Assignment 2: Applying the A* search

Håkon Måløy   Xavier F.C. Sánchez-Díaz[*]

28th August 2023

Deadline: 22.09.2023, 23:59 hrs

1 Overview

In this assignment, you will become familiar with the A* algorithm. You'll use it to find the shortest path in a 2D grid-like world.

The assignment consists of **three parts**, the **first two are mandatory** and the **third one is optional**. You need to make **passable effort** on at least the first two parts in order to get this **assignment approved**.

Each section specifies a set of required deliverables. All parts must include **both programming and report writing**. The main purpose of the report is for you to present your results so that the student assistants don't have to run your code to evaluate your assignment.

2 A* Implementation

A common demonstration problem for A* is that of finding shortest paths in 2D square grids. This is usually known as *pathfinding*, and it is important because some of the locations of a grid may not be *walkable*.

In order to solve the problems in this assignment, you first need to code an implementation of the A* algorithm. The AIMA book is a good place to start. Your implementation will need to read a board configuration—a text file describing the obstacles and their locations, as well as the starting point and goal points, and then find the shortest path between them.

- In **Part I**, the game board will only consist of cells that are either walkable or non-walkable.

[*]adapted from the work of Håkon Måløy

- In **Part II**, the cells will have different costs. This means that the shortest path does not depend entirely on the distance, but also on the *type* of each cell.
- In **Part III** (which is optional) the goal will be moving, so you need to consider this movement in your implementation.

In all three parts, you will be asked to visualise your program results. The specifics of the visualisation are up to you, but the board configuration and the calculated path should be clearly visible.

Attached, you will find some Python code for reading the board, loading and visualising it. You can use it as a starting point, or you can work on your own implementation.

3 Part 1 – Grid with obstacles

In Part 1 of this assignment you will take on the (familiar?) role of a student trying to navigate Studenter Samfundet. See Figure 1. We recommend you also take a look at the map on their website.

3.1 Task 1

You and your friend arrived at Samfundet only five minutes ago, but you've already managed to get separated. Being the resourceful person that you are, you call your friend, whom tells you that they went looking for you and is currently located at **Strossa**. Your task is therefore to find the shortest path from **Rundhallen** (your location) to **Strossa** using your implementation of the A* algorithm.

3.2 Task 2

When you arrive at Strossa, your friend is nowhere to be found. Applying your intellect, you deduce that they have probably moved on and you missed them in the stairs. You call your friend again and find out that they are now at **Selskapssiden**. Your task is now to use your A* implementation to find the shortest path from **Strossa** to **Selskapssiden**.

For a given state in the A* search, i.e. a given cell on the board, the successor states will be the adjacent cells to the north, south, west and east that are within the boundaries of the board and that do not contain obstacles. Suggestions for the heuristic function h for this problem are to calculate either the Manhattan distance or the Euclidean distance between the current cell and the goal cell.

In the provided code, you can generate the two scenarios above by instantiating a `Map_Obj` with the corresponding task number i.e.:

