Probabilistic planning with clear preferences on missing information

Maxim Likhachev [a],[*], Anthony Stentz [b]

[a] *Computer and Information Science, University of Pennsylvania, Philadelphia, PA, USA*
[b] *The Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, USA*

A B S T R A C T

For many real-world problems, environments at the time of planning are only *partially-known*. For example, robots often have to navigate partially-known terrains, planes often have to be scheduled under changing weather conditions, and car route-finders often have to figure out paths with only partial knowledge of traffic congestions. While general decision-theoretic planning that takes into account the uncertainty about the environment is hard to scale to large problems, many such problems exhibit a special property: one can clearly identify beforehand the best (called *clearly preferred*) values for the variables that represent the unknowns in the environment. For example, in the robot navigation problem, it is always preferred to find out that an initially unknown location is traversable rather than not, in the plane scheduling problem, it is always preferred for the weather to remain a good flying weather, and in route-finding problem, it is always preferred for the road of interest to be clear of traffic. It turns out that the existence of the clear preferences can be used to construct an efficient planner, called PPCP (Probabilistic Planning with Clear Preferences), that solves these planning problems by running a series of deterministic low-dimensional A*-like searches.

In this paper, we formally define the notion of clear preferences on missing information, present the PPCP algorithm together with its extensive theoretical analysis, describe several useful extensions and optimizations of the algorithm and demonstrate the usefulness of PPCP on several applications in robotics. The theoretical analysis shows that once converged, the plan returned by PPCP is guaranteed to be optimal under certain conditions. The experimental analysis shows that running a series of fast low-dimensional searches turns out to be much faster than solving the full problem at once since memory requirements are much lower and deterministic searches are orders of magnitude faster than probabilistic planning.

© 2008 Elsevier B.V. All rights reserved.

1. Introduction

A common source of uncertainty in planning problems is lack of full information about the environment. A robot may not know the traversability of the terrain it has to traverse, an air traffic management system may not be able to forecast with certainty future weather conditions, a car route-finder may not be able to predict well future traffic congestions or even be sure about present traffic conditions, a shopping planner may not know whether a particular item will be on sale at one of the stores it considers. Ideally, in all of these situations, to produce a plan, a planner needs to reason over the probability distribution over all the possible instances of the environment. Such planning is known to be hard [1,2].

---

* Corresponding author.
  E-mail address: maximl@seas.upenn.edu (M. Likhachev).

For many of these problems, however, one can clearly name beforehand the "best" values of the variables that represent the unknowns in the environment. We call such values *clearly preferred* values. Thus, in the robot navigation problem, it is always preferred to find out that an initially unknown location is traversable rather than not. In the air traffic management problem it is always preferred to have a good flying weather. In the problem of route planning under partially-known traffic conditions, it is always preferred to find out that there is no traffic on the road of interest. And finally, in the shopping planning example, it is always preferred for a store to hold a sale on the item of interest. These are just few of what we believe to be a large class of planning problems that exhibit clear preferences on missing information. One of the reasons for this is that the knowledge of clear preferences on missing information is not the same as the knowledge of a best action at a state or the value of an optimal policy. Instead, we often know at intuitive level what would be the best event for us (i.e., no traffic congestion, sale, etc.), independently of whether we choose to make use of this event or not. All the other outcomes, on the other hand, are of less preference to us. This intuitive information can be used in planning.

In this paper we present an algorithm called PPCP (Probabilistic Planning with Clear Preferences) that is able to scale up to very large problems by exploiting the fact that these preferences exist. PPCP constructs and refines a plan by running a series of deterministic A*-like searches. Furthermore, by making an approximating assumption that it is not necessary to retain information about the variables whose values were discovered to be clearly preferred values, PPCP keeps the complexity of each search low and independent of the amount of the missing information. Each search is extremely fast, and running a series of fast low-dimensional searches turns out to be much faster than solving the full problem at once since the memory requirements are much lower and deterministic searches can often be many orders of magnitude faster than probabilistic planning techniques. While the assumption PPCP makes does not need to hold for the algorithm to converge, the returned plan is guaranteed to be optimal if the assumption does hold.

The paper is organized as follows. We first briefly go over A* search and explain how it can be used to find least-cost paths in graphs. We then explain how a planning problem changes when some of the information about the environment is missing. In Section 4, we introduce the notion of clear preferences on missing information and briefly talk about the problems that exhibit them. In Section 5, we explain the PPCP algorithm and how it makes use of the clear preferences. The same section gives an extensive theoretical analysis of PPCP that includes the correctness of the algorithm, some complexity results as well as the conditions for the optimality of the plan returned by PPCP. In Section 6 of the paper, we describe two useful extensions of the algorithm such as how one can interleave PPCP planning and execution. In the same section, we also give two optimizations of the algorithm which at least for some problems can speed it up by more than a factor of four. On the experimental side, Section 7 shows how PPCP enabled us to successfully solve the path clearance problem, an important problem in defense robotics. The experimental results in Section 8.1, on the other hand, evaluate the performance of PPCP on the problem of robot navigation in partially-known terrains. They show that in the environments small enough to be solved with methods guaranteed to converge to an optimal solution (such as Real-Time Dynamic Programming [3]), PPCP always returns an optimal policy while being much faster. The results also show that PPCP is able to scale up to large (costmaps of size 500 by 500 cells) environments with thousands of initially unknown locations. The experimental results in Section 8.2, on the other hand, show that PPCP can also solve large instances of path clearance problem and results in substantial benefits over other alternatives. We finally conclude the paper with a short survey of related work, discussion, and conclusions.

2. Backward A* search for planning with complete information

Notations. Let us first consider a planning problem that can be represented as a search for a path in a fully known deterministic graph G. The fact that the graph G is completely known at the time of planning means that there is no missing information about the domain (i.e., environment). We use S to denote a state (a vertex, in the graph terminology) in the graph G. State S_{start} refers to the state of the agent at the time of planning, while state S_{goal} refers to the desired state of the agent. We use $A(S)$ to represent a set of actions available to the agent at state $S \in G$. Each action $a \in A(S)$ corresponds to a transition (i.e., an edge) in the graph G from state S to the successor state denoted by $succ(S, a)$. Each such transition is associated with the cost $c(S, a, succ(S, a))$. The costs need to be bounded from below by a (small) positive constant.

Backward A* search. The goal of shortest path search algorithms such as A* search [4] is to find a path from S_{start} to S_{goal} for which the cumulative cost of the transitions along the path is minimal. The PPCP algorithm we present in this paper is based on running a series of deterministic searches. Each of these searches is a modified backward A* search—the A* search that searches from S_{goal} towards S_{start} by reversing all the edges in the graph. In the following, we therefore briefly describe the operation of a backward A* search.

Suppose for every state $S \in G$ we knew the cost of a least-cost path from S to S_{goal}. Let us denote such cost by $g^*(S)$. Then a least-cost path from S_{start} to S_{goal} can be easily followed by starting at S_{start} and always executing such action $a \in A(S)$ at any state S that $a = \arg\min_{a \in A(S)}(c(S, a, succ(S, a)) + g^*(succ(S, a)))$. Consequently, A* search tries to compute g^*-values. In particular, A* maintains g-values for each state it has visited so far. $g(S)$ is always the cost of the best path found so far from S to S_{goal}. The pseudocode in Fig. 1 gives a simple version of backward A*. In this version, *besta* pointers are used to store the actions that follow the found paths.

The code starts by setting $g(S_{goal})$ to 0 and inserting S_{goal} into *OPEN*. The code then repeatedly selects states from *OPEN* and expands them—executes lines 6 through 9. At any point in time, *OPEN* is a set of states that are candidates for

```
1  g(S_start) = ∞, OPEN = ∅;
2  g(S_goal) = 0, besta(S_goal) = null;
3  insert S_goal into OPEN with the priority equal to g(S_goal) + heur(S_goal);
4  while( g(S_start) > min_{S∈OPEN}(g(S) + heur(S)))
5      remove state S with minimum priority (g(S) + heur(S)) from OPEN;
6      for each action a and S' such that S = succ(S', a)
7          if search hasn't seen S' yet or g(S') > c(S', a, S) + g(S)
8              g(S') = c(S', a, S) + g(S), besta(S') = a;
9              insert S' into OPEN with the priority equal to g(S') + heur(S');
```

Fig. 1. Backward A* search.

expansion. These are also the states from which new paths to S_{goal} have been found but have not been propagated to their predecessors yet. As a result, the expansion of state S involves checking if a path from any predecessor state S' of S can be improved by using the found path from S to S_{goal}, and if so then: (a) setting the g-value of S' to the cost of the new path found; (b) setting action $besta(S')$ to the action a that leads to state S; and (c) inserting S' into *OPEN*. The last operation makes sure that S' will also be considered for expansion and, when expanded, the cost of the found path S' to S_{goal} will be propagated to the predecessors of S'.

The goal of A* is to expand states in such order as to minimize the number of expansions required to guarantee that the states on at least one of the least-cost paths from S_{start} to S_{goal} are expanded. Backward A* expands states in the order of $g(S) + heur(S)$, where *heur*-values estimate the cost of a least-cost path from S_{start} to S. The *heur*-values must never overestimate (i.e., must be admissible), or otherwise A* may return a suboptimal solution. In order for each state not to be expanded more than once, *heur*-values need to be also consistent: $heur(S_{start}) = 0$ and for any two states $S, S' \in G$ such that $S \in succ(S', a)$ for some $a \in A(S')$, $heur(S') + c(S', a, S) \geqslant heur(S)$. If *heur*-values are consistent then every time the search expands a state S, a least-cost path from S to S_{goal} has already been found and therefore a better path will never show up later and the state will never be re-inserted into *OPEN*. Ordering expansions based on the summation of g- and *heur*-values makes the search focus expansions on the states through which the whole path from S_{start} to S_{goal} looks most promising.

The search terminates when $g(S_{start})$—the cost of the best path found so far from S_{start} to S_{goal}—is at least as small as the smallest summation of g and *heur* values in *OPEN*. Consequently, *OPEN* no longer contains states that belong to the paths with smaller costs than $g(S_{start})$. This means that A* can terminate and guarantee that the found path is optimal. The proof of this guarantee relies in one way or another on the fact that the g^*-values of the states on an optimal path are monotonically decreasing: if an optimal path from S_{start} to S_{goal} contains a transition $S \to S'$ via some action a, then $g^*(S) > g^*(S')$. This monotonicity property will show up later in the paper. In particular, while in a general case optimal plans in domains with missing information do not necessarily exhibit monotonicity of state values, we will show that in case of clear preferences, the state values on optimal plans are indeed monotonic in some sense. This will allow us to use a series of backward A* searches to plan.

Example. To make later explanations clearer, let us consider a trivial planning problem shown in Fig. 2(a). Suppose an agent needs to buy wine and cheese, and there are two stores, store A and store B. Both stores have both products but at different prices as shown in the figure. Initially, the agent is at home and the cost of traveling from home to each store and in between stores can also be translated into money (all the costs are shown in Fig. 2(a)). The planning problem is for the agent to purchase wine and cheese with the minimal cost (including the cost of travel) and return home.

Fig. 2(b) shows how this problem can be represented as a graph. Each state encodes the position of the agent and what it has already bought. Thus, S_{start} is {*Agent = Home, Bought = ∅*} and S_{goal} is {*Agent = Home, Bought = wine, cheese*}. Fig. 2(c) shows g-values, heuristics and priorities $f = g + heur$ of states as computed by backward A* search that was used to find a least-cost path. The found path is shown by thicker lines. The states expanded by A* are shown in grey. For each state S, the heuristic $heur(S)$ is the cost of moving from home to the store the agent is in at state S plus the cost of purchasing the items that are bought at state S assuming the price is the minimum possible price across both stores (remember that the search is backward and therefore the heuristics estimate the cost of a least-cost path from start state to state in question). Thus, an optimal plan for the agent is to go to store A, buy cheese there, go to store B, buy wine there and then return home.

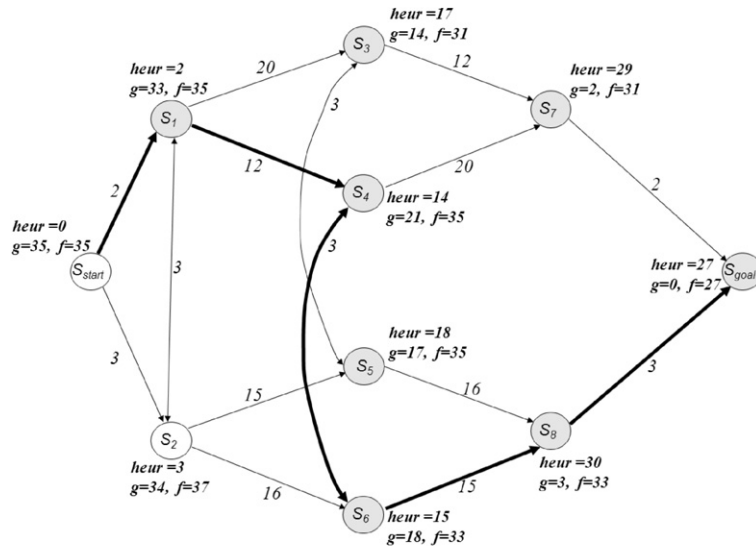## 3. Planning with missing information

In the example above, the graph G that represents the planning problem and all of its edge costs were fully known. By planning with missing information, on the other hand, we refer to the case when the outcomes of some actions and/or some edge costs are not known at the time of planning. In particular, there are some *hidden variables* whose status affects the outcomes and/or costs of certain actions. The status of these hidden variables is unknown to the agent at the time of planning. Instead, the agent has probability distribution (belief) over the possible values of these hidden variables. During execution, however, the agent may sense one or more of these hidden variables at certain states. Once it senses them, the actual values of these hidden variables become known. By sensing we refer to any action that results in knowing a hidden variable. Sometimes, it is an explicit sensing action such as seeing if a region in front of the robot is traversable. In other

(a) shopping example
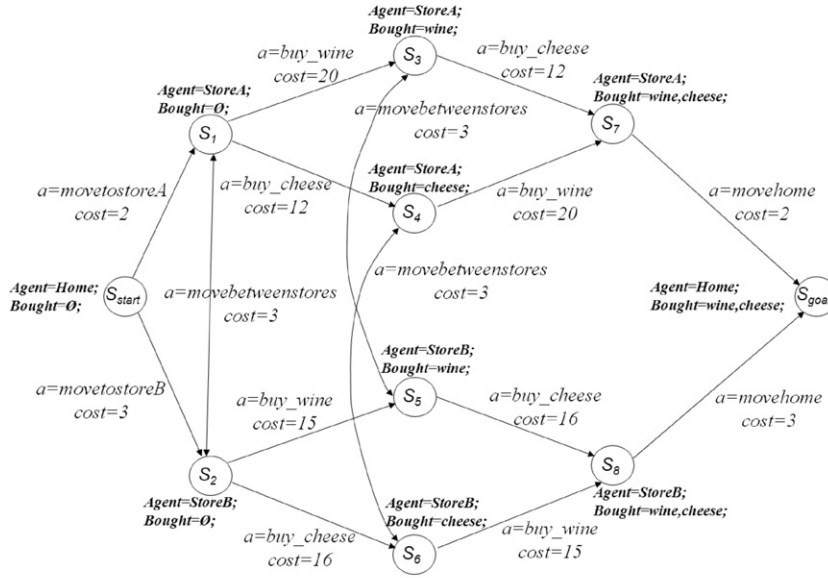
(b) corresponding graph

(c) state values after A* search and the path it finds

Fig. 2. Simple example of planning with complete information.

cases, the value of a hidden variable can be deduced from an outcome of an action such as trying to pick up an object and realizing that it is too heavy to be picked up with a single robot arm. In either case, we assume that the value of the hidden variable that controls the outcome of an action becomes known after the action is executed. In terms of explicit sensing, this corresponds to assuming perfect sensing.

