universidade de aveiro

DEPARTAMENTO DE ELECTRÓNICA, TELECOMUNICAÇÕES E INFORMÁTICA

Simulação e Otimização (2024/2025)

Optimization Mini-Project

Daniel Pedrinho Nº107378 (50%)

Sara Almeida Nº108796 (50%)

6th of June 2025

Contents

1 Introduction

The overall objective of this project is to develop and test some well-known methods to solve optimization problems, using both exact and metaheuristic methods, and to compare the results obtained from each one.

2 Exact Method using ILP

The objective of this method was to create **MATLAB** code to generate a **linear programming (LP)** file, which is later fed to **LPSolve IDE**. For that purpose, we created multiple functions to assist us.

2.1 generate_sdn_lp()

This function serves as the orchestrator of the entire process. It loads the graph representation from **Links200.txt** and **L200.txt**, computes the distances between nodes and writes the .lp file to be used in **LPSolve IDE**.

2.2 compute_shortest_paths()

This functions checks if **L200.txt** is already a distance matrix (with **Floyd-Warshall** output) or if its simply an adjacency matrix. If an adjacency matrix is inputted, it runs **Floyd-Warshall** to compute all-pairs shortest paths and replaces any infinite distances with the infinite equivalent of LP.

2.3 floyd_warshall()

An integral part of our code and overall project is this function. Floyd-Warshall is a dynamic programming algorithm that computes the shortest distance between every pair of nodes in a graph. Unlike algorithms like Dijkstra's that find shortest paths from a single source, Floyd-Warshall finds shortest paths between all pairs of vertices simultaneously.

It works by initializing the adjacency matrix provided. Then, in a triple loop structure, it iterates through:

- Each vertex **k**, as they are considered a potential intermediate vertex
- All vertices **i**, considered as source vertices
- All vertices **j**, considered as goal vertices

For each triple (i, j, k), we check if going from i to j via intermediate vertex k gives a shorter path than the current known shortest path from i to j.

If the distance of going from i to j through k is smaller than the current known shortest distance from i to j, then we update the distance from i to j to be equal to the distance from i to k plus the distance from k to j.

2.4 generate_lp_file()

This functions generated the .lp file that defines the objective (minimize the average assignment cost), all the binary declarations, and the constraints:

- 12 controllers
- Each node is assigned to exactly 1 controller
- Assignments can only be made to active controllers
- Two controllers cant be more than 1000 units apart
- A controller must, at least, serve itself

2.5 Results

With the conditions defined in the project guidelines, we obtained:

- **145.22 units** of average assignment cost
- **13.9%** optimality gap

Although we did not obtain the most optimal solution, we believe that, for the allotted time frame, a 13.9% optimality gap is a high-quality solution. We can also conclude from here that the optimal value is **127.49 units**

3 GRASP Algorithm

GRASP (Greedy Randomized Adaptive Search Procedure) is a metaheuristic that combines greedy construction with randomization, followed by local search improvement.

3.1 Implementation

3.1.1 Load and Preparation

We load **L200.txt** and convert it into a distance matrix using Floyd-Warshall algorithm. The constraints include $C_{max} = 1000$ (maximum communication delay) and placing 12 controllers on a 200-node network.

3.1.2 GRASP Construction

The algorithm runs 10 iterations, each with a 30-second timeout. Each iteration consists of:

1. **Construction**: Build a solution by selecting controllers one by one
2. **Local Search**: Improve the solution using controller swapping

During construction, controllers are selected using a Restricted Candidate List (RCL) based on their minimum distance to already selected controllers.

3.1.3 The Alpha Parameter

The **alpha parameter** ($\alpha \in [0, 1]$) controls the trade-off between greedy and random selection. The RCL threshold is:

$$\text{threshold} = g_{min} + \alpha \cdot (g_{max} - g_{min})$$

Alpha Values:

- $\alpha = 0$: Pure greedy - only best candidates selected
- $\alpha = 1$: Pure random - all candidates equally likely
- $0 < \alpha < 1$: Balance between quality and diversification

Low alpha values produce higher-quality but less diverse solutions, while high values increase diversity but may reduce quality.

