PROST: Probabilistic Planning Based on UCT

Thomas Keller and **Patrick Eyerich**
Albert-Ludwigs-Universität Freiburg
Institut für Informatik
Georges-Köhler-Allee 52
79110 Freiburg, Germany
{tkeller,eyerich}@informatik.uni-freiburg.de

Abstract

We present PROST, a probabilistic planning system that is based on the UCT algorithm by Kocsis and Szepesvári (2006), which has been applied successfully to many areas of planning and acting under uncertainty. The objective of this paper is to show the application of UCT to domain-independent probabilistic planning, an area it had not been applied to before. We furthermore present several enhancements to the algorithm, including a method that is able to drastically reduce the branching factor by identifying *superfluous actions*. We show how *search depth limitation* leads to a more thoroughly investigated search space in parts that are influential on the quality of a policy, and present a sound and polynomially computable detection of *reward locks*, states that correspond to, e.g., dead ends or goals. We describe a general *Q-value initialization* for unvisited nodes in the search tree that circumvents the initial random walks inherent to UCT, and leads to a faster convergence on average. We demonstrate the significant influence of the enhancements by providing a comparison on the IPPC benchmark domains.

Introduction

In its fourth incorporation, the International Probabilistic Planning Competition (IPPC) has undergone a radical change by replacing PPDDL, the probabilistic dialect of the Planning Domain Definition Language, with the Relational Dynamic Influence Diagram Language (RDDL) (Sanner 2010). RDDL overcomes shortcomings of PPDDL in probabilistic settings like difficulties when modeling exogenous or independent effects, especially when combined with concurrency. Thereby, it is well suited to design domains that are probabilistically complex. This is in contrast to the domains that were used in previous competitions which were mostly probabilistically simple (Little and Thiébaux 2007). This kind of problem can usually be tackled easily by determinization based replan approaches like FF-Replan (Yoon, Fern, and Givan 2007), the winner of the first IPPC, or RFF (Teichteil-Königsbuch, Infantes, and Kuter 2008), the winner of IPPC 2008. Probabilistically complex problems, on the other hand, require planning systems to take probabilities into account in their decision making process, for example to predict the influence of exogenous events, to avoid dead ends, or to consider situations where the optimal policy cannot be determined by only considering the most likely or desired outcome. Indeed, for IPPC 2011, it was a predominant strategy to opt for an algorithm that takes probabilities into account, with the PROST planner described in this paper coming out on top of the field, and Glutton (Kolobov et al. 2012) on a close second place.

Copyright © 2012, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

The contribution of this paper is a detailed description of the techniques that were implemented on top of the UCT (Kocsis and Szepesvári 2006) skeleton, a popular algorithm in planning and acting under uncertainty. We start by formalizing probabilistic planning as given by the fragment of RDDL that was used for IPPC 2011 in the next section. The basics underlying our planning system are sketched thereafter, including modifications of UCT to adapt the algorithm to the given circumstances, e.g., with respect to the search space, the search depth, or pruning of *superfluous* actions. We believe that one of the main reasons for the success of our planner is the *initialization* procedure that is described in the subsequent section. An initialization procedure gives UCT initial guidance to prevent random walks through the search space. This is followed by a discussion of a method to detect *reward locks*, states that can be regarded as *goals* or *dead ends*, and we briefly discuss how this information can be used to improve the system's behavior. We finish with an empirical evaluation of the presented methods before we conclude.

Probabilistic Planning

An MDP (Puterman 1994; Bertsekas and Tsitsiklis 1996) is a 4-tuple $\langle S, A, P, R \rangle$, where S is a finite set of *states*; A is a finite set of *actions*, including the noop action (denoted $a_{\varnothing}$); $P : S \times A \times S \rightarrow [0, 1]$ is the *transition function*, which gives the probability $P(s'|a, s)$ that applying action $a \in A$ in state $s \in S$ leads to state $s' \in S$; and $R : S \times A \times S \rightarrow \mathbb{R}$ is the *reward* function.

Schematic RDDL incorporates the semantics of a dynamic Bayesian network extended with an influence diagram utility node representing immediate rewards. We consider fully observable probabilistic planning with rewards and finite horizon as given by the fragment of RDDL used as the input language of IPPC 2011 when fully grounded (which is meant in the following when we mention RDDL). RDDL specifies a factored MDP (Boutilier, Dearden, and

Goldszmidt 2000; Guestrin et al. 2003), that is a tuple $T = \langle V, A, P, R, H, s_0 \rangle$, where the set of states S is induced by the set of binary[1] *state variables* V and the *remaining steps* $h \in \{0, \ldots, H\}$ as $S = 2^V \times \{0, \ldots, H\}$, where $H \in \mathbb{N}$ is the *finite horizon* and $s_0 \in S$ the *initial state*. With $s(v)$ and $s(h)$, we denote the value of variable v and the number of remaining steps h in state s. As the remaining steps must decrease by 1 in each transition, we demand $P(s'|a, s) = 0$ if $s'(h) \neq s(h) - 1$. States with $s(h) = 0$ are *terminal states*. In contrast to PPDDL, where changes are encoded as effects of actions, changes are encoded via *transition functions for variables* in RDDL. We write $P_v(s'(v) = \top|a, s)$ and $P_v(s'(v) = \bot|a, s)$ for the probability that applying action $a \in A$ in state $s \in S$ leads to some state s' where $s'(v) = \top$ and $s'(v) = \bot$, respectively. The transition functions for variables are probability distributions over state-action pairs, and the transition function satisfies the independence criterion (Degris, Sigaud, and Wuillemin 2006; Chakraborty and Stone 2011), i.e., it is induced by the transition functions for variables as the product of the probabilities of all variables.

RDDL also allows to model applicability conditions for actions by using so-called *state-action constraints*. As IPPC 2011 did not make use of this feature[2] we allow execution of all actions at any state (possibly without effects).

