Exploiting the Rubik's Cube 12-edge PDB by Combining Partial Pattern Databases and Bloom Filters

Nathan R. Sturtevant
Dept. of Computer Science
University of Denver
Denver, CO, USA
sturtevant@cs.du.edu

Ariel Felner
Dept. of Information System Engineering
Ben Gurion University
Beer-Sheva, Israel
felner@bgu.ac.il

Malte Helmert
Dept. of Math. and Computer Science
Universität Basel
Basel, Switzerland
malte.helmert@unibas.ch

Abstract

Pattern Databases (PDBs) are a common form of abstraction-based heuristic which are often compressed so that a large PDB can fit in memory. Partial Pattern Databases (PPDBs) achieve this by storing only layers of the PDB which are close to the goal. This paper studies the problem of how to best compress and use the 457 GB 12-edge Rubik's cube PDB, suggesting a number of ways that Bloom filters can be used to effectively compress PPDBs. We then develop a theoretical model of the common min compression approach and our Bloom filters, showing that the original method of compressed PPDBs can never be better than min compression. We conclude with experimental results showing that Bloom filter compression of PPDBs provides superior performance to min compression in Rubik's cube.

Introduction

Heuristic search algorithms such as A* (Hart, Nilsson, and Raphael 1968) and IDA* (Korf 1985) are guided by the cost function $f(n) = g(n) + h(n)$, where $g(n)$ is the cost of the current path from the start node to node n and $h(n)$ is a heuristic function estimating the cost from n to a goal node. If $h(n)$ is *admissible* (i.e., a lower bound on the true cost) these algorithms are guaranteed to find optimal paths.

A large variety of techniques have been used to build heuristics. A prominent class of heuristics are *abstraction-based heuristics* usually designed for exponential (implicit) domains. These heuristics abstract the search space and then solve the abstract problem exactly. The distances in the abstract state space are used as heuristic estimates for the original state space. Pattern databases (PDBs) (Culberson and Schaeffer 1998) are *memory-based heuristics* where exact distances to the goal from all of the abstract states are stored in a lookup table. These distances are then used during the search as h-values.

A key feature of PDBs is that they are designed to fit into memory, ensuring fast random access. When a PDB is too large to fit in memory, a number of techniques have been developed to compress it into memory. The main direction is denoted by *min compression* (Felner et al. 2007) where the PDB is compressed by a factor of k by dividing the PDBs into buckets of size k and only storing the minimum value for each bucket, preserving the admissibility of the heuristic. Another direction is called *partial pattern databases* (PPDBs) (Anderson, Holte, and Schaeffer 2007) where only entries up to a given value V are stored in memory, and $V+1$ can be admissibly used for those abstract states that are not in memory. PPDBs use a hash table where each item stores the *key* (i.e., an explicit description of the abstract state), its associated value, and possible a pointer to the next abstract state in the same entry. Thus, PPDBs are inefficient in their constant memory needs per entry. By contrast, regular and min-compressed PDBs can directly map states into PDB entries, and thus only need to store the PDB values.

This paper focuses on the Rubik's cube 12-edge PDB, one of the largest PDBs built, with 981 billion entries. Our challenge is to find the best way to compress the PDB to fit into RAM. To do this, we introduce new techniques for compressing PDBs by combining the ideas of Bloom filters and PPDBs. Bloom filters (Bloom 1970) are an efficient data structure for performing membership tests on a set of data, where all members of the set are correctly identified, and non-members are identified with some chance of a *false positive*. We present two ways of compressing PDBs into Bloom filters. The first is called a *level-by-level Bloom filter* where states at each depth of the PDB are stored in a separate Bloom filter. The second is called a *min Bloom filter* where states at all depths are stored in the same Bloom filter.

Anderson, Holte, and Schaeffer (2007) originally suggested a method for compressing a PPDB to improve memory efficiency. We show that their idea is a degenerative version of the min Bloom where only 1 hash function is present. Furthermore, we show that compressed PPDBs are functionally equivalent to min compressed PDBs.

We continue by providing a theoretical analysis for predicting the average value of a PDB compressed by min compression and by level-by-level Bloom filters. Based on this we introduce a new measure that quantifies the *loss of information* within a compressed PDB. We conclude with experimental results on Rubik's cube showing that Bloom filters combined with PPDBs are more effective than min compression and are the most efficient way to compress the large 12-edge PDB into current memory sizes.

Background: Pattern Database Heuristics

Admissible heuristics provide an estimate of the distance to the goal without overestimation. A heuristic function $h(\cdot)$ is *admissible* if $h(a) \leq c(a, g)$ for all states a and goal

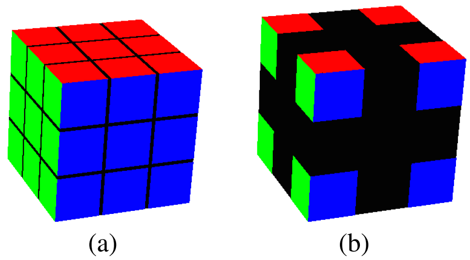

Figure 1: Abstraction applied to Rubik's Cube.

states g, where $c(a,b)$ is the minimal cost of reaching b from a. *Consistent heuristics* also have the property that $h(a) - h(b) \leq c(a,b)$.

