Wegesuche Gelände: Suchverfahren A*

Programmentwurf

des Studienganges Informatik

an der Dualen Hochschule Baden-Württemberg Stuttgart

von

Juliette Hild und Julia Bai

Abgabe: 11.01.2023

Bearbeitungszeitraum	08.11.2022 bis 11.01.2023
Vertraulichkeit	Vertraulich
Matrikelnummer, Kurs	6897684 und 2090617, TINF20C
Ausbildungsfirma	Philipps Medical Systems, Böblingen
	Mercedes-Benz Group AG, Stuttgart
Dozent	Prof. Dr. Dirk Reichardt

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Basics and Terms

This section presents key concepts needed to develop the program.

1.1 Adjacency List

The Adjacency List is a list that represents a graph as an array of linked lists. The size of the list is equal to the number of nodes in the graph. In figure 1 you can see a graph on the left side and its Adjacency List at the right side. The value of adjacency_list[“0”] contains every node that is connected to node “0”. The value of adjacency_list[“1”] contains every node that is connected to node 1 etc.

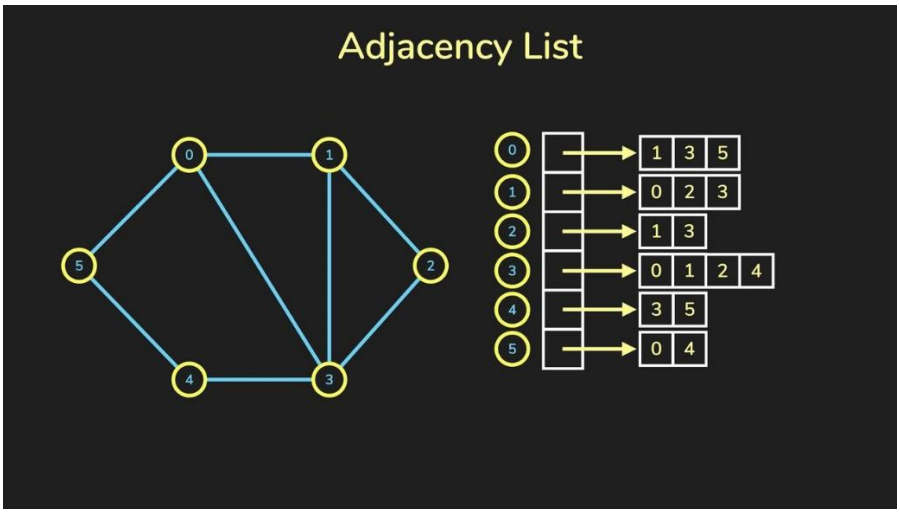


Figure 1 - Adjacency List (cf. (Joma Class, 5))

1.2 Manhattan Distance

The Manhattan Distance is a distance metric between two points measured along axes.
This distance is calculated by the sum of the absolute differences between the coordinates of the points: $d = \left|dst_x - src_x\right| + \left|dst_y - src_y\right|$ (cf. (Szabo, 2015)).

The Manhattan Distance is shown in blue and green in the image above.
The Euclidean metric allows diagonal distance measuring between two points, unlike the Manhattan metric does.

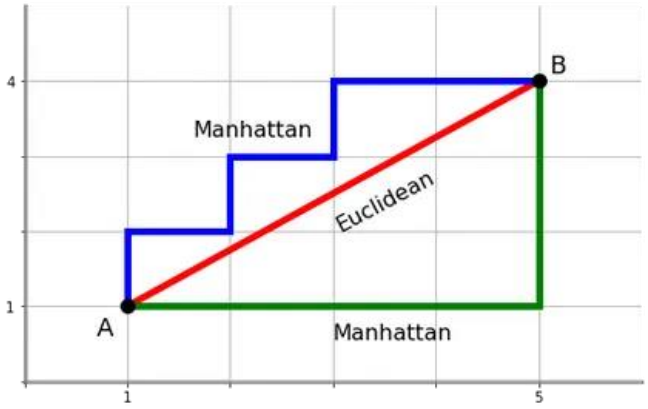


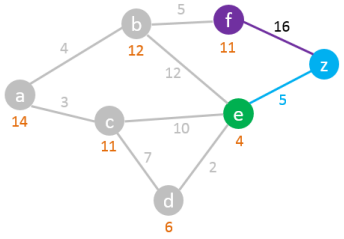
Figure 2 - Comparison between Manhattan and Euclidean Distance (cf. (Omni Calculator, kein Datum))

