CS3105 Practical 1: Search in Sliding Block Puzzles

Analysis

For this practical I was asked to implement and compare different search algorithms on sliding block puzzles. The different search algorithms were: Best-First Search using the Manhattan distance as a heuristic, Best-First as well but using Total Distance as the heuristic, and then the third and final search algorithm is named price and it puts weight on the actual number value of the piece being moved in the form of a monetary cost.

Design

For keeping track of the current state, I created a Board and Node object. The Board object stores information such as the cells of the board, the number of columns and rows for example. Additional values such as the Manhattan distance of a board is also stored in the object. This is to reduce as many repeat calculations as possible as they can be costly.

The Node object is used to keep a track of the games progress. I have a parent attribute that will give the previous node, so in practise given a node of a board that has been solved you can see all the moves leading up to the board being in a solved state.

Another reason why I decided to have a node class is because of my use of a priority queue to store the different nodes. I use a Comparator that uses the function used to calculate the heuristic to order the priority queue. I chose to use a priority queue because it orders the nodes in the correct order and could be easily changed to a different heuristic without difficulty. Due to the large amount of memory that will be used I wanted to take advantage of an already optimised and efficient data structure rather than using a differing data structure where I would have to compare the heuristics manually.

To optimise the search algorithm for all 3 of the searches, I created a set that would store all the nodes that have already been visited so if they are visited again then the whole process doesn't have to go again so there are no repetitions, this will limit the amount of times the Manhattan algorithm would end up in an infinite loop as it eliminates going back to repeated nodes.

Evaluation

The Manhattan distance and total distance are both heuristics that can be used to determine the next 'best' relative to the heuristic move. The total distance always finds the most optimal solution as it uses Manhattan distance and the amount of moves that have currently been moved, this produces A*.

I managed to find cases where the A* heuristic found a solution and the Manhattan distance times out. However, this is what is to be expected because due to the nature of the Manhattan distance heuristic it can go into an infinite loop which will therefore result in the

solution timing out. This doesn't happen to the A* algorithm as it takes into account the amount of moves that have already been made.

For example:

One case I found is where the board is set up as '3 3 0 2 7 1 8 6 5 4 3'. The A* algorithm finds the optimal solution of '1 20 1 5 4 8 6 7 2 6 7 3 8 7 6 2 3 6 5 4 7 8' whereas the Manhattan distance algorithm timed out and couldn't find a solution in time.

Another example is where the board is as follows '3 3 3 8 2 6 0 5 4 7 1'. The A* algorithm finds a solution '1 24 6 4 7 6 8 2 5 1 6 8 2 3 4 2 1 5 3 1 2 4 1 2 5 6' whereas similarly to the previous example the Manhattan distance algorithm timed out.

This is because of the depth that is needed to be reached, the A* can handle the solutions that require a larger depth and iterations. For example, with the second example according to an 8-puzzle solver I found on the internet [1] the most optimal solution is 24 moves as shown above but the iterations are at 2168.

My Manhattan distance solution can solve a lot of 3x3 solutions as long the number of moves needed to be made isn't above around 10, after this point the Manhattan distance enters a loop and cannot find the correct solution before running out of time or memory. This is partly due to my algorithm using expensive operations such as many for loops that when a solution has a lot of nodes being explored can really slow down the efficiency of the algorithm finding a solution. To attempt to make it more efficient for getting all the possible moves I use the position of the empty cell to keep track of what moves can be made, rather than looking for the empty cell each time. This improved the efficiency drastically however there are still other operations that are costly. Such as having to go back up the node path using 'parent', I could have added the output to a variable and added to it each time which would have been less costly. As can be seen in the stacschecks the Manhattan algorithm can solve simple 4x4 boards but as soon as it moves past 10 moves, the algorithm cannot find a solution.

My A* is able to solve practically all 3x3 boards that are solvable. For example, one of the highest depths and iterations for a 3x3 I found was from this board '3 3 5 8 7 6 2 4 0 1 3' which has an optimal solution of 28 moves and 4465 iterations [1].

