Symmetry-breaking in Planning: Schematic Constraints

Ian Miguel

Artificial Intelligence Group
Department of Computer Science
University of York
York, England
ianm@cs.york.ac.uk

Abstract. This paper considers the twin problems of automatically detecting and exploiting symmetry in typed STRIPS planning problems. Symmetry is detected by comparing the initial conditions of uniformly typed plan objects. During search, a schematic constraint representing a class of symmetric nogoods is constructed from each nogood learnt and the stored symmetry information. Algorithms to perform both of these steps are described and initial experimental results are given.

1 Introduction

Many recent advances in AI planning have centred upon the use of constraint satisfaction techniques. One example [4] views the planning graph produced by the popular Graphplan [11] system as an activity-based Dynamic CSP [7], while a second [6] views plan synthesis as the solution of a hierarchy of restriction/relaxation-based dynamic CSPs [2]. Common to both of these approaches is the need to learn nogoods in order to prune the (often large) search trees. Since symmetry is often inherent in planning problems, a planner can waste a lot of time exploring symmetric subtrees and recording symmetric nogoods. We propose a method of automatically detecting symmetries in typed STRIPS planning problems [3] and efficiently pruning the search tree with schematic constraints, each of which represents a symmetric class of nogoods.

2 The Gripper Problem

Figure 1 presents a planning problem known as the Gripper problem [5]. The goal of this problem is to transport all the balls from room A to room B. To accomplish this, the robot is allowed to move from one room to the other, pick up, and put down a ball. Each gripper can hold one ball at a time.

A solution to this problem is as follows:

1. - Pick up ball1
2. - Pick up ball2
2. Move to Room B
3.  - Drop ball
    - Drop ball2
4. Move to Room A
5. etc...

A suite of gripper problems exist, of which that shown in figure 1 is the most simple, with an increasing number of balls. The input for the problem with four balls is presented in table 1.

<table>
<thead>
<tr>
<th>Plan Objects</th>
<th>Initial Conditions</th>
<th>Goal Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>(room1 ROOM)</td>
<td>(at room1)</td>
<td>(at ball1 room1)</td>
</tr>
<tr>
<td>(room2 ROOM)</td>
<td>(free left)</td>
<td>(at ball2 room1)</td>
</tr>
<tr>
<td>(ball1 BALL)</td>
<td>(free right)</td>
<td>(at ball3 room1)</td>
</tr>
<tr>
<td>(ball2 BALL)</td>
<td>(at ball1 room2)</td>
<td>(at ball4 room1)</td>
</tr>
<tr>
<td>(ball3 BALL)</td>
<td>(at ball3 room2)</td>
<td></td>
</tr>
<tr>
<td>(ball4 BALL)</td>
<td>(at ball4 room2)</td>
<td></td>
</tr>
<tr>
<td>(left GRIPPER)</td>
<td>(at ball1 room2)</td>
<td></td>
</tr>
<tr>
<td>(right GRIPPER)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1. Input for the Gripper Problem with Four Balls
3 Symmetry Detection

The input to a typed STRIPS planning problem is:
- A list of plan objects, with types.
- A list of propositions that capture the initial state.
- A list of propositions that capture the goal state.

By considering the nature of the nogoods a planner typically records, it is only necessary to compare the initial conditions of plan objects with the same type to detect symmetry. For example, the following nogood may be discovered during search:
- At Step 3: \{at ball1, RoomA, at ball2, RoomB, at ball3, RoomB\}

This nogood expresses the fact that these three propositions are **conjunctively unsupportable**, i.e., that they cannot be made true simultaneously at time-step 3 from the initial conditions using the available operators. Once the nogood is recorded, it is used during the rest of the search by matching against the set of propositions that must be supported at the current time-step. If a match is made, then the search backtracks (it is known that these propositions are conjunctively unsupportable). Since nogoods record what cannot be achieved forwards in time from the initial conditions, the goal conditions are irrelevant in the detection of symmetry for this purpose.

