Departamento de Eletrónica, Telecomunicações e Informática

Simulação e Otimização

Relatório Final do Mini-Projeto de Otimização

André Oliveira, 107637
Alexandre Cotorobai, 107849

Mestrado em Engenharia Informática

Professor: Prof. Amaro de Sousa

31 de maio de 2025

Índice

Capítulo 1

Introdução

O presente relatório foi desenvolvido no âmbito da unidade curricular de *Simulação e Otimização*, inserida no plano de estudos do Mestrado em Engenharia Informática da Universidade de Aveiro. O trabalho enquadra-se no segundo mini-projeto da unidade curricular, cujo objetivo consiste na aplicação de métodos de otimização, tanto metaheurísticos como exatos, à resolução de um problema concreto relacionado com redes definidas por software (*Software Defined Networks* - SDN).

Neste contexto, é considerada uma rede SDN cuja topologia é representada por um grafo $G = (N, A)$, sendo N o conjunto de nós (switches) e A o conjunto de arcos (ligações), cada um com um comprimento l_{ij} associado. Pretende-se selecionar um subconjunto de n nós onde serão instalados controladores SDN, garantindo que a distância mais curta entre quaisquer dois destes nós não ultrapassa um limite máximo C_{max}. A função objetivo do problema visa minimizar o comprimento médio do caminho mais curto entre cada nó da rede e o seu controlador mais próximo.

A instância do problema fornecida inclui um grafo com $|N| = 200$ nós e $|A| = 250$ ligações, sendo fixados os parâmetros $n = 12$ e $C_{max} = 1000$. Os dados da rede são disponibilizados nos ficheiros `Nodes200.txt`, `Links200.txt` e `L200.txt`.

Este mini-projeto tem como principais objetivos:

- Implementar e aplicar os métodos GRASP (*Greedy Randomized Adaptive Search Procedure*), Algoritmo Genético (GA) e Programação Linear Inteira (ILP) à resolução do problema;
- Avaliar e comparar os resultados obtidos por cada abordagem, considerando tanto a qualidade das soluções como os tempos de execução;
- Justificar as decisões metodológicas e os parâmetros adotados em cada implementação;
- Consolidar os conhecimentos teóricos lecionados na unidade curricular, através de uma análise crítica e comparativa dos métodos de otimização utilizados.

A estrutura do relatório é organizada por método de resolução, permitindo uma análise aprofundada e segmentada das três abordagens estudadas. Em cada secção, são descritas a motivação teórica, a metodologia de implementação em MATLAB, os resultados obtidos e, sempre que relevante, excertos comentados de código. A última secção apresenta uma análise comparativa dos métodos, discutindo as suas vantagens e limitações à luz dos dados recolhidos.

Capítulo 2

GRASP

2.1 Introdução

O método GRASP (*Greedy Randomized Adaptive Search Procedure*) é uma metaheurística multi-start composta por duas fases principais: a construção de soluções viáveis através de uma heurística gulosa aleatorizada, e a subsequente melhoria dessas soluções por meio de uma busca local do tipo *Steepest Ascent Hill Climbing*. Esta abordagem é particularmente eficaz em problemas combinatórios complexos, como a seleção de nós controladores em redes *Software Defined Network* (SDN), onde se procura um compromisso entre diversidade de soluções e qualidade local.

Durante cada iteração, o algoritmo constrói uma solução inicial com base numa heurística gulosa modificada, onde, a cada passo, é selecionado aleatoriamente um dos r elementos mais promissores segundo uma métrica de custo. Esta lista de candidatos restrita (*Restricted Candidate List* - RCL) permite controlar o grau de aleatoriedade do processo de construção. A solução gerada é então submetida a uma busca local, onde se explora a vizinhança gerada por substituição de um único nó, procurando sempre a solução com melhor valor da função objetivo.

A estrutura geral do GRASP é ilustrada no seguinte pseudocódigo:

Algorithm 1 Esquema geral do GRASP

```
1: s ← GreedyRandomized()
2: s_best ← AdaptiveSearch(s)
3: while condição de paragem não satisfeita do
4:     s ← GreedyRandomized()
5:     s ← AdaptiveSearch(s)
6:     if f(s) < f(s_best) then
7:         s_best ← s
8:     end if
9: end while
```

Neste trabalho, a fase de construção utilizou um método *Greedy Randomized* parametrizado pelo valor r, que define o tamanho da RCL. A busca local explorou a vizinhança definida por soluções que diferem em apenas um nó da solução corrente, adotando a estratégia *Steepest Ascent Hill Climbing*. A execução do algoritmo é repetida até atingir um tempo máximo de 30 segundos por execução, de acordo com as especificações do projeto.

2.2 Implementação

O algoritmo foi implementado em `MATLAB` com uma arquitetura modular. Esta abordagem facilitou o controlo das fases de construção e melhoria de soluções, bem como a gestão de tempos de execução. A função principal é `GRASP_SNS`, que coordena o processo iterativo da metaheurística, recorrendo às funções auxiliares `GreedyRandomizedConstruction`, `LocalSearch_SA_HC` e `PerfSNS`.

Função Principal: `GRASP_SNS`

A função `GRASP_SNS` constitui o núcleo da implementação do algoritmo GRASP, sendo responsável pela execução controlada no tempo das duas fases fundamentais do método: construção de soluções iniciais viáveis e respetiva melhoria por busca local. A sua assinatura define claramente os parâmetros de entrada e saída:

```
function [bestScore, bestNodes, totalIterations, bestFoundTime] = ...
    GRASP_SNS(G, time, n, r, Cmax, seed)
```

Esta função recebe o grafo `G` representando a topologia da rede, o tempo máximo de execução em segundos (`time`), o número de nós servidores a selecionar (`n`), o tamanho da lista restrita de candidatos (`r`), o limite máximo de distância entre servidores (`Cmax`), e opcionalmente uma semente para o gerador aleatório (`seed`). Os valores retornados incluem a melhor pontuação encontrada (`bestScore`), a solução correspondente (`bestNodes`), o número total de iterações da fase de busca local (`totalIterations`) e o instante temporal em que a melhor solução foi obtida (`bestFoundTime`).

A primeira parte da função trata da inicialização das variáveis de controlo. Caso tenha sido especificada uma semente válida, esta é utilizada para inicializar o gerador de números aleatórios, assegurando reprodutibilidade dos resultados. As variáveis de estado são inicializadas com os valores padrão apropriados: uma pontuação inicial infinita, um vetor de solução vazio, e contadores a zero. É também iniciado o cronómetro principal com `tic`, e é calculada, de forma antecipada, a matriz de distâncias `D` entre todos os pares de nós da rede, através da função `distances(G)`. Este pré-processamento evita redundâncias e acelera as avaliações durante as iterações.

```
if nargin >= 6 && ~isempty(seed)
    rng(seed);
end
bestScore = Inf;
bestNodes = [];
totalIterations = 0;
bestFoundTime = 0;
globalStartTime = tic;
D = distances(G);
```

Segue-se o ciclo principal do algoritmo, responsável por explorar o espaço de soluções durante o tempo disponível. A cada iteração, verifica-se o tempo decorrido com `toc(globalStartTime)` e interrompe-se o processo assim que este ultrapassar o limite especificado. Em alternativa, se o tempo restante for demasiado reduzido para uma nova iteração significativa, o ciclo termina preventivamente.

```
while true
    elapsed = toc(globalStartTime);
    if elapsed >= time
        break;
```

```
    end
    ...
end
```

Dentro deste ciclo, realizam-se as duas fases centrais do GRASP: construção e melhoria. A função `GreedyRandomizedConstruction` é chamada para gerar uma solução inicial com base numa heurística gulosa e aleatorização guiada por uma lista restrita de candidatos. Esta construção é interrompida e a iteração descartada caso se verifique inviabilidade estrutural da solução ou violação da restrição de distância entre servidores. As soluções viáveis passam então por uma fase de otimização local, recorrendo à função `LocalSearch_SA_HC`, que aplica uma estratégia de *Steepest Ascent Hill Climbing* para refinar a solução, tentando substituir elementos da solução por candidatos externos de forma a melhorar o valor da função objetivo, sempre respeitando a restrição $C_{\max}$.

