

Building Large Compressed PDBs for the Sliding Tile Puzzle

Robert Döbbelin, Thorsten Schütt^(✉), and Alexander Reinefeld

Zuse Institute Berlin, Berlin, Germany

schuett@zib.de

<http://www.zib.de>

Abstract. The performance of heuristic search algorithms depends crucially on the effectiveness of the heuristic. A pattern database (PDB) is a powerful heuristic in the form of a pre-computed lookup table. Larger PDBs provide better bounds and thus allow more cut-offs in the search process. We computed 9-9-6, 9-8-7, and 8-8-8 PDBs for the 24-puzzle that are three orders of magnitude larger (up to 1.4TB) than the 6-6-6-6 PDB. This was possible by performing a parallel breadth-first search in the compressed pattern space. Our experiments indicate an average 8-fold improvement of the 9-9-6 PDB over the 6-6-6-6 PDB on the 24-puzzle. Combining several large PDBs yields a 13-fold improvement.

1 Introduction

Heuristic search algorithms are widely used to solve combinatorial optimization problems. While traversing the problem space, the search process is guided by a heuristic function that provides a lower bound on the cost to a goal state. This allows to prune large parts of the search space and thus reduces the overall search effort. The more accurate the heuristic is, the more states can be pruned in the search. *Pattern Databases (PDBs)* are powerful heuristic functions in form of a lookup table. They store the exact solution of a relaxed version of the problem. The less the original problem is relaxed the larger is the size of the PDB and thereby the tighter are its bounds.

In this paper we present for the first time very large complete PDBs for the 24-puzzle: a 8-8-8 PDB with 122 GB, a 9-8-7 PDB with 733 GB, and a 9-9-6 PDB with 1381 GB. The largest one gave node savings by up to a factor of 37 compared to the 6-6-6-6 PDB [11].

We present a parallel algorithm that performs a breadth-first search in the compressed pattern space and thereby allows to compute very large PDBs on compute clusters with a modest amount of memory. The application of such large PDBs in heuristic search, however, requires a computer that allows to load the whole PDB into main memory. This can be as much as 1.4 TB for the 9-9-6 PDB, for example. While systems with more than 1 TB of main memory are not yet common, we believe that our work will help in studying the pruning-power of large PDBs.

The remainder of this paper is structured as follows. Section 2 sets the context of our work by reviewing relevant literature. Thereafter, PDBs are introduced in Sect. 3 and the algorithms and compressed data structures for generating large PDBs are presented in Sect. 4. In Sect. 5 we provide a statistical and empirical analysis and we summarize our work in Sect. 6.

2 Background

PDBs were first mentioned by Culberson and Schaeffer [3] and have been improved by several researchers. For instance, Felner *et al.* [8] presented additive PDBs in which the heuristic estimate is computed as the sum of the values of several smaller PDBs. The same authors also proposed a method for compressing a PDB for sliding tile puzzles by disregarding the blank tile and by computing the minimum distance over all possible blank positions. PDBs can be built with a backward breadth-first search over the complete state space. Large breadth-first searches have been used by Korf and Schultze [12] to expand the complete graph of the 15-puzzle for the first time. This was achieved by keeping the search front on disks and hiding the disk latency with multiple threads.

Orthogonal to additive PDBs is the idea of Holte *et al.* [9] to take the maximum h -value from several smaller PDBs instead of a single large one. They show that the accuracy of small h -values is especially important for reducing the number of expanded nodes.

Felner and Adler [6] use instance dependent PDBs to utilize large PDBs without completely creating them. They build on the observation of Zhou and Hansen [13] that only the nodes generated by the best-first search algorithm A^* are needed in the pattern space to solve an individual instance. For each pattern of the given instance, Felner and Adler perform an A^* search from the goal pattern towards the start pattern until the available memory is exhausted. This database is then used for the forward search. When h -values are missing, several smaller PDBs are used instead.

Breyer and Korf [1] apply a dense representation for problem spaces [2] to pattern databases. They store the heuristic estimates modulo three and restore the actual h -value during search. This results in a new compression technique using 1.6 bits per entry in the PDB.

Edelkamp *et al.* [4] created large symbolic pattern databases using an external breadth-first search with Binary Decision Diagrams (BDDs). They built a set of 7-tile PDBs for the 35-puzzle with a total size of 195 GB.

