Solution model and code-base

Marcos Ferrer Zalve (LTF)

February 28, 2025

Contents

1 Introduction

The following document aims to explain the model created for the proposed problem, as well as the algorithm developed to find a solution, the tests defined for the code-base and other code and functions created within the project.

2 Graph model

The problem will be considered as a graph, with the bins being the nodes of the graph and the "path" between any two bins being the edges of it.

2.1 Parts of the graph

The next part of this document describes each of these elements of a graph in depth, detailing the attributes they have and some of their functionalities, although they are mostly used as a wrapper for data.

Nodes

The nodes are made up of three main things; an **index** unique and used to search for a node with a cleaner and more understandable code, a **weight** which is the *estimated* weight a bin (node) will have (this is used to plan routes and avoid trucks being full prematurely) and the **coordinates** of it.

Two more elements are part of a node, the **center** and **visited** parameters. The first defaults to false and is used to indicate if a node is the center of a graph (the start point of the trucks), while the latter marks a node as being visited while finding a path. We can see all this by having a look at the code of the *constructor* of the `Node class`

Listing 1: Node constructor

```
def __init__(self,
                index: int, weight: float,
                x: float,
                y: float,
                center: bool = False):
        self.index = int(index)
        self.weight = float(weight) if not
        ↪  bool(center) else 0
        self.center = bool(center)
        self.visited = False
        self.coordinates = (float(x),
        ↪  float(y))
        self.angle = 0.0
```

A couple of questions may arise from looking at this code snippet. One of them may be why would the weight be set to 0 and, that is because if a Node is the center Node of a graph, it won't have a weight. The other question is what is the `angle` attribute? This attribute is used when dividing the graph into zones. This will be talked more in a latter part of the document.

Edges

The edges include data about the nodes it comes from and goes to, as well as the distance and value of it. Looking at the `Edge class` constructor:

Listing 2: Edge constructor

```
def __init__(self, speed: float, origin:
↪  Node, dest: Node):
        self.length =
        ↪  origin.get_distance(dest)
        self.speed = float(speed)
        self.origin = origin
        self.dest = dest
        self.time = (
            (float(self.length)/1000)
            /self.speed
        )
        self.value = self.length + self.time
```

We can see that to create an edge, we need the average speed of it and the origin and destination nodes. To calculate the length of an edge, a function called `get_distance` is used. This function returns the **Manhattan distance** between two nodes. This distance ecuation is prefered over the **Euclidean distance** as it resembles reality more than the second one. With these data we can get (an estimation of) the time that it takes to take an edge from node x to node y; as well as a value (the actual cost of an edge) that is just the sum of the length and time of an edge. This is so edges of the same length can have different values depending on the time they take and vice versa (more time implies a higher cost, as well as a higher length implies more cost for maintaining a truck).

2.2 Defining and creating a graph instance

Having established the two main components of a graph, we can start to take a deeper look into the `Graph class`.

First, let's have a look at its constructor as usual:

Listing 3: Graph constructor

```
def __init__(self):
        self.graph = {}
        self.node_list = []
        self.edge_list = []
        self.nodes = 0
        self.edges = 0
        self.center = None
```

A graph instance starts with an empty `dict` (the actual representation of the graph) as well as a bunch of auxiliary values such as the number of nodes and edges and a list of all the nodes and edges. These parameters are used in different methods along the `class`. To start populating the graph we can use one of the two options described below:

2.2.1 Manually adding all the desired nodes and edges

The `Graph` class has two methods used for adding all the nodes and edges manually, `self.add_node(node)` and `self.add_edge(edge)` respectively. This functions check that both the nodes and edges added are not in the graph already, throwing custom exceptions in case this check fails (see **Exceptions** for more info on this custom exceptions). For a more detailed explanation of the functions described above, refer to the function's `docstring` in the module `model.Graph`.

An example of how to use these functions to create a graph is shown in the code snippet below.

