Memory-Efficient Symbolic Heuristic Search

Rune M. Jensen
IT University of Copenhagen
2300 Copenhagen S, Denmark
rmj@itu.dk

Eric A. Hansen and **Simon Richards**
Dept. of Computer Science and Eng.
Mississippi State University
Mississippi State, MS 39762 USA
hansen@cse.msstate.edu

Rong Zhou
Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, CA 94304 USA
rzhou@parc.com

Abstract

A promising approach to solving large state-space search problems is to integrate heuristic search with symbolic search. Recent work shows that a symbolic A* search algorithm that uses binary decision diagrams to compactly represent sets of states outperforms traditional A* in many domains. Since the memory requirements of A* limit its scalability, we show how to integrate symbolic search with a memory-efficient strategy for heuristic search. We analyze the resulting search algorithm, consider the factors that affect its behavior, and evaluate its performance in solving benchmark problems that include STRIPS planning problems.

Introduction

There has been much recent interest in improving the scalability of AI planning algorithms by using symbolic data structures and search techniques developed originally for model checking, a widely-used approach to hardware and software verification (Clarke, Grumberg, & Peled 2000). A model checker determines whether a system satisfies a formal property by using a search algorithm to explore the state space that represents all possible execution paths. The central challenge in scaling up model-checking algorithms is the same as the challenge in scaling up planning algorithms – the *state explosion problem*, that is, the size of the state space grows exponentially in the number of variables in the problem description. Among the techniques that have been developed in the model checking community to deal with the state-explosion problem, one of the most effective is an approach to state abstraction that uses a data structure called a decision diagram to implicitly represent sets of states and operations on sets of states. Use of decision diagrams in model checking is called *symbolic model checking*. Over the past few years, planning researchers have successfully adopted a symbolic approach based on decision diagrams to improve the scalability of algorithms for planning in non-deterministic domains (Cimatti, Roveri, & Traverso 1998), for multi-agent planning (Jensen & Veloso 2000), for planning using Markov decision processes (Hoey *et al.* 1999; Feng & Hansen 2002), for conformant planning (Cimatti, Roveri, & Bertoli 2004), and for planning in partially observable domains (Bertoli *et al.* 2001).

In this paper, we consider a symbolic approach to deterministic planning. As a starting point, we consider recent work that uses decision diagrams to improve the scalability of the A* search algorithm (Edelkamp & Reffel 1998; Jensen, Bryant, & Veloso 2002; Hansen, Zhou, & Feng 2002; Jensen 2003; Nymeyer & Qian 2003; Edelkamp 2005). Symbolic implementations of A* have been shown to outperform traditional A* in several domains. But in common with traditional A*, the bottleneck of a symbolic implementation of A* is its memory requirement, that is, the need to store a representation of all visited states in memory. Although various techniques for reducing the memory requirements of A* have been developed, they have not yet been adapted for use in a symbolic heuristic search algorithm that uses decision diagrams. In fact, techniques that reduce the memory requirements of A* by using depth-first search, such as Depth-First Iterative-Deepening A* (Korf 1985), are not obviously compatible with decision diagrams, which were developed for use in breadth-first search. Other approaches to memory-efficient A*, such as frontier search (Korf *et al.* 2005) and sparse-memory graph search (Zhou & Hansen 2003), seem incompatible with a symbolic approach because they rely on data structures that assume states are represented individually.

In this paper, we show how to integrate decision diagrams with a recent approach to memory-efficient search called *breadth-first heuristic search* (Zhou & Hansen 2006). As in frontier and sparse-memory graph search, this approach reduces memory requirements by using a divide-and-conquer technique for solution recovery that makes it unnecessary to store all states generated during the search in memory. In a symbolic algorithm, this means it is unnecessary to store all decision diagrams generated during the search in memory. Although development of a symbolic version of breadth-first heuristic search (BFHS) requires some innovation, especially in the method of solution recovery, we show that a breadth-first approach (BFHS relies on breadth-first branch-and-bound search) makes it easier to integrate these two techniques for improving scalability – symbolic search using decision diagrams, and use of divide-and-conquer solution recovery to reduce memory requirements. We analyze the resulting algorithm and study the factors that affect its performance in solving a range of benchmark search problems, including propositional planning problems.

Background

We begin by reviewing relevant background about state-space search, decision diagrams, symbolic reachability analysis and previous work on BDD-based heuristic search.

State-space search

A state-space search problem is defined as a tuple (S, G, s^0, A, T), 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); and T denotes a set of transition relations $\{T^a : S \times S\}$, one for each action $a \in A$, such that $(s, s') \in T^a$ if and only if taking action a in state s results in a transition to state s'. We assume transitions are deterministic. In this paper, we also assume that all transitions have unit cost and the objective is to find a shortest path from the start state to a goal state.

Decision diagrams and symbolic graph traversal

Decision diagrams, especially binary decision diagrams, are widely used as a form of state abstraction in symbolic model checking. *Binary decision diagrams* provide an efficient way of representing and manipulating Boolean functions. A binary decision diagram (BDD) is a rooted directed acyclic graph with internal nodes of out-degree two, and two terminal nodes, one for the value of 0 and one for the value of 1. Every non-terminal node is labeled with a Boolean variable and its two outgoing edges correspond to the values of 0 and 1. The value of the Boolean input variables is mapped to an output Boolean value by following a path from the root of the BDD to a terminal node. An *ordered binary decision diagram* is a BDD with the constraint that the input variables are ordered and every path in the BDD visits them in the same order. A *reduced ordered BDD* is a BDD in which every node represents a distinct logic function. In their reduced, ordered form, and with a fixed variable ordering, BDDs provide a canonical representation of Boolean functions. This canonicity allows efficient algorithms for manipulating them. BDDs can leverage problem structure to achieve significant state abstraction in representing and manipulating Boolean functions, and this often leads to a reduction in time and space complexity from exponential to polynomial in the number of state variables. Furthermore, a set of BDDs can be represented compactly in a global multi-rooted BDD. Since the BDDs often are related, this can lead to further space savings.

