Generating Admissible Heuristics by Abstraction for Search in Stochastic Domains

Natalia Beliaeva and Shlomo Zilberstein

Department of Computer Science
University of Massachusetts Amherst
{nbeliaev, shlomo}@cs.umass.edu

Abstract. Search in abstract spaces has been shown to produce useful admissible heuristic estimates in deterministic domains. We show in this paper how to generalize these results to search in stochastic domains. Solving stochastic optimization problems is significantly harder than solving their deterministic counterparts. Designing admissible heuristics for stochastic domains is also much harder. Therefore, deriving such heuristics automatically using abstraction is particularly beneficial. We analyze this approach both theoretically and empirically and show that it produces significant computational savings when used in conjunction with the heuristic search algorithm LAO*.

1 Introduction

The Markov decision process (MDP) is widely used in artificial intelligence to solve problems of planning and learning under uncertainty. The most common way to solve an MDP is by using a dynamic programming algorithm such as value iteration or policy iteration. The major drawback of this approach is that the entire state space has to be evaluated. More recently, heuristic search algorithms have been developed for solving MDPs [6]. These algorithms can avoid evaluating states that are not reachable from the start state by an optimal policy. The effectiveness of heuristic search mostly depends on the heuristic function being used to guide the search process. One way to generate an *admissible* heuristic is to use search in abstract spaces [7, 11]. Abstraction works by replacing an original state space by an abstract space, which is easier to search. This idea is not new. It has been previously applied to creating admissible heuristics for A* search [8]. More recently, there has been growing interest in developing heuristics using a form of abstraction called *pattern database* [3, 9, 10]. The goal of this paper is to extend the use of abstraction as a means of creating admissible heuristics for search in stochastic domains. Heuristic estimates generated by abstraction are then used to guide LAO*, which is a heuristic search algorithm that can be used to solve stochastic planning problems. To test whether heuristics generated by abstraction produce any savings as compared to uninformed search, LAO* algorithm is applied to a task planning problem that involves uncertainty regarding the use of resources. The structure of this problem facilitates the creation of an abstract space very easily by varying the resolution of resource usage, always rounding up the amount of resources left for future activity. We show that heuristic estimates obtained by search in such abstract space are always admissible. That is, the heuristic value is an optimistic estimate (overestimate) of the

actual value of a state. We also show that the effectiveness of such heuristic estimates depends on the problem and that they could result in significant savings compared to blind search.

The rest of the paper is organized as follows. In Section 2, we review some related work on search in abstract spaces and alternative approximation techniques for MDPs. Section 3 describes the general methodology used in the current research. Section 4 describes the specific model used in the paper. Section 5 analyzes experimental results. Section 6 concludes the paper with a summary of contributions and further work.

2 Related Work

We describe briefly related work in two research areas. First, we examine previous work on the problem of creating heuristics by abstraction in deterministic settings. This paper extends this body of research to stochastic domains. We then describe existing exact and approximate techniques for solving MDPs; heuristic search presents an alternative approach to these techniques.

2.1 Creating Heuristic by Abstraction for Search in Deterministic Domains

Several researchers have looked at the problem of creating heuristic by abstraction for search in deterministic domains (for example, [7, 8, 11, 12]). The most relevant work to the current study is Holte *et al.* [8]. This paper focuses on one type of abstraction called homomorphism (grouping together states of the original state space to create a single abstract state). The heuristic created by abstraction is then used to guide A* search. The goal of the paper was to develop a technique that would break Valtorta's barrier. To achieve this goal the number of states expanded by a heuristic search has to be less than the number of states expanded by uninformed search. The authors use an abstraction-based search algorithm called hierarchical A*. To create an abstraction they use the STAR abstraction technique, which groups together neighboring states within a certain radius. Once one level of abstraction is created, the procedure is repeated recursively until a trivial abstract level is created. The implementation of hierarchical A* is standard except for the way heuristic values are estimated. Every time A* needs a heuristic estimate, it is computed by searching at the next level of abstraction. It was found that a naive version of the algorithm ends up expanding many more states as compared to uninformed search, i.e. Valtorta's barrier is not broken. This could be explained by the fact that although A* never expands the same state twice in a single search, it has to expand the same state many times while performing multiple searches of the abstract levels. To overcome this problem the authors implemented two types of cashing techniques and as a result the Valtorta's barrier was broken in every domain. The authors have also discovered that as the radius of abstraction increases, the number of nodes expanded by hierarchical A* decreases until it reaches some minimum value. Increasing the abstraction radius further caused the number of expanded nodes to increase. In every case the best abstraction radius represented a large fraction of the search space and as a result the abstraction hierarchy contained only one non-trivial level. In this paper we examine

the applicability of these same ideas in stochastic search and evaluate the effectiveness of the approach.

