Pushing the Limits for Planning Pattern Databases

Stefan Edelkamp and Shahid Jabbar
Computer Science Department
Otto-Hahn-Str. 14
University of Dortmund

April 25, 2007

Abstract

In this paper we illustrate efforts to perform memory efficient large-scale planning. We first generate sets of disjoint pattern databases space-efficiently using symbolic PDDL planning on disk. To improve the quality of the heuristic, temporal constraints are imposed. We then apply external memory heuristic search and propose its integration with IDA*. Different options for parallelization to save time and memory are presented. The general techniques are mapped to the (n^2-1)-Puzzle as a running case study.

1 Introduction

Heuristic search [27] corresponds to solving an abstract problem exactly. This notion of abstraction makes it possible to compute search heuristics automatically, opposed to domain-dependent solutions using human intuition. If abstract goal distances are computed on-the-fly, A* can be more time-consuming than breadth-first search [31]. As one solution, [3] propose pattern databases (PDBs) to precompute all abstract goal distances for table look-ups in the concrete state space.

To improve the scaling of PDBs in planning practice we integrate symbolic planning PDBs with explicit-state external memory heuristic search. Besides this new combination of planning technologies, we study a number of refinements to push the limits of PDBs in planning, including: control rule PDBs, shared memory PDBs, partitioned PDB construction, distributed PDBs. Furthermore, we propose the integration of iterative-deepening with external memory A* search, the delayed generation of successors, and relay A* to find approximate plans. To push the limits for these general planning techniques, we report on a large-scale case study for the (n^2-1)-Puzzle domain.

The paper is structured as follows. First, we introduce the case study and turn to external explicit-state A* search for solving problems that are larger than main memory. This solves 15-Puzzle instances optimally. We then consider planning abstractions and planning PDBs and turn to PDBs whose entries can be added. We then address different approaches for PDB compaction and introduce symbolic PDB based on BDDs, their construction and their addressing. We next show how simple temporal control rules improve the quality of planning PDBs. The integration of IDA* to External A* search shows trade-off and optimally solves random 24-Puzzle instances with one disjoint PDB. For solving the 35-Puzzle, more and larger disjoint PDBs are needed. For external symbolic PDB construction, BFS-levels and sub-images are manipulated on disk. When loading different symbolic planning PDBs in a shared BDD, we observe additional memory gains. Subsequently we discuss and analyze the impact of a delayed generation of successors in terms of disk accesses. Furthermore, we address the distributed, space-efficient pattern lookup, which on each client selectively loads only the PDBs that are needed to incrementally compute the heuristic estimate. This allows to load larger PDBs on individual processors. Last but not least, we propose relay A* to generate approximate solution for the 35-Puzzle. Finally, we draw conclusions.

2 The (n^2-1)-Puzzle Domain

To make our techniques explicit, we choose one specific planning benchmark, the (n^2-1)-Puzzle. It consists of (n^2-1) numbered tiles, squarely arranged, that can be slid into a single empty location, called the blank. The goal is to re-arrange the tiles such that a specific layout like the following ones is reached. For modeling the (n^2-1)-puzzle every state is represented as a vector, each of whose components corresponds to one location, indicating by which of the tiles (including the blank) it is occupied.

1	2	3
8		4
7	6	5

	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

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

	1	2	3	4	5
6	7	8	9	10	11
12	13	14	15	16	17
18	19	20	21	22	23
24	25	26	27	28	29
30	31	32	33	34	35

The state spaces consist of $9!/2 \approx 1.81 \cdot 10^5$ states for the 8-Puzzle, $16!/2 \approx 1.05 \cdot 10^{13}$ for the 15-Puzzle, and $25!/2 \approx 7.75 \cdot 10^{24}$ for the 24-Puzzle, while the set of reachable states for the 35-Puzzle consists of $36!/2 \approx 1.86 \cdot 10^{41}$ states. Optimally solving the $(n^2 - 1)$-Puzzle is NP-hard [28]. Optimal plans have been computed by [29] (8-Puzzle), [17] (15-Puzzle), and [22] (24-Puzzle). Approximate plans for the 35-Puzzle are given by [11]. A first step towards optimal solutions is the construction of 5-tile PDBs [9].

The problem can be easily encoded in PDDL [23] with object types `tile` and `position` as well as predicates `(conn ?p1 ?p2 - position)` to encode which locations are adjacent, and `(on ?t - tile ?p - position)` to encode on which position a tile is located. The sliding action is specified as follows:

```
(:action move
:parameters (?t1 - tile ?p1 ?p2 - position)
:precondition (and (on ?t1 ?p1)(conn ?p2 ?p1)
     (forall (?t2 - tile)(not (on ?t2 ?p2))))
:effect (and (not (on ?t1 ?p1))(on ?t1 ?p2)))
```

By its fast depth-first node generation and its linear depth space requirements, IDA* [17] is one of the best choices for solving the $(n^2 - 1)$-Puzzle. However, IDA* comes with limited duplicate detection, which makes it less applicable to other planning domains. For improved state space coverage, different solutions have to be found.

3 External Heuristic Search

The limitation of main memory is the major bottleneck for practical planning applications. External memory search algorithms explicitly manage the disk access, since they are more informed to predict future memory accesses than the operating system. It is common to measure such algorithms in the number of scans for a stream of records.

In external memory breadth-first search (BFS) the internal memory queue is substituted with a file. Naively running internal memory BFS in the same way on external memory results in many block file accesses for finding out whether neighboring nodes have already been visited.

The external memory BFS graph algorithm [26] is designed for undirected explicit graphs. After the preprocessing step, the graph is stored in adjacency lists. The BFS-level $i-1$ is scanned and the set of successors are put into a buffer of a size close to the main memory capacity. If the buffer becomes full, internal sorting generates a sorted duplicate-free state sequence that is flushed. The outcome of this phase is a file containing k partially sorted parts. Duplicate elimination is realized (without an internal hash table) via external sorting followed by an external scan. In the next step, external merging is applied to unify the k parts into BFS-Level i by a simultaneous scan using k buffered file pointers. Within this process duplicates are eliminated. Since the files are sorted and unless the number of file pointers (together with their internal buffers exceed main memory), the I/O complexity is determined by the time for scanning the entire file. One also has to eliminate the BFS-level $i-1$ and $i-2$ from BFS-level i to avoid re-computations. As all these levels are completely sorted, duplicates can be eliminated using a parallel scan. The process is repeated until BFS-level i becomes empty, or the goal has been found. The correctness of restricting the duplicate scope to a constant number of levels builds the basis for internal sparse-memory algorithms like breadth-first heuristic search [33], and has been extended to certain classes of directed graphs.

For implicit undirected problem spaces (like the 35-Puzzle) external memory BFS has been coined to the term breadth-first frontier search with delayed duplicate detection [18]. The algorithm has been applied to perform a complete external exploration of the 15-Puzzle with 1.4 terabytes hard disk in three weeks [21].

