Journal of Artificial Intelligence Research 24 (2005) 851-887 Submitted 08/05; published 12/05

The First Probabilistic Track of the International Planning Competition

Håkan L. S. Younes LORENS@CS.CMU.EDU
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213 USA

Michael L. Littman MLITTMAN@CS.RUTGERS.EDU
David Weissman DWEISMAN@CS.RUTGERS.EDU
John Asmuth JASMUTH@CS.RUTGERS.EDU
Department of Computer Science
Rutgers University
Piscataway, NJ 08854 USA

Abstract

The 2004 International Planning Competition, IPC-4, included a probabilistic planning track for the first time. We describe the new domain specification language we created for the track, our evaluation methodology, the competition domains we developed, and the results of the participating teams.

1. Background

The Fourth International Planning Competition (IPC-4) was held as part of the International Conference on Planning and Scheduling (ICAPS'04) in Vancouver, British Columbia in June 2004. By request of the ICAPS'04 organizers, Sven Koenig and Shlomo Zilberstein, we were asked to create the first probabilistic planning track as part of IPC-4.

The overriding goal of the first probabilistic planning track was to bring together two communities converging on a similar set of research issues and aid them in creating comparable tools and evaluation metrics. One community consists of Markov decision process (MDP) researchers interested in developing algorithms that apply to powerfully expressive representations of environments. The other consists of planning researchers incorporating probabilistic and decision theoretic concepts into their planning algorithms. Cross fertilization has begun, but the intent of the probabilistic planning track was to create a set of shared benchmarks and metrics that could crystallize efforts in this area of study.

We created a new domain description language called PPDDL1.0, described in Section 2. PPDDL stands for "Probabilistic Planning Domain Definition Language", in analogy to PDDL (McDermott, 2000), which was introduced in IPC-1. PPDDL is modeled on PDDL2.1 (Fox & Long, 2003), the domain-description language for deterministic domains used in IPC-3. Syntactically, this language has a STRIPS/ADL-like flavor, but includes probabilistic constructs. To focus the energy of participants on issues of dealing with uncertainty, we chose not to include constructs for durative actions in PPDDL1.0.

By basing the domain-description language on PDDL, we sought to remain in the spirit of the existing planning competition, which we hope will further bring the communities together. The PPDDL representation itself is relational. Although representations with explicit objects are not a traditional feature of MDP-based domain-description languages, algorithms that exploit these features have begun to appear. We expected participants to propositionalize the domains before running their planning algorithms and, for the most part, they did so.

A fully functional parser for PPDDL was provided to participants in C++ in the form of a plan validator and very simple planner. Some basic tools to convert PPDDL to a decision-diagram representation were also provided. In many ways, handling the rich constructs of PPDDL was the main hurdle for many participants and we tried to provide as much assistance as we could on this dimension.

Although PPDDL supports numerical fluents, this feature was not used to its fullest extent in the competition. Numerical quantities were used only for representing reward values, and reward effects were required to be additive.

Since the classical track is well established at this point, it is helpful to contrast how the probabilistic track differs. The defining difference, of course, is that actions can have uncertain effects. That is, a "pickup" action in a Blocksworld might behave differently on different occasions, even from the same state. This single difference has a number of significant consequences. First, the optimal action choices for reaching a goal may be a function of the probabilistic outcomes along the way—a single sequence of actions may not be sufficient. As a result, it can be difficult to output a "plan". For this reason, we decided not to separate plan synthesis and execution into two phases, but instead evaluated planners online. Second, because of the unpredictability of effects, even an optimal plan for reaching a goal may get "unlucky" and fail with some probability. For this reason, we evaluated each planner multiple times on each problem and did not include a separate "optimal planner" track. In addition, since some planners may fail to reach the goal state for some executions, we needed a way of trading off between goal attainment and action cost. We decided to score an execution as goal reward minus action cost and chose a goal reward for each problem. Section 3 describes the evaluation methodology in further detail.

In total, we designed eight domains for the competition (Section 4). Some domains were simply noisy versions of classical planning domains, while other domains were designed specifically to thwart greedy replanning approaches that ignore uncertainty.

Ten planners from seven different groups entered the competition. The results of the competition are presented in Section 5. A deterministic replanner performed best overall, primarily due to a disproportionate number of noisy classical planning problems in the evaluation suite. Some domains proved challenging for all participating planners. These latter domains could serve as a basis for future probabilistic planning competitions.

2. Probabilistic PDDL

This section describes the input language, PPDDL1.0, that was used for the probabilistic track. PPDDL1.0 is essentially a syntactic extension of Levels 1 and 2 of PDDL2.1 (Fox & Long, 2003). The complete syntax for PPDDL1.0 is given in Appendix A. We assume that the reader is familiar with PDDL2.1, so we focus on the new language features, which include probabilistic effects and rewards. A more detailed account of PPDDL1.0 is provided

Name	Type	Init 1	Init 2
$\textit{bomb-in-package}_{\text{package1}}$	boolean	true	false
$\textit{bomb-in-package}_{\text{package2}}$	boolean	false	true
toilet-clogged	boolean	false	false
bomb-defused	boolean	false	false

Table 1: State variables and their initial values for the “Bomb and Toilet” problem.

by Younes and Littman (2004). The semantics of a PPDDL1.0 planning problem is given in terms of a discrete-time Markov decision process (Howard, 1960, 1971; Puterman, 1994).

2.1 Probabilistic Effects

To define probabilistic and decision theoretic planning problems, we need to add support for probabilistic effects. The syntax for probabilistic effects is

$$\texttt{(probabilistic } p_1\ e_1\ \ldots\ p_k\ e_k\texttt{)}$$

meaning that effect e_i occurs with probability p_i. We require that the constraints $p_i \geq 0$ and $\sum_{i=1}^{k} p_i = 1$ are fulfilled: a probabilistic effect declares an exhaustive set of probability-weighted outcomes. We do, however, allow a probability-effect pair to be left out if the effect is empty. In other words,

$$\texttt{(probabilistic } p_1\ e_1\ \ldots\ p_l\ e_l\texttt{)}$$

with $\sum_{i=1}^{l} p_i < 1$ is syntactic sugar for

$$\texttt{(probabilistic } p_1\ e_1\ \ldots\ p_l\ e_l\ q\ \texttt{(and))}$$

with $q = 1 - \sum_{i=1}^{l} p_i$ and `(and)` representing an empty effect (that is, no state changes). For example, the effect `(probabilistic 0.9 (clogged))` means that with probability 0.9 the state variable *clogged* becomes true in the next state, while with probability 0.1 the state remains unchanged.

Figure 1 shows an encoding in PPDDL of the “Bomb and Toilet” example described by Kushmerick, Hanks, and Weld (1995). The requirements flag `:probabilistic-effects` signals that probabilistic effects are used in the domain definition. In this problem, there are two packages, one of which contains a bomb. The bomb can be defused by dunking the package containing the bomb in the toilet. There is a 0.05 probability of the toilet becoming clogged when a package is placed in it, thus rendering the goal state unreachable.

The problem definition in Figure 1 also shows that initial conditions in PPDDL can be probabilistic. In this particular example, we define two possible initial states with equal probability (0.5) of being the true initial state for any given execution. Table 1 lists the state variables for the “Bomb and Toilet” problem and their values in the two possible initial states. Intuitively, we can think of the initial conditions of a PPDDL planning problem as being the effects of an action forced to be scheduled right before time 0. Also, note that the goal of the problem involves negation, which is why the problem definition declares the `:negative-preconditions` requirements flag.

```
(define (domain bomb-and-toilet)
   (:requirements :conditional-effects :probabilistic-effects)
   (:predicates (bomb-in-package ?pkg) (toilet-clogged)
                (bomb-defused))
   (:action dunk-package
            :parameters (?pkg)
            :effect (and (when (bomb-in-package ?pkg)
                               (bomb-defused))
                         (probabilistic 0.05 (toilet-clogged)))))

(define (problem bomb-and-toilet)
   (:domain bomb-and-toilet)
   (:requirements :negative-preconditions)
   (:objects package1 package2)
   (:init (probabilistic 0.5 (bomb-in-package package1)
                         0.5 (bomb-in-package package2)))
   (:goal (and (bomb-defused) (not (toilet-clogged)))))
```

Figure 1: PPDDL encoding of "Bomb and Toilet" example.

PPDDL allows arbitrary nesting of conditional and probabilistic effects (see example in Figure 2). This feature is in contrast to popular encodings, such as probabilistic STRIPS operators (PSOs; Kushmerick et al., 1995) and factored PSOs (Dearden & Boutilier, 1997), which do not allow conditional effects nested inside probabilistic effects. While arbitrary nesting does not add to the expressiveness of the language, it can allow for exponentially more compact representations of certain effects given the same set of state variables and actions (Rintanen, 2003). Any PPDDL action can, however, be translated into a *set* of PSOs with at most a polynomial increase in the size of the representation. Consequently, it follows from the results of Littman (1997) that PPDDL, *after grounding* (that is, full instantiation of action schemata), is representationally equivalent to dynamic Bayesian networks (Dean & Kanazawa, 1989), which is another popular representation for MDP planning problems.

Still, it is worth noting that a single PPDDL action *schema* can represent a large number of actions and a single predicate can represent a large number of state variables, meaning that PPDDL often can represent planning problems more succinctly than other representations. For example, the number of actions that can be represented using m objects and n action schemata with arity c is $m \cdot n^c$, which is not bounded by any polynomial in the size of the original representation $(m+n)$. Grounding is by no means a prerequisite for PPDDL planning, so planners could conceivably take advantage of the more compact representation by working directly with action schemata.

2.2 Rewards

Markovian rewards, associated with state transitions, can be encoded using *fluents* (numeric state variables). PPDDL reserves the fluent *reward*, accessed as `(reward)` or `reward`, to represent the total accumulated reward since the start of execution. Rewards are associated

```
(define (domain coffee-delivery)
   (:requirements :negative-preconditions
                  :disjunctive-preconditions
                  :conditional-effects :mdp)
   (:predicates (in-office) (raining) (has-umbrella) (is-wet)
                (has-coffee) (user-has-coffee))
   (:action deliver-coffee
            :effect (and (when (and (in-office) (has-coffee))
                              (probabilistic
                                 0.8 (and (user-has-coffee)
                                          (not (has-coffee))
                                          (increase (reward) 0.8))
                                 0.2 (and (probabilistic 0.5 (not (has-coffee)))
                                          (when (user-has-coffee)
                                             (increase (reward) 0.8)))))
                         (when (and (not (in-office)) (has-coffee))
                            (and (probabilistic 0.8 (not (has-coffee)))
                                 (when (user-has-coffee)
                                    (increase (reward) 0.8))))
                         (when (and (not (has-coffee)) (user-has-coffee))
                            (increase (reward) 0.8))
                         (when (not (is-wet))
                            (increase (reward) 0.2))))
    ... )
```

Figure 2: Part of PPDDL encoding of "Coffee Delivery" domain.

with state transitions through update rules in action effects. The use of the *reward* fluent is restricted to action effects of the form `(`$\langle$*additive-op*$\rangle$ $\langle$*reward fluent*$\rangle$ $\langle$*f-exp*$\rangle$`)`, where $\langle$*additive-op*$\rangle$ is either `increase` or `decrease`, and $\langle$*f-exp*$\rangle$ is a numeric expression not involving *reward*. Action preconditions and effect conditions are not allowed to refer to the *reward* fluent, which means that the accumulated reward does not have to be considered part of the state space. The initial value of *reward* is zero. These restrictions on the use of the *reward* fluent allow a planner to handle domains with rewards without having to implement full support for fluents.

A new requirements flag, `:rewards`, is introduced to signal that support for rewards is required. Domains that require both probabilistic effects and rewards can declare the `:mdp` requirements flag, which implies `:probabilistic-effects` and `:rewards`.

Figure 2 shows part of the PPDDL encoding of a coffee delivery domain described by Dearden and Boutilier (1997). A reward of 0.8 is awarded if the user has coffee after the "deliver-coffee" action has been executed, and a reward of 0.2 is awarded if *is-wet* is false after execution of "deliver-coffee". Note that a total reward of 1.0 can be awarded as a result of executing the "deliver-coffee" action if execution of the action leads to a state where both *user-has-coffee* and $\neg$*is-wet* hold.