More recently, an effective approach to exploit abstraction in the form of a pattern database has been developed. The idea was introduced by Culberson and Schaeffer [3] who applied it to permutation problems, like the 15-puzzle. To form a pattern database, a search space is projected into an abstract space, which is small enough to allow an efficient computation of the value function for each abstract state. The computed values are stored in a look-up table. Each abstract state is called a pattern and the table that stores the optimal values is called a pattern database. These precomputed values are then used as heuristic estimates for the search in the original state space. Usually more than one pattern database is defined for the same problem. Heuristic estimates for the states of the original space are computed as maximum of several pattern database heuristics. For example, Korf defined three pattern databases to solve the Rubik's cube problem [10]. Similarly, Korf and Felner used eight pattern databases to solve the 24-puzzle [9].

2.2 Approximate Solutions to MDPs

A common way to solve MDPs is by using dynamic programming techniques such as value iteration or policy iteration. The problem with this approach is that the entire state space–which grows exponentially with the number of state variables–has to be evaluated. This makes it hard to find exact solutions, leading to a vast literature on techniques that can approximate the optimal solution. Both planning and learning techniques for approximation of MDP solutions have been developed. The goal of both planning and learning under uncertainty is to discover an optimal or near-optimal policy of action, represented as a mapping from states to actions. The main difference is that planning problems assume that the action model and the reward function are known, whereas learning problems assume both of these to be initially unknown and attempt to learn them. While planning is typically performed off-line, learning algorithms are frequently designed for on-line operation.

A large body of research that attempts to find an approximate solution to an MDP in the context of planning deals with reducing the level of detail in the problem representation by aggregating states with similar or identical values and/or action choices. These aggregate states are then treated as a group by the dynamic programming algorithm (see [5]). Another way to reduce the complexity of the problem is by pruning the tree representation of value functions by removing such nodes in the tree that induce small differences in value (see [2]) or by substituting the values at the terminals with ranges of values (see [13]). Another class of approximation procedures used in planning under uncertainty involves searching local regions or so called envelopes of the state space (see [4, 14]).

Solving MDPs has also been a focus area in reinforcement learning (RL). Most RL algorithms adapt dynamic programming algorithms so that they could be used on-line. To avoid the curse of dimensionality, many methods have been proposed to approximate MDP solutions. Barto and Mahadevan, for example, identify the following three methods for finding approximate solutions using RL algorithms [1]: (1) Restricting computation to states along sample trajectories to avoid the exhaustive sweeps of dynamic

programming; (2) Sampling from the appropriate distribution to simplify the basic dynamic programming backup; and (3) Representing the value function and/or policies more compactly by using function approximation methods, such as linear combinations of basic functions or neural networks.

The technique we present in this paper is an exact algorithm, but it can be easily transformed into an approximation technique. In previous work, we have shown how to convert any exact heuristic search algorithm into a "well-behaved" anytime algorithm that could produce approximate solutions with error-bounds that improve with computation time. However, examining the anytime characteristics of hierarchical LAO* is beyond the scope of this paper.

3 Methodology

3.1 The LAO* Algorithm

The LAO* algorithm was developed by Hansen and Zilberstein [6] as a heuristic approach to finding optimal solutions to MDPs. What distinguishes LAO* from other classical heuristic search algorithms, such as AO*, is the fact that it allows to find solutions that contain loops. Since LAO* is a heuristic search algorithm, it can avoid evaluating the entire search space which makes it a good alternative to dynamic programming algorithms that are commonly used to solve MDPs. Since LAO* does not evaluate every state of the problem, it is not necessary to supply the entire graph to the algorithm. Instead, a graph is specified implicitly by a start state and a successor function.

For complete details of the implementation of LAO* see Hansen, Zilberstein [6]. Generally, the algorithm has two main steps: a forward search step and a dynamic programming step. The forward step identifies and expands the best partial solution graph. The dynamic programming step updates the evaluation function and marks best action for each state that belongs to the current best solution. Although LAO* works correctly independently of which state of the best partial solution is expanded next, the performance of the algorithm can be improved by a good heuristic function. One way to construct a heuristic is by search in abstract spaces.

3.2 Heuristic Construction by Abstraction

Heuristics are designed to speed up search. However, construction of a good heuristic usually comes at a cost. The goal is to come up with a heuristic such that the cost of computing it is less than the savings from using it. The use of heuristic h is said to be beneficial if the total number of states expanded by search with heuristic h is less than the number of states expanded by "blind" or uninformed search. In a stochastic setting "blind" search is also equivalent to reachability analysis.

There are two types of abstraction that can be used to construct a heuristic:

1. Embedding – relaxing a problem by "adding edges" to a state space (for example, by dropping preconditions from, or adding macro-operators to the state-space definition).

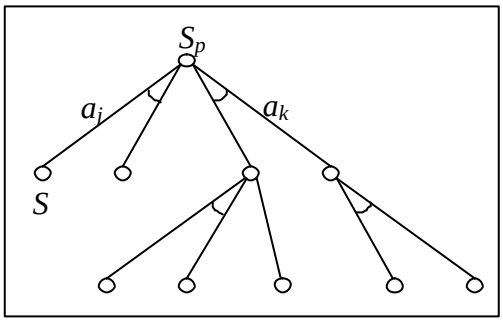

Fig. 1. State S_p is expanded and action a_k is chosen as the best action. State S is visited but not expanded

2. Homomorphism – grouping together several states in the original state space to create a single state in the abstract space.

