Kyle Tranfaglia

COSC 411

Project 02

11 Nov. 2024

# 15-Puzzle A* Implementation Report

a) Choosing Heuristic: Manhattan Distance

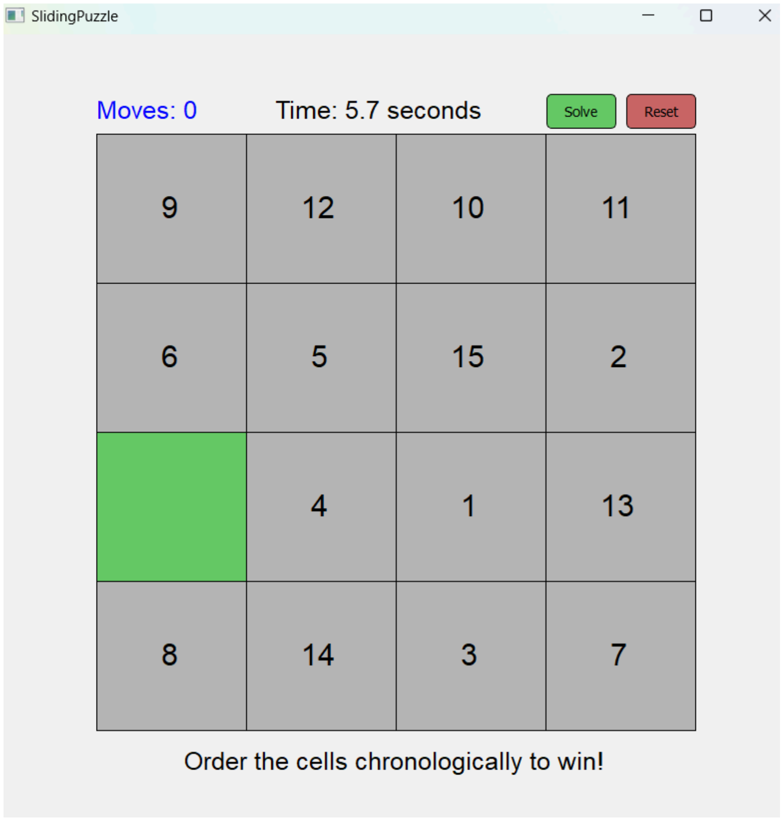
I chose to use Manhattan Distance not only because it is a popular heuristic that we have reviewed before and is relatively simple to implement but also because it provides the most useful estimation of cost. It is used to provide a balanced measure of how far the tiles are from their goal positions as a measurement of the total distance traveled to move from the current state to the goal state. Unlike Euclidean Distance, which is a straight-line path, Manhattan Distance works well in this case as it measures the total number of moves it would take to get that tile to its goal position if it could only move horizontally or vertically in a straight line, which is true for the 15-Puzzle game. Manhattan distance for 15-Puzzle means calculating the distance for each tile to get to the goal state and summing them up. Each tile's Manhattan Distance is the difference in its row position and column position relative to its target position (sum of |x_current - x_goal| + |y_current - y_goal| for all tiles). This heuristic represents the lower bound on the number of moves required to solve the puzzle, which is good since it makes the A* search admissible such that it never overestimates the goal. This property ensures that the A* algorithm will always find the optimal solution if one exists. Additionally, Manhattan Distance balances computation and guidance toward the goal state. It doesn't involve complex calculations, making it efficient to repeatedly compute each state in the more computationally heavy

A* search. I tried to implement a linear combination heuristic with Manhattan distance, but the performance (in terms of time) was worse, meaning there was not enough guidance improvement to make up for the computations. I did not attempt to implement a misplaced tiles heuristic as I figured it would not guide well since it simply counts the number of tiles not in their goal positions. Although admissible, it is likely less informative than the Manhattan Distance, as it doesn't consider how far tiles are from their destinations. Also, a scalar of 1.5 was used to improve the heuristic estimate and significantly improve algorithm performance; as mentioned before, the heuristic alone represents the lower bound. Overall, I found Manhattan Distance to be a good balance between accuracy and time complexity, and it was simple to implement.

