PAO* for Planning with Hidden State

Dave Ferguson[†], Anthony Stentz[†], and Sebastian Thrun[‡]

[†] Robotics Institute
Carnegie Mellon University
Pittsburgh, PA, USA

[‡] Computer Science Department
Stanford University
Stanford, CA, USA

***Abstract*— We describe a heuristic search algorithm for generating optimal plans in a new class of decision problem, characterised by the incorporation of hidden state. The approach exploits the nature of the hidden state to reduce the state space by orders of magnitude. It then interleaves heuristic expansion of the reduced space with forwards and backwards propagation phases to produce a solution in a fraction of the time required by other techniques. Results are provided on an outdoor path planning application.**

I. Introduction

The path planning problem has been addressed extensively by the robotics research community. A number of approaches exist to solve the planning problem in deterministic domains (e.g. A*). Recently, incremental approaches have been developed which allow for corrections to the original cost values associated with state transitions [1], [2]. These algorithms are both optimal and efficient given the information they ascribe to. However, they are unable to cope optimally with uncertain apriori information.

Consider a robot navigating outdoors equipped with an overhead map of the surrounding area (generated by satellite or an aerial vehicle). The resolution of the map may be much lower than the resolution used by the robot to navigate. Due to this low resolution, there is some uncertainty as to whether portions of the terrain are traversable or not. As a result, certain cells in the final map will hold incomplete information: the robot knows some information about what the terrain is like in the general vicinity but not the exact value at the particular cell. Some of these cells may be crucial for the robot's planning task, such as those residing in narrow passageways.

Current planners deal with such cells in one of two ways [2], [1]. Firstly, they may assign a default value to all such cells. Typically, this method (known as "assumptive planning" [3]) allows the robot to update the information about cells as it moves, so that it can plan using the actual terrain of the cell when it comes close enough to determine it. Alternatively, they may compute an expected value of the cell. Given some probability density function over possible terrains, one can compute the expected terrain of the cell and plan using this expectation.

However, neither of these methods makes use of the information provided to act optimally. If the cell has a non-zero probability of being untraversable, then planning using the expected terrain (or terrain cost) will not in general produce the best path. This is because the cost of the cell must reflect the possibility that the cell turns out to be untraversable and the robot must find an alterior path to the goal. The cost of this alterior path is a global property and cannot be derived directly from the terrain cost of the cell in question.

This paper explores a family of problems that extend classical deterministic path planning problems with a limited type of hidden state (discussed below), enabling us to model exactly the above type of uncertainty. This family encompasses several key application problems, in particular mobile robot navigation in environments with detectable state (such as indoor environments with doors which may be open or closed, or outdoor environments with gaps that may turn out to be too narrow to pass through). It also includes the graph-theoretical Canadian Traveller's Problem, which consists of planning a route through a graph where edges may be untraversable [4].

We present a solution to such problems which performs search in a reduced information state space using a heuristic to guide the search, as with AO* [5]. The key theoretical contribution of this paper is a new algorithm, *PAO**, which updates the heuristic value of states throughout the state space in such a way as to reduce the required computation considerably.

II. Deterministic Decision Problems with Hidden State

This section describes the basic *Decision Problems with Hidden State* (DPHS) framework as applied to robotic path planning, beginning with a brief review on deterministic decision problems. We go on to discuss how such problems can be reduced to a search over the space of their hidden state component.

A. Deterministic Decision Problems

A Deterministic Decision Problem (DDP) consists of the following components:

- **States**. The state of the DDP is denoted by x. At any point in time, the state is fully observable. In path planning the state corresponds to the position of the robot in the environment.
- **State transitions**. The states of the system are related through an adjacency list. An agent may transition between the current state and any adjacent state in a deterministic manner.
- **Cost Function**. DDP's need some measure of the cost of transitioning between two adjacent states, $Cost(x, y)$,