External A* [7] combines delayed duplicate detection, frontier search and best-first enumeration to one algorithm. The organizational structure is borrowed from SetA* [16]. External A* maintains the search space on disk. The priority queue data structure to allow extracting elements with smallest f-value is a list of buckets. In the course of the algorithm, a bucket (i, j) contains all states s with path length $g(s) = i$ and the heuristic estimate $h(s) = j$. As same states have same heuristic estimates, duplicate detection is restricted to buckets of the same h-value. Each bucket is implemented as a buffered file. If a buffer becomes full, it is flushed to disk; if it becomes empty, it is refreshed with new states from disk. Figure 1 depicts the working of external A*. We assume that for all states s and successors s' of s we have $|h(u) - h(v)| \in \{-1, 0, 1\}$.

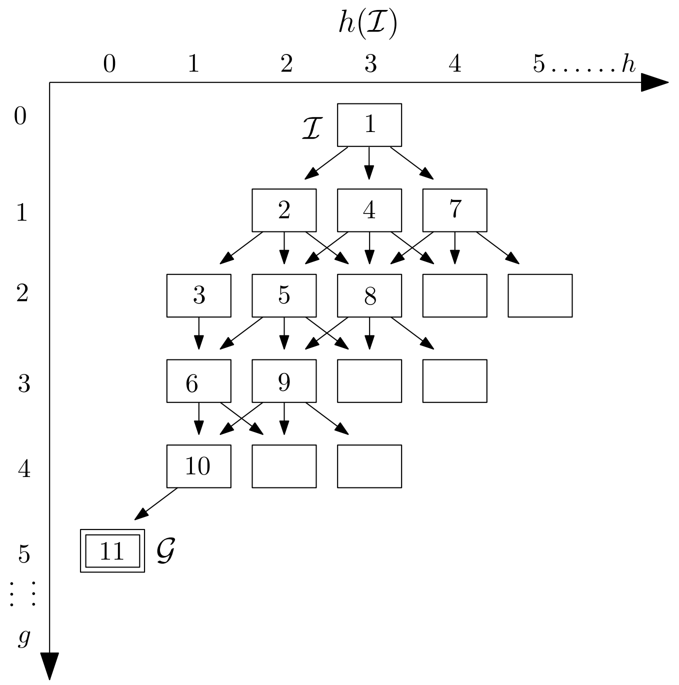

Figure 1: Buckets' selection in External A*. Rectangles represent buckets. Numbers in rectangles correspond to the order of expansion. Unnumbered buckets are not *expanded* but *generated*.

Using the Manhattan distance estimate, External A* can optimally solve all 15-Puzzle instances. For the hardest of Korf's sample instances [17] it expands 999,442,568 nodes in about 54m[1] using 17,769 megabytes of disk space. By the alternating parity and the undirected nature of the $(n^2 - 1)$-Puzzle, duplicates for bucket (i, j) can only exist in the buckets (i, j) and $(i - 2, j)$.

Similar to BFS but in difference to ordinary A*, External A* terminates while generating the goal, since all states in the search frontier with a smaller g-value

[1]We experimented on a Linux cluster, equipped with Opteron 2.2-2.6GHz processor nodes (2.6GHz 4-processor nodes with 16 GB RAM, 2.2GHz 2-processor nodes with 4 GB RAM) and 3 terabytes disk space via NFS. The time limit for 4-processor nodes is 48h, while for 2-processors it varies from 48h–480h.

have already been expanded. The core problem for solving larger puzzles are good heuristics. We will use heuristics from planning.

4 Planning Pattern Databases

If a state in a planning problem is described as a vector of state variables, the pattern variables denote a subset of these variables and induce an abstract space. A pattern is a specific assignment of values to the pattern variables. For the (n^2-1)-Puzzle domain, a pattern reflects a particular configuration of tiles. A pattern database is a lookup table containing the goal distance from any abstract state. It serves as an admissible search heuristics for the concrete state space. The size of a PDB is the number of states it contains.

According to [4] a planning abstraction corresponds to a selection of propositions as don't cares in the grounded problem description. These are removed from the action in the domain description followed by a projection of the initial and goal states. For alternative planning abstractions [15], the parameterized PDDL domain is not changed at all. For example, abstractions simply remove domain objects in the problem description followed by omitting atoms in the goal states. For pattern database construction, the initial state is irrelevant. For example, in the abstraction for the (n^2-1)-Puzzle with remaining domain objects `tile-1`, `tile-2`, `tile-3`, `tile-4` and `tile-5` the goal reduces to

```
 (on tile-1 posn-1) (on tile-2 posn-2)
 (on tile-3 posn-3) (on tile-4 posn-4)
 (on tile-5 posn-5)
```

For symbolic construction, we used a PDDL planner for which input files were generated automatically. The selection of pattern objects is provided manually, but can be automated [13]. Given the abstract instance, the planner finds the following encoding for the position of each tile fully automatically.

```
(:group-1
 (on tile-1 posn-1) (on tile-1 posn-2) ...)
(:group-2
 (on tile-2 posn-1) (on tile-2 posn-2) ...)
 ... )
```

This representation is actually the dual of the state vector [32]. By using k bits for each group ($k = 4$ for the 15-Puzzle, 5 for the 24-Puzzle, and 6 for the 35-Puzzle), $\lfloor 32/k \rfloor$ tiles fit into one 32-bit integer, resulting in $4\lceil n^2/\lfloor 32/k \rfloor \rceil$ bytes for the packed state vector.

As the search space is undirected, for the construction of each PDB we selected forward search, starting with the abstract goal states using the original operators (This state space is different to backward search with inverted operators, for which many infeasible states can be generated.)

One frequently chosen option for storing the pattern databases is a perfect hash table, which stores all goal distances in abstract space in a plain array addressed by the pattern's hash value. It relies on a bijection of a space with n states to the set $\{1, \ldots, n\}$.

5 Disjoint Planning Pattern Databases

Disjoint planning PDBs [19] allow the addition of entries with pairwise disjoint atom sets, while preserving the consistency of the estimate. As only one tile moves at a time, for the $(n^2 - 1)$-Puzzle the move action acts local in one group. (If an action modifies more than one group, it has to have zero costs in at least one abstraction.)

Given a partition of the $(n^2 - 1)$-Puzzle into tiles, disjoint PDBs can be computed fully automatically. Together with IDA* search, explicit-state disjoint PDBs are able to solve fully random 24-Puzzle instances [19]. Each of the PDBs in the *standard* disjoint 6-tile PDBs consists of 127,512,000 abstract states (because of the structural regularities, out of four pattern databases only two were actually constructed).

As already observed by [3], symmetries in the state vector allow multiple usage of disjoint PDBs. For the $(n^2 - 1)$-Puzzle the reflection along the main diagonal leads to a symmetric PDB that is addressed by posing symmetric state queries. By using larger patterns explicit-state PDB grow exponentially.

