Symbolic and Explicit Search Hybrid Through Perfect Hash Functions – A Case Study in Connect Four

Stefan Edelkamp
Institute for Artificial Intelligence
Universität Bremen

Peter Kissmann
Foundations of Artificial Intelligence Group
Universität des Saarlandes

Martha Rohte
Institute for Artificial Intelligence
Universität Bremen

Abstract

This work combines recent advances in AI planning under memory limitation, namely bitvector and symbolic search. Bitvector search assumes a bijective mapping between state and memory addresses, while symbolic search compactly represents state sets. The memory requirements vary with the structure of the problem to be solved. The integration of the two algorithms into one hybrid algorithm for strongly solving general games initiates a BDD-based solving algorithm, which consists of a forward computation of the reachable state set, possibly followed by a layered backward retrograde analysis. If the main memory becomes exhausted, it switches to explicit-state two-bit retrograde search. We use the classical game of *Connect Four* as a case study, and solve some instances of the problem space-efficiently with the proposed hybrid search algorithm.

Introduction

This work combines two recent advances in AI planning under memory limitation, namely planning with bitvectors and planning with BDDs (binary decision diagrams).

Planning with bitvectors (Cooperman and Finkelstein 1992) assumes a perfect hash function, in form of a one-to-one mapping between state and memory address indices, so that each state only takes a constant number of bits (the state itself is implicitly encoded in the memory address), while planning with BDDs compactly represents and progresses state sets (McMillan 1993). Bitvectors have been successfully applied in solving single-agent planning problems like *Pancake Flipping* (Korf 2008), and *Rubik's Cube*, for analyzing multi-agent games like *Chinese Checkers* (Sturtevant and Rutherford 2013), *Nine-Men-Morris* (Edelkamp, Sulewski, and Yücel 2010), and *Awari* (Romein and Bal 2002). They have also been employed for constructing endgames, e.g., in *Checkers* (Schaeffer et al. 2005), and for generating pattern databases (Breyer and Korf 2010; Sievers, Ortlieb, and Helmert 2012).

Symbolic planning has been successfully applied in exploring single-agent challenges (Edelkamp and Reffel 1998), cost-optimal tasks (Edelkamp and Kissmann 2009; Torralba, Edelkamp, and Kissmann 2013; Torralba and Alcázar 2013), for computing pattern databases (Edelkamp 2005) and for strongly solving *general games* (Edelkamp and Kissmann 2008; Kissmann and Edelkamp 2010).

When addressing practical needs in the applications, the memory requirements for planning based on a bitvector encoding of the state space, and for planning with BDDs based on a binary encoding of a state, vary with the structure of the problem to be solved. The sweet spot of efficient space usage is often located between the two extremes.

We will address the integration of the two algorithms into a single hybrid. We *strongly* solve the planning problem by computing what is sometimes referred to as the *universal plan*: an optimal policy (here in form of winning sets) that returns the best possible action for each state (this policy can be extracted to form an optimal controller.) Our setting is adversarial, where a protagonist plays a game against an antoganist. More specifically, we use the technique to strongly solve general games described in the game description language GDL (Love, Hinrichs, and Genesereth 2006).

First, we initiate a BDD-based planning algorithm, which consists of a forward computation of the reachable state set, followed by a layered backward retrograde analysis. For the exploration and solving we will apply the layered approach that is described in (Kissmann and Edelkamp 2010). After storing BDDs for each layer of the set of reachable states, the solving algorithm chains backward layer by layer, with decreasing distance to the initial state instead of increasing distance to the goal states. This way, the BDDs for all but the currently addressed layers reside on disk.

For the explicit-state space analysis we will use a bitvector-based retrograde analysis algorithm. The important contribution of this paper is the connection from the explicit to the symbolic representation obtained in the reachability analysis by applying efficient ranking and unranking with BDDs (Dietzfelbinger and Edelkamp 2009).

If the forward stage is completed or if main memory requirements become exhausted, the hybrid algorithm switches to backward bitvector search. The BDD representation of the forward-layer that is currently worked on in the retrograde analysis serves as a perfect hash function to address the index in the bitvector with the state and to retrieve and reconstruct it from the index.

We use *Connect Four* as a case study, and solve GDL instances of the problem space-efficiently with the proposed hybrid planning algorithm. Moreover, we predict the search

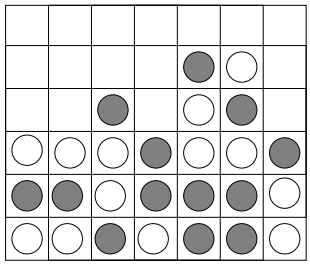

Figure 1: The game *Connect Four*: The player with the pieces shaded in gray has won.

efforts needed for *strongly* solving larger *Connect Four* problems. (The game was weakly solved von James Dow Allen and Victor Allis; and close-to-strongly solved by John Tromp.) We center around this problem, since memory limitations are *real* on current technology, and none of the existing algorithms can solve it. Nonetheless, the hybrid planning approach itself is domain-independent.

The paper is structured as follows. First, we motivate the problem of limited main memory capacity that we encountered, while trying to strongly solve the game of *Connect Four*. Next, we introduce planning with BDDs and with bitvectors and turn to ranking and unranking with BDDs. We then show how to combine the algorithms into one hybrid. Finally, we provide initial experiments in smaller instances of the *Connect Four* problem, predict the search efforts for solving larger instances, and discuss the results.

Case Study

Although most of the algorithms are applicable to several two-player games, our focus is on one particular case, namely the game *Connect Four* (see Fig. 1). The game (Allis 1988; Allen 2011) is played on a grid of c columns and r rows. In the classical setting we have $c = 7$ and $r = 6$. While the game is simple to follow and play, it can be rather challenging to win. This game is similar to *Tic-Tac-Toe*, with two main differences: The players must connect four of their pieces (horizontally, vertically, or diagonally) in order to win and gravity pulls the pieces always as far to the bottom of the chosen column as possible. The numbers of states for different settings of $c \times r$ are shown in Table 1.

BDD search can efficiently execute a breadth-first enumeration of the state space in (7×6) *Connect Four* (Edelkamp and Kissmann 2008). It has been formally shown that – while the reachable set leads to polynomially-sized BDDs – the symbolic representation of the termination criterion appears to be exponential (Edelkamp and Kissmann 2011). The set of all 4,531,985,219,092 reachable states can be found within four hours of computation, while explicit-state disk-based search took more than one year.