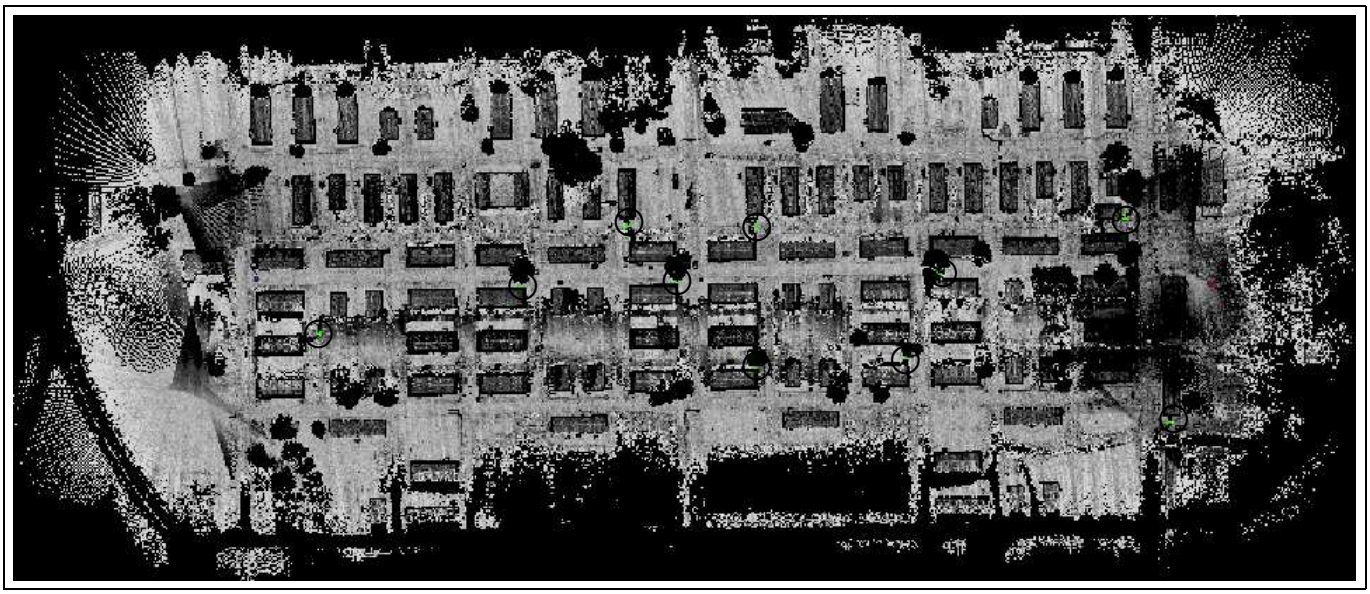

Fig. 1. A gradient map of an outdoor environment generated by a helicopter. Several pinch points are shown in green (and circled in black). The gray scale value of each cell reflects the cell's traversability: the darker the cell, the more difficult the terrain (with black cells representing obstacles). The displayed area is 300 $\times$ 700 meters. Data courtesy of Omead Amidi and Ryan Miller.

which is usually computed from the terrain costs of the cells associated with the two states.

The central problem in path planning DDPs is to calculate a path from a start state s to a goal state g while minimizing the overall cost incurred. The overall cost of a cell x is defined as:

$$Cost(x, g) = \min_{y \in nbrs(x)} Cost(x, y) + Cost(y, g). \quad (1)$$

To solve for the cost of an individual state s a fast A* based focussed propagation of values can be performed [6].

B. Hidden State

The DPHS model allows for the incorporation of hidden state into the DDP framework. The extension is as follows:

- **Hidden state**. We deal with p elements comprising our hidden state (known as 'pinch points'). In our path planning application, each relates to a particular cell in the environment which may be *traversable* or *untraversable*. Not only is there uncertainty associated with the traversability of these cells, but there are great consequences if they are untraversable (i.e., an alternative route holds a much higher cost). Each hidden state element holds a probability distribution over its possible values (for us a single number corresponding to its probability of being *untraversable*). The elements are assumed static: their true values are fixed throughout.
- **Observations**. The true value of a hidden state element may be observed by an agent. We assume noiseless observations through a proximity sensor which allows the agent to determine the value of a pinch point from any adjacent cell in the environment. A proximity sensor approximates the near range navigation sensors typically employed by indoor and outdoor robots.

The hidden state thus relates the DPHS framework to Partially Observable Markov Decision Problems (POMDP's)[7]. However, under the current model there is no uncertainty in robot action and all observations are noiseless. As we will see, these characteristics allow us to use heuristic search algorithms over state spaces intractable with POMDP's [8].

C. DPHS Information States

An *information state* is the state of knowledge an agent may have concerning the true state of the environment, both the known and the hidden elements. An information state in a DPHS is written as (x, H), where x is the observable state and $H = \{v(h_1), v(h_2), \ldots, v(h_p)\}$ is the agent's knowledge concerning the hidden state. Each hidden state element may be known to be *traversable* ($v(h_i) = t$), known to be an *obstacle* ($v(h_i) = o$), or *unknown* to the agent ($v(h_i) = u$). Thus, given a DPHS with an $m \cdot n$ known state space (planning grid) and p pinch points, the number of information states is $m \cdot n \cdot 3^p$.

The noiseless nature of our observations thus restricts the resulting information state space to be finite, allowing us to perform discrete search to achieve an optimal result. However, this state space grows exponentially with the number of hidden state elements. Planning over the entire space is therefore prohibitively expensive.

Fortunately, there are a few key properties of our problem which enable us to substantially reduce the amount of computation required.

Firstly, we can reduce the computation to consider only the hidden state elements. If a cell k in the environment holds a cost to each of the hidden state elements (computed *without* passing through any other hidden state element) and to the goal, then the overall cost of that cell in any information state I can be computed given the overall costs of each hidden state element in I. In particular, since we are planning from

a start state s we need only compute the cost for this single cell in the single information state the agent starts in. We thus simplify our information state description to the agent's knowledge concerning the hidden state, H. Secondly, because the true values of the hidden state elements are fixed, we know that an agent will only ever gain information. In other words, because our environment is static, once an agent observes a pinch point to be traversable/untraversable it will never again be uncertain of that pinch point's true value. So our planning space is without cycles. Finally, because each unknown hidden state element in an information state can turn out to be only either traversable or untraversable, we are presented with a natural admissible heuristic to use for searching the restricted information space (that is, assume all unknown hidden state elements are *traversable*).