3 Pattern Databases

In this paper we are concerned with sliding tile puzzles. An instance of the $(n - 1)$ -puzzle can be described by n state variables, one for each tile. Each *state variable* describes the position of one specific tile in the tray. A *pattern* considers only a subset of the state variables; the remaining state variables are ignored. Hence, patterns abstract from the original problem by mapping several states to

the same point in the pattern space. The number of ignored state variables can be used to control the information loss.

In the $(n-1)$ -puzzle, a pattern is defined by a subset of the tiles. The position of the pattern tiles, the *pattern tile configuration*, and the blank defines a node in the pattern space. Move operations in the original problem can be analogously applied to nodes in the pattern space by moving either a pattern tile or a non-pattern tile, i.e. a *don't care tile*. Although we count the moves of don't cares, they are indistinguishable from each other. The size of the pattern space for a pattern with k tiles for the $(N-1)$ -puzzle is $\frac{N!}{(N-k-1)!}$.

The number of moves needed to reach the goal in the pattern space can be used as an admissible heuristic for the move number in the original space. Because of the don't care tiles, a path in the original search space can only be longer than the corresponding path in the pattern space and hence the heuristic is admissible, i.e. non-overestimating.

To compute a PDB, we perform a backward breadth-first search from the goal to the start node and record for each visited node the distance from the goal.

3.1 Additive Pattern Databases

Because of space limitations, only small PDBs can be built. To get better heuristic estimates, several PDBs must be combined. However, with the above method, which also counts the movements of don't care tiles, we cannot simply add the h values of PDBs, even when the patterns are disjoint, because the same move would be counted several times. For additive PDBs [7] we only count the moves of pattern tiles.

The search space is mapped to the pattern space in the following way. Two states of the original space map to the same state in the pattern space, if the pattern tiles are in the same position and the two blank positions can be reached from each other by moving only don't care tiles. There is an edge between two nodes a and b in the pattern space if and only if there are two nodes c and d in the puzzle space where c maps to a and d maps to b and there is an edge between c and d .

Figure 1 shows an example for the 8-puzzle. Positions (a) and (b) map to the same state in the pattern space, because the blank positions are reachable from each other without moving pattern tiles. Positions (a) and (c), in contrast, do not map to the same state in the pattern space, because at least one pattern tile must be moved to shift the blank to the same position.

To further reduce the memory consumption, we compress the databases by the blank position as described in [7]. This is done by storing for any pattern-tile configuration, independent of the different blank positions, only the minimal distance from the goal node. For the three examples shown in Fig. 1 we only store one (the smallest) distance g in the PDB.

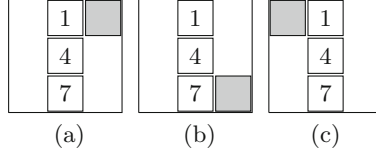


Fig. 1. Patterns with different blank positions (8-puzzle).

4 Building Compressed PDBs

When building large PDBs we ran into two limits: space and time. Not only do we need to keep the PDB itself in main memory, but also the Open and Closed lists must be stored. In Sect. 4.1 we describe a sequential algorithm and a compressed data structure for computing large PDBs. In Sect. 4.2 we describe a parallel implementation that uses the combined memory and compute capacity of a cluster as a single resource.

4.1 Sequential Algorithm

Our algorithm for building PDBs builds on ideas of [2]. To store the k -tile PDB, we use an array of $\frac{N!}{(N-k)!}$ elements, one entry for each state of the compressed pattern space. For our 9-tile PDB this results in $\frac{25!}{16!} = 741 \cdot 10^9$ entries. We use a perfect hash function to map a configuration of the pattern to this array. The hash function is reversible so that we can map an array index back to its pattern tile configuration. Each entry in the array is made up of three values: **g**, **open_list** and **closed_list**.

```

struct {
    byte g;
    byte open_list;
    byte closed_list;
} array_entry;

```

The variable **g** in Algorithm 1 stores for each entry the minimum g in which we found that state. Additionally, we need to store for each tuple of a pattern tile configuration and blank position whether it is in the Open or in the Closed list. This could be done by simply storing two bit strings of length $N - k$ in each PDB entry and setting the responsible bit whenever a new blank position is visited.

