Optimizing Critical Node Detection: A Comparative Study of Metaheuristic and Exact Methods

Simulação e Otimização
Departamento de Electrónica, Telecomunicações e Informática, Universidade de Aveiro.

May 2024

1 Introdction

This report details the work conducted as part of the Optimization Mini-Project for the Simulação e Otimização course in the academic year 2023/2024, under the guidance of Professors Amaro de Sousa. The primary objective of the project is to tackle the Critical Node Detection (CND) problem within a graph structure, utilizing both metaheuristic and exact optimization methods.

The CND problem is defined on a graph $G = (N, A)$ where the goal is to identify critical nodes whose removal would minimize the number of surviving node pairs capable of communication. This problem is addressed using two distinct approaches: two metaheuristic method and an exact method based on integer linear programming (ILP).

For the metaheuristic approach, the project involves implementing the Greedy Randomized Adaptive Search Procedure (GRASP) and a Genetic Algorithm (GA). These methods are evaluated based on their performance on the given problem instance, which consists of a graph with 200 nodes and 250 links. The metaheuristic method(s) are tested for different values of critical nodes (8, 10, and 12) with a runtime limit of 60 seconds per run.

For the exact approach, the project employs the ILP model to solve the optimization problem, using `lpsolve` with a running time limit of 5 minutes. The solutions obtained from both methods are then compared in terms of objective values and running times, providing a comprehensive analysis of their efficiency and effectiveness [1].

2 GRASP

In this project, the Critical Node Detection (CND) problem was solved in a first phase using the Greedy Randomized Adaptive Search Procedure (GRASP), a well-known metaheuristic method. GRASP combine elements of greedy algorithms with randomized processes to explore a broader solution space and escape local optima.

The GRASP method operates in two primary phases: construction and local search. In the construction phase, a feasible solution is incrementally built by making a series of randomized choices guided by a greedy function.

Once a feasible solution is constructed, the local search phase commences to improve this solution. For this purpose, we integrate the Steepest Ascent Hill Climbing algorithm. Steepest Ascent Hill Climbing is a local search technique that iteratively evaluates the neighboring solutions and moves to the best neighbor that improves the objective function. This process continues until no further improvements can be made, thereby reaching a local optimum.

2.1 Implementation

The development of the GRASP algorithm for solving the Critical Node Detection (CND) problem involved several key steps. Below is a detailed explanation of how the code was structured and developed to achieve the desired functionality. Data Import and Initialization

The first step involves importing the necessary data to create the graph representation of the problem. The data files Nodes200.txt, Links200.txt, and L200.txt contain information about the nodes and links of the graph, which are loaded into MATLAB. The graph is then created using MATLAB's graph function.

```
Nodes = load('Nodes200.txt');
Links = load('Links200.txt');
L = load('L200.txt');
nNodes = size(Nodes, 1);
nLinks = size(Links, 1);
G = graph(L);
time_limit = 60;
number_execution = 10;
```

[1] Work Load: Gallo Ilaria 50%, Irtuso Remo 50%

```
c = [8, 10, 12]; % number of nodes in the solution
```

Parameters for the GRASP algorithm, such as the number of top candidates `r` and an array to store results, are initialized. Additionally, arrays for recording statistics like minimum, maximum, and mean values are set up.

```
r = 2;
results_grasp = cell(length(c), 1);
min_grasp = zeros(size(c));
max_grasp = zeros(size(c));
mean_grasp = zeros(size(c));
min_nodes = cell(size(c));
max_nodes = cell(size(c));
```

The main program loop iterates over each value of `c` and performs multiple executions of the GRASP algorithm. For each execution, the nodes selected and the corresponding solution value are stored. The minimum and maximum solutions found are tracked, and relevant statistics are computed at the end of each set of executions.

```
% Main program to evaluate the GRASP algorithm
%For each value of c
for idx = 1:length(c)
    %Creating the results matrix
    results_grasp{idx} = zeros(number_execution, c(idx) + 1); % +1 to store the solution cost
    local_min = inf;
    local_max = -inf;

    % GRASP execution
    for j = 1:number_execution
        %Calling Grasp Function
        [nodes, solution, ~] = grasp(G, c(idx), r, time_limit);
        %Appending the reuslts
        results_grasp{idx}(j, 1:end-1) = nodes;
        results_grasp{idx}(j, end) = solution;

        % Check for new min or max
        if solution < local_min
            local_min = solution;
            min_nodes{idx} = nodes;
        end
        if solution > local_max
            local_max = solution;
            max_nodes{idx} = nodes;
        end
    end

    % Compute min, mean, max
    min_grasp(idx) = local_min;
    mean_grasp(idx) = mean(results_grasp{idx}(:, end));
    max_grasp(idx) = local_max;
end
```