III. Solving the DPHS

As mentioned previously, our solution exploits the fact that we can reduce the problem to a search over just the space of the hidden state component. To do this, we reduce the environment to an adjacency graph between hidden state elements.

A. The Face Graph

Each pinch point in a planning environment may have a number of *faces*, which consist of adjacent cells opening out into different cost regions of the environment. These faces can be thought of as different entrances and exits associated with the pinch point (see Figure 2). Each neighboring (non-obstacle) cell to a pinch point resides on exactly one of the pinch point's faces.

The adjacency graph (*Face Graph*) links up these faces to one another and, in doing so, provides a compact representation of the hidden state elements of the environment. Figure 1 illustrates an environment with 10 pinch points and Figure 2 shows a section of its corresponding face graph. The cost of an arc between two faces represents the lowest cost associated with moving along an optimal (pinch-point free) path *between* the faces and is used to propagate values from one face to another.

1) Creating the Face Graph: In order to produce the appropriate arcs and arc costs associated with the face graph, we run an initial cost propagation through our environment (using prioritized sweeping [9]) which determines, for each cell in the planning grid, the cost to each different face and the cost to the goal. For this propagation we treat each pinch point as if it were an obstacle, so that we ascertain which faces are directly accessible to one another. Then, given some face f_1 and its corresponding costs to each other face $Cost(f_1, f_2)...Cost(f_1, f_n)$, we create an arc to each face f_i for which $Cost(f_1, f_i) < \infty$ and label it with its associated cost. Similarly, an arc is created to the goal if it is accessible (i.e., if $Cost(f_1, Goal) < \infty$).

After this pass, we then compute the cost between faces associated with the same pinch point. This cost (labelled $CostThru(f_i, f_j)$) is then used in information states which have the respective pinch point holding the value t (traversable).

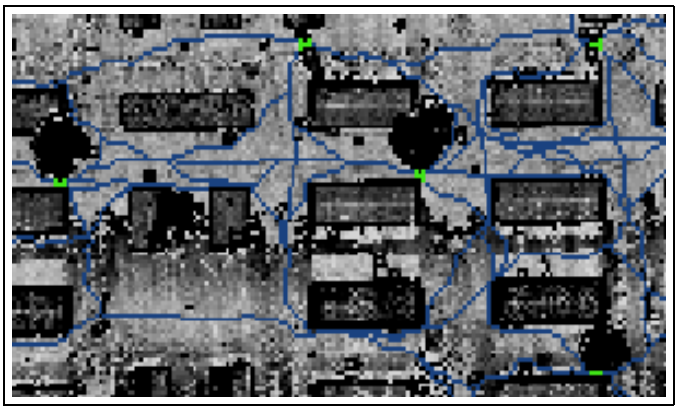

Fig. 2. The lowest cost arcs between a subset of the faces in the previous traversability map. Each pinch point has a number of faces attached to it, corresponding to its different general entries. Here the pinch points are in green, enlarged slightly to aid in illustration, and the cost arcs are in blue.

After constructing our face graph, we have reduced the planning DPHS to the graph-theoretic Canadian Traveller's Problem (CTP) concerning the hidden state elements. We now introduce four methods for solving this problem. The first three draw from ideas common to the planning and MDP communities. The final approach involves our novel algorithm: PAO*.

B. The Complete Solution

The first approach we consider uses the monotonicity of the agent's information concerning the hidden state to derive an iterative solution to the CTP. Ultimately, we are trying to compute an optimal path for an agent which starts out in the information state $H = \{u, u, \ldots, u\}$. However, the cost values of each face in our graph at this state can be recovered directly from the costs of the faces in the information states which have *exactly one* pinch point of known value. These face costs in turn can be computed from the costs of the faces in the information states which have *two* known pinch points, and so on.

The reason for this is as follows. As soon as the agent moves to a face associated with a pinch point which is of value u, the agent learns (through its proximity sensor) what the actual value of that pinch point is. As a result, our agent is constantly increasing its knowledge of the state of the environment, one pinch point at a time. To solve for the values of information state i we must have the values of every information state which is reachable from i. These are exactly the information states which have one more pinch point of known value.

In short, we iterate from the base-case information states where the environment is completely deterministic (all pinch points are of known values) up to information states with increasing numbers of pinch points holding the value u.

The costs of each face in the deterministic information states (there are 2^p such states for p pinch points) are solved using standard value iteration (VI), since we are solving for all faces

Compute Cost $C[f_k, i]$**:**
Cost $\leftarrow Cost(f_k, Goal)$
$v \leftarrow v(h(f_k))$

If $v = u$
  Cost $\leftarrow p(h(f_k) = o) \cdot C[f_k, i_o] + p(h(f_k) = t) \cdot C[f_k, i_t]$
Else
  Cost $\leftarrow \min(\text{Cost}, \min_{f_i \in f}(C[f_i, i] + Cost(f_k, f_i)))$
  If $v = t$
    Cost $\leftarrow \min(\text{Cost}, \min_{f_i \in a(f_k)}(C[f_i, i] + CostThru(f_k, f_i)))$

Return Cost

Fig. 3. General value iteration algorithm to compute cost of face f_k in information state i.

at once[1]. Each face has its cost initialized to its arc cost to the goal. If no such arc exists (i.e. the face has no path to the goal without needing to traverse some hidden state element), the cost is initialized to infinity.