1.3 A* Algorithm

The A Star Algorithm is a searching algorithm that identifies the shortest bath between a start- and an end position. It contains 3 parameters:

- g: cost from start node to current node
- h: heuristic value, estimated cost from current node to the destination node
- f: total cost $f = g + h$

The algorithm makes decisions by comparing the f value of each node. It selects the node with the smallest f value and moves to that node. The process terminates when the algorithm reaches the destination node. All the nodes to which the algorithm has been moved to will be included to the best path solution (cf. (Geeks for Geeks, 2022)).



Node	Status	Shortest Distance From A	Heuristic Distance to Z	Total Distance*	Previous Node
A	Visited	0	14	14	
B	Visited	4	12	16	A
C	Visited	3	11	14	A
D	Visited	10	6	16	C
E	Current	12	4	16	D
F		9	11	20	B
Z		∞ 12+5=17	0	17	E

* Total Distance = Shortest Distance from A + Heuristic Distance to Z

Figure 3 - Example of the A* Algorithm (cf. (101 Computing, 2018))

2 Design and Implementation

In the following the Design and Implementation of the program is explained.

2.1 Design

The software consists of different classes that have different relationships between each other. In the following, the classes and their relationships are presented.

2.1.1 Classes

There are three classes in the program: The graph, the grid and the node. In Figure 4 you can see the UML Class Diagram of each class. The graph is represented by an adjacency list, the grid by a pandas DataFrame and the node by the key information of a cell in the grid, like position, name, code, cost etc.

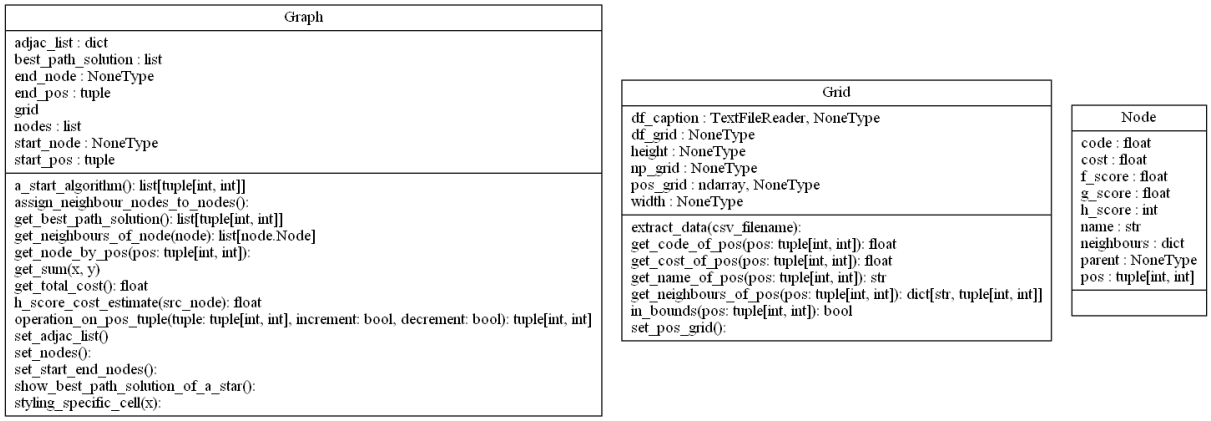


Figure 4 - UML Class Diagram

2.1.2 Graph

The graph class represents a graph. The nodes of the graph are node objects, and the edges represent the connections between the nodes. The graph is represented as an adjacency list. You can apply the a* algorithm on the graph.

2.1.3 Grid

The grid class represents the map and its content information. It contains the extracted map and meta information about it, like the map width and the map height. It also holds the caption information like the name and the cost of a cell with their respective code.