Listing 4: Example of node instantiation

```
def instantiate_graph():
    nodes = [
            Node(0, 0, 0, 0, True),
            Node(1, 1, 1, 2),
            Node(2, 2, 3, -2),
            Node(3, 3, 0, 5),
            Node(4, 4, -7, 0)
        ]

    edges = []
    edges.append(Edge(10, self.nodes[0],
    ↪  self.nodes[2]))
    edges.append(Edge(4, self.nodes[1],
    ↪  self.nodes[4]))
    edges.append(Edge(2, self.nodes[3],
    ↪  self.nodes[2]))
    edges.append(Edge(4, self.nodes[2],
    ↪  self.nodes[0]))

    g = Graph()

    for node in self.nodes:
    ↪  self.g.add_node(node)
    for edge in self.edges:
    ↪  self.g.add_edge(edge)
```

But, as using this method for bigger graphs is quite time consuming, an alternative has been developed.

2.2.2 Creating a graph from a data file

This is the faster and easier option and it is recommended to use if possible, as the process is done automatically. This is done by calling the `self.populate_from_file(file)` function with the path of a properly formatted text file. The code used inside this function automatically creates the nodes and edges needed. The format of the data files must follow the format rules of this example:

```
n
idx_1 weight_1 x_1 y_1
idx_2 weight_2 x_2 y_2
         ...
idx_n weight_n x_n y_n
m
speed_1 origin_1 dest_1
speed_2 origin_2 dest_2
         ...
speed_m origin_m dest_m
```

By just having a fast look at this format rules, the enforced format should be obvious.

This format was chosen as it is similar to the one normally used in competitive programming when defining graphs, as well as being really easy to read using a function. For more information on how this works, refer to the function's `docstring` in the `module model.Graph`.

3 Path-finding inside a graph

The graph class includes various functions to find a good path inside an instance of it. The paths generated by different algorithms are discussed in the **Proofing our solution** section, where they are compared against each other to find if our implementation is truly better than a simpler algorithm. The different algorithms used in this model are *BFS*[1], and the genetic algorithm used, that is really similar to the standard one used on *TSPs.*[2]

4 Algorithm design

In this section, it is explained how the algorithm woks. It consists of three main phases:

- **Dividing the graph.** The graph is divided in n zones, where n is the number of rounds needed to pick up every bin (node).
- **Equilibrating the sections.** The zones are post-processed to try and make their weights similar and delete any unnecessary zone.
- **Finding a path.** A path is searched for every zone of the graph.

The steps of this algorithm are explained in the following sections, detailing the algorithms used and showing how they work in depth.

4.1 Dividing the graph

Starting from the initial graph that contains the nodes, their weights and the connections (edges) between them, divisions are made so that, each truck $t \in TruckFloat$ *can* be assigned a zone. One problem is that sometimes, more zones are created than there are trucks. This can be either because it is impossible for all of the available trucks to pick up all the bins in one go or because doing so would create convoluted and illogical zones.

Logically, each of this zones is converted into it's own graph, effectively making them sub-graphs. This sub-graphs must comply to both having a **total weight lower than the truck's capacity** and having a **total weight similar to all other sub-graphs**. This is enforced by the zone-creating algorithm.

[1] Breadth First Search algorithm - Geeks for Geeks
[2] Travelling Salesman Problem - Wikipedia

This algorithm divides the graph based on a simple concept: the **angle** a bin forms with the center node. Using this, we get contiguous zones that **don't intersect** and are all ordered around one point. To do so, the `self.angle` attribute of the nodes is calculated using the formula $\arctan(\frac{b_1-c_1}{b_0-c_0})$, where b is the bin whose angle is being calculated, c is the center node of the graph and $(0, 1)$ is either the x or y coordinates of each node respectively. In python, this can be done by using the `math.atan2()` function.

Once this angle is calculated, a list of the nodes ordered by this angle is created and the function `self.create_zones()` is called. This function used the ordered list of nodes and the maximum capacity of a truck to create the zones. It uses a **greedy**[3] approach where, before adding a node to a zone, it checks if doing so would make the zone exceed the capacity of a truck. Despite being quite a simple approach, it can be easily modified to make use of more data (such as the time it takes to visit a node vs an hypothetic maximum time) and, thanks to the use of the mentioned ordered list, the zones created by it are usually solid. However, this won't be the final zones, as post-processing is applied to improve them. This is discussed in the next section. Finally, the central node is added as the first node of each zone, as it will always be present on a route.

