

Symbolic Heuristic Search using Decision Diagrams

Eric Hansen¹, Rong Zhou¹, and Zhengzhu Feng²

¹ Computer Science Department, Mississippi State University, Mississippi State MS
`{hansen,rzhou}@cs.msstate.edu`,

² Computer Science Department, University of Massachusetts, Amherst MA
`fengzz@cs.umass.edu`

Abstract. We show how to use symbolic model-checking techniques in heuristic search algorithms for both deterministic and decision-theoretic planning problems. A symbolic approach exploits state abstraction by using decision diagrams to compactly represent sets of states and operators on sets of states. In earlier work, symbolic model-checking techniques have been used to find plans that minimize the number of steps needed to reach a goal. Our approach generalizes this by showing how to find plans that minimize the expected cost of reaching a goal. For this generalization, we use algebraic decision diagrams instead of binary decision diagrams. In particular, we show that algebraic decision diagrams provide a compact representation of state evaluation functions. We describe symbolic generalizations of A* search for deterministic planning and of LAO* search for decision-theoretic planning problems formalized as Markov decision processes. We report experimental results and discuss issues for future work.

1 Introduction

There is currently great interest in using symbolic model-checking to solve AI planning problems with large state spaces [1–8]. This interest is based on recognition that the problem of finding a plan (i.e., a sequence of actions and states leading from an initial state to a goal state) can be treated as a reachability problem in model checking. The planning problem is solved symbolically in this sense: reachability analysis is performed by manipulating sets of states, rather than individual states. Sets of states (and operators on sets of states) are represented compactly using *decision diagrams*. This approach exploits state abstraction to solve problems with large state spaces. For problems with regular structure, decision diagrams can often provide a polynomial representation of an exponential number of states. A symbolic approach to model-checking has made it possible to verify systems with more than 10^{30} states, and a similar approach to scaling up planning algorithms shows much promise.

Symbolic model-checking techniques have been explored for both deterministic and nondeterministic planning. However, the algorithms that have been developed only find plans that are “optimal” in the sense that they minimize the

number of actions needed to reach a goal. The more general problem of finding minimal-cost plans, where there is a varying or state-dependent cost for actions, has not yet been addressed (or has not been addressed adequately). We argue that this follows, in part, from the use of binary decision diagrams in previous work. We adopt algebraic decision diagrams as a more general data structure for symbolic heuristic search. Whereas binary decision diagrams represent Boolean functions, algebraic decision diagrams can represent arbitrary real-valued functions. We show that this allows us to develop algorithms for deterministic and decision-theoretic planning that can find minimal-cost plans.

The paper is organized as follows. In section 2, we review the framework of state-space heuristic search and present some relevant background for a symbolic approach to heuristic search. We review binary decision diagrams and algebraic decision diagrams. We also review previous work in which decision diagrams are used to support state abstraction in AI planning. In section 3, we describe a symbolic generalization of A* that can be used to solve deterministic planning problems. In section 4, we describe a symbolic generalization of LAO*, a heuristic search algorithm for solving decision-theoretic planning problems formalized as Markov decision processes. We present experimental results for both algorithms. We conclude the paper with a comparison of the two algorithms and a discussion of issues for future work.

2 Background

2.1 State-space search

We begin by reviewing the framework of state-space search and introduce some notation that will be used in the rest of the paper.

A state-space search problem is defined as a tuple (S, G, s^0, A, T, c) , where S denotes a set of states; $G \subset S$ denotes a set of goal states; s^0 denotes a start state; A denotes a set of actions (or operators); T denotes a set of transition models $\{T^a\}$, one for each action $a \in A$, describing the state transitions caused by each action; and c denotes a set of cost models $\{c^a\}$, one for each action, specifying the state-dependent (expected) cost of actions.

We consider two kinds of transition model in this paper. In a deterministic transition model, an action $a \in A$ in state $s \in S$ always leads to the same successor state $s' \in S$. We can represent this by a function $T^a : S \times S \rightarrow \{0, 1\}$, where the value of the function is 1 when action a causes a transition from state s to state s' . In a stochastic transition model, an action $a \in A$ in state $s \in S$ has several possible successor states, each occurring with some probability. We can represent this by a function $T^a : S \times S \rightarrow [0, 1]$, where the value of the function is the probability that action a causes a transition from state s to state s' . The first kind of state-space search problem is a deterministic shortest-path problem. The second is a special kind of Markov decision process called a stochastic shortest-path problem [9]. Both can be solved by dynamic programming. But given a start state $s^0 \in S$ and an admissible heuristic function $h : S \rightarrow \Re$, an optimal solution

can also be found by a heuristic search algorithm that gradually expands a partial solution, beginning from the start state and guided by the heuristic, until a complete solution is found. The heuristic search approach has the advantage that it only needs to evaluate part of the state space to find an optimal solution. We consider two heuristic search algorithms in this paper; the A* algorithm for deterministic planning problems [10] and the LAO* algorithm for stochastic planning problems [11]. The contribution of this paper is to describe how both of these algorithms can exploit state abstraction using decision diagrams.