2.1.4 Node

The node class represents a node in a graph. Nodes and edges are components of a graph. A node object contains the information of a specific cell in the map, like their cost, name, position in the grid and their code. They also contain relevant parameters for the A* Algorithm like their g-, h- and f-score.

2.2 Relationship between the classes

The following figure shows the relationships between the classes. The graph is being constructed by a grid object and by node objects. In the main function a grid-object will be instantiated from a certain csv-file and the graph will be instantiated from the information of the attributes of the grid object. For each cell in the grid, a node object will be instantiated and added as a component to the graph.

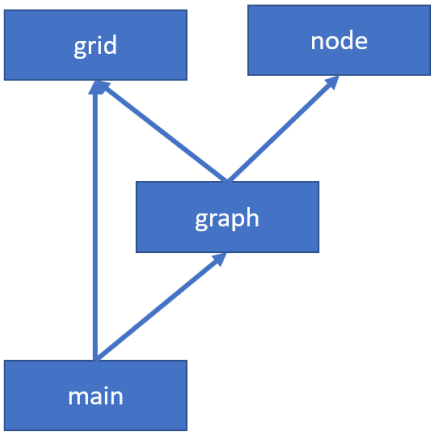


Figure 5 - Class Relationships

2.3 Implementation of the A* algorithm

The Algorithm finds the best path solution from a source node to a destination node of a graph. The best path solution is the path with the lowest cost. It contains these following steps:

1. Start with the starting node in the open_list
2. Remove node from open_list when smallest f_score value
3. Append node to closed_list
4. If the node is the end node -> successful return
5. Else find all neighbours of the node
6. Calculate g-,h- and f-score values of the neighbour nodes
7. Append all neighbour nodes to the open_list
8. GoTo step 2
9. Exit

2.4 Flow of information

In the following, the flow of information from the user input to the user output will be explained.

2.4.1 User Input to Graph

The user inputs consist of a csv file containing the map data. Each cell in the map will later be represented by a node object. The map caption information should also be included in the csv file. So, that for each cell its name and cost can be assigned depending on the cell code. The following picture demonstrates the flow of information from the user Input to the graph object.

In Figure 6 you can see that csv file contains the map and the map caption. With this data the grid object will be instantiated. The grid object as well as the start and end position for the pathfinding will be passed to the graph object. For each cell of the map (*df_grid)* a node object will be instantiated, so that the graph merely consists of nodes and the connections between the nodes.

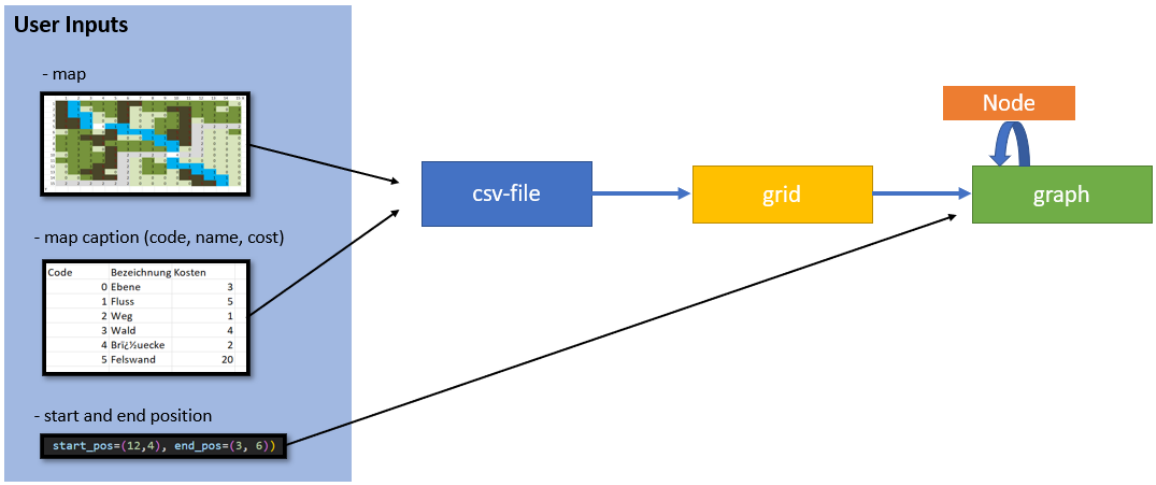


Figure 6 – Flow of information from the user inputs to the graph

2.4.2 CSV file to User Output

The user outputs consist of the information if a minimal cost path is found or not as well as the total cost of the path if it is found. It also outputs a list of tuples that contain the position of each cell in the best path solution in the order from the start node to the end node.