Symmetry detection begins by dividing the plan objects into sets of a uniform type, \{U_1, U_2, \ldots \}. Comparisons for symmetry are subsequently made only within each \( U_i \) by the procedure `compareUniformTypes()`, below. For each plan object, \( o_a \in U_i \), a list is created of the initial propositions, \( P_a \), involving it (line 5). Comparisons are then made between elements of the uniform type set (line 6) and symmetries recorded in the `symmetricalWith` list associated with each plan object (lines 7-8). Symmetry detection could be made more efficient by using the fact that symmetry is transitive, but it already executes very quickly taking less than one second on the largest of the gripper suite, with 26 plan objects and 47 propositions.

1. **Procedure compareUniformTypes()**
2. **Foreach** \( U_i \)
3.  **For** \( a \leftarrow 0 \) **To** \( |U_i| - 2 \)
4.  **For** \( b \leftarrow a+1 \) **To** \( |U_i| - 1 \)
5.  \( P_a \leftarrow \text{propositions containing } o_a \in U_i, P_b \leftarrow \text{propositions containing } o_b \in U_i \)
6.  **If** `symmetrical(\( o_a \in U_i, o_b \in U_i, P_a, P_b \))`
7.  \( o_a\text{.symmetricalWith} \leftarrow o_a\text{.symmetricalWith} \cup \{o_b\} \)
8.  \( o_b\text{.symmetricalWith} \leftarrow o_b\text{.symmetricalWith} \cup \{o_a\} \)

Given two uniformly typed plan objects, \( o_a \) and \( o_b \), involved in sets of propositions \( P_a \) and \( P_b \) respectively, if \( P_a \) can be transformed into \( P_b \) by mapping occurrences of \( o_a \) onto \( o_b \) and vice versa, then \( o_a \) and \( o_b \) are symmetrical. This test is performed by the `symmetrical()` procedure below. In the following, we assume that the proposition lists are sorted, to avoid searching for a match.
1. Procedure symmetrical($o_a$, $o_b$, $P_a$, $P_b$)
2. If $|P_a| \neq |P_b|$ Return False
3. For $i \leftarrow 0$ To $|P_a| - 1$
4. $p_a \leftarrow$ $i$th element of $P_a$, $p_b \leftarrow$ $i$th element of $P_b$
5. If $p_a$.predSymbol $\neq p_b$.predSymbol Return False
6. If $p_a$.arity $\neq p_b$.arity Return False
7. $obj_a \leftarrow$ plan objects related by $p_a$, $obj_b \leftarrow$ plan objects related by $p_b$
8. For $j \leftarrow 0$ To $|obj_a| - 1$
9. $obj_{a, j} \leftarrow j$th element of $obj_a$, $obj_{b, j} \leftarrow j$th element of $obj_b$
10. If $(obj_{a, j} = o_a \land obj_{b, j} \neq o_b) \lor (obj_{b, j} = o_b \land obj_{a, j} \neq o_a)$ Return False
11. If $obj_{a, j} \neq o_a \land obj_{b, j} \neq o_b \land obj_{a, j} \neq obj_{b, j}$ Return False
12. Return True

The first test is whether $|P_a|$ and $|P_b|$, the lengths of the lists of propositions involving plan objects $o_a$ and $o_b$, respectively, are equal (line 2). If not, the mapping must fail. Otherwise, the (sorted) proposition lists are compared pairwise (lines 3–11). Given two propositions, $p_a \in P_a$, $p_b \in P_b$, we check for equality of predicate symbols and arity (lines 5–6). If they match, we compare the individual plan objects related by $p_a$ and $p_b$ (7–11). For each pair of plan objects, the test is whether an occurrence of $o_1$ is matched by an occurrence of $o_2$ and that all other objects related by the propositions are identical (lines 10–11).

Consider the comparison of $ball_1$ and $ball_2$ in the Gripper problem given in Table 1. They are of the same type and each is involved in just one proposition in the initial conditions, namely $(at\ ball_1\ room_a)$ and $(at\ ball_2\ room_a)$. The predicate symbol and arity of both propositions match, so we compare the individual plan objects that they relate. In the first position, an occurrence of $ball_1$ is matched by an occurrence of $ball_2$, so the test succeeds. The second position is occupied by the plan object $room_a$ in both cases. Hence, $ball_1$ and $ball_2$ are symmetrical.

