State-Set Branching: Leveraging BDDs for Heuristic Search ⋆

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

Computer Science Department, Carnegie Mellon University, 5000 Forbes Ave., Pittsburgh, PA 15213-3891, USA

Abstract

In this article, we present a framework called state-set branching that combines symbolic search based on reduced ordered Binary Decision Diagrams (BDDs) with best-first search, such as A* and pure heuristic 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 BDD technique called branching partitioning can be used to efficiently expand sets of states. The framework is general. It applies to any heuristic function, evaluation function, and transition cost function defined over a finite domain. Moreover, branching partitioning applies to both disjunctive and conjunctive transition relation partitioning. An extensive experimental evaluation of the two A* implementations proves state-set branching to be a powerful framework. The algorithms consistently outperform the ordinary A* algorithm. In addition, they can improve the complexity of A* exponentially and often dominate both A* and blind BDD-based search by several orders of magnitude. Moreover, they have substantially better performance than BDDA*, the currently most efficient BDD-based implementation of A*.

Key words: Heuristic Search, BDD-based Search, Boolean Representation

⋆ This work is an extended version of a paper presented at AAAI-02 [27]. The 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.

* Corresponding author.
Email addresses: {runej, mmv, bryant}@cs.cmu.edu.
URLs: www.cs.cmu.edu/~{runej, mmv, bryant}.

1 Introduction

Informed or *heuristic* best-first search (BFS) algorithms [1] such as pure heuristic search and A* [26], 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 and breadth-first search, is that they use heuristics to guide the search toward the goal and in this way significantly reduce the number of visited states. Like depth-first search and breadth-first search, BFS algorithms construct (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 most widely known BFS algorithm. 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 a node with minimum expected completion cost $f = h + g$. A* can be shown to have much better performance than uninformed search algorithms. However, an unresolved problem for this algorithm is that the number of expanded search nodes may grow exponentially even if the heuristic has only a small constant relative error [41]. Such heuristic functions are often encountered in practice, since many heuristics are derived from a relaxation of the search problem that is likely to introduce a relative error. Furthermore, in order to detect cycles and construct a solution, A* must keep all expanded nodes in memory. For this reason, the limiting factor of A* is often space rather than time.

In *symbolic model checking* [38], 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* are 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 the time and space complexity exponentially. Indeed during the last decade, remarkable results have been obtained using reduced ordered *Binary Decision Diagrams* (BDDs [9]) as the Boolean function representation. Systems with more than 10^{100} states have been successfully verified with the BDD-based model checker SMV [38]. 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 cost estimates such as the g and

[1] In this article, BFS always refers to best-first search and not breadth-first search.

h-costs associated with individual states when representing states implicitly.

In this article, we present a new framework called *state-set branching* that combines BDD-based search and best-first search (BFS) and efficiently solves the problem of representing cost estimates. State-set branching applies to any BFS algorithm and any transition cost function, heuristic function, and node-evaluation function defined over a finite domain. The state-set branching framework consists of two independent parts. The first part extends a general BFS algorithm to an algorithm called best-set-first search (BSFS) that expands sets of states in each iteration. The second part is an efficient BDD-based implementation of BSFS using 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 their associated cost estimates. The approach applies both to disjunctive and conjunctive partitioning [15].

Two implementations of A* based on the state-set branching framework called fSetA* anf ghSetA* have been experimentally evaluated in 8 search domains ranging from VLSI-design with synchronous actions, to classical AI planning problems such as the (n^2-1)-Puzzles, Blocks World and problems used in the AIPS 1998, 2000 and 2002 planning competition [35,2,36]. 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)-Puzzles, and HSPr [8] for the planning problems. In this experimental evaluation, the two A* implementations consistently outperform the ordinary A* algorithm. In addition, the results show that they can improve the complexity of A* exponentially and that they often dominate both the ordinary A* algorithm and blind BDD-based search by several orders of magnitude. Moreover, they have substantially better performance than BDDA*, the currently most efficient symbolic implementation of A*.

The remainder of this article is organized as follows. We first describe related work in Section 2. We then define search problems in Section 3 and describe the general BFS algorithm in Section 4. In Section 5, we extend this algorithm to expand sets of states and study a number of example applications of the new best-set-first search algorithm. In Section 6, we introduce branching partitioning and other BDD-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 BDD-based search. All previous work has been restricted to particular algorithms. BDD-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. [55] studies a bidirectional pure heuristic search algorithm pruning frontier states according to their minimum Hamming distance [2] to error states. BDDs representing Hamming distance equivalence classes are precomputed and conjoined with BDDs representing the search frontier during search. Yang and Dill [54] also consider minimum Hamming distance as heuristic function in an ordinary pure heuristic search algorithm. They develop a specialized BDD 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 in the size of the BDD 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 their cost estimates in two separate phases. Recent applications of BDD-based heuristic search in symbolic model checking include error directed search [46] and using symbolic pattern databases for guided invariant model checking [44].

In general, heuristic BDD-based search has received little attention in symbolic model checking. The reason is that the main application of BDDs in this field is verification where all reachable states must be explored. For Computation Tree Logic (CTL) checking [15], guiding techniques have been proposed to avoid a blow-up of intermediate BDDs during a reachability analysis [7]. 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, Edelkamp and Reffel [21] developed the first BDD-based implementation of A* called BDDA*. BDDA* can use any heuristic function defined over a finite domain and has been applied to planning as well as model checking [46]. Several extensions of BDDA* have been published including cycle detection, weighted evaluation function, pattern data bases, disjunctive transition relation partitioning, and external storage [18,19]. BDDA* is currently the most efficient symbolic implementation of A*. It contributes a combination of A* and BDDs where a single BDD is used to represent the search queue of A*. In each iteration, all states with minimum f-costs are extracted from this BDD. The successor states and their associated f-cost are then computed via

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

arithmetic BDD operations and added to the BDD representing the search queue.

There are two major differences between BDDA* and the SetA* algorithms presented in this article.

(1) Our experimental evaluation of BDDA* shows that its successor state function scales poorly (see Section 7.8). A detailed analysis of the computation shows that the complexity mainly is due to the symbolic arithmetic operations. For this reason, a main philosophy of state-set branching is to use BDDs only to represent state information. Cost estimates like the f-cost of a state is represented explicitly in a search tree.
(2) State-set branching introduces a novel approach called *branching partitioning* that makes it possible to use a transition relation partitioning to propagate cost estimates efficiently between *sets* of states during search. In this way, a best-first search algorithm called *best-set-first search* that expands sets of states in each iteration can be efficiently implemented with BDDs. As shown by our experimental evaluation in Section 7, this has a dramatic positive effect on the efficiency of the algorithms.

An ADD-based[3] implementation of A* called ADDA* has also been developed [25]. ADDs [3] generalize BDDs to finite valued functions and may simplify the representation of numeric information like the f-cost of states [53]. Similar to BDDA*, however, ADDA* performs arithmetic computations via complex ADD operations. It has not successfully been shown to have better performance than BDDA* [25].

A recent comparison of A* and a symbolic implementation of A* called SA* on 500 random 8-puzzle problems shows that SA* consistently uses more memory than A* and is outperformed by A* if the heuristic is strong [40]. These results are not confirmed by the experimental evaluation in this article where ghSetA* consistently uses less memory than A* and often finds solutions much faster than A*. We believe that there are several reasons for the observed differences. First, SA* does not use state-set branching to compute child nodes but instead relies on the less efficient two phase approach developed by Yuan et al.. Second, SA* stores expanded nodes without merging nodes with the same g-cost. This is done by ghSetA* and may lead to significant space savings. Third, 8-puzzle problems are very small ($< 10^6$ states) compared with the benchmark problems considered in our evaluation. It is unclear to what extend symbolic approaches pay off on such small problems. Fourth, the state-space of an $(n^2 - 1)$-Puzzle is a permutation space. It is easy to show that a BDD representation of a permutation space is exponentially more compact than an explicit representation. It is, however, still exponential in the

[3] ADD stands for Algebraic Decision Diagram [3].

number of elements in the permutation. For this reason, we may expect a high memory consumption of BDD-based search on $(n^2 - 1)$-Puzzles. Indeed, we get the weakest results for FSETA* and GHSETA* on the 24 and 35-Puzzle benchmarks.