In order to use BDDs in state-space search, the problem state must be described by a set of Boolean state variables, $\vec{x} = \{x_1 \dots x_n\}$. For propositional planning problems, as well as for circuit verification problems, this is the natural representation. For other state-space search problems, an efficient Boolean encoding must be designed. Given such an encoding, a set of states F can be represented by its Boolean characteristic function $F(\vec{x})$, such that $s \in F \iff F(\vec{x}) = 1$, and this Boolean characteristic function can be represented compactly by a BDD. The transition relation for each action can also be represented by a BDD. Given a set of present state variables $\vec{x} = \{x_1 \dots x_n\}$, and a set of next state variables $\vec{x}' = \{x_1' \dots x_n'\}$, the transition relation $T^a(\vec{x}, \vec{x}')$ has a Boolean characteristic function with a value of 1 when action a in state $\vec{x}$ causes a transition to state $\vec{x}'$, and otherwise has a value of 0.

A symbolic approach to state-space search manipulates sets of states, instead of individual states. Given a set of states F, a central task is to find all successors of these states after action a. This task is called *image computation*, and the image of F according to T^a is given by

$$Image(F, T^a) = \left(\exists_{\vec{x}}.F(\vec{x}) \wedge T^a(\vec{x}, \vec{x}') \right) [\vec{x}'/\vec{x}].$$

The operator on the right-hand side of the assignment is called the *relational product* operator. It represents a series of nested existential quantifications, one for each variable in $\vec{x}$. The conjunction $F(\vec{x}) \wedge T^a(\vec{x}, \vec{x}')$ selects the set of valid transitions and the existential quantification extracts and unions the successor states together. Similarly, it is possible to compute the predecessors of a set of states, called the *pre-image*, as follows:

$$PreImage(F, T^a) = \left(\exists_{\vec{x}'}.F(\vec{x}') \wedge T^a(\vec{x}, \vec{x}') \right) [\vec{x}/\vec{x}'].$$

The image and pre-image operators are used in symbolic search algorithms that traverse the state space of a transition system. For example, the set of states that is reachable from an initial state can be determined by a simple breadth-first search starting from an initial state. Let R^i denote the set of states reachable from the initial state s^0 in i steps, initialized by $R^0 = \{s^0\}$. Given any set R^i, we can use the image operator to compute the set R^{i+1} as follows:

$$R^{i+1} \rightarrow \cup_a Image(R^i, T^a)$$

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.

Symbolic A*

The heuristic search algorithm A* organizes its search using a priority queue (or Open list) to store unexpanded states on the frontier of the search, and a Closed list to store all expanded states. A* determines the order in which to expand states by using a state evaluation function, $f(s) = g(s) + h(s)$, where $g(s)$ is the cost of a best path from the start state to state s, and $h(s)$ is a lower-bound function that estimates the cost of a best path from state s to a goal state.

Edelkamp and Reffel (1998) were the first to develop a symbolic A* algorithm that is implemented using BDDs. A symbolic implementation of A* must do more than symbolic reachability analysis; it must *evaluate* states, that is, compute their f-costs, in order to focus the search and find an optimal path. Their algorithm, called BDDA*, represents the Open list as a set of BDDs, one for each distinct f-cost. It selects the BDD with least f-cost to expand, and computes successor states using the image operator. Then it computes the f-costs of the successors states using BDDs that encode the search heuristic and transition costs in binary. They show that this symbolic implementation of A* significantly outperforms symbolic breadth-first search, and is effective in solving propositional planning problems.
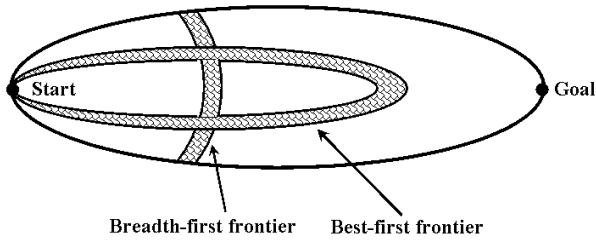
Figure 1: Comparison of best-first and breadth-first frontiers. The outer ellipse encloses all nodes with f-cost less than or equal to an (optimal) upper bound.

Hansen et al. (2002) describe a symbolic implementation of A* called ADDA* that uses *algebraic decision diagrams*, an extension of BDDs that allows more than two terminal nodes in order to represent multi-valued functions. ADDA* can represent the heuristic function, as well as state and transition costs, without encoding them in binary. But both BDDA* and ADDA* suffer from the overhead of performing arithmetic using decision diagrams when computing f-costs of successor states.

Jensen et al. (2002) significantly improved the performance of symbolic A* by using a partitioning technique called *branching partitioning* that avoids BDD-level arithmetic. In a branching partitioning, the transition relation is partitioned into a set of BDDs, such that all transitions in the same BDD change the f-cost by the same amount. (If all transitions have unit cost, this is equivalent to changing the h-cost by the same amount.) When a BDD representing a set of states with the same f-cost is selected for expansion, the image operator is performed once for each BDD in the branching partitioning, and a set of successor BDDs is produced, one for each possible change in the f-cost. Using this method of partitioning, the successor states are partitioned by f-cost, and the f-cost of all successor states is known without performing any arithmetic at the BDD level.

The approach is called *state-set branching*. Because it doesn't perform arithmetic using decision diagrams, it avoids the bottleneck of BDDA* and ADDA* and dramatically improves performance. We use this same partitioning technique in the algorithm developed in the rest of this paper.

Memory-efficient BDD-based heuristic search

We now describe a memory-efficient approach to BDD-based heuristic search. The approach is based on the *breadth-first heuristic search* algorithm of Zhou and Hansen (2006), with some differences to accommodate the use of decision diagrams. Breadth-first heuristic search focuses its search using the same heuristic used by A*. But as shown in Figure 1, it expands the search frontier in breadth-first order, instead of best-first order. Essentially, it is a memory-efficient form of breadth-first branch-and-bound search, where memory efficiency is achieved by only storing layers of the search graph that are on (or near) the search frontier, and recovering the solution path by a divide-and-conquer technique.

We begin by explaining how to implement breadth-first branch-and-bound search using BDDs. Then we introduce our techniques for memory-efficient search.

Symbolic breadth-first branch and bound

Breadth-first branch-and-bound search generates a search graph that consists of a sequence of *layers*, in which all states in the same layer have the same depth and g-cost. (Since we assume unit-cost actions, the depth of a layer is equal to its g-cost.) We could use a single BDD to represent the states in each layer. But in order to apply state-set branching, we partition each layer into a set of BDDs, one for each f-cost (or, equivalently, h-cost, since the g-cost of every node in a layer is the same). We will call each of these BDDs, an *f-cost layer-BDD*. We also use the branching partitioning method of Jensen et al. to partition the transition relation into a set of BDDs, one for each change in f-cost.