A *pattern database* (PDB) is an admissible heuristic that is built through an exhaustive search of an abstracted state space. We demonstrate this with Rubik's cube. Figure 1(a) shows the $3{\times}3{\times}3$ Rubik's cube puzzle, which has $4.3{\times}10^{19}$ states and cannot be fully enumerated and stored by current computers. Now assume that all of the edges (cubies with two faces) are blacked out and rendered identical, as shown in Figure 1(b). This $2 \times 2 \times 2$ cube is an abstraction of the original which only has 8.8×10^7 states. Because the corner pieces must be solved in order to solve the full puzzle, the distance in the abstract state space is a lower bound on the cost of the full solution, and thus is an admissible heuristic for the original $3 \times 3 \times 3$ puzzle.

PDBs are usually built through backwards breadth-first searches from the abstract goal states until all abstract states are reached and the optimal distance to all these abstract states is stored in a lookup table – the *pattern database*.

When the search in the original $3 \times 3 \times 3$ state space reaches a new state n, n is abstracted by blacking out the edges to reach an abstract state, n'. n' is used as the index into the PDB, and its distance to the abstract goal is looked up. That distance is used as the h-value for the original state n. This is called a *pattern database lookup*. In permutation domains, PDBs can be stored compactly in memory (in a table with M entries, one entry per state) because they can be efficiently enumerated and indexed by a ranking function (Myrvold and Ruskey 2001) that uniquely and compactly maps the states to integers. Thus, from a description of the state, we can calculate the index into the PDB entry that contains the corresponding h-value. This is called a *compact mapping*.

The work in this paper is motivated by recent research that has built large PDBs, including for the sliding-tile puzzle (Döbbelin, Schütt, and Reinefeld 2013) and Rubik's cube (Sturtevant and Rutherford 2013). These PDBs are hundreds of gigabytes in size, and so won't fit into the memory of most machines. In fact, even when the PDBs do fit in RAM, they won't necessarily reduce the time required to solve a problem compared to a smaller PDB because the constant time per lookup is expensive when the PDB is extremely large. For instance, no gain in time was reported[1] over smaller PDBs when using PDBs up to 1 TB in size on the sliding-tile puzzle (Döbbelin, Schütt, and Reinefeld 2013). Thus, compression techniques for PDBs are an important area of study.

[1] Confirmed via personal communication with the authors.

Compressing Pattern Databases

Existing compression techniques for PDBs can characterized as *lossless* or *lossy* (Felner et al. 2007).

Lossless Compression

Assume that the original PDB includes M entries, and that each entry needs B bits. In lossless compression the exact information of the M entries of the PDB is stored using less than B bits per entry on average.

Felner et al. (2007) suggested a lossless compression technique that groups k entries together into a single entry containing the minimum, m, of the k entries; a second small table for each entry stores the delta above m for each of the states that map to that entry.

Breyer and Korf (2010) introduced *1.6 bit pattern databases* which only need 1.6 bits per entry. The main idea is to store the PDB value modulo 3. They showed that if the heuristic is consistent, then we only need to know the actual heuristic of the start state. The 1.6 bit PDB will tell us whether each child's heuristic increased or decreased, allowing us to infer the other heuristic values.

A more effective technique, particularly on domains with large branching factor and significant structure is Algebraic Decision Diagram (ADD) compression (Ball and Holte 2008). This approach was able to compress the PDBs for many domains by several orders of magnitude, although it is not universally effective.

Lossy Compression

Lossy compression techniques may lose information by the compression and they must be carefully created to maintain admissibility. Despite the loss of information, lossy compression can often provide far greater compression than lossless compression in practice. Lossy compression techniques can also result in inconsistent (but admissible) heuristics. The main challenge with lossy compression is to minimize the loss of information.

Samadi et al. (2008) used an Artificial Neural Network (ANN) to learn the values of the PDBs. This ANN is in fact a lossy compression of the PDB. To guarantee admissibility a side table stores the values for which the ANN overestimated the actual PDB value.

A simple form of PDB compression (Felner et al. 2007) is what we denote by *min compression*. To compress the PDB with M entries by a factor of k the entire PDB is divided into M/k buckets, each of size k. The compressed PDB is only of size M/k and only stores one value for each bucket. To guarantee admissibility, the minimum value from each bucket in the original PDB is stored. We illustrate how one bucket is compressed in Figure 2. The top row represents one bucket of original PDB values. In this example we perform $8\times$ compression, and all the entries map into a single value which is the minimum value among the original 8 values, 4

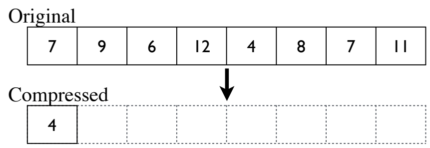

Figure 2: Example showing min compression

in our case. Min compression is a lossy operation because information from the original PDB is lost as all abstract states in this bucket will return 4.

Felner et al. (2007) showed that the key to minimize the loss of information is to analyze the original PDB and to group values that are as similar as possible. One way of doing this is to group together cliques (or set of nodes with small diameter). In some cases nearby entries in the PDBs exhibit this behavior and then a simple DIVk operation on the original PDB provides a simple yet strong way to compress the PDB into buckets while maintaining a small rate of loss of information. However, achieving this best-case behavior is domain-dependent and will not be applicable in all domains. This approach worked very well for the 4-peg Towers of Hanoi, for instance, but its success for the sliding tile puzzles was limited and no significant advantage was reported for the Top-Spin domain (Felner et al. 2007).