For example, consider a variation of the grocery shopping problem described above. The variation, shown in Fig. 3(a), is that store A may conduct a 50% sale on wine products, whereas store B may conduct a 50% sale on cheese products. The agent does not know whether either of the sales is actually happening but estimates the probability of having a sale at store A to be 40% and the probability of having a sale at store B to be 80%. This modified version of the problem corresponds to the problem of planning with missing information, in which the underlying state now includes two additional boolean variables, each indicating whether there is a sale at the corresponding store. Let us denote these variables by $Sale_A$ and $Sale_B$. We will use $Sale_A = 1$ (0) to mean that store A has (doesn't have) a sale on wine and similar notation for $Sale_B$. The problem differs from the original deterministic planning because the agent does not know the status of variables $Sale_A$ and $Sale_B$ until it visits stores A and B respectively. At the same time, the problem is related but much narrower than planning for Partially Observable Markov Decision Problems (POMDPs) due to the following two assumptions we make:

- **Perfect sensing.** There is no noise in sensing: once the agent visits the store, it knows whether there is a sale or not.
- **Deterministic actions.** There is no uncertainty in the outcomes of actions if values of hidden variables are known: the agent moves deterministically and the cost of each purchase is known if the status of $Sale_A$ and $Sale_B$ variables is known.

Despite these restrictions, we explain the problem of planning with missing information using the notation and terminology of POMDPs, similar to how it was done in [5].

Belief MDP formulation. In POMDPs, an agent does not know its full state. Instead, it has a probability distribution over the possible states it can be in. A belief state is any particular value of this distribution (the dimensionality of the distribution can be as high as $N - 1$, where N is the number of possible states in the original graph G). Let us denote belief states by X. For instance, in the grocery shopping example, the initial belief state of the agent, X_{start}, is: $\{Agent = Home, Bought = \emptyset, P(Sale_A = 1) = 0.4, P(Sale_B = 1) = 0.8\}$. Note that this concisely represents a probability distribution over all possible states since the position of the agent and what items it has already purchased is always known and the probability distribution over possible store sales can be represented as the probability of a sale at each store assuming store sales are independent events.

Fig. 3(b) shows the corresponding belief state-space. The notation $Sale_A = u$ and $Sale_B = u$ represents $P(Sale_A = 1) = 0.4$ and $P(Sale_B = 1) = 0.8$ respectively. The notation $Sale_A = 0$ ($Sale_A = 1$), on the other hand, represents the knowledge of the agent that there is no sale (there is sale) at store A. In other words, $Sale_A = 0$ is equivalent to the belief $P(Sale_A = 1) = 0$. Similar notation is used for $Sale_B$.

The belief state-space is a Markov Decision Process (MDP). After every action, the agent knows precisely the belief state it is in. Some actions, however, are probabilistic. The outcomes of these actions depend on the actual status of the corresponding hidden variables. Essentially, these are the actions that incorporate sensing of hidden variables. The probability of the outcomes of these actions thus follows the probability distribution over the hidden variables that are being sensed. Because we assume that sensing is perfect, the subgraphs that result from different outcomes of a single stochastic action are disjoint. This also implies that an optimal policy in such a belief state-space is acyclic.

Let us now explicitly split X into two sets of (discrete and of finite-range) variables, $S(X)$ and $H(X)$: $X = [S(X); H(X)]$.

- $S(X)$ is the set of variables whose values are always observed. These are the variables that define the state in the original graph G (i.e., Fig. 2). $S(X)$ can also be thought of as the projection of X onto the state-space defined by the completely observed variables.
- $H(X)$ is the set of hidden variables that represent the missing information about the environment.

So, in our example, $S(X)$ is the location of the agent and what the agent has purchased so far, and $H(X)$ is the status of variables $Sale_A$ and $Sale_B$. We will use $h_i(X)$ to denote an ith variable in $H(X)$.

The goal of the planner is to construct a policy that reaches any state X such that $S(X) = S_{goal}$ (i.e., $\{Agent = Home, Bought = wine, cheese\}$) while minimizing the expected cost of execution. Fig. 3(c) shows this policy for our example. A full policy is more than a single path since some actions are probabilistic and a full policy dictates to the agent what to do at any state it may end up in during execution. According to the policy, the agent will visit store A, and then, if there is no sale at store A, it will go to store B, and it may even return to store A again, if there is no sale at store B either. This is very different from a single path found in case of planning with complete information (Fig. 2(c)).

Finding a policy of good quality is difficult for two reasons: first, a belief state-space is probabilistic and therefore deterministic planners such as A* do not typically apply; second, and perhaps more importantly, the size of a belief state-space is exponential in the number of hidden variables. More specifically, given that $X = [S(X), H(X)]$, the size of a belief state-space is roughly the number of states in the original graph G times the number of possible beliefs over hidden variables. In our example, the latter is 3^2 since there are three possible beliefs—unknown, 1 and 0—about each of the two hidden variables (i.e., $Sale_A$ and $Sale_B$).

(a) shopping example
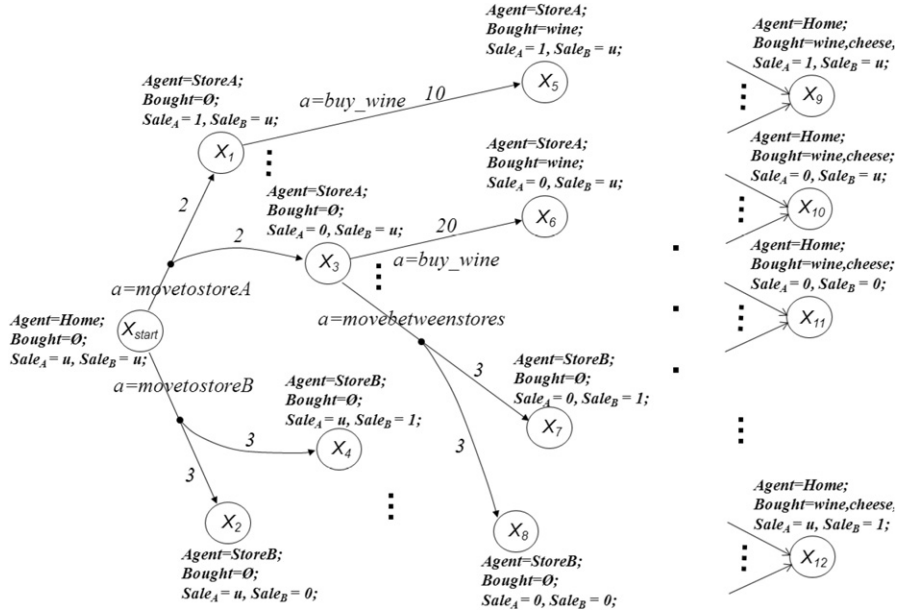
(b) corresponding belief state-space
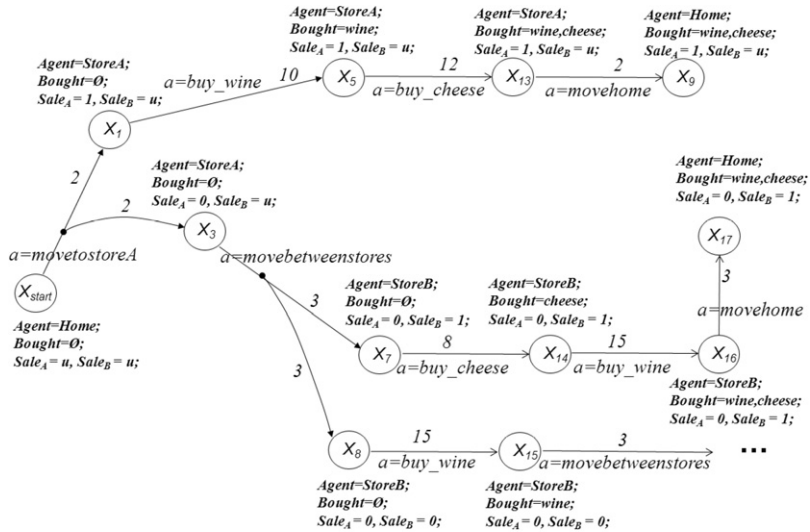
(c) optimal policy

Fig. 3. An example of planning with missing (incomplete) information.

4. Clear preferences on missing information

In this section we introduce the notion of clear preferences. This notion is central to the idea behind PPCP. We first, however, define several other notations and assumptions. For the sake of simplicity, the notation $h_i(X) = u$ at state X will represent the fact that the value of h_i is unknown at X. If $h_i(X) \neq u$, on the other hand, then the actual value of h_i is known at X (since sensing is perfect).

Assuming at most one hidden variable per action. We assume that any time there is an action with uncertainty in outcomes or uncertainty in costs (i.e., an action that involves sensing a hidden variable), the uncertainty is due to a single hidden variable. Thus, the execution of any single action can result in deducing the value of at most one hidden variable. (In some domains, in case two or more factors control the outcome of an action, they can be combined into a single hidden variable.) We do allow, however, for a single hidden variable to control more than one action though. For example, in our example above, there could be a single hidden variable *Sale*, which, if 1, would imply a sale at both stores.

Denoting hidden variables. We use $h^{S(X),a}$ to represent the hidden variable that controls the outcomes and costs of action a taken at $S(X)$. By $h^{S(X),a} = $ **null** we denote the case when there was never any uncertainty about the outcome of action a taken at state X. In other words, none of the variables in H control the outcomes of a executed at $S(X)$, and since the underlying environment is deterministic, there is only one outcome. Thus, in the above example, the action a of moving from store A to home has $h^{Agent=StoreA,Bought=wine,cheese,a=move} = $ **null** since its outcome does not depend on the values of any of the hidden variables. The action a of moving to store A from home, on the other hand, has $h^{Agent=Home,Bought=\emptyset,a=movetostoreA} = u$ since once the agent enters the store A it finds out whether there is a sale or not. Therefore, the action a executed at $X_{start} = [Agent = Home, Bought = \emptyset, Sale_A = u, Sale_B = u]$ has two possible outcomes: $X_1 = [Agent = StoreA, Bought = \emptyset, Sale_A = 1, Sale_B = u]$ and $X_3 = [Agent = StoreA, Bought = \emptyset, Sale_A = 0, Sale_B = u]$.

Denoting successors. Sometimes, we will also need to refer to the set of successors in the belief state-space. In these cases we will use the notation $succ(X, a)$ to denote the set of belief states Y such that $S(Y) \in succ(S(X), a)$ and $H(Y)$ is the same as $H(X)$ except for $h^{(S(X),a)}(Y)$ which becomes known if it was unknown at X and remains the same otherwise. The function $P_{X,a}(Y)$, the probability distribution of outcomes of action a executed at state X, follows the probability distribution of $h^{S(X),a}$, $P(h^{S(X),a})$. As mentioned above, once action a was executed at state X the actual value of $h^{S(X),a}$ can be deduced since we assumed the sensing is perfect and the environment is deterministic. Thus, in our example, for $X_{start} = [Agent = Home, Bought = \emptyset, Sale_A = u, Sale_B = u]$, $a = movetostoreA$, and $X_1 = [Agent = StoreA, Bought = \emptyset, Sale_A = 1, Sale_B = u]$, $P_{X_{start},a}(X_1) = 0.4$.

Assuming independence of hidden variables. PPCP also assumes that the variables in H can be considered independent of each other. In other words, the sale event at store A is independent of the sale event at store B.

Assuming clear preferences. The main assumption PPCP makes is that clear preferences on the values of the hidden variables are available. It requires that for each variable $h_i \in H$, it is given its preferred value, denoted by b (i.e., best). This value is defined as follows.

Definition 1. A clearly preferred value b for a hidden variable $h^{S(X),a}$ is such a value that given any state X and any action a such that $h^{S(X),a}$ is not known (that is, $h^{S(X),a}(X) = u$), there exists a successor state X' which has $h^{S(X),a}(X') = b$ and satisfies the following:

$$X' = \underset{Y \in succ(X,a)}{\operatorname{argmin}} \ c(S(X), a, S(Y)) + v^*(Y)$$

where $v^*(Y)$ is the expected cost of executing an optimal policy at state Y.

We will use the notation $succ(X, a)^b$ (i.e., the best successor) to denote the state X' whose $h^{S(X),a}(X') = b$ if $h^{S(X),a}(X) = u$ and whose $h^{S(X),a}(X') = h^{S(X),a}(X)$ otherwise. (In the latter case it may even be possible that $h^{S(X),a}(X') \neq b$ if $h^{S(X),a}(X) \neq u$. There were simply no other outcomes of action a executed at X.)

Our grocery shopping example satisfies this property since for each sensing action there always exist two outcomes: a sale is present or not and the former one is the preferred outcome. Thus, as shown in Fig. 4, for action $a = movetostoreA$ executed at $X_{start} = [Agent = Home, Bought = \emptyset, Sale_A = u, Sale_B = u]$, the preferred outcome $succ(X, a)^b$ is $X_1 = [Agent = $
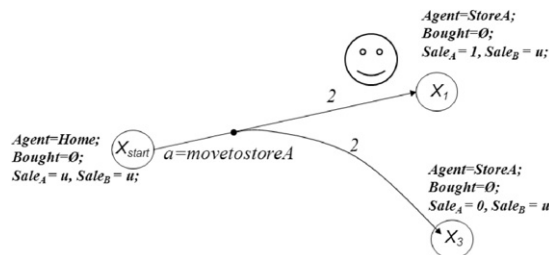

Fig. 4. An example of clear preferences: it is always better (or at least no worse) to have a sale.

