Optimization Mini-Project

Mestrado em Engenharia Informática

Universidade de Aveiro

DETI - Aveiro, Portugal

2023/2024

SO - Simulação e Otimização

Guilherme Antunes [103600] (50%), Pedro Rasinhas [103541] (50%)

Index

1 Approach 1 - Metaheuristic Method - GRASP **3**
1.1 Brief Description 3
1.1.1 Pseudocode 3
1.2 Implementation 4
1.3 Results 8
1.4 Conclusions 9
2 Approach 1 - Metaheuristic Method - GeneticAlgorithm **10**
2.1 Brief Description 10
2.1.1 Pseudocode 10
2.2 Implementation 11
2.3 Results 16
2.4 Conclusions 22
3 Approach 2 - Exact method **23**
3.1 Brief Description 23
3.2 Implementation 23
4 Results discussion **29**
5 References and resources **31**

1 Approach 1 - Metaheuristic Method - GRASP

1.1 Brief Description

The Greedy Randomized Adaptive Search Procedure (GRASP) is a multi-start metaheuristic method used for combinatorial optimization problems. It is a combination of a *Greedy Randomized* method, to compute good initial solutions, and of an *Adaptive Search* method that starts from each initial solution aiming to find its closest local optimum solution. The *Adaptive Search,* in its original proposal, is the *Steepest Ascent Hill Climbing* method.

1.1.1 Pseudocode

$s \leftarrow GreedyRandomized()$

$s_{best} \leftarrow AdaptiveSearch(s)$

$While\ not\ (stopping\ condition)\ do$

$\quad s \leftarrow GreedyRandomized()$

$\quad s \leftarrow AdaptiveSearch(s)$

$\quad If\ f(s)\ better\ than\ f\ (s_{best})\ do$

$\quad\quad s_{best} \leftarrow s$

$\quad EndIf$

$EndWhile$

The pseudocode for GRASP begins with the initialization phase. Initially, a solution s is generated using the *GreedyRandomized*() function. This function constructs the solution incrementally by randomly selecting elements from a restricted candidate list, which includes elements that provide the best improvements to the objective function. The initial solution s is then refined using the *AdaptiveSearch*(s) function, which typically employs the *Steepest Ascent Hill Climbing* method, resulting in s_{best}, the best solution found so far.

Following the initialization, the algorithm enters a loop that continues until a specified stopping condition is met, in our case a runtime limit. In each iteration, a new initial solution s is generated using the *GreedyRandomized*() function.

This new solution s is then refined using the AdaptiveSearch(s) function to find a local optimum. The fitness of the new solution s is compared to the fitness of the best solution s_{best} found so far. If the new solution s is better, s_{best} is updated to this new solution.

The loop continues, generating and refining new solutions, until the stopping condition is met. Once the stopping condition is reached, the algorithm terminates, and s_{best}, the best solution found, is returned.

1.2 Implementation

The implementation of the GRASP method is as follows:

```
% Load data
Nodes = load('Nodes200.txt');
Links = load('Links200.txt');
L = load('L200.txt');

% Create Graph
nNodes = size(Nodes, 1);
% disp(nNodes);
G = graph(L);


function [best_solution, best_value, timeTaken] = GRASP(G, c, max_time, r)
    t_start = tic;

    solution = GreedyRand(G, c, r);
    [best_value, best_solution, duration] = steepestAscentHillClimbing(G, solution, @bestNeighbor1);
    timeTaken = duration;
    while toc(t_start) < max_time
        % shuffle seed
        rng('shuffle');

        % Constructive Phase (Greedy Randomized)
        solution = GreedyRand(G, c, r);

        % Search Phase (Adaptive Search - SA-HC)
        [value, solution, duration] = steepestAscentHillClimbing(G, solution, @bestNeighbor1);
        % [value, solution, duration] = steepestAscentHillClimbing(G, solution, @bestNeighbor2);
        %fprintf("Solution: %s, Value: %f\n", mat2str(solution), value);
        % Evaluate the solution
        if value < best_value
            best_solution = solution;
            best_value = value;
            timeTaken = duration;
        end
    end

end
```

Figure 1 - GRASP main loop

As shown in **Figure 1,** we can see the *GRASP* function that takes in the graph (*G*), the number o critical nodes (*c*) the time limit for a single run, and the *r* value, that is used in the *GreedyRandomized* function, which will be explained below.

```
% Greedy Randomized function
function sol = GreedyRand(G, c, r)
    E = 1:numnodes(G);
    sol = [];
    for i = 1:c
        R = [];
        for j=E
            R = [ R ; j ConnectedNP(G, [sol j]) ];
        end
        R = sortrows(R, 2);
        e = R(randi(min(r, size(R, 1))), 1); % Ensure r is within bounds
        sol = [sol e];
        E = setdiff(E, e);
    end

    %fprintf('Greedy Randomized solution: %s\n', mat2str(sol));
end
```

Figure 2 - *GreedyRandomize* function

The *GreedyRandomize* function takes the graph, the number of critical nodes and the *r* that controls the level of randomization (i.e., the number of best candidates to consider at each iteration). This method overcomes the limitation of a purely Greedy method, which always generates the same solution, by introducing randomness in the selection process. It aims to generate a good randomized solution by iteratively selecting elements. The function starts by initializing the set E with all nodes and an empty solution sol. For each of the c iterations, it computes the RCL, which contains the top r elements from E that, when added to sol, yield the best partial solutions based on $ConnectedNP(G, [sol\, j])$. A new element e is randomly selected from the RCL R, ensuring diversification in the selection process. The selected element e is then added to the solution $sol$and removed from the set E. By the end of the loop, the function constructs a randomized solution sol that includes c critical nodes.

```
% SA-HC function
function [nodesConnected, bestSelected, iteractions, duration] = steepestAscentHillClimbing(G, bestSelected, selectFunction)
    nodesConnected = inf;
    improved = true;

    while improved
        improved = false;
        iteractions = iteractions + 1;
        [bestNeighbor, connected] = selectFunction(G, bestSelected);
        if connected < nodesConnected
            nodesConnected = connected;
            bestSelected = bestNeighbor;
            improved = true;
            % fprintf('Improved in local search: %s with cost %f\n', mat2str(bestSelected), nodesConnected);
        end

    end
    % fprintf('Best solution: %s, cost: %f\n', mat2str(solution), nodesConnected);

    duration = toc(t);
end
```

Figure 3 - SA-HC function

The *steepestAscentHillClimbing* function, described in Figure 3 is designed to refine a given solution to a local optimum by iteratively exploring its neighborhood. The function takes a graph G, an initial selection of critical nodes $bestSelected$, and a $selectFunction$ that evaluates and selects the best neighboring solution.

