Computational Intelligence for Optimization
Final Project

Technicalities

- Link to git repository: https://github.com/mpjan/cifo-final-project
- You can recreate the conda environment used to develop the project using the file in `env/conda_environment.txt`.
- Use `route_inspection_problem.ipynb` to run the solution to the problem. Note that we have included some commented-out cells to create example populations, evolve them, and inspect the individuals in this population.

Group members

- Alberto Parenti (20211304)
- Natalia Poles (20210675)
- Marcelo Jannuzzi (20210674)

Problem definition

Our objective was to use genetic algorithms to solve the *route inspection problem*, also known as the *Chinese postman problem*.

Given a graph of nodes and edges, the route inspection problem involves finding the shortest closed path that visits every edge. That is, starting from an origin node, we must traverse the graph and visit every edge at least once, finishing back at the origin.

We made the following general assumptions when solving this problem:

- An edge can be visited more than once. This means that our solutions have different sizes, and we had to make major changes to the Charles Genetic Algorithm Library developed throughout the semester.
- We are dealing with an undirected graph. That is, if an edge exists between nodes A and B, we can move freely from A to B and from B to A.
- Node A is the origin node. That is, we start at node A and must end back at A.
- The edges of the graph may have different weights.

Representation

Our individuals (which are potential solutions to the route inspection problem) are composed of a sequence of nodes drawn from the target graph we're basing ourselves on. For example, if the target graph were the one shown in Figure 1, a potential solution can be represented by: `A, B, C, D, C, A, D, C, B, A, B, A`.

However, random strings of nodes don't necessarily form a valid path through the graph, because there may not exist an edge connecting two random nodes drawn from the graph.

Taking the graph shown in Figure 1 of the appendix as an example, we can see that no edge exists directly linking nodes B and D, so any solution that contains a path linking B and D would be invalid.

We opted to deal with this issue by both penalizing invalid solutions in the fitness function and also coming up with ways to make it more likely to generate valid solutions, by opting to create our generation 0 solutions and the mutation and crossover functions in a way that takes this into account.

To create our initial random individuals, for example, we simply start at node A and then randomly traverse the edges of the graph until our solutions reach a certain size, and then chop the end off these solutions so that the last node in the solution is the origin node, A. These functions were implemented in `charles/utils.py`.

Fitness function

As mentioned above, the route inspection problem involves finding the shortest closed path that visits every edge, and as can be seen in the example graphs included in the appendix, the edges of the graph are weighted, so we are interested in the path with the smallest weight.

In general terms, the fitness of a path is given by the sum of the weights of the edges in the path. The smaller the fitness, the better the path. However, the fitness of a path also depends on other factors, such as:

- Does the path start and end at the origin?
- Does the path contain all the edges in the graph?
- Is the path valid? A path is valid if it contains a possible traversal of the graph (only edges that exist and in a valid order).

The fitness of a particular individual is initialized as the sum of the weights of the edges in the path, and then we apply a penalization of +1 million if it violates any of the conditions listed above. This is defined in the `get_fitness()` function in `charles/utils.py`.

We realize this is a rather simple fitness function, but since it worked quite well, we opted to keep it.

Selection methods

Apart from the fitness proportional and tournament selection functions implemented in class throughout the semester, we also implemented ranking selection.

Crossover operators

As described above, because of the nature of the route inspection problem, our solutions don't necessarily all have the same size and the regular crossover operators seen during the semester would lead to a great number of invalid individuals. For these reasons, the crossover functions were designed to take this into account. We came up with the two crossover functions described below.

Crossover on one common node

Given two parent individuals, we do a crossover between them by choosing a random node they have in common, and then swapping the paths that lead into and out of this node between the two parents. Figure 3 in the appendix illustrates this visually.

Crossover on two common nodes

Given two parent individuals, we do a crossover between them by choosing two random nodes they have in common, breaking each parent into 3 parts (start, middle and end) and then swapping the middle paths of the two parents. Figure 4 in the appendix illustrates this visually.

Mutation operators

As described above, because of the nature of the route inspection problem, our solutions don't all have the same size and the regular mutation operators seen during the semester would lead to a great number of invalid individuals. For these reasons, the mutation functions were designed to take this into account. We came up with the two mutation functions described below.