2.2 Factored state representation

To use decision diagrams for state abstraction, we adopt a binary encoding of states. We assume the set of states S is described by a set of Boolean variables $\mathbf{X} = \{X_1, \dots, X_n\}$, where $\mathbf{x} = \{x_1, \dots, x_n\}$ denotes a particular instantiation of the variables, corresponding to a unique state $s \in S$. In the planning literature, such Boolean variables are called *fluents*. In the literature on decision-theoretic planning, a state space structured in this way is called a *factored* Markov decision process. Thus we represent the transition and cost models of a state-space search problem as a mapping defined over a Boolean encoding of the state space. For example, we represent the transition model of a planning problem by the function $T^a(\mathbf{X}, \mathbf{X}')$, where $\mathbf{X} = \{X_1, \dots, X_n\}$ refers to the set of state variables before taking action a and $\mathbf{X}' = \{X'_1, \dots, X'_n\}$ refers to the set of state variables after taking the action. Similarly, we represent the cost model by the function $c^a(\mathbf{X})$. Although the set of states $S = 2^{\mathbf{X}}$ grows exponentially with the number of variables, decision diagrams can exploit state abstraction to represent these models compactly.

2.3 Decision diagrams

The decision diagram data structure was developed by the symbolic model-checking community for application to VLSI design and verification [12, 13]. A decision diagram is a compact representation of a mapping from a set of Boolean state variables to a set of values. A binary decision diagram (BDD) represents a mapping to the values 0 or 1. An algebraic decision diagram (ADD) represents a more general mapping to any discrete set of values (including real numbers) [14].

A decision diagram is a directed acyclic graph with internal nodes of out-degree two (corresponding to Boolean variables) and a terminal node for each value of the function it represents. In an ordered decision diagram, the Boolean variables in all paths through the graph respect a linear order $X_1 \prec X_2 \prec \dots \prec X_n$. An ordered decision diagram can be reduced by successive applications of two rules. The *deletion rule* eliminates a node with both of its outgoing edges leading to the same successor. The *merging rule* eliminates one of two nodes with the same label as well as the same pair of successors. A reduced decision diagram with a fixed variable ordering is a canonical representation of a function. The canonicity of decision diagrams gives rise to the existence of efficient algorithms for manipulating them. Because the complexity of operators on decision diagrams

depends on the number of nodes in the diagrams and not on the size of the state space, decision diagrams can exploit regularity in the model to solve problems with large state spaces more efficiently.

A symbolic approach to both model checking and heuristic search manipulates sets of states, instead of individual states. A set of states S is represented by its characteristic function χ_S , so that $s \in S \iff \chi_S(\mathbf{X}) = 1$. The characteristic function is represented compactly by a BDD. (From now on, whenever we refer to a set of states, S , we implicitly refer to its characteristic function, represented by a decision diagram.) In both model checking and heuristic search, a central task is to determine the set of states that is reachable from an initial state. This can be achieved through a simple breadth-first search starting from the initial state, until no more new states can be added. Let S^i denote the set of states reachable from the initial state s^0 in i steps, initialized by $S^i = \{s^0\}$. There is an efficient algorithm for determining the next step characteristic functions $\chi_{S^{i+1}}(\mathbf{X})$ given the current characteristic function $\chi_{S^i}(\mathbf{X})$. Recall that $T(\mathbf{X}, \mathbf{X}')$ is true if and only if \mathbf{X} is the encoding of a given state and \mathbf{X}' is the encoding of its successor state. The next-step characteristic function is computed as follows:

$$\chi_{S^{i+1}}(\mathbf{X}') = \exists_{\mathbf{X} \in \mathbf{X}} (\chi_{S^i}(\mathbf{X}) \wedge T(\mathbf{X}, \mathbf{X}')) \quad (1)$$

Equation (1) is called the *relational product* operator. It represents a series of nested existential quantifications, one for each variable in \mathbf{X} . The conjunction $T^a(\mathbf{X}, \mathbf{X}') \wedge \chi_S(\mathbf{X})$ selects the set of valid transitions and the existential quantification extracts and unions the successor states together. Given $\chi_{S^0}(\mathbf{X})$, we can iteratively compute the characteristic function $\chi_{S^1}(\mathbf{X}), \chi_{S^2}(\mathbf{X}), \chi_{S^3}(\mathbf{X}), \dots$, until the transitive closure of the transition relation is computed. Both the relational-product operator and symbolic traversal algorithms are well studied in the symbolic model checking literature, and we refer to that literature for further details [15].

2.4 Deterministic planning

We are not the first to explore the use of decision diagrams in heuristic search. Edelkamp and Reffel [3] describe a symbolic generalization of A*, called BDDA*, that can solve deterministic planning problems. It uses BDDs to represent sets of states and operators on sets of states in a compressed form that exploits structure in the problem domain. They show that symbolic search guided by a heuristic significantly outperforms breadth-first symbolic search. They also show that this approach is effective in solving a class of deterministic planning problems [4].

In Section 3, we describe an alternative implementation of symbolic A* that uses ADDs. Because BDDA* uses BDDs as a data structure, the cost-functions c^a , the state evaluation function f , the heuristic evaluation function h , and the open list, must all be encoded in binary. Although it is possible to encode anything in binary, this makes the implementation of BDDA* awkward in several respects. In fact, the original version of BDDA* could only solve problems for which all actions have unit cost and the objective is to find the *shortest* plan.

Edelkamp [16] later described a more general implementation of BDDA* does not assume unit-cost actions. However, we argue that ADDs provide a more natural representation for such problems.