Despite this detailed study, no theoretical model of min compression has been proposed. We will propose a novel model which predicts the average performance of min compression. We then suggest a metric for quantifying the loss of information by a given compression method.

Partial Pattern Databases

Partial Pattern Databases (PPDBs) (Anderson, Holte, and Schaeffer 2007) perform a lossy compression by only storing the first D levels (i.e., depths from the goal) of the original PDB instead of all levels. Edelkamp and Kissmann (2008) also use this approach for compression of symbolic PDBs. Such a PPDB is denoted $PPDB_D$. When looking up an abstract state t, if $PPDB_D$ includes t then its h-value $PPDB_D(t)$ (which is $\leq D$) is retrieved and used. If t is not in the PPDB, $D+1$ is used as an admissible heuristic for t. This compression technique is lossy because all values larger than D are now treated as $D+1$.

Full PDBs are usually stored with compact mappings from abstract states into PDB entries based on a ranking of the abstract states. Since PPDBs only store part of the PDB, a compact mapping cannot be used as appropriate ranking functions are not known. Therefore, the simplest PPDB implementation uses a hash table. Each item in the hash table contains the *key* of the state (i.e., its description) and its associated heuristic value. This is very inefficient because the extra memory overhead of storing the keys. In addition, the number of entries in a hash table is usually larger than the number of items stored in it to ensure low collision rate. Finally, the common way to deal with collisions is to have a linked list of all items in a given entry. Thus, we also store a pointer for each of the items. Therefore, the constant memory per item in a PPDB can be much larger than that of a compactly mapped PDB, which only stores the h-value for each abstract state. As a result, Anderson, Holte, and Schaeffer (2007) concluded that 'partial PDBs by themselves are not an efficient use of memory', but that 'compressed partial PDBs are an efficient use of memory'. We will describe their compression approach below and show that their technique can be seen as a degenerative case of our min Bloom filter. Furthermore, we also show that this compression approach can never outperform regular min compression (Felner et al. 2007). On the other hand, we will show that PPDBs can be very effective when implemented by level-by-level Bloom filters.

Depth	# States	Bloom Size	Hash Size
0	1	0.00 MB	0.00 MB
1	18	0.00 MB	0.00 MB
2	243	0.00 MB	0.00 MB
3	3240	0.00 MB	0.05 MB
4	42807	0.05 MB	0.65 MB
5	555866	0.66 MB	8.48 MB
6	7070103	8.43 MB	107.88 MB
7	87801812	104.67 MB	1339.75 MB
8	1050559626	1252.36 MB	16030.27 MB
9	11588911021	13.49 GB	172.69 GB
10	110409721989	128.53 GB	1645.23 GB
11	552734197682	643.47 GB	8236.38 GB
12	304786076626	354.82 GB	4541.67 GB
13	330335518	393.79 MB	5040.52 MB
14	248	0.00 MB	0.00 MB
Total	980,995,276,800	1,142 GB	14,618 GB

Table 1: Number of states at each depth of the 12-edge Rubik's Cube PDB plus approximate sizes for a Bloom filter (10 bits/state) and hash table (128 bits/state)

PPDBs for Rubik's Cube

To validate the general approach of PPDBs for Rubik's cube, we first perform an analysis of their potential performance using the full 12-edge PDB which has 981 billion states and requires 457 GB of storage with 4 bits per abstract state. This PDB is too large for RAM, but we obtained a copy from Sturtevant and Rutherford (2013) and stored it on a solid state drive (SSD). While it is not as fast as RAM it is fast enough to benchmark problems for testing purposes.

Table 1 provides details on the distribution of the h-values of the states in this PDB. The second column counts the number of states at each depth. More than half the states are at level 11, about 12% are at level 10, and less than 2% of the total states can be found at levels 0 through 9. If we want a PPDB to return a heuristic value between 0 and 10, we only need to store states at levels 0 through 9 in the PPDB ($PPDB_9$); states which are not found are known to be at least 10 steps away from the goal. The question is, how can this PPDB be stored efficiently? Estimating that a hash table will require 128 bits per state (for the keys, values, and pointers), this PPDB would need 172 GB of memory, larger than the

Depths	# States (billions)	Nodes	Time
$PPDB_9$	12.7	51,027,608	8,615.6
$PPDB_{10}$	123.1	25,567,143	4,477.7
Full PDB	981.0	24,970,537	4,379.4

Table 2: Nodes generated by a perfect PPDB.

memory of any machine available to us. Bloom filters, described below, are estimated to use around 10 bits per state, suggesting that they may be far more efficient than regular hash tables for implementing PPDBs. These values are shown in the last columns of Table 1.

We evaluated a number of PPDBs built on this PDB using a 100-problem set at depth 14 from the goal as used by Felner et al. (2011). We solve these problems using IDA* (Korf 1985) using the full 12-edge PDB stored on our SSD. We also stored the 8-corner PDB in RAM and took the maximum among the two for the h-value for IDA*. Table 2 provides the total number of generated nodes for solving the entire test suite with various levels of a PPDB. To reiterate, this approach is slow because it uses disk directly; these numbers are simply for validating the general approach. The results show that $PPDB_{10}$ (which returns 11 if the state is not found in the PPDB) only increases the total number of node generations by 2% compared to the full PDB, while using about 13% of the PDB entries. $PPDB_9$ increases the number of nodes by a factor of two (from 24.97 to 51.03 million nodes), but requires less than 1% of the memory of the full PDB. From this, we can conclude that the general notion of a PPDB is effective for Rubik's cube.

