Solving a 15 puzzles using IDA* and Pattern Database Heuristics
A Report for CMPUT 658 Single Agent Search

Talha Ibn Aziz[1]

[1]Student ID: 1669108
Email: taziz@ualberta.ca
Course Instructor: Nathan R. Sturtevant
University of Alberta

Abstract

One of the most popular sliding tile puzzles is the 15-puzzle which is a 4×4 grid containing 16 values starting with 0 and ending with 15 - hence, the name '15-puzzle'. Sliding tile puzzles are great problems for applying single-agent search algorithms as there is one goal state for every possible problem instance. Researchers have performed many experiments using the puzzle as a problem set. In this paper, I solve multiple instances of the 15-puzzle using IDA* and Pattern Database Heuristics.

1 Introduction

The sliding tile puzzles have been a source of entertainment, brain-teasing, and research for the past 2-3 centuries. These puzzles are usually arranged as a square set of moving tiles of different sizes - (4×4), (5×5), etc. (the shape of the puzzle may vary). Each tile is numbered from 1 to $n - 1$, where n is the total number of tiles in the puzzle. There is a blank tile (often denoted as a '0') which can be used to move the other tiles by sliding the adjacent tiles towards the blank. In this way the tiles adjacent to the blank, can be swapped with the blank tile to change the orientation of the puzzle. The end goal is to form an orientation where the tile numbers are ordered start from the 0 (blank) tile to the $n - 1^{th}$ tile. Each swapping with the blank tile can be considered as a *move* and the sequence of moves to reach the end goal is called a valid *solution* to a puzzle instance. For example, the given scenario below, models a 3-move solution for a 4×4 puzzle instance, which is popularly known as the "15-puzzle" (Johnson, Story et al. 1879):

$$\begin{bmatrix} 1 & 5 & 2 & 3 \\ 4 & 9 & 6 & 7 \\ 8 & 0 & 10 & 11 \\ 12 & 13 & 14 & 15 \end{bmatrix} \rightarrow UP \rightarrow \begin{bmatrix} 1 & 5 & 2 & 3 \\ 4 & 0 & 6 & 7 \\ 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 5 & 2 & 3 \\ 4 & 0 & 6 & 7 \\ 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 \end{bmatrix} \rightarrow UP \rightarrow \begin{bmatrix} 1 & 0 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 \end{bmatrix} \rightarrow LEFT \rightarrow \begin{bmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 \end{bmatrix}$$

In recent years, much research has been done to solve the puzzles using the minimum number of moves. The *best solution* to a puzzle instance is the shortest sequence of moves required to convert the initial puzzle orientation into the end goal - which is the sorted puzzle. For any given instance, there are four possible moves - moving the blank tile up, down, left, or right. Moving a blank tile means swapping with an adjacent tile, unless there is no tile in the mentioned direction (i.e. the blank has reached an edge of the puzzle).

In this paper, I solve a standard set of 15-puzzle instances (Korf 1985) using well-known search algorithms and techniques. Section 2 describes the different search algorithms, section 3 details the implementation of the procedure used, section 4 presents the experimental results, and section 5 concludes the paper.

2 Background

The sliding tile puzzles always have a fixed goal for a fixed sized puzzle and there is usually only one solver. Therefore, mostly Single-Agent Search techniques are used to sovle these puzzles. One approach can be to use Breadth-First Search (BFS) to find the shortest possible sequence of moves to the end goal. This is possible because BFS finds the shortest path to the goal while performing only the necessary number of node expansions (meaning it is optimal). However, being an uniformed search technique, BFS requires the ability to store all the necessary node states or at least their representation to solve it. This memory-hungry approach becomes impractical when the search space exceeds the memory limit of the system.

The 15-puzzle - for instance, has $20,922,789,888,000$ (16!) or in short, nearly 20 trillion possible states. In the worst case scenario, even if only one bit is used per state, the storage space required would be 2.38 TB (terabytes), approximately. This would require a high-end personal computer to use their entire Hard Disk (assuming it is more than 2 TB) to store the states. Furthermore, if the puzzle size increases, it becomes infeasible to store all the states. In order to solve this problem, Depth-First Iterative Deepening

(DFID) uses a combination of BFS and Depth-First Search (DFS) to find the shortest path to the goal state while storing only d number of states compared to BFS, which stores b^d states ($b = branching\ factor, d = solution\ depth$, of search tree). DFID (Korf 1985) performs DFS for a fixed depth and increases the depth iteratively until a solution is found. This requires revisiting states searched previously as the search starts from scratch for every iteration. Consequently, the algorithm increases it's computational complexity and greatly reduces space complexity. However, as the number of nodes in the search tree increases exponentially for the 15-puzzle, the overhead of searching the previous depths is negligible compared to the maximum depth of the solution.

The search can also be improved computationally if an informed search like the A* algorithm (Hart, Nilsson, and Raphael 1968) is used. A* expands the initial state and adds the newly opened states to a *frontier*. For each iteration, a state is selected from the frointer and expanded - the new states are again added to the frontier. The expansion of a node from the frontier is guided by both the cost from the initial to the current state (path cost) and the projected cost from the current state to the goal (heuristic). If the heuristic is admissible and consistent, A* is optimal (Dechter and Pearl 1983) in the number of node expansions and the number of expansions can generally be reduced further with better heuristics.