Once the costs of these states have been determined, the costs of faces of subsequent information states can be solved using the modified value iteration algorithm in Figure 3.

In this generalized algorithm, $C[f_k, i]$ represents the cost of face f_k in information state i, $h(f_k)$ is the pinch point to which face f_k belongs, $v(h_j)$ is the value of pinch point h_j in information state i (one of t, o, or u), i_o and i_t are the information states similar to i in all respects except that $h(f_k)$ is of value o and t, respectively, and $a(f_k)$ is the set of all faces associated with pinch point $h(f_k)$.

The algorithm works by finding the complete set of successor faces (combined with information states) from a given face f_k. If f_k is attached to a pinch point with value u (in our information state i), then we use the probability measure associated with this pinch point to generate an expected cost of the current face. This expected cost combines the values of the face in the information states i_o and i_t. If the pinch point is known, then we update our current cost to be the minimum of the cost associated with moving to any adjacent face (and the goal, if reachable). If the pinch point is known *and is traversable* then we can add to our contention the faces on the other side of the current pinch point as these, too, are available successors.

The cost computation is performed for each face in the information state repeatedly until convergence.

C. Reachability Analysis

A major drawback of the above approach is that *every* possible information state is examined and solved for, including states that can never be realized given the initial state the agent resides in.

Consider a robot navigating outdoors. If there are a number of pinch points that the robot cannot directly reach (i.e., without going through some other pinch point), it does not make sense for it to process any information states where it holds information concerning these pinch points without knowing the values of the pinch points it would have to pass through to get this information. Such states are impossible given the initial position and information state of the robot.

Reachability analysis has been used extensively by the Markov Decision Processes community (and others) to restrict computation to information states which are physically reachable from the initial state [10], [11]. The idea is to propagate outwards from the initial state, marking each subsequent state as reachable. All states left unmarked can be ignored in our solution derivation.

Incorporating reachability considerations, the algorithm described above changes in two ways. Firstly, an initial propagation step is performed, branching out from the initial state, to mark all the reachable states. Secondly, the iteration phase only considers the states marked in the first step, thus ignoring the irrelevant areas of our information space.

D. AO*

The number of examined states can be further reduced by performing heuristic-based search over the information space. AO* is a classic search algorithm which performs such a heuristic search over an AND-OR graph [5], [12]. An AND-OR graph contains two types of nodes: AND nodes obtain their values from combining *all* their child nodes, while OR nodes compute their values from choosing *a single node* from their children.

The planning CTP can be represented as an AND-OR graph as follows. Each node in the graph corresponds to a face in a particular information state. The root of the graph (an OR node) is the start cell s in the information state $H = \{u, u, \ldots, u\}$. The next level of the graph corresponds to all the faces which have arcs to s (and each node at this level has information state H). These are AND nodes: each has two children representing the two possible information states realizable from visiting the node. These two children each have the same face as their parent but reside in different information states (one has the hidden state element associated with the face of value *t*, the other *o*). These children are OR nodes, the next level are AND nodes, and so on.

Intuitively, from s the agent can choose to move to any adjacent face or directly to the goal (if clear). Thus, its cost is a function of the *minimum* cost of the adjacent faces. Once it has moved to one of these faces, it learns the true value of the hidden state element associated with the face. It does not choose this value: it is taken from the range of possibilities (in our case just {*traversable, obstacle*}) according to the hidden state element's probability measure. Thus, the cost of the parent node is a combination of the cost of *both* its children.

AO* searches an AND-OR graph by gradually building a solution graph from the start state through two alternating phases. First, it grows the best partial solution by expanding one of the non-terminal leaf nodes and assigning admissible heuristic costs to its children. Then it uses the newly computed costs to propagate cost revisions to the parent node and

[1]If we were only interested in the value of one face it would make sense to use A* rather than value iteration.

1) The initial solution graph consists solely of the start node, s, in the original information state H.

2) While the solution graph has some nonterminal leaf node:
   (a) *Expand best partial solution*: Expand a nonterminal leaf node and compute heuristic values for its two children. Add the children to the solution graph, noting whether they are nonterminal.
   (b) *Propagate cost changes and update solution*: Compute an updated cost estimate of the original leaf node given the costs of its children. If its cost has changed, update its parent's cost to reflect this change. If the parent is an OR node, the current child may be replaced if it no longer provides the minimum cost. Continue propagating up the graph until a node is reached whose cost does not change.
3) Return the optimal solution graph.

Fig. 4. The AO* algorithm

onwards up the graph. The full algorithm as used is illustrated in Figure 4.