Initially, the function sets $nodesConnected$ to infinity and $improved$ to true, indicating that improvement is possible. The function then enters a while loop that continues as long as improvements are being found.

In each iteration of the loop, the function sets $improved$ to false and increments the iteration counter. It then calls $selectFunction$, which examines the neighbors of the current solution $bestSelected$ and returns the best neighboring solution $bestNeighbor$ along with its associated cost $connected$.

If the cost $connected$of the new solution $bestNeighbor$is lower than the current best cost $nodesConnected$, the function updates $nodesConnected$ and $bestSelected$ with the new values, and sets $improved$to true, indicating that a better solution has been found.

This process repeats until no better neighboring solution can be found, at which point $improved$ remains false, and the loop terminates.

This is the same as the Hill Climbing method, except that in the SA-HC, at each iteration, the best neighbor solution becomes the current solution.

As mentioned above, the *steepestAscentHillClimbing* function takes a $selectFunction$. This $selectFunction$ was applied in two different ways:

1. $bestNeighbor1$ - where a neighbor solution is obtained by swapping a node in the current solution by a node not in the current solution;
2. $bestNeighbor2$ - where a neighbor solution is obtained by swapping a node in the current solution by a neighbor node not in the current solution;

Both approaches, described in Figure 4, were tested, and the results are shown in the next section.

```
% a neighbor solution is obtained by swapping a node in the current solution by a node not in the current solution
function [bestNeighbor, bestConnected] = bestNeighbor1(G, current)
    nNodes = numnodes(G);
    others = setdiff(1:nNodes, current);
    bestConnected = inf;
    bestNeighbor = current;
    for a = current
        for b = others
            neighbor = [setdiff(current, a), b];
            connected = ConnectedNP(G, neighbor);
            if connected < bestConnected
                bestConnected = connected;
                bestNeighbor = neighbor;
            end
        end
    end
end

% a neighbor solution is obtained by swapping a node in the current solution by a neighbor node not in the current solution
function [bestNeighbor, bestConnected] = bestNeighbor2(G,current)
    bestConnected = inf;
    bestNeighbor = current;
    for a = current
        others = setdiff(neighbors(G,a)',current);  % get the neighbors of the current node that are not in the current solution
        for b = others
            neighbor = [setdiff(current, a), b];    % swap a node in the current solution by a neighbor node not in the current solution
            connected = ConnectedNP(G,neighbor);
            if connected < bestConnected
                bestConnected = connected;
                bestNeighbor = neighbor;
            end
        end
    end
end
```

Figure 4- *SelectFunction* used in SA-HC

```
% best settings
r = 50;
c_values = [8, 10, 12];
max_time = 60; % seconds
% Create results folder if it doesn't exist
results_folder = 'results_time_GRASP';
if ~exist(results_folder, 'dir')
    mkdir(results_folder);
end

for c = c_values
    results = zeros(10, 2); % Initialize results matrix for 10 runs
    for run = 1:10
        [~, best_value, timeTaken] = GRASP(G, c, max_time, r);
        results(run, 1) = best_value;
        results(run, 2) = timeTaken;

        fprintf('Run %d/%d for c=%d completed\n', run, 10, c);
        fprintf("Best value: %f, Time taken: %f\n", best_value, timeTaken)

    end
    % Write results to file
    filename = sprintf('%s/GRASP_res_%d_%d.txt', results_folder, c, r);
    save(filename, 'results', '-ascii', '-tabs');
    fprintf('Results saved to %s\n', filename);
end
```

Figure 5- Main settings for the GRASP program

In Figure 5**,** we can see how we defined the different configurations to run the program with, how we ran the program and how we stored the results.

1.3 Results

To find the best settings for the GRASP method we ran, 10 times for each value of *c*, the program with 4 different values of *r* (3, 5, 10, 50). We will omit these results here, as they are too extensive, and will just present the ones with the best settings, for $r = 50$.

We also implemented two variations of the $selectFunction$ described above, and for those settings we tested both variations and obtained the results described by Table 1 and Table 2.

N_run	c = 8	TimeTaken (s)	c = 10	TimeTaken (s)	c = 12	TimeTaken (s)
1	5519.00	16.42	**3571.00**	22.85	3108.00	33.55
2	**4790.00**	18.05	**3571.00**	23.15	3108.00	30.39
3	**4790.00**	18.21	**3571.00**	26.11	3108.00	20.60
4	**4790.00**	17.42	4163.00	21.20	3108.00	27.69
5	**4790.00**	18.56	4163.00	14.31	3088.00	34.06
6	**4790.00**	18.27	4163.00	24.05	3108.00	35.47
7	**4790.00**	13.46	**3571.00**	23.82	3108.00	31.56
8	**4790.00**	15.64	5565.00	20.78	3108.00	24.89
9	**4790.00**	20.31	**3571.00**	22.57	3123.00	34.20
10	5519.00	18.30	**3571.00**	22.08	3108.00	23.28
Average	4862.90	17.66	3924.50	22.39	3108.60	29.56

Table 1 - results for GRASP, with configuration ($r = 50$) using $bestNeighbor1$;

N_run	c = 8	c = 10	c = 12
1	9766.00	3571.00	3007.00
2	11164.00	4163.00	3733.00
3	9766.00	4163.00	3770.00
4	4790.00	4181.00	3733.00
5	8665.00	6539.00	3108.00
6	9100.00	4163.00	3108.00
7	9766.00	4163.00	3874.00
8	9255.00	7508.00	3108.00
9	5518.00	4163.00	3108.00
10	9976.00	4163.00	3108.00
Average	**8775.70**	**4673.60**	**3464.70**

Table 2 - results for GRASP, with configuration ($r = 50$) using $bestNeighbor2$;

In Table 2 we decided not to include the time taken, as they were very similar to the ones in Table 1, and therefore, the main point of comparison would be the objective values.

1.4 Conclusions

The results reveal that the $bestNeighbor1$ selectFunction consistently outperforms bestNeighbor2, providing lower and more reliable objective values across all values of c (8, 10, and 12). This indicates that selecting a random node not in the solution yields better results than selecting a neighboring node not in the solution.

The time taken increases with the number of critical nodes c, as expected, due to the increased complexity of the problem. The best configuration was found with $r = 50$, suggesting that a larger RCL size effectively balances randomness and greediness in generating good initial solutions.

Overall, using $bestNeighbor1$ with $r = 50$ is the most effective GRASP configuration for solving the CND problem (from the configurations tested), providing consistent and high-quality solutions, as we will discuss later.

2 Approach 1 - Metaheuristic Method - GeneticAlgorithm

2.1 Brief Description

A Genetic Algorithm (GA) is a population-based metaheuristic optimization method inspired by the process of natural selection from Darwin's theory of evolution.

