An Efficient BDD-Based A* Algorithm

Rune M. Jensen, Randal E. Bryant, and Manuela M. Veloso
Computer Science Department,Carnegie Mellon University,
5000 Forbes Ave, Pittsburgh,PA 15213-3891, USA
{runej,bryant,mmv}@cs.cmu.edu

Abstract

In this paper we combine the goal directed search of A* with the ability of BDDs to traverse an exponential number of states in polynomial time. We introduce a new algorithm, SetA*, that generalizes A* to expand sets of states in each iteration. SetA* has substantial advantages over BDDA*, the only previous BDD-based A* implementation we are aware of. Our experimental evaluation proves SetA* to be a powerful search paradigm. For some of the studied problems it outperforms BDDA*, A*, and BDD-based breadth-first search by several orders of magnitude. We believe exploring sets of states to be essential when the heuristic function is weak. For problems with strong heuristics, SetA* efficiently specializes to single-state search and consequently challenges single-state heuristic search in general.

Introduction

During the last decade, powerful search techniques using an implicit state representation based on the *reduced ordered binary decision diagram* (BDD, Bryant 1986) have been developed in the area of *symbolic model checking* (McMillan 1993). Using blind exploration strategies these techniques have been successfully applied to verify systems with very large state spaces. Similar results have been obtained in well-structured AI search domains (Cimatti *et al.* 1997). However for hard combinatorial problems the search fringe often grows exponentially with the search depth.

A classical AI approach for avoiding the state explosion problem is to use heuristics to guide the search toward the goal states. The question is whether heuristics can be applied to BDD-based search such that their ability to efficiently expand a large set of states in each iteration is preserved. The answer is non-trivial since heuristic search algorithms require values to be associated with each state and manipulated during search. A task for which BDDs often have proven less efficient.

In this paper, we present a new search algorithm called SetA*. The main idea is to avoid the above problem by generalizing A* (Hart, Nilsson, & Raphael 1968) from single states to sets of states in the search queue. Recall that A* associates two values g and h to each state in the search queue. g is the cost of reaching the state and h is an estimate of the remaining cost of reaching the goal given by a *heuristic function*. In SetA* states with similar g and h values are merged such that we can represent them implicitly by a BDD without having to store any numerical information. In each iteration, SetA*: 1) pops the set with highest priority, 2) computes its next states, and 3) partitions the next states into child sets with unique g and h values, which are reinserted into the queue. A straightforward implementation of the three phases has disappointing performance (see PreSetA*, Table 2). A key idea of our work is therefore to combine phase 2 and 3. The technique fits nicely with the so called disjunctive partitioning of BDD-based search (Clarke, Grumberg, & Peled 1999). In addition it can be applied to any heuristic function. Our experimental evaluation of SetA* proves it a powerful search paradigm. For some problems it dominates both A* and BDD-based breadth-first search (see Table 2). In addition, it outperforms the only previous BDD-based implementation of A* (BDDA*, Edelkamp & Reffel 1998), we are aware of, with up to two orders of magnitude (see Table 4).

The remainder of the paper is organized as follows. First we briefly describe BDDs and BDD-based search. We then define the SetA* algorithm and evaluate it experimentally in a range of search and planning domains. Finally we discuss related work and draw conclusions.

BDD-based Search

A BDD is a canonical representation of a Boolean function with n linear ordered arguments $x_1, x_2, ..., x_n$. It is a rooted, directed acyclic graph with one or two terminal nodes labeled 1 or 0, and a set of variable nodes u of out-degree two. The two outgoing edges are given by the functions $high(u)$ and $low(u)$ (drawn as solid and dotted arrows). Each variable node is associated with a propositional variable in the Boolean function the BDD represents. The graph is ordered in the sense that all paths in the graph respect the ordering of the variables. A BDD representing the function $f(x_1, x_2) = x_1 \wedge x_2$ is shown in Figure 1 (left). Given an assignment of the arguments x_1 and x_2, the value of f is determined by a path starting at the root node and iteratively following the high edge, if the associated variable is true, and the low edge, if the associated variable is false. The value of f is $True$ if the label of the reached terminal node is 1; otherwise it is $False$. A BDD is reduced so that no two distinct nodes u and v have the same variable name

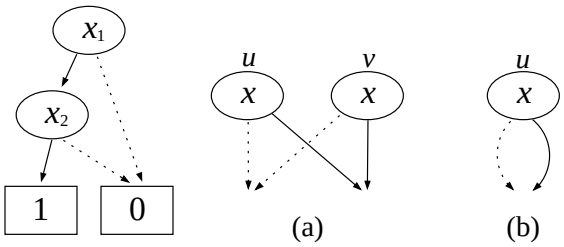

Figure 1: A BDD representing the function $f(x_1, x_2) = x_1 \wedge x_2$. True and false edges are drawn solid and dotted, respectively. (a) and (b) Reductions of BDDs.

and low and high successors (Figure 1(a)), and no variable node u has identical low and high successors (Figure 1(b)). The BDD representation has two major advantages: first, many functions encountered in practice have a polynomial size. Second, any operation on two BDDs, corresponding to a Boolean operation on the functions they represent, has a low complexity bounded by the product of their node counts.