4.2 Equilibrating the sections

Now, the zones that we get from the previous step are pretty good in **most** cases, but that doesn't mean they are pretty good in **all** cases. There are two problems we get with the previous step. The first one is that there could be zones where we could pick up more weight. The second (and more important) problem is that, because of the nature of lists (they have a set start and end), the final zone can end up with only one or two bins, and such having a tiny total weight, or needing a truck for just a couple of bins (nodes). To fix this two issues, we call the `self.postprocess_zones()` function of the graph.

The function evaluates each zone individually and checks if frontier nodes can be moved to the next or previous zones. This is somewhat trivial as implementing each zone as an ordered list of nodes makes finding the frontier nodes as easy as looking at the first and last nodes in said list. We can encounter two cases. One is when the current zone has more than 3 nodes (center and two other nodes). This means it won't be eliminated by moving the nodes to contiguous zones. In this case, we check if we can move (both or any) the frontier nodes another zone while keeping the destination zone's weight under the truck's capacity. It is also checked that the destination zone does not have 3 or less nodes, as it can be a candidate for elimination.

The other case we might get is having a zone with 3 or less nodes. This is the case we want to prioritize, as eliminating a zone can decrease the total cost more than moving a node. The algorithm checks moving each one of the zone's nodes but the center one to either the next or previous zone, as when a zone is of that size, all of its nodes are frontier nodes.

Once the algorithm has run, our hope is having reduced the total number of zones, as well as equilibrating each area's total weight. This has been tested with a set graph with random edges and node weights and has stood true. This is shown in the **Tests for zoning**.

4.3 Finding a path

Here is the most important part of the whole process. Each zone is evaluated using a **Genetic Algorithm** whose hyperparameters have been optimized (see the **Training the algorithm** section) to find the best paths that visit all nodes while minimizing the cost of the path (the objective function).

The genetic algorithm works by generating a set number of individuals. Each of this individuals is a possible path of our graph, generated randomly and made up of genes (our nodes). Then, each individual is evaluated for its **fitness value**. This fitness value is calculated as:

$$f(x) = (Edge(c, x_1)_{value} + \sum_{k=1}^{n-2} Edge(k, k+1)_{value} + (\alpha \cdot \sum_{k=0}^{n-1}(k_{weight} \cdot (n - k_{idx}))$$

Where x is an individual (a list of nodes), $Edge(x, y)$ is a function that returns the edge between two nodes $x, y \in Graph$, c is the center node of the zone/graph, k is the k-sm element of the individual and n is the length of the individual.

This objective function returns the value of the path plus a penalization for picking up heavy nodes early, as this can make the truck run into more maintenance costs. This penalization has a weight α that can be adjusted to make it more or less relevant This α has been set to 0.2.

Then, after each individual has it's fitness value, the genetic algorithm selects the ones that have a lower value using the tournament selection method. This method works as it sounds, by creating a sort of "tournament" and comparing t individuals. The individual with a lower fitness value will pass to the next round and so on. This t value is what is known as a **tournament size**. Then, it compares them and performs two more operations: **mating** and **mutation**.

Mating combines two individuals to create new ones in order to populate a new generation. This mating can use a variety of algorithms to do so. Mutation takes a single individual and performs an operation on one or more of its genes to modify it. The algorithms used for both of this functions will be discussed in the **Training the algorithm** section. This operations aren't always performed on an individual. For mating, the default

[3] Greedy algorithms - Geeks for Geeks

probability (*cxpb*) is 0.9 while for mutating, the probability (*mutpb*) is 0.1 and the probability of each gene to be mutated *indpb* is 1.0. This probabilities, as well as the tournament size are part of the hyperparamenters and the final values for them are also discussed in the section **Training the algorithm**.

Finally, the algorithm repeats all of this again. This whole process is called a generation. The amount of individuals per generation and the number of generations (*popsize* and *ngen*) are also hyperparameters and for now, we will keep them at 200 and 100 respectively.

