Using Partitions and Superstrings for Lossless Compression of Pattern Databases

Ethan L. Schreiber and **Richard E. Korf**
Department of Computer Science, University of California, Los Angeles
Los Angeles, California 90095
ethan@cs.ucla.edu, korf@cs.ucla.edu

Abstract

We present an algorithm for compressing pattern databases (PDBs) and a method for fast random access of these compressed PDBs. We demonstrate the effectiveness of our technique by compressing two 6-tile sliding-tile PDBs by a factor of 12 and a 7-tile sliding-tile PDB by a factor of 24.

Introduction

A pattern database (PDB) (Culberson and Schaeffer 1996) is a precomputed table storing the cost of solving all subgoals of a particular type from a larger problem. For example, with sliding-tile puzzles, a PDB stores the number of moves a subset of N tiles need to make from each of their possible configurations to the goal state. In practice, PDBs typically are stored as large arrays of relatively small numbers.

Algorithms such as A* (Hart, Nilsson, and Raphael 1968) and IDA* (Korf 1985) find optimal solutions to search problems using a cost function $f(n) = g(n) + h(n)$ to direct the search where $g(n)$ is the cost from the start state to node n and $h(n)$ is an estimate of the cost from node n to a goal. If $h(n)$ never overestimates the true cost, we call it admissible, and can use it with A* or IDA* to find optimal solutions. A set of PDBs can be used to compute an admissible $h(n)$.

A problem with PDBs is that they can become quite large. For example, in the twenty-four puzzle (Korf and Felner 2002), an N-tile PDB has $\frac{25!}{(25-N)!}$ entries. With one byte per entry, this requires approximately 122MB, 2.25GB and 40.6GB for N=6,7 and 8 respectively. A natural solution to this is to compress the PDBs. Various Groups (Felner et al. 2007; Breyer and Korf 2010; Ball and Holte 2008) have examined both lossy and lossless methods for compressing PDBs. Our methods are lossless, domain independent and fast for random access. Furthermore, we attain a better compression ratio than the existing lossless methods.

In the next three sections, we will describe the components of our algorithm. This is followed by an example to clarify our methods and illuminate each of the components.

Difference Arrays

While in general, a PDB stores the total cost from a node to the goal state, we can alternatively store values other than total cost. For example, it is common practice with sliding-tile puzzles to subtract the Manhattan distance (MD) from the cost stored in the PDB. While performing our search, we compute the MD for a state and add the PDB value to it.

Similarly, in more general terms, we can subtract the value of a subset PDB, which typically is much smaller. For example, we could compute a 5-tile and 6-tile PDB where all of the tiles in the 5-tile PDB are also in the 6-tile PDB. The 6-tile PDB is 122MB while the 5-tile is only 6MB. In the 6-tile PDB, we store the difference between the total cost of solving the 6 tiles minus the total cost of solving the 5 tiles. While performing the search, we lookup both the 5 and 6-tile values and sum them as our heuristic function. This method generalizes to any domain in which we can compute PDBs.

In our experiments, difference arrays tend to have many sequences of values that repeat in the array, especially zeros.

Partioning the PDB

How can we exploit the repetitive nature of the PDB difference array? For a PDB of size n, we partition it into $\frac{n}{k}$ sub-arrays each of size k. We save space by storing only the unique sub arrays. Instead of having a PDB of size n with many repeated sub-arrays, we store a PDB of size $\frac{n}{k}$ with each element being an index into a second array which stores one copy of each of the unique sub-arrays.

Shortest Common Superstring

The shortest common superstring (SCS) problem (Tarhio and Ukkonen 1988) is given a set S of strings over the alphabet Σ, find the shortest string that contains all $s_i \in S$. This is a standard problem from computational biology. It is MAX SNP-hard (Papadimitriou and Yannakakis 1988), but it has been proven that the greedy algorithm finds strings within four times the optimal length. (Blum et al. 1994). This algorithm repeatedly chooses the two strings from S with maximum overlap and merges them until S contains one string which we call the superstring.

After partitioning the PDB, we are left with a set of unique sub-arrays. We compress this set using the greedy algorithm to find an approximation of the SCS. We modify the PDB index array so that each index references the index within the superstring where the original sub-array begins.

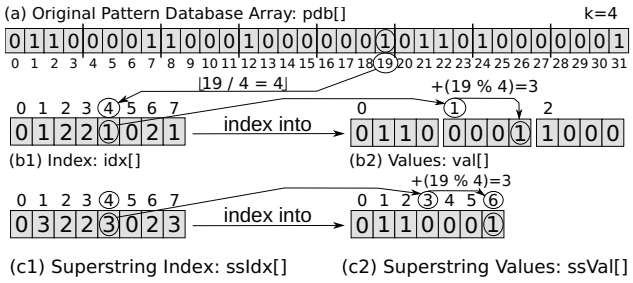

Figure 1: A compressed PDB example described below.

