State-Set Branching: Leveraging OBDDs for Heuristic Search

Rune M. Jensen Randal E. Bryant
Manuela M. Veloso

September 2002
CMU-CS-02-174

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

This work was supported in part by the Danish Research Agency and the United States Air Force under Grants Nos F30602-00-2-0549 and F30602-98-2-0135. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA), the Air Force, or the US Government.

Abstract

In this paper, we introduce a framework called state-set branching that combines symbolic search based on the reduced Ordered Binary Decision Diagram (OBDD) with classical heuristic search, such as A* and best-first search. The framework relies on an extension of these algorithms from expanding a single state in each iteration to expanding a set of states. We prove that it is generally sound and optimal for two A* implementations and show how a new OBDD technique called branching partitioning can be used to efficiently expand sets of states. The framework is general. It applies to any heuristic function, any evaluation function, and any transition cost function. Moreover, branching partitioning applies to both disjunctive and conjunctive transition relation partitioning. An extensive experimental evaluation involving several new and classical domains proves state-set branching to be a powerful framework. It consistently outperforms single-state A*, except when the heuristic is very strong. In addition, we show that it can improve the complexity of single-state search exponentially and that it often dominates both single-state A* and blind OBDD-based search by several orders of magnitude. Moreover, it has substantially better performance than BDDA*, the only previous OBDD-based implementation of A*.

Keywords: Heuristic Search, OBDD-based Search, Boolean Representation

1 Introduction

Informed or *heuristic* search algorithms such as best-first search, A* [22], and IDA* [30], are considered important contributions of *artificial intelligence* (AI). The advantage of these algorithms, compared to *uninformed* or *blind* search algorithms such as depth-first search (DFS) and breadth-first search (BFS), is that they use heuristics to guide the search toward the goal and in this way significantly reduce the number of visited states. Like DFS and BFS, classical heuristic search algorithms constructs (perhaps implicitly) a *search tree* during the search process. The root of the search tree is the initial state and in each iteration the most promising unexpanded leaf node is expanded by generating its child nodes. The algorithms differ mainly by the way they evaluate nodes. A* is probably the best known of the heuristic search algorithms. Each search node of A* is associated with a cost g of reaching the node and a heuristic estimate h of the remaining cost of reaching the goal. In each iteration, A* expands the node with minimum expected completion cost $f = h + g$. A* can be shown to have much better performance than uninformed search algorithms [42]. However, an unresolved problem for the classical heuristic search algorithms is that the complexity of the algorithms is linear with the number of visited states. This number, however, can grow exponentially if the heuristic has a constant relative error [37]. Such heuristic functions are often encountered in practice, since many heuristics are computed by a relaxation of the search problem that is likely to introduce a relative error.

In *symbolic model checking* [35], a quite different approach has been taken to verify systems with large state spaces. Instead of representing and manipulating sets of states explicitly, this is done implicitly using Boolean functions. Given a bit vector encoding of states, *characteristic functions* can be used to represent subsets of states. In a similar way, a Boolean function can be used to represent the transition relation of a domain and find successor states via Boolean function manipulation. The approach potentially reduces both time and space complexity exponentially. Indeed during the last decade, remarkable results have been obtained using reduced *Ordered Binary Decision Diagrams* (OBDDs [8]) as the Boolean function representation. Systems with more than 10^{100} states have been successfully verified with the OBDD-based model checker SMV [35]. For several reasons, however, only very limited work on using heuristics to guide these implicit search algorithms has been carried out. First of all, the solution techniques considered in formal verification often require traversal of all reachable states making search guidance irrelevant. Secondly, it is nontrivial to efficiently handle search information such as the g and h values associated with individual states when representing states implicitly.

In this paper, we introduce a new framework called *state-set branching* that combines OBDD-based search and heuristic search and efficiently solves the problem of representing search information. State-set branching applies to any search tree based heuristic search algorithm, any transition cost function, heuristic function, and node-evaluation function, and any finite state search problem. The state-set branching framework consists of two independent parts. The first part extends classical single-state heuristic search algorithms to expand sets of states in each iteration. The second part is an efficient OBDD-based implementation of this famility of algorithms based on a partitioning of the transition relation of the search domain called *branching partitioning*. Branching partitioning allows sets of states to be expanded implicitly and sorted according to search information. The approach applies both to disjunctive and conjunctive partitioning [14].

The state-set branching framework has been experimentally evaluated in 8 search domains ranging from VLSI-design with synchronous actions, to classical AI problems such as (n^2-1)-puzzles, Blocks World and problems used in the AIPS 1998, 2000 and 2002 planning competition [32, 2, 33]. We apply four different families of heuristic functions ranging from the minimum Hamming distance to the sum of Manhattan distances for the (n^2-1)-Puzzle, and HSPr [7] for planning problems. The experimental evaluation shows that state-set branching consistently outperforms single-state A*, except when the heuristic is very strong. In addition, we show that it can improve the complexity of single-state search exponentially and that it often dominates both single-state A* and blind OBDD-based search by several orders of magnitude. Moreover, it has substantially better performance than BDDA*, the only previous OBDD-based implementation of A*.

The remainder of this paper is organized as follows. We first describe related work in Section 2. We then define graph search problems in Section 3 and describe a general algorithm for classical single-state heuristic search in Section 4. In Section 5, we generalize this algorithm to expand sets of states and study a number of example applications of the new algorithm. In Section 6, we introduce branching partitioning and other OBDD-based techniques to efficiently implement these algorithms. The experimental evaluation is described in Section 7. Finally, we conclude and discuss directions for future work in Section 8.

2 Related Work

State-set branching is the first general framework for combining heuristic search and OBDD-based search. All previous work has been restricted to particular algorithms. OBDD-based heuristic search has been investigated independently in symbolic model checking and AI. The pioneering work is in symbolic model checking where heuristic search has been used to falsify design invariants by finding error traces. Yuan et al. [49] studies a bidirectional search algorithm pruning frontier states according to their minimum Hamming distance[1] to error states. OBDDs representing Hamming distance equivalence classes are precomputed and conjoined with OBDDs representing the search frontier during search. Yang and Dill [48] also consider minimum Hamming distance as heuristic function in an ordinary best-first search algorithm. They develop a specialized OBDD operation for sorting a set of states according to their minimum Hamming distance to a set of error states. The operation is efficient with linear complexity with respect to the size of the OBDD representing the error states. However, it is unclear how such an operation can be generalized to other heuristic functions. In addition, this approach finds next states and sorts them according to search information in two separate phases.

In general, heuristic OBDD-based search has received little attention in symbolic model checking. The reason is that the main application of OBDDs in this field is verification where all reachable states must be explored. For Computation Tree Logic (CTL) checking [14], guiding techniques have been proposed to avoid a blow-up of intermediate OBDDs during a reachability analysis [6]. However, these techniques are not applicable to search since they are based on defining lower and upper bounds of the fixed-point of reachable states.

In AI, an implementation of A* called BDDA* was developed by Edelkamp and Reffel [17]. BDDA* can use any heuristic function and has been applied to planning as well as model checking [41]. Edelkamp [16] later describes a more general implementation of BDDA* not assuming unit-cost transitions and with cycle detection for monotonic heuristic functions [37]. Both of these

[1] The Hamming distance between two Boolean vectors is the number of bits in the two vectors with different value.

versions of BDDA*, however, are fairly direct implementations of A* with OBDDs that imitates the usual explicit application of the heuristic function via complex symbolic arithmetic. Our experimental results show that the successor state computation of BDDA* scales poorly. For this reason a major philosophy in the design of state-set branching has been to avoid arithmetic operations at the OBDD level. An ADD-based implementation of A* called ADDA* has recently been developed [21]. ADDs [3] generalize OBDDs to finite valued functions. ADDA* is similar to BDDA* but implements cycle detection for general heuristic functions. The ADD may handle arithmetic computations more efficiently than the OBDD [47]. However, ADDA* has not succesfully been shown to have better performance than BDDA* [21].

Other applications of OBDDs in AI include STRIPS planning [10, 16, 45, 19, 27], universal planning [12, 13, 28, 29], conformant planning [11], planning with extended goals [38], and planning under partial observability [5].

3 Search Problems

A search domain is a finite graph where vertices denote world states and edges denote state transitions. Transitions are caused by activity in the world that changes the world state deterministically. Sets of transitions may be defined by *actions*, *operator schemas*, or *guarded commands*. In this paper, however, we will not consider such abstract transition descriptions. If a transition is directed from state s to state s', state s' is said to be a *successor* of s and state s is said to be the *predecessor* of s'. The number of successors emanating from a given state is called the *branching degree* of that state. Since the domain is finite, the branching degree of each state is also finite. Each transition is assumed to have a non-negative *transition cost*.

Definition 1 (Search Domain) *A search domain is a triple $\mathcal{D} = \langle \mathcal{S}, \mathcal{T}, c \rangle$ where $\mathcal{S}$ is a finite set of states, $\mathcal{T} \subseteq \mathcal{S} \times \mathcal{S}$ is a transition relation, and $c : \mathcal{T} \to \mathbb{R}^+$ is a transition cost function.*

A search problem is a search domain with a single initial state and a set of goal states.

Definition 2 (Search Problem) *Let $\mathcal{D} = \langle \mathcal{S}, \mathcal{T}, c \rangle$ be a search domain. A search problem for $\mathcal{D}$ is a triple $\mathcal{P} = \langle \mathcal{D}, s_0, \mathcal{G} \rangle$ where $s_0 \in \mathcal{S}$ and $\mathcal{G} \subseteq \mathcal{S}$.*

A solution π to a search problem is a path from the initial state to one of the goal states. The *solution length* is the number of transitions in π and the *solution cost*, $cost(\pi)$, is the sum of the transition costs of the path. All states reachable from the initial state by a sequence of transitions is called the *state space*.

Definition 3 (Search Problem Solution) *Let $\mathcal{D} = \langle \mathcal{S}, \mathcal{T}, c \rangle$ be a search domain and $\mathcal{P} = \langle \mathcal{D}, s_0, \mathcal{G} \rangle$ be a search problem for $\mathcal{D}$. A solution to $\mathcal{P}$ is a sequence of states $\pi = s_0, \cdots, s_n$ such that $s_n \in \mathcal{G}$, and $\mathcal{T}(s_j, s_{j+1})$ for $j = 0, 1, \cdots, n-1$.*

An *optimal solution* to a search problem is a solution with minimum cost. We will use the symbol C^* to denote the minimum cost. Figure 1 shows a search problem example and an optimal solution.

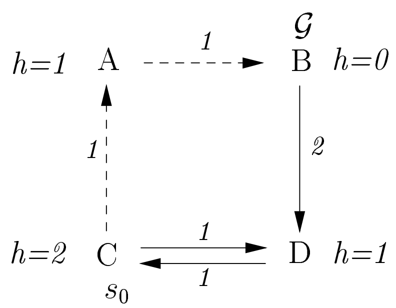

Figure 1: An example search problem consisting of four states, five transitions, initial state $s_0 = \mathrm{C}$, and a single goal state $\mathcal{G} = \{B\}$. The dashed path is an optimal solution. The h-values associated with each states defines a heuristic function used in Section 4.

4 Single-State Heuristic Search

Single-state heuristic search algorithms are characterized by building a search tree superimposed over the state space during the search process. Each *search node* in the tree is a pair $\langle s, I \rangle$ where s is a single state and $I \in \Re^d$ is a d-dimensional real vector representing the search information associated with the node (e.g., the f-value for A*). The root of the search tree contains the initial state. We will assume that the initial state is associated with search information I_0. The leaf nodes of the tree correspond to states that do not have successors in the tree, either because they have not been expanded yet, or because they were expanded, but had no children. At each step, the search algorithm chooses one leaf node to expand. The collection of unexpanded nodes is called the *fringe* or *frontier* of the search tree. It is important to distinguish between the search domain and the search tree. For finite but cyclic search domains, the search tree may be infinite. The heuristic search algorithms differ by what node they select to expand in the frontier. To lower the complexity of the node selection, the frontier is often implemented as a priority queue with the next node to expand at the top. Figure 2 shows a general single-state heuristic search algorithm. The solution extraction function in line 5 simply obtains a solution by tracing back the transitions from the goal node to the root node. The expand function in line 6 finds the set of child nodes of a single node, and enqueueAll inserts each child in the frontier queue.

function General Single-State Heuristic Search
1 $\mathit{frontier} \leftarrow \mathrm{makeQueue}(\langle s_0, I_0 \rangle)$
2 **loop**
3   **if** $|\mathit{frontier}| = 0$ **then return** failure
4   $\langle s, I \rangle \leftarrow \mathrm{removeTop}(\mathit{frontier})$
5   **if** $s \in \mathcal{G}$ **then return** $\mathrm{extractSolution}(\mathit{frontier}, \langle s, I \rangle)$
6   $\mathit{frontier} \leftarrow \mathrm{enqueueAll}(\mathit{frontier}, \mathrm{expand}(\langle s, I \rangle))$

Figure 2: A general single-state heuristic search algorithm.

The A* algorithm is probably the best known and most well studied of the single-state heuristic search algorithms. A* sorts the unexpanded nodes in the priority queue in ascending order of the search information given by a *heuristic evaluation function* f. The evaluation function is defined by

$$f(n) = g(n) + h(n),$$

where $g(n)$ is the cost of the path in the search tree leading from the root node to n, and $h(n)$ is a *heuristic function* estimating the cost of a minimum cost path leading from the state in n to some goal state. Thus $f(n)$ measures the minimum cost over all solution paths constrained to go through the state in n. The search tree built by A* for the example problem and heuristic function defined in Figure 1 is shown in Figure 3.