2.5 Nondeterministic and decision-theoretic planning

Decision diagrams have also been used for nondeterministic and decision-theoretic planning. Nondeterministic planning considers actions with multiple outcomes and allows plan execution to include conditional and iterative behavior. However, it does not associate probabilities and costs with state transitions, as in decision-theoretic planning. In decision-theoretic planning, the problem is not simply to construct a plan that can reach the goal, but to find a plan that minimizes expected cost (or equivalently, maximizes expected value).

There has been substantial work on using BDDs for nondeterministic planning [1, 2]. There has also been work on using ADDs to exploit state abstraction in dynamic programming for MDPs [7, 8]. In Section 4, we show how to integrate these two approaches to create a symbolic heuristic search algorithm that can solve decision-theoretic planning problems formalized as MDPs.

3 Deterministic planning: Symbolic A*

We first describe an implementation of A* that uses ADDs instead of BDDs as a data structure. We call this algorithm ADDA*, in contrast to Edelkamp and Reffel’s BDDA* algorithm. We show that this change of data structure leads to a simpler and more natural implementation of symbolic A* that can solve cost-minimization problems.

3.1 Algorithm

In the preceding section, we reviewed how symbolic model-checking techniques are used to compute the set of states reachable from a start state. For reachability analysis, it is sufficient to compute successor states. But for a heuristic search algorithm such as A*, we also need to compute and store state costs. We begin by pointing out that an algebraic decision diagram provides a symbolic representation of a real-valued function, and thus a more natural representation for the cost function, c , heuristic function h , and state evaluation function f , of A*. No binary encoding of these real-valued functions is necessary, as in BDDA*. As we will show, this simplifies our implementation of A*, without sacrificing the advantages of a symbolic representation.

Recall that A* stores the search frontier in a list called *OPEN*, which organizes nodes in increasing order of their f -cost. The node with the least f -cost is selected to generate all its successor nodes, a process known as a node expansion. Note that the *OPEN* list for A* can be viewed as a function that maps states to costs. If the f -cost of many nodes are the same, then a considerable degree of state abstraction can be achieved by treating states with the same f -cost as

an aggregate or abstract state. This is the approach adopted by BDDA* and it is the same approach we adopt for ADDA*. It makes it possible to perform node expansion symbolically by expanding a set of states (with the same f -cost), instead of a single state at a time.

BDDA* represents the *OPEN* list as a set of BDDs, one for each distinct f -cost. Each BDD represents the characteristic function of a set of states with the same f -cost. By contrast, ADDA* represents the open list as a single ADD, where the leaves of the ADD correspond to the distinct f -costs in the open list. This representation is more natural, and as we will show, it makes it easier to implement the rest of the A* algorithm.

When we expand a set of states in A*, we determine their successor states and compute their g and f -costs. In graph search, we also need to detect states that represent duplicate paths and preserve the one with the least g -cost. Edelkamp and Reffel’s original implementation of A* determines the set of successor states separately from computing their g and f -costs, and does not detect duplicate nodes.¹ Using ADDs instead of BDDs as a data structure makes it possible to do all of these things at the same time.

But to do so, we need to generalize the relational product operator defined for BDDs so that it also works with ADDs. The standard operator cannot be used with ADDs because the terminals of ADDs correspond to real values, and not necessarily the Boolean constants $\{0, 1\}$. In Boolean algebra, the relational product is computed by using existential quantification given by the following formula:

$$\exists_X f(\mathbf{X}) = f(\mathbf{X})|_{X=0} \vee f(\mathbf{X})|_{X=1} \quad (2)$$

A naive way of adapting existential quantification to ADDs is to substitute the algebraic addition operator (+) for the Boolean disjunction operator (\vee) in equation (2). However, this won’t generalize to the case of graph search. In graph search, there may be more than one path from the start state to a given state. Only one of these paths should be preserved and the minimal-cost path is preferred. But if the naive approach to adapting existential quantification to ADDs is used, the cost of a successor state would be the sum of the costs of all duplicate paths to that state! Because this does not make sense, we invented a new operator called *existential least-cost quantification*. It is defined by the following equation:

$$\exists_X^{LC} f(\mathbf{X}) = \min(f(\mathbf{X})|_{X=0}, f(\mathbf{X})|_{X=1}) \quad (3)$$

The intuitive meaning of the existential least-cost quantification operator is to preserve only the path that has the least cost to a given state, discarding any suboptimal paths.

In order to make existential least-cost quantification work seamlessly with the transition function T , we change the definition of the latter as follows: $T(\mathbf{X}, \mathbf{X}')$