We use state-set branching to expand the search frontier. Hence, for each pair of f-cost layer-BDD and branching partitioning BDD, we use the image operator to compute a BDD that represents the set of successor states. Thus the number of image computations performed in generating the next layer of a breadth-first search graph is the product of the cardinality of the layer partition and the cardinality of the branching partitioning. Because we know the f-cost of the parent states and the amount by which each partition in the branching partitioning changes the f-cost, we know the f-cost of all of the successor states without performing any arithmetic at the BDD level. Moreover, we know what the f-cost of the successor states will be before performing the image computation. Therefore, given an upper bound on the optimal cost, *we can prune successor states before even generating them* – an important advantage of this approach. The search algorithm generates successive layers of the breadth-first search graph until it generates a layer-partition that contains a goal state. (Such a layer partition must have an h-cost of zero, given an admissible heuristic.)

Table 1 shows the pseudocode of the symbolic breadth-first branch-and-bound algorithm (SBFBnB). It includes three subroutines that support the memory-saving techniques we introduce next. The subroutine used in line 12, $DeleteLayers$, recovers memory by deleting previous layers of the breadth-first search graph. Layers are deleted in order from the beginning of the search graph, and only as many layers are deleted as needed to recover enough memory to continue the search. The challenge for this approach to reducing memory requirements is that previous layers of the graph are used in both duplicate elimination (line 8) and solution reconstruction (line 11). In the rest of this section, we describe how to perform duplicate elimination and solution reconstruction after layers have been deleted.

Solution reconstruction

Line 11 of the pseudocode in Table 1 invokes a subroutine for solution reconstruction. Although removing previous layers of the search graph from memory can substantially reduce the memory requirements of BDD-based heuristic search, one challenge this creates is how to reconstruct a solution after the search terminates.

procedure *SBFBnB* (BDD $start, goal, transitions$, INT U)
1. $BP \leftarrow BranchingPartitioning(transitions)$
2. $Layer[0, h(start)] \leftarrow start$
3. **for** $g = 1$ **to** U **do**
4.   **for** $h = 0$ **to** $U - g$ **do**
5.   **for** $i = 0$ **to** $|BP|$ **do**
6.   **if** $h + \delta h(BP[i]) < U - g$ **then**
7. $Succ \leftarrow Image(Layer[g - 1, h], BP[i])$
8. $Succ \leftarrow RemoveDuplicates(Succ)$
9. $Layer[g, h + \delta h(BP[i])] \leftarrow Succ$
10.   **if** $h + \delta h(BP[i]) = 0$ **and** $Layer(g, 0) \cap goal \neq \emptyset$ **then**
11.   **return** $ExtractSolution(start, goal, g, Layer)$
12.   **if** memory is low **then** $DeleteLayers(Layer, g)$

Table 1: Pseudocode for symbolic breadth-first branch and bound search.

We first review how to reconstruct a solution when all layers of the breadth-first search graph are retained in memory. Then we describe a divide-and-conquer method for reconstructing a solution that does not require retaining all layers of the search graph in memory.

Sequential solution reconstruction After a symbolic heuristic search algorithm finds a goal state, it cannot reconstruct a solution path in the same way as a non-symbolic heuristic search algorithm – by tracing pointers backward from the goal state to the start state. Not only are there no pointers, but individual states are not even represented. Path information must be maintained in a different way.

One way to allow reconstruction of the solution path is to keep all of the BDDs generated during the search in memory, as well as a record of their successor relationships. Jensen (2003) describes a symbolic implementation of A* that builds an *abstract search graph* that preserves path information. Each node of the abstract search graph corresponds to a BDD generated during the search. One node in the abstract search graph is a successor of another if its corresponding BDD was generated by the image operator from the BDD corresponding to its predecessor node; it follows that the states it represents are successors of the states represented by the prececessor node. A solution path begins at the start state (represented by the BDD corresponding to the start node of the abstract search graph); it terminates at a goal state (represented by the BDD corresponding to the goal node of the abstract search graph); and it consists of a sequence of successor states, with one state represented by each BDD corresponding to an abstract node along a shortest path from the start node to the goal node of the abstract search graph. In breadth-first branch-and-bound search (BF-BnB), the search space is divided into layers and a solution path consists of one state from each layer of the search graph such that each state is a successor of the previous one. In SBFBnB search, each layer is represents by a set of BDDs.

To explain how the solution path is recovered, we consider the case of BFBnB; a similar idea works in A*. Once the search terminates, a goal state on an optimal path has been identified. The state in an optimal path that comes just before the goal state must be in the next-to-the-last layer. For a unit-cost search problem, it can be identified by determining which state in this layer has the goal state as a successor. (For a non-unit cost problem, it is the state in the previous layer such that it's g-cost and the cost of a transition to the goal state is the minimum.) This can be determined efficiently by computing the pre-image of the goal state, for each action, and intersecting the result with the previous layer. The process is repeated for each layer backwards from the goal state to the start state, in order to identify every state and action along an optimal path. Because the pre-image is only computed for a single state in each layer, this method of solution reconstruction is very fast. The problem is that it requires keeping all layers of the search graph (i.e., all BDDs generated during the search) in memory. We next consider how to avoid this.

Divide-and-conquer solution reconstruction To reconstruct a solution path without saving all generated BDDs in memory, we propose to use a divide-and-conquer technique for solution reconstruction. The basic idea is to use the layers that can fit in memory to reconstruct part of the optimal solution path, and then recursively call the same search algorithm to reconstruct the rest of the solution, by solving smaller subproblems of the original search problem.

