Realtime Concurrent Planning and Plan Execution in Stochastic Domains

Luis Pineda and **Shlomo Zilberstein**
Technical Report UM-CS-2014-21
School of Computer Science, University of Massachusetts
{lpineda, shlomo}@cs.umass.edu

Abstract

In realtime planning domains, such as service robot control, an agent receives a task and must minimize the combined cost of planning and plan execution necessary to complete the task. To reduce the total cost, we examine the feasibility of performing planning continuously, while parts of the intermediate plan are being executed. The main challenges are to guarantee the completeness of the approach and make sure that planning does concentrate on regions of the state space that are most crucial given the state of execution. Surprisingly, simple modifications of existing stochastic planners yield an efficient approach for concurrent planning and plan execution. We formalize this approach and analyze its characteristics. Experimental results show that such a continuous planning paradigm offers significant benefits, most notably a significant cost reduction relative to existing realtime planning and execution strategies.

1 Introduction

We address realtime stochastic planning scenarios in which an agent receives a goal and must minimize the *combined* cost of planning and plan execution (Horvitz 1987; Dean and Boddy 1988; Zilberstein and Russell 1993). The cost can be simply the total time needed to reach the goal—be it used to search for a plan or for plan execution—or it can be measured using a time-dependent utility function (Boddy and Dean 1989). Examples of such scenarios include service mobile robots (Boddy and Dean 1994; Koenig *et al.* 1996), real-time fault diagnosis (Rish *et al.* 2002), and real-time planning for satellite systems (Kortenkamp *et al.* 2011).

Figure 1 illustrates four possible strategies for realtime scenarios, with black and red line segments representing, respectively, the planning and plan execution durations. The first three cases represent existing paradigms in which the agent alternates between planning and executions, never performing both concurrently. In method (a), first the agent computes the optimal plan and then executes it. While execution time is minimized, the combined time is substantial. In method (b), first the agent computes a complete but non-optimal plan and then it executes that plan. While plan execution takes more time, the combined cost is reduced. Research aimed at minimizing the combined cost of planning and execution—including any required meta-reasoning time—was proposed by Dean and Boddy (1988) and Horvitz and Breese (1990) among others.

In method (c), the agent performs a limited amount of search that results in an incomplete plan and then it executes that plan until additional planning is needed, repeating this cycle until the task is completed. This approach has been used in determinization-based planners such as FF-Replan (Yoon *et al.* 2007) and in various other online planning methods and autonomous robot architectures (Nourbakhsh 1997). The combined cost in this case can be lower or higher compared to the previous methods depending on the application domain and online planning technique.

The fundamental question addressed in this paper is whether it is feasible to perform planning continuously during plan execution so that the plan is constantly improving in both value and domain coverage, and neither planning nor execution delay the other process. This approach, illustrated by method (d), requires planning and plan execution to share access to the current partial plan with proper synchronization mechanisms to guarantee the correctness and integrity of each process. The potential benefit of such a continuous planning paradigm is that it can fully exploit execution time for planning, reducing the overall cost of the solution.

There have been significant efforts to develop *continuous planning* or *continual planning* systems in which plan generation and plan execution are interleaved (Brenner and Nebel 2009; desJardins *et al.* 1999; Myers 1999). Existing continual planning architectures often include sequential planning and execution stages, sometimes due to lack of domain knowledge. That is, they generate an approximate plan and execute it without making any further changes to the plan until a problem is detected during execution. At that point, execution is stopped and the plan is modified to account for previously unforeseen conditions; the new plan is then executed and the process repeats until a goal is reached. Our

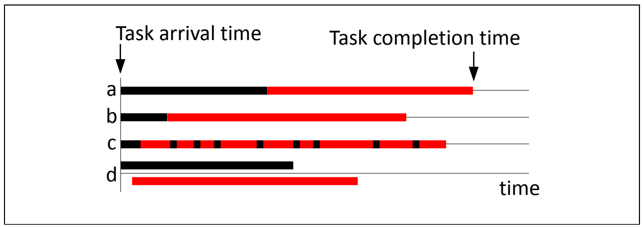

Figure 1: Four strategies for interleaving periods of planning (shown in black) and period of execution (shown in red).

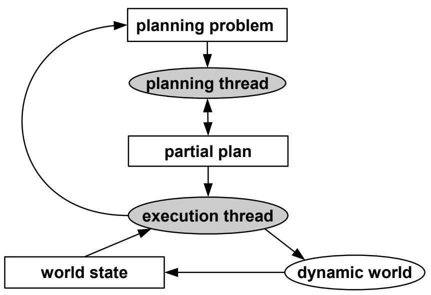

Figure 2: Data flow between planning and execution threads.

goal is to better exploit the fact that executing actions in the real-world can take a significant amount of time that can be used for planning, potentially reducing idle time during replanning and the overall cost.

Some continual planning efforts do allow planning to be performed *during* plan execution, most notably the AMPLE system (Chanel *et al.* 2014). However, our approach is easier to implement and, unlike previous works, is provably complete. Our approach effectively minimizes the combined execution and added planning time and it can achieve near optimal results with respect to that objective.