Tail mutation

We mutate the tail (the end) of an individual by picking a random node in the individual, and then randomly traversing the graph until we are back at the origin node.

Middle mutation

We mutate a random sub-path in the middle of an individual by picking random start and end nodes in this individual and then randomly traversing the graph between them, substituting the original sub-path by the random traversal.

Final results

After implementing the ideas described above, we tested the performance of each possible configuration of selection method, crossover method, mutation method and presence of elitism on the graph depicted in Figure 2 of the appendix. For each of the 24 possible configurations (3 selection options × 2 crossover options × 2 mutation options × 2 elitism options) we ran 100 iterations for 100 generations. Additionally, we used the following parameters:

- Populations of size 10
- 90% probability of crossover
- 10% probability of mutation

The mean fitness of the best individual in the last generation in each configuration is described in Table 1 of the appendix. From this we can see that the best configuration was to use:

- Tournament selection
- Crossover on two common nodes
- Tail mutation
- Elitism

However, many of the best configurations performed very similarly. This can be seen in Figure 5 of the appendix, where we plot the mean fitness (with their 95% confidence intervals) in the final generation for each configuration on the algorithm. Note the large overlap (including the mean) between confidence intervals in the top performing solutions. Therefore, we can't conclude with a high degree of statistical confidence that the best configuration described above is in fact the best one. However, looking at the results, we can notice that:

- The top-performing configurations tend to include
    - Elitism
    - Tournament or ranking selection
    - Crossover on two common nodes
- The worst-performing solutions tend to include
    - No elitism
    - Fitness proportional selection

Note that the worst-performing solutions didn't even converge.

In Figure 6 of the appendix we plot the evolution of the mean fitness over generations for each configuration. The solutions which didn't converge were excluded, but even so, we can see that the configurations cluster into two groups right from the start. That is, if the configuration starts off performing better, it tends to continue performing better.

Division of labor

Each group member contributed equally to the project.

Appendix

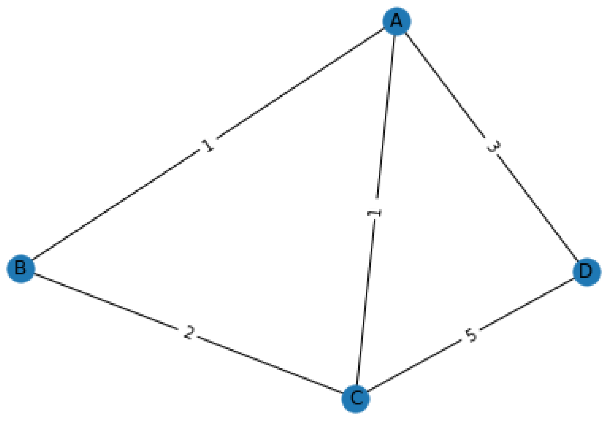

Figure 1: An example graph for the route inspection problem.

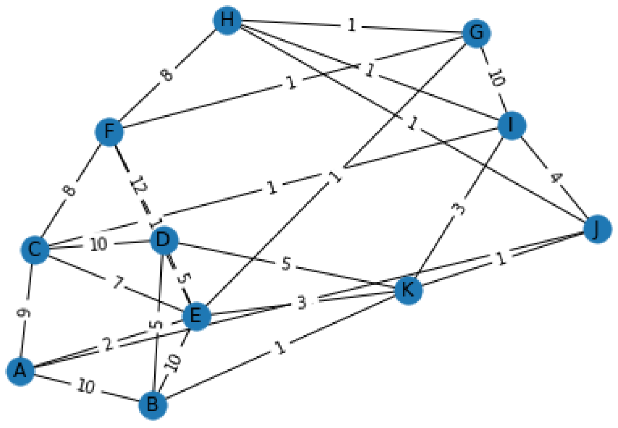

Figure 2: The graph on which we tested our implementations and ran our solutions.

