

UNIVERSITETI I PRISHTINËS “HASAN PRISHTINA”
FAKULTETI I SHKENCAVE MATEMATIKO-NATYRORE
DEPARTAMENTI I MATEMATIKËS
PROGRAMI: SHKENCA KOMPJUTERIKE



Title:

8 puzzle game with A* algorithm implementation

Subject: Artificial Intelligence

Worked by:

- 1. Eljesa Kqiku**
- 2. Rinesë Morina**
- 3. Rona Latifaj**

Profesor: Besnik Duriqi

Content of work:

1. A* algorithm
 - a. A* algorithm overview
 - b. A* algorithm pseudocode
2. The 8 puzzle problem
 - a. The 8 puzzle problem overview
 - b. Solution of a case of 8 puzzle problem with A*
 - c. Code - The 8 puzzle problem implementation
 - d. The 8 puzzle problem technical documentation of code
 - e. Class diagrams

1.1 A* algorithm overview

A* Algorithm is an informed search algorithm that is commonly used in pathfinding and graph traversal problems. It is a type of best-first search algorithm that uses a heuristic function to guide the search towards the goal node. The algorithm works by maintaining two lists - an open list and a closed list. The open list contains nodes that have not yet been visited, while the closed list contains nodes that have already been visited.

The input to the algorithm is a graph $G(V,E)$ with a source node start and a goal node end. The output is the least cost path from the start node to the end node. The algorithm works by initializing the open list with the start node and the closed list as empty. It then calculates the cost from the start node to each node using the function $g(\text{start}) = 0$ and the heuristic function $h(\text{start}, \text{end})$.

The algorithm then enters a loop that continues until the open list is empty. In each iteration, the algorithm selects the node m with the least total cost $f(n) = g(n) + h(n)$ from the open list. If m is the goal node, the algorithm terminates and returns the node m . Otherwise, the algorithm removes m from the open list and adds it to the closed list. It then checks the child nodes of m and calculates the cost to reach each child node.

If a child node n is already in the closed list, the algorithm skips it and continues to the next child node. If n is in the open list and the new path to n has a lower cost than the previous path, the algorithm updates the cost and removes n from the open list. If n is in the closed list and the new path to n has a lower cost than the previous path, the algorithm updates the cost and removes n from the closed list. If n is neither in the open list nor the closed list, the algorithm adds n to the open list and calculates the cost and heuristic function for n .

The time complexity of the A* algorithm depends on the quality of the heuristic function used. If the heuristic function is admissible (never overestimates the actual cost), the A* algorithm is guaranteed to find the optimal solution in the shortest amount of time. However, if the heuristic function is not admissible, the A* algorithm may not find the optimal solution but will still find a solution. In the worst case, the time complexity of the A* algorithm is exponential.

The space complexity of the A* algorithm is also dependent on the size of the graph and the quality of the heuristic function. In the worst case, the space complexity can be exponential.

The A* algorithm is a completed algorithm, meaning that it will always return a solution if one exists. However, the quality of the solution returned depends on the quality of the heuristic function used.

The A* algorithm is most commonly used in pathfinding problems in robotics, video games, and other applications where finding the shortest path between two points is important. It is also used in other applications such as route planning, scheduling, and data analysis. The algorithm has a number of advantages, including its ability to handle complex environments and its relatively fast performance compared to other pathfinding algorithms.

1.2 A* algorithm pseudocode

Input: A graph with source node start and goal node end.

Output: Least cost path from start to end.

S-1: Initialize

$open_list = \{ start \}$ //

$closed_list = \{ \}$ //

$g(start) =$ //

$h(start) = heuristic_function(start, end)$ //

S-2: while open_list is not empty do S3 –

S-3: Node on top of open_list , with least

S-4: if end

return m

S-5: remove from open_list

add to closed_list

S-6: for each in child

S-7: if in closed_list

continue in S-2

S-8: distance

S-9: if in open_list and cost (n)

S-10: remove from open_list as new path is better

S-11: if in closed_list and cost

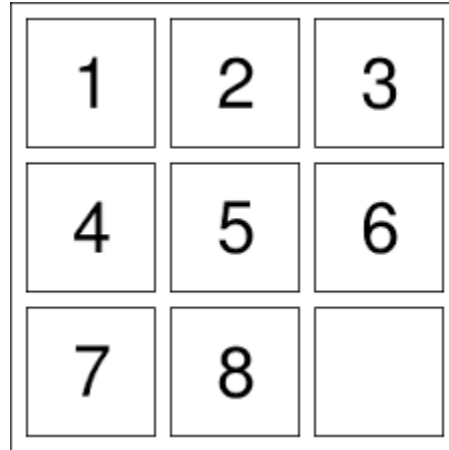
remove n from closed list

S-12: if not in open_list and not in closed_list

add to open_list

S-13: return failure

2.1 The 8 puzzle problem overview



| | | |
|---|---|---|
| 1 | 2 | 3 |
| 4 | 5 | 6 |
| 7 | 8 | |

Fig. 8 puzzle problem goal state

The 8-puzzle problem is a classic problem in the field of artificial intelligence and computer science. It involves a 3x3 board with eight numbered tiles and one empty space, and the goal is to move the tiles to achieve a specific arrangement or goal state. The problem can be framed as a search problem in which the search algorithm must explore the space of possible moves until the goal state is reached.

