Report for assignment 2, TDT4316

Eliah Mathias Hudgins

September 22, 2023

1 Introduction

The provided Python code implements the A* algorithm for pathfinding in grid-based environments. The A* algorithm is widely used for finding the shortest path between two points on a grid, taking into account the cost of moving through cells and an admissible heuristic to guide the search. This report aims to show the results of the implementation of the A* algorithm for locating lost friends at Studentersamfundet in Trondheim. This was done on a 2D grid, provided by the assignment in a separate Map.py file, along with other help full functions.

2 Code overview

1. **Import and Initialization:** It starts by importing necessary modules and initializing the Map-Obj class. The Map-obj class is imported from the Map.py file, which came with functions which were already encoded. However, one extra function was added here, the Map-array function. The task to be solved is specified by changing the task parameter in the fill-critical-positions method.

2. **Main function (main):** The main function serves as the entry point for the code execution. It performs the following actions:

    - Initializes a Map-Obj instance, specifying the task to be solved.
    - Sets up the grid map, starting position, goal position, and cost map.
    - Calls the A* algorithm (Astar) to find the path.
    - If a path is found, it prints the path, otherwise, it informs that no path was found.
    - Calls the visualize-path function to visualize the grid and the path found.

3. **Node class (node):** The node class represents a node in the search space. It stores the state, parent node, and cost to reach the current node. It also defines a custom comparison method based on cost.

4. **Heuristic function (node):** This function calculates the heuristic cost from a current node to the goal node. It uses the Manhattan distance as the heuristic.

5. **A* algorithm:** The heart of the code is the A algorithm. It takes the grid, start node, goal node, and cost map as input and returns the path from the start to the goal if found. It uses a priority queue to explore nodes based on their total cost.

6. **Find neighbors function:** This function finds neighboring nodes that can be reached from the current node. It takes into account valid movements and the cost associated with different cells in the grid.

7. **Visualization function:** This function visualizes the grid and the path found. It uses the show-map method from the Map-Obj class to display the grid. The path is marked with 'Z' symbols

3 Task 1 and 2

For task 1 and 2, in part 1 of the assignment, there were only grids which were either walk-able or none-walkable. There were no additional costs for the nodes, and the algorithm simply finds the shortest path possible. This is calculated based on the Heuristic cost and node cost. I have chosen to define the heuristic cost as the Manhattan distance from the current node to the goal node, as this was fairly straightforward to implement in my code, and i don't really understand euclidean distance.

The board configuration as mentioned was supplied by the Map.py file which was readily available as an appendix, containing all the different maps of samfundet, each with separate start and end goals.

For the complete explanatioin fo the code and how it works, see the uploaded python file for further specification

Shortest paths from Rundhallen to Strossa (task 1)

The path for the first task was relatively straight forward. As the costs of all nodes was equal, there was only one straight-forward path the algorithm should have chosen, which it eventually did. See figure 1 and 2 for the complete paths and visited nodes

Supplied here are visualizations of the path walked, as well as all the nodes which the algorithm visited when it was traversing the graph.

Shortest paths from Strossa to Selskapssiden (task 2)

The same concept for the previous task was applied to this one, as the node cost was still the same, and only the start and goal positions differ. For simplicity in the code, the only

parameter which is needed to change is the "task", from task 1 to task 2. This will yield the following results. (See figure 3 and 4 for the complete paths and visiting algorithm.)

4 Task 3 and 4

For task 3 and 4 in part two of this assignment, the grids were unwalkable, and there were also certain restrains on the different nodes in the grid. Some nodes in the walkable paths had different costs, thereof influencing the path the algorithm chose to take. The heuristic cost is still calculated the same, however this is where the cost-map variable, part of the Neighbors-function comes into play. It takes the specified costs of the 2D grid, adds them to a numpy array and then feeds them to the A* function. This is when the A* algorithm really starts to shine, as it demonstrates its ability to take into account both the current-node cost, as well as the overall heuristic cost of traversing the grid.