3.1.4 Parameter Optimization

To find optimal alpha, we tested values from 0 to 1 (step 0.1) using 3 runs of 10 seconds each per alpha value.

3.1.5 Final Script

The final experiment uses the optimized alpha value with 10 runs of 30 seconds each.

3.2 Results

3.2.1 Alpha Optimization Results

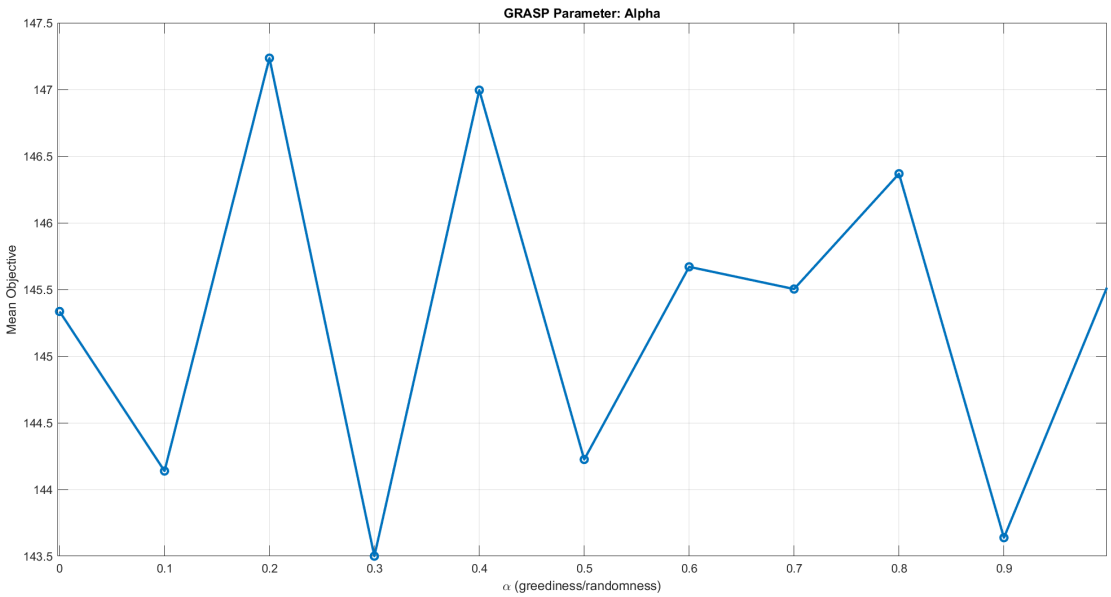

Figure 1: Alpha Optimization: Mean Objective vs. Alpha

The graph shows that $\alpha = 0.3$ provides the best balance between solution quality and algorithm diversification.

3.2.2 Final Results

Using $\alpha = 0.3$, the final results are:

- **Minimum Objective:** 143.0850 Units
- **Average Objective:** 143.6640 Units
- **Maximum Objective:** 144.8900 Units

The GRASP method achieved an **average objective of 143.66 units** with a **12.69% optimality gap**. The small variance between runs (range: 1.80 units) indicates consistent algorithm performance.

