Assignment 2

March 4, 2021

INTRODUCTION

Our next challenge is to enable our robot to navigate in the supermarket to find multiple products. We will use a different method for this than RL. The supermarket will be modeled as a labyrinth (further referred to as 'maze'), just like in assignment 1. In order to save time and energy, your bot should determine the fastest route, given a set of locations to visit within the supermarket. This problem is usually referred to as the *Traveling Salesmen Problem* (TSP) and can be solved with a *Genetic Algorithm*.

Before it can determine the fastest way to visit all locations, it should have an idea of what the distance is between these locations. To achieve this, you will implement *Swarm Intelligence*. Specifically, path finding based on *Ant Colony Optimization* (ACO).

Hence this assignment is split into 2 sub-problems: path finding and a TSP. You will hand-in code individual for each part, but will have to hand in a single report answering all the questions. You can work through this assignment in the order detailed below.

We provide you with a template for the solution in Python and Java. Information about the template is included in Appendix A

1.1. Part 1: Path Finding Through Ant Colony Optimization

In the first part of this assignment you will teach your robot how to move through a maze and implement a method for finding a quick route between a specified start and end point. The goal of the assignment is to use the methods learned in this course to tackle the problem, hence you are not allowed to use search algorithms like Dijkstra or A*.

What Should You Hand-In

You should hand-in the following files:

- The source code of your (Part 1) program as a compressed archive, with a name of the following format:

  <group>_code_ACO.zip.
  Remember to keep your code 'clean', use proper indentation, useful commentary, logical variable names etc. - you will need this code for part 2.

- Route File for each Grading Maze

  <group>_<maze>.txt.
  For example, "3_easy.txt".

- Report in PDF:

  <group>_report.pdf.
  Containing your answers to the questions*.

*Questions are numbered items. Bulleted items are tasks. You should perform these tasks, but you need not mention them in your report. Note that you will hand in one single report answering the questions from both part 1 and 2 of this assignment.

Deadlines: The deadline for the report and the action files is Friday **19th of March at 23:59**.

Maze Visualizer

For this assignment, you can use the maze visualizer to create and test mazes. You can obtain this visualizer from blackboard. The application is written in Java and can be used to visualize the routes computed. This way, you can check whether your answer is reasonable.

The application consists of two different modes: 'editor' and 'visualizer'. The first is used to edit/create mazes yourself. The latter can be used to visualize your routes in a given maze.

Grading Mazes

Besides creating, analyzing and testing mazes yourself, you will need to run your code on each of our grading mazes. These mazes are available on Brightspace. There are 4 mazes

available; *Easy, Medium, Hard* & *Insane*. Note that it's not mandated to solve the insane maze, but can be used to show-off your code.

The mazes are encoded as a matrix of 1's (accessible) and 0's (inaccessible). Your robot is allowed to walk single discrete steps within this matrix, however only the tiles that are accessible. The maze files start with two numbers which indicate the width and height of the matrix, this is added for your convenience (a 10 by 20 matrix (10 rows, 20 columns) is written as '20 10'). You can open the maze files in any text editor (they are ASCII encoded) to see what they look like.

```
20 10
1 0 1 1 1 1 1 1 1 1 0 0 0 1 0 1 0 1 1 1
1 0 1 1 1 1 0 1 0 1 1 1 0 1 0 1 0 1 0 1
1 0 1 1 1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
1 1 1 0 0 1 0 1 0 1 0 1 0 1 1 1 1 1 0 1
0 0 0 0 0 1 1 1 0 1 0 1 1 1 0 1 0 1 1 1
1 1 1 1 1 1 0 1 0 1 0 0 1 0 0 1 0 0 0 1
1 0 1 0 1 0 0 1 0 1 0 0 1 0 0 1 0 1 1 1
1 1 1 0 1 0 0 1 1 1 0 1 1 0 1 1 1 1 0 1
0 1 0 0 1 1 1 1 0 1 1 1 0 0 1 0 0 1 0 1
0 1 1 1 1 0 0 1 1 1 0 1 1 1 1 0 0 1 1 1
```

Accompanied with each grading maze is a coordinate file. This file consists of 4 numbers, i.e. 2 coordinates. The first two integers correspond to the starting position of the maze: "x, y;". Please note that these are x- and y-coordinates, not a row & column number! The second two numbers correspond to the ending point of the maze. Hence for a 50x50 maze, starting at the top left and finishing in the bottom right, the coordinate file would read:

```
0, 0;
49,49;
```

1.1.1. ROUTE SYNTAX FOR PART 1

You need to encode your results (i.e. routes) by the route length, starting location, and a sequence of 'actions'.