We have demonstrated the potential of PPDBs, but using an ordinary hash table will require too much memory to be effective. Thus, a more efficient idea for storing PPDB is needed. This issue is addressed below by our new Bloom filter approach.

Bloom Filters

Bloom filters (Bloom 1970) are a well-known, widely used data structure that can quickly perform membership tests, with some chance of a false positive. By tuning the parameters of a Bloom filter, the false positive rate can be controlled, and false positive rates under 1% are common. Bloom filters are memory efficient because they do not store the keys themselves; they only perform membership tests on the keys which were previously inserted.

A Bloom filter is defined by a bit array and a set of q hash functions $hash_1, \dots, hash_q$. When a key l is added to the Bloom filter we calculate each of the $hash_i$ functions on l and set the corresponding bit to *true*. A key l is looked up in the Bloom filter by checking whether all the q bits associated $hash_i(l)$ of a state are *true*. A false positive occurs if all q hash functions are set for a state that is not in the Bloom filter. This means that q other items share one or more hash values with l. As a result, a Bloom filter can return a false positive – identifying that an item is in the set while it isn't actually there. However, it cannot return false negatives. Any item that it says is not in the set is truly not in the set.

Given m bits in the bit array, n keys in the filter, and q (uniform and independent) hash functions per state, the probability that a given bit is set to 1 is (Mitzenmacher and Upfal 2005):

$$p_{set} = 1 - \left(1 - \frac{1}{m}\right)^{qn} \quad (1)$$

Now assume that an item is **not** in the Bloom filter. The probability that all its q bits are set (by other items), i.e., the probability for a false positive (denoted p_{fp}) is[2]:

$$p_{fp} = (p_{set})^q \quad (2)$$

Thus, if a Bloom filter has 50% of the bits set and it uses 5 hash functions, the false positive rate will be 3.125%.

Next we describe two methods for using Bloom filters to store PPDBs. The first method stores the PPDB level-by-level, while the second method uses a form of Bloom filters to store the entire PPDB.

Level-by-level Bloom Filter for PPDB

The result of a lookup in a PPDB is the depth of an abstract state, but Bloom filters can only perform membership tests. Thus, a natural application of Bloom filters would be to build a Bloom filter for every level of the PPDB. Thus, $PPDB_D$ will have D Bloom filters for levels 1–D. The Bloom filter for level i is denoted by $BF(i)$. Then, given an item t, we perform a look-up for item t in $BF(i)$ for each level i in sequential order, starting at $BF(1)$. If the answer is *true* for level i, then the look-up process stops and i is used as $h(t)$. In case of a false positive, this is smaller than the actual value, but it is still admissible. If the answer is *false* we know for sure that the PDB value of t is not i and we proceed to the next level. If we look up the abstract state in all D Bloom filters and never receive a $true$ answer, $D + 1$ is used as an admissible heuristic.

This method is a form of lossy compression of the PPDB (which itself has loss of information from the original PDB) – when a false positive occurs, we return a value which is lower than the true heuristic value. However, we can tune the memory and number of hashes for each level to minimize the false positive rate as defined in Equation 2.

Hybrid Implementation In practice, this process will be time-consuming because up to D lookups may be required for each state. Instead, we can use a hybrid between ordinary hash functions and Bloom filters as follows. We store the first x levels of the PPDB directly in a hash table, and then use Bloom filters to store later levels of the PPDB. This is especially efficient if the number of abstract states at each depth grows exponentially, as happens in the Rubik's Cube 12-edge PDB (see Table 1). Thus, the memory used in the first x levels will be limited. In this hybrid implementation, the lookup process requires 1 hash lookup to check whether the state is in the first x levels. If the answer is no, then at most $D - x$ Bloom filter lookups will be performed. In

[2] It should be noted that this isn't strictly correct (Bose et al. 2008; Christensen, Roginsky, and Jimeno 2010), but it is sufficient for our purposes.

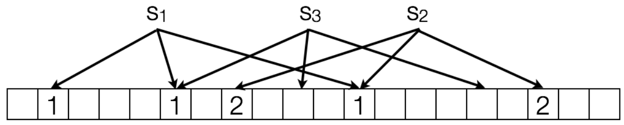

Figure 3: A min Bloom filter

our experiments below, we used this hybrid implementation. Levels 1–7 were stored in a hash table and levels 8 and 9 were stored in Bloom filters. Furthermore, since the underlying PPDB is consistent, we can use the parent heuristic value to reduce the number of Bloom filter queries required at each state. For instance, if the parent heuristic is 10, we only need to query if the child heuristic is in the depth-9 Bloom filter to know the heuristic value.

Single Min Bloom Filter for PPDBs

The previous approach will require multiple lookups to determine the heuristic value for a state. We propose a novel use of Bloom filters, denoted as a *min Bloom filter*, where only a single Bloom filter lookup is performed to determine an admissible heuristic. In a min Bloom filter, we no longer use a bit vector for testing membership and no longer build a different Bloom filter for each level. Instead, we use only one Bloom filter and for each abstract state, we store its h-value directly in that Bloom filter. The Bloom filter is first initialized to a maximum value higher than any value that will be stored. An abstract state t and its h-value $h(t)$ is added to a min Bloom filter by, for each hash function, storing the minimum of $h(t)$ and the value already stored in that entry. Each entry will now have the minimum of the h-values of all abstract states that were mapped to that entry and is thus admissible. When looking up an abstract state t in the min Bloom filter, we lookup in all its associated $hash_i$ entries and the final h-value for t is the maximum over all these entries. This is admissible as we are taking the maximum of a number of admissible heuristics.