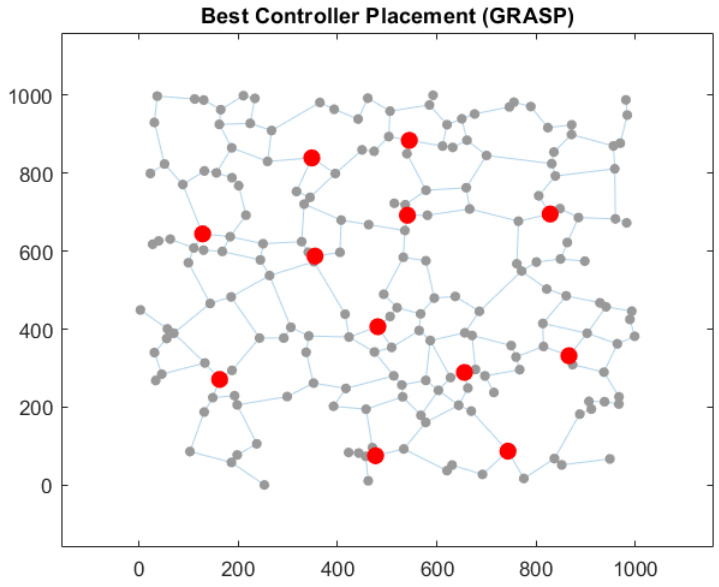

Figure 2: Resultant Best GRASP Solution Graph (red dots are the controllers)

4 Genetic Algorithm

The **genetic algorithm** is a metaheuristic based on the process of natural selection from Darwin's Theory of Evolution. This theory states that species evolve over time through natural selection: individuals with traits better suited to their environment are more likely to survive and reproduce, passing those traits to the next generation — with occasional mutations.

4.1 Implementation

In similarity to the previous exercise's implementation, we start by constructing a distance matrix with the shortest path lengths between all node pairs using the *Floyd-Warshall* method, with the data from *L200.txt*.

The algorithm begins by generating a feasible initial population. It then evolves this population (towards solutions with lower average communication costs) through selection, crossover, and mutation, while preserving a number of elite individuals across generations. This while also ensuring feasibility under the distance constraint of *Cmax=1000* for every new individual. The evolution process continues until the maximum running time of *max_ time=30* is reached.

4.1.1 First Key Components

- **Individual Representation:** Each individual in the population is a set of 12 unique node indices, representing the selected controllers.
- ***Cmax* constraint:** The function ***is_ valid()*** is called in every change to an individual and uses the distance matrix to verify if each node pair fulfills the restraint.
- **Fitness Function:** The function ***evaluate_ population()*** calculates the objective value of an individual which is defined as the average shortest path from each node in the network to its closest controller.

4.1.2 Implementation choices

1. **Selection:** We decided to use **tournament selection** with a size of **2**. In each selection round, the following is repeated 2 times to choose 2 parents: two individuals are randomly chosen from the population, and the one with the lower fitness (better solution) is selected to be a parent. This method is simple but we think it is efficient and helps maintaining diversity within the population.

2. **Crossover:** To generate a new individual (child), the crossover function combines the two parent solutions. It starts by keeping all nodes that are common to both parents. Then, it tries to add more nodes from the union of the two parents, checking each time that the resulting individual remains feasible (doesn't violate *Cmax* constraint). If there are still fewer than 12 nodes, the remaining positions are filled by randomly selecting valid nodes from the network. This method helps preserve useful traits from parents while ensuring the solution remains diverse and valid.

3. **Mutation:** Mutation is applied to individuals with a certain probability. When triggered, the algorithm attempts - up to 10 times (number we decided to be reasonable) - to replace one of the 12 nodes with a randomly selected new node from the network. The replacement is only accepted if the resulting individual is still feasible and contains no repeated nodes. This mutation strategy allows for exploration of new areas in the solution space without compromising solution validity.

4. **Elitism:** We ensure that the best solutions from the current generation are preserved in the next one. At each generation, the algorithm first identifies the top *elitism_count* individuals from the current population — those with the lowest fitness values — and copies them directly into the next generation. Simultaneously, a new population is generated through the same selection, crossover, and mutation process. Once this offspring population is evaluated, the algorithm selects only the best individuals from it — this is the top *population_size - elitism_count* — to complement the remaining elites spots and form the next generation. This means that only the **strongest individuals from the new generation survive**, alongside the elite from the previous one.

