

External Memory PDBs: Initial Results

Nathan R. Sturtevant

Department of Computer Science
University of Denver
Denver, CO, USA
sturtevant@cs.du.edu

Abstract

Pattern databases (PDBs) have been widely used as heuristics for many types of search spaces, but they have always been computed so as to fit in the main memory of the machine using the PDB. This paper studies the how external-memory PDBs can be used. It presents results of both using hard disk drives and solid-state drives directly to access the data, and of just loading a portion of the PDB into RAM. For the time being, all of these approaches are inferior to building the largest PDB that fits into RAM.

Introduction and Overview

In optimal problem solving, one of the key advancements in work to solve large problems was the development of techniques for automatically building heuristics, or estimates of the distance to the goal from a given state in the state space. The better the heuristic, the easier it is to find a path from the start to the goal.

Many different types of heuristics have been built for different types of problems. One early approach that has been well-studied is that of pattern databases (PDBs) (Culberson and Schaeffer 1996). Pattern databases abstract away some portion of the original state space and solve the remaining state space in a way that provides a heuristic estimate for the original unabstracted state space. Work on pattern databases lead to the first solutions of random Rubik's cube instances, and also led to deeper insights on how heuristics influences the cost of search (Korf 1997; Korf, Reid, and Edelkamp 2001).

One idea that has been dominant in this work is that pattern databases must be built to fit in memory, as the cost of random access from disk will not necessarily offset the savings of using a stronger heuristic in practice. While these arguments are well-grounded, they have not, to our knowledge, been tested in practice.

Furthermore, new forms of external storage are now available which have different properties from traditional drives. In particular, because hard disk drives (HDDs) contain a spinning platter that physically turns, there is lag in random access to a drive while the drive waits for data to move under the magnetic head which is reading the data. Solid state

drives (SSDs) have no moving parts, storing data in non-volatile memory. This improves the random access performance of SSDs, although they are still significantly slower than access to main memory.

This paper provides experiments which directly measure the performance of external memory pattern databases on both HDDs and SSDs, confirming the expected results that these devices are too slow for practical use. We then experiment with a technique that has been used in heuristics for two-dimensional maps to see if it can be used for pattern databases.

Background and Problem Formulation

We define a search problem by a graph $G = (V, E)$, a start state s , a goal state g , and a heuristic function $h(a, b)$ which estimates the cost of the shortest path between a and b . While general edge costs are allowed for many problems, we assume that all edges have uniform cost of 1. All the problems considered in this paper have a single goal state, so we can write our heuristic function as $h(a)$, with the goal implicitly defined. Furthermore, we assume that our heuristic is admissible in that it never over-estimates the actual cost of the path to the goal. If $h^*(a)$ is the true cost between a and the goal, an admissible heuristic always has $h(a) \leq h^*(a)$.

One property that is often assumed is heuristic consistency. In an undirected domain with unit edge costs, a heuristic is consistent if for all neighbors a and b , $|h(a) - h(b)| \leq 1$. That is, the heuristic never changes by more than 1 between two adjacent states. If this property does not hold, then the heuristic is inconsistent. A consistent heuristic is also admissible. Inconsistent heuristics can arise in many situations, but are most commonly result when heuristics are compressed or when multiple heuristics are used from different sources.

An inconsistent heuristic can significant impact the performance of A* (Mero 1984), but the effects of inconsistency in states spaces with few cycles (which we generally use IDA* to search), are generally positive (Felner et al. 2011).

IDA*

IDA* (Korf 1985) is a best-first algorithm which finds optimal solutions to a search problem using space linear in the solution depth. It does this through cost-limited depth

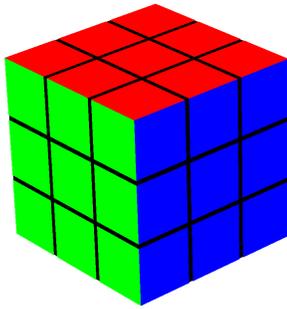


Figure 1: The Rubik’s cube problem.

first searches. The cost limit for an iteration is defined by $f = g + h$, where the g -cost of a state is the cost of the path from the start to that state and the h -cost of a state is the heuristic estimate of the distance from that state to the goal. IDA* does not detect cycles, so it is best used in domains that have relatively few cycles. IDA* has been a standard algorithm for solving the Rubik’s cube, so we use the algorithm in this work as well.

