Generating Plans in Concurrent, Probabilistic, Over-Subscribed Domains

Li Li **Nilufer Onder**
Department of Computer Science
Michigan Technological University
1400 Townsend Drive
Houghton, MI 49931
{lili,nilufer}@mtu.edu

Abstract

Planning in realistic domains involves reasoning under uncertainty, operating under time and resource constraints, and finding the optimal set of goals to be achieved. In this paper, we present an algorithm called CPOAO* (Concurrent, Probabilistic, Oversubscription AO*) that is based on AO* and hence can reduce the size of the search space using informative heuristic functions. We introduce a novel dichotomy of concurrent actions. We call the concurrent actions that serve the purpose of decreasing the execution time "all-finish" actions because each action serves a different purpose and all must finish. We call the redundant, concurrent actions that increase the expected rewards of a plan "early finish" actions because the actions serve the same goal and the success of the earliest finishing one is sufficient. We explain our implemented algorithm, the heuristic functions we use, the experimental results, and our future work based on this framework.

Introduction

Generating plans for agents that operate in the real world presents two main challenges. First, there is uncertainty about the action effects as well as the state of the world. Second, the resources for carrying out the tasks are limited. Research in *probabilistic planning* deals with issues involving uncertainty (Kushmerick, Hanks, & Weld 1995; Bonet & Geffner 2005). *MDP-based planners* (Boutilier, Dean, & Hanks 1999) deal with both uncertain actions and resource limitations. Heuristics have been developed for dealing with time and resources in deterministic domains (Haslum & Geffner 2001). *Oversubscription planners* must select a subset of the goals to plan for because resource limitations do not allow all the goals to be achieved (Smith 2004; Benton, Do, & Kambhampati 2005). *Concurrent planners* generate plans with shorter execution times (makespan) by using parallel actions (Mausam & Weld 2005; Little & Thiebaux 2006). Our work extends this research by exploring the use of parallel actions to increase expected rewards, and by providing a framework to terminate actions to optimize resource usage.

The examples we use are inspired by the Mars rover domain (Bresina *et al.* 2002). To motivate our work, consider a problem where two pictures must be taken within 5 minutes. For simplicity, we assume that actions have fixed known durations. The rover has two cameras where cam_0 succeeds with probability 0.6, and takes 5 minutes; and cam_1 succeeds with probability 0.5, and takes 4 minutes. In a case where both pictures have a value of 10, the best strategy is to use cam_0 for one picture and cam_1 for the other in parallel. Because all the actions need to finish to collect the rewards, we call this *all-finish parallelism*. The total expected reward will be $10 \times 60\% + 10 \times 50\% = 11$. In a different case where the picture values are 100 and 10, the best strategy is to use both cam_0 and cam_1 in parallel to achieve the larger reward. When both cameras are used for the same picture, if cam_1 succeeds in achieving the target reward, we can abort cam_0 immediately. Because we use concurrent actions to achieve the same target reward and will abort all the other actions after the earliest successful action finishes, we call this case *early-finish parallelism*. Cam_1 finishes earlier than cam_0 and the expected reward for using cam_1 is $100 \times 50\% = 50$. If cam_1 fails, the expected remaining unachieved reward will be $100 \times (1 - 50\%) = 50$. Then the expected reward for action cam_0 is $50 \times 60\% = 30$. Therefore, the total expected reward is $50 + 30 = 80$. This is larger than the expected reward of using the cameras for different pictures, which is, $100 \times 60\% + 10 \times 50\% = 65$.

We have developed an algorithm called CPOAO* (Concurrent, Probabilistic, Oversubscription AO*) which extends the AO* framework to use concurrent actions of both kinds defined above. Other AO* based planners include LAO* (Hansen & Zilberstein 2001) which finds solutions with loops, and HAO* (Mausam *et al.* 2005) which deals with continuous resources and stochastic consumptions. Recent concurrent planners are CPTP (Mausam & Weld 2005), an MDP-based planner, and Paragraph (Little & Thiebaux 2006) which uses a Graphplan framework. Both planners use parallel actions to decrease the total execution time of plans and thus consider parallel actions that achieve different goals. These algorithms prevent a pair of actions that achieve the same goal from executing in parallel. Para-

graph (Little & Thiebaux 2006) implements this "restricted" concurrency model because rewards are not present in the model. In domains where an action cost model is defined but explicit rewards are not assigned to individual goals, there is no advantage in having parallel actions that serve the same goal, i.e., early-finish parallel actions. In our approach, we provide means for using redundant parallel actions to maximize expected rewards. To minimize resource consumption, our algorithm aborts the remaining redundant actions when the earliest finishing action establishes the common goal. Another example of the use of redundant actions comes from ProPL, a process monitoring language (Pfeffer 2005) where processes might include redundant parallel actions such as seeking task approval from two managers when one approval is sufficient. The focus of ProPL is on expressing and monitoring such actions. Our focus is in generating plans that use redundant actions.