If a jupyter notebook is used to get the output, you can also visualize the best path solution as a colored pandas dataframe. Required for that is the call of the function *show_best_path_solution_of_a_star* in the graph class, after having started the algorithm.

Figure 7 shows the flow of information from the csv-file to the user output. If the graph is finished being initialized, you can start the a*-algorithm and get its solution. Optionally, if a jupyter notebook is used, you can get a visualized output of the best path solution.

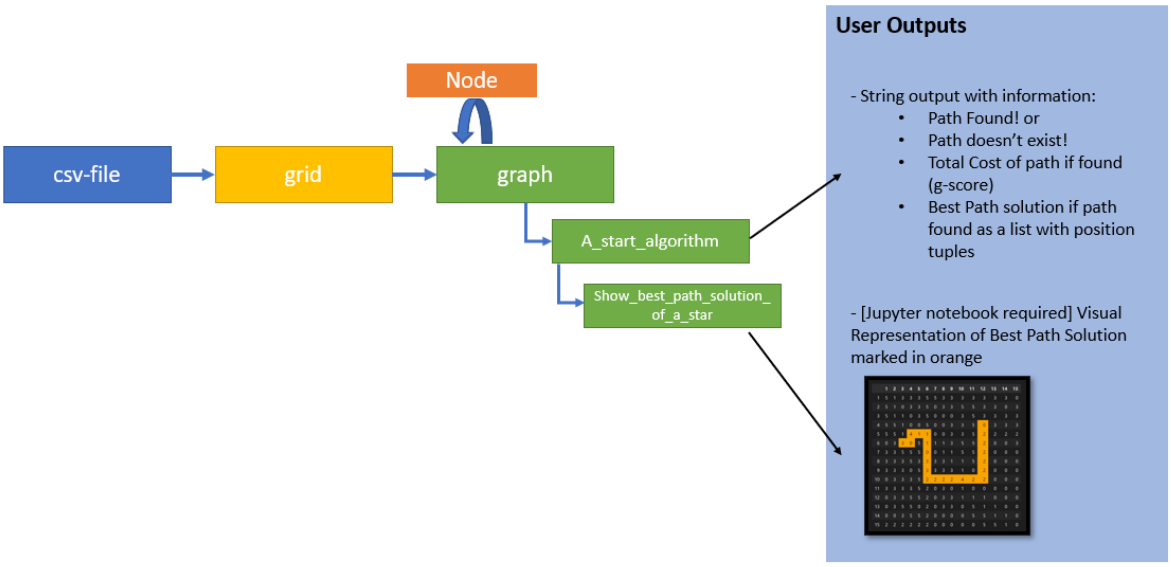


Figure 7 – Flow of information from csv-file to the user outputs

3 Discussion

In the following, the configuration of the program, the solution of the developed program for the exercise problem and the solution of the program for certain test cases is shown and discussed.

3.1 Discussion of the configuration

Figure 8 shows the flow of information within the program. You can see which classes are relevant (grid, node, graph) and how the relationships them need to be so that, in the end, the a*-algorithm can be started, and the best path solution can be shown.