For this divide-and-conquer technique to work, the search algorithm must have at least one layer of the search graph in memory (not including the start and goal layers) when the search terminates. We will assume it is the last layer before the goal state (although it does not have to be). The search algorithm should also store as many additional layers as fit in memory, backwards from the goal layer. If all layers fit in memory, sequential solution reconstruction can be used. If not, the algorithm traces the solution path backwards from the goal state for as many layers as fit in memory, in the same way as in sequential solution reconstruction. When it reaches a missing layer, it creates a new search problem in which the start state is the original start state and the goal state corresponds to the shallowest state (i.e., the state closest to the start state) on the partial solution path constructed backwards from the goal. This subproblem is exponentially easier to solve than the original search problem, assuming search complexity is exponential in the length of a solution path, and so it can be solved relatively quickly to identify the rest of the solution path. The number of subproblems that need to be solved in the worst case is bounded by the length of the solution path, but usually just one or two levels of recursion will be sufficient to solve the problem within the constraints on available memory. This offers a memory-time tradeoff in which memory is saved in exchange for the time needed to reconstruct some of the layers.

Duplicate elimination

The subroutine $RemoveDuplicates$ in line 8 of the pseudocode performs duplicate elimination. This is not necessary for the correctness of the search algorithm, but has a big effect on its efficiency. Since the set of successor

states generated by the image operator may include states that have previously been generated, efficiency is improved by eliminating these duplicates. In the model-checking literature, duplicate elimination is called *frontier (or forward) set simplification*, and involves comparing the successor states against already-generated states in the current and previous layers of the search graph.

Although it is possible to perform duplicate elimination by using a single BDD to represent the set of all previously-generated states, the layered structure of the search graph encodes path information that is needed for solution reconstruction. Therefore, an implementation of BDD-based breadth-first branch-and-bound search needs to keep distinct BDDs in memory for the different layers of the search graph.

For many graph-search problems, it is not necessary to store and check all previously visited states in order to completely eliminate duplicates. In particular, Zhou and Hansen (2006) point out that in breadth-first search of undirected graphs, it is only necessary to store and check the immediate previous layer, in addition to the current layer and the next layer, in order to eliminate all duplicates. The reason for this is that in undirected graphs, every potential successor of a state is also a potential predecessor. In directed graphs, it is not possible to say, in general, how many previous layers need to be stored in memory in order to completely eliminate duplicates. It depends on the structure of each problem. But given knowledge of the structure, it is often possible to bound the number of layers that need to be retained in order to completely eliminate duplicates.

In complex directed graphs in which it is impossible to completely eliminate duplicates without retaining all layers, retaining as many previous layers as fit in memory allows the number of duplicates to be reduced in a principled way. If l is the number of layers in a breadth-first search graph, and k is the number of previous layers stored in memory for duplicate elimination, then $\frac{l}{k}$ bounds the number of layers in which a particular state can occur as a duplicate. This suggests a tradeoff between the memory used to store previous layers in memory, and the performance improvement that results from duplicate elimination. It is also a reason for deleting layers from the beginning of the graph, when memory is low.

One of the advantages of breadth-first search is that it makes it easy to determine which layers of the search graph can be deleted to recover memory, and which should be retained for duplicate elimination. It is not as easy to see how a similar layered approach to duplicate detection could be used in a memory-efficient implementation of symbolic A*.

Upper bounds and iterative deepening

Use of breadth-first branch-and-bound search requires an upper bound on the cost of an optimal solution, and the tighter the upper bound, the more efficient the search. An upper bound can be obtained by running a fast approximate search algorithm that finds a (possibly) sub-optimal solution relatively quickly; possibilities include weighted A* and beam search, in either their symbolic or non-symbolic forms. As an alternative to finding an approximate solution, an iterative-deepening strategy can be used in which the algorithm is run with a hypothetical upper bound, and the hypothetical upper bound is increased as necessary, until an optimal solution is found. Zhou and Hansen (2006) call this algorithm *breadth-first iterative-deepening A**. In this iterative-deepening algorithm, solution reconstruction is not performed until the last iteration, when a solution is found.

Symbolic admissible heuristics

Symbolic heuristic search requires a lower-bound function (i..e, an admissible heuristic) that is represented symbolically as a decision diagram. For specific domains, such heuristics can be created by hand. For example, a symbolic encoding of the Manhattan distance heuristic for the Fifteen Puzzle is easily created. For domain-independent planning, an automatic method of creating symbolic admissible heuristics is necessary. Edelkamp (2002) describes how to create *symbolic pattern databases* for use by a domain-independent planner. We consider a different and more widely-used approach to automatically creating admissible heuristics for a domain-independent planner, the h^m heuristic of Haslum and Geffner (2000), and describe how to efficiently compute a symbolic generalization of it.

Briefly, the $h^m (m = 1, 2, \ldots)$ heuristic (or family of heuristics) is computed as follows. For each group of m atoms in a state, an optimistic estimate of the cost of achieving these atoms is determined via a relaxed reachability analysis that can be computed efficiently using dynamic programming. The heuristic estimate is the maximum of these costs and is guaranteed to be both admissible and consistent. For large values of m, it converges to the optimal cost but is prohibitively expensive to compute. In practice, the *max-pair heuristic* (where $m = 2$) is most widely used. It is particularly efficient when used for regression planning, since the reachability analysis of each pair of atoms only needs to be computed once.

For search algorithms that use an explicit state representation, the maximization part of the heuristic is computed in each iteration. For symbolic algorithms, on the other hand, we need to compute a branching partitioning of the heuristic. Since the max-pair branching partitioning turns out to be somewhat challenging to compute efficiently, we describe our approach in detail below. To our knowledge, this is the first use of the max-pair heuristic in symbolic search.

For most heuristics, there is a simple relation between the state the action is applied in and the change of the heuristic estimate. For instance, when moving a tile down in the Fifteen Puzzle, we have $\delta h = -1$ if the move is towards the goal row of the tile, and otherwise $\delta h = 1$. Thus in many cases, it is easy to compute the branching partitioning by splitting the transition relation of each action according to these different situations. For the max-pair heuristic, however, there is no simple relation between the state an action is applied in and the change of the heuristic estimate.

To overcome this problem, our approach is to build a BDD $h_i(\vec{x})$ for each possible heuristic value i that represents the set of states with $h = i$. It is easy to use these BDDs to build a branching partitioning. We first compute a vector of BDDs $\delta_i(\vec{x}, \vec{x}')$ for each transition with $\delta h = i$ and then use it to partition the transition relation of the search space. For the

max-pair heuristic, however, the $\delta_i(\vec{x}, \vec{x}')$ BDDs turns out to be prohibitively large. To avoid this problem, we exploit the fact that the planning domains we consider have bidirectional search graphs. For a consistent heuristic like max-pair this means that $|\delta h| \leq 1$. Thus, we can make the branching partitioning a function of the heuristic value i of the set of states it expands and split the transition relation of an action by conjoining only with h_{i-1}, h_i, and h_{i+1}. This technique is quite efficient, and we believe that it can be used for other heuristics with high combinatorial complexity. Even if the search graph is not bidirectional, it may be possible to bound $|\delta h|$ and use a similar technique.