Após concluída a fase de busca local, a solução resultante é comparada com a melhor solução global registada até ao momento. Se o valor da função objetivo for inferior, então esta nova solução é adotada como a nova melhor solução, sendo atualizados os vetores de estado `bestScore`, `bestNodes` e `bestFoundTime`, este último registando o instante exato (em segundos) desde o início da execução em que a solução foi encontrada:

```
if currentScore < bestScore
    bestScore = currentScore;
    bestNodes = currentNodes;
    bestFoundTime = toc(globalStartTime);
end
```

O ciclo repete-se até esgotar o tempo, promovendo uma exploração diversificada do espaço de soluções com reforço local da qualidade. Esta função define, assim, o esqueleto do algoritmo GRASP, integrando eficientemente os componentes de construção, melhoria e controlo temporal, fundamentais para a obtenção de boas soluções dentro do orçamento de tempo imposto.

Fase 1 – Construção Gulosa Aleatorizada: `GreedyRandomizedConstruction`

A construção de soluções iniciais viáveis é realizada pela função `GreedyRandomizedConstruction`, cujo objetivo é selecionar iterativamente n nós da rede que satisfaçam a restrição de conectividade máxima $C_{\max}$, enquanto promovem uma boa cobertura da rede no que respeita ao encaminhamento eficiente dos pacotes. A abordagem utilizada combina uma heurística gulosa com aleatorização controlada através de uma Lista Restrita de Candidatos (RCL), promovendo a diversidade de soluções geradas ao longo das iterações do GRASP.

O processo inicia-se com o cálculo da centralidade de cada nó, definida como o inverso da soma das distâncias a todos os outros nós, a partir da matriz `D` previamente calculada. A centralidade é usada como critério heurístico para o primeiro nó da solução, assumindo-se que nós mais centrais oferecem melhor cobertura da rede:

```
centrality = 1 ./ sum(D, 2)';
```

Após a ordenação dos nós por ordem crescente de centralidade, é construída a RCL com os r nós mais centrais. Um deles é selecionado aleatoriamente e adicionado à solução como primeiro nó controlador:

```
[~, sortedIndices] = sort(centrality);
rcl = sortedIndices(1:rclSize);
```

```
selectedIdx = randi(rclSize);
nodes(1) = rcl(selectedIdx);
```

A construção prossegue de forma iterativa até a solução conter n nós. A cada passo k, avaliam-se todos os nós ainda não selecionados. Para cada candidato, verifica-se se a sua inclusão com os nós já presentes respeita a restrição $C_{\max}$. Esta verificação é feita determinando a maior distância entre o nó candidato e os restantes já selecionados:

```
for a = 1:k-1
    serverA = nodes(a);
    dist = D(serverA, candidate);
    maxDistBetweenServers = max(maxDistBetweenServers, dist);
end
if maxDistBetweenServers > Cmax
    continue;
end
```

Se o candidato for viável, calcula-se o seu benefício, definido como o valor negativo da soma das distâncias mínimas entre cada nó da rede e o conjunto de controladores atuais mais o candidato. Este valor serve como aproximação inversa da função objetivo, permitindo ordenar os candidatos pela sua capacidade de melhorar o desempenho da solução:

```
totalDistance = 0;
for node = 1:numNodes
    if ~ismember(node, tempSolution)
        dist = min(D(node, tempSolution));
        totalDistance = totalDistance + dist;
    end
end
benefit(i) = -totalDistance;
```

Uma vez calculado o benefício de todos os candidatos viáveis, estes são ordenados por ordem decrescente de benefício (ou seja, crescente de custo total), e forma-se uma nova RCL com os r melhores. Um nó é selecionado aleatoriamente desta lista e adicionado à solução:

```
[~, sortedIndices] = sort(validBenefits, 'descend');
rcl = remaining(validIndices(sortedIndices(1:rclSize)));
selectedIdx = randi(rclSize);
selectedNode = rcl(selectedIdx);
nodes(k) = selectedNode;
```

Este processo continua até que sejam escolhidos n nós válidos ou até que não existam candidatos viáveis, caso em que a função retorna uma solução vazia.

A construção gulosa aleatorizada desempenha um papel crucial no algoritmo GRASP: permite gerar soluções iniciais de qualidade moderada, com boa diversidade, e respeitando restrições de viabilidade fundamentais para o problema. A combinação da heurística com aleatoriedade controlada garante que o espaço de soluções é explorado de forma eficaz ao longo de múltiplas iterações.

Fase 2 – Busca Local: `LocalSearch_SA_HC`

Após a construção de uma solução inicial viável, esta é refinada através da aplicação de uma busca local do tipo *Steepest Ascent Hill Climbing*, implementada na função `LocalSearch_SA_HC`. O objetivo desta fase é explorar a vizinhança da solução corrente para encontrar melhorias incrementais que conduzam a soluções de menor custo, sem violar a restrição de distância entre servidores. A execução é iterativa, e prossegue enquanto forem encontradas melhorias e houver tempo para continuar.

Inicialmente, é determinado o conjunto de nós não selecionados, isto é, todos os nós da rede que não pertencem à solução atual. Em seguida, para cada nó da solução corrente, testa-se a substituição por cada um dos nós fora da solução. Este processo corresponde a um varrimento exaustivo de todas as vizinhanças obtidas por trocas 1-a-1 (*swap*):

```
notSelected = setdiff(1:numNodes, bestNodes);
for i = 1:length(bestNodes)
    for j = 1:length(notSelected)
        neighborNodes = bestNodes;
        neighborNodes(i) = notSelected(j);
```

Para cada solução vizinha gerada por uma troca, é invocada a função `PerfSNS` que calcula dois indicadores: (i) o valor médio dos caminhos mais curtos entre cada nó da rede e o seu controlador mais próximo (`avgSP`), e (ii) a maior distância entre quaisquer dois servidores da solução candidata (`maxSP`). A segunda métrica é utilizada para verificar a admissibilidade da solução, descartando de imediato todas as que violam a restrição imposta por $C_{\max}$:

```
[neighborScore, neighborMaxSP] = PerfSNS(G, neighborNodes);
if neighborMaxSP > Cmax
    continue;
end
```

Se a solução for viável, compara-se o valor da função objetivo com o da melhor solução vizinha conhecida até ao momento. Apenas se esta for estritamente melhor, a melhoria é registada e o par de índices da troca é armazenado:

```
if neighborScore < bestNeighborScore
    bestNeighborScore = neighborScore;
    bestSwap = [i, notSelected(j)];
    improved = true;
end
```

Após varrer todas as combinações possíveis de trocas, aplica-se, se existir, a melhor substituição encontrada nessa iteração. O processo é repetido enquanto forem encontradas melhorias e o tempo de execução o permitir. A cada iteração, o número de movimentos locais realizados é acumulado, permitindo medir o esforço computacional gasto na fase de intensificação.

Esta abordagem corresponde a uma estratégia de busca local voraz, que em cada passo procura a maior melhoria possível (no espírito de *Steepest Ascent*), o que a torna particularmente eficaz na eliminação de más decisões tomadas durante a construção inicial. Embora limitada a uma vizinhança simples baseada em trocas, esta técnica mostrou-se suficiente para alcançar melhorias significativas nas soluções iniciais, contribuindo para a eficácia global do método GRASP.

Otimizações

A versão `GRASP_SNS_Optimized` introduz várias melhorias com o objetivo de reduzir o tempo de execução e aumentar a eficiência na construção e avaliação de soluções. Estas otimizações incidem sobretudo em três componentes do algoritmo: a construção gulosa, a avaliação de soluções, e a busca local. Abaixo detalham-se as principais alterações introduzidas, com justificações técnicas e excertos representativos de código.

1. Pré-filtragem baseada em $C_{\max}$

Antes de iniciar o ciclo GRASP, é feita uma análise à matriz de distâncias `D` com o intuito de identificar, para cada nó, quais os vizinhos com os quais pode coabitar numa solução válida. Esta verificação permite, mais tarde, restringir os candidatos possíveis de forma eficiente, evitando avaliações desnecessárias de soluções inviáveis.

```
validPairs = D <= Cmax;
validNeighbors = cell(numNodes, 1);
for i = 1:numNodes
    validNeighbors{i} = find(validPairs(i, :));
end
```

Esta estrutura é usada tanto na construção da solução como na busca local, reduzindo significativamente o número de combinações testadas e o custo computacional por iteração.

