

The Compression Power of Symbolic Pattern Databases

Marcel Ball and Robert C. Holte

Department of Computing Science, University of Alberta
Edmonton, Alberta, Canada
{marcel,holte}@cs.ualberta.ca

Abstract

The heuristics used for planning and search often take the form of pattern databases generated from abstracted versions of the given state space. Pattern databases are typically stored as lookup tables with one entry for each state in the abstract space, which limits the size of the abstract state space and therefore the quality of the heuristic that can be used with a given amount of memory. In the AIPS-2002 conference Stefan Edelkamp introduced an alternative representation, called symbolic pattern databases, which, for the Blocks World, required two orders of magnitude less memory than a lookup table to store a pattern database. This paper presents experimental evidence that Edelkamp's result is not restricted to a single domain. Symbolic pattern databases, in the form of Algebraic Decision Diagrams, are one or more orders of magnitude smaller than lookup tables on a wide variety of problem domains and abstractions.

Introduction

Heuristic search planners (Bonet and Geffner 1999; Hoffmann and Nebel 2001) and single-agent search algorithms (Hart, Nilsson, and Raphael 1968; Korf 1985) require a heuristic function estimating the distance from any given state to the nearest goal state. An effective method for defining heuristics is to abstract the given state space and to use exact distances in the abstract space as estimates of the distances in the original space (Prieditis 1993). If all the abstract distances are precomputed and stored so that they can be efficiently looked up when a heuristic value is needed during search, the data structure storing the heuristic is called a pattern database (PDB) (Culberson and Schaeffer 1998). Edelkamp (2001) introduced PDBs, and an abstraction method for defining them, to the planning community.

PDBs are most often stored as hash tables, but alternative data structures are possible. Edelkamp (2002) proposed using Binary Decision Diagrams (BDDs) to represent PDBs, and called PDBs represented in this way "symbolic pattern databases". He reported that symbolic PDBs for the Blocks World required two orders of magnitude less memory than the conventional hash table representation.

This paper examines the generality of Edelkamp's result using Algebraic Decision Diagrams (ADDs) instead of

BDDs. We ask this question: does an ADD representation for PDBs usually require significantly less memory than a hash table? Our investigation is experimental: we compare the memory required by the ADD representation with the memory required for a hash table representation with a perfect hashing function on a wide variety of problem domains, problem instances, and abstractions; preliminary performance evaluations of ADD-based PDBs are also conducted. Our main finding is that ADDs are generally a much more compact representation of a PDB than a hash table.

This paper focuses on reducing the memory needed to store a PDB without changing its contents. This is called lossless compression, in contrast to lossy compression, which has been the primary focus of previous research on PDB compression (Felner et al. 2007; Samadi et al. 2008). Lossy techniques in conjunction with the ADD representation would very likely result in significantly improved compression with only minor impact on the heuristic quality, but that topic is beyond the scope of this paper.

Algebraic Decision Diagrams (ADDs)

An ADD (R.I. Bahar et al. 1993) is a directed acyclic graph representing a numerical function over a set of boolean variables, and so is perfectly suited to store the contents of a PDB when states are represented with binary variables, as is done in planning domains. Conceptually the ADD can be viewed as a tree with $n + 1$ layers if there are n variables. Each internal node in the tree at level i tests the boolean variable v_i , and has two outgoing edges leading to nodes at level $i + 1$, one associated with the value *true* for v_i , the other associated with the value *false*. Each leaf node is associated with an output value of the function being represented. Nodes in the tree are then removed or merged together, and the variables can be reordered, to get a more compact ADD. The function value is computed for a given input by traversing the ADD from top to bottom following the appropriate edge for the input's values of each variable node encountered.

There are several software packages implementing ADDs; we used the CUDD package (Somenzi 2005) augmented with our own "read-only" ADD data structure to reduce the number of bytes needed for each ADD node from 20 to 10 for the final representation of the PDB. These packages implement techniques for reducing the size of the ADD, and methods for re-ordering the variables to improve

the effectiveness of the size-reducing methods. For the experiments in this paper the size 3 window permutation reordering method from the CUDD package was used, although other reordering methods were similarly effective.

In order to save even more space we devised and applied a final post-processing technique to remove nodes in the ADD whose only purpose is to distinguish abstract states that are reachable from the goal state from those that are not. The technique removes every internal ADD node that has one child that is a terminal node representing an unreachable state, pointing the parents of that node directly to the other child node. While this technique can be usefully applied to the domains studied in this paper, it may be detrimental in domains that contain states from which the goal is unreachable.

In addition to the nodes directly removed by this technique it often enables further simplifications to be made, resulting in an even smaller ADD. This simple technique often produces substantial memory savings, and in certain cases is required to make the ADD smaller than the hash table. For example, without this technique the ADD for the 13-disk Towers of Hanoi with Abs=12 (see Table 3 and the related text) requires 93,057,910 bytes, which is more than the 67,108,864 bytes required by the hash table, whereas with the technique the ADD requires only 41,292,790 bytes.

Abstraction Technique

The technique used in this paper for creating abstractions is the standard one for PDDL domains introduced by Edelkamp (2001; 2002). The first step is to identify groups of atoms that are mutually exclusive, *i.e.*, in each reachable state at most one of the atoms in a group can be true. For example, Table 1 shows the 6 groups of mutually exclusive atoms for the 3-action (handless) Blocks World with 3 blocks (a, b, and c). Note that the groups are of two “types”. Groups 1 to 3 are one type, representing the restriction that a block can have at most one block on it at a time. Groups 4 to 6 are the other type, representing the constraint that in a given state a block can be on top of one, and only one, other thing (either the table or one of the other blocks). We used the method in HSP for identifying mutually exclusive groups of atoms automatically (Bonet and Geffner 1999). The groups found by this technique are guaranteed to be mutually exclusive, but there is no guarantee the technique will find all pairs of mutually exclusive atoms.