The efficiency of AO* is obtained through its use of a heuristic to limit the amount of the AND-OR graph that is examined. The resulting solution graph can often be constructed through observing only a fraction of the complete graph. In our case, the heuristic value of a new node n is computed through solving a VI over the 'heuristic counterpart' of n: the deterministic state characterized by the best-possible true values the hidden state elements in n could have. For elements already known in n (i.e. elements h_p such that $v(h_p) = t$ or $v(h_p) = o$) the elements are left untouched. Elements still unknown (with $v(h_p) = u$) are assigned the value t. The resulting values are guaranteed to be admissible and the VI over the deterministic state is very fast.

*E. PAO**

AO* works very well in certain situations, typically where most of the reachable states are clearly undesirable. This is because its use of a heuristic allows it to focus its search away from highly sub-optimal faces. However, given the current problem domain and heuristic, it is possible for the algorithm to examine far more states than necessary. Furthermore, because the partial solution graph is altered during the course of the algorithm, AO* can re-expand the same state several times. In the worst case, this can result in execution times that surpass the complete solution by orders of magnitude (see results).

PAO*, short for *Propagating AO**, is an algorithm which attempts to capture the clear benefits of using a heuristic-based search while minimizing the possible drawbacks of using partial solution graphs. It does this by propagating cost changes not only upwards to parents in the partial solution graph, but sideways to neighbors (in the complete AND-OR graph), and downwards to children. The resulting approach

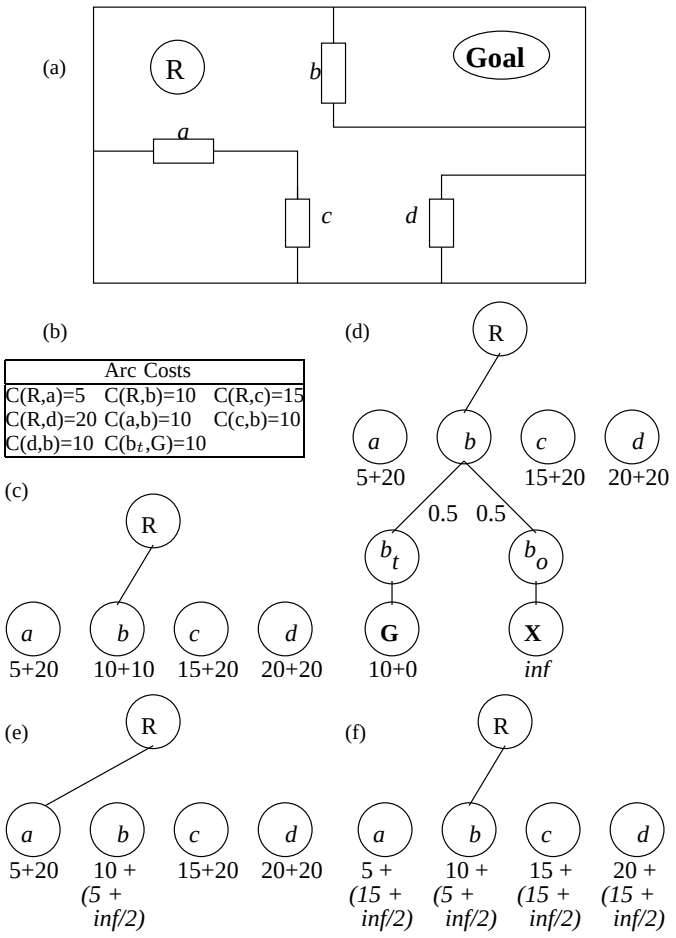

Fig. 5. Sample planning problem involving a robot, a goal, and four doors which may be open or closed. Some sample arc costs between door faces and points of interest are shown in (b). The first two expansions of the AO* and PAO* solution graphs are shown in (c) and (d). The values under each node represent the parent's cost to the node plus the cost of the node itself. The resulting AO* and PAO* solution graphs after propagating the cost change from (d) are shown in (e) and (f).

makes full use of all received information and thus allows for more informed decisions to be made at each stage of the process.

The key insight behind PAO* is that cost changes are rarely isolated. If a node updates its cost based on an altered child cost, it is likely this update will affect the costs associated with that node's neighbors. Consider the simple scenario described in Figure 5, where a robot (at the position R) must navigate to a goal within an environment containing 4 doors. Each door may be open or closed and the robot is equipped with a proximity sensor to tell, upon reaching a doorway, which possibility prevails. For clarity we have only dealt with the four reachable faces from the robot's initial position and have shown only relevant arc costs. We have slightly abused notation and used $C(b_t, G)$ to express the cost between face b and the goal (assuming there is no door at b).

Initially, all the faces have heuristic cost values associated with them, corresponding to their lowest possible costs. In this particular scenario, all these costs correspond to paths through face b. As explained previously, these heuristic values are the

result of a value iteration over the state of the environment where all the doors are open. Given these initial face costs, the best successor from R is *b*, giving R an initial cost of $C[R] = C(R, b) + C[b] = 20$. So *b* is placed as the child of R in the partial solution graph (see Figure 5(c)).