StoreA, *Bought* = ∅, *Sale$_A$* = 1, *Sale$_B$* = *u*]. It is trivial to show this. If X_3 is the second outcome of action *a* (the one that corresponds to *Sale$_A$* = 0 outcome), then $c(S(X_{\text{start}}), a, S(X_1)) = c(S(X_{\text{start}}), a, S(X_3))$, and $v^*(X_1) \leqslant v^*(X_3)$ since X_1 and X_3 are exactly the same states with the only difference that X_1 has a sale event at store A and X_3 does not.

We believe that there is a wide range of problems that exhibit clear preferences. People can often predict the outcomes that optimize their costs-to-goal. For example, when planning a car route, a person would clearly prefer for any single road to be free of traffic. Similarly, when choosing a sequence of connecting flights, a person would clearly prefer for the weather to be a good flying weather and for the planes to have no delays. These preferences can be determined without computing optimal values. One of the frequent reasons for this is the fact that a particular value of a hidden variable does not commit the agent to any particular action. The agent is still free to choose any route or any sequence of planes. The only requirement is that an optimal plan in case of no traffic, a good weather, and no plane delays can not be worse than the corresponding optimal plan in case of a traffic, bad weather, and plane delays.

Obviously, there is also a wide range of problems that do *not* exhibit clear preferences. For example, in case of a long flight delay, there may be a non-zero probability that the air carrier will provide customers with some sort of compensation (i.e., first-class upgrade or a free ticket). In this case, it may be less clear to a person whether he/she prefers to have a flight on-time or delayed. The actual preference depends on other penalties that will be incurred by the person if the flight is delayed (e.g., missing connecting flights, dinners, etc.) and can therefore be computed only after the expected costs of optimal plans for both outcomes are computed. Apart from the compensation and other non-obvious factors though, typically, it is clearly preferred to have a non-delayed flight: a person flying a non-delayed flight has always a freedom to stay at the arrival airport for the length of the possible delay if it is more optimal to leave the airport later. Clear preferences just capture the fact that the cost of an optimal plan with a non-delayed flight can not be worse than the cost of an optimal plan with a delayed flight. These preferences are often known without computing the costs of actual optimal plans.

5. The PPCP algorithm

The explanation of the algorithm can be split into two steps. In the first section we present a planner that constructs a provably optimal policy using a series of A*-like searches in the full belief state-space. It gains efficiency because each search is a deterministic search and can use heuristics to focus its efforts. Nevertheless, each search is still very expensive since the size of the belief state-space is exponential in the number of hidden variables. In the following section we show how the planner can be extended to use a series of searches in the underlying environment instead. These resulting searches are exponentially faster and are independent of the number of hidden variables. The overall solution though, is guaranteed to be optimal only under certain conditions.

5.1. Optimal planning via repeated searches

The algorithm works by executing a series of deterministic searches. We will first describe how the deterministic search works and then we will show how the main function of the algorithm uses these searches to construct the first policy and then refine it.

The function that does the deterministic search is called ComputePath and is shown in Fig. 5. The search is done in the belief state-space. It is very similar to (backward) A*. It also computes *g*-values of states, uses heuristic values to focus its search and performs repeated expansions of states in the order of *g*-value plus heuristic (known as *f*-values). The meaning of the *g*-values and the criterion that the solution minimizes, however, are somewhat different from the ones in A* search.

Suppose that for each state *X* we have an estimate $v(X)$ of $v^*(X)$, the minimum expected cost of reaching a goal from the state. These estimates are provided to the search by the function Main() of the algorithm (the estimates are always non-negative). Then, every state-action pair $X', a \in A(S(X'))$ has a value $Q_{v,g}(X', a) > 0$ associated with it. It is calculated using action costs $c(S(X'), a, S(Y))$, value estimates $v(Y)$ for the outcome states $Y \in succ(X', a)$, and the *g*-value of the preferred outcome state $Y^b = succ(X', a)^b$. $Q_{v,g}(X', a)$ is defined as follows:

$$Q_{v,g}(X', a) = \sum_{Y \in succ(X', a)} P_{X', a}(Y) \cdot \max\big(c\big(S(X'), a, S(Y)\big) + v(Y), c\big(S(X'), a, S(Y^b)\big) + g(Y^b)\big) \tag{1}$$

To understand the meaning of the above equation, consider first the standard definition of an undiscounted *Q*-value of an action [6] (in terms of costs, rather than rewards though):

$$Q(X', a) = \sum_{Y \in succ(X', a)} P_{X', a}(Y) \cdot \big(c\big(S(X'), a, S(Y)\big) + v(Y)\big) \tag{2}$$

In other words, a *Q*-value is the expectation of the sum of the immediate cost plus the value of an outcome over all possible outcomes of action *a* executed at state X'. If *v*-values are perfect estimates (i.e., equal to corresponding v^*-values), then $Q(X', a)$ gives the expected cost of an optimal policy that starts at state X' with the execution of action *a*. If the *v*-values are under-estimates of v^*-values, however, then the computed $Q(X', a)$ will also be an under-estimate. Now consider the definition of clear preferences (Definition 1). According to it, $c(S(X'), a, S(Y^b)) + v^*(Y^b) \leqslant c(S(X'), a, S(Y)) + v^*(Y)$ for any successor $Y \in succ(X', a)$. This also implies that $c(S(X'), a, S(Y^b)) + v(Y^b) \leqslant c(S(X'), a, S(Y)) + v(Y)$ since *v*-values

are under-estimates and therefore $v(Y^b) \leqslant v^*(Y^b)$. Consequently, if our current v-values are imperfect, we can (potentially) improve the Q-value estimate in Eq. (2) using the v-value of the preferred outcome:

$$Q_{v,v}(X',a) = \sum_{Y \in succ(X',a)} P_{X',a}(Y) \cdot \max\left(c\big(S(X'), a, S(Y)\big) + v(Y), c\big(S(X'), a, S(Y^b)\big) + v(Y^b)\right) \tag{3}$$

Finally, the ComputePath function computes g-values of some states. The property that it guarantees is that any computed g-value is also an under-estimate of the corresponding v^*-value. These g-values, however, are improvements over the previous v-values (one can think of them as state values after a series of specially ordered backups). Thus, instead of $v(Y^b)$, the ComputePath function uses the newly computed g-value, $g(Y^b)$. This is exactly Eq. (1).

It should now be straightforward to see that if the provided v-values are equal to the corresponding v^*-values and $g(Y^b)$ is equal to $v^*(Y^b)$, then Eqs. (1) and (2) are identical. As a result, the plan that has the minimum expected cost of reaching the goal would then be given by a simple strategy of always picking an action with the smallest $Q_{v,g}(X,a)$ at any current state X.

In reality, v-values may not necessarily be equal to the corresponding v^*-values at first. Nevertheless, the search computes g-values based on $Q_{v,g}$-values. In particular, let us define g^*-values as the solution to the following fixpoint equation:

$$g^*(X) = \begin{cases} 0 & \text{if } S(X) = S_{\text{goal}} \\ \min_{a \in A(S(X))} Q_{v,g^*}(X,a) & \text{otherwise} \end{cases} \tag{4}$$

These g^*-values are the expected costs of optimal plans under the assumption that the v-values of the non-preferred outcomes are also the expected costs of optimal plans (in other words, the v-values of non-preferred outcomes are assumed to be perfect estimates). The g-values computed by the search in the ComputePath function are estimates of these g^*-values. In fact, it can be shown that the g-values of the states expanded by the search are exactly equal to the corresponding g^*-values (Theorem 3).

Also, because of the max operator in Eq. (1) and the fact that all costs are positive, finite $Q_{v,g^*}(X',a)$ is always strictly larger than $g^*(succ(X',a)^b)$, independently of whether v-values are correct estimates or not. This means that g^*-values along optimal (finite-cost) paths are monotonically decreasing: $g^*(X') = Q_{v,g^*}(X',a') > g^*(succ(X',a')^b)$, where $a' = \arg\min_{a \in A(S(X'))} Q_{v,g^*}(X',a)$. This monotonicity allows us to perform a deterministic (A*-like) search which computes g-values, and sets the g-values of relevant states to their corresponding g^*-values.

The ComputePath function, shown in Fig. 5, searches backwards in the belief state-space from goal states towards the state X_p on which it was called. (It is important to remember that the number of goal states in the belief state-space is exponential in the number of hidden variables, since a goal state is any state X whose $S(X) = S_{\text{goal}}$.) The trajectory the

---

Initially, v-values of states need to be non-negative and smaller than or equal to the costs of least-cost trajectories to a goal under the assumption that all hidden variables are set to their clearly preferred values.

```
 1 procedure ComputePath(Xₚ)
 2 g(Xₚ) = ∞, OPEN = ∅;
 3 for every H whose every element hᵢ satisfies:
      [(hᵢ = u ⋁ hᵢ = b) ⋀ hᵢ(Xₚ) = u] OR [hᵢ = hᵢ(Xₚ) ⋀ hᵢ(Xₚ) ≠ u]
 4   X = [S_goal; H];
 5   g(X) = 0, besta(X) = null;
 6   insert X into OPEN with g(X) + heur(Xₚ, X);
 7 while(g(Xₚ) > min_{X'∈OPEN} g(X') + heur(Xₚ, X'))
 8   remove X with smallest g(X) + heur(Xₚ, X) from OPEN;
 9   for each action a and state X' s.t. X ∈ succ(X', a)
10     compute Q_{v,g}(X', a) according to formula 1;
11     if this search hasn't seen X' yet or g(X') > Q_{v,g}(X', a)
12       g(X') = Q_{v,g}(X', a), besta(X') = a;
13       insert/update X' in OPEN with the priority g(X') + heur(Xₚ, X');
14 procedure UpdateMDP(X_pivot)
15 X = X_pivot;
16 while (S(X) ≠ S_goal)
17   v(X) = g(X);
18   X = succ(X, besta(X))ᵇ;
19 procedure Main()
20 X_pivot = X_start;
21 while (X_pivot != null)
22   ComputePath(X_pivot);
23   UpdateMDP(X_pivot);
24   find state X on the current policy such that S(X) ≠ S_goal and it has
        v(X) < E_{X'∈succ(X,besta(X))} c(S(X), besta(X), S(X')) + v(X');
25   if found set X_pivot to X;
26   otherwise set X_pivot to null;
```

Fig. 5. Optimal planning via repeated searches.

search returns uses only the transitions that correspond to either deterministic actions or preferred outcomes of stochastic actions. This is implemented by starting off the search with all and only those goal states, whose hidden variables assume unknown or preferred values if they are also unknown in $H(X_\mathrm{p})$ and values equal to the corresponding hidden variables in $H(X_\mathrm{p})$ otherwise (lines 3–6). The first time ComputePath is called, X_p is X_start and therefore all the hidden variables are unknown. For the subsequent calls, however, X_p can be a different state, and the values of some of its hidden variables can be known.

Just like (backward) A* search, the ComputePath function expands states in the order of g plus heuristic and during each expansion (lines 8–13) updates the g-value and *besta* pointer of each predecessor state of the expanded state. (The ComputePath function does not retain g-values in between searches. The states that are encountered for the first time within a specific call to the function have their g-values reset anyway on lines 11–12.) It differs from A* though in that g-values are computed according to formula 1. In fact, it is the computation of this formula that requires the ComputePath function to search backwards (and not forwards). Heuristics are used to focus the search. Since the search is backward, they estimate the cost of following a least-cost trajectory from X_p to state in question. In the pseudocode, a user-provided function $heur(X_\mathrm{p}, X)$ returns a heuristic value for state X. These values need to be consistent in the following sense: $heur(X_\mathrm{p}, X_\mathrm{p}) = 0$ and for every other state X and action $a \in A(S(X))$, $heur(X_\mathrm{p}, X) + c(S(X), a, S(succ(X, a)^b)) \geqslant heur(X_\mathrm{p}, succ(X, a)^b)$. This reduces to normal consistency requirement on heuristics used by a backward A* if the state-space is fully deterministic, that is, no information is missing at the time of planning (Section 2). For instance, for our grocery shopping example it could be the same heuristics as the one used for planning with complete information in Section 2 except that in computing the heuristics we would use the sale prices. This is equivalent to computing heuristics under the assumption that all hidden variables are known to have preferred values.

The search finishes as soon as the g-value of X_p is no larger than the smallest priority of states in *OPEN*. (A min operator over empty set is assumed to return infinity. The same assumption was made about the expectation operator on line 24.) Once the ComputePath function exits the following holds for the path from X_p to a goal state constructed by always picking action $besta(X)$ at any state X and then moving into the state $succ(X, besta(X))^b$ if $besta(X)$ has more than one outcome: the g-value of every state on the trajectory is equal to the g^*-value of the same state.

The Main function of the algorithm (shown in Fig. 5) uses this fact. Starting with X_start, it repeatedly executes searches on states that reside on the current policy (defined by *besta* pointers) and whose v-values are smaller than what they should be according to the v-values of the successors of the policy action (line 24). The initial v-values need to be non-negative and smaller than or equal to the costs of least-cost trajectories to a goal under the assumption that all hidden variables are equal to b (a simple initialization to zero suffices). In particular, these values can be set to heuristics if these are admissible with respect to the underlying graph generated when values of all hidden variables are set to their clearly preferred values. The Main function terminates when no state on the current policy has its v-value smaller than what it should be according to the v-values of its successors.

After each search, the UpdateMDP function iterates over the found path (lines 16–18) and updates the v-values of the states on the path found by the search by setting them to their g-values (line 17). As mentioned above, these are equal to their corresponding g^*-values. (The v-values of states are retained until the termination of the algorithm.) On one hand, this increases v-values and is guaranteed to correct the error between the v-values of these states and the v-values of their successors in the policy. On the other hand, g^*-values are bounded from above by v^*-values as long as v-values do not overestimate v^*-values. As a result, the algorithm converges, and at that time, the states on the found policy have their v-values equal to their v^*-values and the found policy itself is optimal (the proof of this and other theorems can be found in [7]):

Theorem 1. *The Main function in Fig. 5 terminates and at that time the expected cost of the policy defined by besta pointers is given by $v(X_\mathrm{start})$ and is equal to the minimum expected cost of reaching a goal from X_start.*

The algorithm remains correct independently of how states satisfying the condition on line 24 are selected. Typically however, out of all the states satisfying the condition, it is most beneficial to always select first the state that has the highest probability of being reached, as it is likely to influence the cost of the policy the most. The probabilities of reaching states on the policy can be computed in a single pass over the states on the policy starting from the start state. In addition, it is even more efficient to select the states using the following simple optimization demonstrated in Section 5.3: whenever a state X is chosen as the next X_pivot, we can backtrack from X up along the policy until we encounter the first stochastic transition (or X_start, whichever comes first), at which point X_pivot is chosen to be the outcome of that transition, the transition that resides on the same branch as X.