As mentioned above, the 2D grid is supplied from the Map.py function. The cost-map variable uses a function called Map-array, which can be located in the Map.py function (it contains the numpy array).

For the complete explanation of the code, check out the Astar-V3.py file

Weighted path from Lyche to Klubben

In this task, there apperently was a lot of people in the stairs from edgar to Rundhallen, so the shortest possible path was some other way. Using the supplied 2D grid with the different node costs and chaning the task variable from either 1 or 2 to 3, the algorithm takes into account the increasing value of the nodes. See figures 5 and 6 for the complete paths as well as the nodes which the algorithm chose to visit with the help of the Heuristic function.

Weighted path from Lyche to Klubben, considering cake party at Edgar

For this task, Edgar was filled with students due to a sudden free chocolate cake party, resulting in a very crowded space, taking a lot of time to walk though. Therefore, the same implementation as above was used, however this time the task variable was changed to 4, making sure that the correct supplied 2D grid was used (the one were edgar is filled with students). This yielded a path which does not go through Edgar, and we can se from the Heuristic function that the algorithm took several different paths first, before eventually finding the correct one. See figures 7 and 8.

5 Conclusion

In conclusion, the provided code demonstrates the application of the A* algorithm for pathfinding in grid-based environments with different characteristics. By specifying the task, users can adapt the code to handle various scenarios, including maps with varying cell

costs. The A* algorithm efficiently explores the grid to find the shortest path, taking into account both movement costs and a heuristic function. Visualizations help in understanding the pathfinding process and the paths found. Further optimizations and improvements can be made based on specific requirements and use cases.

Figure 1: **Task 1, correct path.** The yellow line shows the path which the algorithm chose to walk for task 1. As there are no additional costs for the different nodes, it simply chose the path which was quickest and shortest.

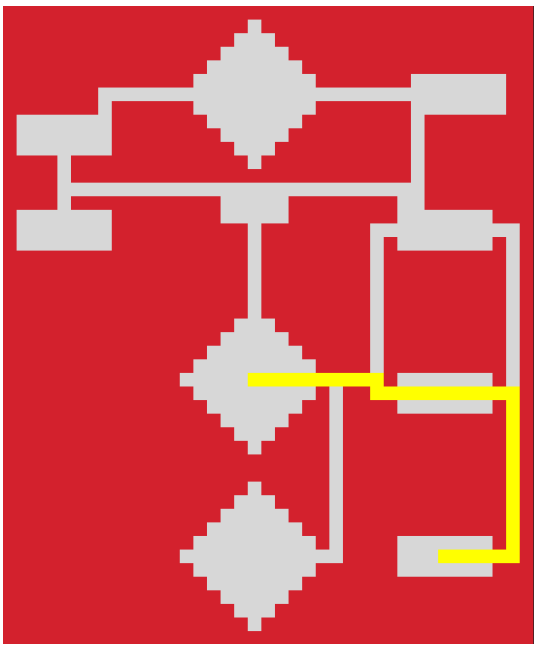

Figure 2: **Task 1, visited nodes.** The colored yellow boxes show all the cells which the algorithm chose to visit before finding the end goal node. This is shown here to demonstrate how the heuristic function works, as it calculates the Manhattan distance from the current node to the goal node for every node it visits. The current node is then added to the queue, and for every node it visits, it checks if there are any other nodes which are lower in the queue than itself, hence the pattern which arises here.)

