ECM2423 Artificial Intelligence and Applications

Coursework Report Part 1

Question 1 | *A** and the 8-puzzle game

Question 1.1: 8-puzzle problem as a search problem

In 8-puzzle game, we have 9 slots with 8 tiles (numbered 1 to 8) and 1 empty slot. To describe it as a search problem, I will address 4 points; the states, operators, goal test and the cost of a path:

- **States:** locations of the 8 tiles (these can be stored in a 1D or 2D python array)
- **Operators:** moving in 4 directions (up, down, left, right; where possible) with the blank slot
- **Goal test:** goal can be any given tile configuration on board (default: 0, 1, 2, 3, 4, 5, 6, 7, 8)
- **Path cost:** 1 cost per move

Based on states and operators, we can generate a set of neighbouring states. Essentially, any 8-puzzle state that can be moved from the current state using any of the operators. Now with neighbours we can generate a solution path, and any such path connecting the start state with goal state can be considered a solution. With these in mind, it should be posssible to write a simple uninformed (Depth-first or Breadth-first) search algorithms.

Question 1.2.1: Briefly outline of the A* algorithm

In general, search algorithms can be divided as uninformed (blind) and informed (heuristic) search algorithms. Unlike uninformed search, in heuristic search we use a problem specific knowledge (typically a heuristic function) to decide on which path we continue with (that is; which node we would expand).

A* is a heuristic (informed) search algorithm, where the heuristic function is:

$$f(n) = g(n) + h(n)$$

Thas is it includes both cost of reaching the node n, and the estimated cost of reaching the goal from n.

A* is complete, exponential in solution length, optimal (that is the obtained solution is always the best) given the heuristic function is admissible and it has to keep all the nodes in memory.

Question 1.2.2: Describe two admissible heuristic functions for the 8-puzzle problem

The two admissible heuristics I will be using are **Manhattan distances** and the **Gaschnig's heuristic**.

Manhattan distances heuristic function is based on calculating the sum of Manhattan distances of each tile from their position in the goal state. In this case, the Manhattan distance refers to shortest number of block moves (horizontal or vertical) between two positions (on an 8-puzzle board).

This heuristic function is derived from the relaxed version of the 8-puzzle game where tiles can be moved to any neighbouring square and hence, it is an admissible heuristic (always optimistic).

Gaschnig's heuristic function is the sum of number of tiles out of row and number of tiles out of column. Where being out of a row/column suggets the tile is not currently in the same row or column that they are in the goal state.

Gaschnig's heuristic is admissible, as every tile out of a row or column would have to be moved at least once (or in case of both out of row and column, twice) to reach the goal state. Meaning, the estimate either equals the actual cost or is optimistic:

$$h(n) \leq h^*(n)$$

Where h(n) is Gaschnig's heuristic function, $h^*(n)$ is the actual cost.

I chose these particular heuristic functions, as they have simple calculations, thus; they have faster calculation time leading to more efficient searches. Futhermore, simple calculations make the heuristics clear and concise, this would help understand them better when analysing the performance. They also use different approaches which should help providing a meaningful comparison.

Question 1.2.3: Implementation of the A* in 8-puzzle problem

The implementation can be found in ***puzzle*** folder consisting of 3 python scripts (***puzzle_solver, puzzle_heuristics, puzzle_base***). To start the program, run the ***puzzle_solver.py***. The default start and goal configurations provided in the specification were used.

Question 1.2.4: Performance of A* heursitics, Manhattan and Gaschnig's

The following table 1 demonstrates the results of running the algorithm on default configurations

Heuristic	Time (Seconds)	Space (Iterations)	Complete?	Optimal?
Manhattan	3.8733	4366	Yes	Yes
Gaschnig's	31.532	14792	Yes	Yes

Table 1 - Manhattan and Gaschnig's heuristics on defulat configurations

Here, *time* refers to the completin time of the search and *space* is represented by the number of (outer loop) iterations; which is equivalent to number of nodes (configurations/states) expanded by the search algorithm.

We can see that, both heuristic functions provided a complete and optimal solution. Althought both also had a reasonable completion time, Manhattan seems to be noticeably faster. Naturally, we can also see that the Manhattan method had less iterations (expanded fewer nodes).

Performance of these two heuristics can be better presented, if we run searches on puzzles of varying difficulty (where difficulty is number of steps to solve the puzzle). I included puzzles from 10 moves (up to the maximum 31 move puzzle) for sensible comparison, as below that level search time is almost instant. Also note that search time and search space axis are in logarithmic scale, in order to better display the widely ranging data.