2.3 Plan Objectives

Regular PDDL goals are used to express goal-type performance objectives. A goal statement `(:goal` ϕ`)` for a probabilistic planning problem encodes the objective that the probability of achieving ϕ should be maximized, unless an explicit optimization metric is specified for the planning problem. For planning problems instantiated from a domain declaring the `:rewards` requirement, the default plan objective is to maximize the expected reward. A goal statement in the specification of a reward oriented planning problem identifies a set of absorbing states. In addition to transition rewards specified in action effects, it is possible to associate a one-time reward for entering a goal state. This is done using the `(:goal-reward` f`)` construct, where f is a numeric expression.

In general, a statement `(:metric maximize` f`)` in a problem definition means that the expected value of f should be maximized. Reward-oriented problems, for example a problem instance of the coffee-delivery domain in Figure 2, would declare `(:metric maximize (reward))` as the optimization criterion (this declaration is the default if the `:rewards` requirement has been specified). PPDDL defines `goal-achieved` as a special optimization metric, which can be used to explicitly specify that the plan objective is to maximize (or minimize) the probability of goal achievement. The value of the `goal-achieved` fluent is 1 if execution ends in a goal state. The expected value of `goal-achieved` is therefore equal to the probability of goal achievement. A declaration `(:metric maximize (goal-achieved))` takes precedence over any reward specifications in a domain or problem definition, and it is the default if the `:rewards` requirement has not been specified (for example, the "Bomb and Toilet" problem in Figure 1).

2.4 PPDDL Semantics

For completeness, we present a formal semantics for PPDDL planning problems in terms of a mapping to a probabilistic transition system with rewards. A planning problem defines a set of state variables V, possibly containing both Boolean and numeric state variables, although we only consider planning problems without any numeric state variables in this section. An assignment of values to state variables defines a state, and the state space S of the planning problem is the set of states representing all possible assignments of values to variables. In addition to V, a planning problem defines an initial-state distribution $p_0 : S \rightarrow [0, 1]$ with $\sum_{s \in S} p_0(s) = 1$ (that is, p_0 is a probability distribution over states), a formula ϕ_G over V characterizing a set of goal states $G = \{s \mid s \models \phi_G\}$, a one-time reward r_G associated with entering a goal state, and a set of actions A instantiated from PPDDL action schemata. For goal-directed planning problems, without explicit rewards, we use $r_G = 1$.

2.4.1 Probability and Reward Structure

An action $a \in A$ consists of a precondition ϕ_a and an effect e_a. Action a is applicable in a state s if and only if $s \models \neg\phi_G \wedge \phi_a$. It is an error to apply a to a state such that $s \not\models \neg\phi_G \wedge \phi_a$. Goal states are absorbing, so no action may be applied to a state satisfying ϕ_G. The requirement that ϕ_a must hold in order for a to be applicable is consistent with the semantics of PDDL2.1 (Fox & Long, 2003) and permits the modeling of forced chains of actions. Effects are recursively defined as follows (see also, Rintanen, 2003):

1. $\top$ is the null-effect, represented in PPDDL by `(and)`.
2. b and $\neg b$ are effects if $b \in V$ is a Boolean state variable.
3. $r \uparrow v$, for $v \in \mathbb{R}$, is an effect.
4. $c \rhd e$ is an effect if c is a formula over V and e is an effect.
5. $e_1 \wedge \cdots \wedge e_n$ is an effect if $e_1, \ldots, e_n$ are effects.
6. $p_1 e_1 | \ldots | p_n e_n$ is an effect if $e_1, \ldots, e_n$ are effects, $p_i \geq 0$ for all $i \in \{1, \ldots, n\}$, and $\sum_{i=1}^{n} p_i = 1$.

The effect b sets the Boolean state variable b to true in the next state, while $\neg b$ sets b to false in the next state. Effects of the form $r \uparrow v$ are used to associate rewards with transitions as described below.

An action $a = \langle \phi_a, e_a \rangle$ defines a transition probability matrix P_a and a state reward vector R_a, with $P_a(i,j)$ being the probability of transitioning to state j when applying a in state i, and $R_a(i)$ being the *expected* reward for executing action a in state i. We can readily compute the entries of the reward vector from the action effect formula e_a. Let χ_c be the characteristic function for the Boolean formula c, that is, $\chi_c(s)$ is 1 if $s \models c$ and 0 otherwise. The expected reward for an effect e applied to a state s, denoted $R(e;s)$, can be computed using the following inductive definition:

$$
\begin{aligned}
R(\top; s) &\doteq 0 \\
R(b; s) &\doteq 0 \\
R(\neg b; s) &\doteq 0 \\
R(r \uparrow v; s) &\doteq v \\
R(c \rhd e; s) &\doteq \chi_c(s) \cdot R(e; s) \\
R(e_1 \wedge \cdots \wedge e_n; s) &\doteq \sum_{i=1}^{n} R(e_i; s) \\
R(p_1 e_1 | \ldots | p_n e_n; s) &\doteq \sum_{i=1}^{n} p_i \cdot R(e_i; s).
\end{aligned}
$$

A factored representation of the probability matrix P_a can be obtained by generating a dynamic Bayesian network (DBN) representation of the action effect formula e_a. We can use Bayesian inference on the DBN to obtain a monolithic representation of P_a, but the structure of the factored representation can be exploited by algorithms for decision theoretic planning (see, for example, work by Boutilier, Dearden, & Goldszmidt, 1995; Hoey, St-Aubin, Hu, & Boutilier, 1999; Boutilier, Dean, & Hanks, 1999; Guestrin, Koller, Parr, & Venkataraman, 2003).

A Bayesian network is a directed graph. Each node of the graph represents a state variable, and a directed edge from one node to another represents a causal dependence. With each node is associated a conditional probability table (CPT). The CPT for state variable X's node represents the probability distribution over possible values for X conditioned on the values of state variables whose nodes are parents of X's node. A Bayesian network is a factored representation of the joint probability distribution over the variables represented in the network.

A DBN is a Bayesian network with a specific structure aimed at capturing temporal dependence. For each state variable X, we create a duplicate state variable X', with X representing the situation at the present time and X' representing the situation one time step into the future. A directed edge from a present-time state variable X to a future-time state variable Y' encodes a temporal dependence. There are no edges between two present-time state variables, or from a future-time to a present-time state variable (the present does not depend on the future). We can, however, have an edge between two future-time state variables. Such edges, called *synchronic* edges, are used to represent correlated effects. A DBN is a factored representation of the joint probability distribution over present-time and future-time state variables, which is also the transition probability matrix for a discrete-time Markov process.

We now show how to generate a DBN representing the transition probability matrix for a PPDDL action. To avoid representational blowup, we introduce a multi-valued auxiliary variable for each probabilistic effect of an action effect. These auxiliary variables are introduced to indicate which of the possible outcomes of a probabilistic effect occurs, allowing the representation to correlate all the effects of a specific outcome. The auxiliary variable associated with a probabilistic effect with n outcomes can take on n different values. A PPDDL effect e of size $|e|$ can consist of at most $O(|e|)$ distinct probabilistic effects. Hence, the number of auxiliary variables required to encode the transition probability matrix for an action with effect e will be at most $O(|e|)$. Only future-time versions of the auxiliary variables are necessary. For a PPDDL problem with m Boolean state variables, we need on the order of $2m + \max_{a \in A} |e_a|$ nodes in the DBNs representing transition probability matrices for actions.

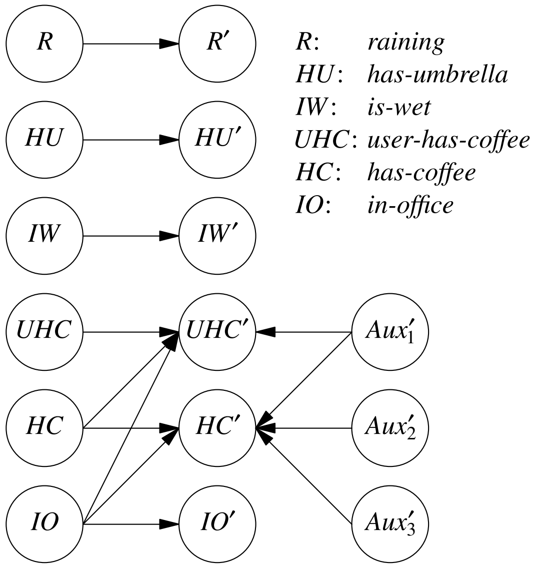
We provide a compositional approach for generating a DBN that represents the transition probability matrix for a PPDDL action with precondition ϕ_a and effect e_a. We assume that the effect is consistent, that is, that b and $\neg b$ do not occur in the same outcome with overlapping conditions. The DBN for an empty effect $\top$ simply consists of $2m$ nodes, with each present-time node X connected to its future-time counterpart X'. The CPT for X' has the non-zero entries $\Pr[X' = \top \mid X = \top] = 1$ and $\Pr[X' = \bot \mid X = \bot] = 1$. The same holds for a reward effect $r \uparrow v$, which does not change the value of state variables.

Next, consider the simple effects b and $\neg b$. Let X_b be the state variable associated with the PPDDL atom b. For these effects, we eliminate the edge from X_b to X'_b. The CPT for X'_b has the entry $\Pr[X'_b = \top] = 1$ for effect b and $\Pr[X'_b = \bot] = 1$ for effect $\neg b$.

For conditional effects, $c \rhd e$, we take the DBN for e and add edges between the present-time state variables mentioned in the formula c and the future-time state variables in the DBN for e.[1] Entries in the CPT for a state variable X' that correspond to settings of the present-time state variables that satisfy c remain unchanged. The other entries are set to 1 if X is true and 0 otherwise (the value of X does not change if the effect condition is not satisfied).

The DBN for an effect conjunction $e_1 \wedge \cdots \wedge e_n$ is constructed from the DBNs for the n effect conjuncts. The value for $\Pr[X' = \top \mid \mathbf{X}]$ in the DBN for the conjunction is set to

1. This transformation can increase the size of the DBNs exponentially unless context-specific DBNs are used (Boutilier, Friedman, Goldszmidt, & Koller, 1996).



				UHC'	
Aux'_1	IO	HC	UHC	$\top$	$\bot$
1	$\top$	$\top$	$\top$	1	0
1	$\top$	$\top$	$\bot$	1	0
1	$\top$	$\bot$	$\top$	1	0
1	$\top$	$\bot$	$\bot$	0	1
1	$\bot$	$\top$	$\top$	1	0
1	$\bot$	$\top$	$\bot$	0	1
1	$\bot$	$\bot$	$\top$	1	0
1	$\bot$	$\bot$	$\bot$	0	1
2	$\top$	$\top$	$\top$	1	0
2	$\top$	$\top$	$\bot$	0	1
2	$\top$	$\bot$	$\top$	1	0
2	$\top$	$\bot$	$\bot$	0	1
2	$\bot$	$\top$	$\top$	1	0
2	$\bot$	$\top$	$\bot$	0	1
2	$\bot$	$\bot$	$\top$	1	0
2	$\bot$	$\bot$	$\bot$	0	1

Figure 3: DBN structure (left) for the "deliver-coffee" action of the "Coffee Delivery" domain, with the CPT for UHC' (the future-time version of the state variable *user-has-coffee*) shown to the right.

the maximum of $\Pr[X' = \top \mid \mathbf{X}]$ over the DBNs for the conjuncts. The maximum is used because a state variable is set to true (false) by the conjunctive effect if it is set to true (false) by one of the effect conjuncts (effects are assumed to be consistent, so that the result of taking the maximum over the separate probability tables is still a probability table).

Finally, to construct a DBN for a probabilistic effect $p_1 e_1 | \ldots | p_n e_n$, we introduce an auxiliary variable Y' that is used to indicate which one of the n outcomes occurred. The node for Y' does not have any parents, and the entries of the CPT are $\Pr[Y' = i] = p_i$. Given a DBN for e_i, we add a synchronic edge from Y' to all state variables X. The value of $\Pr[X' = \top \mid \mathbf{X}, Y' = j]$ is set to $\Pr[X' = \top \mid \mathbf{X}]$ if $j = i$ and 0 otherwise. This transformation is repeated for all n outcomes, which results in n DBNs. These DBNs can trivially be combined into a single DBN for the probabilistic effect because they have mutually exclusive preconditions (the value of Y).