5 Testing our model

In order to prove all of our code is working as expected, a `tests.py` module was created. In here, we test all the methods and classes work as intended. This is part of the test-driven development (TDD) approach, where tests are written in par with the actual functions, making it easier to write working code from the get-go, refactor code and fix bugs that may appear. The only exception for this was the actual algorithm, as the result of the execution is undetermined until it is done. Instead of testing this result, the tests aim to prove the code works, although a solution cannot be tested for (as opposed to other algorithms like *BFS* or *Dijsktra*). For other parts of the algorithm, like the zoning one, the tests are performed against a set graph with random edges and node weights/coordinates.

Let's have a look at this "special" tests. First, the graph division test uses said test graph and checks that the zones it creates are correct. Looking at the code:

Listing 5: Testing the zoning algorithm

```
def test_divide_graph(self):
    """Tests graph division into zones."""
    g2 = Graph()
    g2.populate_from_file(
        f"{os.getcwd()}/files/test2.txt"
    )
    self.assertEqual(
        g2.divide_graph(725),
        [[0, 9, 3, 4, 10], [0, 11, 2, 7, 8],
        ↪  [0, 5, 6, 1, 12]]
    )
```

We can see that it checks the zones are the ones we predicted. The graph used is one where, without post-processing, the node 9 would have ended up on its own zone.

Now, the Genetic Algorithm itself is a bit more tricky to test, as due to its random nature, the results won't be always the same. This is why we instead test for 3 things that we know must hold true:

- The start and end of the path are the same nodes.
- The start and end of the path returned are the center node.
- The genetic algorithm gives a result that is better than a random path.

Knowing this, we can look at the code and see how we test for each of this statements:

Listing 6: Testing the Genetic Algorithm (TSP)

```
def test_ga_tsp(self):
    """Tests the Genetic Algorithm (TSP)"""
    g2 = Graph()
    g2.populate_from_file(
        f"{os.getcwd()}/files/test2.txt"
    )
    p, v = g2.run_ga_tsp(
        dir=f"{os.getcwd()}/files/plots",
        vrb=False
    )
    os.remove(
        f"{os.getcwd()}"
        "/files/plots/Path0.png"
    )
    os.remove(
        f"{os.getcwd()}"
        "/files/plots/Evolution0.png"
    )
    self.assertEqual(p[-1], p[0])
    self.assertEqual(p[0], 0)
    random_path = (
        [n.index - 1 for n in random.sample(
            g2.node_list, g2.nodes - 1
        )]
    )
    self.assertTrue(
        g2.evaluate(random_path)[0] > v
    )
```

The first two cases are simple, we just check the last and first element of the returned list are the same and then that they are the same as the node 0 (our center). The third case involves creating a randomly generated **valid** path. For that, we sample n elements from the graph's list of nodes, where n is equal to the number of nodes minus the center one.

To test the code, the `unittest`[4] native python module was used, as it can easily be used in any machine with python installed, making portability easier and thus improving the quality of the solution. The tests for a class always follow a similar structure. Each class has it's own test class named like `TestX`, where X is the name of the class to test. For example, to find the tests for the `Graph class` you just have to find the `TestGraph class` inside `tests.py`.

Although some default methods have been re-coded for our custom implementation, they have not been tested for as for example, testing `__repr__` is trivial and rather pointless. The `__init__` methods have also not been tested for implicitly, but testing other methods inside each class proves `__init__` works as expected.

Exceptions are tested for in the methods they can be raised by forcing them to happen and checking the

[4] Unittest library documentation

text of the exception using the `assertRaisesRegex()` function of the module `unittest`.