Pattern Databases

Pattern databases are a form of heuristic commonly built for domains that can be represented by a permutation of elements. The pancake puzzle, for instance, is represented by a pile of pancakes of different sizes, with the goal being to sort the pancakes from smallest to largest. The goal state for a 4 pancake problem is $(0, 1, 2, 3)$, with legal states represented by any possible permutation of the elements, such as $(3, 1, 2, 0)$. The size of the domain is the number of possible permutations of the values in the domain. Thus, for the pancake puzzle example there are $4! = 24$ possible states in the state space. If we could store the distance from every permutation in the state space to the goal, we would have a perfect heuristic, but the size of the state space is prohibitively large for most interesting problems.

We assume, but will not discuss here, that we have a function which can map back and forth between a state representation (e.g. an array of numbers) and a compact integer representation of a state. The process of converting an explicit state into a unique id is called *ranking*, and the process of generating a state back from the unique id is *unranking*.

The idea of pattern databases is to abstract the permutations in a way that results in a state space small enough to fit into RAM. This state space can then be searched exhaustively, and the distances in the abstract state space can be used as estimates for distances in the actual state space. To continue the pancake example, we could abstract away the two largest pancakes, giving a goal state that looks like $(0, 1, -, -)$. This state space has only $4!/2! = 12$ possible permutations, and so is smaller than the original state space. Given the state $(3, 1, 2, 0)$, we get a heuristic value by abstracting away the largest pancakes and then looking up the resulting state, $(-, 1, -, 0)$, to find its distance from the goal. It is strictly easier to solve this sub-problem, so pattern databases result in admissible heuristics.

Pattern databases have traditionally been built to fit into memory, because the heuristic for a state must be looked up at every node in the search tree. Permutation puzzles tend to have simple rules for generating the neighbors of a state in a search problem, meaning that a single lookup to an expensive heuristic might be equivalent to searching many nodes in the state space. Thus, it is important that the gains in node reductions from the heuristic outweigh the cost of the heuristic. Recent work on the sliding-tile puzzle (Döbbelin, Schütt, and Reinefeld 2013), for instance, has significant node reductions, but time reductions are not reported, even though the entire PDB is loaded into RAM.

A conjecture of Korf (1997) is that the time required for search, as measured by nodes generated/expanded, is proportional to n/m where n is the size of the problem space and m is the size of the memory used for the heuristic¹. Later work (Holte and Hernádvolgyi 1999) confirmed a slightly revised version of this hypothesis. Further research led to robust predictions of work given a state space and the heuristic distribution on that state space (Korf, Reid, and Edelkamp 2001; Zahavi et al. 2010).

Rubik’s Cube

In this work we focus on the domain of Rubik’s cube, shown in Figure 1. Rubik’s cube has 8 corner cubes and 12 edge cubes. The corner cubes can be in one of three orientations (rotations), while the edge cubes can be in 2 different orientations. Given the orientation of the first $n - 1$ cubes, the n th cube has a fixed orientation. So, there are $8! 3^7 = 88,179,840$ possible corner configurations and $12! 2^{11} = 980,995,276,800$ edge configurations. Multiplying these together, but dividing by two for reasons of parity between the configurations gives 4.33×10^{19} states in the state space.

The original work on building PDB’s for Rubik’s Cube (Korf 1997) built three PDBs for obtaining heuristic values. The first PDB was built just on the corner cubes, shown in Figure 2(a). This sub-space, as described above, has 88 million configurations. Because only 4 bits are needed for storing the depth in the PDB, this requires 42MB of memory to store. The remaining PDBs were based on subsets of the state space containing only 6 edge cubes, which requires 20MB to store. This combination of PDBs fit in to memory in 1997 when the work was done, and trivially fits into memory today.

Other work in Rubik’s cube has been oriented towards finding diameter of the state space – the minimum number of moves to solve any cube (Kunkle and Cooperman 2007; Rokicki et al. 2010), or the diameter of a particular sub-space of the cube (Robinson, Kunkle, and Cooperman 2007; Korf 2008).

PDBs with larger sets of edge cubes can more easily fit into memory in a modern machine. We list the size of the PDBs as the number of edge cubes in the PDB grows in Table 1. While we are acquainted with colleagues that have machines that could fit any of these PDBs in RAM, our

¹This conjecture only applies to exponentially growing domains

Table 2: Search statistics using disk directly.