A search problem is a 4-tuple (S, T, i, G). S is a set of states. $T : S \times S$ is a *transition relation* defining the search graph. $(s, s') \in T$ iff there exists a transition leading from s to s'. i is the initial state of the search while G is the set of goal states. A solution to a search problem is a path $\pi = s_0, \cdots, s_n$ where $s_0 = i$ and $s_n \in G$ and $\bigwedge_{j=0}^{n-1}(s_j, s_{j+1}) \in T$. Assuming that states can be encoded as bit vectors, BDDs can be used to represent the *characteristic function* of a set of states and the transition relation. To make this clear, consider the simple search problem shown in Figure 2. A state s is represented by a bit vector with two elements $\vec{s} = (s_0, s_1)$. Thus the initial state is represented by a BDD for the expression $\neg s_0 \wedge \neg s_1$. Similarly we have $G = s_0 \wedge s_1$. To encode the transition relation, we need to refer to current state variables and next state variables. We adopt the usual notation in BDD literature of primed variables for the next state

$$\begin{aligned} T(s_0, s_1, s_0', s_1') \;=\; & \neg s_0 \wedge \neg s_1 \;\wedge\; s_0' \wedge \neg s_1' \\ \vee\; & \neg s_0 \wedge \neg s_1 \;\wedge\; \neg s_0' \wedge s_1' \\ \vee\; & \neg s_0 \wedge s_1 \;\wedge\; s_0' \wedge s_1' \\ \vee\; & s_0 \wedge s_1 \;\wedge\; s_0' \wedge \neg s_1'. \end{aligned}$$

The main idea in BDD-based search is to stay at the BDD level when finding the next states of a set of states. This can be done by computing the image of a set of states V encoded in current state variables

$$\text{Img} = \big(\exists \vec{s}\,.\, V(\vec{s}) \wedge T(\vec{s}, \vec{s}')\big)[\vec{s}/\vec{s}'].$$

Consider the first step of the search from i in the example domain. We have $V(s_0, s_1) = \neg s_0 \wedge \neg s_1$. Thus,

$$\begin{aligned} \text{Img} &= \big(\exists \vec{s}\,.\, \neg s_0 \wedge \neg s_1 \wedge T(s_0, s_1, s_0', s_1')\big)[\vec{s}/\vec{s}'] \\ &= \big(s_0' \wedge \neg s_1' \;\vee\; \neg s_0' \wedge s_1'\big)[\vec{s}/\vec{s}'] \\ &= s_0 \wedge \neg s_1 \;\vee\; \neg s_0 \wedge s_1. \end{aligned}$$

The image computation is applied for searching in forward direction. For searching backward an analogous computation called the *preimage* is applied. In this section, we focus on techniques for performing the image computation efficiently, but similar techniques exist for the preimage computation.

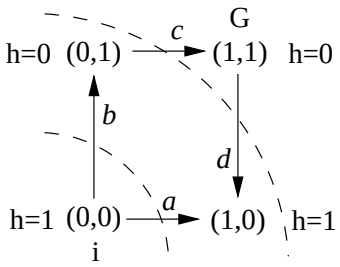

Figure 2: An example search problem consisting of four states and four transitions *a*,*b*,*c*, and *d*. The dashed lines indicate the two search fringes of a BDD-based breadth-first search from the initial state $i = (0, 0)$ to the goal states $G = \{(1, 1)\}$. The h-values is a heuristic function equal to the vertical goal distance.

A common problem in BDD-based search is that intermediate BDDs in the image computation tend to be large compared to the BDD representing the result. In symbolic model checking, a range of techniques has been proposed to avoid this problem. Among the most successful of these are *transition relation partitioning*. For search problems, where each transition normally only modifies a small subset of the state variables, the suitable partitioning technique is *disjunctive partitioning* (Clarke, Grumberg, & Peled 1999). To make a disjunctive partitioning, the part of the individual transition expressions keeping the unmodified variables unchanged is removed. The transition expressions are then partitioned according to what variables they modify. For our example we get two partitions

$$\begin{aligned} P_1 &= \neg s_0 \wedge \neg s_1 \wedge s_0' \;\vee\; \neg s_0 \wedge s_1 \wedge s_0' \\ m_1 &= (s_0) \\ P_2 &= \neg s_0 \wedge \neg s_1 \wedge s_1' \;\vee\; s_0 \wedge s_1 \wedge \neg s_1' \\ m_2 &= (s_1). \end{aligned}$$

In addition to large space savings, disjunctive partitioning often lowers the complexity of the image computation which now can skip the quantification of unchanged variables and operate on smaller expressions

$$\text{Img} = \bigvee_{j=1}^{|\mathbf{P}|} \big(\exists m_j\,.\, V(\vec{s}) \wedge P_j(\vec{s}, m_j')\big)[m_j/m_j'].$$

The complexity of the image computation depends on the number of partitions. Notice that for each new partition, a new conjunction with $V(\vec{s})$ is introduced. For this reason the best performance is often obtained by clustering some of the partitions according to an upper bound on the size of the BDD representing a partition (Burch, Clarke, & Long 1991; Ranjan *et al.* 1995).

