8 Puzzle using A* Algorithm

Group Members:

Julius Breuß

Ling Yan Chan

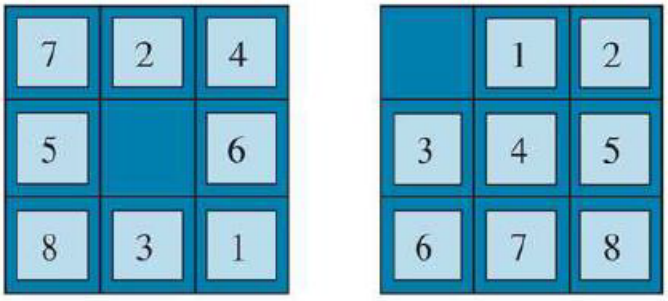
Johannes Jeremias Dittmoser-Pfeifer

Georg Fehringer

Merve Gökce

Contents

1. Short Task Description



8 puzzle consists of a 3x3 grid (containing 9 squares) with one empty square. We start from a random start state, and try to move to a goal state, as shown in the above figures. The idea is to get each square into the assigned position by shifting them around with the empty square. A* algorithm will be used in order to solve the 8-puzzle problem.

A* Search: f(n) = g(n) + h(n), where h(n) is the estimated cost of steps to go from n to reach the end point; g(n) is the number of nodes traversed from the start node to current node.

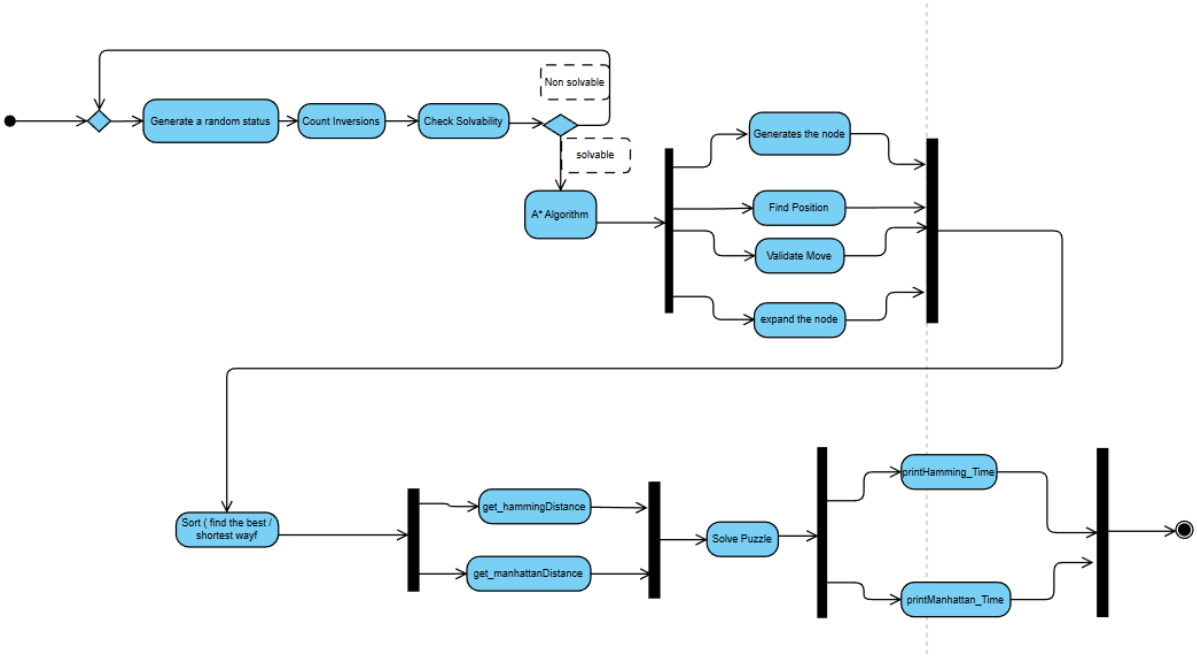
First, a function for checking the puzzle's solvability should be included in this task.

Additionally, the solution needs to provide information on the algorithm's complexity, mean and standard deviation of memory usage and execution time for each heuristics while solving 100 puzzles each.

The two different heuristic functions used in this task are Hamming-Distance and Manhattan-Distance.

In the above states, the hamming distance is 9, because all 9 tiles are misplaced, and the manhattan distance is 18 (the sum of vertical and horizontal distance from the tiles to their goal positions)

2. Software Architecture Diagram



3. Short descriptions of modules and interfaces

In the following part, we explained all modules we implemented in our task.

Main: Our main method only consists of our “solve_all()” method and printouts for the results of average running time, average expanded nodes and standard deviation for both heuristics inside this module.