PDB	Memory (corner)	Avg. Value (corner)	Memory (edge)	Avg. Value (edge)	Avg. Nodes	Avg. Time
Corner+7edge	42MB	8.76	238MB	8.51	19,891,069	11.80
Corner+9edge	42MB	8.76	19 GB	9.97	1,926,871	1.49
Corner+10edge (SSD)	42MB	8.76	114GB	10.63	596,684	47.80
Corner+12edge (SSD)	42MB	8.76	457GB	11.27	249,705	43.79
Corner+12edge (HDD)	42MB	8.76	457GB	11.27	82,166	465.40

largest machine has 64GB of RAM, which can store up to the 9 edge cube pattern database.

However, we have built the 12 edge PDB, shown in Figure 2(b) using external storage (Sturtevant and Rutherford 2013) to store the results. We have also built the 10 edge PDB. This work raises the question: Given that we have a PDB which is larger than available RAM, how can we use that PDB for search. A common answer to this has been compression (Felner et al. 2007). While we will look at a form of compression here, we are interested in studying this problem from a broader perspective – that is, to ask what alternate approaches exist, and whether these are or are not viable. We first look at the problem of using the PDBs directly from disk, and then attempt an alternate approach.

Direct Use of External Memory

In this section we experiment with the direct use of disk for heuristics. It is known that HDDs are not well-suited to random access, but we have not seen experimental results which validate this and measure exactly how slow HDDs are in practice. Additionally, SSDs have far better random access performance than HDDs, so we are interested in measuring results with both devices to compare performance.

We take a set of 100 problems generated and given to us by Ariel Felner which have, for the most part, an optimal solution of 14 moves; the average solution length is 13.91 moves. We solve these problems using IDA* with the corner PDB plus one additional edge PDB. We used edge PDBs with 7 edges and 9 edges, both of which fit into RAM, and also with 10 and 12 edges, stored on disk, although our primary focus is on the 12 edge PDB. We put the data both on a

HDD and a SSD to measure the difference in performance. Our implementation uses standard techniques to minimize cycles in the IDA* search.

The results of the search are in Table 2, including statistics about the average heuristic value in the pattern databases. Notice that in going from the 7 edge to the 9 edge PDB we increase the size of the PDB by a factor of 80, and increase the average heuristic value in the PDB by 1.5. Going from the 9 edge PDB to the 10 edge PDB we increase the average heuristic value by 0.66 with a six times increase in the size of the PDB. Going from the 9 edge PDB to the 12 edge PDB we increase the average heuristic value by 1.2 with a 24 times increase in the size of the PDB. The larger PDBs result in a significant reduction in average node expansions. Interestingly, the 10 edge PDB on the SSD is not much slower than the 12 edge PDB, even though it performs more than twice as many node expansions.

But, as expected, the direct use of disk for accessing the heuristics is much slower than using a smaller PDB that fits in memory. The 10 edge and 12 edge PDBs on the SSD are approximately 30 times slower than the 9 edge PDB. The HDD on the 12 edge PDB was so slow that we only ran it on the first 11 problems of the problem set, and the 10 edge results were significantly slower. Over this subset of relatively easy problems (as compared by the number of nodes expanded), the HDD was over 10 times slower than the SSD, and, if were to have run it to completion, we expect it to be about 30 times slower than the SSD. This matches analysis which states that SSD performance is at about the mean between main memory and RAM (Edelkamp and Schrödl 2012), although we would need to load the 12 edge data into RAM to confirm this.

On the problem set, the SSD for the 12 edge PDB was, however, faster than the 9 edge PDB in RAM on three of the

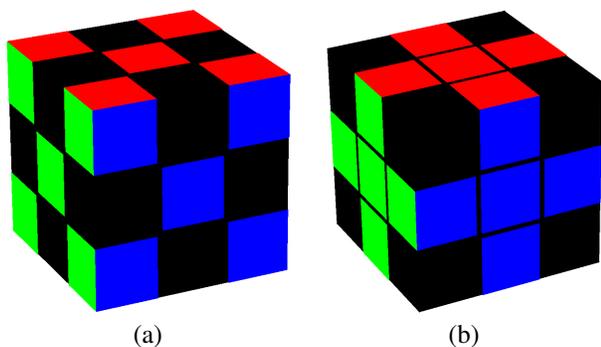


Figure 2: Rubik's Cube edge and corner abstractions.