SetA*

SetA* is a generalization of weighted A* where the definition of f is changed from $f = g + h$ to $f = (1 - w)g + wh$, $w \in [0, 1]$ (Pohl 1970). Similar to BDDA*, SetA* assumes a finite search domain and unit-cost transitions. SetA* expands a set of states instead of just a single state. The main input is what we will call, an *improvement partitioning*. That is, a disjunctive partitioning where the transitions of a partition reduce the h-value by the same amount. The improvement partitioning is non-trivial to compute. The reason is that it may be intractable to calculate each transition expression in turn. Fortunately large sets of transitions are often described in more abstract terms (e.g. by *actions* or *guarded commands*) that can be directly translated into BDDs. This allows for an implicit way to partition a set of transitions according to their improvement. Assume that a set of transitions are represented by a BDD $T(\vec{s}, \vec{s}')$. Given a BDD $h(\vec{s}, \vec{v})$ encoding the heuristic function, such that $\vec{v}$ is a bit vector representation of the h-value associated with state s, the set of transitions with improvement equal to k is

$$T(\vec{s}, \vec{s}') \wedge h(\vec{s}, \vec{v}) \wedge h(\vec{s}', \vec{v}') \wedge \vec{v} - \vec{v}' = \vec{k}.$$

The improvement partitioning is computed only once prior to the search, and in practice it turns out that it often can be produced directly from the description of transitions or by splitting the disjunctive partitioning. In fact, for the heuristics we have studied so far, no BDD encoding of the heuristic function has been necessary. The improvement partitioning may containing several partitions with similar improvement. This may be an advantage if the partitions otherwise grow too large.

SetA* uses two main data structures: a prioritized queue Q and a reach structure R. Each node in Q contains a BDD representing a set of states with particular g and h values. The node with lowest f-value has highest priority. Ties are solved by giving highest priority to the entry with lowest h-value. An important parameter of Q is an upper bound u on the BDD sizes. When inserting a new node it is unioned with an existing node in Q with the same g and h value if the sum of the size of their two BDDs is less than u. Otherwise a new entry is created for the node. The reach structure is for loop detection. R keeps track of the lowest g-value of every reached state and is used to prune states from a set of next states already reached with a lower g-value. The algorithm is shown in Figure 3. All sets and set operations are carried out with BDDs. SetA* takes five arguments. **IP** is the improvement partitioning described above. $init$ and $goal$ are the initial and goal states of the search. u is the upper bound parameter of Q and w is the usual weight parameter of weighted A*. Initially the algorithm inserts the initial state in Q. Observe that the h-value of the initial state has to be found. However since $init$ is a single state this is trivial. Similar to the regular A* algorithm, SetA* continues popping the top node of the queue until the queue is either empty or the states of the top node overlaps with the goal. The top node is expanded by finding the image of it for each improvement partition in turn (l.9). Before being inserted in Q, the new nodes are pruned for states seen at a lower

```
function SetA*(IP, init, goal, u, w)
1    Q.initialize(u, w, goal)
2    g ← 0
3    h ← h(init)
4    Q.insert(init, g, h)
5    R.update(init, g)
6    while ¬Q.empty() and ¬Q.topAtGoal()
7        top ← Q.pop()
8        for j = 0 to |IP|
9            next ← image(top, IP_j)
10           R.prune(next)
11           g ← top.g + 1
12           h ← top.h − impr(IP_j)
13           Q.insert(next, g, h)
14           R.update(next, g)
15   if Q.empty() then NoPathExists
16       else R.extractPath()
```

Figure 3: The SetA* algorithm.

search depth, and the reach structure is updated (l.10-14). If the loop was aborted due to Q being empty no solution path exists. Otherwise the path is extracted by applying transitions backwards on the states in R from one of the reached goal states.

SetA* is *sound* due to the soundness of the image computation. Since no states reached by the search are pruned, SetA* is also *complete*. Given an *admissible heuristic* and $w = 0.5$, SetA* further finds *optimal length paths*. As for A*, the reason is that a state on the optimal path eventually will reach the top of Q because states on finalized but suboptimal paths have higher f-value (Pearl 1984).