However, this simple approach can be improved to achieve a further data compression. A *blank partition* is a set of blank positions with a common pattern tile configuration where all blank positions are reachable from each other by only moving don't care tiles [5]. This is shown in Fig. 1: (a) and (b) belong to the same blank partition, while (a) and (c) do not. For patterns with 9 tiles, the pattern tile configurations have no more than 8 blank partitions. We can simply enumerate the blank partitions and only store one bit for each partition

Algorithm 1 BFS in compressed, indexed PDB space.

```

1: PDBArray A
2: initialize array
3: expandedNodes = -1;
4: g = 1;
5: while expandedNodes ≠ 0 do
6:   expandedNodes = 0;
7:   for i = 0 → A.size - 1 do
8:     if A[i].open_list = ∅ then
9:       continue;
10:    end if
11:    expandedNodes++;
12:    pattern = unindex(i);
13:    blanks = unpackBlanks(A[i].open_list);
14:    succs = genSuccs(pattern, blanks);
15:    for j = 0 → succs.size - 1 do
16:      sIndex = index(succs[j]);
17:      rBlanks = reachableBlanks(succs[j]);
18:      pBlanks = packBlanks(rBlanks);
19:      pBlanks -= A[sIndex].closed_list;
20:      A[sIndex].open_list += pBlanks;
21:      A[sIndex].g = min(A[sIndex].g, g);
22:    end for
23:    A[i].closed_list += A[i].open_list;
24:    A[i].open_list = ∅;
25:  end for
26:  g++;
27: end while

```

in the *open_list* or *closed_list*. In the backward breadth-first search we used pre-computed lookup tables to map the blank positions to blank partitions. To build a PDB with up to 9 tiles, this scheme requires 3 bytes per state, one for *g*, *open_list*, and *closed_list*, respectively.

The breadth-first search over the pattern space is performed as follows (Algorithm 1): All *open_lists* and *closed_lists* are initialized with zeroes. The *g* for each state is set to the maximum value. For the initial state, the blank partition of the initial position is set in the *open_list*.

Then the array is scanned repeatedly (line 4). For each entry, we check if the Open list is empty (line 7). If not, we create the pattern tile configuration (line 11), extract all blank positions from the Open list (line 12) and finally generate the successors (line 13). For each successor, we calculate the index in the PDB (line 15), compress the blank positions (line 16-17) and update the successor's entry in the PDB (line 18-20). Note that backward steps are eliminated with the update. Finally, we update the *open_list* and *closed_list* of the current position. This is repeated until the complete pattern space has been visited. Note that the final PDB is stored using one byte per entry. The two bytes used for *open_list* and *closed_list* can be discarded.

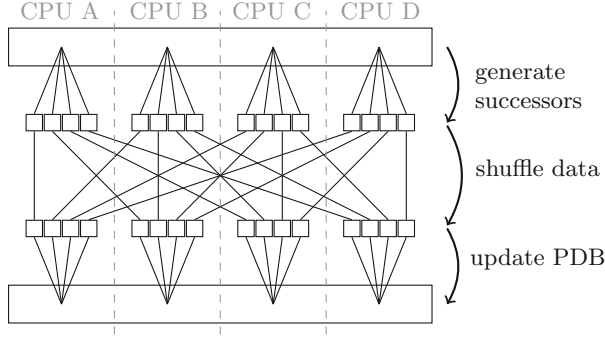


Fig. 2. Workflow of the parallel implementation.

4.2 Parallel Algorithm

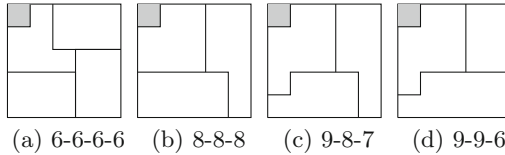
For the parallel algorithm, we distribute the array (in disjoint partitions) over all compute nodes. To avoid imbalances in the work load, we do not assign contiguous parts to the nodes but use a hash function for assigning partitions of the array to the compute nodes. The parallel algorithm has the same structure as the sequential algorithm (see Fig. 2) but it needs additional communication to move the results to remote compute nodes.

For each g , first each node scans its part of the array and generates the successors as described in Algorithm 1. But instead of directly updating the PDB, each node collects the successors locally. In the shuffle phase (Fig. 2), these successors are sent to the nodes storing the corresponding partitions in the PDB. Finally, the PDB is updated locally.