Table 1: Size of the PDBs with a growing number of edge cubes.

Edge Cubes	Entries	Storage
6	42,577,920	20.3 MB
7	510,935,040	243.6 MB
8	5,109,350,400	2.4 GB
9	40,874,803,200	19.0 GB
10	245,248,819,200	114.2 GB
12	980,995,276,800	456.8 GB

problems – all of which required less than 100,000 node expansions. In the worst case it was 75 times slower – this was on the hardest problem out of the set, requiring 3,058,490 node expansions with the 12 edge PDB and 8,361,800 with the 9 edge PDB.

This suggests that performance degrades the harder the problem becomes. The larger the search, the larger the portion of the states space that will need to have heuristic values looked up. This hurts the OS-level disk caching, and therefore degrades performance.

There are two approaches which then might be viable. First, if we could do a better job caching disk access, we might be able to amortize the cost of looking up heuristics on disk and improve performance. The SSD-based PDB is approximately 30 times slower than the RAM-based PDB. This suggests that, if we were able to perform 30 heuristic lookups from a single read from disk, the performance would be on par. The second approach is to only load a portion of the PDB into RAM, and to use the RAM-based PDB when possible. We will evaluate this second approach in Rubik’s cube shortly, but first we describe previous work which only loads a portion of a heuristic into RAM.

Compressed/Interleaved Differential Heuristics

Different types of heuristics are required for different types of state spaces. While PDBs have been successful in exponentially growing domains, they will not necessarily work well in domains that grow polynomially (Felner, Sturtevant, and Schaeffer 2009). In these domains, heuristics fall into a class of true-distance heuristics (Sturtevant et al. 2009), where the heuristic is estimated from actual distances in the state space instead of abstract differences.

One form of a true-distance heuristic is the differential heuristic (Sturtevant et al. 2009). This heuristic works by storing the distance between all states to a single pivot state, p . A heuristic between two arbitrary states can be estimated using the triangle inequality (Goldberg and Harrelson 2005) giving $h(a, b) = |d(a, p) - d(b, p)|$. Multiple such heuristics are combined by taking the maximum. Note that in this state space there isn’t a single goal state, so multiple heuristics are needed to cover all possible goals.

Because this approach is memory intensive, a type of compression has been suggested, in which a large fraction of the data is simply discarded. For instance, if we have n heuristics, we only have heuristic i available at a state with an id/rank modulo n equal to i . Then, heuristic lookups are only possible in states where data is available to make the heuristic computation. This approach, has been called both interleaving and compression (Felner et al. 2011; Goldenberg et al. 2011). It works for two reasons. When compressing multiple different heuristics, the diversity of heuristic values is likely to lead to better estimates than when using a single heuristic, even though the heuristic estimates are not available at every state. The approach also works because local propagation (BPMX) (Felner et al. 2005) is used to propagate heuristic values between neighboring states and avoid known problems with inconsistent heuristics and A*

search (Mero 1984). We propose to use the same approach for Rubik’s cube, and argue why we initially expect the approach to work well in the following section.

Fractional PDBs

We adapt the idea of compressed/interleaved heuristics to pattern databases and rename the idea *Fractional PDBs*, as the approach is not really a form of compression, and we aren’t interleaving values in the same manner as previous work. In a fractional PDB we just load some fraction of the PDB into RAM, and only use the PDB value when the ranking falls into the values available in RAM.

We analyze when a fractional PDB might be useful in practice from the perspective of successor locality. Given a ranking of states in the state space, the state space has high locality if the ranks of the successors of a state are found nearby their parent state. A state space has low locality if the successors are far from the parent state. Researchers have often looked for state spaces or ranking functions that exhibit high locality. Two-Bit Breadth First Search (TBBFS) (Korf 2008), an external-memory search algorithm, takes advantage of locality to solve problems more quickly because it loads adjacent portions of the state space into RAM and can process states in the same portion of the state space quickly. It performs poorly in state spaces with low locality. Structured duplicate detection algorithm (Zhou and Hansen 2004) takes advantage locality to improve parallelism. Writing-Minimizing Breadth-First Search (WMBFS) (Sturtevant and Rutherford 2013) is not as reliant on locality for performance, and has been shown to have good performance in state spaces with low locality.

High Locality A state space like the sliding-tile puzzle is considered to have high locality. The locality of a state space relies on the upper bits of the ranking function, which might, for instance, be determined by the first two tiles in the puzzle. Because the blank tile moves relatively slowly across the state space, most actions will not significantly change the ranking function.