The upper bound u can be used to adjust how many states SetA* expands. If each partition in **IP** contains a single transition and $u = 0$ then SetA* specializes to A*. Interestingly it is even a highly efficient implementation of A*. The memory sharing of BDDs robustly scales to tens of thousands of BDDs, and loop detection is still handled implicitly via BDDs in the R structure. For problems with many shortest length solution paths like the DVM and Logistics described in the next section, it may be an advantage to focus on a subset of them by choosing a low u-value. A similar approach is used by A^*_ϵ described in (Pearl 1984)

The weight w has the usual effect. For $w = 0.5$ Set A* behaves like A*. For $w = 1.0$ it performs best-first search, and for $w = 0.0$ it carries out a regular breadth-first search. The fact that w can take any value in the range $[0, 1]$ is important in practice, since it can be used to strengthen a conservative heuristic and vice versa.

We end this section by demonstrating SetA* on our example problem. For this demonstration we assume $w = 0.5$ and $u = \infty$. The heuristic function is the vertical distance to the goal state. In Figure 2 the states have been labeled with h-values. We see that **IP** must contain at least three partitions: one containing transition d that improves by minus one, one containing a and c that improve by zero, and

one containing b that improves by one. Initially we have

$$\begin{aligned} Q_0 &= < (f = 0.5, g = 0, h = 1, \{(0,0)\}) > \\ R_0 &= < (g = 0, \{(0,0)\}) > . \end{aligned}$$

In the first iteration, state $(0,0)$ is expanded to one child containing state $(1,0)$ and one child containing $(0,1)$. According to the improvements of the partitions, we get

$$\begin{aligned} Q_1 &= < (f = 0.5, g = 1, h = 0, \{(0,1)\}), \\ &\quad (f = 1.0, g = 1, h = 1, \{(1,0)\}) > \\ R_1 &= < (g = 0, \{(0,0)\}), (g = 1, \{(0,1), (1,0)\}) > . \end{aligned}$$

In the second iteration, only the *c* transition can fire resulting in

$$\begin{aligned} Q_2 &= < (f = 1.0, g = 2, h = 0, \{(0,1)\}), \\ &\quad (f = 1.0, g = 1, h = 1, \{(1,0)\}) > \\ R_2 &= < (g = 0, \{(0,0)\}), (g = 1, \{(0,1), (1,0)\}), \\ &\quad (g = 2, \{(1,1)\}) > . \end{aligned}$$

The tie breaking rule causes the goal state to be at the top of Q at the beginning of the third iteration. Thus the while loop is aborted and the solution path $(0,0), (0,1), (1,1)$ is extracted from R_2.

Experimental Evaluation

SetA* has been implemented in the UMOP multi-agent planning framework (Jensen & Veloso 2000) to study its performance characteristics relative to blind bidirectional BDD-based breadth-first search (also implemented in UMOP) and an A* implementation with explicit state representation and cycle detection. In a second evaluation round we developed a domain independent STRIPS planning system called DOP. The state encoding and heuristic function used by the MIPS planner (Edelkamp & Helmert 2001) was reproduced in order to conduct a fair comparison with BDDA* implemented in MIPS. In addition to SetA*, two blind BDD-based breadth-first search algorithms were implemented in DOP, one searching forward and one searching backward. MIPS also includes an algorithm called Pure BDDA*. Pure BDDA* performs best-first search.

All experiments were carried out on a Linux 5.2 PC with a 500 MHz Pentium 3 CPU, 512 KB L2 cache and 512 MB RAM. The time limit (TIME) was 600 seconds and the memory limit (MEM) was 450 MB. For UMOP and DOP the number allocated BDD nodes and the cache size used by the BDD-package were hand-tuned for best performance. A disjunctive partitioning with a minimum number of partitions was applied unless otherwise noted.

Artificial Problems

Two problems IG^k and $D^x V^y M^z$ were defined and studied using the minimum *Hamming distance* to the goal states as heuristic function (the mimimum number of different bits between the bit vector representing the state and a goal state). In these experiments the improvement partitioning was computed by splitting a disjunctive partitioning using a specialized BDD-function. Given an improvement k, this function traverses the BDD of an action and picks transitions of the action improving k. The complexity of the function is linear in the size of the action BDD when the goal is a conjunction and the variable ordering interleaves current and next state variables.

k	SetA* #it	SetA* T (sec)	A* #it	A* T (sec)
1	16	0.2	16	0.13
2	16	0.2	145	0.39
3	16	0.2	514	1.26
4	16	0.2	2861	7.46
5	16	0.2	9955	29.02
6	16	0.2	24931	80.10
7	16	0.2	51098	181.77
8	16	0.2	90080	344.00
9	16	0.2	140756	579.22
10	16	0.2	-	TIME
11	16	0.2	-	-
12	16	0.2	-	-
13	16	0.2	-	-
14	16	0.2	-	-
15	16	0.2	-	-

Table 1: Results for the IG^k problem. #it is the number of iterations, and T is the total CPU time.