b) A* Implementation

The A* Search algorithm I implemented relies on a priority queue, which I made using a min-heap for efficiency. I used a min-heap because I was experiencing some performance issues with the algorithm, especially on my laptop. I recalled using a min-heap before to help optimize a program, and it worked well as a substitution for the priority queue in this case as well. The min-heap allows for the exploration of puzzle states in an order that minimizes the estimated total cost to reach the solution. Of course, to be an A* Search, the min-heap is ordered by the evaluation function $f(n) = g(n) + h(n)$. This priority queue is essential because it ensures that states with the lowest estimated total cost are explored first, keeping the search efficient. My $h(n)$ is calculated using Manhattan Distance, as described above. The true cost to reach a state, g, is represented by the move count from the initial to the current configuration. Each move increments this cost by one. The algorithm begins by pushing the initial puzzle configuration onto the priority queue (min-heap). If the initial state is solved, the algorithm terminates; otherwise, it explores neighboring states. To further optimize the program, I added a dictionary of moves for each empty cell location so that it would not explore anything out of bounds or spend excess time checking valid moves. Continuing with the algorithm, for the current state (the one with the lowest f-value), the algorithm explores all legal moves (e.g., up, down, left, or right). For each neighbor, g is incremented to account for the additional move to reach the state, an h-value is recalculated based on the state's Manhattan Distance, and an updated f-value is calculated by summing g and h. Other neighboring states are added to the priority queue for evaluation in the upcoming steps. To avoid re-exploring states and to optimize performance, each visited state is stored in a dictionary. If a generated neighbor state has been visited before, it's skipped. This prevents the algorithm from getting stuck in loops and reduces redundant computations, which my algorithm fell victim to before this addition. After generating a neighbor, the algorithm checks if it matches the goal configuration. If it does, the algorithm terminates, as the optimal path to the goal has been found. After reaching the goal, the program traces back through the solution path. This sequence is then displayed in the real-time visual animation, which shows each move with a 0.75-second delay to allow for step tracking from start to finish.

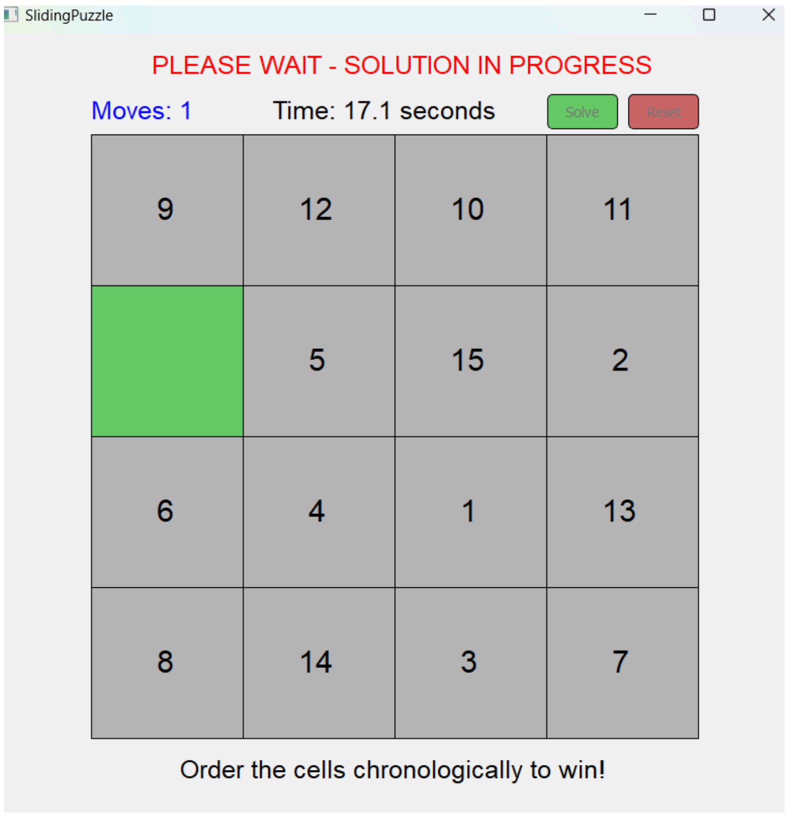
c) Demo



Moves: 0 Time: 5.7 seconds

|  |  |  |  |
|---|---|---|---|
| 9 | 12 | 10 | 11 |
| 6 | 5 | 15 | 2 |
| 🟩 | 4 | 1 | 13 |
| 8 | 14 | 3 | 7 |

Order the cells chronologically to win!