6 Creating training files

To check how our genetic algorithm works in a wide variety of example graphs that have similar characteristics to our map, a helper script `create_models.py` has been coded. In this script, several graph files are created, with a random range of nodes, edges and the values of them. By default, said values are:

Listing 7: Training file creation script constants

```
DATA_SIZE = 20
↪  # Number of training files created.
MIN_NODES, MAX_NODES = 200, 1000
↪  # Minimum and Maximum number of nodes.
MIN_WEIGHT, MAX_WEIGHT = 100, 1500
↪  # Minimum and Maximum weight of the
↪  nodes.
MIN_X, MAX_X = -100, 100
↪  # Minimum and Maximum x coordinates of
↪  the nodes.
MIN_Y, MAX_Y = -100, 100
↪  # Minimum and Maximum x coordinates of
↪  the nodes.
MIN_SPEED, MAX_SPEED = 20, 60
↪  # Minimum and Maximum speed between two
↪  nodes.
```

To ensure the training files are somewhat similar to the real world scenario, the graphs created have a **high density**, as in a city, it can be assumed that we can get from a node x to any another node y without having to pass by any intermediate node, this is assumed for every node pair $\{x, y\} \in Graph$ we can choose. In other words, **the density of our graphs will be of 1**. As a note, the density of a graph is determined by the function $d(g) = \frac{E}{N \cdot (N-1)}$. From this formula, we get $E = d(N \cdot (N-1)$ and thus we can obtain the number of edges needed to obtain the desired density of 1.

The script runs as a CLI tool and can be called with the command:

Listing 8: Creating training files

```
[root@localhost ~]# cd path/to/script
[root@localhost ~]# python3 create_models.py
```

to run with the default values or it can be added some extra arguments to modify those values. Below is a table of all the extra arguments as well as a few examples of command line calls to the script.

Flag	Description
-f	Number of training files to be created
-n	Minimum number of nodes
-N	Maximum number of nodes
-w	Minimum weight of a node
-W	Maximum weight of a node
-x	Minimum x coordinate of a node
-X	Maximum x coordinate of a node
-y	Minimum y coordinate of a node
-Y	Maximum y coordinate of a node
-s	Minimum speed between two nodes
-S	Maximum speed between two nodes

```
/* Generate 12 training files: */
[root@localhost ~]# python3
↪  create_models.py -f 12
/* Make a minimum of 10000 nodes and a
↪  maximum of 100000: */
[root@localhost ~]# python3
↪  create_models.py -n 10000 -N 1000000
/* Make the distance between two nodes
↪  always be 2.000: */
[root@localhost ~]# python3
↪  create_models.py -d 2.000 -D 2.000
/* Generate only one file, with 150
↪  nodes all separated between 6.000
↪  and 35.000 and a minimum speed of
↪  25: */
[root@localhost ~]# python3
↪  create_models.py -f 1 -n 150 -N 150
↪  -d 6.000 -D 35.000 -m 25
```

7 Training the algorithm

A Genetic Algorithm is not trained *per-se*. What is meant by training is searching for the best hyperparameters, as well as the best mating and mutating functions. This section aims to explain how the algorithm has evolved until it has reached its final status.

To help with finding where and why the genetic algorithm fails, plots have been generated showing the path calculated, as well as the evolution of both the value of the best individual's fitness and the average fitness of a generation. This plots are generated using the `matplotlib.pyplot` library, used in an auxiliary module called `plotter` created to extract this functionality out of the `model` module.

The first version of the Genetic Algorithm used the default values for the hyperparameters discussed in the **Finding a path** subsection. The mating function used was **Ordered crossover (OX1)**, commonly used when the individuals are represented as lists that must contain all the elements of a group and can't have repeated genes. It is a variation of the *two-point crossover* where after choosing our sections, the parent lists are iterated through to fill the rest of the offspring's genes, without repeating them.

The mutation function used was **Scramble mutation**, where a random sequence of genes is selected and

the order of them is shuffled. The figure below shows how it would transform a list.

$$mutate([1, 5, 2, 4, 8, 3]) \rightarrow [1, 5, 4, 3, 2, 8]$$

Finally, this iteration uses the method **eaSimple** provided by the *DEAP* framework. Let's have a look at the plots created when running the genetic algorithm with this parameters to see how they made the algorithm work:

