2015

GENERATING PLANS IN CONCURRENT, PROBABILISTIC, OVER-SUBSCRIBED DOMAINS

Li Li
Michigan Technological University

GENERATING PLANS IN CONCURRENT, PROBABILISTIC, OVER-SUBSCRIBED DOMAINS

By

Li Li

A DISSERTATION

Submitted in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

In Computer Science

MICHIGAN TECHNOLOGICAL UNIVERSITY

2015

This dissertation has been approved in partial fulfillment of the requirements for the Degree of DOCTOR OF PHILOSOPHY in Computer Science.

Department of Computer Science

Dissertation Advisor: *Dr. Nilufer Onder*

Committee Member: *Dr. Laura Brown*

Committee Member: *Dr. Charles Wallace*

Committee Member: *Dr. Renfang Jiang*

Department Chair: *Dr. Min Song*

Contents

[1] ©2008 AAAI. Portions reprinted with permission, from **Li Li**, Nilufer Onder, "*Generating Plans in Concurrent, Probabilistic, Over-Subscribed Domains*", in Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence (AAAI 2008), pp. 957–962.

[2] ©2008 AAAI. Portions reprinted with permission, from **Li Li**, Nilufer Onder, "*Generating Plans in Concurrent, Probabilistic, Over-Subscribed Domains*", in Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence (AAAI 2008), pp. 957–962.

List of Figures

List of Tables

Preface

This dissertation contains the research work done for my PhD degree in Computer Science at Michigan Technological University. The work presented here has not been submitted for any other degree or diploma. The main contributions of this work are the design and implementation of an automated planner that can find plans with early-finish and all-finish parallelism, and the development and evaluation of several domain independent heuristics that can work with temporal and resource constraints.

Chapters 3 and 4 contain part of material previously published in the proceedings of the AAAI conference in 2008. My contributions to this publication were to design and implement the CPOAO* algorithm, develop and evaluate heuristics functions, and compare to existing planners. The work has been done under the guidance of my advisor, Dr. Nilufer Onder, who is the second author in the paper. In this dissertation, we used the base CPOAO algorithm and time based heuristic functions from the AAAI 2008 publication. We extended the algorithm to handle resource constraints in addition to time constraints. We designed novel heuristic functions which utilize both time and resource constraints.

The previous work being included in this dissertation is: **Li Li**, Nilufer Onder. "*Generating Plans in Concurrent, Probabilistic, Over-Subscribed Domains*". In Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence (AAAI-08), pp. 957–962.

Acknowledgments

I would like to thank my advisor Dr. Nilufer Onder with deepest appreciation for her excellent guidance, kindness and patience during my graduate study at Michigan Technological University. There were difficult times during my research and Dr. Nilufer was always there to give me a lift, from research ideas to advice on how to balance work and life. It is a long journey and I honestly believe that I would not be able to reach the finish line without her encouragement and understanding. I also want to thank my committee members, Dr. Laura Brown, Dr. Charles Wallace, and Dr. Renfang Jiang. I appreciate their cheerful willingness to be on my committee, spending time to review my dissertation, and providing very helpful comments and suggestions.

I want to thank my wife, Yang Xue, for taking care of the family all these years so that I can focus on my studies. I want to thank my parents, Yangzong Li and Anliang Li for inspiring me to pursue science and knowledge since I was very young. I fondly remember all their support and warm encouragement.

Finally, I want to thank Dr. Mausam from the University of Washington and Dr. Eric Beaudry from the University of Quebec in Montreal for providing me their software and the instructions so that I can complete the experiments and comparisons with their planners. Their prompt reply to my questions and e-mail messages were of great help to my research.

Abstract

Planning in realistic domains typically involves reasoning under uncertainty, operating under time and resource constraints, and finding the optimal subset of goals to work on. Creating optimal plans that consider all of these features is a computationally complex, challenging problem. This dissertation develops an AO* search based planner named CPOAO* (Concurrent, Probabilistic, Over-subscription AO*) which incorporates durative actions, time and resource constraints, concurrent execution, over-subscribed goals, and probabilistic actions. To handle concurrent actions, action combinations rather than individual actions are taken as plan steps. Plan optimization is explored by adding two novel aspects to plans. First, parallel steps that serve the same goal are used to increase the plan's probability of success. Traditionally, only parallel steps that serve different goals are used to reduce plan execution time. Second, actions that are executing but are no longer useful can be terminated to save resources and time. Conventional planners assume that all actions that were started will be carried out to completion. To reduce the size of the search space, several domain independent heuristic functions and pruning techniques were developed. The key ideas are to exploit dominance relations for candidate action sets and to develop relaxed planning graphs to estimate the expected rewards of states. This thesis contributes (1) an AO* based planner to generate parallel plans, (2) domain independent heuristics to increase planner efficiency, and (3) the ability to execute redundant actions and to terminate useless actions to increase plan efficiency.

Chapter 1

Introduction

Automated planning is an important field of artificial intelligence. It is an essential component of developing intelligent systems which can act autonomously to complete complex tasks. *Planning* is the process of choosing and organizing actions that will achieve a set of prestated objectives [1]. Similar to other areas of artificial intelligence and computer science, research in planning focuses on developing expressive representations and efficient algorithms. Planners operate using a structured problem representation which defines an *initial state*, a set of *goal states*, and a set of state modifiers called *actions.* When applied in a particular state, an action can cause a change to a different state. The conditions for the applicability of an action are called *preconditions*, and the resulting changes are called *effects*. The goal of a planner is to find a set of actions that will be executed to change the initial state to a goal state through the effect-precondition linkages [2, 3].

Intelligent planning is useful for many real life applications ranging from supply chain management to autonomous spacecraft [4, 5]. For example, the modern-day enterprise commonly has a very sophisticated supply chain which includes geographically distributed raw material suppliers, external manufacturers, warehouses, plants, and distribution centers [6]. Adding to the complexity, many factors such as supplier capacity, resource availability, transportation time, alternate routing, component substitution, manufacturing cost also need to be considered to create an optimal plan [7]. The traditional material requirement planning uses a simple waterfall model [8] flowing from the top level assembly to the lowest level raw materials and generates a baseline requirement plan. These requirements have to be further adjusted manually before being sent to the purchase department for acquisition or to the shop floor for execution. This approach becomes unfeasible when the supply chain is complex and has many variables. In the more and more intense global competition, having an intelligent supply chain planning system is the key to reduce costs and increase productivity and on time delivery. The planning software

currently available usually take two approaches to tackle the problem of constrained supply chain planning and optimization. The first approach is to convert the supply chain planning problem into a mathematical optimization problem and use linear programming or integer programming to solve it. In this approach, the common key performance indicators such as inventory turn, total profit, and on-time delivery can be selected as the objectives of the optimization. However, there is usually no opportunity for the user to enter priorities to specific orders. The second approach is a heuristic guided search where a domain dependent, possibly suboptimal heuristic which simulates the strategy of a human planner is used.

Both approaches of supply chain management assume a deterministic model, not considering the possibility of production failure, quality issues, transportation delays, or machine breakdowns. As such, there is no way to generate a contingency plan and the entire plan needs to be re-generated for each scenario. In the real world of supply chain, resources and time are always limited. Therefore, it is unavoidable for some tasks to fail. The common practice is to assign priorities to orders and have an alternate manufacturing process in case the primary process cannot be carried through. These shortcomings constitute a motivation to develop a planner which considers uncertainty, respects time and resource constraints, and give flexibility to assign priorities to goal conditions. Furthermore, it is desirable to develop domain independent admissible heuristic functions which can be applied in a range of industries.

Automated Planning is also an important component in autonomous vehicles such as the Mars rover. Challenges similar to the world of supply chain planning are faced by Mars rovers: Actions may have uncertain outcomes, placement of an instrument can fail, driving from one waypoint to another may take longer time than expected, and actions may consume an uncertain amount of power. Furthermore, the tasks to be accomplished by the Mars rover may have different level of priorities and deadlines. For example, Mars rover must get back to the solar recharge location before using up the power. Both Mars rover Opportunity and Curiosity have an onboard system called OASIS (Onboard Autonomous Science Investigation System) [9, 10, 11] which plans rovers' activities on Mars. OASIS generates a sequenced plan and monitors its execution. When there is a change to the environment or Mars rover's state deviates from the plan, OASIS regenerates the plan. An alternative approach is to generate a more robust contingent plan when the Mars rover is still on Earth. We believe having a more robust contingent plan as the baseline plan can improve the overall plan quality and turnaround time. The replan process can be invoked when there is a situation which is not covered by the preloaded contingent plan.

These requirements of real-world problems give the motivation to develop a planner which considers uncertainty, models parallel execution, respects time and resource constraints, and has flexibility to assign priorities to goal conditions. In addition, the planner needs to be guided with domain independent heuristic functions that guarantee optimal

plans. This dissertation develops a model and planner named CPOAO* (Concurrent, Probabilistic, Over-subscription AO*) which incorporates probabilistic actions, concurrent action execution, durative actions with time and resource constraints, and over-subscribed goals.

To give an example of the reasoning in our work, consider a simplified Mars rover domain [12] where two pictures must be taken within 5 minutes, and actions have fixed known durations. The rover has two cameras: cam0 succeeds with probability 0.6, and takes 5 minutes; and cam1 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 cam0 for one picture and cam1 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 cam0 and cam1 in parallel to achieve the larger reward. In this case, the success of the earliest finishing action is sufficient. We call this case *early-finish parallelism*. Cam1 finishes earlier than cam0 and the expected reward for using cam1 is $100 \times 50\% = 50$. If cam1 fails, the expected remaining unachieved reward will be $100 \times (1 - 50\%) = 50$. Then the expected reward for action cam0 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, $100 \times 60\% + 10 \times 50\% = 65$.

When both cameras are used for the same picture, if cam1 succeeds in achieving the target reward, we can abort cam0 immediately unless it serves other goals. Such termination avoids unnecessary expenditure of resources. If plan steps are marked with termination conditions during plan generation, they can be monitored during plan execution for these conditions.

In domains where explicit rewards are assigned to goals, there is advantage in having parallel actions that serve the same goal, i.e., early finish parallelism. In our approach, we provide the means for using such parallel actions to maximize expected rewards. In multi-agent domains or in parallel single-agent domains, there is advantage in terminating actions as soon as the expected result is obtained, or as soon as it becomes impossible to obtain the expected results. Our algorithm is capable of marking the actions to be terminated so that resources can be saved to achieve other goals.

The main contributions of this thesis are threefold. First, we designed and implemented an AO* search based planner that can find plans with early-finish and all-finish parallelism. Second, we developed and evaluated several domain independent heuristics that can work with temporal and resource constraints. Third, we explore the notion of interruptible actions. Our research improves artificial intelligence planning research as evidenced by the empirical evaluation we conducted.

The organization of this dissertation is as follows. In Chapter 2, previous work related to this research is discussed. Techniques and algorithms that this research is built upon are reviewed in the same chapter. Chapter 3 gives an overview of the CPOAO* algorithm. The heuristic functions designed and implemented for CPOAO* are described in Chapter 4. In Chapter 5, we describe the empirical evaluation of our CPOAO* planner and present a comparison to state of the art planners. The summary of our research, conclusion and future work are given in the concluding chapter.

Chapter 2

Background

Plan generation is a computationally complex problem because there are typically an overwhelming number of actions and states resulting in a large number of options to consider while constructing the optimal plan. The complexity increases exponentially when the real world features such as probabilistic actions, parallel actions, temporal reasoning, and over-subscribed goals are modeled. We briefly describe the planners that model each feature below.

- Probabilistic actions: Uncertainty is represented using actions that have multiple possible effects. Each effect has an associated probability which reflects how likely it is for this effect to happen when the action is executed [13, 14, 15, 16, 17, 18].
- Parallel actions: Actions can be executed concurrently rather than in a serialized fashion. The planner generates sets of actions instead of individual actions to shorten the makespan of the plan [19, 20, 21, 22, 23].
- Temporal reasoning: Action durations are represented as a numeric quantity, i.e., a plan metric [24]. Deterministic planners that use temporal reasoning include TP4 [25], Sapa [26], MIPS [27], TLPLAN [28], and HPlan-P [29, 30, 31].
- Over-subscribed goals: The planner must select a subset of the goals to plan for when resource limitations do not allow all the goals to be achieved. Each goal is represented with an associated numeric reward. The solution is a plan that collects the maximum rewards based on the resources available [32, 26, 33, 34].

CPOAO*'s plan generation algorithm is based on the AO* heuristic search framework. In the following section, we review the basics of the A* and AO* algorithms followed by two AO* based planning algorithms, LAO* and HAO*.

2.1 AO*: Heuristic Search in Hypergraphs

AO* is an extension of the A* algorithm [35, 36, 37, 38]. A* is a best-first search algorithm that was originally developed to solve the shortest path problem. In the shortest path problem, the objective is to find the shortest path from the root node to a goal node in a graph (there can be multiple goal nodes in the graph). For each node x visited, the algorithm calculates the cost function $f(x)$ using the equation $f(x) = g(x) + h(x)$ where $g(x)$ is the actual distance from the root node to x and $h(x)$ is a heuristic estimate of the minimum distance from x to any goal node. The algorithm maintains a queue of nodes to be expanded and sorts the queue by the value of function f in ascending order. At each step, the algorithm takes the first node n from the queue and calculates the value of function f for all of n's child nodes. Then node n is removed from the queue and its children are added into the queue. After that, the queue is sorted again and the expansion repeats until the first node in the queue is a goal node. At this moment, a path is found from the root node to a goal node, and the search terminates.

A* searches for the best solution to a problem. The definition of "best" is problem-specific. It can mean either the minimum value, as in the case of shortest path problem, or the maximum value, as in the case of reward based goal metrics. A heuristic function is said to be *admissible* if the heuristic value returned by the heuristic function never misleads the search away from the potentially promising branches in the search space [39]. In the case of finding the minimum value, an admissible heuristic function should never overestimate. In the case of finding the maximum value, it should never underestimate. If the heuristic function $h(x)$ is admissible, then A* guarantees that the path found is the shortest path.

AO* is also a heuristic guided best-first search algorithm. However, it searches in an *AND/OR graph*. An AND/OR graph, also called a *hypergraph*, has hyperarcs connecting one node to multiple other nodes [40]. AND/OR graphs were first introduced in the area of problem solving to represent the structure of problem reduction. In this setting, the root node represents the original problem. Each hyperarc represents one way of dividing the problem into subproblems. The children nodes under the same hyperarc represent the sub-problems which need to be solved all together to complete the solution. The sub-problems can be further divided until the tasks are indivisible. There is a fixed cost for completing every task. AO* searches in this AND/OR graph and it can find the optimal way to divide the original problem such that the total additive cost of completing all the required indivisible tasks is minimized.

In Algorithm 1, we show the steps of finding the optimal solution of problem reduction using AO* search. Initially both working graph G and the solution graph S contain only the root node. Then AO* repeats the following two steps. In the first step, all divisible

leaf nodes which are part of the solution graph are expanded in the working graph. For the new nodes which are not indivisible tasks, the cost values are estimated using a heuristic function. For nodes representing indivisible tasks, the cost values are directly calculated. Afterwards, the costs from the new nodes are propagated backwards to their ancestor nodes. In the second step, the solution graph is re-generated by selecting the best hyperarcs. As the working graph expands, the values of the expanded nodes are updated iteratively. This process repeats until all the leaf nodes in the solution graph are indivisible tasks.

Algorithm 1 AO* Algorithm

Initialize the working graph G to the root node.
Initialize the solution graph S to the root node.
while Solution graph S contains divisible leaf nodes **do**
1. In working graph G, expand all non-divisible nodes which are included in the solution graph S. Back propagate to update the cost values of all the ancestor nodes.
2. Re-construct the solution graph S by selecting the best hyperarcs in the working graph G from the root node.
end while
Return solution graph S.

In Figures 2.1 through 2.4 we illustrate an example. In this example, the round nodes represent sub-problems. The square nodes represent indivisible tasks. The numbers represent the costs. In Figure 2.1, the root node is expanded. The 3 hyperarcs represent the 3 ways to divide the original problem. The number on the hyperarc is the sum of the costs of all its children nodes. Because the middle hyperarc has the lowest total cost, at first it is included in the solution graph. In Figure 2.2, the nodes under the middle hyperarc are expanded. After the expansion, the cost values are updated using backward propagation. The total cost on the middle hyperarc is updated from 13 to 18 ($8 + 10 = 18$). Now the best hyperarc is the left hyperarc and it is included in the solution graph. In Figure 2.3, the nodes under the left hyperarc are expanded. After value propagation, the total cost of left hyperarc is changed from 14 to 16. However, it is still the best hyperarc. Therefore the solution graph is extended to include the new nodes under the left hyperarc. Figure 2.4 shows the last expansion. After this expansion, all the sub-problem nodes in the solution graph are expanded into indivisible task nodes and the final solution graph is found.

The calculation of cost values is done differently in AO* and A*. In A*, for a new leaf node, the cost is calculated as the cost incurred from the root to this new node plus the estimated cost from this node to a goal node (the second part is calculated by a heuristic function). This can be done because the solution is a path. In AO*, because the solution is a graph, the cost value associated with each node is the cost of solving the sub-problems represented by this node. Hence, a backward propagation is required to incorporate the cost from the new nodes into the cost of the ancestor nodes. The main idea of AO* is to find the optimal solution graph without completely expanding the working graph by pruning some

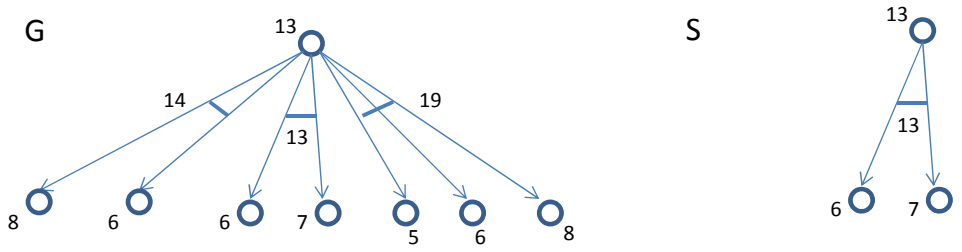


Figure 2.1: First expansion of the AND/OR graph.

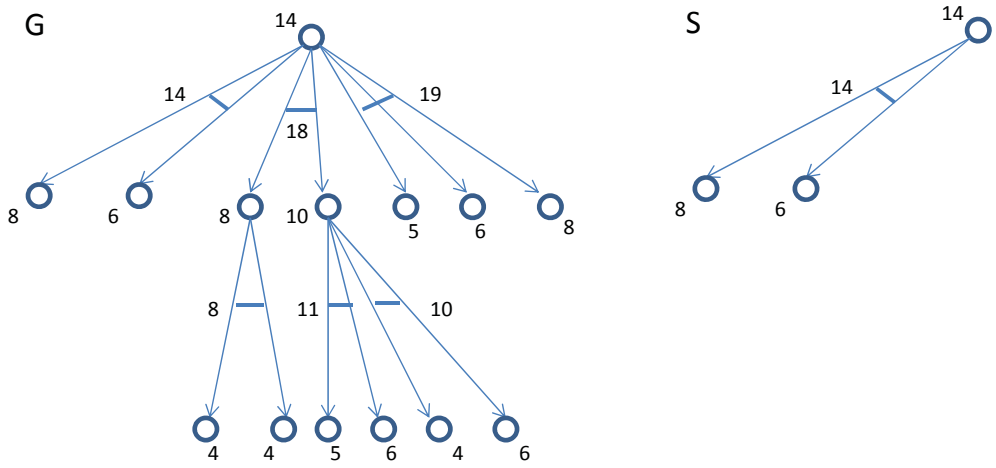


Figure 2.2: Second expansion of the AND/OR graph.

branches using heuristic values. Similar to A*, if the heuristic function is admissible, the solution graph found is guaranteed to be optimal.

Hyperarcs can be used in many different ways. In the area of planning, hyperarcs are very convenient to represent the probabilistic actions which have multiple resulting children states. Therefore, when designing a probabilistic planner, an AND/OR graph can be used as the search graph and AO* can be used as the search algorithm. Each node in the search graph represents a state and the hyperarcs represent the probabilistic actions. When a probabilistic action is applied at state s, it can have multiple outcome states. All of the children states are connected to state s by a hyperarc and each branch has an associated probability. In the example shown in Figure 2.5, there are two probabilistic actions applied in state s_0. The left one has 3 branches, connecting to 3 children states. The right one has 2 children states. For each probabilistic action, the sum of the probabilities of the outcomes

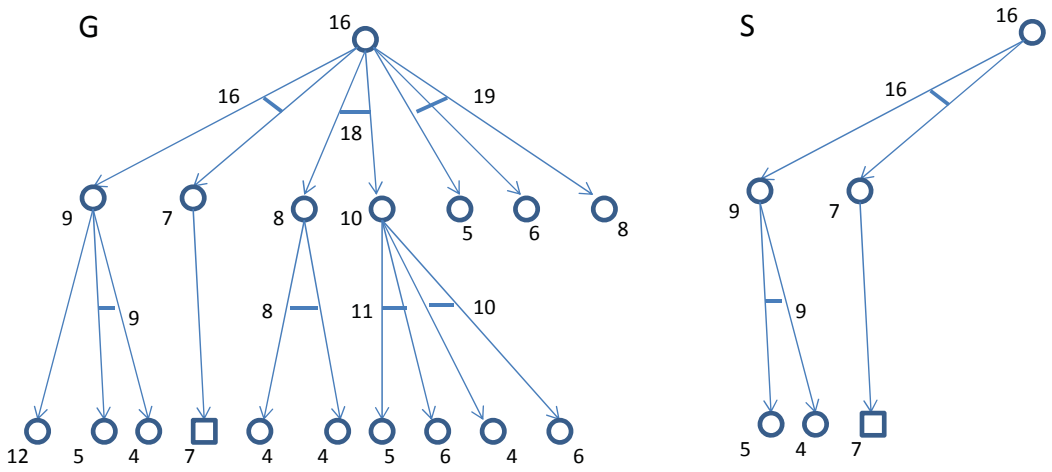


Figure 2.3: Third expansion of the AND/OR graph.

equals to 1.