We assume that the domain is modeled as a Markov decision process (MDP), specified compactly using PPDDL encoding of action schemas (Younes and Littman 2004). We examine ways to perform planning and execution concurrently in an *uninterrupted* fashion. Our objective is to develop a paradigm that is amenable to an analytical evaluation and could provide performance guarantees.

Figure 2 shows the interaction between two threads that perform planning and execution concurrently. The planning thread maintains a data structure that represents the best partial solution, given the current planning problem. The execution thread selects actions based on that plan, observes the outcome and possibly updates the planning problem to account for the current state of execution. Implementing such a system in which planning and execution progress simultaneously presents several fundamental questions. What happens to the current best partial solution when the planning problem is updated? Is it possible to guarantee the completeness of the approach? How can computation be focused on the states that are most relevant to the actual execution path? What sorts of synchronization mechanisms are necessary? In this paper we answer these questions and present an approach that leverages effectively the time spent on action execution for planning. The approach is designed for stochastic shortest path problems and it exploits prior knowledge of the time it takes to execute any action in the domain.

The rest of the paper is structured as follows. In Section 2 we formulate the problem. In Section 3 we introduce our concurrent planning approach, followed by an analysis of its theoretical properties in Section 4. In Section 5 we present experimental results on three complex stochastic domains and compare our approach with other continual planning approaches. Finally, in Section 6 we summarize the contributions and discuss future work.

2 Problem Formulation

We consider problems in which an agent is assigned a task and the objective is to minimize the total time spent computing and executing a plan to complete it. The task itself can be modeled as a special type of MDP called *stochastic shortest-path problem* (SSPP), which is a tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{C}, s_0, \mathcal{G} \rangle$, where $\mathcal{S}$ is a finite set of states; $\mathcal{A}$ is a finite set of actions; $\mathcal{T}(s'|s,a)$ is a stationary transition function specifying the probability of outcome state s' when action a is executed in state s; $\mathcal{C}(s,a)$ is the positive cost of executing action a in state s; $s_0 \in \mathcal{S}$ is a given start state; and $\mathcal{G} \subset \mathcal{S}$ is a set of absorbing goal states.

Starting in state s_0, the objective is to reach one of the goal states while minimizing the expected cumulative cost. We assume that the domain does not include *dead ends*—states from which a goal cannot be reached; under this assumption, there is always a *proper policy* that eventually reaches a goal with probability 1, so there is no need to use discounting.

When the cost of planning is discarded, an optimal solution to a SSPP can be represented as a policy π^* that maps each (reachable) state to an action: $\pi^* : \mathcal{S} \rightarrow \mathcal{A}$. The Bellman equation defines a value function over states, $V^*(s)$, from which an optimal policy π^* can be extracted:

$$V^*(s) = \min_a [\mathcal{C}(s,a) + \sum_{s' \in \mathcal{S}} \mathcal{T}(s'|s,a) V^*(s')] \qquad (1)$$

Computing optimal solutions to SSPPs is intractable. While SSPPs can be solved in time polynomial in the size of the state space, the state space itself is exponential in the size of the problem description (we use compact representations such as PPDDL to represent the SSPP).

We use Π to denote a planning strategy that continuously improves the partial plan and works concurrently with execution. Since the planner produces a sequence of partial plans that are influenced by the progress being made towards the goal, it is not meaningful in this case to talk about the value of a specific intermediate partial plan. Instead, we propose the following framework to define the comprehensive utility of Π.

Each planning strategy Π induces a distribution of runtime *execution trajectories*, where each trajectory $\mathcal{E}$ consists of a sequence of state-action pairs $(s_0, a_0), (s_1, a_1), ..., (s_k, \cdot)$, such that $s_k \in \mathcal{G}$ and $\forall_{1 \leq i \leq k} \mathcal{T}(s_i | s_{i-1}, a_{i-1}) > 0$. The value of a trajectory is defined as $V(\mathcal{E}) = \sum_{i=0}^{k} \mathcal{C}(s_i, a_i)$.

Given a planning strategy Π (e.g., one of the methods in Figure 1), we use the notation $\mathcal{E} \in \Pi$ to indicate that applying strategy Π to the SSPP could result in execution trajectory $\mathcal{E}$. Using this notation, our overall goal is to find a planning strategy that minimizes the expected combined cost of planning and execution, represented as follows:

$$\min_{\Pi} \mathop{\mathbb{E}}_{\mathcal{E} \in \Pi} [V(\mathcal{E}) + \kappa T(\mathcal{E})] \qquad (2)$$

where $T(\mathcal{E})$ is the total amount of time spent exclusively on planning (i.e., not performed concurrently with action execution) when trajectory $\mathcal{E}$ occurred, and κ is a constant that relates units of time in the real world to unit of costs in the

SSPP formulation of the problem. For example, if executing each action in the real world takes 1 second and planning time is measured in milliseconds, then $\kappa = 10^{-3}$, provided that the goal is to minimize the cumulative combined planning and execution time.