The Planning Problem

The planning problems we consider are probabilistic and over-subscribed. The solution plans contain concurrent actions. A *planning problem* is formally defined as a 5-tuple (S, A, s_0, R, T) where S is the state space, A is the set of probabilistic actions, s_0 is the initial state, R is the reward set, and T is the time limit for plan execution.

A state $s \in S$ is a triple (s_p, s_r, s_a) where $s_p \subseteq P$ is a set of propositions, P is the set of all the domain propositions, s_r is a vector of numeric resource values, and s_a is the set of currently executing concurrent actions (we define *concurrent action sets* (CASs) below). Each resource is assumed to be discrete, we leave continuous resources to future work (Younes & Simmons 2004; Mausam *et al.* 2005).

An action a consists of a precondition list, a resource consumption vector, and a set of possible outcomes. An outcome o_i is defined as a triple (add(o_i), del(o_i), prob(o_i)) where add(o_i) is the add list, del(o_i) is the delete list, and prob(o_i) is the probability that this outcome happens. The total probability of all the possible outcomes of an action should be 1. We adopt the common semantics for action execution. Before an action can be executed, the preconditions must hold, and the amount of each resource must be greater than or equal to the value specified for that resource in the resource consumption vector. After an action is executed, the result is a set of states where for each outcome, the resources used are subtracted, the propositions in the add list are added, and the propositions in the delete list are deleted. The probability of the new state resulting from outcome o_i in state s_j is the probability of s_j multiplied by prob(o_i). The process of finding the results of an executed step is defined formally below after concurrent action sets are defined. In addition to the "regular" actions, the set of actions also includes a special *do-nothing* action as explained in the next section.

Each reward in R is a proposition-value pair. When a proposition is achieved, the corresponding value is added to the total rewards. When it becomes false, its value is subtracted from the total rewards. This way, the value of a proposition can contribute at most once to the total rewards collected. Our problem model does not include *hard goals*, i.e., goals that must be achieved, because a plan that guarantees the achievement of a goal with probability 1.0 might not exist. Instead, important goals are represented by assigning large rewards to them. In this regard, our probabilistic actions are similar in spirit to PPDDL actions with the exception of rewards being associated directly with propositions rather than through state transitions (Younes *et al.* 2005, pp. 854-855).

The solution to a planning problem is an optimal contingent plan which maximizes the expected reward of the initial state $s_0 = (s_{p0}, s_{r0}, s_{a0})$. In state s_0, the propositions in s_{p0} hold, the propositions in $P - s_{p0}$ do not hold, the resources have the levels given in the vector s_{r0}, and there are no actions that have been started ($s_{a0} = \emptyset$). We formally define a plan as a set of (state, CAS) pairs where a CAS is a *concurrent action set.*

For each action in a CAS, we specify the number of time units left to execute. The *duration* of CAS$_i$ (dur(CAS$_i$)) is defined as the duration of the shortest action in it. This allows us to evaluate the states resulting from the termination of the earliest action. The *result* of executing a concurrent action set CAS$_i$ in a state $s_i = (s_{pi}, s_{ri}, s_{ai})$ is a probability distribution over the resulting states and is defined with respect to the action that requires the minimum time to execute. Each state in the probability distribution corresponds to a distinct action outcome and is obtained by (1) merging s_{ai} and CAS$_i$ to obtain CAS$_m$, the set of all the concurrent actions that will be executing (2) using the add and delete lists to find the new set of propositions that hold, (3) using the resource consumption vector of the action to update the resource levels, and (4) updating the actions in the CAS set to reflect the execution time that has passed. Suppose that dur(CAS$_m$) $= t_j$ and action a_j has this duration. This means that a_j will be the first action to complete in CAS$_m$. Further suppose that a_j has n possible outcomes $o_1, \ldots, o_n$. Then, *result* (s_i, CAS$_i$) is defined as a probability distribution over n states where each state is obtained by applying an outcome to state s_i. The result of applying outcome o_k to s_i is a new state s_k defined as: $s_k = (s_{pk} = s_{pi} - \text{del}(o_k) + \text{add}(o_k), s_{rk} = s_{ri} - s_{rj}, s_{ak})$. s_{ak} is obtained by subtracting the action duration t_j from all the action durations in CAS$_m$, and removing the completed action a_j from CAS$_m$. The probability of the new state s_k is defined as prob(s_i) $\times$ prob(o_k). If outcome o_k represents a successful outcome then the redundant actions achieving the same goal can be aborted. If outcome o_k represents an unsuccessful outcome, only the finished action is removed.