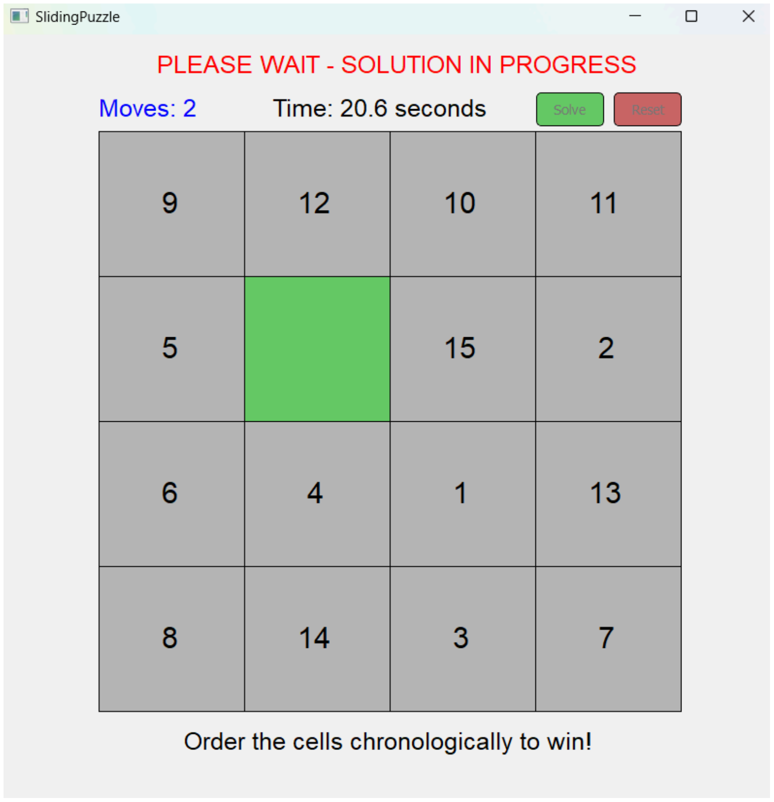


PLEASE WAIT - SOLUTION IN PROGRESS

Moves: 1 Time: 17.1 seconds

|  |  |  |  |
|---|---|---|---|
| 9 | 12 | 10 | 11 |
| 🟩 | 5 | 15 | 2 |
| 6 | 4 | 1 | 13 |
| 8 | 14 | 3 | 7 |

Order the cells chronologically to win!

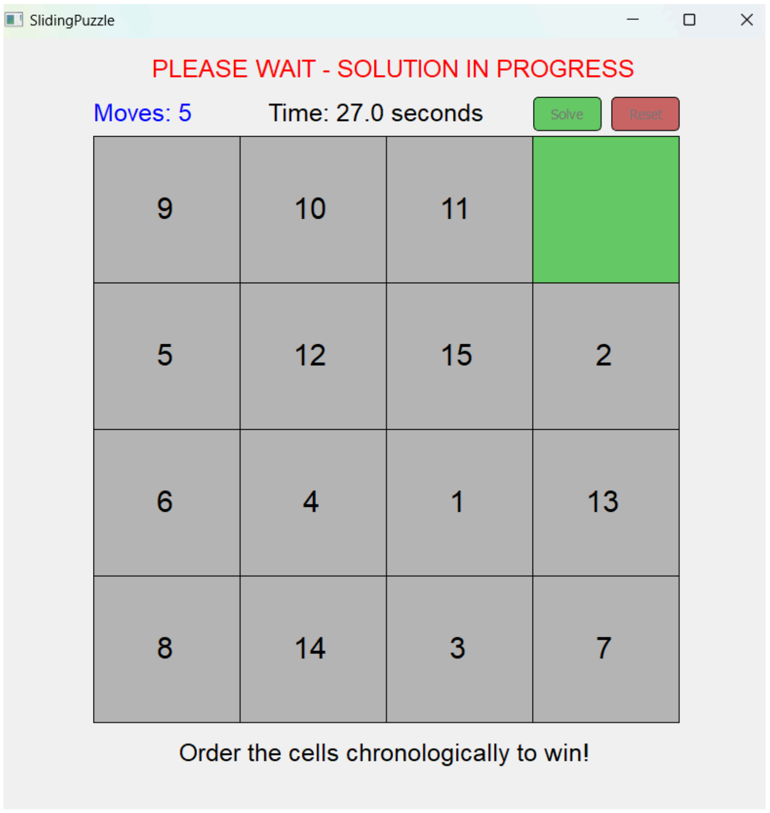


PLEASE WAIT - SOLUTION IN PROGRESS

Moves: 2 Time: 20.6 seconds

| | | | |
|---|---|---|---|
| 9 | 12 | 10 | 11 |
| 5 | 🟩 | 15 | 2 |
| 6 | 4 | 1 | 13 |
| 8 | 14 | 3 | 7 |

Order the cells chronologically to win!