Group	atoms in the group
Group 1	(clear a), (on b a), (on c a)
Group 2	(clear b), (on a b), (on c b)
Group 3	(clear c), (on a c), (on b c)
Group 4	(on-table a), (on a b), (on a c)
Group 5	(on-table b), (on b a), (on b c)
Group 6	(on-table c), (on c a), (on c b)

Table 1: Blocks World groups of mutually exclusive atoms

Once the mutually exclusive groups have been identified, an abstraction is defined by selecting some of the groups.

All atoms that are not in the selected groups are removed from the state descriptions and operator definitions to create the abstract states and operators. A PDB is then created by applying the abstract operators, in retrograde fashion, to the abstract goal state(s). For this we used an ADD version of the symbolic search technique described by Edelkamp (2002). The distance to each abstract state thus reached is recorded in the PDB. It is possible for this retrograde process to produce “spurious” abstract states, *i.e.*, ones that do not correspond to any reachable state in the original state space. In general, there is no efficient method to determine if an abstract state is spurious or not. This topic is beyond the scope of this paper, but it is important to note that in our experiments we have used automatic methods to eliminate those spurious states that violate pairwise mutual exclusions.

Experimental Evaluation

The problem domains used in this study are as follows. **Blocks World**, **Logistics**, **Depots**, **Satellite**, and **Rovers** are standard domains that have been used in AIPS planning competitions (Long and Fox 2003). We used the basic STRIPS version of Depots, Satellite, and Rovers because the more sophisticated versions use fluents, which are not readily handled by our ADD representation or abstraction method. Two non-standard variations of the Blocks World, and one of Satellites, are also used. The **4-peg Towers of Hanoi** puzzle, the **Arrow puzzle**, and the **4×4 Sliding Tile Puzzle** have been used as test domains in the single-agent heuristic search community; here they are implemented in PDDL and abstracted using the method described above. We assume the reader is familiar with all these domains except the Arrow puzzle, which is described in detail below. In all the domains every action has a cost of 1.

The experimental results are presented in a uniform format for all the domains. The discussion of each domain begins with an explanation of the types of mutually exclusive groups of atoms that occur in the domain and the problem instances used in the experiments. As described above, some of the mutually exclusive groups will be used to define each abstraction. The groups chosen are described in the **Abs** column of the experimental results tables (see Table 2, for example). The **States** column gives the number of abstract states whose distances are stored in the PDB. The distances in all the abstract spaces are small enough to require only one byte of storage, so this number is also the number of bytes required to store the PDB as a lookup table assuming a perfect hashing function is known. The next column, **ADD Size**, is the size in bytes of the PDB when it is represented as an ADD. Note that the number of nodes in the ADD is one-tenth of this size, because each node in the ADD requires 10 bytes of storage in the system we used. The rightmost column in the experimental results tables, **Ratio**, is the lookup table size (**States**) divided by **ADD Size**. We call this the compression ratio, and it indicates how effective the ADD representation is at compressing the lookup table. A ratio less than 1 indicates that the ADD representation is larger than the lookup table (assuming a perfect hashing function is known) and it is therefore inadvisable to use the ADD representation. Finally, the leftmost column in the experimental

results tables provides key information about the problem instance used to produce the results in the given row of the table. The exact nature of this information varies from domain to domain and will be described in each section. For example, in the Blocks World (see Table 2) this column gives the number of blocks in the problem instance.

# blocks	Abs	States	ADD Size	Ratio
10	1-8	4,596,553	786,710	5.84
10	1-9	17,572,114	3,168,940	5.55
13	1-7	11,109,337	152,860	72.68
13	1-8	76,751,233	839,160	91.46
13	1-9	472,630,861	4,939,510	95.68
15	1-6	4,010,455	29,600	135.49
15	1-7	38,398,641	152,780	251.33
15	1-8	335,262,313	832,420	402.76

Table 2: Blocks World

Blocks World

This study used the 3-action (handless) Blocks World and problem instances with 10, 13, and 15 blocks in which the goal state has all blocks stacked in one pile, in order, with block 1 on top. As discussed above for the 3-blocks instance of this domain, there are two types of mutually exclusive groups of atoms. All the abstractions were defined by keeping the groups that encoded what was on top of the blocks listed in the **Abs** column. For example “1-8” in the **Abs** column of the first row of Table 2 means that the groups used for the abstraction were those representing what was on top of blocks 1 through 8. Abstractions keeping the groups of mutually exclusive atoms that encode what a block is on top of were not used because these abstractions resulted in poor quality heuristics. This is probably because such abstractions do not keep any (*clear ?b*) atoms, allowing any block and all the blocks above it to be moved in a single action.

The results in the last three rows of Table 2 are consistent with Edelkamp’s finding that symbolic PDB representations compress the lookup table for a 15-block instance by up to two orders of magnitude (Edelkamp 2002). The results cannot be compared exactly because Edelkamp used BDDs instead of ADDs, used a different goal state, and did not report precisely what groups he used to define the abstractions.

Two trends are evident in Table 2. First, as the problem size increases the compression ratio tends to increase for PDBs with similar numbers of states. The two rows highlighted in bold illustrate this. Second, as the abstraction is made more fine-grained for a particular problem instance the compression ratio tends to increase. One exception to this is the two abstractions for the 10-blocks instance. Here the more fine-grained abstraction has a slightly lower compression ratio (5.545 compared to 5.843). This may simply be an artifact of different variable reordering by the ADD software, but it might be systematic since it is also seen in the 4-peg Towers of Hanoi and the Blocks World variant (below) in which there are a limited number of table positions.

4-peg Towers of Hanoi

The 4-peg Towers of Hanoi puzzle is a challenging extension of the standard 3-peg puzzle (Korf and Felner 2007). The groups of mutually exclusive atoms are exactly analogous to those for the Blocks World: there are groups representing the constraint that no disk or peg-bottom can have more than one disk on top of it and other groups representing the constraint that a disk must be on top of exactly one thing (other disk or peg-bottom) in each state. The abstractions used in this experiment keep the groups of mutually exclusive atoms that encode what is at the bottom of each peg as well as those indicating what is on each of the m largest disks (m is listed in the **Abs** column of Table 3). Since nothing can ever be placed on the smallest disk it does not serve any purpose to include the group for what is on top of disk 1. The problem instances tested had 11, 13, 14 and 15 disks. In all cases the goal state has all of the disks stacked on the fourth peg.