Other applications of BDDs for search include HTN planning [33], STRIPS planning [12,16,51,23,29,10], universal planning [13,30], adversarial planning [31,17], fault tolerant planning [32], conformant planning [14], planning with extended goals [42], planning under partial observability [5,6], and shortest path search [48,47].

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 article, 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 factor* of that state. Since the domain is finite, the branching factor of each state is also finite. Each transition is assumed to have positive *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 $c : \mathcal{T} \rightarrow \mathbb{R}^+$ 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* is the sum of the transition costs of the path.

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 $j = 0, 1, \cdots, n-1$.*

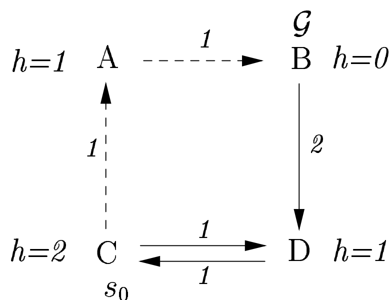

Fig. 1. An example search problem consisting of four states, five transitions, initial state s_0 = C, and a single goal state $\mathcal{G} = \{B\}$. The dashed path is an optimal solution. The h-costs associated with each state define the heuristic function used in Section 4.

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.

4 Best-First Search

Best-first 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 $\langle s, \vec{e} \rangle$ where s is a single state and $\vec{e} \in \mathbb{R}^d$ is a d-dimensional real vector representing the cost estimates associated with the node (e.g., the f-cost for A*). The root of the search tree contains the initial state. We will assume that the initial state is associated with cost estimate $\vec{e}_0$. 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 BFS 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 BFS 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. EXPAND in line 6 finds the set of child nodes of a single node, and ENQUEUEALL inserts each child in the frontier queue.

The A* algorithm is probably the most widely known and most well studied of the BFS algorithms. A* sorts the unexpanded nodes in the priority queue in ascending order of the a cost estimate given by a *heuristic evaluation function*

function BFS($s_0, \vec{e}_0, \mathcal{G}$)
1 *frontier* $\leftarrow$MAKEQUEUE($\langle s_0, \vec{e}_0 \rangle$)
2 **loop**
3 **if** $|frontier| = 0$ **then return** failure
4 $\langle s, \vec{e} \rangle \leftarrow$REMOVETOP(*frontier*)
5 **if** $s \in \mathcal{G}$ **then return** EXTRACTSOLUTION(*frontier*, $\langle s, \vec{e} \rangle$)
6 *frontier* $\leftarrow$ENQUEUEALL(*frontier*,EXPAND($\langle s, \vec{e} \rangle$))

Fig. 2. The general best-first search algorithm.

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.

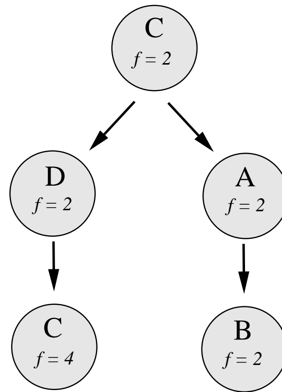

Fig. 3. Search tree example.

The properties of A* have been surveyed by Pearl [41]. 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. The heuristic function is called *consistent* if $h(n) \leq c(n, n') + h(n')$ for every successor node n' of n. It is trivial to show that any consistent heuristic function is admissible and that A* expands nodes with increasing f-cost when using a consistent heuristic. Therefore if the heuristic function is consistent, A* finds an optimal path to a state the first time a node of the state is expanded. Thus, future visits to the state can be ignored. It can be shown that every

```
function BSFS
1  frontier ←MAKEQUEUE(⟨{s_0}, e⃗_0⟩)
2  loop
3    if |frontier| = 0 then return failure
4    ⟨S, e⃗⟩ ←REMOVETOP(frontier)
5    if S ∩ G ≠ ∅ then return EXTRACTSOLUTION(frontier, ⟨S ∩ G, e⃗⟩)
6    frontier ←ENQUEUEANDMERGE(frontier, STATESETEXPAND(⟨S, e⃗⟩))
```

Fig. 4. The best-set-first search algorithm.

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 ($h(n) = h^*(n)$) and f-cost ties are broken by giving highest priority to the node with lowest h-cost, 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* with cycle detection using a consistent heuristic 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 [41]

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

The complexity results are discouraging 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 [43]

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

Weights $w = 0.0, 0.5$, and 1.0 correspond to uniform cost search, A*, and pure heuristic search. Weighted A* is optimal in the range $[0.0, 0.5]$ but often finds solutions faster in the range $(0.5, 1]$.

5 State-Set Branching

The state-set branching framework has two independent parts: a modification of the general BFS algorithm to a new algorithm called *best-set-first search* (BSFS), and a collection of BDD-based techniques for implementing the new

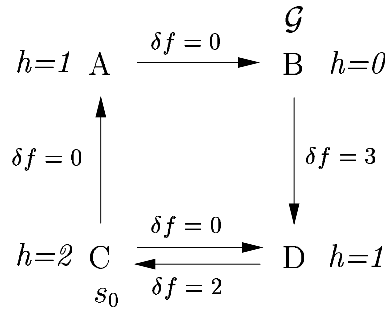

Fig. 5. The example search problem with δf costs.

algorithm efficiently. In this section, we will describe the BSFS algorithm. In next section, we show how it is implemented with BDDs.

5.1 Best-Set-First Search

Assume that each transition $\mathcal{T}(s, s')$ for a particular heuristic search algorithm changes the cost estimates with $\delta\vec{e}(s, s')$. Thus if s is associated with cost estimates $\vec{e}$ and s' is reached by $\mathcal{T}(s, s')$ then s' will be associated with cost estimates $\vec{e} + \delta\vec{e}(s, s')$. For A*, the cost estimates can be one or two dimensional: either it is the f-cost or the g and h-cost of a search node. In the first case $\delta\vec{e}(s, s')$ is the f-cost change caused by the transition. The δf costs of our example problem are shown in Figure 5. The BSFS algorithm shown in Figure 4 is almost identical to the BSFS 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 cost estimates. Multiple states in each node emerge because child nodes having identical cost estimates are coalesced by STATESETEXPAND in line 6 and because ENQUEUEANDMERGE may merge child nodes with nodes on the *frontier* queue having identical cost estimates. The state-set expansion function is defined in Figure 6. The next states of some child associated with cost estimates

function STATESETEXPAND($\langle S, \vec{e} \rangle$)
1 *child* $\leftarrow$ emptyMap
2 **foreach** state s **in** S
3   **foreach** transition $\mathcal{T}(s, s')$
4     $\vec{e}_c \leftarrow \vec{e} + \delta\vec{e}(s, s')$
5     $child[\vec{e}_c] \leftarrow child[\vec{e}_c] \cup \{s'\}$
6 **return** MAKENODES(*child*)

Fig. 6. The state set expand function.

$\vec{e}$ are stored in $child[\vec{e}]$. The outgoing transitions from each state in the parent node are used to find all successor states. The function MAKENODES called

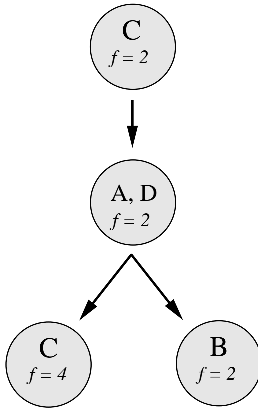

Fig. 7. State-set search tree example.

at line 6 constructs the child nodes from the completed child map. Each child node contains states having identical cost estimates. However, there may exist several nodes with the same cost estimates. In addition, MAKENODES may prune some of the child states (e.g., to implement cycle detection in A*).

As an example, Figure 7 shows the search tree traversed by the BSFS algorithm for A* applied to our example problem. In order to reduce the number of search nodes even further, ENQUEUEANDMERGE of the BSFS algorithm may merge nodes on the search frontier having identical cost estimates. This, however, transforms the search tree into a *Directed Acyclic Graph* (DAG).