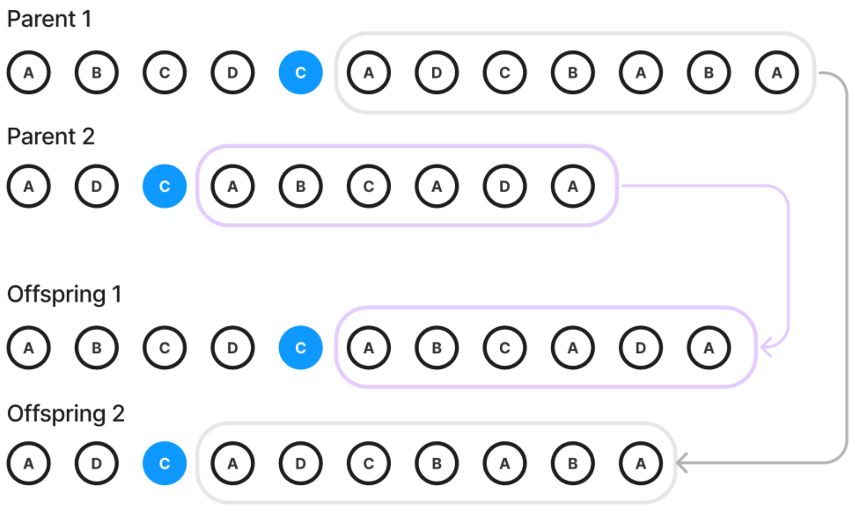

Figure 3: Crossover on one common node.

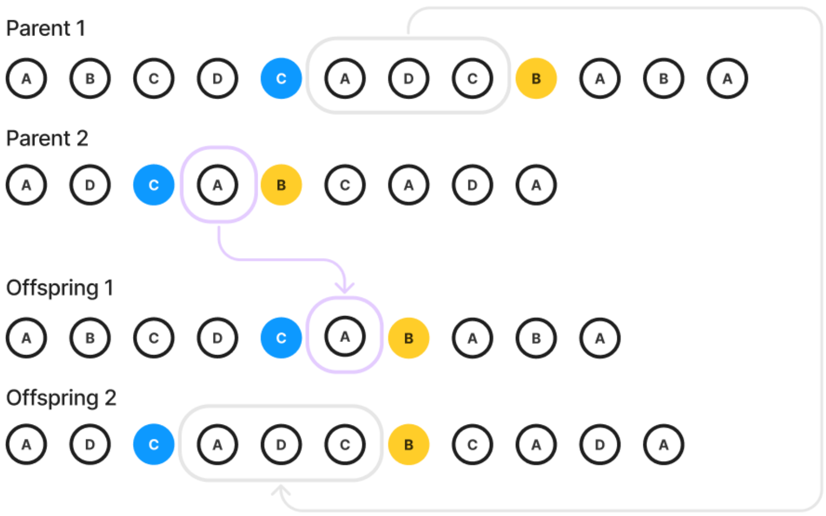

Figure 4: Crossover on two common nodes.