As an example, Figure 3 shows the DBN encoding of the transition probability matrix for the "deliver-coffee" action, whose PPDDL encoding was given in Figure 2. There are three auxiliary variables because the action effect contains three probabilistic effects. The node labeled UHC' (the future-time version of the state variable *user-has-coffee*) has four parents, including one auxiliary variable. Consequently, the CPT for this node will have $2^4 = 16$ rows (shown to the right in Figure 3).

2.4.2 Optimality Criteria

We have shown how to construct an MDP from the PPDDL encoding of a planning problem. The plan objective is to maximize the expected reward for the MDP. This objective can be interpreted in different ways, for example as expected *discounted* reward or expected *total* reward. The suitable interpretation depends on the problem. For process-oriented planning problems (for example, the "Coffee Delivery" problem), discounted reward is typically desirable, while total reward often is the interpretation chosen for goal-oriented problems (for example, the "Bomb and Toilet" problem). PPDDL does not include any facility for enforcing a given interpretation or specifying a discount factor.

For the competition, we used expected total reward as the optimality criterion. Without discounting, some care is required in the design of planning problems to ensure that the expected total reward is bounded for the optimal policy. The following restrictions were made for problems used in the planning competition:

1. Each problem had a goal statement, identifying a set of absorbing goal states.
2. A positive reward was associated with transitioning into a goal state.
3. A negative reward (cost) was associated with each action.
4. A "done" action was available in all states, which could be used to end further accumulation of reward.

These conditions ensure that an MDP model of a planning problem is a *positive bounded model* (Puterman, 1994). The only positive reward is for transitioning into a goal state. Since goal states are absorbing (that is, they have no outgoing transitions), the maximum value for any state is bounded by the goal reward. Furthermore, the "done" action ensures that there is an action available in each state that guarantees a non-negative future reward.

3. Evaluation Methodology

In classical planning, a plan is a series of operators. A successful plan is one that, when applied to the initial state, achieves the goal. In probabilistic planning, there are many proposals for plan representations (straight-line plans, plan trees, policy graphs, and triangle tables, for example), but none is considered a widely accepted standard. In addition, even simple plans are challenging to evaluate exactly in a non-deterministic environment, as all possible outcomes need to be checked and results combined (Littman, Goldsmith, & Mundhenk, 1998).

For these reasons, we chose to evaluate planners by simulation. That is, our plan validator was a server, and individual planning algorithms acted as clients. Planners connected to the validator, received an initial state, and returned an operator/action. This dialog continued until a terminating condition was reached at which point the validator evaluated the performance of the planner during that trajectory from initial state to terminating condition. This entire process was repeated several times and results averaged over the multiple runs.

Because this evaluation scheme blurs the distinction between a planner and an executor, it means that computation is no longer a one-time preprocessing cost, but something integrated with action selection itself. Planner quality, therefore, needs to be a combination of expected utility and running time. For simplicity, we set a time threshold and only allowed reward to be gathered until time ran out. This time threshold was known to competitors, whose planners could take it into consideration when deciding how to balance computation and action. Since we did not know whether participants would reuse results from one trajectory to speed planning in the next, we set an overall time limit that applied to the total of all repetitions of the evaluator for a given domain.

Concretely, in our evaluations, participants were presented with twenty previously unseen problems in PPDDL format. To evaluate each problem, participants connected to one of our evaluation servers (at CMU or Rutgers). The server provided the planner with an initial state and the planner selected and returned an action. This dialogue was iterated until a goal was reached, time ran out, or the planner sent a "done" action. The value obtained for the problem was the goal reward, if the goal was reached, minus the sum of the action costs (if any). For each problem, this procedure was repeated 30 times in a maximum of 15 minutes and the results averaged.

There were two types of problems in the evaluation set: reward problems and goal problems. For goal problems, the success percentage determined a participant's score for the problem (no action costs). In reward problems, every action had a fixed cost. The times to completion were recorded, but not explicitly used for ranking. Planners that completed less than 30 runs in 15 minutes were given a score of 0 for the unfinished runs.

In the design of the server, we believed that the time needed for computation in the planner would far outweigh any possible communication delay. However, in preliminary evaluations, participants—especially those halfway across the world—experienced disruptive levels of latency when evaluating their planners by connecting remotely to the server. Before the formal evaluation, we offered participants local accounts at CMU and nearly all availed themselves of this option.

3.1 Communication between Client and Server

The communication between a participant's client program and our server took place in XML. We made this decision for two reasons: The first is that parsing the messages into an easily managed format was trivial for all parties involved—many solid XML parsers exist in the public domain. The second is that bandwidth was not a great concern—as mentioned in the previous section, most participants ran their clients on the machine that hosted the server. While it is true that excessively large messages can take up valuable processing time, in our specific case those large messages corresponded to large state spaces, which took somewhat longer to process altogether, and the XML parsing was not the limiting factor.

When a client connected to a server, it would request a certain problem to run. The server would then lead the client through running that problem 30 times, sending the state of the problem, receiving the client's action, and then creating a new state from the old state and the action, and sending it back again. Figure 4 gives a schematic illustration of the conversation between the client and server. The specific format of each XML element is described in Appendix B.

Prior to the competition, an example client was written in C++ and distributed to the participants to minimize difficulties in dealing with the nuts and bolts of the protocol, allowing them to instead focus on the design of their algorithms.

client: ⟨*session-request*⟩
server: ⟨*session-init*⟩
–loop through 30 rounds
  client: ⟨*round-request*⟩
  server: ⟨*round-init*⟩
  –loop until termination conditions
    server: ⟨*state*⟩
    client: ⟨*act*⟩ | ⟨*noop*⟩ | ⟨*done*⟩
  server: ⟨*end-round*⟩
– **server**: ⟨*end-session*⟩

Figure 4: The interaction between client (planners) and server (environment) in our evaluation system.

3.2 Generator-Based Domains

Several example domains were provided to participants in advance to serve as testbeds for parser and planner development. In addition, parameterized problem generators were provided for two domain classes—Blocksworld and Boxworld, described in more detail in Section 4. The availability of these domains served to allow participants to learn, either manually or automatically, about the domains and to create domain-specific solutions. These approaches were evaluated independently in a separate category.

4. Competition Domains

This section describes the domains used in the competition. Machine readable versions of the domains themselves can be found online at the competition Web site:

http://www.cs.rutgers.edu/~mlittman/topics/ipc04-pt/

4.1 Blocksworld (Traditional)

Our traditional Blocksworld domain does not stray far from the original Blocksworld domain. The domain consists of two types of objects, blocks and tables. The domain has exactly one table and each problem instance has some number of blocks (the number of blocks is problem specific). The actions of the domain are "pick-up-block-from" and "put-down-block-on". For each problem instance, an initial configuration and a goal configuration of the blocks is given. The goal of the problem is to move the blocks from the initial configuration into the goal configuration. This domain comes in two flavors: a goal version and a reward version. Within the reward version, there is a cost of one unit every time the action "pick-up-block-from" is executed, and the reward is 500 for reaching the goal configuration.

As with all of the other domains used in the competition, the Blocksworld domain incorporates probabilistic effects and does so by adding a "slip" probability. That is, each time a block is picked up or put down, the block will slip and fall onto the table with probability 0.25. (Of course, if the intended action is to put the block down onto the table, then this effect will always be achieved.) The Blocksworld domain is an extremely simple domain, yet it offers a lot of insight into the planning process. Two important features of the domain are:

1. A basic policy to solve the domain is:
   (a) From the initial configuration, place all of the blocks onto the table with no block on top of another block.
   (b) Starting from the bottom up, place each block into its place in the final configuration.

   Note that without noise, if there are n blocks, this policy takes $4n$ steps (2 steps for each block on Part 1a, and 2 steps for each block on Part 1b) and hence costs $2n$ units. So, there is a very basic, very inexpensive way to solve this domain.
2. The state space of this domain increases exponentially with the number of blocks.

Thus, this domain aims at testing if planners could find the easy (maybe slightly more expensive) policy when the state space was too large to find a good policy. As far as the complexity of this domain is concerned, it is one of the easier domains to plan for and our hope was that many planners would do quite well in this domain.

A generation program for random traditional Blocksworld domains was provided to participants and the competition problems were generated from this same program. The availability of the generator allowed participants to test their planners on as many problems as they liked in advance of the evaluation.

4.2 Blocksworld (Color)

The colored Blocksworld domain is a variant of the traditional Blocksworld presented above. As in the traditional Blocksworld, colored Blocksworld consists of two types of objects, tables and blocks. Again, the domain has exactly one table and each problem instance has some number of blocks. The actions of the domain are still "pick-up-block-from" and "put-down-block-on", and this domain also comes in two flavors: goal and reward. The major difference from the traditional Blocksworld domain is that each block in the colored Blocksworld domain is assigned a color, and the goal configuration is specified in terms of block colors rather than specific blocks. Thus, in general, there are many different valid goal configurations. Goal conditions are expressed with existential quantification. For example, the PPDDL fragment

```
(:goal (and (exists (?b1) (is-green ?b1))
            (exists (?b2) (and (is-blue ?b2) (on-top-of ?b1 ?b2)))))
```

states that the goal is to have any green block on top of any blue block.

The noise in the colored Blocksworld domain is the same as in the traditional Blocksworld domain. That is, the colored Blocksworld domain incorporates probabilistic effects by adding a "slip" probability. Each time a block is picked up or put down, the block will slip and fall onto the table with probability 0.25.

Notice that although the goal configuration is existentially quantified and hence not precisely specified, the same basic policy that can be used to solve the traditional Blocksworld can be used to solve the colored Blocksworld. To solve a colored Blocksworld problem, unstack all of the blocks and then, in a bottom up fashion, choose a block that satisfies a color constraint and place it in the appropriate position.

The colored Blocksworld domain aims to add complexity to the traditional Blocksworld domain by incorporating existential quantification into the goal configuration. The indeterminacy of the goal in the colored Blocksworld domain can make the planning problem considerably harder than its traditional counterpart. Thus, a colored Blocksworld problem may be impossible for a given planner to solve in a reasonable amount of time, whereas that same planner would have no problem on a traditional Blocksworld problem of the same size.[2]

A generation program for random colored Blocksworld domains was provided to participants and the competition problems were generated from this same program.

4.3 Boxworld

The Boxworld domain is modeled after the traditional logistics domain. The domain consists of four types of objects: cities, boxes, trucks and planes. For each problem, there is a graph superimposed on the cities with two different types of edges, one denoting the ability to drive from one city to another and the other denoting the ability to fly from one city to the other. The actions of the domain are "load-box-on-truck-in-city", "unload-box-from-truck-in-city", "load-box-on-plane-in-city", "unload-box-from-plane-in-city", "drive-truck" and "fly-plane". Both goal and reward versions of this domain were included in the evaluation. Within the reward version, there was a cost of 1 unit every time either "load-box-on-truck-in-city" or "load-box-on-plane-in-city" was executed, a cost of 5 units every time "drive-truck" was executed and a cost of 25 units every time "fly-plane" was executed. For each problem instance, the initial configuration determines the graph that is superimposed on the cities, identifies the locations of the boxes, trucks and planes and determines the final destination where each box should arrive. The goal configuration specifies a destination for every box. The goal of the problem is to move from the initial configuration to a state where each box is in its destined location.

Noise enters this domain in the action "drive-truck". When this action is executed, the desired effect is achieved with probability 0.8 (that is, with probability 0.8 the truck will end up in its expected destination). However, with probability 0.2, the truck will get lost and end up in the wrong destination. For each city, there are three cities that the truck may get lost to when trying to execute the "drive-truck" action. If the truck actually gets lost it will end up in each of these cities with equal probability (that is, with probability 1/3).

As with the Blocksworld domains, a generation program for random Boxworld domains was provided to participants and the competition problems were generated from this same program.

2. It is important to note that the existentially quantified goal formula for colored Blocksworld, when grounded, can be excessively long. This fact is a serious bottleneck for larger instances of this domain. Planners that avoid grounding should have a benefit here, but did not in the competition because our plan validator grounded the goal formula.

4.4 Exploding Blocksworld

The exploding Blocksworld domain is a “dead-end” version of the traditional Blocksworld domain described earlier. As in the traditional Blocksworld domain, there are two types of objects (tables and blocks) and two actions (“pick-up-block-from” and “put-down-block-on”). An initial configuration and a goal configuration of the blocks are given and the goal of the domain is to move the blocks from the initial configuration to the goal configuration.

