Solving the 15-puzzle using Pattern Databases

Matheus Jun Ota
University of New Hampshire
mo1045@wildcats.unh.edu

Abstract

The 15-puzzle has a huge number of possible permutations. Searching for the goal arrangement using A* with heuristics like Misplaced Tiles or Manhattan Distance may not compute an optimal solution to the puzzle in a feasible time. The solution presented in this paper is using another heuristic called Pattern Databases that has the purpose of mapping in a table the solution of a smaller problem, which is called a pattern. The construction of a Pattern Database can be made on a pre-processing step by doing a backwards Breadth-first search from the target pattern. The first time 15-puzzle was solved they used IDA*, a linear memory version of classic A*. Today's systems have much more memory available so we compared the efficiency of both A* and IDA* on random instances of the problem.

Introduction

When talking about searching algorithm it is common to talk about the CLOSED list, which contains the nodes already expanded, and the OPEN list, which contains nodes that were generated but not expanded. The Best-first search algorithms are those that visit the nodes at the OPEN list in the order of a particular f(n) function. The most well-known Best-first search algorithm is A*(Hart, Nilsson & Raphael, 1968), which defines f(n) as being the sum of g(n) and the heuristic h(n), where g(n) is the cost to reach node n and h(n) is the estimated cost of the optimal path from node n to the goal. If h(n) does not overestimate the true cost, then A* is both complete and optimal.

One famous heuristic for the 15-puzzle is the Manhattan Distance(MD), which consists of computing the sum of the Manhattan Distances of each numbered tile. Unfortunately using A* and MD is not a good approach for the 15-puzzle because of two reasons: A* has an exponential space complexity, and Manhattan Distance is not a very good approximation to the true cost of reaching the goal - making the search visit too many nodes.

The goal of this paper is to solve the 15-puzzle using a better way of computing heuristics called Pattern Databases(Culberson & Schaeffer, 1998) and Additive Pattern Databases(Felner, Korf & Hanan, 2004) while searching with A* and also with a modified version of A* called Iterative Deepening A*, or IDA*(Korf, 1985), which have linear space complexity.

$(n^2 - 1)$-puzzles

The 15-puzzle or the (n^2-1)-puzzles, in general, consists of a grid of size n x n with numbered tiles in random order and one tile missing. Moving only tiles adjacent to the blank tile, one needs to reach a goal position, where the numbered tiles are ordered. Sliding-puzzles like these are NP-hard problems(Ratner & Warmuth, 1986) and have long been used as a domain for searching algorithms because of their huge state space.

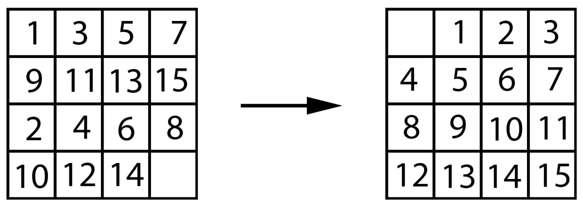

Figure 1: Example of a start state and goal state for the 15-puzzle

Before proceeding to the method used to solve such a puzzle, it is worth to mention that, contrary to intuition, the state space is not $(n^2-1)!$, in other words, not all permutations are reachable from the goal state. Johnson & Story (1879) used a parity argument to show that only half of these permutations are reachable. For example, Figure 2 shows the 14-15 Puzzle, created by the famous puzzle maker Sam Loyd, and it is impossible to solve, because the initial state is unreachable from the goal(Loyd & Gardner, 1959).

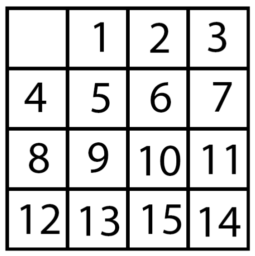

Figure 2: Initial state of the 14-15 Puzzle

Pattern Databases

We define patterns as partial specifications of a state, while Pattern Databases(PDBs) are lookup tables that specify for each pattern the cost to reach the target pattern(the goal). In the context of the 15-puzzle the patterns can be created by not specifying certain tiles.