The results for this domain are presented in Table 3. The result for 14 disks and **Abs**=13 are directly comparable to the lossless compression results for the PDB based on 14 disks reported by Felner *et al.* (2007), see the bottom row of their Table 4). Our results are not quite as good—a compression ratio of 2.202 compared to their 2.67. The difference is probably due to Felner *et al.* exploiting the fact that their technique compresses cliques in the abstract space and the difference in the distance to the goal from the states within a clique cannot exceed one.

# disks	Abs	States	ADD Size	Ratio
11	5	557,056	73,530	7.58
11	6	1,196,032	119,470	10.01
11	7	2,228,224	228,490	9.75
11	8	3,487,872	431,290	7.90
11	9	4,194,304	824,540	5.09
<i>11</i>	<i>10</i>	<i>4,194,304</i>	<i>1,481,680</i>	<i>2.83</i>
13	6	3,751,936	496,570	7.56
13	7	8,912,896	987,800	9.02
13	8	19,136,512	2,108,020	9.08
13	9	35,651,584	5,478,370	6.51
13	10	54,525,952	11,262,840	4.84
13	11	67,108,864	22,272,850	3.01
13	12	67,108,864	41,292,790	1.63
14	6	5,865,472	630,200	9.31
14	7	15,007,744	1,252,520	11.98
14	8	35,651,584	2,594,840	13.74
14	9	76,546,048	5,810,420	13.17
14	10	142,606,336	17,380,290	8.25
14	11	218,103,808	37,930,360	5.75
14	12	268,435,456	70,932,920	3.78
14	13	268,435,456	121,927,650	2.20
15	7	23,461,888	1,550,940	15.13
15	8	60,030,968	3,200,650	18.76
15	9	142,606,336	6,992,120	20.40
15	10	306,184,192	24,380,420	12.56

Table 3: 4-peg Towers of Hanoi

The first trend seen in the Blocks World results holds here as well: for PDBs with a similar number of states the one for the larger problem will tend to have a higher compression ratio (e.g., compare the row for 13 disks and **Abs**=10 with the row for 15 disks and **Abs**=8). However, the other trend seen in the Blocks World doesn't seem to be repeated here; instead the compression ratio tends to increase and then decrease as the abstraction is made more fine-grained.

Blocks World Variants

The Towers of Hanoi can be seen as a variant of the Blocks World in which the blocks (disks) all have distinct sizes that restrict which blocks can be on a given block and in which there are a limited number of positions on the table (the pegs). Since we see much higher compression in the Blocks World than in the Towers of Hanoi, it is of interest to study the variants of the Blocks World that lie in between the two.

Blocks World with limited table positions

In this Blocks World variant there are a limited number of named positions on the table on which a block can be placed. The groups of mutually exclusive atoms that encode the restriction that a block can be on top of at most one thing change to reflect the fact that they can be on specific table positions as opposed to just being generically on the table. In addition a new type of group is introduced that encodes that at most one block can be on the table in each position.

Table 4 presents the compression results for this Blocks World variant with four table positions. Two problem sizes were used: one had eight blocks, the other had ten. In both cases the goal state has all blocks stacked in position four in order with the lowest number block at the top. All abstractions keep the groups encoding what is on top of the table positions as well as the groups encoding what is on top of the blocks listed in the **Abs** column (for example, "4-8" means blocks 4 through 8 inclusive).

# blocks	Abs	States	ADD Size	Ratio
8	4-8	2,624,788	93,590	28.05
8	3-8	6,691,305	137,000	48.84
8	2-8	12,975,561	265,490	48.87
10	8-10	1,878,476	165,710	11.33
10	7-10	8,786,057	263,390	33.36
10	6-10	35,958,326	369,410	97.34
10	5-10	129,229,547	563,480	229.34

Table 4: Blocks World with four table positions

Overall this domain achieves high compression with compression ratios tending to increase as the abstraction was made more fine-grained; although compression ratios do not tend to increase when moving to a larger problem instance when the number of states in the PDBs are similar.

This domain is also of interest as it contains spurious states that are not removed by the automated spurious-state filtering that is performed. While hand-crafted filters are not useful in an automated planning setting, it is still interesting to study their effect on the PDBs. In this domain the removal of spurious states tends to have a negative effect on

compression. For example the PDB for the 10 blocks instance with abstraction "5-10" with additional hand-crafted spurious-state filtering has 122,995,840 entries but requires 4,786,790 bytes to store, a compression ratio of 25.69. This is almost an order of magnitude less compression than for only automated filtering. Note that hand-crafted filters are likely to generate a significantly better heuristic (the average heuristic value is 11.15 compared to 10.41 in the version with only automated filtering).

Blocks World with distinct block sizes

In this Blocks World variant the table is infinite, as in the original Blocks World, but each block has a distinct size and a larger block cannot be put on top of a smaller one. This modification does not add or remove any types of mutually exclusive atoms, but it does modify the existing ones to reflect which blocks can possibly be on top of a given block.

Table 5 presents the compression results for three instances of this Blocks World variant, with 10, 13, and 15 blocks respectively. In each the goal state has all of the blocks in one stack. The abstractions for this variant are based on the mutually exclusive groups encoding what is on top of the m largest blocks (m is given in the **Abs** column).

# blocks	Abs	States	ADD Size	Ratio
10	5	10,427	7,470	1.40
10	6	29,371	10,730	2.74
10	7	60,814	12,720	4.78
13	7	1,322,035	77,390	17.08
13	8	3,967,195	100,620	39.43
13	9	9,131,275	114,240	79.93
13	10	16,360,786	119,620	136.77
15	8	112,287,418	370,500	303.07
15	9	378,716,908	481,550	786.45
15	10	993,098,339	550,240	1804.85
15	11	2,046,540,334	589,320	3472.71
15	12	3,397,658,503	634,650	5353.59

Table 5: Blocks World with distinct block sizes

ADDs produce very good compression here, reducing memory requirements by more than 3 orders of magnitude for the PDBs based on 10 or more blocks in the 15-block instance (bottom three rows). As in the original Blocks World, the compression ratio increases for a given problem instance as the abstraction becomes more fine-grained, but here the rate of increase is much greater. For example, for the 15-block instance the PDB based on 12 blocks has more than 30 times the number of states as the PDB based on 8 blocks, but its ADD representation is less than twice the size.