The key difference between this domain and the traditional Blocksworld domain is that every block in the exploding Blocksworld domain is initially set to “detonate”. Every time a “put-down-block” action is executed, if the block that is being put down has not yet detonated it will detonate with probability 0.3; this is the only noise in the domain. If a block detonates when executing a “put-down-block” action, the object *beneath* the block (whether it is the table or another block) is destroyed and is no longer accessible within the domain. Once a block detonates, it is *safe* and can no longer detonate.

The exploding Blocksworld domain aims at testing a planner’s ability to “think ahead”. More formally, as actions are executed it is possible to reach a state from which the goal cannot be reached. Consider, for example, executing the standard Blocksworld approach in which all blocks are unstacked to the table before the goal configuration is constructed. After seven blocks have been unstacked, there is a 92% $(1-(1-0.3)^7)$ probability that the table is destroyed, rendering the problem unsolvable.

One strategy for solving an exploding Blocksworld problem is to never place an unsafe block on top of something valuable (the table or a block needed in the final stack). Instead, a block should first be “disarmed”, by placing it on top of some block that is not needed for the final configuration, if such a block exists.

We illustrate this strategy on the problem instance that was used in the planning competition, shown in Figure 5. Four blocks are not needed for the goal configuration: 4, 8, 9, and 10. We start by repeatedly picking up Block 0 and placing it on Block 9 until Block 0 detonates. Next, we detonate Block 1 in the same way using Block 10. With both Block 0 and Block 1 safe, we place Block 1 on the table and Block 0 on top of Block 1. This completes the left-most tower. At this stage, there are no safe moves because Blocks 4 and 8 are not clear. We pick up Block 6 and put it on Block 2. The last action leads to failure with probability 0.3. If successful, the right-most tower is completed. Block 8 is now clear and we use it to detonate Block 3. Block 3 is then safely placed on top of Block 5. Finally, the center tower is completed by placing Block 7 on top of Block 3, which can result in failure with probability 0.3. In total, the success probability of the given plan is $(1-0.3)^2 = 0.49$, which, in fact, is optimal for the given problem (there are no action costs).

Along with several other test domains, exploding Blocksworld was specifically designed so that a replanning strategy performs suboptimally (gets stuck with high probability). A replanning strategy would be to ignore the 0.3 probability of detonation and try to replan if something unexpected happens. However, there is a high probability that this approach will render the goal state unreachable.

4.5 Fileworld

Fileworld is a fairly basic domain. It consists of k files and n folders for the files to be filed into. The actions of the domain are “get-type” (reports which folder the given file

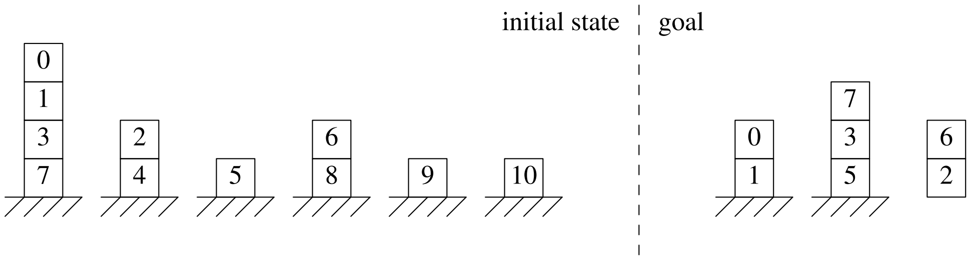


Figure 5: Exploding Blocksworld problem used in the planning competition. Note that the goal condition does not require Block 2 to be on the table.

belongs in), "get-folder-Fi" (one for each $i \in \{0, \ldots, n-1\}$, retrieves Folder i from the filing cabinet), "file-Fi" (one for each $i \in \{0, \ldots, n-1\}$, inserts the given file into Folder i) and "return-Fi" (one for each $i \in 0, \ldots, n-1\}$, returns Folder i to the filing cabinet). This domain comes only in a reward version. There is a cost of 100 for executing the action "get-folder-Fi" and a cost of 1 for executing the action "file-Fi". The other actions have no explicit costs since they must be used in conjunction with get-folder-Fi and file-Fi. The initial configuration of the problem specifies how many folders there are (the competition problem used 30 files and 5 folders) and the goal configuration specifies that all of the files must be filed. Note that the initial configuration does not specify which folder a file is to go into, but files cannot be filed into just any folder; this constraint is where the noise comes into the domain.

Before a file can be filed, its destination folder must be determined. The destination folder of a file is obtained by executing the action "get-type" with the file in question as a parameter. When this action is executed, the file passed as a parameter is assigned a folder, with each folder being the file's destination with equal probability (that is, probability $1/n$). Once a file has a destination folder, it can be filed into this (and only this) folder.

The Fileworld domain tests a planner's ability to consider all of its strategies and choose the one that minimizes the cost. In particular, a straightforward plan to achieve the goal is to carry out the following series of actions on each file in turn:

1. Get its type with "get-type"
2. Get its destination folder by executing "get-folder-Fi"
3. Place the file in the appropriate folder by executing the "file-Fi" action
4. Return the folder by executing the "return-Fi" action

Although this plan works, it is very costly. Its cost would be $101k$ where k is the number of files, because "get-folder-Fi" (expensive) and "file-Fi" (cheap) are executed for every file. A less costly (in fact, the optimal) plan can be described. It first executes "get-type" on every file. Then, for each folder $i \in \{0, \ldots, n-1\}$ that at least one file has as its destination, it runs "get-folder-Fi". Next, it files every file that belongs in folder i using "file-Fi". It then uses "return-Fi" in preparation for getting the next folder.

The expected reward for the optimal plan is $600 - (100n + k)$, where n is the number of folders and k is the number of files (this analysis gives 70 as the optimal expected reward for the competition problem). The domain is designed to reward planners that are able to reason about the initial destination uncertainty of the files and recognize that the second plan is much less costly and should be preferred to the straightforward brute-force plan.

4.6 Tireworld

Tireworld is another domain that tests the planners' ability to plan ahead under uncertainty. The domain consists of one type of object, namely locations. This domain comes in two flavors, a goal version and a reward version. The actions common to both versions are "move-car", "load-tire" and "change-tire". In the reward version, there is the additional action "call-AAA".

Within the reward version, there is a cost of 1 every time one of the actions "move-car", "load-tire" or "change-tire" is executed and a cost of 100 every time the action "call-AAA" is executed. The initial configuration of the problem defines a set of locations, a superimposed graph on these locations (roads), a subset of all locations representing the locations with spare tires, and the starting location on the graph. The goal configuration defines a destination location on the graph. The goal of the problem is to move from the starting location to the goal location.

The noise in Tireworld comes from the action "move-car". Each time this action is executed, the car drives from one city to another and will get a flat tire with probability 0.15. Once a car has a flat tire, it cannot execute the action "move-car" again until the tire is fixed. The car has the ability to store at most one spare tire, which it can pick up by executing the action "load-tire" when it is in a location with a spare tire. If the car is holding a spare tire, the "change-tire" action can be invoked to fix the flat. However, if the car does not currently have a spare then this action is disabled. In the goal version, a flat tire may result in a dead end if a car gets a flat and carries no spare tire. In the reward version, the planner has the choice of executing one of the actions "change-tire" (if the car has a spare) or "call-AAA" (at a high cost) to repair the flat. Thus, in the reward version, there are no dead ends and the goal is always reachable. Notice that since the cost of "call-AAA" is large compared to the costs of "change-tire" and "load-tire", fixing a flat is always less expensive if the car has a spare tire.

Figure 6 illustrates the Tireworld problem used in the competition. We next compare the probability of reaching a goal state for two different plans for this problem to illustrate what an ideal plan looks like in this domain.

An optimal plan would look ahead and attempt to keep spare tires as accessible as possible to avoid dead ends. From the start state, the car must make three steps without a flat tire to reach the first spare at `cc`, which will occur with probability $0.85^3 \approx 0.61$. Now, the car needs to go four steps without getting *two* flats to make it to the next spare at `d5`. It gets zero flats with probability $0.85^4 \approx 0.52$ and one flat with probability $4 \times 0.85^3 \times 0.15 \approx 0.37$, so a four-step segment can be traversed with probability $0.52 + 0.37 = 0.89$ with one spare tire. There are three four-step segments that must be traversed successfully to reach `ck`. Finally, with a spare, the last two steps can be traveled with certainty. Thus, the total success probability of this event sequence is $0.61 \times 0.89^3 \approx 0.43$. Note that this estimate is a

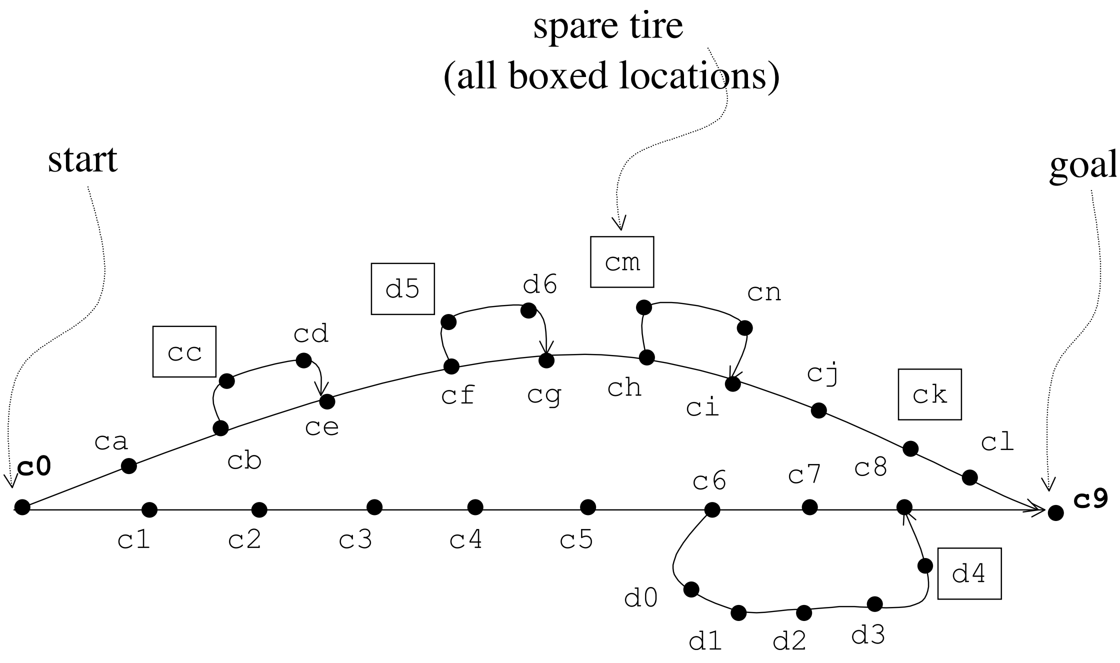


Figure 6: The Tireworld domain used in the competition.

lower bound on the success probability of the optimal strategy, because it does not factor in the probability of getting a flat tire upon arrival to a state with a spare tire. Furthermore, if the car is in location `cf` or `ch` with a spare and no flat, it is unnecessary to traverse the loop to pick up the spare tire in location `d5` or `cm`. By accounting for these factors we get a success probability of just over 0.57.

In contrast, a greedy replanning algorithm would not gather spares, since their utility comes from the realization that something might go wrong. For such a planner, the best plan is to go directly from `c0` to `c9` on the shortest (9-step) route. Its success probability is $0.85^9 \approx 0.23$, which is just 40 percent of the best success probability computed above.

In the reward version of the planning problem, the optimal success probability is one because the "call-AAA" action is always available. However, the cost of this action equals the reward for reaching the goal, so it is always better to end execution with the "done" action than to repair a flat tire with the "call-AAA" action. Hence, the best strategy for the goal version is optimal for the reward version as well and gives a reward of just under 45. The greedy strategy outlined above would result in an expected reward of just over 22. If the "call-AAA" action is used to fix flat tires, then the expected reward drops to -29.

4.7 Towers of Hanoise

As the name suggests, this domain is a noisy version of the famous Towers of Hanoi problem. The domain has two types of objects, disks and pegs. The problem that was used in the competition had five disks and three pegs. The actions of the domain are "single-move-big-not-moved", "single-move-big-moved", "double-move-big-not-moved" and "double-move-big-moved". As the action names suggest, one can move either one or two disks at a time (single-move/double-move) and the outcome of the move is dependent on whether or not the biggest disk has been moved yet (big-not-moved/big-moved). The objective of the domain is to maximize the probability of reaching a goal configuration (no rewards).