As usual, a solution for an MDP is a *policy*, i.e. a mapping from states to actions. Let

$$E(a, s) := \sum_{s' \in S} P(s'|a, s) \cdot R(s, a, s') \tag{1}$$

be the *expected immediate reward* of applying action a in state s. The *expected reward* of a policy π in MDP T with initial state s_0 is given by the *state-value function* V_π as $V_\pi(T) := V_\pi(s_0)$ with

$$V_\pi(s) := \begin{cases} 0 & \text{if } s \text{ is a terminal state} \\ Q_\pi(\pi(s), s) & \text{otherwise,} \end{cases} \tag{2}$$

where the *action-value function* $Q_\pi(a, s)$ is defined as

$$Q_\pi(a, s) := E(a, s) + \sum_{s' \in S} P(s'|a, s) \cdot V_\pi(s'). \tag{3}$$

Related to Equation 2, the *Bellman equation* (Bellman 1957; Bertsekas 1995) characterizes the expected reward of any state in an optimal policy.

Due to the exponential number of states in a factored MDP T, it is usually impractical to compute the expected reward of a policy π for T. Therefore, it is a common way to estimate $V_\pi(T)$ empirically by sampling a fixed number of *runs*, i.e., interactions with an environment that provides the simulated outcomes of executed actions as in IPPC 2011. Interleaving planning with execution additionally allows us to restrict ourselves to the generation of partial policies defined only for the current state, i.e., we repeatedly generate a partial policy π, execute $\pi(s_0)$, and update the current state s_0 to match the observed outcomes until a terminal state is reached.

[1]Note that our algorithms can be adapted easily to work with a framework extended to variables in finite-domain representation.

[2]IPPC 2011 made use of state-action constraints to check legality of concurrent actions. This was done in a static manner, though, i.e. a combination of actions was either always applicable or never.

UCT: Basic Algorithm

The PROST planning system is based on the *upper confidence bounds applied to trees* (UCT) algorithm (Kocsis and Szepesvári 2006), a state-of-the-art approach for many problems of acting under uncertainty, such as the two player game of Go (Gelly and Silver 2007; 2011), the nondeterministic single player game of Klondike solitaire (Bjarnason, Fern, and Tadepalli 2009) or the Canadian Traveler's Problem (Eyerich, Keller, and Helmert 2010), a stochastic path planning problem. To the best of our knowledge, it has not been applied to domain-independent probabilistic planning before IPPC 2011.

As an *anytime* algorithm, UCT returns a non-random decision whenever queried, and terminates based on a timeout given to the system as a parameter. In the given time, UCT performs *rollouts*, where, as usual in Monte-Carlo approaches, outcomes of actions are sampled according to their probability. The state transition graph of a factored MDP is a directed graph. In our case, where the remaining steps are part of the state, this graph is acyclic. UCT works on the tree that results from regarding the path that leads to a state as a part of the state. While this tree might be much larger than the original graph, especially if the graph contains a lot of transpositions, we can keep the tree's size moderate by eliminating superfluous actions as shown below. In our context, a *UCT node* n is a tuple $\langle s, a, N^k, R^k, \{n_1, \ldots, n_m\} \rangle$,where

- s is a (partial) state,
- a is an action,
- N^k is the number of rollouts among the k first rollouts where node n was chosen,
- R^k is the expected reward estimate based on the k first rollouts, and
- $\{n_1, \ldots, n_m\}$ are the successor nodes of node n.

In each rollout, the search tree is traversed from the root node n_0 with $n_0.s = s_0$ to a node that has a terminal state assigned. We distinguish between two kinds of nodes in the tree, *chance nodes* and *decision nodes*, and describe how to determine the path of a rollout in Algorithm 1.

A decision node n has exactly one successor node n_i for each action a_i with $n_i.a = a_i$. The attribute that distinguishes UCT most from other Monte-Carlo methods is that it bases the decision which action to take in a decision node on all previous rollouts: It favors successors that led to *high rewards* or have been *rarely tried* in previous rollouts. To do so, UCT distinguishes two cases: If a decision node has at least one successor node n_i that has never been selected, i.e., where $n_i.N^k = 0$, it randomly selects any of the unvisited successor nodes and applies the corresponding action. If all successor nodes have been visited at least once, UCT balances the classical trade-off between *exploitation* of previously good policies and *exploration* of little examined

Algorithm 1: UCT search with decision and chance nodes.

```
1  def search():
2      while not timeoutReached() do
3          rollout(n_0, s_0, ∅,∅, searchDepth)
4      return greedyPolicy(n_0)
5  def rollout(n, s, a, s', depth):
6      res = 0.0
7      if isDecisionNode(n) then
8          if hasUnvisSucc(n) then n', a = getUnvisSucc(n)
9          else n', a = getUCTSucc(n)
10         s' = applyAllDetSuccFunc(a, s)
11     else
12         n', s' = sampleNextProbSuccFunc(n, s, a, s')
13         if isChanceNodeLeaf(n) then
14             res = R(s, a, s')
15             s := s', s' := ∅, a := ∅, depth := depth −1
16             if depth = 0 then return res
17     res := res + rollout(n', s, a, s'depth)
18     updateNode(n,res)
19     return res
```

choices by picking the successor node n_i that maximizes the UCT formula

$$B\sqrt{\frac{\log n.N^k}{n_i.N^k}} + n_i.R^k$$

over all successor nodes n_i, where B is a bias parameter discussed later.

In a chance node n a successor is chosen by sampling the outcome n_i according to its transition probability $P(n_i.s|n_i.a, n.s)$. Even though this seems like a simple process, it raises some technical difficulties: As transition functions for variables are independent from each other, the number of outcomes and with it the number of successors of a chance node might be exponential in the number of variables. As an example where this has heavy influence take the SYSADMIN domain of IPPC 2011, where computers in a network are shut down with independent probabilities. Any chance node n where $n.s$ is the state where all computers are running has a successor for each state in the state space, as each state has a non-zero probability to be an outcome of any action leading to n. To avoid the need of a data structure of exponential size, we exploit the structure of RDDL that provides us with a separate transition probability for each variable and apply these *sequentially* similar to the way it is done when generating the forked normal form for probabilistic PDDL effects (Keller and Eyerich 2011). While this does not reduce the size of the search space, it drastically reduces the branching factor of chance nodes (each chance node has a branching factor of two, as depicted in Figure 1). Given a factored MDP, our technique generates at most $|V| \cdot H$ successors in chance nodes per rollout, while a standard technique generates up to $2^{|V|} \cdot H$. Note that the same technique can be used in decision nodes if concurrent actions are allowed, leading to a search tree where successors of decision nodes can be decision nodes themselves and where actions are scheduled sequentially. Due to the abundance of benchmarks with a high grade of concurrency, i.e., a high number of possible action combinations, this procedure did not lead to significantly different results.