The header of the file, the total route length, is stored as an integer. This value is equal to the number of actions. A ';' should follow this integer. The starting point should be given by two integers: the x-coordinate and the y-coordinate, separated by a comma (,). A semicolon (';') should follow the y-coordinate.

There are four possible actions: step East, North, West or South. Those four actions are encoded using an integer;

- 0 = East
- 1 = North

- 2 = West
- 3 = South

Within the actions encoding, all characters other than $\{0,1,2,3\}$ are invalid. Again, each action should be followed by a semicolon. You can test the validity of the route file with the visualizer. The provided templates will put the files in the right format for you.

1.1.2. ANT PREPARATION: OBSERVE THE PROBLEM

Before you go ahead and implement the Ant Algorithm you learned in class, you should stop and think for a second: “What kind of features/problems can the maze contain and which should we take into account when implementing the algorithm?”.

1. Make a list of the features you can expect a maze might have. Features that increase the difficulty of 'finding the finish'. Features that will require creative solutions. For example; loops. Name at least 2 other features.
2. Give an equation for the amount of pheromone dropped by the ants. Answer the questions "Why do we drop pheromone?" and "What is the purpose of the algorithm?".
3. Give an equation for the evaporation. How much pheromone will vaporise every iteration? This equation should contain variables which you can use to optimize your algorithm. What is the purpose of pheromone evaporation?

1.1.3. IMPLEMENTING SWARM INTELLIGENCE

Now, implement the standard ant algorithm. The output of your algorithm should be a route between the given start and end point. Section 1.1 describes the syntax your route should use when you hand in your results, you may want to take this into account prematurely.

Your algorithm should satisfy a few conditions:

I. At least the following parameters should be kept variable (i.e., easily modifiable in e.g. the header of your code):
   a) Maximum number of iterations (to prevent infinite loops)
   b) Number of ants patrolling in each iteration
   c) Amount of pheromone dropped by ants
   d) Evaporation parameters
   e) Convergence criterion

II. The beginning- and ending point of the route should be a variable. Your code should be able to find a short route between two random points.

At this point it is sufficient to assume that the maze does not have any of the hard features you thought of in the first question. Just implement a standard Ant Algorithm. You will update your algorithm later to deal with the harder features, but its very useful to be able to test how the ants behave in advance. Still, its good programming practice to have the future upgrades in the back of your mind while writing your standard framework to facilitate later adjustments.

4. Give a short pseudo-code of your ant-algorithm at this stage. If you added any extra functionality to the normal algorithm, please mention and explain them briefly.

1.1.4. UPGRADING YOUR ANTS WITH INTELLIGENCE

You will now adapt the ant algorithm to deal with features like open areas and whatever other tricky features you came up with. How you do this, is totally up to you. It is a very good idea to create small test mazes to see how your code behaves when these features occur.

5. Improve the ant algorithm using your own insight. Describe how you improved the standard algorithm; which problems are you tackling and how? Limit yourself to 1 A4 text (excluding aiding figures).

1.1.5. PARAMETER OPTIMIZATION

We have provided 3 different grading mazes for you to work with; easy, medium and hard. You are expected to run your code on each of these mazes to see how your code behaves for an increasing maze complexity. While doing so, you are expected to tweak the algorithm parameters, such that your code may converge quickly:

The optimal set of parameters (evaporation constant, amount of pheromone dropped, etc.) depends strongly on the precise shape of the maze. A bigger maze will typically have different optimal parameters, but so will a maze with other features (e.g., many open areas).

6. Your task is to find a decent set of parameters for each of the grading mazes. You may do so by varying the parameters and subsequently running your algorithm. If your algorithm converges fast to a good route, your parameters are decent. What is 'converging fast'? Figure that out by varying the parameters.
   Report your tactic upon how to vary the parameters. Assist your text with graphs showing relationships between the parameters and the speed of convergence. Limit yourself to $\frac{1}{2}$A4 text (excluding aiding graphs - and we like nice informative graphs, so use them!).