To obtain the firs solution in the GRASP algorithm the `greedyRnadomized` method was implemented. This function builds an initial solution by selecting nodes in a greedy yet randomized manner.

```
function s = greedyRandomized(G, n, r)
    E = 1:numnodes(G);
    s = [];
    for i = 1:n
        R = [];
        for j = E
            R = [R; j, ConnectedNP(G, [s, j])];
        end
        R = sortrows(R, 2);
        e = R(randi(r), 1);
        s = [s, e];
        E = setdiff(E, e);
    end
end
```

A list of candidates `E` is maintained, then, for each node to be added to the solution, a list `R` of candidate nodes and their associated objective values is created. The list is sorted, and a node is randomly selected from the top `r` candidates. This process ensures a good balance between exploration and exploitation.

Finally, the `grasp` function in was designed to find an optimal solution for the CND problem by combining a greedy randomized construction phase with a local search phase using Steepest Ascent Hill Climbing. Below is a step-by-step explanation of the function. First of all there is the initialization of the timer and the iteration counter.

```
%--------------- GRASP ALGORITHM

function [solution_nodes, solution, num_iter] = grasp(G, n, r, time_limit)
    % Initialize timer and counter
    t = tic;
    num_iter = 0;
    %...
```

An initial solution is constructed using the greedy randomized function shown previously.

```
    %...
    % Construct initial solution
    solution_nodes = greedyRandomized(G, n, r);
    % Evaluate the initial solution
    solution = ConnectedNP(G, solution_nodes);
    %...
```

The main loop runs until the specified time limit is reached.

```
    %...
    while toc(t) < time_limit
        % Current set of nodes
        current_nodes = solution_nodes;
        % Current objective value
        current_solution = solution;
        % Flag to check if improvement is possible
        improved = true;
    %...
```

Initialize variables for the local search phase using Steepest Ascent Hill Climbing.

```
    %...
    while improved
        % Total number of nodes
        N = numnodes(G);
        % Nodes not in the current solution
        aux = setdiff(1:N, current_nodes);
        % Best neighbor solution
    initialization
        best_neigh_solution =
    current_solution;
        % Best neighbor nodes initialization
        best_curr_neigh = current_nodes;
    %...
```

Explore the neighborhood by evaluating possible node swaps.

```
    %...
    for a = current_nodes
        for b = aux
            % Swap node a with node b
            neighbors = [setdiff(
    current_nodes, a), b];
            % Evaluate new solution
            neigh_solution = ConnectedNP(G,
    neighbors);
            if neigh_solution <
    best_neigh_solution
                % Update best neighbor
    solution
                best_neigh_solution =
    neigh_solution;
                % Update best neighbor nodes
                best_curr_neigh = neighbors;
            end
        end
    end
    %...
```

Update the current solution if a better neighbor is found.

```
    %...
        if best_neigh_solution <
    current_solution
            % Update current solution value
            current_solution =
    best_neigh_solution;
            % Update current solution nodes
            current_nodes = best_curr_neigh;
        else
            % No improvement found
            improved = false;
        end
    end
    %...
```

Update the global best solution if the current solution is better.

```
    %...
        if current_solution < solution
            % Update global best solution
    value
            solution = current_solution;
            % Update global best solution
    nodes
            solution_nodes = current_nodes;
        end
        % Increment iteration
        num_iter = num_iter + 1;
    end
end
```

2.2 Results

One of the objectives of this project was to evaluate the impact of varying the parameter `r` in the GRASP algorithm on the quality of the solutions. The parameter `r` represents the number of top candidates considered in the greedy randomized selection phase. By experimenting with different values of `r`, the goal was to determine whether adjusting this parameter can lead to significant improvements or variations in the solutions obtained. The results for different values of `r` are summarized from Table 1 to Table 4 at the end of the report.