2. Ordenação dos nós por centralidade e acesso direto

Na seleção inicial dos candidatos, em vez de calcular a centralidade a cada execução, esta é pré-computada uma vez e armazenada numa lista ordenada. Tal como na versão base, a centralidade é definida como o inverso da soma das distâncias a todos os outros nós. O vetor `centralityOrder` é então utilizado para gerar as listas RCL com acesso imediato aos nós mais relevantes.

```
centrality = 1 ./ sum(D, 2)';
[~, centralityOrder] = sort(centrality, 'descend');
```

3. Cache de avaliações de soluções

A avaliação da função objetivo para cada solução candidata, efetuada através da função `PerfSNS`, pode ser computacionalmente dispendiosa. Para evitar repetições, foi implementado um mecanismo de *memoization* utilizando um `containers.Map`, onde as soluções já avaliadas são armazenadas com uma chave textual baseada na ordenação dos nós selecionados.

```
scoreCache = containers.Map('KeyType', 'char', 'ValueType', 'double');

nodeKey = mat2str(sort(currentNodes));
if isKey(scoreCache, nodeKey)
    currentScore = scoreCache(nodeKey);
else
    [currentScore, maxSP] = PerfSNS(G, currentNodes);
    if maxSP > Cmax
        continue;
    end
    scoreCache(nodeKey) = currentScore;
end
```

Este mecanismo reduz drasticamente o número de chamadas à função de avaliação, especialmente durante a busca local, onde muitas soluções vizinhas podem coincidir com anteriores.

4. Avaliação incremental na construção

Durante a construção da solução, ao invés de recalcular a função objetivo completa para cada candidato, esta é avaliada de forma incremental apenas para os nós ainda não selecionados. Isto permite ignorar partes da rede que não são afetadas pela adição de novos nós, acelerando o cálculo do benefício de cada candidato.

```
for node = 1:numNodes
    if ~usedNodes(node) && node ~= candidate
        minDist = min(D(node, tempSolution));
        totalDistance = totalDistance + minDist;
    end
end
```

Esta abordagem minimiza o número de chamadas à função `min()` e evita cópias desnecessárias de grandes vetores.

5. Busca local com early stopping e pré-filtragem

Na fase de melhoria, a vizinhança é construída apenas com movimentos que respeitam $C_{\max}$ à partida, eliminando desde logo movimentos inviáveis. Além disso, a ordem dos movimentos é randomizada para promover diversidade, e o processo de melhoria pode ser interrompido assim que a primeira melhoria válida é encontrada, reduzindo o número de iterações internas e agilizando a convergência local.

```
if neighborScore < bestNeighborScore
    bestNeighborScore = neighborScore;
    bestSwap = [i, candidate];
    improved = true;
    break; % early stopping
end
```

A estrutura `validNeighbors` é novamente utilizada aqui para restringir os candidatos válidos a trocas, aumentando a eficiência sem comprometer a qualidade das soluções.

No seu conjunto, como mostrado no gráfico da Figura 2.1, estas otimizações introduzem ganhos significativos em tempo de execução, permitindo realizar mais iterações dentro do mesmo limite temporal. Em particular, a combinação de pré-filtragem, cache e avaliação incremental mostrou-se eficaz para melhorar tanto o desempenho como a escalabilidade do algoritmo.

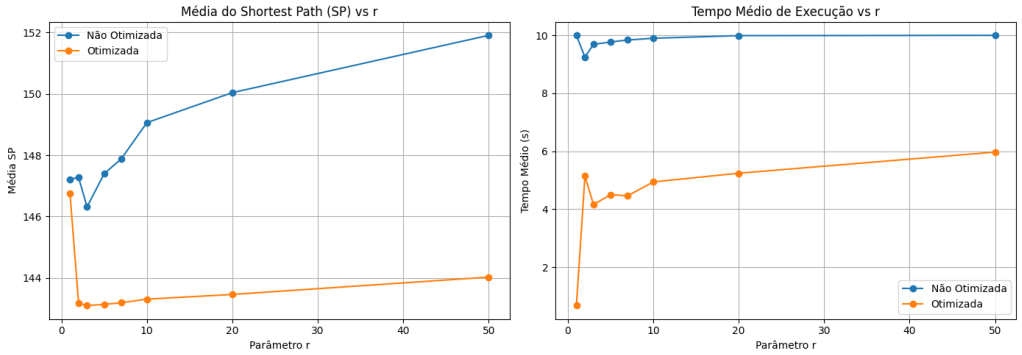

Figura 2.1: Comparação entre as versões otimizada e não otimizada do GRASP quanto à média do comprimento dos caminhos mais curtos (SP) e ao tempo médio de execução, para diferentes valores do parâmetro r.

2.3 Escolha de Parâmetros

A escolha do parâmetro r, que define o tamanho da Lista Restrita de Candidatos (RCL) utilizada na fase de construção do GRASP, é essencial para equilibrar a exploração e a qualidade das soluções iniciais. Valores baixos de r conduzem a uma construção altamente determinística, com pouca diversidade entre soluções geradas. Por outro lado, valores excessivamente elevados resultam em soluções iniciais potencialmente fracas, exigindo maior esforço da busca local para obtenção de melhorias.

Para fundamentar a escolha de um valor adequado de r, foram realizados dois conjuntos de experiências, tanto com a versão original como com a versão otimizada do GRASP, utilizando o script `tune.m`. Em ambas as variantes, variou-se r entre os valores $\{1, 2, 3, 5, 7, 10, 20, 50\}$, sendo testadas duas abordagens:

Cenário A – 30 execuções de 10 segundos Esta configuração favorece a robustez estatística, permitindo avaliar a consistência dos resultados para cada valor de r, com um número elevado de execuções mas tempo limitado por execução. O objetivo foi identificar padrões globais e variações introduzidas pela aleatoriedade.

Cenário B – 10 execuções de 30 segundos Neste caso, o enfoque é posto na qualidade final das soluções, permitindo mais tempo por execução e, portanto, maior profundidade de busca local. Esta configuração oferece uma avaliação mais realista do comportamento do GRASP quando o tempo não é tão restritivo.

As Figuras 2.2 e 2.3 apresentam os resultados das médias de SP e dos tempos médios para ambas as versões do algoritmo. Os gráficos confirmam que os valores de $r = 2$ e $r = 3$ apresentam um bom compromisso entre custo computacional e desempenho da solução, com o valor $r = 3$ a destacar-se por manter uma performance robusta em todos os cenários.

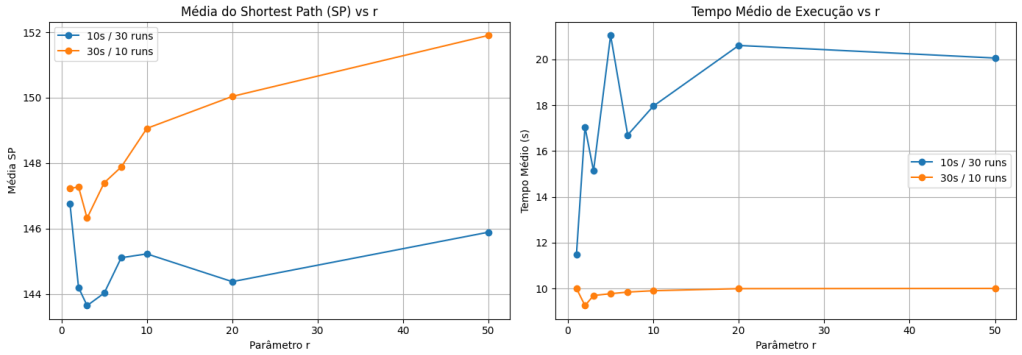

Figura 2.2: Comparação da média do SP e do tempo de execução nos dois cenários (versão não otimizada)