It is particularly effective for solving complex optimization problems where traditional methods may struggle. GAs are characterized by their ability to maintain and improve a population of solutions across iterations, simulating the process of evolution through selection, crossover, and mutation.

2.1.1 Pseudocode

$P \leftarrow Compute\ |P|\ random\ individuals$
$While\ not\ (stopping\ condition)\ do$
    $P' \leftarrow \{\}$
    $For\ i\ =\ 1 \ldots |P|\ do$
        $s \leftarrow Crossover(P)$
        $If\ random([0,1])\ <\ q$
            $s \leftarrow Mutation(s)$
        $EndIf$
        $P' \leftarrow P' \cup \{s\}$
    $EndFor$
    $P \leftarrow Selection(P,\ P')$
$EndWhile$
$s_{best} \leftarrow Best(P)$

The algorithm begins by generating an initial population of potential solutions randomly, where each solution represents a possible answer to the optimization problem. The quality of each solution is then assessed using a fitness function, which quantifies how well it meets the optimization criteria.

Based on their fitness values, individuals are selected for reproduction, favoring better solutions to increase their chances of passing genes to the next generation. Pairs of selected individuals are combined through crossover to produce new solutions, and with a certain probability, mutation introduces

random changes to maintain genetic diversity and prevent premature convergence.

Elitism ensures that the best-performing solutions are preserved and carried over to the next generation unchanged. This cycle of selection, crossover, mutation, and fitness evaluation is repeated for a set number of generations or until a convergence criterion is met, allowing the population to evolve towards better solutions.

When the stopping condition is satisfied, the algorithm evaluates the final population and selects the best solution found, returning it as the output of the algorithm.

2.2 Implementation

```
function [bestSolution, bestCost] = GeneticAlgorithmCND(NodesFile, LinksFile, LFile, c, runtimeLimit, populationSize, mutationRate, elitismCount)
    % Load the graph data
    Nodes = load(NodesFile);
    Links = load(LinksFile);
    L = load(LFile);
    nNodes = size(Nodes, 1);
    % Create graph
    G = graph(L);

    % Initialize population
    population = InitializePopulation(populationSize, nNodes, c);

    startTime = tic;
    while toc(startTime) < runtimeLimit
        % Initialize new population - P'
        newPopulation = zeros(populationSize, c);
        costs = EvaluateCosts(G, population);

        for i = 1:populationSize
            s = Crossover(population, costs, nNodes, c);

            if rand < mutationRate
                s = Mutation(s, nNodes);
            end

            newPopulation(i, :) = s;
        end

        % Update population
        population = Selection(G, population, newPopulation, elitismCount);
    end

    costs = EvaluateCosts(G, population);
    [~, bestIdx] = min(costs);
    bestSolution = population(bestIdx, :);
    % cost for the best solution
    bestCost = ConnectedNP(G, bestSolution);

    % Output the best solution
    fprintf("\nGenetic Algorithm for CND Problem\n");
    fprintf("C value: %d\n", c);
    fprintf("Best solution: %s\n", mat2str(bestSolution));
    fprintf("Best cost: %f\n", bestCost);

    plotTopology(Nodes, Links, bestSolution);
end
```

Figure 6 - *GeneticAlgorithm* function

The function described in Figure 6 begins by loading the graph data from specified files, including node coordinates, links, and link lengths, and constructs a graph G. It then initializes a population of potential solutions, where each solution is a set of c nodes randomly selected from the graph.

The main loop of the GA runs until the specified runtime limit is reached. During each iteration, the fitness (cost) of each solution in the population is evaluated. New solutions are generated through crossover, which combines pairs of parent solutions, and mutation, which introduces random changes based on the mutation rate. The population is then updated by selecting the best individuals from the combined old and new populations, with the top individuals carried over to the next generation through elitism.

After the runtime limit is reached, the function identifies the best solution in the final population based on its fitness.

```
function population = InitializePopulation(populationSize, nNodes, c)
    population = zeros(populationSize, c);
    for i = 1:populationSize
        individual = randperm(nNodes, c);
        while length(unique(individual)) < c
            individual = randperm(nNodes, c);
        end
        population(i, :) = individual;
    end
end

function costs = EvaluateCosts(G, population)
    populationSize = size(population, 1);
    costs = zeros(populationSize, 1);
    for i = 1:populationSize
        costs(i) = ConnectedNP(G, population(i, :));
    end
end
```

Figure 7 - Functions InitializePopulation and EvaluateCosts

As described in Figure 7**,** the *InitializePopulation* function creates the initial population of potential solutions. Each solution is an array of c unique nodes randomly selected from the total nNodes nodes in the graph, ensuring diversity in the initial population.

On the other hand, the *EvaluateCosts* function calculates the fitness of each solution in the population. For each individual, it computes the cost using the ConnectedNP function, which measures the impact of removing the selected nodes on network connectivity. The resulting costs are stored in a vector, guiding the Genetic Algorithm's selection, crossover, and mutation processes.

On Figure 8 we can see the *Crossover* operator that combines the genes of two parent individuals to generate the genes of an offspring individual. It is composed of two main steps: the parent selection and the gene combination.

The parent selection aims to select two random parents from the current population giving higher probability to the most fit individuals. This step can follow one of the following strategies, represented in Figure 9**:**

- Fitness based selection - where each parent is selected with a probability proportional to its fitness;
- Tournament selection - where, for each parent, two random individuals are selected and then choose the most fit among the two as the parent;
- Rank based selection - where all individuals are ranked by their fitness and then each parent is randomly selected, with a probability proportional to its rank;

```
function offspring = Crossover(population, costs, nNodes, c)
    % Parent Selection - Choose one of the strategies
    parents = TournamentSelection(population, costs);
    % parents = FitnessBasedSelection(population, costs);
    % parents = RankBasedSelection(population, costs);

    crossoverPoint = randi(c - 1);
    offspring = [parents(1, 1:crossoverPoint), parents(2, crossoverPoint+1:end)];
    offspring = unique(offspring, 'stable');

    % Ensure the offspring has exactly c unique nodes
    while length(offspring) < c
        newGene = randi(nNodes);
        if ~ismember(newGene, offspring)
            offspring = [offspring, newGene];
        end
    end
end
```