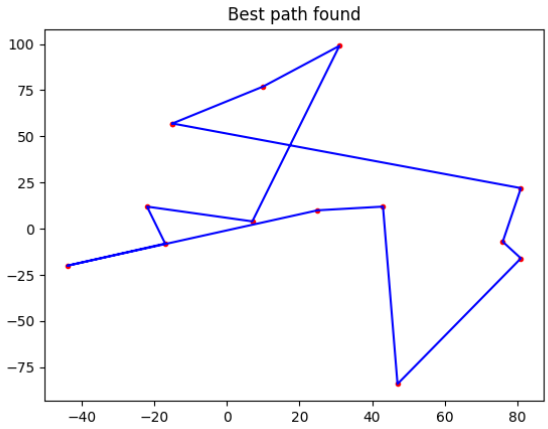

Figure 1: Path of the first iteration

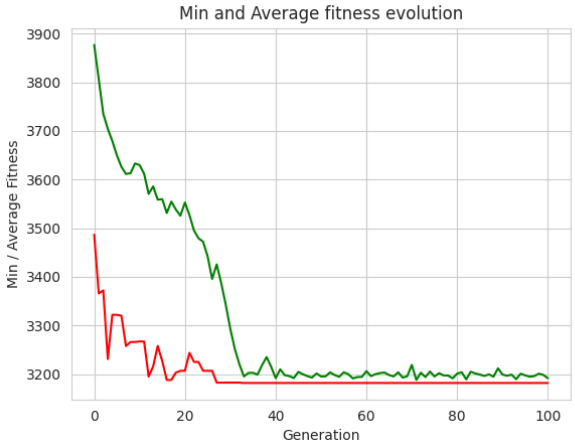

Figure 2: Evolution of the first iteration

By looking at the provided figures, it can be noticed that the genetic algorithm can be tuned as it seems to have stopped improving around generation 25 while not finding the optimum path. The best path also seems to fluctuate a lot. To fix this, we can try to introduce elitisim. This enables us to keep the best solutions by letting they skip the genetic operations. To implement it, the **eaSimple** algorithm must be modified to add the individuals from the hall of fame to the offspring. The implementation used is the one from the book referenced in the **Bibliography**, and it can be found publicly here. We can also try if making the **Hall of Fame** (the best individuals that will be saved) bigger helps. After changing the algorithm to use elitisim, and setting the size of the Hall of Fame to 30, the plots generated look like this:

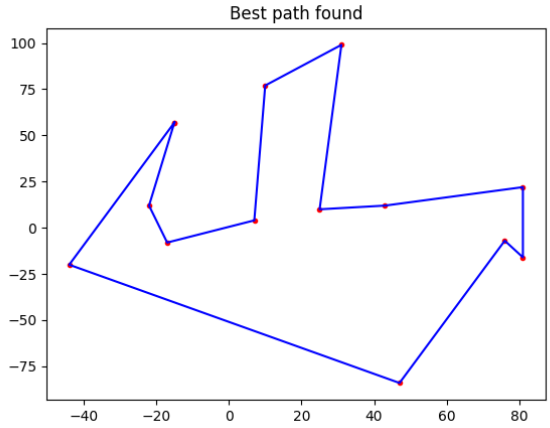

Figure 3: Path of the second iteration

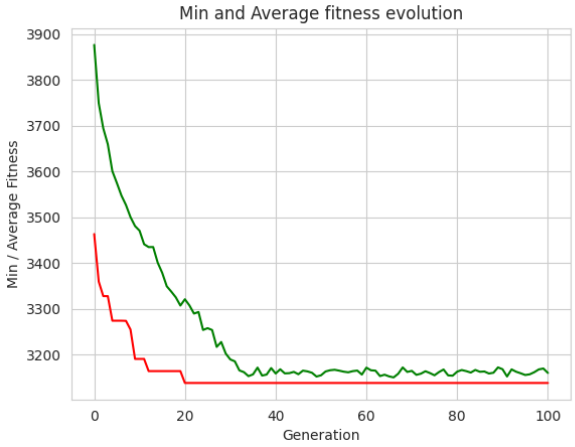

Figure 4: Evolution of the second iteration

The solution looks a lot better, and the evolution graph shows that the best solution found is not fluctuating as much, resembling more the shape and form of an asymptotic function. This will help on keeping good solutions. For bigger graphs, increasing the number of generations and the population size might be better, as it would give the algorithm a better chance to find the optimum solution, rather than ending up in a local minimum.