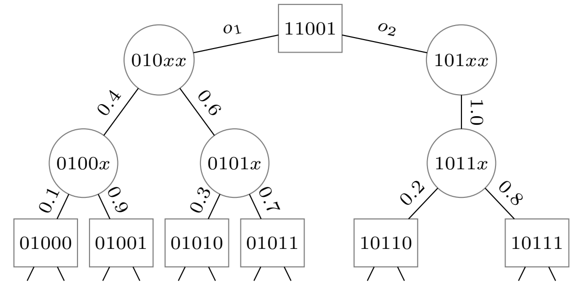

Figure 1: Example statespace with decision nodes (rectangles) and chance nodes (circles). An x in the valuation of state variables stands for an undetermined value in the state.

Bias Parameter Kocsis and Szepesvári (2006) show that UCT converges to the optimal policy in the limit, given a timeout that is sufficiently high. The analysis in the convergence proof suggests that the bias parameter B should be such that it grows linearly with the expected reward of an optimal policy. This is also desirable because it means that the policy is invariant to applying a scaling constant to the rewards. As the expected reward under an optimal policy is of course unavailable, we estimate it for the $(k + 1)$-th rollout by the expected reward estimate of the previous k rollouts for s_0. The fact that this is undefined for $k = 0$ does not affect the algorithm, as the parameter does not affect the choices of the first rollout anyway.

Regret Minimization The UCT formula is derived from the UCB1 algorithm, which minimizes the *regret* in a multi-armed bandit problem (Auer, Cesa-Bianchi, and Fischer 2002), i.e., the number of times a suboptimal arm is chosen. As a procedure that was developed in a machine learning context, a *generative model* of state transitions is sufficient for UCT: it does not reason over probabilities but achieves convergence of the state value function by choosing outcomes with the correct probability a sufficient number of times. For this reason, UCT is not perfectly suited for the circumstances we are facing in probabilistic planning, where we are provided with a *declarative model* of state transitions, i.e., with concrete probabilities, which are not directly used by our planning system when updating search nodes. It is also not clear if regret minimization is a good optimization criterion for an approach that has access to a declarative model. The experimental evaluation in this paper and the results of IPPC 2011 indicate that the algorithm is able to generate competitive policies in domain-independent probabilistic planning, though.

Search Depth Limitation While UCT converges towards the optimal policy in the limit, in practice it often needs a prohibitively large number of rollouts to converge. It does improve bit by bit, though, the more rollouts are made available. As we have little influence on the available time per decision, it is important to make sure that the provided time is spent properly. Decisions far in the future often influ-

ence the expected reward of a policy less than immediate actions, as the uncertainty grows with the number of simulated steps. Consider, for example, the ELEVATORS domain of IPPC 2011, where passengers who want to get from one floor to another arrive with a given probability and have to be served by opening and closing doors, choosing the right direction and moving the elevator. If a passenger arrives, the most influential part of good policies is to bring the passenger to their destination with the correct sequence of actions, while reasoning about other passengers who might or might not arrive some time in the distant future is less important as long as service for a passenger is not completed.

It is possible to exploit this by relaxing a factored MDP $T = \langle V, A, P, R, H, s_0 \rangle$ to $T^\lambda = \langle V, A, P, R, H^\lambda, s_0 \rangle$ with horizon $H^\lambda = \min(H, \lambda)$, where the *search depth limit* λ is given to the planner as a parameter. Our experiments show that the policy derived from UCT on T^λ is often better than a policy on T, as a higher number of rollouts is performed, and as the search space in the more important parts close to the root node is more thoroughly investigated. It must be considered, however, that this modification can also be the cause of problems, most notably that the better of two policies on T might be worse on T^λ, a problem that is discussed for the special case of *reward locks* later in this paper. This also removes the possibility to reuse the part of the search tree in the next iteration that corresponds to the outcome of the last submitted action, as search nodes might have been updated based on assigned states with a different horizon.

Reasonable Actions It is quite often the case that domains are modeled in a way where actions only affect the state if certain circumstances are given, similar to conditional effects in PPDDL. In the ELEVATORS domain, for example, there is no difference if we choose to close the door in a state where it is already closed, or if we choose to apply $a_\varnothing$. It is worth to look into this in more detail, as each action that can safely be excluded leads to a lower branching factor, an important criterion not only for UCT. For this reason, we exclude an action from the search space if there is a obviously better decision:

Definition 1. *Action a* dominates *action a' in state s (written $a >_s a'$) if $P(s'|a, s) = P(s'|a', s)$ and $R(s, a, s') \geq R(s, a', s')$ for all states $s' \in S$, and $R(s, a, s'') > R(s, a', s'')$ for at least one $s'' \in S$.*

Theorem 1. *Let T be a MDP, and a, a' and s such that $a >_s a'$. For each policy π' on T with $\pi'(s) = a'$ there is a policy π with $V_\pi(T) > V_{\pi'}(T)$.*

Proof sketch: Given a policy π' with $\pi'(s) = a'$ and $a >_s a'$, we construct π such that $\pi(s') = \pi'(s')$ for all $s' \neq s$ and $\pi(s) = a$. Then, with Equations 1 to 3, $V_\pi(T) > V_{\pi'}(T)$, as $E(\pi(s'), s') \geq E(\pi'(s'), s')$ for all $s' \neq s$ and $E(\pi(s), s) > E(\pi'(s), s)$. ■

We can also prune an action if there is no obviously better alternative, but one that is obviously equally good:

Definition 2. *Two actions a and a' are* equivalent *in state s (written $a =_s a'$) if $P(s'|a, s) = P(s'|a', s)$ and $R(s, a, s') = R(s, a', s')$ for all states $s' \in S$.*

Theorem 2. *Let T be a MDP, and a, a' and s be such that $a =_s a'$. For each policy π on T with $\pi(s) = a$ there is a policy $\pi' \neq \pi$ with $V_{\pi'}(T) = V_\pi(T)$.*

Proof sketch: Consider policy $\pi' \neq \pi$ with $\pi'(s') = \pi(s')$ for all $s' \neq s$ and $\pi'(s) = a'$. Then, with Equations 1 to 3, obviously $V_{\pi'}(T) = V_\pi(T)$. ■

While it is clear in the case of action dominance which actions are *superfluous*, action equivalence is a symmetric relation. For this reason, we arbitrarily choose one element from the equivalence class with respect to the equivalence relation given in Definition 2, and declare all other actions in that class as superfluous. Actions that are not superfluous are called *reasonable*.