5.2. Scaling up searches

Each search in the version of the algorithm just presented can be very slow because it operates in the belief state-space whose size is exponential in the number of hidden variables. We now describe the actual version of PPCP that addresses this inefficiency. At a high level, the main idea is to forget the outcomes of sensing within each particular search (not within the overall planning though!). This means that within a particular search the values of hidden variables are not being tracked—they remain the same. Consequently, each search can be performed in the original graph G (i.e., underlying environment)

Initially, v-values of states need to be non-negative and smaller than or equal to the costs of least-cost trajectories to a goal under the assumption that all hidden variables are set to their clearly preferred values.

```
 1 procedure ComputePath(X_p)
 2 g(S(X_p)) = ∞, OPEN = ∅;
 3 g(S_goal) = 0, besta(S_goal) = null;
 4 insert S_goal into OPEN with g(S_goal) + heur(S(X_p), S_goal);
 5 while(g(S(X_p)) > min_{S(X)∈OPEN} g(S(X)) + heur(S(X_p), S(X)))
 6   remove S(X) with smallest g(S(X)) + heur(S(X_p), S(X)) from OPEN;
 7   for each action a and S(X') s.t. S(X) = S(succ([S(X'); H^u(X_p)], a)^b)
 8     compute Q̃_{v,g}(S(X'), a) according to formula 5;
 9     if this search hasn't seen S(X') yet or g(S(X')) > Q̃_{v,g}(S(X'), a)
10       g(S(X')) = Q̃_{v,g}(S(X'), a), besta(S(X')) = a;
11       insert/update S(X') in OPEN with the priority g(S(X')) + heur(S(X_p), S(X'));

12 procedure UpdateMDP(X_pivot)
13 X = X_pivot;
14 while (S(X) ≠ S_goal)
15   v(X) = g(S(X)), v(X^u) = g(S(X)), besta(X) = besta(S(X));
16   X = succ(X, besta(X))^b;
17 procedure Main()
18 X_pivot = X_start;
19 while (X_pivot != null)
20   ComputePath(X_pivot);
21   UpdateMDP(X_pivot);
22   find state X on the current policy such that S(X) ≠ S_goal and it has
        v(X) < E_{X'∈succ(X,besta(X))} c(S(X), besta(X), S(X')) + v(X');
23   if found set X_pivot to X;
24   otherwise set X_pivot to null;
```

Fig. 6. PPCP: Planning via repeated efficient searches.

with costs and outcomes modified to reflect the initial settings of the hidden variables. This graph is exponentially smaller than the full belief state-space. In other words, a search state consists of $S(X)$ variables only and the size of the search state-space is therefore independent of the amount of missing information. This results in a drastic increase in efficiency of searches and the ability to solve much bigger problems without running out of memory. This efficiency, however, comes at the expense of optimality, which can only be guaranteed under the conditions described later in Theorem 7. In brief, this theorem states that optimality guarantees require that no branch of an optimal policy executes two actions that rely on the same hidden variable whose value is a clearly preferred value. In other words, when executing optimal policy, it is not necessary to retain information about the variables whose values were discovered to be clearly preferred values.

Suppose the agent executes some action a whose outcome it is uncertain about at a belief state X. Suppose also that the execution puts the agent into $succ(X, a)^b$. This means that the agent can deduce the fact that the value of the hidden variable $h^{S(X),a}$ that represents the missing information about action a is b. During each search, however, we assume that in the future the agent will not need to execute action a or any other action whose outcome is dependent on the value of this hidden variable (remember that the value of a single hidden variable is allowed to control the outcomes of more than one action). In case it does need to execute such action again, the search assumes that the value of the hidden variable is unknown again. As a result, the search does not need to remember whether $h^{S(X),a}$ is unknown or known to be equal to b. In fact, whenever the search needs to compute a $Q_{v,g}$-value of a stochastic action, it assumes that the values of all hidden variables are unknown unless they were known to have non-preferred values in the belief state X_p, the state the ComputePath function was called on. Under this assumption, the calculation of $Q_{v,g}(X)$-value (Eq. (1)) becomes independent of $H(X)$ since any hidden variable in $H(X)$ whose value is different from the same variable in $H(X_p)$ can only be equal to b, and these are replaced by u. Thus, each search can be done in the original graph G rather than the exponentially larger belief state-space. The search no longer needs to keep track of $H(\cdot)$ part of the states, it operates directly on $S(\cdot)$ states.

The ComputePath function, shown in Fig. 6 performs this search. While the function is called on a belief state X_p, it searches backwards from a single state, S_{goal}, towards a single state $S(X_p)$, and the states in the search state-space consist only of variables in S. The search assumes that each action a has only one outcome. It assumes that $S(X)$ is an outcome of action a executed at $S(X')$ if and only if $S(X) = S(succ([S(X'); H^u(X_p)], a)^b)$ (line 7), where $H^u(X_p)$ is defined as $H(X_p)$ but with each hidden variable equal to b set to u. This corresponds to setting up a deterministic environment in which each action a executed at $S(X')$ has a single outcome corresponding to the value of the hidden variable $h^{S(X'),a}$ in $H(X_p)$ if it is known there and to the preferred value of $h^{S(X'),a}$ if it is unknown in $H(X_p)$. The heuristics need to be consistent with respect to this environment we have just set up. The ComputePath function performs backward A* search in this state-space with the exception of how g-values are computed.

To implement the assumption that the agent has no memory for the preferred values of the hidden variables that were previously observed we need to compute $Q_{v,g}$-values appropriately. For any state-action pair $(S(X), a)$, $a \in A(S(X))$ the ComputePath function now computes g-values based on $Q̃_{v,g}(S(X), a)$ instead. Let X^u denote a belief state $[S(X); H^u(X_p)]$. We then define $Q̃_{v,g}(S(X), a)$ as:

$$\tilde{Q}_{v,g}\big(S(X), a\big) = Q_{v,g}\big(X^u, a\big), \quad \text{where } g\big(succ(X^u, a)^b\big) = g\big(S\big(succ(X^u, a)^b\big)\big) \tag{5}$$

According to this formula, during the computation of the Q-value of action a we assume that we execute this action at a belief state X^u, at which we are unaware of any hidden variables with preferred values. In the calculation of $Q_{v,g}(X^u, a)$ (Eq. (1)) we then use v-values of the corresponding belief states. The calculation of $Q_{v,g}(X^u, a)$ also requires the g-value of $succ(X^u, a)^b$. The search does not compute g-values for full belief states. Instead, $g(succ(X^u, a)^b)$ is substituted with $g(S(succ(X^u, a)^b))$. This computation of $\tilde{Q}_{v,g}$ implements the assumption that the agent does not remember the values of the hidden variables whenever they are detected as b.

For example, in the grocery shopping problem described in Section 3, this ComputePath function corresponds to searching the original graph G shown in Fig. 2. The search proceeds from S_{goal} towards a state whose *Agent* and *Bought* variables are given by $S(X_p)$. The search is done in graph G but the prices of items are *not* on sale if the corresponding values of $Sale_A$ or $Sale_B$ variables are known to be 0 in $H(X_p)$. Otherwise, the prices are assumed to be unknown and the computation of the g-values is done by taking the expectation according to Eq. (5). Apart from how the g-values are computed, the ComputePath functions is identical to the backward A* search performed on graph G.

The Main and UpdateMDP functions (Fig. 6) operate in the exact same way as before with the exception that for each state X the UpdateMDP function updates, it also needs to update the corresponding belief state X^u (line 15). Note that the UpdateMDP function updates an actual policy that can be executed. Therefore, the successors of $besta(X)$ depend on the value of the hidden variable $h^{S(X),a}$ in $H(X)$, which is not necessarily equal to the one used to set up the search environment or the value of $h^{S(X),a}$ in $H(X^u)$.

5.3. Example

We now demonstrate the operation of the PPCP algorithm as we have described it in the previous section on the problem of robot navigation in a partially-known terrain example shown in Fig. 7. At the time of planning, the robot is in cell $A4$ and its goal is to go to cell $F4$. In black are shown cells that are untraversable. There are two cells (shaded in grey) whose status is unknown to the robot: cell $B5$ and $E4$. For each, the probability of containing an obstacle is 0.5. In this example, we restrict the robot to move only in four compass directions. Whenever the robot attempts to enter an unknown cell, we assume the robot moves towards the cell, senses it and enters it if it is free and returns back otherwise. The cost of each move is 1, the cost of moving towards an unknown cell, sensing it and then returning back is 2.

Fig. 7(b) shows a belief state-space for the robot navigation problem. The fully observed part of X, $S(X)$ is the location of the robot, while the hidden part of X, $H(X)$, is the status of cells $E4$ and $B5$. For example, $X_4 = [R = B4; E4 = u, B5 = 1]$, where $R = B4$ means that the robot is at cell $B4$ and $E4 = u, B5 = 1$ means that the status of cell $E4$ is still unknown
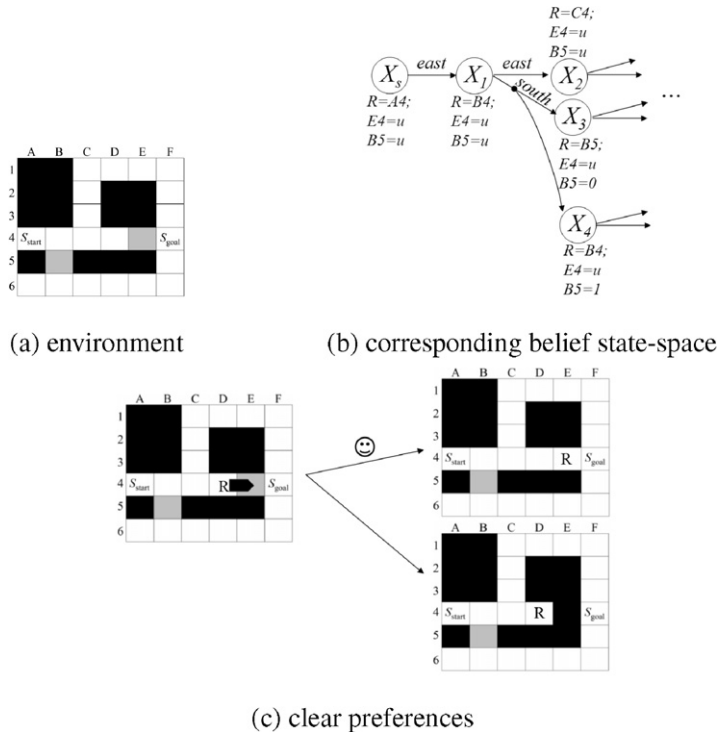

(a) environment   (b) corresponding belief state-space

(c) clear preferences

Fig. 7. The problem of robot navigation in a partially-known terrain.

(a) environment

(b) initial PPCP policy

(c) $X_\mathrm{p} = [R = A4; E4 = u, B5 = u]$

(d) PPCP policy after update

(e) $X_\mathrm{p} = [R = D4; E4 = 1, B5 = u]$

(f) PPCP policy after update

(g) $X_\mathrm{p} = [R = A4; E4 = u, B5 = u]$
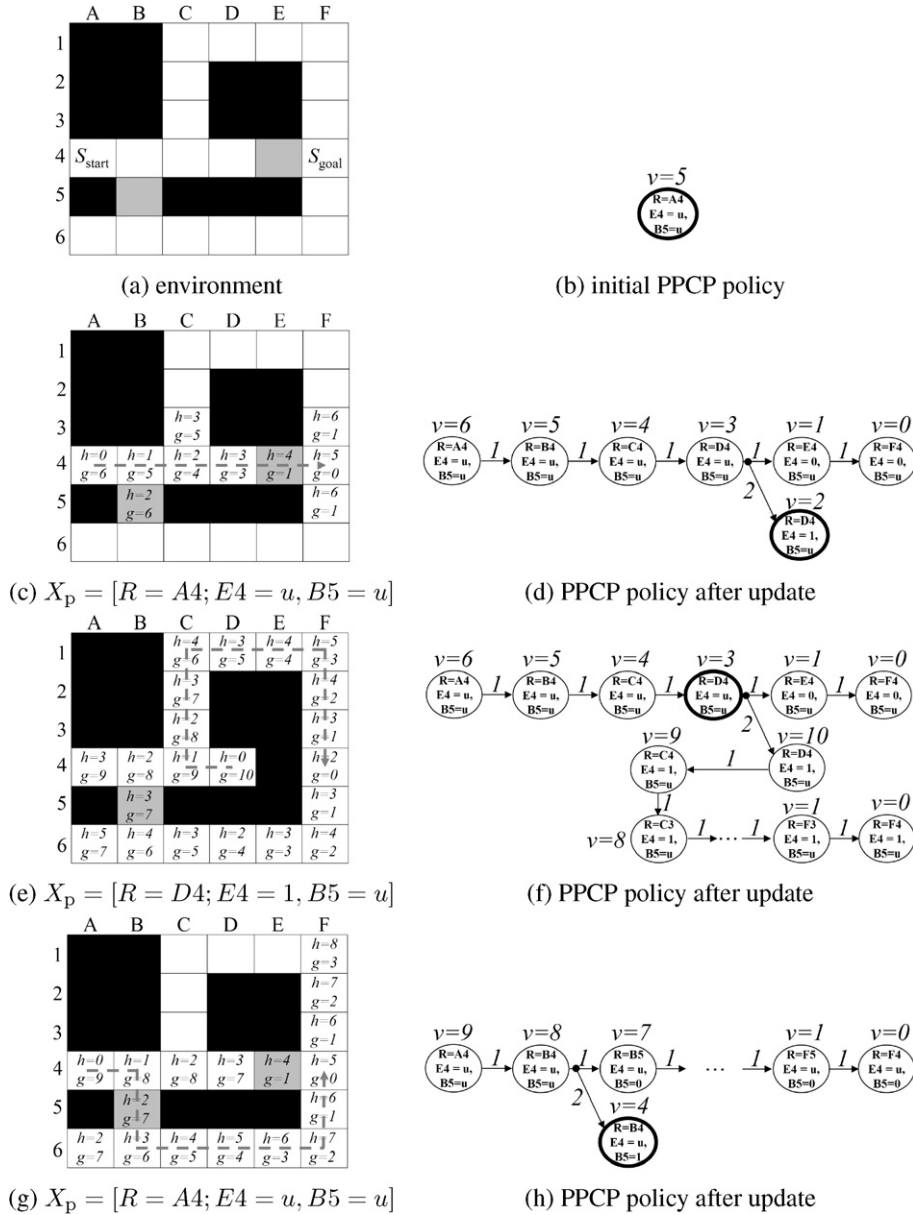
(h) PPCP policy after update

Fig. 8. An example of PPCP operation.

and cell $B5$ is known to be blocked. The robot navigation problem exhibits clear preferences since for each sensing action there always exist two outcomes: a cell is blocked or unblocked and the latter one is the preferred outcome. For example, suppose the robot state is $X = [R = D4; E4 = u, B5 = u]$ (Fig. 7(c)). Then the action $a = East$ has two outcomes: $Y_1 = [R = D4; E4 = 1, B5 = u]$ and $Y_2 = [R = E4; E4 = 0, B5 = u]$. The latter is the preferred outcome ($Y_2 = succ(X, a)^b$), and it satisfies the definition of clear preferences because the robot can always move from $E4$ back to $D4$ at cost of 1, implying that $1 + v^*(Y_1) \geqslant v^*(Y_2)$ and therefore $c(S(X), a, S(Y_1)) + v^*(Y_1) = 2 + v^*(Y_1) \geqslant 1 + v^*(Y_2) = c(S(X), a, S(Y_2)) + v^*(Y_2)$.