6 Symbolic Planning Pattern Databases

Compressed PDBs [10] can be constructed either by compiling the database posterior to its construction, or on-the-fly during the construction process [8]. Moreover, one can approximate the PDB content, by hash table folding [20].

Trie PDBs [30] (commonly used for storing patters in the multiple sequence alignment problem) are compact PDBs that allow sharing of state vectors. Each path from the root to a leaf corresponds to a complete scan of the state vector. Tries are either multi-variate (each branch corresponds to a state vector entry of finite domain, e.g., the tiles' label) or binary (each path corresponds to the binary representation of the state vector). They can be compressed by contracting edges that contain no branching.

Symbolic PDBs [5] are compact representations of trie PDBs on binary state vectors. The functional representation of state set is a binary decision diagram (BDD) [2]. For a given state vector, the BDD evaluates to 1, if the corresponding state is contained in the encoded set. The additional compaction refers to the two BDD reduction rule, which lead to a unique diagram representation. In terms of BDDs, a symbolic PDB can be seen as a sequence of BDDs $\mathit{PDB}_1, \ldots, \mathit{PDB}_k$ with PDB_i covering the state vector representations of all abstract states that have a h-value of i.

The symbolic construction of the standard disjoint 6-tile PDB for the 24-Puzzle (in about half an hour each) lead to PDB of 96,352,162 and 95,766,454 bytes ($\approx$ 30% savings).

In theory, the lookup in a symbolic PBDs can be faster than in a hash table, as not all state variables have to be looked at to determine, whether or not a state is contained in it. In practice, however, the retrieval in symbolic PDBs is often slower due to changing the encoding, transforming numbers into binary, calling interfaces to BDD libraries, a worse cache reputation. For external search the evaluation is sufficiently fast compared to manipulating states on disk.

Instead of compacting the explicit-state PDB, the construction process of the abstract state space itself is symbolic with the abstract goal and all abstract operators represented in form of a BDD. The abstract operators represent transition relation in partitioned form and are used to compute the one-step image of a state set [24].

7 Pattern Database Control Rules

Temporal constraints have been recently added to PDDL [12]. The constraint (`hold-during` t_1 t_2 ϕ) requires ϕ to be true in step $t_1 \leq i < t_2$. Such constraints map to timed initial literals as introduced by [14].

Using temporal constraints, the quality of PDBs can be improved. Consider a PDB for the sliding-tile puzzle with a set of pattern tiles that surround the blank in the top-left corner. We know that the last action necessarily has to move a tile that is adjacent to the blank. In the 35-Puzzle these are the `tile-1` and `tile-5`. When constructing the PDB that includes these two tiles in the pattern (starting with the goal) we simply add the following constraint to the PDDL encoding.

```
(hold-during 1 2 (or (on tile-1 posn-0)
                     (on tile-5 posn-0)))
```

The second last move may also have restrictions. Suppose that `tile-10` and `tile-6` are also contained in the pattern for the PDB. Then

```
(hold-during 2 3 (or (on tile-1 posn-0)
                   (or (on tile-10 posn-6)
                       (on tile-6 posn-6))))
```

There are some subtleties. First, temporal constraints may transform undirected search spaces to a directed ones, implying an increase of the duplicate detection scope (needed for frontier search construction). Secondly, the PDBs may loose consistency as successors in concrete space may not be in adjacent levels in the abstract space. As such successors produce sub-optimal solution, such states can safely be omitted from the search.

In planning practice, the integration of these control rules leads to an increase of radius 32 to radius 35 for the irregular 6-tile pattern database of the 24-Puzzle. We use these improved PDBs for the 24-Puzzle experiments.

8 Incremental External Search

While External A* requires a constant amount of memory due to internal read and write buffers for the files, IDA* requires memory that scales linear with the search depth. External A* removes all duplicates from the search but require slow disk to succeed. Moreover, in search practice disk space is limited, too.

Therefore, one may wish to combine advantages of IDA* and External A*. First of all, simple pruning strategies such as not generating predecessor states again, help to save external work for delayed duplicate detection. Moreover, as we will see, incremental heuristics (the successor's h-value is computed by the old h-value and the heuristic difference) stored together with the state accelerates PDB lookup time.

The integration of IDA* in External A* is as follows. Given a search strategy splitting threshold (e.g., on the f-value) starting with External A*, the remaining unexpanded buckets are only read and each state is fed into IDA* with an appropriate threshold. All IDA* searches run independently and can easily be distributed using different processors.

Split Value	Length	Nodes Generated	Time
68 (IDA*)	82	94,769,462	3m:06s
70 (Hybrid)	82	133,633,811	4m:23s
72 (Hybrid)	82	127,777,529	4m:03s
74 (Hybrid)	82	69,571,621	2m:24s
76 (Hybrid)	82	63,733,384	2m:22s
78 (Hybrid)	82	108,334,552	3m:35s
80 (Hybrid)	82	96,612,234	3m:36s
82 (Hybrid)	82	229,965,025	8m:25s
84 (External A*)	82	1.71814e+08	8m:48s

Table 1: Combining IDA* with external A* in simpler 24-Puzzle instance.

In Table 1 we show results of solving one simpler 24-Puzzles instance (40 in [19]), according to different split values. The first entry shows that (likely due to different tie-breaking and further fine-tuning) our implementation of IDA* generates more nodes than the one of Korf and Felner (65,099,578). Another important observation is that by its breadth-wise traversal ordering, External A* expands the entire f^*-diagonal, while IDA* can stop at the first goal node generated. In the sample 24-Puzzle instance, IDA* generated 94 million nodes, while External A* generated 171 million. With raising split value, we see a potential for a better algorithm in between the two.

There are, however, other 24-Puzzle instances in which pure IDA* performs better. Optimally solving instance 49 of the problem set in [19] (with an optimal step plan of 100 moves) given a split value of 94 yields 367,243,074,706 generated nodes in 217h:19m:11s and 4.9 gigabytes on disk. Pure IDA* generated in 314,556,297,173 nodes in 176h:48m:31s and 31 bytes on disk. A split at 98 resulted in 451,034,974,741 nodes in 256h:48m:47s and 169 gigabytes on disk. (Korf and Felner report 108,197,305,702 nodes generated for this instance.)

Nonetheless, this study shows that even with only one disjoint PDB on board, even hard 24-Puzzle instances are tractable with symbolic pattern databases. Therefore, in the following we concentrate on scaling the number and size of planning PDBs for solving the 35-Puzzle.