As illustrated in Table 2, of the 4,531,985,219,092 reachable states *only* 1,211,380,164,911 (about 26.72%) have been left unsolved in the layered BDD retrograde analysis. (More precisely, there are 1,265,297,048,241 states left unsolved by the algorithm, but the remaining set of 53,916,883,330 states in layer 30 is implied by the solvability status of the other states in the layer.) Even while providing space in form of 192 GB of RAM, however, it was not possible to proceed the symbolic solving algorithm to layers smaller than 30. The reason is that while the peak of the solution for the state sets has already been passed, the BDDs for representing the state sets are still growing.

This motivates looking at other options for memory-limited search and a hybrid approach that takes the symbolic information into account to eventually compute the complete solution of the problem.

BDDs for Strongly Solving Games

Binary decision diagrams (BDDs) are a memory-efficient data structure used to represent Boolean functions (Bryant 1986) as well as to perform set-based search (McMillan 1993). In short, a BDD is a directed acyclic graph with one root and two terminal nodes, the 0- and the 1-sink. Each internal node corresponds to a binary variable and has two successor eges, one (along the *Then*-edge) representing that the current variable is true (1) and the other (along the *Else*-edge) representing that it is false (0). For any assignment of the variables derived from a path from the root to the 1-sink the represented function will be evaluated to 1.

Bryant (1986) imposes a fixed variable ordering, for which he also provided two reduction rules (eliminating nodes with the same Then- and Else-edge and merging two nodes representing the same variable that share the same Then-edge as well as the same Else-edge). These BDDs are called *reduced ordered binary decision diagrams* (ROBDDs). Whenever we mention BDDs in this paper, we actually refer to ROBDDs. We also assume that the variable ordering is the same for all the BDDs and has been optimized prior to the search. BDDs have been shown to be very effective in the verification of systems, where BDD traversal is referred to as symbolic model checking (McMillan 1993). Adopting terminology to state space search, we are interested in the *image* of a state set S with respect to a transition relation *Trans*. The result is a characteristic function of all states reachable from the states in S in one step.

For the application of the image operator we need two sets of variables, one, x, representing the current state variables, another, x', representing the successor state variables. The image *Succ* of the state set S is then computed as $\mathit{Succ}(x') = \exists x\ (\mathit{Trans}(x, x')\ \wedge\ S(x))$. The *preimage Pre* of the state set S is computed as $\mathit{Pre}(x) = \exists x'\ (\mathit{Trans}(x, x')\ \wedge\ S(x'))$ and results in the set of predecessor states.

Using the image operator, implementing a layered symbolic breadth-first search (BFS) is straight-forward. All we need to do is to apply the image operator to the initial state resulting in the first layer, then apply the image operator to the first layer resulting in the second and so on. The search ends when no successor states can be found. General games (and in this case, *Connect Four*) are guaranteed to terminate after a finite number of steps, so that the forward search will eventually terminate as well.

For strongly solving two-player games (Kissmann and Edelkamp 2010), we find all the reachable states by performing layered symbolic BFS, storing all layers separately. The

Table 1: Reachable states in *Connect Four* Variants.

l	7×6	6×6	6×5	5×6	5×5
0	1	1	1	1	1
1	7	6	6	5	5
2	49	36	36	25	25
3	238	156	156	95	95
4	1,120	651	651	345	345
5	4,263	2,256	2,256	1,075	1,075
6	16,422	7,876	7,870	3,355	3,350
7	54,859	24,330	24,120	9,495	9,355
8	184,275	74,922	72,312	26,480	25,060
9	558,186	211,042	194,122	68,602	60,842
10	1,662,623	576,266	502,058	169,107	139,632
11	4,568,683	1,468,114	1,202,338	394,032	299,764
12	12,236,101	3,596,076	2,734,506	866,916	596,136
13	30,929,111	8,394,784	5,868,640	1,836,560	1,128,408
14	75,437,595	18,629,174	11,812,224	3,620,237	1,948,956
15	176,541,259	39,979,044	22,771,514	6,955,925	3,231,341
16	394,591,391	80,684,814	40,496,484	12,286,909	4,769,837
17	858,218,743	159,433,890	69,753,028	21,344,079	6,789,890
18	1,763,883,894	292,803,624	108,862,608	33,562,334	8,396,345
19	3,568,259,802	531,045,746	165,943,600	51,966,652	9,955,530
20	6,746,155,945	884,124,974	224,098,249	71,726,433	9,812,925
21	12,673,345,045	1,463,364,020	296,344,032	97,556,959	9,020,543
22	22,010,823,988	2,196,180,492	338,749,998	116,176,690	6,632,480
23	38,263,228,189	3,286,589,804	378,092,536	134,736,003	4,345,913
24	60,830,813,459	4,398,259,442	352,607,428	132,834,750	2,011,598
25	97,266,114,959	5,862,955,926	314,710,752	124,251,351	584,249
26	140,728,569,039	6,891,603,916	224,395,452	97,021,801	
27	205,289,508,055	8,034,014,154	149,076,078	70,647,088	
28	268,057,611,944	8,106,160,185	74,046,977	40,708,770	
29	352,626,845,666	7,994,700,764	30,162,078	19,932,896	
30	410,378,505,447	6,636,410,522	6,440,532	5,629,467	
31	479,206,477,733	5,261,162,538			
32	488,906,447,183	3,435,759,942			
33	496,636,890,702	2,095,299,732			
34	433,471,730,336	998,252,492			
35	370,947,887,723	401,230,354			
36	266,313,901,222	90,026,720			
37	183,615,682,381				
38	104,004,465,349				
39	55,156,010,773				
40	22,695,896,495				
41	7,811,825,938				
42	1,459,332,899				
Σ	4,531,985,219,092	69,212,342,175	2,818,972,642	1,044,334,437	69,763,700

solving starts in the last reached layer and performs regression search towards the initial state, which resides in layer 0. The last reached layer contains only terminal states (otherwise the forward search would have progressed farther), which can be solved immediately by calculating the conjunction with the BDDs representing the rewards for the two players. Once this is done, the search continues in the preceding layer. If another layer contains terminal states as well, these are solved in the same manner before continuing with the remaining states of that layer. The rewards are handled in a certain order: In case of a zero-sum game the order is always according to the highest reward of the active player, modeling the *MiniMax* procedure (von Neumann and Morgenstern 1944). All the solved states of the successor layer are loaded in this order and the preimage is calculated, which results in those states of the current layer that will achieve the same rewards, so that they can be stored on the disk as well. Once the initial state is solved and stored the strong solution resides completely on the hard disk.