When using AO* to solve planning problems, each node also has a numeric value associated to it. This numeric value can represent either the cost to reach the goal or the total reward that can be collected starting from this state. For the leaf nodes in the hypergraph, this value can be returned by a heuristic function. For internal nodes, the value is calculated from the children state values using the backward update formula shown below:

$$V(s) = \text{best}_{a \in A}(U(s,a) + \sum_{s' \in T(s,a)} P(s,a,s')V(s'))$$

where, $V(s)$ is the value of state s, A is the set of all probabilistic actions, $U(s,a)$ is the single step reward or cost function of applying action a in state s, $T(s,a)$ is the set of all possible resulting children states of applying a in state s, and $P(s,a,s')$ is the probability of reaching state s' from state s when applying action a.

In the next two sections, we describe HAO* and LAO* planning algorithms, which implement probabilistic over-subscribed planning and plans with loops, respectively.

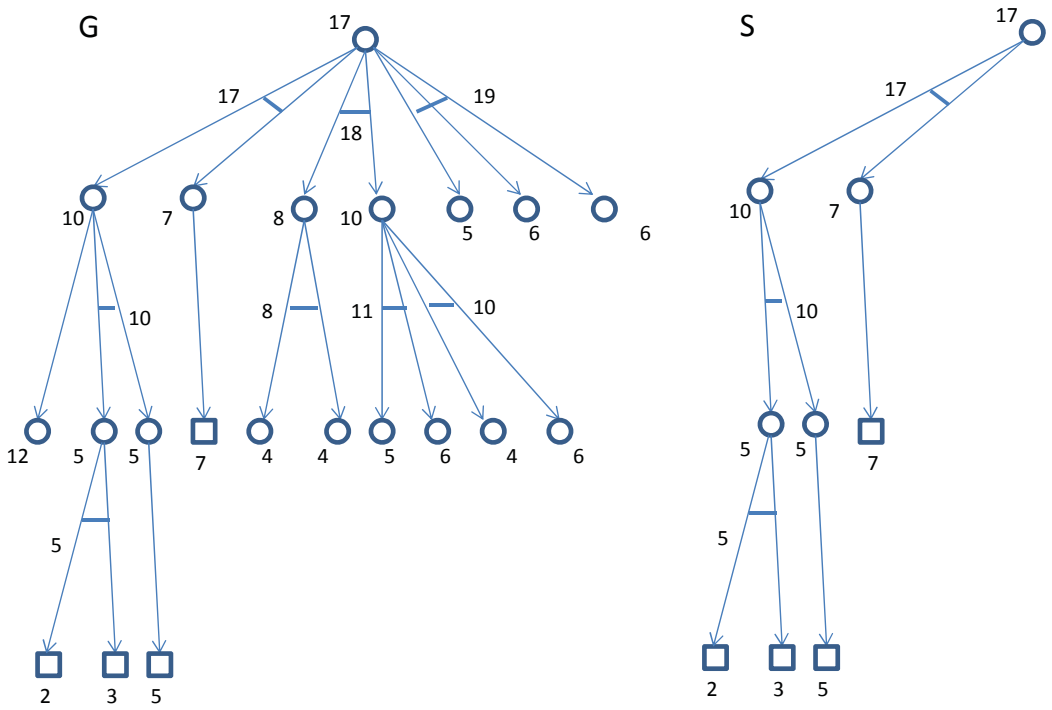


Figure 2.4: Fourth expansion of the AND/OR graph.

2.1.1 HAO*: Planning with Continuous Resources

HAO* (Hybrid AO*) is an extension of the AO* search algorithm for probabilistic over-subscription planning problems that involve continuous resources [41]. In an over-subscription problem, the agent tries to achieve as many goals as possible given the constraints on the resources. Each goal has an associated numeric reward indicating its importance. Due to the constraints of the resources, the agent can only achieve a subset of the goals. Selecting the optimal subset is an NP-hard problem. In the HAO* framework, each action consumes some resources and the total available resources are limited. Therefore, the search graph is bounded by the resource constraints. The expansion stops when the resources are completely consumed or all the rewards have been collected. HAO* finds the optimal plan which maximizes the total expected rewards.

The HAO* framework assumes that the resources can not be replenished and each action consumes at least one kind of resource. If this assumption holds, the search graph does not have any cycles because subsequent states always have fewer resources. Therefore, we can use AO* to solve over-subscription probabilistic planning problems. Furthermore, the

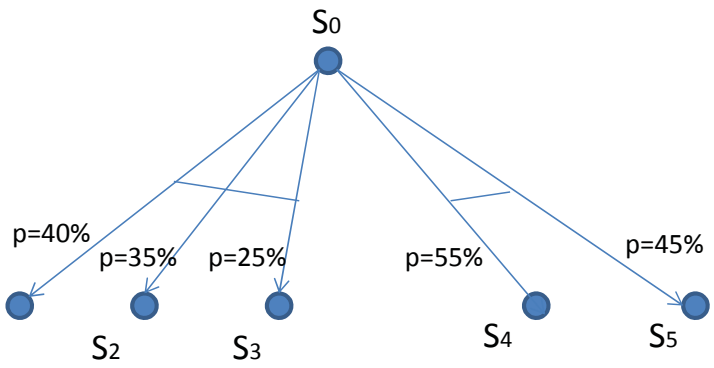


Figure 2.5: Example of AO* hyperarcs.

heuristic functions that make use of the constraints on the resources can be designed to greatly reduce the search space.

The main contribution of HAO* is that it gives a method to integrate the continuous state variables representing continuous resources into the framework of AO* algorithm. In HAO*, a planning state consists of a discrete component and a continuous component. The discrete component includes all the discrete variables of the state. The continuous component describes the continuous resources of the state. The domain of each particular resource is an interval of the real number line. The exact form of the continuous component is a vector of continuous variables defined over a hypercube which is bounded by the domains of the resources. Because the continuous component is uncountable, the states in HAO* can not be represented as graph nodes directly. Instead, the nodes in the HAO* search graph are aggregates of the states which have the same discrete component. Therefore, each node represents a region of the state space. A node contains a probability distribution on the values of the continuous component. It specifies the reachability of a particular value of the continuous component from the root node. Each node also includes a value function to represent the expected future reward of the states within the aggregate. For a new node which has not been expanded, this value function is a heuristic estimate for the states within the node. For the expanded nodes, the value function can be computed precisely by a dynamic programming algorithm that solves the Bellman optimality equation with the continuous integral.

HAO* search starts with the root node which may contain only one state. The probability distribution of the continuous component can also be set to represent a group of possible initial states. When a node is expanded, based on the probability distribution of the continuous resources, only the actions which are possible for this node are considered. The unreachable states are ignored. HAO* expands the search graph and updates the value function and probability distribution of the continuous component along the way. Because

a node is an aggregate of the states, it can have different best actions for different subsets of the states within it. Due to the same reason, sometimes an already expanded node may need to be re-expanded after a new path which changes the probability distribution of the continuous component is found.

HAO* is a novel approach to solving planning problems that involve continuous resources. It puts all states with the same discrete component into the same node and use the probability distribution to track their reachability. It exploits the fact that although the states in one node are uncountable, but they can be partitioned into different regions and the states within the same region have the similar behaviors.

In HAO* or classic AO*, the search graph is an acyclic AND/OR graph which does not have loops. However, it is common that infinite horizon planning problems contain loops in the search graph. For those problems, backward propagation of values does not work. LAO*, which is described in the next section was designed to address this issue.

2.1.2 LAO*:A Heuristic Search Algorithm that Finds Solutions with Loops

In many cases, it is hard to guarantee that there are no cycles in the search graph for a probabilistic planning problem. Sometimes there are loops even in the optimal solution plans due to the uncertainty of the probabilistic actions. In a deterministic world, it does not make sense to go back to a state that was already visited and thus form a loop. In a probabilistic world, however, this may be unavoidable because an action may have many possible outcomes. For example, an action may fail. If it fails, the state does not change forming a cycle that starts and ends in the same state. The classical AO* algorithm can not search in a cyclic graph. If there exist cycles in the search path, its backwards propagation of the values results in endless loops. To fix this problem, the LAO* algorithm was developed to handle the updating of the values in a cyclic search graph [42].

Instead of doing backward propagation, LAO* uses dynamic programming methods such as value iteration or policy iteration [43, 44, 45] to update the values for all the nodes directly or indirectly linked to the newly expanded nodes. In value iteration, state values are updated iteratively by the greedy selection of the best action given the values of other states in the previous iteration. This update step is written as follows:

$$V_{i+1}(s) = \text{best}_{a \in A}(\sum_{s'} P_a(s,s')(U_a(s,s') + \gamma V_i(s')))$$

where V_i is the value function in iteration i, A is the action set, s' is a child state when applying action a in state s, P is the probability or reaching state s', U is the single step utility function, and γ is the discount factor. To determine when the iteration can stop, an error bound is calculated as follows. For all states, if the value difference between two iterations is less than ε, then the maximum difference between the values in the current iteration and their optimal values are bounded by $2\varepsilon\gamma/(1-\gamma)$. In policy iteration, the iterative updates are made directly to improve the policy, which is a mapping between the states and their best actions. Each iteration consists of 2 steps. In the first step, for the current policy π, the following linear equations are solved to find all the optimal values for all states under the current policy.

$$V_\pi(s) = \sum_{s'} P(s,s')(U(s,s') + \gamma V_\pi(s')))$$

In the second step, the policy is updated based on the updated state values. The iterations continue until there is no change on the policy between two iterations. The final policy found is guaranteed to be optimal. Algorithm 2 presents LAO*.

LAO* is a generalization of AO* in that it relaxes the condition that the search graph of the problem must be acyclic. It maintains a working graph and a solution graph. The working graph contains all the states generated. The solution graph has the initial state as the root node and consists of the best action arcs and their children states. It is re-constructed every time the values of the new states and their ancestor states are updated by dynamic programming. When the algorithm stops, the solution graph represents the solution of the planning problem.

In the remaining sections, we first explain $Sapa^{ps}$, an over-subscription deterministic planner [26]. Afterwards, we explain three recent probabilistic planners that can generate plans with parallel actions: ActuPlan deals with uncertainty in action duration [46, 47], and coMDP and CPTP deal with uncertainty in action outcomes [20].

Algorithm 2 LAO* Algorithm

1: The working graph G initially consists of the start state s.
2: **while** the partial solution contains non-goal terminal states: **do**
3: Expand the working graph: Expand a non-goal terminal state in the partial solution using its current best action so that the best action and all the new successor states are added into the working graph.
4: Create a set Z that contains the expanded state and all of its ancestors in the working graph along with the best action arcs (only include ancestor states from which the expanded state can be reached by following the current best actions).
5: Perform dynamic programming (policy iteration or value iteration) on the states in set Z to update the state values and then determine the best action of each state.
6: Re-construct the partial solution.
7: **end while**
8: Convergence test: If policy iteration was used in step 5, go to step 9. Otherwise, perform value iteration on the states in the partial solution. Continue until one of the following two conditions is met.
(i) If the maximum error falls below the error bound, go to step 9.
(ii) If the solution graph changes so that it has an unexpanded non-goal terminal node, go to step 2.
9: Return an optimal(or ε-optimal) solution.

2.2 $Sapa^{ps}$: Solving Deterministic Over-Subscription Planning Problems

$Sapa^{ps}$ is an A* heuristic search planner designed to solve deterministic over-subscription planning problems [26]. The target problem of $Sapa^{ps}$ is slightly different from the oversubscribed problems discussed in previous section. In a $Sapa^{ps}$ problem, the resource consumption is represented by the costs of the actions. The cost of each action is a single number. Therefore, $Sapa^{ps}$ can represent at most one type of resource. $Sapa^{Mps}$ is an extension of $Sapa^{ps}$ to handle numerical goals and soft/hard goal constraints [34]. The solution to a $Sapa^{ps}$ or $Sapa^{Mps}$ planning problem is a plan that has a good trade-off between the reward achieved and the total cost of the actions in the plan. The term "good" rather than "best" is used here because $Sapa^{ps}$ is not an optimal planner due to its inadmissible heuristic function.

Despite being inadmissible, $Sapa^{ps}$'s heuristic function is very informative. To find a heuristic value of a state, $Sapa^{ps}$ generates a *relaxed plan graph* with this state as the root. The plan graph is relaxed in the sense that the delete effects of actions are ignored [48, 49]. The plan graph is expanded until a fixed point is reached. Then a relaxed plan is extracted

from this graph by regression from the goals to the propositions in the initial state. In this extraction, the most cost-effective action is chosen for each goal and the subgoals derived from the goals. The cost of an action includes both the cost of executing this action and the cost of achieving its preconditions. The cost of the preconditions is computed additively. However, an action that supports precondition A may also support precondition B. Therefore, computing the total cost additively may over-estimate the actual cost. This is the reason that the heuristic function is not admissible. After the relaxed plan is generated, it is analyzed to find out which actions support which goals. The result is a mapping from the subsets of actions to the subsets of goals where each subset of actions only supports the goals in the corresponding goal subset. Based on this mapping, $Sapa^{ps}$ forms a subset of goals that is most cost-effective. The net benefit of this subset of goals is taken as the heuristic value of the start state.

In summary, $Sapa^{ps}$ employs an inadmissible heuristic function to guide its A* search algorithm in finding the plans. It trades optimality for efficiency. It can efficiently generate relatively good plans for over-subscription planning problems.

In our work, we also solve oversubscription planning problems. The heuristic functions we developed use a relaxed planning graph to estimate the rewards for the new states. The difference is that our planning framework is capable of handling concurrent probabilistic actions and our heuristic functions are admissible.

2.3 ActuPlan: Planning with Continuous Probabilistic Action Durations

ActuPlan is a recent planner which explores the planning domains with both action concurrency and uncertainty on action duration. In its current version, it assumes the action effects are deterministic and the uncertainty is only with action durations [46, 47]. ActuPlan uses a continuous time model where time is represented using random variables instead of a series of discrete time intervals. For each action, two random variables are created to represent the start event and the end event. For example, suppose that there are two actions a_1 and a_2 in the plan and a_1 needs to be executed before a_2. Further suppose that $r1_s$, $r1_e$, $r2_s$, and $r2_e$ represent the start and the end of actions a_1 and a_2, respectively. Then the dependency between the actions is represented as $r1_e < r2_s$ and this constraint is factored into a Bayesian network. This Bayesian network is used in the evaluation of the plan's makespan.

The ActuPlan algorithm returns the plan which achieves all the goals and has the shortest

makespan. The main algorithm is a forward chaining search algorithm guided by a relaxed planning graph based heuristic function which estimates the lower bound of the cost (plan makespan) to satisfy the goals. Along with the expansion of the plan, the random variables are generated and added into the Bayesian network. This Bayesian network is also used by the heuristic function to calculate the probability of satisfying the goals within a certain makespan.

ActuPlan can run in two modes: conformant mode and contingent mode. In the conformant mode, the agent does not have any visibility on the actual action duration and the steps in the plan are fixed. In contingent mode, the agent has full observability and can take different paths based on the actual action duration. In the contingent mode, contingent sub-plans are included in the plan. In ActuPlan, deadlines can also be attached to goal conditions. In this case, the plan needs to meet these deadlines while trying to minimize the overall makespan.

2.4 CoMDP and CPTP: Planning with Concurrent Actions in the MDP Framework

CoMDP and CPTP are recent MDP-based planners that can generate plans with concurrent actions considering uncertainty in action outcomes. CoMDP uses actions that have a unit duration, CPTP uses actions with numeric durations. Both planners are based on Markov decision process (MDP) framework. Therefore, we begin by reviewing the MDP algorithms [43].

A probabilistic planning problem with full observability can be modeled as an MDP because it satisfies the Markov property in that the decision of choosing the next action only depends on the current state and has nothing to do with the state history. In probabilistic planning, besides the initial state there is a set of goal states. The ultimate goal of the agent is to enter a goal state. The actions are probabilistic. Each action has a cost associated with it. The agent also wants to minimize the total cost of entering a goal state. Going into a dead end state where no action can be applied should be avoided by the agent since it means the failure of entering a goal state.

To formulate a probabilistic planning problem as an MDP problem, actions are represented as a transition function which specifies the probability of going from one state to another under some action a. The goal states and the dead end states are taken as the terminal states. When the agent enters a terminal state, it stays there forever. The solution of the resulting MDP problem is a policy that dictates which action should be taken for each state reachable from the initial state such that the expected total future cost for each state is minimal. This

policy is equivalent a contingent plan for the corresponding probabilistic planning problem.

Several techniques have been developed to reduce the search space size by modeling the MDP problem in a more compact way. One approach is to utilize a *dynamic Bayesian network (DBN)* to represent the actions instead of using a tabular transition function [50, 51, 52, 53]. The state space can then be factored into a set of abstract states based on the common outcomes under the actions. Another approach is to symbolically represent the entire MDP problem using algebraic decision diagrams (ADDs) [54, 55, 56]. Actions, reward function, value function and policy are all represented in the form of ADDs. Through ADD manipulation, the value function and policy are updated without enumerating the underlying states.

Many algorithms have been developed to solve MDP problems. Among them, *Real-Time Dynamic Programming (RTDP)* is an asynchronous value iteration algorithm that exploits the reachability property of the states to speed up the search [57]. Instead of indiscriminately updating every state in the state space, it focuses on updating the states that are more likely to be visited. RTDP can quickly produce a policy that is relatively good. The drawback of RTDP is that it converges very slowly because the rarely visited states are infrequently updated but they are needed for full convergence. To fix this problem, the Labeled RTDP (LRTDP) algorithm was developed [14] . LRTDP marks the states that have already converged so that the computation efforts can be switched to states that have not yet converged. Another approach is to generate an approximate solution. Sampled RTDP only simulates the scenarios that are likely to happen and ignores the ones that are less likely [58]. It alleviates the problem of exponential explosion of the search space. However, the solution found is not guaranteed to be optimal.

LRTDP was originally developed to solve regular (non-parallel) probabilistic planning problems and was then extended to solve the concurrent probabilistic planning problem where actions can be executed in parallel. In concurrent probabilistic planning, the state transitions are based on action combinations rather than individual actions. Thus, at each state, the agent needs to decide which subset of actions should be applied. The key step in formulating a concurrent probabilistic planning problem as an MDP problem which LRTDP can solve is to create the set of applicable action combinations for each state. Action combinations are sets of compatible actions that are applied in parallel similar to the individual actions in regular probabilistic planning problems. The MDP problem constructed from a concurrent probabilistic planning problem is called a CoMDP [21]. After a concurrent probabilistic planning problem is transformed into a CoMDP, LRTDP can be used to solve it.

Concurrent probabilistic temporal planning (CPTP) is a planning model that represents actions with different durations. If we let the duration of an action combination to be the

duration of the longest action in it, then a lot of time is wasted in waiting for the longer actions to finish. CPTP solves this problem by augmenting each state to include not only the discrete logical variables but also the unfinished actions together with their remaining durations. When a state is expanded, the time is advanced by the duration of the shortest action in the union of the applied action combination and the set of unfinished actions in this state. All the remaining unfinished actions are put into the set of unfinished actions of its children states. When any single action finishes, it is time to consider applying new action combinations.

CPTP uses two heuristic functions. In the first heuristic function, the base MDP problem is solved to find the cost. The cost is then divided by the maximum number of concurrent actions and is used as the initial cost for CPTP. The second heuristic function solves a relaxed version of CoMDP generated from the CPTP problem by ignoring the action duration and the mutual exclusivity (mutex) between the actions. Both heuristics are admissible. The second heuristic function is more informative because concurrency is taken into account.

In the next Chapter, we describe the details of the planning problem that we solve and then introduce the CPOAO* algorithm which we developed. Our base CPOAO* algorithm can be considered as an extension of HAO*. In contrast with HAO*, we use hyperarcs to represent action combinations rather than individual actions. Actions can be executed in parallel and can be aborted in the middle of execution. Because the planning problems we solve have time and resource constraints and actions consume at least some time, we can safely use backward propagation to update the values. One simplification we make is to use a discrete time model and not include uncertainty on action durations. This is an interesting feature to add in the future research because many actions are expected to have variable durations in the real world.

Chapter 3

CPOAO*: Solving Concurrent, Probabilistic, Over-Subscribed Planning Problems with AO* search [1]

CPOAO* is a planner that allows concurrent execution of probabilistic actions with discrete durations[59]. The goals have associated reward values which enable the planner to select the optimal set of goals to be achieved. CPOAO* uses an input and domain description language similar to PPDDL (Probabilistic Planning Domain Definition Language) [60, 24]. We extended PPDDL to include time, resources, and numeric reward values for the goal conditions. The planning algorithm conducts a state-space search using the AO* framework. In this Chapter, we describe the representation and algorithm CPOAO* uses.

3.1 The Planning Problem

A concurrent, probabilistic, over-subscribed planning problem is defined as a five-tuple (S, A, s_0, R, t_{max}), where,

- A is the set of actions,
- S is the state space,

[1] ©2008 AAAI. Portions reprinted with permission, from **Li Li**, Nilufer Onder, "*Generating Plans in Concurrent, Probabilistic, Over-Subscribed Domains*", in Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence (AAAI 2008), pp. 957–962.

- s_0 is the initial state,
- R is the reward set, and
- t_{max} is the maximum time allowed for plan execution.

The action set A contains all the actions that are available to the agent. Each action in set A consists of a precondition list l_{pre}, an outcome list l_o, a time duration t, and a list of resource consumption requirements l_{res}. The precondition list represents the logical requirements that must hold for the action to be applied in a state. An outcome o_n is defined as a triple $(add_n, del_n, prob_n)$ denoting the add list, the delete list, and the probability that this outcome happens. The total probability of all the outcomes in the outcome list of an action should be 1. For each outcome, the add-list contains the propositions that will be true after the execution of the action. The delete-list contains the propositions that will be false after the execution of the action. We assume that every action has non-zero duration. This assumption is often the case in the real world and we use it to guarantee the termination of our algorithm.

We adopt the common semantics for action execution. Before an action can be executed in a state, the preconditions must hold, the required resources specified in the resource consumption list must be available, and the remaining time must be greater than or equal to the time duration of the action. After an action is executed and completed, the result is a set of new states which correspond to the list of outcomes. For each outcome, a new child state is generated where the propositions in the add list are added, the propositions in the delete list are deleted, the resources used are subtracted, and the new time is recorded according to the duration of the action. The probability of the new state generated by outcome o_n from state s_j is the probability of s_j multiplied by the probability of outcome o_n.

