Practice 1: Navigation

Part 2: Informed search
Artificial Intelligence
2021–2022
Universitat Autònoma de Barcelona

1 Introduction

During the previous practice we saw how to work with the Path class, implementing two methods of Uninformed Search (Width search and Depth search) to find a route between two points. These two methods are not optimal, as they do not consider the costs and simply look for the first path to the solution without considering any cost, and without considering any kind of heuristic that can help reduce the cost of the search.

In this second part of the project we will introduce the concepts of **cost** and **heuristics** to search algorithms, that will allow us to find optimal solutions and also trying to get there as quickly as possible.

2 Needed files

We will continue working with the same files as in Part 1 of the practice, and again everything you will program will be done on the `SearchAlgorithm.py` file where all the functions of part 1 of the project were already programmed.

3 Preliminaries

Before starting to program it is highly recommended to understand the concepts that we will work on in this part of the practice: Cost and Heuristics. In case of doubt, you can find an explanation of how they can be calculated in the notes of the theory. Therefore, before proceeding, you should be able to:

1. Understand the concept of Cost and how to calculate it with the information you have in the Map class, which we will represent by the function g.

See how it can be extended to different criteria (stops, time, distance, transshipments). To use the different criteria, it is important to keep the following informations in mind:

- The time between all the stops on all the lines is given.
- Each line has a given constant speed.
- Each station has given coordinates that are shared by the different lines of the same station (we will assume that a transfer does not involve a distance cost)
- Roads between stations do not always go in a straight line.

2. Understand the concept of Heuristics as an estimator of the cost that is left from a given station to the solution, which is unknown. As we have seen in theory, heuristics will be represented by the function h. This heuristic function will depend on the cost you estimate, we will have to define a heuristic for each criterion we want to optimize, be it time, transfers or distance. Therefore, each g has its own h.

4 What do we have to code?

In the second part of the practice, you will program two Search algorithms: Uniform Cost Search and A* Search. Both algorithms perform optimal search using the cost of the paths, and A* also uses heuristics to improve the time it takes to find the solution.

In this Part 2 of the project we will use the same functions `Expand` and `Remove_cycles` that we implemented in Part 1 of the project. Cost calculations and path heuristics will be applied to the path lists resulting from these two functions. We will first program the Uniform Cost Search functions and then program the A*.

4.1 Uniform Cost search

We have to code the following three functions:

`Calculate_cost`: Function that takes in input the list of child Paths, the map (Map), and a number (int) that refers to the criteria we will calculate (stops, time, distance, transfers). Calculate the cost from the penultimate station we were exploring to the last (current child) and add it to what you already had. It does this for each of the Paths children in the list, and updates the total cost value for each path.

EXAMPLE:

Input: `expand([14,13,8,12,g=15]),MAP)`
Output: `[[14,13,8,12,8,g=15],[14,13,8,12,11,g=15],[14,13,8,12,13,g=15]]`
Input: `calculate_cost([[14,13,8,12,8,g=15],[14,13,8,12,11,g=15],`

`[14,13,8,12,13,g=15]],1)`
Output: `[[14,13,8,12,8,g=17],[14,13,8,12,11,g=21],[14,13,8,12,13,g=20]]`

*** Hint: Use the function "`update_`g" in the class Path.

`Insert_cost`: Function that takes as input the list of child Paths that we have expanded and the global list of Paths explored from the tree. The function returns the union of these two lists ordered by cost, so that the path of best cost is ahead of everything, as this will be the next path we will expand.