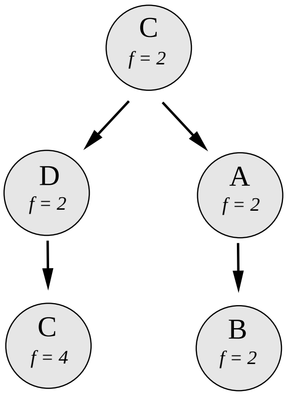

Figure 3: Search tree example.

The properties of A* have been surveyed by Pearl [37]. A* is *sound* and *complete*, since the node expansion operation is assumed to be correct, and infinite cyclic paths have unbounded cost. A* further finds optimal solutions if the heuristic function $h(n)$ is *admissible*, that is, if $h(n) \leq h^*(n)$ for all n, where $h^*(n)$ is the minimum cost of a path going from the state in n to a goal state. A* is *optimally efficient* for any admissible heuristic function. That is, no other optimal algorithm is guaranteed to expand fewer nodes than A* [15]. It can be shown that every node on the frontier with $f(n) < C^*$ eventually will be expanded by A*. Thus, the complexity of A* is directly tied to the accuracy of the estimates provided by $h(n)$. When A* employs a perfectly informed heuristic ($h(n) = h^*(n)$), it is guided directly toward the closest goal. At the other extreme, when no heuristic at all is available ($h(n) = 0$), the search becomes exhaustive, normally yielding exponential complexity. In general, A* has linear complexity if the absolute error of the heuristic function is constant, but it may have exponential complexity if the relative error is constant . Subexponential complexity requires that the growth rate of the error is logarithmically bounded [42]

$$|h(n) - h^*(n)| \in O(\log h^*(n)).$$

The complexity results are disappointing due to the fact that practical heuristic functions often are based on a relaxation of the search problem that causes

$h(n)$ to have constant or near constant relative error. The results show that practical application of A* still may be very search intensive. Often better performance of A* can be obtained by weighting the g and h-component of the evaluation function [39]

$$f(n) = (1-w)g(n) + wh(n), \text{ where } w \in [0,1]. \tag{1}$$

Weights $w = 0.0, 0.5$, and 1.0 correspond to uniform cost search (Dijkstra's

```
function General State-Set Heuristic Search
frontier ← makeQueue(⟨{s₀}, I₀⟩)
loop
  if |frontier| = 0 then return failure
  ⟨S, I⟩ ← removeTop(frontier)
  if S ∩ 𝒢 ≠ ∅ then return extractSolution(frontier, ⟨S ∩ 𝒢, I⟩)
  frontier ← enqueueAndMerge(frontier, stateSetExpand(⟨S, I⟩))
```

Figure 4: A general state-set heuristic search algorithm.

algorithm), A*, and best-first search. Weighted A* is optimal in the range $[0.0, 0.5]$ but often finds solutions faster in the range $(0.5, 1]$.

Another drawback of A* is that its space complexity is very high due to the explicit representation of the search tree. For that reason an iterative deepening version of A* called IDA* has been developed [30]. This algorithm has space complexity linear with the search depth. However, unless the search domain is a tree, it may perform a highly redundant search.

5 State-Set Branching

The state-set branching framework has two independent parts: a modification of the general single-state heuristic search algorithm to a new algorithm called *general state-set heuristic search*, and a collection of OBDD-based techniques for implementing the new algorithm efficiently. In this section, we will describe the general state-set heuristic search algorithm. In next section, we show how it is implemented with OBDDs.

5.1 General State-Set Heuristic Search

Assume that each transition $\mathcal{T}(s, s')$ for a particular heuristic search algorithm changes the search information with $\delta I(s, s')$. Thus if s is associated with search information I and s' is reached by $\mathcal{T}(s, s')$ then s' will be associated with search information $I + \delta I(s, s')$. For A*, the search information can be one or two dimensional: either it is the f-value or the g and h-value of a search node. In the first case $\delta I(s, s')$ is the f-value change caused by the transition. The δf values of our example problem are shown in Figure 5. The general state-set heuristic search algorithm shown in Figure 4 is

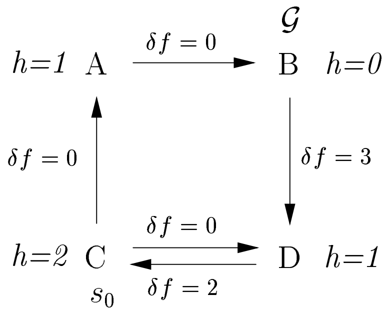

Figure 5: The example search problem with δf values.

almost identical to the general single-state heuristic search algorithm defined in Figure 2. However, the state set version traverses a search tree during the search process where each search node contains a set of states associated with the same search information. Multiple states in each node emerge because child nodes having identical search information are coalesced by the stateSetExpand function in line 6 and because the enqueueAndMerge function may merge child nodes with nodes on the *frontier* queue having identical search information. The state-set expansion function is defined in Figure 6. The next states of some child associated with search information I are stored in *child*$[I]$. The outgoing transitions from each state in the parent node are used to find all successor states. The function makeNodes called at line 6 constructs the child nodes from the completed child map. Each child node contains states having identical search information. However, there may exist several nodes with the same search information. In addition, makeNodes may prune some of the child states (e.g., to implement cycle detection in A*).

function stateSetExpand($\langle S, I \rangle$)
$\quad$ $\mathit{child} \leftarrow$ emptyMap
$\quad$ **foreach** state s **in** S
$\quad\quad$ **foreach** transition $\mathcal{T}(s, s')$
$\quad\quad\quad$ $I_c \leftarrow I + \delta I(s, s')$
$\quad\quad\quad$ $\mathit{child}[I_c] \leftarrow \mathit{child}[I_c] \cup \{s'\}$
$\quad$ **return** makeNodes(child)

Figure 6: The state set expand function.

As an example, Figure 7 shows the search tree traversed by the state-set search algorithm for A* applied to our example problem. In order to reduce

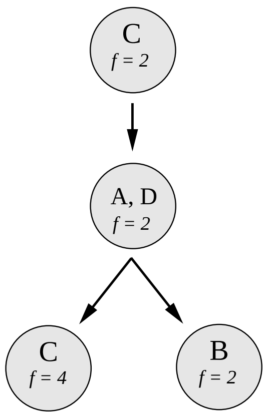

Figure 7: State-set search tree example.

the number of search nodes even further, the enqueue function of the general state-set heuristic search algorithm may merge nodes on the search frontier having identical search information. This, however, transforms the search tree into a *Directed Acyclic Graph* (DAG).

Lemma 1 *The search structure build by the general state-set search algorithm is a DAG where every node $\langle S', I' \rangle$ different from a root node $\langle \{s_0\}, I_0 \rangle$ has a set of predecessor nodes. For each state $s' \in S'$ in such a node there exists a predecessor $\langle S, I \rangle$ with a state $s \in S$ such that $\mathcal{T}(s, s')$ and $I' = I + \delta I(s, s')$.*

Proof: By induction on the number of loop iterations, we get that the search structure after the first iteration is a DAG consisting of a root node $\langle \{s_0\}, I_0 \rangle$. For the inductive step, assume that the search structure is a DAG with the desired properties after n iterations of the loop (see Figure 4). If the algorithm in the next iteration terminates in line 3 or 5, the search structure is unchanged and therefore a DAG with the required format. Assume that the algorithm does not terminate and that $\langle S, I \rangle$ is the node removed from the top of *frontier*. The node is expanded by forming child nodes with the stateSetExpand function in line 6. According to the definition of this function, any state $s' \in S'$ in a child node $\langle S', I' \rangle$ has some state $s \in S$ in $\langle S, I \rangle$ such that $\mathcal{T}(s, s')$ and $I' = I + \delta I(s, s')$. Thus $\langle S, I \rangle$ is a valid predecessor for all states in the child nodes. Furthermore, since all child nodes are new nodes, no cycles are created in the search structure which therefore remains a DAG. If a child node is merged with an old node when enqueued on *frontier* the resulting search structure is still a DAG because all nodes on *frontier* are unexpanded and therefore have no successor nodes that can cause cycles. In addition, each state in the resulting node obviously has the required predecessor nodes. □

Lemma 2 *For each state $s' \in S'$ of a node $\langle S', I' \rangle$ in a finite search structure of the general state-set heuristic search algorithm there exists a path $\pi = s_0, \cdots, s_n$ in $\mathcal{D}$ such that $s_n = s'$ and $I = I_0 + \sum_{i=0}^{n-1} \delta I(s_i, s_{i+1})$.*

Proof: We will construct π by tracing the edges backwards in the search structure. Let $b_0 = s'$. According to Lemma 1 there exists a predecessor $\langle S, I \rangle$ to $\langle S', I' \rangle$ such that for some state $b_1 \in S$ we have $\mathcal{T}(b_1, b_0)$ and $I' = I + \delta I(b_1, b_0)$. Continuing the backward traversal from b_1 must eventually terminate since the search structure is finite and acyclic. Moreover, the traversal will terminate at the root node because this is the only node without predecessors. Assume that the backward traversal terminates after n iterations. Then $\pi = b_n, \cdots, b_1$. $|Box$

The extractSolution function in line 5 of the general state-set heuristic search algorithm uses the backward traversal described in the proof of Lemma 2 to extract a solution. We can now prove soundness of the general state-set heuristic search algorithm.

Theorem 1 *The general state-set heuristic search algorithm is sound.*

Proof: Assume that the algorithm returns a path $\pi = s_0, \cdots, s_n$ with search information I. Since $s_n \in \mathcal{G}$ it follows from Lemma 2 and the definition of extractSolution that π is a solution to the search problem associated with search information I. $\Box$

It is not possible to show that the general state-set heuristic search algorithm is complete since it covers incomplete algorithms such as best-first search.

5.2 Example Implementations

The general state-set heuristic search algorithm can be used to implement variants of best-first search, A*, weighted A*, uniform cost search, beam search, and hill climbing. With some modifications, it also covers iterative deepening heuristic search algorithms such as IDA*.

Best-first search is implemented by using the values of the heuristic function as search information and sorting the nodes on the frontier in ascending order, such that the top node contains states with least h-value. The search information of the initial state is $I_0 = h(s_0)$ and each transition $\mathcal{T}(s, s')$ is associated with the change in h, that is, $\delta I(s, s') = h(s') - h(s)$. In each iteration, this best-first search algorithm will expand all states with least h-value on the frontier.

A* can be implemented by setting $I_0 = h(s_0)$ and $\delta I(s, s') = c(s, s') + h(s') - h(s)$ such that the search information equals the f-value of search nodes. Again nodes on the frontier are sorted in ascending order. We call this implementation *fSetA**. An A* implementation with cycle detection must keep track of the g and h separately and prune child states reached previously with a lower g-value. Thus, $I_0 = (0, h(s_0))$ and $\delta I(s, s') = (c(s, s'), h(s') - h(s))$. The frontier is, as usual, sorted according to the evaluation function $f(n) = g(n) + h(n)$. The resulting algorithm is called *ghSetA**. Compared to fSetA*, ghSetA* does not merge nodes that have identical f-value but different g and h-values. In each iteration, it may therefore only expand a subset of the states on the frontier with minimum f-value. A number of other improvements have been integrated in ghSetA*. First, it uses a tie breaking rule for nodes with identical f-value choosing the node with the least h-value. Thus, in situations where all nodes on the frontier have $f(n) = C^*$, the algorithm focuses the search in a DFS fashion. The reason is that a node at depth level d in this situation must have greater h-value than a node at level $d + 1$ due to the non-negative transition costs. In addition, it only merges two nodes on the frontier if the space used by the resulting node is less than an upper-bound u. This may help to focus the search further in situations where the space requirements of the frontier nodes grow fast with the search depth. Both ghSetA* and fSetA* can easily be extended to the weighted A* algorithms described in Section 5. Using an approach similar to Pearl [37], fSetA* and ghSetA* can be shown to be optimal given an admissible heuristic. In particular this is true when using the trivial admissible heuristic function $h(n) = 0$ of uniform cost search. The proofs are given in the Appendix.

IDA* performs a depth-first search in the search tree bounded by a limit f_{limit} on the f-values of search nodes. Initially f_{limit} is equal to the f-value of the initial state. In each iteration f_{limit} is increased by the minimum value that the previous search exceeded f_{limit} by. A similar algorithm can be defined for a state-set search tree, where child nodes with identical f-values are combined.

6 OBDD-based Implementation

The motivation for defining the general state-set heuristic search algorithm is that it can be efficiently implemented with OBDDs. In this section, we describe how to represent sets of states implicitly with OBDDs and develop a technique called *branching partitioning* for expanding search nodes efficiently.