Experimental evaluation

We evaluate the performance of our memory-efficient approach to symbolic heuristic search in three domains: the Fifteen puzzle and two planning problems from the STRIPS track of the International Planning Competitions of 1998 and 2002, respectively. Our evaluation includes a comparison of the time and space consumption of the following algorithms; symbolic and non-symbolic versions of breadth-first branch-and-bound search (SBFBnB and BFBnB), A*, and a symbolic generalization of A* called SetA* (Jensen, Bryant, & Veloso 2002). Results of the comparison are summarized in Table 2 and in additional graphs to be explained below.

Domains and implementation details We tested the algorithms on six instances of the Fifteen puzzle of increasing difficulty taken from Korf's 100 random instances (Korf 1985). In Table 2, we use Korf's numbering scheme to identify the instances. For the Fifteen puzzle, we implemented the search algorithms in C++/STL. States are represented as vectors of tile positions and STL hash sets are used to represent sets of states. SBFBnB and SetA* were implemented in the BIFROST planning framework (Jensen 2003). The Boolean state representation used by SBFBnB and SetA* consists of a bit vector for each tile position that is a binary encoding of the identity of the tile located at the position. The representation is similar to the one used in (Nymeyer & Qian 2003). All algorithms use the sum of Manhattan distances as an admissible heuristic.

The two STRIPS planning problems used in our evaluation are Blocks and Zenotravel. The domains are defined in PDDL. The planning algorithms are implemented in BIFROST and use the max-pair heuristic to guide a backward search. When given a PDDL input, BIFROST reduces the state space by using a preprocessing reachability analysis that identifies static and balanced predicates (Edelkamp 2001). Static predicates cannot change truth-value and are compiled out of the state and action representation. Balanced predicates like $at(x, y)$ have a variable (in this case x) such that when this variable is bound to an object, exactly *one* of the facts obtained by binding the remaining variables is true in a state. Balanced predicates typically encode locations. Thus, if there are n balanced facts, we only need $\lceil \log n \rceil$ bits to represent their truth-values. This is exploited by the Boolean state representation used by SetA* and SBFBnB. The explicit state representation of A* and

BFBnB uses an STL integer set to identify the true facts in a state. More compact state representations may be possible, but have not been explored in this work.

For all of the domains used in our evaluation, all actions have unit cost and the search graph is undirected. In such domains, BFBnB and SBFBnB can achieve duplicate elimination by keeping only three layers in memory (Zhou & Hansen 2006). To achieve best performance, SBFBnB only deletes layers from memory when it approaches its memory limit. For evaluation purposes, however, we arranged for SBFBnB to use as little memory as possible while still ensuring complete duplicate elimination. Often, this means that it deletes all but three layers from memory, even when more memory is available.

In Table 2, we report results for BFBnB and SBFBnB given an optimal upper bound. This corresponds to the last iteration of breadth-first iterative-deepening A*, which uses iterative deepening to find the optimal upper bound (Zhou & Hansen 2006). The last iteration determines the peak memory use of the algorithm and takes much longer than earlier iterations. The time required for solution recovery is included in the running time shown in Table 2. In divide-and-conquer solution recovery, the divide-and-conquer process never required more than one level of recursion for any of the problem instances.

The experiments were conducted on a Linux PC with a 3.2 Ghz Pentium Xeon CPU, 1024 KB L2 cache, and 3 GB RAM. To avoid excessive duration of the experiments, however, the available memory was adjusted down to 1.8 GB for the Fifteen puzzle and 1 GB for Blocks and Zenotravel. The memory consumption of an algorithm was measued as the peak amount of memory allocated by its process. The algorithms were adjusted to use the least possible amount of memory in order to solve a problem. Thus, BFBnB and SBFBnB would at some point have only three layers in memory. Note also that the symbolic algorithms reserve 5% of the BDD nodes to operator caches. (Experimental studies of reachablility analysis in formal verification show that caches with between 5 and 10 percent of the total number of nodes have fairly good performance (Yang *et al.* 1998).)

Analysis of results Table 2 shows the time and space consumption of the algorithms. Our new algorithm, SBFBnB, is the only one able to solve all problems within the memory bounds. For the largest problems in each domain, it also consistently uses less memory than any other algorithm.

For the Fifteen Puzzle problems, the performance of A* and SetA* confirms previous results in (Jensen 2003).[1] A* has lower time overhead than SetA*, but quickly exhausts memory. As for the algorithms that use divide-and-conquer solution recovery, BFBnB uses less memory than both A* and SetA*, and SBFBns scales even better than BFBnB.

In our experiments, BFBnB and SBFBnB remove all but three layers of the search graph from memory. To esti-

---

[1]The results, however, do not support the hypothesis in (Nymeyer & Qian 2003) that the sum of Manhattan distances is too strong a heuristic for SetA*, and prevents if from being more memory efficient than A*