Ranking and Unranking with BDDs

Ranking is a minimal perfect hash function from the set of satisfying assignments to the position of it in the lexicographical ordering of all satisfying assignments. *Unranking* is the inverse operation. In other words, *rank* is a unique number of a state and the inverse process of *unranking* reconstructs the state given its rank. Perfect hash functions to efficiently rank and unrank states have been shown to be very successful in traversing single-player problems like *Rubik's Cube* or the *Pancake Problem* (Korf 2008) or two-player games like *Awari* (Romein and Bal 2002). They are also used for creating pattern databases (Breyer and Korf 2010). The problem of the construction of these perfect hash functions is that they are domain dependent.

The domain-independent approach exploited in this paper builds on top of findings by Dietzfelbinger and Edelkamp

Table 2: Result of symbolic retrograde analysis (excl. terminal goals; l is the layer).

l	nodes (won)	states (won)	nodes (draw)	states (draw)	nodes (lost)	states (lost)
⋮	⋮	⋮	⋮	⋮	⋮	⋮
29	o.o.m.	o.o.m.	o.o.m.	o.o.m.	o.o.m.	o.o.m.
30	589,818,676	199,698,237,436	442,186,667	6,071,049,190	o.o.m.	o.o.m.
31	458,334,850	64,575,211,590	391,835,510	7,481,813,611	600,184,350	201,906,000,786
32	434,712,475	221,858,140,210	329,128,230	9,048,082,187	431,635,078	57,701,213,064
33	296,171,698	59,055,227,990	265,790,497	10,381,952,902	407,772,871	194,705,107,378
34	269,914,837	180,530,409,295	204,879,421	11,668,229,290	255,030,652	45,845,152,952
35	158,392,456	37,941,816,854	151,396,255	12,225,240,861	231,007,885	132,714,989,361
36	140,866,642	98,839,977,654	106,870,288	12,431,825,174	121,562,152	24,027,994,344
37	68,384,931	14,174,513,115	72,503,659	11,509,102,126	105,342,224	57,747,247,782
38	58,428,179	32,161,409,500	44,463,367	10,220,085,105	42,722,598	6,906,069,443
39	19,660,468	2,395,524,395	27,201,091	7,792,641,079	35,022,531	13,697,133,737
40	17,499,402	4,831,822,472	13,858,002	5,153,271,363	8,233,719	738,628,818
41	0	0	5,994,843	2,496,557,393	7,059,429	1,033,139,763
42	0	0	0	0	0	0

(2009), who illustrated that, after some preprocessing, un/ranking of states in a state set represented as a BDD is as efficient as a hash table lookup. One obvious application was the uniform random choice of satisfying assignments.

In our case, BDD ranking aims at the symbolic equivalent of constructing a perfect hash function for fully enumerated state sets (Botelho, Pagh, and Ziviani 2007). For such perfect hash functions, the underlying state set to be hashed is generated in advance. This is plausible when computing strong solutions to problems, where we are interested in the game-theoretical value of all reachable states. Applications are planning tasks where the problem to be solved is harder than computing the reachability set.

For ranking and unranking satcount values are precomputed at each node of the BDD. Roughly speaking, the rank is computed by adding all satcount values at Else-edges adjacent to nodes in the path from root to sink that corresponds to the assignment. While unranking, the assignment is reconstructed from the number by subtracting these values from top to bottom and following the Then-edges in this case. Based on the lack of node indices when chaining down from top to bottom (due to BDD reduction rules), the actual implementation is tricky and requires access to the binary value of the omitted BDD node indices.

Let n be the size of the binary state vector. We now describe an $O(n)$ ranking and unranking scheme on top of precomputed satcount values in more detail. With negation on edges there are subtle problems to be resolved for storing the satcount values that we do not dwell on.

The *index*(n) of a BDD node n is its unique position in the shared representation and *level*(n) its position in the variable ordering. Moreover, we assume the 1-sink to have index 1 and the 0-sink to have index 0. Let $C_f = |\{a \in \{0,1\}^n \mid f(a) = 1\}|$ denote the number of satisfying assignments (*satcount*, here also sc for short) of f. With *bin* (and *invbin*) we denote the *conversion* of the binary value of a bitvector (and its inverse). The *rank* of a satisfying assignment $a \in \{0,1\}^n$ is the position in the lexicographical ordering of all satisfying assignments, while the *unranking* of a number r in $\{0,\ldots,C_f-1\}$ is its inverse.

Figure 2 shows the ranking and unranking functions in

```
rank(s)
  i = level(root);
  d = bin(s[0..i-1]);
  return d*sc(root) + rankAux(root,s) - 1;


rankAux(n, s)
  if (n <= 1) return n;
  i = level(n);
  j = level(Else(n));
  k = level(Then(n));
  if (s[i] == 0)
    return bin(s[i+1..j-1]) * sc(Else(n))
      + rankAux(Else(n),s);
  else
    return 2^(j-i-1) * sc(Else(n))
      + bin(s[i+1..k-1]) * sc(Then(n))
      + rankAux(Then(n),s);


unrank(r)
  i = level(root);
  d = r / sc(root);
  s[0..i-1] = invbin(d);
  n = root;
  while (n > 1)
    r = r mod sc(n);
    j = level(Else(n));
    k = level(Then(n));
    if (r < (2^(j-i-1) * sc(Else(n))))
      s[i] = 0;
      d = r / sc(Else(n));
      s[i+1..j-1] = invbin(d);
      n = Else(n);
      i = j;
    else
      s[i] = 1;
      r = r - (2^(j-i-1) * sc(Else(n)));
      d = r / sc(Then(n));
      s[i+1..k-1] = invbin(d);
      n = Then(n);
      i = k;
  return s;
```

Figure 2: Ranking and unranking of states.

pseudo-code. The procedures determine the rank given a satisfying assignment and vice versa. They access the satcount values on the Else-successor of each node (adding for the ranking and subtracting in the unranking). Missing nodes (due to BDD reduction) have to be accounted for by their binary representation, i.e., gaps of l missing nodes are accounted for 2^l. While the ranking procedure is recursive the unranking procedure is not.