Logistics

The logistics domain is a well-known planning domain; the version used here is the STRIPS version of the domain from the AIPS-98 planning competition. There are 3 types of mutually exclusive groups of atoms in the domain, as follows:

- **Airplane Locations** – Groups of this type encode the constraint that an airplane may be in only one location, which

must be an airport, in any given state. There is one group of this type for each airplane in the problem instance.

- **Truck Location** – Groups of this type encode the constraint that a truck may be in only one location, which must be in the city the truck is located in, in any given state. There is one group of this type for each truck.
- **Package Location** – Groups of this type encode the constraint that a package can only be in one place in any given state; this can be one of the locations in the cities, or in one of the trucks or airplanes. There is one group of this type for each package.

Two instances of this puzzle are used. One with 4 cities (each with 4 locations and 1 truck), 1 airplane and 3 packages (listed as “4/4/1/3” in the **Prb** column); the other has 4 cities (each with 5 locations and 1 truck), 2 airplanes and 5 packages (listed as “4/5/2/5” in the **Prb** column). In both instances the goal location of the packages were randomized, as were the initial locations of the packages, trucks and airplanes. “A” in the **Abs** column indicates the airplane locations were used in the abstraction, “T” indicates the truck locations were used, and “xP” indicates that the groups for x packages were used.

Prb	Abs	States	ADD Size	Ratio
4/4/1/3	3P	9,261	4,010	2.31
4/4/1/3	A+T+1P	21,504	1,370	15.69
4/4/1/3	A+T+2P	451,584	18,540	24.36
4/4/1/3	A+T+3P	9,483,264	217,060	43.69
4/5/2/5	A+T+1P	260,000	2,160	120.37
4/5/2/5	5P	11,881,376	15,550	764.08
4/5/2/5	A+T+3P	1.76×10^8	1,208,700	145.41

Table 6: Logistics

The results for the Logistics domain are shown in Table 6. Overall this domain compresses quite well, with ratios easily reaching more than 2 orders of magnitude. This domain generally exhibits the trend that as the abstraction is made more fine-grained the compression ratio increases. One exception to this is the PDB for problem “4/5/2/5” with abstraction “5P”. Despite being smaller than the PDB with abstraction “A+T+3P” it has a larger compression ratio. In addition, this domain follows the trend that greater compression ratio tends to be achieved on larger problem sizes if the number of entries is similar; this is illustrated by the two entries in bold.

Depots

The Depots domain combines two other planning domains—Logistics and the 4-action Blocks World (with a hand)—in such a way that trucks move crates around between locations at which hoists load and unload the crates onto pallets. There are four types of mutually exclusive groups of atoms in this domain, as follows:

- **Truck location** – Groups of this type encode the constraint that a truck may only be at one location in any

given state. There is one group of this type for each truck in the problem instance.

- **Hoist Information** – Groups of this type encode the constraint that a hoist cannot be simultaneously unoccupied and lifting something. There is one group of this type for each hoist in the problem instance.
- **Pallet information** – Groups of this type encode the constraint that a pallet is either empty or has exactly one crate sitting directly on it. There is one group of this type for each pallet in the problem instance.
- **Crate location** – Groups of this type encode the constraint that the following types of atoms are mutually exclusive for any given crate: (*on ?crate ?pallet*), (*on ?crate1 ?crate2*), (*in ?crate ?truck*), and (*lifting ?crate ?hoist*). There is one group of this type for each crate in the problem instance.

The “No Truck” abstractions used in this experiment keep all the atoms except those in the truck location groups. Similarly, the “No Hoist”/“No Pallet” abstractions keep all the atoms except those in the hoist/pallet information groups. The “No Crate” abstractions leave out the crate location groups for the crates that are listed in the **Abs** column. Finally, the “Complete” abstractions keep all groups.

Prb	Abs	States	ADD Size	Ratio
P02	No Hoist/Pallet	4,554	160	28.46
P02	No Trucks	4,480	5,550	0.81
P02	No Crate 1-3	15,138	6,190	2.45
P02	No Pallet	17,802	2,600	6.85
P02	No Crate 1, 2	22,518	11,200	2.01
P02	No Hoist	25,515	5,880	4.34
P02	No Crate 1	31,176	17,440	1.79
P02	Complete	40,320	15,850	2.54
P03	No Hoist/Pallet	194,409	24,300	8.00
P03	No Crate 1-5	271,143	62,930	4.31
P03	No Trucks	393,376	356,420	1.10
P03	No Pallet	811,458	136,430	5.95
P03	No Crate 1-3	983,007	365,400	2.69
P03	No Crate 1	2,494,683	711,850	3.5
P03	No Hoist	2,811,303	113,730	24.72
P03	Complete	3,540,384	1,491,070	2.37

Table 7: Depots

Overall, the compression in the Depots domain is poor; often with ratios near 1 or even below 1 (highlighted in bold). It is also hard to find any clear patterns in the compression ratios. This may be because there are spurious states in the PDBs for the Depots domain that are not being removed; these may affect the compression.

Satellite

The Satellite domain involves planning and scheduling a collection of observation tasks for multiple satellites that are each differently equipped. The following types of mutually exclusive atom groups are used for abstraction.

- **Direction information** – Groups of this type encode the constraint that in any given state a satellite is pointing in exactly one direction. There is one group of this type for each satellite in the problem instance.
- **Power information** – Groups of this type encode the constraints that in any given state a satellite either has power or it doesn't and that a satellite with power has at most one of its instruments powered on. There is one group of this type for each satellite in the problem instance.
- **Calibration information** – Groups of this type encode the constraint that in any given state an instrument either is calibrated or it is not. There is one group of this type for each instrument in the problem instance.
- **Observation information** – Groups of this type encode the constraint that an observation either has been made or it hasn't. There is one group of this type for each direction-mode pair in the problem instance. Only groups that contain a (*have_image ?direction ?mode*) atom that is present in the goal are used for creating abstractions.