Problem	Len	A* Time	A* Space	BFBnB Time	BFBnB Space	SetA* Time	SetA* Space	SBFBnB Time	SBFBnB Space
15puzzle-12	45	0.4	10	1.3	8	12.1	26	11.7	16
15puzzle-55	41	2.3	44	1.7	10	39.0	67	13.7	23
15puzzle-19	46	2.2	44	6.1	25	51.3	67	43.8	35
15puzzle-20	52	48.1	531	89.4	277	618.7	456	484.8	176
15puzzle-69	53	163.1	1173	206.3	732	1389.2	820	832.5	387
15puzzle-32	59	-	>1800	-	>1800	-	>1800	3727.9	1733
Blocks-4	6	0.1	3	0.1	3	0.1	4	0.4	4
Blocks-5	12	0.3	3	0.3	3	0.4	4	1.1	4
Blocks-6	12	0.6	4	0.6	4	0.7	4	2.4	4
Blocks-7	20	1.7	4	1.7	4	2.0	5	5.7	6
Blocks-8	18	1.9	4	2.1	4	3.5	8	10.5	9
Blocks-9	30	6.3	6	11.1	5	7.0	9	19.6	10
Blocks-10	34	11.3	9	47.3	8	13.0	16	22.1	16
Blocks-11	32	35.7	24	143.8	12	69.2	33	90.8	31
Blocks-12	34	132.7	129	689.2	45	33.1	35	52.3	32
Blocks-13	42	1025.5	737	14982.5	344	226.6	95	375.4	71
Blocks-14	38	-	>1000	-	>1000	363.9	132	500.8	111
Zenotravel-1	2	0.1	3	0.2	3	0.1	4	0.2	4
Zenotravel-2	8	0.2	4	0.3	3	0.2	4	0.6	4
Zenotravel-3	8	0.6	4	1.5	4	0.7	4	2.2	4
Zenotravel-4	12	10.4	15	33.4	7	1.0	5	3.2	5
Zenotravel-5	16	94.5	80	204.2	21	1.9	7	5.5	6
Zenotravel-6	16	560.6	332	1169.0	90	2.7	10	7.2	10
Zenotravel-7	17	-	>1000	-	>1000	13.3	31	22.2	28
Zenotravel-8	16	-	>1000	-	>1000	28.1	48	66.7	35

Table 2: Time (sec) and space (MB) consumption of the algorithms on the 15-Puzzle, Blocks, and Zenotravel problems.

mate the additional time overhead for divide-and-conquer solution recovery in the symbolic algorithm, we compared SBFBnB to a version of the same algorithm that keeps all layers in memory and recovers a solution in the conventional way (SBFBnB*). We ran the algorithms on the five Fifteen Puzzle problems that both could solve. The results, presented in Figure 2, show that the additional overhead for performing divide-and-conquer solution recovery instead of conventional solution recovery is negligible.

In the two planning domains, the symbolic state representation is quite compact leading to good performance of SetA* and SBFBnB compared to A* and BFBnB. There may be several reasons for this. First, since the state-space of the Fifteen Puzzle is a permutation subspace, it does not have an efficient BDD representation. Second, the general explicit-state representation used by BIFROST is not as memory efficient as the specialized representation used for the Fifteen Puzzle problems.

The space savings of BFBnB and SBFBnB compared to A* and SetA* are significant but less dramatic in the Blocks and Zenotravel domains. In part, this is because the memory needed to store the max-pair heuristic is included in the results reported in Table 2. The symbolic max-pair heuristic, in particular, requires a significant amount of memory. This affects the ratio of memory used by SBFBnB compared to memory used by SetA*. They both must allocate space for the max-pair heuristic, but if we only consider the memory

used to store the layers of the search graph, the memory saving of SBFBnB relative to SetA* is closer to that observed for the Fifteen Puzzle. But other factors also limit the effectiveness of SBFBnB for these planning problems. To un-
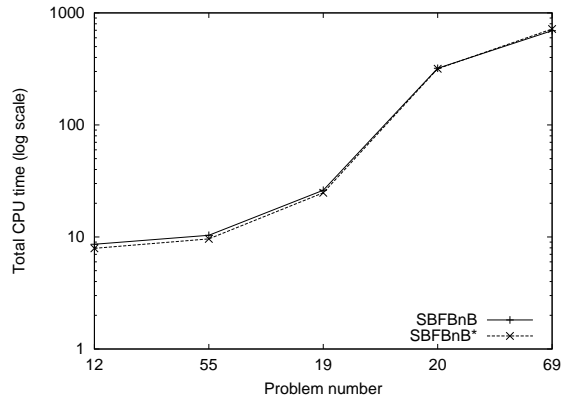

Figure 2: The time overhead for deleting layers and recovering a solution by the divide-and-conquer method is shown by comparing the time it takes SBFBnB to solve the first five Fifteen Puzzle problems with and without using divide-and-conquer solution recovery. (SBFBnB* denotes the algorithm that does not use the divide-and-conquer method.)
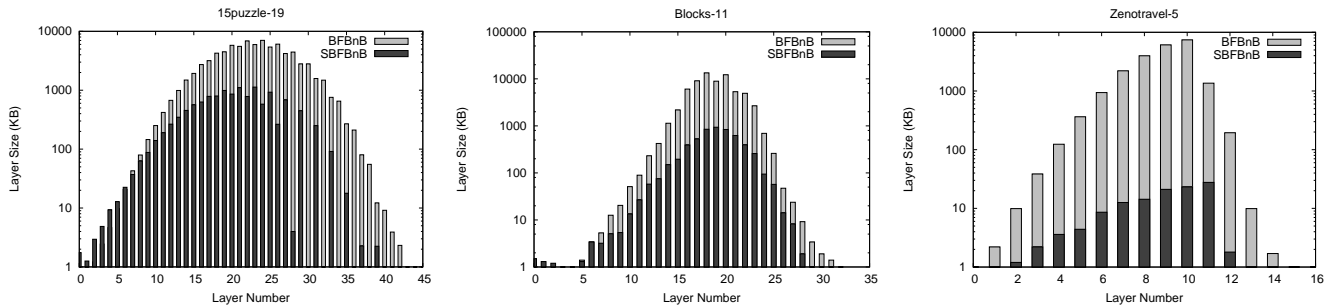
Figure 3: Layer size as a function of search depth for BFBnB and SBFBnB on 15puzzle-19, Blocks-11, and Zenotravel-5. (Note that the y-axis is logarithmically scaled.)

derstand this better, we plot the layer size as a function of search depth of BFBnB and SBFBnB on a selected problem for each domain. The results are shown in Figure 3.

Because the BDD package uses a multi-rooted BDD to share structure between active BDDs, it is non-trivial to estimate how much memory each layer of SBFBnB uses. In the graphs, a BDD l_i representing the states in layer 0 to i is constructed and the estimated size of layer i is defined to be $|l_i| - |l_{i-1}|$. In the last half of the search, l_i often becomes more structured when adding more states. For this reason a layer may have negative size. Due to the logarithmic scale, such layers have been given size 1 in the graphs.