¹ Edelkamp [16] describes a later implementation of BDDA* that does detect duplicate nodes using a technique called *forward set simplification*, although this technique can only detect duplicate paths to nodes that are already expanded.

```

procedure ADDA* ( $\chi_{S^0}, \chi_{S^g}$ )
1.  $OPEN(\mathbf{X}, f) \leftarrow (\chi_{S^0}, h(\chi_{S^0})) \wedge (\neg\chi_{S^0}, \infty)$ 
2. while  $OPEN \neq \emptyset$  do
3.    $\mathcal{B} \leftarrow \{\beta \mid (\beta, n \in OPEN) \wedge (f(\beta) = \min f(n))\}$ 
4.   if  $\exists_{X \in \mathbf{X}}(\mathbf{X}(\mathcal{B}) \wedge \chi_{S^g})$  then return  $f(\mathcal{B})$ 
5.    $OPEN^- \leftarrow OPEN \setminus \mathcal{B}$ 
6.    $\mathcal{G}(\mathbf{X}, g) \leftarrow (\mathbf{X}(\mathcal{B}), f(\mathcal{B}) - h(\mathcal{B}))$ 
7.    $\mathcal{G}(\mathbf{X}', g) \leftarrow \exists_{X \in \mathbf{X}}^{LC} \bigcup_a (\mathbf{X}(\mathcal{G}) * T^a(\mathbf{X}, \mathbf{X}') + c^a(\mathbf{X}))$ 
8.    $\mathbf{X}(\mathcal{G}) \leftarrow SwapVar_{\mathbf{X}', \mathbf{X}}(\mathbf{X}'(\mathcal{G}))$ 
9.    $OPEN^+ \leftarrow (\mathbf{X}(\mathcal{G}), g(\mathcal{G}) + h(\mathbf{X}(\mathcal{G})))$ 
10.   $OPEN \leftarrow \min(OPEN^-, OPEN^+)$ 

```

Table 1. Pseudocode for ADDA*

maps to constant 1 if and only if \mathbf{X} is the encoding of a given state and \mathbf{X}' is the encoding of its successor state; otherwise, it maps to an infinity (∞) constant. This modification of the transition function T is necessary in order to make the cost of any illegal move infinity. As a consequence, all illegal moves will be ignored.

In order to detect duplicate paths in graph search and preserve the one with the least g -cost, ADDA* performs existential least-cost quantification on an ADD that represents the g -costs of the set of states being expanded. First it selects the set of states to expand by selecting those with the least f -cost. ADDA* computes the ADD representing the g -cost of the states by subtracting the heuristic estimate from their f -costs, as follows: $g(\mathbf{X}) = f(\mathbf{X}) - h(\mathbf{X})$. Then ADDA* uses least-cost existential quantification to compute the successors of this set of states and their g -costs. The f -costs of the successor states after taking action a are computed as follows:

$$\begin{aligned}
 f^a(\mathbf{X}') &= g(\mathbf{X}') + h(\mathbf{X}') \\
 &= g(\mathbf{X}) + c^a(\mathbf{X}) + h(\mathbf{X}') \\
 &= f(\mathbf{X}) - h(\mathbf{X}) + c^a(\mathbf{X}) + h(\mathbf{X}'),
 \end{aligned}$$

and the successor states for each action are inserted into the open list.

Figure 1 shows the pseudocode of ADDA*. Let χ_{S^0}, χ_{S^g} denote the characteristic function of the starting node(s) and goal node(s), respectively. Initially, $OPEN$ contains a single starting node with its f value set to the heuristic estimate h and the f value for everything else is set to infinity. Then the algorithm finds the set of nodes with the least f value, \mathcal{B} . Once the intersection between the set \mathcal{B} and the goal set χ_{S^g} is not the trivial zero function, this indicates a goal node has been found. If no goal node is found, the set \mathcal{B} is detached from the $OPEN$ and the g values of node $\in \mathcal{B}$ are computed and stored in \mathcal{G} . Since the transition function T has a value of 1 for any legal move, which means after the product of “ $\mathbf{X}(\mathcal{G})$ ” and “ $T(\mathbf{X}, \mathbf{X}')$ ” in line# 7, the g value of all legal moves

will be preserved, while the g value of all illegal moves will be set to infinity and discarded later. Furthermore, according to the definition of existential least-cost quantification, among all the legal moves, only the move that produces the least-cost path will be kept. Which means for the successors of the nodes with the minimum f value, only the best (shortest) paths are stored in the *OPEN*. The last line (line# 10) adds the successor nodes back to the original *OPEN*. The purpose of using a “min” operator is to check for duplicate paths between the set of newly generated successors nodes and the set of old nodes in *OPEN*. We note that the data structure we refer to as *OPEN* is the union of the open and closed lists of A* and contains all states evaluated by A*. Although the open and closed lists can be represented by distinct data structures in symbolic A*, we found that it is more memory-efficient to represent them by a single ADD.

3.2 Experimental Results

Problem	Size		Time	
	BDDBFS	ADDA*	BDDBFS	ADDA*
8-puzzle	28096	5465	8.91	3.89
15-puzzle	Unsolvable	548546	Unsolvable	902.85

Table 2. Performance comparison of ADDA* and breadth-first search (BDDBFS).

We implemented ADDA* and compared its performance to a symbolic breadth-first search algorithm implemented using BDDs, called BDD-BFS.² We tested both algorithms on 200 randomly generated instances of the Eight Puzzle and the Fifteen Puzzle. (For the Fifteen Puzzle, we only considered instances with solution depth up to 40.) As shown in Table 2, ADDA* stores about one fifth as many nodes as BDD-BFS and is more than twice as fast in solving the Eight Puzzle. None of the Fifteen Puzzle instances could be solved by BDD-BFS. The performance of ADDA* on the Fifteen puzzle is shown in the table. We note that these results are consistent with the results reported by Edelkamp and Reffel for BDDA* in solving the Eight and Fifteen Puzzles. Our current algorithm does not seem to be more or less efficient than theirs. However, it is simpler and more general, by virtue of being implemented using ADDs instead of BDDs.

