Optimization Mini-Project

Paulo Pereira[98430],Rui Fernandes[92952]

Simulação e Otimização, Universidade de Aveiro, 3810-193 Aveiro, Portugal

1 Introduction

This document covers the approach for the second mini-project assignment in the course *Simulação e Otimização*.

The assignment proposed that two methods were developed to solve an optimization problem, which were: a meta-heuristic method and an exact method based on integer linear programming. And then make a comparative analysis of the solutions and running times of both methods.

2 Problem

The proposed problem was the same for both methods and was defined as follows.

Consider a SDN (Software Defined Network) network with a data plane represented by a graph with N switches. Each link has an associated length.

Considering an integer parameter n¡N and a length parameter Cmax. The aim is the selection of n switches to connect one SDN controller to each of them guaranteeing that the shortest path length between any pair of SDN controllers is not higher than Cmax.

The objective is to minimize the average shortest path length from each switch to its closest controller.

In this case, we were asked to solve an instance of this problem where the graph had 200 nodes and 250 total links, with n=10 and Cmax=1000.

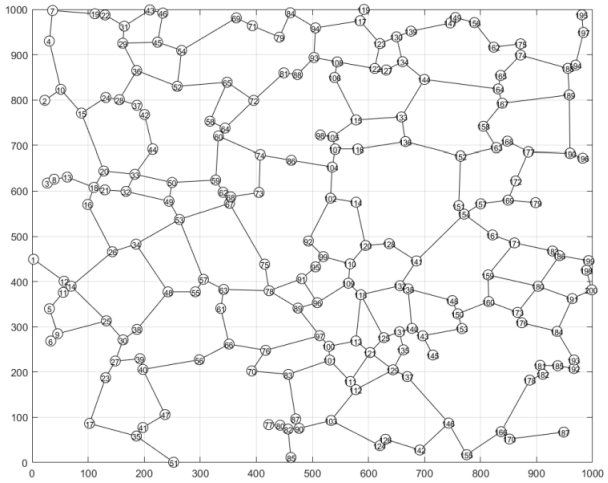

Fig. 1. Results for Normal scenario (2 type A, 1 type B server)

3 Metaheuristic Methods

The first methodology to solve the optimization problem was a choice between two metaheuristic methods taught in the classes, the GRASP algorithm and the Genetic algorithm. Both were implemented in matlab and adapted to solve the proposed problem.

3.1 GRASP Introduction

GRASP (Greedy Randomized Adaptive Search Procedure) is a metaheuristic optimization method that offers a balanced approach to solving combinatorial optimization problems. It consists of two key phases: the construction phase and the local search phase. In the construction phase, a feasible solution is incrementally constructed using a greedy randomized strategy, considering both exploitation and exploration. The local search phase follows, where the Steepest Ascent Hill Climbing method is employed to improve the solution's quality.

The Steepest Ascent Hill Climbing is a local search algorithm that iteratively explores neighboring solutions, selecting the one that provides the highest improvement in the objective function value. Starting from an initial solution, it evaluates the current solution's objective function value and generates neighboring solutions through small modifications. The algorithm moves to the neighboring solution with the greatest improvement, repeating the process until no further improvement is possible. This iterative procedure refines the solution by systematically exploring the local search space.

Integrating the Steepest Ascent Hill Climbing method into the GRASP framework enhances the overall optimization process. After each construction phase,

the resulting solution undergoes the Steepest Ascent Hill Climbing procedure to iteratively improve its quality. This integration allows for the fine-tuning of solutions obtained during the construction phase, potentially reaching higher-quality solutions. By combining the strengths of GRASP's randomized construction with the systematic exploration of Steepest Ascent Hill Climbing, this approach provides a powerful optimization method for combinatorial problems.

3.2 GRASP Implementation

As before said, the GRASP algorithm was implemented in matlab code, and adapted to the given problem. It is partitioned in a few main functions and some auxiliary ones, that will be sucinctly explained.