6.1 The OBDD Representation

The OBDD is a canonical representation of a Boolean function with linear ordered arguments. It is a rooted DAG with one or two terminal nodes labeled 1 or 0, and a set of variable nodes. Each variable node is associated with a Boolean variable of the function the OBDD represents. It has two outgoing edges *low* and *high*. Given an assignment of the arguments, the value of the Boolean function is determined by a path starting at the root node and recursively following the high edge, if the associated variable is true, and the low edge, if the associated variable is false. The function value is *True*, if the label of the reached terminal node is 1; otherwise it is *False*. The graph is ordered such that all paths in the graph respect the ordering of the variables. An OBDD representing the function $f(x_1, x_2) = x_1 \wedge x_2 \vee \neg x_1 \wedge \neg x_2$ is shown in Figure 8. OBDDs are canonical due to two reduction rules ensuring that no

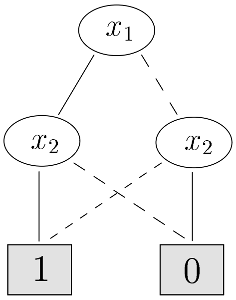

Figure 8: An OBDD representing the function $f(x_1, x_2) = x_1 \wedge x_2 \vee \neg x_1 \wedge \neg x_2$. High and low edges are drawn with solid and dashed lines, respectively.

two distinct nodes have the same variable name and low and high successors, and no variable node has identical low and high successors. The OBDD representation has several advantages: first, many functions encountered in practice (e.g., symmetric functions) have polynomial size, second, graphs of several OBDDs can be shared and efficiently manipulated in multi-rooted OBDDs, third, with the shared representation, equivalence and satisfiability tests on OBDDs take constant time, and finally, fourth, binary synthesis $x \otimes y$ has time and space complexity $O(|x||y|)$ [8]. A disadvantage of OBDDs is that there may be an exponential size difference depending on the ordering of the variables. However, powerful heuristics exist for finding good variable orderings [36].

6.2 OBDD-based State Space Exploration

OBDDs were originally applied to digital circuit verification [1]. More relevant, however, for the work presented in this paper, they were later applied in *model checking* using a range of techniques collectively coined *symbolic model checking* [35]. During the last decade OBDDs have successfully been applied to verify very large transition systems. The essential computation applied in symbolic model checking is an efficient rechability analysis where OBDDs are used to represent sets of states and the transition relation.

Search problems can be solved using the standard machinery developed in symbolic model checking. Let $\mathcal{D} = \langle \mathcal{S}, \mathcal{T}, c \rangle$ be a search domain. Since the number of states $\mathcal{S}$ is finite, a vector of Boolean variables $\vec{v} \in \mathbb{B}^{|\vec{v}|}$ can be used to represent the state space. The variables in $\vec{v}$ are called *state variables* A set of states S can be represented by a Boolean function on $\vec{v}$

equal to the characteristic function of S. Thus, an OBDD can represent any set of states. The main efficiency of the OBDD representation is that the cardinality of the represented set is not directly related to the size of the OBDD. For instance the OBDD equal to the terminal node 1 represents all states in the domain no matter how many there are. In addition, the set operations union, intersection and complementation simply translate into disjunction, conjunction, and negation on OBDDs.

In a similar way, the transition relation can be represented by a Boolean function $T(\vec{v}, \vec{v}\,') = \mathcal{T}(S_{\vec{v}}, S_{\vec{v}\,'})$. We refer to $\vec{v}$ and $\vec{v}\,'$ as *current* and *next state variables*, respectively. The symbol $S_{\vec{v}}$ denotes the state encoded by $\vec{v}$. To make this clear, two Booelan variables $\vec{v} = (v_0, v_1)$ is used in Figure 9 to represent the four states of our example problem. The initial state is

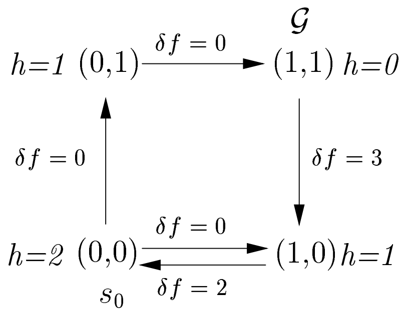

Figure 9: Boolean state encoding of the example search problem.

represented by an OBDD of the expression $\neg v_0 \wedge \neg v_1$. Similarly we have $\mathcal{G} = v_0 \wedge v_1$. The transition relation is represented by an OBDD equal to the Boolean function

$$\begin{aligned} T(\vec{v}, \vec{v}\,') \quad = \quad & \neg v_0 \wedge \neg v_1 \wedge v_0' \wedge \neg v_1' \quad \vee \quad \neg v_0 \wedge \neg v_1 \wedge \neg v_0' \wedge v_1' \\ \vee \quad & \neg v_0 \wedge v_1 \wedge v_0' \wedge v_1' \quad \vee \quad v_0 \wedge v_1 \wedge v_0' \wedge \neg v_1' \\ \vee \quad & v_0 \wedge \neg v_1 \wedge \neg v_0' \wedge \neg v_1'. \end{aligned}$$

The crucial idea in OBDD-based or symbolic search is to stay at the OBDD level when finding the next states of a set of states. A set of next states can be found by computing the *image* of a set of states S encoded in current state variables

$$\mathrm{image}(S(\vec{v})) = \Big(\exists \vec{v} \,.\, S(\vec{v}) \wedge T(\vec{v}, \vec{v}\,')\Big)[\vec{v}\,'/\vec{v}].$$

The previous states of a set of states is called the *preimage* and are computed in a similar fashion. The operation $[\vec{v}\,'/\vec{v}]$ is a regular variable substitution. Existential quantification is used to abstract variables in an expression. Let v_i be one of the variables in the expression $e(v_0, \cdots, v_n)$, we then have

$$\exists v_i \,.\, e(v_0, \cdots, v_n) \;=\; e(v_0, \cdots, v_n)[v_i/\mathit{False}] \;\vee\; e(v_0, \cdots, v_n)[v_i/\mathit{True}].$$

Existentially quantifying a Boolean variable vector involves quantifying its variables in turn.

To illustrate the image computation, consider the first step of a search from s_0 in the example problem. We have $S(v_0, v_1) = \neg v_0 \wedge \neg v_1$. Thus

$$\begin{aligned}
\mathrm{image}(S(v_0, v_1)) &= (\exists (v_0, v_1)\,.\, \neg v_0 \wedge \neg v_1 \wedge T(v_0, v_1, v_0', v_1'))[(v_0', v_1')/(v_0, v_1)] \\
&= (v_0' \wedge \neg v_1' \;\vee\; \neg v_0' \wedge v_1')[(v_0', v_1')/(v_0, v_1)] \\
&= v_0 \wedge \neg v_1 \;\vee\; \neg v_0 \wedge v_1.
\end{aligned}$$

It is straight forward to implement uninformed or blind OBDD-based search algorithms using the image and preimage computations. All sets and set operations in these algorithms are implemented with OBDDs. The forward breadth-first search algorithm, shown in Figure 10, computes the set of frontier states with the image computation. The set *reached* contains all explored states and is used to prune a new frontier from previously visited states. Backward breadth-first search can be implemented in a similar fash-

function Forward Breadth-First Search
$\quad$ *reached* $\leftarrow \emptyset$; $\mathit{forwardFrontier}_0 \leftarrow \{s_0\}$; $i \leftarrow 0$
$\quad$ **while** $\mathit{forwardFrontier}_i \cap \mathcal{G} = \emptyset$
$\quad\quad$ $i \leftarrow i + 1$
$\quad\quad$ $\mathit{forwardFrontier}_i \leftarrow \mathrm{image}(\mathit{forwardFrontier}_{i-1}) \setminus \mathit{reached}$
$\quad\quad$ $\mathit{reached} \leftarrow \mathit{reached} \cup \mathit{forwardFrontier}_i$
$\quad\quad$ **if** $\mathit{forwardFrontier}_i = \emptyset$ **return** failure
$\quad$ **return** extractSolution(*forwardFrontier*)

Figure 10: OBDD-based Forward Breadth-First Search.

ion using the preimage to find the frontier states. The two algorithms are easily combined into a bidirectional search algorithm. In each iteration, this algorithm either computes the frontier states in forward or backward direction. If the set of frontier states is empty the algorithm returns failure. If an overlap between the frontier states and the reached states in the opposit direction is found the algorithm extracts and returns a solution. Otherwise the search continues. A good heuristic for deciding which direction to search in is simply to choose the direction where the previous frontier took least time to compute. When using this heuristic, bidirectional search has similar or better performance than both forward and backward search, since it will transform into one of these algorithms if the frontiers always are faster to compute in a particular direction.

6.3 Partitioning

A common problem when computing the image and preimage is that the intermediate OBDDs tend to be large compared to the OBDD representing the result. Another problem is that the transition relation may grow very large if represented by a single OBDD (a *monolithic* transition relation). In symbolic model checking one of the most successful approaches to solve these problems is *transition relation partitioning*. For search problems, where each transition only modifies a small subset of the state variables, the suitable partitioning technique is *disjunctive partitioning* [14]. In a disjunctive partitioning, unmodified next state variables are unconstrained in the transition expressions. The abstracted transition expressions are partitioned according to which variables they modify. For our example, we get two partitions: P_1 consisting of transitions $(0,0) \rightarrow (1,0)$, $(0,1) \rightarrow (1,1)$, and $(1,0) \rightarrow (0,0)$ that modify variable v_0, and P_2 consisting of transitions $(0,0) \rightarrow (0,1)$ and $(1,1) \rightarrow (1,0)$ that modify variable v_1

$$\begin{aligned}
P_1 &= \neg v_0 \wedge \neg v_1 \wedge v'_0 \;\vee\; \neg v_0 \wedge v_1 \wedge v'_0 \;\vee\; v_0 \wedge \neg v_1 \wedge \neg v'_0, \qquad && \vec{m}_1 = (v_0) \\
P_2 &= \neg v_0 \wedge \neg v_1 \wedge v'_1 \;\vee\; v_0 \wedge v_1 \wedge \neg v'_1, && \vec{m}_2 = (v_1).
\end{aligned}$$

In addition to large space savings, disjunctive partitioning often reduces the complexity of the image (and preimage) computation that now may skip the quantification of unchanged variables and operate on smaller expressions [9, 40]

$$Image(S(\vec{v})) = \bigvee_{i=1}^{|\mathbf{P}|} \Big(\exists \vec{m}_i \,.\, S(\vec{v}) \wedge P_i(\vec{v}, \vec{m}'_i) \Big) [\vec{m}'_i / \vec{m}_i].$$

In domains where the transitions represent synchronous composition of parallel activity (e.g., in multi-agent domains [28]) disjuncitve partitioning often becomes intractable due to a large number of transitions. In this case, a dual partitioning technique called *conjunctive partitioning* may apply. Assume that activity j only modifies state varaibles $\vec{m}_j$ as defined by its local transition relation $P_j(\vec{v}, \vec{m}'_j)$. Given a total of n activities, we then have

$$T(\vec{v}, \vec{v}\,') = P_1(\vec{v}, \vec{m}'_1) \wedge \cdots \wedge P_n(\vec{v}, \vec{m}'_n).$$

Although existential quantification does not distribute over conjunction, subexpressions can be moved out of the scope of existential quantification if they do not depend on any of the variables being quantified. To simplify the presentation, assume that the activities modify disjoint sets of variables. An efficient image computation combining conjunctive partitioning with early quantification is then defined by

$$Image(S(\vec{v})) = (\exists \vec{m}_n \,.\, (\cdots (\exists \vec{m}_1 \,.\, S(\vec{v}) \wedge P_1(\vec{v}, \vec{m}'_1)) \cdots) \wedge P_n(\vec{v}, \vec{m}'_n))[\vec{v}\,'/\vec{v}].$$

In a similar fashion an efficient preimage computation can be defined.

6.4 The OBDD-based General State-Set Heuristic Search Algorithm

The general state-set heuristic search algorithm represents the states in each search node by an OBDD. This may lead to exponential space savings compared to the explicit state representation used by the single-state heuristic search algorithms. However, if we want the exponential space saving to translate into an exponential time saving, we also need an implicit approach for computing the expand operation. The image computation can be applied to find all next states of a set of states implicitly, but we need a way to partition the next states into child nodes with the same search information. The expand operation could be carried out in two phases, where the first finds all the next states using the image computation, and the second splits this set of states into child nodes [48]. A more direct approach, however, is to split up the image computation such that the two phases are combined into one. We call this branching partitioning.

For disjunctive partitioning the approach is straight-forward. We simply ensure that each partition contains transitions with the same search information change. The result is called a *disjunctive branching partitioning.* For our example it is

$$
\begin{array}{lll}
P_1 = \neg v_0 \wedge \neg v_1 \wedge v_0' \;\vee\; \neg v_0 \wedge v_1 \wedge v_0', & \vec{m}_1 = (v_0), & \delta f_1 = 0 \\
P_2 = v_0 \wedge \neg v_1 \wedge \neg v_0', & \vec{m}_2 = (v_0), & \delta f_2 = 2 \\
P_3 = \neg v_0 \wedge \neg v_1 \wedge v_1', & \vec{m}_3 = (v_1), & \delta f_3 = 0 \\
P_4 = v_0 \wedge v_1 \wedge \neg v_1', & \vec{m}_4 = (v_1), & \delta f_4 = 3.
\end{array}
$$

Notice, that there may exist several partitions with the same information change. In practice, it is often more efficient to merge some of these partitions even though more variables will be modified by the resulting partitions. Assume that $\mathbf{P}$ is a disjunctive branching partition. The stateSetExpand function in Figure 6 can then be implemented with OBDDs as shown in Figure 11.

function disjunctiveStateSetExpand$(\langle S(\vec{v}), I \rangle)$
child $\leftarrow$ emptyMap
for $i = 1$ **to** $|\mathbf{P}|$
  $I_c \leftarrow I + \delta I_i$
  $child[I_c] \leftarrow child[I_c] \cup \text{image}(P_i, S(\vec{v}))$
return makeNodes(*child*)

Figure 11: The state set expand function for a disjunctive branching partitioning.