The satcount values of all BDD nodes are precomputed

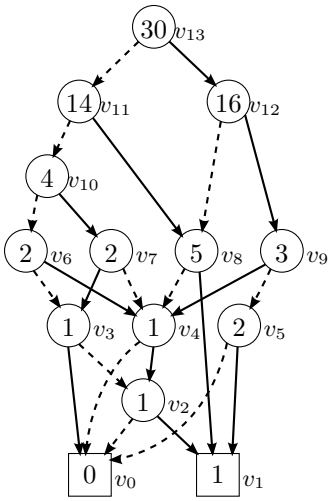

Figure 3: BDD for the ranking and unranking examples. Dashed arrows denote Else-edges; solid ones Then-edges. The numbers in the nodes correspond to the satcount. Each v_i denotes the index (i) of the corresponding node.

and stored along with the nodes. As BDDs are reduced, not all variables on a path are present but need to be accounted for in the satcount procedure. The time (and space) complexity of it is $O(|G_f|)$, where $|G_f|$ is the number of nodes of the BDD G_f representing f. With the precomputed values, rank and unrank both require linear time $O(n)$, where n is the number of variables in the function represented in the BDD. Dietzfelbinger and Edelkamp (2009) provide invariances showing that the procedures work correctly.

Ranking and Unranking Examples

To illustrate the ranking and unranking procedures, take the example BDD given in Figure 3. Assume we want to calculate the rank of state $s = 110011$. The rank of s is then $rank(s) = 0 + rA(v_{13}, s) - 1$

$$
\begin{aligned}
&= (2^{1-0-1} \cdot sc(v_{11}) + 0 + rA(v_{16}, s)) - 1 \\
&= sc(v_{11}) + \\
&\quad (2^{3-1-1} \cdot sc(v_8) + bin(0) \cdot sc(v_9) + rA(v_9, s)) - 1 \\
&= sc(v_{11}) + 2sc(v_8) + (0 + rA(v_5, s)) - 1 \\
&= sc(v_{11}) + 2sc(v_8) + \\
&\quad (2^{6-4-1} \cdot sc(v_0) + bin(1) \cdot sc(v_1) + rA(v_1, s)) - 1 \\
&= sc(v_{11}) + 2sc(v_8) + 2sc(v_0) + sc(v_1) + 1 - 1 \\
&= 14 + 2 \cdot 5 + 2 \cdot 0 + 1 + 1 - 1 = 25
\end{aligned}
$$

with $rA(s, v_i)$ being the recursive call of the rankAux function for state s in node v_i and $sc(v_i)$ the satcount stored in node v_i.

For unranking the state with index 19 ($r = 19$) from the BDD depicted in Figure 3 we get:

- $i = 0, n = v_{13}$: $r = 19 \mod sc(v_{13}) = 19 \mod 30 = 19 \not< 2^{1-0-1} sc(v_{11}) = 14$, thus $s[0] = 1$; $r = r - 2^{1-0-1} sc(v_{11}) = 19 - 14 = 5$
- $i = 1, n = v_{12}$: $r = 5 \mod sc(v_{12}) = 5 \mod 16 = 5 < 2^{3-1-1} sc(v_8) = 2 \cdot 5 = 10$, thus $s[1] = 0$; $s[2] = invbin(r/sc(v_8)) = invbin(5/5) = 1$
- $i = 3, n = v_8$: $r = 5 \mod sc(v_8) = 5 \mod 5 = 0 < 2^{4-3-1} sc(v_4) = 1$, thus $s[3] = 0$
- $i = 4, n = v_4$: $r = 0 \mod sc(v_4) = 0 \mod 1 = 0 \not< 2^{6-4-1} sc(v_0) = 0$, thus $s[4] = 1$; $r = r - 2^{6-4-1} sc(v_0) = 0 - 0 = 0$
- $i = 5, n = v_2$: $r = 0 \mod sc(v_2) = 0 \mod 1 = 0 \not< 2^{7-6-1} sc(v_{12}) = 0$, thus $s[5] = 1$; $r = r - 2^{7-6-1} sc(v_{12}) = 0 - 0$
- $i = 6; n = v_1$: return s ($= 101011$)

Retrograde Analysis on a Bitvector

Two-bit breadth-first search has first been used to enumerate so-called *Cayley Graphs* (Cooperman and Finkelstein 1992). As a subsequent result the authors proved an upper bound to solve every possible configuration of *Rubik's Cube* (Kunkle and Cooperman 2007). By performing a breadth-first search over subsets of configurations in 63 hours together with the help of 128 processor cores and 7 TB of disk space it was shown that 26 moves always suffice to unscramble it. Korf (2008) has applied the two-bit breadth-first search algorithm to generate the state spaces for hard instances of the *Pancake* problem I/O-efficiently.

In the two-bit breadth-first search algorithm every state is expanded at most once. The two bits encode values in $\{0, \ldots, 3\}$ with value 3 representing an unvisited state, and values 0, 1, or 2 denoting the current search depth *mod* 3. This allows to distinguish generated and visited states from ones expanded in the current breadth-first layer.

Adaptation to Our Setting

In our implementation (see Algorithm 4) we also use two bits, but with a different meaning. We apply the algorithm to solve two-player zero-sum games where the outcomes are only won/lost/drawn from the starting player's point of view. This is reflected in the interpretation of the two bits: Value 0 means that the state has not yet been evaluated; value 1 means it is won by the starting player (the player with index 0); value 2 means it is won by the player with index 1; value 3 means it is drawn. Retrograde analysis solves the entire set of positions in backward direction, starting from won and lost terminal ones. Bit-state retrograde analysis applies backward BFS starting from the states already decided.

For the sake of simplicity, the rank and unrank functions are both context-sensitive wrt. the layer of the search in which the operations take place. In the implementation we use BDDs for the different layers.

The algorithm assumes a maximal number of moves, that terminal drawn states appear only in the last layer (as is the case in *Connect Four*; extension to different settings is possible), that the game is turn-taking, and that the player can be found in the encoding of the game. It takes as input a decision procedure for determining whether a situation is *won* by one of the players as well as the index of the last reached

```
retrograde(won, maxlayers)
for layer in maxlayers,...,0
  m = sc(bdd(layer))
  for i in 0,...,m − 1
    B[layer][i] = 0
  for i in 0,...,m − 1
    state = unrank(i)
    if (won(state))
      if (layer mod 2 == 1)
        B[layer][i] = 1
      else
        B[layer][i] = 2
    else if (layer == maxlayer)
      B[layer][i] = 3
    else
      succs = expand(state)
      process(succs)


process (succs)
  if (layer mod 2 == 1)
    for all s in succs
      if B[layer+1][rank(s)] == 2
        B[layer][rank(i)] = 2
        break
      else if (B[layer+1][rank(s)] == 3)
        B[layer][rank(i)] = 3
    if (B[layer][rank(i)] == 0)
      B[layer][rank(i)] = 1
  else
    for all s in succs
      if B[layer+1][rank(s)] == 1
        B[layer][rank(i)] = 1
        break
      else if (B[layer+1][rank(s)] == 3)
        B[layer][rank(i)] = 3
    if (B[layer][rank(i)] == 0)
      B[layer][rank(i)] = 2
```

Figure 4: Retrograde analysis with bits for two-player zero-sum game (ranking is sensitive to the layer it is called in).