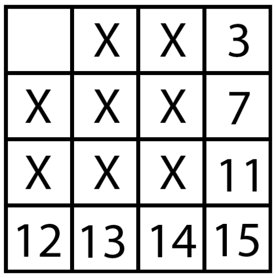

Figure 3: The fringe pattern

To create a PDB we apply Breadth-first search on the target pattern and associate each pattern n (or node)in a hash table with the value of the cost to reach it. This value can then be used as the heuristic value for the same state that have all the tiles numbered and follows the pattern. To understand this, one can observe that to solve the whole problem, we also need to solve the pattern, because the target pattern is part of the goal state. Since the pattern has a smaller quantity of numbered tiles, the cost to reach the solution to the pattern will never overestimate the true cost of the original puzzle, making the heuristic admissible.

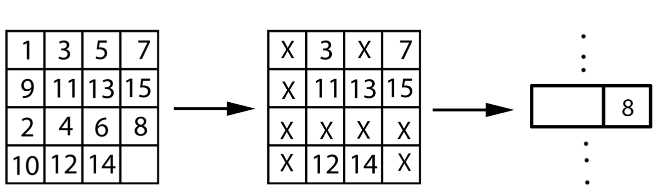

Figure 4: Example of the process to find a heuristic value in the PDB. We first pick the original state and then abstract it, obtaining a corresponding pattern. Next, we use this pattern to search at a hash table and get the associated heuristic value.

Currently PDBs are the best heuristics that can be used to solve some hard combinatorial puzzles like the $(n^2 - 1)$-puzzle, the 4-Peg Towers of Hanoi or the Rubik's Cube(Korf, 1997). Although the computer may take a huge time to compute the PDB, it only needs to be done once, therefore the cost can be amortized between all the times a search process is ran using the PDB.

Additive Pattern Databases

So far, the approach when using Pattern Databases just uses one at a time, or the maximum of two PDBs, it would be more interesting if our implementation could use separate PDBs together. One possible way of achieving this is by using Additive Pattern Databases, where we use the sum of the heuristic value on the different PDBs.

To use Additive PDBs we first need to partition the problem into smaller sets of disjoint patterns. In the case of the 15-puzzle, this means that no numbered tile will be in two patterns at the same time.

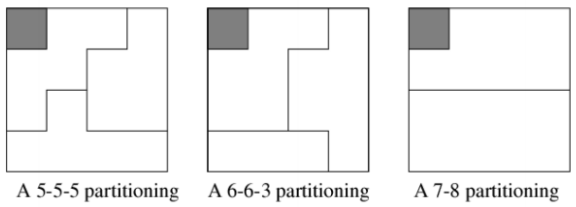

Figure 5: Some possible partitions for the 15-puzzle

The creation process follows just like with normal PDBs with the difference that, when computing the values on the tables, instead of counting all moves to reach a node, we just count the moves of the tiles that are part of the pattern. Although this makes the heuristic value smaller, and therefore pruning a smaller number of nodes, it guarantees that the Additive Pattern Databases heuristic is admissible.

Hashing Method

When inserting a pattern, or a full state, into a hash table we are concerned about two major processes: ranking and hashing. The first one is how we convert the grid to an integer and the latter is how we use this integer to compute an index on the hash table.

For ranking we first represent the grid in an array format, representing the blank with a 0(zero). To illustrate this concept better, the starting state on figure 1 would be like:

$$(1, 3, 5, 7, 9, 11, 13, 15, 2, 4, 6, 8, 10, 12, 14, 0)$$

When dealing with a pattern, we assign -1 to unspecified tiles, therefore the fringe pattern would be like:

$$(-1, -1, -1, 3, -1, -1, -1, 7, -1, -1, -1, 11, 12, 13, 14, 15)$$

Next we initiate a 64-bit integer u with 0(zero) and iterate through the array representation so that each 4-bit group that starts at $u[4i]$ and ends at $u[4i + 3]$ represents the position of the tile of number i in the grid. For example, if we rank the starting state of figure 1, then the bits $u[20]$ up to $u[23]$ would store the value of 2, which is the position of tile 5. We ignore the tile if it is unspecified($i = -1$). The following is the pseudocode of this algorithm:

```
function RANKSTATE(grid)
    array ← array representation of grid
    $u \leftarrow 0$
    for $i \leftarrow 0, 15$ do
        $x \leftarrow array[i]$
        if $x \neq -1$ then
            $u| = i << (4x)$
        end if
    end for
    return u
end function
```

Once we have the integer representation of the grid, we need to use it to find the index on the hash table, for this we simply define a function $hash(u) = u\ mod\ k$ where k is a prime number and also the size of the hash table. Although this is a rather simple hashing function, we used it because its easy implementation.