An efficient implicit node expansion computation is also possible to define for conjunctive partitioning if we assume that the search information change of a global transition is equal to the sum of the information change of each of its local transitions. Recall that each partition of a conjunctive partitioning is equal to the transition relation of a single activity. In a *conjunctive branching partitioning* each of these transition relations is disjunctively partitioned such that each subpartition contains transitions with the same search information change. Let $P_i^j(\vec{v}, \vec{m}'_i)$ denote subpartition j of activity i with search information change δI_i^j. The state-set expansion function is then defined as shown in Figure 12. In the worst case, the number of child nodes will grow exponentially with the number of activities. However, in practice this blow-up of child nodes may be avoided due to the merging of nodes with identical search information during the computation.

function conjunctiveStateSetExpand($\langle S(\vec{v}), I\rangle$)
$\quad$ *child* $\leftarrow$ emptyMap
$\quad$ $child[I] = S(\vec{v})$
$\quad$ **for** $i = 1$ **to** n
$\quad\quad$ *newChild* $\leftarrow$ emptyMap
$\quad\quad$ **foreach** entry $\langle S(\vec{v}, \vec{v}\,'), \delta I\rangle$ **in** *child*
$\quad\quad\quad$ **for** $j = 1$ **to** $|\mathbf{P}_i|$
$\quad\quad\quad\quad$ $I_c \leftarrow \delta I + \delta I_i^j$
$\quad\quad\quad\quad$ $newChild[I_c] \leftarrow newChild[I_c] \cup \exists \vec{m}_i \,.\, S(\vec{v}, \vec{v}\,') \wedge P_i^j(\vec{v}, \vec{m}\,'_i)$
$\quad\quad$ *child* $\leftarrow$ *newChild*
$\quad$ **return** makeNodes($child[\vec{v}\,'/\vec{v}]$)

Figure 12: The state set expand function for a conjunctive branching partitioning.

7 Experimental Evaluation

Even though weighted A* and best-first search are subsumed by the state-set branching framework, the experimental evaluation in this paper focuses on algorithms performing search similar to A*. There are several reasons for this. First, we are interested in finding optimal or near optimal solutions, and for best-first search, the whole emphasis would be on the quality of the heuristic function rather than the efficiency of the search approach. Second, the behavior of A* has been extensively studied, and finally, we compare with BDDA*. Readers interested in the performance of state-set branching algorithms of weighted A* with other weight settings than $w = 0.5$ (see Equation 1) are referred to [25].

We have implemented a general search engine in C++/STL using the BuDDy OBDD package[2] [31]. This package has two major parameters: 1) the number of OBDD-nodes allocated to represent the shared OBDD (n), and 2) the number of OBDD nodes allocated to represent OBDDs in the operator caches used to implement dynamic programming (c). The input to the search engine is a search problem defined in the STRIPS part of PDDL

[2]We also made experiments [24] using the CUDD package [44], but did not obtain significantly better results than with the BuDDy package.

[34] or an extended version of *NADL* [28] with action costs. The output of the search engine is a solution found by one of the following six search algorithms:

ghSetA*	:	The ghSetA* algorithm with evaluation function $f(n) = g(n) + h(n)$.
fSetA*	:	The fSetA* algorithm with evaluation function $f(n) = g(n) + h(n)$.
Bidir	:	OBDD-based blind breadth-first bidirectional search using the heuristic for choosing search direction described in Section 6.2.
A*	:	Single-state A* with cycle detection, explicit state manipulation, and evaluation function $f(n) = g(n) + h(n)$.
BDDA*	:	The BDDA* algorithm as described in [17].
iBDDA*	:	An improved version of BDDA* described below.

The ghSetA*, fSetA*, and Bidir search algorithms have been implemented as described in this paper. The A* algorithm manipulates and represents states explicitly. It has been implemented in several versions to cover the range of different problems studied. The most general of them is an algorithm for planning problems with states encoded explicitly as sets of facts and actions represented in the usual STRIPS fashion. All of the single-state A* algorithms are implemented with cycle detection. The BDDA* algorithm has been implemented as described in [17]. The algorithm presented in this paper is shown in Figure 13. It can only solve search problems in domains with unit transition costs. The search frontier is represented by a single OBDD $open(\vec{f}, \vec{v})$. This OBDD is the characteristic function of a set of states paired with their f-value. The state is encoded as usual by a Boolean vector $\vec{v}$ and the f-value is encoded in binary by the Boolean vector $\vec{f}$. Similar to fSetA*, BDDA* expands all states $min(\vec{v})$ with minimum f-value (f_{min}) in each iteration. The f-value of the child states is computed by arithmetic operations at the OBDD level (line 5 and 6). The change in h-value is found by applying a symbolic encoding of the heuristic function to the child and parent state. BDDA* is able to find optimal solutions, but the algorithm only returns the path cost of such solutions. In our implementation, we therefore added a function for tracing a solution backward. In the domains we have investigated, this extraction function has low complexity, as did those for ghSetA* and fSetA*. Our investigation of the BDDA* algorithm shows that

function BDDA$*$

1 $open(\vec{f}, \vec{v}) \leftarrow h(\vec{f}, \vec{v}) \wedge s_0(\vec{v})$

2 **while** $(open \neq \emptyset)$

3 $\quad (f_{min}, min(\vec{v}), open'(\vec{f}, \vec{v})) \leftarrow \text{goLeft}(open)$

4 $\quad$ **if** $(\exists \vec{v} \,.\, (min(\vec{v}) \wedge \mathcal{G}(\vec{v}))$ **return** f_{min}

5 $\quad open''(\vec{f}\,', \vec{v}\,') \leftarrow \exists \vec{v} \,.\, min(\vec{v}) \wedge T(\vec{v}, \vec{v}\,') \wedge$

6 $\quad\quad \exists \vec{e} \,.\, h(\vec{e}, \vec{v}) \wedge \exists \vec{e}\,' \,.\, h(\vec{e}\,', \vec{v}\,') \wedge (\vec{f}\,' = f_{min} + \vec{e}\,' - \vec{e} + 1)$

7 $\quad open(\vec{f}, \vec{v}) \leftarrow open'(\vec{f}, \vec{v}) \vee open''(\vec{f}\,', \vec{v}\,')[\vec{f}\,' \setminus \vec{f}, \vec{v}\,' \setminus \vec{v}]$

Figure 13: The BDDA* algorithm.

it often can be improved by: (1) Defining a computation of $open''$ using a disjunctive partitioned transition relation instead of monolithic transition relation as in line 5 and 6, (2) Precomputing the arithmetic operation at the end of line 6 for each possible f-value, (3) Interleaving the OBDD variables of $\vec{f}$, $\vec{e}$, and $\vec{e}\,'$ to improve the arithmetic OBDD operations, and (4) Moving this block of variables to the middle of the OBDD variable ordering to reduce the average distance to dependent state variables. The last improvement is actually antagonistic to the recommendation of the BDDA* inventors who locate the $\vec{f}$ variables at the beginning of the variable ordering to simplify the goLeft operation. However, we get up to two times speed up with the above modification. The improved algorithm is called iBDDA*.

In order to factor out differences due to state encodings and OBDD computations, all OBDD-based algorithms use the same bit vector representation of states, the same variable ordering of the state variables, and similar space allocation and cache sizes of the OBDD package. We believe, we did an extensive empirical validation. However, it is necessary since a dissimilarity in just one of the above mentioned properties may cause an exponential performance difference. All algorithms share as many subcomputations as possible, but redundant or unnecessary computations are never carried out for a particular instantiation of an algorithm. The following table shows the performance parameters of the search engine:

T_{total}	:	The total elapsed CPU time of the search engine.
T_{rel}	:	Time to generate the transition relation. For BDDA* and iBDDA*, this also includes building the symbolic representation of the heuristic function and f-formulas.
T_{search}	:	Time to search for and extract a solution.
$\|sol\|$	:	Solution length.
$\|expand\|$	:	For Bidir this is the average size of the OBDDs representing the search frontier. For fSetA* and ghSetA*, it is the average size of OBDDs of search nodes being expanded. For BDDA* and iBDDA*, it is the average size of $open''$.
$\|maxQ\|$	:	Maximal number of nodes on the frontier queue.
$\|T\|$	:	The sum of the OBDDs representing the partitioned transition relation.
it	:	Number of iterations of the algorithm.

Time is measured in seconds. The time $T_{total} - T_{rel} - T_{search}$ is spent on allocating memory for the OBDD package, parsing the problem description and in case of PDDL problems analysing the problem in order to make a compact Boolean state encoding. All experiments are carried out on a Linux 7.1 PC with kernel 2.4.16, 500 MHz Pentium III CPU, 512 KB L2 cache and 512 MB RAM. Time out and out of memory are indicated by *Time* and *Mem*. Time out changes between the experiments. The algorithms are out of memory when they start page faulting to the hard drive at approximately 450 MB RAM.

Our experiments cover a wide range of search domains and heuristics. The first domain FG^k is artificial and uses the minimum Hamming distance as heuristic function. It demonstrates that state-set branching may have exponentially better performance than single-state A*. Next, we consider the $D^xV^yM^z$ puzzle and the 24 and 35 Puzzle using minimum Hamming distance and sum of Manhattan distance as heuristic function, respectively. We then consider a number of STRIPS [18] planning problems from the AIPS planning competitions [32, 2, 33] using the HSPr heuristic [7], and finally we study the channel routing problem from VLSI design using a specialized heuristic function.

7.1 FG^k

This problem is a modification of Barret and Weld's D^1S^1 problem [4] and has been constructed to show that state-set branching may have exponentially better performance than single-state A*. The problem is easiest to describe in STRIPS. Thus, a state is a set of facts and actions are fact triples defining sets of transitions. In a given state S, an action defined by $\langle pre, add, del \rangle$ is applicable if $pre \subseteq S$, and the resulting state is $S' = (S \cup add) \setminus del$. The actions are

$$
\begin{array}{lll}
\mathbf{A}_1^1 & & \\
pre & : & \{F^*\} \\
add & : & \{G_1\} \\
del & : & \{\}
\end{array}
\qquad
\begin{array}{lll}
\mathbf{A}_i^1, & \multicolumn{2}{l}{i = 2, \cdots, n} \\
pre & : & \{F^*, G_{i-1}\} \\
add & : & \{G_i\} \\
del & : & \{\}
\end{array}
\qquad
\begin{array}{lll}
\mathbf{A}_i^2, & \multicolumn{2}{l}{i = 1, \cdots, n} \\
pre & : & \{\} \\
add & : & \{F_i\} \\
del & : & \{F^*\}.
\end{array}
$$

The initial state is $\{F^*\}$ and the goal state is $\{G_i | k < i \leq n\}$. Only $\mathbf{A}_i^1$ actions should be applied to reach the goal. Applying an $\mathbf{A}_i^2$ action in any state leads to a wild path since F^* is deleted. The states on wild paths contain F_i facts. Since any subset of F_i facts is possible, the number of states on wild paths grows exponentially with n. The only solution is $\mathbf{A}_1^1, \cdots, \mathbf{A}_n^1$ which is non-trivial to find, since the heuristic gives no information to guide the search on the first k steps. Each action is assumed to have unit cost.

In this experiment, we only compare the total CPU time and number of iterations of ghSetA* and single-state A*. The FG^k problems are defined in *NADL*. A specialized poly-time OBDD operation for splitting *NADL* actions into transitions with the same search information change is used for ghSetA*. No upper bound ($u = \infty$) is used by ghSetA* and no upper limit of the branching partitions is applied. For the FG^k problems considered, n equals 16. This corresponds to a domain with 2^{33} states. The parameters of the OBDD package are hand tuned in each experiment for best performance. Time out is 600 seconds. The results are shown in Figure 14. The performance of A* degrades quickly with the number of unguided steps. A* gets lost expanding an exponentially growing set of states on wild paths. The ghSetA* algorithm is hardly affected by the lack of guidance. The reason is that ghSetA* performs a regular OBDD-based blind forward search on the unguided part where the frontier states can be represented by a symmetric function with polynomial OBDD size. Thus the performance difference between A* and ghSetA* grows exponentially.

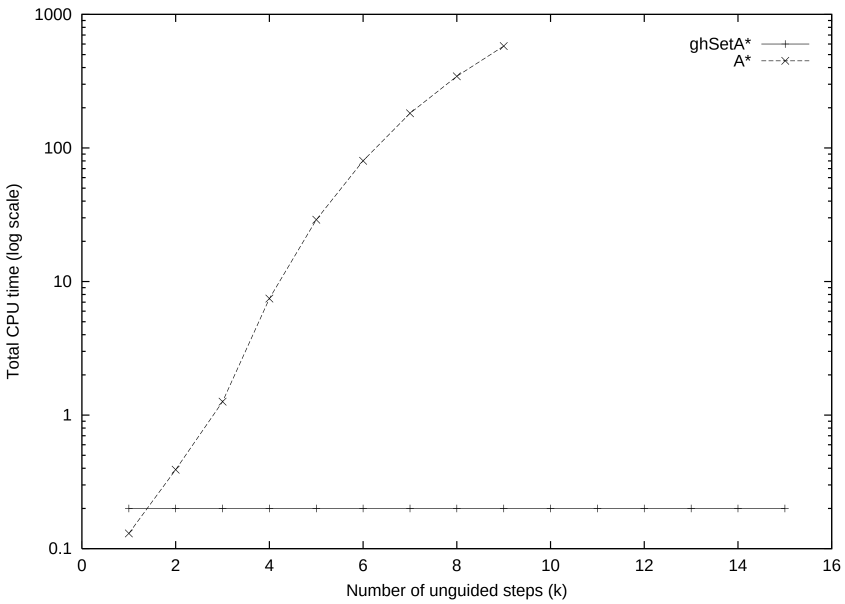

Figure 14: Total CPU time of the FG^k problems.

