Time-Saving Tips for Problem Solving with Incomplete Information

Michael R. Genesereth
Computer Science Department
Stanford University
Stanford CA 94305

Illah R. Nourbakhsh
Computer Science Department
Stanford University
Stanford CA 94305

Abstract

Problem solving with incomplete information is usually very costly, since multiple alternatives must be taken into account in the planning process. In this paper, we present some pruning rules that lead to substantial cost savings. The rules are all based on the simple idea that, if goal achievement is the sole criterion for performance, a planner need not consider one "branch" in its search space when there is another "branch" characterized by equal or greater information. The idea is worked out for the cases of sequential planning, conditional planning, and interleaved planning and execution. The rules are of special value in this last case, as they provide a way for the problem solver to terminate its search without planning all the way to the goal and yet be assured that no important alternatives are overlooked.

Introduction

In much of the early literature on robot problem solving, the problem solver is assumed to have complete information about the initial state of the world. In some cases, the information is provided to the robot by its programmer; in other cases, the information is obtained through a period of exploration and observation.

In fact, complete information is rarely available. In some cases, the models used by our robots are quantitatively inaccurate (leading to errors in position, velocity, etc.). In some cases, the incompleteness of information is more qualitative (e.g. the robot does not know the room in which an essential tool is located). In this paper, we concentrate on problem solving with incomplete information of the latter sort.

There are, of course, multiple ways to deal with qualitatively incomplete information. To illustrate some of the alternatives, consider a robot in a machine shop. The robot's goal is to fabricate a part by boring a hole in a piece of stock, and it decides to do this by using a drill press. The complication is that there might or might not be some debris on the drill press table.

In some cases, it may be possible to formulate a *sequential* plan that solves the problem. One possibility is a sequential plan that covers many states by using powerful operators with the same effects in those states. In our example, the robot might intend to use a workpiece fixture that fits into position whether or not there is debris on the table. Another possibility is a sequential plan that coerces many states into a single known state. For example, the robot could insert into its plan the action of sweeping the table. Whether or not there is debris, this action will result in a state in which there is no debris.

A second possibility is for the planner to insert a *conditional* into the plan, so that the robot will examine the table before acting, in one case (debris present) clearing the table, in the other case (table clear) proceeding without delay.

A more interesting possibility is for the planner to *interleave planning and execution*, deferring some planning effort until more information is available. For example, the robot plans how to get its materials to the drill press but then suspends further planning until after those steps are executed and further information about the state of the table is available.

The difficulty with all of these approaches is that, in the absence of any good pruning rules, the planning cost is extremely high. In the case of deferred planning, the absence of good termination rules means that the problem solver must plan all the way to the goal, thus eliminating the principal value of the approach.

In this paper, we present some powerful pruning rules for planning in the face of incomplete information. The rules are all based on the simple idea that, if goal achievement is the sole criterion for performance, a planner need not consider one "branch" in its search space when there is another "branch" characterized by equal or greater information.

Fikes introduced interleaved planning and execution in the limited instance of plan modification during execution [Fikes 1972]. Rosenschein's work on dynamic

[1]Funding was provided by the Office of Naval Research under contract number N00014-90-J-1533.

logic formalized conditional planning but paid little attention to computational aspects [Rosenschein 1981]. More recent works do provide domain dependent guidance, but have not uncovered methods that generalize across domains [Hsu 1990], [Olawsky 1990], [Etzioni 1992].

In the next section, we give our definition for problem solving. In section 3, we present a traditional approach to problem solving with complete information. In sections 4-6, we present pruning rules for the three approaches to problem solving mentioned above. Section 7 offers some experimental results on the use of our rules. The final section summarizes the main results of the paper and describes some limitations of this work.

Problem Solving

Our definition of problem solving assumes a division of the world into two interacting parts – an agent and its environment. The outputs of the agent (its actions) are the inputs to the environment, and the outputs of the environment are the inputs to the agent (its percepts).

Formally, we specify the behavior of our agent as a tuple $\langle P, B, A, int, ext, b_1 \rangle$, where P is a set of input objects (the agent's *percepts*), B is a set of internal states, A is a set of output objects (the agent's *actions*), *int* is a function from $P \times B$ into B (the agent's state transition function), *ext* is a function from $P \times B$ into A (the agent's action function), and b_1 is a member of B (the agent's initial internal state).