Dealing with Memory Limitations. The parallel implementation requires more memory than the sequential algorithm, because successors are cached locally before they are stored in the array. The generated successors in a large search front could exceed the available memory of a compute node. Thus, we implemented the following scheme to bound the overall memory consumption. If a processor is about to run out of memory, it stops scanning the array and raises a flag. In this case all processor mark updates to the Open lists in the BFS array as new, g is not incremented and the array is scanned again. Then only those Open lists are considered, which are not marked as new. Once all processors succeeded scanning the array, the flag is removed from all Open lists and the algorithm proceeds with the next g .

5 Evaluation

We used the presented parallel algorithm to build three large PDBs, 8-8-8, 9-8-7, and 9-9-6, with sizes of 122 GB, 733 GB and 1381 GB, respectively. For comparison, the 6-6-6-6 PDB has a size of only 488 MB.



In our cluster, each compute node has 2 quad-core Intel Xeon X5570 with 48 GB of main memory. It took about 6 hours to build a single 9-tile PDB on 255 nodes. The maximum amount of memory required to build such a PDB was 3 TB. For the empirical analysis we used an SGI UV 1000, a large shared-memory machine with 64 octo-core Intel Xeon X7560 and 2 TB of main memory.

In the following, we first present a statistical analysis of the performance of our PDBs on a large number of randomly generated positions. Thereafter we show the performance on Korf’s set of random 24-puzzle instances [12]. In both cases, we used mirroring [3] to improve the accuracy of the heuristics.

5.1 Statistical Evaluation

We created 100,000,000 random instances of the 24-puzzle and recorded the h -values obtained with the 6-6-6-6, 8-8-8, 9-8-7, and 9-9-6 PDB. Figure 3 shows the cumulative distribution, i.e. the probability $P(X \leq h)$, that the heuristic value for a random state is less or equal to h . The higher the h -value, the better the pruning power of the heuristic. This is because all heuristics are admissible, i.e. they never overestimate the goal distance. Higher h -values represent therefore tighter bounds on the true value. As can be seen in Fig. 3, all graphs lie close together and their order corresponds to the size and pruning power of the PDBs. Interestingly, the new PDBs are distinctively better than the 6-6-6-6 PDB (see the dashed line).

Note that the increased number of small h -values is especially important for the performance of the heuristic [9]. Figure 4 shows a magnification of the lower left corner of the data in Fig. 3. It can be seen that all curves are clearly distinct and that the large PDBs provide a considerable improvement over the 6-6-6-6 PDB.

Table 1 lists the average, minimum, and maximum values. In accordance with Fig. 3, larger PDBs return on average a higher h -value. Checking the extreme values reveals an interesting fact: While the minimum value of the 9-9-6 PDB is 4 moves higher than the lowest value of the 6-6-6-6, its maximum value is only

Table 1. Average, minimum and maximum h -values of 100,000,000 random instances.

PDB	Size [GB]	Avg. h	Min. h	Max. h
6-6-6-6	0.488	81.85	40	115
8-8-8	122	82.84	40	116
9-8-7	733	83.10	43	116
9-9-6	1381	83.56	44	116

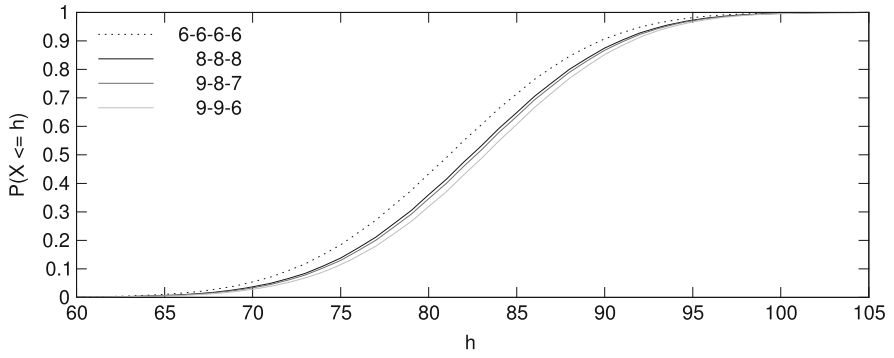


Fig. 3. Cumulative distribution of h -values of 100,000,000 random samples.