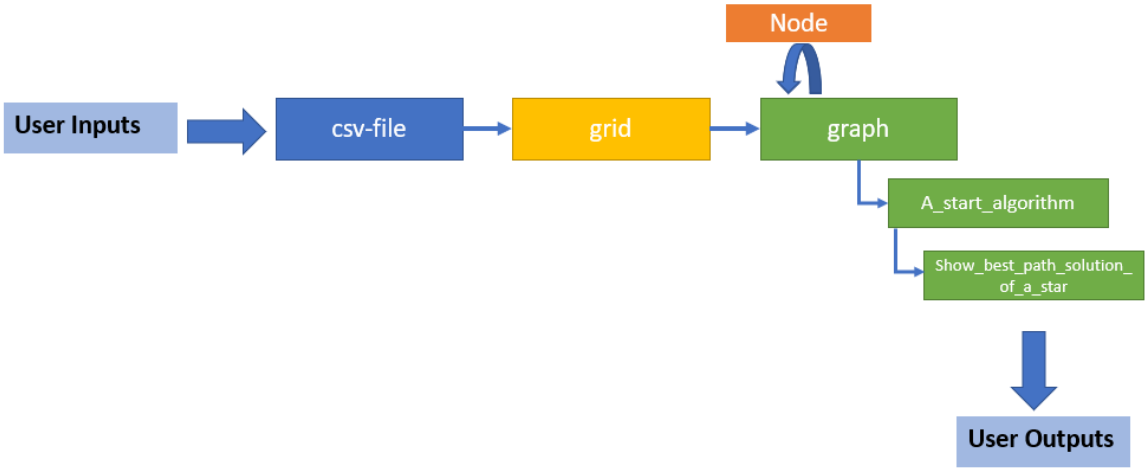


Figure 8 - Flow of information within the program

For solving the exercise problem, the program first reads the given csv-file and creates a grid object from it. From the csv-file the map will be extracted and saved in a pandas DataFrame in *grid.df_grid.* Furthermore, the map caption will be extracted from the csv-file and saved in a separate pandas DataFrame *grid.caption*.

After that, the graph is being instantiated and initialized with the created grid-object and the start- and end position information for the pathfinding algorithm (cf. Figure 9).

```python
import graph
import grid

if __name__ == '__main__':
    # initialization
    csv_filename = "S_001_Daten.csv"                                    # in directory ./SEARCH_001/S001
    landscape = grid.Grid(csv_filename)                                 # Creates grid from extracted data
    g = graph.Graph(grid=landscape, start_pos=(12,4), end_pos=(3, 6))   # Creates Graph with nodes

    # start a* algorithm and show best path solution
    g.a_start_algorithm()
```

Figure 9 - Main Program

During the Initialization of the graph the nodes of the graph as well as the start- and end nodes and the adjacency list is being set.

The nodes are being set by iterating through every cell in the *grid.df_grid* and get the cells code, cost and name. To get this information certain help functions in the grid class exists. This data will then be used to instantiate and initialize a new node object. The node object will then be appended to the nodes list of the graph class (cf. Figure 10).

```python
for x_row in self.grid.pos_grid:
    for pos_el in x_row:
        code = self.grid.get_code_of_pos(pos_el)
        cost = self.grid.get_cost_of_pos(pos_el)
        name = self.grid.get_name_of_pos(pos_el)
        n = node.Node(cost=cost, code=code, name=name, pos=pos_el)
        self.nodes.append(n)
self.assign_neighbour_nodes_to_nodes()
```

Figure 10 - Set nodes function in graph class

After that the position of the neighbour nodes for each node is being set. For that there is the help function *get_neighbours_of_pos(pos: tuple[int,int]) -> Dict[str, tuple[int,int]]* in the *grid* class.

The nodes and the positions of the neighbour nodes for each node is set, the adjacency list can therefore be created. For that, the program iterates through every node and saves its neighbour nodes in a list. For each node a dictionary entry with the node as key and its neighbour-nodes-list as value is being added to the adjacency list (cf. Figure 11).

```
for n in self.nodes:
    neighbour_nodes = []
    for key_dir in n.neighbours:
        pos = n.neighbours[key_dir]
        node = self.get_node_by_pos(pos)
        neighbour_nodes.append(node)
    self.adjac_list[n] = neighbour_nodes
```

Figure 11 - Set adjacency list function in graph class

Finally, the a*-algorithm can be used on the graph since the graph is fully initialized. The open as well as the closed list consist of a set of nodes. The g-, h- and f-score for each node can be calculated and saved into the specific attributes of the node. The *graph.best_path_solution* that is being returned if a path is found, consist of the position of each best path node. This position is also saved in the node objects attribute.