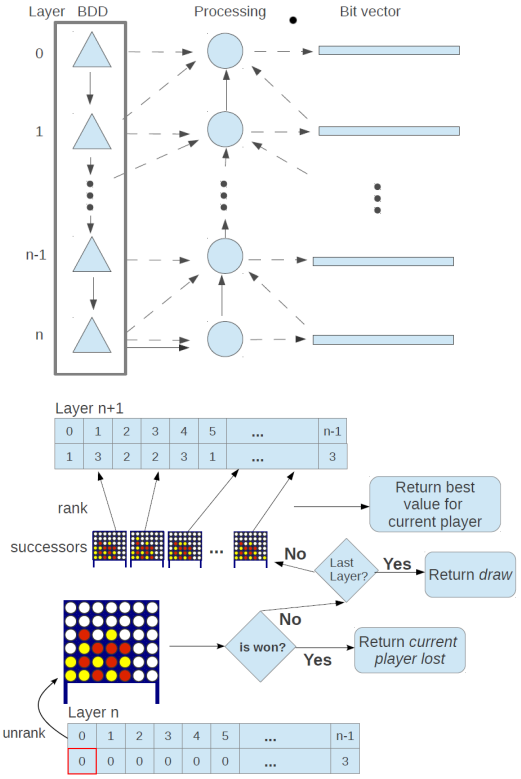

Figure 5: Hybrid algorithm: Visualization of data flow in the strong solution process (top). Processing a layer in the retrograde analysis (bottom).

layer (*maxlayer*). Starting at the final layer, it iterates toward the initial state residing in layer 0.

For each layer, it first of all determines the number of states. Next it sets all values of the vector B for the states in the current layer to 0 – not yet solved. Then it iterates over all states in that layer.

It takes one state (by unranking it from the layer), checks whether it is won by one of the players. If so, it can be solved correspondingly (setting its value to either 1 or 2). Otherwise, if it resides in the final layer, it must be a drawn state (value 3). In case neither holds, we calculate the state's successors. For each successor we check whether it is won by the currently active player, which is determined by checking the current layer's index. In this case the state is assigned the same value and we continue with the next state. Otherwise if the successor is drawn, the value of this state is set to draw as well. In the end, if the state is still unsolved that means that all successors are won by the opponent, so that the corresponding value is assigned to this state as well.

Hybrid Algorithm

The hybrid algorithm combines the two precursing approaches. It generates the state space with symbolic forward search on disk and subsequently applies explicit-state retrograde analysis based on the results in form of the BDD encoded layers read from disk.

Figure 5 illustrates the strong solution process. On the right hand side we see the bitvector used in retrograde analysis and on the left hand side we see the BDD generated in forward search and used in backward search.

The process of solving one layer is depicted in Figure 5 (right). While the bitvector in the layer n (shown at the bottom of the figure) is scanned and states within the layer are unranked and expanded, existing information on the solvability status of ranked successor states in the subsequent layer $n + 1$ is retrieved.

Ranking and unranking wrt. the BDD is executed to look up the status (won/lost/drawn) of a node in the set of successors. We observed that there is a trade-off for evaluating immediate termination. There are two options, one is procedural by evaluating the goal condition directly on the explicit state, the other is a dictionary lookup by traversing the corresponding reward BDD. In *Connect Four* the latter was not only more general but also faster. A third option would be to determine if there are any successors and set the rewards according to the current layer (as done in the pseudo-code).

To increase the exploration performance of the system we distributed the explicit-state solving algorithms on multiple CPU cores. We divide the bitvector for the layer to be solved into equally-sized chunks. The bitvector for the next layer is shared among all the threads.

For the ease of implementation, we duplicate the query BDDs for each individual core. This is unfortunate, as we

only use concurrent read in the BDD for evaluating the perfect hash function but the computation of the rank involves setting and reading local variables and requires significant changes in the BDD package to be organized lock-free. There are distributed usages of BDD libraries, e.g., reported by Christian Stangier documented in the CUDD files, but – up to our knowledge – there is no currently available multi-core version. Recent research work shows some steps into that direction (van Dijk, Laarman, and van de Pol 2012), but the status is far from being at library use.

Compared to the size of the bitvector the BDDs for the reachability layers are considerably smaller, so that we can afford re-reading the BDDs for each running thread.

Our prototype currently works with reachability sets from forward symbolic search and a complete explicit-state retrograde analysis. In a subsequent refinement of the implementation the hybrid search algorithm can be extended to start solving using symbolic search and then switch over to explicit search using the stored solutions if the BDD solving cannot finish within certain amounts of RAM.

Experiments

Experiments other than in the motivating section have been conducted on a Desktop PC with an Intel i7(R)Core(TM)-920 CPU (with 2.66 GHz), 24 GB RAM, with four hyper-threaded cores. RAM mainly limits the maximum number of elements we could handle (operating system: Ubuntu Linux; Compiler: GNU's `g++` optimized with flag `-O3`). For multi-core parallelization we used pthreads. The BDDs for the (7×6) *Connect Four* have been generated on a much larger cluster computer, running a single Java program via a native interface to CUDD that was compiled and optimized on 64 bits with GNU's `g++`. The machine used in the infeasibility argument of the 7×6 problem was with wrt. Intel Xeon X5690@3.47GHZ, 192GB RAM.

The results of the solving of (5×5) *Connect Four* are shown in Table 3. We see that more cores are helpful to reduce the running time significantly. Moreover, the performance for the three strategies to elevate the goal conditions varies only a little. By a small margin, the direct evaluation that has the lowest memory requirements is best.

For the (5×5) case using a bitvector the total space requirements are 17 MB, while the BDDs take 28 MB. However, when loading the data in RAM we also need the reachability sets in form of a BDD taking 5.4 MB, and the goal BDDs taking 4.7 MB. All in all, we obtain memory needs in the order of 22.4 MB, which still documents a possible saving in memory. The last 4.7 MB could be saved by not using the goal BDDs, but rather by evaluating the condition explicitly.