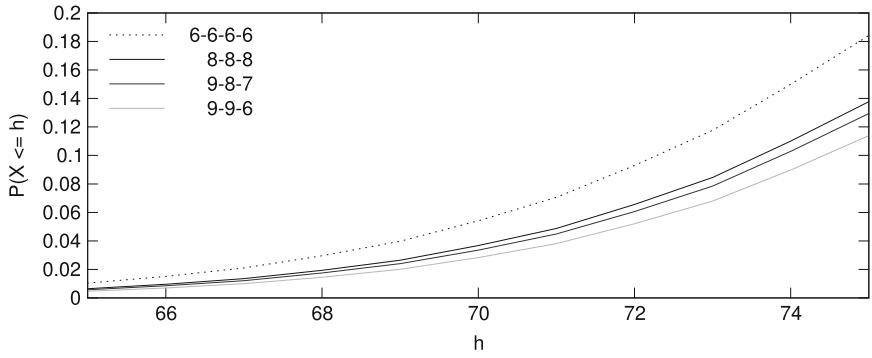


Fig. 4. Magnification of the lower left corner of Fig. 3.

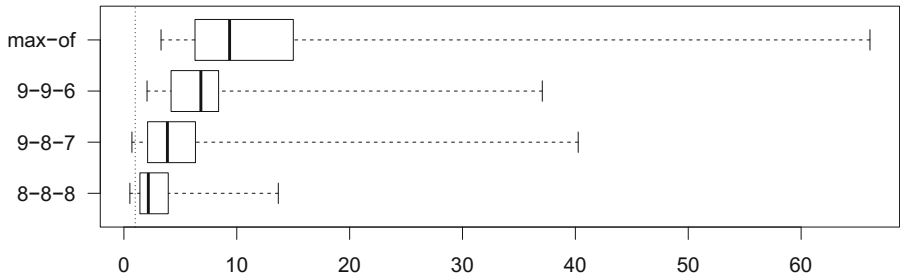


Fig. 5. Reduction factors compared to 6-6-6-6 PDB.

increased by one. Thus, the large PDBs return fewer small values but they do not provide a significantly higher maximum.

5.2 Empirical Evaluation

For the second set of experiments, we used Korf’s fifty random instances and solved them optimally. We present data on the breadth-first iterative deepening A* algorithm (BF-IDA*) [14], a breadth-first variant of IDA* [10]. We chose BF-IDA* over IDA* because its performance does not depend on the node ordering and it therefore allows to better assess the performance of the heuristic. We sorted the 50 instances by the number of expanded nodes with BF-IDA* using the 6-6-6-6 PDB.

Figure 6 shows the reduction of node expansions in comparison to the 6-6-6-6 PDB. For each bar we divided the nodes expanded by the 6-6-6-6 PDB by that of the other PDBs. In general, larger PDBs tend to perform better than smaller ones and the gain seems to be independent from the problem difficulty. However, there are a number of outliers in both directions.

Figure 5 summarizes Fig. 6 and groups the reduction factors by PDB. For the *max-of* line on the top, the maximum of the 6-6-6-6, 8-8-8, 9-8-7 and 9-9-6 PDBs for the heuristic. The memory consumption is only marginally larger because of the overlapping partitions. The four PDBs reduce the number of expanded nodes by a median factor of 2.16, 3.86, 6.81 and 9.36. However, there are some outliers towards both ends of the scale. For some instances the number of expanded nodes was higher compared to the 6-6-6-6 PDB. On the other hand, it could be reduced by a factor of up to 10 with the 8-8-8 PDB and up to 40 with the 9-8-7

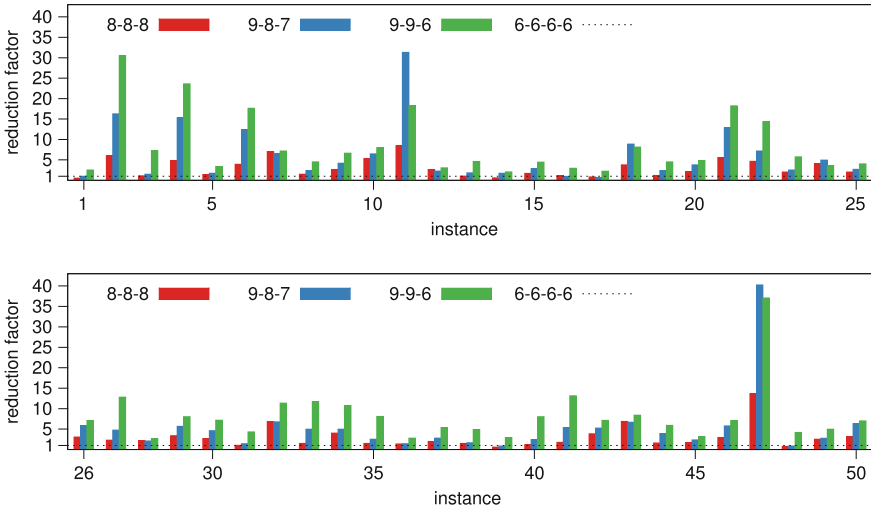


Fig. 6. Reduction factor to 6-6-6-6 PDB on Korf’s random set (ordered by IDA* nodes) using BF-IDA*.

and 9-9-6 PDBs. The standard deviation seems to slightly increase with the size of the heuristic.