From the results presented in the tables, it is evident that varying the parameter `r` affects the quality of the solutions obtained by the GRASP algorithm. Here are some key observations for the different values of `c`:

1. `c = 8`: The minimum value is around 9766 most of the time, but when `r` is equal to 4, the minimum values get a drastic decreasment reaching 4790. The maximum value fluctuates, with the highest at 12520 when `r` is 4. The mean value and mean iterations also show some variation, indicating that the parameter `r` impacts the convergence and solution quality. In particular, to reach the best value when `r` is equal to 4 the number of iteration increases indicating lower convergence of the algorithm.

2. `c = 10`: The results show significant differences, especially in the minimum values. For example, with `r` = 3, the minimum value drops to 4163, suggesting better performance for this setting. Mean iterations decrease slightly with higher values of `r`, reflecting a possible faster convergence.

3. `c = 12`: There is a noticeable improvement in the minimum value for `r` equal 5 with a minimum value of 3007. The mean value also shows improvements, while mean iterations do not vary significantly, suggesting that `r` equal 5 might be an optimal setting for this value of `c`.

In conclusion, the parameter `r` in the GRASP algorithm plays a crucial role in determining the quality and consistency of the solutions. Careful tuning of this parameter can lead to significant improvements in performance, making it a vital aspect of the optimization process. Future work can further explore the interplay between `r` and other parameters to optimize the algorithm's performance across different problem instances.

3 Genetic Algorithm

Genetic Algorithms (GAs) are a class of evolutionary algorithms inspired by the process of natural selection.

They are used to solve optimization and search problems by evolving solutions over successive generations. GAs mimic biological evolution processes such as selection, crossover, and mutation to explore the solution space and identify optimal or near-optimal solutions.

In this project, we implemented a Genetic Algorithm to solve the Critical Node Detection (CND) problem. The main components of our GA implementation are as follows:

1. **Initialization**: population of potential solutions is generated randomly.
2. **Crossover**: two individuals are selected to generate a new one derived from the combination of their genes (in this case the nodes of the network).
3. **Mutation**: offspring are subjected to mutation to introduce genetic diversity.
4. **Selection**: individuals are selected from the population based on their fitness.

The Genetic Algorithm's parameters, such as population size, number of elite individuals, mutation rate, are the one that were meant to be tuned for this project.

3.1 Implementation

The Genetic Algorithm (GA) implementation for solving the Critical Node Detection (CND) problem follows a structured approach to simulate the process of natural selection. This section provides a detailed explanation of the implementation steps and the functions used.

As for the GRASP algorithm all the necessary parameters and data structures for the results were implemented.

```
% GA parameter initialization
population_size = 100;
q = 0.1; % mutation probability
num_elite = 10;
min_ga = zeros(size(c));
max_ga = zeros(size(c));
mean_ga = zeros(size(c));
min_nodes_ga = cell(size(c));
max_nodes_ga = cell(size(c));
iterations_ga = cell(length(c), 1);
```

After that the main function was implemented. In the first part of the function some parameters were initialized and the initial random population was generated.

```
%--------------- GENETIC ALGORITHM
function [solution_nodes, solution,
    num_generation] = ga(G, n, nNodes,
    population_size, q, num_elite, time_limit
    )
    num_generation = 0;
    %Genereting the first population
    % an additional cell in the matrix is
    %created to store the fitness
    current_population = zeros([
population_size, n+1]);
    for i = 1:population_size
        %we select random nodes of the graph
        %for each individual
        current_individual = randperm(nNodes,
     n);
        current_population(i, 1:end-1) =
current_individual;
    end

    %a new generation is created so
    %the counter is incremented
    num_generation = num_generation + 1;

    %initial fitness value computation
    for i = 1:population_size
        current_fitness = ConnectedNP(G,
current_population(i, 1:end-1));
        current_population(i, end) =
current_fitness;
    end

    %selecting the first best individual
    current_population = sortrows(
current_population, (n+1));
    solution = current_population(1, n+1);
    solution_nodes = current_population(1, 1:
end-1);
```