Four problem instances from the AIPS 2002 problem set were used; column **Prb** in Table 8 gives the problem number. Problem 3 has two satellites, four instruments and eight directions. Problem 5 has three satellites, nine instruments and ten directions. Problem 6 has three satellites, five instruments, and eleven directions. Problem 7 has four satellites, eight instruments and twelve directions. The relative size of the problems is: 7 (largest), 5, 6, and 3 (smallest).

The abstractions used in this experiment (see the **Abs** column in Table 8) are as follows. The “No D” abstractions use all of the groups except those relating to the direction a satellite is pointing. The “No C” abstractions use all of the groups except those relating to instrument calibration. The “All” abstraction keeps all the groups of all types except for the Observation Information groups that do not contain a (*have_image ?direction ?mode*) atom that is present in the goal. This abstraction produces a perfect heuristic because the only information it drops from the original problem instance is irrelevant to solving the problem.

Prb	Abs	States	ADD Size	Ratio
3	No D	2,048	490	4.18
3	No C	8,192	2,870	2.85
3	All	131,072	4,020	32.60
6	No D	65,536	2,190	29.93
6	No C	2,725,888	28,210	96.65
6	All	87,228,416	39,760	2,193.87
5	No D	209,152	2,190	95.50
5	No C	4,096,000	86,330	47.45
5	All	2,097,153,000	130,020	16,129.46
7	No D	2,359,296	4,070	579.68
7	No C	191,102,976	406,320	470.33
7	All	48,922,361,856	949,450	51,527.05

Table 8: Satellite

Table 8 presents the compression results for this experiment. Clearly, ADDs are very effective in this domain re-

ducing memory requirements by over 4 orders of magnitude for the PDB based on the “All” abstraction for problem 7. The two trends observed in the Blocks World domain are also seen here: (1) within a problem instance the compression ratio increases as the abstraction is made more fine-grained; and (2) if two PDBs for different problem instances have a similar number of states the PDB for the larger problem instance compresses more (for example, compare the rows highlighted in bold in Table 8).

An interesting variation on the Satellite domain imposes restrictions on how a satellite's direction can be changed instead of allowing it to freely move from its current direction to any other direction. For a given problem instance we randomly generated a connected graph in which the nodes represented the directions, and a satellite could only move between two directions if there was an edge between the two corresponding nodes in the graph.¹

Table 9 lists the compression results for this Satellite variant. For each problem instance 8 different movement topologies were created and the resulting ADD sizes were averaged to produce the **ADD Size** shown in the table. The “No D” abstraction is not reported here since it is the same as in Table 8. While this modified version still compresses well, its compression ratio is consistently lower than the original version's. One possible reason for this is that in the movement-constrained variant the branching factor is lower (fewer direction changes are allowed in any given state), with the consequence that the maximum heuristic value is greater because the PDBs still contain the same number of states. A larger range of heuristic values means more leaf nodes in the ADD. While this does not necessitate a larger ADD overall, it could be a contributing factor.

Prb	Abs	States	ADD Size	Ratio
3	No C	8,192	6,320	1.30
3	All	131,072	11,280	11.62
5	No C	4,096,000	252,990	16.19
5	All	2,097,153,000	1,413,300	1,483.87
6	No C	2,725,888	150,690	18.09
6	All	87,228,416	317,610	274.64
7	No C	191,102,976	1,990,590	96.00
7	All	48,922,361,856	10,658,540	4,589.97

Table 9: Satellite with constrained motion

Rovers

The Rovers domain involves planning and scheduling tasks for multiple rovers, as well as a lander, that are differently equipped. This domain is similar to the Satellite domain with constrained motion but is more complex. The mutually exclusive atom groups that are used for abstraction in

¹The directions were placed at random positions in a 2-dimensional coordinate system. Two directions were connected to each other if the distance between their positions was less than or equal to a threshold. The threshold was defined as the minimum value needed to make the resulting graph connected.

the Satellite domain are of the following types. The abbreviation used for each type in the **Abs** column of Table 10 is shown in brackets after the name of the type (the “Rock” and “Soil” abbreviations each denote two types that are included or excluded together in the abstractions we defined).

- **Location Information (Loc)** – Groups of this type encode the constraint that a rover can be at only one location at a time. There is one group of this type for each rover in the problem instance.
- **Rock Analysis (Rock)** – Groups of this type encode the constraint that a rock sample is either at the waypoint it starts at or has been analyzed by one of the rovers equipped for rock analysis. There is one group of this type for each rock sample in the problem instance.
- **Rock Communication (Rock)** – Groups of this type encode the constraint that a rock sample cannot be at the waypoint it started at if the data for that rock sample has been communicated to a lander. There is one group of this type for each rock sample in the problem instance.
- **Soil Analysis (Soil)** – Groups of this type encode the constraint that a soil sample is either at the waypoint it starts at or has been analyzed by one of the rovers equipped for soil analysis. There is one group of this type for each soil sample in the problem instance.
- **Soil Communication (Soil)** – Groups of this type encode the constraint that a soil sample cannot be at the waypoint it started at if the data for that soil sample has been communicated to a lander. There is one group of this type for each soil sample in the problem instance.
- **Storage Information (Stor)** – Groups of this type encode the constraint that a storage unit cannot be both full and empty at the same time. There is one group of this type for each storage unit in the problem instance.

Three types of atoms are independent of all other atoms and therefore form groups of their own. Atoms of the form (*calibrated ?camera ?rover*) are included in all abstractions. Atoms of the forms (*have_image ?rover ?objective ?mode*), and (*communicated_image_data ?objective ?mode*) are referred to as “Img” in the **Abs** column of Table 10. The (*communicated_image_data ?objective ?mode*) atoms that do not occur in the goal state are never included in an abstraction.