With x tiles in the pattern, the abstract state space for the 35-Puzzle consists of $36!/(36-x)!$ states. For $x = 1$ a PDB stores the Manhattan distances for the selected tile, inducing an abstract state space consisting of 36 states. The sizes for larger values of x are: $36!/34! \approx 1.26 \cdot 10^3$ ($x = 2$), $36!/33! \approx 4.28 \cdot 10^4$ ($x = 3$), $36!/32! \approx 1.41 \cdot 10^6$ ($x = 4$), $36!/31! \approx 4.52 \cdot 10^7$ ($x = 5$), $36!/30! \approx 1.40 \cdot 10^9$ ($x = 6$), and $36!/29! \approx 4.2 \cdot 10^{10}$ ($x = 7$). Assuming one byte per entry, a perfect hash-based PDB for the 35-Puzzle calls for about 1.23 kilobytes ($x = 2$), 41.83 kilobytes ($x = 3$), 1.34 megabytes ($x = 4$), 43.14 megabytes ($x = 5$), 1.3 gigabytes ($x = 6$), and 39.1 gigabytes ($x = 7$) RAM. Note that for generating the abstract state space, the search frontier additionally has to be stored, but to keep the comparison simple, we assume this frontier to be maintained on disk.

9 External Planning Pattern Databases

External PDBs [6] are constructed in layers and maintained on the hard disk. Each BFS level is flushed in form of a BDD to disk, so that the memory for representing this level can be re-used. As the BDD representation of a state set is unique, no effort for eliminating duplicates in one BFS-level is required. Before expanding a state set, however, we apply delayed duplicate detection wrt. the set of the two previous BFS-levels. As the external construction of large-scale PDBs may take long, the process can be paused and resumed at any time.

For the 35-Puzzle either seven 5-tile, five 6-tile and one 5-tile, or five 7-tile databases complete a disjoint set. The cumulated space consumption for explicit-state disjoint sets of PDBs (without exploiting structural regularities) is 302 megabytes (for $x = 5$), 6.9 gigabytes (for $x = 6$), and 195 gigabytes (for $x = 7$). For the disjoint 6-tile PDB in the 35-Puzzle the results are shown in Fig. 2. In total the additive set consumes 2.6 gigabytes; a gain of factor 2.65 wrt. the explicit construction. We generated four more additive 6-tile PDBs with sizes 4.0, 2.3, 2.3, and 3.2 gigabytes. The construction of all five disjoint 6-tile PDBs took about 50h.

Together with the space of about one gigabyte for the search buckets ($5 \cdot 10^6$ entries), one would expect a main memory requirement of more than 15 gigabytes when loading all 5 disjoint PDBs. However, about 13 gigabytes RAM were actually neede. This additional memory gain is due to loading the different layers stored on disk into a shared BDD [25]. There are also cross-PDB savings, ob-

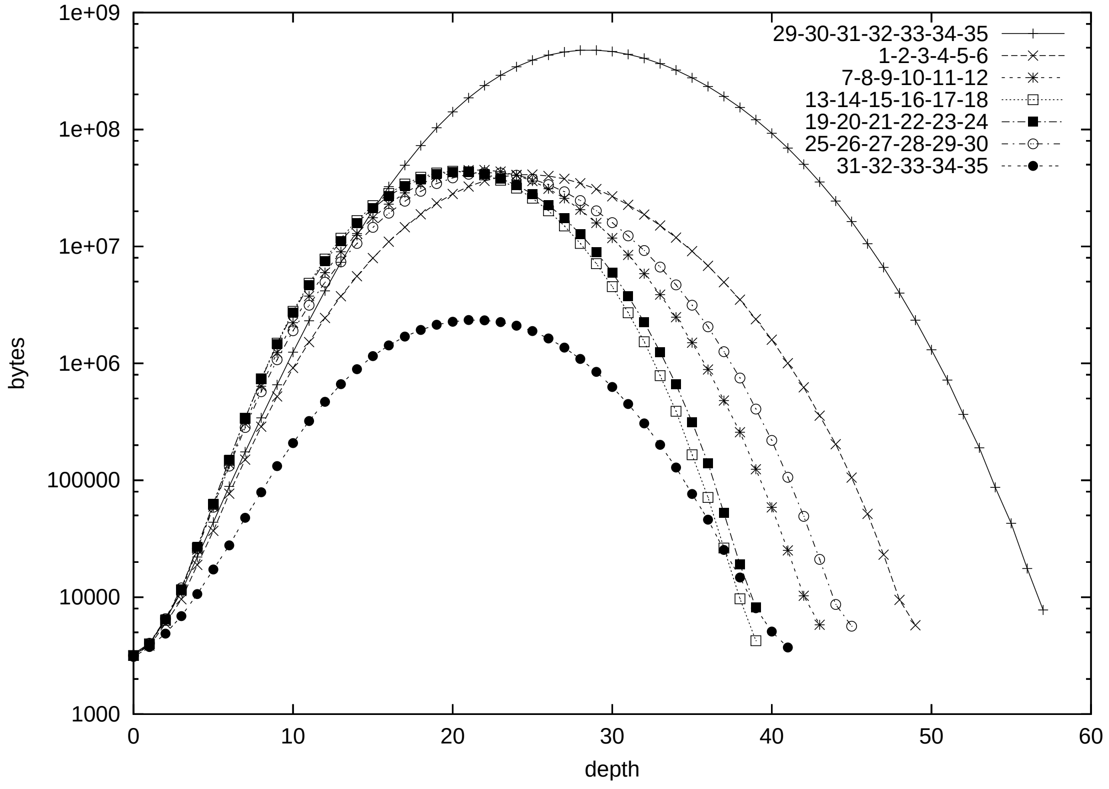

Figure 2: Memory profile for sample 5-, 6- and 7-tile symbolic PDBs (logscale).

tained by BDD sharing. These savings are due to shifting the set of BDD variable indices for every PDB to the same range, i.e., all BDDs are rooted at variable index 0.

For the construction of large PDBs, we found that the intermediate result of the symbolic successor set computation (the image) became too large to be completed in main memory. As a solution, we computed all sub-images (one for each individual move), flushed them, and computed their union in form of a binary tree using a separate program.

Fig. 2 include the memory profile for generating one 7-tile PDB[2]. The total size of the pattern database is 6.6 gigabytes which compares well with the space for explicit construction. When RAM becomes sparse, for constructing larger

[2]The full disjoint 7-tile pattern databases is not completed.

BFS levels partitioned images were calculated and unified.

Based on [20] the question arises, whether symbolic PDBs cooperate with symbolic perimeter search. For this we draw a small experiment with the planner constructing a limited-depth PDB for the full problem. The results comparing the memory need for an explicit-state search tree and breadth-first exploration is compared with symbolic exploration (residing on disk) in Figure 3.