For solving (6×5) *Connect Four* the results are depicted in Table 4. Again, we see that more cores clearly reduce runtime (with two cores by a factor of rougly two; with eight cores by a factor of roughly four – note that our CPU uses Hyperthreading, i.e., it has only four physical cores, so the speedup seems to be linear in the number of used cores). Concerning the goal evaluations, we can see that again the direct evaluation is a bit faster. The third criterion was not tested because it was expected to take more space without any significant speed-up.

Table 3: Results for (5×5) *Connect Four* with 69,763,699 states, different termination criteria, single- and multi-core.

Algorithm	Time	States/Sec
1 Core Direct Goal Evaluation	283s	246,514
8 Core Direct Goal Evaluation	123s	567,184
1 Core BDD Goal	291s	239,737
8 Core BDD Goal	131s	532,547
1 Core No Succ. BDD	288s	242,235
8 Core No Succ. BDD	127s	549,320

Table 4: Results for (6×5) *Connect Four* with 2,938,430,579 states, different termination criteria, single- and multi-core.

Algorithm	Time	States/Sec
1 Core Direct Goal Evaluation	14,197s	206,975
2 Core Direct Goal Evaluation	7,540s	389,712
8 Core Direct Goal Evaluation	3,510s	837,159
1 Core No Succ. BDD	14,944s	196,629
2 Core No Succ. BDD	7,665s	383,356
8 Core No Succ. BDD	3,600s	816,230

Table 5 gives some insight into the actual sizes required by the BDDs and the bitvectors in the solving of (6×5) *Connect Four*. For each forward layer we provide the number of states in that layer, the number of BDD nodes needed to represent them, the size of the BDD representation of all states in that layer (assuming that each node requires 20 bytes), the size of the corresponding bitvector as well as a theoretical estimation of the memory required to solve that layer with the hybrid approach in case of a single-core based setting. The required memory is simply the sum of the BDD sizes of the current and the successor layer (both must be loaded for the ranking and unranking in the algorithm) as well as the sizes of the bitvectors (both must be loaded for the actual solving). Performing pure symbolic search we arrive at a peak node count of 136,001,819. Assuming the same node size of 20 bytes this corresponds to roughly 2.5 GB. Seeing that the highest required size in the hybrid approach is 254 MB the savings become apparent.

Note that the column mem_{req} of Table 5 refers to theortical values under ideal circumstances of a static BDD package allocating only space for the BDD nodes that appear on disk. The BDD package in use, however, has its own memory pool implementation, and, therefore, a significant overhead. Hence, we also computed the practical values (by analyzing the output of the Unix command top after finalizing a layer). E.g., in layer 23 of the retrograde classification algorithm, we observed the maximal real peak memory requirements of 390 MB (VIRT) and 287 MB (RES)[1].

The CPU time used to process all 2,938,430,579 states of

[1] By inspecting values from top in between two consecutive layers, we detected slightly higher intermediate RAM requirements of 351 MB (RES).

Table 5: Space consumption computing (6×5) *Connect Four*. l is the layer, s the number of states in that layer, n the number of BDD nodes needed to represent it, $size_{bdd}$ the size of the corresponding BDD (assuming 20 Bytes per node), $size_{bv}$ the size of the corresponding bitvector, and mem_{req} the memory needed for solving a layer (i.e., loading the current and successor layer's bitvectors and BDDs) in a single-core setting.

l	s	n	$size_{bdd}$	$size_{bv}$	mem_{req}
0	1	61	1.2KB	1B	3.5KB
1	6	116	2.3KB	2B	6.6KB
2	36	223	4.4KB	9B	12KB
3	156	366	7.2KB	39B	20KB
4	651	637	13KB	163B	34KB
5	2,256	1,080	21KB	564B	57KB
6	7,870	1,702	33KB	1.9KB	96KB
7	24,120	2,793	55KB	5.9KB	171KB
8	72,312	4,772	93KB	18KB	305KB
9	194,122	7,498	146KB	47KB	526KB
10	502,058	10,722	209KB	123KB	964KB
11	1,202,338	17,316	338KB	294KB	1.8MB
12	2,734,506	25,987	508KB	668KB	3.4MB
13	5,868,640	43,898	857KB	1.4MB	6.4MB
14	11,812,224	68,223	1.3MB	2.8MB	12MB
15	22,771,514	122,322	2.3MB	5.4MB	21MB
16	40,496,484	187,493	3.6MB	9.7MB	36MB
17	69,753,028	327,553	6.2MB	17MB	58MB
18	108,862,608	475,887	9.1MB	26MB	89MB
19	165,943,600	769,944	15MB	40MB	127MB
20	224,098,249	1,004,398	19MB	53MB	171MB
21	296,344,032	1,437,885	27MB	71MB	210MB
22	338,749,998	1,656,510	32MB	81MB	242MB
23	378,092,536	2,080,932	40MB	90MB	254MB
24	352,607,428	2,123,251	41MB	84MB	243MB
25	314,710,752	2,294,960	44MB	75MB	210MB
26	224,395,452	2,004,090	38MB	54MB	162MB
27	149,076,078	1,814,442	35MB	36MB	111MB
28	74,046,977	1,257,586	24MB	18MB	64MB
29	30,162,078	789,650	15MB	7.2MB	29MB
30	6,440,532	282,339	5.4MB	1.5MB	6.9MB

Table 6: Classification of all states in won, draw and lost in (6×6) *Connect Four*.

l	won(black)	draw	won(white)
0	1	0	0
1	6	0	0
2	6	24	6
3	98	52	6
4	131	220	300
5	1,324	534	398
6	1,752	1,580	4,544
7	13,868	3,982	6,480
8	18,640	10,280	46,002
9	118,724	25,104	67,214
10	156,360	56,710	363,196
11	815,366	129,592	523,156
12	1,050,857	267,636	2,277,583
13	4,597,758	565,760	3,231,266
14	5,831,790	1,098,276	11,699,108
15	21,523,754	2,144,618	16,310,672
16	27,021,039	3,911,893	49,751,882
17	83,960,708	7,060,426	68,412,756
18	104,937,956	12,096,840	175,768,828
19	272,162,860	20,210,438	238,672,448
20	339,135,354	32,320,349	512,669,271
21	725,182,660	50,189,136	687,992,224
22	901,278,168	75,033,304	1,219,869,020
23	1,561,655,780	108,518,894	1,616,415,130
24	1,929,105,096	150,351,002	2,318,803,344
25	2,645,054,112	202,034,082	3,015,867,732
26	3,223,332,998	259,072,600	3,409,198,318
27	3,392,753,538	322,390,736	4,318,869,880
28	4,030,760,404	384,569,265	3,690,830,516
29	3,089,884,946	435,398,174	4,469,417,644
30	3,476,328,802	471,148,650	2,688,933,070
31	1,785,392,828	468,490,136	3,007,279,574
32	1,841,291,613	450,464,011	1,144,004,318
33	554,598,782	373,041,874	1,167,659,076
34	502,035,320	276,221,222	219,995,950
35	54,244,612	149,951,066	197,034,676
36	40,044,990	49,981,730	0

(6×5) *Connect Four* is about 3,500 seconds. That's about 800,000 states per second on average.