3.2 Discussion of the result

The result of the test with start_pos = (12, 4) and end_pos = (3, 6) looks very plausible. It takes a path containing a lot of "2" coded cells since the cost of a cell coded with 2 is only 1. It avoids taking a "5" coded cell since its cost is 20.

```
Path Found with total cost of 48.0!
[(12, 4), (12, 5), (12, 6), (12, 7), (12, 8), (12, 9), (12, 10), (11,
10), (10, 10), (9, 10), (8, 10), (7, 10), (6, 10), (6, 9), (6, 8), (6,
7), (6, 6), (6, 5), (5, 5), (4, 5), (4, 6), (3, 6)]
```

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	5	1	3	3	3	5	5	3	3	3	3	3	3	3	0
2	5	1	0	3	3	5	0	3	3	5	5	3	3	0	3
3	5	1	1	0	3	5	0	0	0	3	5	3	3	3	3
4	5	5	1	0	0	5	0	0	3	3	5	0	3	3	3
5	5	5	1	4	1	1	0	0	3	3	5	2	2	2	2
6	0	3	3	0	5	1	1	1	3	5	5	2	0	0	3
7	3	3	5	5	5	0	0	1	1	5	5	2	0	0	0
8	3	3	3	5	3	3	3	3	1	1	5	2	0	0	0
9	3	3	3	0	5	3	3	3	3	1	0	2	0	0	0
10	0	3	3	3	5	2	2	2	2	4	2	2	0	0	0
11	3	3	3	3	5	2	0	3	0	1	0	0	0	0	0
12	0	3	3	5	5	2	0	3	3	1	1	1	0	0	0
13	0	3	5	5	0	2	0	3	3	0	5	1	1	0	0
14	0	0	3	5	5	2	0	0	0	0	5	5	1	1	0
15	2	2	2	2	2	2	0	0	0	0	0	5	5	1	0

Figure 12 – best path solution for start_pos = (12, 4) and end_pos = (3, 6)

3.3 Discussion of the results in test cases

In the following, the results of 4 different test cases are being shown and discussed.

3.3.1 Test case 1

The result of the test with start_pos = (4, 15) and end_pos = (9, 10) looks very plausible. It takes the "2" coded path since the cost of a cell coded with 2 is only 1.

```
Path Found with total cost of 10.0!
[(4, 15), (5, 15), (6, 15), (6, 14), (6, 13), (6, 12), (6, 11), (6, 10),
(7, 10), (8, 10), (9, 10)]
```

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	5	1	3	3	3	5	5	3	3	3	3	3	3	3	0
2	5	1	0	3	3	5	0	3	3	5	5	3	3	0	3
3	5	1	1	0	3	5	0	0	0	3	5	3	3	3	3
4	5	5	1	0	0	5	0	0	3	3	5	0	3	3	3
5	5	5	1	4	1	1	0	0	3	3	5	2	2	2	2
6	0	3	3	0	5	1	1	1	3	5	5	2	0	0	3
7	3	3	5	5	5	0	0	1	1	5	5	2	0	0	0
8	3	3	3	5	3	3	3	3	1	1	5	2	0	0	0
9	3	3	3	0	5	3	3	3	3	1	0	2	0	0	0
10	0	3	3	3	5	2	2	2	2	4	2	2	0	0	0
11	3	3	3	3	5	2	0	3	0	1	0	0	0	0	0
12	0	3	3	5	5	2	0	3	3	1	1	1	0	0	0
13	0	3	5	5	0	2	0	3	3	0	5	1	1	0	0
14	0	0	3	5	5	2	0	0	0	0	5	5	1	1	0
15	2	2	2	2	2	2	0	0	0	0	0	5	5	1	0

Figure 13 - best path solution for start_pos = (4, 15) and end_pos = (9, 10)