Iterative Deepening A*

So far we have talked about the heuristic, but we did not talk about the search algorithm that uses this heuristic. The most popular heuristic search is definitely A*, but it cannot solve some instances of the 15-puzzle using Manhattan Distance because of the memory overflow.

To overcome this problem we will start thinking about Depth-first Search, which is linear in memory, because you just need to save the path to the current node from the root. The problem with DFS is that, although it is guaranteed to find a solution, it may not be optimal. In the the context of the sliding-puzzles, this means that the solution is not with the minimum number of steps. The Iterative Deepening Depth-first Search(ID-DFS) addresses this problem by iteratively running DFS with an incrementing bound, limiting the depth of the search. The following code illustrates better the ID-DFS:

```
function ID-DFS(root)
    $i \leftarrow 1$
    while Solution not found do
        DFS(root) with bound $g(n) \leq i$
        $i++$
    end while
end function
```

Finally, the idea of the Iterative Deepening A* is to make the ID-DFS use the information given by the heuristic(just like A* does) so the search visits a smaller number of nodes. Therefore, instead of limiting on the depth of the search $g(n)$, we will limit on the $f(n) = g(n) + h(n)$ value. The following is the code of IDA*:

```
function IDA*(root)
    $i \leftarrow h(root)$
    while Solution not found do
        DFS with bound $g(n) + h(n) \leq i$
        $i \leftarrow$smaller f(n) that overpass $i$
    end while
end function
```

Implementation

The whole code of both the solver and the pdb creator was done in Standard C(C-99) and was all written by the author. During most of the program the puzzle is represented by an integer matrix, which is not the most efficient way(we could use an array of chars instead), but was the simple to implement. To ensure that our results were correct we compared it to other optimal sliding-puzzle solvers(Dramaix, 2012).

Empirical Results

Firstly, we tested our solver of the 15-puzzle in 10 random instances of Korf's test suite(Korf, 1985) and compared the time of computation, the numbers of nodes expanded and the number of nodes generated. We evaluated A* and IDA* with Manhattan Distance, 5/5/5 PDBs and 6/6/3 PDBs. The results are shown on table 2 and table 3. To evaluate the time taken by the algorithm while only performing the search, we also computed the time to transfer the PDB from the disk to the memory(table 1).

	5/5/5 PDB	6/6/3 PDB
time(s)	0.727115	5.126073

Table 1: Time taken to transfer the PDB from disk to memory.

	A* - MD	A* - 5/5/5 PDB	A* - 6/6/3 PDB
time(s)	12.154277	1.453978	5.809331
expanded	4067541.5	212166.9	127031.1
generated	7518747.2	430230.3	250090.3

Table 2: Mean of the table 2 values for A*. Reduction of 94.8% of nodes expanded for 5/5/5 and 96.8% for 6/6/3

	IDA* - MD	IDA* - 5/5/5 PDB	IDA* - 6/6/3 PDB
time(s)	12.866775	1.990139	6.387031
expanded	12995446.2	1183247.5	911386.5
generated	12995448.5	1183253.7	911393.4

Table 3: Mean of the table 2 values for IDA*. Reduction of 90.9% of nodes expanded for 5/5/5 and 93% for 6/6/3

Felner, Korf & Hanan(2004) reports a reduction of 99% compared to Manhattan Distance running IDA*. We expect that our code would achieve this performance if we had ran it through more tests.

Next, we plotted the histograms of the PDBs for the purpose of visualizing the distribution of the heuristic values. When looking at the histograms we noted that the heuristics on both partitions are close to a gaussian distribution.

