. Key to the definition of functional symmetry in PDDL is a notion of function, inferred from types, facts and actions. We shall begin by seeking analogues for each of these.

First, let us consider types. A PDDL type defines, at an abstract level, the capabilities of an entity. The analogue to this, within SAS+ is the Domain Transition Graph (DTG) (Helmert 2006b). A DTG is a directed graph, denoting the paths between the values a variable can hold. For each variable v, the domain transition graph $DTG(v)$ contains a vertex for each possible value of v (i.e. each of $D(v)$, and additionally one for $undef$). Edges are defined according to O: for each operator o with a pre_post condition $\langle v, pre, post$, $DTG(v)$ contains an edge $pre \rightarrow post$, labelled with o. Thus, the topology of a variable's DTG can be thought of characterising its type. 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, the problem of determining whether two DTGs are symmetrical 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 denoted in Figure 3.2 (two packages, in a problem with a single truck). If we denote their vertices $a_0..a_n$ and $b_0..b_n$, a mapping from B onto A, $mapping_{B\rightarrow A}(b_i)$, defines the vertex in A that corresponds to each b_i. The function is one-to-one: each vertex of B maps onto a single vertex in A, and vice-versa. For the example shown, the mapping will consist of $b_1 \leftrightarrow a_4$, and a one-to-one mapping for the other vertices. As the vertices correspond to variable values, the mapping can then be used to map values of b onto values of a, with $mapping_{B\rightarrow A}(b_i)$ dictating the value of a symmetric to $b = i$.[1]

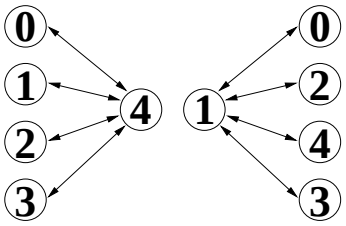

Figure 1: Package Domain Transition Graphs

In this manner, graph isomorphism appears to be an effective tool for detecting variable symmetry. However, as stated, this is only the case if labels on edges are ignored. Clearly, this is not feasible: the labels denote operators, with $prevail$ conditions and pre_post involving it and other variables. Intuitively, returning to Figure 3.2, the edges correspond to loading/unloading packages from the truck at various locations. These will have a specific $prevail$ condition that the truck must be at the appropriate location. Hence, although the DTGs may suggest we can permute some vertices arbitrarily when constructing a mapping, we cannot: we must consider the implications of the mapping decisions made. Further, we must also consider operators outwith the DTGs: the DTG for a variable v contains only operators which modify the value of v; not those in which v is present within a $prevail$ condition. To capture these, along with the variables within the DTG, for each variable v we can define its relevant operators $ro(v)$ as:

$$ro(v) = \{o \in O \mid \exists\langle v, k\rangle \in prevail(o) \vee \exists\langle v, p, q\rangle \in pre\_post(o)\}$$

With these, we can then perform augmented graph isomorphism detection: both the topology and the operator implications of the mapping must hold. We can apply a mapping $mapping_{B\rightarrow A}(b_i)$ to an operator by modifying its $prevail$ and pre_post conditions accordingly:

- replace a $prevail$ $\langle b, k\rangle$ with $\langle a, mapping_{B\rightarrow A}(b_k)\rangle$;
- replace a pre_post $\langle b, p, q\rangle$ with $\langle a, mapping_{B\rightarrow A}(b_p)$, $mapping_{B\rightarrow A}(b_q)\rangle$;

By modifying $ro(b)$ according to these rules, if the resulting operator set is equivalent to $ro(a)$ under the mapping found, then the two variables are symmetrical: a mapping has been found which reflects the DTG topology *and* which assuages any interactions with other variables. Importantly,

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

this augmented isomorphism detection on domain variables can be performed as a static pre-processing step, with minimal overheads incurred during search: it records *potential* symmetries, and whether or not variables are actually symmetric can be determined with minimal cost at each state.

3.3 Use of Symmetry in Regression Search

Having identified potential variable symmetries, we can exploit this information during regression search. Here, as we are aiming to find a path back from a state s to the initial state s_0 we must first split potentially symmetric variables based on their values in s_0. If two variables a, b are potentially symmetric, but $s_0[a]$ does not equal $mapping_{B\rightarrow A}(s_0[b])$, then in each state when regressing towards s_0 we cannot consider a and b to be symmetrical.

What remains is to exploit symmetry at each state s. Here, we use the mapping: for potentially symmetric variables a, b if $s[a] = mapping_{B\rightarrow A}(s[b])$ then the variables are currently symmetrical. This can be ascertained with minimal overheads. During state expansion, the recognised symmetries can then be used to reduce the branching factor. Referring to Section 2, a state s is expanded by finding operators to achieve the variable values in that state. If we have a pair of symmetric variables a, b, we only need consider operators to achieve the value of one of them.