We illustrate this in Figure 3, where s_i is at depth i for $i = 1, 2, 3$. An arrow is drawn from each state to the locations associated with each hash function. Looking up s_1, we see that all its associated entries in the Bloom filter are 1, so a value of 1 is used. Looking up s_2, we see values of 1 and 2. We can take the max of these, so the value used is 2. For s_3, some entries are not set, so we assert that s_3 is not in the min Bloom filter and use $D + 1$ as $h(s_3)$.

As a trade-off to requiring fewer lookups, the min bloom filter requires more memory, as values are stored instead of bits indicating set membership.

Compressed PPDBs and Min Bloom Filters The original PPDB paper proposed a form of *compressed PPDBs* (Anderson, Holte, and Schaeffer 2007) where the key for each entry is not stored. Instead, a hash function is used to map abstract states to entries. If a number of states map to the same entry, the min of these entries is stored. This is exactly a min Bloom filter using only a single hash function. Unfortunately, as we will show later, the use of additional hash functions did not improve performance in practice.

Compressed PPDBs vs. Min Compression While compressed PPDBs are a special case of min Bloom filters, they are also a special case of min compression using a randomized mapping instead of a structured one. In particular, min compression is commonly performed on k adjacent entries. If we were to use a general hash function to map all states to an array of size M/k and then take the min of each entry, we are still performing min compression, just with a different mapping of states. This is effectively the approach being used by compressed PPDBs. (Note that this is no longer true when we use a min Bloom filter with more than one hash function.)

If domain structure caused min compression using adjacent entries to perform poorly, this is an alternate approach to improve performance. In Top-Spin, for instance, adjacent entries did not compress well, and so entries with the same index $mod(M/k)$ were compressed together (Felner et al. 2007).

Theoretical Models

In this section we introduce theoretical models for the effectiveness of standard min compression of PDBs and for level-by-level Bloom filter compression.

Theoretical Model for Min Compression

One crucial question when performing min compression is the distribution of values within the PDB. In particular, if the PDB contains regions with similar heuristic values, it may be possible to compress the PDB with very little loss of information (Felner et al. 2007). This demands a domain-dependent analysis to find these mappings and is not possible for many domains. In particular, no such similar regions are known for the 12-edge PDB for Rubik's cube.

We build a theoretical model assuming that values are sampled uniformly across the PDB entries. Thus, when k entries are compressed into one bucket, the values in that bucket are chosen uniformly from the PDB and have the same distribution as the entire PDB. We simplify the analysis by sampling with replacement – ignoring that states are removed from the PDB when compressed. The expected value in the compressed PDB is determined by the expected min of the k states selected.

Assume we are compressing by a factor of k and that $P(h = i)$ denotes the probability that a uniformly randomly sampled state has an h-value of i in the uncompressed PDB. (This is simply the number of abstract states at depth i divided by the total number of abstract states.) $P(h \geq i)$ denotes the probability of obtaining an h-value of at least i, i.e., $P(h \geq i) = \sum_{j \geq i} P(h = j)$.

Let $P(v = i)$ be the probability that the minimum of k random selections has a value of i. We can rewrite this as follows using a reformulation of Bayes' Rule:

$$P(v = i) = \frac{P(v \geq i)}{P(v \geq i \mid v = i)} \cdot P(v = i \mid v \geq i) \qquad (3)$$

We always have $P(v \geq i \mid v = i) = 1$, so we can drop this term from the remaining equations.

k	L	Pred.	IPR	Actual	IPR	H
15	9.79	9.94	0.89	10.10	0.90	11.17
20	9.71	9.81	0.88	9.99	0.89	11.17
25	9.64	9.72	0.87	9.90	0.89	11.17
30	9.57	9.66	0.86	9.83	0.88	11.17
35	9.50	9.60	0.86	9.77	0.87	11.17
40	9.43	9.55	0.85	9.72	0.87	11.17
45	9.36	9.50	0.85	9.68	0.87	11.17
50	9.29	9.46	0.85	9.64	0.86	11.17

Table 3: Predicted vs actual min compression in the 12-edge PDB with various compression factors (k).

The probability that $v \geq i$ is easy to compute as it is:

$$P(v \geq i) = P(h \geq i)^k \tag{4}$$

The probability that $v = i$ given that $v \geq i$ is 1 minus the probability that all choices are strictly greater than i given that they are at least i:

$$P(v = i \mid v \geq i) = 1 - \left(\frac{P(h \geq i+1)}{P(h \geq i)} \right)^k \tag{5}$$