Lemma 4 *The search structure build by the BSFS algorithm is a DAG where every node $\langle S', \vec{e}' \rangle$ different from a root node $\langle \{s_0\}, \vec{e}_0 \rangle$ has a set of predecessor nodes. For each state $s' \in S'$ in such a node, there exists a predecessor $\langle S, \vec{e} \rangle$ with a state $s \in S$ such that $\mathcal{T}(s, s')$ and $\vec{e}' = \vec{e} + \delta\vec{e}(s, s')$.*

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 $\langle \{s_0\}, \vec{e}_0 \rangle$. 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 $\langle S, \vec{e} \rangle$ 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 $\langle S', \vec{e}' \rangle$ has some state $s \in S$ in $\langle S, \vec{e} \rangle$ such that $\mathcal{T}(s, s')$ and $\vec{e}' = \vec{e} + \delta\vec{e}(s, s')$. Thus $\langle S, \vec{e} \rangle$ 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 5 *For each state $s' \in S'$ of a node $\langle S', \vec{e}' \rangle$ in a finite search structure of the BSFS algorithm there exists a path $\pi = s_0, \cdots, s_n$ in $\mathcal{D}$ such that $s_n = s'$ and $\vec{e} = \vec{e}_0 + \sum_{i=0}^{n-1} \delta\vec{e}(s_i, s_{i+1})$.*

PROOF. We will construct π by tracing the edges backwards in the search structure. Let $b_0 = s'$. According to Lemma 4 there exists a predecessor $\langle S, \vec{e} \rangle$ to $\langle S', \vec{e}' \rangle$ such that for some state $b_1 \in S$ we have $\mathcal{T}(b_1, b_0)$ and $\vec{e}' = \vec{e} + \delta\vec{e}(b_1, b_0)$. 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 $\pi = b_n, \cdots, b_1$. □

The EXTRACTSOLUTION function in line 5 of the BSFS search algorithm uses the backward traversal described in the proof of Lemma 5 to extract a solution. We can now prove soundness of the BSFS algorithm.

Theorem 6 *The BSFS algorithm is sound.*

PROOF. Assume that the algorithm returns a path $\pi = s_0, \cdots, s_n$ with cost estimates $\vec{e}$. Since $s_n \in \mathcal{G}$ it follows from Lemma 5 and the definition of EXTRACTSOLUTION that π is a solution to the search problem associated with cost estimates $\vec{e}$. □

It is not possible to show that the BSFS algorithm is complete since it covers incomplete algorithms such as pure heuristic search.

5.2 Example Implementations

The BSFS algorithm can be used to implement variants of pure heuristic search, A*, weighted A*, uniform cost search, and beam search.

Pure heuristic search is implemented by using the values of the heuristic function as cost estimates and sorting the nodes on the frontier in ascending order, such that the top node contains states with least h-cost. The cost estimate of the initial state is $\vec{e}_0 = h(s_0)$ and each transition $\mathcal{T}(s, s')$ is associated with the change in h, that is, $\delta\vec{e}(s, s') = h(s') - h(s)$. In each iteration, this

pure heuristic search algorithm will expand all states with least h-cost on the frontier.

A* can be implemented by setting $\vec{e}_0 = h(s_0)$ and $\delta\vec{e}(s,s') = c(s,s') + h(s') - h(s)$ such that the cost estimates equal the f-cost of search nodes. Again nodes on the frontier are sorted in ascending order. We call this implementation fSetA*. An A* implementation with cycle detection that does not require the heuristic function to be admissible or consistent must keep track of the g and h-cost separately and prune child states reached previously with a lower g-cost. Thus, $\vec{e}_0 = (0, h(s_0))$ and $\delta\vec{e}(s,s') = (c(s,s'), h(s') - h(s))$. 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-cost but different g and h-costs. In each iteration, it may therefore only expand a subset of the states on the frontier with minimum f-cost. A number of other improvements have been integrated in ghSetA*. First, it applies the usual tie breaking rule for nodes with identical f-cost choosing the node with the least h-cost. 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-cost 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* algorithm described in Section 5. Using an approach similar to Pearl [41], 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. [4]

6 BDD-based Implementation

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

[4] Notice that it follows from the optimality proof that fSetA* and ghSetA* are complete.

6.1 The BDD Representation

A BDD is a decision tree representation of a Boolean function on a set of linearly ordered arguments. The tree is reduced by removing redundant tests on argument variables and reusing structure. This transforms the tree into a rooted directed acyclic graph and makes the representation canonical. BDDs have several advantages: first, many functions encountered in practice (e.g., symmetric functions) have polynomial size, second, graphs of several BDDs can be shared and efficiently manipulated in multi-rooted BDDs, third, with the shared representation, equivalence and satisfiability tests on BDDs take constant time, and finally, fourth, the 16 Boolean operations on two BDDs x and y have time and space complexity $O(|x||y|)$ [9]. A disadvantage of BDDs 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 [39]. For a detailed introduction to BDDs, we refer the reader to Bryant's original paper [9] and the books [39,53].

6.2 BDD-based State Space Exploration

BDDs were originally applied to digital circuit verification [1]. More relevant, however, for the work presented in this article, they were later applied in *model checking* using a range of techniques collectively coined *symbolic model checking* [38]. During the last decade BDDs have successfully been applied to verify very large transition systems. The essential computation applied in symbolic model checking is an efficient reachability analysis where BDDs 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, a BDD can represent any set of states. The main efficiency of the BDD representation is that the cardinality of the represented set is not directly related to the size of the BDD. For instance a BDD with a single node can represent 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 BDDs.

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*

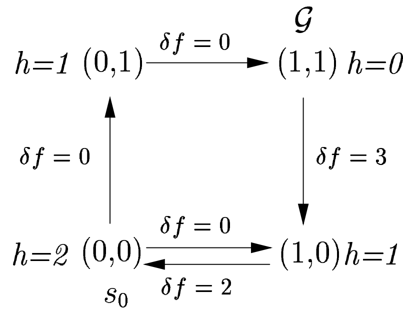

Fig. 8. Boolean state encoding of the example search problem.

variables, respectively. The symbol $S_{\vec{v}}$ denotes the state encoded by $\vec{v}$. To make this clear, two Boolean variables $\vec{v} = (v_0, v_1)$ are used in Figure 8 to represent the four states of our example problem. [5] The initial state is represented by a BDD 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 a BDD 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 BDD-based or symbolic search is to stay at the BDD 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

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

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 each variable in turn.

To illustrate the image computation, consider the first step of a search from

[5] Readers interested in studying the structure of BDD graphs representing sets of states and transition relations are referred to the papers [21,16]. In this article, we consider BDDs an abstract data type for manipulating Boolean functions and focus on explaining how implicit search can be performed by manipulating these functions.

s_0 in the example problem. We have $S(v_0, v_1) = \neg v_0 \wedge \neg v_1$. Thus

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

It is straight forward to implement uninformed or blind BDD-based search algorithms using the image and preimage computations. All sets and set operations in these algorithms are implemented with BDDs. The forward breadth-first search algorithm, shown in Figure 9, 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. A solution is

function FORWARD BREADTH-FIRST SEARCH
1 $\;$ $\mathit{reached} \leftarrow \emptyset$; $\mathit{forwardFrontier}_0 \leftarrow \{s_0\}$; $i \leftarrow 0$
2 $\;$ **while** $\mathit{forwardFrontier}_i \cap \mathcal{G} = \emptyset$
3 $\quad$ $i \leftarrow i + 1$
4 $\quad$ $\mathit{forwardFrontier}_i \leftarrow \text{IMAGE}(\mathit{forwardFrontier}_{i-1}) \setminus \mathit{reached}$
5 $\quad$ $\mathit{reached} \leftarrow \mathit{reached} \cup \mathit{forwardFrontier}_i$
6 $\quad$ **if** $\mathit{forwardFrontier}_i = \emptyset$ **return** failure
7 $\;$ **return** EXTRACTSOLUTION(*forwardFrontier*)

Fig. 9. BDD-based forward breadth-first search.

constructed by traversing the forward frontiers backward from a reached goal state to the initial state. This computation always has much lower complexity than the forward search, since the preimage computation in each iteration can be restricted to a BDD representing a single state.

Backward breadth-first search can be implemented in a similar fashion 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 opposite 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.