In some situations, the best choice is to remain in a particular state to guarantee the maximum total reward. Therefore, CPOAO* domains include a special "Do-nothing" action in the action set A. The result of applying the "Do-nothing" action is a new child state that is exactly the same as the parent state but is flagged as a *terminal* state. No actions can be applied in a terminal state. In addition to the states resulting from the "Do-nothing" actions, the second type of terminal state in our framework are the states where all the rewards have been collected and thus there is no need to execute further actions.

CPOAO* provides a mechanism to stop actions before they finish executing. If an action is aborted, then the add and delete lists are not applied and the only impact is the resources used so far. The consumption is calculated in proportion to the elapsed execution time. For example, consider an action that has a duration of 5 time units and requires 4 power units to complete the execution. If this action is aborted after 3 time units, then it consumes 4*(3/5) = 2.4 power units.

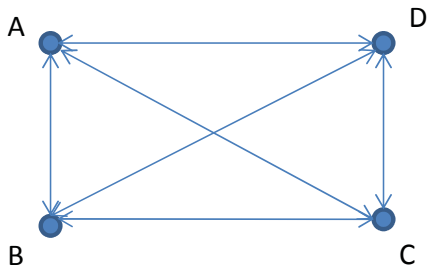


Figure 3.1: Mars rover map.

The actions in the action set A can be executed in parallel if they are compatible. A pair of actions are considered to be compatible if they don't have contradictory effects or one action doesn't delete a precondition of the other. Executing actions in parallel has value in the real world because usually time is a very critical resource in a task. In addition, in a domain where action results are uncertain, we can exploit parallelism to make the plan more robust. For example, we can execute two actions in parallel for the same goal to increase the probability of achieving the goal.

S represents the set of all states that are in the agent's environment. Each state in S is a 4-tuple (s_p,s_a, t,s_r) where s_p is a set of propositions which are *true* in this state, s_a is the set of currently executing actions, t is the available time, and s_r is a list of numeric resource values. The propositions in s_p are similar to the state variables in classical planning problems. For example, the proposition At_Location_A represents whether the agent is at location A or not. Because we allow actions to run in parallel, when one action finishes, there might be some other actions still running. Set s_a is used to track these unfinished actions. An unfinished action is an action with the additional information about the remaining time duration and resource requirements. Variables t and s_r have numeric values and represent the remaining time and resource levels in a state, respectively.

Each reward in R is a proposition-value pair. In a state, if a reward proposition is true, the corresponding reward value is counted as an achieved reward and is added to the total rewards. In our model, we do not have the concept of hard goals, i.e., goals that must be achieved, because the actions are probabilistic and the results are uncertain. Therefore, a plan that guarantees the achievement of a goal with probability 1 might not exist. Instead, important goals are represented by assigning very high rewards. The last item in the problem definition is t_{max}, which defines a time limit for plan execution to finish.

In the initial state $s_0 = (s_{p0}, s_{a0}, t_0, s_{r0})$, the propositions in s_{p0} are true, the resources have the initial values given in the resource list s_{r0}, no actions have been started so s_{a0} is empty, and t_0 is the time limit given to the problem.

Table 3.1
Actions in the Mars rover domain.

Action	Time Required	Energy Required	Success Probability	Description
move	5	5	100%	Move the rover from one location to another location
collect-sample	6	6	70%	Collect soil sample at a location
take-picture-long	4	4	60%	Take a picture at a location (higher success probability)
take-picture-short	3	3	50%	Take a picture at a location (lower success probability)

As our running example, we use the Mars rover domain which is a popular planning domain that has been used widely [21]. In this domain, a Mars rover has landed on Mars and has a list of tasks to complete. In our first example, the Mars rover needs to collect soil samples and take pictures. At the beginning, it is at location A. It can travel to locations B, C and D as shown in Figure 3.1. At location B, it can collect a soil sample and can take a picture. At location C, it can collect a soil sample. At location D, it can take a picture. To collect the soil sample, the Mars rover needs to execute the collect-sample action. To take a picture, it can either execute take-picture-long action or take-picture-short action. Both of them can take a picture but they have different durations, energy costs, and success probabilities. Table 3.1 shows the details of these actions.

The action space consists of all the ground actions (instantiated actions). In the following, we show example actions from the action space. The collect-sample action is probabilistic and has two outcomes, with probabilities 0.7 and 0.3. Outcome o_1 is the successful outcome with the desired effect achieved. Outcome o_2 is the failure case. The move action is deterministic and has one outcome.

$$\text{collect-sample(B)}=\begin{cases} l_{pre} = & \{\text{At\_Location\_B}\} \\ l_o = & \begin{cases} o_1 = \begin{cases} add_1 = & \{\text{Sample\_Collected\_B}\} \\ del_1 = & \emptyset \\ prob_1 = & 0.7 \end{cases} \\ o_2 = \begin{cases} add_2 = & \emptyset \\ del_2 = & \emptyset \\ prob_2 = & 0.3 \end{cases} \end{cases} \\ t = & 6 \\ l_{res} = & \{\text{energy} = 6\} \end{cases}$$

$$\text{move(A,B)} = \begin{cases} l_{pre} = & \{\text{At\_Location\_A}\} \\ l_o = & \begin{cases} o_1 = \begin{cases} add_1 = & \{\text{At\_Location\_B}\} \\ del_1 = & \{\text{At\_Location\_A}\} \\ prob_1 = & 1 \end{cases} \end{cases} \\ t = & 5 \\ l_{res} = & \{\text{energy} = 5\} \end{cases}$$

The rover has 20 energy units at the beginning and is given 11 time units to work on the task list. The goal is to maximize the reward points collected. When we translate the above planning problem description into the formal definition given in this Chapter, we have the following.

$$\text{The initial state } s_0 = \begin{cases} s_{p0} = & \{\text{At\_Location\_A}\} \\ s_{a0} = & \emptyset \\ t_0 = & 11 \\ s_{r0} = & \{\text{energy} = 20\} \end{cases}$$

The state space includes all the possible states which can be reached from the initial state. For example, by taking action move(A,B) to move from Location A to Location B, the Mars rover will reach state s_1 shown below.

$$s_1 = \begin{cases} s_{p1} = & \{\text{At\_Location\_B}\} \\ s_{a1} = & \emptyset \\ t_1 = & 6 \\ s_{r1} = & \{\text{energy} = 15\} \end{cases}$$

From state s_1, the Mars rover can start 3 actions at the same time, which are collect-sample(B), take-picture-long(B), and take-picture-short(B). Because take-picture-short(B) is the shortest action, it will complete first. After it completes, the Mars rover will be in state s_2. In this state, collect-sample(B) and take-picture-long(B) are unfinished actions. The remaining time duration for collect-sample(B) is 3 time units. The remaining time duration for take-picture-long(B) is 1 time unit.

$$s_2 = \begin{cases} s_{p1} = & \{\text{At\_Location\_B}, \text{Picture\_Taken\_B}\} \\ s_{a1} = & \{(\text{collect-sample(B)}, time = 3), (\text{take-picture-long(B)}, time = 1)\} \\ t_2 = & 3 \\ s_{r2} = & \{\text{energy} = 6\} \end{cases}$$

The reward proposition-value pairs for this example are listed below. A high reward value is assigned to At_Location_A to make sure that the rover moves back to the start location.

$$R = \begin{cases} (\text{At\_Location\_A}, 10) \\ (\text{Sample\_Collected\_B}, 3) \\ (\text{Picture\_Taken\_B}, 3) \\ (\text{Sample\_Collected\_C}, 3) \\ (\text{Picture\_Taken\_D}, 3) \end{cases}$$

In general, there are two cases in which we can use parallel execution to achieve the planning goals. We illustrate this with an example from the Mars rover domain. In this setting, the Mars rover is currently located in location A. There are two rewards "Picture-Taken-A1" and "Picture-Taken-A2" the Mars rover can collect at location A. The time allowed is 5 units. Consider a situation where each picture reward has a value of 10. In this case, the best strategy is to execute take-picture-long to achieve one reward and execute take-picture-short to achieve another. The total expected reward will be $10 \times 60\% + 10 \times 50\% = 11$. In this case, we execute actions in parallel to achieve different rewards. Because all the actions need to finish to collect the rewards, we call it the case of *all-finish*. In another situation, suppose one picture reward has a value of 100 and the other has a value of 10. Then the best strategy is to use both take-picture-long and take-picture-short to achieve the same reward which has the value of 100. The expected reward for action take-picture-short is $100 \times 50\% = 50$. The take-picture-short finishes earlier than the take-picture-long. The expected remaining unachieved reward after take-picture-short finishes is $100 \times (1 - 50\%) = 50$. So, the expected reward for action take-picture-long is $50 \times 60\% = 30$. Therefore, the total expected reward is $50 + 30 = 80$ which is the maximum expected reward. In this case, if take-picture-short succeeds in achieving the target reward, we can abort take-picture-long immediately. Because we use concurrent actions to achieve the same target reward and we will abort all other actions

after the earliest successful action finishes, we call it the case of *early-finish*.

Another example of using redundant actions comes from ProPL, a process monitoring language where processes might include redundant parallel actions such as seeking task approval from two managers when one approval is sufficient [61]. The focus of ProPL is on expressing and monitoring such actions. Our focus is in generating plans that can use redundant actions, as well as marking the actions that need to be terminated.

The solution to the planning problem defined in this section is a contingent plan that produces the maximum expected total reward for the initial state s_0. Different from the policies created by classic MDP planning programs, each plan step in our planning problem is a set of concurrent actions which we call a concurrent action set (*CAS*). The plan consists of pairs of states and CASs. In a CAS, 3 sets are maintained, i.e., start action set l_{start}, terminating action set l_{term}, and ongoing action set $l_{ongoing}$. The start action set includes all the new actions that are going to be started. The terminating action set has the actions that will be aborted. The ongoing action set includes the actions that can continue to execute. The actions in the union of start action set and ongoing action set are required to be compatible with each other. The duration of a CAS is determined by taking the shortest duration from the actions in the start action set and the remaining duration of the actions in the ongoing action set. At the end of the application of a CAS, new children states are created by applying the outcomes of the finished actions in the CAS. The remaining unfinished actions are added into the unfinished action list of the new children states.

Using the Mars rover example, suppose that the current state s_i is the following.

$$s_i = \begin{cases} s_{pi} = & \{\text{At\_Location\_B}\} \\ s_{ai} = & \{(\text{collect-sample(B)}, time = 5), (\text{take-picture-long(B)}, time = 3)\} \\ t_i = & 5 \\ s_{ri} = & \{\text{energy} = 11\} \end{cases}$$

An example of applicable CAS for state s_i is

$$CAS_x = \begin{cases} l_{start} = & \{(\text{take-picture-short(B)}, time = 3)\} \\ l_{term} = & \{(\text{collect-sample(B)}, time = 5)\} \\ l_{ongoing} = & \{(\text{take-picture-long(B)}, time = 3)\} \end{cases}$$

l_{start} can be an empty set, but the union of l_{term} and $l_{ongoing}$ is aways equal to the set of unfinished action s_{ai}. Below we present another example to illustrate how the CAS outcomes are computed.

$$CAS_y = \begin{cases} l_{start} = & \emptyset \\ l_{term} = & \{(\text{collect-sample(B)}, time = 5)\} \\ l_{ongoing} = & \{(\text{take-picture-long(B)}, time = 3)\} \end{cases}$$

Both CASs above have a duration of 3 time units. At the end of a CAS, the outcomes of all finished actions are applied and merged to create new children states. For CAS_x, the two children states are the following.

$$s_{i+1} = \begin{cases} s_{pi} = & \{\text{At\_Location\_B}, \text{Picture\_Taken\_B}\} \\ s_{ai} = & \{(\text{collect-sample(B)}, time = 2)\} \\ t_i = & 2 \\ s_{ri} = & \{\text{energy} = 2\} \end{cases} \quad \text{(probability = 80\%)}$$

and

$$s_{i+2} = \begin{cases} s_{pi} = & \{\text{At\_Location\_B}\} \\ s_{ai} = & \{(\text{collect-sample(B)}, time = 2)\} \\ t_i = & 2 \\ s_{ri} = & \{\text{energy} = 2\} \end{cases} \quad \text{(probability = 20\%)}$$

Applying CAS_y would generate the same children states but with different probabilities which are 60% and 40%.

In a valid CAS, the union of l_{start} and $l_{ongoing}$ should never be empty. In the case where there are no applicable actions, the start action set l_{start} will have the do-nothing action in it. If the start action set of a CAS includes do-nothing, it generates a terminal state with 100% probability. Below is an example CAS that generates a terminal state.

$$CAS_z = \begin{cases} l_{start} = & \{(\text{do-nothing}, time = 0)\} \\ l_{term} = & \{(\text{collect-sample(B)}, time = 5), (\text{take-picture-long(B)}, time = 3)\} \\ l_{ongoing} = & \emptyset \end{cases}$$

The optimal solution is the plan that maximizes the expected total reward of initial state s_0. In the next section we explain the CPOAO* planning algorithm.

3.2 The CPOAO* Algorithm

We developed a new algorithm named CPOAO* (Concurrent Probabilistic Oversubscribed AO*) which can find the optimal solution plan for the planning problems defined in this Chapter. This algorithm searches in the state space of the planning problem and maintains two AND/OR graphs. The first graph is the *working graph*, WORK-G, which contains the entire search space explored. The second graph is the *solution graph*, SOLN-G, which is a sub-graph of WORK-G and represents the current best plan according to the working graph expanded so far. The nodes in these two graphs are the states and the hyper-arcs are the CASs. Each CAS starts from a parent state and ends in an "and" set of states which consists of all the possible children states. The "or" relations between CASs in the WORK-G model the fact that for a given parent state there are multiple choices of CASs that can be taken.

Initially, both WORK-G and SOLN-G only contain the initial state s_0. Then WORK-G is expanded iteratively by applying all the applicable concurrent action sets on the unexpanded states which are also included in graph SOLN-G. When a state s_i is expanded and multiple CASs and their corresponding sets of children states are added into WORK-G, an admissible function is used to estimate the expected reward for each newly generated child state. The CAS which gives the maximum expected reward for state s_i is selected as the best CAS and the expected reward of state s_i is thus updated. After this, the expected total reward and the best CAS are re-calculated for each ancestor state of state s_i. If the expected reward of an ancestor state s_a changes, the parent states of s_a also have their expected reward re-calculated and best CAS re-selected. Finally, the solution plan graph SOLN-G is re-constructed by starting from the initial state and following the best CAS along the way. This process is repeated until every state in SOLN-G is either expanded or is a terminal state. At that time, SOLN-G is the optimal solution plan. This plan returns the maximum expected reward for the initial state. It specifies which CAS should be taken for each state that are reachable following the plan. It is a contingent plan because it takes care of both the success and failure outcomes of an action. The algorithm is shown in Algorithm 3.

The main loop of the search starts at Line 3, and continues until the solution graph doesn't have any non-terminal, unexpanded states. From Line 5 to Line 15, all the non-terminal unexpanded states in the current solution graph SOLN-G are expanded. To expand a state s, first the set of all applicable concurrent action sets is generated for state s. Then, every applicable CAS is applied to create the child states. Following that, an upper bound estimate of the expected reward is calculated for each new state and the best CAS is found for state s. At the end of the expansion, the parent states of the expanded state are added into the set Z. At Line 16 through Line 24, all states in set Z and their ancestor states are traversed bottom-up to re-calculate the best CAS and expected reward. In the last part of

Algorithm 3 CPOAO* Main Algorithm

```
1: The working graph WORK-G initially only contains the initial state s_0.
2: The solution graph SOLN-G initially only contains the initial state s_0.
3: while SOLN-G has unexpanded non-terminal states do
4:    Let Z be the empty set.
5:    for all s which is an unexpanded non-terminal state in SOLN-G do
6:       Generate the set of all applicable CASs for state s, denoted as C_s, by calling the
         Generate Concurrent Action Sets procedure
7:       for all c in C_s do
8:          Apply c on state s to generate the child states of s.
9:          Calculate the upper bound of expected rewards for newly generated child states
            using heuristic functions.
10:      end for
11:      Find the best CAS for state s. The best CAS is the one that provides the the
         maximum expected reward based on the estimated rewards of the child states.
12:      Mark the best CAS on graph WORK-G.
13:      Update the estimated expected reward of state s based on its best CAS.
14:      Add the parent states of s into set Z if the estimated expected reward of s has
         changed.
15:   end for
16:   while Z is not empty do
17:      Choose a state s' ∈ Z that has no descendant in Z.
18:      Re-select the best CAS for state s'
19:      Update the estimated expected reward of s' using the new best CAS
20:      if The estimated expected reward of s' has changed then
21:         Add the parent states of s' into Z.
22:         Remove state s' from Z
23:      end if
24:   end while
25:   Re-generate SOLN-G by following the best CASs from the initial state s_0 to the leaf
      states in the graph.
26: end while
27: When every state in SOLN-G is either expanded or is a terminal state,
    SOLN-G contains the optimal plan,
    Return SOLN-G.
```

the algorithm, SOLN-G is re-generated from initial state following the path of the updated best CASs.

At the terminal states, plan execution stops and the total rewards are calculated by adding up the rewards specified in the reward proposition-value pairs for the true propositions. Because an achieved reward in a non-terminal state can be removed in order to achieve

some other reward, we don't calculate the exact total reward for non terminal states using reward proposition-values. For non terminal states, we calculate the expected total rewards by adding up the expected total rewards of its children states weighted by probability. Given a plan, the expected total reward of a state, $E(s_i)$, is calculated using the following formula:

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

where $C(s_i)$ is the set of child states of s_i after applying the corresponding CAS in the plan, $P_{(i,j)}$ is the probability of entering state s_j from state s_i according to the plan, and R_{s_i} is the exact total reward achieved in the terminal state s_i.

At Line 6, the procedure shown in Algorithm 4 is called to generate the set of applicable concurrent actions sets. First, the set of all applicable actions for state s is created. This set contains all candidate actions to put into the start action set of the new CAS. Then the actions which are also an unfinished action in state s are removed from this set because there is no benefit in aborting an executing action restart it immediately. Afterwards, two power sets are built: one from the newly starting actions and one from the unfinished actions. Each power set contains all the combinations of the actions from the base set. From the power set of newly starting actions, the incompatible action combinations are removed. This is not needed for the power set of unfinished actions because they must be compatible given the fact that they are already running at the same time. Next, a Cartesian product is taken between the two power sets. This creates the set which has pairs of start action set and ongoing action set. Finally, the pairs are extended to include terminating action set by subtracting the ongoing action set from the set of unfinished actions in state s. The output of this procedure is the set of applicable concurrent action sets for state s. In the worst case, when all actions are applicable and are also compatible with each other, the number of applicable CASs is 2^n, where n is the number of actions. Any technique which can safely prune the set of applicable CASs is highly desired. In the next Chapter, we present a technique which can reduce the size of the set of applicable CASs when time is the only constraint.

In Figures 3.2 to 3.8, we show how the working graph and the solution graph are generated by the CPOAO* algorithm with the Mars rover example. At each iteration, we expand the tip unexpanded non-terminal states in the current solution graph. After the expansion, the expected total rewards are updated and the solution graph is recomputed. These iterations continue until there are no unexpanded non-terminal states in the solution graph. In this example, we only show the first 3 iterations.

At the beginning, both working graph and solution graph only contain the initial state s_0 (Figure 3.2). In this example, the initial expected total rewards of the new states are

Algorithm 4 Procedure: Generate applicable concurrent action sets for state s

1: Create set AP_s that includes all the applicable actions for state s.
2: Let AU_s be the set of unfinished actions in state s.
3: Delete from AP_s all the actions that exist in AU_s.
4: Create set $AP_s{}^p$ which is the power set of AP_s.
5: Delete from $AP_s{}^p$ all the action combinations that are not compatible.
6: Create set $AU_s{}^p$ which is the power set of AU_s.
7: Create the product set $AS_s{}^p = \{(x,y) : x \in AP_s{}^p; y \in AU_s{}^p\}$.
8: Delete from $AS_s{}^p$ all the (x,y) pairs that are not compatible.
9: Extend each pair in $AS_s{}^p$ to be a triple by adding the termination action set z which equals AU_s - y.
10: The resulting set $AS_s{}^p = \{(x,y,z) : x \in AP_s{}^p; y \in AU_s{}^p; z = AU_s - y\}$ is the set of all applicable CASs for state s.
Return $AS_s{}^p$.

calculated using a hypothetical admissible heuristic function h. After a state is expanded, the expected total reward is calculated from its children states.

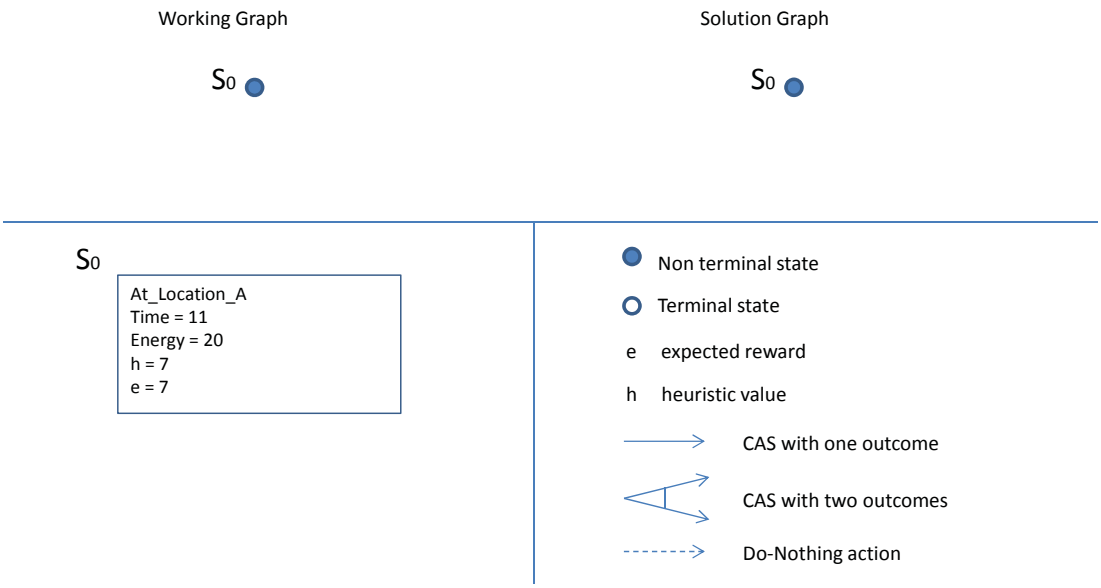


Figure 3.2: Initial state.

