Universidade Federal
do Rio de Janeiro
Escola Politécnica

PROPOSTAS DE META-HEURÍSTICAS PARA O PROBLEMA *MINI-MAX K-ROOTED SPANNING FOREST*

Marcos Aurélio Constant de Souza Filho

Projeto de Graduação apresentado ao Curso de Computação e Informação da Escola Politécnica da Universidade Federal do Rio de Janeiro como parte dos requisitos necessários para a obtenção do grau de Engenheiro de Computação e Informação.

Orientador: Luidi Gelabert Simonetti

Rio de Janeiro
Setembro de 2018

PROPOSTAS DE META-HEURÍSTICAS PARA O PROBLEMA *MINI-MAX K-ROOTED SPANNING FOREST*

Marcos Aurélio Constant de Souza Filho

PROJETO SUBMETIDO AO CORPO DOCENTE DO CURSO DE COMPUTAÇÃO E INFORMAÇÃO DA ESCOLA POLITÉCNICA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE ENGENHEIRO DE COMPUTAÇÃO E INFORMAÇÃO.

Examinadores:

Prof. Luidi Gelabert Simonetti, D. Sc.

Prof. Pedro Braconnot Velloso, D. Sc.

Profa. Laura Silvia Bahiense da Silva Leite, D. Sc.

RIO DE JANEIRO, RJ – BRASIL
SETEMBRO DE 2018

Constant de Souza Filho, Marcos Aurélio

Propostas de Meta-Heurísticas para o problema *minimax K-rooted Spanning Forest*/Marcos Aurélio Constant de Souza Filho. – Rio de Janeiro: UFRJ/POLI – COPPE, 2018.

XI, 61 p.: il.; 29, 7cm.

Orientador: Luidi Gelabert Simonetti

Projeto (graduação) – UFRJ/ Escola Politécnica/ Curso de Computação e Informação, 2018.

Referências Bibliográficas: p. 39 – 40.

1. Otimização Combinatória. 2. Teoria dos Grafos. 3. Floresta Geradoras. 4. Meta-Heurística. I. Gelabert Simonetti, Luidi. II. Universidade Federal do Rio de Janeiro, Escola Politécnica/ Curso de Computação e Informação. III. Título.

Agradecimentos

Primeiramente, gostaria de agradecer aos meus pais, Marcia e Marcos, que sempre me apoiaram e me incentivaram para que eu chegasse até esta fase da minha vida. Agradeço também a minha tia, Vera, que me acolheu em sua casa durante toda minha graduação.

Além dos citados acima, agradeço a todos os amigos com os quais compartilhei momentos de alegrias e tristezas durante esta trajetória.

Agradeço também a todos os professores da UFRJ, de Engenharia de Computação e Informação ou não, com os quais tive o prazer de aprender, em especial ao meu orientador, Luidi Simonetti, por compartilhar seus conhecimentos e me auxiliar durante este trabalho.

Por fim, agradeço a todos que não foram citados mas diretamente ou indiretamente fizeram parte da minha formação.

Resumo do Projeto de Graduação apresentado à Escola Politécnica/COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Engenheiro de Computação e Informação.

PROPOSTAS DE META-HEURÍSTICAS PARA O PROBLEMA *MINI-MAX K-ROOTED SPANNING FOREST*

Marcos Aurélio Constant de Souza Filho

Setembro/2018

Orientador: Luidi Gelabert Simonetti

Curso: Engenharia de Computação e Informação

O problema *mini-max K-Rooted Spanning Forest* é tal que, dado $G = (V, E)$ um grafo não direcionado, conexo e simples onde para cada arestas $e_{(i,j)} \in E$ existe um custo associado $c_{(i,j)}$ e além disso um conjunto de raízes $R = \{r_1, \ldots, r_k\}, \quad R \subseteq V$, desejamos encontrar uma floresta geradora F formada de árvores com raízes em R de forma à minimizar o custo da maior árvore da floresta F. Neste trabalho são apresentados propostas de heurísticas e meta-heurísticas para resolução deste problema. Essas meta-heurísticas são baseadas em métodos conhecidos, como por exemplo o *simulated annealing* e algoritmos genéticos. Ao final são realizados comparações entre os métodos utilizados, e uma breve discussão sobre os resultados.

Palavras-Chave: Otimização Combinatória, Teoria dos Grafos, Floresta Geradoras, Meta-Heurística.

Abstract of the Undergraduate Project presented to Poli/COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Computer and Information Engineer.

PROPOSAL OF METAHEURISTIC FOR THE MINI-MAX K-ROOTED SPANNING FOREST PROBLEM

Marcos Aurélio Constant de Souza Filho

September/2018

Advisor: Luidi Gelabert Simonetti

Course: Computer and Information Engineering

Given a connected, undirected and simple graph $G = (V, E)$ that for each edge $e_{(i,j)} \in E$ existis an associated cost $c_{(i,j)}$, moreover, let $R = \{r_1, \ldots\}, \quad R \in V$ be a set of roots, we wish to find the spanning forest F with trees rooted in R that the cost of the most expensive tree in the spanning forest is minimial among all others spanning forest. The aim of this project is to present proposals of heuristics and metaheuristics to solve this problem. The metaheuristics are based on well-known methods like simulated annealing and genetic algorithm. And finally, a short discussion about the results will be made, comparing the results of each method.

Keywords: Combinatorial Optimization, Graph Theory, Spanning Forest, Metaheuristic.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Algoritmos

Capítulo 1

Introdução

Neste capítulo são apresentados a definição do problema, o objetivo e como este trabalho está estruturado.

1.1 O Problema

O problema *Mini-Max K-rooted Spanning Forest* (*K-MMSFP* abreviando) tem por objetivo encontrar a floresta geradora de um grafo não direcionado de maneira que o custo da árvore geradora mais cara dentre as K árvores geradoras desta floresta seja mínimo.

Existem diversas aplicações práticas para esse problema, YAMADA *et al.* cita como exemplo o problema de conectar cidades a usinas de energia de maneira justa, ou seja, o custo que cada usina de energia irá gastar para realizar as conexões entre as cidades que estão sob seu uso deve ser mínimo. Outras podem ser encontradas em [4] e [2].

Para o caso de $K = 1$ este problema é conhecido como *Minimum Spanning Tree* (*MST* abreviando) e existem algoritmos com complexidade polinomial capazes de resolvê-lo, como por exemplo o algoritmo de Prim [5] ou algoritmo de Kruskal [6]. YAMADA *et al.* prova que *K-MMSFP* é *NP-Difícil* para o número de raízes $K \geq 2$ transformando este problema em um Problema de Partição.

De uma maneira mais formal *K-MMSFP* pode ser demonstrado da seguinte maneira. Seja $G = (V, E)$ um grafo não direcionado, conexo e simples, onde $V : \{1, 2, \ldots, N\}$ é o conjunto de vértices e $E \subseteq N \times N$ é o conjunto de arestas. Para cada aresta $(i, j) \in E$ existe um custo associado $c_{(i,j)} \in \mathbb{Z}_{\geq 0}$.