4.1.3 Parameter Tuning

To determine the best configuration of parameters for our genetic algorithm, we implemented the tuning function **ga_tune()**. This function tested multiple combinations of the following three key parameters through *10 runs* of *30 seconds* each for each parameter combination:

- **Population Size:** {20, 50, 100, 150, 200}
- **Mutation Probability:** {0.01, 0.05, 0.1}
- **Elitism Count:** {1, 2}

4.2 Results

The parameters values selected as the best solution were the following:

- **Population Size:** 100
- **Mutation Probability:** 0.1
- **Elitism Count:** 1

And the best result of GA, after *10 runs* of *30 seconds* each, for those values was:

1. **Minimum Objective:** 143.1600 Units
2. **Average Objective:** 145.3230 Units
3. **Maximum Objective:** 148.3550 Units

In conclusion, the GA method achieved an average objective of 145.32 units with a 13.99% optimality gap. This algorithm is not as consistent, presenting a wide range of values, which was expected, as it will be shown bellow.

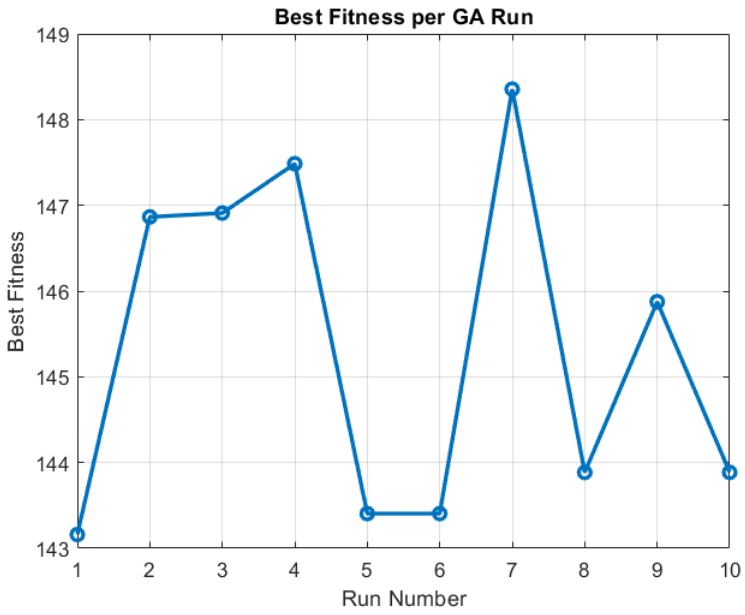

Figure 3: Fitness Solution per Run

The plot above shows the best objective value (fitness) obtained in each of the 10 runs. The fluctuations confirm the stochastic nature of GA — different runs can produce different outcomes due to the random components in selection, crossover, and mutation. Despite the variation, the plot shows that most runs converge to relatively good solutions (between 143 and 148), with the best run achieving the fitness of 143.16.

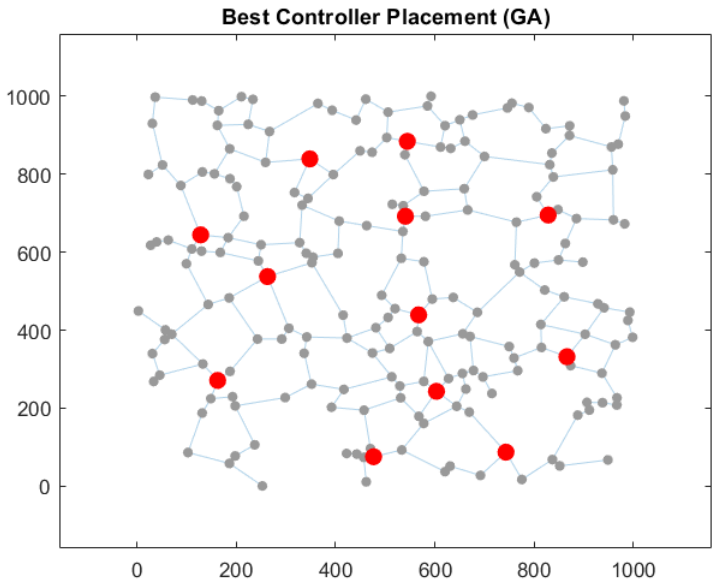

Figure 4: Resultant Best GA Solution Graph (red dots are the controllers)

5 Results Comparison and Conclusion

This section will serve as a agglomeration of all the results obtained during this project.