After a single further expansion of the graph, the cost associated with *b* changes dramatically. Its two possible outcomes are computed and its resulting cost is forced to reflect the possibility that the adjacent doorway may be blocked, in which case no path to the goal is possible. However, AO* only uses this new information to update the value of the root node, which in turn chooses a new child (one of the faces whose cost is the original heuristic value - see Figure 5(e)). This does not make effective use of the information gained in the previous expansion. Because all the faces depend on *b* to reach the goal, their costs are affected by any cost changes associated with *b*. By ignoring this, AO* ends up expanding each of the faces reachable from R one by one in order to arrive at the same cost values that could have been computed directly from this initial expansion.

PAO* propagates information concerning updated costs more thoroughly through the information state space. The complete algorithm is given in Figure 6. There are four key differences between its operation and that of AO*.

The first difference allows PAO* to overcome the difficulty AO* faces in domains such as our simple robot navigation example of Figure 5. In its propagation of cost changes, PAO* propagates the updated child cost through the *entire* child information state, so that dependencies between faces will be reflected in their costs and the parent will be able to use the most accurate information possible in determining its own cost and (currently) optimal child (see Figure 5(f)).

Secondly, PAO* propagates cost values *down* the solution graph. Given an AND node with two children corresponding to the two possible true values of the node's pinch point (traversable and obstacle), the cost of the parent node should never be greater than the cost of the obstacle child node. Similarly, the parent should never have a lower cost than the traversable child. This makes intuitive sense: if the true value of a given pinch point is known to be traversable, then we are certainly in at least as desirable a state as if we did not know anything about that pinch point's true value. However, because a face in a given information state can be reached through a number of different paths, often this will not hold for a given parent and child combination. It is even more likely that *other* faces in the same information state as the face of the child/parent node will have unrealistic costs. To take advantage of this piece of intuition, PAO* updates the face costs of the states associated with obstacle nodes so that they are lower bounded by their parent state values. The update looks at each face in the child state and assigns it the maximum of the costs assigned to it by the two respective states, parent and child. PAO* performs this update as it traverses down the solution graph to select the next nonterminal node for expansion.

As an example, consider again the simple environment given in Figure 5(a). Assume this time the robot starts not from the position marked *R* but rather in the room blocked by the doors attached to faces *a* and *c*. Let's assume further that the robot initially expands the face on the other side of the door from *a* (call this face a'). If it then chooses to expand the *traversable* child of a', it will receive an updated value for face *b* which takes into account its probability of being untraversable and precluding any solution. When it propagates this information back to its parent and on up to node a', suddenly the traversable child of a' has a higher cost than its obstacle child, since the obstacle child still uses the heuristic cost of each unexpanded face (including *b*). PAO* propagates the updated information to the obstacle child on its next pass down the graph and, as a result, arrives at a much better heuristic estimate of its cost.

The relationship works both ways, however, and PAO* also updates the face costs of *parent* information states from their *traversable* child states. It performs this update as part of its propagation of cost changes back up the solution graph. These two propagation steps combine to allow information gained at one end of the solution graph to be accessible at the other.

1) The initial solution graph consists solely of the start node, s, in the original information state H.
2) While the solution graph has some nonterminal leaf node:
   (a) *Generate fringe node*: Starting from the root, traverse down the solution graph until a nonterminal leaf node is encountered. Along the way, update obstacle child states to have their face costs lower bounded by their parent states.
   (b) *Expand best partial solution*: Expand the nonterminal leaf node and compute cost values for the information states of its children. Traversable child states are given heuristic costs. Obstacle child states inherit their parents' cost values as lower bounds then perform limited VI's over their heuristic counterparts to potentially increase these values. Add the children to the solution graph, noting whether they are terminal.
   (c) *Propagate cost changes and update solution*: Compute an updated cost of the original leaf node given the costs of its children. If the node's cost has changed, update the cost estimates for its *entire information state* and update its parent's cost to reflect these changes. If the parent is an OR node, the current node may be replaced if it no longer provides the minimum cost. If the node is a traversable child, update the costs associated with the entire parent state to be lower bounded by the current state. Continue propagating up the graph until a node is reached whose cost does not change.
3) Return the optimal solution graph.

Fig. 6. The PAO* algorithm

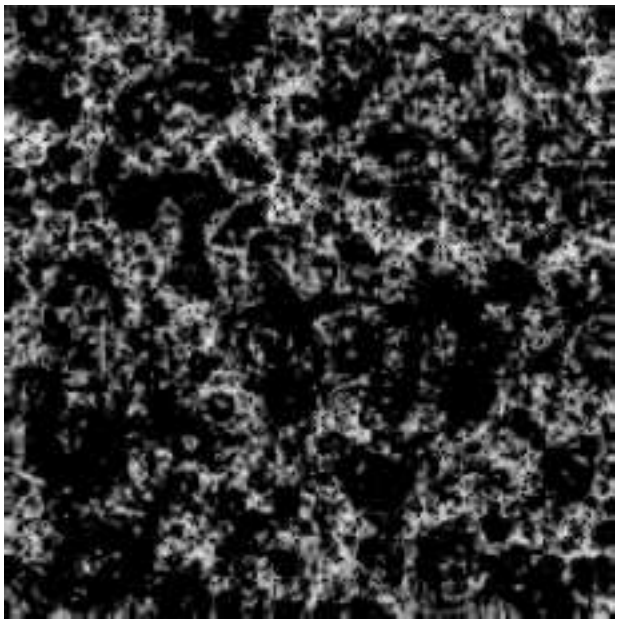

Fig. 7. An example fractal terrain used in testing. 10 pinch points were manually selected from each environment.