- `GRASP` Function: This function executes the GRASP algorithm itself. The function generates and initial solution using the greedy-randomized construction method and then improves it iteratively using the Steepest Ascent Hill Climbing (SAHC) algorithm. It returns the best solution found in the defined time along with its corresponding objective function value (The average shortest path length).
- `sahc` Function: The `sahc` function implements the Steepest Ascent Hill Climbing (SAHC) algorithm. The objective of this function is to iteratively improve the initial solution by exploring its neighborhood. It starts with the initial solution and keeps moving to the best neighbor if it leads to an improvement in the objective function value. The function returns the improved solution as well as the `AverageSP_v2` result for it.
- `greedy_random` Function: This function generates a greedy-randomized solution. The function iteratively selects the best node to add to the solution based on the average shortest path values. It ensures that the selected nodes satisfy the `Cmax` constraint. The function returns the generated solution.
- `get_best_neighbour` Function: This auxiliary function is called by the `sahc` function and aims to find the best neighbor of a given solution `s`. The function goes through all neighbors of the given solution and evaluates their objective function values. It selects the neighbor with the lowest value that also satisfies the `Cmax` constraint.
- `get_neighbours` Function: This function is called by the `get_best_neighbour` function and aims to generate all possible neighbors of a given solution `s`. The function iterates over each element in the current solution and each element in the remaining nodes to form all possible neighbors. It returns an array of all generated neighbors.

3.3 Genetic Algorithm Introduction

The Genetic Algorithm (GA) is a popular optimization method inspired by natural selection and genetic evolution. It operates on a population of potential solutions and iteratively improves them over generations. Solutions, represented

as strings of genes, undergo evaluation using an objective function to measure their fitness.

The GA utilizes genetic operators, including selection, crossover, and mutation, to generate new offspring solutions. Selection favors solutions with higher fitness, while crossover combines genetic material from selected parents to create diverse offspring. Mutation introduces random changes, maintaining genetic diversity. These operators drive the exploration and exploitation of the search space.

Through multiple generations of evaluation, selection, crossover, and mutation, the GA gradually improves the population's fitness. Termination occurs when a specified condition is met, such as reaching a maximum number of generations or achieving a desired fitness threshold. The GA's ability to handle complex problems with large search spaces and its capacity to explore diverse regions make it a widely applicable optimization method across various fields.

3.4 Genetic Algorithm Implementation

A basic genetic algorithm (GA) for optimization was implemented in matlab. The goal is to find the best solution that minimizes a given objective function while satisfying certain constraints. In this case the objective function is the average shortest path length and it needs to take into consideration the Cmax constraint.

The GA follows the basic steps of initialization, selection, crossover, and mutation. The algorithm maintains a population of candidate solutions, where each solution is represented as a set of nodes. The best solution is iteratively improved over multiple generations.

Functions The code includes several functions that perform specific tasks:

- `GA`: Implements the main GA algorithm. It initializes the population, performs selection, crossover, and mutation, and returns the best solution found.
- `mutation`: Performs mutation on a solution by randomly changing a node to explore new search space.
- `crossover`: Combines two parent solutions to create a new individual by inheriting characteristics from both parents.
- `parent_selection`: Selects parent solutions based on their fitness probabilities to maintain diversity and promote better solutions.
- `generate_valid_solution`: Generates a valid solution by random permutation until a feasible solution satisfying constraints is found.
- `is_valid`: Checks the validity of a solution by evaluating whether it satisfies the constraints imposed by the problem.

These functions work together to execute the GA, generate and evaluate candidate solutions, and ensure the feasibility of solutions according to the problem constraints.

3.5 Result Comparison

It's important to note that as asked, the algorithms were run ten times, with a set runtime of thirty seconds for each run. Experiments were done with the randomization factor in GRASP and the population size and mutation chance variables in GA to make sure we were using optimal parameters.

For the GRASP algorithm, these were the best solutions found in each thirty-second run:

Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9	Run 10
159.02	159.27	156.08	160.50	160.42	160.54	160.30	161.74	159.17	164.18

Obtaining a minimum value of 156.08, a maximum value of 164.18, and an average value of 160.12.