The 8-puzzle problem has many different variations, but the most common one is the "sliding tiles" version, where the tiles can only be moved horizontally or vertically into the empty space. The problem is often used for testing search algorithms because it is simple enough to understand but complex enough to be challenging for many algorithms.

The 8-puzzle problem can be solved using a variety of search algorithms, including uninformed search algorithms such as breadth-first search or depth-first search, and informed search algorithms such as A* search. In general, uninformed search algorithms tend to be less efficient for solving the 8-puzzle problem because they explore a large number of states that are unlikely to lead to a solution. In contrast, informed search algorithms such as A* search use heuristics to guide the search towards the goal state, leading to more efficient search.

The most commonly used heuristic for the 8-puzzle problem is the "manhattan distance" heuristic, which calculates the total distance that each tile is away from its correct position in the

goal state. The sum of these distances provides an estimate of the total cost to reach the goal state, which can be used to guide the A* search algorithm towards the most promising states.

We use the heuristic function called the Manhattan distance, also known as "L1 norm" and "Taxicab Distance", which in mathematics calculates the distance between a pair of vectors.

| | | |
|---|---|---|
| 2 | 8 | 3 |
| 1 | 6 | 4 |
| 7 | | 5 |

The metric is calculated by summing the absolute distance between the components of the two vectors. We use this heuristic to determine which of the transversal states of the graph is closer to the goal state, and the value that we get from this calculation determines the h parameter of our search in a state. For example: in the following case we have the string “283164705”, and this might be our initial or transversal state. If we choose the Manhattan distance to be our heuristic function, then in this case we would have $h=1+2+0+1+1+2+0+1+2=10$.

2.2 Solution of a case of 8 puzzle problem with A*

In our example we start with the initial state of “123485706”. In the first step the A* algorithm evaluates the heuristic function, which in this case is the Manhattan distance, where we get the value of 3 and since this is our initial state the g function returns the value 0. And summing these two we get $g+h=3$ which represents the value of this state thus far.

Initial state

| | | |
|---|---|---|
| 1 | 2 | 3 |
| 4 | 8 | 5 |
| 7 | 0 | 6 |

In the next step the A* algorithm finds the neighbor states in the graph, and it does so by expanding the graph in all of the ways which it still hasn't explored. Since this is the initial state, we know that any state has yet to be discovered. Now we have three new states which we obtain by moving the space (or in this case number “0”) in all possible directions- left, up, and right,

since down is a state not allowed by the constraints of the game. At this point we evaluate the heuristic functions and we get the following values for the neighboring states: 6, 5, and 5. Since

we have two equal values we go with the first one that was represented to us- so we decide to move down by bringing “8” in the middle of the board.

Now we repeat the process by expanding our current states and obtaining new values: 3, 8, and 8. Since 3 is the lowest value of the heuristic function we expand that state, and we find that the goal state is just around the corner.

This presents us with the solution to our problem, with just three steps. We obtain the path of the solution by backtracking into the initial state.

Initial state

| | | |
|---|---|---|
| 1 | 2 | 3 |
| 4 | 8 | 5 |
| 7 | 0 | 6 |

 $g=0, h=3, g+h=0+3$

| | | |
|---|---|---|
| 1 | 2 | 3 |
| 4 | 8 | 5 |
| 0 | 7 | 6 |

 $g=1, h=5, g+h=6$

| | | |
|---|---|---|
| 1 | 2 | 3 |
| 4 | 0 | 5 |
| 7 | 8 | 6 |

 $g=1, h=4, g+h=5$

| | | |
|---|---|---|
| 1 | 2 | 3 |
| 4 | 8 | 5 |
| 7 | 6 | 0 |

 $g=1, h=4, g+h=5$

| | | |
|---|---|---|
| 1 | 2 | 3 |
| 4 | 5 | 0 |
| 7 | 8 | 6 |

 $g=2, h=1, g+h=3$

| | | |
|---|---|---|
| 1 | 2 | 3 |
| 0 | 4 | 5 |
| 7 | 8 | 6 |

 $g=2, h=6, g+h=8$

| | | |
|---|---|---|
| 1 | 0 | 3 |
| 4 | 2 | 5 |
| 7 | 8 | 6 |

 $g=2, h=6, g+h=8$

| | | |
|---|---|---|
| 1 | 2 | 3 |
| 4 | 5 | 6 |
| 7 | 8 | 0 |

 $g=3, h=0, g+h=3$

| | | |
|---|---|---|
| 1 | 2 | 0 |
| 4 | 5 | 3 |
| 7 | 8 | 6 |

 $g=3, h=3, g+h=6$

Goal state

2.4 The 8 puzzle problem implementation class diagram

The solution to this implementation is written in Java, and it includes five core components: Graph, InformedGraph, Node, PuzzleState, and Utilities. The class diagram is seen below where InformedGraph extends the Graph interface, and the abstract class Node is extended by the PuzzleState class. The four entities mentioned above the “brain of the program”, whereas the public class Utilities is a “helping” entity which deals with mathematical functions such as permutations and factorial.