3 Concurrent Planning and Execution

The combined cost to be minimized in Eq. (2) depends on the expected value of the resulting trajectory as well as on the total time the strategy spends exclusively on planning. Therefore, ideally we would like to perform most of the planning during action execution, yet produce trajectories with expected cost that is near optimal.

A common approach to solving SSPPs is to employ search-based algorithms that use heuristics to focus computation on relevant parts of the domain. In this work we use FIND-and-REVISE (Bonet and Geffner 2003a), a general schema for heuristic search algorithms that subsumes popular MDP solvers such as LAO* (Hansen and Zilberstein 1998; 2001a) and LRTDP (Bonet and Geffner 2003b).

FIND-and-REVISE (FaR) algorithms iteratively search (FIND) the so-called *greedy graph* of the current value function for states that don't satisfy some convergence criteria, and then update these states using Bellman backups (REVISE). The greedy graph of a value function—also known as the best partial plan—is the set of states that can be reached from the start state following a greedy policy on the value function. To guide earlier stages of the search towards the more relevant parts of the domain, FaR algorithms use an heuristic function as an initial estimate for state values. Moreover, FaR algorithms are guaranteed to converge to the optimal value function (within some tolerance) as long as the FIND procedure is systematic, that is, an infinite number of calls to FIND would not starve any states in the greedy graph. The planners we use satisfy this assumption.

To minimize the combined cost of execution and planning time, we could try running a FaR algorithm with a fixed initial state in parallel to execution, and simply execute actions as soon as they are available, regardless of whether the algorithm has converged or not. We call this strategy *naive continual planning* and show that it is incomplete.

Proposition 1. *The naive continual planning strategy is incomplete.*

Proof. The strategy is incomplete because, before convergence, FaR algorithms continuously modify the best partial plan, and nothing prevents them from creating—during early iterations in particular—a partial plan that leads to states that are not reachable by the final optimal plan. Consequently, by executing actions according to a plan generated in early iterations, the agent could reach a state that is no longer part of the plan and may be ignored by all future iterations until convergence. Therefore, this strategy cannot guarantee that the agent reaches the goal and is thus incomplete. □

Modifying the Planning Algorithm

We can get around this problem by forcing FaR to compute plans rooted at the current state of execution, instead of continuing to build a plan that starts at the original initial state. This has the additional advantage of potentially reducing computation effort, since parts of the domain become unreachable from the current state and can be ignored.

Algorithm 1: PLAN-EX: A concurrent planning and execution strategy for solving MDPs

```
execution-thread
    input: SSPP problem $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{C}, s_0, \mathcal{G} \rangle$
1   $s_c \leftarrow s_0$
2   Start planning-thread
3   Wait $t_I$ for initial plan
    while $s_c \notin \mathcal{G}$ do
4       $a_c \leftarrow$ Partial-Plan.get-action($s_c$)
5       $s_0^c \leftarrow$ create-new-SSPP ($s_c$,$a_c$)
6       Update $s_0^c$ in planning-thread
7       $s_c \leftarrow$ ExecuteAction($s_c$,$a_c$)

planning-thread
    input: MDP problem $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{C}, s_0, \mathcal{G} \rangle$
8   $\hat{s} \leftarrow s_0$
9   while $s_0^c \notin \mathcal{G}$ do
10      $\hat{s} \leftarrow$ Updated $s_0^c$ from execution-thread
11      FIND a set of states $S_F$ in the greedy graph of $\hat{s}$
12      REVISE $V(s)$ for all $s' \in S_F$ with a Bellman
        backup
```

Furthermore, when an action is executed, the planner should take that into account in order to further reduce computational effort. However, since MDP algorithms compute stationary policies, one challenge is to ensure that executing a particular action (that appears best) at a given state, does not commit us to executing that same action—which might not be optimal—if this state is reached again in the future.

We solve this problem by introducing a new SSPP problem every time an action is to be executed, and modifying FaR so that it always attempts to solve the latest SSPP available. Before explaining how we modify the FaR schema, we first describe how these SSPPs are constructed. Formally, let s_c be the agent's current state, and let a_c be the current best action for that state. Consider SSPP $\langle \mathcal{S} \cup \{s_0^c\}, \mathcal{A}, \mathcal{T}^c, \mathcal{C}^c, s_0^c, \mathcal{G} \rangle$, where the transition function $\mathcal{T}^c$ is identical to $\mathcal{T}$, except for the addition of:

$$\begin{cases} \mathcal{T}^c(s'|s_0^c, a_c) = \mathcal{T}(s'|s_c, a_c) & \text{if } s' \neq s_0^c \\ \mathcal{T}^c(s_0^c|s_0^c, a) = 1 & \text{if } a \neq a_c \\ \mathcal{T}^c(s'|s_0^c, a) = 0 & \text{if } a \neq a_c \text{ and } s' \neq s_0 \end{cases} \quad (3)$$

This transition function is illustrated in Figure 4 in the Appendix. The new cost function $\mathcal{C}^c$ is identical to $\mathcal{C}$ with the addition of $\forall_{a \in \mathcal{A}}$, $\mathcal{C}^c(s_0^c, a) = 0$. Therefore, this new SSPP is the same as the original one, except for the introduction of s_0^c, a copy of the current state s_c for which the only action that can make progress towards the goal is a_c—the actual action to be executed next. This SSPP has the following property.