Figure 8 - *Crossover* operator

```
% Parent Selection strategies
function parents = TournamentSelection(population, costs)
    % Tournament selection for choosing two parents
    % for each parent, select two random individuals and choose the most fit among the two as the parent
    parents = zeros(2, size(population, 2));
    for i = 1:2
        competitors = randperm(size(population, 1), 2);
        if costs(competitors(1)) < costs(competitors(2))
            parents(i, :) = population(competitors(1), :);
        else
            parents(i, :) = population(competitors(2), :);
        end
    end
end

function parents = FitnessBasedSelection(population, costs)
    % Fitness-based selection for choosing two parents
    % select each parent with a probability proportional to its fitness
    [~, sortedIdx] = sort(costs);
    parents = population(sortedIdx(1:2), :);
end

function parents = RankBasedSelection(population, costs)
    % Rank-based selection for choosing two parents
    % rank all individuals by their fitness value and
    % randomly select each parent with a probability proportional to its rank

    % Rank the combined population based on costs
    [~, sortedIdx] = sort(costs);

    % Calculate selection probabilities
    selectionProbabilities = 1:length(population);
    selectionProbabilities = selectionProbabilities / sum(selectionProbabilities);

    % Select parents based on probabilities
    parents = zeros(2, size(population, 2));
    for i = 1:2
        selectedIdx = randsample(1:length(population), 1, true, selectionProbabilities);
        parents(i, :) = population(selectedIdx, :);
    end

    % Return the selected parents
    parents = population(sortedIdx(1:2), :);
end
```

Figure 9 - Crossover strategies

The gene combination step creates an offspring by randomly selecting genes from each parent, ensuring that the new solution inherits characteristics from both parents.

```
function mutatedOffspring = Mutation(offspring, nNodes)
    % One gene of the offspring individual is randomly mutated
    mutationPoint = randi(length(offspring));
    newGene = randi(nNodes);
    while ismember(newGene, offspring)
        newGene = randi(nNodes);
    end
    mutatedOffspring = offspring;
    mutatedOffspring(mutationPoint) = newGene;
end
```

Figure 10 - Mutation function

The mutation function, described by Figure 10, introduces random changes to an offspring individual to maintain genetic diversity. It randomly selects a gene in the offspring (mutationPoint) and replaces it with a new gene that is not already in the offspring, ensuring the mutated offspring has unique genes and promoting variation in the population.

```
function selected = Selection(G, population, newPopulation, elitismCount)
    % Selection operator:
    % Selects the |P| individuals s from set P U P' with the best value of f(s)
    % f(s) in this case is the ConnectedNP function (s)
    % Limiting the number of elitist individuals (i.e., elements of P) to a maximum value m (elitismCount)

    % all_p = population U newPopulation
    fullPopulation = [population; newPopulation];
    % Make sure this is a set by removing duplicates
    fullPopulation = unique(fullPopulation, 'rows');

    % Evaluate the fitness of each individual
    fullCosts = EvaluateCosts(G, fullPopulation);

    % Sort the population based on costs
    [~, sortedIdx] = sort(fullCosts);

    % Select the best m individuals
    selected = fullPopulation(sortedIdx(1:elitismCount), :);

    % Fill the rest of the selected population
    remainingCount = size(population, 1) - elitismCount;
    for i = 1:remainingCount
        competitors = randperm(size(fullPopulation, 1), 2);
        if fullCosts(competitors(1)) < fullCosts(competitors(2))
            selected(elitismCount + i, :) = fullPopulation(competitors(1), :);
        else
            selected(elitismCount + i, :) = fullPopulation(competitors(2), :);
        end
    end
end
```

Figure 11- Selection phase

The Selection function, in Figure 11 updates the population by choosing the best individuals from the combined set of the current population and the new offspring. It begins by merging the two populations and removing any duplicates. The fitness of each individual is then evaluated using the EvaluateCosts function. The population is sorted based on these fitness values, and the top *elitismCount* individuals are selected directly. The remaining spots in the population are filled by comparing pairs of randomly selected individuals and choosing the fitter one, ensuring that only the best solutions are carried forward to the next generation. This process maintains high-quality solutions while promoting diversity.

```
Nodes_file = 'Nodes200.txt';
Links_file = 'Links200.txt';
L_file = 'L200.txt';

% Genetic algorithm configurations
configs = {
    struct('populationSize', 100, 'mutationRate', 0.2, 'elitismCount', 10),
    struct('populationSize', 100, 'mutationRate', 0.2, 'elitismCount', 50),
    struct('populationSize', 100, 'mutationRate', 0.2, 'elitismCount', 5),
    struct('populationSize', 100, 'mutationRate', 0.5, 'elitismCount', 10),
    struct('populationSize', 100, 'mutationRate', 0.5, 'elitismCount', 50),
    struct('populationSize', 100, 'mutationRate', 0.5, 'elitismCount', 5),
    struct('populationSize', 100, 'mutationRate', 1.0, 'elitismCount', 10),
    struct('populationSize', 100, 'mutationRate', 1.0, 'elitismCount', 50),
    struct('populationSize', 100, 'mutationRate', 1.0, 'elitismCount', 5),
    struct('populationSize', 200, 'mutationRate', 0.2, 'elitismCount', 10),
    struct('populationSize', 200, 'mutationRate', 0.2, 'elitismCount', 50),
    struct('populationSize', 200, 'mutationRate', 0.2, 'elitismCount', 5)
};

c_values = [8, 10, 12];
runtime_limit = 60; % seconds

% Loop through each configuration
for k = 1:length(configs)
    config = configs{k};
    populationSize = config.populationSize;
    mutationRate = config.mutationRate;
    elitismCount = config.elitismCount;
    % Run the GA for different values of c and store the results
    results = zeros(10, length(c_values));
    for j = 1:10
        for i = 1:length(c_values)
            % Run the GA
            [bestSolution, bestCost] = GeneticAlgorithmCND(Nodes_file, Links_file, L_file, c_values(i), runtime_limit, populationSize, mutationRate, elitismCount);
            % Store the best cost in the results matrix
            results(j, i) = bestCost;
        end
    end

    % Save the results to a file
    resultsFile = sprintf('GA_results_%d_%.1f_%d.txt',populationSize, mutationRate, elitismCount);
    save(resultsFile, 'results', '-ascii', '-tabs');
    fprintf('Results saved to %s\n', resultsFile);
end
```

Figure 12 - Program Initialization

The script described in Figure 12 sets up and runs multiple configurations of the Genetic Algorithm (GA) for the Critical Node Detection (CND) problem. It begins by defining file paths for the nodes, links, and link lengths of the graph. Various GA configurations are specified, varying in population size, mutation rate, and elitism count. For each configuration, the script runs the GA for different values of c (number of nodes to be removed) and stores the results.

The script loops through each configuration and runs the GA ten times for each value of c. After running the GA, the best cost is recorded for each run. The results are then saved to a file, with the filename indicating the population size, mutation rate, and elitism count used in that configuration. Note that besides these configurations, we still tested the different typical strategies for the crossover operator, which, as expected, provided different results.