A common problem when computing the image and preimage is that the intermediate BDDs tend to be large compared to the BDD representing the result. Another problem is that the transition relation may grow very large if represented by a single BDD (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* [15]. 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

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

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 [11,45]

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

In domains where the transitions represent synchronous composition of parallel activity (e.g., in multi-agent domains [30]) disjunctive 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 modifies state variables $\vec{m}_j$ given the value of the state variables $\vec{d}_j$ as defined by its local transition relation $P_j(\vec{d}'_j, \vec{m}'_j)$. Given a total of n activities, we then have

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

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 sets of variables in the vectors $\vec{d}'_1, \cdots, \vec{d}'_n$ form a partitioning of the current state variables. An efficient image computation

combining conjunctive partitioning with early quantification is then defined by

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

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

6.4 The BDD-based BSFS Algorithm

The BSFS algorithm represents the states in each search node by a BDD. This may lead to exponential space savings compared to the explicit state representation used by the BFS algorithm. In addition, search nodes with similar BDDs may share structure in the multi-rooted BDD representation. This may further reduce the memory consumption substantially.

However, if we want an 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 cost estimates. 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 [54]. A more direct approach, however, is to split up the image computation such that the two phases are combined into one. We call this a branching partitioning.

For disjunctive partitioning the approach is straight-forward. We simply ensure that each partition contains transitions with the same cost estimate 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 cost estimate 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 BDDs as shown in Figure 10.

An efficient implicit node expansion computation is also possible to define for conjunctive partitioning if we assume that the change in cost estimates of a

```
function DISJUNCTIVESTATESETEXPAND(⟨S(v⃗), e⃗⟩)
1  child ← emptyMap
2  for i = 1 to |P|
4     e⃗_c ← e⃗ + δe⃗_i
5     child[e⃗_c] ← child[e⃗_c] ∪ IMAGE(P_i, S(v⃗))
6  return MAKENODES(child)
```

Fig. 10. The state set expand function for a disjunctive branching partitioning.

```
function CONJUNCTIVESTATESETEXPAND(⟨S(v⃗), e⃗⟩)
1  child ← emptyMap
2  child[e⃗] = S(v⃗)
3  for i = 1 to n
4     newChild ← emptyMap
5     foreach entry ⟨S(v⃗, v⃗'), δe⃗⟩ in child
6        for j = 1 to |P_i|
7           e⃗_c ← δe⃗ + δe⃗_i^j
8           newChild[e⃗_c] ← newChild[e⃗_c] ∪ ∃d⃗_i . S(v⃗, v⃗') ∧ P_i^j(d⃗'_i, m⃗'_i)
9     child ← newChild
10 return MAKENODES(child[v⃗'/v⃗])
```

Fig. 11. The state set expand function for a conjunctive branching partitioning.

global transition is equal to the sum of estimate changes 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 change in cost estimates. Let $P_i^j(\vec{d}'_i, \vec{m}'_i)$ denote subpartition j of activity i with cost estimate change $\delta\vec{e}_i^j$. To simplify the presentation, we assume again that the sets of variables in $\vec{d}'_1, \cdots, \vec{d}'_n$ form a partitioning of the current state variables. The state-set expansion function is then defined as shown in Figure 11. 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 cost estimates during the computation.

Expanded nodes with identical cost estimates may also be merged without compromising the ability to construct a solution. This is crucial since an individual storage of the nodes reduces the space efficiency of a symbolic representation.

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	:	BDD-based blind breadth-first bidirectional search using the heuristic for choosing search direction described in Section 6.2.
A*	:	Ordinary A* with cycle detection, explicit state manipulation, and evaluation function $f(n) = g(n) + h(n)$.
BDDA*	:	The BDDA* algorithm as described in [21].
iBDDA*	:	An improved version of BDDA* described below.

Table 1
The six search algorithms compared in the experimental evaluation.

7 Experimental Evaluation

Even though weighted A* and pure heuristic search are subsumed by the state-set branching framework, the experimental evaluation in this article 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 pure heuristic 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 [27].

We have implemented a general search engine in C++/STL using the BuDDy BDD package [6] [34]. This package has two major parameters: 1) the number of BDD-nodes allocated to represent the shared BDD (n), and 2) the number of BDD nodes allocated to represent BDDs 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 [37] or an extended version of *NADL* [30] with action costs. The output of the search engine is a solution found by one of the six search algorithms described in Table 1.

The GHSETA*, FSETA*, and BIDIR search algorithms have been implemented as described in this article. 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

[6] We also made experiments using the CUDD package [50], but did not obtain significantly better results than with the BuDDy package.

```
function BDDA*
1  open(f⃗, v⃗) ← h(f⃗, v⃗) ∧ s_0(v⃗)
2  while (open ≠ ∅)
3    (f_min, min(v⃗), open'(f⃗, v⃗)) ←GOLEFT(open)
4    if (∃v⃗ . (min(v⃗) ∧ G(v⃗)) return f_min
5    open''(f⃗', v⃗') ← ∃v⃗ . min(v⃗) ∧ T(v⃗, v⃗')∧
6      ∃e⃗ . h(e⃗, v⃗) ∧ ∃e⃗' . h(e⃗', v⃗') ∧ (f⃗' = f_min + e⃗' − e⃗ + 1)
7    open(f⃗, v⃗) ← open'(f⃗, v⃗) ∨ open''(f⃗', v⃗')[f⃗' \ f⃗, v⃗' \ v⃗]
```

Fig. 12. The BDDA* algorithm.

represented in the usual STRIPS fashion. All of the ordinary A* algorithms are implemented with cycle detection. The BDDA* algorithm has been implemented as described in [21]. The algorithm presented in this article is shown in Figure 12. It can only solve search problems in domains with unit transition costs. The search frontier is represented by a single BDD $open(\vec{f}, \vec{v})$. This BDD is the characteristic function of a set of states paired with their f-cost. The state is encoded as usual by a Boolean vector $\vec{v}$ and the f-cost is encoded in binary by the Boolean vector $\vec{f}$. Similar to FSETA*, BDDA* expands all states $min(\vec{v})$ with minimum f-cost (f_{min}) in each iteration. The f-cost of the child states is computed by arithmetic operations at the BDD level (line 5 and 6). The change in h-cost 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 implementation of BDDA* shows that 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-cost, (3) interleaving the BDD variables of $\vec{f}$, $\vec{e}$, and $\vec{e}'$ to improve the arithmetic BDD operations, and (4) moving this block of variables to the middle of the BDD variable ordering to reduce the average distance to dependent state variables. All of these improvements except the last have been considered to some degree in later versions of BDDA* [16]. The last improvement, however, 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 a factor of two speed up with the above modification. The improved algorithm is called iBDDA*.

In order to factor out differences due to state encodings and BDD computations, all BDD-based algorithms use the same bit vector representation of states, the same variable ordering of the state variables, and similar space

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.
$\lvert sol \rvert$	:	Solution length.
$\lvert expand \rvert$	:	For BIDIR this is the average size of the BDDs representing the search frontier. For FSETA* and GHSETA*, it is the average size of BDDs of search nodes being expanded. For BDDA* and iBDDA*, it is the average size of $open''$.
$\lvert maxQ \rvert$	:	Maximal number of nodes on the frontier queue.
$\lvert T \rvert$	:	The sum of the BDDs representing the partitioned transition relation.
it	:	Number of iterations of the algorithm.

Table 2
The performance parameters of the search engine.

allocation and cache sizes of the BDD package. This 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 performance parameters of the search engine are shown in Table 2. Time is measured in seconds. The time $t_{total} - t_{rel} - t_{search}$ is spent on allocating memory for the BDD package, parsing the problem description and in case of PDDL problems analyzing the problem in order to make a compact Boolean state encoding. For all domains, the size of the state space is given as the number of possible assignments to the Boolean state variables used to represent the domain. All experiments are carried out on a Linux 2.4 PC with 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 uses the minimum Hamming distance as heuristic function. It has been artificially designed to demonstrate that GHSETA* may have exponentially better performance than single-state A*. Next, we consider another artificial domain called the $D^xV^yM^z$ puzzle again using the minimum Hamming distance as heuristic function. The purpose of this domain is to show the scalability of state-set branching as a function of the dependency between objects in the domain. In particular, it demonstrates how the u parameter of GHSETA* can be used to focus the search on a subset of optimal paths when there is an abundance of these. We then turn to studying several well-known

search domains including the $(n^2\text{-}1)$-Puzzles and STRIPS [22] planning problems from the AIPS planning competitions [35,2,36]. We start by examining the 24 and 35-Puzzles using the usual sum of Manhattan distances as heuristic function. The planning domains include the Blocks World, Logistics, and Gripper. The experiments are interesting since we now consider a backward search guided by an approximation to the HSPr heuristic [8]. In all experiments to this point, we have considered state-set branching based on a disjunctive branching partitioning. In the final experiment, we show an example of state-set branching using a conjunctive branching partitioning. We study a range of channel routing problems from VLSI design produced from two circuits of the ISCAS-85 benchmarks [52] using a specialized heuristic function. We believe that this large set of experiments supporting an evaluation of our techniques is one of the main contributions of our work.