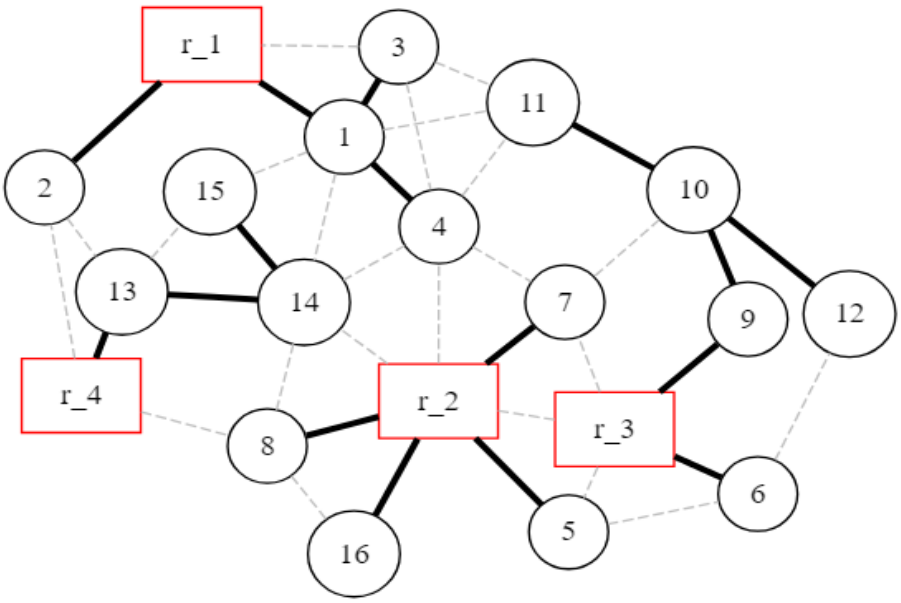


Figura 1.1: Exemplo de uma floresta geradora para $K = 4$

Além disso, seja $R : \{r_1, \ldots, r_k\}$ o conjunto de raízes tal que $R \subseteq V$ e F a floresta geradora formada pelo conjunto de árvores geradoras T_r, $\forall r \in R$. Dado isso podemos definir a função objetivo.

$$\begin{aligned} &\min \ W(F) \\ &\quad W(F) = \max_{\forall T \in F} \{C(T)\} \\ &\text{Onde,} \\ &\quad C(T) = \sum_{\forall (i,j) \in T} c_{(i,j)} \end{aligned} \tag{1.1}$$

Desta maneira queremos encontrar a floresta geradora F^* dentre todas a floresta geradoras possíveis de G que minimiza $W(F)$.
Conceitos apresentados acima serão explicados mais adiante na seção 3.

1.2 Proposta

Apesar da necessidade de algoritmos que encontrem a solução ótima para determinados problemas, atualmente a computação vem cada vez mais realizando o *trade-off* entre precisão da solução e tempo, desta forma, podendo encontrar soluções suficientemente boas em um tempo viável.

Tendo isso em vista, a proposta principal deste trabalho consiste em desenvolver heurísticas determinísticas e meta-heurísticas, baseadas em processos físicos e biológicos, eficientes para a solução do problema descrito 1.1 e comparar os resultados obtidos por essas soluções com os encontrados em trabalhos relacionados.

Embora os trabalhos já realizados apresentarem propostas de soluções eficientes para resolução deste problema de forma exata, essas soluções normalmente são para instâncias pequenas ou apenas para duas raízes. Desta forma, analisar os casos de instâncias maiores e com mais raízes é uma proposta secundária do trabalho.

1.3 Organização do Trabalho

Este trabalho contém 6 capítulos. O capítulo 2 apresenta uma revisão sobre propostas existentes na literatura.

No capítulo 3 são apresentados definições e conceitos relacionados a grafos, *simulated annealing* e algoritmos genéticos necessários para o entendimento do trabalho e do problema.

No capítulo 4 são apresentados as propostas de algoritmos, os seus respectivos pseudocódigos e seus detalhes de implementação.

No capítulo 5 são apresentados as instâncias utilizadas, os resultados obtidos e suas devidas comparações.

Por fim, o capítulo 6 apresenta as considerações finais, a conclusão que pode ser tomada pelos resultados obtidos e trabalhos futuros.

Capítulo 2

Trabalhos Relacionados

Nesta seção serão apresentadas soluções propostas por alguns artigos utilizados como referência.

2.1 Algoritmos existentes

YAMADA *et al.* [3] foram os primeiros a introduzir o problema e provar que ele era *NP-Difícil*. Eles propõem uma heurística baseada em uma busca local. Utilizando o custo da subárvore de um determinado vértice somado com o custo da aresta que liga este vértice a outra árvore, é verificado se a função objetivo diminui. Caso ela diminua, a aresta testada é conectada ao vértice e a que o ligava a antiga árvore removida.

TAKAHASHI e KATAOKA [2] apresentam o primeiro algoritmo exato para solução do problema, um algoritmo *branch-and-bound*, onde novos ramos(*branches*) são gerados a partir de seleções de arestas, de modo que uma aresta pode estar ou não em um ramo se esta forma uma componente conexa com a raiz. Além disso ele utiliza versões modificadas do algoritmo de árvore geradora mínima para propor limitantes superior e inferior que serão utilizados pelo algoritmo proposto por ele.

MEKKING e VOLGENANT [4] também propõem um algoritmo *branch-and-bound*, porém este é baseado em seleções de nós. Ele também introduz limitantes derivados dos propostos acima.

DA CUNHA *et al.* [1] propõem um algoritmo *branch-and-cut* para a resolução do problema. Seus limitantes são dados pelas soluções da relaxação linear do problema.

Capítulo 3

Conceitos e Definições

No presente capítulo serão introduzidos conceitos e definições nas quais estão baseados este trabalho.

3.1 Grafo

Grafo é uma abstração matemática que nos permite codificar relações entre pares de objetos. O conjunto de objetos forma os vértices e as relações entre esses objetos formam as arestas. Arestas em um grafo indicam uma relação, ou assimétrica ou simétrica [7].

Normalmente utiliza-se como notação para representar um grafo $G = (V, E)$, onde V é conjunto de vértices e E é o conjunto de arestas que formam o grafo G.

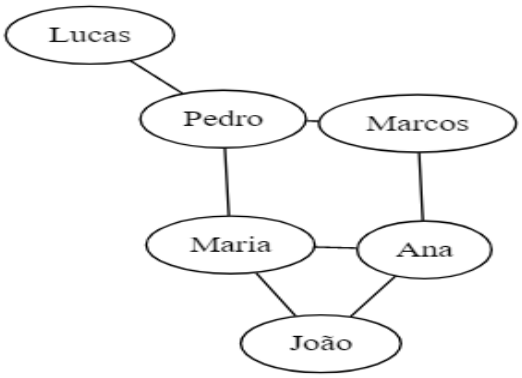


Figura 3.1: Grafo que codifica amizades entre pessoas

Um grafo constituído de relações assimétricas é chamado de grafo direcionado, enquanto um grafo constituído por relações simétricas é chamado de grafo não direcionado. Além disso, arestas podem conter custos que codificam a intensidade das relações entre os objetos, vamos utilizar a notação $c_{(i,j)}$, onde (i, j) é uma aresta do grafo, para representar o custo de uma aresta.

Ao modificar a figura 3.10 codificando os custos como sendo tempo de amizade entre as pessoas, damos origem a um grafo com pesos.