Three instances of the Rovers problem from the AIPS 2002 competition were tested. Problem 6 consists of 2 rovers, 6 waypoints, 3 cameras with 3 modes, 2 objectives, 4 soil samples, and 4 rock samples. Problem 7 has 3 rovers, 6 waypoints, 2 cameras with 3 modes, 2 objectives, 2 soil samples and 4 rock samples. Problem 8 has 4 rovers, 6 waypoints, 4 cameras with 3 modes, 3 objectives, 3 soil samples, and 4 rock samples. The relative sizes of these instances are: 8 (largest), 7, and 6 (smallest).

Table 10 presents the compression results for this experiment. One trend that is observed is that larger PDBs tend to compress better than smaller ones. In addition, if two PDBs are of similar size the one with the larger range of heuristic values tends to compress less than the one with the smaller range of heuristic values. Consider, for example, the

Prb	Abs	States	ADD Size	Ratio
6	No Rock, Img, or Cal	11,664	2,220	5.25
6	No Img or Cal	209,952	12,490	16.81
6	No Rock, Soil, or Stor	294,912	1,680	175.54
6	No Soil	5,308,416	6,700	729.30
6	No Loc	11,943,369	6,190	1,929.46
6	No Rock	23,887,872	15,730	1,518.62
6	No Cal	53,747,424	26,430	2,033.58
6	No Stor	1.07×10^8	25,290	4,250.51
7	No Rock, Img, or Cal	43,200	2,210	19.55
7	No Soil, Img, or Cal	338,688	25,090	13.50
7	No Loc	1,254,400	2,000	627.20
7	No Rock	1,382,400	4,290	322.24
7	No Img or Cal	8,467,200	136,730	61.93
7	No Soil	10,838,016	41,980	258.17
7	No Stor	33,868,800	48,970	691.62
7	No Cal	67,737,600	124,120	545.74
8	No Soil, Img, or Cal	1,679,616	187,570	8.95
8	No Rock, Img, or Cal	2,592,000	27,610	93.88
8	No Rock, Soil, or Stor	42,467,328	52,520	808.59
8	No Img Or Cal	2.10×10^8	1,215,470	172.73

Table 10: Rovers

two PDBs for Problem 6 highlighted in bold. The maximum heuristic value for the abstraction “No Img or Cal” is 31, while the maximum heuristic value for the abstraction “No Rock, Soil, or Stor” is only 10.

Arrow Puzzle

In the arrow puzzle there is a set of arrows, each of which can point either up or down, a symmetric adjacency relation indicating which arrows are neighbours of one another, and an operator for each adjacent pair of arrows that reverses the direction each arrow is pointing (this is a generalization of the original version (Korf 1980), which simply had the arrows arranged in a line). Three parameterized predicates are used to represent this puzzle in PDDL. (*up ?x*) is true when arrow *?x* is pointing up, (*down ?x*) is true when arrow *?x* is pointing down, and the static predicate (*adjacent ?x ?y*) is true if arrow *?x* is adjacent to arrow *?y*. There is only one type of mutually exclusive group of atoms, encoding the fact that each arrow is either up or down. There is one such group for each arrow in the problem instance.

The problem instances used in this experiment were rectangular arrangements of arrows, in either 2 or 3 dimensions,

in which an arrow was adjacent to its neighbours in the cardinal directions. Column **Prb** in Table 11 describes the arrangement. The abstractions in this experiment are based on the atom groups for the arrows in a subrectangle anchored at a corner of the original puzzle of the size given in the **Abs** column of Table 11.

Prb	Abs	States	ADD Size	Ratio
3×3×2	3×2×1	4,096	2,170	1.89
3×3×2	3×2×2	131,072	11,820	11.09
5×5	4×4	65,536	4,380	14.96
5×5	5×4	1,048,576	13,040	80.41
5×5	5×5	16,777,216	24,090	696.44
3×9	2×4	256	310	0.83
3×9	3×6	262,144	2,820	92.96
3×9	3×9	67,108,864	42,440	1,581.26
3×3×3	2×2×2	256	510	0.50
3×3×3	3×3×2	262,144	15,430	16.99
3×3×3	3×3×3	67,108,864	278,890	240.63
6×6	5×5	33,554,432	36,430	921.07
6×6	5×6	1.07×10^9	93,750	11,453.25
6×6	6×6	3.44×10^{10}	162,760	2.11×10^5
4×3×3	3×3×2	262,144	11,910	22.01
4×3×3	4×3×2	16,777,216	66,990	250.44

Table 11: Arrow puzzle

The compression results for this experiment are shown in Table 11. The two PDBs with the fewest states have a compression ratio less than one (highlighted in bold) but, as in most other domains, the compression ratio increases as the abstractions become more fine-grained, with very impressive ratios for the most fine-grained abstractions in the larger problem instances. The rate of increase of the compression ratio is sufficiently large that the ADD size grows extremely slowly compared to the size of the lookup table. For example, the 6×6 abstraction of the 6×6 problem instance has a lookup table that is 1024 times larger than the lookup table for the 5×5 abstraction, but an ADD that is only 4.5 times larger. Unlike several other domains, the compression ratio for similar size PDBs for different problem instances does not increase with the problem size. For example, the best compression ratio by far of the three PDBs with 262,144 states is for the smallest problem instance. The dimensionality of the puzzle might play a role in this (there are more interactions between the arrows in a 3-dimensional arrangement than in a 2-dimensional one).

4×4 Sliding Tile Puzzle

States of the sliding tile puzzle can be represented using three parameterized predicates: (*in ?tile ?position*) is true if *?tile* is located at *?position*; (*blank ?position*) which is true if the blank is located at *?position*; and the symmetric static predicate (*adjacent ?position1 ?position2*) is true if *?position1* is adjacent to *?position2* in the physical puzzle. There are two types of mutually exclusive groups with this representation. One type encodes the constraint that only one

tile, or the blank, can be in a particular position at a time. There is one of these groups for each position in the problem instance. The second type encodes the constraint that a particular tile, and the blank, can only be in one position at a time. There is one group of this type for each tile and blank in the problem instance. The abstractions in this experiment kept groups of the second type for the blank and for the tiles listed in the **Abs** column of Table 12.