For the Genetic algorithm, the results were clearly worse, these were the best solutions found in each thirty-second run:

Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9	Run 10
225.88	205.55	200.39	216.79	190.99	202.58	256.14	232.51	196.87	220.46

Obtaining a minimum value of 190.99, a maximum value of 256.14 and an average value of 214.82.

To further visualize the comparison between both algorithms the following graph shows the information in the tables:

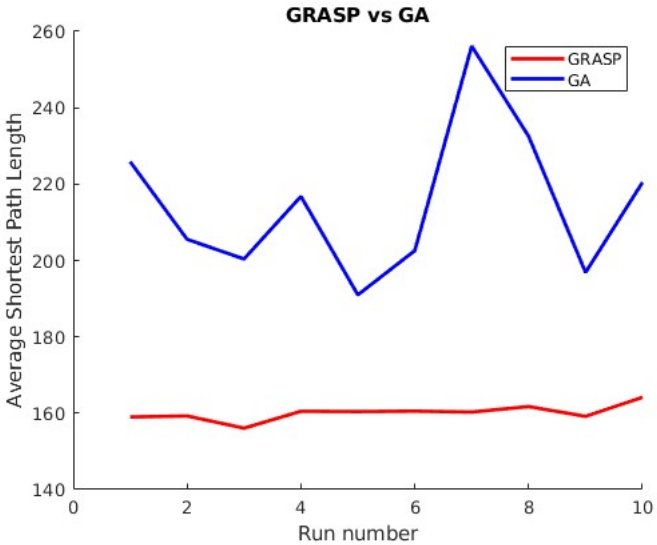

Fig. 2. GRASP vs GA

4 Exact method based on integer linear programming

This portion of the report will explain the thought process and implementation of the method based on integer linear programming.

In this method, we used ILP modeling techniques and four different types of constraints. To solve the problem itself we ran a script on the program *lpsolve*.

4.1 Matlab code

We used MatLab in order to generate the *.lpt* script. The MatLab program read the files *Nodes2.txt*, *Links2.txt* and *L2.txt*, and from these files we created a cost matrix that we further used to create a graph.

After having created the graph, another matrix was created in order to have the distances, in order to calculate these we used the distances function from MatLab with the graph together with two nodes as the arguments. This matrix then was used to create the coefficients in the objective.

The MatLab script then wrote in a *.lpt* file the script that *lpsolve* would later run with the correct syntax. After that, we then run another Matlab script with a call to a function we defined as *plotTopology* in order to create a visual representation of the graph with the selected nodes clearly visible.

4.2 lpsolve

As previously mentioned the *lpsolve* file was written by the MatLab script.

This file is divided into different parts.

- **Objective** - this portion of the script defines the objective function for our script, in our case the objective was to minimize the average shortest path length from each switch to its closest controller, as mentioned in the introduction.
- **Constraints** - as mentioned we defined four types of constraints
- **Binary** - this portion of the file simply defines all variables as binary.

Objective As mentioned, the objective was to minimize the average shortest path length from each switch to its closest controller.

To do this we made the sum of the distances of all paths, each path was defined as *g1_2* under the minimize keyword of *lpsolve*. Since the path variables are binary if a path was active it would become 1 and if not it would be 0, this way we only made the sum of the active paths.

This sum was then divided by 200, which is the number of nodes present in the graph.

Constraints The constraint section consists of the vast majority of the lines inside of the *.lpt* script, but it can be divided into four components.

The first of which defines how many server nodes must be selected, to accomplish this we make the **sum of all x variables and make sure it equals n**, in our case, n is 10.

The second constraint makes sure one server must be assigned to each node, in the **g** variables the first number can be interpreted as the controller and the second number as a switch, in this constraint we create a sum of all the g variables for each switch and equal it to one, meaning each switch will have one and only one controller.