We characterize the behavior of an agent's environment as a tuple $\langle A, E, P, see, do, e_1 \rangle$, where A is a finite set of actions, E is a set of world states, P is a finite set of distinct percepts, *see* is a function that maps each world state into its corresponding percept, *do* is a function that maps an action and a state into the state that results from the application of the given action in the given state, and e_1 is an initial state of the world.

Note the strong similarity between our characterization of an agent's behavior and that of its environment. There is only one asymmetry – the *see* function is a function only of the environment's state, whereas the *ext* function of an agent is a function of both the percept and the internal state. (For automata theorists, our agent is a Mealy machine, whereas our environment is a Moore machine.) This asymmetry is of no real significance and can, with a little care, be eliminated; it just simplifies the analysis.

The behavior of an agent in its environment is cyclical. At the outset, the agent has a particular state b_1, and the environment is in a particular state e_1. The environment presents the agent with a percept p_1 (based on *see*), and the agent uses this percept and its internal state to select an action a_1 to perform (based on the *ext* function). The agent then updates its internal state to b_2 (in accordance with *int*), and the environment changes to a new state e_2 (in accordance with *do*). The cycle then repeats.

In what follows, we define a *goal* to be a set of states of an environment. We say that an agent *achieves* a goal G if and only if there is some time step n on which the environment enters a state in the goal set:

$$\exists n \; e_n \in G$$

In problem solving with complete information, the agent has the advantage of complete information about the environment and its goal. In problem solving with incomplete information, some of this information is missing or incomplete. The pruning rules presented here are fully general and apply equally well in cases of uncertainty about initial state, percepts, and actions. However, for the sake of presentational simplicity, we restrict our attention to uncertainty about the robot's initial state. In our version, the robot's job is to achieve a goal G, when started in an environment $\langle A, E, P, see, do, e \rangle$, where e is any member of a set of states $I \subseteq E$.

Problem Solving with Complete Information

The traditional approach to problem solving with complete information is sequential planning and execution. An agent, given a description of an initial state and a set of goal states, first produces a plan of operation, then executes that plan.

In *single state sequential planning*, information about the behavior of the agent's environment is represented in the form of a *state graph*, i.e. a labelled, directed graph in which nodes denote states of the agent's environment, node labels denote percepts, and arc labels denote actions. There is an arc (s_1, s_2) in the graph if and only if the action denoted by the label on the arc transforms the state denoted by s_1 into the state denoted by s_2. By convention, all labelled arcs that begin and end at the same state are omitted.

To find a plan, the robot searches the environment's state graph for a path connecting its single initial state to a goal state. If such a path exists, it forms a sequential plan from the labels on the arcs along the path.

Obviously, there are many ways to conduct this search — forward, backward, bidirectional, depth-first, breadth-first, iterative deepening, etc. If the search is done in breadth-first fashion or with iterative deepening, the shortest path will be found first.

As an illustration of this method, consider an application area known as the *Square World*. The geography of this world consists of a set of 4 cells laid out on a 2-by-2 square. The cells are labelled `a`,`b`,`c`,`d` in a clockwise fashion, starting at the upper left cell. There is a robot in one of the cells and some gold in another.

One state of the Square World is shown on the left in Figure 1. The robot is in cell `a` and the gold is in cell `c`. The picture on the right illustrates another state. In this case, the robot is in cell `b` and the gold is in cell `d`.

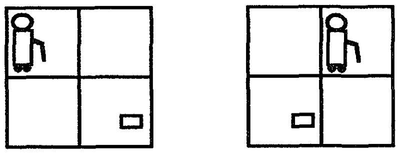
Figure 1: Square World

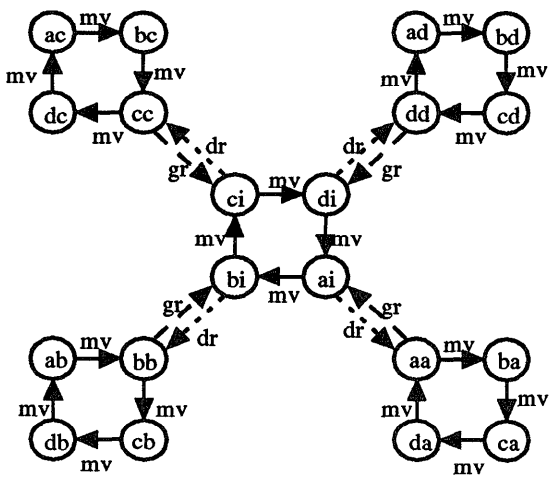

Figure 2: State Graph of Square World

If we concentrate on the location of the robot and the gold only, then there are 20 possible states. The robot can be in any one of 4 cells, and the gold can be in any one of 4 cells or in the grasp of the robot (5 possibilities in all).