IG^k This problem is simplest to define using the STRIPS language (Fikes & Nilsson 1971). Thus a state is a set of facts and an action is a triple of sets of facts. In a given state S, an action (pre, add, del) is applicable if $pre \subseteq S$, and the resulting state is $S' = (S \cup add) \setminus del$. The actions are

$\mathbf{A}_1^1$	$\mathbf{A}_j^1$ $j = 2, \cdots, n$	$\mathbf{A}_j^2$ $j = 1, \cdots, n$
$pre : \{I^*\}$	$pre : \{I^*, G_{j-1}\}$	$pre : \{\}$
$add : \{G_1\}$	$add : \{G_j\}$	$add : \{I_j\}$
$del : \{\}$	$del : \{\}$	$del : \{I^*\}$.

The initial state is $\{I^*\}$ and the goal state is $\{G_j | k < j \leq n\}$. Only $\mathbf{A}_j^1$ actions should be applied to reach the goal. Applying an $\mathbf{A}_j^2$ action in any state leads to a wild path since I^* is deleted. The states on wild paths contain I_j facts. Since any subset of I_j facts is possible, the number of states on wild paths grows exponentially with n. The only solution is $\mathbf{A}_1^1, \cdots, \mathbf{A}_n^1$ which is non-trivial to find, since the heuristic gives no information to guide the search on the first k steps. The purpose of the experiment is to investigate how well SetA* copes with this situation compared to A*. For SetA* $w = 0.5$ and $u = \infty$. For the IG^k problems considered, n equals 16. This corresponds to a state space size of 2^{33}. The results are shown in Table 1.

The experiment shows a fast degradation of A*'s performance with the number of unguided steps. A* gets lost expanding an exponentially growing set of states on wild paths. SetA* is hardly affected by the lack of guidance. The reason is that all transitions on the unguided part improve by zero. Thus on this part, SetA* performs a regular BDD-based breadth-first search, which due to the structure

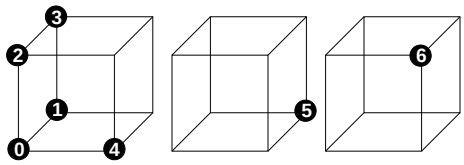

Figure 4: The initial state of $D^5V^3M^7$.

of the problem scales well.

$\mathbf{D^xV^yM^z}$ In this domain a set of sliders are moved between the corner positions of hypercubes. In any state, a corner position can be occupied by at most one slider. The dimension of the hypercubes is y. There are z sliders of which x are moving on the same cube. The remaining $z - x$ sliders are moving on individual cubes. Figure 4 shows the initial state of $D^5V^3M^7$. When $x = z$ all sliders are moving on the same cube. If further $x = 2^y - 1$ all corners of the cube except one will be occupied. In this form, DVM is a *permutation problem* similar to the 15puzzle and Rubik's Cube. We choose to investigate DVM instead of these well-known problems because it has a direct Boolean encoding. In this way, the complexity of the problem is solely caused by the interaction between sliders, allowing us to adjust the dependency of sliders linearly with the x parameter. For the 15puzzle and Rubik's Cube there would be two sources of complexity. One due to the interaction between objects and one due to the physical constraints of the puzzles.

The purpose of the first experiment is to investigate how SetA* degrades when the dependency of the domain is increased and to compare its performance to A* and BDD-based breadth-first search. In this experiment we study the $D^xV^4M^{15}$ problem. For all experiments the size of the state space is 2^{60}. We also show the results of PreSetA*, a premature version of SetA* finding the next states and splitting them in two separate phases. Both versions of SetA* were run with $w = 0.5$ and $u = 200$. In this experiment an upper bound of 1000 on the size of the disjunctive partition BDDs was chosen. The results are shown in Table 2.

The upper bound on the partitions is crucial for large values of x. Despite applying this technique, BDD-based bidirectional search does not scale due to a blow-up of the search fringe in both directions. A* works well when x is small since f is a perfect or near perfect discriminator. However when the quality of the heuristic degrades A* gets lost tracking equally promissing paths. The good performance of SetA* is due to the low upper bound of the size of BDDs in the search queue. It focuses the search on a reasonable subset of the paths. Interestingly the search time is very low even for the hardest problems. Time and memory are spent on building and splitting the transition relation. Separating the next state computation and the splitting as done by the earlier version of SetA*, seems to come with a large performance penalty.

In the second experiment we measure the performance of SetA* for increasing upper bounds (u) of the size of BDDs in the search queue. The results are shown in Table 3 and were obtained for the $D^9V^4M^{15}$ problem using the same disjunctive partitioning as in the previous experiment.

u	#it	T (sec)	T_t (sec)	T_s (sec)
100	73	7.4	3.3	1.3
200	34	6.8	3.3	0.7
400	34	7.3	3.3	1.1
800	52	8.3	3.3	2.2
1600	51	10.1	3.3	4.0
3200	49	14.1	3.3	8.0
6400	49	24.4	3.3	17.7
12800	45	47.5	3.3	39.6
25600	42	110.4	3.3	102.8
51200	34	474.4	3.3	466.8

Table 3: Upper bound results for SetA* on the $D^9V^4M^{15}$ problem. u is the upper bound, #it is the number of iterations, T is the total CPU time, T_t is the time used to generate the improvement partitioning, and T_s is time used on search.