On one core, the (6×6) exploration finished in 958,283 seconds, or approx. 11 days. It generated about 72,184.34 states/second. The peak of the real memory requirements was encountered in layers 29 and 28 with 6.5 GB. At the end of the exploration, a 14.5 GB-sized strong solution bitvector database was computed and flushed to disk. The outcome is that the second player wins, validating published results (e.g., by Tromp). Table 6 shows the final classification result. On our machine we could not finalize the BDD-based solving due to the limited amount of RAM.

Investing 2 Bits per state for (6×7) *Connect Four* results in $2 \cdot 4{,}531{,}985{,}219{,}092$ Bits or more than 1 TB of RAM. Even for a layered retrograde analysis our computer infrastructure is not sufficient to provide sufficient amount of RAM to finalize the solving experiment for the (7×6) *Connect Four* instance. In layers 32/33 the bitvectors alone would occupy $2 \cdot (488{,}906{,}447{,}183 + 496{,}636{,}890{,}702)$ Bits ≈ 229.5 GB. Therefore, we would better start with the partial classification result of Table 2. Assuming the solving speed for the remaining unclassified states goes down to about 200,000 states per second for solving the entire layer 29 would take about $352{,}626{,}845{,}666/200{,}000/60^2 \approx 488$ hours, or little more than 20 days. For the remaining total of $1{,}265{,}297{,}048{,}241$ states to be solved we estimate a CPU search time of (at least) 1,752 hours or 73 days. In layer 30, the BDD for the reachable states and the two BDDs computed for the solution consume disk space in the order of a few GB only but still take several hours to be loaded. Investing two bits for each state in layer 29 requires $2 \cdot 352{,}626{,}845{,}666$ Bits ≈ 82.1 GB. Therefore, it may be possible for the hybrid search algorithm in this paper to finalize the entire solution process within 192 GB, while the symbolic algorithm could not finish it.

Therefore, we looked at the memory profile of smaller *Connect Four* variants. In all cases, we could show savings in RAM in trade-off with a – still acceptable – slow-down in the run-time behavior: e.g., the entire BDD exploration for (6×5) (running on a single core of a modern PC) took about 40m, while the hybrid exploration (running on a multi-

core PC) lasted for about one hour. At the edge of RAM sparse memory algorithms certainly become faster, due to paging. Even clever I/O-efficient algorithm designs often show a larger trade-off.

Conclusion and Discussion

We introduced a hybrid planner for two-player general games and showed in the case study of *Connect Four* that memory savings are indeed possible. We predicted the space and time efforts needed to finalize the solution for (6×7) *Connect Four* to give a feasibility assessment on the strong solvability of the game on current technology. Based on the expected significant resources in running time and RAM usage for the remaining solution, however, we presented experimental results only for smaller *Connect Four* instances.

Memory-limitation often is a more severe problem to planning algorithms than computation time. In this paper we have combined two promising approaches to cope with the problem within main memory, namely the compact representation of state sets in form of a BDD and the implicit representation of states by main memory addresses. Depending on the problem at hand, the one or the other can be advantageous. For cases like *Connect Four*, it is also the case that the memory profile for the one or the other is better in different parts of the search, so that we addressed the problem of how to change from one representation to the other.

For the approach to work, we expect the state sets of the reachability analysis (computed with BDDs) to be available, which for finding one step-optimal plan appears to be an overkill. For many cases of planning, however, it is more difficult to solve the planning problem than generating the plan space, e.g., for synthesizing an optimal controller in form of a universal plan. With the use of ADDs, for finding the optimal value function in MDPs similar problems arise. We also expect progress in the area of automata-based model checking, where (accepting) cycles, rather than simple paths, have to be found in the state space graph.

The bridge between the explicit-state and symbolic memory-limited search is the design of a linear-time perfect hash function based on the BDD representation of the state sets encountered. Generalizing from the studied case of game playing and strongly solving *Connect Four* the approach is rather fundamental: the approach can be interpreted as being the symbolic equivalent to (minimum) perfect hashing for explicit state sets (Botelho, Pagh, and Ziviani 2007; Botelho and Ziviani 2007). An external-memory model checker exploiting the exploration based on such perfect hash functions (Edelkamp, Sanders, and Simecek 2008) has been shown to be effective and is able to find minimal counter-examples (Edelkamp et al. 2011).

As the binary state encoding is found in the SAS$^+$ encoding file that is generated when grounding the problem the generality of the approach in our implementation is mainly affected by two aspects. The one is the so-called layered approach, which allow to externalize the backward computation wrt. BFS layers established in forward search. For this to work, an incremental progress measure has to be provided. Fortunately, there are many other games that include such a progress measure. In general games this measure is often imposed by providing a step counter. The other domain-specific aspect are some goal test options. From the offered choices for the goal test, the more general one has to be based on the goal state set(s) represented as a BDD rather than a specific test function.

The approach in its current form is not applicable to solve the history problem (of cyclic and path-dependent behavior) that is apparent in games like Chess and Checkers. Unfortunately, our input format GDL cannot express the history problem, so that the Checkers and Chess implementations in the repository do not contain such drawing rules.

There are BDD model checking algorithms that can deal with cycles, e.g., for solving Parity Games (Bakera et al. 2009; Kant and van de Pol 2014); some of them inflate the search space (Schuppan and Biere 2006). GGP problems do not allow cyclic game graphs and often introduce a step counter to avoid this trouble.

Duplicate states in same depth are represented only once, thanks to BDD reduction. Duplicates in different depths of the retrograde analysis do not impose burden, as the forward BFS-layers in our layered approach are handled individually.