Given *our* point of view, we can distinguish every one of these states from every other state. By contrast, consider an agent with a single sensor that determines whether the gold is in the grip of the robot, in the same cell, or elsewhere. This sensory limitation induces a partition of the Square World's 20 states into 3 subsets. The first subset contains the 4 states in which the robot grasps the gold. The second subset consists of the 4 states in which the gold and the robot are in the same cell. The third subset consists of the 12 states in which the gold and the robot are located in different cells.

The Square World has four possible actions. The agent has a single movement action *move*, which moves the robot around the square in a clockwise direction one cell at a time. In addition, the agent can grasp the gold if the gold is occupying the same cell, and it can drop the gold if it is holding the gold, leading to 2 more actions *grab* and *drop*. Finally, it can do nothing, i.e. execute the *noop* action.

Our robot's objective in the Square World problem is to get itself and the gold to the upper left cell. In this case, the goal G is a singleton set consisting of just this one state.

Figure 2 presents the state graph for the Square World. The labels inside the nodes denote the states. The first letter of each label denotes the location of

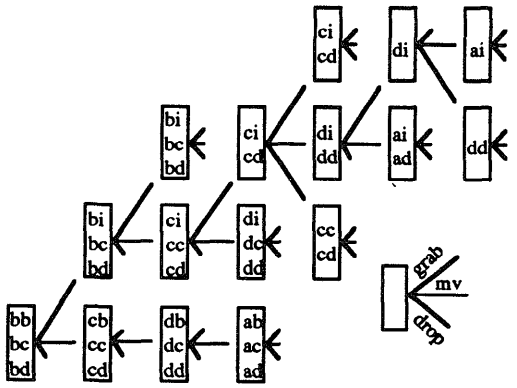

Figure 3: State-set Graph of Square World

the robot. The second letter denotes the location of the gold using the same notation as for the robot, with the addition of i indicating that the gold is in the grip of the robot. The structure of the graph clarifies the robot's three percepts. The four inner states indicate that the robot is holding the gold. The next four states indicate that the gold is in the same cell as the robot (aa,bb,cc,dd). The outermost twelve states indicate that the robot and the gold are in different locations.

Looking at the graph in Figure 2, we see that there are multiple paths connecting the Square World state ac to state aa. Consequently, there are multiple plans for achieving aa from ac. It is a simple matter for *sssp* to find these paths. If the search is done in breadth-first fashion, the result will be the shortest path — the sequence *move*, *move*, *grab*, *move*, *move*, *drop*.

Sequential Planning with Incomplete Information

In problem solving with incomplete information, our robot knows that its initial state is a member of a set I of possible states. How can this robot reach the goal, given a state graph of the world and this set I? One approach is to derive a single sequential plan that is guaranteed to reach the goal no matter which state in I is the actual initial state. The robot can execute such a plan with confidence.

A *multiple state sequential planner* finds a sequential plan if a sequential solution exists using a *state-set graph* instead of the *state graph* that *sssp* uses. In the state-set graph, a node is a set of states. An action arc connects node n_1 to node n_2 if n_2 contains exactly the states obtained by performing the corresponding action in the states of n_1.

Figure 3 illustrates a partial state-set graph for the Square World. In this case, the robot knows at the outset that it is in the upper right cell. However, it does not know the whereabouts of the gold, other than that it is not in its grasp or in its cell. Therefore, the initial state set I consists of exactly three states: *bb*, *bc*, and *bd*.

Note that actions can change the state-set size, both increasing node size and *coercing* the world to decrease node size. The *mssp* architecture begins with node I and expands the state-set graph breadth-first until it encounters a node that is a subset of the goal node. *Msspa* expands nodes using *Results*(N), which returns all nodes that result from the application of each $a \in A$ to node N. The following is a simple version of such an algorithm. For a more thorough treatment of problem solving search algorithms, see [Genesereth 1992].