It was proven by Valtorta (see [15]) that A* search using heuristic constructed by embedding transformation cannot be beneficial. Holte *et al.* [8] have generalized Valtorta's theorem to any abstraction transformation. They have shown that if the abstraction used to direct A* is a homomorphism, then it can be beneficial. The potential savings are due to the fact that expansion of many states in the original space can be replaced by an expansion of a single state in the abstract space. The goal of this research is to see whether the same idea holds in a stochastic setting, i.e. whether an admissible heuristic can be constructed by homomorphism and whether it can be beneficial if used to direct LAO* search. The next section generalizes Holte *et al.* version of Valtorta's theorem to stochastic search spaces.

3.3 Valtorta's Theorem Generalized to Stochastic Search

Let *SP* be the original state space, *SP'* the abstraction of *SP*. Let S be any state necessarily expanded when the given problem (S_0, G) is solved by reachability analysis directly in space *SP*. Let f be any abstraction mapping from *SP* to *SP'* and $h_f(S)$ be computed by reachability analysis in *SP'* from $f(S)$ to $f(G)$. If the problem is solved in *SP* by LAO* search using $h_f(\cdot)$ as heuristic estimate, then either:

1. S itself will be expanded, or
2. $f(S)$ will be expanded

Proof. When LAO* terminates, every state will either be

1. expanded,
2. visited, or
3. or unvisited.

We examine each one of these cases below:

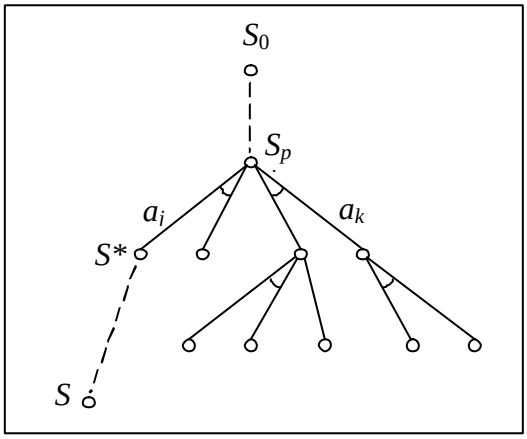

Fig. 2. State S^* is visited but not expanded. State S is unvisited

1. In the first case, the state *S* itself is expanded.

2. In the second case, the parent node S_p of state S, is expanded and the best action for S_p is computed (see Fig. 1). Because S is not expanded, the action a_j which leads to state S with a certain probability is suboptimal. However, to pick an optimal action for state S_p, $h_f(S)$ must have been computed. To compute $h_f(S)$, it is necessary to solve a problem $(f(S), f(G))$ in the abstract space by reachability analysis. Therefore, $f(S)$ has to be expanded at the first step.

3. In the third case, the state S is unvisited (see Fig. 2). It means that on every path from S_0 to S there must be a state which was visited but not expanded. Let S^* be such state on any shortest path from S_0 to S. As in the previous case, $h_f(S^*)$ must have been computed. To compute $h_f(S^*)$, it is necessary to solve a problem $(f(S^*), f(G))$ in the abstract space by reachability analysis. Since state S is reachable in the original state space, the corresponding state, $f(S)$. in the abstract space has to be reachable as well. Therefore, while solving the problem $(f(S^*), f(G))$ by reachability analysis, the state $f(S)$ has to be expanded.

3.4 General Problem Description

One type of problems that can be solved using LAO* with heuristics created by abstraction is executing multiple tasks that involve uncertainty about resources. In such problems, an agent has to perform a series of tasks. Every task is associated with a set of actions that an agent can undertake to complete the task. Each action uses an uncertain amount of one or several resources (for example, time, energy, etc.) and compensates the agent with a certain reward. The process stops once the agent either performs all tasks or runs out of at least one of the resources. The goal is to maximize the collected

reward while performing a sequence of tasks. The structure of such problems allows creating an abstract space very easily by grouping states by resources.

Each state is defined by the amount of resources left:

$$R_i = \{0,\ 1,\ 2, \ldots, N_i\}\,, \quad i = 1, \ldots, n$$

and by values of the variables:

$$V_i = \{1,\ 2, \ldots, M_i\}\,, \quad i = 1, \ldots, m$$

Formally, $S = \{R_1, \ldots, R_n;\ V_1, \ldots, V_m\}$. The start state can be defined as

$$S_0 = \{N_1, \ldots, N_n;\ 1, \ldots, 1\}$$

There are many possible terminal states. One example of a terminal state is

$$G = \{0, \ldots, R_n;\ V_1, \ldots, V_m\}$$

3.5 Creating an Abstract Space

To create the abstract space, it is first necessary to choose the desired granularity of abstraction or *abstraction step*. Selecting larger steps for grouping resources will result in a smaller total number of states in an abstract space and therefore fewer states to expand while performing a "blind" search. On the other hand, the generated heuristic estimates will be coarser and subsequently more states will need to be expanded in the original space. If smaller abstraction steps are used, then there will be more work in the abstract space, and less in the original. Therefore, it is important to use such abstraction steps that result in an optimal trade-off between the number of states expanded in the abstract and the original spaces.