Then the main cycle begins in which a new generation is created. Each new individual is generated with the tournament selection crossover and then, with a probability of `q` it could mutate. The method for crossover and mutation are explained later in the paper.

```
%settinc the timer
t = tic;

    while toc(t) < time_limit

        %generating the new population
        new_population = zeros(
    population_size, n+1);

        % Generate new individuals with
        %crossover and possible mutation
        for i = 1:population_size
            %using the crossover
            new_individual =
    crossover_tournament(current_population,
    n);

            %veryfing mutation probability
            if rand < q
                new_individual = mutation(
    new_individual, nNodes, n);
            end

            %evaluating the fitness of the
    new indiviual
            new_fitness = ConnectedNP(G,
    new_individual(1:end-1));
            new_individual(end) = new_fitness
    ;
            %adding the new individual to the
     new population
            new_population(i, :) =
    new_individual;
```

```
        end
```

After that the elite individuals are selected. Elite individuals are the first m who have the best fitness value. After their selection they are added to the previous generation. The new generation is then created by selecting the first n individuals with the best fitness among all. Inn this case n indicates the size of the population.

```
    % Sort the new population by fitness
    new_population = sortrows(new_population,
     (n+1));

    % Select the top m individuals as elite
    elite_individuals = new_population(1:
    num_elite, :);

    % Combine current population and elite
    %individuals
    combined_population = [current_population
    ; elite_individuals];
    %sorting rows of the combined population
    combined_population = sortrows(
    combined_population, (n+1));

    % Keep the best population_size from
    combined population
    current_population = combined_population
    (1:population_size, :);

    %updating the counter
    num_generation = num_generation+1;

    % Track the best solution
    if current_population(1, end) < solution
        solution = current_population(1, end)
    ;
        solution_nodes = current_population
    (1, 1:end-1);
    end
```

Is also important to understand how the crossover and mutation work. Beginning with the mutation function. To mutate an individual, a random gene (node) is selected. The algorithm identifies all nodes that are not currently part of the individual, referred to as `possible_nodes`. A random node is then chosen from these `possible_nodes` and replaces the selected gene in the individual.

```
function mutated_individual = mutation(
    individual, nNodes, n)

    mutated_individual = individual;

    % selecting a random gene
    mutation_point = randi(n);

    % selecting a node that is not
    %currently in the individual
    possible_nodes = setdiff(1:nNodes,
    individual(1:n));
    %selecting the new node computing a
    random
    %index of possible node
    new_node = possible_nodes(randi(length(
    possible_nodes)));

    % applying mutation
    mutated_individual(mutation_point) =
    new_node;
    %fintess = 0 because it need to
    %be evaluated in the main algorithm
    mutated_individual(n+1) = 0;

end
```

Finally the crossover. To generate a new individual two parents need to be selected. The procedure is simple: two individuals are selected randomly from the current population, then, the individual with the best fitness is going to be one of the parent. To select the second one the same steps are used.

```
function new_individual =
    crossover_tournament(population, n)
    % Tournament selection to pick parent 1
    %random individual selection
    idx1 = randi(size(population - 1, 1));
    idx2 = randi(size(population - 1, 1));
    %choosing the parent based on fitness
    if population(idx1, n+1) > population(
    idx2, n+1)
        parent_1 = population(idx1, 1:end-1);
    else
        parent_1 = population(idx2, 1:end-1);
    end

    % Tournament selection to pick parent 2
    idx1 = randi(size(population - 1, 1));
    idx2 = randi(size(population - 1, 1));
    if population(idx1, n+1) > population(
    idx2, n+1)
        parent_2 = population(idx1, 1:end-1);
    else
        parent_2 = population(idx2, 1:end-1);
    end
```

After the selection the parents are combined to obtain the new indivdual.

```
    %combining the parend
    aux = union(parent_1, parent_2);
    %creating a permutation of the
    %gens of the two parents
    aux2 = randperm(length(aux), n);
    %selecting the genes
    new_individual = aux(aux2);
    new_individual(n+1) = 0; % Temporary
    value, compute actual fitness later
end
```

3.2 Results

The Genetic Algorithm (GA) was applied to the Critical Node Detection (CND) problem with various parameter settings. The the results obtained for different population sizes, mutation probabilities, and elite sizes are reported from Table 5 to Table 8 at the end of the document. As for GRASP, the tables include the maximum, minimum, and mean values of the objective function, the mean number of generations (instead of iterations), and the nodes corresponding to the best solutions.

The results obtained from the GA with various parameter settings provide valuable insights into the performance of the algorithm under different configurations.