Histograms of the 5/5/5 PDB

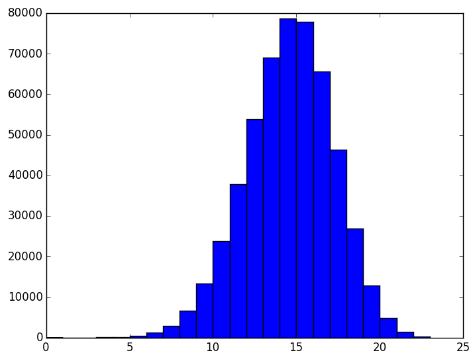

Figure 6: Pattern with tiles 9, 12, 13, 14 and 15 mapped

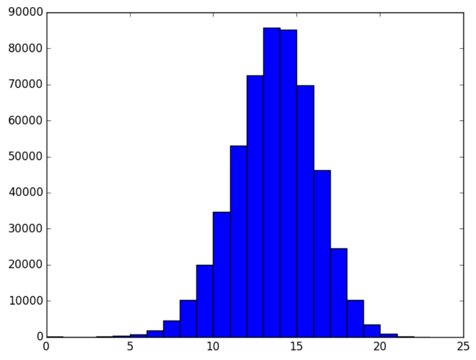

Figure 7: Pattern with tiles 3, 6, 7, 10 and 11 mapped

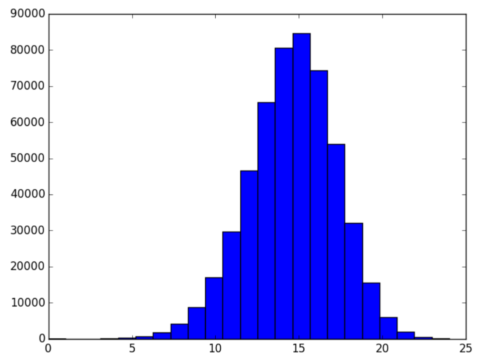

Figure 8: Pattern with tiles 1, 2, 4, 5 and 8 mapped

Histograms of the 6/6/3 PDB

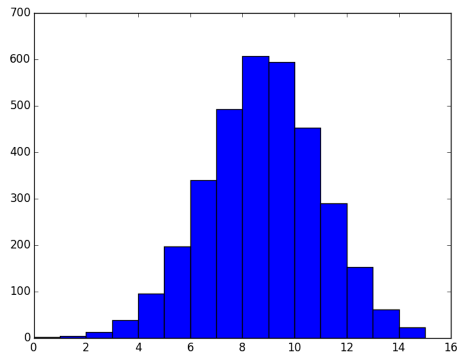

Figure 9: Pattern with tiles 12, 13 and 14 mapped

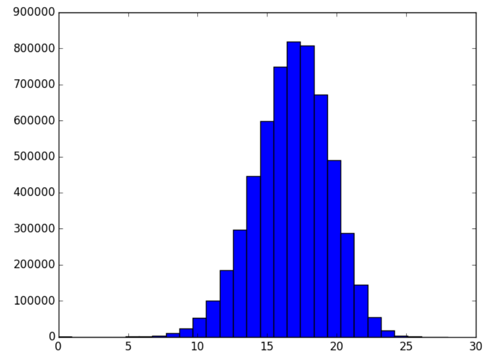

Figure 10: Pattern with tiles 3, 6, 7, 10, 11 and 15 mapped

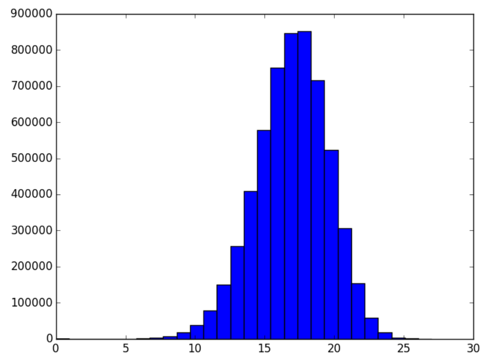

Figure 11: Pattern with tiles 1, 2, 4, 5, 8 and 9 mapped

Comparing Histograms

The following picture was obtained by normalizing the histograms, so the area of the bars sum to one, and comparing the best 5/5/5 PDB(tiles 3, 6, 7, 10, 11) with the best 6/6/3 PDB(tiles 1, 2, 4, 5, 8, 9).