If there are multiple actions which require t_j time units, then the effects of all of these actions are applied

in the above procedure. We assume that all action effects take place at the completion of the action. We therefore make a simplifying assumption and do not register any of the effects of the aborted redundant actions. One way to relax this assumption is to calculate the resource consumption using the ratio of the time an action executed to the total time it needs and subtract a proportional amount of the resources.

The underlying theoretical model of our framework can be thought of as a concurrent MDP (Mausam & Weld 2004). In order to provide the ability to utilize redundant parallel actions and to abort useless actions, the concurrent MDP model must be extended. For the former capability, the possible action combinations should include redundant as well as non-redundant actions. For the latter capability, the set of actions defined for the MDP should include all the possible abort actions.

Our objective is to find an optimal contingent plan which maximizes the expected reward of the initial state s_0. We use the following definition for the expected reward of a state s_i with respect to a plan π:

$$E_\pi(s_i) = \begin{cases} \sum_{s_j \in C(s_i)} P_{(i,j)} E_\pi(s_j) & \text{if } s_i \text{ is not a terminal state} \\ R_{s_i} & \text{if } s_i \text{ is a terminal state} \end{cases}$$

$C(s_i)$ is the set of resulting states under plan π, $P_{(i,j)}$ is the probability of entering state s_j from state s_i, and R_{s_i} is the sum of the rewards achieved when ending in state s_i. The *terminal states* are the leaves of the search graph and are explained in the next section. Note that a discount factor is not needed because a reward associated with a proposition can be received only once, and the time limit given in the planning problem limits the planning horizon.

Require: A planning problem (5-tuple).
Ensure: A solution plan that can contain concurrent actions.

1: WORK-G ← MAKEROOTNODE(s_0).
2: SOLN-G ← INSERT(SOLN-G, s_0)
3: **while** SOLN-G contains non-terminal tip states **do**
4: CHANGE-E ← $\emptyset$
5: **for each** s which is an unexpanded non-terminal tip state in SOLN-G **do**
6: APP-CAS ← COMPUTEAPPLICABLECAS(s)
7: **for all** ac in APP-CAS **do**
8: Apply ac on state s to generate the child states of s.
9: Calculate the heuristic values of the expected rewards for newly generated children states.
10: **end for**
11: Find BEST-CAS, the best concurrent action set for state s.
12: Expand SOLN-G to include BEST-CAS.
13: Update the expected reward of state s based on BEST-CAS.
14: Add the ancestor states of s into the set CHANGE-E if the expected reward of s has changed.
15: **end for**
16: **while** CHANGE-E is not empty **do**
17: Choose and remove a state $s' \in$ CHANGE-E that has no descendant in CHANGE-E.
18: Update the expected reward of s' by reselecting its best CAS..
19: **if** The expected reward of s' has changed **then**
20: Add the ancestor states of s' into CHANGE-E.
21: **end if**
22: **end while**
23: Recompute the best solution graph B by following the best CASs from the initial state s_0 to the tip states.
24: **end while**
25: **return** SOLN-G

Figure 1: The CPOAO* algorithm.

The CPOAO* Algorithm

The CPOAO* algorithm depicted in Figure 1 extends the AO* algorithm (Nilsson 1980). AO* is a heuristic based search algorithm which searches in an "and-or" graph. The hyperarcs in the "and-or" graph are particularly suitable to represent probabilistic actions. Each hyperarc represents one probabilistic action and each "and" set represents the set of possible results of one action. The nodes represent the states. The input to the CPOAO* algorithm is a planning problem, the output is a function that maps the reachable states to concurrent action sets. The output function represents an acyclic finite automaton, i.e., the solution plan does not contain loops due to two reasons similar to the HAO* framework (Mausam *et al.* 2005). First, the time limit that is given as part of the planning problem bounds the horizon. Second, every action has a non-zero duration and time is part of the state. Thus, even if the propositions and resources of the state remain the same after the execution of an action, the time component of the resource vector will be different.