Prb	Abs	States	ADD Size	Ratio
4×4	B,1,2,4,5	524,160	1,619,900	0.32
4×4	B,1,2,4-6	5,765,760	13,431,610	0.43
4×4	B,1-6	57,657,600	102,510,700	0.56

Table 12: 4×4 Sliding-Tile puzzle

The compression results for the 4×4 sliding tile puzzle with the standard goal state (blank in the top left corner) are shown in Table 12. In no case did an ADD use less memory than a lookup table. A similar result was obtained when a different representation and method of abstraction was used for this domain. This is in contrast with the results by Felner *et al.* (2007) whose lossless “edge” technique achieved a compression ratio of 1.25 on this domain (see their Table 7).

Time Performance Evaluation

One important factor to consider when evaluating any compression technique for PDBs is the computation time required by the technique. Two separate times need to be evaluated; the time required to build the PDB and the time required to evaluate the heuristic value which impacts the time when performing a search using the heuristic.²

Prb	Abs	Hash	ADD
Log. 4/4/1/3	A+T+2P	0.91	0.41
Log. 4/4/1/3	A+T+1P	22.36	3.83
Sat. P03	No D.	0.04	0.04
Sat. P03	No C.	0.31	0.16
Sat. P03	Comp	5.63	0.28
Sat. P06	No D.	2.48	0.10
Sliding 4×4	B 1 2 4 5	203.2	104.9

Table 13: Time required to build pattern databases

Table 13 lists times to build several PDBs both as a hash table (using an explicit retrograde search) and an ADD (using a symbolic retrograde search).³ In the table **Prb** is the domain and instance, **Abs** is the abstraction, and **Hash** and **ADD** are the times in seconds required to build the hash table based and the ADD-based PDBs respectively. The symbolically built ADD-based PDB is consistently quicker to build than the explicitly built hash table based PDB; there-

²Times reported in this section are from runs on 1 core of a Intel 2.2 GHz Core 2 Duo with 4 GB of RAM running Mac OS X 10.5.2.

³These methods were chosen as they are the typical methods used to build PDBs of these types.

fore the time required to build the PDB is unlikely to be an obstacle to the use of ADD based symbolic PDBs.

Prb	Abs	Hash	ADD
Log. 4/4/1/3	A+T+2P	16.73	17.42
Log. 4/4/1/3	A+T+1P	16.58	16.99
Sat. P03	No D.	12.33	12.20
Sat. P03	No C.	12.70	12.37
Sat. P03	Comp	12.84	12.65
Sat. P06	No D.	25.63	25.65
Sliding 4×4	B 1 2 4 5	18.62	20.84

Table 14: Time required to query pattern databases

Table 14 lists the time required to make 1,000,000 randomly generated heuristic value lookups in the PDBs that were generated. **Prb** and **Abs** have the same meaning as the previous table, and **Hash** and **ADD** are the times in seconds required to do the 1,000,000 lookups for the hash table-based and ADD-based PDBs respectively. In all cases the times required to perform the lookups for the ADD-based PDBs were similar to those for the hash table based PDB, with it being slightly faster for Satellite P03, and slightly slower for Satellite P04, Logistics 4/4/1/3 and Sliding Tile 4×4. Since the lookup times are quite similar it is unlikely that the lookup time would be a problem for using ADD-based PDBs. In addition, since the ADD-based PDB is quite often significantly smaller than the hash table-based implementation, some of the memory saved by using the ADD representation can be used for other memory-based techniques to speed up search, resulting in actual search times that are quicker.

Factors Predicting Poor Compressibility

This paper focuses on the question “Does a symbolic representation of a PDB, in the form of an ADD, usually require significantly less memory than a hash table representation?”. The empirical results presented indicate that ADD representation is beneficial in a wide variety of domains.

One question that remains unanswered is why symbolic representations work very well in some domains, but poorly in others. A theoretical approach to answering this question is taken by Edelkamp and Kissmann (2008). Analyzing our experiments revealed two trends: (1) domains with low branching factors tend to compress poorly compared to those with higher branching factors; and (2) domains that have large changes to the state description from a state to its successors tend to compress poorly.

Figure 1 plots the branching factor of the domain compared to the compression achieved in PDBs for the domain, and Figure 2 plots the state encoding disruption (as a percent of the state encoding length) compared to the compression achieved.⁴ Each point in these plots is the compression ratio of the PDB compared to the branching factor, or encoding disruption, of the problem the PDB was built for. These plots

⁴These plots include data for domains that are not covered in this paper because of space constraints.

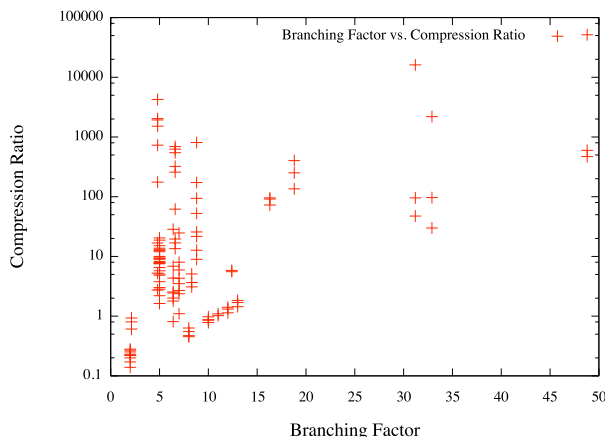


Figure 1: Branching factor vs. compression ratio

illustrate the trends observed between the branching factor of a domain and the compression achieved and the changes to the state encoding and the compression achieved respectively. In particular domains with branching factors below 5 and/or encoding disruptions above 10% tend to have PDBs that compress poorly. Further investigation into factors that affect compression can be found in (Ball 2008).