The third constraint ensures that the controller associated with a switch is, in fact, a controller, this is done by solving the inequality $gi_s <= xi$, since this exact notation isn't possible in the file format we simply defined it as $gi_s - xi <= 0$.

Finally, the last constraint is the one that defines that the path between two controllers can't be more than Cmax (in our case 1000). To create this constraint we created in Matlab a for loop that will, for each pair of nodes, check if their distance is bigger than Cmax, if that is the case it will then be written a constraint that looks like $xi + xj <= 1$, this means that both nodes i and j can't be selected at the same time.

Binary In the introduction to this section, it was mentioned that all variables would be binary, this means that there was the need to use the **binary** keyword in order to define each x and g variables as binary.

4.3 Results

In the document it was recommended to run the script for 5 minutes, this result will be used to compare to the other implemented methods, but we were also curious to find better solutions by giving *lpsolve* more time to compute a better solution.

5 minutes of timeout Using 5 minutes as a timeout in *lpsolve* the best solution presented provided us with an average shortest path of 161.245, and the selected nodes [18,27,65,78,90,104,108,121,152,173].

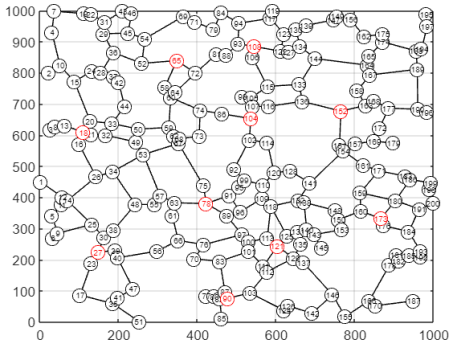

Fig. 3. Results for Normal scenario (2 type A, 1 type B server)

10 and 15 minutes of timeout Using 10 or 15 minutes as a timeout (they reached the same solution) in *lpsolve* the best solution presented provided us with an average shortest path of 161.215, and the selected nodes [18,27,65,78,90,104,108,121,152,173]. Despite the improvement being very small, we found this case curious because the selected nodes are the same. We consider that the average lowered because one switch got assigned a different controller.

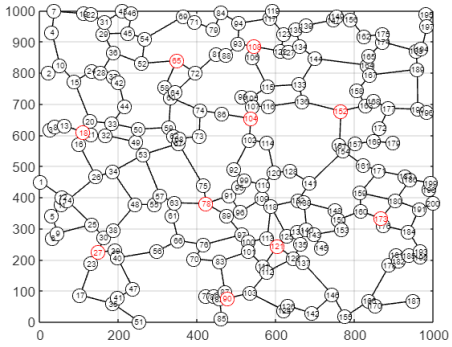

Fig. 4. Graph after 5 minutes of search

30 minutes of timeout Using 30 minutes as a timeout in *lpsolve* the best solution presented provided us with an average shortest path of 156.55, and the selected nodes [18,30,65,78,103,174,122,131,173,177].

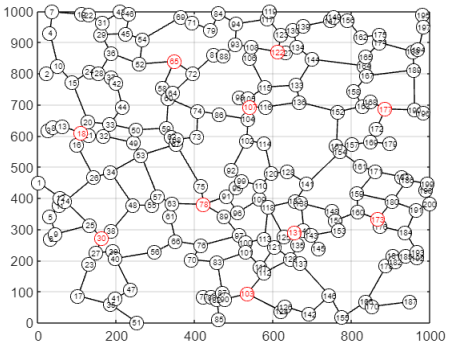

Fig. 5. Graph after 10 and 15 minutes of search)

No timeout Without any timeout on *lpsolve*, we got major changes at 3 hours of run time as we received a new solution with an average shortest path of 155.995, and the selected nodes [18,30,65,78,103,107,122,131,173,177]. Only after 2 more hours, we got another solution (A total of 5 hours had elapsed) this time the average shortest path was 155.985 but the nodes changed to [20,30,65,68,90,108,110,129,163,173]. After 7 total hours of elapsed time, another improvement was made, this time the objective function returned a value of 155.37 and the controllers changed to [20,30,65,78,103,107,122,131,173,177].