Solvability: To check if the puzzle is solvable, we need to check the inversions in our puzzle. To accomplish this, we convert the 2D array containing our puzzle into a 1D array to make it easier to test for our inversions function. After calling the function, solvability is determined by looking at the returned number of inversions. If this number is even, the puzzle is solvable. If not, the puzzle cannot be solved.

Inversions: 8 puzzles are only solvable if there is an even number of inversions. If the values on tiles are in reverse order of their appearance in the target state, a pair of tiles

forms an inversion. This function is counting all inversions, while ignoring the empty tile, and returns the value.

Node class: This class establishes the state's structure, setting the associated values for each node. These values include the parent node, the previous move and the puzzle array itself, as well as the g- and h-score for each state

Expand Node: This function is responsible for the main part of this task. It checks if a move into each direction is possible and if yes, creates that state with all the associated values. Then it checks if this state is not already saved as a node using our hash function and our hashlist. If not, it creates the node and adds it to the nodelist as well as the hashlist.

Validate Move: This function checks if the wanted move is allowed. This could be denied based on the position in the array (e.g. the empty square is in the top left field and wants to move up or left) and also if it doesn't return to the state it was before.

Cost Calculation: This function, called "calc_cost" was adding together the depth of the search and the heuristic distance. The result is returned. Now this function does not add the depth because our program is faster by searching the goal state just by the heuristic distance. Therefore, we commented the depth out.

Find position: The function is called "find_pos". An 3x3 array and a number are served as input here, then the position of the number in the array will be returned.

Get hamming distance: Two 3x3 arrays will be the input here, which are the current state and the goal state respectively. After it runs successfully, the number of not correctly placed numbers will be returned.

Get manhattan distance: This is similar to the function above, but the Manhattan distance is calculated here. x1 and y1 represent the current location, while x2 and y2 represent the goal location. This distance is calculated by adding the vertical and horizontal difference from each square to its representative goal location.

Solve 8 puzzle: We have two functions which are "solve_8_puzzle" and "solve_all" here. We use the above functions here in order to solve our 8 puzzle, such as

“expand_node”, “solvability”, “get_hamming_distance” and “get_manhattan_distance” etc. It includes the current node, and starts to count the time. After each puzzle is finished, the time gets stopped and added to the respective array. It then displays a counter in the console for each puzzle that was solved by both heuristics.

4. Explain design decisions

Start generate a random state check Solvability implement Heuristics A Algorithms*

superior functionality:

- Generate a random Status in Python with NumPy
- Check Solvability with a Permutation. The permutation has the parity property; it refers to the parity of the number of inversions. The permutation can have even parity or odd parity

A* Algorithms:

- Manage the Queue: expand the node or delete the node
- How do I store the states? In which data type of a 2D Array
- Cost: F(n) = G(n) + H(n)

Admissible Heuristics H(n)

- Implement Manhattan Distance
- Implement Hamming Distance

The A* algorithm helped us find the correct solution of the 8 puzzles because it returned the best 8 puzzle solution. The program first creates a randomly solvable puzzle and reaches the target state using the A* search algorithm and the two heuristics.

5. Discussion and conclusions

5.1 Describe you experience

As a team, we divided the tasks among ourselves. The implementation was commented and documented accordingly. GitHub allowed us to work on our 8-puzzle project at the same time.

The first step in creating the 8-puzzle was to generate random numbers. We downloaded NumPy as a module and wanted to represent our 8-puzzle as an array for easier progression. The next step was to develop a method to check whether an 8-puzzle is solvable or not. Then came the big hurdle: implementing the A* Algorithm and the heuristics. We opted for Manhattan and Hamming, as also defined in the statement.

5.2 Provide a table with complexity comparisons of different heuristics

	Hamming (Number of misplaced tiles)	Manhattan
Average Number of Expanded Nodes	1071860.80	1073596.65
Average Computation Time	23.3808	0.3746
Average Standard Deviation Expanded Nodes	577923.34	577743.69
Average Standard Deviation Time	41.2904	0.8212

In the table above we measured the average time,expanded nodes and the respective standard deviation for both heuristics with the calculated cost (depth + heuristic distance).

	Hamming (Number of misplaced tiles)	Manhattan
Average Number of Expanded Nodes	57078.34	57346.12
Average Computation Time	0.0535	0.0225
Average Standard Deviation Expanded Nodes	30898.85	30883.46
Average Standard Deviation Time	0.0495	0.0146

Since we had a lot of problems with the duration of the respective runs, we realised that by removing the depth from our cost calculation, both heuristics performed a lot better.

5.3 Possible improvements in future

We could try other heuristics and compare them based on their memory cost (number of nodes expanded) and runtime for each random state and heuristic to make them more efficient.

It would also be possible to make the 8-puzzle more user-friendly by implementing a GUI where users define numbers of the fields themselves, or by making the steps of the algorithm visible and showing each step as an animation in the console output.

There are also still performance improvements possible in the code by more efficient Node selection or using better search algorithms for already created states.