Jordan Rowley
Computer Science
University of St. Andrews

CS3105 Practical 1 Report

Search in Sliding Blocks Puzzles

For part one of the practical, I have implemented a 'Best-First Search' for both the manhattan distance heuristic and the A* total distance heuristic.

The Manhattan class contains some utility functions used by all three search implementations. These could've been housed in their own class but it didn't seem necessary. These functions calculate: the goal coordinates of a square given the number of columns and the square's number; the coordinates of a square given the puzzle's board and the square's number; the manhattan distance of a puzzle board given the board; the squares adjacent to a given square given the board, the coordinates of the square, and the position of the square in relation to the edges of the board; the possible moves that each square can make given the board; and finally the new puzzle board after a piece has been moved given a state. Each of these functions are used across all three of my implementations.

I also made a state class with some attributes to track the puzzles in the search tree. The attributes include: the puzzle board as an integer array, the move history as a linked list of integers, the manhattan distance of the board as an integer, and lastly a boolean flag to tell the compareTo function which heuristic to use in the PriorityQueue. Additionally, there is a didntWorkException class which is the custom exception I used throughout my classes to throw exceptions with a message explaining where the error occurred. I primarily used this exception alongside printing variables to test and debug my code. I also used paper and a pen to check solutions and come up with my own puzzles to test the implementations with.

My implementation of the Best-First search using the manhattan heuristic takes the input file from the command line and converts it into an array of integers. It then calls solve() with the parameters: initial state, a new priority queue, and false for the astar flag. The solve function adds the initial state to the priority queue before opening a while(true) loop. Within the loop is a counter to break the loop once a certain number of iterations have run. The

algorithm loops through taking the first state off of the queue and checking if any of its child nodes are a solution. If it is not a solution, the new state will be added to the queue, if it is a solution, the history of that solution state is returned and written to an output.txt file. This works because the priority queue orders the states by manhattan distance so a state removed from the queue will be the state with the lowest manhattan distance.

The only difference with the astar implementation is that the flag for the solve function will be true. This switches the compareTo function for the priority queue to take the 'Total Distance' as a heuristic instead of just a Manhattan distance. This will order the priority queue by Manhattan distance + total moves so far.

Comparing the runtimes and solution length of the Manhattan implementation vs astar implementation:

Examples from project spec:	Manhattan runtime (ns)	Manhattan output	Astar runtime (ns)	Astar output
Example 1	113099	1	304813	1
Example 2	296233	1 3	514603	1 3
Example 3	4191974	1 3 6 8 7 5 4 7 8	5649527	1 3 6 8 7 5 4 7 8
Example 4	N/A	N/A	268930121	1 6 7 3 5 1 3 7 8 5 7 2 4 3 2 4 6 8 5 7 4 5 8
Example 5	N/A	N/A	2739820252	1 1 2 5 4 7 1 2 5 3 8 6 7 4 6 7 4 1 2 5 3 6 5 3 6 8 7 4 1 2 3 6
Example 6	N/A	N/A	1780407298	0

As seen above, for the easier puzzles Manhattan is quickest but for harder puzzles Astar completed the search within a time I could wait and Manhattan didn't. I also didn't have time to wait for example 6 to exhaustively search.

The examples in which the manhattan implementation was faster were very quick and the time saved was negligible. Where astar could solve larger problems in a matter of seconds, manhattan was left running for minutes on end. In example 5, astar returned a depth 32 solution quite quickly but this may not be the optimal solution. The worst case for both the astar and manhattan implementations is when the number of iterations needed to solve the puzzle exceeds the hard-coded counter bound. This is because you will just have to wait for that bound to be reached and -1 to be returned.

The iterative deepening with branch and bound implementation uses the utility functions from the Manhattan class but has its own two versions of the solve function. These are solveID and solveBNBDP. The latter implements a branch and bound depth first depth bounded algorithm to search for a solution. It uses two linked lists: one for the search frontier and one to store visited states. This solve function will add the initial state to the search tree before opening a loop which will run as long as the search tree is not empty. Within the loop, the current state is set as the element removed from the front of the search list. Each child node is then checked to see if it is a solution or not, if it is then that state's history is returned. If it is not a solution then the depth bound is checked and, if sufficed, the new state is added to the list of visited nodes and added to the start of the search list. The solveID function calls the solveBNBDF with gradually increasing depth bounds, starting at the manhattan distance of the initial state and incrementing by 2 between each function call. If and when the solution is found, it is written to an output.txt file.

Comparing the runtimes and solution length of the iterative deepening branch and bound implementation vs astar implementation:

Examples from project spec:	IDBNB runtime (ns)	IDBNB output	Astar runtime (ns)	Astar output
Example 1	47856	1	304813	1
Example 2	725598	1 3	514603	1 3
Example 3	12840573	1 3 6 8 7 5 4 7 8	5649527	1 3 6 8 7 5 4 7 8
Example 4	~ 20 minutes	1 6 7 3 5 1 3 7 8 5 7 2 4 3 2 4 6 8 5 7 4 5 8	268930121	1 6 7 3 5 1 3 7 8 5 7 2 4 3 2 4 6 8 5 7 4 5 8
Example 5	N/A	N/A	2739820252	1 1 2 5 4 7 1 2 5 3 8 6 7 4 6 7 4 1 2 5 3 6 5 3 6 8 7 4 1 2 3 6
Example 6	92416948354	0	1780407298	0

As seen above, the IDBNB implementation always gave the optimal solution but at a huge time cost. For example 5 I couldn't wait long enough for it to exhaustively search.

Astar is much more reliable in terms of reaching a solution within a given time but the solution may not be optimal. IDBNB proved to have some extremely long runtimes for more difficult puzzles but always promised an optimal solution if there exists one.