We also compared ADDA* and ordinary A* in solving the Fifteen Puzzle. On average, A* generates 434,282 nodes. Although this seems less than the number of nodes generated by ADDA* (548,546), a node in ADDA* is not the same as a node in A*. A decision diagram node can only represent a single bit (true

² The decision diagram package used to implement our algorithm is the CUDD package from University of Colorado [17]. We modified the package to support the existential least-cost quantification operator. Experiments were performed on a Sun UltraSPARC II with a 300MHz processor and 2 gigabytes of memory.

or false) in the encoding of a Fifteen Puzzle state; whereas a node in A* can represent the entire state of the board. In our implementations, a node in A* takes roughly two and a half times the physical memory required for a node in ADDA* and we found that the amount of physical memory saved using ADDA* instead of ordinary A* is slightly more than 50%. This reflects the benefit of state abstraction. In solving sliding-tile puzzles, however, there is not sufficient state abstraction to compensate for the significant overhead involved in using decision diagrams and ordinary A* is faster than ADDA*.

Our experimental results for ADDA* are very preliminary. There are several ways in which its performance may be improved, and we plan to test it on a range of different problems. We discuss some of the possibilities for future work in the conclusion of the paper.

4 Decision-theoretic planning: Symbolic LAO*

We now turn to a more complex class of planning problems in which state transitions are stochastic, and, as a result, plans have a more complex structure that includes branches and loops. Previous authors have described how to use symbolic reachability analysis to solve nondeterministic planning problems. But nondeterministic planning does not consider the probability of state transitions or their cost. Hoey et al. [7] have described a symbolic dynamic programming algorithm for Markov decision processes. It does consider the probability of state transitions and their cost, and uses ADDs to aggregate states with the same value – the same approach to state abstraction we used for A*. However, their SPUDD planner is a dynamic programming algorithm that solves the problem for all possible starting states, without considering reachability. In the rest of this paper, we describe a heuristic search approach to solving this class of problems that integrates both approaches – reachability analysis and dynamic programming.

4.1 Algorithm

Our algorithm is a symbolic generalization of the LAO* algorithm, a heuristic search algorithm for stochastic shortest-path problems (and other Markov decision problems) [11]. Given a start state, LAO* finds an optimal policy that can include cycles and reaches the goal state with probability one. It is interesting to note that a policy that reaches the goal state with probability one can be viewed as a decision-theoretic generalization of the concept of a *strong cyclic plan* in nondeterministic planning [18].

LAO* is an extension of the classic search algorithm AO* [19]. Like AO*, it has two alternating phases. First, it expands the best partial solution (or policy) and evaluates the states on its fringe using a heuristic evaluation function. Then it performs dynamic programming on the states visited by the best partial solution, to update their values and possibly revise the best current solution. The two phases alternate until a complete solution is found, which is guaranteed to be optimal. LAO* differs from AO* by allowing solutions with loops, requiring it

```

procedure solutionExpansion( $\pi, \chi_{S^0}, G$ )
1.  $E \leftarrow F \leftarrow \phi$ 
2.  $from \leftarrow \chi_{S^0}$ 
3. repeat
4.    $to \leftarrow \bigcup_a \exists X \in \mathbf{X} ((from \cap \chi_{S^a_\pi}) \wedge T^a(\mathbf{X}, \mathbf{X}'))$ 
5.    $F \leftarrow F \cup (to - G)$ 
6.    $E \leftarrow E \cup from$ 
7.    $from \leftarrow to \cap G - E$ 
8. until ( $from = \phi$ )
9.  $E \leftarrow E \cup F$ 
10.  $G \leftarrow G \cup F$ 
11. return ( $E, F, G$ )

procedure dynamicProgramming( $E, f$ )
12.  $savef \leftarrow f$ 
13.  $E' \leftarrow \bigcup_a \exists X \in \mathbf{X} (E \wedge T^a(\mathbf{X}, \mathbf{X}'))$ 
14. repeat
15.    $f' \leftarrow f$ 
16.   FOR each action  $a$ 
17.      $f^a \leftarrow c^a_E + \sum_{E'} T^a_{E \cup E'} f'_{E'}$ 
18.    $M \leftarrow \min_a f^a$ 
19.    $f \leftarrow M \cup savef_{\bar{E}}$ 
20.   residual  $\leftarrow \|f_E - f'_E\|$ 
21. until stopping criterion met
22.  $\pi \leftarrow extractPolicy(M, \{f^a\})$ 
23. return ( $f, \pi, residual$ )

procedure LAO*( $\{T^a\}, \{c^a\}, \chi_{S^0}, h, threshold$  )
24.  $f \leftarrow h$ 
25.  $G \leftarrow \phi$ 
26.  $\pi \leftarrow 0$ 
27. repeat
28.    $(E, F, G) \leftarrow solutionExpansion(\pi, \chi_{S^0}, G)$ 
29.    $(f, \pi, residual) \leftarrow dynamicProgramming(E, f)$ 
30. until ( $F = \phi$ ) and (residual  $\leq$  threshold)
31. return ( $\pi, f, E, G$ ).