Q-value Initialization

The UCT algorithm chooses the successor node that maximizes the UCT formula in decision nodes only if all successor nodes have already been visited (Kocsis and Szepesvári 2006). If there is at least one reasonable action that has not been applied, UCT chooses one of the unvisited successor nodes uniformly randomly. As described so far, it does not take into account any problem specific information that would bias the rollouts towards promising parts of the search space, but follows a random policy initially. This weakness has been observed in several areas of planning and acting under uncertainty, and is typically addressed by introducing a *initialization* that assigns a quality estimate to unvisited children of decision nodes, as *blind* UCT would require a prohibitively large number of rollouts to converge to a good policy. As each initialized successor node contains information that corresponds to an action-value function (or Q-value) estimate, we refer to this as a *Q-value initialization*. Publications on this topic range from general game playing (Finnsson and Björnsson 2011) over specific game solvers like the most successful player for Go (Gelly and Silver 2007; 2011) to domain-specific planning problems like the highly probabilistic Canadian Traveler's Problem (Eyerich, Keller, and Helmert 2010). While there is no reason why initializations should not be applicable to domain-independent probabilistic planning, there is no obvious way how to do so either, as the mentioned techniques are tailored to their specific use-cases: Finnsson and Björnsson (2010) compare several search control techniques that resemble initializations in their general game player Cadia Player, winner of the AAAI 2008 general game playing competition, including, e.g., *Rapid Action Value Estimation*. This technique has also been applied in the context of Go by Gelly and Silver (2007), who compare it to an offline-computed initialization based on *Temporal Difference Learning*, and Eyerich, Keller, and Helmert (2010) initialize with an *optimistic* estimation based on *Dijkstra's* algorithm.

All these techniques have in common that they gain information about the problem by relaxing it, which is used to guide UCT to promising parts in the search tree. For the purpose of this paper, we regard a Q-value initialization as a function $I : S \times A \to \mathbb{R}$, and we initialize the successors of decision node n by setting $n_i.R^k = I(n.s, n_i.a)$, and $n_i.N^k = \delta$ for all successor nodes n_i, where the *num-*

ber of virtual rollouts δ is a parameter given to the system. While initializing nodes with any function I does not alter the convergence result of UCT in the limit, faster convergence on average is only achieved if $I(s,a)$ has some degree of *informativeness*, and if the obtained information is worth more than the additional (uninformed) rollouts that would have been possible without the computational overhead. In PROST, we use a Q-value initialization that is based on a *single-outcome determinization* of the MDP that is generated by expecting that the *most likely outcome* occurs:

Definition 3. *The* most likely outcome *of applying action* $a \in A$ *in state* s *is the state* $s^+(a,s)$*, where*

$$s^+(a,s)(v) = \begin{cases} \top & \text{if } P_v(s'(v) = \top | a, s) \geq 0.5 \\ \bot & \text{otherwise} \end{cases}$$

and $s^+(a,s)(h) = s(h) - 1$*. The* most likely determinization *of the factored MDP* $T = \langle V, A, P, R, H, s_0 \rangle$ *is* $T^+ = \langle V, A, P^+, R, H, s_0 \rangle$*, where*

$$P^+(s'|a,s) = \begin{cases} 1 & \text{if } s' = s^+(a,s) \\ 0 & \text{otherwise} \end{cases}$$

It should be mentioned that the same determinization is also used in FF-Replan (Yoon, Fern, and Givan 2007), albeit in a completely different way. The advantage of simplifying the MDP by removing the uncertainty is that we can use a computationally more expensive search algorithm to initialize. In PROST, we have implemented a *depth first search* (DFS) procedure, which maximizes the reward that can be achieved in the determinization in a given number of steps. As it is not desirable to use a constant search depth in all domains, we extend this approach to an *iterative deepening search* (IDS) with maximal search depth D that is described in Algorithm 2. We determine a good value for D in a simple learning procedure on a set of arbitrarily generated, reachable states, which is preformed as a preprocess. It is even possible that a maximal search depth of zero is chosen in the preprocess, with the result that the initialization is aborted.

During initialization, which is shown in Algorithm 2, we additionally check if a *satisfying degree of informativeness* has been reached in the previous iteration. We approximate the degree of informativeness by checking if an action is regarded as better than doing nothing, i.e. if there is an action $a \in A$ where $I(s,a) > I(s,a_\varnothing)$. The reason why we regard this as an appropriate metric for measuring the degree of informativeness is best explained with an example, in which we ignore the remaining steps in states for the sake of simplicity. Consider a determinization where $a_\varnothing$ does not change the initial state s_0 (as given in all IPPC 2011 domains). A sequence of actions starting with $a_\varnothing$ will always be one step behind the optimal sequence of actions, as that optimal sequence can simply be applied after $a_\varnothing$ (as it did not change the initial state). If, for example, the reward function always yields 0 or 1, and the optimal sequence of actions $a^\star, a_2, \ldots, a_D$ leads to rewards $1 + \ldots + 1 = D$ for search depth D, then DFS for $a_\varnothing$ on the same state will lead to sequence $a_\varnothing, a^\star, a_2, \ldots, a_{D-1}$ with rewards $0{+}1{+}\ldots{+}1 = D{-}1$. As we normalize the result, i.e., initialize the action-value function with $I(s,a) = \text{IDS}(s,a) \cdot \frac{s(h)}{D}$,