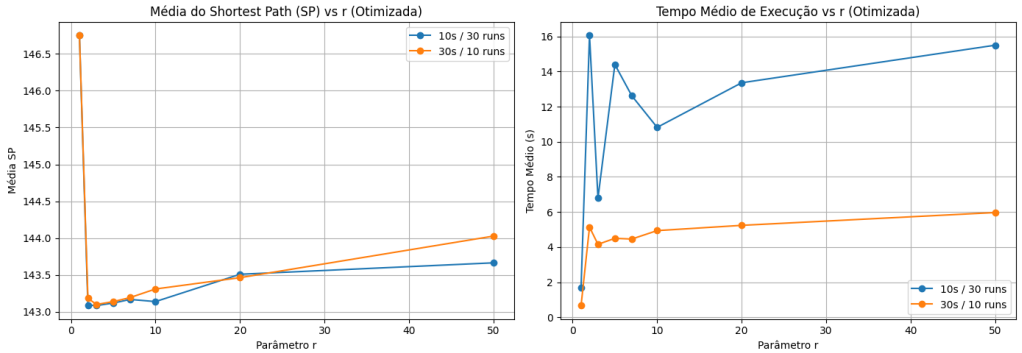

Figura 2.3: Comparação da média do SP e do tempo de execução nos dois cenários (versão otimizada)

A Tabela 2.1 apresenta os valores mínimos de média do SP obtidos para cada configuração testada, permitindo uma comparação direta do melhor desempenho conseguido por valor de r. A análise evidencia que, na versão otimizada, os resultados mínimos atingem um patamar estável a partir de $r = 2$, com valores de SP praticamente invariáveis, mesmo em execuções mais curtas.

Tabela 2.1: Valores mínimos de média do SP por valor de r nas versões otimizada e não otimizada

r	**Não otimizado (10x30s)**	**Não otimizado (30x10s)**	**Otimizado (10x30s)**	**Otimizado (30x10s)**
1	146.750	146.750	146.750	146.750
2	143.085	143.085	143.085	143.085
3	143.085	143.085	143.085	143.085
5	143.160	143.280	143.085	143.085
7	143.160	143.335	143.085	143.085
10	143.085	143.405	143.085	143.085
20	143.085	144.580	143.085	143.085
50	143.085	144.635	143.085	143.085

Com base nesta análise, foi adotado o valor $r = 3$ como valor de referência para as restantes experiências. Esta escolha permite manter um equilíbrio eficiente entre diversidade na construção, qualidade da solução inicial e tempo de execução total, tanto na versão otimizada como na original do GRASP.

2.4 Resultados

Para avaliar o desempenho final do algoritmo GRASP otimizado, foram realizadas 10 execuções independentes com tempo máximo de 30 segundos cada, utilizando $r = 3$, valor previamente identificado como o mais eficaz.

A Tabela 2.2 apresenta os valores da função objetivo (SP médio), a distância máxima entre pares de servidores (SP máximo) e o tempo de execução até à melhor solução, por execução.

Execução	SP Médio	SP Máximo	Tempo até Melhor Solução (s)
1	143.0850	998.0000	9.57
2	143.0850	998.0000	12.12
3	143.0850	998.0000	18.26
4	143.0850	998.0000	4.04
5	143.0850	998.0000	10.03
6	143.0850	998.0000	1.01
7	143.0850	998.0000	18.14
8	143.0850	998.0000	2.45
9	143.0850	998.0000	11.50
10	143.0850	998.0000	16.41

Tabela 2.2: Resultados obtidos pelo GRASP em 10 execuções com limite de 30 segundos.

Verifica-se que todas as execuções retornaram exatamente o mesmo valor de SP médio, igual a 143.0850, o que confirma a estabilidade do algoritmo mesmo perante diferentes *seeds* aleatórias. O tempo médio até à obtenção da melhor solução foi de 10.35 segundos, reforçando a eficiência do método na convergência para soluções de elevada qualidade dentro do orçamento temporal definido.

A melhor solução encontrada, obtida logo na primeira execução, consistiu na seleção dos seguintes nós como servidores:

$$20,\ 30,\ 65,\ 68,\ 90,\ 91,\ 107,\ 108,\ 131,\ 146,\ 163,\ 173$$

A Figura 2.4 apresenta a topologia da rede com a localização desta solução, destacando visualmente os nós selecionados a vermelho:

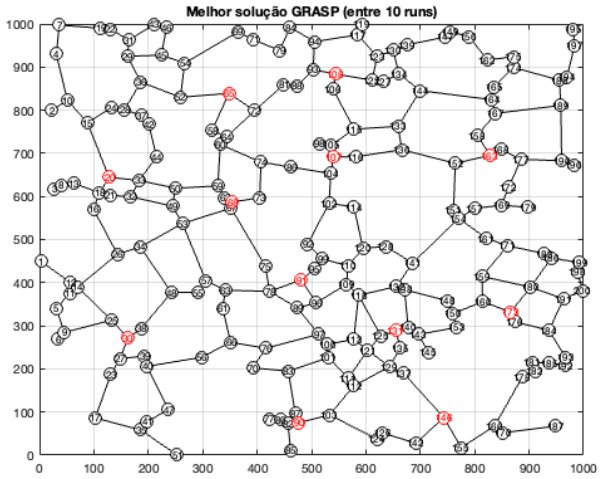

Figura 2.4: Melhor solução GRASP (entre 10 execuções), com os 12 nós selecionados destacados a vermelho.

Capítulo 3

GA

3.1 Introdução

O Algoritmo Genético (GA) é uma metaheurística populacional inspirada no processo evolutivo de seleção natural, conforme proposto por Charles Darwin. Esta abordagem baseia-se na evolução de uma população de soluções candidatas ao longo de várias gerações, promovendo a combinação e mutação dos seus componentes para explorar eficientemente o espaço de soluções. Cada solução candidata, designada por indivíduo, é avaliada através de uma função de aptidão que mede a sua qualidade em relação ao objetivo do problema.

O princípio fundamental do GA reside na reprodução seletiva de indivíduos mais aptos. A cada geração, indivíduos da população atual são selecionados para reprodução com base no seu desempenho. Dois operadores genéticos principais são aplicados: *crossover*, que combina genes de dois progenitores para formar um novo indivíduo (descendente), e *mutação*, que introduz pequenas alterações aleatórias com o objetivo de preservar a diversidade genética e evitar convergência prematura. Adicionalmente, pode ser aplicada uma política de elitismo que garante que os melhores indivíduos da geração atual sejam preservados na geração seguinte.

Para relembrar, no contexto deste projeto, o objetivo do GA é identificar subconjuntos de $n = 12$ nós num grafo com $|N| = 200$ que minimizem o comprimento médio do caminho mais curto entre cada nó da rede e o seu nó controlador mais próximo, garantindo simultaneamente que a distância entre quaisquer dois controladores não excede um valor máximo $C_{max} = 1000$.

A modelação deste problema de otimização combina elementos estruturais (topologia da rede), funcionais (distâncias mínimas) e restritivos (limites de conectividade entre servidores). A utilização de uma representação genotípica adequada, bem como a definição criteriosa dos operadores genéticos, são cruciais para a eficácia do algoritmo. A capacidade do GA de manter e refinar uma população diversificada ao longo do tempo torna-o uma abordagem robusta para encontrar boas soluções em tempo limitado, mesmo para problemas altamente não lineares e com restrições complexas como o presente.

A secção seguinte detalha a metodologia de implementação adotada, abordando a representação das soluções, os mecanismos de seleção e reprodução, os critérios de paragem e as decisões de parametrização do algoritmo.

3.2 Implementação

A implementação do Algoritmo Genético (GA) foi organizada em módulos distintos, promovendo clareza e eficiência. O código foi desenvolvido em *MATLAB* e segue uma arquitetura evolutiva clássica: geração inicial aleatória, reprodução com operadores genéticos e seleção com elitismo. A função principal

`GA_SNS` coordena todo o processo, assegurando que a execução respeita o tempo limite imposto.