As depicted the performance degrades substantially for large values of u. The problem is that the sets of most promising states is large and have no compact BDD representation. By choosing a low u-value we focus on a subset of the most promising states in each iteration. As long as the problem has many solution paths this approach may work well. For $D^9V^4M^{15}$ this is reasonable to assume, since the sliders still are fairly independent. For highly dependent problems, however, a low u-value may lead to SetA* getting lost on wild paths.

Planning Problems

Like MIPS, the DOP planning system uses an approximation to the HSPr heuristic (Bonet & Geffner 1999) for STRIPS domains. In addition, it performs similar analysis to minimize the state encoding length. HSPr is an efficient but non-admissible heuristic. We approximate it by summing the depth $d(f)$ of each fact in a state given by a relaxed forward breadth-first search. The heuristic is applied in a backward search from the goal states to the initial state. For any action (pre, add, del) leading from S to S' (when applied in forward direction), we assume

$$del \subseteq pre \text{ and } add \not\subseteq pre.$$

Since the search is backward the improvement of the action is

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

Thus the improvement of an action can be computed without any BDD-based encoding of the heuristic function. Each action is partitioned in up to $2^{|add|}$ sets of transitions with different improvement.

x	SetA*		PreSetA*		A*		BiDir	
	#it	T (sec)	#it	T (sec)	#it	T (sec)	#it	T (sec)
1	34	0.6	34	0.8	34	1.1	34	0.7
2	34	0.7	34	0.9	34	1.1	34	0.7
3	34	0.6	34	1.4	34	1.1	34	1.6
4	34	0.6	34	1.5	34	1.1	34	8.1
5	34	0.6	34	3.5	34	1.0	34	334.0
6	34	0.8	34	14.4	-	TIME	-	TIME
7	34	1.3	34	39.8	-	TIME	-	TIME
8	34	2.1	34	50.7	-	TIME	-	TIME
9	94	6.8	34	202.6	-	TIME	-	TIME
10	58	16.3	34	297.2	-	TIME	-	TIME
11	34	39.3	-	TIME	-	TIME	-	TIME
12	-	MEM	-	TIME	-	TIME	-	TIME

Table 2: Results for the $D^x V^4 M^{15}$ problem. #it is the number of iterations, and T is the total CPU time.

The problems we consider, are *Gripper* from the STRIPS track of the AIPS-98 planning competition (Long 2000) and *Logistics* from the first round of the STRIPS track of the AIPS-00 planning competition (Bacchus 2001). The purpose of these experiments is to compare the performance of SetA* and BDDA*, not to solve the problems particularly fast. In that case, a more informative heuristic like the FF heuristic (Hoffmann 2001) should be applied.

Gripper This domain considers a robot with two grippers moving an increasing number of balls between two connected rooms. The first experiment compares forward BDD-based breadth-first search, SetA* with $w = 1.0$ and $u = \infty$, backward BDD-based breadth-first search, pure BDDA*, and BDDA*. Recall that Pure BDDA* performs best-first search like SetA* with $w = 1.0$. The results are shown in Table 4.

This domain is efficiently solved using blind BDD-based breadth-first search. The reason is that the search fringe grows only polynomially with the search depth. As is often observed both for planning and model checking problems, the best performance is obtained when searching forward. The performance of SetA* is almost as good as forward search even though, this algorithm relies on the slower backward expansion. BDDA* spends considerable time prior to search computing BDD formulas for arithmetic operations. During search the fringe expansion of BDDA* seems to degrade fast with the size of the fringe. Pure BDDA* on the other hand successfully completes a large number of iterations due to a lower growth rate of the fringe. All algorithms find shortest plans.

The second experiment shows the impact of the weight setting in problem 20. The results are shown in Table 5. Since the problem can be solved efficiently by blind BDD-based breadth-first search, it is not surprising that the weight setting turns out to be less important for the performance of SetA*.

Logistics This domain considers moving packages with trucks between sub-cities and with airplanes between cities. In the first experiment SetA* was run with $w = 1.0$ and $u = 200$. The upper bound of the size of partitions in the disjunctive partitioning was 400. The results are shown in Table 6.

w	#it	$\|p\|$	T (sec)	T_s (sec)
0.0	360	125	7.6	4.6
0.1	358	125	7.7	4.5
0.2	354	125	7.9	4.7
0.3	347	125	8.0	4.7
0.4	338	125	8.0	4.8
0.5	373	125	9.2	5.9
0.6	204	125	6.1	2.9
0.7	204	125	6.1	3.0
0.8	204	125	6.2	3.0
0.9	204	125	6.2	3.2
1.0	204	125	5.9	2.9

Table 5: Results of the second Gripper experiment. w is the weight, $|p|$ is the solution length, #it is the number of iterations, T is the total CPU time, and T_s is time used on search.