7.2 $D^xV^yM^z$

This problem has the minimum Hamming distance as an admissible heuristic. The domain consists of a set sliders that can be moved between the corner positions of hypercubes. In any state, a corner position can be occupied by at most one slider. The dimension of the hypercubes is y. There are z sliders of which x are moving on the same cube. The remaining $z-x$ sliders are moving on individual cubes. The sliders are numbered. Initially, they are given corner positions that, when encoded in binary, correspond to an ascending order of their numbers. The goal is to change their positions to a descending order. Each action is assumed to have unit cost. Figure 15 shows the initial state of $D^5V^3M^7$.

When $x = z$ all sliders are moving on the same cube. If further $x = 2^y - 1$ all corners of the cube except one will be occupied making it a permutation problem similar to the 8-Puzzle. The key idea about this problem is that the x parameter allows the dependency of sliders to be adjusted linearly without changing the size of the domain. For the OBDD-based algorithms, the $D^xV^4M^{15}$ problems are defined in *NADL*. Again a specialized poly-

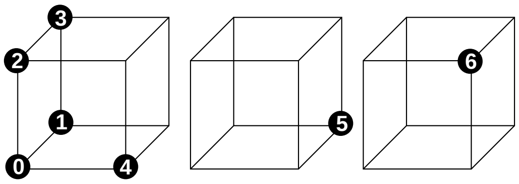

Figure 15: The initial state of $D^5V^3M^7$.

time OBDD operation for splitting *NADL* actions into transitions with the same search information change is applied by ghSetA* and fSetA*. For all problems, the number of states is 2^{60}. For ghSetA* the upper bound for node merging is 200 ($u = 200$). All OBDD-based algorithms except BDDA* utilize a disjunctive partitioning with an upper bound on the OBDDs representing a partition of 5000. Time out is 500 seconds. For all problems, the OBDD-based algorithms use 2.3 seconds on initializing the OBDD package ($n = 8000000$ and $c = 700000$). The results are shown in Table 1. Figure 16 shows a graph of the total CPU time for the algorithms.

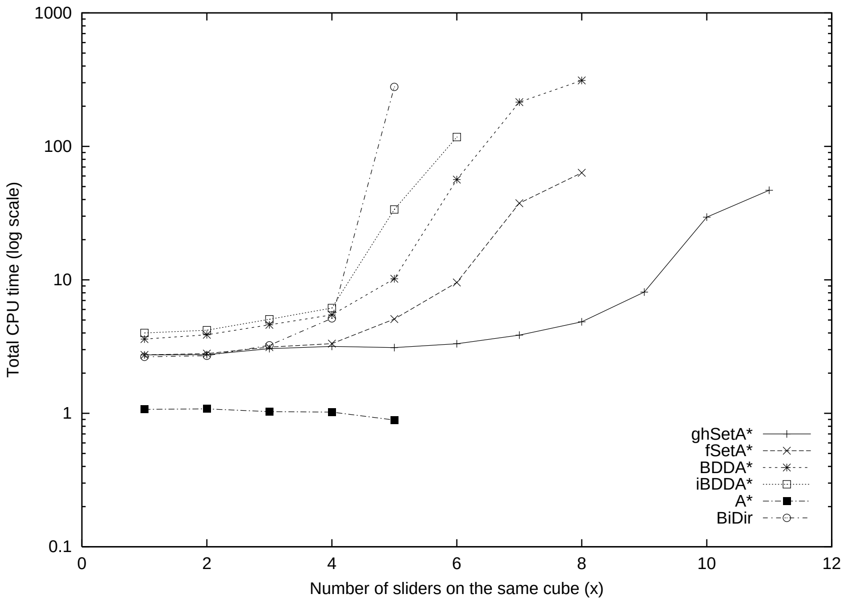

Figure 16: Total CPU time of the $D^xV^4M^{15}$ problems.

All solutions found are 34 steps long. For BDDA* and iBDDA* the size of the OBDD representing the heuristic function is 2014 and 1235, respectively. Both the size of the monolithic and partitioned transition relation grows fast with the dependency between sliders. The problem is that there is no efficient way to model whether a position is occupied or not. The most efficient algorithm is ghSetA*. The fSetA* algorithm has worse performance than ghSetA* because it has to expand all states with minimum f-value in each iteration, whereas ghSetA* focus on a subset of them by having $u = 200$. A subexperiment shows that ghSetA* has similar performance as fSetA* when setting $u = \infty$. The impact of the u parameter is significant for this problem since, even for fairly large values of x, it has an abundance of optimal solutions. BDDA* has much worse performance than fSetA* even though it expands the exact same set of states in each iteration. As we show in Section 7.8, the problem is that the complexity of the computation of $open''$ grows fast with the size of the OBDD representing the states to expand. Surprisingly the performance of iBDDA* is worse than BDDA*. This is unusual, as the remaining experiments will show. The reason might be that only a little space is saved by partitioning the transition relation in this domain. This may cause the computation of $open''$ for iBDDA* to deteriorate because it must iterate through all the partitions. A* performs well when $f(n)$ is a perfect or near perfect discriminator, but it soon gets lost in keeping track of the fast growing number of states on optimal paths. It times out in a single step going from about one second to more than 500 seconds. The problem for Bidir is the usual for blind OBDD-based search algorithms applied to hard combinatorial problems: the OBDDs representing the search frontiers blow up.

7.3 The 24 and 35-Puzzle

We now turn to investigating the $(n^2 - 1)$-puzzles. The domain consists of an $n \times n$ board with $n^2 - 1$ numbered tiles and a blank space. A slide adjacant to the blank space can slide into the space. The goal is to reach a configuration where the tiles are ordered ascendingly as shown for the 24-Puzzle in Figure 17. For our experiments, the initial state is generated by performing r random moves from the goal state.[3] We assume unit cost transitions and use the usual sum of Manhattan distances of the tiles to their

[3] In each of these steps choosing the move back to the previous state is illegal.

Algorithm	x	T_{total}	T_{rel}	T_{search}	$\|expand\|$	$\|Q\|_{max}$	$\|T\|$	it
ghSetA*	1	2.7	0.3	0.2	307.3	33	710	34
	2	2.8	0.3	0.2	307.3	33	1472	34
	3	3.1	0.4	0.3	671.0	33	4070	34
	4	3.2	0.5	0.4	441.7	72	10292	34
	5	3.1	0.4	0.4	194.8	120	20974	34
	6	3.3	0.6	0.4	139.9	212	45978	34
	7	3.9	1.0	0.5	128.4	322	104358	34
	8	4.9	1.9	0.6	115.9	438	232278	34
	9	8.1	5.0	0.8	132.0	557	705956	34
	10	29.5	14.3	12.8	146.1	5103	1970406	373
	11	46.9	43.8	0.8	107.3	336	5537402	34
	12	*Mem*						
fSetA*	1	2.7	0.3	0.2	307.3	1	710	34
	2	2.8	0.3	0.2	307.3	1	1472	34
	3	3.1	0.4	0.4	671.0	1	4070	34
	4	3.3	0.4	0.6	671.0	1	10292	34
	5	5.1	0.5	2.3	1778.6	1	20974	34
	6	9.6	0.6	6.6	2976.5	1	45978	34
	7	37.5	1.0	34.2	9046.7	1	104358	34
	8	63.4	2.0	59.1	9046.7	1	232278	34
	9	408.3	4.9	401.1	24175.4	1	705956	34
	10	*Time*						
BDDA*	1	3.6	0.5	0.4	314.3		355	34
	2	3.9	0.5	0.6	314.3		772	34
	3	4.6	0.6	1.3	678.0		2128	34
	4	5.5	0.8	2.0	678.0		6484	34
	5	10.2	1.3	6.2	1785.6		20050	34
	6	56.4	3.4	50.4	2983.5		64959	34
	7	214.8	10.8	201.1	9053.7		234757	34
	8	312.1	52.7	256.1	9053.7		998346	34
	9	*Time*						
iBDDA*	1	4.0	0.4	0.8	307.3		355	34
	2	4.2	0.4	1.1	307.3		772	34
	3	5.1	0.5	1.9	671.0		2128	34
	4	6.2	0.4	3.0	671.0		6791	34
	5	33.7	0.4	30.4	1778.6		25298	34
	6	117.6	0.5	113.9	2976.5		84559	34
	7	*Time*						
A*	1	1.1				1884		34
	2	1.1				1882		34
	3	1.0				1770		34
	4	1.0				1750		34
	5	0.9				1626		34
	6	*Time*						
Bidir	1	2.7	0.2	0.1	568.5		355	34
	2	2.7	0.2	0.2	630.8		772	34
	3	3.2	0.3	0.7	2305.1		2128	34
	4	5.2	0.2	2.6	3131.1		5159	34
	5	278.9	0.2	276.4	30445.0		10610	34
	6	*Time*						

Table 1: Results of the $D^xV^4M^{15}$ problems.

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20
21	22	23	24	

Figure 17: Goal state of the 24-Puzzle.

goal position as heuristic function. This heuristic function is admissible. For ghSetA* and fSetA* a disjunctive branching partitioning is easy to compute since δh of an action changing the position of a single tile is independent of the position of the other tiles. The two algorithms have no upper bound on the size of OBDDs in the frontier nodes ($u = \infty$). For the OBDD-based algorithms, the problems are defined in *NADL* and the best results are obtained when having no limit on the partition size. Thus, BDDA*, iBDDA*, and Bidir use a monolithic transition relation. The number of states for the 24-puzzle is 2^{125}. The results of this problem are shown in Table 2. For all 24-puzzle problems, the OBDD-based algorithms spend 3.6 seconds on initializing the OBDD package ($n = 15000000$ and $c = 500000$). Time out is 10000 seconds. For BDDA* and iBDDA* the size of the OBDD representing the heuristic function is 33522 and 18424, respectively. For ghSetA* and fSetA* the size of the transition relations is 70582, while the size of the transition relation for BDDA* and iBDDA* is 66673. Thus a small amount of space was saved by using a monolithic transition relation representation. However, ghSetA* and fSetA* has better performance than BDDA* and iBDDA* mostly due to the their more efficient node expansion computation. Interestingly, both BDDA* and iBDDA* spend significant time computing the heuristic funtion in this domain. The ghSetA* and fSetA* also scale better than A* and Bidir. A* has good performance because it does not have the substantial overhead of computing the transition relation and finding actions to apply. However, due to the explicit representation of

Algorithm	r	T_{total}	T_{rel}	T_{search}	$\|sol\|$	$\|expand\|$	$\|Q\|_{max}$	it
ghSetA*	140	28.8	22.1	2.7	26	187.5	23	93
	160	30.0	22.2	3.8	28	213.2	24	175
	180	31.4	22.2	5.3	32	270.2	28	253
	200	43.7	21.9	14.9	36	786.2	31	575
	220	36.3	22.2	10.1	36	411.1	31	490
	240	199.3	22.0	173.2	50	2055.5	44	1543
	260	5673.7	23.9	5644.5	56	10641.2	48	2576
	280	*Mem*						
	300	4772.7	20.9	4743.97	60	9761.3	53	2705
	320	*Mem*						
fSetA*	140	29.7	21.0	4.7	26	669.9	1	42
	160	32.2	20.9	7.4	28	1051.6	1	57
	180	34.3	21.0	9.5	32	1207.0	1	69
	200	50.1	21.0	25.3	36	5276.0	1	93
	220	41.8	21.0	17.0	36	3117.6	1	88
	240	205.2	21.0	180.5	50	18243.3	1	156
	260	*Mem*						
BDDA*	140	98.5	83.0	11.3	26	676.9		42
	160	114.7	83.2	27.4	28	1058.6		57
	180	129.8	82.9	42.7	32	1214.0		69
	200	425.0	83.1	337.1	36	5283.0		93
	220	267.7	82.8	180.6	36	3124.6		88
	240	4120.1	83.1	4032.8	50	18250.3		156
	260	*Time*						
iBDDA*	140	79.8	66.7	5.9	26	669.9		42
	160	85.3	65.7	11.8	28	1051.6		57
	180	93.6	65.7	20.0	32	1207.0		69
	200	314.6	65.8	240.9	36	5276.0		93
	220	156.9	65.6	83.5	36	3117.6		88
	240	2150.3	65.9	2076.6	50	18243.3		156
	260	*Mem*						
A*	140	0.1			26		300	221
	160	0.9			28		725	546
	180	0.6			32		1470	1106
	200	7.4			36		15927	12539
	220	2.3			36		5228	4147
	240	87.1			50		159231	133418
	260	*Mem*						
Bidir	140	68.1	36.6	27.9	26	34365.2		26
	160	96.0	36.8	55.6	28	55388.4		28
	180	214.7	36.8	174.3	32	106166.0		32
	200	1286.0	36.8	1245.6	36	359488.0		36
	220	3168.8	36.8	3128.4	36	421307.0		36
	240	*Mem*						

Table 2: Results of the 24-puzzle problems.

states, it runs out of memory for solution depths above 50. For Bidir, the problem is the usual: the OBDDs representing the search frontiers blow up. Figure 18 shows a graph of the total CPU time of the 24 and 35-puzzle. Again time out is 10000 seconds.