	fitness
config	
selection:tournament, crossover:co_on_two_common_nodes, mutation:tail_mutation, elitism:True	166.31
selection:ranking, crossover:co_on_two_common_nodes, mutation:middle_mutation, elitism:True	167.34
selection:tournament, crossover:co_on_two_common_nodes, mutation:middle_mutation, elitism:True	168.83
selection:ranking, crossover:co_on_two_common_nodes, mutation:tail_mutation, elitism:True	170.07
selection:ranking, crossover:co_on_common_node, mutation:middle_mutation, elitism:True	175.48
selection:tournament, crossover:co_on_common_node, mutation:middle_mutation, elitism:True	179.12
selection:tournament, crossover:co_on_common_node, mutation:tail_mutation, elitism:True	179.31
selection:ranking, crossover:co_on_common_node, mutation:tail_mutation, elitism:True	181.80
selection:tournament, crossover:co_on_common_node, mutation:middle_mutation, elitism:False	188.23
selection:ranking, crossover:co_on_common_node, mutation:tail_mutation, elitism:False	190.00
selection:tournament, crossover:co_on_common_node, mutation:tail_mutation, elitism:False	190.33
selection:ranking, crossover:co_on_common_node, mutation:middle_mutation, elitism:False	192.80
selection:fitness_proportional, crossover:co_on_common_node, mutation:middle_mutation, elitism:True	219.41
selection:fitness_proportional, crossover:co_on_common_node, mutation:tail_mutation, elitism:True	224.41
selection:fitness_proportional, crossover:co_on_two_common_nodes, mutation:tail_mutation, elitism:True	228.61
selection:fitness_proportional, crossover:co_on_two_common_nodes, mutation:middle_mutation, elitism:True	240.14
selection:tournament, crossover:co_on_two_common_nodes, mutation:tail_mutation, elitism:False	220196.90
selection:ranking, crossover:co_on_two_common_nodes, mutation:middle_mutation, elitism:False	240224.25
selection:ranking, crossover:co_on_two_common_nodes, mutation:tail_mutation, elitism:False	260229.48
selection:tournament, crossover:co_on_two_common_nodes, mutation:middle_mutation, elitism:False	290181.89
selection:fitness_proportional, crossover:co_on_common_node, mutation:middle_mutation, elitism:False	510550.64
selection:fitness_proportional, crossover:co_on_two_common_nodes, mutation:middle_mutation, elitism:False	600535.62
selection:fitness_proportional, crossover:co_on_common_node, mutation:tail_mutation, elitism:False	750340.18
selection:fitness_proportional, crossover:co_on_two_common_nodes, mutation:tail_mutation, elitism:False	780316.70

Table 1: Mean fitness of the best individual in the last generation in each configuration

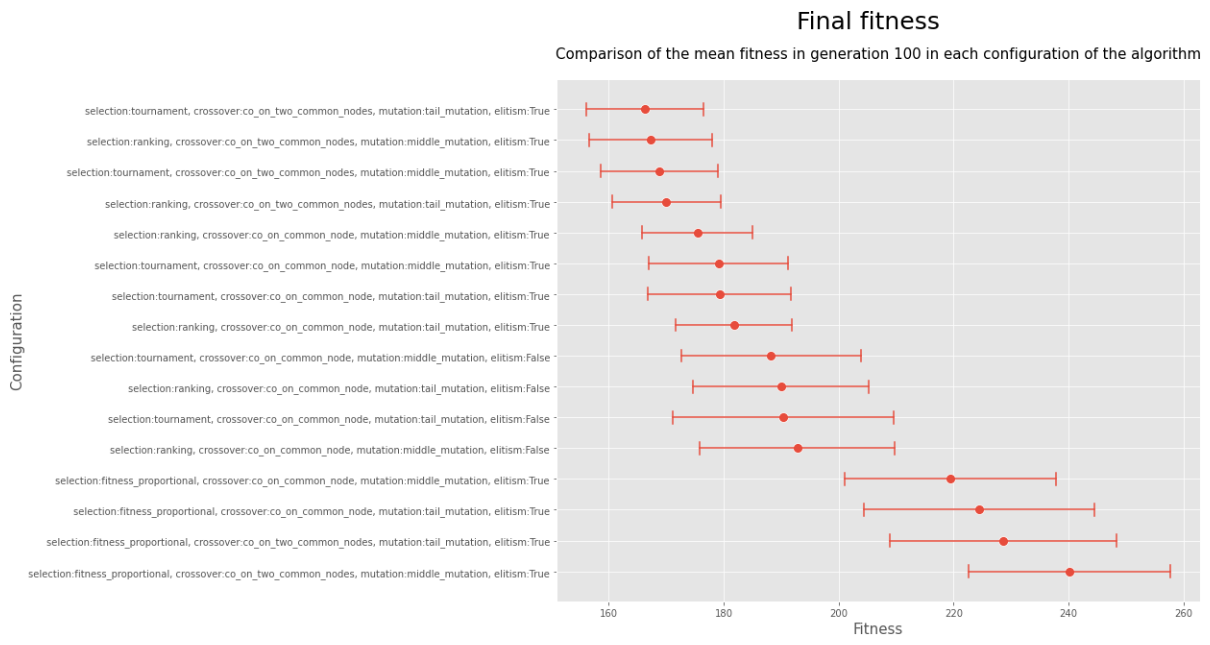

Figure 5: Mean fitness in generation 100 in each configuration of the algorithm. Configurations which didn’t converge were excluded from this view.