It is useless for SBFBnB and BFBnB to remove layers from memory if the layer size doubles or more in each iteration. In this case, deleting all previous layers does not release enough memory for even a single new layer, and at least three layers must reside in memory. The stronger the heuristic, the smaller the growth rate of layer size. Furthermore, using a strong heuristic, layer size reaches a maximum in the middle of the search; using a weak heuristic, layer size reaches its maximum later. Thus, the stronger the heuristic, the wider the area in which SBFBnB and BFBnB can benefit from removing layers from memory. The BDD representation has an effect similar to a stronger heuristic. It reduces the growth rate and thus widens the area where it is fruitful to delete layers in memory.

Going back to the graphs, we see that the area with sufficiently small growth rate is much narrower for Blocks-11 than 15puzzle-19. For Zenotravel-5, the peak is pushed even further to the right. This reflects the fact that the max-pair heuristic is much less informative than the sum of Manhattan distances. This leads to some performance degradation of SBFBnB and BFBnB.

Another important factor that affects the relative performance of these algorithms is their tie-breaking behavior. Because all of the test problems we used have unit action costs, there are *many* ties. Unfortunately, a breadth-first approach to heuristic search has worst-case tie-breaking behavior. Nodes with the same f-cost can occur in all layers of the search graph; because all nodes in one layer are expanded before the next layer is considered, most nodes with an f-cost equal to the optimal f-cost are expanded before an optimal solution is found. By contrast, A* and SetA* break

ties by preferring to expand nodes with the least h-cost and this leads to near-optimal tie-breaking behavior. In the results reported in Table 2, A* and SetA* enjoy near-optimal tie-breaking and BFBnB and SBFBnB suffer worst-case tie-breaking. This has a significant effect on the results. Although BFBnB and SBFBnB still show a benefit, their relative performance would be significantly improved by finding a way to improve on their worst-case tie-breaking behavior.

Enhancements

We are optimistic that we can improve on the results reported in the previous section by continuing to develop the approach proposed in this paper. Here we briefly outline some enhancements that we will consider in future work.

Bidirectional search

We have seen that the effectiveness of the memory-efficient approach is somewhat diminished for the STRIPS planning problems because of the weakness of the max-pair heuristic. With a more informative heuristic, we expect the relative improvement to be greater. Developing more informative admissible heuristics for domain-independent planners is an important direction for future research.

But for search problems where the heuristic is very weak, another way to improve memory efficiency might be to use bidirectional search. The divide-and-conquer solution reconstruction technique described in this paper is used in a unidirectional search algorithm that searches forward from the start state to a goal state, and then traces the solution path backwards from the goal to the start state. But it is possible to use the same divide-and-conquer approach to improve the memory efficiency of bidirectional search. In fact, Korf (1999) first used divide-and-conquer solution recovery in a bidirectional search algorithm.

In bidirectional breadth-first heuristic search (Richards 2004), one search algorithm searches forward from the start state, the other searches backwards from the goal state, and when they reach the same layer, the states in the intersection of the two frontiers must be on an optimal path. Given a state on the optimal path, two subproblems can be created, one that involves searching for an optimal path from the original start state to an intermediate state in the intersection of the
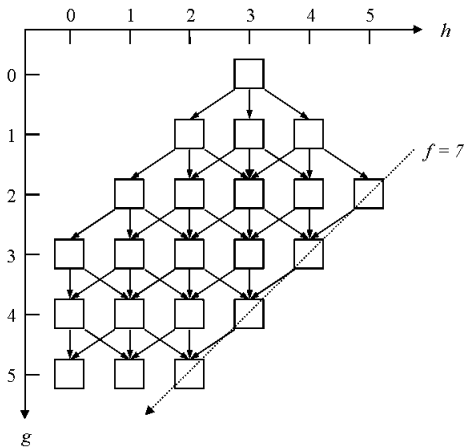
Figure 4: Successor relationships between sets of states with the same g-cost and h-cost illustrate transition locality. Sets of states in the same row have the same g-cost; sets of states in the same column have the same h-cost; and sets of states in the same diagonal have the same f-cost.

two frontiers, and one that involves searching for an optimal path from the intermediate state to the goal state. A similar divide-and-conquer recursion is then used to reconstruct the optimal path for the original search problem.

When the heuristic is very weak, the layers of the breadth-first search graph generated in unidirectional search continue to grow in size past the midpoint of the search. Figure 3 shows this for the Zenotravel problem. A bidirectional search algorithm would not generate these largest layers. Instead, its two frontiers would meet in the middle of the search space and its memory requirements would be roughly equal to twice the size of the middle layer. If the heuristic is very weak, this could be less than the size of the largest layer generated by unidirectional search. In that case, bidirectional search would have an advantage. In fact, the advantage it would have in the case of a very weak heuristic is similar to the advantage that bidirectional search is already known to have in the case of blind search. If the heuristic is strong enough, however, the largest layers will tend to be the middle of the search graph, as they are for the Fifteen Puzzle, and bidirectional search will not have an advantage.

Leveraging transition locality

Recall that in branching partitioning, the transition relation is partitioned by the change in h-cost caused by a transition. In undirected graphs with unit edge costs, and when the heuristic is consistent, it is easy to prove that a transition can only change the h-cost by +1, 0, or -1. To leverage this information to simplify state evaluation, we have already represented the set of states on the frontier of the search graph by a set of BDDs, where each BDD contains states with the same g-cost and h-cost. Figure 4 illustrates this by showing a matrix in which each row corresponds to a distinct g-cost, each column corresponds to a distinct h-cost, and each cell of the matrix (represented by a box or "bucket") corresponds

to a set of states with the same g-cost and the same h-cost. Note that each bucket can be represented by a distinct BDD. Figure 4 shows buckets that are generated when the upper bound on f-cost is 7.

Figure 4 also shows that each bucket of states has only three child buckets, where each child bucket has a g-cost that is one greater than the g-cost of the states in its parent bucket, and the three child buckets correspond to the sets of states with an h-cost that differs from the h-cost of the states in the parent bucket by -1, 0, or +1, respectively. This is a form of *transition locality* that can be exploited in breadth-first heuristic search to reduce memory requirements, as follows. In the algorithm presented in this paper, the previous layer of the search graph is not removed from memory until the current layer is completely expanded. But if buckets in the matrix are expanded in row-major order, as in breadth-first search, it is clear that a bucket with g-cost equal to i and h-cost equal to j can be deleted as soon as the breadth-first search algorithm has finished expanding the bucket with g-cost equal to $i+1$ and h-cost equal to $j+1$. In other words, parts of the previous layer can be removed from memory while the current layer is still being expanded. In practice, this means that breadth-first heuristic search only needs to keep about two layers in memory instead of three. Although this only applies to undirected search graphs with unit edge costs and a consistent heuristic, it could significantly improve efficiency in this case.