Table 2 in the Appendix shows the detailed results for each problem instance. The first column gives the *Id* used in [12] and the second column states the length d of the shortest path. The number of expanded nodes with the individual PDBs are listed in columns three to seven. Columns eight to eleven give the reduction factor of the 8-8-8, 9-8-7, 9-9-6, and *max-of* PDBs relative to the 6-6-6-6 PDB.

6 Conclusion

We presented an efficient parallel algorithm and a compact data structure that allowed us to compute for the first time very large compressed PDBs. The parallel algorithm utilizes the aggregated memory of multiple parallel computers to compute and store the PDB in the main memory.

We computed three additive PDBs for the 24-puzzle, an 8-8-8, 9-8-7 and 9-9-6 PDB. To the best of our knowledge, these are the largest PDBs reported for this domain.

The 9-9-6 PDB gives on average an 8-fold node reduction compared to a 6-6-6-6 PDB on Korf’s random instances of the 24-puzzle. We observed a high variance on the reduction rate, which ranges from 2x to 37x savings (Table 2). Hence, we suggest to use the maximum over several additive PDBs in practice. This is feasible, because multiple additive PDBs do not proportionally increase the memory consumption. This is because the same PDB can be utilized by multiple additive PDBs. As an example, the same 9 PDB can be used in both of our 9-9-6 PDB and the 9-8-7 PDB.

Acknowledgments. This work was partly supported by the EU project CONTRAIL, the DFG project FFMK and the North German Supercomputer Alliance HLRN.

Appendix

Table 2. Expanded nodes of all 50 random instances (r_1 : 6-6-6-6 / 8-8-8, r_2 : 6-6-6-6 / 9-8-7, r_3 : 6-6-6-6 / 9-9-6, r_4 : 6-6-6-6 / max-of).

Id	d	6-6-6-6	8-8-8	9-8-7	9-9-6	max-of	r_1	r_2	r_3	r_4
40	82	26,320,497	49,291,000	26,655,910	10,486,000	7,166,383	0.53	0.99	2.51	3.67
38	96	58,097,633	9,577,883	3,573,949	1,906,127	1,638,334	6.07	16.26	30.48	35.46
25	81	127,949,696	118,780,897	85,141,009	17,658,986	15,217,162	1.08	1.50	7.25	8.41
44	93	181,555,996	37,853,812	11,869,090	7,686,937	5,547,600	4.80	15.30	23.62	32.73
32	97	399,045,498	281,515,091	232,222,028	117,317,314	67,570,393	1.42	1.72	3.40	5.91
28	98	450,493,295	114,571,662	36,263,727	25,552,985	19,743,793	3.93	12.42	17.63	22.82
22	95	581,539,254	82,503,279	88,652,504	81,038,427	37,858,513	7.05	6.56	7.18	15.36

(continued)

Table 2. (continued)