2.3 Results

Before choosing the best settings, we opted to verify which strategy to use in the crossover operator would produce the best results, for this we tested all three strategies, for two different settings: ($populationSize = 100$; $mutationRate = 0.5$; $elitismCount = 10$) and ($populationSize = 100$; $mutationRate = 0.2$; $elitismCount = 5$); We also opted not to print the time elapsed for each run, as our

only stopping condition is the time elapsed (60 seconds), the time values are all above but really close to 60 seconds (as expected and according to our program).

- ***TournamentSelection*** strategy:

N_run	c = 8	c = 10	c = 12
1	6652.00	7222.00	4702.00
2	8665.00	**3571.00**	3108.00
3	9754.00	**3571.00**	3088.00
4	**4790.00**	**3571.00**	3088.00
5	6055.00	**3571.00**	3076.00
6	9901.00	**3571.00**	3007.00
7	9901.00	7388.00	3166.00
8	9145.00	3571.00	3166.00
9	9901.00	4737.00	3088.00
10	**4790.00**	4163.00	3007.00
Average	7945.40	4593.60	3248.60

Table 3 - results for GA using the ***TournamentSelection*** ($populationSize = 100$; $mutationRate = 0.5$; $elitismCount = 10$)

N_run	c = 8	c = 10	c = 12
1	5770.00	4943.00	3007.00
2	9901.00	5546.00	3088.00
3	7208.00	3611.00	3088.00
4	9055.00	**3571.00**	3166.00
5	9538.00	3648.00	3166.00
6	9187.00	7738.00	3210.00
7	4830.00	**3571.00**	5915.00
8	5770.00	5678.00	3088.00
9	6697.00	5005.00	3088.00

10	6162.00	3611.00	5660.00
Average	7411.80	4692.20	3646.60

Table 4 - results for GA using the ***TournamentSelection*** ($populationSize = 100$; $mutationRate = 0.2$; $elitismCount = 5$)

- ***RankBasedSelection*** strategy:

N_run	c = 8	c = 10	c = 12
1	10425.00	9610.00	8307.00
2	7707.00	6241.00	7801.00
3	11381.00	9146.00	3132.00
4	10441.00	9255.00	8450.00
5	11831.00	10447.00	8104.00
6	10369.00	6153.00	7554.00
7	11801.00	8202.00	7320.00
8	10830.00	3954.00	9365.00
9	8051.00	9322.00	3623.00
10	10931.00	9356.00	6123.00
Average	10276.60	8308.60	6797.40

Tabel 5 - results for GA using the ***RankBasedSelection*** ($populationSize = 100$; $mutationRate = 0.2$; $elitismCount = 5$)

N_run	c = 8	c = 10	c = 12
1	11672.00	7307.00	6781.0
2	10547.00	3807.00	3389.0
3	10637.00	6755.00	3829.0
4	10836.00	7601.00	3754.0
5	7677.00	3935.00	4922.0
6	10413.00	9022.00	3520.0

7	10591.00	8107.00	3319.0
8	7140.00	9478.00	3768.0
9	5059.00	6575.00	5660.0
10	11286.00	4297.00	6660.0
Average	9791.80	6684.40	4460.20

Table 6 - results for GA using the ***RankBasedSelection*** ($populationSize = 100$; $mutationRate = 0.5$; $elitismCount = 10$)

- ***FitnessBasedSelection*** strategy:

N_run	c = 8	c = 10	c = 12
1	9145.00	7222.00	3513.00
2	8665.00	4943.00	4400.00
3	10426.00	4163.00	3230.00
4	8839.00	7724.00	3088.00
5	10777.00	5163.00	3513.00
6	8839.00	5163.00	3195.00
7	11223.00	5056.00	3088.00
8	7207.00	10922.00	5038.00
9	6221.00	7724.00	3166.00
10	9207.00	5163.00	3088.00
Average	8964.90	6452.20	3571.90

Tabel 7 - results for GA using the ***FitnessBasedSelection*** ($populationSize = 100$; $mutationRate = 0.5$; $elitismCount = 10$)

N_run	c = 8	c = 10	c = 12

1	9207.00	7802.00	3092.00
2	6221.00	6142.00	3175.00
3	5770.00	3571.00	3195.00
4	10426.00	4181.00	3195.00
5	7257.00	5779.00	3123.00
6	8665.00	8233.00	3195.00
7	9901.00	4737.00	4343.00
8	9330.00	9172.00	3123.00
9	6221.00	4943.00	3069.00
10	11640.00	4163.00	3088.00
Average	8216.20	5872.30	3259.00

Table 8 - results for GA using the ***FitnessBasedSelection*** ($populationSize = 100$; $mutationRate = 0.2$; $elitismCount = 5$)

From tables 3 to 8, we can verify that the best strategy (for both of those settings) was the ***TournamentSelection*** strategy.

Based on those results, we chose to use that strategy as the one to compare and to dictate which are the best settings for the Genetic algorithm.

populationSize	**mutationRate**	**elitismCount**
100	0.2	5
100	0.2	10
100	0.2	50
100	0.5	5
100	0.5	10
100	0.5	50
100	1.0	5
100	1.0	10

100	1.0	50
200	0.2	5
200	0.2	10
200	0.2	50

Table 9 - All the settings combinations tested

From each of those settings combinations (ran 10 times each, for each value of *c*), described in Table 9, we obtained 12 files, with 3 columns each (corresponding to the different values of c (8, 10 and 12). Then we calculated the average values of each column, and the average of the average values, to sort them in an ascendent way (lower average cost to higher average cost). For that we ran a small script (Figure 13) on all of those files, speeding up that process.

From that script, that won't be explained in detail as it's out of the scope of the course, we obtained the following results:

population Size	**mutation Rate**	**elitism Count**	**Average** c=8	**Average** c=10	**Average** c=12	**Average** overall
100	1.0	5	5709.20	3798.50	3083.80	4107.17
100	1.0	10	6736.50	3666.50	3070.40	4491.13
100	1.0	50	6494.50	3912.40	3102.60	4503.17
100	0.5	50	7316.20	4156.80	3146.60	4873.20
100	0.5	10	7955.40	4493.60	3249.60	5232.87
100	0.2	5	7411.80	4692.20	3647.60	5250.53
200	0.2	10	8489.40	4544.80	3677.40	5570.53
100	0.5	5	8629.60	4703.60	3481.80	5605.00
200	0.2	5	8029.10	5735.70	3438.60	5734.47
100	0.2	50	9132.70	5790.50	3571.90	6165.03
100	0.2	10	9219.00	5464.50	4227.10	6303.53
200	0.2	50	9219.20	6604.10	3330.50	6384.60