Função Principal: `GA_SNS`

A função `GA_SNS` é o ponto de entrada da execução do algoritmo. Recebe como parâmetros a topologia da rede (grafo `G`), o tempo máximo de execução, o número de nós a selecionar por solução, o tamanho da população, a probabilidade de mutação, o número máximo de soluções elitistas a preservar, o limite de conectividade máxima ($C_{\max}$) e uma *seed* opcional.

```
function [bestScore, bestNodes, generations, bestFoundTime] = ...
    GA_SNS(G, time, n, populationSize, mutationProb, elitistParam, Cmax, seed)
```

Se a *seed* for fornecida, o gerador de números aleatórios é inicializado para garantir reprodutibilidade. A seguir, a população inicial é gerada e avaliada. O ciclo principal de execução mantém-se ativo até esgotar o tempo disponível:

```
 [population, fitness, feasible, bestScore, ...
 bestNodes, bestFoundTime] = initializePopulation(...);

while true
    elapsed = toc(startTime);
    if elapsed >= time
        break;
    end

    [offspringPopulation, offspringFitness, offspringFeasible, ...
     bestScore, bestNodes, bestFoundTime] = generateOffspring(...);

    [population, fitness, feasible] = applySurvivalSelection(...);

    generations = generations + 1;
end
```

Este ciclo gera descendentes, aplica seleção com elitismo e incrementa o número de gerações até o tempo limite ser atingido.

Inicialização da População: `initializePopulation`

Nesta fase, cada indivíduo da população inicial é criado aleatoriamente com `randperm`, que seleciona um subconjunto de `n` nós distintos. A viabilidade da solução é verificada através da função `PerfSNS`, que calcula o comprimento médio dos caminhos (*fitness*) e a distância máxima entre quaisquer dois servidores (restrição de viabilidade).

```
population{i} = randperm(numNodes, n);
[avgSP, maxSP] = PerfSNS(G, population{i});
```

Se a solução respeitar o limite $C_{\max}$, é considerada viável:

```
if maxSP <= Cmax
    feasible(i) = true;
    fitness(i) = avgSP;
```

```
4
5       if avgSP < bestScore
6           bestScore = avgSP;
7           bestNodes = population{i};
8           bestFoundTime = toc(startTime);
9       end
10  else
11      fitness(i) = avgSP + (maxSP - Cmax);
12  end
```

Soluções inviáveis são penalizadas proporcionalmente à violação da restrição, o que permite que participem no processo evolutivo, embora com menor probabilidade de seleção.

Seleção de Progenitores: `selectParents`

A seleção dos progenitores é feita através de torneio binário. Dois candidatos são escolhidos aleatoriamente. Se existirem candidatos viáveis entre os dois, seleciona-se o com melhor *fitness*. Caso contrário, seleciona-se o com menor penalização.

```
1  candidates = randi(length(fitness), 1, 2);
2  feasibleCandidates = candidates(feasible(candidates));
3
4  if ~isempty(feasibleCandidates)
5      [~, bestIdx] = min(fitness(feasibleCandidates));
6      idx = feasibleCandidates(bestIdx);
7  else
8      [~, bestIdx] = min(fitness(candidates));
9      idx = candidates(bestIdx);
10 end
```

Este mecanismo garante robustez, mesmo em populações onde soluções viáveis são inicialmente escassas.

Cruzamento: `crossover`

O operador de cruzamento combina subconjuntos dos dois progenitores, substituindo elementos de `parent1` por elementos únicos de `parent2`. Isto preserva a estrutura base de um dos pais, enquanto injeta diversidade genética do outro.

```
1  selected = parent1;
2  uniqueToParent2 = setdiff(parent2, parent1);
3
4  if ~isempty(uniqueToParent2)
5      numToSwap = randi([1, floor(n/2)]);
6      numToSwap = min(numToSwap, length(uniqueToParent2));
7      posToReplace = randperm(n, numToSwap);
8      elemsToInsert = uniqueToParent2(randperm(length(uniqueToParent2), numToSwap));
9      selected(posToReplace) = elemsToInsert;
10 end
11 child = selected;
```

Este cruzamento é conservador: não permite duplicação de nós, e o número de trocas é limitado, garantindo diversidade sem perda de viabilidade estrutural.

Mutação: `mutate`

A mutação substitui um único nó da solução por outro que não esteja presente. Isto permite escapar de ótimos locais e explorar novas regiões do espaço de busca.

```
notSelected = setdiff(1:numNodes, solution);
pos = randi(length(solution));
newNode = notSelected(randi(length(notSelected)));
mutated(pos) = newNode;
```

A operação é simples, mas eficaz, e é aplicada com uma probabilidade definida pelo parâmetro `mutationProb`.

Avaliação e Atualização da Melhor Solução: `evaluateAndUpdateBest`

Após gerar e mutar uma solução, esta é avaliada quanto ao seu custo médio e viabilidade. Se for viável e melhorar a melhor solução conhecida, o algoritmo atualiza os registos:

```
[avgSP, maxSP] = PerfSNS(G, solution);

if maxSP <= Cmax
    solutionFeasible = true;
    solutionFitness = avgSP;

    if avgSP < bestScore
        bestScore = avgSP;
        bestNodes = solution;
        bestFoundTime = toc(startTime);
    end
else
    solutionFeasible = false;
    solutionFitness = avgSP + (maxSP - Cmax);
end
```

A penalização para soluções inviáveis é utilizada aqui da mesma forma que na população inicial.

Elitismo e Seleção de Sobreviventes: `applySurvivalSelection`

Após gerar todos os descendentes, os melhores indivíduos da geração anterior são preservados, até um limite definido por `elitistParam`:

```
[parentFitness, parentIndices] = sort(fitness);
filteredParentCount = min(elitistParam, populationSize);
filteredParents = cell(filteredParentCount, 1);

for i = 1:filteredParentCount
    filteredParents{i} = population{parentIndices(i)};
end
```

Progenitores e descendentes são combinados e ordenados com prioridade para viabilidade e qualidade:

```
selectionPool = [filteredParents; offspringPopulation];
```

```
[selectionPool, selectionPoolFitness, selectionPoolFeasible] = ...
    sortSolutionsByFeasibilityAndFitness(selectionPool, ...);
```

As melhores soluções viáveis são então selecionadas para compor a próxima geração:

```
newPopulation = selectionPool(1:populationSize);
newFitness = selectionPoolFitness(1:populationSize);
newFeasible = selectionPoolFeasible(1:populationSize);

for i = 1:populationSize
    newPopulation{i} = selectionPool{i};
    newFitness(i) = selectionPoolFitness(i);
    newFeasible(i) = selectionPoolFeasible(i);
end
```

Este mecanismo garante que o algoritmo converge para soluções cada vez melhores, sem sacrificar diversidade, e respeitando sistematicamente a restrição $C_{\max}$.

3.3 Escolha dos Parâmetros

A escolha dos parâmetros seguiu a mesma abordagem adotada na secção correspondente do método GRASP. Utilizando execuções repetidas e análise estatística dos resultados, procurou-se equilibrar a qualidade das soluções com o tempo de execução, respeitando o limite de 30 segundos por execução.

Testaram-se combinações dos três principais parâmetros do GA: tamanho da população (*populationSize*), probabilidade de mutação (*mutationProb*) e grau de elitismo (*elitistParam*). Cada combinação foi avaliada em dois cenários experimentais:

- Execuções curtas de 10 segundos ao longo de 30 repetições — para explorar o espaço de parâmetros de forma estatisticamente robusta.
- Execuções longas de 30 segundos ao longo de 10 repetições — para validar o comportamento das melhores configurações em cenários realistas.

O objetivo era minimizar o valor médio de shortest paths (*SP*) das soluções viáveis, assegurando ao mesmo tempo uma taxa razoável de viabilidade, ou seja, soluções que respeitassem o valor imposto de C_{max}, e tempos de execução compatíveis com os limites do projeto.

As tabelas seguintes apresentam os resultados estatísticos agregados para cada configuração testada, com base nas execuções registadas.