7.1 $\mathbf{FG}^k$

This problem is a modification of Barret and Weld's D^1S^1 problem [4]. 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

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

Each action is assumed to have unit cost. The initial state is $\{F^*\}$ and the goal state is $\{G_i | k < i \leq n\}$. Action $\mathbf{A}_i^1$ produces G_i given that G_{i-1} and F^* belong to the current state. In each state, however, the actions $\mathbf{A}_1^2, \cdots, \mathbf{A}_n^2$ are also applicable and they consume F^*. Thus, if one of these actions is applied no further $\mathbf{A}_i^1$ actions can be applied. This means that the only solution is $\mathbf{A}_1^1, \cdots, \mathbf{A}_n^1$. The purpose of the $\mathbf{A}_i^2$ actions is to make the decision of which action to apply in each state non-trivial. Without guidance the average number of states that must be visited in order to find a solution grows exponentially with the search depth.

This domain has been artificially designed to demonstrate the advantage of using BDDs to implicitly represent sets of states as done by GHSETA* compared to representing states explicitly as done by the ordinary single-state A* algorithm.

A state is represented by a vector of Boolean state variables

$$(G_1, \cdots, G_n, F_1, \cdots, F_n, F^*).$$

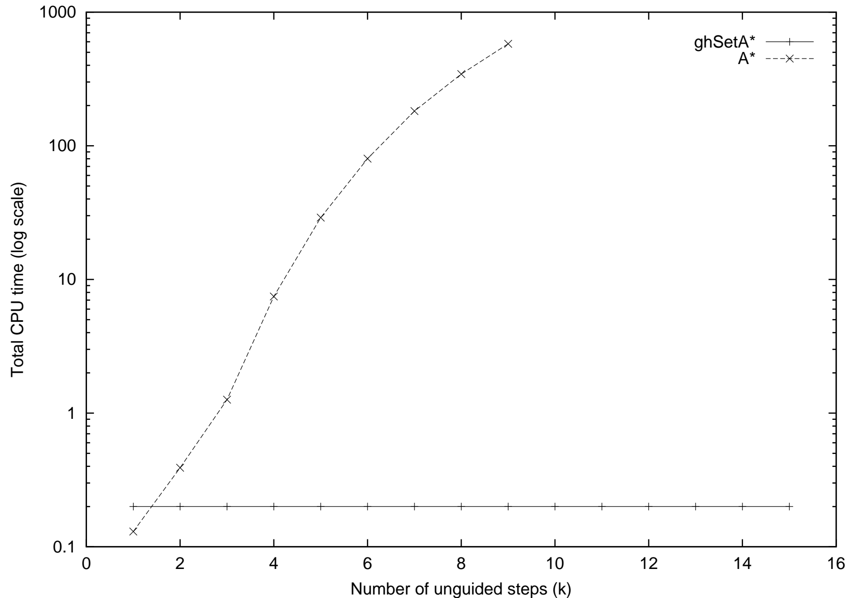

Fig. 13. Total CPU time of the FG^k problems.

Hence, in the initial state F^* is true, while all the other state variables are false. In a goal state, the state variables $G_{k+1}, \cdots, G_n$ are true while all other state variables may have arbitrary truth value. The heuristic value $h(s)$ of a state s is the minimum Hamming distance to a goal states. That is the number of goal state variables $(G_{k+1}, \cdots, G_n)$ that are false in the state s. Since the heuristic function gives no information to guide the search on the first k steps, we may expect the complexity of the ordinary A* algorithm to grow exponentially with k.

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 BDD operation for splitting *NADL* actions into transitions with the same cost estimate 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 BDD package are hand tuned in each experiment for best performance. Time out is 600 seconds. The results are shown in Figure 13. The performance of A* degrades quickly with the number of unguided steps. A* gets lost expanding an exponentially growing set of states. The GHSETA* algorithm is hardly affected by the lack of guidance. The reason is that GHSETA* performs an ordinary BDD-based blind forward search on the unguided part where the frontier states can be represented by a symmetric function with polynomial BDD size. Thus, the performance difference between A* and GHSETA* grows exponentially.

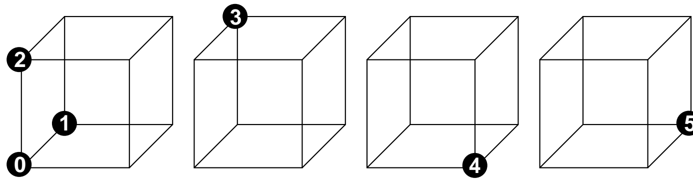

Fig. 14. The initial state of $D^3V^3M^6$.

7.2 $D^xV^yM^z$

The $D^xV^yM^z$ domain is an artificial puzzle domain where the dependency between objects in the domain can be adjusted without changing the size of the state space. The domain has the minimum Hamming distance as an admissible heuristic. It 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. Each action moves a single slider to an empty adjacent corner. The dimension of the hypercubes is y. That is, the hypercubes are described by y Boolean variables. For $y = 3$ the hypercubes are regular three dimensional cubes with 8 corners. Each corner is associated with a particular assignment of the y Boolean variables. We enumerate the corners according to the value encoded in binary of the Boolean variables. Hence, an action simply flips the value of one of these Boolean variables. There are z sliders of which x are moving on the same hypercube. The remaining $z-x$ sliders are moving on individual hypercubes. This means that there is a total of $z-x+1$ hypercubes. Sliders on individual hypercubes do not interact with other sliders. Thus, the x parameter can be used to adjust the dependency between sliders without changing the size of the state space.

The sliders are numbered. Initially, each slider is located at a corner position with the same number. There are 2^y corners on each hypercube. The goal is to move a slider with number n to the corner with number 2^y-n-1. Each action is assumed to have unit cost. Figure 14 shows the initial state of $D^3V^3M^6$.

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. In addition, it demonstrates how the u parameter of the GHSETA* algorithm can be used to focus the search when there is an abundance of optimal paths to explore. For the BDD-based algorithms, the $D^xV^4M^{15}$ problems are defined in *NADL*. Again a specialized poly-time BDD operation for splitting *NADL* actions into transitions with the same cost estimate 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 BDD-based algorithms except BDDA* utilize

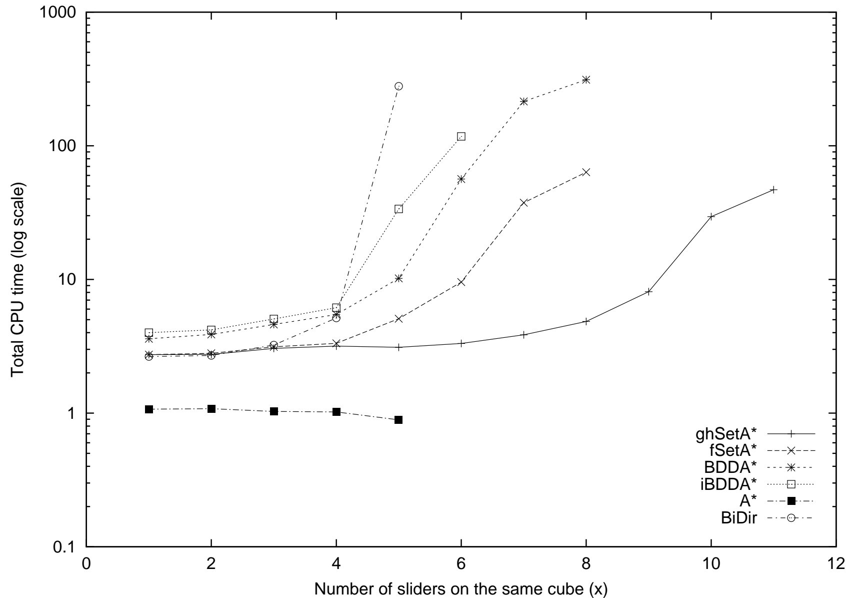

Fig. 15. Total CPU time of the $D^xV^4M^{15}$ problems.

a disjunctive partitioning with an upper bound on the BDDs representing a partition of 5000. Time out is 500 seconds. For all problems, the BDD-based algorithms use 2.3 seconds on initializing the BDD package ($n = 8000000$ and $c = 700000$). The results are shown in Table 3. Figure 15 shows a graph of the total CPU time for the algorithms.