Using the compareUniformTypes() and symmetrical() procedures, all the balls can be shown to be symmetrical in the Gripper problem. The same is true for the grippers themselves. The two rooms are not symmetrical, however, since Robby and the balls are in room A initially.

4 Schematic Constraints

Reconsider the example nogood given in section 3. For this simple problem, there are 4 such symmetrical nogoods at time-step 3, each of which may have to be rechecked before a fruitless branch of search is pruned. Generally, this leads to a large number of symmetrical nogoods being recorded.

However, since we know that all balls are symmetrical, we can use a nogood to create a schematic constraint which fulfils the function of this one nogood and all its symmetric equivalents. A schematic constraint is created by generating a template from the input nogood. For example, the above case is a nogood containing 3 elements, each of which is an ‘at’ proposition. For each proposition, instead of recording the exact nature of the input nogood, each plan object, $o_a$, is replaced by a list containing $o_a$ and all of the plan objects it is symmetrical with.
Note that we cannot simply use the object type, because it is not guaranteed generally that all objects with a particular type are symmetrical. The schematic constraint for the example nogood is as follows:

- Schematic Constraint, Time-step 3:
  - (Proposition 1: (at \{ball_1, ball_2, ball_3, ball_4\}, \{room_B\})
  - (Proposition 2: (at \{ball_1, ball_2, ball_3, ball_4\}, \{room_B\})
  - (Proposition 3: (at \{ball_1, ball_2, ball_3, ball_4\}, \{room_B\})

This captures the fact that it is impossible for any three of the four balls to be in Room B after just 3 steps. During search, when considering whether the set of propositions at a particular time-step is supportable, schematic constraints are used to generate all symmetrical nogoods. If any match, this branch of the search can be pruned immediately. Consider the following set of propositions at time-step 3 which the planner must try and find support for:

- (at \ball_1 \room_B)
- (at \ball_2 \room_B)
- (at \ball_3 \room_B)
- (at \ball_4 \room_B)

Rather than using generate-and-test to do this, a simple backtracking search is used. The procedure getSymNgs() below checks all schematic constraints recorded for this time-step against the set, \(P_n\), of propositions (in this case the four propositions above) that must be supported. Rather than using generate-and-test to do this, a simple backtracking search is used. A cheap, but powerful, pre-check is whether all the predicate symbols present in the schematic constraint are present in \(P_n\) (line 3). If not, then there is no way that this schematic constraint can ever match a subset of \(P_n\). In the example, only “at” propositions are present, matching our schematic constraint.

1. Procedure getSymNgs(\(P_n\))
2. For each schematic constraint, \(c_o\), at this timestep
3.  If all predicate symbols in \(c_o\) are present in \(P_n\)
4.    result \leftarrow \text{genSymNgs}(0, \phi, c_o, P_n)
5.  If result \neq \phi Return result

The procedure \text{genSymNgs()} below recursively through the elements of the schematic constraint, \(c_o\), using \text{genSymNLogEle}() to generate corresponding proposition elements of a nogood (line 6). Our example constraint comprises three such elements, as shown above. When this process is complete, if the generated nogood is a subset of \(P_n\), then \(P_n\) has been shown to be unsupportable.

1. Procedure genSymNgs(i, partialNg, c_o, P_n)
2. If \(i = |\phi| - 1\)
3.  If partialNg \subseteq P_n
4.    Return partialNg
5.  Else Return \phi
6. Else Return genSymNLogEle(i, 0, partialNg, c_o, P_n)
Finally, the procedure genSymm[Elem](i, j, partialNg, cS, Pm) takes a single element of cS, and, using the lists of symmetrical plan objects generates a single proposition element of a nogood (lines 6-10). For example, we might take:

- Proposition 1: (at \{ball1, ball2, ball3, ball4\}, \{roomB\})

and generate (at ball1 roomB). This choice is backtrackable, so that all possible elements are generated as necessary to find an entire nogood that matches Pm.

Note also that, rather than waiting for the whole nogood to be generated, each proposition is matched against Pm so that we can backtrack early (line 3).