MSSPA Algorithm

```
1. graph= I, frontier= (I)
2. S = Pop(frontier)
3. If S ⊆ G go to 6.
4. frontier = Append(frontier, Results(S))
5. Go to 2.
6. Execute the actions of the path from I to S in graph.
```

One nice property of this approach is that it is guaranteed – the robot will achieve its goal if there is a guaranteed sequential plan. Furthermore, it will find the plan of minimal length.

However, the cost of simple *mssp* is very high. Given $i = |I|$, $g = |G|$, $a = |A|$, and search depth k, the cost $cost_{mssp}$ of finding a plan is proportional to iga^k.

Fortunately, many of the paths in the state-set graph can be ignored; these are *useless* partial plans. Any path reaching a node that is identical to some earlier node in that path is accomplishing nothing. Furthermore, any path that leads to a state from which there is no escape is simply trapping the robot. Finally, if we compare two paths and can show that one path is always as good as the other path, we needn't bother with the inferior path. We formally define *useless* in terms of any partial plan that begins at any node in the graph. Therefore, note the distinction between the *root* node, the current node being expanded, and node I, the node at which our solution plan must begin:

A partial plan q is *useless* with respect to *root* (the current node) and *result-node*(q) (the resultant node of executing plan q from *root*) if (1) there is a node n on the path from I to *root* (inclusive) such that n is a subset of *result-node*(q), (2) there is a state s in *result-node*(q) that has no outgoing arcs in the state graph, or (3) there is a plan r such that q is not a sub-plan of r and *result-node*(r) is a proper subset of *result-node*(q).

Pruning Rule 1: Sequential Planning

Prune any branch of the state-set graph that leads only to useless plans.

Theorem: *Pruning Rule 1 preserves completeness.*

Furthermore, we can guarantee the minimal solution by modifying condition (3) to (3e): there is some plan r such that *length*(r) $\leq$*length*(q) and *result-node*(r) is a proper subset of *result-node*(q).

Note that once the planner finds a *useless* partial plan, it can prune all extensions of that plan since any solution from the result-node of a useless plan must work either from an earlier node (1) or from some other plan's result-node (3).

This rule can lead to significant cost savings. Recall that $cost_{mssp}$ was iga^k. If the pruning rule decreases the branching factor from a to a/b and searches to depth d for case 3, the cost of *mssp* including the cost of *Pruning Rule 1* is proportional to $(ki^2 + ai + a^d i^2 + ig)a^k/b^k$. We would have savings when:

$$\frac{new\ cost}{old\ cost} = \frac{ki + a + a^d i + g}{gb^k} < 1$$

As a result of the k term in the denominator, $cost_{mssp+heuristics}$ will grow significantly more slowly than $cost_{mssp}$ as the solution length increases.

Conditional Planning

Sequential planning has a serious flaw: some problems require perceptual input for success. In these cases, a sequential planner would fail to find a solution although the system can reach the goal if it consults its sensory input. We need a planner that will find such solutions.

A *multiple state conditional planner* finds the minimal conditional solution using a *conditional state-set graph*. This graph alternates *perceptory* and *effectory* nodes. An effectory node has action arcs emanating from it and percept arcs leading to it. A perceptory node has percept arcs emanating from it and action arcs leading to it. Action arcs connect nodes exactly as in state-set graphs. Percept arcs are labelled with percept names and lead to nodes representing the subset of the originating states that is consistent with the corresponding percept. Figure 4 illustrates part of a conditional state-set graph.

Mscp begins with just the state set I and expands the conditional state-set graph in a breadth-first (or iterative deepening) manner until it finds a solution. The planner uses both *Results* and *Sees*, which expands a perceptory node into a set of nodes, to accomplish the construction. Searching this graph is much less trivial and often more costly than the state-set graph search that *mssp* conducts. This is basically an and-or graph search problem.

Mscp returns a *conditional plan*. This plan specifies a sequence of actions for every possible sequence of inputs. It is effectively a series of nested case statements that branch based upon perceptual inputs. *Mscpa* then executes the conditional plan by checking the robot's percepts against case statements and executing the corresponding sub-plans. Below is a greatly simplified version of the *mscp* algorithm.