All solutions found are 34 steps long. For BDDA* and iBDDA* the size of the BDD 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-cost 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 BDD 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

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 3
Results of the $D^x V^4 M^{15}$ problems.

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 BDD-based search algorithms applied to hard combinatorial problems: the BDDs representing the search frontiers blow up.

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

Fig. 16. Goal state of the 24-Puzzle.

7.3 The 24 and 35-Puzzle

We have further analyzed "non-artificial" domains. We aim at using domains that embed a search with a potential significant large number of search states. We turned to investigating the (n^2-1)-Puzzles in particular the 24 Puzzle $(n=5)$ and the 35 Puzzle $(n=6)$. The domain consists of an $n \times n$ board with n^2-1 numbered tiles and a blank space. A slide adjacent to the blank space can slide into the space. The task is to reach the European goal configuration as shown for the 24-Puzzle in Figure 16. For our experiments, the initial state is generated by performing r random moves from the goal state. [7] We assume unit cost transitions and use the usual sum of Manhattan distances of the tiles to their 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 BDDs in the frontier nodes $(u=\infty)$. For the BDD-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 25!. [8] The results of this problem are shown in Table 4. For all 24-puzzle problems, the BDD-based algorithms spend 3.6 seconds on initializing the BDD package $(n=15000000$ and $c=500000)$. Time out is 10000 seconds. For BDDA* and iBDDA* the size of the BDD 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.

[7] In each of these steps choosing the move back to the previous state is illegal.
[8] The number of solvable 24 puzzle states is about 10^{25}.

Algorithm	r	t_{total}	t_{rel}	t_{search}	$\lvert sol\rvert$	$\lvert expand\rvert$	$\lvert Q\rvert_{max}$	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 4
Results of the 24-puzzle problems.

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 function 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 states, it runs out of memory for solution depths above 50. For BIDIR, the problem is the usual: the BDDs representing the search frontiers blow up. Figure 17 shows a graph of the total CPU time of the 24 and 35-puzzle. Again time out is 10000 seconds.

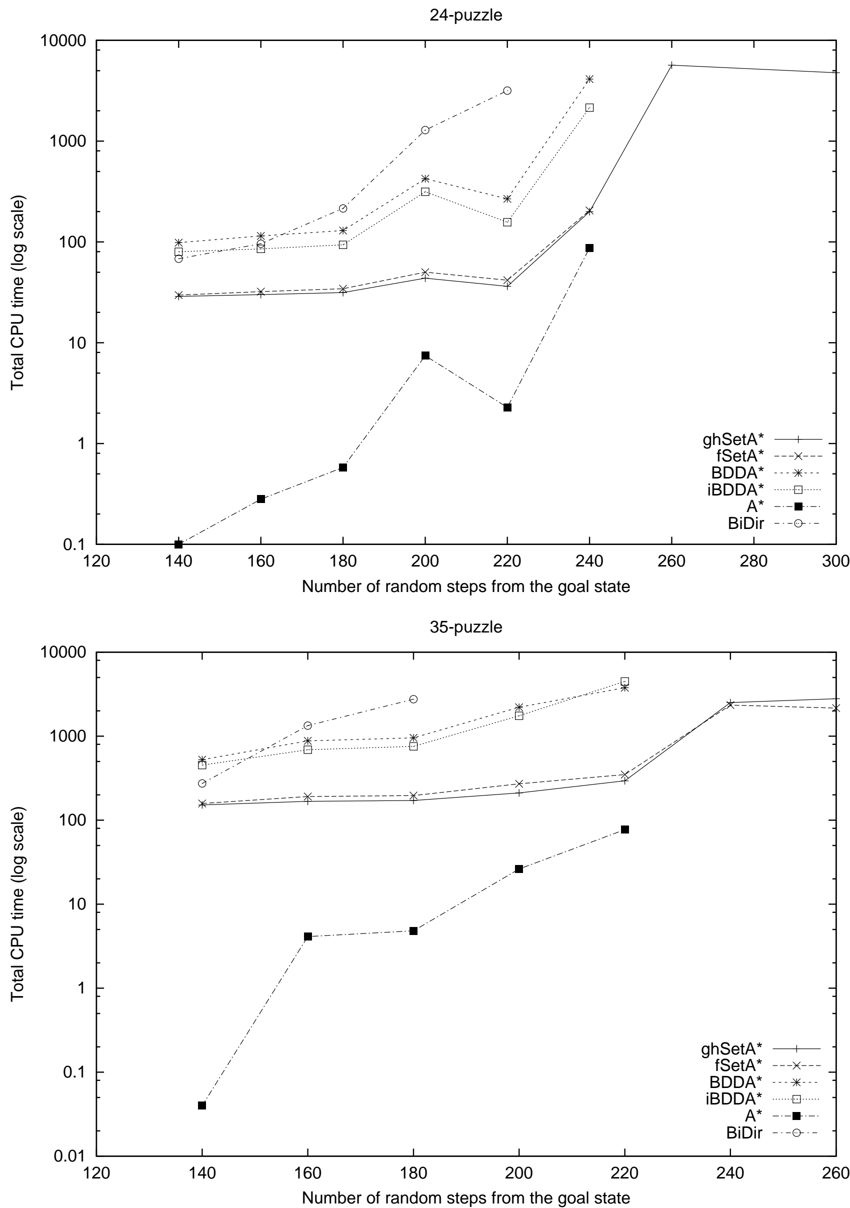

Fig. 17. Total CPU time for the 24 and 35-puzzle problems.

7.4 Planning Problems

In this section, we consider four planning problems from the STRIPS track of the AIPS 1998 [35], 2000 [2], and 2002 [36] 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

analyze 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* [24] 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 implemented based an approach described in [20]. It is fast for the problems considered in this article (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 [8]. 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 heuristic 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 article. 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-cost. In order to simplify the computation of the initial heuristic cost, 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 GHSETA* and FSETA* algorithms have no upper bound on the size of BDDs of the nodes on the frontier ($u = \infty$). For all BDD-based algorithms, the partition limit was 5000. For each problem, these algorithms spend about 2.5 seconds on initializing the BDD package ($n = 8000000$ and $c = 800000$). Time out is 500 seconds in all experiments. The results are shown in Table 5. The top graph of Figure 18 shows the total CPU time of the algorithms. For BDDA* and iBDDA* the size of the BDD 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 BDDs 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 performance of A*.

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 linearly from 2^{12} to 2^{88}. The GHSETA* and FSETA* algorithms have no upper bound on the size of BDDs in the frontier nodes ($u = \infty$). For all BDD-based algorithms no partition limit is used, and they spend about 0.8 seconds on initializing the BDD package ($n = 2000000$ and $c = 400000$). All algorithms generate optimal solutions. The results are shown in Table 6. The bottom graph of Figure 18 shows the total CPU time of the algorithms. Interestingly BIDIR is the fastest algorithm in this domain since the BDDs 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 BDDs of frontier nodes grow quite large for the harder problems.

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 5
Results of the Blocks World problems.

7.4.3 Logistics

The logistics domain considers moving packages with trucks between sub-cities 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

Algorithm	p	t_{total}	t_{rel}	t_{search}	$\lvert expand \rvert$	$\lvert Q \rvert_{max}$	it	$\lvert T \rvert$
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 6
Results of the Gripper problems.

upper bound on the size of BDDs in the frontier nodes ($u = \infty$). For all BDD-based algorithms a partition limit of 5000 is used and they spend about 2.0 seconds on initializing the BDD package ($n = 8000000$ and $c = 400000$). Due to systematic under estimation, the HSPr heuristic is scaled with a factor of 1.5. The top graph of Figure 19 shows the total CPU time of the algorithms.