1. Procedure genSymm[Elem](i, j, partialNg, cS, Pm)
2. If j = \|cS\| - 1
3. If partialNg \notin Pm Return \phi
4. Else Return genSymm[Elem](i + 1, partialNg, cS, Pm)
5. Else
6. symDomain ← symmetrical objects at slot j of ith element of cS
7. For k ← 0 To \|symDomain\| - 1
8. partialNg[k] ← symDomain[k]
9. result ← genSymm[Elem](i, j + 1, partialNg, cS, Pm)
10. If result \neq \phi Return result
11. Return \phi

4.1 Symmetrical Propositions

Unfortunately, this scheme is inadequate in two ways. Firstly, when a nogood is first discovered, it may contain multiple occurrences of the same object. Similarly, it may contain instances of plan objects which are mutually-symmetrical, but which must be treated as distinct for the purpose of this nogood: the example nogood contains three symmetrical but distinct balls. In order to remain sound, the schematic constraint must ensure that the slots corresponding to these occurrences are equal or different as appropriate for each generated symmetrical nogood. Secondly, it suffers from symmetry. This is because the propositions that comprise the nogood may be symmetrical, as is the case for the example.

In order to deal with both of these cases, sub-constraints over the slots for plan objects are added to each schematic constraint. For the first case, slots that were equal in the original recorded nogood are constrained to be equal. In addition, symmetrical slots which were different in the original nogood are constrained to be so for each symmetrical nogood. For the second case, symmetry-breaking constraints are added between slots of symmetrical propositions. These take the form of lexicographic constraints, based on the names of the symmetrical plan objects, e.g. ball1, ball2, etc.

Figure 2 presents the final schematic constraint for the example nogood. The ordering sub-constraints between the ball slots remove the symmetry between the propositions. Note that '>1' and not '>2' is used since it was already known that these slots were different. The sub-constraints are checked in procedure genSymm[Elem](i, j, partialNg, cS, Pm) at line 8 to ensure that the assignment of a plan object to a slot is consistent.
5 Results

Table 2 presents the results of solving four problems from the Gripper suite. The planner used is GP-γDCSP, a development of Graphplan presented in [6]. Three versions of the planner are used, one with no symmetry-breaking, a second that uses symmetry information to record all symmetric equivalents of each nogood found, and a third that uses schematic constraints. The hardware platform is a 750 Mhz PentiumIII with 256Mb.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Number of Balls</th>
<th>Plan Length</th>
<th>No Symmetry Breaking</th>
<th>Record all Nogoods</th>
<th>Schematic Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>17</td>
<td>0.5s</td>
<td>3.2s</td>
<td>0.2s</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>11</td>
<td>5.8s</td>
<td>2.5s</td>
<td>4.1s</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>15</td>
<td>323.2s</td>
<td>31.8s</td>
<td>25.9s</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
<td>19</td>
<td>&gt; 1000</td>
<td>Out of Memory</td>
<td>289.3</td>
</tr>
</tbody>
</table>

*Table 2. Results for the GP-γDCSP Planner on Four Gripper Problems.*

The use of symmetry-breaking is clearly beneficial to this problem, with an order of magnitude improvement in evidence for the larger instances. On the small instances, simply recording all symmetrical equivalents of a nogood suffices, and out-performs schematic constraints which incur some overhead. Since, however, there are an exponential number of nogoods to record this scheme becomes inefficient very quickly as the problem size grows. The use of schematic constraints therefore provides an effective remedy for the symmetry in these problems, without the crippling space cost.

6 Conclusions

We have discussed symmetry detection and exploitation in simple typed STRIPS planning problems through the use of schematic constraints. The Gripper prob-
lem considered here has a much higher degree of symmetry than the average, but many other planning problems contain some symmetry which could be exploited using schematic constraints. Although typed STRIPS problems are simple by today’s standards, we expect that the techniques presented here could be extended to more complex planners and domains.

Acknowledgements

The author is supported by EPSRC Grant GR/N16129. I thank the anonymous referees for their useful comments and Angela Miguel for the diagram of Robby.

References