Algorithm 2: Q-value initialization based on IDS.

```
1  def initialize(s,a):
2      depth := 0, result := −∞
3      while not terminate() do
4          depth := depth+1
5          result := dfs(s, a, depth) ·s(h)/ depth
6      return result
7  def terminate(depth):
8      if resultIsInformative() then return true
9      return depth = maxDepth
10 def dfs(s, a, depth):
11     s′ := apply(s, a)
12     res := R(s, a, s′)
13     if depth = 0 then return res
14     if depth = 1 and earlyTerminationPossible() then
15         return res +R(s′, a∅, ∅)
16     bestRew := −∞
17     for a′ ∈ A do
18         futRew := dfs(s′, a′, depth − 1)
19         if futRew > bestRew then bestRew := futRew
20     return res+bestRew
```

this will lead to estimates of $I(s,a^\star) = D \cdot \frac{s(h)}{D} = s(h)$ and $I(s,a_\varnothing) = (D-1) \cdot \frac{s(h)}{D}$. As the impact of the initialization is largest if the *relative* difference between sibling nodes is largest, we are interested to terminate IDS in depth 1, where $I(s,a^\star) = s(h)$ and $I(s,a_\varnothing) = 0$. Even though this procedure is prone to overestimate local maxima, it is both faster than a search to the maximal depth and leads to better results on the IPPC 2011 benchmarks.

The last search layer in DFS can be skipped if the reward function is such that $R(s,a,s') = R(s,a,s'')$ for all $s, s', s'' \in S$ and $a \in A$, which is given for all IPPC 2011 domains, and if $R(s,a,s') \leq R(s,a_\varnothing,s')$ for all $s,s' \in S$ and action $a \neq a_\varnothing$, which is given if the reward depends on actions only in the form of costs. Note that, even though this might seem negligible at first glance, it saves half the computational effort for a constant branching factor, and basically turns a DFS of depth D to one of depth $D-1$.

Reward Locks

The search depth limitation presented earlier in this simplifies an MDP T to an MDP with limited horizon T^λ. While this leads to a more thoroughly investigated search space close to the current state, a policy that is optimal in T^λ is not necessarily optimal in T.

We illustrate this with an example: Let T be an MDP with horizon $H = 20$, where the decision which of two operators a_1 and a_2 is applied in s_0 determines the reward of the whole run. Choosing a_1 leads to sequence of states $s_1^+, \ldots, s_{20}^+$ with a probability of 0.6, incurring a reward of 1 in each transition, and to a sequence of states $s_1^-, \ldots, s_{20}^-$ with probability 0.4 with reward of 0 in each transition. Executing a_2 always leads to the sequence of states $s_1, \ldots, s_{20}$, incurring a reward of 0 in the first 5 transitions, and a reward of 1 thereafter. The expected reward in T of policy π with $\pi(s_0) = a_1$ is $V_\pi(T) = 12$, and $V_{\pi^\star}(T) = 15$ for

Algorithm 3: Reward Lock Detection.

```
1  isARewardLock(s):
2      s' := apply(a_∅, s)
3      r = R(s, a_∅, s')
4      return checkRewardLock(s, r)
5  checkRewardLock(s, r):
6      add s to closed list
7      if s(h) = 0 then return true
8      for a ∈ A do
9          s' := s^k(a, s)
10         if s' not in closed list then
11             if r^k(s, a, s') ≠ r then return false
12             if r^k(s, a, s') = u then return false
13             if not checkRewardLock(s', r) then return false
14     return true
```

policy $\pi^\star$ with $\pi^\star(s_0) = a_2$. In T^λ, where the search depth is limited to $\lambda = 10$, policy π is the better policy: The average reward per transition does not change for policy π, as it does not depend on the length of the state sequence, and therefore $V_\pi(T^\lambda) = 6$. This is different for policy π', where $V_{\pi^\star}(T^\lambda) = 5$ when only accounting for the states $s_1, \ldots, s_{10}$.

It is, in general, not possible to eliminate the risk of biased estimates completely when using a relaxation T^λ with limited search depth $\lambda < H$ to generate a policy for a factored MDP T. Nevertheless, our experiments show the potential of search depth limitation on several benchmarks. For this reason, we present a method to detect special cases called *reward locks*, where it is likely that we overestimate some policy and underestimate another:

Definition 4. *A set of states $S^l \subseteq S$ is a* reward lock *with reward $r^\star(S^l)$, if $R(s, a, s') = r^\star(S^l)$ for all $s \in S^l$, $a \in A$ and $s' \in S$ with $P(s'|a, s) > 0$, and if $s' \in S^l$ for all $s \in S^l$ if $P(s'|a, s) > 0$.*

In other words, a reward lock is a state where, no matter which action is applied and which outcome occurs, we end up in a state where we receive the same reward as before, and which is also a reward lock. We talk about *goals* if $r^\star(S^l) = \max_{s,a,s'} R(s, a, s')$, and about a *dead end* if $r^\star(S^l) = \min_{s,a,s'} R(s, a, s')$. Note that dead ends in our context do not render policies improper as in infinite horizon MDPs, and goals do not terminate a run prematurely – both incur the minimal or maximal reward until a terminal state is reached. As we have no influence on future rewards once a reward lock state is reached, we can profit in two ways from a method that detects this kind of states: We can stop a rollout as soon as we encounter a reward lock, as all successor states that are reachable under any policy will yield the same reward, i.e., we can calculate the total reward by multiplying the remaining steps with the reward of the reward lock. More importantly, we can react appropriately to minimize the error incurred by the limited search depth by simulating the horizon of T rather than T^λ. The question which reaction is appropriate is not answered for good in this paper but left open for future research. Possible reactions include switching back to searching on T rather than T^λ as soon as a reward lock is encountered; or to actively search for goals initially, and use a search strategy that applies any of the well studied, goal-oriented heuristics for classical planning (e.g., Helmert and Domshlak 2009). The technique used in the experiments described in the next section is a much simpler approach that punishes dead ends and rewards goals by regarding rollouts that end in a reward lock as if the used horizon was H (even though we continue to search in T^λ), essentially giving states in reward locks a higher weight. This is both useful to guide the search towards goal states, and to avoid dead ends.