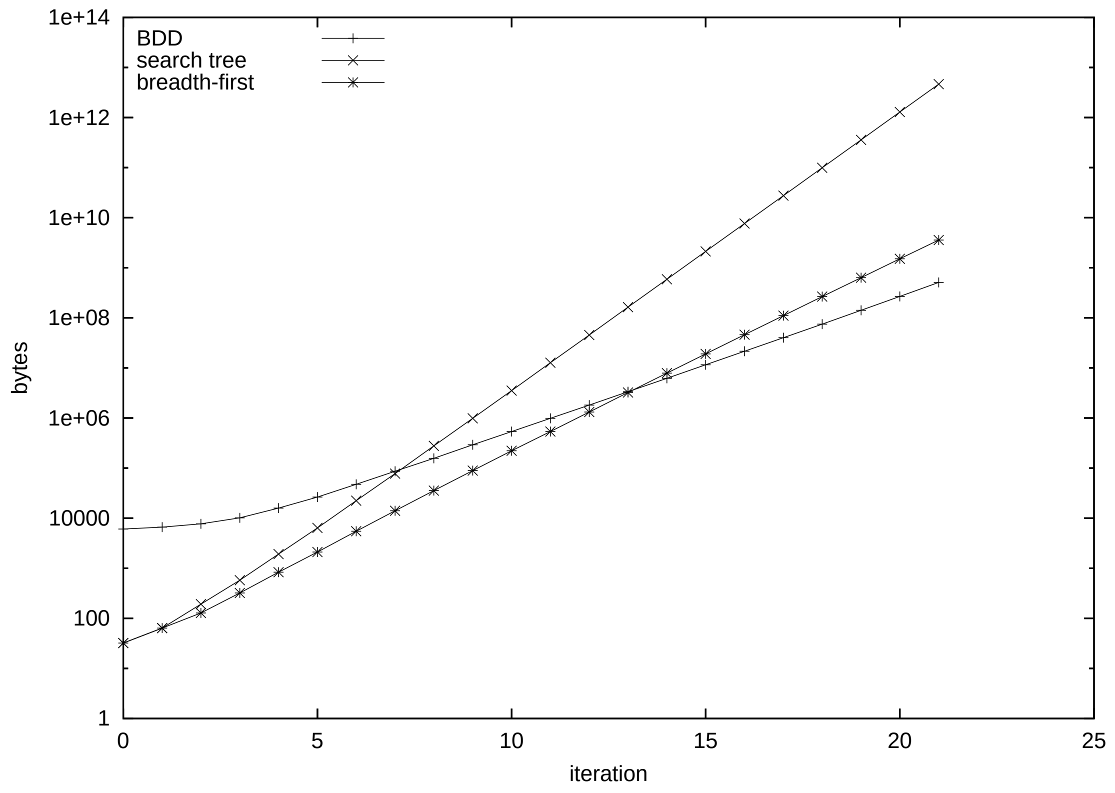

Figure 3: Memory profile for backward search (logscale).

10 Delayed Successor Generation

First, we emphasize that provided large buffers (a multi-level merge is not needed) external sorting reduces to external scanning. This is due to the fact that in most operating systems, the number of file pointers is larger than the ratio between disk and main memory and can also be increased by recompiling the kernel. Given that a single merging pass suffices, the I/O complexity is bounded by $O(|E|/B)$, with $E = \{(s, s') \mid s'$ is successor of $s \wedge f(s) \leq f^*\}$ and B being the block size.

We have extended external memory A* by expanding each f-diagonal twice. In the first pass, only the successors on the active diagonal are generated, leaving out the generation of the successors on the $(f + 2)$-diagonal. In the second pass, the remaining successors on the $(f + 2)$-diagonal are generated. We can avoid computing the estimate twice, since all successor states that do not belong to the bucket $(g + 1, h - 1)$, belong to $(g + 1, h + 1)$. In the second pass we can, therefore, generate all successors and subtract bucket $(g + 1, h - 1)$ from the result. The I/O complexity of this strategy decreases to $O(|E'|/B)$, with $E' = \{(s, s') \mid s'$ is successor of $s \wedge f(s') \leq f^*\}$.

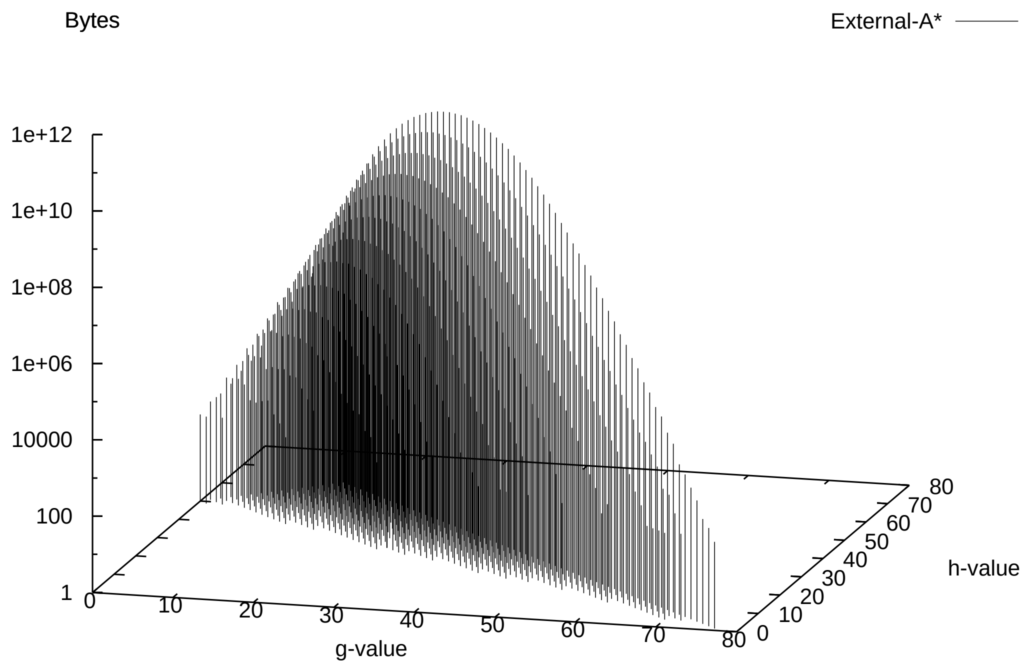

Figure 4: Memory profile external memory A* (logscale) for solving a fully random instance of the 35-Puzzle with symbolic PDBs.