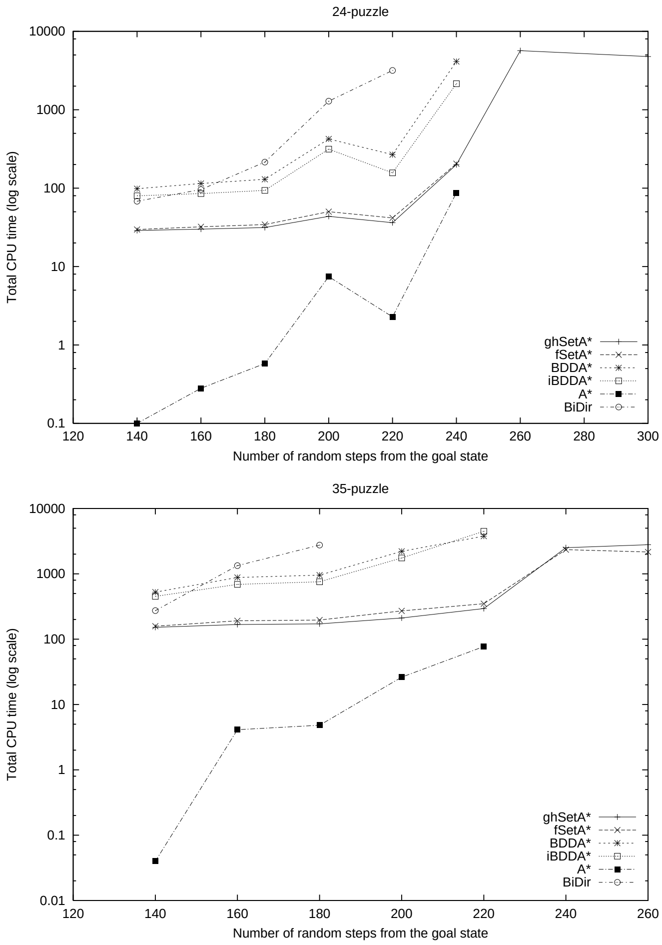

Figure 18: Total CPU time for the 24 and 35-puzzle problems.

7.4 Planning Problems

In this section, we consider four planning problems from the STRIPS track of the AIPS 1998 [32], 2000 [2], and 2002 [33] planning competition. The problems are defined in the STRIPS fraction of PDDL. An optimal solution is a solution with minimal length, so we assume unit cost actions. A Boolean representation of a STRIPS domain is trivial if using a single Boolean state variable for each fact. This encoding, however, is normally very inefficient due to its redundant representation of static facts and facts that are mutual exclusive or unreachable. In order to generate a more compact encoding, we analyse the STRIPS problem in a three step process.

1. Find static facts by subtracting the facts mentioned in the add and delete sets of actions from the facts in the initial state.

2. Approximate the set of reachable facts from the initial state by performing a relaxed reachability analysis, ignoring the delete set of the actions.

3. Find sets of *single-valued predicates* [20] via inductive proofs on the reachable facts.

If a set of predicates are mutual exclusive when restricting a particular argument in each of them to the same object then the set of predicates is said to be single-valued. Consider for instance a domain where packages can be either inside a trucks $in(P, T)$ or at locations $at(P, L)$. Then in and at at are single-valued with respect to the first argument. The reachability analysis in step 2 is based an approach described in [16]. It is fast for the problems considered in this paper (for most problems less than 0.04 seconds). The algorithm proceeds in a breadth-first manner such that each fact f can be assigned a depth $d(f)$ where it is reached. Similar to the MIPS planning system [16], we use this measure to approximate the HSPr heuristic [7]. HSPr is an efficient but non-admissible heuristic for backward search. For a state given by a set of facts S, the approximation to HSPr is given by

$$h(S) = \sum_{f \in S} d(f)$$

A branching partitioning for this heurisitic is efficient to generate given that each action (pre, add, del) leading from S to $S' = (S \cup add) \setminus del$ satisfies

$$del \subseteq pre \quad \text{and} \quad add \cap pre = \emptyset.$$

These requirements are natural and satisfied by all the planning domains considered in this paper. Due to the constraints, we get

$$\begin{aligned}\delta h &= h(S') - h(S) \\ &= h(add \setminus S) - h(del) \\ &= \sum_{f \in add \setminus S} d(f) - \sum_{f \in del} d(f).\end{aligned}$$

Thus, each action is partitioned in up to $2^{|add|}$ sets of transitions with different δh-value. In order to simplify the computation of the initial heuristic value, all problems have been modified to a single goal state. Furthermore, in domains where the HSPr approximation either systematically under or over estimates the true remaining cost, we have scaled it accordingly.

7.4.1 Blocks World

The Blocks World is a classical planning domain. It consists of a set of cubic blocks sitting on a table. A robot arm can stack and unstack blocks from some initial configuration to a goal configuration. The problems, we consider, are from the STRIPS track of the AIPS 2000 planning competition. The number of states grows from 2^{17} to 2^{80}. The HSPr heuristic is scaled by a factor of 0.4. The gsSetA* and fSetA* algorithms have no upper bound on the size of OBDDs of the nodes on the frontier ($u = \infty$). For all OBDD-based algorithms, the partition limit was 5000. For each problem, these algorithms spend about 2.5 seconds on initializing the OBDD package ($n = 8000000$ and $c = 800000$). Time out is 500 seconds in all experiments. The results are shown in Table 3. Figure 19 shows the total CPU time of the algorithms. For BDDA* and iBDDA* the size of the OBDD representing the heuristic function is in the range of $[8, 1908]$ and $[8, 1000]$, respectively. The ghSetA* and fSetA* algorithms have significantly better performance than all other algorithms. As usual BDDA* and iBDDA* suffer from an inefficient expansion computation while the frontier OBDDs blow up for Bidir. The general A* algorithm for STRIPS planning problems is less domain-tuned than the previous A* implementations. In particular, it must check the precondition of all actions in each iteration in order to find the ones that are applicable. This, in addition to the explicit state representation, may explain the poor performace of A*.

Algorithm	p	T_{total}	T_{rel}	T_{search}	$\|sol\|$	$\|expand\|$	$\|Q\|_{max}$	it	$\|T\|$
ghSetA*	4	2.6	0.0	0.0	6	19.5	1	6	706
	5	2.7	0.1	0.1	12	33.4	11	31	1346
	6	2.6	0.1	0.1	12	57.7	9	30	2608
	7	3.1	0.2	0.4	20	53.8	48	152	4685
	8	4.1	0.3	1.3	18	540.4	12	72	7475
	9	17.0	0.4	14.1	32	331.8	94	991	8717
	10	116.2	0.6	113.1	38	744.9	111	2309	11392
	11	133.5	0.7	130.2	32	1404.9	91	1200	16122
	12	14.8	1.0	11.2	34	410.3	120	557	18734
	13	*Time*							
	14	112.1	1.7	107.8	38	1067.8	125	1061	30707
	15	*Time*							
fSetA*	4	2.5	0.0	0.0	6	29.8	1	6	706
	5	2.7	0.1	0.1	12	68.7	4	23	1346
	6	2.7	0.1	0.1	12	126.8	2	20	2608
	7	3.2	0.2	0.5	20	121.9	8	92	4685
	8	3.9	0.3	1.1	18	1328.8	2	35	7475
	9	30.0	0.4	27.1	32	935.5	10	610	8717
	10	217.0	0.6	213.8	38	2594.4	12	1098	11392
	11	259.8	0.8	256.4	32	4756.0	9	671	16122
	12	39.2	1.0	35.7	34	817.0	13	860	18734
	13	*Time*							
	14	274.3	1.7	270.0	38	1555.1	13	1462	30707
	13	*Time*							
BDDA*	4	3.3	0.0	0.1	6	37.8		6	706
	5	3.6	0.2	0.2	12	76.7		23	1365
	6	3.6	0.2	0.2	12	134.8		20	2334
	7	4.9	0.5	1.2	20	129.9		92	4669
	8	6.0	0.5	2.2	18	1336.8		35	6959
	9	100.8	1.1	96.5	32	943.5		610	9923
	10	*Time*							
iBDDA*	4	2.7	0.0	0.0	6	29.8		6	706
	5	2.8	0.1	0.1	12	68.7		23	1365
	6	2.9	0.1	0.1	12	126.8		20	2334
	7	3.7	0.3	0.7	20	121.9		92	4669
	8	6.2	0.4	3.2	18	1328.8		35	7123
	9	113.7	0.6	110.3	32	935.5		610	10361
	10	*Time*							
A*	4	0.0		0.0	6		8	15	
	5	0.2		0.2	12		62	70	
	6	0.4		0.4	12		115	102	
	7	1.3		1.2	20		287	287	
	8	31.9		31.6	18		7787	5252	
	9	233.9		232.9	32		38221	31831	
	10	*Time*							
Bidir	4	2.6	0.0	0.0	6	124.5		6	706
	5	2.6	0.1	0.0	12	228.3		12	1423
	6	2.7	0.1	0.1	12	438.8		12	2567
	7	3.6	0.2	0.8	20	1931.3		20	5263
	8	9.7	0.3	6.8	18	11181.8		18	8157
	9	146.8	0.4	143.9	30	75040.9		30	11443
	10	*Time*							

Table 3: Results of the Blocks World problems.

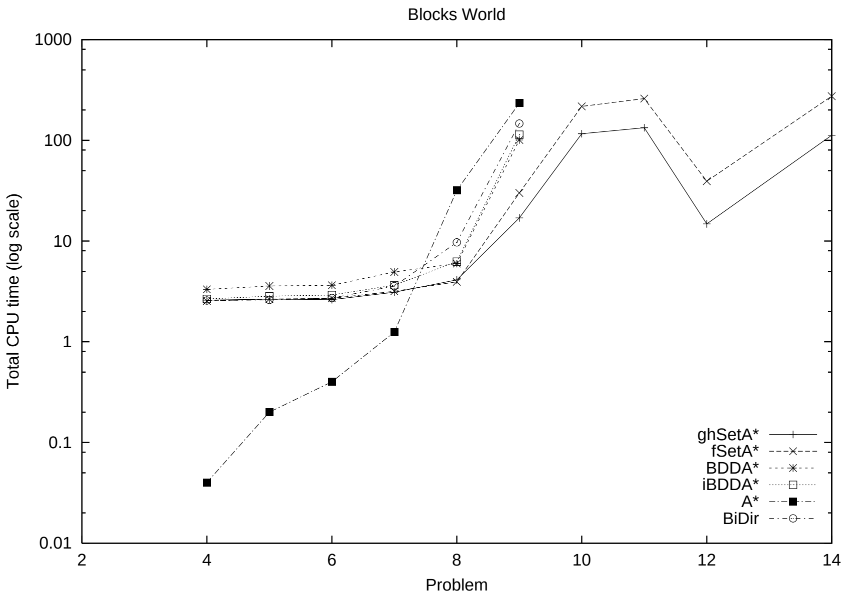

Figure 19: Total CPU time for the Blocks World problems.

7.4.2 Gripper

The Gripper problems are from the first round of the STRIPS track of the AIPS 1998 planning competition. The domain consists of two rooms, A and B, connected with a door and robot with two grippers. Initially, a number of balls are located in room A, and the goal is to move them to room B. The number of states grows liniearly from 2^{12} to 2^{88}. The ghSetA* and fSetA* algorithms have no upper bound on the size of OBDDs in the frontier nodes ($u = \infty$). For all OBDD-based algorithms no partition limit is used, and they spend about 0.8 seconds on initializing the OBDD package ($n = 2000000$ and $c = 400000$). All algorithms generate optimal solutions. The results are shown in Table 4. Figure 20 shows the total CPU time of the algorithms. Interestingly Bidir is the fastes algorithm in this domain since the OBDDs representing the search frontier only grows moderately during the search. The ghSetA* and fSetA* algorithms, however, have almost as good performance. BDDA* and iBDDA* has particularly bad performance in this domain. The problem is that the OBDDs of frontier nodes grow quite large for the harder problems.