Figs. 8 and 9 show how PPCP solves the problem. In the left columns of Figs. 8 and 9 we show the environment each ComputePath function sets up when executed on X_p (specified underneath the figure). Thus, when executed on $X_\mathrm{p} = [R = D4; E4 = 1, B5 = u]$ in Fig. 8(e), the ComputePath function assumes cell E4 is blocked whereas cell B5 is free. We also show the heuristics (shown as h-values), the g-values and the path from $S(X_\mathrm{p})$ to S_goal (shown in grey dashed line) computed by the search. The heuristics are Manhattan distances (summation of x and y differences) from $S(X_\mathrm{p})$ to the cell in question.

In the right column of each figure we show the current policy found by PPCP after the UpdateMDP function incorporates the results of the most recent search, which is shown in the left column in the same row. The states whose v-values are smaller than what they are supposed to be according to their successors are outlined in bold. These states are candidates for being X_pivot in the next search iterations. As mentioned earlier, we exercise the following simple optimization in this
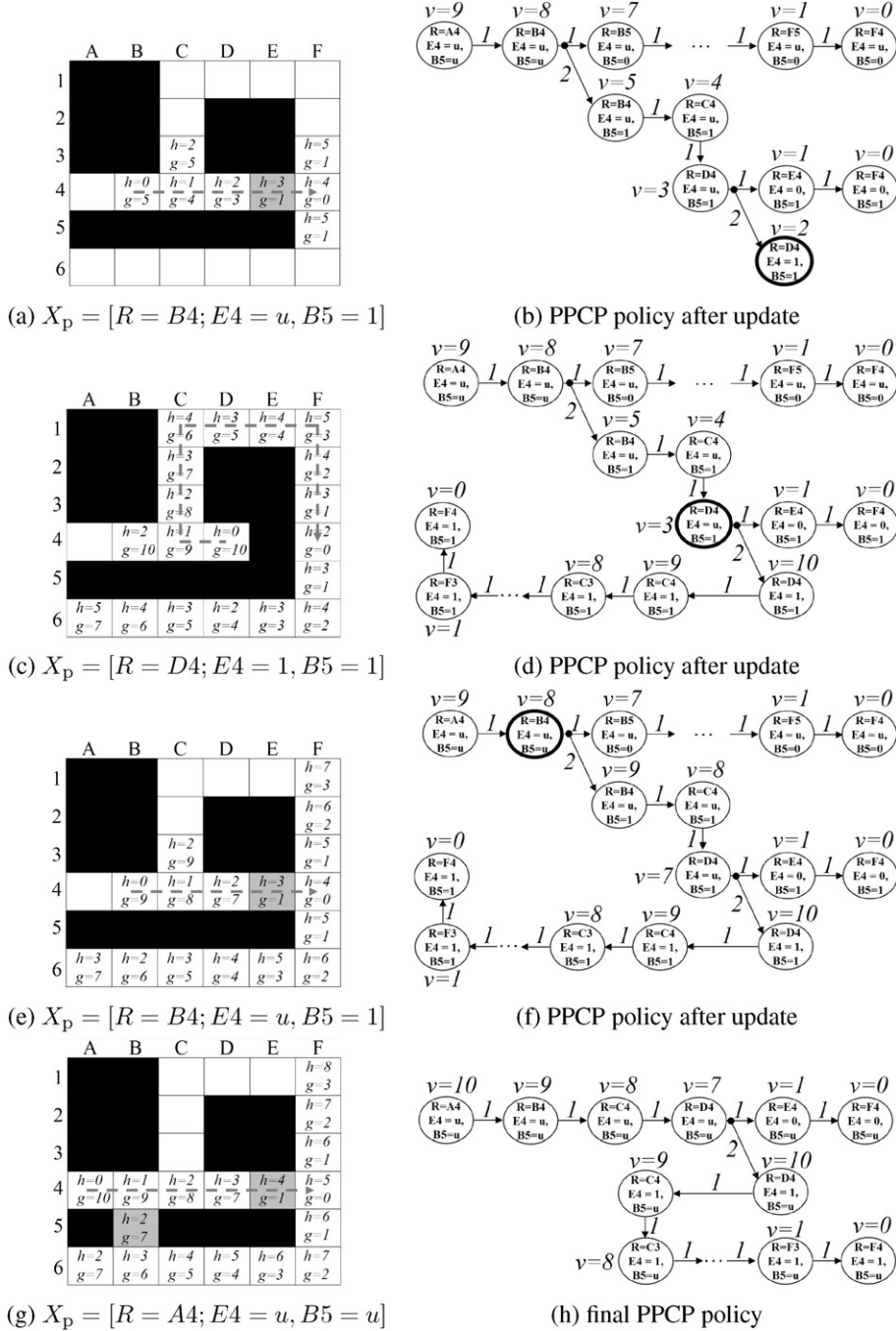
Fig. 9. An example of PPCP operation (continued).

example and all experiments: whenever a state X is chosen as the next X_{pivot}, we backtrack from X up along the policy until we encounter the first stochastic transition (or X_{start}, whichever comes first), at which point X_{pivot} is chosen to be the outcome of this transition that resides on the same branch as X. For example, in Fig. 9(d) X_{pivot} is chosen to be state $[R = B4; E4 = u, B5 = 1]$ (robot is at $B4$, cell $E4$ is unknown and cell $B5$ is known to contain an obstacle) as a result of backtracking from state $[R = D4; E4 = u, B5 = 1]$.

In computing $\tilde{Q}_{v,g}$, the v-values of belief states that do not yet exist are assumed to be equal to the Manhattan distances to the goal. These distances are also used to initialize the v-values of new belief states. The difference between the way the ComputePath function computes g-values and the way A* computes them can be seen well in Fig. 8(g). There, the g-value of cell C4 is 8, which is $1 + g(D4)$. The g-value of cell D4, on the other hand, is 7, despite the fact that $g(E4) = 1$, and

the reason is that the v-value of the state $[R = D4; E4 = 1, B5 = u]$ that corresponds to the "bad" outcome of going east at cell D4 is 10. The v-value of this state has just been updated in the previous iteration (Fig. 8(f)). When computing $\tilde{Q}_{v,g}$ according to Eq. (5), the ComputePath function accounts for this v-value. As a result, the path the ComputePath function comes up in this iteration is going through cell $B5$, which is different from the path that would have been produced by running normal A* assuming all cells free. In fact, the latter path is exactly the path computed by the ComputePath function in its first iteration (Fig. 8(c)).

Fig. 9(h) shows the policy that PPCP returns after it converges. In general, the expected cost of the found policy is bounded from above by the cost of the policy in which the robot always forgets the outcome of sensing if it was a preferred outcome. If an optimal policy does not require remembering preferred outcomes, then the policy returned by PPCP is also guaranteed to be optimal. In our example, this memoryless property would mean that during each search, the planner assumes that as soon as the robot successfully enters cell $E4$, for example, and therefore finds out that is free, it resets back the status of cell $E4$ to an unknown cell. (The non-preferred outcomes of sensing are never reset, and so if the robot finds that the cell $E4$ is blocked, then this information is always used.)

While in the example, an optimal plan does not need to remember the status of any cell the robot has successfully entered, it is possible to set up an environment when it will be sub-optimal. An optimal policy for such environment would require the robot to sense a cell but then come back from it, and use the fact that the cell is free at a later time. In Section 6.1 we will discuss how this memoryless assumption of PPCP can be relaxed.

Independently of whether the memoryless property is satisfied or not however, the algorithm *is* guaranteed to converge in a finite amount of time. It is important to emphasize that forgetting the best outcomes of hidden variables is only happening at the level of each search iteration (i.e., execution of the ComputePath function). The Main function constructs a policy in which none of the states forget anything. Thus, in Fig. 6, the Main function calls UpdateMDP function that updates the policy in the full belief state-space using the path found by each search iteration. This UpdateMDP function iterates over the found path while retaining the values of the hidden variables, both preferred and non-preferred (lines 15–16, Fig. 6). This can also be seen in the Figs. 8–9 example, where the constructed policy does *not* forget the preferred outcomes. Instead, forgetting these preferred outcomes happens only at the level of each search iteration.

5.4. Theoretical properties

We now present some of the theorems about the algorithm. They are supposed to give a sense as to why the algorithm converges and what it converges to. All the theorems together with their proofs can be found in [7].

The first few theorems relate the properties of the ComputePath function to the properties of the (backward) A* search. For example, the following theorem, perhaps unsurprisingly, states that each execution of the ComputePath function expands each state at most once, the same guarantee A* makes. (Here and in the following when comparing with A* we assume the heuristics are consistent.)

Theorem 2. *No state is expanded more than once during a single execution of the ComputePath function.*

The statement of the next theorem is once again very similar to the property (backward) A* maintains: every state with f-value (which is the summation of g-value and heuristics) smaller than or equal to the smallest f-value of the states in *OPEN* has its g-value equal to its goal distance, the cost of a shortest path from the state to the goal. While A* computes the cost of a shortest path, the ComputePath function computes the cost of a path which takes into account v-values of "bad" outcomes. To put it formally, let us re-define g^*-values for states $S(X)$ (as opposed to g^*-values defined in Eq. (4) for states X in the full belief state-space). For given v-values and a given pivot state X_{p}, g^*-values are the solution to the following fixpoint equation:

$$g^*\big(S(X)\big) = \begin{cases} 0 & \text{if } S(X) = S_{\text{goal}} \\ \min_{a \in A(S(X))} \tilde{Q}_{v,g^*}(S(X), a) & \text{otherwise} \end{cases} \qquad (6)$$

These g^*-values are goal distances for the ComputePath function, and the g-values of the computed states are equal to them.

Theorem 3. *Assuming function v is non-negative, at line 5 in Fig. 6, for any state $S(X)$ with $(heur(S(X)) < \infty \land g(S(X)) + heur(S(X)) \leqslant g(S(X')) + heur(S(X')) \,\forall S(X') \in OPEN)$, it holds that $g(S(X)) = g^*(S(X))$.*

Given the terminating condition of the while loop in ComputePath and the fact that $h(S(X_{\text{p}})) = 0$ since heuristics are consistent, it is clear that after ComputePath terminates $g(S(X_{\text{p}})) = g^*(S(X_{\text{p}}))$. Also, same as in A*, the states on the found path all have g- plus *heur*-values smaller than or equal to the g-value of the goal of the search ($S(X_{\text{p}})$). Therefore, they all are going to have their g-values equal to their corresponding g^*-values according to Theorem 3. The proof of the next theorem uses this fact, since UpdateMDP sets v-values of the states on the found path to their g-values. This theorem shows that updating the v-values of states on the trajectory found by ComputePath makes states at least as consistent as the v-values of the policy successors. In other words, the update removes negative Bellman errors on the path returned by the ComputePath function.

Theorem 4. *Suppose that before the ComputePath function is executed, for every state X it is true that $0 \leqslant v(X) \leqslant v(X^u)$. Then after the UpdateMDP returns, it holds that for each state X whose v-value was just updated by UpdateMDP function it holds that $v(X) = 0$ if $S(X) = S_{\text{goal}}$ and $v(X) \geqslant E_{X' \in succ(X, besta(X))}(c(S(X), besta(X), S(X')) + v(X'))$ otherwise.*

For the purpose of the following theorem let us also define v^u-values which are somewhat similar to v^*-values. $v^u(X)$ is the minimum expected cost of a policy under which the preferred values of hidden variables are forgotten as soon as they are observed. v^u-values are upper bounds on the expected cost of the policy PPCP returns.

$$v^u(X) = \begin{cases} 0 & \text{if } S(X) = S_{\text{goal}} \\ \min_{a \in A(S(X))} Q_{v^u, v^u}(X^u, a) & \text{otherwise} \end{cases}$$

The next theorem says that v-values after each iteration can never decrease. The reason is that each ComputePath function computes the lowest possible g-values, namely g^*-values, for the states on the found path and then the UpdateMDP function sets v-values to them. Assuming properly initialized v-values, the proof that after each iteration v-values can only increase can be done by induction on executions of the UpdateMDP functions. The next theorem also proves that v-values are bounded from above by the corresponding v^u-values. The proof of this statement can also be done by induction based on the fact that the g^*-values the ComputePath function sets g-values to can never be larger than v^u-values as long v-values were not exceeding corresponding v^u-values before the ComputePath function was executed.

Theorem 5. *v-values of states are monotonically non-decreasing but are bounded from above by the corresponding v^u-values.*

After each iteration of the algorithm the value of state X_p (and possibly others) is corrected by either changing its *besta* pointer or making an increase in its v-value. The number of possible actions is finite. The increases, on the other hand, are bounded from below by a positive constant because the belief state-space is finite (since we assumed perfect sensing and a finite number of possible values for each hidden variable). Therefore, the algorithm terminates. Moreover, at the time of termination v-value of every state on the policy is no smaller than the expectation over the immediate cost plus v-value of the successors. Therefore, the expected cost of the policy can not be larger than $v(X_{\text{start}})$. This is summarized in the following theorem.

Theorem 6. *PPCP terminates and at that time the cost of the policy defined by besta pointers is bounded from above by $v(X_{\text{start}})$ which in turn is no larger than $v^u(X_{\text{start}})$.*

The final theorem gives the conditions under which the policy found by PPCP is optimal. It states that the found policy is optimal if the memory about preferred outcomes is not required in an optimal policy, notated by ρ^*. We use $\rho^*(X)$ to denote a pointer to the action dictated by policy ρ^* at the belief state X.

Theorem 7. *Suppose there exists a minimum expected cost policy ρ^* that satisfies the following condition: for every state $X \in \rho^*$ it holds that $h^{S(X), \rho^*(X)}(X) \neq b$. Then the policy defined by besta pointers at the time PPCP terminates is also a minimum expected cost policy.*

The condition $h^{S(X), \rho^*(X)}(X) \neq b$ means that whenever an agent executes a policy action $\rho^*(X)$ at state X, the hidden variable that controls the outcomes of this action is *not* known to have a preferred outcome or there is no hidden variable that controls the outcomes (i.e., there was never any uncertainty about the outcome of the action). If this property holds for any action on the policy, then there is no need for the agent to retain information about the values of hidden variables it has already observed to have clearly preferred values. Typically, this means that the knowledge about these hidden variables has been exercised immediately and there is no need to remember them anymore. Another way of stating the memoryless property of PPCP is that if no branch of an optimal policy executes two actions that rely on the same hidden variable and assume its value is a clearly preferred value, then PPCP is guaranteed to find an optimal policy.