The main data structure is a search graph called the *working graph (WORK-G)* (Hansen & Zilberstein 2001; Mausam *et al.* 2005). WORK-G is a *hypergraph* consisting of nodes that represent states and *hyperarcs* that represent the alternative concurrent action sets (CASs) which can be executed in a state. Each ending node of a hyperarc represents one possible outcome of the CAS. In our implementation we represent a plan by a *solution graph* (SOLN-G). CHANGE-E is a set that is used to propagate the changes in the expected rewards upward in the solution graph.

The main loop of the search starts at line 3, and continues until the solution graph is "complete," i.e., has no non-terminal tip states. The leaves of the search graph are the *terminal states.* There are three types of terminal states. The first type includes the states in which there are no applicable actions due to lack of preconditions or resources. The second type includes the states where all the rewards have been collected, and thus there is no need to execute any action. The third type includes the states entered after executing the special "do-nothing" action from a state s_j. In this case, s_j is a state where there are unachieved rewards and resources available, but leaving the current state

might result in the loss of already achieved rewards. For example, s_j might be a state where the rover is at the base location to which it needs to return, but the amount of resources will not be sufficient to take the rover back if it leaves. Thus, the best action to execute at s_j is the do-nothing action.

In line 5, all the unexpanded non-terminal tip states in the solution graph are expanded. To expand a search state s, we first find a set of all the applicable concurrent action sets using the procedure in Figure 2. In line 2 of the COMPUTEAPPLICABLECAS procedure we find all the single actions that are applicable and store it in the set I. An action is applicable in a state if its preconditions hold, there are sufficient resources, and it is not already executing. Next, we generate I^p, the power set of I. Each subset in I^p represents a set of actions that are candidates for concurrent execution. From I^p, we eliminate the subsets that contain incompatible actions. Two actions a_1 and a_2 are said to be *incompatible* if (1) a proposition occurs in one of the add lists of a_1 and in one of the delete lists of a_2, or (2) a_1 produces the opposite of a precondition of a_2. We assume that resources can be accessed in parallel, so we do not consider resources while computing incompatible actions. As known, computing the power set of the applicable actions generates an exponential number of subsets. Therefore, heuristics that can decrease the number of concurrent actions sets are valuable. We describe such a heuristic in the next section.

Require: A state.
Ensure: A set of applicable concurrent action sets.
1: Initialize I with all the unfinished actions of state s.
2: Add all applicable actions for state s into set I.
3: Generate I^p which is the power set of set I.
4: If a CAS contains incompatible actions, delete it from set I^p. The resulting set,denoted I^c, contains all applicable CASs for state s. For each applicable CAS, set its duration to the duration of the shortest action in it.
5: **return** I^c.

Figure 2: The COMPUTEAPPLICABLECAS algorithm.

After finding the set of concurrent actions applicable, we apply each CAS to state s using the *result* procedure defined in the previous section. The working graph is updated to include the resulting states. The heuristic value of each new state added to the working graph is computed using a heuristic function which is described in the next section. At the next step (line 11), the best CAS for state s is found using the following formula: $\text{argmax}_{cas \in \text{APP-CAS}} \text{result}(s, cas)$. The best CAS and its resulting states are marked to become part of the best solution graph (line 12). In lines 13 through 21, the expected value of the state s as well as its ancestors in the solution graph are updated using the best CAS.

When every state in the current best solution graph is either expanded or a terminal state, the optimal plan is found. The algorithm returns the best solution graph (SOLN-G) that encodes this plan. CPOAO* inherits the optimality property from AO* when an admissible heuristic is used under the assumptions listed. CPOAO* is guaranteed to terminate because a finite time limit for plan execution is given in the planning problem and each action takes a non-zero duration.

Heuristics

In our implementation of CPOAO* we used an optimistic heuristic function where we consider each unachieved reward in a state and find the best action that achieves it based on the costs and success probabilities of the actions. The reward-cost ratio of a reward r_i is defined as: $Ratio(r_i) = max_{a \in A}((Value(R) \times Prob(a))/Cost(a))$, where A is the set of actions, cost(a) is the resource usage of action a, and $prob(a)$ is the success probability of action a.