Algorithm	Objective	Opt. Gap
Exact	145.22 Units	13.90%
GRASP	143.66 Units	12.69%
GA	145.32 Units	13.99%

Table 1: Results Comparison

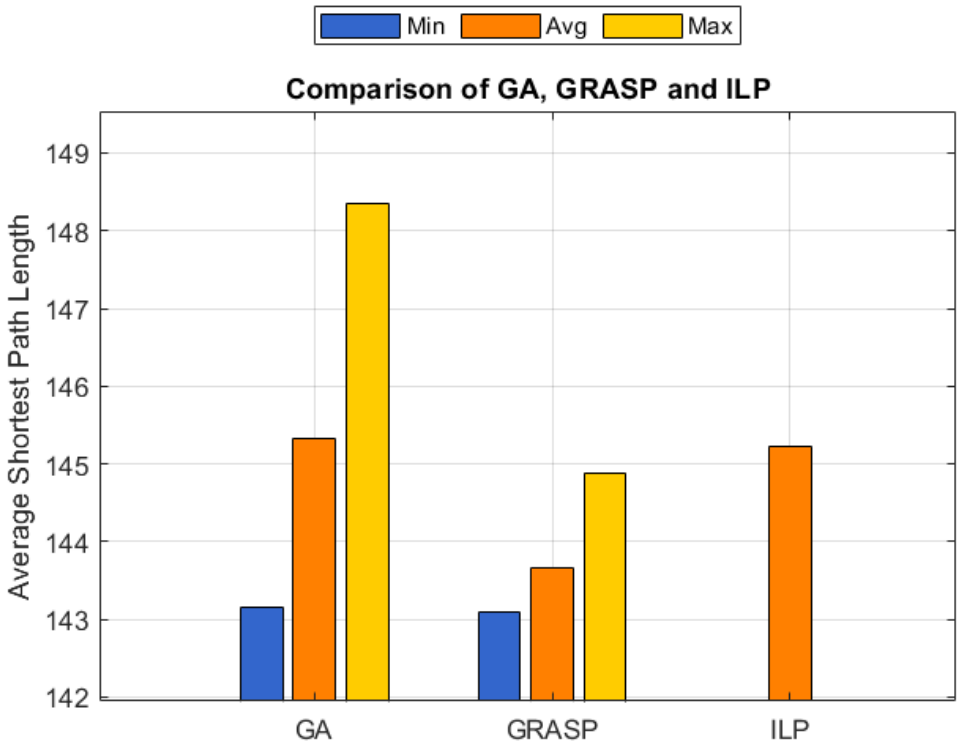

Figure 5: GA vs GRASP vs ILP Results

When comparing the three approaches through the table and bar chart above, we can see that **GRASP achieved the best average performance** (143.66 units) with the lowest optimal-

ity gap (12.69%), while also being the most consistent. The Exact method produced a slightly worse result (145.22 units, 13.90% gap), possibly limited by solver constraints. The GA, despite yielding results close to ILP (145.32 units), showed greater variability across runs. This variability is also evident in the bar chart, where GA has the widest range, GRASP remains stable, and ILP returns a fixed result.

In summary, while the Genetic Algorithm performed slightly worse on average than GRASP, it still proved capable of producing competitive results, especially considering its flexibility. GRASP, however, emerged as the best overall approach in this project, balancing quality, speed, and consistency.