The initial configuration of parameters was set as follows: population size = 100, elite individuals = 10, mutation probability = 0.1. The results obtained in this case, shown in Table 5, are not excellent when compared to the best results obtained by GRASP. Subsequently, only the mutation probability parameter was modified, increasing it from 0.1 to 0.3. The mutation probability is a very delicate parameter to configure: if it is too high, this probability can lead to excessive randomness, disrupting the optimization process and preventing convergence. Conversely, if it is too low, the algorithm may become stuck in local optima, failing to explore the solution space adequately. By applying this single modification, a significant improvement in the optimization results can be observed, which approach (and in the case of `c=10` even surpass) the best results of GRASP.

Next, the population size was increased while the mutation value was reset to its initial value. In this case, the results deteriorated significantly. This was expected because computing for twice the number of individuals requires more time, resulting in the evaluation of fewer populations and thus slower convergence of the algorithm.

Finally, multiple parameters were modified simultaneously, but even in this case, the results were not better than those achieved solely with a mutation probability of 0.3. This underscores the impact of this parameter: indeed, by modifying only this, a significant improvement in results was achieved.

It is also possible to conclude that Increasing the number of elite individuals helps preserve the best solutions across generations, contributing to a more consistent performance. Overall, tuning these parameters is crucial for optimizing the Genetic Algorithm's performance. The GA require more effort to obtain the right set of parameter, future work can further explore these aspect trying different combinations to enhance the algorithm's performance.

4 Exact Methods

The project also involved the application of exact methods to solve the Critical Node Detection (CND) problem, specifically using Integer Linear Programming (ILP). Exact optimization methods, such as ILP, are designed to provide guaranteed optimal solutions by precisely defining the problem through mathematical models and solving them with dedicated solvers. These methods are particularly useful when the goal is to ensure the best possible solution within the constraints of the problem. The CND problem can be described with the following formalization:

$$\text{minimize: } \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} u_{ij}$$

subject to:

$$\sum_{i=1}^{n} v_i = c$$

$$u_{ij} + v_i + v_j \geq 1, (i,j) \in E$$

$$u_{ij} \geq u_{ik} + u_{kj} - 1 + v_k, (i,j) \notin E, k \in V(i)$$

$$v_i \in [0,1]$$

$$u_{ij} \in \mathbf{R}_0^+$$

As explained earlier, the objective function aim to minimize the number of connected node pairs u_{ij}. This summation represents the total number of node pairs that remain connected after the removal of the selected critical nodes. It is also important to understand the costraints of the problem to be able to solve it:

1. The first constraint ensures that exactly c nodes are selected as critical nodes. Here, v_i is a binary variable indicating whether node i is a critical node (1 if it is critical, 0 otherwise).

2. The second constraint ensures that for each pair of connected nodes (i, j) in the set of edges E, either one of the nodes is a critical node or the pair is counted as disconnected ($u_{ij} \geq 1$). This means that if neither v_i nor v_j is 1 (i.e., neither node is critical), then u_{ij} must be 1, indicating that nodes i and j remain connected.

3. The third constraint handles the cases where nodes i and j are not directly connected. It ensures that if i and j are indirectly connected through a common neighbor k, then u_{ij} should reflect this connection, unless k is a critical node. $V(i)$ represents the set of neighbors of node i. Note that u_{ij} represents the variable u_{ij} if $i < j$, otherwise, u_{ji}.

The last two costraints state that the variables v_i are binary and the u_{ij} is a non-negative real number. In our project, we utilized the `lp_solve` solver, a free and widely-used software for solving linear, integer, and mixed-integer linear programming problems. This solver was chosen for its accessibility and capability to handle the problem's complexity efficiently.

4.1 Implementation

To apply ILP using `lp_solve`, we needed to convert our problem into a format compatible with the solver. This involved writing a MATLAB script that generates `.lp` files, which describe the CND problem in a structured format. The script, for each value of `c`, creates a new file:

```
for value = c
    file_name = sprintf('exact_method_c_%d.lp', value);
    fid = fopen(file_name,'wt');
    %...
```

Than the objective function is written to minimize the number of connected node pairs.

```
    %...
    %objective function
    fprintf(fid,'Min: ');
    for i = 1:nNodes-1
        for j = i + 1:nNodes
            if i ~= nNodes-1
                fprintf(fid,'+ u%d_%d ', i, j);
            else
                fprintf(fid,'+ u%d_%d;', i, j);
            end
        end
    end
    fprintf(fid,'\n');
    %...
```