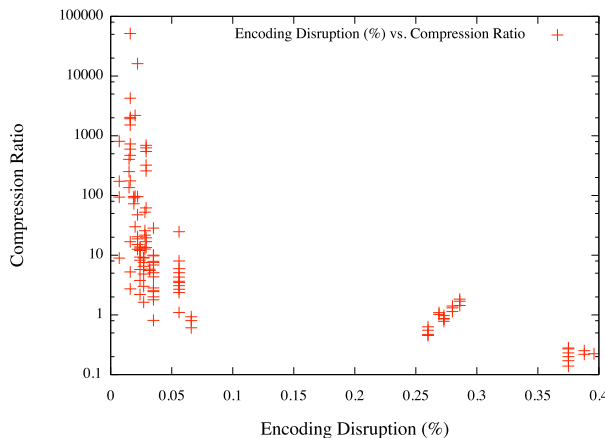


Figure 2: Encoding disruption vs. compression ratio

Conclusions

The primary goal of this paper was to test the generality of the impressive memory reductions reported by Edelkamp (2002) when ADD/BDDs are used to represent PDBs instead of hash tables. Our results show that ADDs are indeed a superior representation for PDBs for a wide variety of domains, problem instances, and abstractions. In a few of the cases we studied ADDs required more memory than hash tables, but in the vast majority ADDs required substantially less memory, often several orders of magnitude less.

It is recommended to use some of the memory freed up

by using an ADD instead of a hash table to speed up search because the time required to look up a heuristic value in an ADD can be greater than a hash table lookup, although the experiments conducted indicate only a minor slowdown, if any. This can be done in several ways. First, ADDs allow the use of PDBs that are much bigger than would fit in memory if represented as a hash table, and larger PDBs usually represent more accurate heuristics (Hernádvölgyi and Holte 2000). Note that the ADD representation is built up incrementally as the abstract space is traversed so that at no time is the entire PDB, in its uncompressed form, required to be stored in memory. Secondly, as an alternative to making a larger PDB, the memory freed up by using the ADD representation can be used for multiple pattern databases, transposition tables, or other memory-intensive methods for speeding up search (Holte et al. 2006; Reinefeld and Marsland 1994).

Two trends were observed in many of the domains. First, in most cases studied the compression ratio tends to increase as the abstraction is made more fine-grained. Second, as the problem size increases the compression ratio tends to increase if the number of states in the PDB remains similar.

The instances where compression of PDBs is most needed are those where the problem size is so large that it is impossible to create high-quality PDBs based upon fine-grained abstractions within the limited memory of today's computers. Fortunately, this is the situation where symbolic PDBs are most likely to be beneficial, as compression tends to increase both as problem size increases and as the abstractions are made more fine-grained.

Even in domains like the Towers of Hanoi, where compression tends to decrease as abstractions become very fine-grained, symbolic PDBs may still be beneficial because the decrease began only when the abstractions were so fine-grained that the size of the abstract state space was approaching the size of the original state space. In practice such fine-grained abstractions will rarely be used because the original state space is so much larger than the available memory.

Acknowledgements

We thank the Natural Sciences and Engineering Research Council of Canada and the Alberta Ingenuity Centre for Machine Learning for their support of this research. We would also like to thank Sandra Zilles and the reviewers for their feedback and insightful comments.

References

Ball, M. A. 2008. Compression of pattern databases using algebraic decision diagrams. Master's thesis, University of Alberta.

Bonet, B., and Geffner, H. 1999. Planning as heuristic search: New results. In Biundo, S., and Fox, M., eds., *Proc. 5th European Conf. on Planning*, 359–371. Durham, UK: Springer: Lecture Notes on Computer Science.

Culberson, J. C., and Schaeffer, J. 1998. Pattern databases. *Computational Intelligence* 14(3):318–334.

Edelkamp, S., and Kissmann, P. 2008. Limits and possibilities of bdds in state space search. In *AAAI'08*.

Edelkamp, S. 2001. Planning with pattern databases. In *Proceedings of the 6th European Conference on Planning*, 13–24.

Edelkamp, S. 2002. Symbolic pattern databases in heuristic search planning. In *Proceedings of Sixth International Conference on AI Planning and Scheduling (AIPS-02)*, 274–283.

Felner, A.; Korf, R. E.; Meshulam, R.; and Holte, R. 2007. Compressed pattern databases. *Journal of Artificial Intelligence Research* 30:213–247.

Hart, P.; Nilsson, N.; and Raphael, B. 1968. A formal basis for the heuristic determination of minimum cost paths. In *IEEE Transactions on Systems Science and Cybernetics*, volume SSC-4(2), 100–107.

Hernádvölgyi, I., and Holte, R. C. 2000. Experiments with automatically created memory-based heuristics. *Proc. SARA-2000, Lecture Notes in Artificial Intelligence* 1864:281–290.

Hoffmann, J., and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research (JAIR)* 14:253–302.

Holte, R. C.; Felner, A.; Newton, J.; Meshulam, R.; and Furcy, D. 2006. Maximizing over multiple pattern databases speeds up heuristic search. *Artificial Intelligence* 170:1123–1136.

Korf, R. E., and Felner, A. 2007. Recent progress in heuristic search: A case study of the four-peg Towers of Hanoi problem. In *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI-07)*, 2324–2329.

Korf, R. E. 1980. Toward a model of representation changes. *Artificial Intelligence* 14(1):41–78.

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

Long, D., and Fox, M., eds. 2003. *Special Issue on the 3rd International Planning Competition, Journal of Artificial Intelligence Research (JAIR)*, volume 20.

Prieditis, A. E. 1993. Machine discovery of effective admissible heuristics. *Machine Learning* 12:117–141.

Reinefeld, A., and Marsland, T. 1994. Enhanced iterative-deepening search. *IEEE Trans. PAMI* 16:701–710.

R.I. Bahar; E.A. Frohm; C.M. Gaona; G.D. Hachtel; E. Macii; A. Pardo; and F. Somenzi. 1993. Algebraic Decision Diagrams and Their Applications. In *IEEE / ACM International Conference on CAD*, 188–191. Santa Clara, California: IEEE Computer Society Press.

Samadi, M.; Siabani, M.; Felner, A.; and Holte, R. 2008. Compressing pattern databases with learning. In *ECAI'2008*.

Somenzi, F. 2005. CUDD: CU Decision Diagram package release 2.4.1. Software Release.