Heuristic_search

April 13, 2021

1 Heuristic and Greedy Search Algorithms

Author: Jessica Cervi
Expected time = 2 hours
Total points = 80 points

1.1 Assignment Overview

In this assignment, we will look at some types of search algorithms. In the first part of the assignment, we will focus on a very popular uniform seach algorithm, the A-star (A*) algorithm. After reiterating the foundations and the theory behind this algorithm, we will solve a simple problem using the `Python` library `pathfinding`. Next, you will be asked to solve a problem using the A** algorithm by hand. We will work on the A* algorithm using the library `networkx`.

The second part of the assignment focuses on another type of algorithm, the Greedy Search algorithm. After re-iterating the basics of this algorithm, you will be asked to solve a simple problem from scratch. The second part of the assignment focuses on another type of algorithm, the Greedy Search algorithm. After reiterating the basics of this algorithm, you will be asked to solve a simple problem from scratch.

This assignment is designed to build your familiarity and comfort coding in Python while also helping you review key topics from each module. As you progress through the assignment, answers will get increasingly complex. It is important that you adopt a data scientist's mindset when completing this assignment. **Remember to run your code from each cell before submitting your assignment.** Running your code beforehand will notify you of errors and give you a chance to fix your errors before submitting. You should view your Vocareum submission as if you are delivering a final project to your manager or client.

Vocareum Tips - Do not add arguments or options to functions unless you are specifically asked to. This will cause an error in Vocareum. - Do not use a library unless you are explicitly asked to in the question. - You can download the Grading Report after submitting the assignment. This will include feedback and hints on incorrect questions.

1.1.1 Learning Objectives

- Use search algorithms such as A* search and A* optimality
- Learn how to implement the A* algorithm using the `pathfinding` and `networkx` library
- Compute the shortest path using the A* algorithm by hand
- Learn the basics of greedy algorithms and implement various problems in Python

1.2 Index:

1.3 Heuristic and Greedy Search Algorithms

1.3.1 Informed Search Algorithms

Informed search algorithms have information on the goal state, which helps in more efficient searching. This information is obtained by something called a **heuristic**.

In an informed search, a heuristic is a function that estimates how close a state is to the goal state. Some examples are Manhattan distance, Euclidean distance, etc. Of course, different heuristics are used in different informed algorithms.

A-star (also referred to as A*) is one of the most successful search algorithms to find the shortest path between nodes or graphs. It is an informed search algorithm, as it uses information about path cost and also uses heuristics to find the solution.

In the first part of this assignment, we will focus on learning how to implement an A* algorithm using the library `pathfinder`. In the second part of the assignment, we will focus on how to build A* search algorithm from scratch using a simple Python code. Finally, we will conclude the section about Informed Search Algorithms by utilizing the library `networkx` and its implemetation of the A* algorithm.

The theory behind the A* algorithm

A* achieve **optimality** and **completeness**, two valuable properties of search algorithms.

When a search algorithm has the property of optimality, it means it is guaranteed to find the best possible solution. When a search algorithm has the property of completeness, it means that if a solution to a given problem exists, the algorithm is guaranteed to find it.

Now to understand how A* works, first we need to understand a few terminologies:

- **Node** (also called State) — All potential positions or stops with a unique identification
- **Transition** — The act of moving between states or nodes
- **Starting Node** — Where to start searching
- **Goal Node** — The target to stop searching
- **Search Space** — A collection of nodes, like all board positions of a board game
- **Cost** — Numerical value (say distance, time, or financial expense) for the path from a node to another node
- **g(n)** — The exact cost of the path from the starting node to any node n
- **h(n)** — The heuristic estimated cost from node n to the goal node
- **f(n)** — Lowest cost in the neighboring node n

Each time A* enters a node, it calculates the cost, f(n)(n being the neighboring node), to travel to all of the neighboring nodes, and then enters the node with the lowest value of f(n). We calculate these values using the following formula:

$$f(n) = g(n) + h(n)$$

We begin by importing the necesserary modules and function from the library `pathfinding`:

```
In [2]: from pathfinding.core.diagonal_movement import DiagonalMovement
        from pathfinding.core.grid import Grid
        from pathfinding.finder.a_star import AStarFinder
```

The first thing we need to define is the *maze* we want to work on.

In the simplest case, this can be defined in Python using a list of lists.

For example, assume that we wanted to find the shortest path from one point to another in this maze:

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$$

then we would need to define our maze in Python in the following way:

```
In [3]: my_maze = [[1,1], [1,0]]
```

Entries meaning Any value smaller or equal to 0 describes an **obstacle**. Any number bigger than 0 describes the weight of a field that can be walked on. The bigger the number the higher the cost to walk that field.

In the next questions, we want the algorithm to create a path from the upper left to the bottom right.

To make it not too easy for the algorithm, we added an obstacle in the middle, so that it cannot use the direct way. We ignore the weight for now, and all fields will have the same cost of 1.

Section **??**

1.3.2 Question 1:

5 points

$$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{pmatrix}$$

From the image above, construct a list of lists with the entries of the matrix above. Assign the result to `ans_1`.

```
In [4]: ### GRADED

        ### YOUR SOLUTION HERE
        ans_1 = [[1, 1, 1], [1, 0, 1], [1, 1, 1]]
        ###
        ### YOUR CODE HERE
        ###
```

```
In [5]: ###
        ### AUTOGRADER TEST - DO NOT REMOVE
        ###
```

Next, we need to create a new grid from this map representation using the function `grid()` from `pathfinding` and by massing the name of our map to it. This will create node instances for every element of our map, and it will also set the size of the map. For now, we assume that your map is a square, so the size height is defined by the length of the outer list and the width by the length of the first list inside it.

```
In [6]: grid = Grid(matrix=ans_1)
```

Next, we need to define the start and end points of our path.

In `Python`, the entries on a regular grid are numbered using a (x, y)-type numbering, where x denotes the row number and y denotes the columns.

Note that both x and y start at 0 and they follow a top-to-bottom and left-to-right order, respectively.

For example, if you wanted to define a point with coordinate (1,2) as a starting point, you would write:

```
In [7]: my_start = grid.node(1,2)
```

Section **??**

1.3.3 Question 2:

5 points

Our goal is to find the shortest path from the top-left corner to endpoint (bottom-right) from the map.

Define the start point as `start` and the end point as `end`.

```
In [8]: ### GRADED

        ### YOUR SOLUTION HERE
        start = grid.node(0,0)
        end = grid.node(2,2)

        ###
        ### YOUR CODE HERE
        ###
```

```
In [9]: ###
        ### AUTOGRADER TEST - DO NOT REMOVE
        ###
```

Next, we need to create a new instance of our finder and let it do its work.

This can done by invoking the function `AStarFinder`.

Notice that by default this instance does not allow for diagonal movement in our search (only left, right, top, and bottom).

In the code cell below, we create a new instance for your path finder and assign it to the variable `finder`. Notice that we make sure to allow for diagonal movement and search by setting the argument `diagonal_movement` equal to `DiagonalMovement.always`