Figure 3.3 shows the result of the first iteration after the initial state is expanded and all the applicable CASs are applied. The do-nothing action is also added (shown in dashed lines). All the new states except for s_4 are non-terminal states. s_4 is a terminal state where expansion stops. Its expected reward is calculated based on the state propositions rather than using the heuristic function. s_4 has zero reward because it doesn't have any reward propositions.

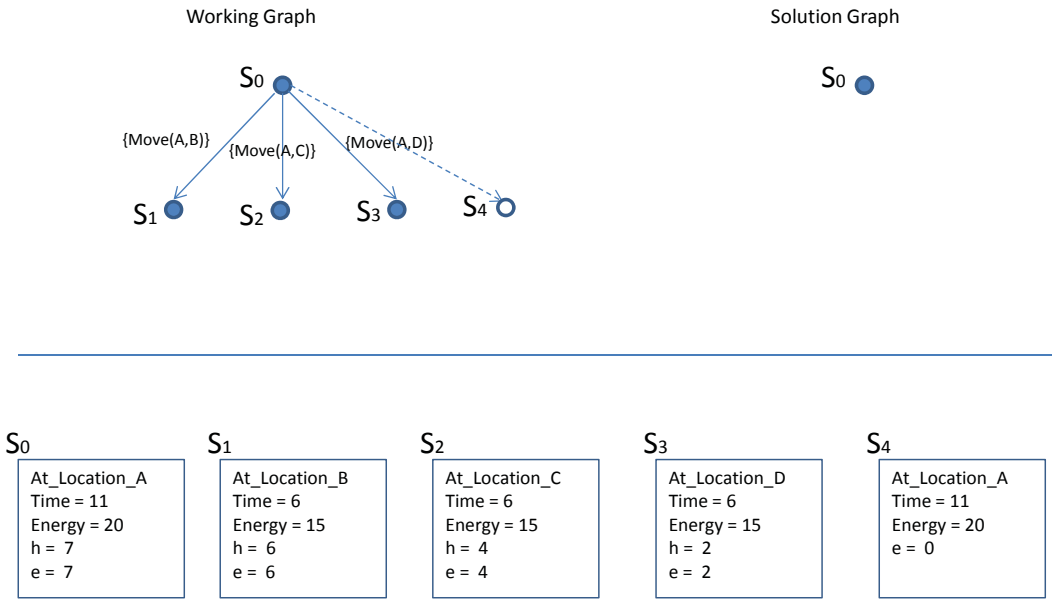


Figure 3.3: Expansion of s_0 (first iteration).

After the expansion, the expected rewards are re-calculated for all the ancestor states of the new states (Figure 3.4). The maximum reward returned by one of the CASs is used as the expected reward of the parent state. In the example, Move(A,B) is the best CAS, so the expected reward of s_0 is updated to 6. The solution graph is regenerated to include Move(A,B) and s_0.

Figure 3.5 shows the next iteration where the unexpanded state s_1 in the solution graph is expanded. New states from s_5 to s_9 are added into working graph.

After the expected reward update, s_1 has a reward of 2.5 which is less than the expected reward for state s_2. As a result, the new solution graph doesn't include s_1 and Move(A,B). Instead it has s_2 and Move(A,C) (Figure 3.6) .

Figure 3.7 shows the expansion of state s_2. As a result of expected reward update, state s_2 now has a reward of 1.4 (Figure 3.8). Therefore, the solution graph is re-generated to switch back to s_1 and its child states. State s_3 is never expanded because it has a low expected reward.

We can see that there is no need to expand the states that have lower expected rewards than the total expected rewards given by the current solution graph. This example shows that several search branches can be pruned by utilizing a good heuristic function. In the next

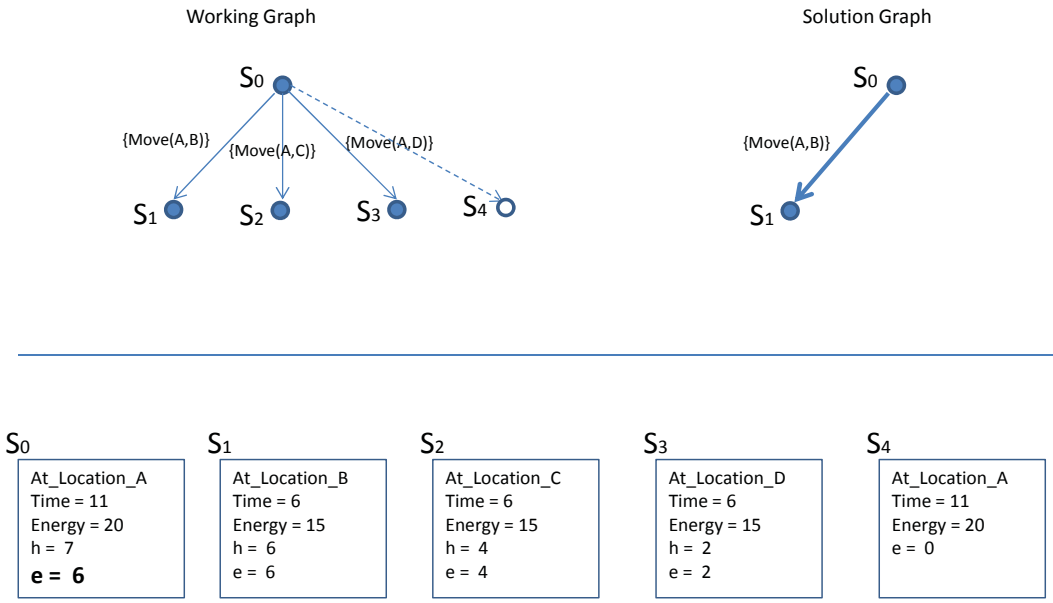


Figure 3.4: Update the expected reward for s_0, re-generate the solution graph.

section, we present the heuristic functions we have developed.

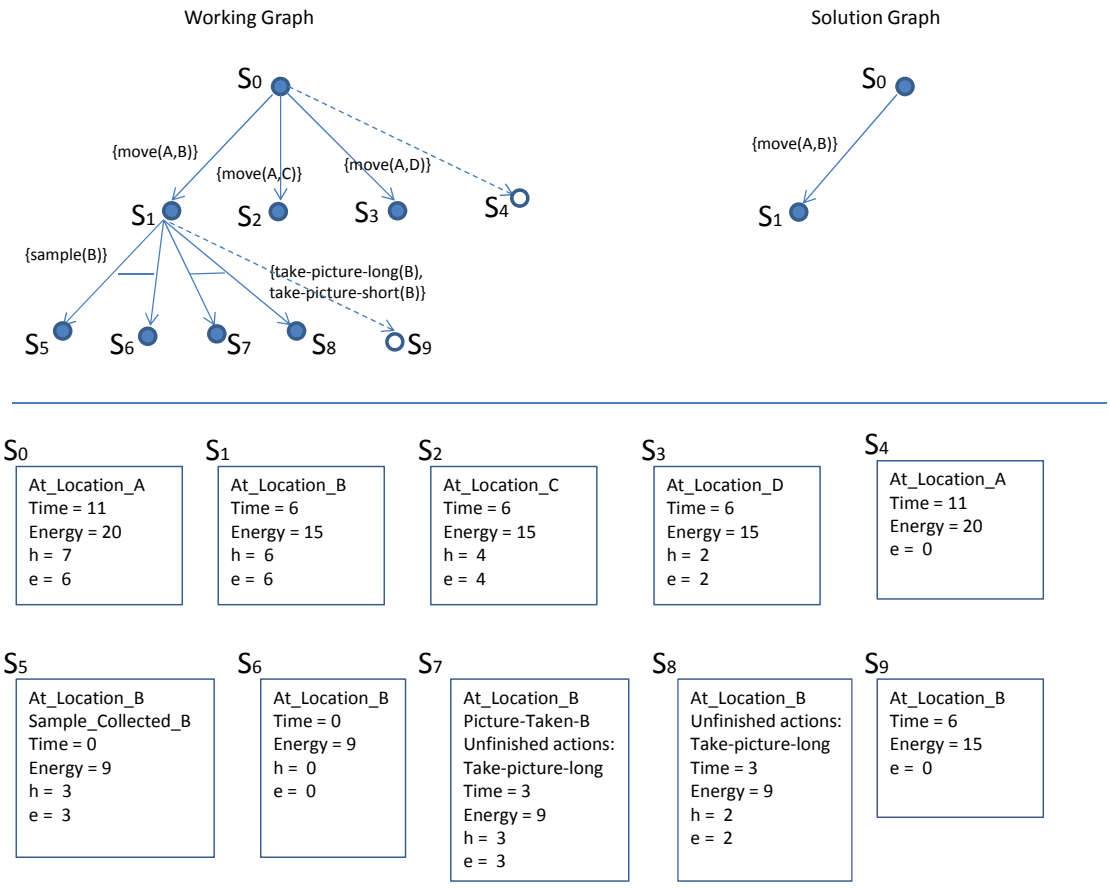


Figure 3.5: Expansion of s_1 (second iteration).

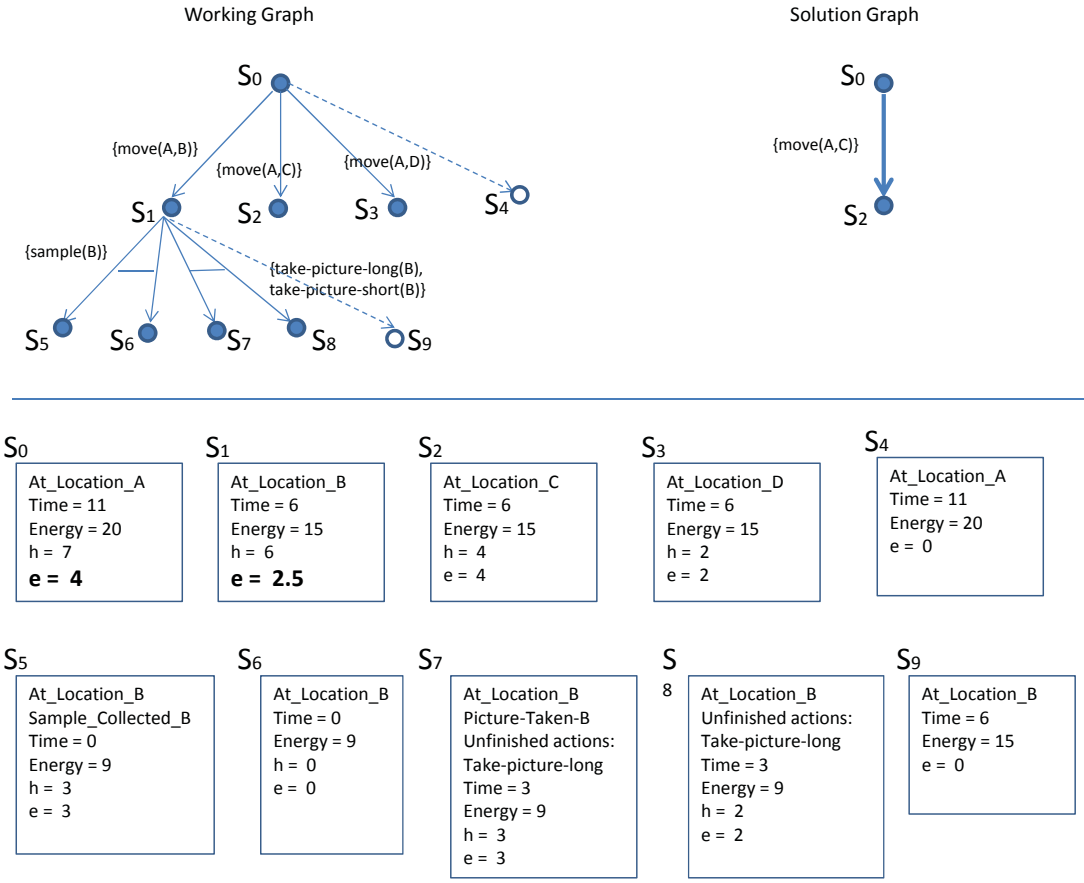


Figure 3.6: Update the expected reward for s_1 and s_0, re-generate the solution graph.

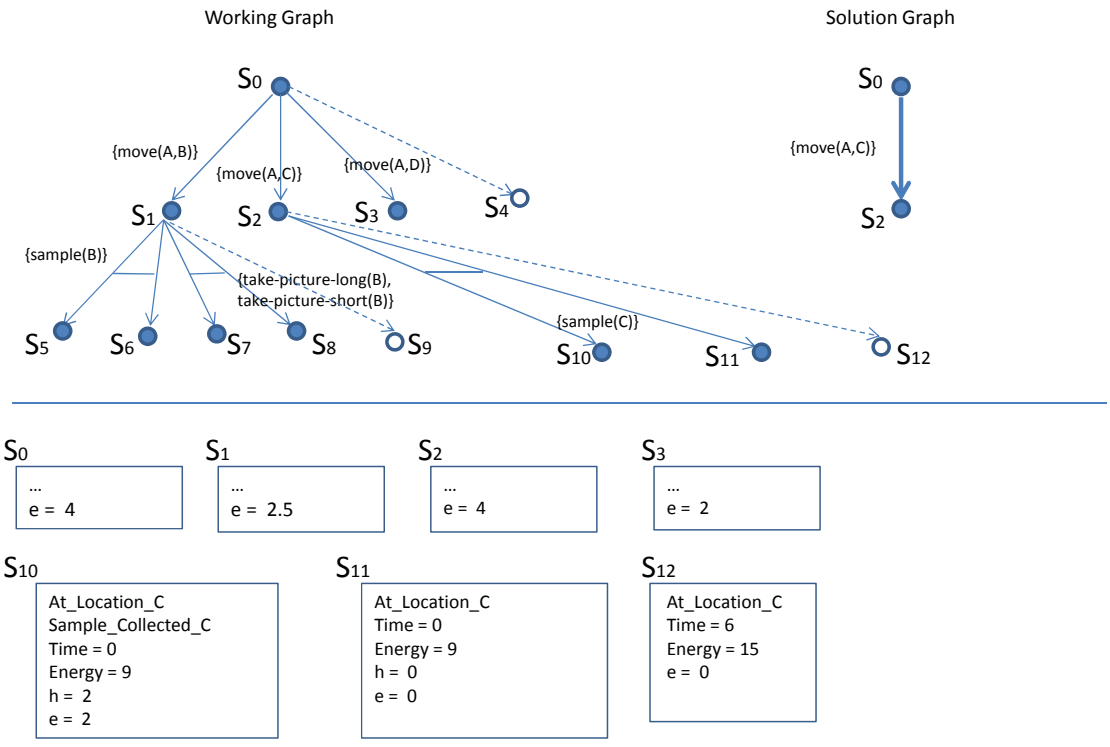


Figure 3.7: Expansion of s_2 (third iteration).

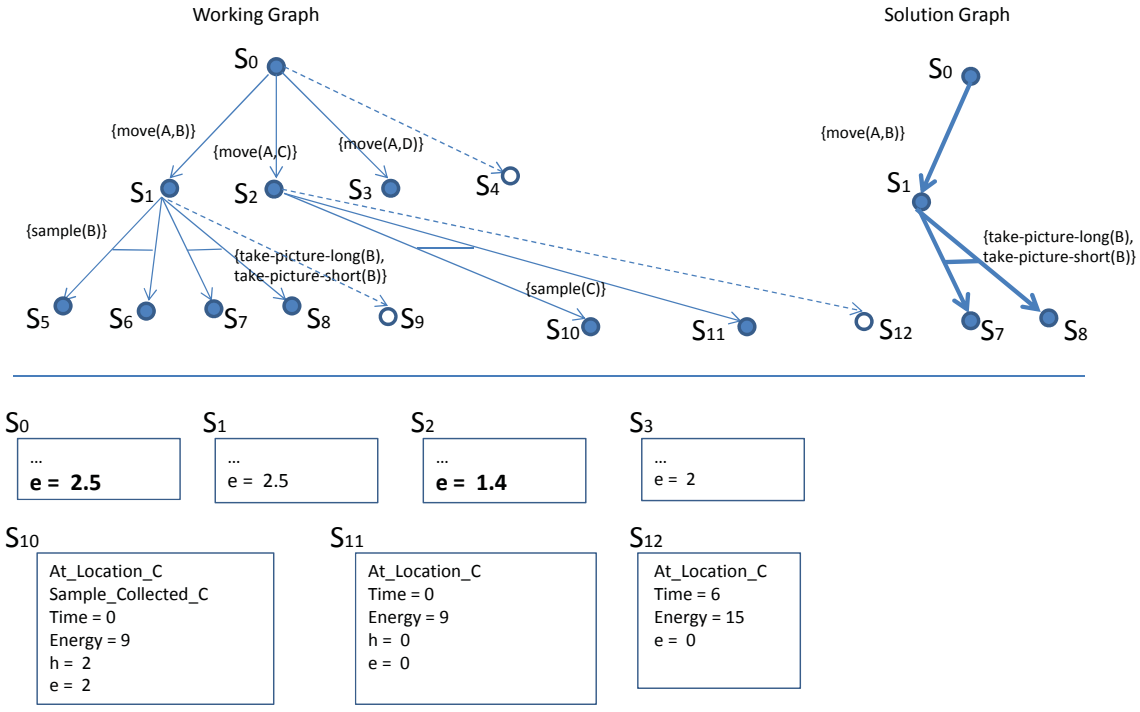


Figure 3.8: Update expected reward for s_2 and s_0, re-generate solution graph.

Chapter 4

Heuristic Functions [1]

Heuristic functions play an important role in search algorithms because they can reduce the search space significantly by steering away from non-promising branches. A good heuristic function is informative, admissible, and easy to compute. However, it is often not possible to achieve all of these characteristics, and tradeoffs must be made. For example, $Sapa^{ps}$, an over-subscription planner, calculates the heuristic value of a state s by constructing the relaxed planning graph with state s as the root node and propagating the action costs to the goals [34]. This heuristic value is not admissible because it uses the summation of the costs of the preconditions to compute the cost of an action. An inadmissible heuristic function can produce a sub-optimal plan which might not be desirable in critical tasks such as in the case of Mars rover. Inadmissible heuristics are useful when they are more informative than their admissible counterparts and the level of suboptimality is acceptable.

The HAO* planner described in Chapter 2 computes the heuristic values by solving a relaxed problem, which is a deterministic version of the original problem, using the same algorithm and a trivial heuristic function. The maximum reward of each state is saved in the memory. It uses the rewards found in the deterministic problem as the heuristic value for the states in the original problem. If it can not find a matching state in the relaxed problem for some state s in the original problem, it simply solves the relaxed problem again with state s as the initial state. The resulting heuristic values are admissible. While solving the relaxed problem is relatively easy compared to solving the original problem, it is still hard to solve. In addition, the heuristic might not be informative because ignoring the uncertainty of the probabilistic actions might generate heuristic values too far away from their actual values.

[1]

In this dissertation we focus on admissible heuristic functions and develop two groups of heuristics. The first group is developed for domains that have only time constraints. The second group contains heuristic functions that are more general and are applicable to domains with both time and resource constraints. We describe the heuristic functions we developed in the next two sections and describe the experimental evaluation in Chapter 5.

4.1 Heuristic Functions for Domains Constrained only by Time

The branching factor in CPOAO* search is very high because it needs to consider concurrent action sets (CASs) rather than individual actions. A CAS is a member of the power set of the action set A, and thus the size of the state space increases exponentially. To make the search space smaller, we apply a pruning technique to decrease the number of branches when time is the only constraint. This technique is based on the fact that we do not need to have a branch for a CAS if there is another CAS which is always *better* than it. We say concurrent action set CAS_i is better than concurrent action set CAS_j if the expected total rewards which could be collected by following the concurrent action set CAS_i is always greater than following the concurrent action set CAS_j. For example, the concurrent action set {collect-sample(A), take-picture-long(A), take-picture-short(A)} is always better than the concurrent action set {take-picture-long(A), take-picture-short(A)} or {collect-sample(A)} if the actions do not consume any resources. For simplicity, we write concurrent action set as an union of start action set and ongoing action set in this chapter. Starting the actions earlier is always better than having the agent stay idle. We can abort an action at any time if it turns out that there is no need to wait for its completion. The following two rules can be used to determine whether two concurrent action sets have the "better" relation.

- Rule 1: If the shortest action a in CAS_i does not delete any propositions, then concurrent action set CAS_i is always better than concurrent action set CAS_j which contains all the actions in CAS_i except action a.
- Rule 2: Suppose CAS_i and CAS_j are two concurrent action sets and $CAS_i = CAS_j \cup \{b\}$. If b is not the shortest action in CAS_i, then CAS_i is always better than CAS_j.

The first rule is saying that if an action does not delete any proposition, it does not harm to start it as early as possible. The second rule is saying that even if an action deletes some proposition, it is still safe to start it as early as possible as long as it is not the shortest

action in the concurrent action set. If it is not the shortest action, there will still be a chance to abort it before it makes any deletions. These two rules are used to reduce the size of applicable CAS set by keeping only the CASs that are better than others. This "better" relation is transitive. These two rules can be applied more than once to find out if one concurrent action set is better than another one. For example, if the concurrent action set CAS_i shown in the following is better than CAS_j by rule 1 and CAS_j is better than CAS_k by rule 2, then CAS_i is better than CAS_k.

CAS_i = {take-picture-short(A), take-picture-long(A), collect-sample(A)}
CAS_j = {take-picture-long(A), collect-sample(A)}
CAS_k = {take-picture-long(A)}

In addition to the CAS pruning technique above, we developed a heuristic function called *relaxed forward probability update* or *forward-relax* in short for domains which have only time constraints. forward-relax returns an estimated expected reward for a given state. The main idea of this heuristic function is to estimate the probabilities of individual propositions at future states given the initial state. To find the probability of a particular proposition at a future state, one way is to add up the probabilities of all future states (at a particular future time point) in which this proposition is true. However, because of the uncertainty of the actions, the number of child states grows exponentially as time moves forward. It is not feasible to track all of the future states. To avoid dealing with the large quantity of future children states the idea of forward-relax is to attach probabilities to propositions rather than states. When time moves forward and actions are applied, the probabilities of propositions are continuously updated. Several relaxing rules are employed to make the updates easier while keeping the admissibility of the heuristic function.