Proposition 2. *An optimal policy for the new SSPP with initial state s_0^c provides an optimal policy for every state in the original SSPP that can be reached after executing a_c.*

Proof. It is easy to see that this is true, since s_0^c has only one action that can make progress towards the goal—leading to all possible successors of s_c after executing a_c—and every other state that can be reached in the new SSPP has the same transition and cost functions as in the original SSPP. □

We now show how to modify FaR so that during execution it repeatedly attempts to solve the most recent of these SSPPs. The resulting strategy, PLAN-EX, uses two threads running simultaneously (see Algorithm 1) that access the shared data structure `Partial-Plan`. The execution thread executes actions specified by the plan and updates the current state accordingly (line 7), calls `create-new-SSPP` to create the next initial state s_0^c (line 5) and updates this information on the planning thread (line 6). The planning thread continuously runs a modified version of FaR on the most recent SSPP (lines 10-12). Note that the procedure `create-new-SSPP` uses a hash map with keys given by state-action pairs, to guarantee that there is a unique copy state s_0^c for each pair (s_c, a_c). Algorithm 2 in the Appendix describes procedure `create-new-SSPP` in more detail.

`Partial-Plan` maintains the values and best actions for all states ever encountered by the FIND procedure. For this it keeps a hash map that stores the current estimates for the state values, using states as keys. All Bellman backups performed by the REVISE procedure are done by accessing the values stored by `Partial-Plan` and updating the hash map. This information can be accessed by the execution thread (see line 4 in Algorithm 1) through a call to method `Partial-Plan.get-action`, which returns an action chosen greedily on the current value estimates (initialized with a heuristic function), breaking ties randomly. Note that to prevent race conditions and ensure data integrity, some care must be taken when performing concurrent Bellman backups and accessing `Partial-Plan`. This is achieved through the use of a mutex to guarantee that all reads/writes to `Partial-Plan` are atomic.

To see how the planning thread works, note that standard FaR consists of two phases performed iteratively until convergence. In our modified version of FaR, at each iteration the planning thread checks if the initial state for the search was changed by the execution thread (line 10 of Algorithm 1), and starts the search (FIND step) on the corresponding latest SSPP. As a consequence, in the REVISE step PLAN-EX only performs Bellman backups of states that can be reached from the current state of execution, ignoring parts of the domain that are no longer reachable. This modified version of FaR has the following property.

Proposition 3. *If the initial heuristic is admissible, then every cost estimated by PLAN-EX represents a lower bound on the optimal cost of reaching a goal.*

Proof. This is easy to show by induction on the Bellman backup operator used by the REVISE procedure. □

4 Completeness of PLAN-EX

In this section we show that PLAN-EX is complete in that it eventually reaches a goal state whenevWe use the fact that `create-new-SSPP` generates a unique initial state for each state-action pair (s_c, a_c). Note that this property is not strictly necessary to ensure completeness of PLAN-EX, but it simplifies the proof.

Lemma 1. *The set of unique start states s_0^c that will be created by line 5 of PLAN-EX during an application of the algorithm to a given SSPP is finite, even when trajectories have unbounded length.*

Proof. This is obviously true because every s_0^c is uniquely determined by s_c and a_c. Since $\mathcal{S}$ and $\mathcal{A}$ are finite sets, there is a finite number of different states s_0^c. □

Theorem 1. *If the heuristic used by modified FaR is admissible, PLAN-EX is complete.*

Proof. Suppose PLAN-EX is not complete, that is, it results in an infinite length trajectory. By Lemma 1 and the pigeonhole principle, there exists a finite set of states I that will be backed-up infinitely often through calls to REVISE (line 12). Since states in $S \setminus I$ appear a finite number of times, at some point during the course of PLAN-EX the FIND procedure (line 11) must only return states in I. Moreover, set I must contain at least one goal state, since it can be shown that otherwise the estimated costs grow without bound, contradicting Proposition 3. Therefore, at some point during execution PLAN-EX becomes equivalent to performing asynchronous value iteration with state set I, which is known to converge to the optimal cost for all states in I. Finally, this implies that PLAN-EX has found an optimal path to the goal for any state within I, contradicting the initial assumption that PLAN-EX is not complete. □

5 Experimental Results

We present experimental results on three stochastic domains, showing the benefits of our concurrent planning and execution approach. For the first two domains, we compare seven planning strategies: computing a complete optimal plan using LAO*/LRTDP and then executing it, computing a complete approximate plan using weighted LAO* and then executing it (w-LAO*), a continual planning approach based on the most-likely-outcome determinization (DET), and three versions of PLAN-EX: one using LAO* (PE-L*) as the core planning algorithm, one using LRTDP (PE-LR) and another using weighted LAO* (PE-w*). Additionally, in the third domain we investigate the scalability of PLAN-EX to problems that cannot be completely solved off-line (e.g., due to memory requirements).