Id d	6-6-6-6	8-8-8	9-8-7	9-9-6	max-of	r ₁	r ₂	r ₃	r ₄
36 90	603,580,192	408,261,989	252,309,866	133,482,919	95,563,302	1.48	2.39	4.52	6.32
30 92	661,835,606	256,431,250	158,409,200	99,557,684	52,338,447	2.58	4.18	6.65	12.65
1 95	1,059,622,872	199,198,406	163,950,295	133,060,463	63,948,759	5.32	6.46	7.96	16.57
29 88	1,090,385,785	128,886,129	34,814,333	59,609,938	21,223,415	8.46	31.32	18.29	51.38
37 100	1,646,715,005	628,890,120	725,323,664	542,573,720	331,223,844	2.62	2.27	3.04	4.97
16 96	1,783,144,872	1,729,554,795	966,783,772	387,360,939	296,519,726	1.03	1.84	4.60	6.01
5 100	1,859,102,197	3,125,977,623	1,078,990,063	905,861,248	565,263,022	0.59	1.72	2.05	3.27
13 101	1,979,587,555	1,181,771,575	690,327,991	444,476,728	268,475,464	1.68	2.87	4.45	7.37
47 92	4,385,270,986	3,825,636,827	4,520,442,316	1,479,759,728	960,463,883	1.15	0.97	2.96	4.57
3 97	4,805,007,493	5,699,072,723	6,731,407,433	2,146,564,697	1,113,194,453	0.84	0.71	2.24	4.32
4 98	5,154,861,019	1,361,290,863	581,368,420	632,299,449	370,467,747	3.79	8.87	8.15	13.91
26 105	6,039,700,647	4,993,857,550	2,525,926,189	1,337,993,889	955,364,988	1.21	2.39	4.51	6.32
31 99	7,785,405,374	3,653,831,114	2,058,364,161	1,622,465,469	992,726,542	2.13	3.78	4.80	7.84
27 99	7,884,559,441	1,415,859,414	611,960,188	432,345,846	337,466,232	5.57	12.88	18.24	23.23
41 106	8,064,453,928	1,737,010,534	1,123,917,776	561,944,277	455,028,148	4.64	7.18	14.35	17.72
43 104	8,816,151,498	4,378,714,353	3,498,876,258	1,532,474,999	1,090,696,435	2.01	2.52	5.75	8.08
6 101	9,810,208,759	2,397,434,227	1,982,606,973	2,739,184,006	1,053,141,115	4.09	4.95	3.58	9.32
49 100	11,220,738,849	5,526,627,744	4,160,235,910	2,792,736,271	1,587,674,537	2.03	2.70	4.02	7.07
45 101	17,068,061,084	5,614,562,048	2,909,124,921	2,408,543,192	1,339,279,458	3.04	5.87	7.09	12.74
20 92	20,689,215,063	9,014,702,404	4,354,383,611	1,615,310,063	1,378,812,797	2.30	4.75	12.81	15.01
46 100	21,674,806,323	9,872,851,915	10,304,210,129	8,017,940,089	3,402,288,275	2.20	2.10	2.70	6.37
19 106	22,761,173,348	6,759,987,121	4,019,764,127	2,836,304,399	2,125,081,076	3.37	5.66	8.02	10.71
35 98	23,049,423,391	8,584,994,059	4,998,934,055	3,208,321,325	2,369,834,229	2.68	4.61	7.18	9.73
7 104	27,686,193,468	26,781,188,637	19,232,502,973	6,429,879,587	4,395,653,789	1.03	1.44	4.31	6.30
8 108	29,575,219,906	4,318,849,565	4,366,429,730	2,609,051,057	1,727,994,805	6.85	6.77	11.34	17.12
39 104	34,198,605,172	22,810,919,845	6,881,101,921	2,912,577,301	2,428,595,642	1.50	4.97	11.74	14.08
42 108	37,492,323,962	9,339,335,844	7,508,532,598	3,490,897,448	2,697,310,294	4.01	4.99	10.74	13.09
24 107	38,272,741,957	25,802,863,114	15,170,752,402	4,724,091,699	3,837,236,834	1.48	2.52	8.10	9.97
2 96	40,161,477,151	29,318,072,174	28,011,360,591	14,446,211,551	8,963,348,921	1.37	1.43	2.78	4.48
15 103	52,178,879,610	26,951,022,561	18,771,225,751	9,741,418,794	8,075,823,446	1.94	2.78	5.36	6.46
23 104	54,281,904,788	36,611,741,317	32,729,241,923	11,103,574,065	8,930,804,356	1.48	1.66	4.89	6.08
48 107	58,365,224,981	99,614,525,233	68,013,167,519	19,890,964,633	12,563,246,704	0.59	0.86	2.93	4.65
34 102	59,225,710,222	49,923,377,951	24,336,781,035	7,384,409,074	5,346,161,078	1.19	2.43	8.02	11.08
12 109	76,476,143,041	43,132,155,298	14,260,876,794	5,820,163,959	4,265,458,902	1.77	5.36	13.14	17.93
21 103	98,083,647,769	25,411,173,479	18,746,227,139	13,731,206,789	8,402,416,300	3.86	5.23	7.14	11.67
18 110	126,470,260,027	18,375,847,744	18,999,810,842	15,070,620,942	7,809,249,544	6.88	6.66	8.39	16.19
9 113	132,599,245,368	82,839,919,151	33,749,539,711	22,489,080,304	16,927,179,096	1.60	3.93	5.90	7.83
33 106	134,103,676,989	77,163,409,262	57,402,766,270	42,219,474,099	25,271,466,707	1.74	2.34	3.18	5.31
17 109	143,972,316,747	49,516,974,145	25,000,824,805	20,405,484,237	15,304,298,302	2.91	5.76	7.06	9.41
11 106	309,253,017,124	22,602,670,676	7,683,989,291	8,343,197,181	4,678,739,173	13.68	40.25	37.07	66.10
14 111	312,885,453,572	419,699,251,120	360,169,788,945	74,779,904,961	63,056,188,490	0.75	0.87	4.18	4.96
10 114	525,907,193,133	207,752,246,775	192,243,603,386	105,311,763,457	63,629,118,230	2.53	2.74	4.99	8.27
50 113	1,067,321,687,213	334,283,260,227	168,384,195,109	152,720,707,871	100,026,128,248	3.19	6.34	6.99	10.67
Average	71,004,578,707.12	33,908,766,050.50	23,611,990,572.06	11,599,129,942.46	7,794,424,738.66	3.00	5.74	8.37	12.85
Median	14,144,399,966.50	5,570,594,896.00	4,257,309,760.50	2,508,797,124.50	1,359,046,127.50	2.16	3.86	6.81	9.36