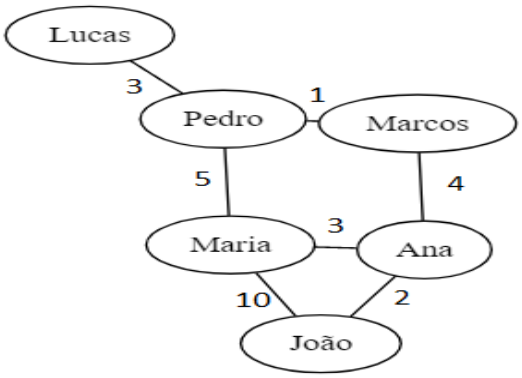
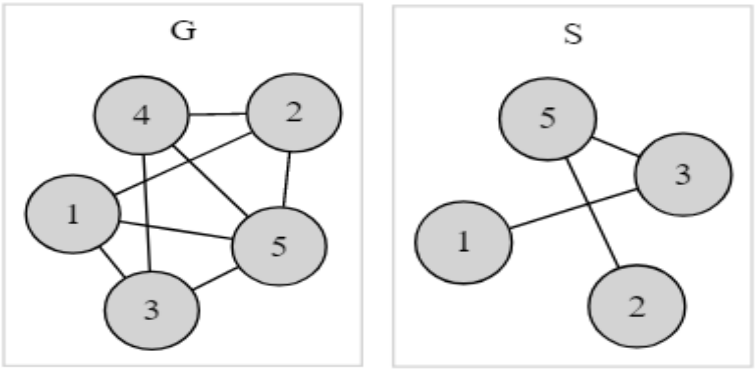


Figura 3.2: Grafo que codifica amizades e o tempo de amizade entre pessoas.

3.1.1 Subgrafo

Dado os grafos $S = (V', E')$ e $G = (V, E)$, o grafo S é dito subgrafo de G se o conjunto de vértices $V' \subseteq V$ e o conjunto de arestas $E' \subseteq E$.



(a) Grafo G (b) Grafo S

Figura 3.3: Grafos S subgrafo de G.

3.1.2 Caminho em um grafo

Um caminho é uma sequência de vértices $S : \{v_1, \ldots, v_n\}$ que para cada par de vértices consecutivos (i, j) existe uma aresta $e_{(i,j)}$ no grafo. Em um caminho simples cada vértice só aparece uma vez em S.

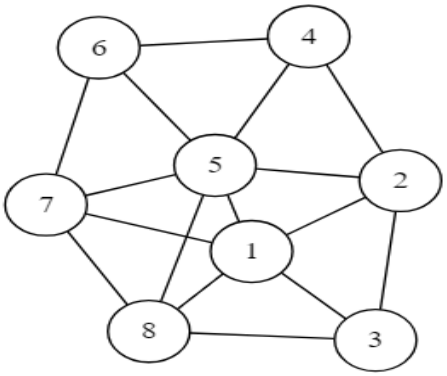
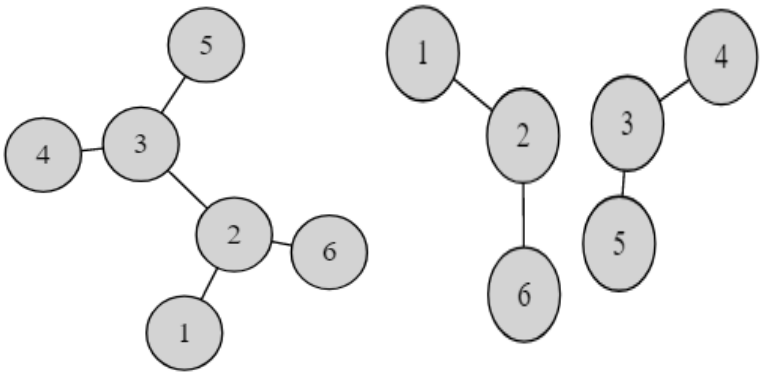


Figura 3.4: Exemplo de caminho em um grafo.

Na figura 3.1.2, podemos verificar que a sequência $S : \{3, 1, 8, 5, 6\}$, pela definição, é um caminho simples do grafo acima.

3.1.3 Grafo Conexo

Um grafo $G = (V, E)$ é dito conexo se existe ao menos um caminho que ligue cada par de vértices do grafo. Caso não exista tal caminho o grafo é considerado não conexo, e portanto existem K componentes conexos tal que $K \in \{2, \ldots, |V|\}$. Uma componente conexa é um subgrafo conexo maximal de G.



(a) Grafo Conexo (b) Grafo não conexo

Figura 3.5: Exemplo de grafos conexo e não conexo

Na figura acima, podemos notar que o grafo não conexo contêm dois componentes conexos, $C_1 = \{1, 2, 6\}$ e $C_2 = \{3, 4, 5\}$.

3.1.4 Árvores

Um grafo $G = (V, E)$ é dito uma árvore se G é conexo e acíclico, ou seja, não possui um subconjunto de arestas que formem um ciclo. Árvores possuem exatamente

$|V| - 1$ arestas. Para cada par de vértices $i, j \in V$ existe somente um caminho que liga i a j em G.

No contexto de grafos, a raiz de uma árvore é um vértice que induz uma sequência de ligação entre os outros vértices. Essas ligações são denominadas ramos, e vértices que não possuem ramos são denominados folhas da árvore. Qualquer vértice do grafo pode ser denominado raiz e para cada raiz a árvore terá uma sequência de ligações diferentes.

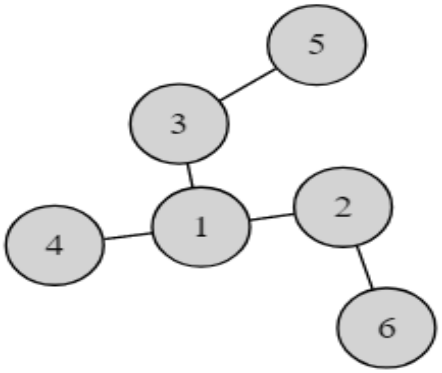


Figura 3.6: Exemplo de uma Árvore

3.1.5 Árvores Geradoras

Uma árvore geradora de um grafo conexo $G = (V, E)$ é um subgrafo árvore $T = (V', E')$ onde $V' = V$, $E' \subset E$.

O custo de uma árvore geradora é definido da seguinte maneira:

$$C(T) = \sum_{\forall (i,j) \in E'} c_{(i,j)} \tag{3.1}$$

Árvore Geradora Mínima

Dado a definição de árvore geradora, a árvore geradora mínima T' (MST abreviando em inglês) é a árvore geradora do grafo G cujo somatório das suas arestas é mínimo. Para grafos G onde o peso das arestas são iguais, qualquer árvore geradora de G é uma MST.

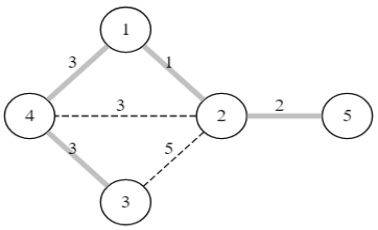


Figura 3.7: Exemplo de uma MST

Na figura 3.1.5, as arestas em cinza representam uma MST de custo 9. Podemos notar que se trocarmos a resta $(1, 4)$ pela aresta $(2, 4)$, o custo da MST se manteria o mesmo.

3.1.6 Floresta Geradora

O conceito de floresta geradora normalmente é aplicado para grafos desconexos, de maneira que uma floresta geradora é o conjunto formados pelas árvores geradoras de cada componente conexa do grafo. Entretanto esse conceito pode ser aplicado para grafos conexos também. Com isso, podemos definir uma floresta geradora da seguinte maneira:

Dado um grafo $G = (V, E)$. F é denominado uma floresta geradora de G se:

- O conjunto de vértices da árvore T é subconjunto de V para toda árvore na floresta.

$$V_i \subseteq V \qquad \forall i \in [1, k] \tag{3.2}$$

- O conjunto formado pela união dos vértices de cada árvore de F_k é igual ao conjunto de vértices de G.

$$\bigcup_{i=1}^{k} V_i = V \tag{3.3}$$

- Cada vértice de G pertence somente a uma árvore da floresta.

$$\bigcap_{i=1}^{k} V_i = \emptyset \tag{3.4}$$

- O somatório das cardinalidades dos conjuntos de arestas de cada árvore em F é igual $|V| - k$ onde k é o numero de árvores que formam F.

$$\sum_{i=1}^{k} |E_i| = |V| - k \tag{3.5}$$

Além disso, para o K-MMSFP apresentado neste trabalho, partindo do princípio que o grafo é conexo, iremos definir também um conjunto de vértices $R = \{r_1, \ldots, r_k\}$ onde cada vértice que pertence a esse conjunto R deve pertencer a árvores geradoras distintas do conjunto de árvores que formam a floresta geradora $F_k = \{T_1, \ldots, T_k\}$.