Algorithm	p	T_{total}	T_{rel}	T_{search}	$\|expand\|$	$\|Q\|_{max}$	it	$\|T\|$
ghSetA*	2	0.9	0.1	0.02	68.8	5	21	594
	4	1.0	0.1	0.08	168.9	6	43	1002
	6	1.3	0.2	0.27	314.9	6	65	1410
	8	1.5	0.3	0.34	504.8	6	87	1818
	10	1.8	0.4	0.54	738.1	6	109	2226
	12	2.3	0.5	0.88	1014.7	6	131	2634
	14	3.0	0.7	1.33	1334.5	6	153	3042
	16	3.6	0.9	1.78	1697.5	6	175	3450
	18	4.5	1.1	2.46	2103.7	6	197	3858
	20	5.7	1.4	3.37	2553.1	6	219	4266
fSetA*	2	1.0	0.1	0.1	95.4	1	17	594
	4	1.0	0.1	0.1	231.2	1	29	1002
	6	1.2	0.2	0.2	423.9	1	41	1410
	8	1.6	0.3	0.3	673.4	1	53	1818
	10	2.0	0.4	0.6	979.9	1	65	2226
	12	2.5	0.6	1.0	1343.3	1	77	2634
	14	3.1	0.8	1.4	1763.5	1	89	3042
	16	3.7	0.9	1.9	2240.7	1	101	3450
	18	5.0	1.2	2.9	2774.7	1	113	3858
	20	5.7	1.5	3.2	3365.6	1	125	4266
BDDA*	2	1.8	0.1	0.2	103.4		17	323
	4	2.4	0.2	0.6	239.2		29	539
	6	3.4	0.3	1.5	431.9		41	755
	8	6.1	0.6	4.0	681.4		53	971
	10	16.9	0.9	14.4	987.9		65	1187
	12	40.7	1.2	37.9	1351.3		77	1403
	14	81.7	1.6	78.5	1771.5		89	1619
	16	149.3	2.2	145.4	2248.7		101	1835
	18	240.4	3.1	235.5	2782.7		113	2051
	20	391.1	3.9	385.5	3373.6		125	2267
iBDDA*	2	1.2	0.1	0.1	95.4		17	323
	4	1.6	0.1	0.4	231.2		29	539
	6	2.3	0.3	1.0	423.9		41	755
	8	3.6	0.4	2.2	673.4		53	971
	10	6.2	0.6	4.5	979.9		65	1187
	12	12.2	0.9	9.2	1343.3		77	1403
	14	23.5	1.1	21.3	1763.5		89	1619
	16	44.8	1.6	42.1	2240.7		101	1835
	18	76.1	2.2	72.4	2774.7		113	2051
	20	120.9	2.7	116.7	3365.6		125	2267
A*	2	3.9		3.9		698	1286	
	4	422.9		422.3		26434	85468	
	6	*Time*						
Bidir*	2	0.9	0.1	0.0	125.4		17	323
	4	1.0	0.1	0.1	290.9		29	539
	6	1.2	0.2	0.1	589.7		41	755
	8	1.4	0.3	0.3	958.2		53	971
	10	1.7	0.4	0.5	1404.3		65	1187
	12	2.2	0.5	0.8	1611.0		77	1403
	14	2.6	0.7	1.0	2025.6		89	1619
	16	3.2	0.9	1.3	3265.6		101	1835
	18	3.8	1.2	1.7	4074.4		113	2051
	20	4.5	1.5	2.1	4944.9		125	2267

Table 4: Results of the Gripper problems.

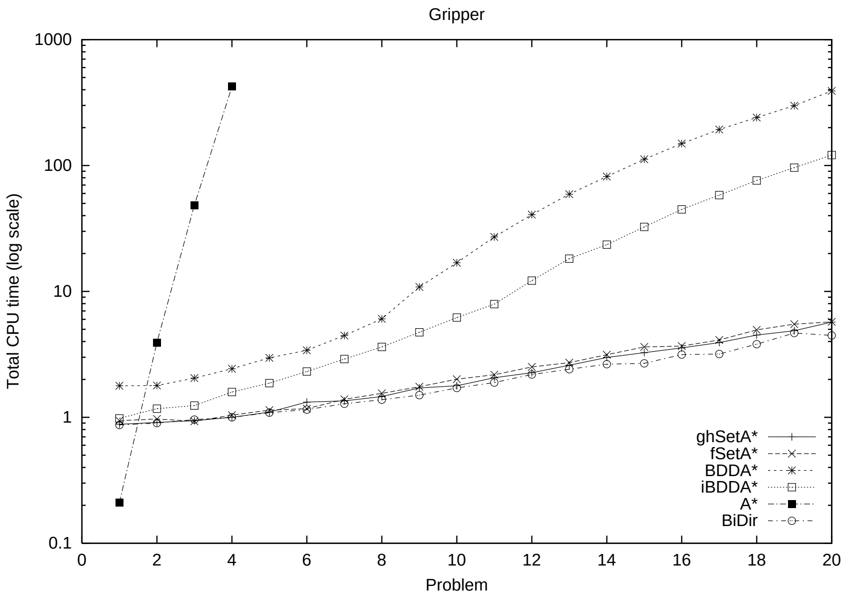

Figure 20: Total CPU time for the Gripper problems.

7.4.3 Logistics

The logistics domain considers moving packages with trucks between subcities and with airplanes between cities. The problems considered are from the STRIPS track of the AIPS 2000 planning competition. The number of states grows from 2^{21} to 2^{86}. The ghSetA* and fSetA* algorithms have no upper bound on the size of OBDDs in the frontier nodes ($u = \infty$). For all OBDD-based algorithms a partition limit of 5000 is used and they spend about 2.0 seconds on initializing the OBDD package ($n = 8000000$ and $c = 400000$). Due to systematic under estimation, the HSPr heursitic is scaled with a factor of 1.5. Figure 21 shows the total CPU time of the algorithms.

7.4.4 ZenoTravel

ZenoTravel is from the STRIPS track of the AIPS 2002 planning competition. It involves transporting people around in planes, using different modes of movement: fast and slow. The number of states grows from 2^{9} to 2^{165}. The ghSetA* and fSetA* algorithms have no upper bound on the size of OBDDs in the frontier nodes ($u = \infty$). For all OBDD-based algorithms a partition

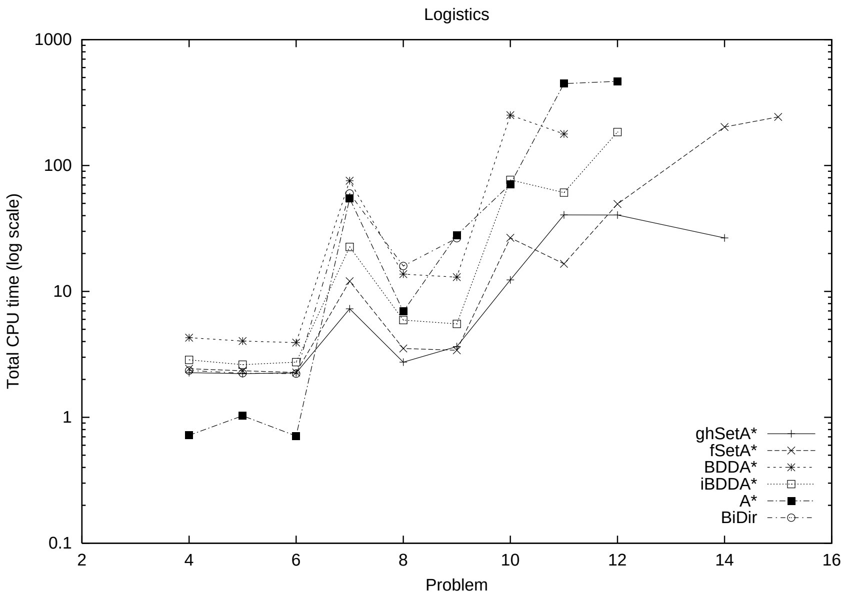

Figure 21: Total CPU time for the Logistics problems.

limit of 4000 is used. About 2.7 seconds is spent on initializing the OBDD package (n = 10000000 and c = 700000). Figure 22 shows the total CPU time of the algorithms. The results are very similar to the results of the logisitcs problems.

7.5 Channel Routing

Channel routing is a fundamental subtask in the layout process of VLSI-design. It is an NP-complete problem which makes exact solutions hard to produce. Channel routing considers connecting pins in the small gaps or channels between the cells of a chip. In its classical formulation two layers are used for the wires: one where wires go horizontal (tracks) and one where wires go vertical (columns). In order to change direction, a connection must be made between the two layers. These connections are called vias. Pins are at the top and bottom of the channel. A set of pins that must be connected is called a net. The problem is to connect the pins optimally according to some cost function. The cost function studied here equals the total number of vias used in the routing. Figure 23 shows an example of an optimal

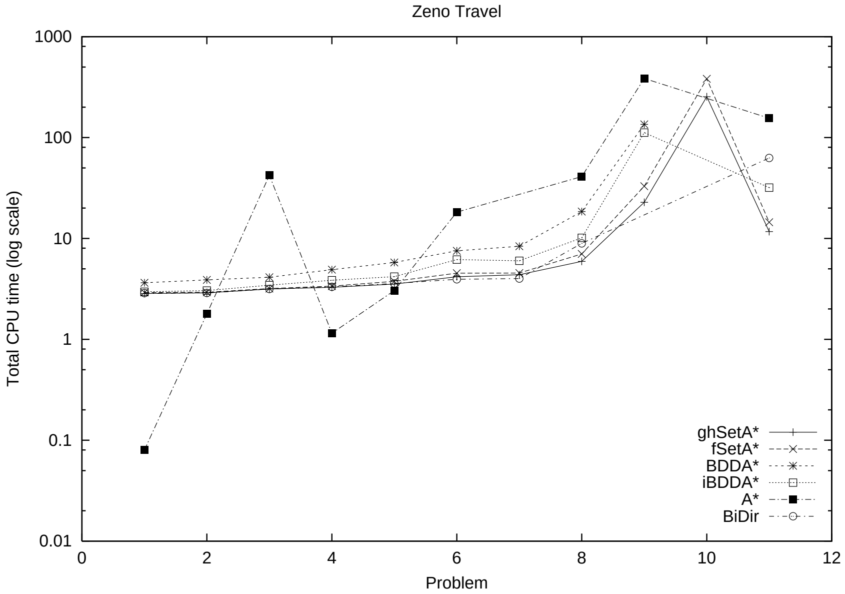

Figure 22: Total CPU time for the ZenoTravel problems. Problem 10 of ZenoTravel can only be solved by ghSetA* and fSetA*.

solution to a small channel routing problem. The cost of the solution is 4. One way to apply search to solve a channel routing problem is to route the nets from left to right. A state in this search is a column paired with a routing of the nets on the left side of that column. A transition of the search is a routing of live nets over a single column. A* can be used in the usual way to find optimal solutions. An admissible heuristic function for our cost function is the sum of the cost of routing all remaining nets optimally ignoring interactions with other nets. We have implemented a specialized search engine to solve channel routing problems with ghSetA* [26]. The important point about this application is that ghSetA* utilizes a conjunctive branching partitioning instead of a disjunctive branching partitioning as in all other experiments reported in this paper. This is possible since a transition can be regarded as the joint result of routing each net in turn.

The performance of ghSetA* is evaluated using problems produced from two ISCAS-85 circuits [46]. For each of these problems the parameters of the OBDD package are hand tuned for best performance. There is no upper bound on the size of OBDDs in frontier nodes ($u = \infty$) and no limit on the

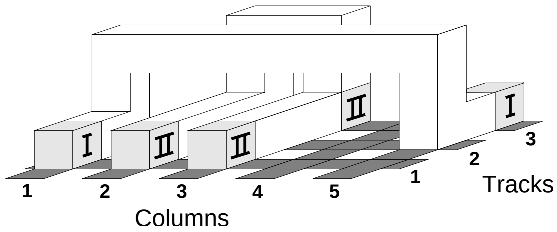

Figure 23: A solution to a channel routing problem with 5 columns, 3 tracks, and 2 nets (labeled I and II). The pins are numbered according to what net they belong.

size of the partitions. Time out is 600 seconds. Table 5 shows the results. The performance of ghSetA* is similar to previous applications of OBDDs

Circuit	$c-t-n$	T_{total}	T_{rel}	T_{search}	$\lvert Q\rvert_{max}$	it
Add	38-3-10	0.2	0.1	0.2	1	40
	47-5-27	0.8	0.7	0.1	24	46
	41-3-12	0.2	0.1	0.1	1	42
	46-7-20	5.0	3.5	1.5	56	89
	25-4-6	0.1	0.0	0.1	1	30
C432	83-4-33	0.4	0.2	0.2	0	93
	89-11-58	*Mem*				
	101-9-57	286.1	61.5	206.6	135	113
	99-8-58	34.0	13.5	20.5	59	448
	97-10-63	295.0	99.7	195.3	129	109
	101-7-53	15.7	11.5	4.2	90	101
	95-9-48	223.8	58.9	164.9	59	399
	95-10-48	*Time*				
	84-5-23	3.2	0.7	2.5	0	92

Table 5: Results of the ISCAS-85 channel routing problems. A problem, $c-t-n$, is identified by its number of columns (c), tracks (t), and nets (n).

to channel routing [43, 46]. However, in contrast to previous approaches, ghSetA* is able to find optimal solutions.

7.6 State-Set Branching versus Single-State Heuristic Search

Heuristic search is trivial if the heuristic function is very informative. In this case, state-set branching may have worse performance than single-state heuristic search due to the overhead of computing the transition relation. Thus, we do not expect state-set branching algorithms to have better performance than the single-state heuristic search algorithms applied in the AIPS planning competitions because the problems considered have very strong heuristics [23]. In this paper, we consider finding optimal or near optimal solutions with state-set branching implementations of A*. The studied heuristic functions are classical but leaves a significant search element for the algorithms to handle. For these problems, state-set branching outperforms single-state A*. Notice that this result is consistent with the fact that single-state A* is optimally efficient. The reason is that a state-set branching implementation of A* may use an exponentially more compact state representation than single-state A*.

7.7 State-Set Branching versus Blind OBDD-based Search

Blind OBDD-based search has been succesfully applied in symbolic model checking and circuit verification. It has been shown that many problems encountered in practice are tactable when using OBDDs [47]. The classical search problems studied in AI, however, seems to be harder and have longer solutions than the problems considered in formal verification. When applying blind OBDD-based search to these problems, the OBDDs used to represent the search frontier often grow exponentially. The experiemtal evaluation of state-set branching shows that this problem can be substantially reduced when efficiently splitting the search frontier according to a heuristic evaluation of the states.