MSCPA Algorithm

```
1. graph = I (a perceptory node)
2. Expand every unexpanded perceptory node n
```

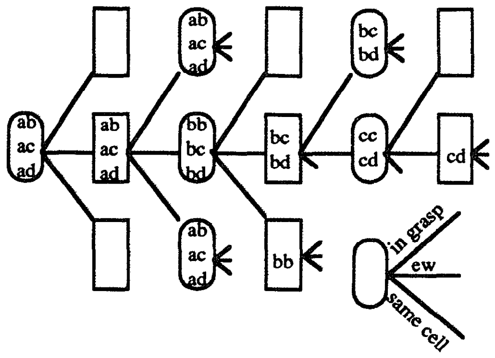

Figure 4: Conditional State-set Graph

using *Sees*(n).
3. If there is a sub-graph of *graph* that specifies all action arcs and reaches a subset of G for every possible series of percept arcs, then go to 6.
4. Expand every unexpanded effectory node m using *Results*(m).
5. Go to 2.
6. Execute that sub-graph as a conditional plan.

Mscpa will reach the goal with a minimal action sequence provided that there is a conditional solution.

Unfortunately, greater power has a price. At its worst, $cost_{mscp}$ is even greater than $cost_{mssp}$ because the space contains perceptual branches: $igp^k a^k$. To extend *Pruning Rule 1* to *mscp*, remember that a sequential plan has a single "result node" while a conditional plan has many possible "result nodes." We define the *result-nodes*(q) to be the set of possible resultant nodes (depending upon perceptory inputs) of conditional plan q. Pruning Rule parts (1) and (2) require trivial changes to take this into account. But part (3) now intends to compare two plans, which amounts to comparing two sets of result-nodes.

We define *domination* such that if plan r dominates plan q, then if there is a solution from *result-nodes*(q), there must be a solution from *result-nodes*(r). Each node in *result-nodes*(r) is dominating if it is a proper subset of some node in *result-nodes*(q). But "result-nodes" that maintain goal-reachability and do not introduce infinite loops are also acceptable. Therefore, we also state that n is dominating if it has reached the goal or even if it is a proper subset of *root*. Below, we define *domination* and revisit *useless* in terms of conditional plans:

Formally, conditional plan r *dominates* a conditional plan q if and only if

(A) $\forall(n_q \in \text{result-nodes}(q))$
$\exists(n_r \in \text{result-nodes}(r))\ n_r \subset n_q, and$

(B) $\forall(n_r \in \text{result-nodes}(r)) either$
1. $\exists(n_q \in \text{result-nodes}(q)) n_r \subset n_q, or$
2. $n_r \subseteq G, or$
3. $n_r \subset root.$

A partial conditional plan q is *useless* with respect to *root* and *result-nodes*(q) if:
(1) There is a node n on the path from I to *root* (inclusive) and there is a node n_q in *result-nodes*(q) such that n is a subset of n_q, or
(2) There is a node n_q in *result-nodes*(q) such that there is a state in n_q with no outgoing action arcs in the state graph, or
(3) There is a partial conditional plan r such that r dominates q.

Pruning Rule 2: Conditional Planning

Prune any branch of the conditional state-set graph that leads only to useless plans.

Theorem: *Pruning Rule 2 preserves completeness.*

Once again, we can reimpose the minimality guarantee by modifying condition (3): (3e) There is a plan r such that $length(r) \leq length(q)$ and r dominates q without use of case $B3$.

Cost analysis of *mscp* with *Pruning Rule 2* yields results identical to $cost_{mssp}$ with the exception that all a^k terms become $a^k p^k$. The pruning rule again provides search space savings as solution length increases.

Interleaved Planning and Execution

Conditional planning is an excellent choice when the planner can extract a solution in reasonable time. But this is not an easy condition to meet. As the branching factor and solution length increase mildly, conditional planning becomes prohibitively expensive in short order. These are cases in which conditional planning wastes much planning energy by examining simply too much of the search space.

What if the system could cut its search short and execute effective partial conditional plans? The system could track its perceptual inputs during execution and pinpoint its resultant fringe node at the end of execution. The planner could continue planning from this particular fringe node instead of planning for every possible fringe node. This two-phase cycle would continue until the system found itself at the goal.

DPA Algorithm

```
1. $states = I$.
2. if $states \subseteq G$ exit (success!!!)
3. Invoke terminating mscp from states and return the resultant conditional plan.
4. Execute the conditional plan, updating states during execution.
5. Go to 2.
```

Dpa will reach the goal provided that there is a conditional solution and the search termination rules preserve completeness.