Than the costraints were implemented, beginning with the one that ensure that exactly `c` nodes are selected as critical nodes.

```
    %...
    for i = 1:nNodes
        fprintf(fid,'+ v%d ', i);
    end
    fprintf(fid,'= %d;\n', value);
    %...
```

The second constraint ensure that for each connected node pair, either one node is critical or the pair is considered disconnected

```
    %...
    % 2nd constraint
    for i = 1:size(L, 1)
        written = false;  % Initialize a flag to track if anything has been written for this 'i'
        for j = 1:size(L, 2)
            if L(i, j) ~= 0 && (i < j)  % Only consider each edge once (i < j)
                if ~written
                    written = true;  % Set flag to true as we are writing something
                    fprintf(fid, '+ u%d_%d + v%d + v%d >= 1;\n', i, j, i, j);
                else
                    fprintf(fid, '+ u%d_%d + v%d + v%d >= 1;\n', i, j, i, j);
                end
            end
        end
    end
    %...
```

The third costraint ensure that for nodes not directly connected, their disconnection considers critical nodes and common neighbors

```
% 3rd constraint
for i = 1:nNodes
    for j = 1:nNodes
        if i ~= j && i < j && L(i, j) == 0  % Consider only non-connected pairs (i < j)
            ni = neighbors(G, i);  % Get neighbors of node i
            if ~isempty(ni)
                for k = ni'  % Iterate over common neighbors
                    % Determine the correct form for u variables
                    if i < k
                        ui_k = sprintf('u%d_%d', i, k);
                    else
                        ui_k = sprintf('u%d_%d', k, i);
                    end
                    if j < k
                        uj_k = sprintf('u%d_%d', j, k);
                    else
                        uj_k = sprintf('u%d_%d', k, j);
                    end
                    if i < j
                        fprintf(fid, '+ u%d_%d - %s - %s - v%d >= - 1;\n', i, j, ui_k, uj_k, k);
                    else
                        fprintf(fid, '+ u%d_%d - %s - %s - v%d >= - 1;\n', j, i, ui_k, uj_k, k);
                    end
                end
            end
        end
    end
end
```

Finally the binary variables costraint was implemented. The last costraint, $u_{ij} \in \mathbf{R}_0^+$, is implicit in `lp_solve`.

```
    % Binary variables constraints: Ensure that all v variables are binary
    fprintf(fid, 'Bin ');
    for i = 1:nNodes
        if i ~= nNodes
            fprintf(fid, 'v%d,', i);
        else
            fprintf(fid, 'v%d;', i);
        end
    end
```

If executed, the script generates three different files that have this structure:

```
Min: + u1_2 + u1_3 + u1_4 ...;
+ v1 + v2 + v3 + v4 + .... +v200 = 8;
+ u1_12 + v1 + v12 >= 1;
+ u2_10 + v2 + v10 >= 1;
...
+ u198_199 + v198 + v199 >= 1;
+ u198_200 + v198 + v200 >= 1;
+ u1_2 - u1_12 - u2_12 - v12 >= - 1;
+ u1_3 - u1_12 - u3_12 - v12 >= - 1;
...
+ u199_200 - u186_199 - u186_200 - v186 >= - 1;
```

```
+ u199_200 - u198_199 - u198_200 - v198 >= -
    1;
Bin v1,v2,v3,v4,v5,v6,v7,v8,...;
```

4.2 Results

To obtain the results using the exact methods, a runtime limit of 5 minutes was set, and then each created file was executed. The results are shown from Table 9 to table 11 in the appendix. The exact methods utilize Integer Linear Programming (ILP) to provide guaranteed optimal solutions. The `lp_solve` solver was employed to solve these ILP formulations. Follows some observation about the obtained results:

- For `c=8` and `c=10`, the exact methods were able to find optimal solutions with a gap of 0.0%, indicating high accuracy and precision in the solutions provided.
- For `c=12`, the solver timed out, indicating the increased complexity and difficulty in solving larger instances within a reasonable time frame. With only five minutes, in fact, it does not reach any integer solution. That is because the file was executed on a virtual machine with lower computation capacity. Was then decided to run it a second time but with a limit of ten minutes. In this case a solution was found with a gap of 16.3%. This indicates that to find the optimal solution the solver need even more time. One final test was conducted with a time limit of one hour. After 44 minutes the algorithm found as optimal solution the value 3140 with a solution gap of 9.1%.