Racetrack Domain

The first domain is a modified version of the racetrack domain (Sutton and Barto 1998), a well-known reinforcement learning (RL) benchmark. The original problem involves a simulation of a race car on a discrete track of some length and shape, where a starting line has been drawn on one end and a finish line on the opposite end. The goal is to reach the finish line in the minimum number of steps. The state of the car is determined by its location and its two-dimensional velocity. The car can change its speed in each of the two

	LAO*	**LRTDP**	**w-LAO***	**DET**	**PE-L***	**PE-w***	**PE-LR**
$\mathbb{E}[V(\mathcal{E})]$	**57.43**	**57.43**	67.72	78.07	62.23	66.75	68.80
$\mathbb{E}[\kappa T(\mathcal{E})]$	41.80	52.80	34.64	1.31	**1.00**	**1.00**	**1.00**
$\mathbb{E}[Total]$	99.23	110.23	102.36	79.38	**63.23**	67.75	69.80
Δ Cost	72.78%	91.93%	78.23%	38.22%	**10.10%**	17.97%	21.54%

Table 1: Performance of seven planning strategies on the racetrack domain. The last row represents the increase in overall cost with respect to the theoretical lower bound.

dimensions by at most 1 unit, giving a total of nine possible actions, all at a unit cost. After applying an action, there is a probability p_s that the resulting acceleration is zero, simulating failed attempts to accelerate/decelerate because of unpredictably slipping on the track.

In the modified version, there is also a probability p_e that the outcome corresponds to that of a similar action, simulating driving errors. Additionally, if the car collides with a wall, it stops at the wall and can take normal actions in the next turn but at a higher cost. Intuitively, higher action costs in this domain correspond to higher execution periods, giving us an opportunity to test PLAN-EX with variable duration actions. In all the experiments we assumed that the cheapest action takes 250 ms to execute, and the durations of more expensive actions are proportional to their costs, which implies a value of $\kappa = 1/250\,\text{ms}^{-1}$. Figure 5 in the Appendix shows the track we used in our experiments, which includes approx. 78,000 states.

Table 1 shows the performance of all planning strategies on the racetrack problem, based on 100 simulations. For w-LAO* the weight was set to 50.0, which was the best value based on trial-and-error with different weights in the range [2.0 – 100.0]; we used the same weight for PE-w*. The last row of Table 1 represents the increase in overall expected cost defined by Eq. (2) as a percentage of the theoretical lower bound, which equals the cost of executing an optimal policy ignoring planning time. PE-L*, PE-w* and PE-LR outperform all other methods by more than a 28%, 20% and 16% decrease in combined cost, respectively, the best one being PE-L* with an increased cost of only 10%. PLAN-EX instances dominate all off-line methods primarily due to a much faster planning time. With respect to DET, whose planning time was also low, the benefits offered by PE-L* and PE-w* are due to a much lower expected cost of the resulting trajectories.

We also analyzed the impact of action duration—which affects the amount of time available for planning—on the average cost of the trajectories obtained with PE-L*. Figure 3 shows the relative error with respect to the optimal cost as a function of action duration, computed from 100 simulations; in all cases the initial planning time was set to the equivalent of one action. As expected, the relative error decreases with increasing action duration, and for durations higher than 200 ms the average cost is very close to the lower bound. These encouraging results suggest that with modest action durations ($> 200\,\text{ms}$), PLAN-EX can obtain near-optimal results, *with little initial planning time*.

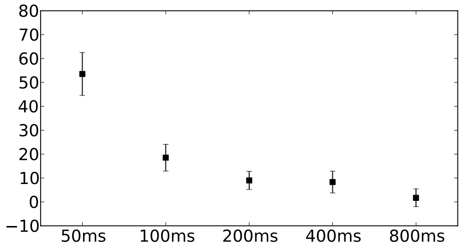

Figure 3: Relative difference between $\mathbb{E}[V(\mathcal{E})]$ (PLAN-EX) and $V^*(s_0)$ as a function of action duration on the racetrack domain. The lines show 95% confidence intervals around the mean.

Sailing Domain

The second domain is the sailing domain described by Vanderbei (1996), in which a sailboat has to find the shortest path between two points on a grid under dynamic wind conditions. The sailboat's position is represented as a pair of coordinates on a two dimensional grid. The sailboat can move to any neighboring position except directly against the direction of the wind, giving a total of 7 applicable actions. Actions are deterministic and uncertainty in the domain is due to stochastic changes in the direction of the wind. Each action has a cost that depends on the direction of movement and the direction of the wind. We experimented on a 50×50 grid with 20,000 states, with the start state at $(0, 0)$ and the goal at $(20, 20)$. As we did for the racetrack domain, we assumed that the cheapest action takes 250 ms to execute, and that durations of more expensive actions are proportional to their costs. Hence, $\kappa = 1/250\,\text{ms}^{-1}$.

We again set an initial planning time for all PLAN-EX instances to 250 ms (lowest action cost). The weight for w-LAO* was 5.0, selected based on trial-and-error in the range [2.0 – 10.0]; the same weight was used for PE-w*. We should note that finding the best weight to use for w-LAO* was a time-consuming process. Moreover, its performance seems to be very sensitive to the choice of weight; with a poor choice, the performance of w-LAO* can degrade to that of standard LAO* or worse. Furthermore, experiments not reported here suggest that the choice of weight doesn't carry over between different problem instances of the same domain. In contrast, PLAN-EX seems to be less sensitive to the choice of initial planning time, and simply using the duration of one action worked well in all of our experiments on both the racetrack and the sailing domain.