To calculate the heuristic value, a proposition set and an action set are created for each time point. The first time point is time 0, which corresponds to the target state s for which we want to calculate the expected total reward. All of the propositions in this state are added into the initial proposition set. Each of these initial propositions have a probability of 1. Then, based on these propositions, all applicable ground actions are found and inserted into the initial action set. An action may have multiple effects. Each effect has its probability, an add list and a delete list. The add list contains the propositions that should be added (positive consequences) whereas the delete list contains the propositions that should be deleted (negative consequences). In our heuristic function, all the negative consequences are ignored. By ignoring the negative consequences we don't need to split the proposition set to cope with effects that have different negative consequences. For each action, we keep its duration and the probabilities of its positive consequences. The probability of a positive consequence of proposition x is calculated using the following formula:

$$P_x = P_{action} \times P_{effect}$$

where P_x is the probability of the positive consequence of proposition x, P_{action} is the probability that the action will be executed, and P_{effect} is the probability of the effect given that the action is executed.

To calculate P_x, we need to calculate the conditional probability that all the preconditions of the action are true given the fact that the proposition x is not true. In a domain where every proposition can be achieved by only one type of action, knowing the truth value of proposition x will not provide any additional information about the probabilities of the preconditions. In such domains, the conditional probability becomes an unconditional probability. In other domains where one proposition can be achieved by multiple actions, we need to modify the action instances in a particular way while generating the plan graph so that we can convert the conditional probabilities into unconditional probabilities. In our testing domain, every proposition can be achieved by only one action. Thus, we can safely ignore the given condition that proposition x is not true and use P_{action}. But it is still complicated to calculate the exact value of this probability because the preconditions can be correlated. To simplify the calculation, we use an approximate probability. The probability of the precondition with the minimum probability is taken as P_{action}. Because this approximate probability is always greater than the exact probability, admissibility is maintained. P_{effect} is the probability of the effect which contains proposition x. This probability is a constant number.

To calculate the proposition set and the action set of the next time point, the first step is to copy the proposition set and the action set of the previous time point. Then, the time duration of the actions in the action set are decremented by 1. If an action has zero duration after decrementing, it is removed from the action set and its positive consequences are added into the proposition set. If a positive consequence introduces a new proposition, this proposition will be directly added into the proposition set with the consequence's probability. If the proposition already exists in the proposition set, its probability is updated using the following formula:

$$P_{x_{new}} = P_{x_{old}} + (1 - P_{x_{old}}) \times P_x$$

where $P_{x_{new}}$ is the updated probability of proposition x and P_x is the probability of the positive consequence of proposition x.

After all positive consequences of zero duration actions are incorporated into the proposition set, the proposition set for the new time point is created. Based on the new proposition set we search for all applicable ground actions. If an applicable action is not part of the current action set, it is added into the action set. If an applicable action already exists in the copied action set, we calculate the P_{action} for this action. At any given moment, only one instance of a ground action is allowed to run. If the P_{action} of a new instance of the action is equal to the P_{action} of the existing instance of the action, the new instance of the action is ignored and will not be inserted into the action set because we won't get any benefits by canceling the existing action and starting the new action. If the P_{action} of the new instance is greater than the P_{action} of the existing instance, then the positive consequences of the new instance will have higher probabilities. Therefore, it may lead to a higher expected total reward by canceling the existing instance and starting the new instance. So we need to add the new instance of the action into the action set. However, the probabilities of positive consequences are modified to offset the effects of the existing instance. The probability of the positive consequence of proposition x is calculated as follows:

$$P_{x_{i2_{new}}} = (P_{x_{i2}} - P_{x_{i1}})/(1 - P_{x_{i1}})$$

where, $P_{x_{i2_{new}}}$ is the updated probability of the positive consequence of proposition x in the new instance,
$P_{x_{i2}}$ is the initially calculated probability of the positive consequence of proposition x in the new instance, and
$P_{x_{i1}}$ is the probability of the positive consequence of proposition x in the existing instance.

Although obviously it is impossible to execute two different ground actions which are generated from the same lifted action at the same time, we relax this restriction and allow them to be executed simultaneously in our calculation of the heuristic function because we do not know which one is the better choice. After all applicable actions are analyzed and the suitable ones are added into the action set based on the above logic, the action set is constructed for the new time point. This process continues to run and builds the proposition set and action set for the ensuing time points until the time limit of the target state s is reached. The proposition set of the final time point is used to calculate the expected total rewards (ETR) for the target state s:

$$ETR_s = \sum(R_x \times P_x)$$

where R_x is the reward of proposition x, and
P_x is the probability that proposition x is true at the final time point.

4.2 Heuristic Functions for Domains Constrained by both Time and Resources

For domains which have both time and resources constraints, we developed two heuristics. The first heuristic, *reachability test*, is based on relaxed plan graph generation. A *relaxed plan graph* is a structure that ignores negative effects of actions [62, 48, 49]. When the negative effects are ignored, the structure contains all the solutions to the original problem and possibly more, and the solutions are easier to extract [1, 63]. To compute the heuristic value, we check if a proposition can be achieved but ignore the exact probability of achieving it. Starting with the propositions in the initial state, a relaxed plan graph is iteratively generated by applying the applicable actions. An action is applicable if all of its preconditions have been achieved and the time and resource requirements are satisfied. For each achieved proposition, we keep track of the maximum possible remaining time and resources after this proposition is achieved. For propositions in the target state, the maximum remaining time and resource levels are set to the specifications given in that state.

To check if time is sufficient to apply an action, we take the minimum of the maximum time left among all the preconditions and use that value, t_a, as the time available for the action. If t_a is greater than the time required, the time requirement is met. Similarly, for each resource, we take the minimum of the maximum resource remaining among the all preconditions and compare it to the required resource usage to find out if the resource requirement is satisfied. Once an action is applied, all of the propositions in its outcomes are merged into the plan graph. If a proposition is new, it is added with time and resource values obtained by subtracting the time and resource usage of the action from the values available to the action. If a proposition already exists in the plan graph, the current time and resource values in the plan graph are compared to the new values calculated. If a new value is greater than the existing value in the plan graph, the proposition is updated with the new value. For an action to qualify for selection, at least one of its preconditions should be a new proposition or one that has been updated in the previous iteration. The expansion stops when no actions can be applied. After the plan graph is generated, all the goals that are included in the final planning graph are considered to be reachable and their corresponding reward values are added together to calculate the heuristic value.

The second heuristic for domains which have both time and resources constraints is called

all-resource forward-relax. It also solves an easier problem but considers probabilistic outcomes, time span constraints, and resource constraints. The basic idea is to first find all possible ways to achieve each goal. Then we calculate the upper bound of the probability to achieve the goal based on the available time and resources. The more resources and time are given, the higher the probability will be, because the failed actions can be retried. At the end, the total probability for each goal is calculated and we use those values to calculate the total expected reward.

In this heuristic function, the planning problem is relaxed in the following ways:

1. All the "delete" outcomes are ignored.
2. All the actions are allowed to execute concurrently except that actions cannot be executed at the same time with their predecessor actions or children actions.
3. When there are multiple ways to achieve the same goal, we treat each of them independently and give them full access to the time allowed and other resources. Then we combine them together to get the total probability assuming that the events of achieving the goals are independent among the different ways of achieving the same goal. Time constraint check and resource constraint check are only applied to the same group of actions that are in the same causal link chain to achieve a single goal.

Because all of the above relaxations result in a higher probability of achieving the goals, we are calculating the upper bound of achieving the goals. Therefore, the heuristic function is an admissible function.

The detailed procedure is described as follows. First, we generate a relaxed planning graph by expanding forward from the start state. This planning graph consists of alternate layers of propositions and actions. All the propositions that are true in the start state form the first layer of the planning graph, and they are marked as "New" propositions. Next, we derive all the actions that are applicable. This group of actions form the second layer. The third layer is generated by adding the propositions in the outcomes of actions. During the iterations, a proposition is marked as "New" when it is freshly added into the graph. Later its status is changed to "Expanded" when the next layer of actions is added into the graph. An action can be added into the action layer only when all of its preconditions are in the graph and at least one of them is marked as "New". This iteration continues until either there are no "New" propositions left, or the resource or time limit has been reached. After that, for each goal, we extract all possible ways of achieving it from the planning graph. Each unique way of achieving a goal is represented by a planning subgraph which consists of all the required actions to achieve that goal. All of the actions in the extracted subgraph have to

be successful in order to achieve the goal. Then for each goal-specific action subgraph, we calculate the upper bound of the success probability using the action outcome probability and the chances to retry the failed branch. Finally, we combine all probabilities together to calculate the probability of achievement and use that value to calculate the total expected reward.

To illustrate the process of calculating the heuristic values using this approach, we use the Mars rover example. Figures 4.1 to 4.4 show how the heuristic value is calculated for initial state s_0.

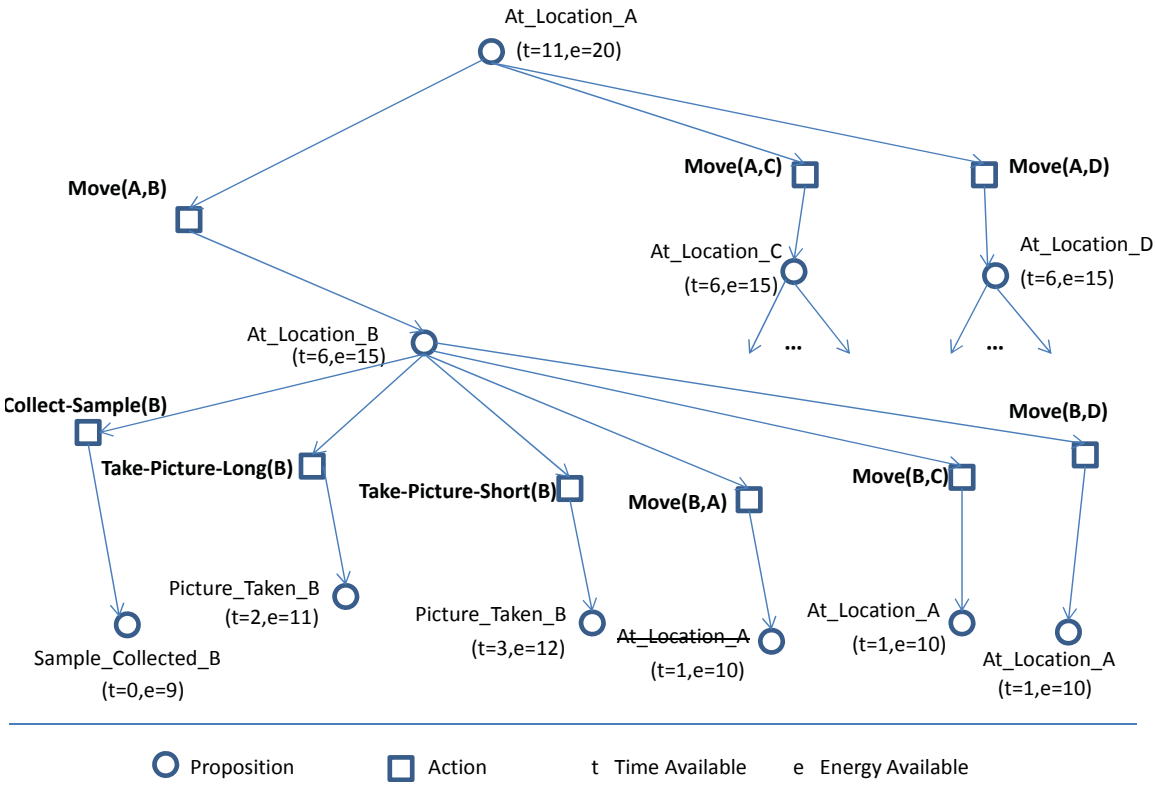


Figure 4.1: Relaxed planning graph.

Figure 4.1 shows the relaxed planning graph. The graph consists of action nodes and proposition nodes. The same action can be applied again if new precondition propositions are generated. At each proposition node, we keep track of the remaining time and resources available after achieving this proposition. These values are calculated based on the consumption of the actions and the available resource levels of the preconditions. For an action which has multiple preconditions, the lowest time or resource level is taken. In the expansion, the loops are removed, e.g., At_Location_A under action Move(B,A) has been crossed out. The expansion stops when no new actions can be added.

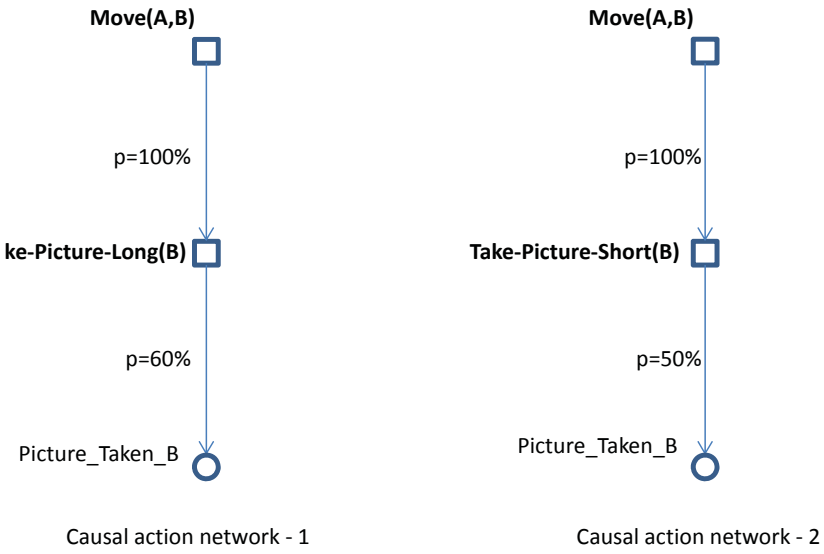


Figure 4.2: Constructing the causal action networks.

In Figure 4.2, two action graphs are extracted for the reward Picture_Taken_B. These two graphs represent the two different ways to achieve Picture_Taken_B. Each action has two outcomes. One represents the outcome where the desired proposition is achieved. This outcome is called the *success outcome*. The other outcome represents the rest of all possible cases and it is called the *failure outcome*. These action graphs consist of only actions and represent the causal relations between the actions. We call them *causal action networks*.

Figure 4.3 shows the full expansion of all possible scenarios of action execution following each causal action network. The actions are labeled with the time and resource levels available after their execution. When an action fails, the same action can be repeated if there are sufficient time and resources to support it. For each fully expanded execution graph, the probabilities of all success scenarios are added together to find the total success probability of the corresponding causal action network. In Figure 4.3, graph 1 has 60% as the success probability, graph 2 has 75%. At the end, these causal action network level success probabilities are combined to find the final probability of achieving the reward Picture_Taken_B. Because we want to have an upper bound of the final success probability so that we can keep the heuristic function admissible, we relax the resource restriction and assume that both of causal action networks can be executed at the same time and have full access to resources. Therefore, we use the addition law of probability and calculate the final probability as $60\% + (1 - 60\%) \times 75\% = 90\%$.

In the above example, the actions are simple as they only have one precondition. Next, we show a more complicated example where actions have more than one precondition. We introduce a new action called *Drill* which needs to be executed before a Collect_Sample

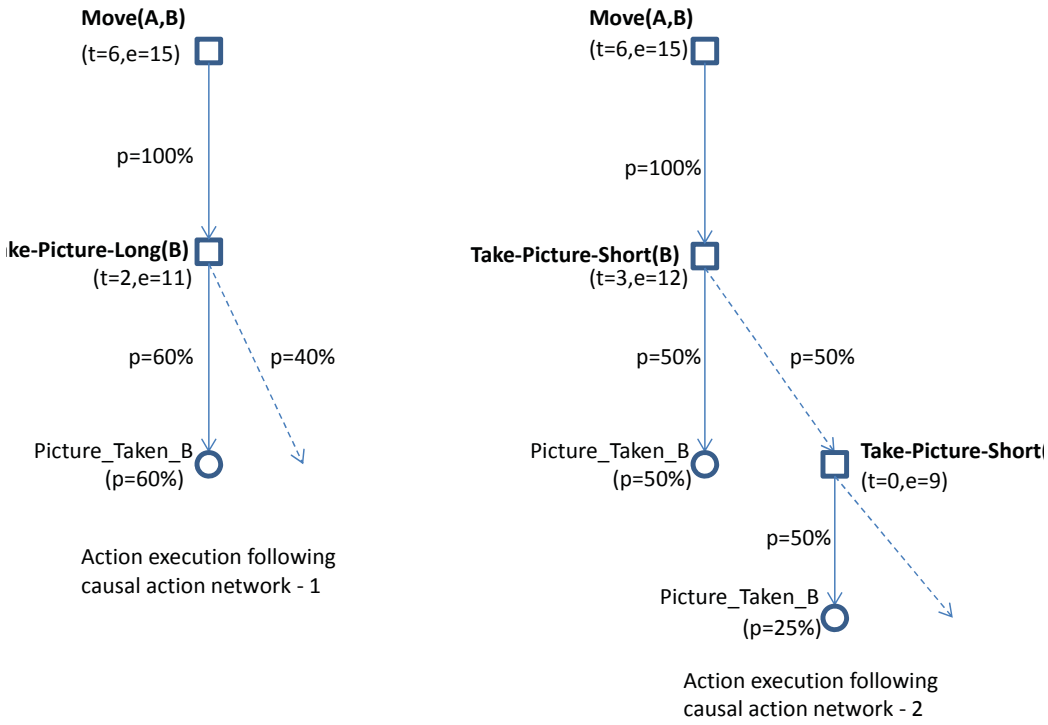


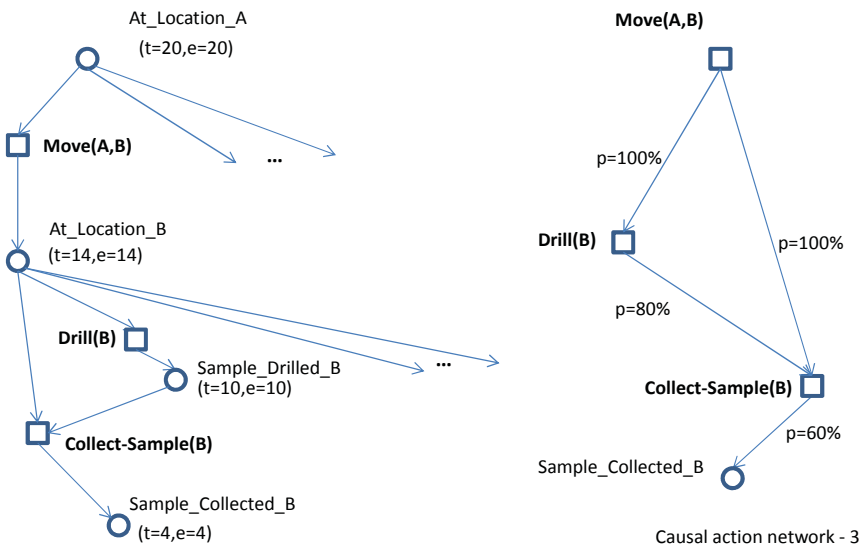
Figure 4.3: Generating all execution scenarios.

action can be executed. Figure 4.4 shows a portion of the relaxed planning graph expanded for reward Sample_Collected_B. When there are multiple preconditions, the minimum time and resource values of the preconditions are taken to calculate the time and resource levels after applying the action.

Figure 4.4 (b) shows the causal action network to achieve goal Sample_Collected_B.

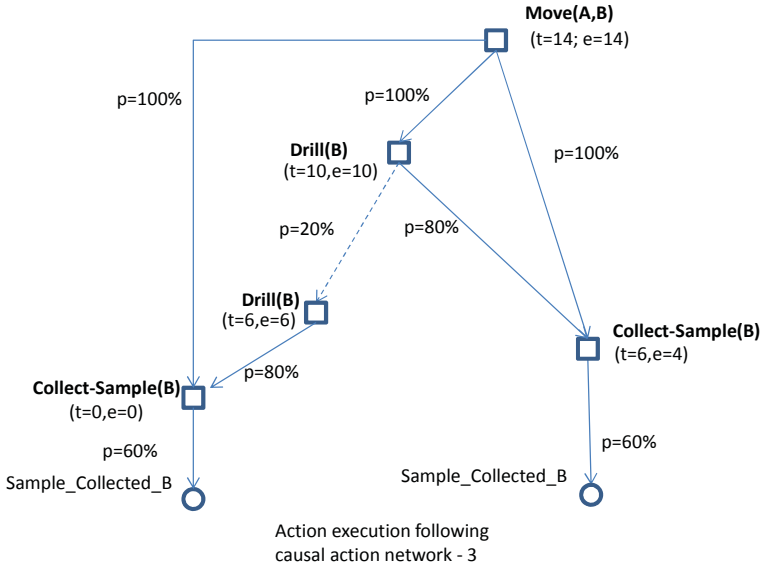
Figure 4.4 (c) is the execution graph following this causal action network. When one action is supported by multiple parent actions, the minimum success probability of the parent actions is taken as the probability to calculate the scenario success probability. For example, the scenario represented by the left branch of the execution graph has success probability of 20% $\times$ 80% $\times$ 60% =9.6%. The scenario represented by the right branch of the execution graph has success probability of 80% $\times$ 60% =48%.

In the next Chapter, we present the experimental evaluation of CPOAO* and related planners.



(a) Relaxed planning graph expanded for Sample_Collected_B

(b) Constructing the casual action network



(c) Generating the execution scenarios

Figure 4.4: Example with actions that contain more than one precondition

Chapter 5

Empirical Evaluation

We designed and conducted three sets of experiments to evaluate the CPOAO* algorithm and heuristics. In the first set, we tested the strengths and limitations of CPOAO* using different heuristic functions across multiple domains. We designed three planning domains and several planning problems. For each domain, we designed a set of problems that are easily solved and gradually increased the problem complexity by adding more propositions and actions as well as raising the time and resource limits until the algorithm is not able to produce the optimal plan within 5 hours. This set of tests allows us to see how each heuristic function performs and the scalability of CPOAO* in general.

In the second set of experiments, we show the results of an additional pruning technique which can be applied when time is the only constraint. The performance of using this technique is compared to the performance of not using it. We demonstrate that this technique can greatly improve the efficiency of the planner.