To maximize the total expected rewards, we pick the rewards with the higher ratios. If we select a reward which is associated with a probabilistic action, only a proportional amount of the reward is added to the total expected reward and we put the remaining reward which is proportional to the probability of the failure of this action back into the reward set. When we are done, the total expected reward is taken as the heuristic value for the state.

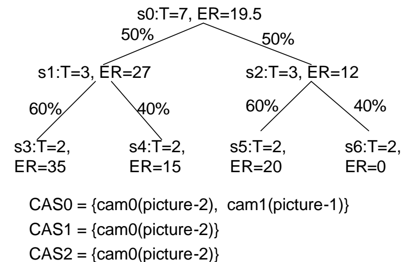

Figure 3: Example working graph.

Figure 3 shows a portion of a working graph corresponding to the above example. Each node represents a state which could be reached under this plan. For each node s_i, we show the time left (T) and the final expected reward (ER) at this state. The concurrent action sets (CAS) for each state are listed below the figure. For the CASs which have probabilistic outcomes, the probability of each outcome is marked on the link. The left branch corresponds to the successful outcome and the right branch is the failed one.

In the above example, the only resource is time. In case more resources are involved, we calculate the expected total rewards for each type of resource using the above routine and we take the smallest one as the heuristic value. The calculation of expected rewards with concurrent actions is similar. We omit the details due to space constraints.

We implemented a second heuristic to prune branches of concurrent action sets (CASs) for domains which have only time as a resource. The technique is based

on the fact that we do not need to have a branch for a concurrent action set if there is another concurrent action set which is always *better* than it. We say that the concurrent action set A is always better than the concurrent action set B if the expected total reward which could be collected by following the concurrent action set A is always greater than following the concurrent action set B. For example, when the actions do not consume any resources other than time and action preconditions are met, starting the actions earlier is always better than letting the operators be idle. We can abort any action at any time if it turns out that we should not wait for its completion. The following rules can be used to determine whether or not two concurrent action sets have the "better" relation. (1) If the shortest action a in CAS A does not consume any resource other than time or delete any propositions, then CAS A is always better than CAS $B = A$-{a}. (2) Let $A = B \cup \{b\}$. If b is not the shortest action in A and b does not consume any resource other than time, then A is always better than B. (3) Suppose CAS A and CAS B contain the same set of actions except action b. If action b is a member of both CASs but action b in A has a shorter remaining time than the action b in B, then CAS A is always better than CAS B.

The first rule states that if an action consumes only time, it does not harm to start it as early as possible. The second rule states that even if an action is deleting a proposition, it is still safe to start it as early as possible as long as it is not the shortest action in the action set. If it is not the shortest action, we still have a chance to abort it so that it will not delete any propositions. The third rule states that there is no need to abort the currently executing action and restart it immediately.

Empirical Evaluation

To evaluate our algorithm we conducted three sets of experiments on Mars rover problems. In the first set, we used CPOAO* with both of the heuristics and tested its limits. In the second set, we performed ablation studies to test the effects of the two heuristics. In the third set, we evaluated a technique for caching node values. For limit studies, we gradually increased the problem complexity by using 5, 10, 12, and 15 locations. For each number of locations, we created four different problems by varying the connectivity of the locations and the number of rewards. The problem names are coded as "m-n-k" where m is the number of locations, n is the number of paths and k is the number of the rewards in the problem. We tested CPOAO* with execution time limits of 20 and 40 given as part of the planning problem. A "-" in the table indicates that the corresponding problem was not solvable within 5 minutes. The results in Table 1 show that the current implementation of CPOAO* can return the optimal plan for up to 15 locations. As expected, the execution time and the number of nodes generated increase exponentially as the time limit increases due to the large branching factor b at each internal node.