Results from testing my algorithms on randomly generated boards (these were all on 3x3 boards) I counted the occurrences of each different output for random boards:

Manhattan:

	Totals	Percentage % (x / (Solved + Couldn't find solution)
Unsolvable (0)	439	N/A
Solved (1)	16	26.23
Couldn't find soluition (-1)	45	73.77

A*:

	Totals	Percentage % (x / (Solved + Couldn't find solution)
Unsolvable (0)	454	N/A
Solved (1)	12	26.09
Couldn't find soluition (-1)	34	73.91

Price:

	Totals	Percentage % (x / (Solved + Couldn't find solution)
Unsolvable (0)	443	N/A
Solved (1)	14	24.56
Couldn't find soluition (-1)	43	75.44

From these results above it can be seen that price is the least likely to find a solution for a given board, followed by A* and then Manhattan being the most likely. However, there are a lot of factors coming into play with these results, i.e. when running the A* algorithm there were more unsolvable boards so it had less chance to get a board that was solvable. However, I believe apart from the A* these results do still follow the same trend as what I have proved previously.

These results were carried out with only giving 10 seconds for the algorithm to carry out. This is not an ideal circumstance as with more time it is likely the algorithms would be able to solve the solution. This is at the detriment to A* in this case, and I don't believe the results above show the whole story about this algorithm.

For implementing the cost to the feature to the algorithm what I opted for was that the heuristic would be the Manhattan distance multiplied by the value added to the cost so far of what was being moved, i.e. if the piece '8' had a Manhattan distance of 3, this would be 24, added to the cost so far with my heuristic and any the total cost of moving any additional pieces. I decided to choose to implement it in this fashion because I believe it was the most optimal way to give an admissible heuristic. This heuristic is admissible as it keeps track of the cost so far, similar to the way the A* algorithm works.

Testing

To test the solutions, I have provided I have added my own tests additionally to the stacschecks provided. To test the basic functionality the I tested the Manhattan program with 3x3, 3x4, 4x3 and 4x4 boards ranging from boards with 0 moves to be made to around 10. This proved useful as I was able to guarantee that my Manhattan distance best first search was making the correct moves each time for different sized boards and boards with different amounts of moves to be made. I generated the boards all myself and came up with the solutions manually, so that I could guarantee that the solution was the correct answer.