When an abstract space is created, states of the original space are grouped by resources with each resource rounded up. Only states with the same variable values can be grouped. Since we are representing the amount of remaining resources (as opposed to the amount of used resources), rounding resources up ensures admissibility of the heuristic as it will always be overestimating the reward. Given that the amount of resources available in each state is overestimated, it might be possible to do more work (i.e. take more actions) and subsequently collect a larger reward.

4 The Model

4.1 Problem Specification

To test whether heuristic estimates generated by abstraction produce any savings as compared to uninformed search, the LAO* algorithm was used to solve the following problem that involves uncertainty regarding the use of resources. An agent operates autonomously for a period of time. Its goal is to perform a sequence of M tasks (see Fig. 3). A terminal state is reached when the agent either performs all tasks or runs out of at least one resource.

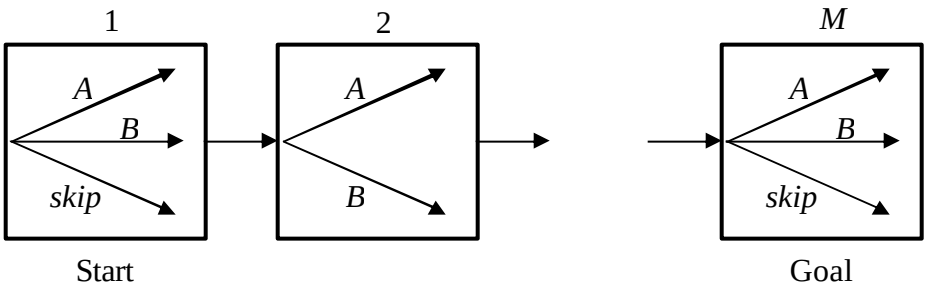

Fig. 3. An illustration of a problem instance

Each task can be executed either by taking an action *A*, or by taking an action *B*. In addition, some tasks may be skipped altogether. When a problem instance is created, the skip action is added to a task with probability p and omitted with probability $1 - p$. (This probability only applies to the process of generating a random problem instance.) Action *A*, on average, uses less resources than action *B*. An agent's goal is to maximize the reward obtained during the time period. The *attractiveness* of the task can be defined as a ratio of expected reward to expected resources used to execute an action. When a sample task is generated, its expected reward is computed as follows. First, an average amount of each resource *i* used to execute an action is generated. Then the expected reward for performing the task is computed as:

$$ER = \sum_{i=1}^{n} k_i \times E[Res\_used_i] \ . \tag{1}$$

where,

k_i is a random number in the range $[0.1, 1]$
n is the number of resources

Each state is defined by the remaining resources, $R_i = \{0,\ 1, \ldots, N_i\}$, and by the current task number, $I = \{1,\ 2, \ldots, M\}$. Formally, $S = \{R_1, \ldots, R_n;\ I\}$. The start state is $S_0 = \{N_1, \ldots, N_n;\ 1\}$. An example of a terminal state is $G = \{0, \ldots, 0;\ I\}$.

4.2 Original and Abstract State Space Construction

State spaces are represented as AND/OR graphs with all unique tree nodes stored in hash tables. To construct the original and abstract state space, first, the average resource use and the reward for action one and two are precomputed. Then whether the task can be skipped is determined according to probability p. Construction of the original tree starts with the root. The root node is assigned the maximum values for each resource and variable value of 1. After that the whole graph is generated as follows. The number of successors for the first two actions is determined at random from the range [5, 15]. In case of multiple resources, it is assumed that resources are correlated, i.e. if an action

requires the minimum amount of resource 1, it will also require the minimum amount of all other resources. Under this assumption, the resource values for successor states are determined as follows. The first successor always gets assigned the resource values of the previous resource levels less some predetermined minimum resource use. The middle successor gets assigned the resource values of the previous resource levels less the precomputed average resource use. The resource levels for the rest of the successors are linearly projected from these two states. The variable value for each of the successors is determined as a previous variable value plus 1. Successor states are assigned probability values according to the linear probability model. The middle state which is associated with the average resources use has the highest probability of occurrence. The first and last states which are associated with the lowest and highest resources use have the smallest probability of occurrence. The probability values associated with all other states are linearly interpolated.

An abstract state space is constructed using the same data and assumptions as the original state space. First, the abstract root node is constructed. For example, if the initial resource is 50 and the abstraction step is 20, the initial level of resource is rounded up and the abstract root state gets assigned the resource value of 60. For the purpose of construction of abstract successors the number of successors in the original state space is assumed to be the maximum. The resource value for each successor is generated in the same way as in the original space. Then, each resource is rounded up according to the abstraction step and identical states are grouped. The probability value for each abstract successor is assigned according to the linear probability model.