In the last set of experiments, we examine two recent planners that solve planning problems similar to CPOAO*. These planners are ActuPlan (Actions Concurrency and Time Uncertainty Planner) and CPTP (Concurrent Probabilistic Temporal Planning) described in Sections 2.3 and 2.4, respectively. We describe the similarities and differences between these two planners and CPOAO*. We show the results of running them with domains and problems that are adapted from the CPOAO* tests. The objective of these experiments is to see how these two planners scale in domains which have similar structures in terms of actions and state variables. Note that the purpose is not to compare the performance of the planners because the problems they solve are not exactly the same as explained in Section 5.3.

5.1 Running CPOAO* in multiple domains

We tested CPOAO* across several different planning domains including the Mars Rover domain that we have described in the previous chapters. All of these domains have been used as the benchmark problems in the experiments done for other planners and have been coded originally in PPDDL (Probabilistic Planning Domain Definition Language) [64, 65]. We extended PPDDL to include time, resources, and numeric reward values for goal conditions. We coded three domains as explained below, and shown in Table 5.1.

In the Mars Rover domain, a Mars rover navigates a network of locations to perform tasks such as taking pictures or collect soil samples. The task of taking a picture can be achieved by two different actions with different costs and success probabilities. The locations to take pictures or collect soil samples are specified in the planning problem as goals. The rover wants to receive the maximum total reward given the limited time and resources. This domain involves both map navigation and early finish concurrency.

In the Machine Shop domain, there are multiple machines which have different capabilities such as lathe, polish, smooth, and paint. One machine may have one or more capabilities. The goal is to work a group of raw material pieces into finished parts which are shaped, polished, smoothed and painted. The agent needs to move parts around to maximize the utilization of machines. This domain encapsulates a scheduling problem since one machine cannot work on multiple pieces at the same time. Many real world planning and scheduling problems bear similarities to this domain.

In the File World domain, there are multiple agents trying to organize files, namely, putting files into correct folders. An agent can only perform one task at a time, so the agents need to cooperate with each other to get the job done. For example, one agent can get the file and recognize the file type while another agent opens the folder. We include this domain as an example of a multi-agent planning problem. The coding of each domain is shown in Appendix A.

In Table 5.1 we show the planning problems coded in each domain. The problem features include the number of ground propositions, the number of ground actions, the time limit and the resource limit. All these factors contribute to the complexity of the problem. The number of ground propositions, the time limit and the resource limit determine the size of the state space. The number of ground actions control the number of branches coming from each non-terminal state. In the last column of the table, we include the number of states in the final optimal plan because it gives a hint on the difficulty of the problem.

Table 5.1
Problem features (MR: Mars Rover, MS: Machine Shop, FW: File World)

Problem	Actions	Propositions	Time Limit	Resource Limit	States
MR 1-1	36	35	20	25	17
MR 1-2			25	30	30
MR 1-3			30	35	42
MR 1-4			35	40	89
MR 2-1	65	60	20	25	24
MR 2-2			25	30	43
MR 2-3			30	35	56
MR 2-4			35	40	101
MR 3-1	126	112	20	25	24
MR 3-2			25	30	43
MR 3-3			30	35	63
MR 3-4			35	40	109
MR 4-1	207	180	20	25	24
MR 4-2			25	30	43
MR 4-3			30	35	63
MR 4-4			35	40	109
MS 1-1	22	18	11	13	9
MS 1-2			12	14	9
MS 1-3			13	15	18
MS 1-4			14	16	22
MS 2-1	42	26	11	13	9
MS 2-2			12	14	9
MS 2-3			13	15	18
MS 2-4			14	16	22
MS 3-1	62	34	11	13	9
MS 3-2			12	14	9
MS 3-3			13	15	18
MS 3-4			14	16	22
MS 4-1	82	42	11	13	9
MS 4-2			12	14	9
MS 4-3			13	15	18
MS 4-4			14	16	22
FW 1-1	34	28	8	12	15
FW 1-2			12	16	61
FW 1-3			16	20	142
FW 1-4			20	24	298
FW 2-1	42	35	8	12	15
FW 2-2			12	16	61
FW 2-3			16	20	142
FW 2-4			20	24	316
FW 3-1	50	42	8	12	15
FW 3-2			12	16	61
FW 3-3			16	20	142
FW 3-4			20	24	316
FW 4-1	58	49	8	12	15
FW 4-2			12	16	61
FW 4-3			16	20	142
FW 4-4			20	24	316

We ran each problem using three heuristic functions: Time Only Reachability Test (TORT), Time Only Forward Relax (TOFR), and All Resource Forward Relax (ARFR). We included a baseline heuristic function called "No heuristic", which always returns the sum of all rewards as the estimate of the total achievable rewards. The Reachability Test heuristic was initially designed to support both time and resources. However, during testing we found that including resources in the calculations does not provide much information gain. This is because allowing the resources to be reused when they are not on the same critical path provides estimates that are too far away from the actual resource requirement. Therefore, we used Reachability Test as a "time-only" heuristic in our experiments.

In Tables 5.2 and 5.3 we show the elapsed running time and the number of states generated for each planning problem. The number of states expanded for each problem is shown in Appendix B. From the experimental results, we can see that All Resources Forward Relax (ARFR) always has much fewer number of expanded states compared to Time Only Reachability Test (TORT) and Time Only Forward Relax (TOFR). This shows that this heuristic function indeed provides more informative guidance to the search. On the other hand, All Resources Forward Relax (ARFR) employs the most complicated calculations so it may take more time to find the results in some cases. In particular, the complexity of All Resource Forward Relax increases exponentially when the number of actions and the number of propositions increase. When there are a large number of actions and propositions, All Resource Forward Relax takes longer to calculate the heuristic value. This is the reason All Resources Forward Relax is taking more time to solve the problems than other heuristic functions in the Mars Rover domain. In the Machine Shop and File World domains, it performs better than other heuristic functions because the number of actions or the number propositions are much fewer. The elapsed time of the two hardest problems from each domain are shown in Figure 5.1.

In Table 5.4 we show the pruning ratio for each heuristic taking "No heuristic" as the baseline. The results are in agreement with the results shown in Tables 5.2 and 5.3: All Resources Forward Relax (ARFR) consistently has a much higher ratio of pruned states. In our implementation, we use an artificial action called 'Do Nothing' to generate terminal states where working graph expansion stops even though time and resource may still be available. When calculating the ratio of states being pruned, these terminal states should be excluded because they would never need to be expanded. Since every expanded state has such a corresponding terminal state, the pruning ratio is calculated as shown below:

$$\text{Ratio of state pruning} = \frac{\text{(number of states generated - } 2 \times \text{ number of states expanded)}}{\text{(number of states generated - number of states expanded)}}$$

Table 5.2
Elapsed running time

Problem	No heuristic	TORT heuristic	TOFR heuristic	ARFR heuristic
MR 1-1	1s	$<$ 1s	$<$ 1s	$<$ 1s
MR 1-2	1s	1s	$<$ 1s	1s
MR 1-3	8s	2s	2s	6s
MR 1-4	57s	9s	17s	37s
MR 2-1	$<$ 1s	$<$ 1s	$<$ 1s	1s
MR 2-2	6s	3s	4s	10s
MR 2-3	1min	41s	1min	2min
MR 2-4	17min	9min	15min	13min
MR 3-1	1s	$<$ 1s	$<$ 1s	4s
MR 3-2	10s	8s	11s	33s
MR 3-3	2min	2min	3min	5min
MR 3-4	36min	28min	49min	48min
MR 4-1	1s	1s	1s	5s
MR 4-2	11s	15s	19s	55s
MR 4-3	3min	4min	6min	9min
MR 4-4	65min	50min	89min	83min
MS 1-1	$<$ 1s	$<$ 1s	$<$ 1s	$<$ 1s
MS 1-2	$<$ 1s	$<$ 1s	$<$ 1s	$<$ 1s
MS 1-3	1s	$<$ 1s	$<$ 1s	$<$ 1s
MS 1-4	1s	$<$ 1s	$<$ 1s	$<$ 1s
MS 2-1	7s	2s	1s	$<$ 1s
MS 2-2	25s	5s	4s	2s
MS 2-3	2min	28s	22s	8s
MS 2-4	5min	59s	41s	11s
MS 3-1	3min	38s	35s	7s
MS 3-2	10min	3min	3min	33s
MS 3-3	35min	12min	11min	3min
MS 3-4	>300min	35min	28min	5min
MS 4-1	21min	5min	4min	28s
MS 4-2	224min	20min	20min	3min
MS 4-3	$>$ 300min	$>$ 300min	293min	16min
MS 4-4	$>$ 300min	$>$ 300min	$>$ 300min	53min
FW 1-1	$<$ 1s	$<$ 1s	$<$ 1s	$<$ 1s
FW 1-2	5s	3s	3s	1s
FW 1-3	1min	49s	2min	19s
FW 1-4	12min	5min	26min	2min
FW 2-1	1s	$<$ 1s	1s	1s
FW 2-2	2min	16s	7s	4s
FW 2-3	16min	2min	3min	2min
FW 2-4	19min	17min	62min	7min
FW 3-1	1s	$<$ 1s	1s	1s
FW 3-2	13s	9s	14s	6s
FW 3-3	3min	3min	7min	1min
FW 3-4	155min	35min	141min	13min
FW 4-1	2s	$<$ 1s	1s	1s
FW 4-2	18s	16s	24s	11s
FW 4-3	5min	4min	12min	2min
FW 4-4	$>$ 300min	72min	273min	23min

Table 5.3
Generated states

Problem	No heuristic	TORT heuristic	TOFR heuristic	ARFR heuristic
MR 1-1	3968	747	606	982
MR 1-2	31294	6579	3754	5427
MR 1-3	242632	33038	14304	19088
MR 1-4	1841042	139723	115520	113316
MR 2-1	13426	2539	2211	2486
MR 2-2	163514	37398	27323	20948
MR 2-3	1806442	420000	306554	156877
MR 2-4	19039098	4856299	3489036	1203673
MR 3-1	18064	2943	2491	3372
MR 3-2	243542	47669	37709	30844
MR 3-3	2964758	627014	527766	298111
MR 3-4	34577730	8469617	6620039	2655556
MR 4-1	18558	2951	2494	3433
MR 4-2	255414	48139	38157	31946
MR 4-3	3159176	641686	547198	322224
MR 4-4	37371910	8766900	7158058	2993419
MS 1-1	2929	291	187	101
MS 1-2	4993	766	367	161
MS 1-3	10247	1268	847	287
MS 1-4	19019	1943	1429	537
MS 2-1	133418	19009	13570	1688
MS 2-2	343858	60571	35591	6309
MS 2-3	1894714	390905	193475	36254
MS 2-4	5121026	678101	288569	46635
MS 3-1	3920094	439313	280324	20394
MS 3-2	13073814	1707567	1137302	106634
MS 3-3	42987842	7631263	4721078	494212
MS 3-4	>49282591	20730732	10075700	976828
MS 4-1	24027806	2407880	1626192	66515
MS 4-2	91340862	10567139	7630804	467355
MS 4-3	$>$ 111042556	54829104	33709944	2741289
MS 4-4	$>$ 119343772	$>$ 87852206	$>$ 69720903	5909975
FW 1-1	8858	884	884	1382
FW 1-2	100886	21922	13640	3089
FW 1-3	1062098	299558	248652	36388
FW 1-4	11141318	2941568	2914577	155552
FW 2-1	13274	1478	1478	2114
FW 2-2	163238	42908	27647	5549
FW 2-3	1824686	759555	458997	59879
FW 2-4	20650178	8100935	6657556	457747
FW 3-1	18626	2180	2180	2990
FW 3-2	243734	67124	52112	11903
FW 3-3	2860754	1217646	1019828	90275
FW 3-4	34757366	14031529	13774160	972588
FW 4-1	24914	2990	2990	4010
FW 4-2	343670	104690	91025	15955
FW 4-3	4203350	1865121	1615572	133925
FW 4-4	>36157627	23712246	22927210	1504605

Table 5.4
Pruning ratio

Problem	TORT heuristic	TOFR heuristic	ARFR heuristic
MR 1-1	0.71	0.71	0.71
MR 1-2	0.66	0.68	0.73
MR 1-3	0.64	0.69	0.73
MR 1-4	0.67	0.69	0.74
MR 2-1	0.69	0.72	0.78
MR 2-2	0.64	0.64	0.81
MR 2-3	0.58	0.59	0.82
MR 2-4	0.51	0.55	0.82
MR 3-1	0.72	0.75	0.80
MR 3-2	0.67	0.69	0.82
MR 3-3	0.61	0.62	0.83
MR 3-4	0.55	0.58	0.84
MR 4-1	0.72	0.75	0.80
MR 4-2	0.68	0.69	0.82
MR 4-3	0.62	0.63	0.83
MR 4-4	0.56	0.59	0.83
MS 1-1	0.68	0.71	0.78
MS 1-2	0.64	0.70	0.78
MS 1-3	0.64	0.67	0.77
MS 1-4	0.64	0.66	0.8
MS 2-1	0.65	0.72	0.92
MS 2-2	0.63	0.70	0.90
MS 2-3	0.61	0.67	0.91
MS 2-4	0.62	0.71	0.92
MS 3-1	0.73	0.82	0.95
MS 3-2	0.72	0.79	0.95
MS 3-3	0.69	0.76	0.95
MS 3-4	0.69	0.76	0.96
MS 4-1	0.79	0.87	0.97
MS 4-2	0.77	0.82	0.97
MS 4-3	0.74	0.81	0.97
MS 4-4	0.88	0.89	0.99
FW 1-1	0.5	0.5	0.92
FW 1-2	0.73	0.79	0.93
FW 1-3	0.45	0.67	0.91
FW 1-4	0.27	0.64	0.93
FW 2-1	0.53	0.53	0.93
FW 2-2	0.73	0.79	0.95
FW 2-3	0.49	0.64	0.93
FW 2-4	0.32	0.58	0.94
FW 3-1	0.58	0.58	0.94
FW 3-2	0.74	0.77	0.96
FW 3-3	0.47	0.59	0.94
FW 3-4	0.31	0.53	0.94
FW 4-1	0.61	0.61	0.94
FW 4-2	0.74	0.79	0.96
FW 4-3	0.48	0.58	0.94
FW 4-4	0.31	0.52	0.94

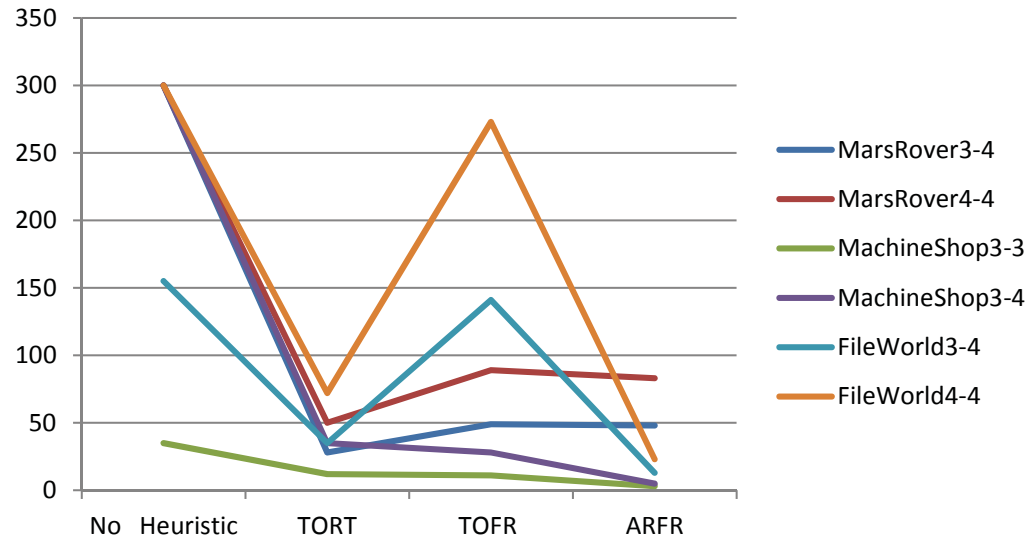


Figure 5.1: Elapsed Time

As the problems are getting harder when more time and resources are allowed, All Resources Forward Relax scales better than the other heuristic functions. A sample of this is shown in Figure 5.2 using the hardest problems in the Mars Rover domain. The x-axis shows a series of problems sorted from easiest to the hardest. The y-axis shows the increase ratio of elapsed time compared to the immediate preceding problem in the series. In almost similar graphs, All Resources Forward Relax (ARFR) has the lowest increase ratio. Therefore, we believe that All Resources Forward Relax will provide better trade-offs for harder problems and hence give better overall performance.

In the Machine Shop domain, to make the problem more complex, we added more actions and propositions into the domain. We noticed that these new actions are not being used in the final optimal plan, i.e., there is no change on the number of states in the optimal plan. However, the elapsed time and the number of generated state increase dramatically because the planner still needs to explore these new options and rule them out. It might be beneficial if we have a pre-planning phase to rule out the actions that will never contribute to achieving any goal. This will greatly improve the planner performance because the number of actions is a primary factor of problem complexity. We will consider this as our future research.

Overall, in all 3 domains tested, the planning complexity and elapsed time grow exponentially as the number of actions, the number of propositions, the resource and time limit increase. This exponential increase is hard to avoid because the planning problem is NP-hard. Thus, the heuristic functions plays a very important role to provide guidance in the planning process. Therefore, we believe the design of an efficient and informative

heuristic functions should be a focus of future research.

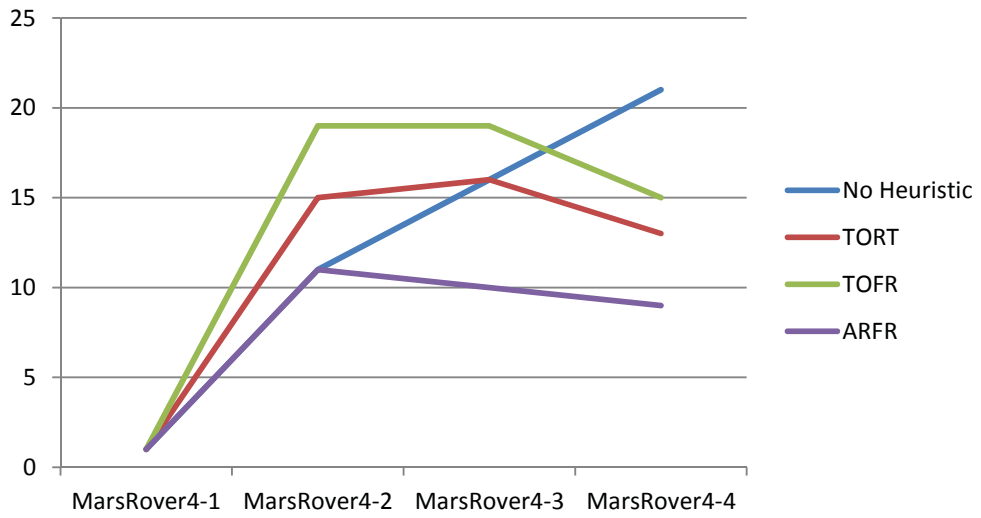


Figure 5.2: Mars Rover 4 Increase Ratio of Elapsed Time

5.2 Pruning Technique for Time Only problems

When time is the only restricting condition, we can apply an additional pruning technique to improve the planner's performance. We used the CAS pruning rules described in Section 4.1 to remove the CASs whose effects are covered by other CASs. We tested this technique by running the hardest problems from the 4 subgroups of each domain. In Table 5.5 we show the effect of CAS pruning on top the Time Only Forward Relax (TOFR) heuristic.

The experimental results show that the pruning technque greatly improves the performence. In the Mars Rover and Machine Shop domains where the number of compatible actions are higher, the improvements are more than 100 fold in terms of both elapsed time and total number of generated states. For problem MarsRover2-4, CPOAO* with pruning takes 23 seconds to solve the problem and generates 108,294 states in total. When pruning is disabled, it takes 281 minutes and has to generate 122,848,455 states to solve the same problem. Similar results are found for problem MachineShop3-2. Elapsed time is shortened from 269 minutes to 1 minute and the number of total generated states is reduced from 134,841,177 to 565,472. In File World domain, there are fewer actions that can be executed concurrently. Therefore, the improvement is less significant than the other two domains. However, the elapsed time and total number of generated states are still cut in half.

Table 5.5
CAS Pruning

	Without CAS pruning		With CAS pruning	
Problem	Elapsed Time	States Generated	Elapsed Time	States Generated
MR 1-4	8s	56334	2s	7306
MR 2-4	281min	122848455	23s	108294
MR 3-4	$>$ 300min	$>$ 34998365	2min	198134
MR 4-4	$>$ 300min	$>$ 30234565	3min	238713
MS 1-2	$<$ 1s	1319	$<$ 1s	314
MS 2-2	3min	2095500	1s	20113
MS 3-2	269min	134841177	1min	565472
MS 4-2	$>$ 300min	$>$ 107462155	22min	7872300
FW 1-2	10s	56905	6s	35222
FW 2-2	33s	184847	16s	82597
FW 3-2	1min	343575	36s	166685
FW 4-2	3min	826417	2min	475005

5.3 Comparison to Other Planners

CPTP and ActuPlan are two recent planners that can generate plans with concurrent actions. Each planner solves a slightly different category of problems. CPTP considers uncertainty in both action effects and action durations. It models the planning problem as an MDP (Markov Decision Process) and uses RTDP (Real Time Dynamic Programming) trials to iteratively update the state values until they converge. ActuPlan focuses on the uncertainty of action duration and uses a continuous time mode. The dependency relationships between the actions are represented by a series of random variables linked in a Bayesian network. We show a comparison of these two planners and CPOAO* in Table 5.6.