7. Using your answer to the previous question, can you say something about the dependency of the parameters on the maze size / complexity? Aim at $\approx \frac{1}{4}$A4.

1.1.6. THE FINAL ROUTE

Run your code using your decent set of parameters on each of the grading mazes. Output your route as described in Section 1.1. See also "What you need to hand-in" at the beginning of this document.

1.2. PART 2: THE TRAVELING ROBOT PROBLEM

In this part of the assignment you will be tackling a modified version of a classic AI problem, commonly known as The Traveling Salesmen Problem (TSP).

As you can imagine (and probably from own experience), there are a lot of possible routes to take through a supermarket if you want to pick up several products. For 3 products, there would be $3 \times 2 \times 1 = 6$ possible routes. This number increases exponentially. For 20 products, we already have $20! = 2.4 \times 10^8$ possible routes.

Clearly, just trying out all these routes (brute force) to find the fastest would take too much computational time. Instead, you will be implementing an intelligent way of converging towards an optimal path.

`http://en.wikipedia.org/wiki/Travelling_salesman_problem`

WHAT WILL YOU NEED

On Brightspace you will find the data you will need for this assignment:

- A maze, provided on Brightspace, in the same syntax as in Part 1
- The x,y-coordinates of a list of products, which your robot must go and pick up.
- Optional: The route visualizer from Brightspace.

WHAT SHOULD YOU HAND IN

You will need to hand in the following things:

- Code (archived):

  "*<groupnumber>_Code_TSP*".

- Final Action-file (see Route Syntax section below):

  "*<groupnumber>_actions_TSP.txt*"

- a Report in PDF:

  "*<groupnumber>_report*". (same file as part 1)

Deadlines: The deadline for the report and the action files is Friday **19th of March at 23:59**.

ROUTE SYNTAX PART 2

The syntax of your results is largely the same as in Part 1. Only here off course your route will be longer, and you'll have to indicate when a product is being picked up. This is done by inserting the line: *take product #<Product Number>;* in between actions. Part of a result file might thus look like:

```
83;
0, 1;
3;
3;
0;
take product #6;
3;
2;
2;
take product #15;
3;
1;
1;
1;
take product #4;
1;
0;
(...)
```

The provided templates contain export functions to get the appropriate file format needed.

PROBLEM ANALYSIS

To get a good understanding of the nature of our problem, start by analyzing its characteristics. Your robot will have to pick up a number of different products from all over the supermarket, which can be found at certain locations.

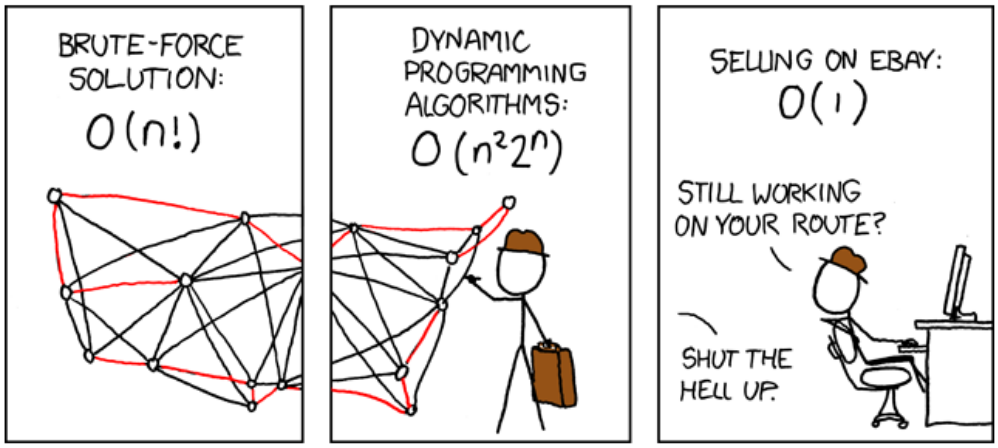
Think about the following questions and answer them in your report.