Fig. 4 shows the memory profile of external memory A* for the partially ran-

CPU Time	Total Time	RAM
12h:32m:20s	48h:00m:10s	4,960,976 kilobytes
9h:55m:59s	48h:00m:25s	4,960,980 kilobytes
8h:35m:43s	45h:33m:02s	4,960,976 kilobytes
8h:45m:53s	39h:36m:22s	4,960,988 kilobytes
10h:25m:31s	46h:31m:35s	4,960,976 kilobytes
11h:38m:36s	48h:00m:40s	4,960,988 kilobytes
8h:04m:14s	26h:37m:30s	4,960,984 kilobytes
66h:58m:16s	302h:19m:44s	

Table 2: Exploration results of external memory A* for 3 disjoint 3- and 3 disjoint 5-tile PDBs.

dom instance:

9	10	8	2		7
3	1	12	4	14	13
5	11	6	15	16	17
18	19	20	21	22	23
24	25	26	27	28	29
30	31	32	33	34	35

The exploration started in bucket $(50, 0)$ and terminated while expanding bucket $(77, 1)$. For this experiment three disjoint 3-tile and three disjoint 5-tile PDBs were loaded, which, together with the buckets for reading and flushing consumed about 4.9 gigabytes RAM. The total disk space taken was 1,298,389,180,652 bytes, or 1.2 terabytes, with a state vector of 188 bytes[3]. The exploration took about 2 weeks (cf. Table 2). As the maximum time for one process is 2 days, we had to resume the algorithm six times – a facility offered by the external A* algorithm. The observed discrepancy between processor and total time can be improved using pipelining [1].

As we work on f-diagonal twice, we avoid the generation of nodes for diagonal $f^* + 2$. For the 35-Puzzle this roughly saves disk traffic by a factor of about 2.22. As a consequence, external memory A* without support for delayed successor generation would have required about 3.9 terabytes disk space.

[3]$32 + 2 \times (6 \times 12 + 6 \times 1) = 188$ bytes: 32 bytes for the state vector + information for incremental heuristic evaluation: 1 byte for each value stored, multiplied by 6 sets of at most 12 PDBs + 1 value each for their sum. Factor 2 is due to symmetry lookups.

CPU Time	Total Time	RAM
20h:57m:43s	48h:00m:35s	13,164,060 kilobytes
2h:18m:47s	5h:08m:44s	13,164,068 kilobytes
18h:28m:38s	42h:23m:08s	13,164,064 kilobytes
22h:55m:48s	45h:33m:47s	13,164,068 kilobytes
12h:38m:31s	24h:16m:32s	13,164,068 kilobytes
77h:19m:27s	165h:22m:46s	

Table 3: Exploration results of external memory A* for three disjoint 5-tile and five disjoint 6-tile PDBs.

In Table 3, we solved the instance again, now with the same three disjoint 5-tile PDBs and additionally with the five disjoint 6-tile pattern databases. The total exploration time reduces to about a week. The hard-disk consumption accumulated to 505 gigabytes with a state of 160 bytes.

11 Parallel Pattern Databases

For distributed construction of disjoint PDBs, no inter-process communication is needed. For computing the partitioned image, the workload can also be distributed. Each process works on the image for a subsets of moves and set unions are computed similar to parallel addition.

The single-process version faces the problem of high memory consumption due to large PDBs. Lazy loading PDBs on demand significantly slows down the performance (loading the large disjoint set from disk takes about half an hour). Instead, we decided to distribute the heuristic evaluation to multiple processes across the network.

The distributed version of the external memory A* works as follows. As buckets are fully expanded, and the order within a bucket does not matter, we can share the work for expansion, evaluation and duplicate elimination. We chose one master to expand and distribute the states in a bucket, and parallelized the heuristic evaluation for them to 35 client processes p_i, each one responsible for one tile i for $i \in \{1, \ldots, 35\}$. All client processes operate individually and communicate with the master via shared files.

During the expansion of a bucket (see Fig. 5), the master writes a file T_i for each client process p_i, $i \in \{1, \ldots, 35\}$. The file T_i contains all the successor gen-

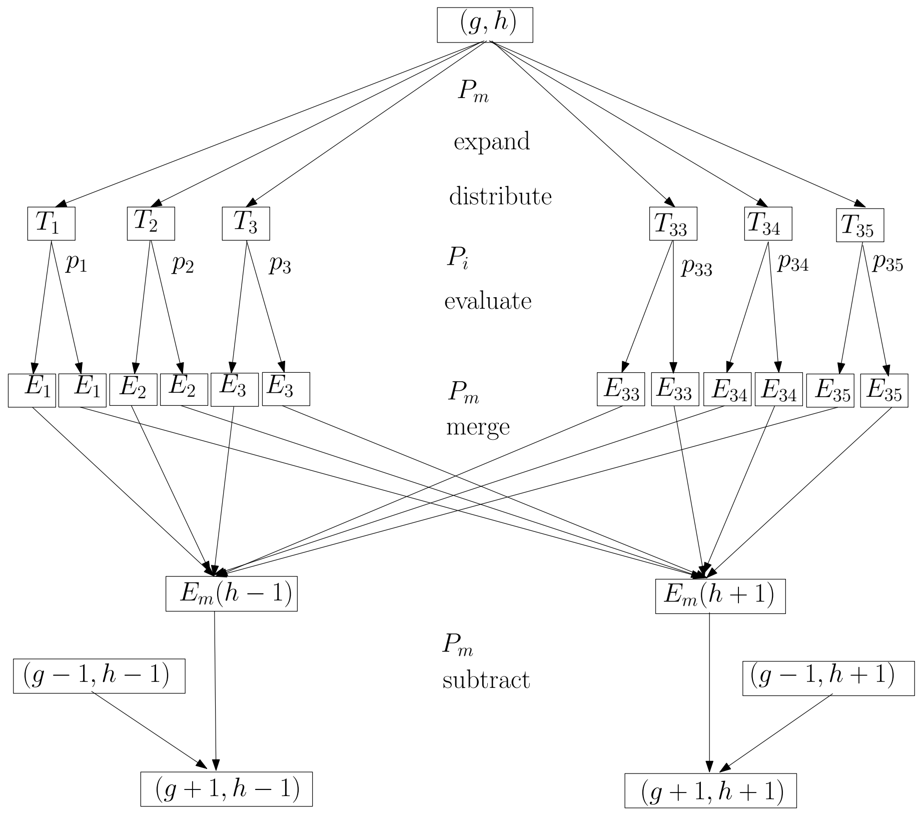

Figure 5: Distributed expansion/evaluation of one bucket.

erated by moving the i-th tile. Once it has finished the expansion of a bucket, the master p_m announces that each p_is should start evaluating its file T_i. Additionally, the client is informed on the current g- and h-value. After that, the master p_m is suspended, and waits for all p_i's to complete their task. To prevent the master from load, no sorting takes place in this phase.

Next, the client processes start evaluating the file T_i and distribute their results into the files $E_i(h-1)$ and $E_i(h+1)$, depending on the observed difference in the h-values. All files E_i are additionally sorted to eliminate duplicates; internally (when a buffer is flushed) and externally (for each generated buffer). As only 3 buffers are needed (one for reading and two for writing) internal buffers can be large.

After its evaluation, each process p_i is suspended. When all clients are done, the master p_m is resumed to merge the individual $E_i(h-1)$ and $E_i(h+1)$ files into

$E_m(h-1)$ and $E_m(h+1)$. The merging preserves the order in the files $E_i(h-1)$ and $E_i(h+1)$, so that the files $E_m(h-1)$ and $E_m(h+1)$ are sorted with all bucket duplicates eliminated. The subtraction of the bucket $(g-1, h-1)$ from $E_m(h-1)$ and $(g-1, h+1)$ from $E_m(h+1)$ now eliminates duplicates from the search using a parallel scan of both files[4].