4.3 Heuristic Construction and its Application to the LAO* Algorithm

Once an abstract space is constructed, the next step is to perform a "blind" search of this abstract space. During this process, all reachable abstract states get expanded and assigned a value. These values are then used as heuristic estimates for the states in the original state space. Every time an algorithm needs to estimate a heuristic value for a state in the original space, it creates an abstract state that corresponds to the original state by rounding the resources up according to the abstraction steps and looks up the abstract state value in the hash table. The abstract state space constructed using the procedure outlined in the previous section is not guaranteed to give the exact representation of the original state space. As a result, some of the original states might not have a counterpart in the abstract state space. In case the corresponding abstract state cannot be found in the hash table, the value of the state with the same variable but higher level of the resource is used. This way the heuristic value is always overestimated and it remains admissible.

The procedure for constructing the heuristic by abstraction can be taken one step further by using multiple abstract spaces, i.e. by creating an abstraction hierarchy. The first abstract space is created in exactly the same way as before. The next abstract space is created by grouping states of the previous abstract space. This process is repeated until the top abstract space becomes trivial. The algorithm starts by performing a complete "blind" search of the top abstract level. After that at each iteration of LAO* whenever it is necessary to estimate the value of the heuristic, the next higher level of abstraction

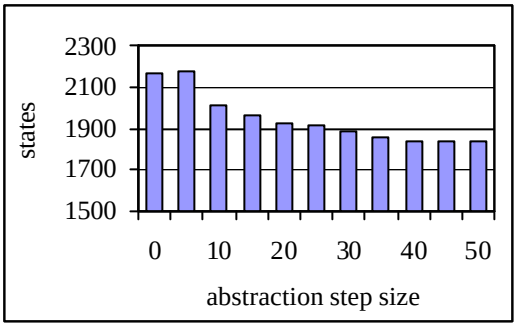

Fig. 4. Average number of states expanded both at the base and abstract level as a function of the abstraction step

is searched. The search of the abstract space starts with the abstract state that corresponds to the state in the level below. When the second to last abstract level is searched, the heuristic estimates are simply looked up in the hash table for the top abstract level. Once an abstract space is searched at least once, the values are known for all states that belong to the solution graph. These states can be cashed and their values can be used as heuristic estimates for the lower level in all subsequent searches.

It is worth noting that some difficulties could arise with this approach. Since abstract states are created by rounding the resources up, there could be no change in the level of resources after executing a series of actions. Therefore, in certain situations an agent can come back to the same state. In general, this is not a problem since LAO* can easily handle solutions with loops. However, it becomes a problem if an agent comes back to the same state with probability 1 because it leads to an infinite loop. Fortunately, this difficulty does not arise in the problem considered here because each state is defined by the amount of resources left and the number of the task to be performed. Even if no resources have been used while executing a task, the task number will increase. Therefore, an agent cannot come back to the same state once an action is executed. Although this problem does not contain loops and therefore could be handled by AO* algorithm, the procedure described here is general enough to handle problems with loops.

5 Experimental Results

5.1 One Resource, One Level of Abstraction

This section analyzes problems with one resource. The heuristic estimates are based on one level of abstraction. Fig. 4 shows the average number of states expanded at the base and abstract levels over 20 problems with 15 tasks and the starting level of resource of 200 units. The first bar corresponds to the average number of states expanded by "blind" search. On average, the use of abstraction step 5 is not beneficial since the algorithm expands more states than the "blind" search. All other abstraction steps can be considered beneficial since they result in some savings as compared to the "blind"

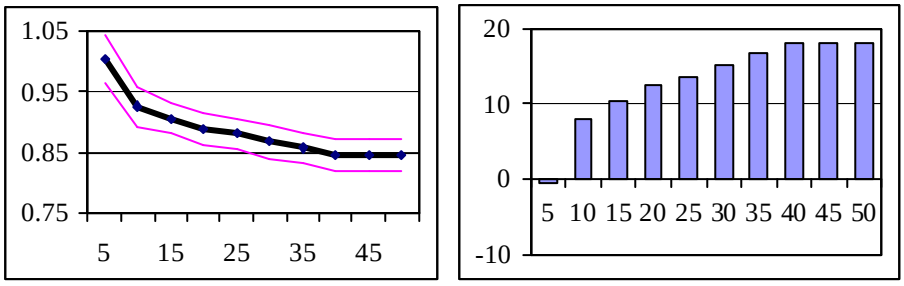

Fig. 5. Left: average amount of work as compared to blind search as a function of the abstraction step. Right: average savings in % as compared to blind search as a function of the abstraction step. An average over 20 problems with initial level of resource of 200, 15 tasks, and probability of skip action of 1

search. The smallest number of states gets expanded when the abstraction step of 40 or above is used.