The initial configuration defines the starting positions of the disks (as in Towers of Hanoi, the five disks are stacked on the first peg in bottom to top, largest to smallest order). The goal configuration defines the destination positions of the disks (again, the destination positions are the same as that of Towers of Hanoi, namely all five disks are stacked in the same order as the initial configuration but on the last peg). The goal of the problem is to move the disks from the starting configuration to the goal configuration. All actions in Towers of Hanoise have noisy outcomes. In particular, when executing an action it is possible to drop a disk and have it be lost forever, thus bringing execution to a dead end. The success probabilities are:

Action	**Success Probability**
"single-move-big-not-moved"	0.99
"single-move-big-moved"	0.95
"double-move-big-not-moved"	0.80
"double-move-big-moved"	0.90

Notice that the probability of succeeding with a move is dependent on the number of disks moved and whether or not the big disk has been moved yet.

Every sequence of actions has some success probability less than one in this problem, so it is not possible to reach the goal with certainty. To maximize the probability of reaching the goal, a careful comparison must be made. To move the big disk from the first to last peg, it is necessary to move the four smaller disks to the middle peg. This subgoal can be achieved by executing "single-move-big-not-moved" fifteen times on the smaller disks, resulting in a success probability of $0.99^{15} \approx 0.86$. It can also be accomplished by moving the four smaller disks as two units of two using "double-move-big-not-moved" three times, resulting in a low success probability of approximately 0.51.

Next, the big disk can be moved from the first to last peg with a success probability of 0.99 ("single-move-big-not-moved"). Then, the four smaller disks again need to be moved, this time from the middle peg to the last peg. Now that the big disk has been moved, the success probabilities change and the two strategies yield success probabilities of about 0.46 for "single-move-big-moved" and 0.73 for "double-move-big-moved".

A planner that chooses optimally at each step would switch from single moves to double moves after the big disk is in place resulting in a total success probability of $0.99^{15} \times 0.99 \times 0.9^3 \approx 0.62$. One that ignores probabilities and always uses single moves has a lower success probability of $0.99^{15} \times 0.99 \times 0.95^{15} \approx 0.39$. A planner that ignores probabilities and minimizes the number of steps by always using double moves has a lower success probability still of $0.8^3 \times 0.99 \times 0.9^3 \approx 0.37$. Thus, for optimum performance, a planner must realize that its policy should consider the success probabilities of actions and how they are influenced by the status of the big disk.

4.8 Zeno Travel

Our last domain is Zeno Travel, based on a domain used in IPC-3. Problem instances of this domain involve using airplanes to move people between cities. An airplane requires fuel to fly. It can be flown at two different speeds—the higher speed requiring more fuel. Our problem instance used one aircraft, two people, three cities and seven fuel levels. The actions of the domain are "start-boarding", "complete-boarding", "start-debarking", "complete-debarking", "start-refueling", "complete-refueling", "start-flying", "complete-flying", "start-zooming", and "complete-zooming". The initial configuration specifies the location of the plane, the initial fuel level of the plane and the location of all people (as well as some initializations to allow for arithmetic type operations on the fuel-level objects). The goal configuration specifies a destination for the plane and destinations for all people. The noise in this domain comes from the family of "complete-X" actions. Each time a "complete-X" action is executed it will have the desired effect with probability $1/k$ for some positive integer k (note that k is a function of the action executed, specifically $k = 20$ for "complete-boarding" and $k = 30$ for "complete-debarking"). If the desired effect is not achieved then there is no effect, and this occurs with probability $1 - (1/k)$. This structure is meant to represent actions with random duration. Each "durative" action X is represented by two primitive actions "start-X" and "complete-X", giving X a duration that is geometrically distributed.

Ultimately, this problem presented no real challenge because we neglected to include action costs. Since actions have either standard desired effect or none at all, a planner can simple continue to execute an action until its effect is achieved, without incurring any cost.

5. Competition Results

Based on the initial announcement of the competition, we put together a mailing list of 87 researchers expressing interest. As development of PPDDL, the server, the evaluation criteria, and the practice domains progressed, we kept the community informed by releasing a series of FAQs (May 2003, FAQ 0.1; September 2003, FAQ 0.5; November 2003 FAQ 1.01).

By early 2004, a core group of participants became evident and the competition logistics were finalized. Leading up to June 2004, participants ran their planners on the previously unseen test problems. We tabulated the scores in each of a set of evaluation categories and presented them at ICAPS'04 in Vancouver, Canada.

The following subsections describe the competition's participants, evaluation tracks, and results.

5.1 Participants

Although twenty teams registered for the competition initially, seven teams from four continents ultimately competed. They produced ten different planners, which were evaluated on various subsets of the problem domains. The groups and their planners were:

- Group C. UMass

  Participants: Zhengzhu Feng (University of Massachusetts) and Eric Hansen (Mississippi State University).

Description: Symbolic heuristic search.

- Group E. Dresden ("FluCaP", formerly "FCPlanner")

  Participants: Eldar Karabaev and Olga Skvortsova (both of Dresden University of Technology).

  Description: First-order heuristic search.

- Group G. ANU ("NMRDPP")

  Participants: Charles Gretton, David Price and Sylvie Thiébaux (all of The Australian National University).

  Descriptions: **G1**: Planner primarily designed for domains with non-Markovian rewards, and **G2**: NMRDPP augmented with control knowledge.

- Group J. Purdue

  Participants: SungWook Yoon, Alan Fern and Robert Givan (all of Purdue University).

  Descriptions: **J1**: Human-written policy in Classy's policy language ("Purdue-Humans"), **J2**: Offline policy iteration by reduction to classification, automatically acquiring a domain-specific policy ("Classy"), and **J3**: Deterministic replanner using FF ("FF-rePlan").

- Group P. Simón Bolívar ("mGPT")

  Participants: Blai Bonet (Universidad Simón Bolívar) and Héctor Geffner (Universitat Pompeu Fabra).

  Description: Labeled RTDP with lower bounds extracted from the problem description.

- Group Q. Michigan Tech ("Probapop")

  Participants: Nilufer Onder, Garrett C. Whelan and Li Li (all of Michigan Technological University).

  Description: POP-style planner (no sensing).

- Group R. CERT

  Participants: Florent Teichteil-Königsbuch and Patrick Fabiani (both of CERT).

  Description: Probabilistic reachability heuristic and DBNs.

5.2 Evaluation Tracks

It was clear from the discussions leading up to the competition that different groups were prioritizing their efforts differently. We wanted to ensure that a diverse set of powerful approaches were recognized and decided to tabulate results in several different ways to acknowledge the value of these different approaches. The six tracks were:

- **Overall**. This track used a reward-based evaluation criterion for all domains (goal achievement counted as 500 for goal-based domains). Domains: Blocksworld (7 problems), Colored Blocksworld (2), Boxworld (5), Exploding Blocksworld (1), Fileworld (1), Tireworld (2), Towers of Hanoise (1), Zeno Travel (1).

- **Goal-based**. For this track, we ignored action costs and counted goal achievement as a unit reward (thus emphasizing approaches that maximized the probability of reaching a goal state). The domains and problems used were the same as in the Overall track: Blocksworld (7), Colored Blocksworld (2), Boxworld (5), Exploding Blocksworld (1), Fileworld (1), Tireworld (2), Towers of Hanoise (1), Zeno Travel (1).

- **Overall, Non-Blocks/Box**. Blocksworld and Boxworld dominated the full set and we wanted to see how subtler problems were handled. Domains: Exploding Blocksworld (1), Fileworld (1), Tireworld (2), Towers of Hanoise (1), Zeno Travel (1).

- **Domain-specific**. "Domain-specific" allowed human-tuned rules; "Domain-specific, No Tuning" did not (only automatically generated rules specific to the domain were allowed). They were evaluated using the generated domains: Blocksworld (8), Colored Blocksworld (6), Boxworld (5).

- **Conformant**. Planners in this category had to produce straight-line plans, "blind" to intermediate states encountered. We prepared "unobservable" versions of the domains to evaluate planners in this category. Domains: Blocksworld (7), Colored Blocksworld (2), Boxworld (5), Exploding Blocksworld (1), Fileworld (1), Tireworld (2), Towers of Hanoise (1), Zeno Travel (1).

5.3 Results

To display the results for each evaluation track, we plotted the cumulative reward achieved by each participant over the set of evaluation problems (reward is accumulated left to right). In the reward-based tracks, goal achievement was counted as 500 for problems without an explicitly specified goal reward. These plots highlight where one planner has an advantage over another (greater slope) as well as the total difference in score (height difference between the lines).

Figure 7 displays the results in the Overall category. Two planners, J3 and P, produced significantly more positive results than the others, with the replanning algorithm J3 clearly dominating the others. J3 was crowned Overall winner, with P as runner up. The figure also displays the results for the Conformant category, which consisted solely of Q, the uncontested winner of the category.

Similar results are visible in the Goal-based track, displayed in Figure 8, in which J3 again comes out ahead with P achieving runner-up status. Comparing Figures 7 and 8 reveals that the margin of victory between J3 and P, R and G1 is diminished in the Goal-based category. This suggests that J3 is more sensitive to the rewards themselves—choosing cheaper paths among the multiple paths available to the goal. In the set of problems used in the competition, this distinction was not very significant and the graphs look very similar. However, a different set of test problems might have revealed the fundamental tradeoff

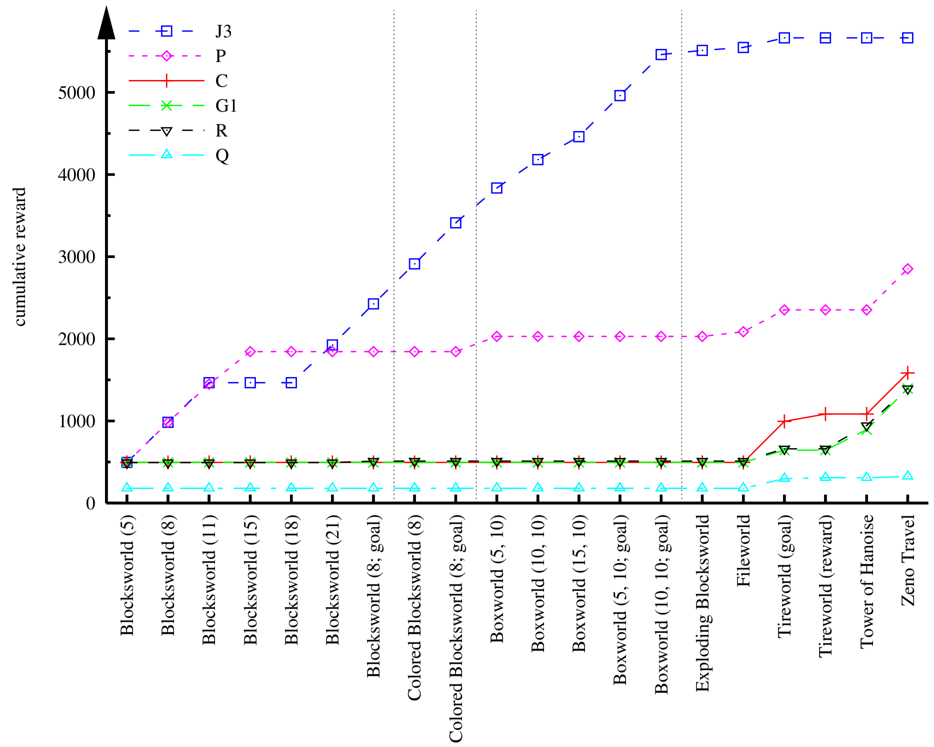


Figure 7: Competition results in the Overall category. The result for the Conformant category is the line marked "Q". The numbers in parentheses indicate problem size: number of blocks for Blocksworld domains; number of cities and boxes, respectively, for Boxworld domains.