In some domains, alternative goal tests are faster. Explicit checks, however, are domain-dependent, so we contributed domain-independent ones using BDDs. Our general interest exceeds game playing: the proposed combination of explicit and symbolic memory-limited search is applicable to many other planning formalisms.

Dropping the BDD library and using a specialized implementation for ranking saves memory. As CUDD has advanced operations like zero-supression, which likely gain more space, we have not followed the approach.

There is no need to know the maximum solution depth in advance. We apply our approach in GGP, where by definition all games end after a finite number of steps – but we do not necessarily know at which depth. We start by performing forward search to determine all reachable states – after this we of course know the maximum solution depth (and the depth of every reachable state).

We have not yet considered the combination of explicit-state and symbolic memory-limited *heuristic* search. For example, in cost-optimal action planning rather accurate BDD exploration heuristics exist (Helmert et al. 2013; Edelkamp, Kissmann, and Torralba 2012b) and have also been applied as lookup tables for explicit-state space forward search.

Wrt. improved BDD exploration one further research avenue is to look at problem representations with preconditions and effects, so that improved image operations based on the concept of transition trees apply (Torralba, Edelkamp, and Kissmann 2013). Another option is to split the BDD in computing the image based on state set splitting into equally sized parts (Edelkamp, Kissmann, and Torralba 2012a).

References

Allen, J. D. 2011. *The Complete Book of Connect 4: History, Strategy, Puzzles*. Puzzlewright.

Allis, L. V. 1988. A knowledge-based approach of connect-four. Master's thesis, Vrije Universiteit Amsterdam.

Bakera, M.; Edelkamp, S.; Kissmann, P.; and Renner, C. D. 2009. Solving μ-calculus parity games by symbolic planning. In *MOCHART*, 15–33.

Botelho, F. C., and Ziviani, N. 2007. External perfect hashing for very large key sets. In *ACM Conference on Information and Knowledge Management (CIKM)*, 653–662.

Botelho, F. C.; Pagh, R.; and Ziviani, N. 2007. Simple and space-efficient minimal perfect hash functions. In *WADS*, 139–150.

Breyer, T. M., and Korf, R. E. 2010. 1.6-bit pattern databases. In *AAAI*, 39–44.

Bryant, R. E. 1986. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers* 35(8):677–691.

Cooperman, G., and Finkelstein, L. 1992. New methods for using Cayley graphs in interconnection networks. *Discrete Applied Mathematics* 37/38:95–118.

Dietzfelbinger, M., and Edelkamp, S. 2009. Perfect hashing for state spaces in BDD representation. In *KI*, 33–40.

Edelkamp, S., and Kissmann, P. 2008. Symbolic classification of general two-player games. In *KI*, 185–192.

Edelkamp, S., and Kissmann, P. 2009. Optimal symbolic planning with action costs and preferences. In *IJCAI*, 1690–1695.

Edelkamp, S., and Kissmann, P. 2011. On the complexity of BDDs for state space search: A case study in Connect Four. In *AAAI*, 18–23.

Edelkamp, S., and Reffel, F. 1998. OBDDs in heuristic search. In *KI*, 81–92.

Edelkamp, S.; Sulewski, D.; Barnat, J.; Brim, L.; and Simecek, P. 2011. Flash memory efficient LTL model checking. *Sci. Comput. Program.* 76(2):136–157.

Edelkamp, S.; Kissmann, P.; and Torralba, Á. 2012a. Lex-partitioning: A new option for BDD search. In *ETAPS-Workshop GRAPHITE*, 66–82.

Edelkamp, S.; Kissmann, P.; and Torralba, Á. 2012b. Symbolic A* search with pattern databases and the merge-and-shrink abstraction. In *ECAI*, 306–311.

Edelkamp, S.; Sanders, P.; and Simecek, P. 2008. Semi-external LTL model checking. In *CAV*, 530–542.

Edelkamp, S.; Sulewski, D.; and Yücel, C. 2010. GPU exploration of two-player games with perfect hash functions. In *ICAPS-Workshop on Planning in Games*.

Edelkamp, S. 2005. External symbolic heuristic search with pattern databases. In *ICAPS*, 51–60.

Helmert, M.; Haslum, P.; Hoffmann, J.; and Nissim, R. 2013. Merge & shrink abstraction: A method for generating lower bounds in factored state spaces. *Journal of the ACM*. Accepted for Publication.

Kant, G., and van de Pol, J. 2014. Generating and solving symbolic parity games. In *ETAPS-Workshop GRAPHITE*.

Kissmann, P., and Edelkamp, S. 2010. Layer-abstraction for symbolically solving general two-player games. In *SOCS*, 63–70.

Korf, R. E. 2008. Minimizing disk I/O in two-bit breadth-first search. In *AAAI*, 317–324.

Kunkle, D., and Cooperman, G. 2007. Twenty-six moves suffice for Rubik's cube. In *International Symposium on Symbolic and Algebraic Computation (ISSAC)*, 235 – 242.

Love, N. C.; Hinrichs, T. L.; and Genesereth, M. R. 2006. General game playing: Game description language specification. Technical Report LG-2006-01, Stanford Logic Group.

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

Romein, J. W., and Bal, H. E. 2002. Awari is solved. *International Computer Games Association (ICGA) Journal* 25(3):162–165.

Schaeffer, J.; Björnsson, Y.; Burch, N.; Kishimoto, A.; and Müller, M. 2005. Solving checkers. In *IJCAI*, 292–297.

Schuppan, V., and Biere, A. 2006. Liveness checking as safety checking for infinite state spaces. In *INFINITY*, 79–96.

Sievers, S.; Ortlieb, M.; and Helmert, M. 2012. Efficient implementation of pattern database heuristics for classical planning. In *SOCS*.

Sturtevant, N., and Rutherford, M. 2013. Minimizing writes in parallel external memory search. In *IJCAI*, 666–673.

Torralba, Á., and Alcázar, V. 2013. Constrained symbolic search: On mutexes, BDD minimization and more. In *SOCS*.

Torralba, Á.; Edelkamp, S.; and Kissmann, P. 2013. Transition trees for cost-optimal symbolic planning. In *ICAPS*, 206–214.

van Dijk, T.; Laarman, A.; and van de Pol, J. 2012. Multi-core and/or symbolic model checking. *ECEASST* 53.

von Neumann, J., and Morgenstern, O. 1944. *Theory of Games and Economic Behavior*. Princeton University Press.