Fig. 5 shows the average amount of work as compared to the "blind" search (the chart on the left) and the average savings (or loss) that occur due to the use of abstraction (the chart on the right). The probability of "skip" action is 1, i.e. each task can be performed by taking an action *A* or an action *B* or the task can be skipped. The three lines on the charts on the left represent the average amount of work over 20 problems with one standard deviation band around it. The average amount of work is determined as a ratio of the total number of states expanded at both base and abstract levels over the total number of states expanded by "blind" search. The left chart shows that as the abstraction step increases, the average amount of work decreases. Similarly, the chart on the right shows that the average savings due to the use of heuristic constructed by abstraction go up as the abstraction step size increases. The maximum savings achieved are 18.1%.

The size of the problem determined by the number of unique states in a hash table can be increased by either increasing the starting level of resource or by increasing the number of tasks the agent needs to perform or by increasing both. Fig. 6 explores the relationship between the average savings due to abstraction and the size of the problem when the size of the problem increases due to increase in the starting level of resource. The chart on the right shows the average number of states in a hash table for each level of resource. The hash table keeps growing until initial level of resource reaches 350. After that the increase in the starting level of resource does not result in additional states being added to the hash table. When initial resource level is set at 50, it is enough to perform only a few tasks. In this case, the algorithm tries to determine which tasks should be skipped and which tasks should be performed. On the other hand, when initial level of resource is set at 500, the resource is plentiful to perform all tasks, so at each step it will be necessary to determine whether an action *A* or *B* should be preferred since the skip action will always be suboptimal (because of zero reward). The chart on the left shows the average savings due to abstraction as a function of the initial level of resource. When the level of resource is low, the savings from abstraction are the lowest

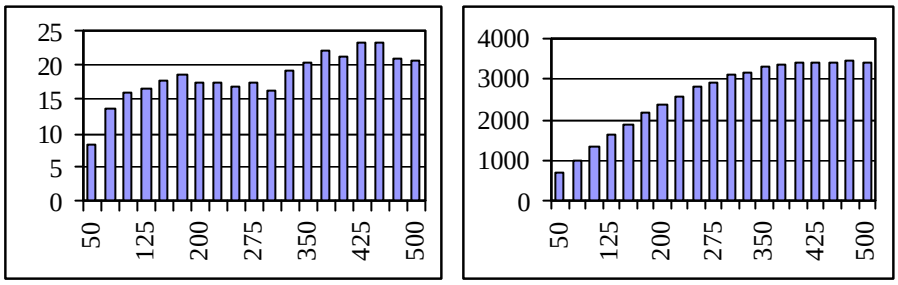

Fig. 6. Left: average savings in % as compared to blind search as a function of the initial level of resource. Right: average size of the hash table as a function of the initial level of resource. Averages are over 50 problems with 15 tasks, and probability of skip action of 1

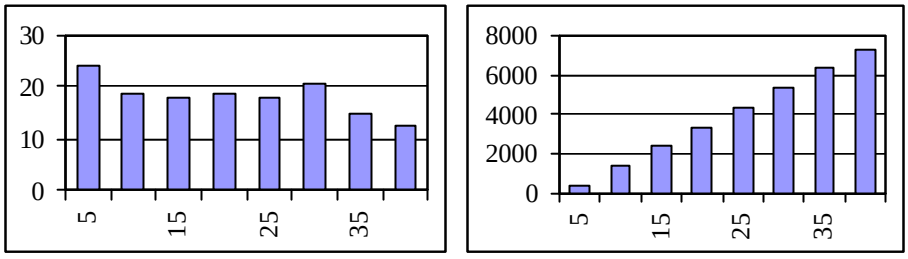

Fig. 7. Left: average savings in % as compared to blind search as a function of the number of tasks. Right: average size of the hash table as a function of the number of tasks. Averages are over 50 problems with initial level of resource set at 200, and probability of skip action of 1

(less than 15%). The savings are the highest (above 20%) when resource is abundant. A lot of the savings will occur because brunches corresponding to the skip action are suboptimal and therefore will be ignored in a heuristic search but expanded in a “blind” search. In general, the most savings occur when the problem tree has a lot of clearly suboptimal branches. Heuristics constructed by abstraction will easily identify those branches and save a lot of work at the base level.

Fig. 7 explores the relationship between the average savings due to abstraction and the size of the problem when the size of the problem increases due to increase in the number of tasks an agent has to perform. The chart on the right shows the average number of states in a hash table as a function of the number of tasks in a problem. As the number of tasks to be performed increases, so does the size of the hash table. On average, addition of 5 tasks to the problem adds roughly 1000 states to the hash table. The chart on the left shows the average savings due to abstraction as a function of the number of tasks to be performed. As in Fig. 6, the most savings occur when the initial level of resource is high relative to the number of tasks to be performed. As the number of tasks to be performed goes up, the average savings go down. The largest savings of 24.2% occur when there are only 5 tasks to be performed. In this case there is enough of the resource to perform all tasks. Savings occur largely due to the possibility to ignore