Note that the reward lock detection is not restricted to be used in UCT. Especially algorithms that label states as solved as, e.g., Labeled Real-Time Dynamic Programming (Bonet and Geffner 2003), can profit immensely from using a reward lock detection technique, as state-value functions of reward lock states can be calculated exactly the first time such a state is expanded, and can therefore be labeled as solved immediately.

Deciding whether a state is a reward lock is very hard, though: Consider the PSPACE-complete plan existence problem in classical planning (Bylander 1994), which can be reduced to the problem of detecting that the initial state of a factored MDP where all non-goal states yield reward 0 and all goal states a reward 1 is not part of a reward lock. For this reason, we use a procedure that approximates a solution in an incomplete but sound way: We use the three-valued Kleene logic (Kleene 1950) with values $\perp$, $\top$ and u (unknown) with their usual semantics, and check for more states than necessary if they are part of a reward lock. As we interpret u as "the value is either $\top$ or $\perp$", we calculate rewards and transition functions on sets of states rather than on a potentially exponential number of states separately.

Definition 5. *The* Kleene outcome *of applying action $a \in A$ in state s is the state $s^k(a, s)$, where*

$$s^k(a,s)(v) = \begin{cases} \top & \text{if } P_v(s'(v) = \top | a, s) = 1 \\ \perp & \text{if } P_v(s'(v) = \perp | a, s) = 1 \\ u & \text{otherwise,} \end{cases}$$

and $s^k(a, s)(h) = s(h) - 1$. The Kleene logic relaxation *of the factored MDP $T = \langle V, A, P, R, H, s_0\rangle$ is $T^k = \langle V^k, A, P^k, R^k, H, s_0\rangle$, where V^k is a set with $v^k \in V^k$ for each $v \in V$, and where*

$$P^k(s'|a,s) = \begin{cases} 1 & \text{if } s' = s^k(a,s) \\ 0 & \text{otherwise.} \end{cases}$$

For the calculation of rewards with states in Kleene logic, the variable-based structure of RDDL is exploited: If it depends on a state variable with value u, the reward is also as unknown, and $R^k(s, a, s') = R(s, a, s')$ otherwise. Given this framework, we can apply successor functions and calculate rewards on states in T^k, which corresponds to doing so on sets of states in T. Algorithm 3 shows the procedure in detail. It is invoked in PROST in every state that is encountered in a rollout. The *reference reward* r is calculated

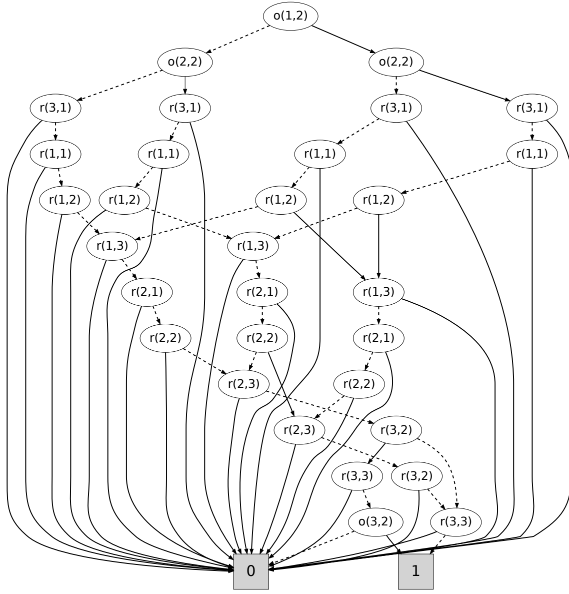

Figure 2: Deadend BDD in CROSSING TRAFFIC instance 1. r(m,n) stands for `robot-at(`x_m`,`y_n`)`, o(m,n) for `obstacle-at(`x_m`,`y_n`)`. A state is a deadend if the robot is at no location or at the same location as an obstacle.

by the algorithm as the reward of applying $a_\varnothing$ in the state that is currently checked and leading to an arbitrary successor state. This is possible because all actions and outcomes must lead to a state with the same reward, so it does not matter which are used for computation of the reference reward.

Even though the closed list that is used in the procedure does not distinguish between states with different remaining steps, the state s that is given recursively as an argument does. Therefore, it can easily be seen that the function is called recursively at most $s(h)$ times, as the remaining steps decrease by one in every recursion. Moreover, as all methods invoked from this procedure are computable in polynomial time, the algorithm is polynomial: The generation of successor states in Kleene logic is as complex as the generation of successors in binary logic by definition. The combination of two states s_1, s_2 in Kleene logic, which is used when a state is added to the closed list, is the state s' where

$$s'(v) = \begin{cases} \bot & \text{if } s_1(v) = \bot \text{ and } s_2(v) = \bot \\ \top & \text{if } s_1(v) = \top \text{ and } s_2(v) = \top \\ u & \text{otherwise,} \end{cases}$$

which is clearly polynomial as well. Finally, the function that checks if a state is in the closed list must compare to at most $s(h)$ other states in Kleene logic.

Even though the presented procedure checks the reward on more states than necessary for equality, and even though it might consider rewards as different even though they are equal, it is able to detect all reward locks that are present in the domains of IPPC 2011. In practice, we speed up this process by saving reward locks that have been detected once in a BDD, and additionally check for all states s' in Algorithm 3 if they are already in the BDD assigned to the current reference reward $r^\star$. A BDD that was created by PROST, the BDD describing dead ends in instance 1 of the CROSSING TRAFFIC domain, is shown in Figure 2. Closer examination shows that the planner is able to reliably detect reward locks in reasonable time.

Experimental Evaluation

To evaluate our planner empirically, we perform experiments on the IPPC 2011 benchmark suite which consists of all RDDL domains that are currently publicly available. We also use the competition's setting of 30 trials for each of the 10 instances in 8 domains that have to be completed within a total time limit of 24 hours.