The final difference resides at the node expansion stage. In the AO* expansion of a nonterminal node, indifference is shown towards the nature of the two children. Both are assigned initial heuristic values and these values are then used to update the parent. However, it is possible to exploit the relationship between the face costs of parent and child states described above to produce more realistic values for at least one of the two children. PAO* allows the obstacle child state to inherit the values of the parent state, then performs a VI over the heuristic counterpart of the obstacle state which carefully ensures the resulting costs are not less than the parent costs. This is done by initializing each face with the cost of its arc to the goal (if one exists), then performing standard value iteration. If, at any point during the VI, the cost of a particular face becomes less than that face's cost in the parent state, the face has its cost fixed to the parent cost for the remainder of the value iteration. This inheritance allows all the information concerning the parent state to be retained and utilised by the obstacle child state.

IV. Results

Figure 8 compares the performance of PAO* to all 3 alternative approaches discussed here. The algorithms were tested over 20 different fractally generated environments[2], each with 10 pinch points (selected manually from the environment). Each fractal environment was generated using a different density to simulate varying degrees of terrain difficulty. For each environment we varied the probabilities associated with each pinch point randomly to produce 10 different test cases.

In each case, the task was to find the optimal path given a start state at one end of the environment and a goal state at the other. Each environment was 200×200 cells in size. The time taken for the initial arc cost propagation is independent of which approach is used and is highly dependent on the size of the environment, so it has been left out of our comparison. On average, this propagation took about 6 seconds. All times reported are for a 1.4 GHz Pentium III Processor.

[2]see [1] for details of the fractal generation process. We used a gain of 20 and varied the number of levels from 1 to 20.

Approach	Complete	Reachability	AO*	PAO*
Examined				
Min	59048	19684	5	5
Max	59048	59048	59048	2146
Avg	59048	52924.8	25272.0	405.8
Expanded				
Min			2	2
Max			26748382	3150
Avg			5794832.5	314.8
Run Time				
Min	1.3278	0.4749	0.0004	0.0002
Max	2.8071	2.5056	1011.5316	0.2827
Avg	1.8250	1.3594	232.5929	0.0302

Fig. 8. The results of our four approaches applied to a test set of 200 pinch point environments. The three criterion displayed are the number of information states examined, the number of states expanded (in the case of AO* and PAO*) and the run time required to find the optimal solution.

We used 10 pinch points in our analysis in order to keep the numbers down. Although the relative performance of each approach alters slightly given an increased quantity of hidden state (the advantage of reachability analysis over the complete solution, for example, will increase), we found that 10 pinch points was enough to portray the general trend.

The first criteria used to evaluate the approaches was the number of information states examined. For the complete approach, this is a fixed number, as it exhaustively solves each information state from the deterministic cases upwards. Reachability analysis allows us to reduce the number of examined states quite considerably. AO* at times examines only a fraction of the states, however on occasion it was forced to deal with the complete information space. PAO* was able to keep the number of considered states extremely low, on average looking at only 406 (out of a state space of 59048).

To compare PAO* with AO* more thoroughly, we generated results for the number of states expanded during the run of each algorithm. This corresponds to the number of fringe elements which were further processed to produce their traversable and obstacle children. Because the partial solution graph maintained by these approaches is continually updated and reshaped, a single state can be expanded several times. Thus, the number of expanded states can be much larger than the total number of distinct states examined. AO* on average expanded more than $1.8 \cdot 10^4$ times as many states as PAO*.

The enormous difference in state expansions carried over into the run time results. AO* performed on average much more poorly than the complete solution, although certain environments it was able to solve very quickly. PAO* performed considerably better than any of the other approaches, with an average run time of 0.03 seconds.