Table 2 shows the performance of the seven planning strategies on the sailing domain, based on 100 simulations. The best strategies for this problem were w-LAO* (5.51%) and PE-w* (6.61%). The optimal off-line planning algorithms (LAO* and LRTDP) performed poorly on this problem, with planning time cost significantly exceeding execution cost. Nevertheless, the concurrent versions of these algorithms (PE-L* and PE-LR) significantly improved upon their respective off-line versions. Finally, as in the case of the racetrack problem, DET performed poorly due to poor plan quality and a large increase in the expected trajectory cost with respect to the optimal (164% increase).

	LAO*	**LRTDP**	**w-LAO***	**DET**	**PE-L***	**PE-w***	**PE-LR**
$\mathbb{E}[V(\mathcal{E})]$	**91.28**	**91.28**	92.51	241.03	109.01	96.31	111.54
$\mathbb{E}[\kappa T(\mathcal{E})]$	116.76	83.60	3.80	3.56	**1.0**	**1.0**	**1.0**
$\mathbb{E}[Total]$	208.04	174.88	**96.31**	244.59	110.01	97.31	112.54
Δ Cost	127.91%	91.59%	**5.51%**	167.96%	20.52%	6.61%	23.29%

Table 2: Performance of the seven planning strategies on the sailing domain. The last row represents the increase in overall cost with respect to the theoretical lower bound.

As we did for the racetrack domain, we analyzed the impact of action duration on the expected trajectory cost of PLAN-EX. The results showed again a similar trend to what is depicted in Figure 3, and are therefore omitted.

Scalability of PLAN-EX to Larger Problems

In the first set of experiments, we selected problems that are complex, but can still be tackled using off-line solvers. This allowed us to compute optimal plans, establish a lower bound on the overall cost, and perform a comprehensive comparison of different planners. Now, we try to assess the scalability of PLAN-EX to harder problems that cannot be solved off-line due to time and/or memory requirements.

To this end, we experimented on the Canadian Traveler Problem (Papadimitriou and Yannakakis 1991). In this problem, an agent must find shortest paths on a graph in which each edge can be blocked with some probability due to bad weather. The objective is to minimize the expected cost of reaching the goal from the start state; this expected cost is evaluated only on those weather instances in which a path between the start and goal exists. This problem can be modeled as a belief-state MDP in which the state of the agent is given by its location and a vector of edge beliefs; for each edge there are three possibilities: unknown, known to be open, known to be blocked. Therefore, the size of the belief space is exponential in the number of edges and thus even small instances are hard to solve.

We experimented on a problem instance reported by (Bonet and Geffner 2012) consisting of 20 nodes and 49 edges. In all cases, both LAO* and w-LAO* quickly ran out of memory (5 GB). However, PE-L* with 500 ms for planning during actions was able to solve 18 out of 20 runs of the problem, with an expected cost of approximately 152, while PE-w* (with a weight of 20.0) solved 14 instances with an expected cost of approximately 155; for reference, the DET strategy had an expected cost of 188. These result suggests that PLAN-EX can be used to efficiently solve hard problems, even when the underlying FIND-and-REVISE planner cannot solve them.

6 Conclusion

We present a novel approach to minimize the combined cost of planning and execution in stochastic environments by performing planning and execution concurrently. The proposed approach, PLAN-EX, is based on a simple modification of an existing heuristic search scheme (FIND-and-REVISE) that allows the planning thread to adapt its operation to the current state of execution with negligible overhead. The approach maintains continuous execution by choosing greedy actions based on the current value estimates (initialized with a heuristic function), and it can be easily extended to exploit user-provided policies to choose actions reactively, without losing any of its formal guarantees. We analyze the characteristics of this new paradigm and show that it is complete. Furthermore, it performs well in practice on three complex stochastic domains.

Among the planning paradigms we considered, one in particular was designed specifically as a way to minimize the combined runtime of planning and execution albeit in a different way. This existing approach employs an anytime planning procedure to create the plan and then executes that plan (Horvitz 1987; Dean and Boddy 1988; Zilberstein and Russell 1993; Zilberstein *et al.* 1999). We used weighted LAO* (w-LAO*) as the anytime planner and compared it to our approach, using the *best* possible weight for w-LAO*. Even using the best weight for w-LAO*, the concurrent planning approach significantly outperformed w-LAO* and the other baseline approaches in one domain, and had similar performance to w-LAO* in another. Furthermore, our approach required no special parameter tuning and it consistently performed well, while w-LAO* was outperformed by a determinization-based re-planner in the racetrack domain. Finally, we also show results suggesting that our concurrent planning approach can be used to efficiently solve problems even when the underlying search algorithm fails to solve them due to memory requirements.