Due to the fact that there are no resource constraints in the Logistics domain, and thus no conflicts between subgoals, the HSPr heuristic is quite efficient. Both SetA* and Pure BDDA* search fast in this domain. However, Pure BDDA* and BDDA* have a significant overhead due to their precomputation of arithmetic formulas. For SetA* the upper bound on the size of the partitions in the disjunctive partitioning is crucial for the larger Logistics problems. In addition the upper bound on the size of BDDs in the search queue speeds up SetA* on the last five problems. The search fringe for blind BDD-based search blows up in both directions. The plans of SetA* are slightly longer than Pure BDDA*. The plans of BDDA* are shorter than both SetA* and Pure BDDA*, but only BDD-based breadth-first search finds optimal length plans.

The second experiment was carried out on problem 7 of the Logistics domain. In this experiment SetA* was run with

#p	$\|S\|$	Forward T (sec)	SetA* #it	SetA* T (sec)	SetA* T_s (sec)	Backward T (sec)	Pure BDDA* #it	Pure BDDA* T (sec)	Pure BDDA* T_s (sec)	BDDA* #it	BDDA* T (sec)	BDDA* T_s (sec)
1	2^{11}	0.1	14	0.1	0.0	0.1	22	3.0	0.0	14	2.8	0.0
2	2^{15}	0.1	24	0.1	0.0	0.1	62	4.0	0.1	22	3.9	0.1
3	2^{19}	0.2	34	0.2	0.1	0.3	138	5.5	0.5	30	5.3	0.4
4	2^{23}	0.2	44	0.3	0.1	0.6	250	8.0	1.8	38	7.1	1.2
5	2^{27}	0.3	54	0.5	0.1	1.0	398	12.9	5.5	46	10.3	3.2
6	2^{31}	0.4	64	0.6	0.2	1.4	582	22.4	13.6	54	15.4	7.0
7	2^{35}	0.6	74	0.7	0.2	2.1	802	40.4	29.5	62	25.2	15.5
8	2^{39}	0.9	84	1.0	0.4	2.9	1058	72.5	59.4	70	47.1	36.2
9	2^{43}	1.0	94	1.2	0.4	4.1	1350	137.4	120.7	78	123.5	111.8
10	2^{47}	1.2	104	1.4	0.5	5.3	1678	317.2	295.8	-	TIME	-
11	2^{51}	1.4	114	1.7	0.7	7.1	-	TIME	-	-	TIME	-
12	2^{55}	1.7	124	2.0	0.8	9.1	-	TIME	-	-	TIME	-
13	2^{59}	2.0	134	2.3	1.0	13.0	-	TIME	-	-	TIME	-
14	2^{63}	2.2	144	2.7	1.2	17.2	-	TIME	-	-	TIME	-
15	2^{67}	2.7	154	3.1	1.4	16.4	-	TIME	-	-	TIME	-
16	2^{71}	3.5	164	3.5	1.6	19.7	-	TIME	-	-	TIME	-
17	2^{75}	3.4	174	4.0	1.9	23.1	-	TIME	-	-	TIME	-
18	2^{79}	3.9	184	4.9	2.3	27.5	-	TIME	-	-	TIME	-
19	2^{83}	4.5	194	5.1	2.6	32.4	-	TIME	-	-	TIME	-
20	2^{87}	5.0	204	5.8	3.0	37.2	-	TIME	-	-	TIME	-

Table 4: Results of the first Gripper experiment. #p is the problem number, $|S|$ is the size of the state space, #it is the number of iterations, T is the total CPU time, and T_s is time used on search.

#p	$\|S\|$	SetA* #it	SetA* $\|P\|$	SetA* T (sec)	SetA* T_s (sec)	Pure BDDA* #it	Pure BDDA* $\|P\|$	Pure BDDA* T (sec)	Pure BDDA* T_s (sec)	Forward $\|P\|$	Forward T (sec)	BDDA* #it	BDDA* $\|P\|$	BDDA* T (sec)	BDDA* T_s (sec)
4	2^{21}	21	21	0.2	0.1	22	22	6.5	0.0	20	0.3	54	22	7.7	1.2
5	2^{21}	33	33	0.3	0.1	30	30	6.7	0.1	27	0.5	65	28	9.5	2.7
6	2^{21}	31	31	0.3	0.1	30	30	6.7	0.1	25	0.4	64	26	8.4	1.6
7	2^{41}	46	44	0.9	0.3	44	42	13.9	0.3	36	99.0	-	-	TIME	-
8	2^{41}	41	40	1.0	0.3	36	36	14.1	0.2	31	59.5	94	32	138.5	118.5
9	2^{41}	48	46	0.9	0.2	46	45	14.0	0.3	36	100.0	102	38	132.6	115.8
10	2^{54}	66	56	2.5	1.1	54	51	25.1	1.1	-	MEM	-	-	TIME	-
11	2^{54}	71	61	2.2	0.7	62	60	25.2	1.2	-	-	-	-	TIME	-
12	2^{54}	60	54	2.0	0.6	52	49	24.9	0.8	-	-	-	-	TIME	-
13	2^{86}	154	94	8.5	5.0	96	94	57.5	6.5	-	-	-	-	TIME	-
14	2^{86}	127	78	7.7	3.9	70	65	56.7	8.3	-	-	-	-	TIME	-
15	2^{86}	140	96	7.3	3.2	92	91	53.9	5.5	-	-	-	-	TIME	-