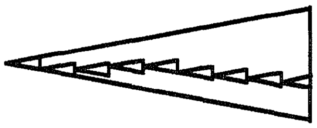

Figure 5: Search Space Savings of DPA

Assume for the moment that our search termination rules return sub-plans of the minimal conditional plan from I to G. We can quantify the dramatic search space savings of delayed planning in this case. Recall that the cost of conditional planning is $igp^k a^k$. The cost of delayed planning is the sum of the costs of each conditional planning episode. If there are j such episodes, then the total cost of delayed planning is $jigp^{k/j} a^{k/j}$. Figure 5 demonstrates the savings, representing *mscpa* with a large triangle and *dpa* by successive small triangles. Note that if the system could terminate search at every step, the search cost would simplify to a linear one: *kigpa*.

Let us return to the search termination problem: how can the planner tell that a particular plan is worth executing although it does not take the system to the goal? The intuition is clear in situations where all our actions are clearly inferior to one action: we might as well execute that one action before planning further. For example, suppose Sally the robot is trying to deliver a package. She is facing the staircase and has two available actions: move forward and turn right 90 degrees. The pruning rules would realize that flying down the stairs is *useless* (deadly) and the planner should immediately return the *turn right* action. We can generalize this rule from single actions to partial plans.

Termination Rule 1 (Forced Plan)*If there exists a plan r such that for all plans q either q is useless or r is a sub-plan of q, then return r as a forced plan.*

Theorem: *Termination Rule 1 preserves completeness and provides a minimal solution.*

The *forced plan* rule has trivial cost when its conditional planner is using *Pruning Rule 2*. Unfortunately, the forced plan criterion can be difficult to satisfy. This rule requires that every non-useless solution from *root* share at least a common first action. This fails when there are two disparate solutions to the same problem. Still, complete conditional planning to the goal may be prohibitively expensive.

We need a termination rule with weaker criteria. The *viable plan* rule will select a plan based upon its own merit, never comparing two plans. The foremost feature of any viable plan is reversibility. We want to insure that the plan does not destroy the ability of the system to reach the goal. This justifies the requirement that each fringe node of a viable plan be a subset of *root*.

A viable plan must also guarantee some sort of progress toward the goal. We guarantee such progress by requiring every fringe node to be a proper subset of *root*. Each viable plan will decrease uncertainty by decreasing the *root* state set size. This can occur at most $|I| - 1$ times.

Termination Rule 2 (Viable Plan)*If there exists a plan r such that for all nodes n_r in result-nodes(r): n_r is a proper subset of root, then return r as a viable plan.*

Theorem: *Termination Rule 2 preserves completeness.*

The fact that the *viable plan* rule does not preserve minimality introduces a new issue: how much of the viable plan should the system execute before returning to planning? Reasonable choices range from the first action to the entire plan. Experimental and qualitative analysis indicates that this variable allows a very mild tradeoff between planning time and execution time.

Average-case cost analysis of *dpa* using the *Viable Plan Rule* yields hopeful results. Recall that pure conditional planning would cost $igp^k a^k$. Suppose a *dpa* system executes n partial plans of depth j, resulting in node I_n with size h. From I_n, there are no search termination opportunities and the planner must plan straight to the goal. Assume that there is some c such that $i = ch$. The cost per node of the *Viable Plan Rule* is i^2.

For case 1, assume $g > h$. The cost from I to I_n is $n(i^2 + ig)p^j a^j$. The worst-case cost from I_n to the goal is $(h^2 + hg)p^k a^k$ when I_n is no closer to the goal than I. This can occur precisely when $g > h$ and *coercion* is not necessary. When we divide the cost of *dpa* by the cost of *mscp* we are left with savings when:

$$\frac{n(i+g)p^j a^j}{gp^k a^k} + \frac{h}{i} + \frac{h^2}{ig} < 1$$

For case 2, assume $g \leq h$. Then a number of *coercive* actions occur along the way from I to G. If we assume that these *coercives* are distributed evenly, then there are $(h - g)/2$ *coercives* from I_n to the goal and $k - (i - h)/2$ total steps from I_n to the goal. The total cost changes to $n(i^2 + ig)p^j a^j + (h^2 + hg)p^{k-(i-h)/2} a^{k-(i-h)/2}$. The third term, h^2/ig, changes to $h/(gp^{(i-h)/2} a^{(i-h)/2})$, which is now less than one since we assumed that $g \leq h$.