`Uniform_cost_search`: Function that given a source station, a destination station, the city map and the type of cost we want to evaluate represented by a number (int), returns an optimal route between the two stations implementing the search algorithm of uniform cost given in Figure 1.

```
Function Uniform_Cost_SEARCH(RootNode,GoalNode)

1. List=[[RootNode]];
2. Until (Head(Head(List))=GoalNode)OR (List=NIL)do
    a) C=Head(List);
    b) E=Expand(C);
    c) E=RemoveCycles(E);
    d) List=Ordered_Insertion_g(E,Tail(List));
3. EndUntil;
4. If (List<>NIL)Return(Head(List));
5. Else Return("No solution exists");
EndFunction
```

Figure 1: Pseudocode for the Uniform Cost Search

4.2 A* search

For this algorithm, we must first program two functions that will implement the use of heuristics, a function that will eliminate redundant paths (that is, non-optimal partial paths), and finally two functions that will implement Search A* itself. We detail it below.

Functions to handle the heuristic

`Calculate_heuristics`: Function that takes as input the list of Paths children, the map (Map), the ID of the destination station and a number (int) that refers to the criterion that tries to estimate the heuristic (stops, time, distance , tran- shipments). Calculate the heuristic between the last stop we are exploring for each child path and the end station. Once calculated, it updates the heuristic value of each child path.

(*** Hint: Use Path class function "`update_h`")

`Update_f`: Function that takes as input the list of Path children, to which we have previously updated the cost and heuristic, and returns the same list with the updated total cost for each child path.

(*** Hint: Use Path class function "`update_f`")

Function that eliminates redundant paths

During navigation we can find that we arrive at the same station using different paths. To optimize our search, we need to stop exploring any path that leads to a station we've explored before at a better partial cost. A non-optimal partial path will be a redundant path that will never be part of an optimal solution. Therefore, it is necessary to define the function `Remove_redundant_paths` which will be responsible for removing these redundant paths.

In order to implement this function we need to save at all times what is the optimal cost to get to each station. We will need to create a dictionary, `visited_stations_cost`, which will store the information of the stations visited and the minimum cost to them at any given time.

`Remove_redundant_paths`: The function takes as input the list of child Paths we just expanded, the global list of Paths of the tree explored, and the dictionary of partial costs. The function is responsible for checking if one of the new child Paths has reached a node that we had already explored before and for which we keep its cost in the dictionary, if so it will check if the new cost is better or worse than the previous one:

- If the previous cost is better or equal, we will remove the new path from the list of Paths children.
- If the previous cost is worse, we will put the new cost in the dictionary, and we will eliminate those paths that contained this node, since all the paths reached that node in a suboptimal way.

The function must return two lists and a dictionary. The first list is that of the child Paths without redundant paths, the second list is the global list of Paths explored, also without redundant paths, and finally the dictionary of all the nodes visited with the updated optimal costs.

Functions to implement the A* search

`Insert_cost_f`: Function that takes as input the list of child Paths that we have expanded, and the global list of Paths explored from the tree. The function returns the union of these two lists ordered according to the estimated total cost ($f = g + h$), so that the path of the best estimated total cost is in front of everything, as this will be the next one that we will expand.

`A_star`: Function that given the COORDINATES of a position of origin, a position of destination, the metro map of a city and the type of criterion we want to optimize, which is represented by a number (int), search an optimal route between the two points using the A* algorithm specified in Figure 2.

```
Function A*_SEARCH(RootNode,GoalNode)

1. List=[[RootNode]];
2. Until (Head(Head(List))=GoalNode)OR (List=NIL)do
    a) C=Head(List);
    b) E=Expand(C);
    c) E=RemoveCycles(E);
    d) List=Sorted_Insertion_f(E,Tail(List));
3. EndUntil;
4. If (List<>NIL)Return(Head(List));
5. Else Return("No solution exists");
EndFunction
```

Figure 2: Pseudocode for the A* Search

5 Additional functions

Here below we give the specifications for performing the additional functions:

Distance heuristics

`calculate_heuristics`: modify this function by `type_preference = 2`
`calculate_cost`: change `type_preference = 2`

Stop heuristics

`calculate_heuristics`: modify this function by `type_preference = 0`
`calculate_cost`: change `type_preference = 0`

Transhipment heuristics

`calculate_heuristics`: modify this function by `type_preference = 3`
`calculate_cost`: change `type_preference = 3`

5.1 Giving coordinates to A*

The function we used to apply A* from a given origin in coordinates is based on going to the nearest station. However, the nearest station may not be the best station to reach our destination. See the example in Figure 3. This part of the project aims at implementing the A* search method, improving the problem seen in the example above. It is suggested that given the initial and final coordinates, the A* search will find the best route for any of the criteria, taking into account that the user can navigate to the most convenient station. The user is considered to be approaching this station by walking in a straight line at a speed of 5.