Table 10 - results obtained from the processing script

```python
import os
import pandas as pd

directory = 'C:/Users/ASUS/Desktop/SO/projeto-so-mei-2/metah'

def process_files(directory):
    file_means = []

    for filename in os.listdir(directory):
        if filename.startswith("GA_results"):
            file_path = os.path.join(directory, filename)
            data = pd.read_csv(file_path, delim_whitespace=True, header=None)
            averages = data.mean(axis=0)
            overall_mean = averages.mean()
            file_means.append((filename, overall_mean, averages))

    # Sort files by their overall mean values
    file_means.sort(key=lambda x: x[1])

    # Sort files by their overall mean values
    for i in range(len(file_means)):
        filename, overall_mean, averages = file_means[i]
        print(f"Filename: {filename}")
        print(f"Smallest mean value: {overall_mean:.2f}")
        for col in range(len(averages)):
            print(f"Average of column {col + 1}: {averages[col]:.2f}")
        print('---------------------------------------')

# Call the function
process_files(directory)
```

Figure 13 - python script to verify the best configurations for the GA

From those results, described on Table 10, we can see that whenever the mutation is sure to happen (=1) the results are lower (better). Well, in fact, what this means in reality is that, when the mutation value is too high (closer to 1), the Genetic Algorithm turns into a random search, which proves to be a better approach, and that is not exactly what we are trying to evaluate here. Therefore, let's ignore the results of the random searches (mutation value 1), and focus our attention on the lower values of the mutation (0.5 and 0.2).

After verifying that the best results were for the mutation values of 0.5 and with higher values of elitism (50) we ran the genetic algorithm again, and obtained the following values:

N_run	c = 8	c = 10	c = 12
1	**4790.00**	**3571.00**	3166.00
2	5770.00	**3571.00**	3088.00
3	5770.00	5160.00	3007.00
4	9901.00	**3571.00**	3007.00
5	9433.00	8043.00	3088.00
6	8754.00	**3571.00**	3166.00

7	9901.00	4943.00	3007.00
8	8665.00	**3571.00**	3088.00
9	5770.00	**3571.00**	3088.00
10	8665.00	**3571.00**	3088.00
Average	7741.90	4394.20	3078.30

Table 11 - results for GA using the ***TournamentSelection*** ($populationSize = 100$; $mutationRate = 0.5$; $elitismCount = 50$)

2.4 Conclusions

To sum up, the configurations with a mutation rate of 1.0 resulted in the lowest average costs. This suggests that when the mutation rate is very high, the GA behaves similarly to a random search. Although this approach yielded the best results in terms of cost, it is not reflective of a true Genetic Algorithm's performance but rather indicative of the effectiveness of random search in this context. When comparing moderate and low mutation rates, a mutation rate of 0.5 provided better balance and performance than a rate of 0.2, as shown in Table 10.

We can also affirm that smaller population sizes of 100 performed better than larger ones of 200, probably due to faster convergence and more frequent updates, despite larger populations offering more genetic diversity. Higher elitism counts generally preserved high-quality solutions better, indicating their importance in maintaining good solutions across generations.

By focusing on configurations with lower mutation rates (0.5 and 0.2) and considering the impacts of population size and elitism count, it is evident that a moderate mutation rate of 0.5, a population size of 100, and higher elitism counts provide a balanced and effective GA configuration. These settings enhance the algorithm's ability to explore and exploit the solution space effectively, resulting in better overall performance.

Therefore, for the given problem instance, the optimal GA parameters include a mutation rate of 0.5, a population size of 100, and higher elitism counts (50 and 10).

3 Approach 2 - Exact method

3.1 Brief Description

The exact method uses Integer Linear Programming (ILP) to solve optimization problems precisely. This approach is applied to the Critical Node Detection (CND) problem, where the goal is to identify critical nodes in a network whose removal minimizes the number of connected node pairs.

ILP models the problem with binary variables representing node inclusion or exclusion as critical. The objective function and constraints ensure that the optimal set of nodes is selected. The branch-and-bound algorithm solves the ILP, iteratively refining the solution by exploring and pruning the solution space.

This method, though computationally intensive, guarantees the best solution, making it ideal for critical network optimization tasks where accuracy is paramount.

3.2 Implementation

```
% exact method based on integer linear programming
nodes = load('Nodes200.txt');
links = load('Links200.txt');
lengths = load('L200.txt');
L = load('L200.txt');

% Create the graph
G = graph(L);

% get the adjacency matrix
adj_m = adjacency(G);

N = 200;    % number of nodes (nodes)
A = 250;    % number of links (L)

% c = 8, 10 and 12 (number of critical nodes)
c = 12;

name = sprintf('CND_lp_%d.lpt', c);
% Open the file for writing
fileID = fopen(name, 'wt');
```

Figure 14 - Start of the exact method script

As described in Figure 14, to start the exact method, we first load the necessary files, create the graph, get the adjacency matrix, initialize some other variables, and open the file, where we will write the ILP model.

```
% Objective function: Minimize the number of connected node pairs
fprintf(fileID, 'min ');
% number of connected node pairs
for i=1:N-1
    for j = i+1 : N
        % u(i,j) = 1 if nodes i and j are connected and 0 otherwise (use the adjacency matrix A)
        fprintf(fileID, ' + u%d_%d', i, j);
    end
end


fprintf(fileID, '\nsubject to\n');
% constraint 1 -> sum{i=1 to n} vi = c
for i=1:N
    fprintf(fileID, ' + v%d', i);
end
fprintf(fileID, ' = %d\n', c);


% constraint 2 -> uij + vi + vj ≥ 1 , i,j ∈ E

for k=1:A
    i = links(k,1);
    j = links(k,2);
    fprintf(fileID, ' + u%d_%d + v%d + v%d >= 1\n', i, j, i, j);
end
```

Figure 15 - Objective function and first two constraints

The objective function, described by Figure 15**,** aims to minimize the number of connected node pairs. This is achieved by summing the binary variables $u_{i,j}$, which indicate whether nodes i and j are connected. The loop iterates through all pairs of nodes and constructs the objective function as a minimization problem.

The first constraint ensures that exactly *c* nodes are selected as critical. The binary variables v_i indicate whether node i is selected as a critical node. The sum of these variables must equal *c*.