Experimental Results

We implemented these planners in four domains using property space representations, in which sets of properties correspond to sets of states satisfying those properties. For DPA, we implemented both termination criteria and executed the first step of viable plans. MJH World is a realistic indoor navigation problem. Wumpus World is a traditional hero, gold, and monster game. The Bay Area Transit Problem [Hsu 1990] models an attempt to travel from Berkeley to Stanford despite traffic jams. The Tool Box Problem [Olawsky 1990] describes two tool boxes that our robot must bolt. The following depicts p, a, i, and g:

	p	a	i	g
MJH1	2	4	4	4
MJH2	2	4	6	6
MJH3	2	4	6	6
WUM1	4	6	24	4
WUM2	4	6	44	4
BAT	16	4	8172	8172
TBOX	3	14	4	4

Below are running times (in seconds) and plan lengths, including average length in brackets, for all architectures with and without pruning rules. The DPA statistics were derived by running DPA on every initial state and averaging the running times. The dash (-) signifies no solution and the asterisk (*) indicates no solution after 24 hours running time.

	SPA	SPA_h	CPA	CPA_h	DPA
MJH1	34.6	4.1	82.8	21.4	1.6
MJH2	-	-	74.6	24.6	1.5
MJH3	-	-	*	623.6	2.4
WUM1	-	-	877.7	104.5	1.3
WUM2	-	-	*	15111	1.7
BAT	-	-	*	*	3.6
TBOX	-	-	*	*	73.1

	Len_{dpa}	Len_{ideal}
MJH1	9-11[10]	7
MJH2	8-12[10]	6-10[8]
MJH3	8-16[11]	6-12[10]
WUM1	7-15[9.2]	7-11[8.5]
WUM2	7-20[10.8]	7-15[9.8]
BAT	5-12[6.5]	5-12
TBOX	10-13[11.7]	10-13

BAT introduces a huge initial state set and a high branching factor. DPA time results for BAT are based upon a random sampling of thirty actual initial states. TBOX is the hardest problem because the action branching factor is so high that even sequential programming with complete information is impossible without pruning. The TBOX running times are based upon running DPA on every I possible in the Tool Box World. Our DPA planner never issued an *unbolt* command in any TBOX solution. Olawsky regards the use of *unbolt* as a failure and, using that definition, our termination rules produced zero failures in TBOX. A surprising result concerning both of these large domains is that the execution lengths were extremely similar to the ideal execution lengths.

Conclusion

This paper presents some powerful pruning rules for problem solving with incomplete information. These rules are all domain-independent and lead to substantial savings in planning cost, both in theoretical analysis and on practical problems. The rules are of special importance in the case of interleaved planning and execution in that they allow the planner to terminate search without planning to the goal.

Although our analysis concentrates exclusively on uncertainty about initial states, the rules are equally relevant to uncertainty about percepts and actions.

Our analysis also assumes that state sets are represented explicitly, but the pruning rules apply equally well to planners based on explict enumerations of property sets (e.g. Strips) and logic-based methods (e.g. Green's method).

One substantial limitation of this work is our emphasis on state goals. We have not considered the value of these methods or rules on problems involving conditional goals or process goals. We have also not considered the interactions of our rules with methods for coping with numerical uncertainty. Further work is needed in both areas.

Acknowledgements

David Smith introduced the machine shop robot example. Sarah Morse provided a helpful early critique of this paper. Tomas Uribe provided useful late-night suggestions.

References

Etzioni, O., Hanks, S., and Weld, D. 1992. An Approach to Planning with Incomplete Information. In Proceedings of the Third International Conference on Knowledge Representation and Reasoning.

Fikes, R. E., Hart, P.E., and Nilsson, N. J. 1972. Learning and Executing Generalized Robot Plans. *Artificial Intelligence* 3(4): 251–288.

Genesereth, M. R. 1992. *Discrete Systems.* Course notes for *CS 222.* Stanford, CA: Stanford University.

Hsu, J. 1990. Partial Planning with Incomplete Information. In Proceedings of AAAI Spring Symposium on Planning in Uncertain, Unpredictable, or Changing Environments. Menlo Park, Calif.: AAAI Press.

Olawsky, D., and Gini, M. 1990. Deferred Planning and Sensor Use. In Proceedings of the DARPA Workshop on Innovative Approaches to Planning, Scheduling, and Control. Los Altos, Calif.: Morgan Kaufmann.

Rosenschein, S.J. 1991. Plan Synthesis: A Logical Perspective. In Proceedings of the Seventh International Conference on Artificial Intelligence. Vancouver, British Columbia, Canada.