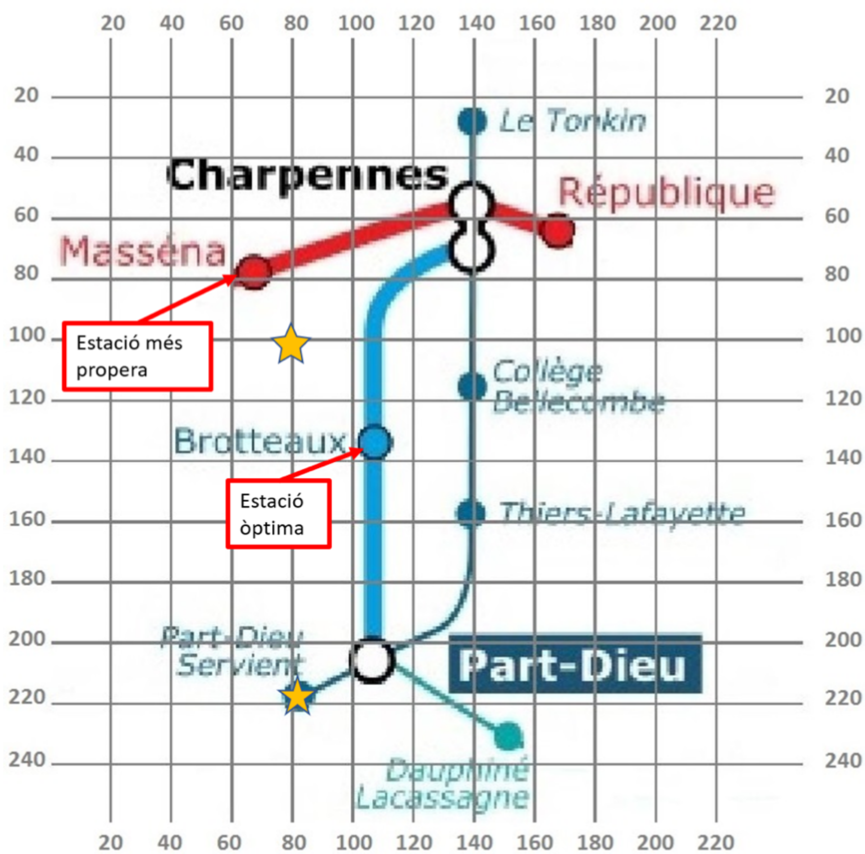

Figure 3: Example of two possible routes between the point of origin $(80, 100)$ and the point of destination $(80, 220)$. Route 1: Go to the nearest station (Masséna), and make 6 stops counting the transfers. Route 2: Go to Brotteaux station, and make only 3 stops counting the transfers.

To program it you will need to create a new function called `Astar_improved` to program it. In the event that you implement this A* feature, you must indicate this to the internship teacher for correction. Since this part is not corrected automatically.

6 Document submission

To evaluate this first part of the practice you will need to upload to Moodle your `SearchAlgorithm.py` file which should contain your NIA in the authors variable and your group in the variable group (at the beginning of the file). Delivery must be made before the day **20/03/2022**

Warning! It is important that you consider the following points:

1. Code correction is done automatically, so be sure to upload the files with the correct nomenclature and format.
2. The code is subject to automatic plagiarism detection during correction.
3. Any part of the code that is in the functions of the `SearchAlgorithm.py` file cannot be evaluated, so do not modify anything outside of it.
4. To prevent code from looping, there is a time limit for each exercise, so if your functions take too long it will count as an error.