Problem	TR	T = 20			T = 40		
		ER	NG	ET	ER	NG	ET
5-5-5	34	22.4	41	<1	23.84	524	<1
5-5-10	51	22.4	95	<1	29.6	7407	3
5-10-5	34	22.4	178	<1	26.08	11374	6
5-10-10	51	25.6	389	<1	32.96	87858	155
10-10-8	41	23.8	69	<1	25.89	1130	<1
10-10-21	84	25.3	265	<1	30.99	27538	19
10-17-8	41	22.4	122	<1	25.6	8179	2
10-17-21	84	23.9	332	<1	28.9	102484	190
12-12-12	52	23.8	124	<1	27.52	3625	1
12-12-23	97	25.3	301	<1	30.14	36344	32
12-21-12	52	22.4	196	<1	25.6	16724	8
12-21-23	97	23.9	396	<1	28.9	125959	272
15-16-14	58	23.8	184	<1	27.52	8664	3
15-16-31	116	25.3	450	<1	-	-	-
15-28-14	58	22.4	337	<1	27.4	51344	75
15-28-31	116	25.3	752	<1	-	-	-

Table 1: Performance of CPOAO*. TR: Sum of all rewards. T: Time limit. ER: Expected total reward of the optimal plan. NG: The number of nodes generated. ET: Execution Time (sec.).

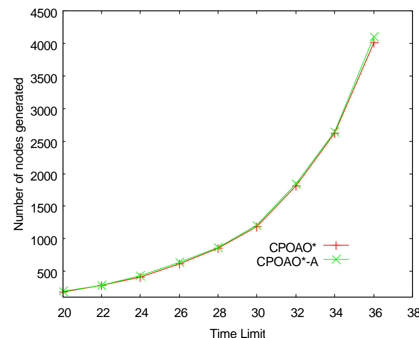

Figure 4: CPOAO* vs. CPOAO*-A for problem 15-16-14 (15 locations, 16 paths and 14 rewards)

To evaluate the effects of the heuristics we created two variants of CPOAO*. The first one named CPOAO*-A uses a constant heuristic function and the pruning technique. The second one named CPOAO*-B has the same heuristic function for estimating the expected total reward as CPOAO* does, but does not implement the rules of pruning the branches of CASs. When expanding a state, it applies all the possible CASs on this state. Figures 4 and 5 compare CPOAO* to CPOAO*-A and CPOAO*-B. The number of generated nodes is used as the metric of the efficiency of heuristics. The results show that our heuristic function for estimating the expected total reward did prune a number of states from the search graph. However, compared to the total number of generated states, the gain is small. This means that a more informative heuristic function of estimating the expected total reward of the states can improve the performance of the algorithm.

Because the shape of the search graph is much like a

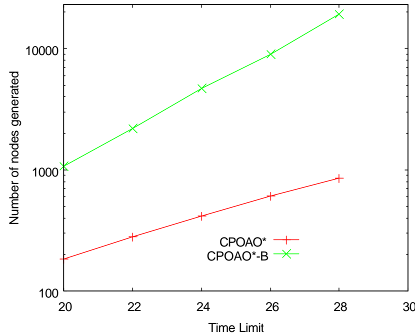

Figure 5: CPOAO* vs. CPOAO*-B for problem 15-16-14 (15 locations, 16 paths and 14 rewards)

pyramid with a large number of nodes residing at the lower levels, a heuristic function that is informative only at the last several steps before reaching the leaves can still effectively prune a large number of nodes. Therefore, we are working on using regression from the leaves to design an effective heuristic function. The results with CPOAO*-B show that the pruning rules for the action combinations did prune a large portion of the reachable states from the search graph. Further research is needed to minimize the set of applicable CASs at each state.

Our final set of experiments is based on the observation that there are many similar states in the search graph. In particular, there are states in which the set of rewards collected and executing actions are the same but the remaining times are different. In such a case, the total expected reward for the state with the greater remaining time serves as an upper bound for the state with the lower remaining time. To exploit this fact, we cache the expected rewards of the states that are completely expanded. When a new state is generated, we try to obtain its expected rewards from the cache, if there are no cached states similar to it, we use the heuristic function explained previously.

Our experimental results in Table 2 show that for some cases, the planner with the cache generates 45% less nodes than the planner without the cache. The gains are negligible when the allowable execution time is short. As the allowable execution time gets longer, a bigger portion of nodes are pruned. This can be explained by the fact that as the execution time gets longer, more previously failed actions can be retried. The nodes generated by these retried actions tend to be similar to the nodes in the cache.