Putting these together, we get:

$$P(v = i) = P(h \geq i)^k \cdot \left(1 - \left(\frac{P(h \geq i+1)}{P(h \geq i)} \right)^k \right)$$

$$= P(h \geq i)^k - P(h \geq i)^k \left(\frac{P(h \geq i+1)}{P(h \geq i)} \right)^k$$

$$= P(h \geq i)^k - P(h \geq i+1)^k \tag{6}$$

If all h-values in the PDB are finite, the expected heuristic value in the compressed PDB is then:[3]

$$\begin{aligned} E[v] &= \sum_{i \geq 0} i \cdot P(v = i) = \sum_{i \geq 0} \sum_{j=0}^{i-1} P(v = i) \\ &= \sum_{j \geq 0} \sum_{i \geq j+1} P(v = i) \\ &= \sum_{j \geq 0} \sum_{i \geq j+1} (P(h \geq i)^k - P(h \geq i+1)^k) \\ &\overset{(*)}{=} \sum_{j \geq 0} P(h \geq j+1)^k = \sum_{i \geq 1} P(h \geq i)^k \end{aligned} \tag{7}$$

Step (*) exploits that $\sum_{i=a}^{b} (P(h \geq i)^k - P(h \geq i+1)^k)$ is a telescopic sum that simplifies to $P(h \geq a)^k - P(h \geq b+1)^k$ and that $\lim_{b \to \infty} P(h \geq b+1)^k = 0$.

[3] If they are not all finite, there is a nonzero chance of obtaining $v = \infty$, and hence $E[v] = \infty$.

Mem	$p_{fp}(8)$	$p_{fp}(9)$	h (Pred.)	h (Act.)	IPR
11 GB	2.8	3.7	9.895	9.894	0.89
16 GB	2.8	1.3	9.918	9.918	0.89
21.3 GB	1.4	0.6	9.952	9.951	0.89
31.6 GB	0.8	0.2	9.968	9.967	0.89

Table 4: Predicted and actual heuristic values for various false-positive rates.

A Metric for the Loss of Information

We now propose a metric that measures the loss of information of a given compression method. In the best case, there would be no loss of information and the expected value of the compressed heuristic would be the same as the uncompressed heuristic, denoted as H. In the worst case, with pathological compression, all entries could be combined with the goal state to return a heuristic of 0. If a compressed PDB has an average value of v the *information preserving rate* (IPR) is *IPR* $= \frac{v}{H}$. This is a value in $[0, 1]$ and indicates how much the information is preserved. The *loss of information rate* can be defined as $1 - \frac{v}{H}$.

Table 3 shows the heuristic predicted by Equation 7 and actual heuristic values for various min compression factors of the 12-edge Rubik's cube PDB. L indicates the lowest possible heuristic achievable from min compression, and H is the highest possible heuristic. We also show the predicted heuristic value as well as the actual heuristic value for each compression factor. While the predicted value is lower than the actual value, suggesting that the items are not stored completely at random with our ranking function, the difference is approximately 0.18 for all rows of the table.

Level by Level Bloom Filter Theory

We now turn to predicting the expected heuristic value of a Bloom filter. Using the false-positive rate of each level (given in Equation 2), we would like to predict the chance that a state is assigned a lower heuristic value than its true value. Assume that the false-positive rate at level i is $p_{fp}(i)$. Then, the probability of getting a heuristic value of i from looking up a state which is at level d for $i \leq d$ is:

$$p_d(i) = p_{true}(i) \cdot \prod_{j=0}^{i-1} (1 - p_{fp}(j)) \tag{8}$$

In Equation 8 the term $p_{true}(i)$ is the probability that we will get a $true$ value from a query at level i. For levels $\leq d$ this is exactly the false positive term (i.e., $p_{true}(i) = p_{fp}(i)$ for $i < d$). For level d, we set $p_{true}(d) = 1$ because at level d we will surely get a positive answer (a true positive). The expected h-value of a random state at depth d is:

$$E(h(\cdot) = d) = \sum_{i=0}^{d} p_d(i) \cdot i \tag{9}$$

The average compressed heuristic will then be:

$$\sum_{i=0}^{n} E(h(\cdot) = d) \cdot P(h = i) \tag{10}$$

k	Memory	Av. h	Nodes	Time
Full PDB (SSD)				
1	457.0	11.17	24,970,537	4,379.00
min compression (RAM)				
15	30.5	10.10	82,717,113	71.53
20	22.8	9.99	101,498,224	87.00
25	18.3	9.90	115,441,421	98.63
30	15.2	9.83	136,521,114	115.56
35	13.1	9.77	147,406,610	124.43
40	11.4	9.72	168,325,109	135.73
45	10.2	9.68	182,096,478	145.87
50	9.1	9.64	191,441,365	159.87
PPDB(9)	16.5	-	54,583,816	64.40

Table 5: Min Compression and PPDBs of the 12-edge PDB.

Memory	1 hash	2 hash	3 hash	4 hash
Nodes (Millions)				
10 GB	**214**	**214**	243	246
15 GB	166	**151**	162	183
20 GB	140	**118**	119	148
30 GB	133	86	**80**	**80**
Time (Seconds)				
10 GB	**147**	204	265	366
15 GB	**110**	141	195	261
20 GB	**94**	113	149	192
30 GB	**75**	81	105	113

Table 7: Min Bloom filter results for different sizes of memory and different number of hash functions.

Actual results are in Table 4. The predicted values are almost identical to the values achieved in practice, and the IPR does not vary significantly with the memory usage. We do not derive the expected min Bloom heuristic value here due to space limitations and the poor results they obtain below.

Experimental Results

We implemented all the ideas above for Rubik's cube with the 12-edge PDB; this section provides detailed results for many parameters of the approaches described in this paper. All results were run on a 2.6GHz AMD Opteron server with 64GB of RAM. All searches use IDA* (Korf 1985) with BPMX (Felner et al. 2011).