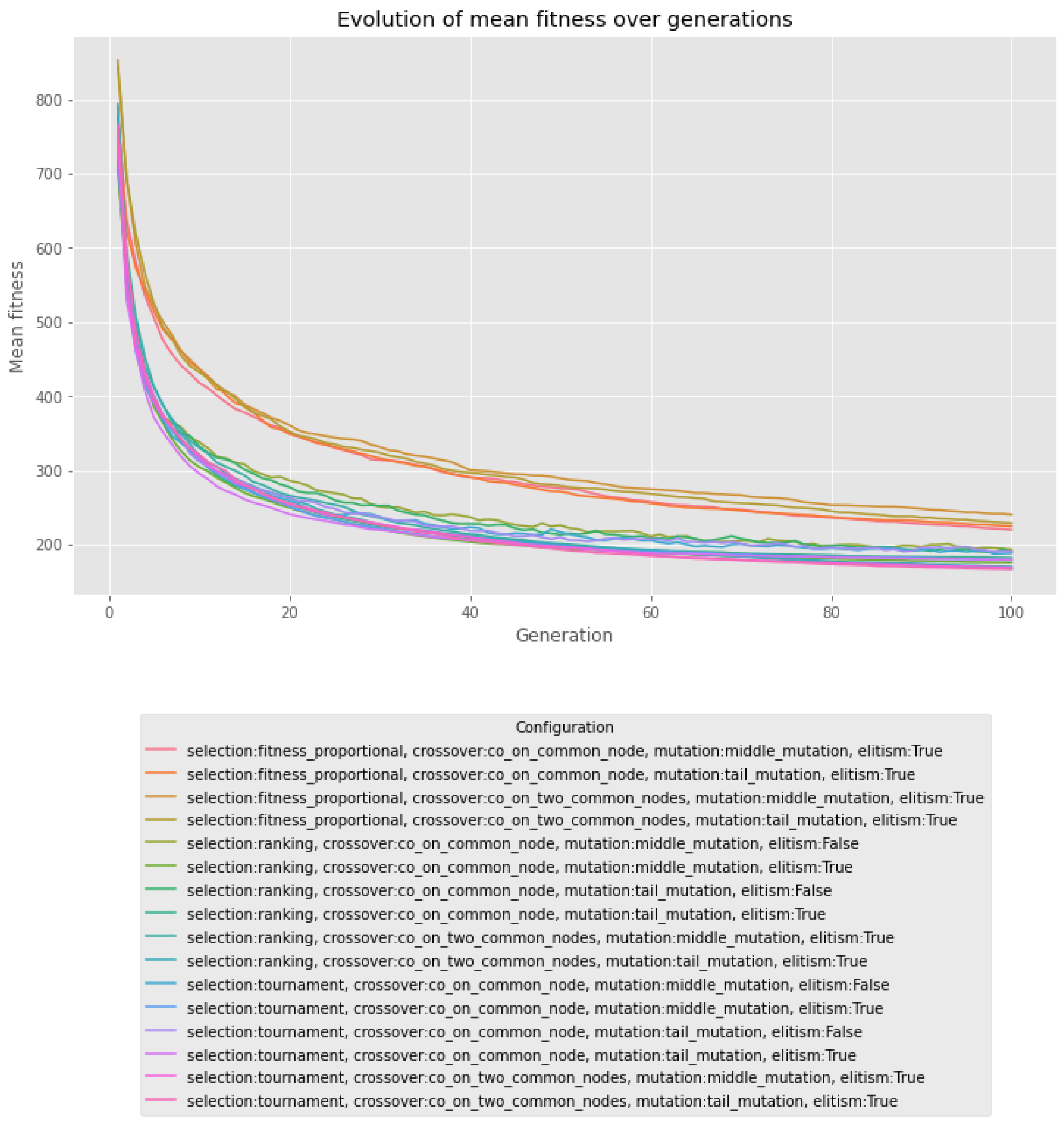

Figure 6: Evolution of mean fitness over generations for each configuration. Configurations which didn't converge were excluded from this view.