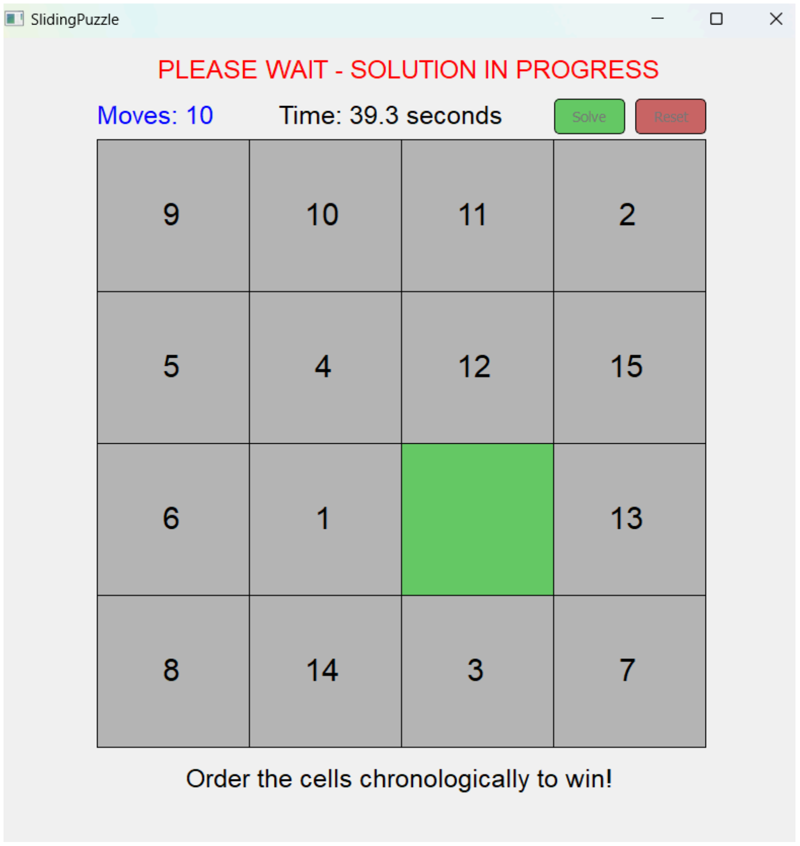


PLEASE WAIT - SOLUTION IN PROGRESS

Moves: 5 Time: 27.0 seconds

| | | | |
|---|---|---|---|
| 9 | 10 | 11 | 🟩 |
| 5 | 12 | 15 | 2 |
| 6 | 4 | 1 | 13 |
| 8 | 14 | 3 | 7 |

Order the cells chronologically to win!