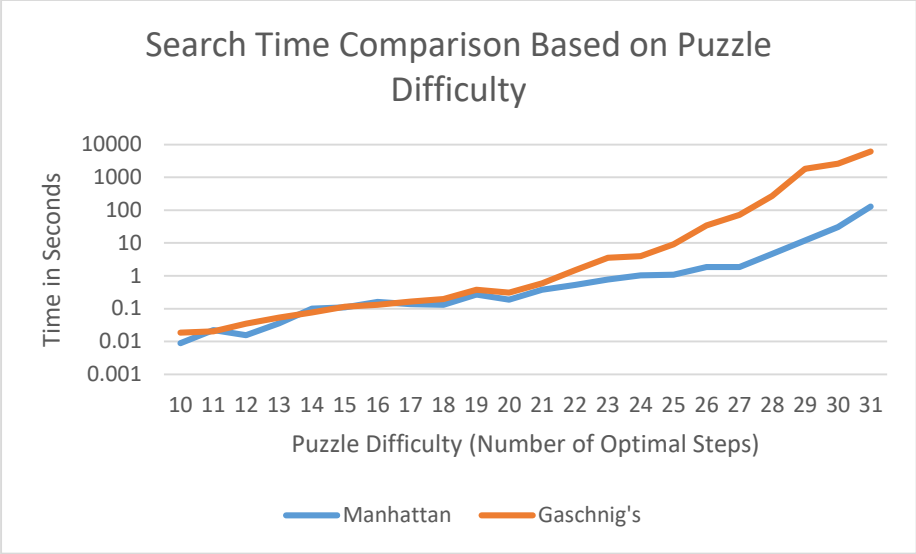

Figure 1 - Manhattan and Gaschnig's Search Time over Puzzle Difficulty

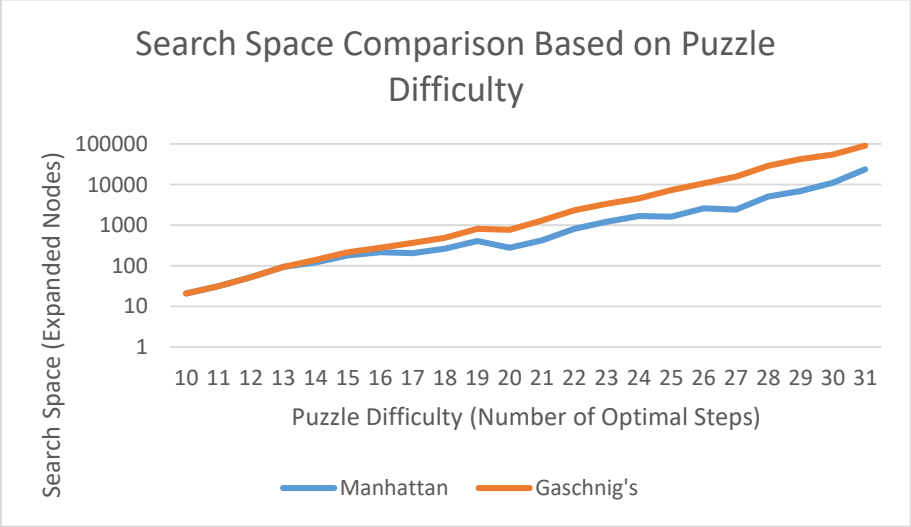

Figure 2 - Manhattan and Gaschnig's Search Space over Puzzle Difficulty

From both Figure 1 and Figure 2 we can see that at lower levels, there is little to no different between the search time and space. However, on more difficult puzzles (similar to the default example) the Manhattan heuristic has an upper edge. Difference in performance (between the heuristics) increases with puzzle difficulty.

At the maximum puzzle difficulty (31 steps), the Manhattan heuristic function completes almost 50 times faster (128 seconds to 6100 seconds). At this difficulty, Manhattan heuristic also has a smaller search, althought the inequality is smaller (23912 iterations to 90503 iterations). Overall, the Manhattan Distances methodology seems to be significantly “smarter” than the Gaschnig’s heuristic.

Finally, note that the performance of the A* search algorithm (completion time) depends on various factors; processing power, operating system (allocated processing power). Hence, the exact search time is for comparison purposes.

The raw tabular representation of the graphs (Figure 1 and Figure 2) can be found at ex1_performance.

Question 1.3: General solution of the 8-puzzle using A*

The ***puzzle*** python folder has been updated to provide a general version of the puzzle solver. Similarly, the ***puzzle_solver.py*** needs to be run to start the program. User will be requested to either enter their own configurations or continue with default start and goal states (as provided in the specification). The user can also determine which heuristic (Manhattan or Gaschnig's) they want the A* algorithm to run with.

basic input validation to ensure user input is in correct format. The configuration input should be an array of size 9 containing tile values (where empty tile is denoted as 0). Below are examples of how user input would be translated into an 8-puzzle state:

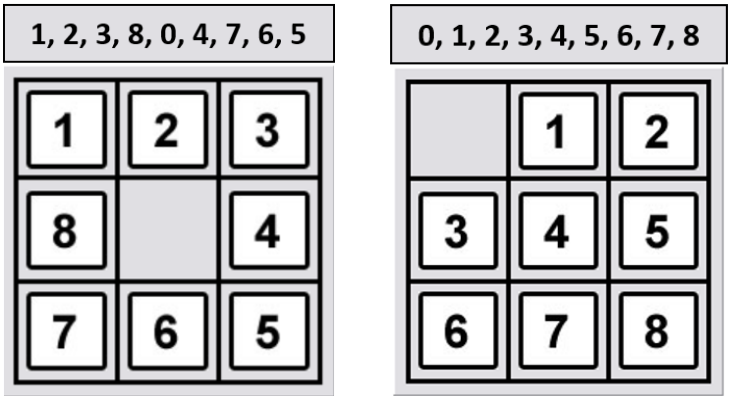

Figure 3 - User Input Format Examples

Can the code solve any pair of configurations?

No, since in the 8-puzzle, there are unsolvable problems, where a path from a given start state to goal state simply does not exist. This can be determined by checking the number of inversions between given start and goal states. If there are odd number of inversions a solution path does not exist, as every legal move would be changing the number of inversions by an even number.

Here an inversion refers to a pair of tiles that are in reverse order (between the start and goal state). To illustrate, if we assume start (left) and goal (right) states as in Figure 3. We can see that this pair is unsolvable, there are 11 inversions (*[0 1] [0 2] [0 3] [0 8] [8 4][8 7] [8 6] [8 5] [7 6] [7 5] [6 5]*).