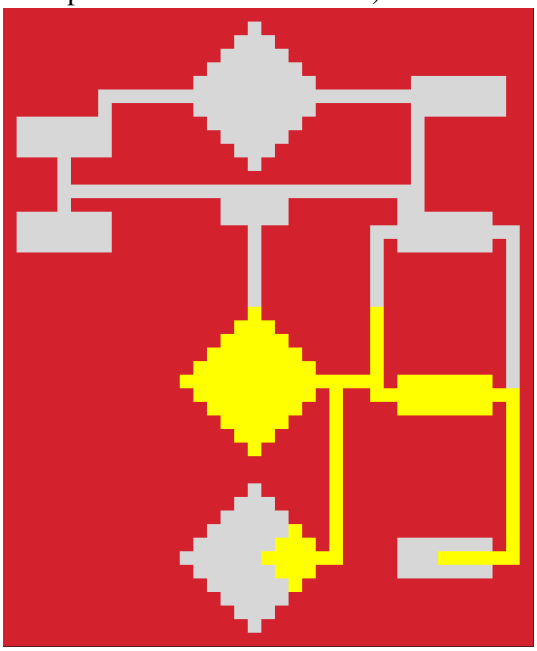

Figure 3: **Task 2, correct path.** The yellow line shows the path which the algorithm chose to walk for task 1. As there are no additional costs for the different nodes, it simply chose the path which was quickest and shortest, based on the heuristic cost and the node cost (which was all 1)

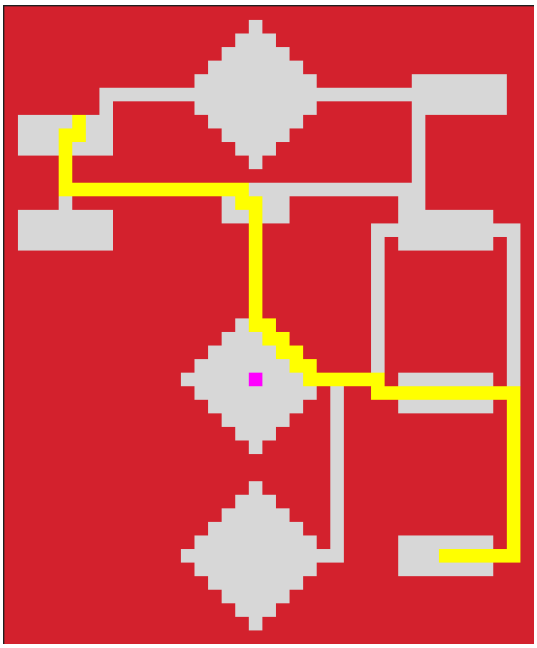

Figure 4: **Task 2, visited nodes.** The colored yellow boxes show all the cells which the algorithm chose to visit before finding the end goal node. The same implementation as for the above instance was used, resulting in a similar pattern of way finding.

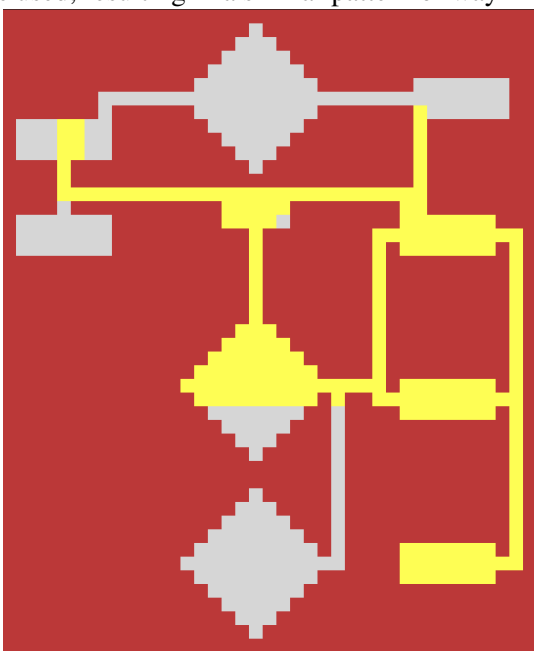

Figure 5: **Task 3, correct path**. Colored yellow line shows the correct path for the algorithm. Here it has taken into account that the stairs from edgar to rundhallen are full, hence taking a different route to the end point.