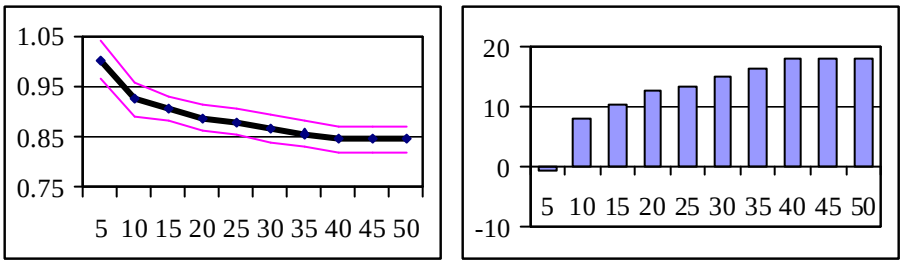

Fig. 8. One level of abstraction. Left: average amount of work as compared to blind search as a function of the abstraction step. Right: average savings in % as compared to blind search as a function of the abstraction step. An average over 20 problems with initial level of resource of 200, 15 tasks, and probability of skip action of 1

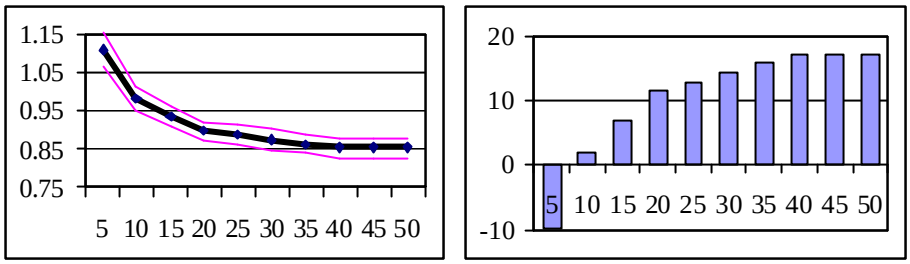

Fig. 9. Two levels of abstraction. Left: average amount of work as compared to blind search as a function of the abstraction step. Right: average savings in % as compared to blind search as a function of the abstraction step. An average over 20 problems with initial level of resource of 200, 15 tasks, and probability of skip action of 1

those branches that correspond to the skip action while performing the heuristic search.

5.2 One Resource, Two Levels of Abstraction

In this section, an identical set of 20 problems with 15 tasks, initial level of resource of 200 and probability of skip action of 1 was solved by LAO*, first, using heuristic constructed with one level of abstraction, second, using heuristic constructed with two levels of abstraction. Fig. 8 shows graphs for one level of abstraction. Fig. 9 shows graphs for two levels of abstraction. In this case X axis values correspond to the abstraction step at the first level, s1. Abstraction step at the second level was assumed to be $s_2 = 2 \times s_1$. Comparison of the two figures shows that heuristic constructed by abstraction with one level of abstraction produces higher savings as compared to heuristic constructed by abstraction with two levels. For example, the highest possible savings with one level of abstraction are 18.1%, whereas the highest possible savings with two levels of abstraction are 17.1%.

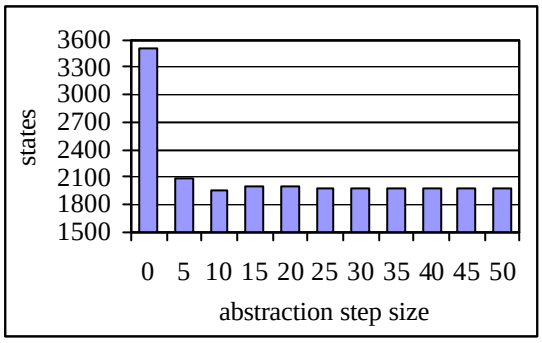

Fig. 10. Average number of states expanded both at the base and abstract level as a function of the abstraction step

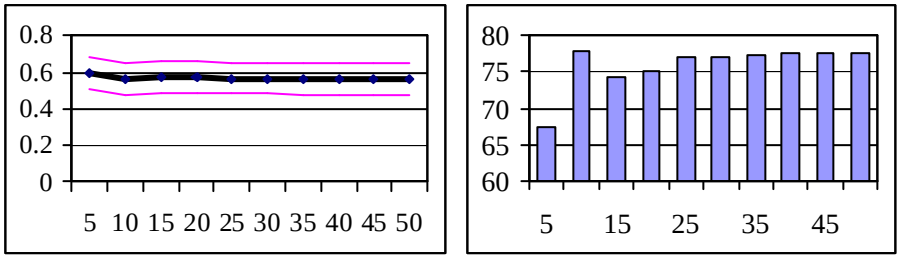

Fig. 11. Left: average amount of work as compared to blind search as a function of the abstraction step. Right: average savings in % as compared to blind search as a function of the abstraction step. An average over 20 problems with initial level of resource 1 of 50, resource 2 of 50, 7 tasks, and probability of skip action of 1

5.3 Two Resources, One Level of Abstraction

This section analyzes problems with two resources. The heuristic estimates are based on one level of abstraction. Fig. 10 shows the total number of states expanded at both base and abstract levels. The first bar corresponds to the average number of states expanded by "blind" search (about 3500). On average, the use of any abstraction step is beneficial. Unlike in the case with one resource, the smallest number of states (around 1960) gets expanded when the abstraction step of 10 is used.