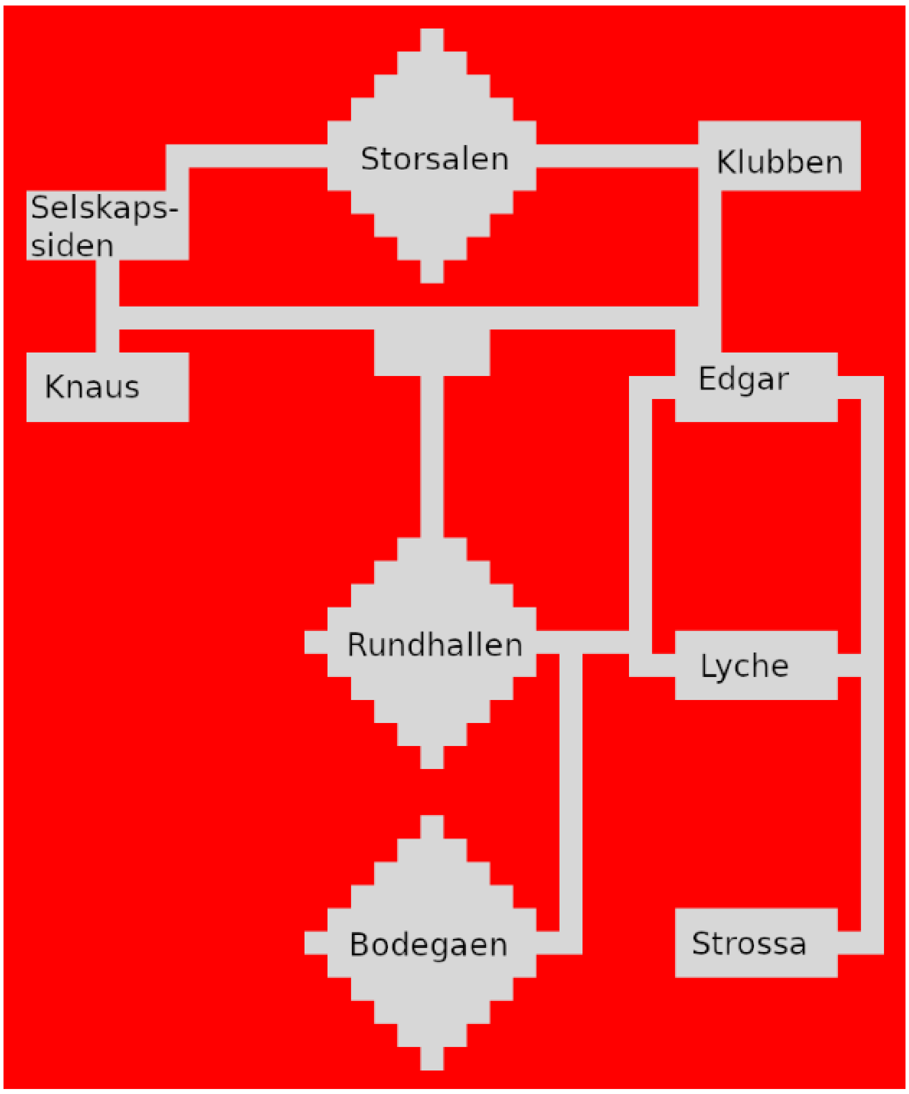

Figure 1: A 2D-world representation of Samfundet.

```
map_obj = Map_Obj(task=1)
```

Further information about the use of the `map_obj` is available in the Appendix.

Deliverables

- Well-documented source code for a program that finds shortest paths for boards with obstacles and that visualises the results. If you did not write the A* implementation yourself, specify where you got the code from.
- Visualisation of the shortest path for the two tasks above.

4 Part 2 – Grids with different costs

Table 1: Cell types and their associated costs.

Char	Description	Cost
.	**Flat ground**	1
,	**Stairs**	2
:	**Crowded Stairs**	3
;	**Crowded room**	4

In Part 1 we modelled Samfundet as if it was a building consisting of a single level. In reality, it is a multi-levelled building with stairs connecting the different rooms. Using stairs is usually more time consuming than walking on flat ground, especially if it is a popular night at Samfundet. To reflect this reality, an additional cost has been added to the stair cells.

In this part of the assignment, your code form Part 1 will be extended to take different cell costs into accounting the path finding process. Table 1 specifies the cell types that should be supported, while Figure 2 shows an example of Samfundet where these cell types are used.

4.1 Task 3

Tonight you are going to a concert at Samfundet. The concert is held at **Klubben** and will start kl. 21. You arrived early to enjoy a Lyche-Burger with some friends before going to the concert. The time is 20:45 and you should get going. The stairs from **Rundhallen** to **Edgar** have become crowded with all the concertgoers arriving. Use your A* implementation to find the path from **Lyche** to **Klubben** with the least cost.