Similar to CPOAO*, CPTP solves planning problems with probabilistic action effects. An action can have multiple effects and each effect has an associated probability. The total probability is always 1. In ActuPlan, the action effects are deterministic i.e., it is certain that the effects will happen at the end of the action. As for action duration, CPTP also considers the uncertainty on action duration. Different than CPOAO, in which the time an action takes to complete is a constant number, CPTP support several probabilistic time models such as normal distribution and uniform distribution. CPTP uses these time models to calculate the probability of completing the action at a time point. For example, if an action's time specification is (uniform 6 9), then this action may take 6, 7, 8, or 9 time units to complete and each outcome has a probability of 25%. In CPTP, new states are added

Table 5.6
Comparison to other planners

	CPTP	ActuPlan	CPOAO
Action effects	Probabilistic	Deterministic	Probabilistic
Action duration	Discrete probabilistic	Continuous probabilistic	Discrete deterministic
Resource consumption	Not considered	Not considered	Resource constrained
Concurrency	All Finish	All Finish	All finish and arly finish
Main Algorithm	Sampled RTDP	Forward chaining search	AO*
Heuristic guided	Yes	Yes	Yes

to represent each possible case of action duration. ActuPlan also considers uncertainty on the action duration and uses a similar denotation such as (uniform 10 15) or (normal 6 1) for duration specification. But unlike CPTP, ActuPlan does not instantiate time as discrete time points and time is continuous. ActuPlan uses a set of random variables to represent the action durations and action start or end events. All these random variables are connected in a Bayesian network. When needed, this Bayesian network is queried to calculate the probability of any event. Both CPTP and ActuPlan only consider time as a cost factor in the search for optimal plan. In CPOAO*, in addition to the time constraint, an unlimited number of resource types are supported.

All three planners support concurrent actions. One difference between CPOAO* and the other two planners is CPOAO* also allows the action abortion. In the case of "Early Finish" parallelism an action can be aborted in the middle if continuing to execute the action does not add any value based on the outcomes of other actions. In CPTP and ActuPlan, all actions have to continue to run until they finish regardless of the outcomes of the other actions.

All three planners search in state space starting from the initial state and going forward. Also, all have heuristic functions to guide the search. CPTP uses a dynamic programming algorithm called RTDP (Real Time Dynamic Programming). Each RTDP trial starts from the initial state and simulates action execution until either a dead end is reached or all goal conditions are satisfied. At each decision point, CPTP selects the best action combination according to the values of the children states. When the action has more than one children state, one is randomly picked. The values of the states are updated in the trial and CPTP repeats the trials until all the state values in the plan converge. ActuPlan performs a depth first search and backtracks if the probability of reaching a goal state before the deadline does not meet the threshold or the plan found is not optimal. ActuPlan uses a heuristic

Table 5.7
Experiment results with CPTP

Problem name	Domain specification	Problem specification	States generated	Elapsed time (sec.)
MarsRover-CPTP-p1	5 lifted actions 6 predicates	2 locations 2 soil targets 2 picture targets	3590	95
MarsRover-CPTP-p2	5 lifted actions 6 predicates	3 locations 3 soil targets 3 picture targets	11224	148
MarsRover-CPTP-p3	5 lifted actions 6 predicates	4 locations 4 soil targets 4 picture targets	74526	499
MarsRover-CPTP-p4	5 lifted actions 6 predicates	5 locations 5 soil targets 5 picture targets	not solved	not solved
MachineShop-CPTP-p1	6 lifted actions 8 predicates	1 piece 2 machines	308	15
MachineShop-CPTP-p2	6 lifted actions 8 predicates	2 pieces 2 machines	13616	64
MachineShop-CPTP-p3	6 lifted actions 8 predicates	3 pieces 2 machines	390885	2016
MachineShop-CPTP-p4	6 lifted actions 8 predicates	4 pieces 2 machines	not solved	not solved
FileWorld-CPTP-p1	13 lifted actions 16 predicates	2 files 2 agents	23286	54
FileWorld-CPTP-p2	13 lifted actions 16 predicates	3 files 2 agents	167916	368
FileWorld-CPTP-p3	13 lifted actions 16 predicates	4 files 2 agents	769371	3979
FileWorld-CPTP-p4	13 lifted actions 16 predicates	5 files 2 agents	not solved	not solved

function to select the best action to apply.

To calculate the heuristic value, CPTP solves a relaxed CoMDP problem in which the information about unfinished actions are ignored. It still tracks when the action effects will become true but the mutexes between the new action and the unfinished actions do not need to be considered. ActuPlan uses a Relaxed GraphPlan (RGP) based heuristic function which is adapted from Fast Forward (FF) planner. In this heuristic function, the "delete" effects are ignored. The scalar values calculated from action duration probabilities are used as expected action durations. Same as CPOAO*, both heuristic functions are admissible.

Table 5.8
Experiment results with ActuPlan

Problem name	Domain specification	Problem specification	States generated	Elapsed time (sec.)
MarsRover-Actu-p1	4 lifted actions 6 predicates	5 locations 2 soil targets 3 picture targets	236	0.3
MarsRover-Actu-p2	4 lifted actions 6 predicates	7 locations 4 soil targets 5 picture targets	9387	4.5
MarsRover-Actu-p3	4 lifted actions 6 predicates	8 locations 5 soil targets 6 picture targets	47828	185
MarsRover-Actu-p4	4 lifted actions 6 predicates	9 locations 6 soil targets 7 picture targets	260305	1202
MarsRover-Actu-p5	4 lifted actions 6 predicates	10 locations 7 soil targets 8 picture targets	not solved	not solved
MachineShop-Actu-p1	10 lifted actions 8 predicates	1 piece 2 machines	13	0.2
MachineShop-Actu-p2	10 lifted actions 8 predicates	2 pieces 2 machines	541	0.3
MachineShop-Actu-p3	10 lifted actions 8 predicates	3 pieces 2 machines	68478	113
MachineShop-Actu-p4	10 lifted actions 8 predicates	4 pieces 2 machines	2283521	851
MachineShop-Actu-p5	6 lifted actions 8 predicates	5 pieces 2 machines	not solved	not solved
FileWorld-Actu-p1	15 lifted actions 13 predicates	2 files 2 agents	305	0.2
FileWorld-Actu-p2	15 lifted actions 13 predicates	4 files 2 agents	9674	0.9
FileWorld-Actu-p3	15 lifted actions 13 predicates	6 files 2 agents	90569	11
FileWorld-Actu-p4	15 lifted actions 13 predicates	8 files 2 agents	729933	121
FileWorld-Actu-p5	15 lifted actions 13 predicates	10 files 2 agents	1305722	not solved

We performed experiments with ActuPlan and CPTP using planning domains similar to the ones that we have used in the CPOAO* experiments. We adapted the planning problems to suit the requirements of these planners but kept most of the domain features. In Table 5.7 and Table 5.8 we show the scalability to see the scalability of these planners. The results are shown in Table 5.7 and Table 5.8. Similar to CPOAO*, we can see that, for both planners, the number of states generated and the time taken to solve the problem grow exponentially as the problem complexity increases linearly. In CPTP's implementation, before the start of the RTDP trials, all the ground actions are instantiated and a transition function is calculated beforehand. This is designed to speed up the generation of new states in the process of RTDP trials. However, at the time this transition function is calculated, there is no information about which states are reachable. Therefore, all the possible ground actions have to be generated resulting in a large number of actions. Calculating the transition function for all the ground actions not only takes time but also consumes a lot of memory. In our experiments, many problems were not solvable because the system ran out of memory in the phase of generating the transition function for all ground actions. This problem can be avoided by generating the transition function just in time, creating it when the action is actually being applied.

ActuPlan can run in both conformant mode and contingent mode. Also it can run with or without the deadlines assigned to the goal conditions. In our experiments, we ran it in contingent mode with deadlines, which we believe is the most complicated scenario for ActuPlan. ActuPlan needs to generate a large number of random variables to track the dependency relationships between the actions and events, occasionally causing memory issues. Some of our testing failed due to memory limitations. In all three planners including CPOAO*, the difficulty of planning problem grows exponentially with respect to the complexity of the planning problem definition. In many cases, adding one more object into the problem adds 10 times as much time to solve the problem or renders the problem not solvable within a reasonable time. This is expected because all three planners deal with concurrency and uncertainty. Making a small change to the domain or problem can cause a big change on the search space. Thus, having an efficient and informative heuristic function is very important in curbing the explosion of the search space and it should be considered as one of the main directions of future research.

Conclusion

In this dissertation, we presented CPOAO*, a model and framework for plan generation in concurrent, probabilistic, and oversubscribed domains with durative actions. The main contributions of this thesis are threefold. First, we designed and implemented an AO* search based planner that can find plans with early-finish and all-finish parallelism. Second, we explored the notion of interruptible actions. Third, we developed and evaluated several domain independent heuristics that can work with temporal and resource constraints. Our research improves artificial intelligence planning research as evidenced by the empirical evaluation we presented.

CPOAO* generates plans with concurrent actions by considering action combinations rather than individual actions at each choice point of the search. The duration of each action combination is the duration of the shortest action in the action combination. The states are augmented to include the unfinished actions together with their remaining executing time. At the end of an action combination, all the unfinished actions are put into the set of unfinished actions of the resulting states. The best action combination is chosen from the set of applicable actions and the set of unfinished actions of the current state. This gives the planner the ability to abort ongoing actions if necessary. CPOAO* addresses both the early-finish and all-finish cases of the concurrent actions.

The difficulty of concurrent planning lies in the exponential number of action combinations that are applicable at each decision point. We provide a pruning technique that can decrease the number of applicable action combinations and several heuristics that decrease the number of states explored. The main factor that determines the performance of a heuristic function is the trade-off between the time spent on calculating the heuristic value and the accuracy of the heuristic value. Among the heuristic functions we developed, Time Only Reachability Test is fast but is less informative. Forward Relax heuristics give better estimates but require more time to calculate. When time is the only constraint, running CPOAO* with CAS pruning and Time Only Forward Relax together provides the best performance. When both time and resource constraints need to be respected, the time horizon and the number of actions are the two important factors affecting the performance of the heuristic function. In general, when the time horizon is short and the number of

ground actions is high, it is better to use the Reachability Test heuristic to avoid expensive heuristic value calculation. On the other hand, for problems with long horizons or fewer actions, Forward Relax heuristics are a better choice because they can prune more states early on and save time in the long run.

Another factor to consider in plan generation is the memory usage. CPOAO* running with Reachability Test needs to generate more states and requires more memory as a result. For complex problems, running with Forward Relax heuristics requires less memory. Between the two Forward Relax heuristic functions, when resources are indeed the bottleneck constraints, Forward Relax Time and Resource perform better than Forward Relax Time Only as shown by our experiments.

As part of future work, the CPOAO* framework can benefit from a pre-processing phase that rules out the actions that will never contribute to achieving any goal. This can greatly improve the planner performance because the number of actions is a primary factor of problem complexity. This dissertation work can be extended by representing and reasoning with actions that have continuous durations and resources. The notion of interruptible actions can be integrated with a plan execution and monitoring framework to monitor changes in the environment and adapt the plan in response.

The problem of plan existence for deterministic domains is PSPACE-complete [66] whereas the extensions to probabilistic domains are EXP-complete for full observability [67] and 2-EXP-complete for partial observability [68]. Therefore, heuristic functions play a very important role to provide guidance in the search process. Consequently, a crucial focus of future research is the design and development of efficient and informative heuristic functions.

References

[1] M. Ghallab, D. Nau, and P. Traverso, *Automated Planning: Theory and Practice*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2004.

[2] D. S. Weld, "An introduction to least commitment planning," *AI Magazine*, vol. 15, no. 4, pp. 27–61, 1994.

[3] D. S. Weld, "Recent advances in AI planning," *AI Magazine*, vol. 20, no. 2, pp. 93–123, 1999.

[4] D. S. Nau, "Current trends in automated planning," *AI Magazine*, vol. 28, no. 4, pp. 43–58, 2007.

[5] U. Visser and P. Doherty, "Issues in designing physical agents for dynamic real-time environments world: Modeling, planning, learning, and communicating," *AI Magazine*, vol. 25, no. 2, pp. 137–138, 2004.

[6] D. Simchi-Levi, P. Kaminsky, and E. Simchi-Levi, *Designing And Managing The Supply Chain*. McGraw-Hill, 2007.

[7] K. Sampath, A. Tezabwala, A. Chabrier, J. Payne, and F. Tiozzo, "Integrated operations (re-)scheduling from mine to ship," in *Proceedings of the 23rd International Conference on Automated Planning and Scheduling (ICAPS-2013)*, pp. 416–424, 2013.

[8] C. Larman and V. R. Basili, "Iterative and incremental development: A brief history," *IEEE Computer*, vol. 36, no. 3, pp. 47–58, 2003.

[9] R. Castano, T. Estlin, R. C. Anderson, D. M. Gaines, A. Castano, B. Bornstein, C. Chouinard, and M. Judd, "Oasis: Onboard autonomous science investigation system for opportunistic rover science," *Journal of Field Robotics, Special Issue on Space Robotics, Part III*, vol. 24, no. 5, pp. 379–397, 2007.

[10] T. Estlin, R. Castano, R. C. Anderson, D. Gaines, F. Fisher, and M. Judd, "Learning and planning for Mars rover science," in *Proceedings of the International Joint*

Conference on Artificial Intelligence (IJCAI) Workshop on Issues in Designing Physical Agents for Dynamic Real-Time Environments: World Modeling, Planning, Learning, and Communicating, 2003.

[11] T. Estlin, D. M. Gaines, C. Chouinard, R. Castano, B. Bornstein, M. Judd, I. Nesnas, and R. C. Anderson, "Increased Mars Rover autonomy using AI planning, scheduling and execution," *IEEE International Conference on Robotics and Automation*, pp. 4911–4918, 2007.

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

[13] B. Bonet and H. Geffner, "GPT: A tool for planning with uncertainty and partial information.," in *International Joint Conference on Artificial Intelligence (IJCAI) Workshop on Planning with Uncertainty and Partial Information*, pp. 82–87, 2001.

[14] B. Bonet and H. Geffner, "Labeled RTDP: Improving the convergence of real-time dynamic programming," in *Proceedings of the 13th International Conference on Automated Planning and Scheduling (ICAPS-2003)*, pp. 12–21, 2003.

[15] B. Bonet and H. Geffner, "mGPT: A probabilistic planner based on heuristic search," *Journal of Artificial Intelligence Research*, vol. 24, pp. 933–944, 2005.

[16] N. Kushmerick, S. Hanks, and D. Weld, "An algorithm for probabilistic planning," *Artificial Intelligence*, vol. 76, pp. 239–86, 1995.

[17] N. Onder, G. C. Whelan, and L. Li, "Engineering a conformant probabilistic planner," *Journal of Artificial Intelligence Research*, vol. 25, pp. 1–15, 2006.

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

[19] F. Bacchus and M. Ady, "Planning with resources and concurrency: A forward chaining approach," in *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI 2001)*, p. 417, 2001.

[20] Mausam and D. Weld, "Concurrent probabilistic temporal planning," in *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS 2005)*, 2005.

[21] Mausam and D. Weld, "Planning with durative actions in stochastic domains," *Journal of Artificial Intelligence Research*, vol. 31, pp. 33–82, 2008.

[22] I. Little and S. Thiebaux, "Concurrent probabilistic planning in the Graphplan framework," *In Proc. ICAPS-06*, pp. 263–272, 2006.

[23] C. Boutilier, "Planning, learning and coordination in multiagent decision processes," in *Proceedings of the Sixth Conference on Theoretical Aspects of Rationality and Knowledge*, pp. 195–210, 1996.

[24] M. Fox and D. Long, "PDDL2.1: An extension to PDDL for expressing temporal planning domains," *Journal of Artificial Intelligence Research*, vol. 20, pp. 61–124, 2003.

[25] P. Haslum and H. Geffner, "Heuristic planning with time and resources," in *Proceedings of the IJCAI-01 Workshop on Planning with Resources*, 2001.

[26] M. Do and S. Kambhampati, "Sapa: A multi-objective metric temporal planner," *Journal of Artificial Intelligence Research*, vol. 20, pp. 155–194, 2003.

[27] S. Edelkamp, "Taming numbers and duration in the model checking integrated planning system," *Journal of Artificial Intelligence Research*, vol. 20, pp. 195–238, 2003.

[28] F. Bacchus and F. Kabanza, "Taming numbers and duration in the model checking integrated planning system," *Journal of Artificial Intelligence Research*, vol. 20, pp. 195–238, 2003.

[29] J. Baier, F. Bacchus, and S. McIlraith, "A heuristic search approach to planning with temporally extended preferences," in *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI 2007)*, pp. 1808–1815, 2007.

[30] J. Baier and S. McIlraith, "Planning with first-order temporally extended goals using heuristic search," in *Proceedings of the 21st National Conference on Artificial Intelligence (AAAI 2006)*, pp. 788–795, 2006.

[31] J. Baier and S. McIlraith, "Planning with temporally extended goals using heuristic search," in *Proceedings of the 16th International Conference on Automated Planning and Scheduling (ICAPS 2006)*, 2006.

[32] D. E. Smith, "Choosing objectives in over-subscription planning," in *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS 2005)*, 2004.

[33] M. Van Den Briel, R. Sanchez, M. B. Do, and S. Kambhampati, "Effective approaches for partial satisfaction (over-subscription) planning," in *Proceedings of AAAI-04*, pp. 562–569, 2004.

[34] J. Benton, M. B. Do, and S. Kambhampati, "Over-subscription planning with numeric goals," in *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI 2005)*, 2005.

[35] N. Nilsson, "Problem solving methods in artificial intelligence," *McGraw-Hill*, 1971.

[36] N. Nilsson, *Principles of artificial intelligence*. Morgan Kaufmann, 1980.

[37] J. Pearl, *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley, 1984.

[38] P. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Transactions on Systems Science and Cybernetics*, vol. SSC4 4 (2), pp. 100–107, 1968.

[39] S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2002.

[40] N. J. Nilsson, *Artificial Intelligence: A New Synthesis*. Morgan Kaufmann, 1998.

[41] Mausam, E. Benazera, R. Brafman, N. Meuleau, and E. A. Hansen, "Planning with continuous resources in stochastic domains," in *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI 2005)*, 2005.

[42] E. Hansen and S. Zilberstein, "Lao*: A heuristic search algorithm that finds solutions with loops," *Artificial Intelligence*, vol. 129(1-2), pp. 35–62, 2001.

[43] R. Bellman, "A Markovian decision process," *Journal of Mathematics and Mechanics*, vol. 6, 1957.

[44] R. A. Howard, *Dynamic Programming and Markov Processes*. The M.I.T. Press, 1960.

[45] M. L. Puterman, *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons, 1994.

[46] E. Beaudry, F. Kabanza, and F. Michaud, "Planning with concurrency under resources and time uncertainty," in *Proceedings of the European Conference on Artificial Intelligence (ECAI 2010)*, 2010.

[47] E. Beaudry, F. Kabanza, and F. Michaud, "Planning for concurrent action executions under action duration uncertainty using dynamically generated bayesian networks," in *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS 2010)*, 2010.

[48] J. Hoffmann and B. Nebel, "The FF planning system: Fast plan generation through heuristic search," *Journal of Artificial Intelligence Research*, vol. 14, pp. 253–302, 2001.

[49] J. Hoffmann, "The metric-FF planning system: Translating "ignoring delete lists" to numeric state variables," *Journal of Artificial Intelligence Research*, vol. 20, pp. 291–341, 2003.

[50] C. Boutilier, R. Dearden, and M. Goldszmidt, "Exploiting structure in policy construction," *Proceedings of IJCAI-95*, pp. 1104–1111, 1995.

[51] S. H. Craig Boutilier, Thomas Dean, "Decision-theoretic planning: Structural assumptions and computational leverage," *Journal of Artificial Intelligence Research*, vol. 11, pp. 1–94, 1999.

[52] C. Guestrin, D. Kolle, R. Parr, and S. Venkataraman, "Efficient solution algorithms for factored mdps," *Journal of Artificial Intelligence Research*, vol. 19, pp. 399–468, 2003.

[53] A. Raghavan, S. Joshi, A. Fern, P. Tadepalli, and R. Khardon, "Planning in factored action spaces with symbolic dynamic programming," in *Proceedings of the Twenty-Sixth National Conference on Artificial Intelligence (AAAI 2012)*, pp. 1802–1808, 2012.

[54] J. Hoey, R. St-Aubin, A. Hu, and C. Boutilier, "Spudd: Stochastic planning using decision diagrams," in *Proceedings of the 15th Conference on Uncertainty in Artificial Intelligence (UAI 1999)*, pp. 279–288, 1999.

[55] Z. Feng, E. A. Hansen, and S. Zilberstein, "Symbolic generalization for on-line planning," in *Proceedings of the 19th Conference on Uncertainty in Artificial Intelligence (UAI 2003)*, pp. 209–216, 2003.

[56] S. Sanner, K. V. Delgado, and L. N. de Barros, "Symbolic dynamic programming for discrete and continuous state MDPs," in *Proceedings of the 27th Conference on Uncertainty in Artificial Intelligence (UAI 2011)*, (Corvallis, Oregon), pp. 643–652, AUAI Press, 2011.

[57] A. Barto, S. Bradtke, and S. Singh, "Learning to act using real-time dynamic programming," *Artificial Intelligence*, vol. 72, no. 1–2, pp. 81–138, 1995.

[58] Mausam and D. Weld, "Solving concurrent Markov decision processes," in *Proceedings of the Nineteenth National Conference on Artificial Intelligence (AAAI 2004)*, pp. 716–722, 2004.

[59] L. Li and N. Onder, "Generating plans in concurrent, probabilistic, over-subscribed domains," in *Proceedings of the Twenty-Third National Conference on Artificial Intelligence (AAAI 2008)*, pp. 957–962, 2008.

[60] H. L. S. Younes and M. L. Littman, "PPDDL 1.0: An extension to PDDL for expressing planning domains with probabilistic effects," Tech. Rep. CMU-CS-04-167, School of Computer Science, Carnegie Mellon University, 2004.

[61] A. Pfeffer, "Functional specification of probabilistic process models," in *Proceedings of the Twentieth National Conference on Artificial Intelligence (AAAI 2005)*, pp. 663–669, 2005.

[62] A. L. Blum and M. L. Furst, "Fast planning through planning graph analysis," *Artificial Intelligence*, vol. 90, pp. 1636–1642, 1995.

[63] D. Byrce and S. Kambhampati, "A tutorial on planning graph-based reachability heuristics," *AI Magazine*, vol. 28, no. 1.

[64] D. Long and M. Fox, "The 3rd international planning competition: Results and analysis," *Journal of Artificial Intelligence Research*, vol. 20, pp. 1–59, 2003.

[65] M. L. Littman and H. L. S. Younes, "Introduction to the probabilistic planning track," 2004.

[66] T. Bylander, "The computational complexity of propositional STRIPS planning," *Artificial Intelligence*, vol. 69, pp. 165–204, 1994.