At the 500-minute mark, we decided to stop the execution, the final results were the same as at the 7-hour mark.

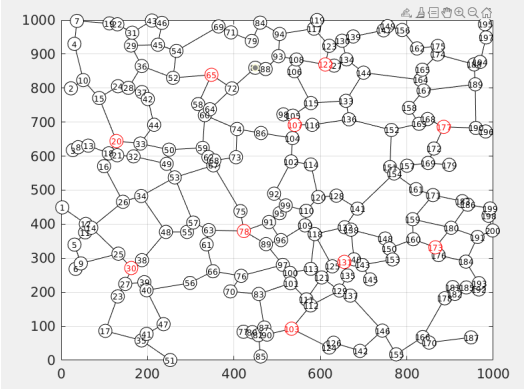

Fig. 6. Best solution graph

Here we can see the product of the method at different points in time using this method.

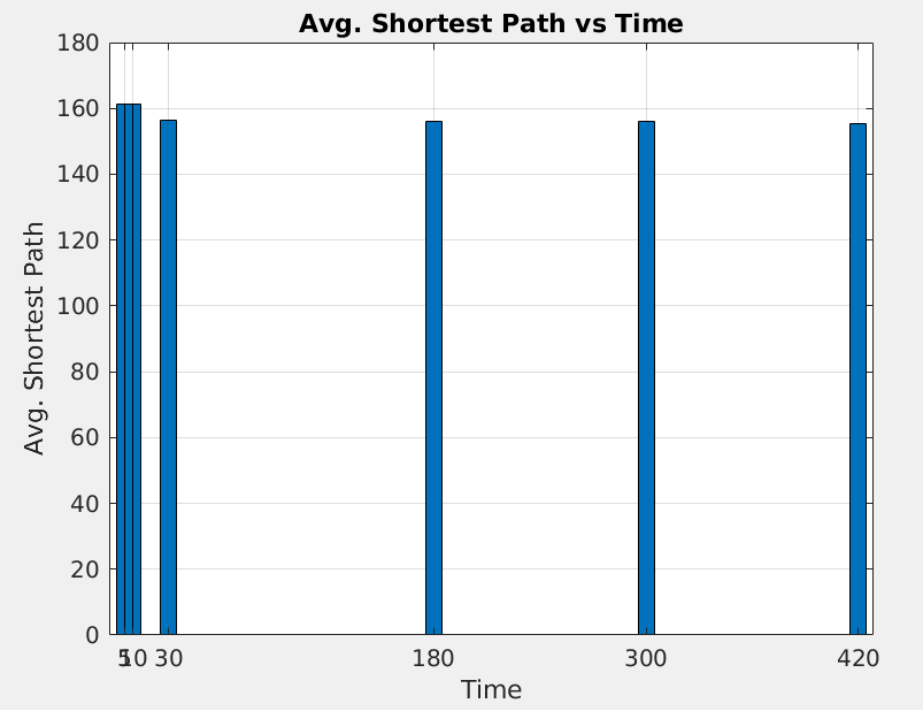

Fig. 7. Results at points in time

5 Result comparison between methods

In this section, we will compare the results of all implementations, and draw the final conclusions.

For a fair comparison, we will evaluate every implementation under the same conditions, this means that since both metaheuristic implementations ran 10 times for 30 seconds each, we will consider the 5 minutes time frame for the exact method based on ILP.

Implementation	ILP	GRASP	Genetic
Avg. Shortest Path	161.245	156.080	190.990

Table 1. Comparison between implementations inside a 5-minute time frame.

As we can see in the table above, in the given time frame, GRASP produced the best value and the genetic algorithm provided the worst.

Despite the fact that GRASP came out ahead in the previous comparison when taking a look at the results provided by the exact method based on ILP, we

can conclude that given enough time this method will provide the best results, since after 10 minutes both methods produced the same average.