```

Table 3. Symbolic LAO* algorithm.

to use a more general dynamic programming algorithm such as value iteration or policy iteration. To create a symbolic generalization of LAO*, we will implement the first phase of the algorithm using a variant of symbolic reachability analysis and the second phase using a variant of the SPUDD algorithm.

To integrate the two phases of LAO* – reachability analysis and dynamic programming – we introduce the concept of *masking*. To motivate this idea, we note that all elements manipulated by our symbolic generalization of LAO* are represented by ADDs (including the transition and cost models, the policy π :

$S \rightarrow A$, the state evaluation function $f : S \rightarrow \mathfrak{R}$, the admissible heuristic $h : S \rightarrow \mathfrak{R}$, etc.), and all computations are performed using ADDs. A potential problem is that an ADD assigns a value to every state in the state space. However, we want to limit computation to the set of states that are reachable from the start state by following the best policy. To focus computation on the relevant state values, we introduce the idea of *masking* an ADD.

Given an ADD D and a set of relevant states U , masking is performed by multiplying D by χ_U , which is the characteristic function of U . This has the effect of mapping all irrelevant states to the value zero. We let D_U denote the resulting *masked ADD*. Mapping all irrelevant states to zero can simplify an ADD considerably. If the set of reachable states is small, the masked ADD often has dramatically fewer nodes. Since the complexity of computations using ADDs is a function of the size of the ADDs, this can dramatically improve the efficiency of computation using ADDs.

The first phase of LAO* uses symbolic reachability analysis to determine the set of relevant states. The second phase updates the state evaluation function for these states only, using the technique of masking to focus computation on the relevant states. The algorithm maintains two kinds of global data structure; characteristic functions for sets of states and value functions for sets of states. The first are used to mask the second.

We summarize symbolic LAO* in Table 3, and give a more detailed explanation of its alternating phases in the following.

Solution expansion In the solution expansion step of the algorithm, we perform reachability analysis to find the set of states F that are not in G (i.e., have not been “expanded” yet), but are reachable from the set of start states, S^0 , by following the partial policy π_G . These states are on the “fringe” of the states visited by the partial policy. We add them to G and add them to the set of states $E \subseteq G$ that are visited by the current partial policy. This is analogous to “expanding” states on the frontier of a search graph in heuristic search. It expands the partial policy in the sense that the policy will be defined for a larger set of states in the dynamic-programming step. We perform this reachability analysis using the same relational product operator reviewed in Section 2 (but not one one that uses least-cost existential quantification).

We note that our symbolic implementation of LAO* does not maintain an explicit search graph. It is sufficient to keep track of the set of states that have been “expanded” so far, denoted G , the *partial value function*, denoted f_G , and a *partial policy*, denoted π_G . For any state in G , we can “query” the policy to determine its associated action, and compute its successor states. Thus, the graph structure is implicit in this representation.

Because a policy is associated with a set of transition functions, one for each action, we need to invoke the appropriate transition function for each action when computing successor states under a policy. For this, it is useful to represent the partial policy π_G in a different way. We associate with each action a the set of states for which the best action to take is a under the current policy, and

call this set of states $\chi_{S_\pi^a}$. Obviously we have $\chi_{S_\pi^a} \cap \chi_{S_\pi^{a'}} = \phi$ for $a \neq a'$, and $\cup_a \chi_{S_\pi^a} = G$. Given this representation of the policy, line 4 in Table 3 computes the set of successor states following the current policy using the *image* operator.

Dynamic programming The dynamic-programming step of LAO* is performed using a modified version of the SPUDD algorithm. The original SPUDD algorithm performs dynamic programming over the entire state space. We modify it to focus computation on reachable states, using the idea of masking. Masking lets us perform dynamic programming on a subset of the state space instead of the entire state space.

The pseudocode in Table 3 assumes that DP is performed on E , the states visited by the best (partial) policy, although a larger or smaller set of states can be updated by LAO* [11]. Because π_G is a partial policy, there can be states in E with successor states that are not in G , denoted E' . This is true until LAO* converges. In line 13, we compute these states in order to do appropriate masking. To perform dynamic programming on the states in E , we assign admissible values to the "fringe" states in E' , where these values come from the current value function. Note that the value function is initialized to an admissible heuristic evaluation function at the beginning of the algorithm.

With all components properly masked, we can perform dynamic programming using the SPUDD algorithm. This is summarized in line 17. The full equation is

$$f^a(\mathbf{X}) = c_E^a(\mathbf{X}) + \sum_{E'} T_{E \cup E'}^a(\mathbf{X}, \mathbf{X}') \cdot f_{E'}^a(\mathbf{X}').$$

The masked ADDs c_E^a and $T_{E \cup E'}^a$ need to be computed only once for each call to *valueIteration()* since they don't change between iterations. Note that the product $T_{E \cup E'}^a \cdot f_{E'}^a$ is effectively defined over $E \cup E'$. After the summation over E' , which is accomplished by existentially abstracting away all post-action variables, the resulting ADD is effectively defined over E only. As a result, f^a is effectively a masked ADD over E , and the maximum M at line 18 is also a masked ADD over E .