Note that although in this work we used FIND-and-REVISE algorithms as the core planners, our approach can be easily adapted to other stochastic problem solvers outside of this schema, such as UCT (Kocsis and Szepesvári 2006). It could also be trivially used in conjunction with the more simple AO* algorithm, or its recent anytime variant, for deterministic problems or problems without cyclic policies (Nilsson 1980; Bonet and Geffner 2012). Furthermore, a concurrent planning approach similar to the one presented here can be combined with incomplete methods that prune parts of the state-space to reduce computation time (Trevizan and Veloso 2012; Pineda and Zilberstein 2014). The benefit of applying our approach in these contexts is that it will continuously increase the *coverage* of the partial plan during execution, which is likely to improve performance.

Another interesting direction for future work is to examine more general objective and cost functions that affect the tradeoff between planning time and execution cost, and study their impact on the time that should be dedicated to planning before execution starts. In our formulation, we assume that planning time and action costs are related by a constant factor κ, and our experiments suggest that an initial planning time equivalent to the duration of one action works well in this case. However, more complex relationships between planning time and action cost are possible, for example when a time-dependent utility function defines the overall value (Dean and Boddy 1988). In that case, it may be beneficial to use utility-driven meta-reasoning mechanisms to determine the most appropriate initial planning time (Hansen and Zilberstein 2001b).

Acknowledgments

Support for this work was provided in part by the National Science Foundation under grants IIS-1405550.

References

Mark Boddy and Thomas Dean. Solving time-dependent planning problems. In *Proceedings of the 11th International Joint Conference on Artificial Intelligence*, pages 979–984, 1989.

Mark Boddy and Thomas L. Dean. Deliberation scheduling for problem solving in time-constrained environments. *Artificial Intelligence*, 67(2):245–285, 1994.

Blai Bonet and Hector Geffner. Faster heuristic search algorithms for planning with uncertainty and full feedback. In *International Joint Conference on Artificial Intelligence*, pages 1233–1238, 2003.

Blai Bonet and Hector Geffner. Labeled RTDP: Improving the convergence of real-time dynamic programming. In *Proceedings of the 13th International Conference on Automated Planning and Scheduling*, pages 12–21, 2003.

Blai Bonet and Hector Geffner. Action selection for mdps: Anytime ao* versus uct. In *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence*, 2012.

Michael Brenner and Bernhard Nebel. Continual planning and acting in dynamic multiagent environments. *Autonomous Agents and Multi-Agent Systems*, 19(3):297–331, 2009.

Caroline P. Carvalho Chanel, Charles Lesire, and Florent Teichteil-Königsbuch. A robotic execution framework for online probabilistic (re) planning. In *Proceedings of the 24th International Conference on Automated Planning and Scheduling*, pages 454–462, 2014.

Thomas Dean and Mark Boddy. An analysis of time-dependent planning. In *Proceedings of the 7th National Conference on Artificial Intelligence*, pages 49–54, Saint Paul, Minnesota, 1988.

Marie E. desJardins, Edmund H. Durfee, Charles L. Ortiz, and Michael J. Wolverton. Continual planning and acting in dynamic multiagent environments. *AI Magazine*, 20(4):13–22, 1999.

Eric A. Hansen and Shlomo Zilberstein. Heuristic search in cyclic AND/OR graphs. In *Proceedings of the 15th National Conference on Artificial Intelligence*, pages 412–418, 1998.

Eric A. Hansen and Shlomo Zilberstein. LAO*: A heuristic search algorithm that finds solutions with loops. *Artificial Intelligence*, 129(1-2):35–62, 2001.

Eric A. Hansen and Shlomo Zilberstein. Monitoring and control of anytime algorithms: A dynamic programming approach. *Artificial Intelligence*, 126(1-2):139–157, 2001.

Eric J. Horvitz and Jack S. Breese. Ideal partition of resources for metareasoning. Technical Report KSL-90-26, Stanford University, Computer Science Department, 1990.

Eric J. Horvitz. Reasoning about beliefs and actions under computational resource constraints. In *Proceedings of the 3rd Conference on Uncertainty in Artificial Intelligence*, pages 429–444, 1987.

Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *Proceedings of the 17th European conference on Machine Learning*, pages 282–293, 2006.

Sven Koenig, Richard Goodwin, and Reid G. Simmons. Robot navigation with Markov models: A framework for path planning and learning with limited computational resources. In Leo Dorst, Michiel Lambalgen, and Frans Voorbraak, editors, *Reasoning with Uncertainty in Robotics*, volume 1093 of *LNCS*, pages 322–337. Springer, 1996.

David Kortenkamp, Peter Bonasso, David Musliner, Michael Pelican, and Jesse Hostetler. Embedding planning technology into satellite systems. In *Proceedings of AIAA Infotech@Aerospace*, pages 1–10, 2011.

Karen L. Myers. CPEF: A continuous planning and execution framework. *AI Magazine*, 20(4):63–69, 1999.

Nils J. Nilsson. *Principles of artificial intelligence*. Morgan Kaufmann, 1980.

Illah Nourbakhsh. *Interleaving Planning and Execution for Autonomous Robots*. Kluwer Academic Publishers, 1997.