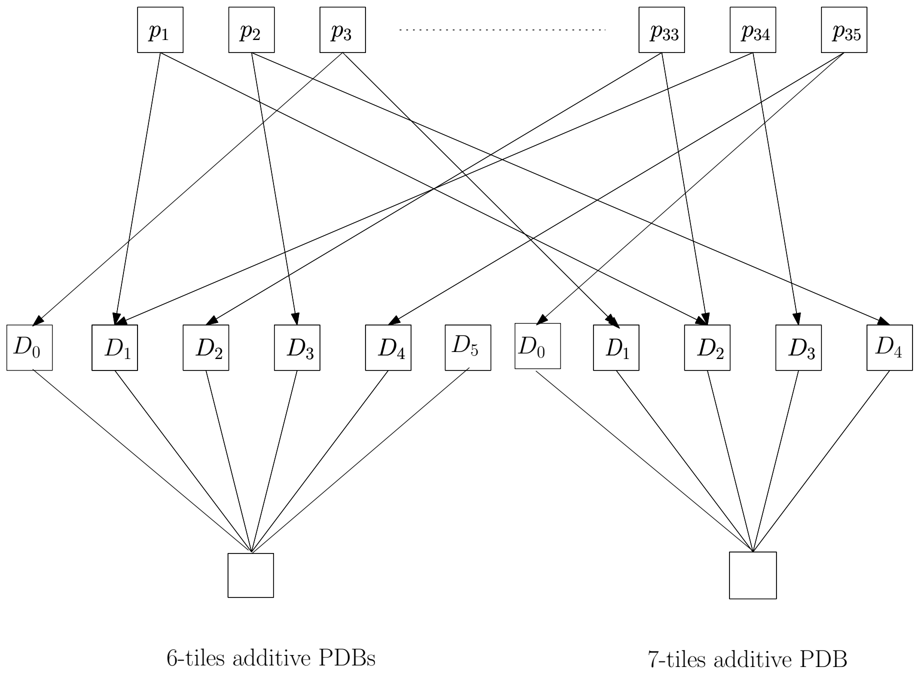

Figure 6: Selection of PDBs for evaluation.

Besides the potential for speeding up the evaluation, the chosen distribution mainly saves space. On the one hand, the master process does not need any additional memory for loading PDBs. It can invest all its available memory for internal buffers required for the distribution, merging and subtraction of nodes. On the other hand, during the entire lifetime of client process p_i, it has to maintain only the PDB D_j that includes tile i in its pattern (see Fig. 6). This saves RAM by a factor[5] of about 6 for $x = 6$ and about 5 for $x = 7$.

[4]A refined implementation only concatenates the E- files using the system command `cat`. As the concatenated files are partially presorted, the merging and subtraction process used for the duplicate elimination of the entire bucket then catches all duplicates.

[5]The consumption is doubled if symmetry lookups are used.

We started two parallel external memory A* explorations for solving the half instances using 35 clients and one master process. The individual RAM requirements for the clients reduced to 1.3 gigabytes so that 2 processes could be run on one node. This proves that a considerable amount of RAM can be saved on a node using parallel execution - the most critical resource for the exploration with planning PDBs. The first exploration (first 17 tiles permuted) took 3h:34m and 4.3 gigabytes to complete, while the second exploration (last 18 tiles permuted) took 4h:29m and 19 gigabytes.

As the above instance is moderately hard (the mean Manhattan distance in the Puzzle is about 135 [9]) we compared the single-process with the distributed version in the following fully random instance:

27	20	35	30	33	26
32	12	4	15	6	23
3	5	8	34	14	29
22		21	7	31	28
25	2	10	1	24	16
18	17	13	9	11	19

The single-process version used the disjoint 5- and 6 tile PDBs, while the parallel version took the 3- and 5-tile PDBs. In two days, the distributed version found its best plan at $(87, 75)$ with 338, while the single version found $(85, 77)$ with 270 gigabytes, so that node generation in the distributed version was slightly faster even though it generated many intermediate files. The masters' CPU time accumulated to less than 1/6 of its total time (using 2.2 gigabytes RAM).

For large buckets the partition based on the tile that moves gives an almost uniform distribution of the state sets so that no additional load balancing between the clients is needed. For the very small savings in time we blame the NFS file system for being slow. The clients' CPU time almost matches their total time, showing that they mostly wait for the master, without wasting time for writing and reading data.

12 External Memory Relay A*

We have also solved the following instance consisting of a random permutation of the upper and lower part with the mentioned three disjoint 3-tile and three disjoint 5-tile PDBs:

7	1	12		3	2
6	4	9	8	10	14
13	15	16	5	17	11
33	21	26	24	22	25
18	20	34	32	35	28
19	30	23	31	27	29

We found optimal plans for the first half using 55 steps in about 10m total time; as well as for the second half of the puzzle using 66 steps in about 40m total time. As the other half remains untouched, this establishes a relay solution of 121 steps. Since the goal distance for the original problem is larger than either of the two, in theory, we obtain a 2-approximation. In practice, the approximation is much better as the following experiment shows. We terminated external memory A* with the disjoint 5- and 6-tile PDBs for solving the full instance at the $f = 121$ diagonal after generating 1.3 terabytes disk space (in about 8 days). The best two states in diagonal $f = 99$ had a h-value of 22. When solving these remaining problems from scratch, we quickly established a minimum of 42 moves giving rise to an upper bound of $77 + 42 = 119$ steps, such that the optimal solution length is an odd number in the interval $[101, 119]$.

Last, but not least, we solved the fully random problem in a relay fashion. As we have not implemented a resume support for the distributed version yet, we invoked the single-process version three times. Figure 7 illustrates the memory profile for a relay A*. We see three exploration peaks. When the search has consumed too much disk space or time, it was restarted with the bucket having the lowest h-value. The initial h-value is 152 and the obtained interval for the optimal solution length is $[166, 214]$. This large-scale exploration consumed $2,566,708,604,768+535,388,038,560+58,618,421,920$ bytes ≈ 2.9 terabytes in about 3 weeks, presumingly the largest successful exploration in planning.