8. Define the regular TSP problem.
9. In a classic TSP, all cities (nodes) are singularly connected to all other nodes and relative distances are known and symmetric (a weighted complete graph). How is our problem different?
10. Why are Computational Intelligence techniques an appropriate tool for solving a TSP?

A GENETIC ALGORITHM

A classical method of solving a TSP are genetic algorithms. This technique is part of the Evolutionary Computation methods. Biological evolution is sometimes described as non-random selection through random gene-mutations. Genetic Algorithms could be described using this same phraseology: random search through non-random selection of the solution space. Since the solution-space is too big to go through all the possible outcomes, we need a method of directing us towards viable.

11. What do the genes represent and how will you encode your chromosomes?
12. Which fitness function will you use?
13. How are parents selected from the population?
14. The functions of your genetic operations (e.g. cross-over, mutation).
15. How do you prevent points being visited twice?
16. How do you prevent local minima?
17. What is elitism? Have you applied it?



source: http://xkcd.com/399/

A. INSTRUCTIONS FOR TEMPLATE

There is a Java and Python template available to write your code in. The description below describes the Java template, but both templates have the same functionality.
The Java template consists a number of Java classes. There are a number of stubs/nonimplemented functions that you will need to implement to solve the assignment. You can create extra functions/modify existing functions as you see fit.
You can import the Python project in an IDE like PyCharm, Spyder or Eclipse + PyDev. The classes AntColonyOptimization, GeneticAlgorithm and TSPData have a main function that can be run.

To import the Java project in Eclipse:

1. Select Import under File.
2. Select "Existing Projects into Workspace" under General.
3. Either select the archive from Brightspace, or the root directory of the unzipped files.
4. Press Finish.
5. You should now have a project with a `src` and `data` folder, where the `src` folder contains a number of Java classes.
6. You will need to add three run configurations. This allows you to select different classes in the same project to be run. This can be edited in "Run Configurations" under Run. Select "Java Application"
7. Double click "Java Application" to generate a new configuration.
8. Rename your new configuration by entering a different value under the "Name" field. Then select the main class which is AntColonyOptimization for the first assignment.
9. Repeat this and create new configurations for the other entry points[1] for the later parts of the assignment (TSPData and GeneticAlgorithm).
10. The last highlighted run configurations will be run when you press the run button in the "Run Configuration" menu or the 'Play' Button in the main window in Eclipse.

For IDE's other than Eclipse, similar import functionality exist, or one could start a new project in the root folder.

We will start with an overview of the classes you will need for part 1.

[1] The class that you "run". In Java this can be any class that contains a `public static void main(String[] args)` method.

A.1. PART 1

The entry point for part 1 is AntColonyOptimization. All of the I/O has been implemented and you should only have to worry about the algorithmic side of the problem.

Some information per class:

Ant represents an ant finding a path through the maze. You will need to implement logic how an ant finds a route through the maze.

AntColonyOptimization driver function that finds an optimal route between two points. Allows you to tune the parameters of the algorithm. You will need to implement the creation of a maze, the ants to allow them to run through the maze.

Coordinate represents a coordinate. All functions are implemented.

Direction represents the directions an Ant can take. All functions are implemented.

Maze represents a maze. You will need to implement the initialization, addition and reset of the pheromones. Besides this you will need to implement a function that returns a function that returns SurroundingPheromone for a given coordinate.

PathSpecification A class representing the pair of coordinates that indicate the start and end points of a route. Used as initialization for shortest paths/ant colony optimzation.

Route represents a route. All functions are implemented.

SurroundingPheromone represents the surrounding pheromone for a given coordinate in the maze. All functions are implemented.

A.2. PART 2

For part 2 you will need to first generate a class containing all the shortest routes between all product combinations. Since this can take a while to generate it is a good idea to persist this to a file.

This file will function as the starting point for your genetic algorithm which will optimise the routes between the products.

Some information per class:

GeneticAlgorithm driver function for the Genetic Algorithm. Takes a file that contains a 2D-matrix containing the routes you found and attempts to optimise those. You will need to implement the genetic algorithm itself.

TSPData builds and stores the data for all the shortest paths between the products. Uses AntColonyOptimzation to fill the matrix. You will only have to tune the parameters. Can write the final action file given a solution to the TSP problem.