The computation time for the exact methods was substantial, with solutions taking over 200 seconds for smaller instances and over 300 seconds for larger instances. The number of iterations also reflects the complexity, with the exact methods requiring a significant number of iterations to converge to an optimal or near-optimal solution.

The solutions provided by the exact methods are of high quality, as expected from ILP approaches. The exact method guarantees the best possible solution within the constraints and model formulated. The solution nodes identified are consistent with the objectives of minimizing the number of connected node pairs when the critical nodes are removed. In conclusion, the exact methods provide a benchmark for optimal solutions.

5 Conclusions

In this project, the Critical Node Detection (CND) problem was tackled using a combination of metaheuristic and exact optimization methods. Our primary focus was on implementing and evaluating the Greedy Randomized Adaptive Search Procedure (GRASP), Genetic Algorithms (GA), and Integer Linear Programming (ILP) approaches to understand their performance, strengths, and limitations.

The GRASP algorithm proved effective in finding high-quality solutions for the CND problem. By integrating the Steepest Ascent Hill Climbing method in the local search phase, we were able to refine the solutions further. We found that the parameter r, representing the number of top candidates in the greedy randomized selection phase, significantly influenced the algorithm's performance. Adjusting r led to noticeable improvements, highlighting the importance of tuning this parameter to optimize GRASP's performance. Overall, GRASP demonstrated its robustness and flexibility in addressing the CND problem. For `c = 8` and $r = 4$ the minimum value of the objective function is really exactly the one that the exact method produced.

The Genetic Algorithm also showed its capability to explore the solution space and evolve towards optimal solutions through processes similar to natural selection, crossover, and mutation. We tested various parameter settings, including different population sizes, mutation probabilities, and numbers of elite individuals. Our findings emphasized the necessity of parameter tuning: larger population sizes generally improved solution quality but required more generations to converge, higher mutation probabilities introduced greater diversity, leading to better average solutions but increased variability, and increasing the number of elite individuals helped preserve the best solutions across generations, enhancing consistency. Particularly, a mutation probability of 0.3 provided a significant improvement in solution quality, demonstrating the GA's effectiveness when properly tuned. Is important to notice also that for `c = 10` the GA reaches the optimal solution gave by the exact method with the base configuration and the mutation rate of 0.3.

Using exact methods with ILP, we achieved guaranteed optimal solutions, especially for smaller problem instances. The ILP approach, implemented with the `lp_solve` solver, successfully found optimal solutions with a 0.0% gap for (`c = 8`) and (`c = 10`). However, for (`c = 12`), the solver timed out within the initial 5-minute limit, highlighting the increased complexity of larger problem instances. Extending the runtime to 10 minutes and then to one hour showed some improvements, but the solver still faced challenges in achieving optimal solutions within practical time frames. These findings underscore the utility of exact methods as benchmarks for optimality, despite their computational intensity.

In comparing the three approaches, we gained several key insights. Both GRASP and GA demonstrated flexibility and effectiveness in solving the CND prob-

lem, with the added advantage of being scalable to larger problem instances. On the other hand, ILP provided high-quality solutions for smaller instances but struggled with the computational complexity of larger problems. For metaheuristic approaches, careful tuning of parameters such as the number of candidates r, population size, mutation probability, and elite size is crucial for optimizing performance. Each method has its strengths—metaheuristics offer scalability and flexibility, while exact methods guarantee optimality but at a higher computational cost.

Looking ahead, future research can focus on hybrid approaches that combine the strengths of metaheuristic and exact methods to enhance performance. Exploring advanced techniques for parameter tuning and leveraging parallel computing resources could further improve the efficiency and effectiveness of these optimization algorithms. The insights gained from this project provide a solid foundation for developing robust solutions to the CND problem and similar optimization challenges.