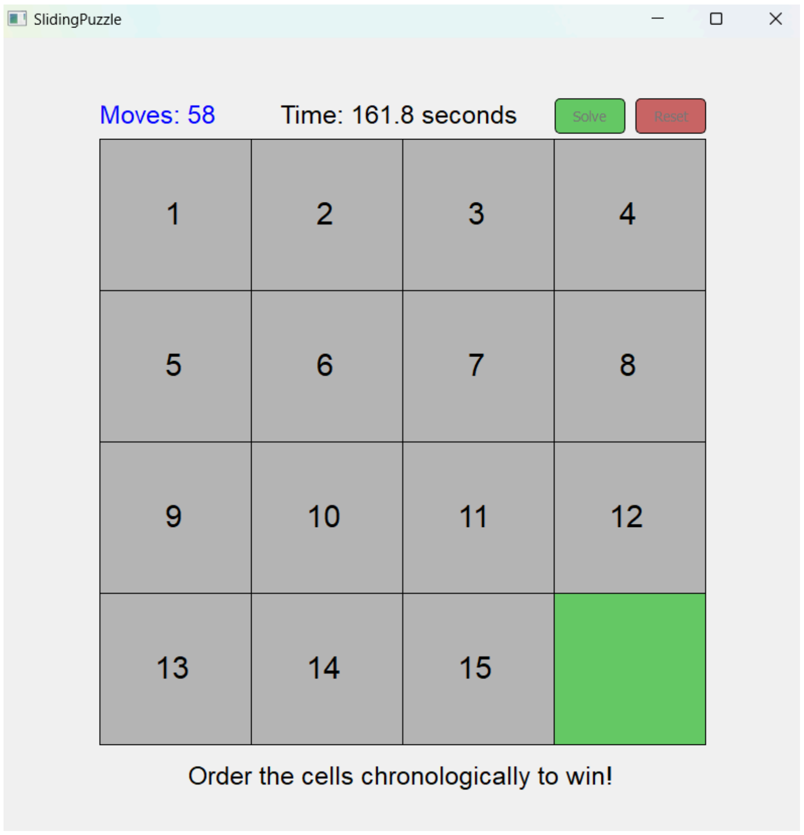


PLEASE WAIT - SOLUTION IN PROGRESS

Moves: 10 Time: 39.3 seconds

| | | | |
|---|---|---|---|
| 9 | 10 | 11 | 2 |
| 5 | 4 | 12 | 15 |
| 6 | 1 | 🟩 | 13 |
| 8 | 14 | 3 | 7 |

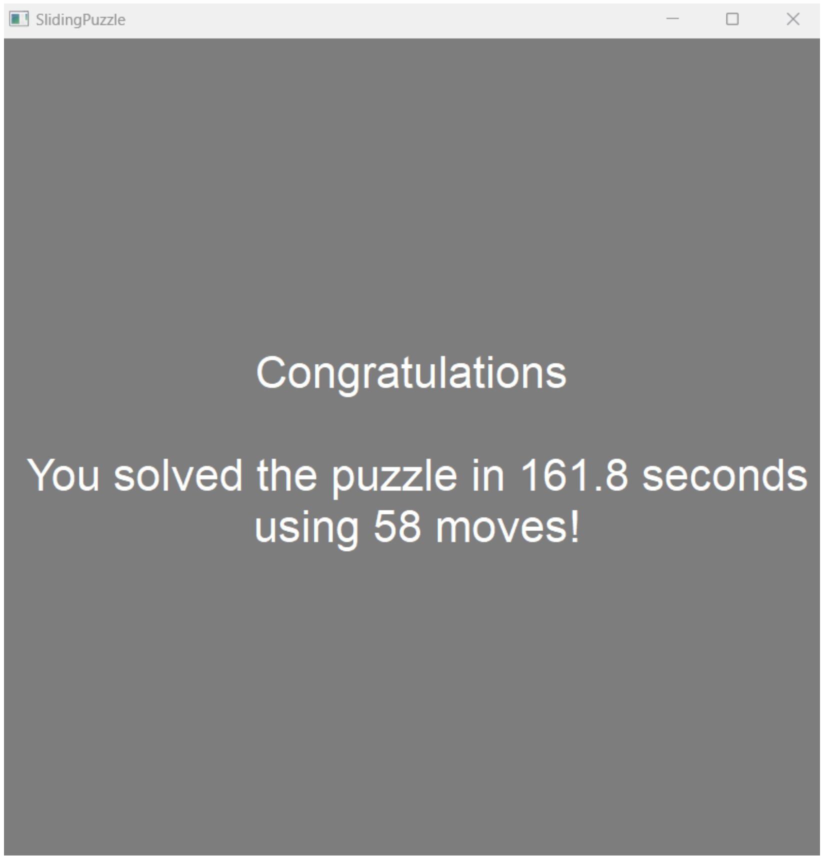
Order the cells chronologically to win!



Moves: 58 Time: 161.8 seconds

| | | | |
|---|---|---|---|
| 1 | 2 | 3 | 4 |
| 5 | 6 | 7 | 8 |
| 9 | 10 | 11 | 12 |
| 13 | 14 | 15 | 🟩 |

Order the cells chronologically to win!



Congratulations

You solved the puzzle in 161.8 seconds using 58 moves!

d) References

https://www.geeksforgeeks.org/pyqt5-how-to-hide-label-label-sethidden-method/

https://www.codecademy.com/resources/docs/ai/search-algorithms/a-star-search

e) Collaborators

This project was completed individually. Nothing was discussed with any individual regarding this project.