Christos H Papadimitriou and Mihalis Yannakakis. Shortest paths without a map. *Theoretical Computer Science*, 84(1):127–150, 1991.

Luis Pineda and Shlomo Zilberstein. Planning under uncertainty using reduced models: Revisiting determinization. In *Proceedings of the 24th International Conference on Automated Planning and Scheduling*, pages 217–225, 2014.

Irina Rish, Mark Brodie, and Sheng Ma. Accuracy vs. efficiency trade-offs in probabilistic diagnosis. In *Proceedings of the 18th National Conference on Artificial Intelligence*, pages 560–566, 2002.

Richard S. Sutton and Andrew G. Barto. *Introduction to Reinforcement Learning*. MIT Press, Cambridge, MA, USA, 1998.

Felipe W. Trevizan and Manuela M. Veloso. Short-sighted stochastic shortest path problems. *Proceedings of the 22nd International Conference on Automated Planning and Scheduling*, pages 288–296, 2012.

Robert J. Vanderbei. Optimal sailing strategies. Technical report, Princeton University, Statistics and Operations Research Program, 1996.

Sung Wook Yoon, Alan Fern, and Robert Givan. FF-Replan: A baseline for probabilistic planning. In *Proceedings of the 17th International Conference on Automated Planning and Scheduling*, pages 352–359, 2007.

Håkan L. S. Younes and Michael L. Littman. PPDDL1.0: An extension to PDDL for expressing planning domains with probabilistic effects. *Technical Report CMU-CS-04-162*, 2004.

Shlomo Zilberstein and Stuart J. Russell. Anytime sensing, planning and action: A practical model for robot control. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence*, pages 1402–1407, 1993.

Shlomo Zilberstein, François Charpillet, and Philippe Chassaing. Real-time problem-solving with contract algorithms. In *Proceedings of the 16h International Joint Conference on Artificial Intelligence*, pages 1008–1015, 1999.

Appendix

Generating the new SSPP

Figure 4 illustrates the transition function of the SSPPs created during the course of running PLAN-EX. Briefly, given a state s_c and an action to be executed a_c, the new SSPP includes an additional copy state s_0^c that is used as initial state. The transition function for s_0^c when action a_c is executed is exactly the same as the transition function for the state-action pair (s_c, a_c), and for any other action the transition results in s_0^c with probability one. Therefore, given that the cost of executing a_c in state s_0^c is 0, planning for initial state s_0^c is equivalent to finding a plan for all successors of (s_c, a_c).

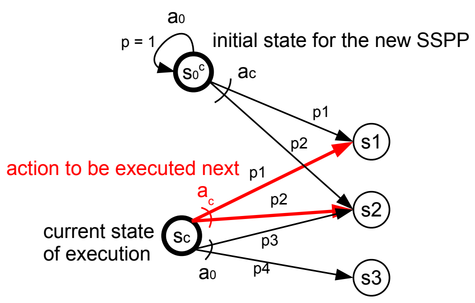

Figure 4: Illustration of the transition function for the new SSPPs.

Algorithm 2 shows the procedure to generate a new SSPP given the current state of execution. We use a hash map (`InitialStates`) to store all copy states generated during the course of PLAN-EX. Every time a copy state is created, `InitialStates` is queried to check if another copy state for the given state-action pair (s_c, a_c) already exists (lines 1-2). If not, then a state is created and stored in the hash map (lines 3-7). This guarantees that for any given state-action pair there is a unique copy state s_0^c.

Algorithm 2: Procedure to generate a new SSPP given the current state of execution

```
create-new-SSPP
    input: State-Action pair $\langle s, a \rangle$
    HashMap InitialStates
1   if InitialStates.hasKey($\langle s, a \rangle$) then
2       return InitialStates.get($\langle s, a \rangle$)
    else
3       Create state $s_0^c$
4       Create $\mathcal{T}^c$ for $s_0^c$ using Eq. (3)
5       Create $\mathcal{C}^c$ for $s_0^c$ s.t. $\forall_{a \in \mathcal{A}}$, $\mathcal{C}^c(s_0^c, a) = 0$
6       InitialStates.set($\langle s, a \rangle$,$s_0^c$)
7       return $s_0^c$
```

Track used for our experiments on the Racetrack domain

Figure 5 shows the track used for our experiments on the Racetrack domain. This track has size 120×60 and 77,954 states can be reached from the start state defined by the location labeled as S and zero speed in all directions. The goal is to reach the location labeled as G.

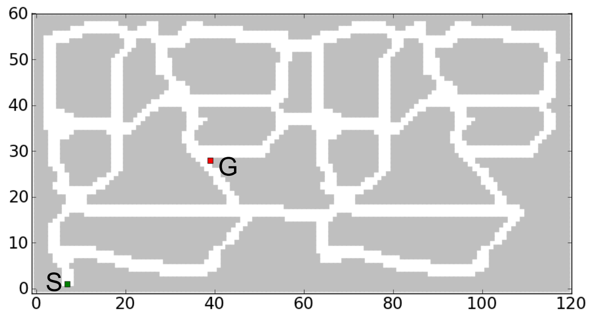

Figure 5: Map of the racetrack domain used in the experiments.