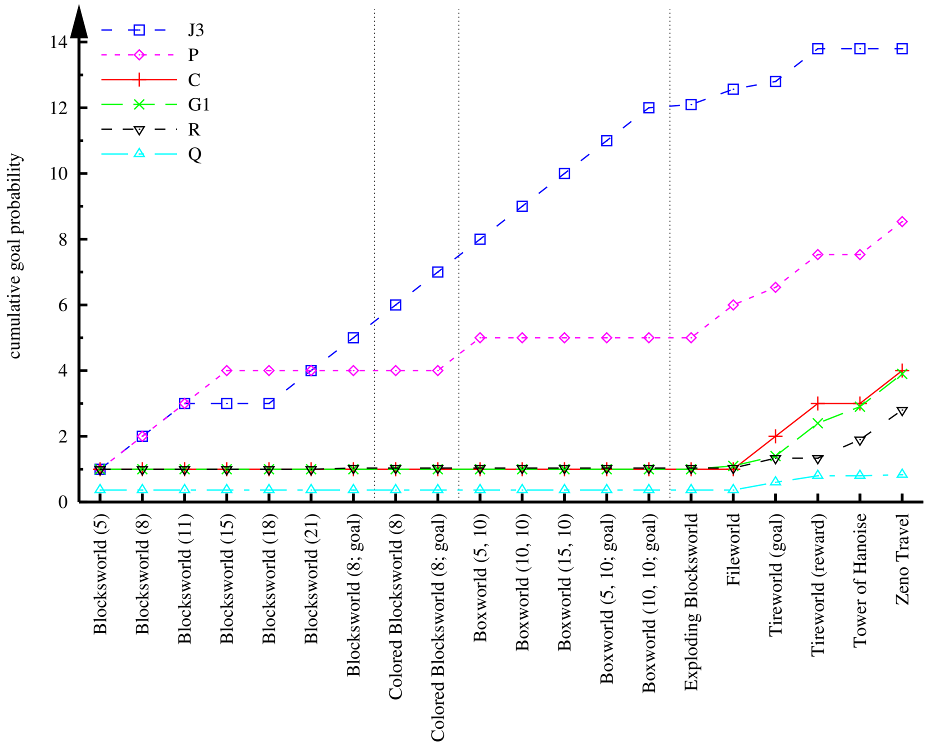


Figure 8: Competition results in the Goal-based category.

between seeking to maximize reward and seeking to reach the goal with high probability. Future competitions could attempt to highlight this important issue.

It is very interesting to note that J3's outstanding performance stems primarily from the early problems, which are the Blocksworld and Boxworld problems that are amenable to replanning. The later problems in the set were not handled as well by J3 as by most other planners.

Figure 9 displays the results for the Non-Block/Box category. Indeed, J3 performed much more poorly on the problems is this category, with Planner C taking the top spot. The runner-up spot was closely contested between planners R and G1, but G1 pulled ahead on the last problem to claim the honors. Planner P also performed nearly as well on this set.

Figure 10 gives a more detailed view of the results in the Non-Blocks/Box category. The optimal score for each problem is indicated in the graphs.[3] Note that Planner C's performance in the Tireworld domain is well above optimal, the result of a now-fixed bug in the competition server that allowed disabled actions to be executed. Planner P displayed outstanding performance on the Fileworld and goal-based Tireworld problems, but did not attempt to solve Tower of Hanoise and therefore fell behind G1 and R overall. Planner R used more time per round than Planner G1 in the Zeno Travel domain, which ultimately cost R the second place because it could only complete 27 of the 30 runs in this domain. Note that some planners received a negative score on the reward-oriented problems. We

3. The optimal scores do not necessarily apply to Planner Q, which is a conformant planner.

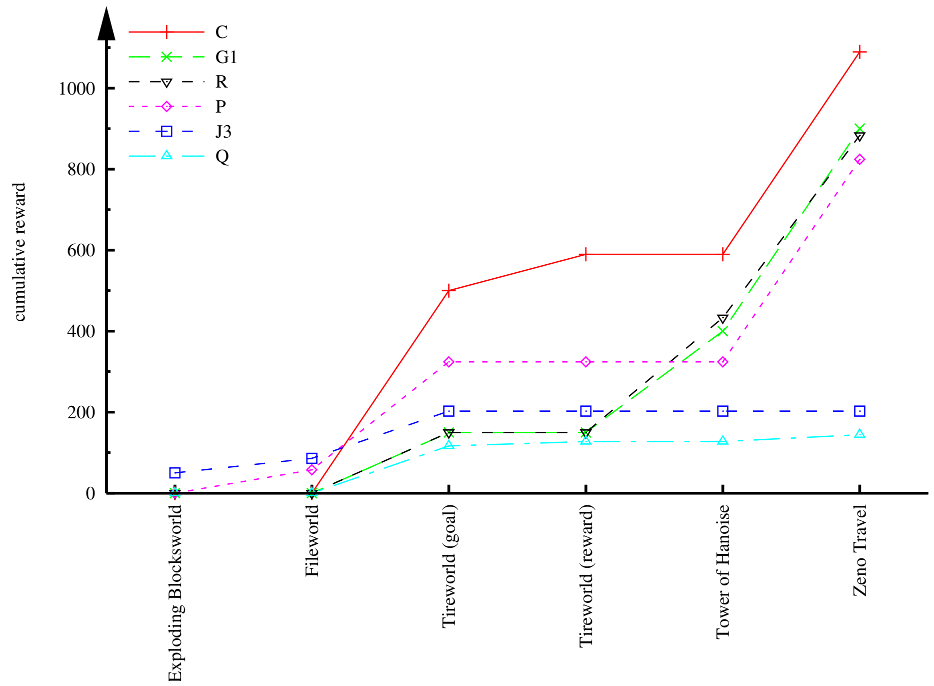


Figure 9: Summary of competition results in the Overall, Non-Blocks/Box category.

counted negative scores on individual problems as zero in the overall evaluation so as not to give an advantage to planners that did not even attempt to solve some problems. Planner Q was the only entrant (except, possibly, for C) to receive a positive score on the reward-based Tireworld problem. The planners with negative score for this problem used the expensive "call-AAA" action to ensure that the goal was always reached.

The results for the domain-specific planners are shown in Figure 11. The highest scoring planners were J1 and G2, with the difference between them primarily due to the two largest Blocksworld problems, which J1 solved more effectively than G2. The performance of the five domain-specific planners on the colored Blocksworld problems is virtually indistinguishable. As mentioned earlier, grounding of the goal condition in the validator prevented us from using larger problem instances, which might otherwise have separated the planners in this domain.

The two planners that won the "domain specific" category were ineligible for the "no tuning" subcategory because they were hand-tuned for these domains. Thus, J3 and J2 took the top spots in the subcategory. It is interesting to note that J3 won in spite of being a general-purpose planner—it was not, in fact, created to be domain specific. It overtook J2 due to two small Boxworld problems that J3 solved but J2 missed.

Figure 12 summarizes the competition results in the six evaluation categories.

6. Conclusion

We are happy with the outcomes of the first probabilistic track of the International Planning Competition. In addition to bringing attention to this important set of planning challenges,

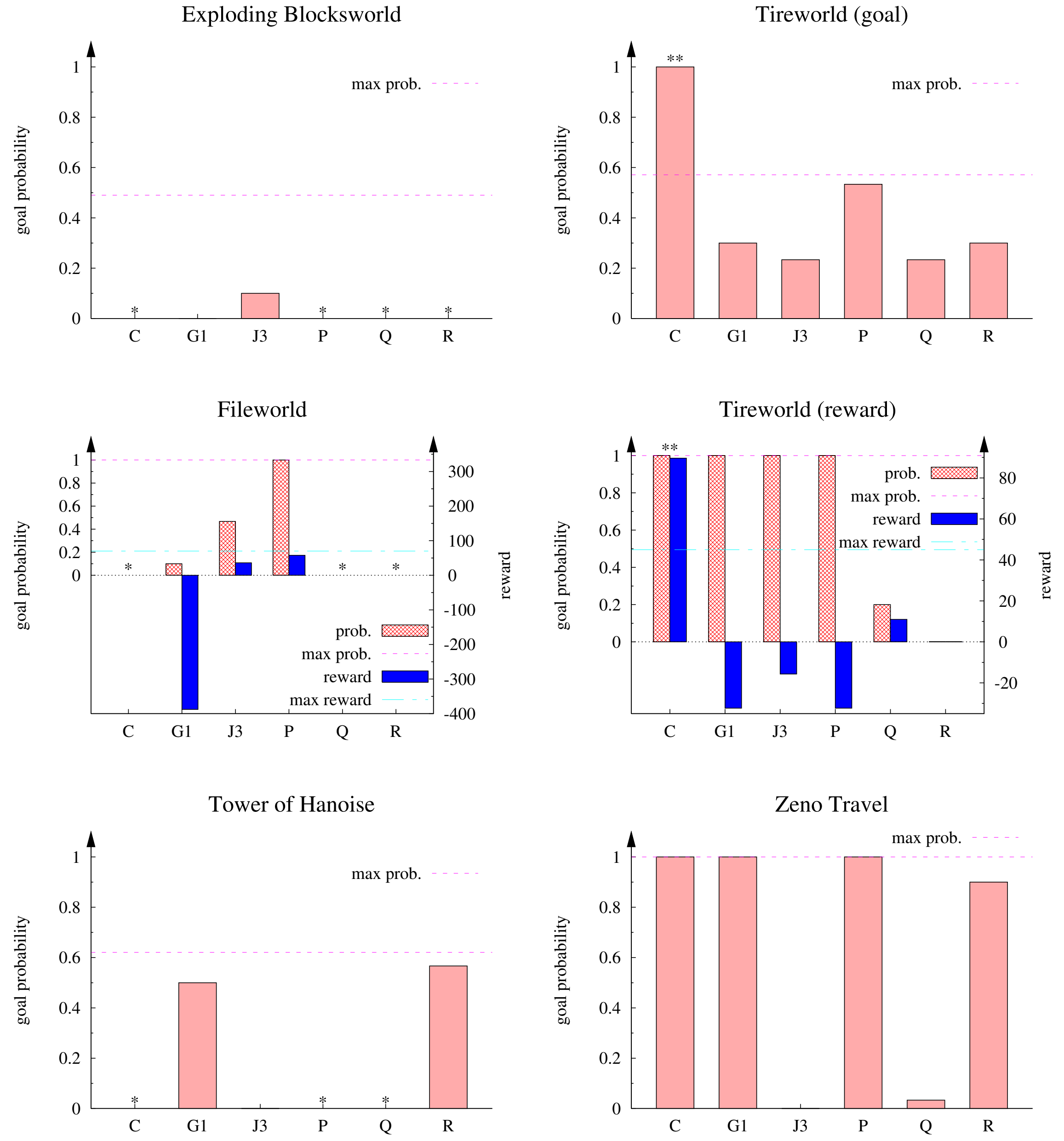


Figure 10: Competition results for Non-Blocks/Box problems (* indicates that a planner did not attempt to solve a problem; ** indicates anomalous results due to a bug in the server that allowed the execution of disabled actions). Note that the two graphs in the center have reward scales to the right.

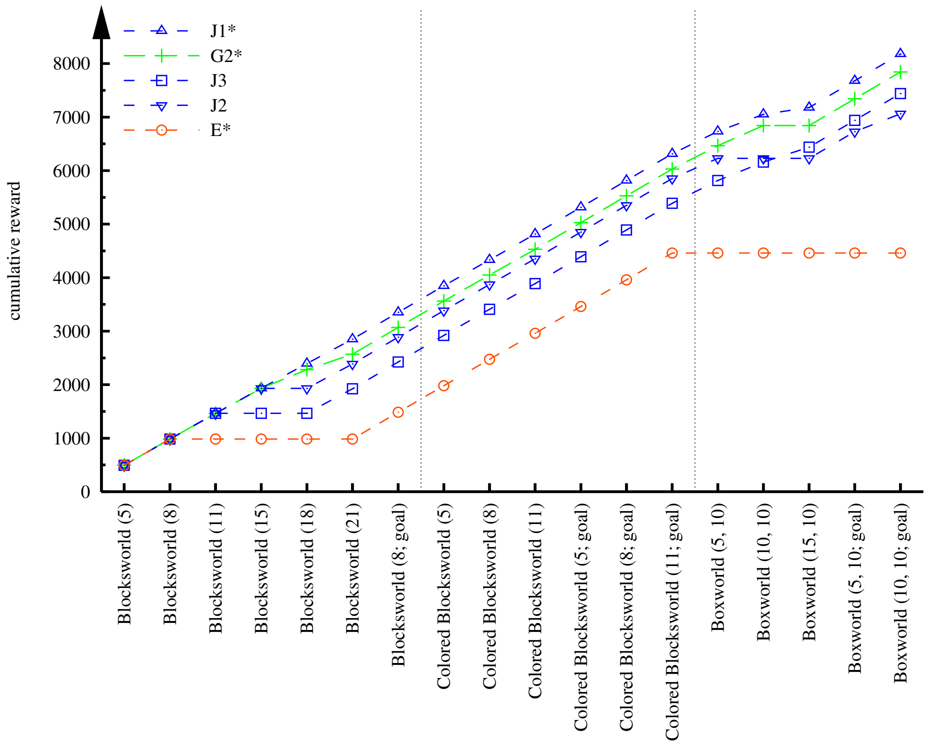


Figure 11: Competition results in the Domain-specific categories. For the "No Tuning" category results, ignore the J1, G2, and E lines on the graph (marked with asterisks).