Com isso, o custo de uma floresta geradora F_k é definido como sendo o maior custo entre os custos das árvore pertencente a F_k

$$W(F_k) = \max_{i \in 1 \ldots k} \{C(T_i)\} \tag{3.6}$$

3.2 Cadeia de Markov

Cadeia de Markov é uma ferramenta matemática que permite modelar a dinâmica de sistemas aleatórios. Ela pode ser definida como sendo um conjunto finito ou infinito de estados para o qual existe uma matriz de transição P, onde P_{ij} denota a probabilidade de transição do estado i para o j. Uma transição é dada a cada tempo discreto t.

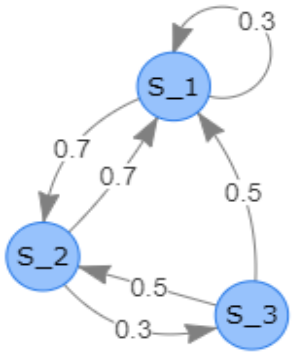


Figura 3.8: Exemplo de Cadeia de Markov

3.2.1 Propriedades

Nesta subseção serão apresentadas algumas propriedades de uma Cadeia de Markov. No seu livro, HÄGGSTRÖM [8] apresenta essas e outras propriedades de forma mais detalhada.

Memoryless

A Cadeia de Markov possui uma propriedade chamada de *Sem memória*. Esta propriedade diz que a probabilidade de estar num estado em um tempo t depende somente do estado no tempo anterior.

Matematicamente falando, suponha uma sequência de variáveis aleatórias $X = \{X_0, \ldots, X_k\}$. A probabilidade de ocorrer uma transição para o estado j dado que o estado atual é i pode ser escrito da seguinte maneira.

$$P_{ij}^{k+1} = P[X_{k+1} = j | X_k = i] \tag{3.7}$$

Distribuição estacionária

Seja uma Cadeia de Markov com k estados e matriz de transição P. Chamamos $\pi = (\pi_1, \ldots, \pi_k)$ de vetor de distribuição estacionária se:

- $\sum_{i=1}^{k} \pi_i = 1$
- $\sum_{i=1}^{k} \pi_i P_{ij} = \pi_j$ ou seja, $\pi P = \pi$

3.2.2 Cadeias de Markov reversíveis

Dado uma Cadeia de Markov com conjuntos de estados S e matriz de transição P. A reversibilidade ocorre nesta cadeia se existe uma distribuição π que satisfaz a seguinte igualdade.

$$\pi_i P_{ij} = \pi_j P_{ji} \quad \forall i, j \in S \tag{3.8}$$

Caso exista tal distribuição π que satisfaça essa igualdade, essa distribuição também é uma distribuição estacionária.

3.3 *Metropolis-Hasting*

O algoritmo de *Metropolis-Hasting* tem como objetivo gerar amostras de um espaço de soluções S do qual não se tem conhecimento. A ideia é induzir uma distribuição estacionária π de interesse a uma Cadeia de Markov conhecida. Para isso devemos alterar a matriz de transição da Cadeia de Markov original.

Vamos supor que queremos induzir uma distribuição estacionária π a uma Cadeia de Markov de conjunto de estados S e matriz de transição P, onde $\pi_i = \frac{f(S_i)}{Z}$ dado que $Z = \sum_{\forall s \in S} f(s)$. Vamos adicionar uma função de aceitação $\alpha(i, j)$ de modo que ela possa moldar a matriz de transição para que qualquer distribuição π seja estacionária. Desta maneira iremos ter uma nova matriz de transição P' onde a probabilidade de transicionar do estado S_i para S_j é dada por:

$$P'_{ij} = \begin{cases} P_{ij}\alpha(i,j) & \text{se } i \neq j \\ 1 - \sum_{\forall k \neq i \in S} P_{ik}\alpha(i,k) & \text{se } i = j \end{cases} \tag{3.9}$$

Com isso, como queremos garantir que a distribuição π seja estacionária, vamos utilizar a nova matriz P' na equação 3.8 e com isso calcular o valor de $\alpha(i, j)$ que satisfaça a equação para garantir que π é estacionário.

CHIB e GREENBERG [9] mostram que o valor $\alpha(i, j)$ deve ser:

$$\alpha(i,j) = \min\left(1, \frac{\pi_j P_{ji}}{\pi_i P_{ij}}\right) \tag{3.10}$$

Para métodos de otimização, a distribuição estacionária π normalmente está relacionada com a função objetivo. Com isso, a função de aceitação permite a exploração

de estados com uma função objetivo maior, que pode levar a saída de uma região de máximo ou mínimo local para uma que possa conter melhores soluções.

3.4 *Simulated Annealing*

Simulated Annealing é uma meta-heurística probabilística baseado em um processo físico de aquecimento e resfriamento de um sólido para buscar seu estado de menor energia[10]. Na computação este método é muito utilizado em problemas de otimização combinatória onde o espaço de busca S é muito amplo.

Algoritmo 1: Simulated Annealing

Entrada: t_0: Temperatura inicial
Entrada: s_0: Estado inicial $\in S$
Entrada: β: Constante de resfriamento

```
1  Início
2      s_i ← s_0
3      t ← t_0
4      steps ← 0
5      Enquanto t > 0 faça
6          s_j ← Modifica(s_i)
7          se f(s_j) ≤ f(s_i) então
8              s_j ← s_i
9          fim
10         senão
11             se rand(0,1) < e^((f(i)−f(j))/t) então
12                 s_j ← s_i
13             fim
14         fim
15         steps ← steps + 1
16         t ← AlteraTemperatura(t, steps, β)
17     Fim
18 Fim
```

O algoritmo se inicia atribuindo os recebidos de estado inicial e da temperatura inicial como sendo os valores atuais. A seguir, enquanto a temperatura não se reduzir a zero, um novo estado é gerado, o valor desse novo estado é calculado e comparado com o valor do estado atual. Caso esse valor seja menor, o estado atual assume o valor do novo estado, caso contrário, com uma certa probabilidade o estado atual também assume o valor do novo estado. Após isto, a temperatura é modificada.

A ideia deste algoritmo é similar ao algoritmo *Metropolis-Hasting*. Entretanto

a distribuição estacionária π é definida por uma função objetivo $f(S_i)$ aplicada a distribuição de *Boltzmann* , ou seja, $\pi_i = e^{\frac{-f(S_i)}{T}}$. Como a função de aceitação depende da variável T, que representa a temperatura, e como a cada iteração a temperatura se altera, a matriz de transição também é alterada. Sendo assim, dando origem a uma sequência de Cadeias de Markov ao longo do tempo. A ideia de alterar a matriz de transição é dar opções do algoritmo realizar buscas em toda região viável a fim de não ficar preso em um ponto de máximo ou mínimo local e, conforme o tempo passa, ir restringindo a busca apenas às soluções que melhorem a função objetivo.

