Kirill Melnikov
301312400
March 28, 2020

15 Puzzle Assignment 5

The 15 puzzle was an interesting problem to solve. The attempt was to create a set of heuristics, based on the Manhattan and the pattern databases, to optimally solve random instances of the 15 puzzle, while also maintaining decent coding practices, despite the need for efficiency.

Some background on how the program works: There are several files within it, Node.h, Node.cpp, PatternDatabase.h, PatternDatabase.cpp, PuzzleSearch.h, PuzzleSearch.cpp, Main.cpp. All code is build in C++.

For the user, the program works simplify enough; To run it, type “make” on a Linux system, then “./Main” to start the executable. The program will automatically load the DB, and then output a random solution of the 15 puzzle using IDA star search. The solution is a string of moves that you can follow through, if you wish, to try to get to the solution from the initial state.

The goal of this program was efficiency, however, I did not want to sacrifice this in favour of poor code. As such, there are the three classes FifteenPuzzleSearch, PatternDatabases, and Node to encourage modularity. Node is our instance of the 15 puzzle. It has some basic helper functions to check the validity of a move, and make moves. The pattern database class uses the BuildPatterns() method to build our database of moves. It uses a backwards breadth-first search from the goal state, to find all the possible positions that can be discerned from the goal, and then stored into a set. Cost of one move in the 15 puzzle was always considered to be one.

Finally, we have the FifteenPuzzleSearch. Here, we initialize our puzzle. Instead of using inversions to check the validity of our state, a predetermined number of random moves were made from the goal state, to generate our new board. The IDA algorithm is implemented within IDA_root() and IDA(). IDA_root runs in an infinite loop until IDA returns a valid solution, and also is used for initialization.

The algorithm in question is recursive, and uses a threshold value to evaluate when to cut-off a branch. The threshold value is initially calculated as part of the f-value of a Node, which is then passed into the recursive IDA. If it so happens that the current node’s f-value exceeds the threshold, then the current branch would be cut off, and removed. The threshold acts as the “depth,” similar in concept to iterative deepening search. To ensure efficiency, most of the objects were passed in by reference, particularly the pattern database, which was used to calculate the heuristic value of a node. To further optimize code, a set was used for the explored list, providing us an O(logn) time complexity. Hash-tables were considered, but to be able to ensure a perfect hash is difficult, which is why ultimately it was eliminated from consideration.

For the pattern database, the design was implemented using an additive pattern. This means that you have a set of tiles that you solve a relaxed problem of the 15 puzzle, and then sum up the result to get the final cost for our heuristic function. For example, you could have the sets {{1, 2, 3}, {4,5,6}, ... , {13, 14, 15}} which would mean that for each state, you would have 5 sub-problems to solve, adding them all up to get the final cost. In contrast is the non-additive database, which takes a subset of the 15 puzzle problem. For example, you could take the first column, first row, and the blank tile of the 15 puzzle, and then attempt to solve this sub problem, of which the total cost of this would be your heuristic function.

The issues with the non-additive pattern is that, generally, it is not as effective as the additive database, and it also takes up a lot of memory. For example, if you were attempting to store what was previously described, which is in total 8 pieces, you need around 500 million positions that you would be required to store. This would take too much time to compute, and also too much memory. In

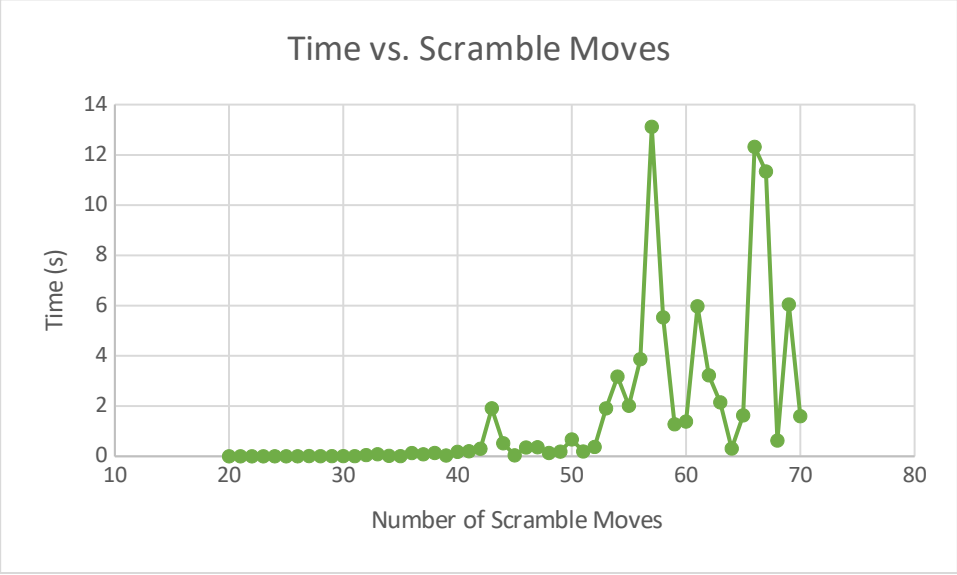
Kirill Melnikov
301312400
March 28, 2020