Suppose that we have loaded only a small percentage of a PDB into RAM. In a state space with high locality, the majority of the successors of a state will be in the same portion of the PDB as the parent. While this might help with the amortization of loading values from disk, it also suggests that there will be many, many states for which a heuristic lookup is not available. When a search touches states that fall into the PDB it is likely that the search will be quickly cut off, and so the access to the PDB will be relatively sparse.

Low Locality Rubik’s Cube is a state space with low locality because any move of one of the faces of the cube will change many cubes, and thus have a higher chance of significantly changing the ranking function. As such, TBBFS performs significantly worse than WMBFS in this domain.

In a state space with low locality, the chances of having a state or one of its successors fall into the portion of a PDB that is in RAM will be significantly increased. We will measure this directly in the next set of experimental results.

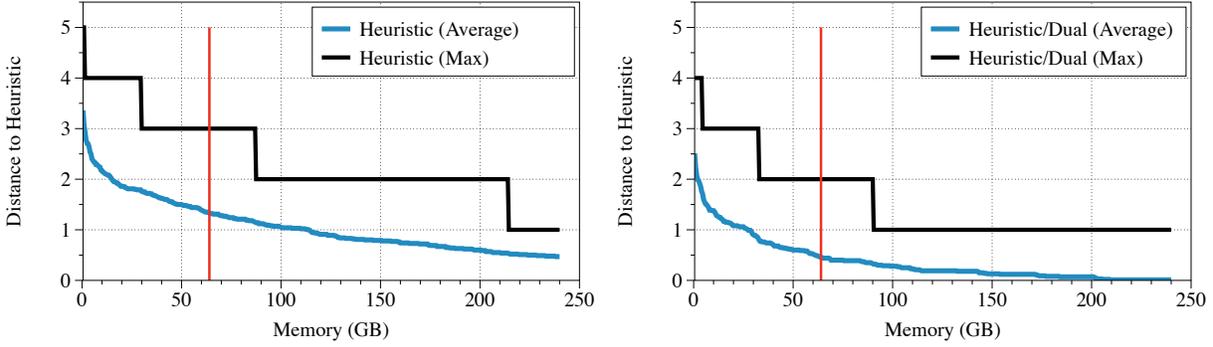


Figure 3: Measurements of the average and maximum distance to a heuristic lookup in the 12 edge PDB given that k GB have been loaded into RAM.

Fractional PDB Experiments

In our first experiment, we measure the effect of locality in Rubik’s cube. We assume that we have loaded k GB of a 12 edge PDB into RAM. Then, we sample the state space. For each sampled state, we measure the depth at which a child state is found for which a heuristic lookup is possible in the k GB of the fractional PDB. The distance to a heuristic value is essentially the error that is introduced into the heuristic as a result of only storing the fractional PDB.

We graph this data in Figure 3; the vertical line marks the memory we have available in our machine. The left graph shows that if we filled memory with the fractional PDB, we would expect to see an error of just under 1.5 in the heuristic value, with a maximum error of 3. This error would result in a heuristic that is worse than the 9 edge PDB (Table 2), and it suggests that we would be better off loading the smaller PDB.

However, state space properties such as symmetry can allow multiple lookups into the same PDB. These lookups will fall into different parts of the PDB, and so this could increase the usefulness of a fractional PDB. We perform the same experiment as before, this time allowing a dual lookup in the PDB (Zahavi et al. 2008) if the regular lookup does not fall into the fractional PDB. Suppose for a state t that the permutation π transforms the start state s into t . Then, the dual of t is the state that applying π to results in s . This data is in the right hand side of Figure 3. Here we see that with 64GB of RAM the expected heuristic error is only 0.5, which does have the potential to provide better performance than the 9 edge PDB.

We validate these results in Table 3. This table compares a large number of different PDB combinations and sizes. The left column indicates which PDBs are used. C stands for the corner PDB. E7 stands for the 7 edge PDB. E9 stands for the 9 edge PDB. E10 and E12 stand for a fractional 10 edge and 12 edge PDB respectively. The amount of data loaded into RAM can be inferred from the total storage used in the second column. The third column is the average number of nodes expanded solving the same set of 100 problems as before, and the time is the average time to solve one of these problems. IDA* was used to solve the problems, and BPMX

was used for propagation of heuristic values, as the resulting heuristics are inconsistent.