População	Mutação	Elitismo	SP Mín	SP Médio	SP Máx	Tempo Médio (s)
20	0.05	1	143.0850	147.1668	152.1000	6.8063
20	0.05	5	143.4050	146.7750	152.3550	5.8627
20	0.05	10	143.4050	147.7612	159.9350	6.5567
20	0.10	1	143.3700	146.9420	154.0950	7.0977
20	0.10	5	143.0850	146.2022	149.8950	4.3813
20	0.10	10	143.0850	146.9807	152.6400	4.6553
20	0.20	1	143.0850	146.5863	151.7550	7.6647
20	0.20	5	143.0850	146.5272	156.0600	2.4680
20	0.20	10	143.4050	146.9143	150.2750	2.7943
50	0.05	1	144.2850	148.9598	154.8600	8.3707
50	0.05	5	143.4050	146.8112	153.8950	6.1477
50	0.05	10	143.0850	146.6735	149.0550	6.9457
50	0.10	1	144.6950	148.3837	151.7900	8.1067
50	0.10	5	143.0850	147.0508	152.6600	4.2997
50	0.10	10	143.0850	146.0212	151.1700	4.8660
50	0.20	1	144.3600	148.1307	153.2550	7.6497
50	0.20	5	143.1600	146.5560	152.6400	3.6863
50	0.20	10	143.0850	146.8235	151.3900	3.3380
100	0.05	1	145.2400	149.3670	155.8050	8.2390
100	0.05	5	143.4050	147.0952	151.0400	8.2877
100	0.05	10	143.0850	146.3405	151.0850	7.0273
100	0.10	1	145.5550	149.1558	155.9300	8.0133
100	0.10	5	143.4450	147.1760	153.8100	7.7197
100	0.10	10	143.0850	146.3407	150.2750	5.6863
100	0.20	1	144.4200	148.9328	152.4600	8.0840
100	0.20	5	143.4900	147.1508	151.2150	7.7720
100	0.20	10	143.0850	145.3347	149.8800	5.0417
150	0.05	1	145.9050	149.8665	154.6300	7.8173
150	0.05	5	143.9350	147.6715	151.9000	7.6320
150	0.05	10	143.1600	146.3883	150.8650	8.5130
150	0.10	1	144.8000	149.5798	153.6600	7.2060
150	0.10	5	143.6400	148.9512	154.2000	7.6677
150	0.10	10	143.0850	146.0257	150.3050	8.7243
150	0.20	1	146.7850	150.3227	154.2100	7.4393
150	0.20	5	143.5650	146.6768	152.4300	7.6840
150	0.20	10	143.4050	146.6275	151.3950	8.4743

Tabela 3.1: Resultados agregados das 30 execuções com 10 segundos por configuração.

População	Mutação	Elitismo	SP Mín	SP Médio	SP Máx	Tempo Médio (s)
20	0.05	1	143.0850	147.0955	152.0450	14.8060
20	0.05	5	143.1600	148.0340	152.6600	17.8600
20	0.05	10	143.6650	147.8985	155.7950	19.3530
20	0.10	1	143.1600	145.7245	148.8500	14.4140
20	0.10	5	143.1600	147.0930	150.7900	9.3570
20	0.10	10	143.4050	146.5800	151.4050	9.5930
20	0.20	1	143.1600	148.9995	155.9600	19.2240
20	0.20	5	143.1600	145.6580	149.8950	6.7060
20	0.20	10	145.4900	147.9660	150.7900	6.1250
50	0.05	1	145.1000	149.0820	152.8150	25.6500
50	0.05	5	143.4050	146.8310	150.7900	15.6350
50	0.05	10	143.4050	146.6840	147.8650	13.9710
50	0.10	1	144.3900	147.2705	149.6450	19.8750
50	0.10	5	143.4050	145.8635	150.2650	8.8850
50	0.10	10	143.0850	145.2265	149.8800	11.9660
50	0.20	1	144.1850	147.9490	151.6600	25.4420
50	0.20	5	143.0850	145.9790	147.6250	8.5820
50	0.20	10	143.4050	147.3110	152.6600	7.0520
100	0.05	1	145.4050	148.6010	151.6600	23.4730
100	0.05	5	143.6350	146.4580	149.2500	25.5790
100	0.05	10	143.1600	146.9320	149.0500	15.2580
100	0.10	1	144.3500	148.4315	153.6750	26.6600
100	0.10	5	143.7550	147.3855	149.8650	21.5420
100	0.10	10	143.4050	146.8640	154.1800	12.4530
100	0.20	1	144.7700	148.2130	151.5400	19.1980
100	0.20	5	143.1600	145.8475	147.9250	23.1530
100	0.20	10	143.1600	145.8060	150.3100	11.2900
150	0.05	1	145.1250	148.9005	152.0800	23.1300
150	0.05	5	146.6300	148.4120	150.6200	24.9900
150	0.05	10	143.0850	145.8730	149.4200	16.9020
150	0.10	1	145.3900	150.1870	156.0700	22.2250
150	0.10	5	144.3350	148.0905	151.5850	23.3340
150	0.10	10	143.3400	146.4095	149.1600	26.0800
150	0.20	1	148.3950	151.7725	153.8100	23.1910
150	0.20	5	144.0150	147.4535	151.0650	23.9620
150	0.20	10	144.0550	146.4650	148.7350	24.5230

Tabela 3.2: Resultados agregados das 10 execuções com 30 segundos por configuração.

A Tabela 3.3 apresenta uma seleção final das configurações mais promissoras, considerando simultaneamente o valor médio de SP, a consistência (baixa variabilidade), e o tempo de execução. A escolha foi guiada por um compromisso entre qualidade da solução e eficiência temporal. Foram priorizadas configurações com:

- Valor médio de *SP* inferior a 146.5;
- Tempo médio de execução inferior a 10 segundos;
- Mínimo absoluto próximo de `143.085`, frequentemente observado nas melhores soluções.

Configurações com valores médios mais baixos, mas tempos de execução muito elevados, foram descartadas para manter a viabilidade prática do algoritmo no contexto do problema.

População	Mutação	Elitismo	SP Mín	SP Médio	SP Máx	Tempo Médio (s)
50	0.10	10	143.085	146.0212	151.170	4.86
100	0.20	10	143.085	145.3347	149.880	5.04
50	0.05	10	143.085	146.6735	149.055	6.95
150	0.10	10	143.085	146.0257	150.305	8.72

Tabela 3.3: Configurações finais selecionadas com melhor equilíbrio entre desempenho e tempo.

A configuração (`População = 100, Mutação = 0.20, Elitismo = 10`) foi a escolhida como padrão para as execuções finais do algoritmo. Apresentou não apenas um dos melhores valores médios de SP, mas também um tempo de execução reduzido e comportamento estável entre execuções, o que a torna especialmente atrativa para o contexto de otimização sob restrições temporais.

3.4 Resultados

Para avaliar o desempenho final do algoritmo genético com os parâmetros otimizados (`populationSize = 50`, `mutationProb = 0.10`, `elitistParam = 10`), foram realizadas 10 execuções independentes, cada uma com um tempo máximo de 30 segundos.

A Tabela 3.4 apresenta os valores da função objetivo (SP médio), a distância máxima entre pares de servidores (SP máximo) e o tempo de execução até à melhor solução encontrada em cada execução.

Execução	SP Médio	SP Máximo	Tempo até Melhor Solução (s)
1	146.1750	990.0000	11.21
2	143.0850	998.0000	19.27
3	143.4050	1000.0000	17.96
4	143.4050	1000.0000	11.68
5	146.6400	997.0000	19.36
6	143.9600	998.0000	24.85
7	146.8650	998.0000	17.68
8	152.2400	994.0000	9.69
9	143.4050	1000.0000	22.55
10	143.1600	998.0000	13.70

Tabela 3.4: Resultados obtidos pelo GA em 10 execuções com limite de 30 segundos.

Observa-se que, embora o algoritmo genético tenha apresentado uma ligeira variabilidade entre execuções, os melhores valores atingidos são bastante competitivos. O valor mínimo da função objetivo foi de 143.0850, igual ao melhor valor encontrado pelo GRASP, o que demonstra a capacidade do GA em explorar eficazmente o espaço de soluções. O tempo médio até à melhor solução foi de 16.79 segundos, evidenciando uma maior dispersão temporal na convergência comparativamente ao GRASP.