Stacscheck results:

```
Testing CS3105 P1
- Looking for submission in a directory called 'P1': Already in it!
* BUILD TEST - build-all : pass
* COMPARISON TEST - AStar/3x3/prog-run-3x3-m20.out : pass
* COMPARISON TEST - AStar/3x3/prog-run-3x3-m24.out : pass
* COMPARISON TEST - AStar/4x4/prog-run-4x4-m0.out : pass
* COMPARISON TEST - AStar/4x4/prog-run-4x4-m1.out : pass
* COMPARISON TEST - AStar/4x4/prog-run-4x4-m10.out : pass
* COMPARISON TEST - AStar/4x4/prog-run-4x4-m11.out : pass
* COMPARISON TEST - AStar/4x4/prog-run-4x4-m12.out : pass
* COMPARISON TEST - AStar/4x4/prog-run-4x4-m13.out : pass
* COMPARISON TEST - AStar/4x4/prog-run-4x4-m2.out : pass
* COMPARISON TEST - AStar/4x4/prog-run-4x4-m3.out : pass
* COMPARISON TEST - AStar/4x4/prog-run-4x4-m4.out : pass
* COMPARISON TEST - AStar/4x4/prog-run-4x4-m5.out : pass
* COMPARISON TEST - AStar/4x4/prog-run-4x4-m6.out : pass
* COMPARISON TEST - AStar/4x4/prog-run-4x4-m7.out : pass
* COMPARISON TEST - AStar/4x4/prog-run-4x4-m8.out : pass
* COMPARISON TEST - AStar/4x4/prog-run-4x4-m9.out : pass
* COMPARISON TEST - AStar/OptimalAny/prog-run-NoMoves.out : pass
* COMPARISON TEST - AStar/OptimalAny/prog-run-RunningExample.out : pass
* COMPARISON TEST - AStar/OptimalUnique/prog-run-NoMoves.out : pass
* COMPARISON TEST - AStar/OptimalUnique/prog-run-RunningExample.out : pass
* COMPARISON TEST - AStar/Solvable/prog-run-NoMoves.out : pass
* COMPARISON TEST - AStar/Solvable/prog-run-RunningExample.out : pass
* COMPARISON TEST - AStar/Unsolvable/prog-run-Impossible2.out : pass
* COMPARISON TEST - AStar/Unsolvable/prog-run-Impossible3.out : pass
* COMPARISON TEST - AStar/UnsolvableOrTimeout/prog-run-Impossible2.out : pass
* COMPARISON TEST - AStar/UnsolvableOrTimeout/prog-run-Impossible3.out : pass
* COMPARISON TEST - Manhattan/3x3/prog-run-3x3-m0.out : pass
* COMPARISON TEST - Manhattan/3x3/prog-run-3x3-m1.out : pass
* COMPARISON TEST - Manhattan/3x3/prog-run-3x3-m2.out : pass
* COMPARISON TEST - Manhattan/3x3/prog-run-3x3-m3.out : pass
* COMPARISON TEST - Manhattan/3x3/prog-run-3x3-m4.out : pass
* COMPARISON TEST - Manhattan/3x3/prog-run-3x3-m5.out : pass
* COMPARISON TEST - Manhattan/3x3/prog-run-3x3-m6.out : pass
* COMPARISON TEST - Manhattan/3x3/prog-run-3x3-m7.out : pass
* COMPARISON TEST - Manhattan/3x3/prog-run-3x3-m8.out : pass
* COMPARISON TEST - Manhattan/3x3/prog-run-3x3-m9.out : pass
* COMPARISON TEST - Manhattan/3x4/prog-run-3x4-m0.out : pass
* COMPARISON TEST - Manhattan/3x4/prog-run-3x4-m1.out : pass
* COMPARISON TEST - Manhattan/3x4/prog-run-3x4-m10.out : pass
* COMPARISON TEST - Manhattan/3x4/prog-run-3x4-m2.out : pass
* COMPARISON TEST - Manhattan/3x4/prog-run-3x4-m3.out : pass
* COMPARISON TEST - Manhattan/3x4/prog-run-3x4-m4.out : pass
* COMPARISON TEST - Manhattan/3x4/prog-run-3x4-m5.out : pass
```

```
* COMPARISON TEST - Manhattan/4x3/prog-run-4x3-m0.out : pass
* COMPARISON TEST - Manhattan/4x3/prog-run-4x3-m1.out : pass
* COMPARISON TEST - Manhattan/4x3/prog-run-4x3-m2.out : pass
* COMPARISON TEST - Manhattan/4x3/prog-run-4x3-m3.out : pass
* COMPARISON TEST - Manhattan/4x3/prog-run-4x3-m4.out : pass
* COMPARISON TEST - Manhattan/4x3/prog-run-4x3-m5.out : pass
* COMPARISON TEST - Manhattan/4x3/prog-run-4x3-m6.out : pass