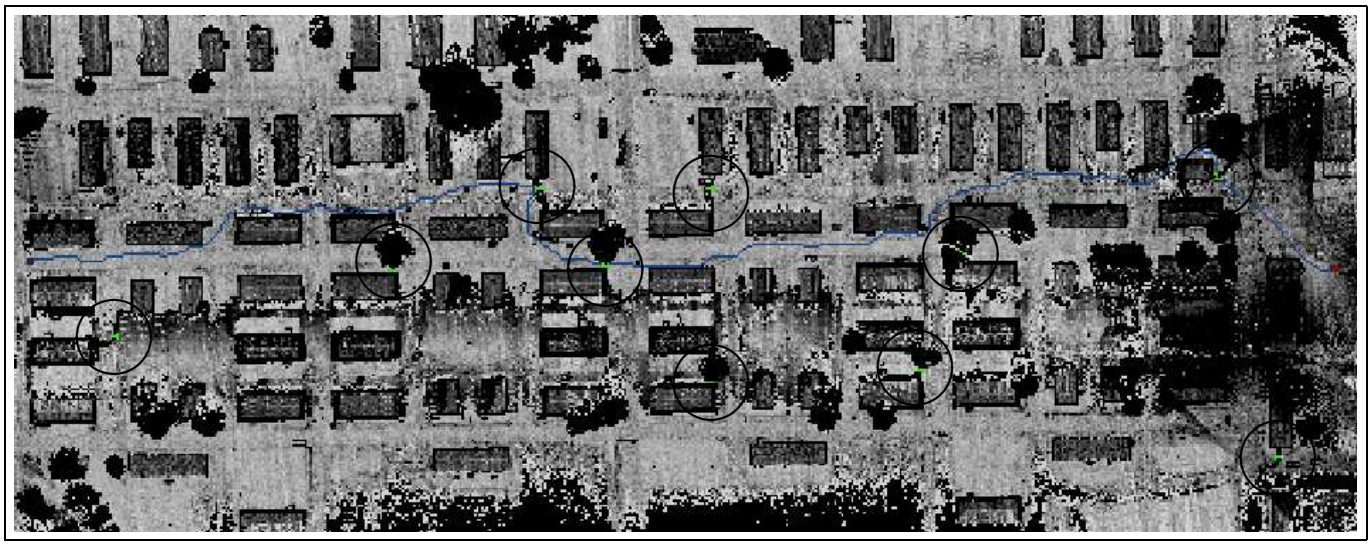

Fig. 9. A simulated traverse (in blue) of the outdoor environment shown in Figure 1.

The effectiveness of each approach is highly dependent on the nature of the environment in which we wish to plan. We have been interested in solving the navigation problem for outdoor environments and generated our range of test scenarios accordingly. However, for different environments, particularly indoor scenarios, the relative performance of the approaches may be a little different. In particular, in run time values the approaches would be even more separated, as only a fraction of the information states are reachable when the order of the adjacency list between faces is small (a typical characteristic of indoor environments), and the inter-dependencies between face costs are even stronger. These changes do not affect the overall performance advantage of PAO*, however, which dominated every criterion in every environment we tested (including some indoor scenarios which have not been reported).

We have included in Figure 9 one possible resulting traverse for an agent planning optimally in the outdoor environment shown earlier. In this particular example, the probabilities associated with each pinch point being untraversable were set to 0.5 and their actual values were generated randomly. The agent started on the left side of the environment (shown in dark blue) and made its way to the goal at the far right (shown in red). It encountered three pinch points, one of which turned out to be untraversable.

V. Conclusion

We have described a new algorithm, PAO*, which applies heuristic search to AND-OR graphs. It is similar to AO* in its maintenance of a partial solution graph but differs in its ability to update heuristic values across the full AND-OR state space. We have presented comparisons between PAO* and three other approaches used to solve a new kind of decision problem, characterized by the incorporation of hidden state.

A number of promising directions exist for future research. In this paper, we have dealt with environments where the pinch points are manually specified. We are currently investigating the automatic extraction of pinch points from outdoor data. We are also looking at how we can replan efficiently when updated information is received concerning the terrain of non-pinch point areas of the environment (as in [1], [2]).

Acknowledgments

The authors would like to express their thanks to Ryan Miller and Omead Amidi for the outdoor terrain data. This work was sponsored by the U.S. Army Research Laboratory, under contract “Robotics Collaborative Technology Alliance” (contract number DAAD19-01-2-0012). The views and conclusions contained in this document are those of the authors and do not represent the official policies or endorsements of the U.S. Government.

References

[1] A. Stentz, “The focussed D* algorithm for real-time replanning,” in *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 1995.

[2] S. Koenig and M. Likhachev, “Incremental A*,” in *Advances in Neural Information Processing Systems*. MIT Press, 2002.

[3] I. Nourbakhsh and M. Genesereth, “Assumptive planning and execution: a simple, working robot architecture,” *Autonomous Robots Journal*, vol. 3, no. 1, pp. 49–67, 1996.

[4] A. Bar-Noy and B. Schieber, “The Canadian Traveller Problem,” in *Proceedings of the second annual ACM-SIAM Symposium on Discrete Algorithms*, 1991, pp. 261 – 270.

[5] C. Chang and J. Slagle, “An admissible and optimal algorithm for searching AND-OR graphs,” *Artificial Intelligence*, vol. 2, pp. 117 – 128, 1971.

[6] N. Nilsson, *Principles of Artificial Intelligence*. Tioga Publishing, 1980.

[7] L. Kaebling, M. Littman, and A. Cassandra, “Planning and acting in partially observable stochastic domains,” *Artificial Intelligence*, vol. 101, 1998.

[8] J. Pineau, G. Gordon, and S. Thrun, “Point-based value iteration: An anytime algorithm for POMDPs,” in *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 2003.

[9] A. Moore and C. Atkeson, “Prioritized sweeping: reinforcement learning with less data and less time,” *Machine Learning*, vol. 13, 1993.

[10] C. Boutilier, R. Brafman, and C. Geib, “Structured reachability analysis for MDPs,” in *Uncertainty in Artificial Intelligence*, 1998.

[11] A. Blum and M. Furst, “Fast planning through graph analysis,” in *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, Montreal, 1995, pp. 1636 – 1642.

[12] E. Rich and K. Knight, *Artificial Intelligence*. McGraw-Hill, 1992.