3.5 Algoritmo Genético

O Algoritmo Genético é uma meta-heurística que simula o processo de seleção natural, ou seja, no algoritmo existe uma população de indivíduos que possuem um nível de aptidão. Um indivíduo é formado por genes, que representam variáveis do problema, o conjunto de genes de um indivíduo é chamado de cromossomo.

Cada indivíduo pode se reproduzir com outros, sofrer mutações, morrer ou sobreviver. A ideia é que a cada geração a população atual gere indivíduos com maior aptidão que a anterior.

Em um problema de otimização, normalmente, um cromossomo representa uma solução viável do problema, e sua aptidão corresponde a função objetivo.

Algoritmo 2: Algoritmo Genético

Entrada: T: Tamanho da população
Entrada: t_c: Taxa de cruzamento
Entrada: t_m: Taxa de mutação
Início
  $P \leftarrow GeraIndividuos(T)$
  $A \leftarrow Aptidao(P)$
  Enquanto *Critério de parada não satisfeito* **faça**
    $Pais \leftarrow SelecionaPais(P, A)$
    $Filhos \leftarrow Reproduz(Pais, t_c)$
    $P \leftarrow P + Filhos$
    $P \leftarrow Muta(P, t_m)$
    $A \leftarrow Aptidao(P)$
    $P \leftarrow Sobrevive(P, A)$
  Fim
  Retorna indivíduo mais apto
Fim

3.5.1 Etapas de um algoritmo genético

No pseudocódigo acima, podemos observar diversas etapas que devem ser realizadas durante a execução do algoritmo. Os tópicos abaixo irão mostrar o que cada etapa significa.

Geração Inicial

Existem diversas maneiras de gerar uma população inicial. Normalmente a mais utilizada é a totalmente aleatória pois a aleatoriedade garante diversidade na população.

Critério de Parada

Uma forma de definir o critério de parada normalmente é estabelecer um número máximo de gerações que irão existir. Outra forma pode ser interromper as gerações se ocorrer um número K de gerações onde a melhor solução encontrada não se altera.

Aptidão

O cálculo da aptidão de cada indivíduo depende do problema. Ela serve como base para escolher a elite de cada população.

Seleção dos Pais

A ideia é dar preferência a indivíduos da elite, porém utilizar somente esses indivíduos pode ocasionar uma convergência prematura, levando a resultados insatisfatórios. Pode ser interessante também a escolha de indivíduos que se diferem muito.

Reprodução

Na reprodução é realizada a combinação dos cromossomos de pares de indivíduos, denominados pais, para geração de novos indivíduos a partir deles. O número de novos indivíduos por par depende da técnica de reprodução que está sendo utilizada. Algumas técnicas de reprodução são:

- Reprodução por um ponto:
  Uma posição aleatórias é escolhida, depois os cromossomos dos pais são divididos em dois pedaços limitados por essa posição e depois cruzados para criação de dois filhos.

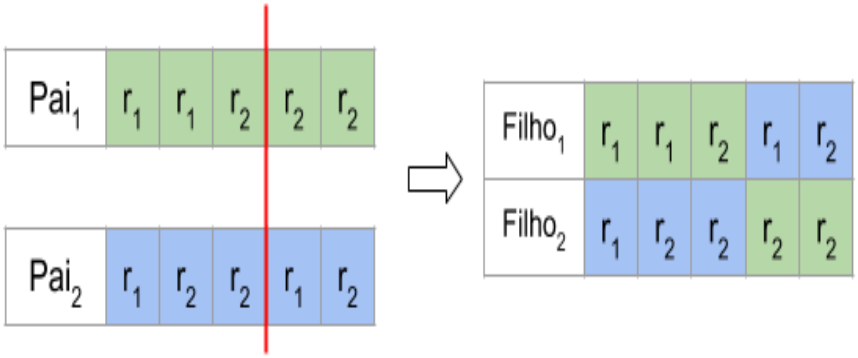


Figura 3.9: Reprodução por um ponto

- Reprodução dois pontos:
  Duas posições aleatórias são escolhidas, depois os cromossomos dos pais são divididos em três pedaços limitados por essas posições e depois cruzados para criação de dois filhos.

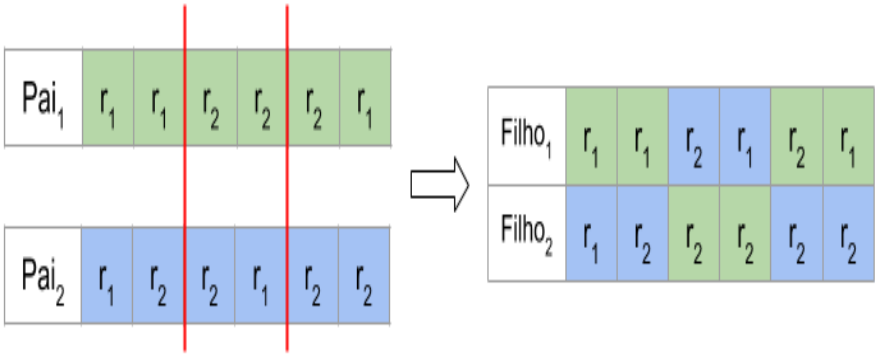


Figura 3.10: Reprodução por dois pontos

- Reprodução uniforme:
  Para cada gene do cromossomo do filho a ser gerado, existe uma probabilidade igual dele assumir o valor do pai ou da mãe.

Mutação

Para cada indivíduo na população existe uma probabilidade de ocorrer uma mutação em um dos seus genes. Essa mutação normalmente é simples, ou seja, um gene do cromossomo do indivíduo é alterado. Em alguns casos acontece uma troca de valores entre dois genes.

Morte

Ao final de uma geração, uma quantidade da população menos apta morre. Essa quantidade pode ser um número fixo de indivíduos ou pode ser baseado na diferença entre os melhores e os piores indivíduos.

Capítulo 4

Soluções Propostas

Neste capítulo serão apresentadas as ideias e os devidos pseudocódigos dos algoritmos propostos para solução do problema.

4.1 Heurísticas

Essa seção busca descrever as duas heurísticas propostas para resolução do problema e o método de busca local aplicado às soluções encontradas por elas para possíveis melhorias.

4.1.1 Heurística 1

Dado um grafo $G = (V, E)$ e as raízes $R = \{r_1, \ldots, r_k\}$ que irão formar a floresta geradora. A ideia dessa heurística é modificar G da seguinte forma:

1. Adicionar um vértice auxiliar v_{aux} à G
2. Para cada raiz $r_i \in R$, adicionar uma aresta $e_{(v_{aux}, r_i)}$ a E com custo zero.

Desta maneira podemos executar um algoritmo que encontre uma árvore geradora mínima a partir do vértice auxiliar[5] no grafo modificado. Tendo a MST do grafo modificado, podemos remover o vértice auxiliar e suas arestas. Isso dará origem a k árvores enraizadas em R.