As expected, the 12 edge PDB performs poorly, even with 60GB of RAM. But, perhaps it is even worse than expected, as the 280MB 7 edge PDB performs better both in terms of node expansions and time. Adding dual lookups to the heuristic does help, but not enough to perform better than the 7 edge PDB. In fact, the fractional 10 edge PDB performs better than the 12 edge because a larger fraction states are in RAM. The best result in terms of node expansions uses both the 9 edge and fractional 12 edge PDB, along with dual lookup (only in the 12 edge PDB). But, the best time result with just with the single corner PDB and 9 edge PDB. In general, performance is degraded when the PDB is close to the size of main memory. In this case the RAM bus is probably too slow to accommodate all the requests and is thus hurting performance.

We illustrate the degradation of heuristic values using fractional 9 edge PDBs in Table 4. These results use the same 100 problems and the heuristic is the max of the fractional 9 edge PDB and the corner PDB. When the majority of the state space is loaded into RAM, the fractional PDB provides reasonable results. But, loading 60% of the 9 edge PDB is only slightly better than using all of the 7 edge PDB, shown in the bottom row.

These results suggest that the idea of fractional PDBs on its own is not nearly as useful as it was proven to be in pathfinding domains. In particular, the results suggest that the runtime distribution of heuristic values is heavily skewed towards states with poorer heuristic values. For fractional PDBs to be successful, there must be a way to shift this distribution back towards the true heuristic distribution.

One potential issue is that the measurements in Figure 3 do not take into account pruning rules intended to reduce duplicates within the search, which could also be reducing the number of heuristic lookups. As such, we conjecture that the difficulty here is partially a problem of the type of state space being searched. The ideas which worked well in polynomial domains may not apply as well to exponential domains. In particular, polynomial domains usually have many cycles and many possible goal states used in different

Table 3: Experiments with fractional PDBs.

PDB	Total Storage	Nodes	Time
C+E12	18.6 GB	755,312,740	291.00
C+E12	60.5 GB	174,754,420	75.20
C+E12+dual	60.5 GB	58,006,372	39.44
C+E10	60.5 GB	11,233,883	31.79
C+E7+E12+dual	60.8 GB	6,186,872	19.07
C+E7	280.0 MB	19,891,069	11.81
C+E7+E12+dual	18.9 GB	12,652,497	11.62
C+E7+E12	18.9 GB	14,652,320	10.98
C+E7+E12	60.8 GB	9,602,075	10.64
C+E9+E12	51.6 GB	1,535,483	7.91
C+E9+E12+dual	51.6 GB	1,359,370	7.35
C+E7+E12+dual	46.8 GB	7,620,426	7.16
C+E9+E12+dual	23.7 GB	1,764,127	1.93
C+E9+E12	23.7 GB	1,821,488	1.68
C+E9	19.0 GB	1,926,871	1.49

Table 4: Fractional 9 edge PDB

% PDB in RAM	Total Storage	Nodes	Time
100%	19.0 GB	1,926,871	1.49
95%	18.1 GB	2,397,627	1.90
90%	17.1 GB	3,109,128	2.39
85%	16.2 GB	4,078,892	3.05
80%	15.2 GB	5,435,597	3.95
75%	14.3 GB	7,718,233	5.45
70%	13.3 GB	9,332,735	6.52
65%	12.4 GB	11,927,444	8.21
60%	11.4 GB	15,689,744	10.62
E7	0.3 GB	19,891,069	11.81

searches. Thus, good coverage of all goals is needed and is an important component of heuristic accuracy. Additionally, the many cycles means that good heuristic value be easily propagated. In the Rubik’s cube there is a single goal state, and we work hard to avoid duplicates in the IDA* search, reducing the potential gains of fractional PDBs.

Conclusions and Future Work

Given the availability of PDBs which are larger than RAM, this paper addresses the question of how not to effectively use these PDBs. We perform disk-based searches to measure the cost of looking up heuristics directly from disk, validating results expected results with experimental numbers. We also attempt to apply the idea of fractional PDBs, something that has been successful in polynomial domains. We suggest that the approach might work based on the locality of the Rubik’s cube state space, but discover that in practice the results are disappointing.

We have several directions for future work, including optimizing our implementation and performing further experiments to explain why fractional PDBs perform poorly. We are looking into other approaches for using external-memory

PDBs to improve the performance of search, including investigating state spaces with varying locality.