* COMPARISON TEST - Manhattan/4x3/prog-run-4x3-m7.out : pass
* COMPARISON TEST - Manhattan/4x3/prog-run-4x3-m8.out : pass
* COMPARISON TEST - Manhattan/4x4/prog-run-4x4-m0.out : pass
* COMPARISON TEST - Manhattan/4x4/prog-run-4x4-m1.out : pass
* COMPARISON TEST - Manhattan/4x4/prog-run-4x4-m10.out : pass
* COMPARISON TEST - Manhattan/4x4/prog-run-4x4-m11.out : pass
* COMPARISON TEST - Manhattan/4x4/prog-run-4x4-m12.out : pass
* COMPARISON TEST - Manhattan/4x4/prog-run-4x4-m13.out : pass
* COMPARISON TEST - Manhattan/4x4/prog-run-4x4-m2.out : pass
* COMPARISON TEST - Manhattan/4x4/prog-run-4x4-m3.out : pass
* COMPARISON TEST - Manhattan/4x4/prog-run-4x4-m4.out : pass
* COMPARISON TEST - Manhattan/4x4/prog-run-4x4-m5.out : pass
* COMPARISON TEST - Manhattan/4x4/prog-run-4x4-m6.out : pass
* COMPARISON TEST - Manhattan/4x4/prog-run-4x4-m7.out : pass
* COMPARISON TEST - Manhattan/4x4/prog-run-4x4-m8.out : pass
* COMPARISON TEST - Manhattan/4x4/prog-run-4x4-m9.out : pass
* COMPARISON TEST - Manhattan/OptimalAny/prog-run-NoMoves.out : pass
* COMPARISON TEST - Manhattan/OptimalAny/prog-run-RunningExample.out : pass
* COMPARISON TEST - Manhattan/OptimalUnique/prog-run-NoMoves.out : pass
* COMPARISON TEST - Manhattan/OptimalUnique/prog-run-RunningExample.out : pass
* COMPARISON TEST - Manhattan/Solvable/prog-run-NoMoves.out : fail
--- expected output ---
1 0
--- submission output ---
1
---

* COMPARISON TEST - Manhattan/Solvable/prog-run-RunningExample.out : pass
* COMPARISON TEST - Manhattan/Unsolvable/prog-run-Impossible2.out : pass
* COMPARISON TEST - Manhattan/Unsolvable/prog-run-Impossible3.out : pass
* COMPARISON TEST - Manhattan/UnsolvableOrTimeout/prog-run-Impossible2.out : pass
* COMPARISON TEST - Manhattan/UnsolvableOrTimeout/prog-run-Impossible3.out : pass
* COMPARISON TEST - Price/3x3/prog-run-3x3-m0.out : pass
* COMPARISON TEST - Price/3x3/prog-run-3x3-m1.out : pass
* COMPARISON TEST - Price/3x3/prog-run-3x3-m2.out : pass
* COMPARISON TEST - Price/3x3/prog-run-3x3-m3.out : pass
* COMPARISON TEST - Price/3x3/prog-run-3x3-m4.out : pass
* COMPARISON TEST - Price/3x3/prog-run-3x3-m5.out : pass
* COMPARISON TEST - Price/3x3/prog-run-3x3-m6.out : pass
* COMPARISON TEST - Price/3x3/prog-run-3x3-m7.out : pass
* COMPARISON TEST - Price/3x3/prog-run-3x3-m8.out : pass
* COMPARISON TEST - Price/3x3/prog-run-3x3-m9.out : pass
* COMPARISON TEST - Price/3x4/prog-run-3x4-m0.out : pass
* COMPARISON TEST - Price/3x4/prog-run-3x4-m1.out : pass
* COMPARISON TEST - Price/3x4/prog-run-3x4-m10.out : pass
* COMPARISON TEST - Price/3x4/prog-run-3x4-m2.out : pass
* COMPARISON TEST - Price/3x4/prog-run-3x4-m3.out : pass
* COMPARISON TEST - Price/3x4/prog-run-3x4-m4.out : pass
* COMPARISON TEST - Price/3x4/prog-run-3x4-m5.out : pass
```

```
* COMPARISON TEST - Price/3x4/prog-run-3x4-m6.out : pass
* COMPARISON TEST - Price/3x4/prog-run-3x4-m7.out : pass
* COMPARISON TEST - Price/3x4/prog-run-3x4-m8.out : pass
* COMPARISON TEST - Price/3x4/prog-run-3x4-m9.out : pass
* COMPARISON TEST - Price/4x3/prog-run-4x3-m0.out : pass
* COMPARISON TEST - Price/4x3/prog-run-4x3-m1.out : pass
* COMPARISON TEST - Price/4x3/prog-run-4x3-m2.out : pass
* COMPARISON TEST - Price/4x3/prog-run-4x3-m3.out : pass