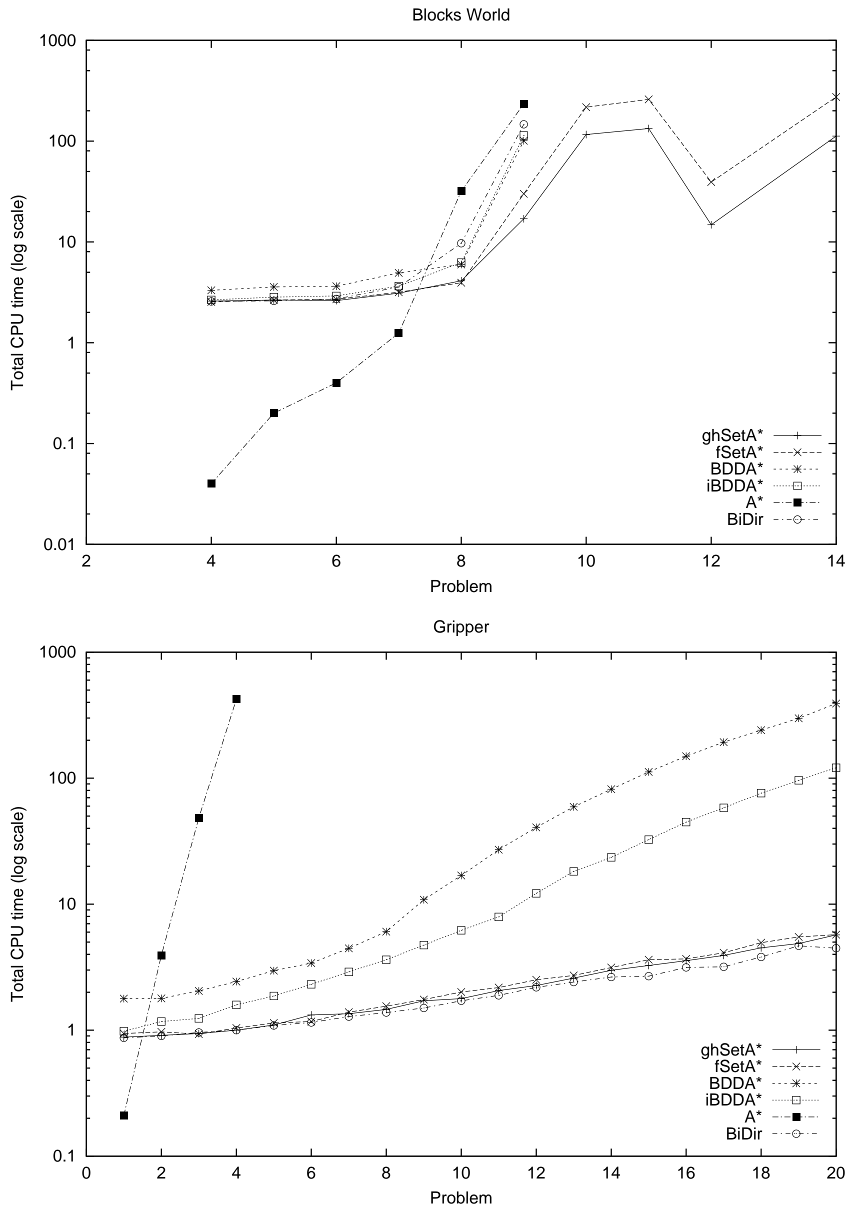

Fig. 18. Total CPU time for the Blocks World and Gripper problems.

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 BDDs in the frontier nodes ($u = \infty$). For all BDD-based algorithms a partition limit of 4000 is used. About 2.7 seconds is spent on initializing the BDD package ($n = 10000000$ and $c = 700000$). The bottom graph of Figure 19 shows the total CPU time of the algorithms. The results are very similar to the results of the logistics problems.

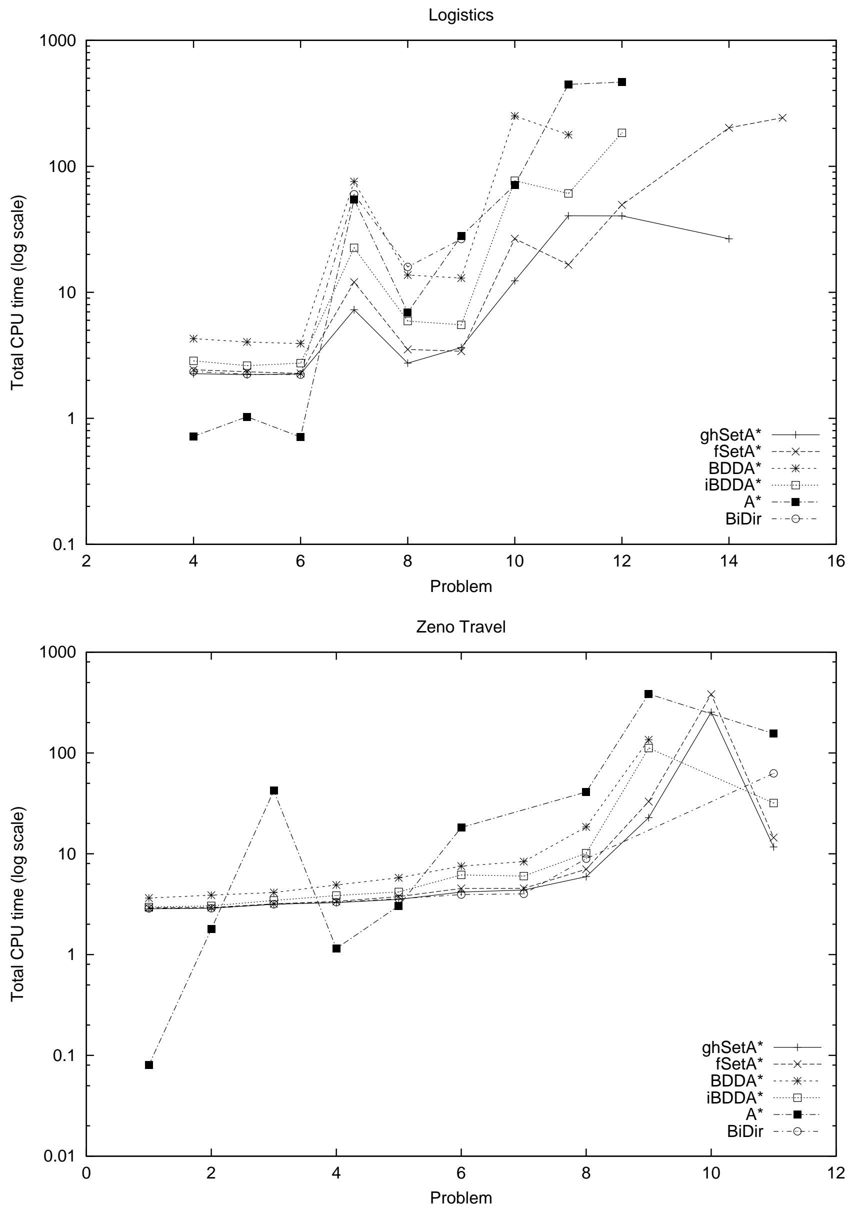

Fig. 19. Total CPU time for the Logistics and ZenoTravel problems. Problem 10 of ZenoTravel can only be solved by GHSETA* and FSETA*.

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 20 shows an example of an optimal solution to a small channel routing problem. The cost of the solution is 4. One way to apply search

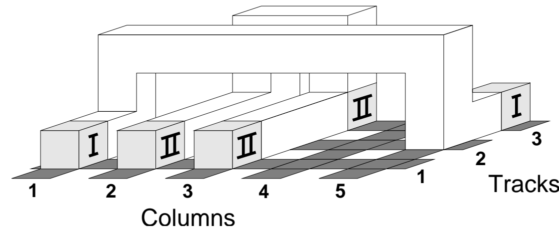

Fig. 20. A solution to a channel routing problem with 5 columns, 3 tracks, and 2 nets (labeled I and II). The pins are numbered according to what net they belong.