The residual in line 20 is computed by finding the largest absolute value of the ADD ($f_E - f_{E'}^a$). We use the masking subscript here to emphasize that the residual is computed only for states in the set E . Dynamic programming is the most expensive step of LAO*, and it is not necessary to run it until convergence each time this step is performed. Often a single iteration gives the best performance. We extract a policy in line 22 by comparing M against the action value function f^a (breaking ties arbitrarily): $\forall s \in E \ \pi(s) = a$ if $M(s) = f^a(s)$.

Convergence test At the beginning of LAO*, the value function f is initialized to the admissible heuristic h . Each time the value iteration is performed, it starts with the current values of f . Hansen and Zilberstein (2001) show that these values increase monotonically in the course of the algorithm; are always

Example	Reachability Results				Size Results				Timing Results				
	$ S $	$ A $	$ E $	$ G $	reach	LAO*		SPUDD		LAO*		SPUDD	
					nodes	leaves	nodes	leaves	expand	DP	total	total	
r1	2^{15}	20	134	413	2^{15}	65	12	1288	406	0.24	17	17	539
r2	2^{20}	25	3014	3281	2^{20}	181	19	15758	4056	0.46	54	544	12774
r3	2^{20}	30	10442	33322	2^{20}	6240	2190	9902	4594	57.71	1679	1738	10891
r4	2^{35}	30	383	383	2^{35}	77	4	NA	NA	0.05	7	7	> 20hr

Table 4. Performance comparison of LAO* and SPUDD.

admissible; and converge arbitrarily close to optimal. LAO* converges to an optimal or ϵ -optimal policy when two conditions are met: (1) its current policy does not have any unexpanded states, and (2) the error bound of the policy is less than some predetermined threshold. Like other heuristic search algorithms, LAO* can find an optimal solution without visiting the entire state space. The convergence proofs for the original LAO* algorithm carry over in a straightforward way to symbolic LAO*.

4.2 Experimental results

Table 4 compares the performance of symbolic LAO* and SPUDD on four randomly-generated MDPs. Because the performance of LAO* depends on the starting state, our results for LAO* are averaged over 50 random starting states.

A simple admissible heuristic function was created by performing ten iterations of an approximate value iteration algorithm similar to APRICODD [8] on an initial admissible value function created by assuming the maximum reward is received each step. The first few iterations are very fast because the ADD representing the value function is compact, especially when approximation is used.

LAO* achieves its efficiency by focusing computation on a subset of the state space. The column labeled *reach* shows the average number of states that can be reached from the starting state, by following any policy. The column labelled $|G|$ is important because it shows the number of states “expanded” by LAO*. These are states for which a backup is performed at some point in the algorithm, and this number depends on the quality of the heuristic. The better the heuristic, the fewer states need to be expanded before finding an optimal policy. The gap between $|E|$ and *reach* reflects the potential for increased efficiency using heuristic search, instead of simple reachability analysis.

The columns labeled “nodes” and “leaves”, under LAO* and SPUDD respectively, compare the size of the final value function returned by LAO* and SPUDD. The columns under “nodes” gives the number of nodes in the respective value function ADDs, and the columns under “leaves” give the number of leaves. Because LAO* focuses computation on a subset of the state space, it finds a much more compact solution (which translates into increased efficiency).

The last four columns compare the running times of LAO* and SPUDD. Timing results are measured in CPU seconds. The total running time of LAO* is broken down into two parts; the column “expand” shows the average time for policy expansion and the column “DP” shows the average time for value iteration. These results show that value iteration consumes most of the running time. This is in keeping with a similar observation about the original LAO* algorithm for flat state spaces. The time for value iteration includes the time for masking. For this set of examples, masking takes between 0.5% and 2.1% of the running time of value iteration. The final two columns show that LAO* is much more efficient than SPUDD in solving these examples. This is to be expected since LAO* solves the problem for only part of the state space. Nevertheless, it demonstrates the power of using heuristic search to focus computation on the relevant part of the state space. The running time of LAO* is correlated with $|G|$, the number of states expanded during the search, which in turn is affected by the starting state, the reachability structure of the problem, and the accuracy of the heuristic function.

We refer to a longer paper for a more detailed description of the symbolic LAO* algorithm and more extensive experimental results [20].

5 Conclusion and future work

We have described symbolic generalizations of A* and LAO* heuristic search that use decision diagrams to exploit state abstraction. In showing how to implement A* using ADDs, we introduced a new ADD operator called least-cost existential quantification. This operator makes it possible to simultaneously compute the successors of a set of states, update their g-values, and detect and remove duplicate paths in graph search. Our symbolic generalization of LAO* integrates a reachability analysis algorithm adapted from symbolic model checking with the SPUDD dynamic programming algorithm for factored MDPs. To integrate these two approaches, we introduced the concept of *masking* an ADD. Masking provides a way to focus computation on the relevant parts of the state space – the hallmark of heuristic search.

We have described work in progress. Our contribution is to show how to use symbolic model-checking techniques for planning problems that require cost minimization. More work needs to be done to improve the performance of these algorithms, before we can fully evaluate this approach. Among the questions we are continuing to investigate, we highlight the following.

- Current techniques require encoding the problem state using Boolean variables. Different encodings may lend themselves more easily to state abstraction. Thus, an important question is how to find the best encoding. Another question is whether recent extensions of decision diagrams that allow non-Boolean variables could be useful.
- The bottleneck of algorithms that use decision diagrams is the need to represent the full transition relation, especially when computing the successors of a set of states. To address this problem, the model-checking community

has developed techniques for *partitioning* the transition relation. These allow significant improvement in efficiency, and adapting these techniques to our more general search algorithms is likely to improve their performance too. Another possibility worth exploring is whether the transition relation can be represented procedurally, as is usually done for deterministic state-space search.