* COMPARISON TEST - Price/4x3/prog-run-4x3-m4.out : pass
* COMPARISON TEST - Price/4x3/prog-run-4x3-m5.out : pass
* COMPARISON TEST - Price/4x3/prog-run-4x3-m6.out : pass
* COMPARISON TEST - Price/4x3/prog-run-4x3-m7.out : pass
* COMPARISON TEST - Price/4x4/prog-run-4x4-m0.out : pass
* COMPARISON TEST - Price/4x4/prog-run-4x4-m1.out : pass
* COMPARISON TEST - Price/4x4/prog-run-4x4-m10.out : fail
--- expected output ---
1 10 8 4 3 2 6 5 9 13 14 15 79
--- submission output ---
-1
---

* COMPARISON TEST - Price/4x4/prog-run-4x4-m11.out : fail
--- expected output ---
1 11 7 8 4 3 2 6 5 9 13 14 15 86
--- submission output ---
-1
---

* COMPARISON TEST - Price/4x4/prog-run-4x4-m12.out : fail
--- expected output ---
1 12 11 7 8 4 3 2 6 5 9 13 14 15 97
--- submission output ---
-1
---

* COMPARISON TEST - Price/4x4/prog-run-4x4-m13.out : fail
--- expected output ---
1 13 10 11 7 8 4 3 2 6 5 9 13 14 15 107
--- submission output ---
-1
---

* COMPARISON TEST - Price/4x4/prog-run-4x4-m2.out : pass
* COMPARISON TEST - Price/4x4/prog-run-4x4-m3.out : pass
* COMPARISON TEST - Price/4x4/prog-run-4x4-m4.out : pass
* COMPARISON TEST - Price/4x4/prog-run-4x4-m5.out : pass
* COMPARISON TEST - Price/4x4/prog-run-4x4-m6.out : fail
--- expected output ---
1 6 6 5 9 13 14 15
--- submission output ---
-1
---

* COMPARISON TEST - Price/4x4/prog-run-4x4-m7.out : fail
--- expected output ---
1 7 2 6 5 9 13 14 15
--- submission output ---
-1
---
```

```
* COMPARISON TEST - Price/4x4/prog-run-4x4-m8.out : fail
--- expected output ---
1 8 3 2 6 5 9 13 14 15
--- submission output ---
-1
---

* COMPARISON TEST - Price/4x4/prog-run-4x4-m9.out : fail
--- expected output ---
1 9 4 3 2 6 5 9 13 14 15
--- submission output ---
-1
---

115 out of 124 tests passed
```

As can be seen from the additional stacschecks tests I created that a majority of them pass and produce the expected output. This confirms that my solution works as expected and that the search algorithms go through the process of making the moves in the correct manner.

A majority of the tests that are failing are from the price search algorithm timing out. This is due to it being a 4x4 matrix and the depth of the search is very high and doesn't have enough time to compute the result.

Also of note, I could not understand why this occurred: 'Manhattan/Solvable/prog-run-NoMoves.out : fail' as when I run the test isolated from stacscheck with the same input I get this outcome:

```
./manhattan
3 4 1 2 3 4 5 6 7 8 9 10 11 0
1 0
```

Which is the correct output but for the stacscheck my solution only outputs '1'.

Conclusion

In conclusion I enjoyed this practical as it gave me a deep insight and greater knowledge of the different types of search algorithm and how they are affected by the heuristic that is chosen.

Given more time I would have liked to be able to create better datasets which more accurately show how each of the algorithms performed. Another aspect would have been to improve the optimisation so that my algorithms would have been able to achieve complex 4x4 solutions, however this wasn't my main focus as I didn't want to optimise the solutions too far so the comparisons between them were highlighted further.

References

[1] – https://deniz.co/8-puzzle-solver/