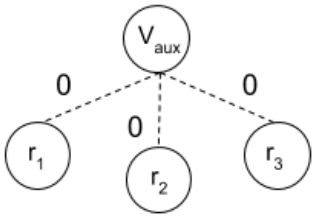


Figura 4.1: Ligações do vértice auxiliar com as raízes

Algoritmo 3: Heurística K-MMSFP

Entrada: G: Grafo = (V, E)

Entrada: R: Conjunto de raízes

Saída: F: Floresta Geradora de k raízes

Início

$V \leftarrow V \cup \{v_{aux}\}$

Para cada $r \in R$ **faça**

$E \leftarrow E \cup \{(v_{aux}, r, 0)\}$

Fim

$F' \leftarrow MST(V, E)$

Para cada $u \in vizinho(v_{aux}, F)$ **faça**

$E_F \leftarrow E_F - \{(v_{aux}, u)\}$

Fim

$V_F \leftarrow V_F - \{v_{aux}\}$

$F \leftarrow compConexas(F')$

Fim

4.1.2 Heurística 2

Dado um grafo $G = (V, E)$ e as raízes $R = \{r_1, \ldots, r_k\}$ que irão formar a floresta geradora, a ideia é manter k árvores com raízes $r_i \in R$ que são subgrafos de G durante cada iteração, para ao final do algoritmo obter k árvores geradoras que formam a floresta geradora F.

Neste algoritmo são mantidas k árvores que podem ser escolhidas para receber um novo vértice. Escolhida a árvore, adicinaremos um vértice seguindo o padrao do algoritmo de *Prim*, ou seja, o vértice que será adicionado a esta árvore será aquele com maior prioridade (menor custo)

Para a escolha da árvore a ser utiliza, dois métodos baseados no custo das árvores foram propostos. A seguir esses métodos serão explicados.

- **Método 1:** Escolher a árvore de menor custo para receber o próximo vértice.

  Para exemplificar, suponha que existam duas raízes, logo, o algoritmo irá manter duas árvores T_1 e T_2. Suponha agora que os custos das árvores são respectivamente 100 e 90. Com isso, utilizando o método acima, a árvore escolhida para a próxima iteração seria a T_2

- **Método 2:** Escolher a árvore que na próxima iteração aumente menos a função objetivo.

  Para exemplificar, suponha o mesmo exemplo do primeiro método onde a função objetivo tem valor 100. Além dos custos de T_1 e T_2, suponha que os valores de maior prioridade (menor custo) do próximo vértice v que podem ser

adicionados sejam respectivamente 5 e 20. A função objetivo passaria a ser 105 em caso de T_1 e 110 em caso de T_2. Logo a árvore T_1 será escolhida.

Algoritmo 4: Heurística Melhorada K-MMSFP

Entrada: G: Grafo
Entrada: R: Conjunto de raízes
Saída: F: Floresta Geradora de $|R|$ raízes
Início
  $Usado[v] \leftarrow Falso \quad \forall v \in Vertices(G)$
  $Subgrafos[r] \leftarrow Lista() \quad \forall r \in R$
  $Custo[r][v] \leftarrow Infinito \quad \forall r \in R \quad \forall v \in Vertices(G)$
  $Prioridade[r] \leftarrow MinHeap() \quad \forall r \in R$
  Para cada $r \in R$ **faça**
    $Usados[r] \leftarrow Verdadeiro$
    Para cada $u \in Vizinhos(G, r)$ **faça**
      $Custo[r][u] \leftarrow c_{(r,u)}$
      $Adiciona(Prioridade[r], (c_{(r,u)}, (r, u)))$
    Fim
  Fim
  Enquanto *tenha vértice não usado* **faça**
    $ArvoreAtual \leftarrow ProximaArvore(Subgrafos)$
    $c, (u, v) \leftarrow Remove(Prioridade[ArvoreAtual])$
    se *Usado[v]=Falso* **então**
      $Subgrafos[ArvoreAtual] \leftarrow (u, v)$
      $Usado[v] \leftarrow Verdadeiro$
      Para cada $u \in Vizinhos(G, v)$ **faça**
        se $Usado[u] = Falso \ e \ Custo[ArvoreAtual][u] > e_{(v,u)}$ **então**
          $Custo[ArvoreAtual][u] \leftarrow c_{(v,u)}$
          $Adiciona(Prioridade[ArvoreAtual], (c_{(v,u)}, (v, u)))$
        fim
      Fim
    fim
  Fim
  $F \leftarrow Subgrafos$
Fim

O pseudocódigo acima utiliza a raiz de uma árvore T para referenciá-la. A função $Remove(A)$ remove e retorna a tupla (custo, aresta) de maior prioridade em A. Além disso a função $ProximaArvore(S)$ assume um dos métodos de seleção descritos acima.

4.1.3 Busca Local

Busca local é um método de otimização que dada uma solução inicial e uma vizinhança, verifica se existe de soluções melhores.
Para nossa busca local, a ideia da busca local é verificar se existe alguma aresta que ligue uma folha de uma árvore T, que pertence a floresta geradora F, a algum outro vértice de outra árvore $T' \in F$ que reduza a função objetivo.

Algoritmo 5: Busca Local

Entrada: G: Grafo = (V, E)
Entrada: F: Floresta Geradora inicial
Saída: F': Floresta Geradora modificada
Início
  $melhorSolucao \leftarrow Verdadeiro$
  Enquanto $melhorSolucao = Verdadeiro$ **faça**
    $melhorSolucao \leftarrow Falso$
    Para cada $v \in Folhas(T_{max})$ **faça**
      Para cada $u \in Vizinhos(G, v, T'), \quad \forall T' \neq T_{max} \in F$ **faça**
        se $W(T') + c_{(v,u)} < W(T_{max})$ **então**
          $melhorSolucao \leftarrow Verdadeiro$
          $Adiciona(T', (v, u))$
          $Remove(T_{max}, v)$
        fim
      Fim
    Fim
  Fim
Fim

No pseudocódigo acima, T_{max} é a árvore de maior custo entre as árvores de F.

4.2 Meta-heurísticas

Essa seção busca descrever como foi modelado o problema de forma que permitisse utilizar meta-heurísticas conhecidas como *Simulated Annealing* e Algoritmo Genético para resolvê-lo, além disso, mostrar decisões de implementação e variações nos algoritmos originais.

4.2.1 *Simulated Annealing*

Abaixo será descrito como é modelado os estados e as transições da Cadeia de Markov que será utilizada para execução do algoritmo. E também as diferentes variações feitas nos métodos de transição de estados.

Estados

Dado o grafo $G = (V, E)$ e o conjunto de raízes $R = \{r_i, \ldots, r_k\}$. Um estado S_i foi definido como sendo um vetor de tamanho $|V|$ onde cada índice i do vetor representa vértice v_i e seu valor representa a qual raiz, e por consequência a qual árvore, v_i está associado.

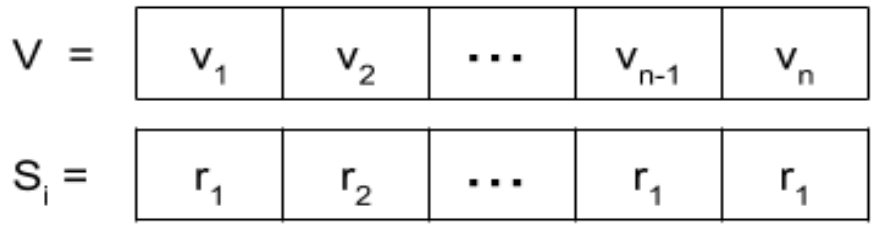


Figura 4.2: Representação de um possível estado para $R = \{r_1, r_2\}$

Os índices de todos os vértices, com exceção aos que são raízes, podem assumir qualquer valor $\in R$. Os vértices raízes, entretanto, só podem assumir seus próprios valores.

Função Objetivo

A função objetivo deste problema é definida como o maior custo entre os custos das árvores que compõem um estado. Para calcular esse valor, é gerado k subgrafos de G a partir de um estado. Os subgrafos são gerados de modo que, vértices que estejam associados a uma mesma raiz pertençam ao mesmo subgrafo de G.