A solução mais eficaz obtida (execução 2) corresponde à seguinte seleção de nós servidores:

$$20,\ 30,\ 65,\ 68,\ 90,\ 91,\ 107,\ 108,\ 131,\ 146,\ 163,\ 173$$

Esta seleção é idêntica à melhor solução encontrada pelo GRASP, o que valida a robustez dos dois métodos em identificar pontos de elevada qualidade no espaço de busca.

A Figura 3.1 apresenta a topologia da rede com a melhor solução obtida pelo GA, com os nós selecionados destacados a vermelho:

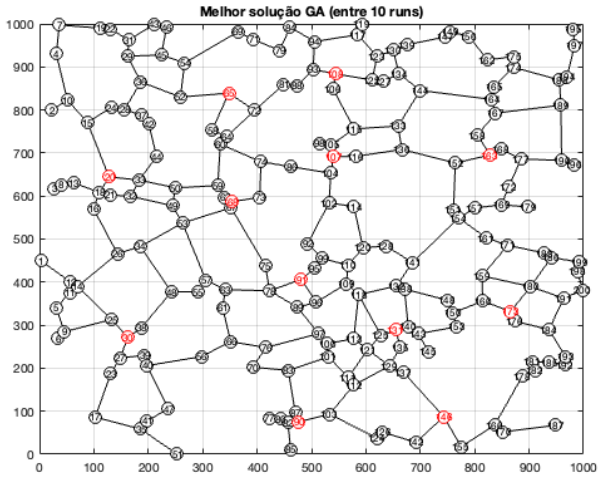

Figura 3.1: Melhor solução GA (entre 10 execuções), com os 12 nós selecionados destacados a vermelho.

Capítulo 4

ILP

4.1 Introdução

A Programação Linear Inteira (ILP) é um método exato de otimização que permite obter soluções ótimas garantidas para problemas cuja formulação pode ser expressa através de funções lineares e variáveis inteiras. Este tipo de abordagem é particularmente indicado para problemas combinatórios com estruturas bem definidas e restrições rígidas, como é o caso do problema presente.

Mais uma vez, o problema consiste em selecionar $n = 12$ nós de um grafo com $|N| = 200$ nós e $|A| = 250$ ligações, de forma a instalar controladores SDN, assegurando que a distância mais curta entre quaisquer dois controladores não excede o valor máximo $C_{max} = 1000$. O objetivo é minimizar o comprimento médio dos caminhos mais curtos entre cada nó da rede e o seu controlador mais próximo.

Para resolver este problema por ILP, é utilizada uma formulação baseada em variáveis binárias que indicam a seleção dos nós servidores e a afetação de cada nó da rede ao seu controlador mais próximo. A formulação matemática permite modelar com precisão tanto a função objetivo como as restrições do problema, nomeadamente: (i) o número exato de controladores a instalar, (ii) a obrigatoriedade de cada nó estar ligado a um controlador, e (iii) a limitação da distância entre quaisquer dois controladores.

A resolução do modelo ILP foi realizada utilizando o software `lpsolve`, com o ficheiro `LP` gerado automaticamente a partir de um script em `MATLAB`. A execução foi limitada a 5 minutos, tal como definido no enunciado, registando-se o valor da função objetivo da melhor solução encontrada, bem como o tempo de execução até à obtenção dessa solução.

Na secção seguinte, será apresentada a formulação matemática completa do problema, seguida da descrição do processo de geração do ficheiro `LP`, com excertos comentados do código desenvolvido. Serão também apresentados os resultados obtidos com a resolução por ILP e discutidas as suas implicações em termos de qualidade da solução e viabilidade computacional.

4.2 Formulação Matemática

A formulação do modelo de Programação Linear Inteira baseia-se na definição de variáveis binárias que codificam a seleção dos nós servidores e a afetação dos restantes nós ao seu controlador mais próximo. Utilizam-se as seguintes variáveis:

- $z_i \in \{0, 1\}$: indica se o nó i é selecionado como servidor;
- $g_s^i \in \{0, 1\}$: indica se o nó s é servido pelo nó i.

Com δ_s^i a representar a distância do caminho mais curto entre os nós s e i, o modelo ILP é definido da

seguinte forma:

$$
\begin{aligned}
\text{Minimizar} \quad & \sum_{s \in N} \sum_{i \in N} \delta_s^i \cdot g_s^i & \\
\text{sujeito a:} \quad & \sum_{i \in N} z_i = n & \\
& \sum_{i \in N} g_s^i = 1 & s \in N \\
& g_s^i \leq z_i & s \in N,\ i \in N \\
& z_i + z_j \leq 1 & i, j \in N : \delta_i^j > C_{max} \\
& z_i \in \{0, 1\} & i \in N \\
& g_s^i \in \{0, 1\} & s \in N,\ i \in N
\end{aligned}
$$

A função objetivo minimiza a soma das distâncias entre cada nó e o seu servidor atribuído. A primeira restrição assegura a seleção de exatamente n servidores. A segunda obriga cada nó a estar ligado a um e apenas um servidor. A terceira impõe que só pode servir outros nós quem for efetivamente selecionado como servidor. A quarta proíbe a seleção simultânea de dois servidores cuja distância exceda C_{max}.

Este modelo garante soluções viáveis que respeitam todas as restrições estruturais do problema, otimizando simultaneamente o critério de desempenho estabelecido.

4.3 Implementação

A geração do ficheiro `LP` foi realizada através de um script desenvolvido em `MATLAB`, que constrói uma representação textual do modelo de Programação Linear Inteira compatível com o formato de entrada do `lpsolve`. O processo é totalmente automatizado, permitindo a aplicação a redes de diferentes dimensões, desde que respeitado o formato de dados fornecido.

A função objetivo é definida como a soma ponderada das distâncias entre cada nó s e o servidor i que o serve, considerando apenas os pares distintos (s, i), uma vez que a distância de um nó para si próprio é nula e não contribui para a otimização:

```
fprintf(fid, 'min: ');
terms = [];
for s = 1:N
    for i = 1:N
        if s ~= i
            terms{end+1} = sprintf('%.6f*g_%d_%d', D(s,i), s, i);
        end
    end
end
fprintf(fid, '%s;\n', strjoin(terms, ' + '));
```

Segue-se a definição das restrições. Primeiro, impõe-se que sejam selecionados exatamente `n_servers` nós como servidores:

```
for i = 1:N
    terms{end+1} = sprintf('z_%d', i);
end
fprintf(fid, '%s = %d;\n', strjoin(terms, ' + '), n_servers);
```

De seguida, garante-se que cada nó s da rede é atribuído a exatamente um servidor:

```
for s = 1:N
    for i = 1:N
        terms{end+1} = sprintf('g_%d_%d', s, i);
    end
    fprintf(fid, '%s = 1;\n', strjoin(terms, ' + '));
    terms = [];
end
```

A restrição de coerência entre variáveis de afetação e de seleção de servidores é imposta com:

```
for s = 1:N
    for i = 1:N
        fprintf(fid, 'g_%d_%d <= z_%d;\n', s, i, i);
    end
end
```

Em seguida, são adicionadas as restrições de viabilidade estrutural: quaisquer dois nós cuja distância seja superior a C_{max} não podem ser ambos selecionados como servidores:

```
for i = 1:N
    for j = i+1:N
        if D(i,j) > Cmax
            fprintf(fid, 'z_%d + z_%d <= 1;\n', i, j);
        end
    end
end
```

Por fim, todas as variáveis são explicitamente definidas como binárias, em conformidade com a sintaxe do `lpsolve`:

```
for i = 1:N
    fprintf(fid, 'bin z_%d;\n', i);
end
for s = 1:N
    for i = 1:N
        fprintf(fid, 'bin g_%d_%d;\n', s, i);
    end
end
```

Este gerador garante a conformidade estrita com a formulação matemática previamente definida, permitindo a resolução do problema com exatidão por via do `lpsolve` com um simples comando de execução.

4.4 Resultados

A resolução do modelo, como já referido, foi realizada através da ferramenta `lpsolve`, com um tempo máximo de execução de 5 minutos, conforme especificado no enunciado do projeto. O ficheiro LP foi gerado automaticamente pelo *script* descrito na secção anterior e submetido diretamente ao `solver`.