Table 1: GRASP Results for r = 2

C Value	Max Value	Min Value	Mean Value	Mean Iterations	Solution Nodes
8	11164	9766	10005.70	55.50	[18,53,78,83, 111,129,184,97]
10	8857	8737	8769.40	46.80	[15,18,31,53,78, 83,111,129,184, 97]
12	4681	3108	3265.30	38.10	[18,53,76,78, 101,103,104,117, 146,150,154,93]

Table 2: GRASP Results for r = 3

C Value	Max Value	Min Value	Mean Value	Mean Iterations	Solution Nodes
8	11164	9766	10005.70	54.60	[18,53,78,83, 111,129,184,97]
10	8857	4163	8300	43.10	[18,53,76,78,101, 103,104,117,154, 93]
12	3108	3108	3108	39.20	[18,53,76,78, 93,101,103,104, 117,146,150,154, 103]

Table 3: GRASP Results for r = 4

C Value	Max Value	Min Value	Mean Value	Mean Iterations	Solution Nodes
8	12520	4790	10063.10	62.20	[18,53,76,78, 104,117,154,93]
10	8821	5163	8295.10	50.00	[18,53,76,78,104, 132,154,166,177, 131]
12	4910	3108	3445.50	47.40	[18,53,76,78, 101,103,104,117 146,150,154,93]

Table 4: GRASP Results for r = 5

C Value	Max Value	Min Value	Mean Value	Mean Iterations	Solution Nodes
8	11164	9766	10105.60	57.20	[18,53,97,111, 83,129,184,78]
10	8857	8737	8774.20	49.60	[15,18,31,53,78, 83,111,129,184, 97]
12	3108	3007	3097.90	43.40	[18,33,52,53, 76,78,93,104 146,150,154,117]

Table 5: GA Results for [populatipn size = 100, mutation prop = 0.1, elite = 10]

C Value	Max Value	Min Value	Mean Value	Mean Generations	Solution Nodes
8	10216	5770	7866.5	1188.2	[146,18,53,78, 150,154,104,76]
10	8483	3571	6760.50	1302.3	[117,93,104,150, 18, 78,53,76, 146,154]
12	6785	3088	3978.50	1638.30	[104,90,150,154, 18,53,166,93, 123,78,54,76]

Table 6: GA Results for [populatipn size = 100, mutation prop = 0.3, elite = 10]

C Value	Max Value	Min Value	Mean Value	Mean Generations	Solution Nodes
8	10087	4830	7619.9	1108.5	[154,76,93,53, 104,78,94,18]
10	7956	3571	5084.80	1410	[104,76,150,53, 93, 154,18,78, 146,117]
12	7077	3014	3518.40	1446.70	[76,33,18,93, 117,154,53,65, 150,104,146,78]

Table 7: GA Results for [populatipn size = 200, mutation prop = 0.1, elite = 10]

C Value	Max Value	Min Value	Mean Value	Mean Generations	Solution Nodes
8	10205	7261	9397.4	433.5	[83,103,72,154 54,150,129,107]
10	5377	3630	4253.10	544.1	[81,146,104,150, 93, 53,154,18,66, 78,94]
12	6147	3102	3909.8	635.1	[104,53,18,66, 54,150,78,166, 90,93,154,123]

Table 8: GA Results for [populatipn size = 150, mutation prop = 0.2, elite = 15]

C Value	Max Value	Min Value	Mean Value	Mean Generations	Solution Nodes
8	9991	4830	8721.1	710.9	[18,104,94,93 76,53,154,78]
10	8172	4214	6444	829.2	[93,185,66,14, 93, 54,104,150,78, 154,30]
12	4850	3088	3569.3	938.8	[53,78,93,150 154,166,18,90, 104,76,54,123]

Table 9: Exact Method Results for $c = 8$

Type of Solution	Value	Iterations	GAP (%)	Exec. Time (s)	Solution Nodes
Relaxed	4789.99	24333	0.0	230.238	[18,53, 76,78, 93,104, 117,154]
Feasible	4789.99	24333	0.0		
Optimal	4789.99	24333	0.0		

Table 10: Exact Method Results for $c = 10$

Type of Solution	Value	Iterations	GAP (%)	Exec. Time (s)	Solution Nodes
Relaxed	3571	22347	0.0	210.991	[18,53, 76,78, 93,104, 117,146 150,154]
Feasible	3571	22347	0.0		
Optimal	3571	22347	0.0		

Table 11: Exact Method Results for c = 12 (10 minutes run time limit)

Type of Solution	Value	Iterations	GAP (%)	Exec. Time (s)	Solution Nodes
Relaxed	2877.32	24234		606.088	[15,18,30, 53,66,78, 93,104,117, 146,150,154]
Feasible	3346	35997	16.3		
Optimal	3346	46547	16.3		