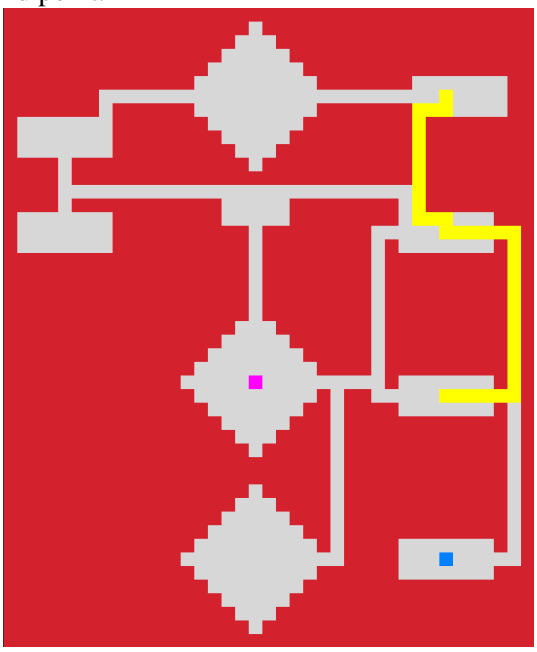

Figure 6: **Task 3, visited nodes**. For this instance, it is interesting to see how the Heuristic function operates. Even though the stairs from edgar to rundhallen are full, it still checked to see if the path would be shorter going that way.

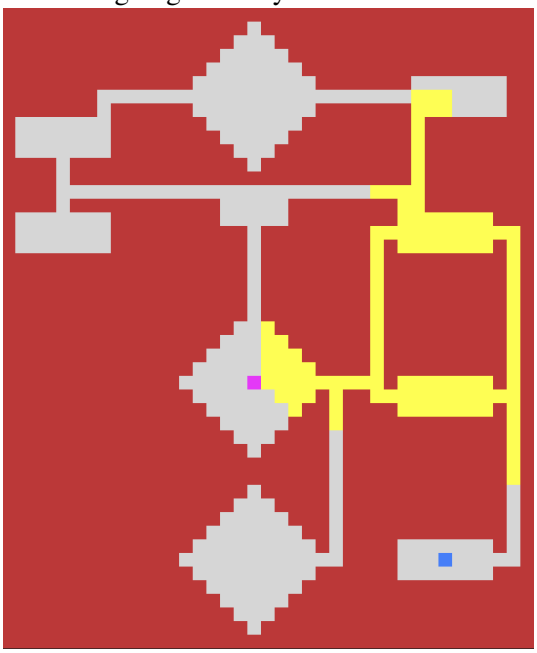

Figure 7: **Task 4, correct path**. Again, the colored yellow line is the correct path. Here, there was a cake party at edgar, and due to this it is not traversable. Heading to klubben therefore required a re-route through somewhere else. It concludes with the quickest path being through rundhallen instead.

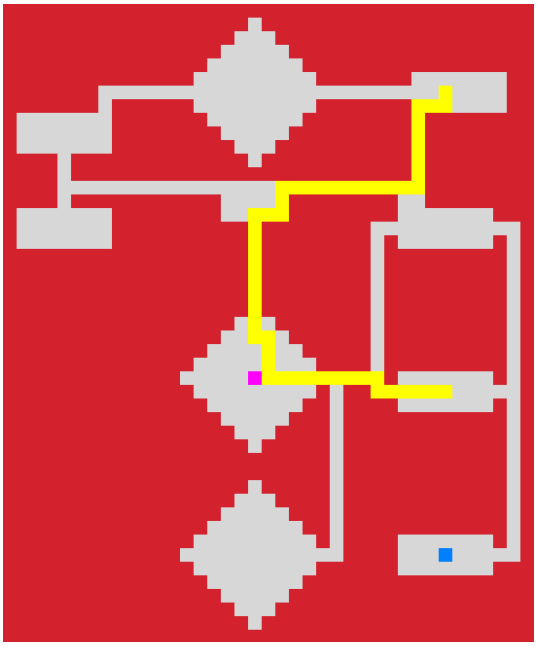

Figure 8: **Task 4, visited nodes**. Again, we can see the heurisitic cost in full effect here. Interestingly, there are two points within edgar which the paths does not try, most likely due the cost increasing beyond some other path through a node which cost less.