6. Extensions and optimizations

In the following first two sections we describe some useful extensions of PPCP algorithm. The first extension makes PPCP applicable to problems for which no acceptable policy exists that forgets the preferred outcomes of sensing. The second extension makes PPCP useable on real-time systems by making it possible to interleave planning with PPCP and execution. The last two sections describe general optimizations of the PPCP algorithm that prove to be very effective for the path clearance problem. Both optimizations leave the theoretical properties of the algorithm such as convergence and optimality under certain conditions unchanged and can be considered general optimizations of PPCP.

6.1. Overcoming memoryless property

Each search of PPCP assumes that the policy does not need to remember the outcomes of sensing if they are preferred outcomes. In the Fig. 8 example, for instance, the robot senses a cell when trying to enter it. If it is free then the robot

enters it and does not really need to remember that it is free afterwards, its future path does not involve entering the same cell again. It only needs to remember if some cells turn out to be blocked, and policies generated by PPCP do retain this information.

There are many problems, however, that do require remembering preferred outcomes, and there are at least several simple approaches to relaxing the memoryless property of PPCP. Before trying them though, perhaps the first thing one should see is if it is enough to just make sure that sensing action is done as part of the action that requires the knowledge of the status of the hidden variable. This is essentially how we set up the robot navigation problem in Fig. 8. The robot senses a cell and then enters it or backs up in a *single* action. Therefore, the preferred outcome of sensing is used in the same action as sensing itself and does not need to be remembered.

A slightly more complex solution to this problem is to augment each state $S(X)$ with the last k preferred outcomes of sensing. During each search done by the ComputePath function, each sensing operation results in a preferred outcome, as before, but now the corresponding hidden variable is pushed onto the queue of maximum size k. Thereafter, sensing does not need to be done for the value of this hidden variable as long as it remains in the queue (i.e., does not get pushed out by the new results of sensing). For example, in the robot navigation problem $S(X)$ will now consist of the location of the robot plus the last k locations that were sensed by the robot. The addition of k last preferred outcomes of sensing makes each execution of the ComputePath function more expensive, however. In particular, the size of the state-space on which the search is done now grows roughly by a factor of $|H|^k$, where $|H|$ is the number of hidden variables. Therefore, k should be set to a small number such as two or three to keep the complexity of each search low. This approach is good for the problems in which the results of most recent sensing are more likely to be useful.

If the addition of k last preferred outcomes of sensing is still not sufficient, one can also just split the vector of hidden variables $H(X)$ into two sets: $H(X)$ and $\underline{H}(X)$. The first one can be hidden variables whose preferred outcomes the ComputePath function *does* keep track of, same as in the pseudocode of the ComputePath function in Fig. 5. The second set of hidden variables, $\underline{H}(X)$, are the ones whose preferred outcomes the ComputePath function does *not* retain, same as in the pseudocode of the ComputePath function in Fig. 6. The version of PPCP that uses this approach generalizes PPCP given in Sections 5.1 and 5.2. The pseudocode of this generalized version is given in [7]. (In fact, this is the version all of the proofs are derived for.) The approach of splitting $H(X)$ into two sets is suitable for the problems in which one can have a good idea about which preferred outcomes are likely to be useful later and which are not.

6.2. Interleaving planning and execution

In many cases we would like to be able to interleave planning with execution. The agent can then start executing whatever current plan it has and while executing it, a planner can work on improving the plan. This way the agent does not need to wait for the planner to fully converge.

Interleaving planning with PPCP with execution can be done as follows. The agent first executes PPCP for several seconds. The loop in the Main function of PPCP is then suspended (right after the UpdateMDP function returns on line 21, Fig. 6), and the agent starts following the policy currently found by PPCP as given by *besta* pointers. During each agent move, X_{start} state maintained by PPCP is updated to the current state of the agent and the main loop of PPCP is resumed for another few seconds. After it is suspended again, the policy that the agent currently follows is compared against the policy that PPCP currently has and is updated to it only if the latter has a higher probability of reaching a goal location. If the policy the agent currently follows has a higher probability of reaching the goal then the agent continues to follow it. This way we avoid changing policies every time PPCP decides to explore a different policy but has not explored much of the outcomes on it yet. The probabilities of reaching a goal for an acyclic policy can be computed in a single pass over the states on the policy in their topological order starting with the start state.

Once PPCP converges to a final policy, the agent can follow the policy without re-executing PPCP again unless the agent deviates from its path due to actuation errors. If the agent does deviate significantly from the plan generated by PPCP, then PPCP can be used to re-plan. There is no need to re-plan from scratch. Instead, X_{start} is updated to the new state of the agent, the old policy is discarded, and the main loop of PPCP is resumed. Note that the values of all state variables in PPCP are preserved. It will then automatically re-use them to find a new policy from the current agent position much faster than if PPCP was re-executed from scratch.

6.3. Reducing the number of search iterations

As mentioned previously, during each search whenever the ComputePath function encounters action a executed at state $S(X)$ and the outcome is not known according to $H(X_p)$, then in evaluating Eq. (1), the ComputePath function uses the v-values of non-preferred outcomes. The v-values are estimates of the goal distances. If these non-preferred outcomes have never been explored by PPCP, then the v-values are initial estimates of the cost-to-goal from them and are likely to be much lower than what they should really be. This means that the ComputePath function will return a path that uses the state-action pair $(S(X), a)$, and only in the future iterations will PPCP find out that the v-value of these non-preferred outcomes should really be higher and this state-action pair should have been avoided.

Fig. 10 gives such example for the robot navigation in a partially-known environment problem. When solving the environment in Fig. 10(a), PPCP at some point invokes the ComputePath function on state $X_p = [R = A4; C4 = 1, C6 = u]$.
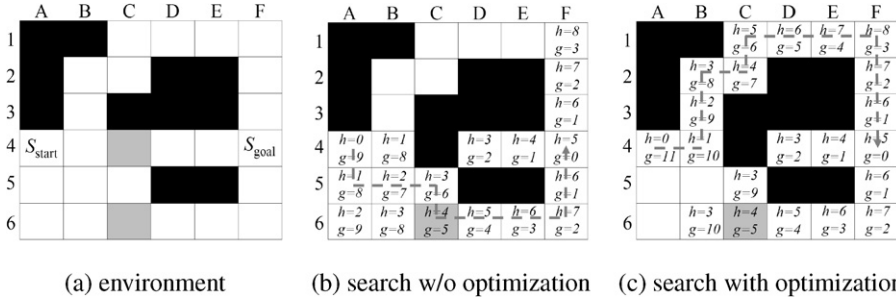
Fig. 10. The comparison of a search by PPCP (b) without the optimization vs. (c) with the optimization described in Section 6.3.

Suppose that by this time PPCP has already computed the v-value of state $[R = B6; C4 = 1, C6 = 1]$ as 12. This is the cost of getting to the goal from cell B6 if both cells C4 and C6 are blocked. During the current search, when computing the g-value of cell C5, PPCP will query the v-value of state $[R = C5; C4 = 1, C6 = 1]$ as needed by Eq. 1. If the v-value of this state has never been computed previously, PPCP initializes it to some admissible estimate such as Manhattan distance from C5 to the goal cell, which is 4 (Fig. 10(b)). After evaluating Eq. (1), the g-value of cell C5 becomes 6 $(= 0.5 \max(2 + 4, 1 + 5) + 0.5 \max(1 + 5, 1 + 5))$. Consequently, the search returns the path shown in Fig. 10(b) that goes through cells C5 and C6.

One optimization we propose is to use the v-values of neighboring states to obtain more informative v-values of states that have not been explored yet. Thus, in the example, we can deduce that the v-value of state $[R = C5; C4 = 1, C6 = 1]$ can be at least 10: the v-value of state $[R = B6; C4 = 1, C6 = 1]$, which is 12, minus an upper bound on the minimum cost of getting from $[R = B6; C4 = 1, C6 = 1]$ to state $[R = C5; C4 = 1, C6 = 1]$, which we can easily compute as 2. More formally, suppose we are interested in estimating the v-value of some state X. We can then take some (small) region R of states around X whose $H(\cdot)$ part is the same as in $H(X)$. Using each state $Y \in R$ and an upper bound $c^u(Y, X)$ on getting from state Y to state X, we can then estimate $v(X)$ as:

$$v(X) = \max_{Y \in R}\big(v(Y) - c^u(Y, X)\big) \tag{7}$$

The upper bounds, $c^u(\cdot, X)$ can be computed via a single backward Depth-First Search from X. In some problems they can also be obtained a priori. To see that Eq. (7) is a valid update for $v(X)$ consider the following trivial proof that $v(X)$ remains admissible (does not overestimate the minimum expected cost of getting to goal) provided an admissible value $v(Y)$ for each $Y \in R$. Let $v^*(Y)$ denote the minimum expected cost of getting to the goal from Y. Then:

$$v(Y) \leqslant v^*(Y) \leqslant c^u(Y, X) + v^*(X)$$

Thus, $v^*(X)$ is bounded from below by $v(Y) - c^u(Y, X)$ and by setting $v(X)$ to it we guarantee the admissibility of $v(X)$.

The only change to the algorithm is that in Fig. 6 on line 15 the v-value update is now a maximum between the old v-value and the g-value because setting the initial v-values according to Eq. (7) can now sometimes result in v-values larger than their estimates computed by the search (i.e., g-values). In other words, the new line 15 is now as follows:

15   $v(X) = \max\big(v(X), g\big(S(X)\big)\big), \quad v\big(X^u\big) = \max\big(v\big(X^u\big), g\big(S(X)\big)\big), \quad besta(X) = besta\big(S(X)\big)$

Fig. 10(c) shows the operation of the ComputePath function that uses this optimization. The g-value of cell C5 now is computed as 9 $(= 0.5 \max(2 + 10, 1 + 5) + 0.5 \max(1 + 5, 1 + 5))$ because it uses the v-value of state $[R = B6; C4 = 1, C6 = 1]$ to better estimate the v-value of $[R = C5; C4 = 1, C6 = 1]$ — the non-preferred outcome of moving from C5 towards C6. Consequently, the search returns a very different path and PPCP never has to explore the path through cells C5 and C6 that would have been returned without the optimization. The proposed optimization can substantially cut down on the overall number of search iterations PPCP has to do. This significantly overcomes the expense of computing better estimates of v-values for non-preferred outcomes.

6.4. Speeding up searches

PPCP repeatedly executes A*-like searches. As a result, much of the search efforts are repeated and it should be beneficial to employ the techniques such as D* [8], D* Lite [9] or Adaptive A* [10] that are known to significantly speed up repeated A* searches. We use the last method because it guarantees not to perform more work than A* search itself and more importantly requires little changes to our ComputePath function.[1]

---

[1] It would be an interesting direction for future work to investigate how the ComputePath function PPCP uses can be made incremental in the same way D* and D* Lite extended A* to an incremental version.
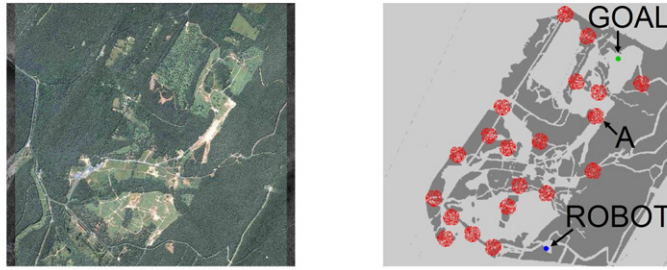
(a) 3.5 by 3.0 km satellite image   (b) corresponding traversability map

Fig. 11. Path clearance problem.

The idea is simple and is as follows. This optimization computes more informed heuristic values, $heur(S(X))$, that are used to focus each search. $heur(S(X))$ is a heuristic value that (under) estimates a distance from $S(X_p)$ to $S(X)$ under the assumption that all hidden variables whose values are unknown are set to b. The heuristics need to be consistent. Initially, before any search iteration is done, we compute the start distance (the cost of a least-cost path from $S(X_{start})$ to the state in question) of every state in $S(\cdot)$ assuming that execution of every stochastic action results in a preferred outcome (in other words, the search is done on the deterministic environment where the value of each hidden variable is set to b). In the Fig. 8 example, it means that we compute the distance from cell A4 to every other cell assuming cells E4 and B5 are free. We can do this computation via a single Dijkstra's search. Let us denote the computed value for each state $S(X)$ by $heur^*(S(X_{start}), S(X))$.

The computed value $heur^*(S(X_{start}), S(X))$ is a perfect estimate of the start distance in the environment where every hidden variable is set to b. Therefore it is a good heuristic value to use when the ComputePath function is invoked with $X_p = X_{start}$. The problem, however, is that the ComputePath function is most of the time called to find a path from some other state $X_p \neq X_{start}$. We employ the same principle as in [10] that allows us to use our $heur^*$-values anyway: for every state $S(X)$, its heuristic value $heur(S(X))$ can be improved as follows

$$heur\big(S(X)\big) = \max\big(heur\big(S(X)\big), heur^*\big(S(X_{start}), S(X)\big) - heur^*\big(S(X_{start}), S(X_p)\big)\big)$$

(Note that $S(X)$ does not retain the value of $heur(S(X))$ from one search to another.) For the same reasoning as in [10], the updated $heur(S(X))$ is guaranteed not to overestimate the actual distance from $S(X_p)$ to $S(X)$ and to remain a consistent function.

7. Application to path clearance

The problem of path clearance is the problem of planning for a robot whose task is to reach its goal as quickly as possible without being detected by an adversary [11,12]. The robot does not know beforehand the precise locations of adversaries, but has a list of their possible locations. When navigating, the robot can come to a possible adversary location, sense it using its long range sensor and go around the area if an adversary is detected or cut through this area otherwise.

The example in Fig. 11 demonstrates the path clearance problem. Fig. 11(b) shows the traversability map of the satellite image of a 3.5 km by 3 km area shown in Fig. 11(a). The traversability map is obtained by converting the image into a discretized 2D map where each cell is of size 5 by 5 meters and can either be traversable (shown in light grey color) or not (shown in dark grey color). The robot is shown by the blue circle and its goal by the green circle.[2] Red circles are *possible* adversary locations and their radii represent the sensor range of adversaries (100 meters in this example). The radii can vary from one location to another. The locations can be specified either manually or automatically in places such as narrow passages. Each location also comes with a probability of containing an adversary (50% for each location in the example): the likelihood that the location contains an adversary. The probabilities can vary from one location to another.