An Example of Compression and Lookup

Figure 1 illustrates an example of our compression technique. (a) is our original uncompressed PDB array of size $n = 32$ called pdb[]. In this case, each entry is 0 or 1 for clarity but in general each element could be any number. We partition this array into sub-arrays of k=4 elements each. There are $\frac{n}{k} = 8$ such sub-arrays. (b2) is the array of unique sub-arrays from (a) called val[]. (b1) is the array of $\frac{n}{k}$ indices called idx[], one for each of the original sub-arrays.

To retrieve the value pdb[19] (whose value is 1) from the compressed array, we first calculate $\lfloor \frac{19}{4} \rfloor = 4$. That is, we take the floor of the original index divided by the partition size. This is the index into the array idx[4]=1. The 1 is our index into val[1]=$\{0,0,0,1\}$. To attain the offset into this array, we compute $19 \ \% \ 4 = 3$. The 3 is the offset into val[1][3] = 1, the same value as pdb[19], our target value.

Arrays (c1) and (c2) are the superstring analogs of (b1) and (b2). We use the greedy superstring algorithm to compress val into ssVal. The sub-array val[0] begins at ssVal[0], val[1] at ssVal[3] and val[2] at ssVal[2]. With the same example of finding pdb[19], we lookup ssIdx$\left[\lfloor \frac{19}{4} \rfloor = 4\right]$ which is 3. We then lookup ssVal[$3 + (19 \ \% \ 4)$] = ssVal[6] = 1 which is the same as pdb[19].

Preliminary Experimental Results

We generated 3 PDBs for the twenty-four puzzle containing tiles: $\{3,4,8,9,13,14\}$, $\{1,2,5,6,7,12\}$ and $\{3,4,8,9,13,14,15\}$ with one byte per entry. We call our PDBs *6-regular*, *6-irregular* and *7* respectively. We compressed each with a sub-array size of 128 entries. We report the uncompressed size, the compressed size and the ratio. Our results:

	6-regular	*6-irregular*	*7*
uncompressed	121.6MB	121.6MB	2.26GB
compressed	10.82MB	9.0MB	96.48MB
ratio	11.24	13.5	23.95

Using the methods of and data from (Korf and Felner 2002), we ran IDA*. We used both the compressed and uncompressed version of 6-regular and 6-irregular as our heuristics. IDA* with the uncompressed PDBs generated 8,860,660 nodes per second while the compressed version generated 6,304,690 nodes per second.

Discussion and Future Work

We have described a general technique for lossless compression of PDBs and have demonstrated a proof of concept based on compressing the 6-tile sliding-tile PDB and successfully using it with IDA*. The true value of this technique will be compressing and using larger PDBs that have previously been too large to store in memory.

We are currently working on compressing the 40.6GB 8-tile sliding-tile PDB. Given the large size of this PDB, there are additional engineering complications such as disk based search both in creating and compressing the PDB. We believe that when we are done, we will be able to use 3 8-tile PDBs in a reasonable amount of main memory to search the 24-puzzle. Furthermore, we plan to explore other domains and the effectiveness of larger PDBs in solving problems such as the 4-peg Towers of Hanoi and Rubik's cube puzzle.

We see room for improvement in terms of compression ratio by experimenting with alternative methods for creating difference arrays. Beyond this, we will look to explore other domains outside of permutation problems in which we can use our ideas for compression of sparse arrays.

Acknowledgements

This work was supported in part by NSF grant IIS-0713178.

References

Ball, M., and Holte, R. 2008. The compression power of symbolic pattern databases.

Blum, A.; Jiang, T.; Li, M.; Tromp, J.; and Yannakakis, M. 1994. Linear approximation of shortest superstrings. *J. ACM* 41:630–647.

Breyer, T. M., and Korf, R. E. 2010. 1.6-bit pattern databases. In *AAAI*.

Culberson, J. C., and Schaeffer, J. 1996. Searching with pattern databases. In *Advances in Artificial Intelligence (Lecture Notes in Artificial Intelligence 1081*, 402–416. Springer-Verlag.

Felner, A.; Korf, R. E.; Meshulam, R.; and Holte, R. C. 2007. Compressed pattern databases. *J. Artif. Intell. Res. (JAIR)* 30:213–247.

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

Korf, R. E., and Felner, A. 2002. Disjoint pattern database heuristics. *Artif. Intell.* 134(1-2):9–22.

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

Papadimitriou, C., and Yannakakis, M. 1988. Optimization, approximation, and complexity classes. In *Proceedings of the twentieth annual ACM symposium on Theory of computing*, STOC '88, 229–234. New York, NY, USA: ACM.

Tarhio, J., and Ukkonen, E. 1988. A greedy approximation algorithm for constructing shortest common superstrings. *Theor. Comput. Sci.* 57:131–145.