- The symbolic approach exploits the structure of a problem to create useful state abstractions. But not all problems have the kind of structure that can be exploited by these techniques. Can we characterize the kind of problem structure for which this approach works well?
- The approach to state abstraction adopted in this paper is to aggregate states with the same value. It may be possible to develop approximation algorithms that aggregate states that have a similar value, achieving greater state abstraction in exchange for a bounded decrease in solution quality [8].

Presenting our symbolic generalizations of A* and LAO* in the same paper allows us to compare and contrast their performance. This leads to an interesting observation. At present, our symbolic implementation of A* cannot solve problems of nearly the same size as single-state A* can solve. Even the fifteen puzzle presents a challenge. By contrast, our symbolic implementation of LAO* can efficiently solve factored MDPs that are as large or larger than any reported in the literature. Given this observation, we find it interesting that the problems solved by both algorithms in our experiments have roughly the same size state space! (In fact, a Boolean encoding of the state of the Fifteen Puzzle has 64 variables, almost twice as large as the largest factored MDP considered in the literature.) In our view, this underscores the fact that work on A* search is very mature and techniques have been developed that allow quite large problems to be solved. By contrast, work on decision-theoretic planning is relatively immature and test problems are still quite small.

We hope that by looking at these two classes of problems together, we will be able to use techniques that have proved successful for one in order to improve our ability to solve the other.

Acknowledgements

This research was supported in part by NSF grant IIS-9984952 and NASA grant NAG-2-1463. We thank the developers of the SPUDD planner [7] for making their code available.

References

1. Cimatti, A., Roveri, M., Traverso, P.: Automatic OBDD-based generation of universal plans in non-deterministic domains. In: Proceedings of the Fifteenth National Conference on Artificial Intelligence. (1998) 875 – 881
2. Cimatti, M., Roveri, M.: Conformant planning via symbolic model checking. Journal of Artificial Intelligence Research **13** (2000) 305–338

3. Edelkamp, S., Reffel, F.: OBDDs in heuristic search. In: German Conference on Artificial Intelligence (KI). (1998) 81–92
4. Edelkamp, S., Reffel, F.: Deterministic state space planning with BDDs. In: Proceedings of the 5th European Conference on Planning (ECP-99). (1999) 381–2
5. Jensen, R., Veloso, M.: OBDD-based universal planning for synchronized agents in non-deterministic domains. *Journal of Artificial Intelligence Research* **13** (2000) 189–226
6. Jensen, R.: OBDD-based deterministic planning using the UMOP planning framework. In: Proceedings of the AIPS-00 Workshop on Model-Theoretic Approaches to Planning. (2000) 26–31
7. Hoey, J., St-Aubin, R., Hu, A., Boutilier, C.: SPUDD: Stochastic planning using decision diagrams. In: Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence. (1999) 279–288
8. St-Aubin, R., Hoey, J., Boutilier, C.: APRICODD: Approximate policy construction using decision diagrams. In: Proceedings of NIPS-2000. (2000)
9. Bertsekas, D.: *Dynamic Programming and Optimal Control*. Athena Scientific, Belmont, MA (1995)
10. Hart, P., Nilsson, N., Raphael, B.: A formal basis for the heuristic determination of minimum cost paths. *IEEE Trans. Syst. Science and Cybernetics (SSC-4)* 100 – 107
11. Hansen, E., Zilberstein, S.: LAO*: A heuristic search algorithm that finds solutions with loops. *Artificial Intelligence* **129** (2001) 35–62
12. Bryant, R.E.: Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers* **C-35** (1986) 677–691
13. K.L. McMillan: *Symbolic Model Checking*. Kluwer Academic Publishers, Norwell Massachusetts (1993)
14. R. Bahar, E. Frohm, C. Gaona, G. Hachtel, E. Macii, A. Pardo, F. Somenzi: Algebraic Decision Diagrams and Their Applications. In: IEEE /ACM International Conference on CAD, IEEE Computer Society Press (1993) 188–191
15. Somenzi, F.: Binary decision diagrams. In Broy, M., Steinbruggen, R., eds.: *Calculational System Design*. Volume 173 of NATO Science Series F: Computer and Systems Sciences. IOS Press (1999) 303–366
16. Edelkamp, S.: Directed symbolic exploration in AI-planning. In: AAAI Spring Symposium on Model-based Validation of Intelligence, Stanford University (2001) 84–92
17. Somenzi, F.: CUDD: CU decision diagram package. <ftp://vlsi.colorado.edu/pub/> (1998)
18. Daniele, M., Traverso, P., Vardi, M.: Strong cyclic planning revisited. In: Proceedings of the 5th European Conference on Planning (ECP-99). (1999)
19. Nilsson, N.: *Principles of Artificial Intelligence*. Tioga Publishing Co., Palo Alto, CA (1980)
20. Feng, Z., Hansen, E.: Symbolic heuristic search for factored Markov decision processes. (2002) Proceedings of the 18th National Conference on Artificial Intelligence (AAAI-02).