Conclusion

Our work is along the lines of improving search techniques which deal with environment uncertainty for plan execution. We have presented the design and evaluation of CPOAO*, an algorithm that provides a modular, heuristic-based framework which deals with realistic planning problem features such as uncertainty,

Problem	Without Cache		With cache	
	NG	ET	NG	ET
5-5-10 (T=20)	95	<1	93	<1
5-5-10 (T=40)	7404	3	4208	1
5-10-10 (T=20)	389	<1	320	<1
5-10-10 (T=40)	87858	155	59578	83
10-10-21 (T=20)	265	<1	252	<1
10-10-21 (T=40)	27538	19	16987	7
10-17-21 (T=20)	332	<1	316	<1
10-17-21 (T=40)	102482	190	53814	58
12-12-23 (T=20)	301	<1	279	<1
12-12-23 (T=40)	36344	32	21911	14
12-21-23 (T=20)	396	<1	348	<1
12-21-23 (T=40)	125959	272	72324	95
15-16-14 (T=20)	184	<1	176	<1
15-16-14 (T=40)	8664	3	6064	2
15-28-14 (T=20)	337	<1	343	<1
15-28-14 (T=40)	51344	75	24982	15

Table 2: Performance of CPOAO* with cache. T: Time limit. NG: The number of nodes generated. ET: Execution Time (sec.).

over-subscription, and concurrency. An important contribution of our algorithm is its ability to use concurrent actions that achieve the same goal. Our experimental results are promising, we are planning to compare our system to Paragraph (Little & Thiebaux 2006), and CPTP (Mausam & Weld 2005).

Our current implementation includes three heuristics designed for concurrent domains with rewards and gives insight into possible improvements in dealing with complex domains. The current heuristic for estimating node values does not take into account how to achieve the preconditions of actions. Our future work involves using reachability based heuristics and pruning techniques and extending our framework to (1) consider the resource cost of reaching the goals, (2) remove the assumption that terminating an action before it finishes does not have any effects on the world, and (3) develop a theory of when it would be useful to abort already started actions. In regards to item 2, we need to extend our model to consider the possibly unknown state of the system after failures. In our current implementation, we only modeled the failure to achieve a goal and assumed that the cameras would remain in a ready state even if a picture cannot be taken. In regards to item 3, currently we only consider aborting actions when the goals they are serving are already achieved. In addition, resource consumption can be minimized by aborting actions which no longer serve a useful purpose due to failure of parallel actions.

Acknowledgements We thank the anonymous reviewers for their detailed comments.

References

Benton, J.; Do, M. B.; and Kambhampati, S. 2005. Oversubscription planning with numeric goals. *Proc. IJCAI-05.*

Bonet, B., and Geffner, H. 2005. mGPT: A probabilistic

planner based on heuristic search. *Journal of Artificial Intelligence Research 24* 24:933–944.

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

Bresina, J. L.; Dearden, R.; Meuleau, N.; Ramakrishnan, S.; Smith, D. E.; and Washington, R. 2002. Planning under continuous time and resource uncertainty: A challenge for AI. In *Proc. UAI-02*, 77–84.

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

Haslum, P., and Geffner, H. 2001. Heuristic planning with time and resources.

Kushmerick, N.; Hanks, S.; and Weld, D. 1995. An algorithm for probabilistic planning. *Artificial Intelligence* 76:239–86.

Little, I., and Thiebaux, S. 2006. Concurrent probabilistic planning in the graphplan framework. In *Proc. ICAPS-06*, 263–272.

Mausam, and Weld, D. 2004. Solving concurrent markov decision processes. In *Proc. AAAI-04*, 716–722.

Mausam, and Weld, D. 2005. Concurrent probabilistic temporal planning. *Proc. ICAPS-05*.

Mausam; Benazera, E.; Brafman, R.; Meuleau, N.; and Hansen, E. A. 2005. Planning with continuous resources in stochastic domains. *Proc. IJCAI-05*.

Nilsson, N. J. 1980. Principles of artificial intelligence. *Tioga Publishing*.

Pfeffer, A. 2005. Functional specification of probabilistic process models. In *Proc. AAAI-05*, 663–669.

Smith, D. E. 2004. Choosing objectives in over-subscription planning. In *Proc. ICAPS-04*.

Younes, H. L., and Simmons, R. G. 2004. Policy generation for continuous-time stochastic domains with concurrency. In *Proc. ICAPS-04*, 325–333.

Younes, H. L.; Littman, M. L.; Weissman, D.; and Asmuth, J. 2005. The first probabilistic track of the international planning competition. *Journal of Artificial Intelligence Research* 24:851–887.