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* [28]. 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 article. 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 [52]. For each of these problems the parameters of the BDD package are hand tuned for best performance. There is no upper bound on the size of BDDs in frontier nodes ($u = \infty$) and no limit on the size of the partitions. Time out is 600 seconds. Table 7 shows the results. The performance of GHSETA* is similar to previous applications of BDDs to channel routing [49,52]. 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. In this article, we consider finding optimal or near optimal solutions with state-set branching implementations of A*. The studied heuristic functions are classical

Circuit	$c-t-n$	t_{total}	t_{rel}	t_{search}	$\|Q\|_{max}$	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 7
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).

but leaves a significant search element for the algorithms to handle. For these problems, state-set branching outperforms the ordinary A* algorithm. The reason is that a state-set branching implementation of A* may use an exponentially more compact state representation than the ordinary A* algorithm.

7.7 State-Set Branching versus Blind BDD-based Search

Blind BDD-based search has been successfully applied in symbolic model checking and circuit verification. It has been shown that many problems encountered in practice are tractable when using BDDs [53]. 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 BDD-based search to these problems, the BDDs used to represent the search frontier often grow exponentially. The experimental 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* 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

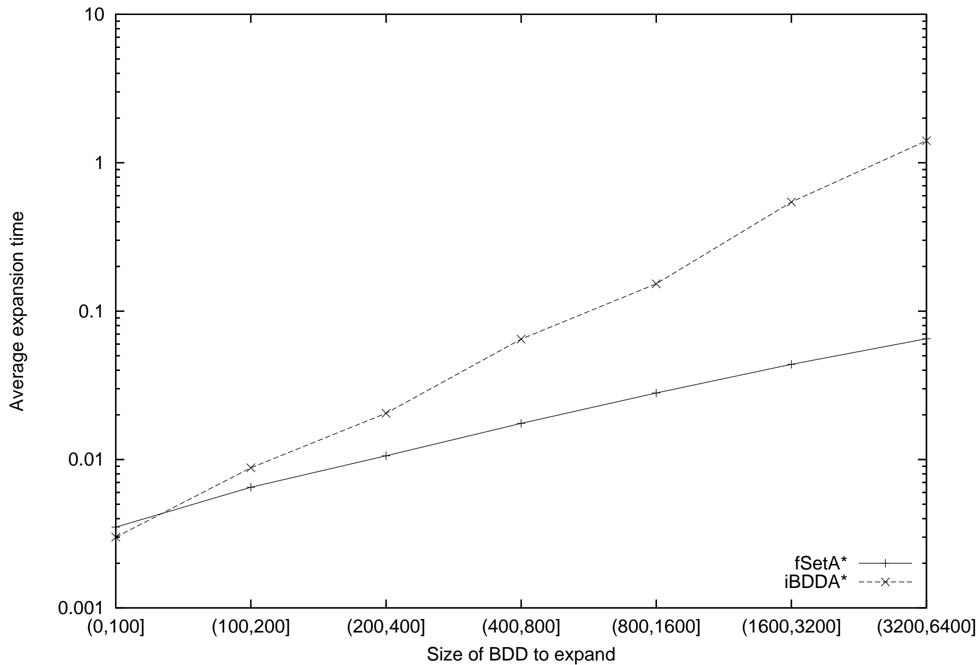

Fig. 21. Node expansion times of fSetA* and BDDA*.

functions studied in this article. The major challenge for BDDA* is that the arithmetic computations at the BDD level scales poorly with the size of the BDD representing the set of states to expand (line 5 and 6 in Figure 12). This hypothesis can be empirically 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 21. 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 D^xV^4M15 with x varying from 1 to 6. For very small frontier BDDs, iBDDA* is slightly faster than fSetA*. This is probably because small frontier BDDs 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 BDDs, BDDA* needs much more expansion time than fSetA*. Another limitation of BDDA* is the inflexibility of BDD-based arithmetic. It makes it hard to extend BDDA* efficiently to general evaluation functions and arbitrary transitions costs.

8 Conclusion

In this article, we have presented a framework called state-set branching for integrating symbolic and heuristic search. The key component of the frame-

work is a new BDD technique called branching partitioning that allows sets of states to be expanded implicitly and sorted according to cost estimates 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 defined over a finite domain. An extensive experimental evaluation of state-set branching proves it to be a powerful approach. It consistently outperforms the ordinary A* algorithm. 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 BDD-based search, and the most efficient current BDD-based A* implementation, BDDA*.

An important question, however, is how to compute branching partitions efficiently in general. Obviously, transitions cannot be sorted explicitly due to their large number. An implicit approach based on a symbolic encoding of the heuristic function exists [27], but interestingly, for the problems we have examined so far, even this implicit computation can be avoided. In many cases, heuristic functions are based on relaxations that decouples the search problem. For instance, for the minimum Hamming distance poly-time BDD 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.

Future 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 7 *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* $\langle S, \vec{e} \rangle$ *with a state* $s_i \in S$ *such that* $\vec{e} \leq C^*$ *and* $s_0, \cdots, s_i$ *is the search path associated with* s_i.

PROOF. A node $\langle S, \vec{e} \rangle$ 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 $\vec{e} = cost(s_0, \cdots, s_i) + h(s_i)$. 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 $\vec{e} \leq C^*$. □

Theorem 8 *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 $\vec{e} > C^*$. Since the node was at the top of the frontier queue, we have

$$C^* < f(n) \; \forall n \in frontier.$$

However, this contradicts Lemma 7 that states that any optimal path has a node on the frontier any time prior to termination with $\vec{e} \leq C^*$. □

Acknowledgements

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. Finally, we thank our anonymous reviewers for their valuable comments and suggestions.

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] P. Bertoli and M. Pistore. Planning with extended goals and partial observability. In *Proceedings of the 14th International Conference on Automated Planning and Scheduling (ICAPS-04)*, pages 270–278, 2004.

[7] 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.

[8] 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.

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

[10] D. Bryce and D. E. Smith. Planning graph heuristics for belief space search. *Journal of Artificial Intelligence Research*, (submitted).

[11] 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.

[12] 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.

[13] A. Cimatti, M. Pistore, M. Roveri, and P. Traverso. Weak, Strong, and Strong Cyclic Planning via Symbolic Model Checking. *Artificial Intelligence*, 147(1-2), 2003. Elsevier Science publishers.

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

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

[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. Symbolic exploration in two-player games: Preliminary results. In *Proceedings of the International Conference on AI Planning and Scheduling (AIPS'02) Workshop on Model Checking*, 2002.

[18] S. Edelkamp. Symbolic pattern databases in heuristic search planning. In *Proceedings of the 12th International Conference on AI Planning and Scheduling (ICAPS-02)*, pages 274–293, 2002.

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

[20] S. Edelkamp and M. Helmert. Exhibiting knowledge in planning problems to minimize state encoding length. In *Proceedings of the 6th European Conference on Planning (ECP'99)*, pages 135–147, 1999.

[21] 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.

[22] 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.

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

[24] 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.

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

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] R. M. Jensen, M. M. Veloso, and R. E. Bryant. Fault tolerant planning: Toward probabilistic uncertainty models in symbolic non-deterministic planning. In *Proceedings of the 14th International Conference on Automated Planning and Scheduling ICAPS-04*, pages 235–344, 2004.

[33] U. Kuter, D. Nau, M. Pistore, and P. Traverso. A hierarchical task-network planner based on symbolic model checking. In *Proceedings of the 15th International Conference on Automated Planning and Scheduling ICAPS-05*, pages 300–309, 2005.

[34] 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`.

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

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

[37] 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 for Computational Vision and Control, 1998.

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

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

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

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

[42] 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.

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

[44] K. Qian and A. Nymeyer. Guided invariant model checking based on abstraction and symbolic pattern databases. In *Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS-04)*, pages 497–511, 2004.

[45] 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.

[46] 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.

[47] D. Sawitzki. Experimental studies of symbolic shortest-path algorithms. In *Proceedings of the 3rd International Workshop on Experimental and Efficientl Algorithms (WEA-04)*, pages 482–498, 2004.

[48] D. Sawitzki. A symbolic approach to the all-pairs shortest-paths problem. In *Proceedings of the 30th International Workshop on Graph-Theoretic Concepts in Computer Science (WG-04)*, pages 154–168, 2004.

[49] 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.

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

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

[52] 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.

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

[54] 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.

[55] 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.