4 Generic Types: Mobiles

Generic types, introduced in TIM (Long & Fox 2000), encapsulate commonly occurring idioms within planning problems. Perhaps one of the most common of these is the mobile type. A mobile object can move (unhindered) over a map of locations, subject to certain conditions holding. For instance, in Driverlog, if a driver is in a truck, then the truck can move freely over its location map. In HybridSTAN (Fox & Long 2001), this was used to safely abstract move operators: any reference to these were removed in a pre-processing step, and once a solution had been found to this reduced problem, the missing operators were inserted.

In the IPC5 version of Downward (Helmert 2006a), the first steps were taken towards incorporating such ideas into a SAS+ setting, using a technique known as *safe abstraction*. There is a strong correlation between SAS+ variables and mobile generic types: a mobile generic type can be represented by a single SAS+ variable, with its map being reflected in the variable's DTG. In Downward, two criteria determine whether a variable v can be safely abstracted, and as in HybridSTAN if these are met the variable is abstracted out of the problem and the missing operators inserted post-hoc:

Definition 4.1 — Safe Abstraction
A variable v can be safely abstracted if:
- $DTG(v)$ can be wholly traversed by applying operators with no $prevail$ conditions and whose pre_post conditions affect only v;
- the DTG is strongly connected: the vertex $s_0[v]$ is reachable from all vertices in the DTG.

The limitation of safe-abstraction is that unlike TIM, it cannot handle $prevail$ constraints on the movement operators. Also, as in HybridSTAN, the post-hoc insertion of missing operators into a solution plan loses optimality. In UPWARDS we address these two issues by introducing a new technique corresponding to the TIM mobile type — *conditional abstraction* — and incorporate the abstraction information into search to preserve optimality, rather than using it only in post-processing.

We define conditional abstraction as follows:

Definition 4.2 — Conditional Abstraction
A variable v can be conditionally abstracted, subject to the conditions $v' = k$, if considering only the operators in $DTG(v)$ with $prevail$ conditions $\langle v', k\rangle$ yields a DTG $DTG_{\langle v',k\rangle}(v)$ which meets the criteria for Safe Abstraction (Definition 4.1).

A variable can have a number of conditional abstractions if there are several $prevail$ conditional paths. Indeed, it can have a conditional abstraction whose condition is null; i.e. safe abstraction is a special case of conditional abstraction. A variable v is wholly conditionally abstracted if the conditional DTGs for v, between them, all of the operators affecting v. In this case, we can eliminate choices of the operators changing v from search. We proceed in three phases:

1) Operator Path Preprocessing. For each conditional abstraction of v we use the Floyd-Warshall all-pairs shortest path algorithm to give cost-optimal operator paths between values of v. This gives us operator paths, and costs, to attain values of v subject to a condition holding.

2) Implicit Operator Selection. During search, operators with pre_post conditions involving v are never considered explicitly. Once we have a state s in which an abstraction condition $v' = k$ holds, we assume v can hold any value; not just its current value, $s[v]$. When generating a successor s', through applying an operator o, if a value of v other than $s[v]$ is needed (as a $prevail$ or as a $post$ condition), the necessary operators are determined according to the Floyd-Warshall lookup table. The resulting plan segment from s' to s then consists of o followed by this operator sequence.

3) Optimal Goal Jumping. We may reach a state s where the operators from $s_0[v]$ to $s[v]$ are implicit for one or more variables v. In this case, we apply the necessary operators to s, leading to a state $s' \subseteq s_0$ and hence a solution plan. If we did not consider this case, we would lose completeness: for variables that are wholly abstracted with only a null condition, we would never otherwise consider adding their final operators to reach s_0. To preserve optimality, we make two further considerations. First, if there are several conditional abstractions over a variable v, we take the path with the lowest cost. Second, if there is a unsatisfied conditional abstraction, and which potentially could lead to a lower-cost goal jump, we consider the normal successors to s also rather than only a goal jump; thereby preserving the opportunity to achieve this other conditional path.

With this technique, given we are performing cost-optimal sequential planning, it is important to note the cost of implicit operator sequences. Given we have cost-optimal paths from the Floyd-Warshall algorithm, we can rely on the optimality of the implicit operator sequences. Then, the cost of regressing through o in s becomes $cost(o)$, plus the cost of the implicit operator sequences.