7.8 State-Set Branching versus BDDA*

State-set branching implementations of A* such as ghSetA* and fSetA* are fundamentally different from BDDA*. BDDA* does not exploit a partitioning of the transitions according to how they change the g and h-value. Instead, it imitates the usual explicit application of the heuristic function via a symbolic computation. It would be reasonable to expect that the symbolic representation of practical heuristic functions often is very large. However, this is seldom the case for the heuristic functions studied in this paper. The major challenge for BDDA* is that the arithmetic computations at the OBDD level scales poorly with the size of the OBDD representing the set of states to expand (line 5 and 6 in Figure 13). This hypothesis can be emperically verified by measuring the CPU time used by fSetA* and iBDDA* to expand a set of states. Recall that both fSetA* and iBDDA* expand the exact same set of states in each iteration. Any performance difference is therefore solely caused by their expansion techniques. The results are shown in Figure 24. The reported CPU time is the average of the 15-Puzzle with 50, 100, and 200 random steps, Logistics problem 4 to 9, Blocks World problem 4 to 9, Gripper problem 1 to 20, and DxV4M15 with x varying from 1 to 6. For

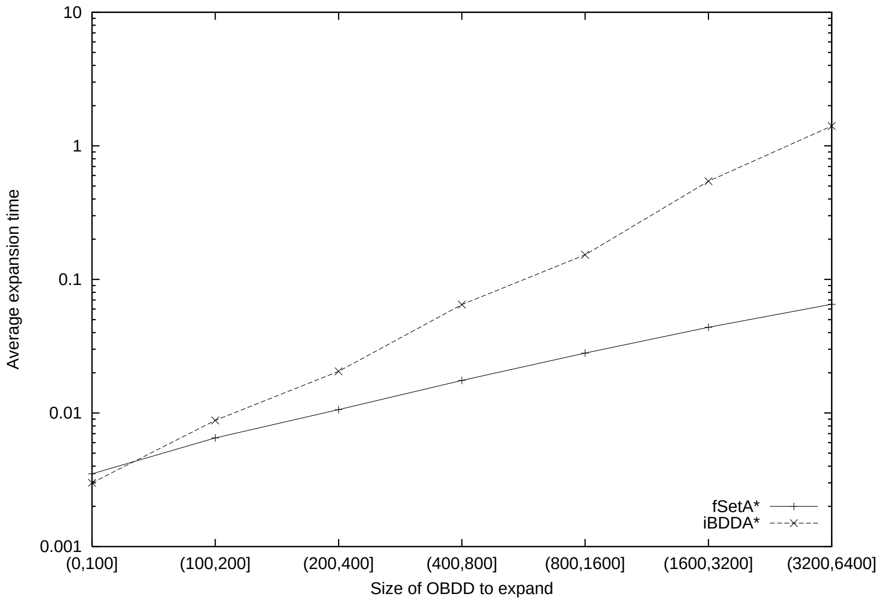

Figure 24: Node expansion times of fSetA* and BDDA*.

very small frontier OBDDs, iBDDA* is slightly faster than fSetA*. This is probably because small frontier OBDDs mainly are generated by easy problems where a monolithic transition relation used by BDDA* is more efficient than the partitioned transition relation used by fSetA*. However, for large frontier OBDDs, BDDA* needs much more expansion time than fSetA*. Another limitation of BDDA* is the inflexibility of OBDD-based arithmetic. It makes it hard to extend BDDA* efficiently to general evaluation functions and arbitrary transitions costs.

8 Conclusion

In this paper, we have introduced a framework called state-set branching for integrating symbolic and heuristic search. The key component of the framework is a new OBDD technique called branching partitioning that allows sets of states to be expanded implicitly and sorted according to search information in one step. State-set branching is a general framework. It applies to any heuristic function, any search node evaluation function, and any transition cost function. An extensive experimental evaluation of state-set branching proves it to be a powerful approach. It consistently outperforms single-state A*, except when the heuristic is very strong. In addition, we show that it can improve the complexity of single-state search exponentially and that, for several of the best-known AI search problems often, it is orders of magnitude faster than single-state heuristic search, blind OBDD-based search, and the previous OBDD-based A* implementation, BDDA*.

An important question, however, is how to compute branching partitions efficiently in general. Obviously, transitions can not be sorted explicitly due to their large number. An implicit approach based on a symbolic encoding of the heuristic function exists [25], but interestingly, for the problems we have examined so far, even this implicit computation can be avoided. In practice, heuristic functions are based on relaxations that decouples the search problem. For instance, for the minimum Hamming distance poly-time OBDD operations exist for sorting the transitions, and for the sum of Manhattan distances and HSPr the transitions can be sorted directly from the problem description.

Further work includes investigating state-set branching on real-world problems. An interesting application is formal verification of finite state machines where state-set branching can be used to find error traces.

Appendix

Lemma 3 *Assume fSetA* and ghSetA* apply an admissible heuristic and* $\pi = s_0, \cdots, s_n$ *is an optimal solution, then at any time before fSetA* and* *ghSetA* terminates there exists a frontier node $\langle S, I\rangle$ with a state $s_i \in S$ such that $I \leq C^*$ and $s_0, \cdots, s_i$ is the search path associated with s_i.*

Proof: A node $\langle S, I\rangle$ containing s_i with associated search path $s_0, \cdots, s_i$ must be on the frontier since a node containing s_0 was initially inserted on the frontier and fSetA* and ghSetA* terminates if a node containing the goal state s_n is removed from the frontier. We have $I = cost(s_0, \cdots, s_i) + h(s_i)$. The path $s_0, \cdots, s_i$ is a prefix of an optimal solution, thus $cost(s_0, \cdots, s_i)$ must be the minimum cost of reaching s_i. Since the heuristic function is admissible, we have $h(s_i) \leq h^*(s_i)$ which gives $I \leq C^*$. □

Theorem 2 *Given an admissible heuristic function fSetA* and ghSetA* are optimal.*

Proof: Suppose fSetA* or ghSetA* terminates with a solution derived from a frontier node with $I > C^*$. Since the node was at the top of the frontier queue, we have

$$I < f(n) \; \forall n \in frontier.$$

Thus, prior to termination, all nodes on the frontier satisfied $f(n) > C^*$. However, this contradicts Lemma 3 that states that any optimal path has a node on the frontier any time prior to termination with $I \leq C^*$. □

acknowledgment

We thank Robert Punkunus for initial work on efficient Boolean encoding of PDDL domains. We also wish to thank Kolja Sulimma for providing channel routing benchmark problems.

References

[1] S. B. Akers. Binary decision diagrams. *IEEE Transactions on Computers*, c-27(6):509–516, 1978.

[2] F. Bacchus. AIPS'00 planning competition : The fifth international conference on artificial intelligence planning and scheduling systems. *AI Magazine*, 22(3):47–56, 2001.

[3] R. Bahar, E. Frohm, C. Gaona, E Hachtel, A Macii, A. Pardo, and F. Somenzi. Algebraic decision diagrams and their applications. In *IEEE/ACM International Conference on CAD*, pages 188–191, 1993.

[4] A. Barret and D. S. Weld. Partial-order planning: Evaluating possible efficiency gains. *Artificial Intelligence*, 67(1):71–112, 1994.

[5] P. Bertoli, A. Cimatti, and M. Roveri. Conditional planning under partial observability as heuristic-symbolic search in belief space. In *Pre-Proceedings of the 6th European Conference on Planning (ECP-01)*, pages 379–384, 2001.

[6] R. Bloem, K. Ravi, and F. Somenzi. Symbolic guided search for CTL model checking. In *Proceedings of the 37th Design Automation Conference (DAC'00)*, pages 29–34. ACM, 2000.

[7] B. Bonet and H. Geffner. Planning as heuristic search: New results. In *Proceedings of the 5th European Conference on Planning (ECP-99)*, pages 360–372. Springer, 1999.

[8] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 8:677–691, 1986.

[9] J.R. Burch, E.M. Clarke, and D.E. Long. Symbolic model checking with partitioned transition relations. In *International Conference on Very Large Scale Integration*, pages 49–58. North-Holland, 1991.

[10] A. Cimatti, E. Giunchiglia, F. Giunchiglia, and P. Traverso. Planning via model checking: A decision procedure for $\mathcal{AR}$. In *Proceedings of the 4th European Conference on Planning (ECP'97)*, pages 130–142. Springer, 1997.

[11] A. Cimatti and M. Roveri. Conformant planning via symbolic model checking. *Journal of Artificial Intelligence Research*, 13:305–338, 2000.

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

[13] A. Cimatti, M. Roveri, and P. Traverso. Strong planning in non-deterministic domains via model checking. In *Proceedings of the 4th International Conference on Artificial Intelligence Planning System (AIPS'98)*, pages 36–43. AAAI Press, 1998.

[14] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[15] R. Dechter and J. Pearl. Generalized best-first search strategies and the optimality of A*. *Association for Computing Machinery*, 32(3):505–536, 1985.

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

[17] S. Edelkamp and F. Reffel. OBDDs in heuristic search. In *Proceedings of the 22nd Annual German Conference on Advances in Artificial Intelligence (KI-98)*, pages 81–92. Springer, 1998.

[18] R. E. Fikes and N. J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.

[19] M. P. Fourman. Propositional planning. In *Proceedings of the AIPS-00 Workshop on Model-Theoretic Approaches to Planning*, pages 10–17, 2000.

[20] A. Gerevini and L. Schubert. Inferring state constraints for domain-independent planning. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI'98)*, pages 905–912, 1998.

[21] E. Hansen, R. Zhou, and Z. Feng. Symbolic heuristic search using decision diagrams. In *Symposium on Abstraction, Reformulation and Approximation SARA'02*, 2002.

[22] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for heuristic determination of minimum path cost. *IEEE Transactions on SSC*, 100(4), 1968.

[23] Jörg Hoffmann. Local search topology in planning benchmarks: An empirical analysis. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI-01)*, pages 453–458. Morgan Kaufmann, 2001.

[24] R. M Jensen. A comparison study between the CUDD and BuDDy OBDD package applied to AI-planning problems. Technical report, Computer Science Department, Carnegie Mellon University, 2002. CMU-CS-02-173.

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

[26] R. M. Jensen, R. E. Bryant, and M. M. Veloso. SetA* applied to channel routing. Technical report, Computer Science Department, Carnegie Mellon University, 2002. CMU-CS-02-172.

[27] R. M. Jensen and M. M. Veloso. OBDD-based deterministic planning using the UMOP planning framework. In *Proceedings of the AIPS-00 Workshop on Model-Theoretic Approaches to Planning*, pages 26–31, 2000.

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

[29] R. M. Jensen, M. M. Veloso, and M. Bowling. Optimistic and strong cyclic adversarial planning. In *Pre-proceedings of the 6th European Conference on Planning (ECP'01)*, pages 265–276, 2001.

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

[31] J. Lind-Nielsen. BuDDy - A Binary Decision Diagram Package. Technical Report IT-TR: 1999-028, Institute of Information Technology, Technical University of Denmark, 1999. `http://cs.it.dtu.dk/buddy`.

[32] D. Long. The AIPS-98 planning competition. *AI Magazine*, 21(2):13–34, 2000.

[33] D. Long and M. Fox. The AIPS-02 planning competition. `http://www.dur.ac.uk/d.p.long/competition.html`, 2002.

[34] D. McDermott, M. Ghallab, A. Howe, C. Knoblock, A. Ram, M. Veloso, D. Weld, and D. Wilkins. PDDL - the planning domain definition language. Technical report, Yale Center fro Computational Vision and Control, 1998.

[35] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publ., 1993.

[36] C. Meinel and T. Theobald. *Algorithms and Data Structures in VLSI Design*. Springer, 1998.

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

[38] M. Pistore, R. Bettin, and P. Traverso. Symbolic techniques for planning with extended goals in non-deterministic domains. In *Pre-Proceedings of the 6th European Conference on Planning (ECP-01)*, pages 253–264, 2001.

[39] I. Pohl. First results on the effect of error in heuristic search. *Machine Intelligence*, 5:127–140, 1970.

[40] R. K. Ranjan, A. Aziz, R. K. Brayton, B. Plessier, and C. Pixley. Efficient BDD algorithms for FSM synthesis and verification. In *IEEE/ACM Proceedings of the International Workshop on Logic Synthesis*, 1995.

[41] F. Reffel and S. Edelkamp. Error detection with directed symbolic model. In *Proceedings of World congress on Formal Methods (FM)*, pages 195–211. Springer, 1999.

[42] S. Russell and P. Norvig. *Artificial Intelligence: a Modern Approach*. Prentice-Hall, 1995.

[43] F. Schmiedle, R. Drechsler, and B. Becker. Exact channel routing using symbolic representation. In *Proceedings of IEEE International Symposium on Circuits and Systems (ISCAS'1999)*, 1999.

[44] F. Somenzi. CUDD: Colorado university decision diagram package. `ftp://vlsi.colorado.edu/pub/`, 1996.

[45] H.-P. Stör. Planning in the fluent calculus using binary decision diagrams. *AI Magazine*, pages 103–105, 2001.

[46] K. Sulimma and K. Wolfgang. An exact algorithm for solving difficult detailed routing problems. In *Proceedings of the 2001 International Symposium on Physical Design*, pages 198–203, 2001.

[47] I. Wegener. *Branching Programs and Binary Decision Diagrams*. Society for Industrial and Applied Mathematics (SIAM), 2000.

[48] C. H. Yang and D. L. Dill. Validation with guided search of the state space. In *Proceedings of the 35th Design Automation Conference (DAC'98)*, pages 599–604. ACM, 1998.

[49] J. Yuan, J. Shen, J. Abraham, and A. Aziz. Formal and informal verification. In *Conference on Computer Aided Verification (CAV'97)*, pages 376–387, 1997.