Obtidos os subgrafos, pode-se então executar um algoritmo para encontrar o custo da árvore geradora mínima de cada subgrafo, e com isso obter a função objetivo.

Restrições para o cálculo da Função Objetivo

Como a representação só leva em conta os vértices, pode acontecer de um subgrafo ser desconexo caso o grafo original G não seja completo. Para evitar esse tipo de situação, o grafo G é alterado antes de inicializar o algoritmo, de forma que as arestas que faltam para torná-lo completo são adicionadas com um peso igual ao somatório de todas as arestas originais do grafo mais uma unidade. Deste modo, evita-se que o subgrafo seja desconexo.

Claramente essa adição de arestas em G leva o algoritmo a ter uma perda de desempenho na calcular a função objetivo, porém, é um método que penaliza a solução inviável de uma maneira tão forte, que a probabilidade dela se tornar um estado

escolhido é muito baixa, e se ela for um estado, o algoritmo facilmente encontrará uma solução melhor.

Transições entre estados

As transições entre estados podem ocorrer de três formas que serão descritas abaixo.

1. **Transição 1:** Dado um estado S_i, um vértice $v \in V - R$ é escolhido com probabilidade uniforme dentre todos os outros. Após a escolha do vértice, é escolhida uma raiz $r \in R - S_i[v]$, para qual o vértice v será associado, também com probabilidade uniforme. Com isso, o estado atual S_i é alterado de forma que $S_i[v] \leftarrow r$, o transformando assim em um novo estado S_j

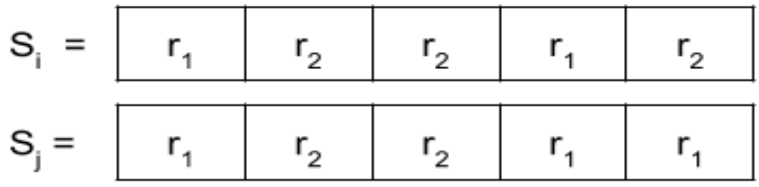


Figura 4.3: Transição 1 - Diferença entre estados

2. **Transição 2:** Similar a transição 1, porém, a ideia é escolher mais frequentemente vértices que pertençam a árvore de maior custo, e associá-lo mais frequentemente com árvores de menor custo.

3. **Transição 3:** Nessa transição a escolha do vértice e da raiz também é similar à transição 1, entretanto, além de associar o vértice escolhido à raiz escolhida, toda subárvore dele também é associada.

   Por exemplo, suponha que a árvore T_{r_2} seja a semelhante à figura abaixo:

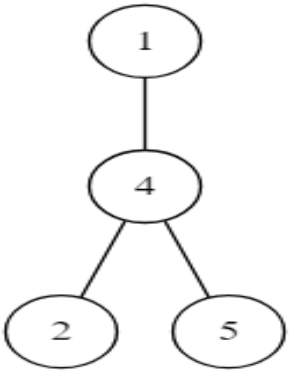


Figura 4.4: Transição 3 - Exemplo subárvore

Caso o vértice 4 seja selecionado, o estado S_i irá ser modificado para S_j como mostrado na figura abaixo.

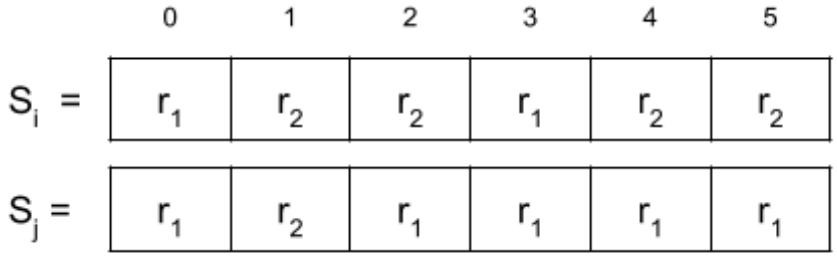


Figura 4.5: Transição 3 - Diferença entre estados

Função de Resfriamento

A função de resfriamento escolhida, foi uma função onde o decaimento da temperatura é linear em relação ao número de iterações.

$$T = T_0 - \beta t \tag{4.1}$$

Estado Inicial

O estado inicial do algoritmo é dado pela solução encontrada pela heurística 2.

4.3 Algoritmo Genético

Esta seção mostra como foi modelado o problema para ser aplicado o algoritmo genético. E também descreve como foi implementada cada etapa descrita na seção 3.5.

4.3.1 Indivíduos e Aptidão

As definições de indivíduos e aptidão são equivalentes às definições de estado e função objetivo no *Simulated Annealing*, este último sujeito às mesmas restrições para o cálculo da aptidão.

4.3.2 Geração Inicial

A geração inicial da população é aleatória, cada cromossomo de um indivíduo pode assumir qualquer valor dentre os valores possíveis de R com probabilidade uniforme, tendo apenas que garantir que as raízes pertençam a elas mesmas, ou seja, $i[r] = r \quad \forall r \in R, \forall i \in$ População.

4.3.3 Seleção dos pais

A cada geração, uma porcentagem, passada como parâmetro, de pares é selecionada para realizar o cruzamento. Dado essa porcentagem, calcula-se o número de pares N_p. Em seguida, são selecionados $2 \times N_p$ indivíduos dando probabilidade maior para escolher indivíduos mais aptos.

4.3.4 Reprodução

Para cada par de indivíduos selecionados é feita uma reprodução uniforme, gerando assim N_p novos indivíduos na população.

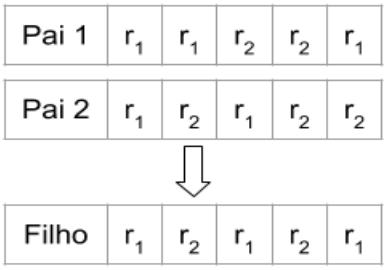


Figura 4.6: Reprodução uniforme entre pares

4.3.5 Mutação

Com a probabilidade de mutação, passada como parâmetro, um indivíduo da população tem um par de genes trocados de posição.

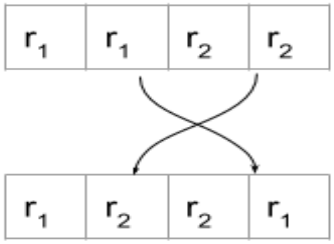


Figura 4.7: Mutação por troca de genes

4.3.6 Morte

Após as etapas anteriores, é recalculada a aptidão dos indivíduos. Após esse cálculo, os N_p indivíduos com menor aptidão são mortos, mantendo a população do mesmo tamanho da população inicial.

Capítulo 5

Avaliação dos Resultados

Serão apresentados neste capítulo as instâncias utilizadas na realização dos experimentos com os algoritmos propostos, as métricas de avaliação, como foram realizados os experimentos e seus devidos resultados.

Parâmetros

Para as avaliações foram utilizados os seguintes parâmetros:

- *Simulatted Annealing*
  - **Temperatura inicial (t_0):** 10^4
  - **Constante de resfriamento (β):** 0.99
  - **Estado inicial:** Solução encontrada pela heurística 2.
- Algoritmo Genético
  - **Critério de parada:** Finalizar após 500 gerações
  - **População inicial:** 100 indivíduos
  - **Taxa de cruzamento:** 30%
  - **Taxa de mutação:** 5%

5.1 Instâncias

O conjunto de instâncias utilizadas para a avaliação dos algoritmos são um subconjunto das utilizadas em DA CUNHA *et al.* [1]. Este conjunto contêm 144 instâncias, divididas de forma igual em quatro tipos:

1. Grafos conexos com distâncias euclidianas.
2. Grafos conexos bipartidos com distâncias euclidianas.

3. Grafos conexos com distâncias randômicas.

4. Grafos conexos bipartidos com distâncias randômicas.

Para cada subconjunto acima, existem três grafos com (n, d) $\forall n \in \{50, 80, 100\}$, $\forall d \in \{100\%, 75\%, 50\%, 25\%\}$ onde n é o número de vértices e d é a densidade de arestas. Além das instâncias, foi possível obter os valores das soluções para os pares de raízes $(1, r)$ $\forall r \in \{2 \dots |V| - 1\}$ para os grafos de $n = \{50, 80\}$.