Category	**1st**	**2nd**
Overall	J3	P
Goal-based Domains	J3	P
Overall, Non-Blocks/Box	C	G1
Domain-specific, No Tuning	J3	J2
Domain-specific	J1	G2
Conformant	Q	

Figure 12: Summary of competition results by category.

it appears to have helped spur the community to use uniform comparison problems by providing a domain language and a set of benchmarks (Yoon, Fern, & Givan, 2005).

In spite of the success, we feel there are changes that could be made in future competitions that would increase their value to the community. First, on the competition logistics side, our server logged outcomes of the interactions between planners and domains, but did not keep an exhaustive record of the actions taken and timing information. In retrospect, such information would have been helpful in identifying *how* the planners addressed the domains and whether they took suboptimal actions or just got unlucky. In addition, our server had no provisions for security. A simple password and/or reservation system would have helped the evaluations go much more smoothly as it would have prevented inadvertent access to the server by one group when another was assigned an evaluation slot.

On the domain side, we hope future competitions are able to focus on more interesting domains. We found that simply adding noisy action failures to a deterministic domain was not enough to produce interesting probabilistic problems—for such domains, straightforward replanning can be very effective. The non-Blocksworld domains we created were not mastered by any of the planners and we hope that they are retained in some form in future evaluations.

Like the progression of competitions in the classical track, we hope future competitions in the probabilistic track move toward domains grounded in real-life data and real-world problems including the handling of partially observability and time. A second competition is slated to be held in conjunction with IPC in 2006 and we urge interested members of the planning community to participate to help keep the competition moving in a productive direction for the benefit of the field.

Acknowledgments

We appreciate the support of the National Science Foundation and the Royal Swedish Academy of Engineering Sciences, as well as the feedback of Sven Koenig, Shlomo Zilberstein, Paul Batchis, Bob Givan, Hector Geffner and other participants who contributed to the design of the competition. JAIR editor David Smith and his anonymous reviewers provided invaluable insights on the document that we tried to reflect in this final manuscript.

This material is based upon work supported by the National Science Foundation under Grant No. 0315909 and the Royal Swedish Academy of Engineering Sciences (IVA) with grants from the Hans Werthén fund. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation or IVA.

Appendix A. BNF Grammar for PPDDL1.0

We provide the full syntax for PPDDL1.0 using an extended BNF notation with the following conventions:

- Each rule is of the form ⟨*non-terminal*⟩ ::= *expansion*.
- Alternative expansions are separated by a vertical bar ("|").
- A syntactic element surrounded by square brackets ("[" and "]") is optional.
- Expansions and optional syntactic elements with a superscripted requirements flag are only available if the requirements flag is specified for the domain or problem currently being defined. For example, [⟨*types-def*⟩]$^{\texttt{:typing}}$ in the syntax for domain definitions means that ⟨*types-def*⟩ may only occur in domain definitions that include the `:typing` flag in the requirements declaration.
- An asterisk ("*") following a syntactic element x means zero or more occurrences of x; a plus ("+") following x means at least one occurrence of x.
- Parameterized non-terminals, for example ⟨*typed list* (x)⟩, represent separate rules for each instantiation of the parameter.
- Terminals are written using `typewriter` font.
- The syntax is Lisp-like. In particular, case is not significant (for example, `?x` and `?X` are equivalent), parenthesis are an essential part of the syntax and have no semantic meaning in the extended BNF notation, and any number of whitespace characters (space, newline, tab, etc.) may occur between tokens.

A.1 Domains

The syntax for domain definitions is the same as for PDDL2.1, except that durative actions are not allowed. Declarations of constants, predicates, and functions are allowed in any order with respect to one another, but they must all come after any type declarations and precede any action declarations.

⟨*domain*⟩	::=	`( define ( domain` ⟨*name*⟩ `)`
		[⟨*require-def*⟩]
		[⟨*types-def*⟩]$^{\texttt{:typing}}$
		[⟨*constants-def*⟩]
		[⟨*predicates-def*⟩]
		[⟨*functions-def*⟩]$^{\texttt{:fluents}}$
		⟨*structure-def*⟩* `)`
⟨*require-def*⟩	::=	`( :requirements` ⟨*require-key*⟩* `)`
⟨*require-key*⟩	::=	*See Section A.4*
⟨*types-def*⟩	::=	`( :types` ⟨*typed list* (*name*)⟩ `)`
⟨*constants-def*⟩	::=	`( :constants` ⟨*typed list* (*name*)⟩ `)`
⟨*predicates-def*⟩	::=	`( :predicates` ⟨*atomic formula skeleton*⟩* `)`

```
⟨atomic formula skeleton⟩ ::= ( ⟨predicate⟩ ⟨typed list (variable)⟩ )
⟨predicate⟩               ::= ⟨name⟩
⟨functions-def⟩           ::= ( :functions ⟨function typed list (function skeleton)⟩ )
⟨function skeleton⟩       ::= ( ⟨function symbol⟩ ⟨typed list (variable)⟩ )
⟨function symbol⟩         ::= ⟨name⟩
⟨structure-def⟩           ::= ⟨action-def⟩
⟨action-def⟩              ::= See Section A.2
⟨typed list (x)⟩          ::= ⟨x⟩* |:typing ⟨x⟩+ - ⟨type⟩ ⟨typed list (x)⟩
⟨type⟩                    ::= ( either ⟨primitive type⟩+ ) | ⟨primitive type⟩
⟨primitive type⟩          ::= ⟨name⟩
⟨function typed list (x)⟩ ::= ⟨x⟩*
                              |:typing ⟨x⟩+ - ⟨function type⟩ ⟨function typed list (x)⟩
⟨function type⟩           ::= number
```

A ⟨*name*⟩ is a string of characters starting with an alphabetic character followed by a possibly empty sequence of alphanumeric characters, hyphens (“`-`”), and underscore characters (“`_`”). A ⟨*variable*⟩ is a ⟨*name*⟩ immediately preceded by a question mark (“`?`”). For example, `in-office` and `ball_2` are names, and `?gripper` is a variable.

A.2 Actions

Action definitions and goal descriptions have the same syntax as in PDDL2.1.

```
⟨action-def⟩          ::= ( :action ⟨action symbol⟩
                                [:parameters ( ⟨typed list (variable)⟩ )]
                                ⟨action-def body⟩ )
⟨action symbol⟩       ::= ⟨name⟩
⟨action-def body⟩     ::= [:precondition ⟨GD⟩]
                          [:effect ⟨effect⟩]
⟨GD⟩                  ::= ⟨atomic formula (term)⟩ | ( and ⟨GD⟩* )
                          |:equality ( = ⟨term⟩ ⟨term⟩ )
                          |:equality ( not ( = ⟨term⟩ ⟨term⟩ ) )
                          |:negative-preconditions ( not ⟨atomic formula (term)⟩ )
                          |:disjunctive-preconditions ( not ⟨GD⟩ )
                          |:disjunctive-preconditions ( or ⟨GD⟩* )
                          |:disjunctive-preconditions ( imply ⟨GD⟩ ⟨GD⟩ )
                          |:existential-preconditions ( exists ( ⟨typed list (variable)⟩ )
                                                            ⟨GD⟩ )
                          |:universal-preconditions ( forall ( ⟨typed list (variable)⟩ )
                                                          ⟨GD⟩ )
                          |:fluents ⟨f-comp⟩
⟨atomic formula (x)⟩  ::= ( ⟨predicate⟩ ⟨x⟩* ) | ⟨predicate⟩
⟨term⟩                ::= ⟨name⟩ | ⟨variable⟩
⟨f-comp⟩              ::= ( ⟨binary-comp⟩ ⟨f-exp⟩ ⟨f-exp⟩ )
⟨binary-comp⟩         ::= < | <= | = | >= | >
⟨f-exp⟩               ::= ⟨number⟩ | ⟨f-head (term)⟩
```

$\mid$ `(` $\langle \textit{binary-op} \rangle$ $\langle \textit{f-exp} \rangle$ $\langle \textit{f-exp} \rangle$ `)` $\mid$ `( -` $\langle \textit{f-exp} \rangle$ `)`

$\langle \textit{f-head (x)} \rangle$ ::= `(` $\langle \textit{function symbol} \rangle$ $\langle x \rangle^*$ `)` $\mid$ $\langle \textit{function symbol} \rangle$

$\langle \textit{binary-op} \rangle$ ::= `+` $\mid$ `-` $\mid$ `*` $\mid$ `/`

A $\langle \textit{number} \rangle$ is a sequence of numeric characters, possibly with a single decimal point ("`.`") at any position in the sequence. Negative numbers are written as `(-` $\langle \textit{number} \rangle$`)`.

The syntax for effects has been extended to allow for probabilistic effects, which can be arbitrarily interleaved with conditional effects and universal quantification.

$\langle \textit{effect} \rangle$ ::= $\langle \textit{p-effect} \rangle$ $\mid$ `( and` $\langle \textit{effect} \rangle^*$ `)`
$\quad \mid^{\texttt{:conditional-effects}}$ `( forall (` $\langle \textit{typed list (variable)} \rangle$ `)` $\langle \textit{effect} \rangle$ `)`
$\quad \mid^{\texttt{:conditional-effects}}$ `( when` $\langle \textit{GD} \rangle$ $\langle \textit{effect} \rangle$ `)`
$\quad \mid^{\texttt{:probabilistic-effects}}$ `( probabilistic` $\langle \textit{prob-effect} \rangle^+$ `)`

$\langle \textit{p-effect} \rangle$ ::= $\langle \textit{atomic formula (term)} \rangle$ $\mid$ `( not` $\langle \textit{atomic formula (term)} \rangle$ `)`
$\quad \mid^{\texttt{:fluents}}$ `(` $\langle \textit{assign-op} \rangle$ $\langle \textit{f-head (term)} \rangle$ $\langle \textit{f-exp} \rangle$ `)`
$\quad \mid^{\texttt{:rewards}}$ `(` $\langle \textit{additive-op} \rangle$ $\langle \textit{reward fluent} \rangle$ $\langle \textit{f-exp} \rangle$ `)`

$\langle \textit{prob-effect} \rangle$ ::= $\langle \textit{probability} \rangle$ $\langle \textit{effect} \rangle$

$\langle \textit{assign-op} \rangle$ ::= `assign` $\mid$ `scale-up` $\mid$ `scale-down` $\mid$ $\langle \textit{additive-op} \rangle$

$\langle \textit{additive-op} \rangle$ ::= `increase` $\mid$ `decrease`

$\langle \textit{reward fluent} \rangle$ ::= `( reward )` $\mid$ `reward`

A $\langle \textit{probability} \rangle$ is a $\langle \textit{number} \rangle$ with a value in the interval $[0, 1]$.

A.3 Problems

The syntax for problem definitions has been extended to allow for the specification of a probability distribution over initial states, and also to permit the association of a one-time reward with entering a goal state. It is otherwise identical to the syntax for PDDL2.1 problem definitions.