13 Conclusion

In this paper external memory, iterative-deepening and distributed versions of A* have been applied to solve partial and fully randomized instances of the $(n^2 - 1)$-Puzzle with large symbolic planning PDBs either optimally or approximately (providing tight lower and upper bounds on the plan length). Even though fully random instances of the 35-Puzzle could not yet be solved step-optimally yet, new trade-offs for heuristic search planning with planning pattern databases have

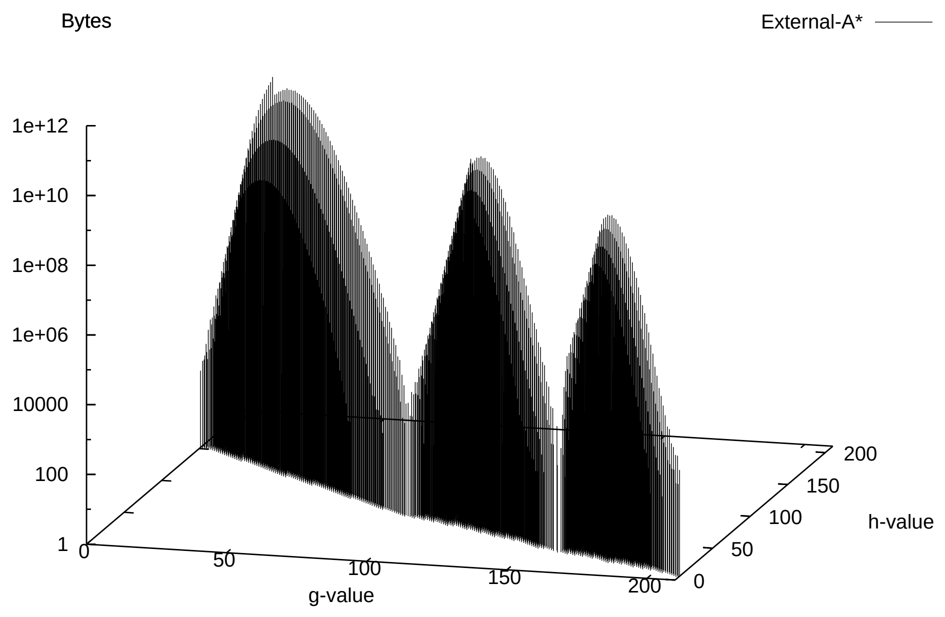

Figure 7: Memory profile of external memory relay A* for for solving a fully random instances of the 35-Puzzle with symbolic PDBs (on a logarithmic scale).

been provided. Substantial memory savings have been achieved in using BDDs. The gain seem to raise with the sizes of the PDBs. The succinctness of the state encoding of the $(n^2 - 1)$-Puzzle limits the structural advantage of BDDs – larger savings are expected for other planning domains.

References

[1] D. Ajwani, R. Dementiev, and U. Meyer. A computational study of external memory BFS algorithms. In *SODA*, pages 601–610, 2006.

[2] R. E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):142–170, 1992.

[3] J. C. Culberson and J. Schaeffer. Pattern databases. *Computational Intelligence*, 14(4):318–334, 1998.

[4] S. Edelkamp. Planning with pattern databases. In *European Conference on Planning (ECP)*, pages 13–24, 2001.

[5] S. Edelkamp. Symbolic pattern databases in heuristic search planning. In *AIPS*, pages 274–293, 2002.

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

[7] S. Edelkamp, S. Jabbar, and S. Schrödl. External A*. In *German Conference on Artificial Intelligence (KI)*, pages 233–250, 2004.

[8] A. Felner and A. Alder. Solving the 24 puzzle with instance dependent pattern databases. In *SARA*, pages 248–260, 2005.

[9] A. Felner, R. Korf, and S. Hanan. Additive pattern databases. *Journal of Artificial Intelligence Research*, 22:279–318, 2004.

[10] A. Felner, R. Meshulam, R. C. Holte, and R. E. Korf. Compressing pattern databases. In *AAAI*, pages 638–643, 2004.

[11] D. Furcy and S. Koenig. Scaling up WA* with commitment and diversity. In *IJCAI*, pages 1521–1522, 2005.

[12] A. Gerevini and D. Long. Plan constraints and preferences in PDDL3. Technical report, Department of Electronics for Automation, University of Brescia, 2005.

[13] P. Haslum. Domain-independent construction of pattern database heuristics for cost-optimal planning, 2007. Personal communications.

[14] J. Hoffmann and S. Edelkamp. The deterministic part of IPC-4: An overview. *Journal of Artificial Intelligence Research*, 24:519–579, 2005.

[15] J. Hoffmann, A. Sabharwal, and C. Domshlak. Friends or foes? An AI planning perspective on abstraction and search. pages 294–304, 2006.

[16] R. M. Jensen, R. E. Bryant, and M. M. Veloso. SetA*: An efficient BDD-based heuristic search algorithm. In *AAAI*, pages 668–673, 2002.

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

[18] R. E. Korf. Breadth-first frontier search with delayed duplicate detection. In *MOCHART*, pages 87–92, 2003.

[19] R. E. Korf and A. Felner. *Chips Challenging Champions: Games, Computers and Artificial Intelligence*, chapter Disjoint Pattern Database Heuristics, pages 13–26. Elsevier, 2002.

[20] R. E. Korf and A. Felner. Recent progress in heuristic search: A case study of the four-peg towers of hanoi problem. In *IJCAI*, pages 2324–2329, 2007.

[21] R. E. Korf and T. Schultze. Large-scale parallel breadth-first search. In *AAAI*, pages 1380–1385, 2005.

[22] R. E. Korf and L. A. Taylor. Finding optimal solutions to the twenty-four puzzle. In *AAAI*, pages 1202–1207, 1996.

[23] D. McDermott. The 1998 AI Planning Competition. *AI Magazine*, 21(2), 2000.

[24] K. McMillan. *Symbolic Model Checking*. Kluwer Academic Press, 1993.

[25] S. Minato, N. Ishiura, and S. Yajima. Shared binary decision diragram with attributed edges for efficient boolean function manipuation. In *Design Automation Conference (DAC)*, pages 52–57, 1990.

[26] K. Munagala and A. Ranade. I/O-complexity of graph algorithms. In *Symposium on Discrete Algorithms*, pages 687–694, 1999.

[27] J. Pearl. *Heuristics*. Addison-Wesley, 1985.

[28] D. Ratner and M. K. Warmuth. The $(n^2 - 1)$-puzzle and related relocation problems. *Journal of Symbolic Computation*, 10(2):111–137, 1990.

[29] P. D. A. Schofield. Complete solution of the eight puzzle. In *Machine Intelligence 2*, pages 125–133. Elsevier, 1967.

[30] S. Schroedl. An improved search algorithm for optimal multiple sequence alignment. *Journal of Artificial Intelligence Research*, 23:587–623, 2005.

[31] M. Valtorta. A result on the computational complexity of heuristic estimates for the A* algorithm. *Information Sciences*, 34:48–59, 1984.

[32] U. Zahavi, A. Felner, R. Holte, and J. Schaeffer. Dual search in permutation state spaces. In *AAAI*, pages 1076–1081, 2006.

[33] R. Zhou and E. A. Hansen. Breadth-first heuristic search. *Artificial Intelligence*, 170(4-5):385–408, 2006.