Rather than presenting average rewards, we show normalized scores that are computed in the same way as in IPPC 2011, including those of Glutton (Kolobov et al. 2012), which finished close behind PROST at the competition, and incorporates a Real Time Dynamic Programming (Barto, Bradtke, and Singh 1995) approach. We evaluate several versions of PROST with different parameter settings. To distinguish between different versions of our planner, we denote the versions with

- $\mathbf{P}^0$ if reasonable action pruning is *not* used,
- $\mathbf{P}_\lambda$ if search depth limitation is set to λ,
- $\mathbf{P}^I$ if initialization is enabled, and
- $\mathbf{P}^R$ if reward lock detection is enabled.

In runs where initialization is enabled, we use $\delta = 5$ virtual rollouts, and for search depth limitation we use $\lambda = 15$. Both numbers were determined empirically.

To measure the influence of reasonable action pruning, we compare results from $\mathbf{P}^0$ and $\mathbf{P}$, which only differ in the use of action pruning. Closing or opening doors are superfluous actions in certain situations in the ELEVATORS domain. Nevertheless, the normalized scores do not differ much, a fact that is probably biased due to the comparably bad results. In CROSSING TRAFFIC and NAVIGATION we can prune those actions that move the agent into the boundaries of the gridmap. Effects of reasonable action pruning can especially be seen in NAVIGATION, where the normalized score is approximately doubled. The domain with the most superfluous actions, RECON, shows the biggest improvement in terms of normalized score: from 0.00 to 0.40. Combined with the fact that enabling action pruning does not diminish the results on any domain in a statistically significant way, this experiment clearly shows the potential of reasonable action pruning.

There are three pairs of PROST versions that only differ in the use of search depth limitation: $\mathbf{P}$ / $\mathbf{P}_{15}$, $\mathbf{P}^I$ / $\mathbf{P}^I_{15}$, and $\mathbf{P}^{I,R}$ / $\mathbf{P}^{I,R}_{15}$. It can clearly be seen that all versions that make use of search depth limitation either achieve better or similar results on all domains besides CROSSING TRAFFIC and NAVIGATION. The strength of limiting the search depth is especially apparent in ELEVATORS, GAME OF LIFE, SYSADMIN and TRAFFIC. Surprisingly, in three of these domains the version of PROST that only limits the search depth and

	CROSSING	ELEVATORS	GAME	NAVIGATION	RECON	SKILL	SYSADMIN	TRAFFIC	TOTAL
$\mathbf{P}^0$	0.46	0.01	0.86	0.14	0.00	0.89	0.86	0.98	0.53 ± 0.09
$\mathbf{P}$	0.51	0.04	0.91	0.27	0.40	0.90	0.86	0.96	0.61 ± 0.08
$\mathbf{P}_{15}$	0.56	0.01	**0.95**	0.30	0.46	0.91	**0.91**	**0.99**	0.63 ± 0.08
$\mathbf{P}^I$	0.84	0.86	0.88	0.65	**0.98**	0.94	0.82	0.84	0.85 ± 0.05
$\mathbf{P}^I_{15}$	0.83	0.93	0.91	0.57	**0.98**	**0.95**	0.88	0.93	0.87 ± 0.05
$\mathbf{P}^{I,R}$	**0.98**	0.85	0.86	0.71	**0.98**	0.89	0.80	0.83	0.86 ± 0.04
$\mathbf{P}^{I,R}_{15}$	0.91	**0.94**	0.92	0.67	0.97	0.92	0.86	0.94	$\mathbf{0.89} \pm 0.04$
Glutton	0.80	0.90	0.67	**0.97**	0.76	0.86	0.34	0.67	0.75 ± 0.06

Table 1: Experimental Results. The scores and the 95% confidence intervales are calculated following the schema of IPPC 2011, and the results of Glutton are taken from IPPC 2011. The best result in each domain and in total are in bold.

does not use any other technique presented in this paper (besides reasonable action pruning) is even the best-performing planner. Lastly, while the total results seem to indicate that search depth limitation is not worth the bias we discussed in this paper, this is only due to the decreased performance in the domains containing reward locks. When ignoring these domains, the total results also improve significantly.

This does, however, not come by surprise. We discussed the reasons in the previous Section, and presented reward lock detection as a potential solution. Our experiments back up our expectations empirically: We achieve significantly better results in CROSSING TRAFFIC and NAVIGATION with $\mathbf{P}^{I,R}$, and even though some of this gain is lost in $\mathbf{P}^{I,R}_{15}$, it still yields better or comparable results than $\mathbf{P}^I$ in all domains. This is especially important as it shows that using both search depth limitation and reward lock detection outperforms using neither, and it also seems that we managed to find the right balance between the quite contrary methods.

If we regard reward lock detection on its own, the results confirm that it is an effective approach for problems with dead ends or goals: We achieve the best normalized score in the CROSSING TRAFFIC domain, and only Glutton outperforms $\mathbf{P}^{I,R}$ in NAVIGATION. We think the improvement in the two domains that actually contain reward locks show the prospects the technique entails. Note that we could have easily created a version of PROST that combines the results of $\mathbf{P}^{I,R}$ in CROSSING TRAFFIC and NAVIGATION with those of $\mathbf{P}^{I,R}_{15}$ on the other domains by, for instance, switching to $\mathbf{P}^{I,R}$ upon detecting any reward lock.

The last approach discussed in this paper is the initialization method, which has the biggest influence on the scores. It is enabled in all of the four presented versions that outperform Glutton in terms of total score. In all domains but GAME OF LIFE, SYSADMIN and TRAFFIC the initialization leads to incomparably better results, increasing the normalized total score by more than 0.2. The trade-off between informativeness and computational overhead appears to be in balance: all domains where we find a good policy without the help of an initialization, i.e., those where it is most likely that the computational overhead dominates the information gain, achieve only slightly worse results. Those domains where long sequences of actions are necessary to achieve high rewards, ELEVATORS, RECON, and, with the aforementioned weaknesses regarding search depth limitation, CROSSING TRAFFIC and NAVIGATION, profit immensely from the determinization-based initialization. Moreover, the results indicate that a blind UCT version of PROST would not have been able to win IPPC 2012.