Min Compression

Table 5 reports the results for standard min compression[4] given the compression factor, k. It reports the number of nodes generated, average heuristic value, and time to solve the full problem set (same 100 problems). The h-value is the average heuristic of the compressed 12-edge PDB, although in practice we maximize these with the 8-corner PDB. The bottom line of the table is for level-by-level bloom filter with 4 hash functions. Here we used the hybrid approach where levels 1–7 were stored in a hash table[5] while levels 8 and 9 were stored in a level-by-level Bloom filter. Our Bloom filter used Zobrist (Zobrist 1970) hashes The results show that we can get the same CPU time as the min compression using half the memory, or nearly twice the time using the same amount of memory.

Bloom Filter Parameters

We note that Equation 2 for false positive depends on the following parameters: (1) m – the number of bits for the filter (2) q – the number of hash functions (3) n – the number of items stored. Since n is fixed, we can only vary m and q. Table 6 presents comprehensive results for the number of nodes (Top) and the CPU time (Bottom) of different combinations for these parameters. Each row represents a different memory/hash parameter for the level 8 Bloom filter, while each column represents a combination for the level 9 Bloom filter. In terms of nodes, four hash functions produce the best results. However, this isn't true for time. More hash functions require more time, and so better timing results are often achieved with fewer hash functions, even if the number of nodes increases.

Min Bloom Filters

We also experimented with min Bloom filters. Here, all levels of 1–9 were mapped to the same Bloom filter and each entry in the Bloom filter stored the minimum among all the values that were mapped to it. We used four bits per state in the min Bloom filter (versus one bit per state in a regular Bloom filter). We varied the number of hash functions and the size of the memory allowed. Table 7 provides the results.

Each row is for a different amount of memory (in GB) allocated for the Bloom filter while each column is for a different number of hash functions. We report the number of nodes (in millions) and the CPU time (in seconds). For each amount of allocated memory we have the best value in bold. Naturally, with more memory, the performance improves. However, the effect of adding more hash functions is more complex. Clearly, it adds more time to the process as we now have to perform more memory lookups. This seems to be very important in our settings and using only 1 hash function was always the best option in terms of time. Adding more hash functions has the following opposite effects on the number of nodes: **(1)** Each abstract state is now mapped into more entries so we take the maximum over more such entries. **(2)** More items are mapped into the same entry so the minimum in each entry decreases. In our settings, it turns out that the first effect was more dominant with larger amounts of memory while the second effect was more dominant with smaller amounts of memory. The best CPU time was always achieved with only 1 hash function. As explained previously, this is identical to the compressed PPDBs idea suggested by Anderson, Holte, and Schaeffer (2007).

[4]Tests suggested that a hash table with levels 1–7 and min compression of levels 8 and 9 would not improve performance.

[5]We used a standard library hash table for which we cannot tune performance or reliably measure memory usage. As such, we report the memory needed for the Bloom filters, which should be the dominant factor.

Nodes generated (in Million)																		
			Depth 9 Bloom filter															
			10 GB Storage				15 GB Storage				20 GB Storage				30 GB Storage			
		H	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4
Depth 8 Bloom filter	1 GB	1	125	99	92	89	110	88	83	81	102	84	80	79	95	81	79	78
		2	105	80	74	71	91	70	66	64	84	66	63	62	76	64	62	61
		3	100	76	69	67	86	66	61	60	79	62	59	58	72	59	57	57
		4	98	74	68	**66**	85	64	60	**58**	78	61	57	**56**	70	58	**56**	**56**
	1.3 GB	1	118	92	85	82	103	81	76	75	95	77	74	73	88	74	72	72
		2	100	76	70	67	86	66	61	60	79	62	59	58	72	59	58	57
		3	96	73	66	64	83	63	58	56	76	59	56	55	69	56	54	54
		4	95	71	65	**63**	82	62	57	**55**	75	58	55	**54**	67	55	**53**	**53**
	1.6 GB	1	113	87	81	78	98	77	72	70	91	73	70	69	83	70	68	68
		2	98	74	67	65	84	64	59	58	77	60	57	56	70	57	56	55
		3	95	71	65	63	81	61	57	55	74	58	54	**53**	67	55	53	53
		4	94	70	64	**62**	80	61	56	**54**	73	57	54	**53**	66	54	**52**	**52**

CPU Time (in seconds)																		
			Depth 9 Bloom filter															
			10 GB Storage				15 GB Storage				20 GB Storage				30 GB Storage			
		H	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4
Depth 8 Bloom filter	1 GB	1	131	112	110	103	107	88	86	88	110	85	84	85	93	81	82	84
		2	115	91	**84**	93	93	73	71	72	85	76	75	77	78	73	72	74
		3	113	88	90	91	102	79	76	78	93	67	72	73	84	**70**	**70**	72
		4	115	89	91	91	92	82	**69**	70	84	67	**65**	66	85	**70**	**70**	71
	1.3 GB	1	125	100	94	94	112	91	88	90	95	87	85	79	94	82	82	84
		2	111	86	88	87	97	70	67	75	90	66	64	66	81	61	67	69
		3	108	84	85	80	97	75	**66**	67	79	70	68	70	79	60	**59**	67
		4	109	84	84	**79**	99	76	**66**	73	85	70	**62**	70	72	66	66	61
	1.6 GB	1	119	95	89	90	107	79	84	79	90	74	73	82	82	77	77	79
		2	108	84	77	77	97	67	65	66	79	63	**61**	63	78	65	65	67
		3	107	82	**76**	**76**	95	73	**64**	65	77	68	66	67	70	**58**	64	60
		4	107	82	**76**	**76**	98	74	**64**	**64**	88	69	67	68	78	**58**	64	66

Table 6: Nodes and time for solving different combinations of level 8 (vertical) and level 9 (horizontal) Bloom filters.