contrast, with the additive pattern, if we were to consider the 3-piece relaxed problems, the total amount of states adds up to around 16800, which is much less expensive compared to its counterpart.

One important aspect to consider when implementing the pattern database is how many tiles we want to use to solve our “relaxed problem.” If you were to use an even number of tiles, you would end up with a problem subset that was smaller from the rest. For example, if you approached the 4-tile relaxed problem, you would have the sets one to four, four to eight, eight to twelve, but then you would be left with a disjoint, 3-tile set, compared to the rest. As such, to simplify implementation, I only considered the odd pattern databases. From what was observed, the 3-tile relaxed problem is relatively fast to compute, and provides better results compared to the Manhattan heuristic.

Below is a picture depicting the behaviour of the 3-tile pattern database, of Time vs. Scramble moves in Figure 1, and Nodes Expanded vs. Scramble move in Figure 2.

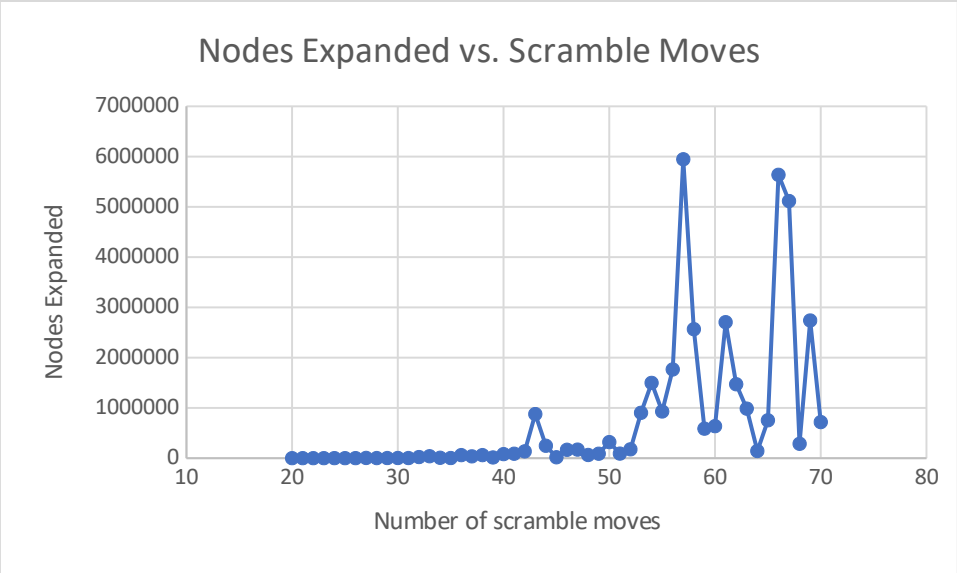
Figure 1



Time vs. Scramble moves shows the duration it took to find an optimal solutions using IDA search with pattern databases. Note that the number of scramble moves does not directly correspond to a solution of the same size. The maximum length of a solution in the 15-puzzle was shown to be 80 moves at most. As such, even if the initialization was scrambled past this amount, the solution set would not change. This property applies to solutions of length less than eighty also.

Kirill Melnikov
301312400
March 28, 2020

Figure 2



These graphs were produced by solving 5 random instance of the 15 puzzle for each value of the x-axis, and then averaged to get the corresponding data. As shown here, the relationship between nodes expanded and time taken is closely tied together. They work in conjunction, and have almost a one to one ratio. It should be noted that the more scramble moves you make, does not necessarily mean that the solution is more complicated. As such, it makes it difficult to estimate the complexity of the problem, unless you have per-calculated the states from the beginning. However, these graphs, when averaged, provide a good estimate in to the performance of the IDA algorithm, when paired with pattern databases.