Fig. 11 shows the average amount of work as compared to the "blind" search (the chart on the left) and the average savings that occur due to the use of abstraction (the chart on the right). On average, savings that occur due to abstraction are significantly higher than in the case with one resource for any abstraction step. The highest savings occur when abstraction step of 10 is used. In contrast, in the case of one resource the highest savings occur when the abstraction step of 40 is used.

6 Conclusions

We have shown that admissible heuristics can be generated using abstraction in stochastic domains. The results are very similar to those obtained by Holte *et al.* in deterministic settings [8]. In general, the use of abstraction in both stochastic and deterministic settings is beneficial. The actual amount of savings depends on the abstraction step. Holte *et al.* have found the abstraction radius to be large as compared to the size of the search space. As a result, only one level of abstraction is necessary. A similar conclusion can be made for the type of problems considered here. The experiments with two levels of abstraction and one resource show that the use of one level of abstraction results in higher savings as compared to two levels. In case of one level of abstraction and one resource, an abstraction step of 40 turns out to be the most beneficial; in case of two resources, an abstraction step of 10 is the most beneficial. In general, the amount of savings depends on the difficulty of the problem. Problems with two resources result in much higher savings as compared to the problems with one resource. We expect the savings to grow with the number of resources.

One benefit of our approach is that it is designed to avoid visiting the entire state space either during search or any preprocessing stage. Although the abstract space is created in advance, it is generated independently of the original space. Moreover, there is no need to go through the entire base-level state space in order to search the abstract space. Another source of savings is the fact that the search in the abstract space is only through reachable states, not all states.

Because it is generally much harder and less intuitive to design admissible heuristics for stochastic domains, it is beneficial to design automated techniques based on abstraction such as the one we present in this paper. Moreover, because it is relatively easy to transform LAO* into an approximation anytime algorithm, the result of this work facilitate the development of both exact and approximate algorithms for search in stochastic domains.

Acknowledgments

Support for this work was provided in part by the National Science Foundation under grant number IIS-0328601.

References

1. Barto, A.G., Mahadevan, S.: Recent Advances in Hierarchical Reinforcement Learning. *Discrete Event Dynamic Systems: Theory and Applications* **13** (2003) 41–77
2. Boutilier, C., Dearden, R.: Approximating Value Trees in Structured Dynamic Programming. In *Proc. of the Thirteenth International Conference on Machine Learning* (1996)
3. Culberson, J.C., Schaeffer, J.: Pattern Databases. *Computational Intelligence* **14(3)** (1998) 318–334
4. Dean, T., Pack Kaelbling, L., Kirman, J., Nicholson, A.: Planning with Deadlines in Stochastic Domains. In *Proc. of the Eleventh National Conference on Artificial Intelligence* (1993) 574–579

5. Dearden, R., Boutilier, C.: Abstraction and Approximate Decision-Theoretic Planning. *Artificial Intelligence* **89** (1997) 219–283
6. Hansen, E.A., Zilberstein, S.: LAO*: A Heuristic Search Algorithm that Finds Solutions with Loops. *Artificial Intelligence* **129** (1-2) (2001) 35–62
7. Holte, R.C., Drummond, C., Perez, M.B., Zimmer, R.M., MacDonald, A.J.: Searching with Abstractions: A Unifying Framework and New High-Performance Algorithm. In *Proc. of the Canadian Artificial Intelligence Conference* (1994) 263–270
8. Holte, R.C., Perez, M.B., Zimmer, R.M., MacDonald, A.J.: Hierarchical A*: Searching Abstraction Hierarchies Efficiently. In *Proc. of the Thirteenth National Conference on Artificial Intelligence* (1996) 530–535
9. Korf, R.E., Felner, A.: Disjoint Pattern Database Heuristics. *Artificial Intelligence* **134(1-2)** (2002) 9–22
10. Korf, R.E.: Finding Optimal Solutions to Rubik's Cube Using Pattern Databases. In *Proc. of the Fourteenth National Conference on Artificial Intelligence* (1997) 700–705
11. Pearl, J.: *Heuristics.* Addison-Wesley (1984)
12. Preditis, A.: Machine Discovery of Admissible Heuristics. *Machine Learning* **12** (1995) 165–175
13. St-Aubin, R., Hoey, J., Boutilier, C.: APRICODD: Approximate Policy Construction Using Decision Diagrams. *Neural Information Processing Systems* **13** (2000)
14. Tash, J., Russell, S.: Control Strategies for a Stochastic Planner. In *Proc. of the Twelfth National Conference on Artificial Intelligence* (1994) 1079–1085
15. Valtorta, M.: A New Result on the Computational Complexity of Heuristic Estimates for the A* Algorithm. *Artificial Intelligence* **55(1)** (1992) 129–142