Conclusion

We have shown how to synthesize complementary techniques for improving the scalability of state-space search: an approach to state abstraction based on decision diagrams, and a memory-efficient approach to heuristic search that relies on divide-and-conquer solution reconstruction. We have also described how to efficiently compute a symbolic generalization of the max-pair heuristic for domain-independent planning. The resulting algorithm leverages this combination of techniques to achieve state-of-the-art performance in solving a selection of benchmark problems, including propositional planning problems.

Testing the algorithm in additional domains will contribute to a better understanding of the factors that affect its performance. It may also further demonstrate the advantages of an algorithm that uses complementary techniques to improve scalability. Since these techniques will not be equally effective in all domains, when one is less effective in a particular domain, another can compensate.

Although we have described a memory-efficient algorithm for symbolic heuristic search, it is not a memory-bounded algorithm. Eventually, it runs out of memory. Zhou and Hansen (2005) describe a memory-bounded search algorithm called *beam-stack search* that is closely related to breadth-first heuristic search. A symbolic version of this algorithm seems promising. Another promising direction is to integrate symbolic search with external-memory search (Edelkamp 2005). Even when a search algorithm can use external memory, it performs better when it uses internal memory as efficiently as possible, and so, these are also complementary techniques.

References

Bertoli, P.; Cimatti, A.; Roveri, M.; and Traverso, P. 2001. Planning in nondeterministic domains under partial observability via symbolic model checking. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI-01)*, 473–478.

Cimatti, A.; Roveri, M.; and Bertoli, P. 2004. Conformant planning via symbolic model checking and heuristic search. *Artificial Intelligence* 159(1-2):127–206.

Cimatti, A.; Roveri, M.; and Traverso, P. 1998. Automatic OBDD-based generation of universal plans in non-deterministic domains. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI-98)*, 875 – 881.

Clarke, E.; Grumberg, O.; and Peled, D. 2000. *Model Checking*. MIT Press.

Edelkamp, S., and Reffel, F. 1998. OBDDs in heuristic search. In *Proceedings of the 22nd Annual German Conference on Artificial Intelligence*, volume 1504 of *Lecture Notes in Computer Science*, 81–92. Springer.

Edelkamp, S. 2001. Directed symbolic exploration in AI-planning. In *Proceedings of the AAAI Spring Symposium on Model-Based Validation of Intelligence*, 84–92.

Edelkamp, S. 2002. Symbolic pattern databases in heuristic search planning. In *Proceedings of the 6th International Conference on Artificial Intelligence Planning Systems (AIPS-02)*, 274–283.

Edelkamp, S. 2005. External symbolic heuristic search with pattern databases. In *Proceedings of the 15th International Conference on Automated Planning and Scheduling (ICAPS-05)*, 51–60.

Feng, Z., and Hansen, E. 2002. Symbolic heuristic search for factored Markov decision processes. In *Proceedings of the 18th National Conference on Artificial Intelligence (AAAI-02)*, 455–460.

Hansen, E.; Zhou, R.; and Feng, Z. 2002. Symbolic heuristic search using decision diagrams. In *Proceedings of the 5th International Symposium on Abstraction, Reformulation, and Approximation*, volume 2371 of *Lecture Notes in Computer Science*, 83–98. Springer.

Haslum, P., and Geffner, H. 2000. Admissible heuristics for optimal planning. In *Proceedings of the 5th International Conference on Artificial Intelligence Planning System (AIPS'00)*, 140–149. AAAI Press.

Hoey, J.; St-Aubin, R.; Hu, A. J.; and Boutilier, C. 1999. SPUDD: Stochastic planning using decision diagrams. In *Proceedings of the 15th Conference on Uncertainty in Artificial Intelligence (UAI-99)*, 279–288.

Jensen, R., and Veloso, M. 2000. OBDD-based universal planning for synchronized agents in non-deterministic domains. *Journal of Artificial Intelligence Research* 13:189–226.

Jensen, R.; Bryant, R.; and Veloso, M. 2002. SetA*: An efficient BDD-based heuristic search algorithm. In *Proceedings of the 18th National Conference on Artificial Intelligence (AAAI-02)*, 668–673.

Jensen, R. 2003. *Efficient BDD-based planning for non-deterministic, fault-tolerant, and adversarial domains*. Ph.D. Dissertation, Carnegie-Mellon University. Technical report no. CMU-CS-03-139.

Korf, R.; Zhang, W.; Thayer, I.; and Hohwald, H. 2005. Frontier search. *Journal of the ACM* 52(5):715–748.

Korf, R. E. 1985. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence* 27(1):97–109.

Korf, R. 1999. Divide-and-conquer bidirectional search: First results. In *Proc. of the 16th International Joint Conference on Artifiicial Intelligence (IJCAI-99)*, 1184–1189.

Nymeyer, A., and Qian, K. 2003. Heuristic search algorithms based on symbolic data structures. In *Proceedings of the 16th Australian Conference on Artificial Intelligence*, volume 2903 of *Lecture Notes in Computer Science*, 966–979. Springer.

Richards, S. 2004. Symbolic bidirectional breadth-first heuristic search. Master's thesis, Mississippi State University. Department of Computer Science and Engineering.

Yang, B.; Bryant, R. E.; O'Hallaron, D. R.; Biere, A.; Coudert, O.; Janssen, G.; Ranjan, R. K.; and Somenzi, F. 1998. A performance study of BDD-based model checking. In *Formal Methods in Computer-Aided Design FM-CAD'98*, 255–289.

Zhou, R., and Hansen, E. 2003. Sparse-memory graph search. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI-03)*, 1259–1266.

Zhou, R., and Hansen, E. 2005. Beam-stack search: Integrating backtracking with beam search. In *Proceedings of the 15th International Conference on Automated Planning and Scheduling (ICAPS-05)*, 90–98.

Zhou, R., and Hansen, E. 2006. Breadth-first heuristic search. *Artificial Intelligence* 170(4–5):385–408.