$\langle \textit{problem} \rangle$ ::= `( define ( problem` $\langle \textit{name} \rangle$ `)`
$\quad$ `( :domain` $\langle \textit{name} \rangle$ `)`
$\quad$ $[\langle \textit{require-def} \rangle]$
$\quad$ $[\langle \textit{objects-def} \rangle]$
$\quad$ $[\langle \textit{init} \rangle]$
$\quad$ $\langle \textit{goal} \rangle$ `)`

$\langle \textit{objects-def} \rangle$ ::= `( :objects` $\langle \textit{typed list (name)} \rangle$ `)`

$\langle \textit{init} \rangle$ ::= `( :init` $\langle \textit{init-el} \rangle^*$ `)`

$\langle \textit{init-el} \rangle$ ::= $\langle \textit{p-init-el} \rangle$
$\quad \mid^{\texttt{:probabilistic-effects}}$ `( probabilistic` $\langle \textit{prob-init-el} \rangle^*$ `)`

$\langle \textit{p-init-el} \rangle$ ::= $\langle \textit{atomic formula (name)} \rangle$ $\mid^{\texttt{:fluents}}$ `( =` $\langle \textit{f-head (name)} \rangle$ $\langle \textit{number} \rangle$ `)`

$\langle \textit{prob-init-el} \rangle$ ::= $\langle \textit{probability} \rangle$ $\langle \textit{a-init-el} \rangle$

$\langle \textit{a-init-el} \rangle$ ::= $\langle \textit{p-init-el} \rangle$ $\mid$ `( and` $\langle \textit{p-init-el} \rangle^*$ `)`

$\langle \textit{goal} \rangle$ ::= $\langle \textit{goal-spec} \rangle$ $[\langle \textit{metric-spec} \rangle]$ $\mid$ $\langle \textit{metric-spec} \rangle$

$\langle \textit{goal-spec} \rangle$ ::= `( :goal` $\langle \textit{GD} \rangle$ `)` `[( :goal-reward` $\langle \textit{ground-f-exp} \rangle$ `)]`$^{\texttt{:rewards}}$

$\langle \textit{metric-spec} \rangle$ ::= `( :metric` $\langle \textit{optimization} \rangle$ $\langle \textit{ground-f-exp} \rangle$ `)`

$$
\begin{array}{lll}
\langle \textit{optimization} \rangle & ::= & \texttt{minimize} \mid \texttt{maximize} \\
\langle \textit{ground-f-exp} \rangle & ::= & \langle \textit{number} \rangle \mid \langle \textit{f-head (name)} \rangle \\
 & \mid & \texttt{(} \ \langle \textit{binary-op} \rangle \ \langle \textit{ground-f-exp} \rangle \ \langle \textit{ground-f-exp} \rangle \ \texttt{)} \\
 & \mid & \texttt{(} \ \texttt{-} \ \langle \textit{ground-f-exp} \rangle \ \texttt{)} \\
 & \mid & \texttt{(total-time)} \mid \texttt{total-time} \\
 & \mid & \texttt{(goal-achieved)} \mid \texttt{goal-achieved} \\
 & \mid^{\texttt{:rewards}} & \langle \textit{reward fluent} \rangle
\end{array}
$$

A.4 Requirements

Below is a table of all requirements in PPDDL1.0. Some requirements imply others; some are abbreviations for common sets of requirements. If a domain stipulates no requirements, it is assumed to declare a requirement for `:strips`.

Requirement	*Description*
`:strips`	Basic STRIPS-style adds and deletes
`:typing`	Allow type names in declarations of variables
`:equality`	Support `=` as built-in predicate
`:negative-preconditions`	Allow negated atoms in goal descriptions
`:disjunctive-preconditions`	Allow disjunctive goal descriptions
`:existential-preconditions`	Allow `exists` in goal descriptions
`:universal-preconditions`	Allow `forall` in goal descriptions
`:quantified-preconditions`	= `:existential-preconditions` + `:universal-preconditions`
`:conditional-effects`	Allow `when` and `forall` in action effects
`:probabilistic-effects`	Allow `probabilistic` in action effects
`:rewards`	Allow reward fluent in action effects and optimization metric
`:fluents`	Allow numeric state variables
`:adl`	= `:strips` + `:typing` + `:equality` + `:negative-preconditions` + `:disjunctive-preconditions` + `:quantified-preconditions` + `:conditional-effects`
`:mdp`	= `:probabilistic-effects` + `:rewards`

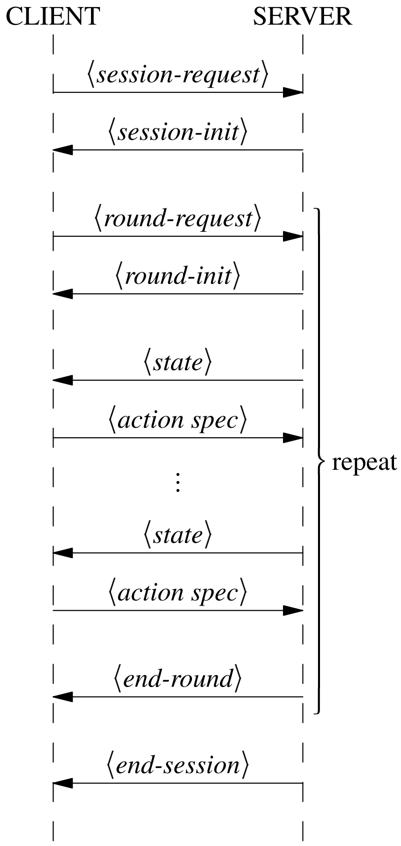


Figure 13: Successful communication session.

Appendix B. Communication Protocol

We adopt an XML-like syntax for the client/server communication protocol. We use the same extended BNF notation as in Appendix A to describe the syntax of protocol messages. The ⟨*name*⟩ and ⟨*number*⟩ terminals are defined in exactly the same way as for PPDDL. An ⟨*integer*⟩ is a nonempty string of numeric characters. A ⟨*message*⟩ is an arbitrary character string, possibly empty.

Figure 13 shows the expected sequence of messages. A session starts by the client sending a ⟨*session-request*⟩ message to the server. The server replies with a ⟨*session-init*⟩ message, which tells the client the number of evaluation rounds that will be run. To start an evaluation round, the client sends a ⟨*round-request*⟩ message, to which the server replies with a ⟨*round-init*⟩ message. At this point the evaluation round starts. The server sends a ⟨*turn-response*⟩ message to the client, which can be a ⟨*state*⟩ message or an ⟨*end-round*⟩ message. For every ⟨*state*⟩ message that the client receives, it sends an ⟨*action spec*⟩ message in return. Once the client receives an ⟨*end-round*⟩ message, it ends the current evaluation round. The client then starts a new evaluation round with a ⟨*round-request*⟩ message to the server, or waits for an ⟨*end-session*⟩ message from the server in case all rounds have already been run. The server sends an ⟨*error*⟩ message to the client if an error occurs, for example if the server receives an unexpected message from the client.

B.1 Client Messages

Client messages have the following form:

```
⟨session-request⟩ ::= <session-request>
                        <name> ⟨name⟩ </name>
                        <problem> ⟨name⟩ </problem>
                      </session-request>

⟨round-request⟩   ::= <round-request/>

⟨action spec⟩     ::= <act> ⟨action⟩ </act> | <done/>
⟨action⟩          ::= <action> <name> ⟨name⟩ </name> ⟨term⟩* </action>
⟨term⟩            ::= <term> ⟨name⟩ </term>
```

B.2 Server Messages

Server messages have the following form:

```
⟨session-init⟩    ::= <session-init>
                        <sessionID> ⟨integer⟩ </sessionID>
                        <setting>
                          <rounds> ⟨integer⟩ </rounds>
                          <allowed-time> ⟨integer⟩ </allowed-time>
                          <allowed-turns> ⟨integer⟩ </allowed-turns>
                        </setting>
                      </session-init>

⟨round-init⟩      ::= <round-init>
                        <round> ⟨integer⟩ </round>
                        <sessionID> ⟨integer⟩ </sessionID>
                        <time-left> ⟨integer⟩ </time-left>
                        <rounds-left> ⟨integer⟩ </rounds-left>
                      </round-init>

⟨turn-response⟩   ::= ⟨state⟩ | ⟨end-round⟩
⟨end-round⟩       ::= <end-round>
                        ⟨state⟩ [<goal-reached/>]
                        <time-spent> ⟨integer⟩ </time-spent>
                        <turns-used> ⟨integer⟩ </turns-used>
                      </end-round>
⟨state⟩           ::= <state> [<is-goal/>] ⟨atom⟩* ⟨fluent⟩* </state>
⟨atom⟩            ::= <atom> ⟨predicate⟩ ⟨term⟩* </atom>
⟨fluent⟩          ::= <fluent> ⟨function⟩ ⟨term⟩* ⟨value⟩ </fluent>
⟨predicate⟩       ::= <predicate> ⟨name⟩ </predicate>
⟨function⟩        ::= <function> ⟨name⟩ </function>
⟨term⟩            ::= <term> ⟨name⟩ </term>
```

```
⟨value⟩          ::= <value> ⟨number⟩ </value>

⟨end-session⟩    ::= <end-session>
                       <sessionID> ⟨integer⟩ </sessionID>
                       <problem> ⟨name⟩ </problem>
                       <rounds> ⟨integer⟩ </rounds>
                       <goals>
                         <failed> ⟨integer⟩ </failed>
                         <reached>
                           <successes> ⟨integer⟩ </successes>
                           [<time-average> ⟨number⟩ </time-average>]
                         </reached>
                       </goals>
                       [<metric-average> ⟨number⟩ </metric-average>]
                     </end-session>

⟨error⟩          ::= <error> ⟨message⟩ </error>
```

References

Boutilier, C., Dean, T., & Hanks, S. (1999). Decision-theoretic planning: Structural assumptions and computational leverage. *Journal of Artificial Intelligence Research*, *11*, 1–94.

Boutilier, C., Dearden, R., & Goldszmidt, M. (1995). Exploiting structure in policy construction. In Mellish, C. S. (Ed.), *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pp. 1104–1111, Montreal, Canada. Morgan Kaufmann Publishers.

Boutilier, C., Friedman, N., Goldszmidt, M., & Koller, D. (1996). Context-specific independence in Bayesian networks. In *Proceedings of the Twelfth Annual Conference on Uncertainty in Artificial Intelligence (UAI 96)*, pp. 115–123, Portland, OR.

Dean, T., & Kanazawa, K. (1989). A model for reasoning about persistence and causation. *Computational Intelligence*, *5*(3), 142–150.

Dearden, R., & Boutilier, C. (1997). Abstraction and approximate decision-theoretic planning. *Artificial Intelligence*, *89*(1–2), 219–283.

Fox, M., & Long, D. (2003). PDDL2.1: An extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research*, *20*, 61–124.

Guestrin, C., Koller, D., Parr, R., & Venkataraman, S. (2003). Efficient solution algorithms for factored MDPs. *Journal of Artificial Intelligence Research*, *19*, 399–468.

Hoey, J., St-Aubin, R., Hu, A., & Boutilier, C. (1999). SPUDD: Stochastic planning using decision diagrams. In Laskey, K. B., & Prade, H. (Eds.), *Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence*, pp. 279–288, Stockholm, Sweden. Morgan Kaufmann Publishers.

Howard, R. A. (1960). *Dynamic Programming and Markov Processes*. John Wiley & Sons, New York, NY.

Howard, R. A. (1971). *Dynamic Probabilistic Systems*, Vol. I: Markov Models. John Wiley & Sons, New York, NY.

Kushmerick, N., Hanks, S., & Weld, D. S. (1995). An algorithm for probabilistic planning. *Artificial Intelligence*, *76*(1–2), 239–286.

Littman, M. L. (1997). Probabilistic propositional planning: Representations and complexity. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence*, pp. 748–754, Providence, RI. American Association for Artificial Intelligence, AAAI Press.

Littman, M. L., Goldsmith, J., & Mundhenk, M. (1998). The computational complexity of probabilistic planning. *Journal of Artificial Intelligence Research*, *9*, 1–36.

McDermott, D. (2000). The 1998 AI planning systems competition. *AI Magazine*, *21*(2), 35–55.

Puterman, M. L. (1994). *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons, New York, NY.

Rintanen, J. (2003). Expressive equivalence of formalisms for planning with sensing. In Giunchiglia, E., Muscettola, N., & Nau, D. S. (Eds.), *Proceedings of the Thirteenth International Conference on Automated Planning and Scheduling*, pp. 185–194, Trento, Italy. AAAI Press.

Yoon, S., Fern, A., & Givan, R. (2005). Learning measures of progress for planning domains. In *Proceedings of the Twentieth National Conference on Artificial Intelligence*, pp. 1217–1222.

Younes, H. L. S., & Littman, M. L. (2004). PPDDL1.0: An extension to PDDL for expressing planning domains with probabilistic effects. Tech. rep. CMU-CS-04-167, Carnegie Mellon University, Pittsburgh, PA.