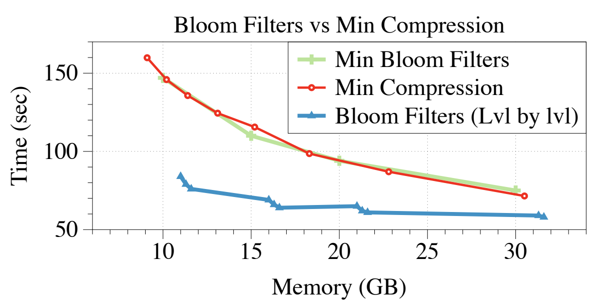

Figure 4: Comparison of min compression, level-by-level Bloom filters and min Bloom filters.

Large Problems

To establish a comparison with previous research, we also ran on 10 random instances (Korf 1997). Bloom filters of size 1.6 GB and 15.0 GB for levels 8 and 9 respectively could solve these problems using 6.4 billion generations as opposed to 14.0 billion (Breyer and Korf 2010) using 1.2 GB. Our code takes on average 7135s to solve each problem; using 30 GB for level 9 decreased the average time taken to 6400s.

Final Comparison

Finally, we compared the three approaches: min compression, level-by-level Bloom filter, and min Bloom filters for various amounts of memory. With the Bloom filters we selected the best settings from Tables 6 and 7. Figure 4 shows the average time needed to solve the test suite as a function of the available memory. Level-by-level Bloom filters clearly outperform the other forms of compression. It achieves near-optimal performance (defined by the best-case PPDB) with far less memory than min compression.

The timing results for min Bloom filter and min compression are nearly identical with even a small advantage to min Bloom filters that likely results from faster hash lookups.

Conclusions

The goal of this research was to study the Rubik's Cube 12-edge PDB; our best performance was achieved with (hybrid) level-by-level Bloom filter PPDBs. But, we have only touched the surface of the potential of Bloom filters by demonstrating their benefits on our single test domain. In the future we aim to continue by testing different sizes of PDBs and different domains. We are also studying ways to use more of the level 10 Rubik's cube data to further improve performance.

Acknowledgments

Computing resources were provided from machines used for the NSF I/UCRC on Safety, Security, and Rescue.

This research was partly supported by the Israel Science Foundation (ISF) under grant #417/13 to Ariel Felner.

This work was partly supported by the Swiss National Science Foundation (SNSF) as part of the project "Abstraction Heuristics for Planning and Combinatorial Search" (AHPACS).

References

Anderson, K.; Holte, R.; and Schaeffer, J. 2007. Partial pattern databases. In *Symposium on Abstraction, Reformulation and Approximation (SARA)*, 20–34.

Ball, M., and Holte, R. 2008. The compression power of symbolic pattern databases. In *International Conference on Automated Planning and Scheduling (ICAPS-08)*, 2–11.

Bloom, B. H. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* 13(7):422–426.

Bose, P.; Guo, H.; Kranakis, E.; Maheshwari, A.; Morin, P.; Morrison, J.; Smid, M.; and Tang, Y. 2008. On the false-positive rate of bloom filters. *Inf. Process. Lett.* 108(4):210–213.

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

Christensen, K.; Roginsky, A.; and Jimeno, M. 2010. A new analysis of the false positive rate of a bloom filter. *Inf. Process. Lett.* 110(21):944–949.

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

Döbbelin, R.; Schütt, T.; and Reinefeld, A. 2013. Building large compressed PDBs for the sliding tile puzzle. In Cazenave, T.; Winands, M. H. M.; and Iida, H., eds., *CGW@IJCAI*, volume 408 of *Communications in Computer and Information Science*, 16–27. Springer.

Edelkamp, S., and Kissmann, P. 2008. Partial symbolic pattern databases for optimal sequential planning. In Dengel, A.; Berns, K.; Breuel, T. M.; Bomarius, F.; and Roth-Berghofer, T., eds., *KI*, volume 5243 of *Lecture Notes in Computer Science*, 193–200. Springer.

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.

Hart, P.; Nilsson, N. J.; and Raphael, B. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics* 4:100–107.

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.

Mitzenmacher, M., and Upfal, E. 2005. *Probability and computing - randomized algorithms and probabilistic analysis*. Cambridge University Press.

Myrvold, W., and Ruskey, F. 2001. Ranking and unranking permutations in linear time. *Information Processing Letters* 79:281–284.

Samadi, M.; Siabani, M.; Felner, A.; and Holte, R. 2008. Compressing pattern databases using learning. In *European Conference on Artificial Intelligence (ECAI-08)*, 495–499.

Sturtevant, N. R., and Rutherford, M. J. 2013. Minimizing writes in parallel external memory search. In *International Joint Conference on Artificial Intelligence (IJCAI)*, 666–673.

Zobrist, A. L. 1970. A new hashing method with application for game playing. Technical Report 88, U. Wisconsin CS Department.