Acknowledgements

The experiments in this paper were run on a machine supported by the NSF I/UCRC on Safety, Security, and Rescue.

References

- Culberson, J. C., and Schaeffer, J. 1996. Searching with pattern databases. *Advances in Artificial Intelligence (Lecture Notes in Artificial Intelligence 1081)* 402–416.
- Döbbelin, R.; Schütt, T.; and Reinefeld, A. 2013. Building large compressed pdbs for the sliding tile puzzle. ZIB-Report 13-21.
- Edelkamp, S., and Schrödl, S. 2012. *Heuristic Search - Theory and Applications*. Academic Press.
- Felner, A.; Zahavi, U.; Schaeffer, J.; and Holte, R. C. 2005. Dual lookups in pattern databases. In *International Joint Conference on Artificial Intelligence (IJCAI-05)*, 103–108.
- Felner, A.; Korf, R. E.; Meshulam, R.; and Holte, R. C. 2007. Compressed pattern databases. *Journal of Artificial Intelligence Research* 30:213–247.
- Felner, A.; Zahavi, U.; Holte, R.; Schaeffer, J.; Sturtevant, N.; and Zhang, Z. 2011. Inconsistent heuristics in theory and practice. *Artificial Intelligence* 175(9-10):1570–1603.
- Felner, A.; Sturtevant, N.; and Schaeffer, J. 2009. Abstraction-based heuristics with true distance computations. In *Symposium on Abstraction, Reformulation and Approximation (SARA-09)*.
- Goldberg, A. V., and Harrelson, C. 2005. Computing the shortest path: A* search meets graph theory. In *SODA*, 156–165.
- Goldenberg, M.; Sturtevant, N. R.; Felner, A.; and Schaeffer, J. 2011. The compressed differential heuristic. In *AAAI Conference on Artificial Intelligence*, 24–29.
- Holte, R. C., and Hernádvölgyi, I. T. 1999. A space-time tradeoff for memory-based heuristics. In *National Conference on Artificial Intelligence (AAAI-99)*, 704–709.
- Korf, R. E.; Reid, M.; and Edelkamp, S. 2001. Time complexity of iterative-deepening-A*. *Artificial Intelligence* 129(1-2):199–218.
- Korf, R. E. 1985. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence* 27(1):97–109.
- Korf, R. E. 1997. Finding optimal solutions to Rubik’s cube using pattern databases. In *National Conference on Artificial Intelligence (AAAI-97)*, 700–705.
- Korf, R. E. 2008. Minimizing disk i/o in two-bit breadth-first search. In *AAAI*, 317–324.
- Kunkle, D., and Cooperman, G. 2007. Twenty-six moves suffice for rubik’s cube. In Wang, D., ed., *ISSAC*, 235–242. ACM.
- Mero, L. 1984. A heuristic search algorithm with modifiable estimate. *Artificial Intelligence* 23:13–27.

- Robinson, E.; Kunkle, D.; and Cooperman, G. 2007. A comparative analysis of parallel disk-based methods for enumerating implicit graphs. In Maza, M. M., and Watt, S. M., eds., *PASCO*, 78–87. ACM.
- Rokicki, T.; Kociemba, H.; Davidson, M.; and Dethridge, J. 2010. God’s number is 20. <http://www.cube20.org/>.
- Sturtevant, N. R., and Rutherford, M. J. 2013. Minimizing writes in parallel external memory search. *International Joint Conference on Artificial Intelligence (IJCAI)*.
- Sturtevant, N.; Felner, A.; Barer, M.; Schaeffer, J.; and Burch, N. 2009. Memory-based heuristics for explicit state spaces. In *International Joint Conference on Artificial Intelligence (IJCAI-09)*, 609–614.
- Zahavi, U.; Felner, A.; Holte, R. C.; and Schaeffer, J. 2008. Duality in permutation state spaces and the dual search algorithm. *Artificial Intelligence* 172(4–5):514–540.
- Zahavi, U.; Felner, A.; Burch, N.; and Holte, R. C. 2010. Predicting the performance of IDA* (with BPMX) with conditional distributions. *Journal of Artificial Intelligence Research* 37:41–83.
- Zhou, R., and Hansen, E. 2004. Structured duplicate detection in external-memory graph search. In *National Conference on Artificial Intelligence (AAAI-04)*, 683–689.