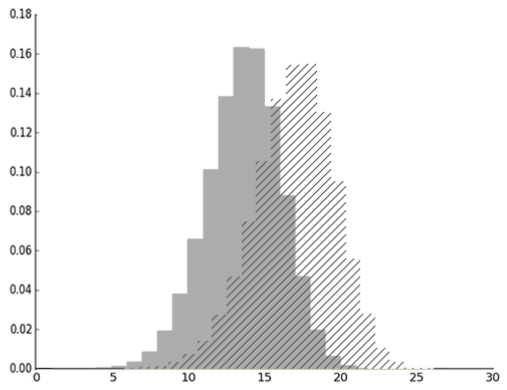

Figure 12: Comparing the patterns. The grey area is the 5/5/5 PDB and the dashed area is the 6/6/3

When looking at the previous tables we note that the 6/6/3 PDBs is more efficient. Since the 5/5/5 histogram have a highest peak, while the 6/6/3 have bigger heuristic values. We conjecture then that the size of the heuristic values is more important then the frequency when analyzing Pattern Databases.

Future Work

Unfortunately, we could not generate the 7/8 PDBs without a memory overflow. However, some optimizations could be done to surpass this problem. We could use chars instead of integers to represent a tile(chars have 8 bits while integers have 32) and we could store the PDB only on the magnetic disk, by using some kind of Database Management System like SQL, instead of storing them into the memory and then saving it to the disk. This would also improve the performance of the solver, because the program would not need to transfer the whole PDB from a file to the RAM memory before starting the search. Therefore, we expect not only to compute the 7/8 PDBs but also to compute PDBs to solve the 24-puzzle.

Another improvement could be made by using a better hash function like FNV(Fowler, Noll & Vo, 2012) achieving a better distribution of states per bucket. Currently approximately 10% of buckets have more than 1 state.

Finally, we could remember that IDA* does not keep a record of the CLOSED list, therefore it cannot detect duplicates states. One possible way to address this problem could be by pruning Duplicate Nodes with a Finite State Machine, which was reported as solving 97.4% of nodes generated on depth 50(Taylor & Korf, 1993).

Conclusion

Pattern Databases are very powerful tools for the development of heuristic search algorithms. In this paper we presented some empirical results on the context of the 15-puzzle, which explicitly showed that Pattern Databases are much better than Manhattan Distance. Most of the research done in this paper was already well known to the field, but we believe that our work bring them together and have some implementation details that were not so explicit in previous works - for example, the hashing process was not found on the references used - and some new data, like the histogram approach to comparing PDBs.

Acknowledgments

I would like to thanks professor Wheeler Ruml for proposing the use of histograms and comparing them. I would also like to thanks professor Carlos Linares López of the University Carlos III for answering a lot of questions I had when creating this project.

References

Russell, Stuart J.; Norvig, Peter (2003), Artificial Intelligence: A Modern Approach (2nd ed.), Upper Saddle River, NJ: Prentice Hall, ISBN 0-13-790395-2

Korf, R. E. (1997, July). Finding optimal solutions to Rubik's Cube using pattern databases. In *AAAI/IAAI* (pp. 700-705).

Felner, A., Korf, R. E., & Hanan, S. (2004). Additive pattern database heuristics. *J. Artif. Intell. Res.(JAIR)*, 22, 279-318.

Culberson, J. C., & Schaeffer, J. (1998). Pattern databases. *Computational Intelligence, 14(3)*, 318-334.

Ratner, D., & Warmuth, M. K. (1986, August). Finding a Shortest Solution for the N x N Extension of the 15-PUZZLE Is Intractable. In *AAAI* (pp. 168-172).

Story, W. E. (1879). Notes on the" 15" Puzzle. *American Journal of Mathematics, 2(4)*, 397-404.

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

Loyd, S., & Gardner, M. (1959). *Mathematical puzzles (Vol. 1)*. Courier Corporation.

Dramaix, J. (2012). *How to solve the famous 15 sliding-tile puzzle*. Retrieved from https://plus.google.com/u/0/+JulienDramaix/posts/4vLG9oghrLy

Fowler, G., Noll, L. C. & Vo, K. (2012) *The FNV Non-Cryptographic Hash Algorithm* Retrieved from https://tools.ietf.org/html/draft-eastlake-fnv-03