O tempo total de execução foi de 5 minutos, tendo o processo sido interrompido ao atingir o limite

máximo estipulado. A melhor solução inteira encontrada nesse intervalo apresentou um valor da função objetivo de 29017, mas este valor corresponde à soma total das distâncias entre cada nó e o seu servidor mais próximo. Para comparação direta com os resultados dos métodos metaheurísticos, que otimizam a distância média por nó, este valor deverá ser dividido por $|N| = 200$, resultando em:

$$145.085$$

A solução ótima não foi encontrada dentro do tempo disponível, com o `solver` a reportar um *gap* de 13.9% em relação à solução relaxada de referência (valor base de 25464.4). Estes indicadores confirmam a dificuldade computacional do problema e o elevado custo de resolução exata em instâncias de grande dimensão.

Apesar de não ter sido atingida a otimalidade, a solução obtida é admissível e respeita todas as restrições do modelo. O valor da função objetivo é ligeiramente superior ao das melhores soluções heurísticas, o que sugere que estas se aproximam bastante do ótimo ou, pelo menos, de soluções de elevada qualidade. O esforço computacional registado evidencia a escalabilidade limitada dos métodos exatos face a problemas com elevado número de variáveis e restrições.

Os nós selecionados como servidores na melhor solução encontrada foram os seguintes:

$$14,\ 18,\ 40,\ 52,\ 78,\ 90,\ 107,\ 108,\ 129,\ 150,\ 154,\ 163$$

A Figura 4.1 apresenta a topologia da rede com a localização desta solução, destacando visualmente os nós selecionados a vermelho:

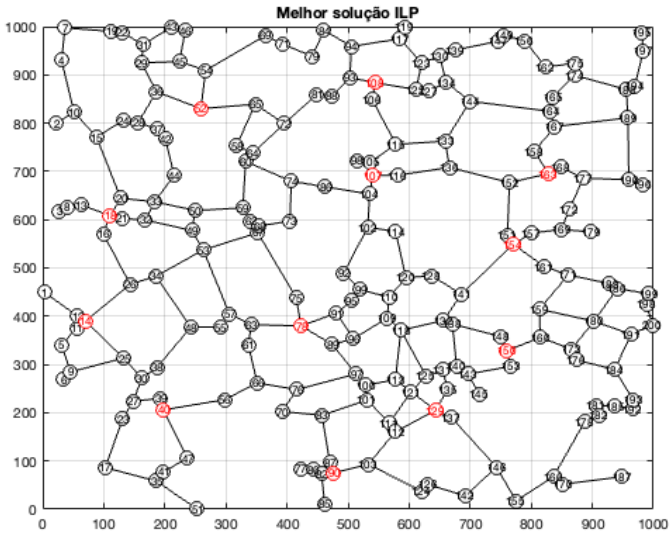

Figura 4.1: Melhor solução ILP, com os 12 nós selecionados destacados a vermelho.

Capítulo 5

Análise Comparativa

A Tabela 5.1 apresenta uma síntese dos resultados obtidos pelos três métodos implementados: GRASP, Algoritmo Genético (GA) e Programação Linear Inteira (ILP). Os valores refletem os dados efetivos registados nas execuções realizadas, nomeadamente o valor mínimo, médio e máximo do SP (Shortest Path médio) entre os nós e os respetivos tempos de convergência ou execução.

Tabela 5.1: Resumo comparativo dos resultados dos métodos de otimização

Método	**SP Mínimo**	**SP Médio**	**SP Máximo**	**Tempo Total por Execução**
GRASP	143.0850	143.0850	143.0850	30 s
GA	143.0850	144.8120	152.2400	30 s
ILP	145.0850	145.0850	145.0850	5 min

No método GRASP, observou-se uma total estabilidade: todas as 10 execuções resultaram exatamente na mesma solução, com SP médio de 143.0850, atingida em média aos 10.35 segundos. Esta consistência demonstra não só a robustez da construção gulosa aleatorizada e da intensificação via busca local, como também a eficácia das otimizações implementadas na versão final.

Já o Algoritmo Genético apresentou maior variabilidade entre execuções, com SP médio global de 144.8120, valor máximo de 152.2400 e valor mínimo igual ao do GRASP (143.0850), demonstrando que é possível alcançar soluções equivalentes às melhores, mas com menor frequência. Esta dispersão reflete o caráter estocástico do método e o seu mecanismo de exploração baseado em população.

Relativamente à Programação Linear Inteira (ILP), o *solver* foi interrompido ao fim do tempo limite de 5 minutos, tendo retornado uma solução viável com valor de SP médio igual a 145.0850. Este valor é superior ao das melhores soluções heurísticas, o que confirma que o método exato, apesar de teoricamente garantir otimalidade, não conseguiu convergir para o ótimo global dentro do tempo estipulado.

Em termos de custo computacional, o ILP revelou-se significativamente mais exigente, sendo superado pelos métodos metaheurísticos em termos de escalabilidade e tempo por iteração. Contudo, a sua principal mais-valia reside na capacidade de validação e benchmarking, servindo como base de comparação para os métodos não exatos.

Em suma, o GRASP destacou-se pelo equilíbrio entre tempo, desempenho e estabilidade, sendo recomendado como abordagem de referência. O GA mostrou-se competitivo e versátil, com espaço para melhorias através de refinamentos adicionais. Já o ILP confirmou-se adequado para instâncias menores ou como ferramenta de validação, sendo menos eficiente para casos de grande dimensão como o presente.

Capítulo 6

Conclusão

O presente mini-projeto teve como objetivo a aplicação de métodos de otimização, metaheurísticos e exatos, à resolução de um problema de seleção de nós servidores em redes definidas por software (SDN). Através das abordagens GRASP, Algoritmo Genético (GA) e Programação Linear Inteira (ILP), foi possível explorar diferentes paradigmas de modelação e resolução para um mesmo problema com elevada complexidade combinatória.

O método GRASP destacou-se pela sua elevada estabilidade, conseguindo consistentemente alcançar a melhor solução em todas as execuções com baixo tempo médio até convergência. As otimizações aplicadas à versão base demonstraram-se cruciais para este desempenho, assegurando eficiência e robustez sem comprometer a qualidade das soluções. A integração de uma construção gulosa aleatorizada com uma busca local do tipo *Steepest Ascent Hill Climbing* provou ser uma estratégia eficaz na exploração e intensificação da vizinhança de soluções promissoras.

O Algoritmo Genético revelou-se igualmente capaz de encontrar soluções de elevada qualidade, explorando o espaço de soluções através de operadores genéticos que promovem a diversidade e previnem convergência prematura. Apesar de apresentar maior variabilidade nos resultados, foi possível observar que as melhores execuções do GA rivalizam com os resultados do GRASP, validando o seu potencial em contextos de otimização populacional.

A formulação e resolução exata por ILP permitiu obter uma solução admissível em tempo limitado, mas sem alcançar a otimalidade. A análise do *gap* residual reforça a dificuldade computacional do problema quando abordado por métodos exatos, sobretudo em redes de grande dimensão. Ainda assim, a solução obtida oferece uma referência útil para avaliação comparativa dos métodos metaheurísticos.

Como principais conclusões, verifica-se que as abordagens metaheurísticas, em particular o GRASP e o Algoritmo Genético, demonstraram desempenho superior à abordagem exata em termos de tempo de execução, especialmente em problemas de maior dimensão. Em alguns casos, estas abordagens conseguiram também obter soluções de melhor qualidade. O método GRASP destacou-se como a solução mais equilibrada, apresentando simultaneamente elevada consistência e excelente desempenho. Por outro lado, o Algoritmo Genético revelou-se competitivo, com potencial para superar o GRASP em instâncias com maior diversidade estrutural, beneficiando da sua natureza populacional. A abordagem exata por Programação Linear Inteira mostrou-se adequada sobretudo como ferramenta de validação, oferecendo uma referência formal de qualidade das soluções, embora a sua aplicabilidade prática esteja limitada pelo custo computacional associado a instâncias de grande escala.