Table 6: Results of the first Logistics experiment. #p is the problem number, $|S|$ is the size of the state space, #it is the number of iterations, $|P|$ is the plan length, T is the total CPU time, and T_s is time used on search.

w	#it	$\|p\|$	T (sec)	T_s (sec)
0.0	279	25	8.6	7.9
0.1	248	25	9.0	8.3
0.2	203	25	8.9	8.1
0.3	154	25	7.9	7.1
0.4	102	25	4.7	4.0
0.5	180	27	2.3	1.6
0.6	49	29	0.9	0.1
0.7	31	31	0.8	0.1
0.8	31	31	0.8	0.1
0.9	31	31	0.8	0.1
1.0	31	31	0.9	0.1

Table 7: Results of the second Logistics experiment. w is the weight, $|p|$ is the solution length, #it is the number of iterations, T is the total CPU time, and T_s is time used on search.

$u = \infty$. The results are shown in Table 7. As depicted HSPr is a good heuristic for this domain increasing the speed significantly while preserving a relative high solution quality. Notice that the relaxation of the upper bound does not affect the performance of SetA for this problem.

Related Work

Directed 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, guiding techniques have been proposed to avoid a blow-up of intermediate BDDs (Bloem, Ravi, & Somenzi 2000). However these techniques are not applicable to search since they are based on defining lower and upper bounds on the fixed-point. Directed search techniques are relevant for *falsification* where the goal is to find a state not satisfying an invariant. The first work on BDD-based directed search, we are aware of, was for this application (Yang & Dill 1998). The proposed algorithm is a simple best-first search where the search fringe is partitioned with a specialized BDD-operator according to the Hamming distance to the goal state. Even though this operation is fairly efficient for the Hamming distance, it is not obvious how to define it in general.

As far as we know, the only previous BDD-based implementation of A* is BDDA*. BDDA* can use a general heuristic function and has been applied to planning as well as model checking. Similar to SetA*, it assumes unit-cost transitions and Boolean encoding of states. In contrast to SetA*, however, BDDA* requires arithmetic operations at the BDD level during search and includes no tools to control the growth of the search fringe or for cycle detection. In addition BDDA* is non-trivial to generalize to weighted A*.

The search queue in BDDA* is represented by a BDD $Open(s, f)$ that associates each state s in the queue with its f-value. Given a BDD encoding of the heuristic function $h(s, v)$ and a BDD encoding of the set of states with minimum f-value $Min(s)$, BDDA* expands all states with minimum f-value by computing

$$\begin{aligned} Open'(s,f) \quad \leftarrow \quad & \exists s'.\, Min(s) \wedge T(s,s') \wedge \\ & \exists e'.\, h(s',e') \wedge \exists e\,.\, h(s,e) \wedge \\ & (f = f_{min} + e - e' + 1). \end{aligned}$$

Since the BDDs representing the heuristic function and the transition relation often are large, a naive implementation of this computation would be very slow. The MIPS implementation of BDDA* seems to use another strategy where the largest possible subset of the computations are carried out prior to the search. However this strategy has not been described in the literature and as indicated by our experiments, it still leads to substantial performance degradation.

Conclusion and Outlook

In this paper, we have combined BDD-based search and heuristic search into a new search paradigm. The experimental evaluation of SetA* proves it a powerful algorithm often several orders of magnitude faster than BDD-based breadth-first search and A*. Today planning problems are efficiently solved by heuristic single-state search algorithms. However as recently noticed, the success may be due to an inherent simplicity of the benchmark domains when using the right heuristics (Hoffmann 2001). For less domain-tuned heuristics, we believe that the ability of SetA* to explore an exponential growing set of paths in polynomial time is essential. Our ongoing research includes identifying such problems and comparing the performance of SetA* and single-state search algorithms.

Acknowledgments

This research is sponsored in part by 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.

References

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

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

Bonet, B., and Geffner, H. 1999. Planning as heuristic search: New results. In *Proceedings of the European Conference on Planning (ECP-99)*. Springer.

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

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

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

Clarke, E.; Grumberg, O.; and Peled, D. 1999. *Model Checking*. MIT Press.

Edelkamp, S., and Helmert, M. 2001. MIPS the model-checking integrated planning system. *AI Magazine* 22(3):67–71.

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

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

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

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

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

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

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

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

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

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

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