References

1. Breyer, T.M., Korf, R.E.: 1.6-bit pattern databases. In: AAAI (2010)
2. Cooperman, G., Finkelstein, L.: New methods for using Cayley graphs in interconnection networks. *Discrete Appl. Math.* **37**, 95–118 (1992)
3. Culberson, J.C., Schaeffer, J.: Pattern databases. *Comput. Intell.* **14**(3), 318–334 (1998)
4. Edelkamp, S., Jabbar, S., Kissmann, P.: Scaling search with pattern databases. In: Peled, D.A., Wooldridge, M.J. (eds.) *MoChArt 2008. LNCS*, vol. 5348, pp. 49–64. Springer, Heidelberg (2009)

5. Felner, A.: Improving search techniques and using them on different environments. Ph.D. thesis (2001)
6. Felner, A., Adler, A.: Solving the 24 Puzzle with instance dependent pattern databases. In: Zucker, J.-D., Saitta, L. (eds.) SARA 2005. LNCS (LNAI), vol. 3607, pp. 248–260. Springer, Heidelberg (2005)
7. Felner, A., Korf, R.E., Hanan, S.: Additive pattern database heuristics. *J. Artif. Intell. Res.* **22**, 279–318 (2004)
8. Felner, A., Meshulam, R., Holte, R.C., Korf, R.E.: Compressing pattern databases. In: AAAI, pp. 638–643 (2004)
9. Holte, R.C., Newton, J., Felner, A., Meshulam, R., Furcy, D.: Multiple pattern databases. In: Proceedings of the Fourteenth International Conference on Automated Planning and Scheduling (ICAPS-04), pp. 122–131 (2004)
10. Korf, R.E.: Depth-first iterative-deepening an optimal admissible tree search. *Artif. Intell.* **27**(1), 97–109 (1985)
11. Korf, R.E., Felner, A.: Disjoint pattern database heuristics. *Artif. Intell.* **134**(1–2), 9–22 (2002)
12. Korf, R.E., Schultze, P.: Large-scale parallel breadth-first search. In: Proceedings of the National Conference on Artificial Intelligence, vol. 20, pp. 1380–1385. AAAI Press/MIT Press (2005)
13. Zhou, R., Hansen, E.A.: Space-efficient memory-based heuristics. In: Proceedings of the National Conference on Artificial Intelligence, pp. 677–682. AAAI Press/MIT Press (2004)
14. Zhou, R., Hansen, E.A.: Breadth-first heuristic search. *Artif. Intell.* **170**(4–5), 385–408 (2006)

Computer Games

Workshop on Computer Games, CGW 2013, Held in
Conjunction with the 23rd International Conference on
Artificial Intelligence, IJCAI 2013, Beijing, China, August
3, 2013, Revised Selected Papers

Cazenave, T.; Winands, M.H.M.; Iida, H. (Eds.)

2014, XI, 133 p. 24 illus., Softcover

ISBN: 978-3-319-05427-8