The drawback of the A* algorithm is - storing all the states of the frontier may consume a big chunk of memory for a large search space (especially, when the heuristic is misleading). Iterative Deepening A* (IDA*) solves this problem by searching upto a bounded depth and iteratively increasing the bound (Korf 1985). The search tree of IDA* can be reduced greatly with accurate heuristics. Therefore, the more accurate the heuristic, the faster IDA* can find the solution. I used a combination of Manhattan Distance and Pattern Database heuristics (Culberson and Schaeffer 1998) to prune the search tree. Manhattan Distance is simple and therefore easy to calculate and Pattern Databases are more accurate and slow to build. If the Pattern Databases are non-overlapping and cover the entire puzzle, further improvement is achievable. The two patterns I used for the 15-puzzle are as follows:

- Pattern 1: This pattern contains the values 0 to 7 and requires about 500 MB to store.
- Pattern 2: This pattern represents the remaining values and stores the position of '0' as it is necessary to capture the change in heuristic for every move. The values are therefore 0, and 8 to 15, and storing the heuristic values require about 4 GB.

Following this distribution, the memory requirement of 2.38 TB is reduced to 4.5 GB, which is much more feasible and can be stored directly in the RAM of a personal system. The computation speed can be improved further if the databases are pre-computed and stored in a file.

3 Implementation

The implementation is done using two classes - "SAS" and "stp_state". The first class implements the different search algorithms and creates the pattern databases. The second class stores the state variables of the puzzle and modifies the puzzle representations as per requirement. The main function calls and uses the class functions, and acts like a driver program. The execution flow of the 15-puzzle solver is as follows:

1. The *main* function initializes an instance of the *SAS* class which initializes some necessary variables required to build the pattern databases. Through the *SAS* class, the main functions calls for the formation of the databases.
2. There are two separate functions in the *SAS* class to build the databases. Both are similar in terms of functionality and only different in terms of the patterns used.
3. The functions create vectors of appropriate size to store the heuristic values for the pattern databases. The functions then peform a Breadth-First Search using the patterns and stores the heuristic values in the vectors. Both the functions use a "ranking" and "unrank" function to compress the state representation and store the distance calculated using BFS.
4. The "ranking" function calculates the rank value using the pattern passed as input parameter and the class variables storing the current puzzle state (and also the dual state). The rank value is returned to the PDB functions.
5. The "unrank" function changes the class state representations according to the passed *rank* value and the pattern. The rank is used as the index of the vector to store the heuristic values.
6. After the databases are formed, they are written to file. If the file is written, the values are read from file instead of using BFS to calculate the heuristic.
7. The *SAS* class then uses IDA* to search for the solution of the given problem. The IDA* driver function declares and uses an instance of the *stp_state* class to perform operations on the puzzle.
8. The heuristic used is the maximum of the Manhattan Distance, Pattern Database 1 and 2. The Manhattan Distance is calculated iteratively for each internal node of the search tree.
9. In the recursive function of IDA*, moves are generated using a direction matrix of the *stp_state* class and parent states are avoided (e.g. no immediate left move after right). The puzzle tiles and move history are maintained by the *stp_state* class as well.
10. The main function keeps calling the search functions of *SAS* and keeps taking input from the input text file until end-of-file (EOF) is reached.

4 Experimental Results

The experiments were done using 100 15-puzzle random test cases (Korf 1985). The result could not be collected when DFID was used as the test cases were taking too long to be solved. The first successful solution of the 100 cases is calculated using Manhattan Distance. The total time taken is $898.9s$. The longest time taken was $162s$ and for test case 88.

The results improve significantly when pattern databases are used. The longest time taken is $16.35s$ for test case 17. However, forming the databases require $284.43s$ and $2705.85s$, respectively. The maximum depth for the database searches are 62 and 68, respectively. When the heuristic values are read from file, the total time of the program is reduced to $52.658s$. The experimental results are saved in the following text files:

- *Initial.txt*: The results using Manhattan Distance as heuristic.
- *PDB.txt*: The results using both Manhattan Distance and pattern databases as heuristic. The databases are built from scratch.
- *PDB_modified.txt*: The results using Manhattan Distance and pattern databases as heuristic, where the heuristic is read from the file.

5 Conclusion and Challenges

Implementing the state representations and the *stp_state* class was fairly simple and without much issues. The search implementations were a bit more difficult and debugging the big searches was troublesome. I created small test cases and ran the search to shallow depths to compare with my manual search models. The sample solution provided by Prof. Nathan was very helpful in this regard.

The pattern databases were the most difficult to implement and the most time consuming. However, the difficulty was expected and so moving slowly, step-by-step, debugging each small step was how I managed to complete the program eventually. The performance can be improved further if additive pattern database (Felner, Korf, and Hanan 2004) heuristic is used.

References

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

Dechter, R.; and Pearl, J. 1983. The Optimality of A* Revisited. In *AAAI*, volume 83, 95–99.

Felner, A.; Korf, R. E.; and Hanan, S. 2004. Additive pattern database heuristics. *Journal of Artificial Intelligence Research* 22: 279–318.

Hart, P. E.; 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(2): 100–107.

Johnson, W. W.; Story, W. E.; et al. 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.