The path the robot follows may change any time the robot senses a possible adversary locations (the sensor range of the robot is 105 meters in our example). A planner, therefore, needs to reason about possible outcomes of sensing *before the execution* and to generate a policy that dictates which path the robot should take as a function of the outcome of each sensing. Ideally, the generated policy should minimize the expected traversal distance. Finding such policy with guarantees on its optimality, however, corresponds to planning with missing information about the environment. In fact, the path clearance problem is very much equivalent to the problem of planning for a robot navigating in a partially-known terrain. The difference is that in the path clearance problem, detecting an adversary blocks a large area resulting in a long detour. An adversary location has also a tendency to be placed in such places that it blocks the whole path and the robot has to backup and choose a totally different route. As a result, the detours can be much costlier than in the case of navigation in a partially-known terrain, even when the amount of uncertainty is much less.

---

[2] For colors see the web version of this article.

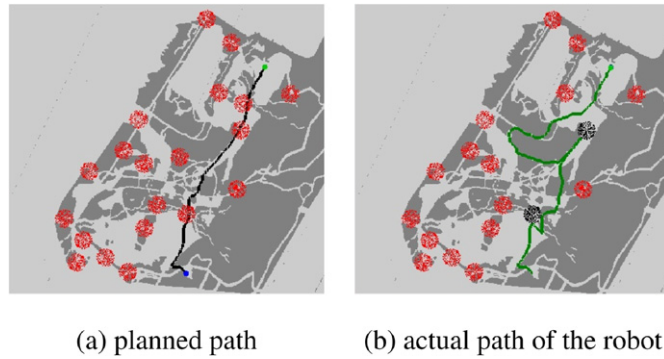(a) planned path   (b) actual path of the robot

Fig. 12. Solving path clearance problem with freespace assumption.

Solving path clearance using assumptive planning. To avoid the computational complexity, a robot operating in a partially-known terrain often performs assumptive planning [8,13,14]. In particular, it often just follows a shortest path under the assumption that all unknown areas in the environment are free unless the robot has already sensed them otherwise. This is known as a freespace assumption [14]. The robot follows such path until it either reaches its goal or senses new information about the environment. In the latter case, the robot re-computes and starts following a new shortest path under the freespace assumption.

The freespace assumption is also applicable to the path clearance problem. The robot can always plan a path under the assumption that no adversary is present unless sensed otherwise. This assumption makes path clearance a deterministic planning problem and therefore can be solved efficiently. The fact that the robot ignores the uncertainty about the adversaries, however, means that it risks having to take long detours, and the detours in the path clearance problem tend to be longer than in the problem of navigation in a partially-known terrain as we have previously explained.

For example, Fig. 12(a) shows the path computed by the robot that uses the freespace assumption. According to the path, the robot tries to go through the possible adversary location A (shown in Fig. 11(b)) as it is on the shortest route to the goal. As the robot senses the location A, however, it discovers that the adversary is present in there (the red circle becomes black after sensing). As a result, the robot has to take a very long detour. Fig. 12(b) shows the actual path traversed by the robot before it reaches its goal.

Solving path clearance using PPCP planning. Turns out that the path clearance problem can be efficiently solved using PPCP. Same as in the robot navigation in a partially-known terrain, in the path clearance problem, there are also clear preferences for the values of unknowns. The unknowns are m binary variables, one for each of the m possible adversary locations. The preference for each of these variables is to have a value false: no adversary is present.

Differently from our simple example in Fig. 7, however, in path clearance, the robot has a long range sensor. It therefore may sense whether an adversary is present before actually reaching the area covered by the adversary. As a result, it needs to remember the preferred outcomes if it wants to sense adversaries from a long distance. To address this, we augment $S(X)$ with the last $k = 3$ preferred outcomes of sensing as described in Section 6.1.

Fig. 13 shows the application of PPCP to the path clearance example in Fig. 11. Before the robot starts executing any policy, PPCP plans for five seconds. Fig. 13(a) shows the very first policy produced by PPCP (in black color). It is a single path to the goal, which in fact is exactly the same as the path planned by planning with the freespace assumption (Fig. 12(a)). PPCP produced this path within few milliseconds in its first iteration. At the next step, PPCP refines the policy by executing a new search which determines the cost of the detour the robot has to take if the first adversary location on the found path contains an adversary. The result is the new policy (Fig. 13(b)). PPCP continues in this manner and at the end of five seconds allocated for planning, it generates the policy shown in Fig. 13(c). This is the policy that is passed to the robot for execution. Each fork in the policy is where the robot tries to sense an adversary and chooses the corresponding branch.

As explained in Section 6.2, we interleave planning with execution. Thus, while the robot executes the plan, PPCP improves it relative to the current position of the robot. Fig. 13(d) shows the new position of the robot (the robot travels at the speed of 1 meter per second) and the current policy generated by PPCP after 15 seconds since the robot was given its goal. Fig. 13(e) shows the position of the robot and the policy PPCP has generated after 30 seconds. At this point, PPCP has converged and no more refinement is necessary. Note how the generated policy makes the robot go through the area on its left since there are a number of ways to get to the goal and therefore there is a high chance that one of them will be available. Unlike the plan generated by planning under freespace assumption, the plan generated by PPCP avoids going through location A. Fig. 13(f) shows the actual path traversed by the robot. It is 4123 meters long while the length of the trajectory traversed by the robot that plans with freespace assumption (Fig. 12(b)) is 4922 meters.

(a) the first policy   (b) the second policy

(c) after 5 secs   (d) after 15 secs

(e) after 30 secs (PPCP converged)   (f) actual path of the robot

Fig. 13. Solving path clearance problem with PPCP.

8. Experimental analysis

8.1. Navigation in a partially-known terrain

In this section, we use the problem of robot navigation in unknown terrain to evaluate the performance of PPCP algorithm (without optimizations). In all of the experiments we used randomly generated fractal environments that are often used to model outdoor environments [15]. A robot was allowed to move in eight directions, and the cost of each move in between two traversable cells was defined as the distance between the centers of the corresponding cells times the cost of traversing the target cell (according to its fractal value). The cost of sensing and discovering an initially unknown cell to be untraversable was set to the cost of moving towards the cell and then moving back into the source cell.

In the first set of experiments we compared the performance of PPCP with three optimal algorithms: VI (value iteration), LAO* [16], and RTDP [3]. All three can be used to plan in finite-size belief state-spaces, and the latter two have been shown to be competitive with other planners in belief state-spaces [5]. To make VI more efficient and scalable, we first performed a simple reachability analysis from the initial belief state, and then ran VI only on the reachable portion of the belief state-space. Both PPCP and LAO* used the following (admissible and consistent) heuristics to estimate distances in between any two states with coordinates (x_1, y_1) and (x_2, y_2):

$$\sqrt{2}\min(|x_1 - x_2|, |y_1 - y_2|) + \left(\max(|x_1 - x_2|, |y_1 - y_2|) - \min(|x_1 - x_2|, |y_1 - y_2|)\right)$$

The same heuristics were also used to initialize the state values when running VI and RTDP algorithms.

Fig. 14(a) shows the time it takes to converge, the percent of solved environments (the environments were declared to be unsolved when an algorithm ran for more than 15 minutes), and the solution costs for the four algorithms for the

# of unknowns	Percent Solved				Time to Convergence (in secs)				Solution Cost
	VI	LAO*	RTDP	PPCP	VI	LAO*	RTDP	PPCP	Same for All
6	92%	72%	100%	100%	7.7	43.9	0.4	0.1	112,284
10	—	36%	92%	100%	—	123.1	19.7	0.2	117,221
14	—	—	80%	100%	—	—	25.8	0.2	113,918
18	—	—	48%	100%	—	—	52.3	0.7	112,884

(a) runs on small environments
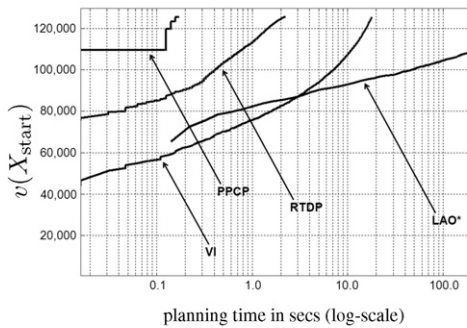
(b) rate of convergence

# of unknowns	Traversal Cost	
	PPCP	Freespace
1,000 (0.4%)	1,368,388	1,394,455
2,500 (1.0%)	1,824,853	1,865,935
5,000 (2.0%)	1,521,572	1,616,697
10,000 (4.0%)	1,626,413	1,685,717
25,000 (10.0%)	1,393,694	1,484,018

(c) runs on large environments

Fig. 14. Experimental results.

environments of size 17 by 17 cells. The number of unknown locations increases from 6 to 18 and for each number the results are averaged over 25 environments.

The figure shows that PPCP converges faster than the other algorithms and the differences in speeds grow large very fast with the increase in the number of unknown locations. More importantly, PPCP was able to solve all environments in all cases. (We do not give numbers for VI for more than 6 unknowns and LAO* for more than 10 unknowns because they were running out of memory on almost all environments.[3]) Fig. 14(a) also shows that in all the cases the solution returned by PPCP turned out to be the same as the one returned by other algorithms, an optimal solution. (An interesting and potentially important by-product of these results is an implication that, at least in randomly generated environments, an optimal navigation in a partially-known environment does not really need to memorize the cells that turn out to be free.) Finally, Fig. 14(b) shows the rate of convergence (v-value of start state) of the algorithms for one of the environments with 6 unknowns (note the log scale of the time).

Besides the algorithms we compared PPCP against, there are other efficient algorithms such as HDP [17], MCP [18], FF-replan [19] and FPG [20] that can be used to plan in finite belief state-spaces. While we have not compared their performance, we believe they would show the performance similar to the one exhibited by RTDP and LAO* since they all *have* to perform planning in the belief state-spaces that are exponential in the number of unknowns.

The second set of experiments shows that PPCP can be applied to the problem of robot navigation in environments of large size and with large number of unknown locations. Fig. 14(c) compares the performance of PPCP against a strategy of planning with freespace assumption. The comparison is done on the environments of size 500 by 500 cells with the number of unknown locations ranging from 1000 (0.4% of overall size) to 25,000 (10%). (The size of the corresponding belief state-spaces therefore ranges from $250,000 \cdot 3^{1000}$ to $250,000 \cdot 3^{25,000}$.) Unlike in the previous experiments, in these ones the robot was moving and was given only 1 second to plan during each of its moves (for both planning with PPCP and planning with freespace assumption). This amount of time was always sufficient for planning with freespace assumption to generate a path. The PPCP planning, however, was interleaved with execution as described in Section 6.2. In most experiments, PPCP converged to a final policy after several tens of moves. Fig. 14(c) summarizes the execution costs of two approaches

---

[3] In many domains, LAO* runs much better than VI. In our domain however, the performance of LAO* was comparable to VI and much worse than that of RTDP. We believe that the reason for this was the fact that the heuristics were not that informative since the costs of cells were often much larger than ones. If the heuristics do not focus efforts well, then VI with a reachability analysis may even become more efficient than LAO* due to its much smaller overhead.

averaged over 25 randomly generated fractal environments for each row in the table. The results show that the cost of the trajectory traversed by the robot with PPCP planning is consistently smaller than the one traversed by the robot with freespace assumption planning.

8.2. Path clearance

In this section, we study the performance of PPCP algorithm on the path clearance problem. In all of the experiments we used the extended version of PPCP that allowed it to remember $k = 3$ last preferred outcomes (described in Section 6.1). In all of our experiments we again used randomly generated fractal environments to model outdoor environments. On top of these fractal environments, however, we also superimposed a number of randomly generated paths in between randomly generated pairs of points. The paths were meant to simulate roads through forests and valleys and that are usually present in outdoor terrains. Figs. 15(a, b) show typical environments that were used in our experiments. The lighter colors represent more easily traversable areas. All environments were of size 500 by 500 cells, with the size of each cell being 5 by 5 meters.

The test environments were split into two groups. Each group contained 25 environments. For each environment in the group I we set up 30 possible adversary locations at randomly chosen coordinates but in the areas that were traversable. (The size of the corresponding belief state-space is $250,000 \cdot 3^{30}$.) Fig. 15(a) shows a plan the PPCP algorithm with both optimizations (described in Sections 6.3 and 6.4) has generated after full convergence for one of the environments in group I. For each environment in the group II we set up 10 possible adversary locations. (The size of the corresponding belief state-space is $250,000 \cdot 3^{10}$.) The coordinates of these locations, however, were chosen such as to maximize the length of detours. This was meant to simulate the fact that an adversary may often be set at a point that would make the robot take a long detour. In other words, an adversary is often set at a place that the robot is likely to traverse. Thus, the environments in group II are more challenging. Fig. 15(b) shows a typical environment from the group II together with the plan generated by PPCP with both optimizations. The shown plan has about 95% probability of reaching the goal (in other words, the robot executing the policy has at most 5% chance of encountering an outcome for which the plan had not been generated yet). In contrast to the plan in Fig. 15(a), the plan for the environment in group II is more complex—the detours are much longer—and it is therefore harder to compute. For each possible adversary location the probability of containing an adversary was set at random to a value in between 0.1 and 0.9.

We have run two sets of experiments on these environments. In the first set we compared the unoptimized PPCP algorithm to the PPCP algorithm with the two optimizations we have described in Sections 6.3 and 6.4. Table 1 shows the results for the group I averaged over all of the environments in it. The algorithms were run until full convergence in order to obtain the comparison results. According to them the number of states expanded by the unoptimized PPCP is about five times more and its run-time is also close to five times longer than for the optimized PPCP. The unoptimized PPCP has also converged on less environments within 15 minutes.

In the second set of experiments we compared the execution cost of the robot planning with our optimized PPCP versus the execution cost of the robot planning with freespace assumption [14]. Unlike in the previous experiments, the robot was moving and had 5 seconds to plan while traversing 5 meter distance. This amount of time was always sufficient for planning with freespace assumption to generate a path. The PPCP planning, on the other hand, was interleaved with execution as we have explained in Section 6.2.
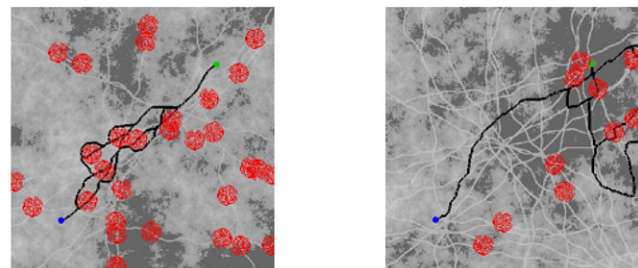
(a) A typical group I environment   (b) A typical group II environment

Fig. 15. The example of environments used in testing and the plans generated by PPCP for each.