Concluding Remarks

In this paper, we have shown how the remarkable performance that the UCT algorithm has demonstrated in other areas of planning and acting under uncertainty can be reflected in domain-independent probabilistic planning. We have shown how to adapt the algorithm to the special characteristics given in the context of stochastic planning, like a strongly connected search space that needs to be created carefully, or reasonable action pruning that is based on actions that are equivalent or dominated. We furthermore discussed search depth limitation, an extension that incurs theoretical flaws with regard to good policies but yields promising results in the empirical evaluation.

We have shown how reward locks, states that can be regarded as goals or dead ends, can be detected, and discussed briefly how that information could be exploited. We showed experimentally that even a simple approach to use this information is able to increase the planner's performance in domains where reward locks are present. Another contribution of this paper is the Q-value initialization that gives UCT initial guidance, preventing the algorithm from performing random walks in the search space initially. It is based on an iterative deepening search, and is automatically adapted to the given circumstances in order to find a good trade-off between informativeness and additional computational effort. In an empirical evaluation, resembling IPPC 2011, we have shown the influence of the presented search enhancements, combining them to a framework that allows UCT to develop its full power.

Acknowldegements

This work was supported by European Communitys Seventh Framework Programme [FP7/2007-2013] as part of the CogX project (215181), and by the German Aerospace Center (DLR) as part of the Kontiplan project (50 RA 1010).

References

Auer, P.; Cesa-Bianchi, N.; and Fischer, P. 2002. Finite-time Analysis of the Multiarmed Bandit Problem. *Machine Learning* 47:235–256.

Barto, A. G.; Bradtke, S. J.; and Singh, S. P. 1995. Learning to Act Using Real-Time Dynamic Programming. *Artificial Intelligence (AIJ)* 72(1–2):81–138.

Bellman, R. 1957. *Dynamic Programming*. Princeton University Press.

Bertsekas, D., and Tsitsiklis, J. 1996. *Neuro-Dynamic Programming*. Athena Scientific.

Bertsekas, D. 1995. *Dynamic Programming and Optimal Control*. Athena Scientific.

Bjarnason, R.; Fern, A.; and Tadepalli, P. 2009. Lower Bounding Klondike Solitaire with Monte-Carlo Planning. In *Proceedings of the 19th International Conference on Automated Planning and Scheduling (ICAPS)*, 26–33.

Bonet, B., and Geffner, H. 2003. Labeled RTDP: Improving the Convergence of Real-Time Dynamic Programming. In *Proceedings of the 13th International Conference on Automated Planning and Scheduling (ICAPS)*, 12–31.

Boutilier, C.; Dearden, R.; and Goldszmidt, M. 2000. Stochastic Dynamic Programming with Factored Representations. *Artificial Intelligence (AIJ)* 121(1–2):49–107.

Bylander, T. 1994. The Computational Complexity of Propositional STRIPS Planning. *Artificial Intelligence (AIJ)* 69:165–204.

Chakraborty, D., and Stone, P. 2011. Structure Learning in Ergodic Factored MDPs without Knowledge of the Transition Function's In-Degree. In *Proceedings of the 28th International Conference on Machine Learning (ICML)*, 737–744.

Degris, T.; Sigaud, O.; and Wuillemin, P.-H. 2006. Learning the Structure of Factored Markov Decision Processes in Reinforcement Learning Problems. In *Proceedings of the 23rd International Conference on Machine Learning (ICML)*, 257–264.

Eyerich, P.; Keller, T.; and Helmert, M. 2010. High-Quality Policies for the Canadian Traveler's Problem. In *Proceedings of the 24th Conference on Artificial Intelligence (AAAI)*, 51–58.

Finnsson, H., and Björnsson, Y. 2010. Learning Simulation Control in General Game-Playing Agents. In *Proceedings of the 23rd Conference on Artificial Intelligence (AAAI)*, 954–959.

Finnsson, H., and Björnsson, Y. 2011. CadiaPlayer: Search Control Techniques. *KI Journal* 25(1):9–16.

Gelly, S., and Silver, D. 2007. Combining Online and Offline Knowledge in UCT. In *Proceedings of the 24th International Conference on Machine Learning (ICML)*, 273–280.

Gelly, S., and Silver, D. 2011. Monte-Carlo Tree Search and Rapid Action Value Estimation in Ccomputer Go. *Artificial Intelligence (AIJ)* 175:1856–1875.

Guestrin, C.; Koller, D.; Parr, R.; and Venkataraman, S. 2003. Efficient Solution Algorithms for Factored MDPs. *Journal of Artificial Intelligence Research (JAIR)* 19:399–468.

Helmert, M., and Domshlak, C. 2009. Landmarks, Critical Paths and Abstractions: What's the Difference Anyway? In *Proceedings of the 19th International Conference on Automated Planning and Scheduling (ICAPS)*.

Keller, T., and Eyerich, P. 2011. A Polynomial All Outcomes Determinization for Probabilistic Planning. In *Proceedings of the 21st International Conference on Automated Planning and Scheduling (ICAPS)*, 331–334. AAAI Press.

Kleene, S. C. 1950. Introduction to Metamathematics.

Kocsis, L., and Szepesvári, C. 2006. Bandit Based Monte-Carlo Planning. In *Proceedings of the 17th European Conference on Machine Learning (ECML)*, 282–293.

Kolobov, A.; Dai, P.; Mausam; and Weld, D. 2012. Reverse Iterative Deepening for Finite-Horizon MDPs with Large Branching Factors. In *Proceedings of the 22nd International Conference on Automated Planning and Scheduling (ICAPS)*. To Appear.

Little, I., and Thiébaux, S. 2007. Probabilistic Planning vs Replanning. In *ICAPS Workshop International Planning Competition: Past, Present and Future*.

Puterman, M. 1994. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley.

Sanner, S. 2010. Relational Dynamic Influence Diagram Language (RDDL): Language Description.

Teichteil-Königsbuch, F.; Infantes, G.; and Kuter, U. 2008. RFF: A Robust, FF-Based MDP Planning Algorithm for Generating Policies with Low Probability of Failure. In *Proceedings of the 6th IPC at ICAPS*.

Yoon, S. W.; Fern, A.; and Givan, R. 2007. FF-Replan: A Baseline for Probabilistic Planning. In *Proceedings of the 17th International Conference on Automated Planning and Scheduling (ICAPS)*, 352–360.