[67] M. L. Littman, J. Goldsmith, and t. . M. Mundhenk"

[68] J. Rintanen, "Complexity of planning with partial observability," in *Proceedings of the 14th International Conference on Automated Planning and Scheduling (ICAPS 2004)*, 2004.

Appendix A

CPOAO* Domain and Problem

A.1 Mars Rover Domain

```
(define (domain mars-rover-domain)
        (types location pTgt sTgt)
        (resource-types power)
        (predicates (rover-at location)
                    (picture-site location pTgt)
                    (sample-site location sTgt)
                    (path location location)
                    (shot pTgt)
                    (collected sTgt))

(action move
       (var (location : loc1 loc2))
       (preconditions (path loc1 loc2)
                      (rover-at loc1))
       (time 4)
       (resources (power 4))
       (effects
          ( 1 (add (rover-at loc2))
              (del (rover-at loc1)))))
```

```
(action sample
       (var (location : loc1)
            (sTgt : sTgt1))
       (preconditions (sample-site loc1 sTgt1)
                      (rover-at loc1))
       (time 7)
       (resources (power 7))
       (effects
          ( 0.7 (add (collected sTgt1))
                (del ))
          ( 0.3 (add )
                (del ))))

(action picture-l
       (var (location : loc1)
            (pTgt : pTgt1))
       (preconditions (picture-site loc1 pTgt1)
                      (rover-at loc1))
       (time 6)
       (resources (power 6))
       (effects
          ( 0.6 (add (shot pTgt1))
                (del ))
          ( 0.4 (add )
                (del ))))


(action picture-s
       (var  (location : loc1)
             (pTgt : pTgt1))
       (preconditions (picture-site loc1 pTgt1)
                      (rover-at loc1))
       (time 5)
       (resources (power 5))
       (effects
          ( 0.5 (add (shot pTgt1))
                (del ))
          ( 0.5 (add )
                (del )))))
```

A.2 Mars Rover Problem

```
(define (problem simple-mars-rover)
  (domain mars-rover-domain)
  (objects
      (location : loc-a loc-b loc-c loc-d loc-e)
      (pTgt     : pTgt-1 pTgt-2 pTgt-3)
      (sTgt     : sTgt-1 sTgt-2))
  (init (rover-at loc-a)
        (path loc-a loc-b)
        (path loc-a loc-c)
        (path loc-a loc-d)
        (path loc-a loc-e)
        (path loc-b loc-a)
        (path loc-b loc-c)
        (path loc-b loc-e)
        (path loc-c loc-a)
        (path loc-c loc-b)
        (path loc-c loc-d)
        (path loc-d loc-a)
        (path loc-d loc-c)
        (path loc-d loc-e)
        (path loc-e loc-a)
        (path loc-e loc-b)
        (path loc-e loc-d)
        (picture-site loc-c pTgt-1)
        (picture-site loc-d pTgt-2)
        (picture-site loc-e pTgt-3)
        (sample-site loc-d sTgt-1)
        (sample-site loc-e sTgt-2))

        (time  25)
        (resources    (power 30))
        (goals ((rover-at loc-a) 20)
                ((shot pTgt-1) 5)
                ((shot pTgt-2) 2)
                ((shot pTgt-3) 6)
                ((collected sTgt-1) 3)
                ((collected sTgt-2) 5)))
```

A.3 Machine Shop Domain

```
(define (domain MachineShop)
        (types Piece Machine)
        (resource-types power)
        (predicates (shaped Piece)
                    (painted Piece)
                    (smooth Piece)
                    (polished Piece)
                    (canpolpaint Machine)
                    (canlatroll Machine)
                    (cangrind Machine)
                    (at Piece Machine)
                    (on Piece Machine)
                    (hasimmersion Machine)
                    (free Machine))

(action polish
       (var (Piece : piece-1)
            (Machine : machine-1))
       (preconditions (canpolpaint machine-1)
                      (on piece-1 machine-1))
       (time 7) (resources (power 10))
       (effects
          ( 0.9 (add (polished piece-1))
                (del ))
          ( 0.1 (add )
                (del ))))

(action spraypaint
       (var (Piece : piece-1)
            (Machine : machine-1))
       (preconditions (canpolpaint machine-1)
                       (on piece-1 machine-1))
       (time 8) (resources (power 6))
       (effects
          ( 0.8 (add (painted piece-1))
                (del ))
          ( 0.2 (add )
                (del ))))
```

```
(action immersionpaint
       (var (Piece : piece-1)
            (Machine : machine-1))
       (preconditions (canpolpaint machine-1)
                      (on piece-1 machine-1)
       (hasimmersion machine-1))
       (time 3) (resources (power 4))
       (effects
          ( 0.57 (add (painted piece-1))
                 (del ))
          ( 0.38 (add (painted piece-1))
                 (del (hasimmersion machine-1)))
          ( 0.02 (add )
                 (del (hasimmersion machine-1)))))

(action lathe
       (var (Piece : piece-1)
            (Machine : machine-1))
       (preconditions (canlatroll machine-1)
                      (on piece-1 machine-1))
       (time 5) (resources (power 5))
       (effects
          ( 0.9 (add (shaped piece-1))
                (del (painted piece-1)
                     (smooth piece-1)))
          ( 0.1 (add )
                (del ))))

(action grind
       (var (Piece : piece-1)
            (Machine : machine-1))
       (preconditions (cangrind machine-1)
                      (on piece-1 machine-1))
       (time 4) (resources (power 4))
       (effects
          ( 0.9 (add (smooth piece-1))
                (del ))
          ( 0.1 (add )
                (del ))))
```

```
(action buyimmersion
       (var (Machine : machine-1))
       (preconditions (canpolpaint machine-1))
       (time 3) (resources (power 5))
       (effects
          ( 1 (add (hasimmersion machine-1))
              (del ))))

(action place
       (var (Piece : piece-1)
            (Machine : machine-1))
       (preconditions (at piece-1 machine-1)
                      (free machine-1))
       (time 1) (resources (power 1))
       (effects
          ( 1 (add (on piece-1 machine-1))
              (del (free machine-1)
                   (at piece-1 machine-1)))))

(action move_from_place
       (var (Piece : piece-1)
            (Machine : machine-1 machine-2))
       (preconditions (at piece-1 machine-1))
       (time 3) (resources (power 3))
       (effects
          ( 0.9 (add (at piece-1 machine-2))
                (del (at piece-1 machine-1)))
          ( 0.1 (add )
                (del ))))

(action move_from_machine
       (var (Piece : piece-1)
            (Machine : machine-1 machine-2))
       (preconditions (on piece-1 machine-1))
       (time 3) (resources (power 3))
       (effects
          ( 0.9 (add (at piece-1 machine-2)
                     (free machine-1))
                (del (on piece-1 machine-1)))
          ( 0.1 (add (free machine-1))
                (del (on piece-1 machine-1))))))
```

A.4 Machine Shop Problem

```
(define (problem machine-shop-problem)
        (domain MachineShop)
        (objects
            (Piece : piece-1 piece-2)
            (Machine : machine-1 machine-2))
        (init
          (at piece-1 machine-1)
          (at piece-2 machine-1)
          (canpolpaint machine-1)
          (cangrind machine-2)
          (canlatroll machine-2)
          (free machine-1)
          (free machine-2))
          (time  12)
          (resources    (power 13))
          (goals ((shaped piece-1) 5)
          ((painted piece-1) 5)
          ((shaped piece-2) 5)
          ((painted piece-2) 5)))
```

A.5 File World Domain

```
(define (domain file-world)
        (types file agent true)
        (resource-types power)
        (predicates (A-File file)
                    (B-File file)
                    (C-File file)
                    (A-Folder-open true)
                    (B-Folder-open true)
                    (C-Folder-open true)
                    (A-Folder-closed true)
                    (B-Folder-closed true)
                    (C-Folder-closed true)
                    (Get-Folder-A agent)
                    (Get-Folder-B agent)
                    (Get-Folder-C agent)
                    (Type-Unknown file)
                    (Free agent)
                    (Get-File agent file)
                    (Filed file))

(action get-file-type
        (var (file: file1)
             (agent : agent1))
        (preconditions (Free agent1)
                       (Type-Unknown file1))
        (time = 3)
        (resources (power 4))
        (effects ( 0.4 (add (A-File file1)
                            (Get-File agent1 file1))
                       (del (Type-Unknown file1)
                            (Free agent1)))
                 ( 0.3 (add (B-File file1)
                            (Get-File agent1 file1))
                       (del (Type-Unknown file1)
                            (Free agent1)))
                 ( 0.3 (add (C-File file1)
                            (Get-File agent1 file1))
                       (del (Type-Unknown file1)
                            (Free agent1))))))
```

```
(action open-folder-A
        (var (agent : agent1)
             (true : true1))
        (preconditions (Free agent1)
                       (A-Folder-closed true1))
        (time = 2)
        (resources (power 4))
        (effects ( 0.8 (add (A-Folder-open true1)
                            (Get-Folder-A agent1))
                       (del (Free agent1)
                            (A-Folder-closed true1)))
                 ( 0.2 (add )
                       (del ))))
(action open-folder-B
        (var    (agent : agent1)
                (true : true1))
        (preconditions (Free agent1)
                       (B-Folder-closed true1))
        (time = 2)
        (resources (power 4))
        (effects( 0.8 (add (B-Folder-open true1)
                           (Get-Folder-B agent1))
                      (del (Free agent1)
                           (B-Folder-closed true1)))
                ( 0.2 (add )
                      (del ))))
(action open-folder-C
        (var (agent : agent1)
             (true : true1))
        (preconditions (Free agent1)
                       (C-Folder-closed true1))
        (time = 2)
        (resources (power 4))
        (effects ( 0.8 (add (C-Folder-open true1)
                            (Get-Folder-C agent1))
                       (del (Free agent1)
                            (C-Folder-closed true1)))
                 ( 0.2 (add )
                       (del ))))
```

```
(action close-folder-A
        (var (agent : agent1)
             (true : true1))
        (preconditions (Free agent1)
                       (A-Folder-open true1))
        (time = 2)
        (resources (power 4))
        (effects ( 0.8 (add (A-Folder-closed true1)
                            (Get-Folder-A agent1))
                       (del (Free agent1)
                            (A-Folder-open true1)))
                 ( 0.2 (add )
                       (del ))))
(action close-folder-B
        (var (agent : agent1)
             (true : true1))
        (preconditions (Free agent1)
                       (B-Folder-open true1))
        (time = 2)
        (resources (power 4))
        (effects ( 0.8 (add (B-Folder-closed true1)
                            (Get-Folder-B agent1))
                       (del (Free agent1)
                            (B-Folder-open true1)))
                 ( 0.2 (add )
                       (del ))))
(action close-folder-C
        (var (agent : agent1)
             (true : true1))
        (preconditions (Free agent1)
                       (C-Folder-open true1))
        (time = 2)
        (resources (power 4))
        (effects ( 0.8 (add (C-Folder-closed true1)
                            (Get-Folder-C agent1))
                       (del (Free agent1)
                            (C-Folder-open true1)))
                 ( 0.2 (add )
                       (del ))))
```

```
(action leave-folder-A
        (var (agent : agent1))
        (preconditions (Get-Folder-A agent1))
        (time = 2)
        (resources (power 4))
        (effects ( 1 (add (Free agent1))
                     (del (Get-Folder-A agent1)))))

(action leave-folder-B
        (var (agent : agent1))
        (preconditions (Get-Folder-B agent1))
        (time = 2)
        (resources (power 4))
        (effects ( 1 (add (Free agent1))
                     (del (Get-Folder-B agent1)))))

(action leave-folder-C
        (var (agent : agent1))
        (preconditions (Get-Folder-C agent1))
        (time = 2)
        (resources (power 4))
        (effects ( 1 (add (Free agent1))
                     (del (Get-Folder-C agent1)))))
```

```
(action insert-folder-A
        (var (agent : agent1)
             (true : true1)
             (file : file1))
        (preconditions (A-Folder-open true1)
                       (Get-File agent1 file1)
                       (A-File file1))
        (time = 4)
        (resources (power 4))
        (effects ( 0.7 (add (Filed file1)
                            (Free agent1))
                       (del (Get-File agent1 file1)))
                 ( 0.3 (add )
                       (del ))))
(action insert-folder-B
        (var (agent : agent1)
             (true : true1)
             (file : file1))
        (preconditions (B-Folder-open true1)
                       (Get-File agent1 file1)
                       (B-File file1))
        (time = 4)
        (resources (power 4))
        (effects ( 0.7 (add (Filed file1)
                            (Free agent1))
                       (del (Get-File agent1 file1)))
                 ( 0.3 (add )
                       (del ))))
(action insert-folder-C
        (var (agent : agent1)
             (true : true1)
             (file : file1))
        (preconditions (C-Folder-open true1)
                       (Get-File agent1 file1)
                       (C-File file1))
        (time = 4)
        (resources (power 4))
        (effects ( 0.7 (add (Filed file1)
                            (Free agent1))
                       (del (Get-File agent1 file1)))
                 ( 0.3 (add )
                       (del ))))
```

A.6 File World Problem

```
(define (problem file-world-problem-1)
                (domain file-world)
                (objects
                        (file : f1 f2 f3)
                        (agent : a1 a2)
                        (true : T))
                (init
                        (A-Folder-closed T)
                        (B-Folder-closed T)
                        (C-Folder-closed T)
                        (Free a1)
                        (Free a2)
                        (Type-Unknown f1)
                        (Type-Unknown f2)
                        (Type-Unknown f3))
                (time 12)
                (resources (power 16))
                (goals
                        ((Filed f1) 1)
                        ((Filed f2) 1)
                        ((Filed f3) 1)))
```

Appendix B

Experimental Data

Table B.1
Mars Rover Expanded States

Problem	No heuristic	TORT heuristic	TOFR heuristic	ARFR heuristic
1-1	1984	108	87	140
1-2	15641	1110	592	728
1-3	121316	5944	2174	2508
1-4	920521	22738	17713	14623
2-1	6713	396	306	262
2-2	81757	6745	4916	2039
2-3	903221	87981	62540	14355
2-4	9519549	1186813	788852	107061
3-1	9032	406	311	333
3-2	121771	7634	5895	2801
3-3	1482379	121262	99711	25626
3-4	17288865	1899310	1382671	218206
4-1	9279	406	311	336
4-2	127707	7639	5900	2888
4-3	1579588	121741	100684	27703
4-4	18685955	1918370	1456119	247674

Table B.2
Machine Shop Expanded States

Problem	No heuristic	TORT heuristic	TOFR heuristic	ARFR heuristic
1-1	1464	46	27	11
1-2	2494	138	54	18
1-3	5112	228	139	33
1-4	9491	350	241	55
2-1	66705	3356	1875	67
2-2	171929	11164	5262	306
2-3	947172	77168	31505	1720
2-4	2554196	129252	41711	1820
3-1	1960047	59996	24006	504
3-2	6536745	239530	116646	2678
3-3	21493432	1187605	562921	11341
3-4	>24640000	3222702	1195344	21166
4-1	12013903	258274	108610	1078
4-2	45669855	1220166	692143	7421
4-3	>55520000	7149781	3226530	38719
4-4	>59670000	>9130000	>6970000	78870

Table B.3
File World Expanded States

Problem	No heuristic	TORT heuristic	TOFR heuristic	ARFR heuristic
1-1	4429	223	223	55
1-2	50443	2951	1411	103
1-3	531049	81970	39849	1489
1-4	5546179	1063000	517527	5336
2-1	6637	343	343	73
2-2	81619	5617	2839	139
2-3	912343	192890	82459	2149
2-4	10294489	2748260	1375839	14543
3-1	9313	463	463	91
3-2	121867	8647	5957	235
3-3	1430377	323024	207626	2895
3-4	17341963	4862669	3204565	30659
4-1	12457	583	583	109
4-2	171835	13171	9373	283
4-3	2101675	485214	342797	3841
4-4	>16200000	8163609	5451835	44679

Appendix C

AAAI Publication Copyright Information

 AAAI Publications

---

Copyright Notice

Authors who publish a paper in this conference agree to the following terms:

1. Author(s) agree to transfer their copyrights in their article/paper to the Association for the Advancement of Artificial Intelligence (AAAI), in order to deal with future requests for reprints, translations, anthologies, reproductions, excerpts, and other publications. This grant will include, without limitation, the entire copyright in the article/paper in all countries of the world, including all renewals, extensions, and reversions thereof, whether such rights current exist or hereafter come into effect, and also the exclusive right to create electronic versions of the article/paper, to the extent that such right is not subsumed under copyright.
2. The author(s) warrants that they are the sole author and owner of the copyright in the above article/paper, except for those portions shown to be in quotations; that the article/paper is original throughout; and that the undersigned right to make the grants set forth above is complete and unencumbered.
3. The author(s) agree that if anyone brings any claim or action alleging facts that, if true, constitute a breach of any of the foregoing warranties, the author(s) will hold harmless and indemnify AAAI, their grantees, their licensees, and their distributors against any liability, whether under judgment, decree, or compromise, and any legal fees and expenses arising out of that claim or actions, and the undersigned will cooperate fully in any defense AAAI may make to such claim or action. Moreover, the undersigned agrees to cooperate in any claim or other action seeking to protect or enforce any right the undersigned has granted to AAAI in the article/paper. If any such claim or action fails because of facts that constitute a breach of any of the foregoing warranties, the undersigned agrees to reimburse whomever brings such claim or action for expenses and attorneys' fees incurred therein.

4. Author(s) retain all proprietary rights other than copyright (such as patent rights).
5. Author(s) may make personal reuse of all or portions of the above article/paper in other works of their own authorship.
6. Author(s) may reproduce, or have reproduced, their article/paper for the author's personal use, or for company use provided that AAAI copyright and the source are indicated, and that the copies are not used in a way that implies AAAI endorsement of a product or service of an employer, and that the copies per se are not offered for sale. The foregoing right shall not permit the posting of the article/paper in electronic or digital form on any computer network, except by the author or the author's employer, and then only on the author's or the employer's own web page or ftp site. Such web page or ftp site, in addition to the aforementioned requirements of this Paragraph, must provide an electronic reference or link back to the AAAI electronic server, and shall not post other AAAI copyrighted materials not of the author's or the employer's creation (including tables of contents with links to other papers) without AAAI's written permission.
7. Author(s) may make limited distribution of all or portions of their article/paper prior to publication.
8. In the case of work performed under U.S. Government contract, AAAI grants the U.S. Government royalty-free permission to reproduce all or portions of the above article/paper, and to authorize others to do so, for U.S. Government purposes.
9. In the event the above article/paper is not accepted and published by AAAI, or is withdrawn by the author(s) before acceptance by AAAI, this agreement becomes null and void.

Privacy Statement

The names and email addresses entered in this conference site will be used exclusively for the stated purposes of this conference and will not be made available for any other purpose or to any other party.

Association for the Advancement of Artificial Intelligence

445 Burgess Drive
Menlo Park, CA 94025

AAAI Copyright Form

Title of Article/Paper: Generating Plans in Concurrent, Probabilistic, Over-Subscribed Domains

Publication in Which Article Is to Appear: AAAI Conference – Doctoral Consortium

Author's Name(s): Li Li

Please type or print your name(s) as you wish it (them) to appear in print

(Please read and sign Part A only, unless you are a government employee and created your article/paper as part of your employment. If your work was performed under Government contract, but you are not a Government employee, sign Part A and see item 5 under returned rights.)

PART A–Copyright Transfer Form

The undersigned, desiring to publish the above article/paper in a publication of the Association for the Advancement of Artificial Intelligence, (AAAI), hereby transfer their copyrights in the above article/paper to the Association for the Advancement of Artificial Intelligence (AAAI), in order to deal with future requests for reprints, translations, anthologies, reproductions, excerpts, and other publications.

This grant will include, without limitation, the entire copyright in the article/paper in all countries of the world, including all renewals, extensions, and reversions thereof, whether such rights current exist or hereafter come into effect, and also the exclusive right to create electronic versions of the article/paper, to the extent that such right is not subsumed under copyright.

The undersigned warrants that they are the sole author and owner of the copyright in the above article/paper, except for those portions shown to be in quotations; that the article/paper is original throughout; and that the undersigned right to make the grants set forth above is complete and unencumbered.

If anyone brings any claim or action alleging facts that, if true, constitute a breach of any of the foregoing warranties, the undersigned will hold harmless and indemnify AAAI, their grantees, their licensees, and their distributors against any liability, whether under judgment, decree, or compromise, and any legal fees and expenses arising out of that claim or actions, and the undersigned will cooperate fully in any defense AAAI may make to such claim or action. Moreover, the undersigned agrees to cooperate in any claim or other action seeking to protect or enforce any right the undersigned has granted to AAAI in the article/paper. If any such claim or action fails because of facts that constitute a breach of any of the foregoing warranties, the undersigned agrees to reimburse whomever brings such claim or action for expenses and attorneys' fees incurred therein.

Returned Rights

In return for these rights, AAAI hereby grants to the above authors, and the employers for whom the work was performed, royalty-free permission to:

1. Retain all proprietary rights other than copyright (such as patent rights).
2. Personal reuse of all or portions of the above article/paper in other works of their own authorship.
3. Reproduce, or have reproduced, the above article/paper for the author's personal use, or for company use provided that AAAI copyright and the source are indicated, and that the copies are not used in a way that implies AAAI endorsement of a product or service of an employer, and that the copies per se are not offered for sale. The foregoing right shall not permit the posting of the article/paper in electronic or digital form on any computer network, except by the author or the author's employer, and then only on the author's or the employer's own web page or ftp site. Such web page or ftp site, in addition to the aforementioned requirements of this Paragraph, must provide an electronic reference or link back to the AAAI electronic server, and shall not post other AAAI copyrighted materials not of the author's or the employer's creation (including tables of contents with links to other papers) without AAAI's written permission.
4. Make limited distribution of all or portions of the above article/paper prior to publication.
5. In the case of work performed under U.S. Government contract, AAAI grants the U.S. Government royalty-free permission to reproduce all or portions of the above article/paper, and to authorize others to do so, for U.S. Government purposes.

In the event the above article/paper is not accepted and published by AAAI, or is withdrawn by the author(s) before acceptance by AAAI, this agreement becomes null and void.

Li Li — Author's Signature

April 7, 2008 — Date

Employer for whom work was performed

Title (if not author)