3.3.2 Test case 2

The result of the test with start_pos = (1, 1) and end_pos = (1, 1) is accurate. The total cost stays 0 because no movement is happening. The path solution only contains the (1, 1) position.

```
Path Found with total cost of 0!
[(1, 1)]
```

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	5	1	3	3	3	5	5	3	3	3	3	3	3	3	0
2	5	1	0	3	3	5	0	3	3	5	5	3	3	0	3
3	5	1	1	0	3	5	0	0	0	3	5	3	3	3	3
4	5	5	1	0	0	5	0	0	3	3	5	0	3	3	3
5	5	5	1	4	1	1	0	0	3	3	5	2	2	2	2
6	0	3	3	0	5	1	1	1	3	5	5	2	0	0	3
7	3	3	5	5	5	0	0	1	1	5	5	2	0	0	0
8	3	3	3	5	3	3	3	3	1	1	5	2	0	0	0
9	3	3	3	0	5	3	3	3	3	1	0	2	0	0	0
10	0	3	3	3	5	2	2	2	2	4	2	2	0	0	0
11	3	3	3	3	5	2	0	3	0	1	0	0	0	0	0
12	0	3	3	5	5	2	0	3	3	1	1	1	0	0	0
13	0	3	5	5	0	2	0	3	3	0	5	1	1	0	0
14	0	0	3	5	5	2	0	0	0	0	5	5	1	1	0
15	2	2	2	2	2	2	0	0	0	0	0	5	5	1	0

Figure 14 - best path solution for start_pos = (1, 1) and end_pos = (1, 1)

3.3.3 Test case 3

The result of the test with start_pos = (1, 1) and end_pos = (1, 6) looks very plausible. The algorithm doesn't take the "5" coded path since the cost of a cell coded with 5 is 20. It finds a path with a lower value of cost.

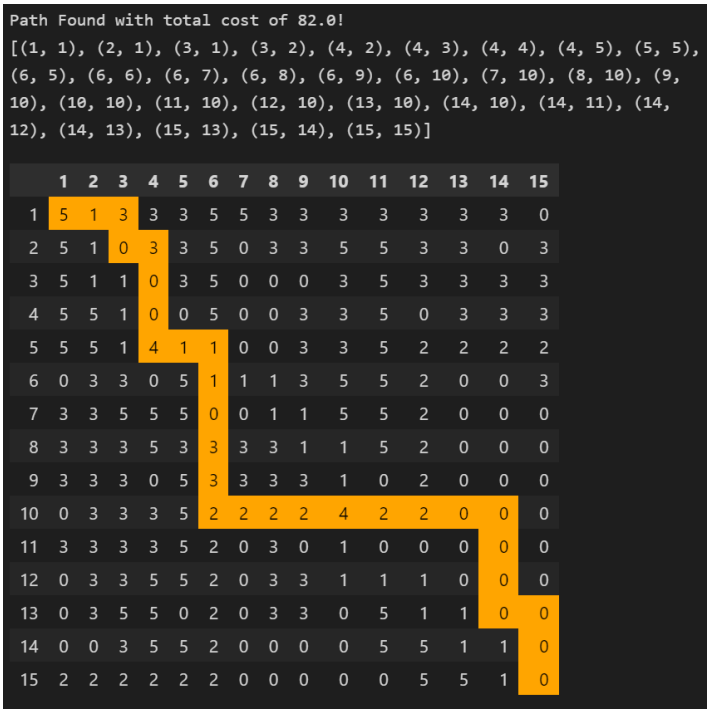
```
Path Found with total cost of 38.0!
[(1, 1), (2, 1), (3, 1), (3, 2), (3, 3), (3, 4), (3, 5), (3, 6), (2, 6),
(1, 6)]
```