8 Proofing our solution

In order to prove our solution is better than others found by non-heuristic algorithms, a benchmark module has been created. This module uses the `create_models.py` module to generate 100 models and runs each path-finding algorithm inside the `Graph()` `class` on each one of the generated maps, getting an average time to find a solution, as well as an average value of the objective function that calculates the cost of using each route.

To create a new benchmark, just execute the module by using the command

```
[root@localhost ~]# python3 benckmark.py
```

The result of the benchmark will be displayed on the terminal screen. If we run this command, we can see an example of the result.

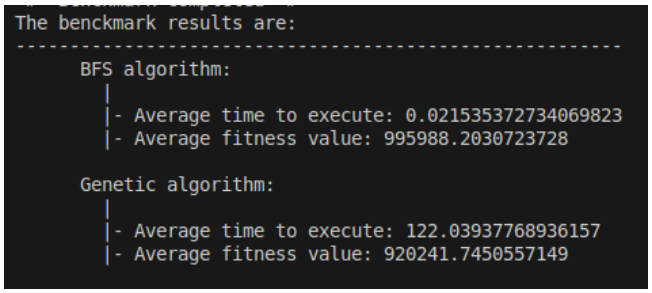

Figure 5: Benchmark. BFS vs Genetic Algorithm

As expected, due to its greater complexity and having to run for a somewhat high number of generations, the GA takes way longer (122s vs 20ms), but the results are way better. The genetic algorithm is, on average, 7,61% better.

9 Exceptions

In order to allow for easier debugging and usage of the software, custom exceptions have been created following python's guidelines. Contrary to other programming languages such as C where the standard practice is to use different return codes for errors in function execution, python uses Exceptions that can stop the execution of the program or be catched and treated properly. All of the custom exceptions are located in the `exceptions.py` module.

The exceptions defined are:

- **NodeNotFound:** Raised when a node with index i is searched in a graph and not found. Returns a message and the index searched for.
- **DuplicateNode:** Raised when a node is being added to a graph, but it already exists in that graph instance.
- **DuplicateEdge:** Raised when an edge is being added to a graph, but it already exists in that graph instance.
- **EdgeNotFound:** Raised when an edge between x and y is searched in a graph and not found.
- **NoCenterDefined:** Raised when a graph doesn't have a center defined.
- **EmptyGraph:** Raised when an operation is performed on an empty graph.

[5]Vehicle Routing Problem - Wikipedia

10 Next steps

The next steps planned for the project are:

- **Adding a Genetic Algorithm for VRP.** During investigation, I've found a problem similar to TSP called *VRP*[5]. I want to test how it performs insted of my own approach of dividing the graph into zones and having n TSP problems to solve.
- **Improving the heuristics.** I would like to investigate on adding more restrictions to the zone-making and fitness function so that the solution can be better applied to our real-world problem. Commenting this project in a talk I had with someone who works in the residue collection industry, who stated that any improvement will be huge, as now routes are not good in most cases and involve a lot of maintenance costs, specially mid-route.
- **Integrating solution with second TFG.** Once this TFG is nearly finished, the code and scripts developed will be used for the second TFG, which will tackle getting the data and updating routes, as well as requesting a new route in case a truck is filled prematurely or can't pick up a bin (node).

11 Bibliography

The book **Genetic Algorithms with Python** has been used for reference and guide on how to code, create and improve the Genetic Algorithm.

12 Needed libraries

In order to run the code, you will need the following libraries and packages in at least the versions specified:

- **Python** → *version 3.10 or higher* Due to the use of `match` in `create_models.py`
- **DEAP** → An *evolutionary computation framework* used to code the `Genetic Algorithm`
- **matplotlib.pyplot** → A *state-based interface to matplotlib.* It provides an implicit, MATLAB-like, way of plotting. Used to plot the solutions of the Genetic Algorithm
- **seaborn** → A *data visualization library* based on matplotlib. It provides an interface for drawing attractive statistical graphics

13 Repository

A Github repository for the project is available at this link. Please, email my URJC email address here to request access to it.