The second constraint enforces that if nodes i and j are connected $(u_{i,j} = 1)$, at least one of them must not be a critical node $(v_i \, or \, v_j)$. It ensures that if both v_i and v_j are zero (i.e., neither node is critical), then $u_{i,j}$ must be one (i.e., they must be connected).

```
% constraint 3 -> uij ≥ u{ik} + u{kj} - 1 + vk , i,j ∈ E, k ∈ V(i)
% if i is connected with its neighbour k and k is connected with j, then node i is connected with node j
for i = 1:N
    for j = i+1:N

        n_i = find(adj_m(i,:)); % get the neighbours of node i

        if any(n_i == j)
            continue;
        end

        % Iterate through all nodes to find common neighbors
        for m = 1:length(n_i)
            k = n_i(m);
            fprintf(fileID, '+ u%d_%d - u%d_%d - u%d_%d - v%d >= -1\n', i, j, min(i, k), max(i, k), min(k, j), max(k, j), k);
        end
    end
end
```

Figure 16- Third constraint

The third constraint (Figure 16) ensures that if node i is connected to node k and node k is connected to node j, then node i should be connected to node j, unless k is a critical node. The condition:

$$u_{i,j} \geq u_{i,k} + u_{k,j} - 1 + v_k$$

enforces this relationship. The loops iterate over all node pairs (i, j) and their neighbors to apply this constraint.

```
% Binary constraints
fprintf(fileID, 'binary\n');
for i = 1:N
    fprintf(fileID, 'v%d ', i);
end
fprintf(fileID, '\n');

fprintf(fileID, 'general\n');
for i = 1:N-1
    for j = i+1:N
        fprintf(fileID, 'u%d_%d ', i, j);
    end
end

fprintf(fileID, '\nend');

% Close the file
fclose(fileID);

disp('File generation complete.');
```

Figure 17 - Binary constraints

At last, the binary constraints described by Figure 17, specify that the variables v_i, which indicate whether a node i is critical, are binary. This means that v_i can only take values 0 or 1, representing the exclusion or inclusion of the node as critical.

The general constraints specify the $u_{i,j}$ variables, which indicate whether nodes i and j are connected, are general integer variables. This allows them to take values greater than or equal to 0.

3.3 Results

The ILP model was run for different values of *c* (8, 10 and 12), with a timeout of 300 seconds (5 minutes) and the results obtained were:

c	Optimal Solution	Gap (%)	Running Time (s)
8	4789.99	0.0%	80.954
10	3571.00	0.0%	71.961
12	-	-	-

Table 12 - Results from the ILP model for different values of c

- For c = 8:

```
Model name:  '' - run #1
Objective:   Minimize(R0)

SUBMITTED
Model size:    49469 constraints,   20100 variables,       197822 non-zeros.
Sets:                                  0 GUB,                   0 SOS.

Using DUAL simplex for phase 1 and PRIMAL simplex for phase 2.
The primal and dual simplex pricing strategy set to 'Devex'.


Relaxed solution       4789.99999999 after      24333 iter is B&B base.

Feasible solution      4789.99999999 after      24333 iter,         0 nodes (gap 0.0%)

Optimal solution       4789.99999999 after      24333 iter,         0 nodes (gap 0.0%).

Relative numeric accuracy ||*|| = 1.78627e-011

 MEMO: lp_solve version 5.5.2.11 for 32 bit OS, with 64 bit REAL variables.
      In the total iteration count 24333, 320 (1.3%) were bound flips.
      There were 96 refactorizations, 0 triggered by time and 1 by density.
       ... on average 250.1 major pivots per refactorization.
      The largest [LUSOL v2.2.1.0] fact(B) had 188900 NZ entries, 1.0x largest basis.
      The maximum B&B level was 1, 0.0x MIP order, 1 at the optimal solution.
      The constraint matrix inf-norm is 1, with a dynamic range of 1.
      Time to load data was 1.482 seconds, presolve used 0.011 seconds,
       ... 79.461 seconds in simplex solver, in total 80.954 seconds.
```

Figure 18 - ILP Model for c = 8

- For c = 10:

```
Model name:  '' - run #1
Objective:   Minimize(R0)

SUBMITTED
Model size:    49469 constraints,   20100 variables,       197822 non-zeros.
Sets:                                  0 GUB,                 0 SOS.

Using DUAL simplex for phase 1 and PRIMAL simplex for phase 2.
The primal and dual simplex pricing strategy set to 'Devex'.


Relaxed solution               3571 after      22347 iter is B&B base.

Feasible solution              3571 after      22347 iter,         0 nodes (gap 0.0%)

Optimal solution               3571 after      22347 iter,         0 nodes (gap 0.0%).

Relative numeric accuracy ||*|| = 1.24345e-014

 MEMO: lp_solve version 5.5.2.11 for 32 bit OS, with 64 bit REAL variables.
      In the total iteration count 22347, 213 (1.0%) were bound flips.
      There were 89 refactorizations, 0 triggered by time and 1 by density.
       ... on average 248.7 major pivots per refactorization.
      The largest [LUSOL v2.2.1.0] fact(B) had 181579 NZ entries, 1.0x largest basis.
      The maximum B&B level was 1, 0.0x MIP order, 1 at the optimal solution.
      The constraint matrix inf-norm is 1, with a dynamic range of 1.
      Time to load data was 1.365 seconds, presolve used 0.015 seconds,
       ... 70.581 seconds in simplex solver, in total 71.961 seconds.
```

Figure 19 - ILP Model for c = 10

- For c = 12;

```
Using DUAL simplex for phase 1 and PRIMAL simplex for phase 2.
The primal and dual simplex pricing strategy set to 'Devex'.


Relaxed solution       2877.32633104 after      24234 iter is B&B base.

spx_run: Lost feasibility 10 times - iter      24307 and         2 nodes.
spx_run: Lost feasibility 10 times - iter      27296 and        11 nodes.
spx_run: Lost feasibility 10 times - iter      27943 and        13 nodes.
spx_run: Lost feasibility 10 times - iter      38926 and        44 nodes.
spx_run: Lost feasibility 10 times - iter      50139 and        57 nodes.
spx_run: Lost feasibility 10 times - iter      54340 and        64 nodes.
spx_run: Lost feasibility 10 times - iter      63249 and       119 nodes.
spx_run: Lost feasibility 10 times - iter      64458 and       126 nodes.
spx_run: Lost feasibility 10 times - iter      79161 and       146 nodes.

lp_solve optimization was stopped due to time-out.
lp_solve unsuccessful after 87088 iter and a last best value of 1e+030
lp_solve explored 161 nodes before termination

 MEMO: lp_solve version 5.5.2.11 for 32 bit OS, with 64 bit REAL variables.
      In the total iteration count 87088, 237 (0.3%) were bound flips.
      There were 349 refactorizations, 0 triggered by time and 1 by density.
       ... on average 248.9 major pivots per refactorization.
      The largest [LUSOL v2.2.1.0] fact(B) had 191636 NZ entries, 1.0x largest basis.
      The maximum B&B level was 153, 0.0x MIP order, with 161 nodes explored.
      The constraint matrix inf-norm is 1, with a dynamic range of 1.
      Time to load data was 1.494 seconds, presolve used 0.013 seconds,
       ... 300.004 seconds in simplex solver, in total 301.511 seconds.
```

Figure 20 - ILP Model for c = 12 (timeout = 5 minutes)

From the results described above, we can see that we found the optimal solution for both c values of 8 and 10 (gap = 0.0%).