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	5	1	3	3	3	5	5	3	3	3	3	3	3	3	0
2	5	1	0	3	3	5	0	3	3	5	5	3	3	0	3
3	5	1	1	0	3	5	0	0	0	3	5	3	3	3	3
4	5	5	1	0	0	5	0	0	3	3	5	0	3	3	3
5	5	5	1	4	1	1	0	0	3	3	5	2	2	2	2
6	0	3	3	0	5	1	1	1	3	5	5	2	0	0	3
7	3	3	5	5	5	0	0	1	1	5	5	2	0	0	0
8	3	3	3	5	3	3	3	3	1	1	5	2	0	0	0
9	3	3	3	0	5	3	3	3	3	1	0	2	0	0	0
10	0	3	3	3	5	2	2	2	2	4	2	2	0	0	0
11	3	3	3	3	5	2	0	3	0	1	0	0	0	0	0
12	0	3	3	5	5	2	0	3	3	1	1	1	0	0	0
13	0	3	5	5	0	2	0	3	3	0	5	1	1	0	0
14	0	0	3	5	5	2	0	0	0	0	5	5	1	1	0
15	2	2	2	2	2	2	0	0	0	0	0	5	5	1	0

Figure 15 - best path solution for start_pos = (1, 1) and end_pos = (1, 6)

3.3.4 Test case 4

The result of the test with start_pos = (1, 1) and end_pos = (15, 15) looks very plausible. The algorithm doesn't move along any "5" coded cells since the cost of a cell coded with 5 is 20. It prefers moving along cells with lower costs. For example the path from (7, 10) to (14, 10) contains a lot of "2" coded cells since the cost of these cells is only 1.



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	5	1	3	3	3	5	5	3	3	3	3	3	3	3	0
2	5	1	0	3	3	5	0	3	3	5	5	3	3	0	3
3	5	1	1	0	3	5	0	0	0	3	5	3	3	3	3
4	5	5	1	0	0	5	0	0	3	3	5	0	3	3	3
5	5	5	1	4	1	1	0	0	3	3	5	2	2	2	2
6	0	3	3	0	5	1	1	1	3	5	5	2	0	0	3
7	3	3	5	5	5	0	0	1	1	5	5	2	0	0	0
8	3	3	3	5	3	3	3	3	1	1	5	2	0	0	0
9	3	3	3	0	5	3	3	3	3	1	0	2	0	0	0
10	0	3	3	3	5	2	2	2	2	4	2	2	0	0	0
11	3	3	3	3	5	2	0	3	0	1	0	0	0	0	0
12	0	3	3	5	5	2	0	3	3	1	1	1	0	0	0
13	0	3	5	5	0	2	0	3	3	0	5	1	1	0	0
14	0	0	3	5	5	2	0	0	0	0	5	5	1	1	0
15	2	2	2	2	2	2	0	0	0	0	0	5	5	1	0

Figure 16 - best path solution for start_pos = (1, 1) and end_pos = (15, 15)

3.3.5 Conclusion

To conclude, the solutions found by the algorithm look very plausible for being the best path solution. It appears to avoid cells with high costs "5" coded cells and to prefer cells with low costs like "2" coded cells.

4 Conclusion

To conclude, the program was developed by creating three classes: grid, node and graph.
The extracted information is passed to the grid object. The graph uses the grid objects information to create nodes that represent each cell of the map and their relevant information.

The a*-algorithm can be applied to the graph object by passing a start- and end position to it. The results of the algorithm are very plausible as the results of certain test cases show.

To solve the problem in the exercise: The best path solution for the given map with the start position start_pos = (12, 4) and the end position end_pos = (3, 6) is shown in Figure 12.

5 Quellverzeichnis

101 Computing. (1. February 2018). Von https://www.101computing.net/a-star-search-algorithm/ abgerufen

Geeks for Geeks. (30. May 2022). Abgerufen am 2022. 12 12 von https://www.geeksforgeeks.org/a-search-algorithm/

Joma Class. (2020. Dezember 5). Abgerufen am 12. Dezember 2022 von https://www.jomaclass.com/blog/graph-representation-edge-list-adjacency-matrix-and-adjacency-lists

Omni Calculator. (kein Datum). Abgerufen am 2022. 12 12 von https://www.omnicalculator.com/math/manhattan-distance

Szabo, F. E. (2015). The Linear Algebra Survival Guide. Academic Press. Von https://www.sciencedirect.com/topics/mathematics/manhattan-distance abgerufen