Table 1
The comparison of unoptimized and optimized PPCP on Group I environments. The convergence times are given for the environments on which *both* algorithms converged within 15 minutes.

	# of expansions	Time to convergence (secs)	Converged within 15 minutes
Unoptimized PPCP	59,759,717	281.83	64%
Optimized PPCP	11,911,585	60.81	92%

Table 2
The overhead in execution cost of navigating using planning with freespace assumption over navigating using planning with PPCP.

	Overhead in execution cost			
	Group I no penalty	Group II no penalty	Group I with penalty	Group II with penalty
freespace	5.4%	5.2%	35.4%	21.6%
freespace2	0.5%	4.9%	4.8%	17.0%
freespace3	2.1%	4.3%	0.0%	12.7%

Table 2 shows the overhead in the execution cost incurred by the robot that plans with the freespace assumption over the execution cost incurred by the robot that uses PPCP for planning. The rows freespace2 and freespace3 correspond to making a cost of going through a cell that belongs to a possible adversary location twice and three times higher than what it really is, respectively. One may scale costs in this way in order to bias the paths generated by the planner with freespace assumption away from going through possible adversary locations. The results are averaged over 8 runs for each of the 25 environments in each group. For each run the true status of each adversary location was generated at random according to the probability having an adversary in there.

The figure shows that planning with PPCP results in considerable execution cost savings. The savings for group I environments were small only if biasing the freespace planner was set to 2. The problem, however, is that the biasing factor is dependent on the actual environment, the way the adversaries are set up and the sensor range of an adversary. Thus, the overhead of planning with freespace for the group II environments is considerable across all bias factors. In the last two columns we have introduced penalty for discovering an adversary. It simulated the fact that the robot runs the risk of being detected by an adversary when it tries to sense it. In these experiments, the overhead of planning with freespace assumption becomes very large. Also, note that the best bias factor for freespace assumption has now shifted to 3 indicating that it does depend on the actual problem. Overall, the results indicate that planning with PPCP can have significant benefits and do not require any tuning.

9. Related work

In general, planning with missing (incomplete) information about the environment and with sensing is a special class of planning for Partially Observable Markov Decision Processes (POMDPs) [5]. As a result, theoretically, algorithms for solving POMDPs are also applicable to solving the problem of planning with missing information. Unfortunately, however, planning optimally for POMDPs, in general, and planning with missing information, in particular, is known to be intractable [1,2]. Various approximations techniques have been proposed instead [21–28]. For example, grid-based approaches such as [28–30] solve POMDPs by putting specialized grids over infinite belief state-spaces, thereby converting the planning problem into solving a finite-size but usually very large MDP. Point-based approaches such as [24,26,27,31] approximate the value function over the whole belief space by computing it for a relatively small set of reachable points in the belief space. Factorization-based approaches such as [21–23] use factored representation of belief states. Baral and Son have developed approximation techniques for solving planning with missing information problems [32].

A number of approaches capable of planning with missing information have also been based on the idea of using heuristic searches in one way or another [3,5,16–18,31,33–35]. For example, LAO* [16]—one of the algorithms that we used in our experiments—is an efficient combination of dynamic programming and A*-like extensions developed specifically for planning in MDPs. It has also be shown, however, to be able to find policies in the belief state-spaces [5]. MCP [18] can also efficiently find optimal policies by running a series of A*-like searches in the belief state-spaces with sparse stochasticity. HSVI [31] and FSVI [35] incorporate some of the ideas behind heuristic searches into the point-based approaches. MAA* [34] is an algorithm for solving finite-horizon decentralized POMDPs optimally using A*-like processing. Similarly to how we used it in our experiments, RTDP [3] can also be used to find solutions to POMDP problems by planning in belief state-spaces [5].

Many of the abovementioned algorithms are capable of solving general POMDP problems. It is important to realize however, that the problem we are addressing in this paper is a much narrower (and simpler) than solving a general POMDP. For one, we assume that the underlying problem is deterministic and there is only uncertainty about some actions due to missing information about the environment. We also assume sensing is perfect which entails a finite size belief state-space with an acyclic optimal policy. Most importantly, however, we concentrate on the class of problems for which there are clear preferences on the missing information. The most relevant to our work, perhaps, is the algorithm in [36], developed for the problem of robot navigation in a partially-known terrain. Similarly to our definition of clear preferences, their planner has taken advantage of the idea that the cost of the plan if a cell is free can not be larger than the cost of the plan if the cell is occupied. Based on this idea, they proposed a clever planner that is capable of finding optimal policies much faster than other optimal approaches.

The goal of our work, however, is to avoid dealing with the exponentially large belief state-spaces altogether, which is required to guarantee the optimality of the solution. This allows us to solve very efficiently and without running out of memory large environments with a large amount of missing information. The cost is the solution optimality guarantee, which can only be made under certain conditions.

10. Discussion and future work

Besides its efficiency and low memory requirements, the other important advantages of the PPCP algorithm in our opinion are its simplicity and ease of implementation. PPCP is easy to implement because it is really just running a series of A* searches on the instances of underlying problem, each of which is made deterministic by making the necessary assumptions about the pieces of missing information. For example, in the path clearance problem, PPCP reduced to running a series of A* searches (with the exception of how g-values are computed) to find paths in the environments. Each environment had some adversaries present and some not, as specified in X_p. Therefore, the implementation of the algorithm was rather trivial.

The main disadvantage of PPCP is that it can only provide optimality guarantees under certain conditions (as described in Section 5.4). It is our hope, however, that it might be possible to derive general bounds on the sub-optimality of the solutions returned by PPCP for the cases when these conditions are *not* satisfied. Interestingly, in our experiments all of the solutions returned by PPCP were optimal when compared on the environments small enough to be solved by algorithms that can find provably optimal solutions.

Experimentally, PPCP works also for problems in which clear preferences are not so clear. That is, even though a particular outcome of sensing is thought to be a preferred outcome, it does not satisfy Definition 1. In our opinion, it would be valuable to analyze the behavior of PPCP for such problems from a theoretical side. In particular, it would be interesting to derive a function that relates the sub-optimality of PPCP to how much the clear preferences are not satisfied.

Finally, in this paper we concentrated on the notion of clear preferences on the missing information. There are other common sources of uncertainty, however. One direction for future research is therefore to explore whether the notion of clear preferences can be extended to cover other types of uncertainty such as sensor noise and uncertainty in actuation. For instance, in the latter case, one can also sometimes name the preferred outcomes of actions. Thus, a robot moving along a cliff clearly prefers not to slip. Sometimes, these preferences are clear and sometimes they can be "nearly" clear (i.e., sometimes a slip outcome may turn out to be a good outcome at the end). In either case, however, it would be interesting to investigate whether clear preferences could be assumed and used to construct a planner capable of dealing in real-time with large-scale problems exhibiting both the uncertainty in actuation and the uncertainty in the environment.

11. Conclusions

Most of us are not very good in planning under uncertainty. When faced with such a task, we never try to derive a plan that minimizes the expected cost. Instead, we will typically reason only about few contingencies and assume that in all the other cases the fortune will look upon us. The key to being able to do this, however, is the fact that we usually know (or assume) ahead of time what is good for us.

One of the goals of this paper was to formally define this notion of *clear preferences* on missing information about the environment. A second goal of the paper was to show how the existence of these clear preferences can be used to construct an efficient planner PPCP. By making use of these preferences, PPCP solves the planning problem by running a series of deterministic A*-like searches in the space of the original (deterministic) planning problem (and not in the belief state-space that is exponential in the number of unknowns). The complexity of each of these searches is the same as the complexity of planning after making some assumptions about all of the unknowns, which is a common way to make real-time planning possible. This makes PPCP highly efficient and scalable to large-scale planning problems with large amounts of uncertainty.

In our theoretical analysis, we have shown that once converged, the plan returned by PPCP is guaranteed to be optimal under certain conditions. In our experimental analysis, we have shown that PPCP can be successfully used for planning in partially-known terrains and for solving the path clearance problem, both important problems in robotics. For both problems, PPCP could scale to much larger environments and with much more uncertainty than previously possible. We are also currently working on applying PPCP to several other planning problems in robotics including navigation under uncertainty in the position of moving objects such as humans and planning an autonomous landing for unmanned helicopters under uncertainty in the safety of multiple landing sites. We therefore hope that this paper will stimulate more research on the notion of clear preferences on uncertainty, will make available to others an efficient algorithm for probabilistic planning with missing information, and finally, will encourage a wider use of planning under uncertainty for real-time robots operating in large-scale environments.

Acknowledgements

References

[1] C.H. Papadimitriou, J.N. Tsitsiklis, The complexity of Markov Decision Processes, Mathematics of Operations Research 12 (3) (1987) 441–450.
[2] C. Baral, V. Kreinovich, R. Trejo, Computational complexity of planning and approximate planning in the presence of incompleteness, Artificial Intelligence 122 (1–2) (2000) 241–267.
[3] A. Barto, S. Bradtke, S. Singh, Learning to act using real-time dynamic programming, Artificial Intelligence 72 (1995) 81–138.
[4] P.E. Hart, N.J. Nilsson, B. Raphael, A formal basis for the heuristic determination of minimum cost paths, IEEE Transactions on Systems, Science, and Cybernetics SSC-4 (2) (1968) 100–107.
[5] B. Bonet, H. Geffner, Planning with incomplete information as heuristic search in belief space, in: S. Chien, S. Kambhampati, C. Knoblock (Eds.), Proc. 6th International Conf. on Artificial Intelligence Planning and Scheduling, AAAI Press, Breckenridge, CO, 2000, pp. 52–61.
[6] M. Puterman, Markov Decision Processes: Discrete Stochastic Dynamic Programming, John Wiley and Sons, 1994.
[7] M. Likhachev, A. Stentz, PPCP: The proofs, Tech. Rep., University of Pennsylvania Philadelphia, PA, 2008.
[8] A. Stentz, The focussed D* algorithm for real-time replanning, in: Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI), 1995.
[9] S. Koenig, M. Likhachev, D* Lite, in: Proceedings of the Eighteenth National Conference on Artificial Intelligence (AAAI), 2002.
[10] S. Koenig, M. Likhachev, Adaptive A*, in: Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS), 2005, poster abstract.
[11] M. Likhachev, A. Stentz, Goal directed navigation with uncertainty in adversary locations, in: Proceedings of the International Conference on Intelligent Robots and Systems (IROS), 2007.
[12] M. Likhachev, A. Stentz, Path clearance using multiple scout robots, in: Proceedings of the Army Science Conference (ASC), 2006.
[13] I. Nourbakhsh, M. Genesereth, Assumptive planning and execution: a simple, working robot architecture, Autonomous Robots Journal 3 (1) (1996) 49–67.
[14] S. Koenig, Y. Smirnov, Sensor-based planning with the freespace assumption, in: Proceedings of the IEEE International Conference on Robotics and Automation (ICRA), 1996.
[15] A. Stentz, Map-based strategies for robot navigation in unknown environments, in: AAAI Spring Symposium on Planning with Incomplete Information for Robot Problems, 1996.
[16] E. Hansen, S. Zilberstein, LAO*: A heuristic search algorithm that finds solutions with loops, Artificial Intelligence 129 (2001) 35–62.
[17] B. Bonet, H. Geffner, Faster heuristic search algorithms for planning with uncertainty and full feedback, in: Proceedings of the 18th International Joint Conference on Artificial Intelligence, 2003, pp. 1233–1238.
[18] M. Likhachev, G. Gordon, S. Thrun, Planning for Markov Decision Processes with sparse stochasticity, in: Advances in Neural Information Processing Systems (NIPS) 17, MIT Press, Cambridge, MA, 2004.
[19] S. Yoon, A. Fern, R. Givan, FF-replan: A baseline for probabilistic planning, in: Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS), 2007.
[20] O. Buffet, D. Aberdeen, The factored policy gradient planner, in: Proceedings of the Fifth International Planning Competition (IPC), 2006.
[21] C. Boutilier, D. Poole, Computing optimal policies for partially observable decision processes using compact representations, in: Proceedings of the National Conference on Artificial Intelligence (AAAI-96), AAAI Press/The MIT Press, Portland, OR, 1996, pp. 1168–1175.
[22] X. Boyen, D. Koller, Tractable inference for complex stochastic processes, in: Proceedings of the International Conference on Uncertainty in Artificial Intelligence (UAI), 1998, pp. 33–42.
[23] C. Guestrin, D. Koller, R. Parr, Solving factored pomdps with linear value functions, in: Proceedings of the Workshop on Planning under Uncertainty and Incomplete Information, 2001.
[24] K. Poon, A fast heuristic algorithm for decision-theoretic planning, Ph.D. thesis, The Hong Kong University of Science and Technology, 2001.
[25] N. Roy, G. Gordon, Exponential family pca for belief compression in POMDPs, in: Advances in Neural Information Processing Systems, 2002.
[26] J. Pineau, G. Gordon, S. Thrun, Point-based value iteration: An anytime algorithm for POMDPs, in: Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI), 2003.
[27] M. Spaan, N. Vlassis, A point-based POMDP algorithm for robot planning, in: Proceedings of the IEEE International Conference on Robotics and Automation, 2004, pp. 2399–2404.
[28] B. Bonet, An ϵ-optimal grid-based algorithm for Partially Observable Markov Decision Processes, in: Proceedings of the International Conference on Machine Learning, 2002.
[29] W.S. Lovejoy, Computationally feasible bounds for partially observed Markov Decision Processes, Operations Research 39 (1) (1991) 162–175.
[30] R. Zhou, E.A. Hansen, An improved grid-based approximation algorithm for POMDPs, in: Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI), 2001.
[31] T. Smith, R.G. Simmons, Heuristic search value iteration for POMDPs, in: Proceedings of the International Conference on Uncertainty in Artificial Intelligence (UAI), 2004.
[32] Baral, Son, Approximate reasoning about actions in presence of sensing and incomplete information, in: Proceedings of International Logic Programming Symposium (ILPS), 1997.
[33] R. Washington, BI-POMDP: Bounded, incremental, partially-observable Markov-model planning, in: Proceedings of the European Conference on Planning (ECP), 1997, pp. 440–451.
[34] F.C.D. Szer, S. Zilberstein, Maa*: A heuristic search algorithm for solving decentralized POMDPs, in: Proceedings of the International Conference on Uncertainty in Artificial Intelligence (UAI), 2005.
[35] G. Shani, R.I. Brafman, S.E. Shimony, Forward search value iteration for POMDPs, in: Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI), 2007.
[36] D. Ferguson, A. Stentz, S. Thrun, PAO* for planning with hidden state, in: Proceedings of the 2004 IEEE International Conference on Robotics and Automation, 2004.