When c was equal to 12, it couldn't find the optimal solution, and therefore settled for a relaxed solution of 2877.326; A possible reason for this is the computational complexity, as solving larger and larger ILP problems require significant computational resources in terms of memory and processing power. Another possible reason was the time limit, but for this, we could make some more tests, and so we did.

Since we couldn't find the optimal solution with a 5 minute timeout, we decided to try it with a 10 minute timeout.

That didn't work either, and the results were as follows:

```
Model name:  '' - run #1
Objective:   Minimize(R0)

SUBMITTED
Model size:    49469 constraints,   20100 variables,       197822 non-zeros.
Sets:                                  0 GUB,                 0 SOS.

Using DUAL simplex for phase 1 and PRIMAL simplex for phase 2.
The primal and dual simplex pricing strategy set to 'Devex'.


Relaxed solution       2877.32633104 after      24234 iter is B&B base.

spx_run: Lost feasibility 10 times - iter      24307 and         2 nodes.
spx_run: Lost feasibility 10 times - iter      27296 and        11 nodes.
spx_run: Lost feasibility 10 times - iter      27943 and        13 nodes.
spx_run: Lost feasibility 10 times - iter      38926 and        44 nodes.
spx_run: Lost feasibility 10 times - iter      50139 and        57 nodes.
spx_run: Lost feasibility 10 times - iter      54340 and        64 nodes.
spx_run: Lost feasibility 10 times - iter      63249 and       119 nodes.
spx_run: Lost feasibility 10 times - iter      64458 and       126 nodes.
spx_run: Lost feasibility 10 times - iter      79161 and       146 nodes.
Feasible solution              4883 after      92929 iter,       162 nodes (gap 69.6%)
Improved solution      4737.99999999 after     115655 iter,       217 nodes (gap 64.6%)
spx_run: Lost feasibility 10 times - iter     120621 and       222 nodes.
spx_run: Lost feasibility 10 times - iter     130574 and       238 nodes.

lp_solve optimization was stopped due to time-out.

Optimal solution       4737.99999999 after     140237 iter,       256 nodes (gap 64.6%).

Relative numeric accuracy ||*|| = 1.70612e-010

 MEMO: lp_solve version 5.5.2.11 for 32 bit OS, with 64 bit REAL variables.
      In the total iteration count 140237, 263 (0.2%) were bound flips.
      There were 586 refactorizations, 0 triggered by time and 1 by density.
       ... on average 238.9 major pivots per refactorization.
      The largest [LUSOL v2.2.1.0] fact(B) had 191636 NZ entries, 1.0x largest basis.
      The maximum B&B level was 181, 0.0x MIP order, 157 at the optimal solution.
      The constraint matrix inf-norm is 1, with a dynamic range of 1.
      Time to load data was 1.633 seconds, presolve used 0.027 seconds,
       ... 599.990 seconds in simplex solver, in total 601.650 seconds.
```

Figure 21- ILP Model for c = 12 (timeout = 10 minutes)

From Figure 21 we can see that at some iterations it finds some feasible solutions, and even improved solutions, although this is not particularly good, as the gap associated with those solutions is fairly high (over 64%). Which means

that a higher gap indicates that the solver is far from the optimal solution, while a small gap indicates that the solver is close to the optimal solution.

4 Results discussion

After verifying the optimal solutions, for both c = 8 and c = 10, calculated by the ILP and shown in Table 12, we can draw some conclusions about all the methods.

For the GRASP method, and using the best settings ($bestNeighbor1$ and $r = 50$), we verified, by comparing the results in Table 1 and in Table 12, that the GRASP method with those settings obtained the optimal solution 8 out of 10 times for $c = 8$ and 6 out of 10 times for $c = 10$. When comparing the time taken to find the optimal solution, we can see that the GRASP method is way faster than the ILP model, reaching the optimal solution 8 out of 10 times for $c = 8$ in under than 20 seconds, while the ILP model took over 80 seconds to reach the optimal solution. The same can be said for $c = 10$, as the GRASP method found the optimal solution 6 out of 10 times, always in less than 27 seconds, while the ILP model took almost 72 seconds. We can't be sure about the optimal solution for $c = 12$, as the ILP didn't find it, but we can see (Table 1) that the GRASP method obtained values a bit higher than the relaxed solution found by the ILP model, but lower than the feasible and improved solutions that the ILP model found while running, even with a timeout of 10 minutes (Figure ILP4).

For the Genetic Algorithm, and following the best settings, using the ***TournamentSelection*** with $populationSize = 100$, $mutationRate = 0.5$ and $elitismCount = 50$, we can verify, by looking at Table 11 and Table 12, that the GA with those settings, for $c = 8$ found the optimal solution only 1 out of 10 times. While for $c = 10$ it found the optimal solution 8 out of 10 times. When $c = 12$ we can see that the GA found better solutions than the GRASP method, proven by comparing the average values for the GRASP method (3108.60) and the average value for the GA (3078.30).

The efficiency of the GRASP method is further highlighted when considering the GA with a mutation rate of 1.0, effectively turning it into a random search. This approach occasionally outperforms structured methods, indicating that GRASP's randomized nature aligns well with the problem's requirements. Thus, transforming GA into a random search puts it on par with GRASP, underscoring GRASP as the superior method.

In conclusion, the GRASP method stands out as the most reliable and efficient method for solving the Critical Node Detection problem in this context. Its ability to frequently find optimal solutions quickly makes it a superior choice compared to the ILP model and the Genetic Algorithm. The randomized approach of GRASP, as evidenced by the performance of GA under random search conditions, proves its effectiveness for this specific optimization problem.

5 References and resources

de Sousa, Amaro, and Nuno Lau. “Exact Optimization Methods based on Integer Linear Programming.” *Exact Optimization Methods based on Integer Linear Programming*, https://elearning.ua.pt/pluginfile.php/1005714/mod_resource/content/27/ExactMethods_SO23_24VF.pdf. Accessed 2024.

de Sousa, Amaro, and Nuno Lau. “Metaheuristic Optimization Methods.” *Metaheuristic Optimization Methods*, https://elearning.ua.pt/pluginfile.php/993741/mod_resource/content/30/Heuristicas_SO23_24VF.pdf. Accessed 2024.

de Sousa, Amaro, and Nuno Lau. “Simulation Mini-Projects.” *Simulation Mini-Projects*, 2024. *elearning@ua*, https://elearning.ua.pt/pluginfile.php/1824265/mod_resource/content/4/simopt2324_SimProject.pdf.