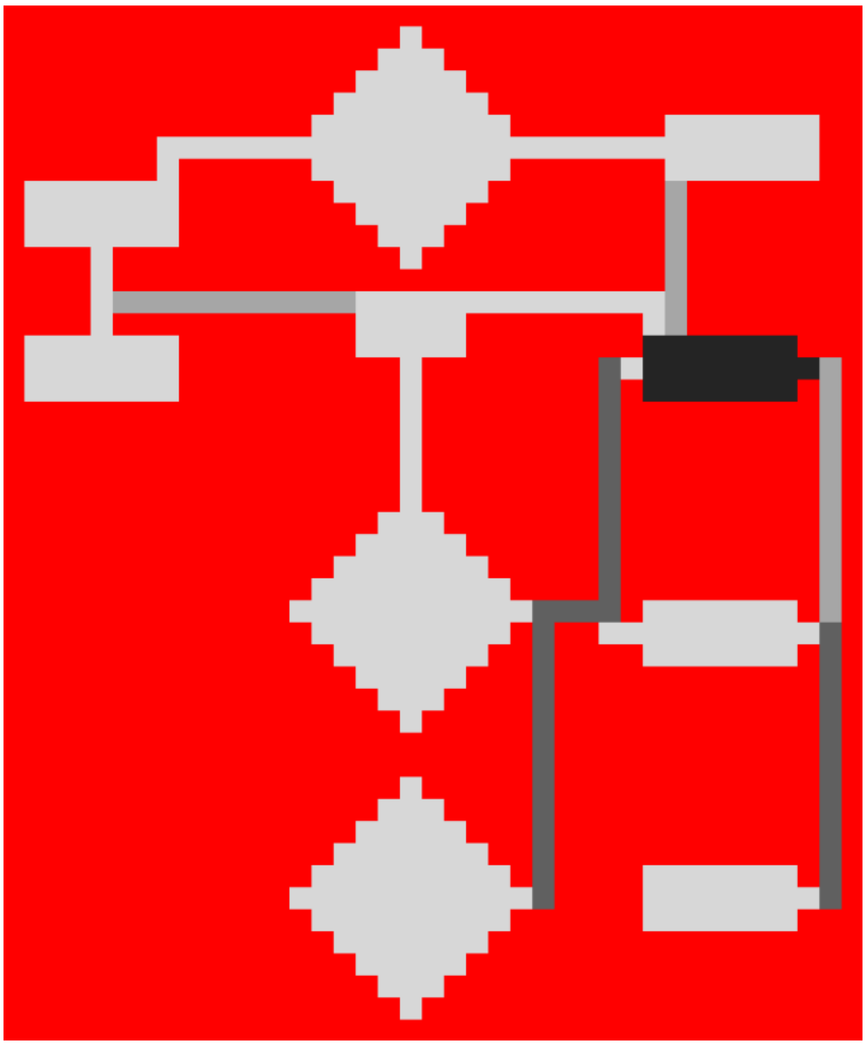

Figure 2: Samfundet modelled with all the different cell types in use.

4.2 Task 4

As you start walking, you remember seeing a poster announcing a free chocolate cake party at **Edgar** this very evening. **Edgar** is therefore filled with hungry students scrambling to eat as much cake as possible. Use your A* implementation to find the new least-cost path from **Lyche** to **Klubben**, now considering the cake party at **Edgar**.

Again, you can generate the two scenarios by instantiating a `Map_Obj` with the corresponding task number.

4.3 Deliverables

- Well-documented source code for a program that finds shortest paths for boards with different cell costs (as specified in Table 1) and that visualises the results.
- Visualisation of the least-cost paths for the two tasks above.

5 Part 3 – Moving Goal (Optional)

In the previous parts, we have only used static goals. In this final part we will use a moving goal. Your algorithm will therefore need to take this movement into account. We have limited the movement of our goal to a straight line, with a constant speed of 25% of your speed and a constant direction. You can therefore use this information to make some assumptions when designing your heuristic function h.

5.1 Task 5

After having finished the concert you feel the need for sugar! You rush down to **Edgar** to get some free chocolate cake. After having satisfied your sugar cravings, you notice that you have separated from your friend again. You head to the top of **Rytterhallen** and call to find that they are still at **Klubben**. Having had one beer too many, they tell you that they will be heading towards **Selskapssiden** and can meet you outside. Having just consumed four chocolate cakes, you experience a sugar high and you quickly estimate that with your sugar high and your friend's reduced balance, you can move a four times their speed. Use this information along with the knowledge that your friend will be starting out at **Klubben** and moving towards **Selskapssiden** to improve your A* implementation to handle a moving target.

A `tick()` function is provided with the code in `Map_Obj`. This function moves the goal one step towards **Selskapssiden** every fourth call. You should call this function every iteration of your A* algorithm for the goal to move properly.

5.2 Deliverables

- Well-documented source code for a program that finds the least-cost path for the board with a moving goal.
- Visualisation of the least-cost path.

Appendix: Code description

This appendix contains information about the source code accompanying this assignment, which is written in Python 3. To run it, you need the `pandas`, `numpy` and `Pillow` (previously `PIL`) packages. For this, we recommend using a Python environment manager (like `venv` or `conda`).

The source code contains one class, `Map_Obj`, which provides functions for reading the `CSV` files containing the maps, placing the start and goal positions, moving goal positions as well as some code for printing the maps.

To start one of the tasks, instantiate a Map_Obj with the corresponding task number i.e.:

```
map_obj = Map_Obj(task=1)
```

You can then call `map_obj.get_maps()` to receive an integer and a string representation of the current map. The integer map contains the cost of each cell, while the string map can for example be used for printing.

The `map_obj.show_map()` function can be called to show the current version of the string map in a human-readable format. You can also `print_map(map)` as well.

There are a bunch of other important methods and attributes you can use. We recommend you take a look at the code documentation where you can get a more detailed description of the methods provided and the type of data they use.

You are free to modify the code as you wish.