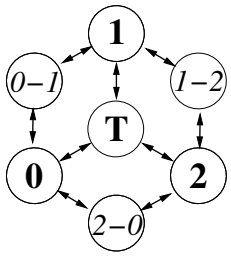

Figure 2: DTG for a Driver

5 Tunnel Macros

Tunnel Macros were first introduced as part of a Sokoban solver (Junghanns & Schæffer 1997). Consider a grid-based maze: here, 'tunnels' arise when a tract of empty squares has blocked squares on each side. Upon entering such a tunnel, assuming the points along it bear no significance, the only sensible choice is to walk to the end. Retreating necessarily results in a cycle, returning to from whence one came. The same is true for each point along the tunnel: the only sensible option is to carry on until the end, skipping heuristic evaluation or decision making. Tunnel Macros capture this concept: entering a tunnel results in being at the other end.

Our next contribution is to demonstrate how this idea can be generalised for use within planning. To achieve this we identify analogues to tunnels within the variable DTGs. Consider by way of example Figure 2, the DTG for a Driverlog driver. Drivers have the capability to be in a truck or at a location. The construction of Driverlog problems, following the format of IPC3, produces problems with major locations (here designated 0,1,2) between which trucks can drive. Between these, path locations are added ((0-1), (1-2), (2-0)), through which drivers can walk to reach adjacent major locations. In SAS+, 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 in search, s, in which $s[d] = 1$. If s is regressed through the operator `walk d 0-1 1`, we reach a state s' where $s'[d]$ =(0-1). This new value is entirely uninteresting: no other variable depends upon it. Returning to 0 would be cyclical, so the only reasonable option is to immediately regress through `walk d 0 0-1`. Hence, path locations are analogues to tunnels. These principles can be generalised to any SAS+ variable, subject to certain criteria:

Definition 5.1 — Tunnel Macro
If a state s is regressed through an operator o, with prevail conditions p and a single pre_post condition $\langle v, pre, s[v]\rangle$, a state s' is reached where $s'[v] = pre$. If the following conditions hold:

1. No operator has a prevail condition $\langle v, pre\rangle$;
2. All operators with pre_post conditions of the form $\langle v, k, pre\rangle$ have no other pre_post conditions; and, further, all have prevail conditions $c \subseteq p$. We denote these operators $tunnels$.

We can regress s' through each of the operators $t \in tunnels$, and then discard s'. This leads to states s'', one for each t. This process can then be applied recursively, to each s'', until the criteria no longer hold and the process terminates.

In the Driverlog example, only a single state s'' was reached, that with $s[d] = 0$, and recursion terminated here: $\langle d, 0, t\rangle$ is a pre_post condition of `board-truck t d 0`. In general, the DTG tunnels may fork leading to a number of possible tunnelled outcomes. By applying recursion and considering each operator in the set $tunnels$ we handle these cases, leading to many different eventual states s''.

6 Upwards

These techniques are implemented in our planner UPWARDS. UPWARDS works in two phases. First, FF (Hoffmann & Nebel 2001) is used to find a starting plan. This is used to determine an upper-bound ub on plan cost: we know the the cost of a solution plan is at most that of the plan found by FF. Then, sequential cost-optimal regression is performed, using the depth-first branch-and-bound (DFBB). The upper-bound for DFBB is first set to ub, and as search proceeds, the bound ub is tightened. Two heuristics are used: for pruning, an admissible heuristic based upon (Haslum, Bonet, & Geffner 2005); for branch ordering, FF's relaxed planning graph (RPG) heuristic. Once search has completed, we return the best plan found: either that of FF, or if quality was improved beyond ub, that found by UPWARDS itself.

Acknowledgements

We would like to thank Maria Fox, Derek Long and Peter Gregory for their helpful discussions concerning this work; and Malte Helmert and Sylvia Richter for making the source code for Fast Downward available.

References

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

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

Fox, M., and Long, D. 2001. Hybrid STAN: Identifying and Managing Combinatorial Optimisation Sub-problems in Planning. In *Proc. IJCAI*, 445–452.

Fox, M., and Long, D. 2003. PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains. *Journal of Artificial Intelligence Research* 20:61–124.

Haslum, P.; Bonet, B.; and Geffner, H. 2005. New Admissible Heuristics for Domain-Independent Planning. In *Proc. AAAI*, 1343–1348.

Helmert, M., and Röger, G. 2008. How Good is Almost Perfect? In *Proc. AAAI*.

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

Helmert, M. 2006a. Fast (Diagonally) Downward. In *IPC5 Competition Booklet, ICAPS 2006*, 37–38.

Helmert, M. 2006b. The Fast Downward Planning System. *Journal of Artificial Intelligence Research* 26:191–246.

Hoffmann, J., and Nebel, B. 2001. The FF Planning System: Fast Plan Generation Through Heuristic Search. *Journal of Artificial Intelligence Research* 14:253–302.

Junghanns, A., and Schæffer, J. 1997. Sokoban: A challenging single-agent search problem. In *Proc. IJCAI Workshop on Computer Games*, 27–36.

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

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