Além das instâncias acima, foi utilizada também a descrita em TAKAHASHI e KATAOKA [2], as raízes também foram definidas conforme o resultados existentes, de maneira a facilitar comparações.

5.2 Comparações

Com o objetivo de avaliar o comportamento dos algoritmos propostos, todos eles foram executados uma vez para cada instâncias. De maneira a dar uma visão geral dos desempenhos dos algoritmos, foi utilizado para gerar as tabelas abaixo a métrica *Gap*, que é a discrepância relativa entre o ótimo e a solução encontrada pelo algoritmo.

$$Gap = \frac{Valor_{\text{Heurística}} - Valor_{\text{Ótimo}}}{Valor_{\text{Ótimo}}} \tag{5.1}$$

Nas tabelas a seguir, além do Gap médio e Tempo médio, é apresentado o percentual de vezes que o Gap foi menor que uma porcentagem estabelecida pela tabela. As siglas que estão presentes na coluna 'Algoritmo' tem os seguintes significados:

- S.A 1 - Simulated Annealing Método de transição 1
- S.A 1 - Simulated Annealing Método de transição 2
- S.A 1 - Simulated Annealing Método de transição 3
- A.G - Algoritmo Genético

Algoritmo	Tempo Médio* (s)	Gap Médio(%)	Gap < #(%)			
			1%	5%	10%	25%
Heurística 1	0.004	71.14	0.67	1.48	5.65	13.26
Heurística 2^1	0.003	7.74	5.77	35.30	76.50	97.94
Heurística 2^2	0.003	6.76	7.42	45.12	81.47	99.60
S.A. 1	6.48	3.39	15.04	78.21	97.82	100
S.A. 2	9.13	3.29	15.94	79.41	98.39	100
S.A. 3	6.34	0.91	59.45	99.98	100	100
A.G.	7.48	27.89	0	0	0.35	45.65
DA Cunha[1] BnB	4.01	0				

Tabela 5.1: Primeiros resultados - Instâncias DA CUNHA *et al.* [1]
* Média do tempo que o algoritmo levou para encontrar a melhor solução.

Algoritmo	Tempo Médio* (s)	Gap Médio(%)	Gap < #(%)			
			1%	5%	10%	25%
Heurística 1	0,0006	14,86	0	0	70	90
Heurística 2^1	0,0008	8,37	60	60	60	100
Heurística 2^2	0,0008	16,98	10	10	10	100
S.A. 1	3,62	3,90	40	50	90	100
S.A. 2	4,73	1,51	80	80	100	100
S.A. 3	1,82	0.0	100	100	100	100
A.G.	4.53	5.1	30	50	100	100
Yamada[2] BnB	21.6	4.7				

Tabela 5.2: Primeiros resultados - Instâncias TAKAHASHI e KATAOKA [2]
* Média do tempo que o algoritmo levou para encontrar a melhor solução.

Nas tabelas 5.2 acima, podemos perceber que as heurísticas, como esperado, encontram soluções com tempo reduzido de execução, entretanto, com alta discrepância em relação ao ótimo. O algoritmo genético também gera soluções com discrepância considerável utilizando mais tempo de processamento em comparação com as outras meta-heurísticas propostas.

Busca Local	Tempo Médio* (s)	Gap Médio(%)	Gap < #(%)			
			1%	5%	10%	25%
Heurística 1	0.006	66.21	0.67	2.38	6.33	13.64
Heurística 2^1	0.006	7.45	6.49	37.14	79.69	98.14
Heurística 2^2	0.007	6.38	8.27	47.17	84.16	99.18

Tabela 5.3: Resultados das buscas locais aplicadas às Heurísticas

Os valores expostos na tabela 5.3 acima ainda estão distantes dos valores atingidos pelas meta-heurísticas propostas, porém esses resultados nos mostram que a busca local acrescenta uma pequena melhoria, além disso, as heurísticas 2 conseguem encontrar soluções de forma a obter uma discrepância relativa menor que 10% em relação ao ótimo em aproximadamente 80% das instâncias. Entretanto, mesmo com os tempos médios de processamento para encontrar as soluções ainda serem baixos em comparação as meta-heurísticas, eles praticamente dobraram em relação as heurísticas sem a busca local.

5.2.1 Avaliação das Meta-heurísticas

O resultado de apenas uma execução de uma meta-heurística não tem muito significado no contexto de eficiência, pois encontrar uma solução ótima pode ser um caso raro para o algoritmo e por sorte pode ter acontecido exatamente nessa única execução. Tendo isto em vista, para verificar o desempenho dos algoritmos, foram escolhidas as piores entradas (instâncias e raízes), em termos de solução ótima, encontrada para serem executadas cinquenta vezes. E com isso, calcular métricas como máximo, mínimo, média e desvio padrão da função objetivo, de forma que tenhamos uma melhor noção de como o algoritmo se comporta.

Além do Algoritmo Genético, foram escolhidas as duas melhores variações do *Simulated Annealing* para serem avaliadas.

A partir disso foram geradas as distribuições e as tabelas abaixo, que apresentam resultados para os algoritmos selecionados que foram executados nas condições ditas acima. Serão mostradas as distribuições de melhor caso, pior caso e para todas as instâncias.

Distribuição empírica

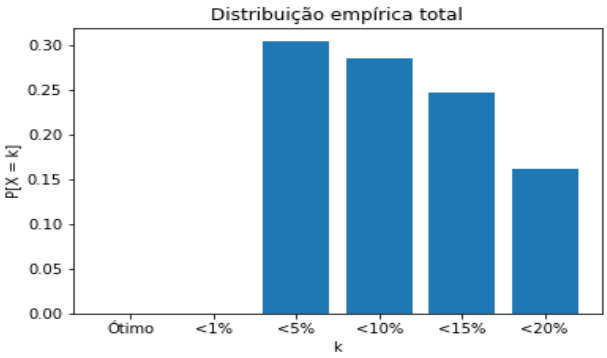


Figura 5.1: Distribuição Empírica Total - *Simulated Annealing* - Método 2

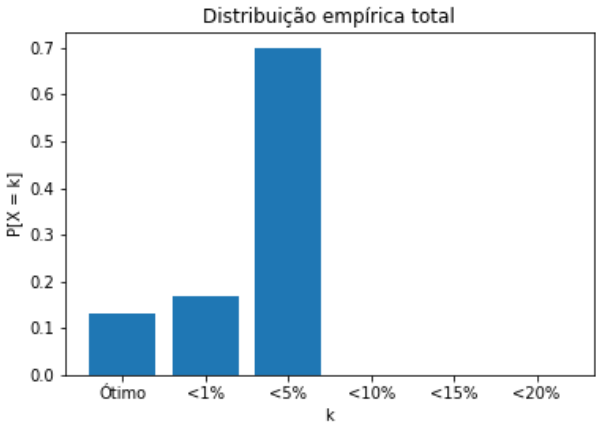


Figura 5.2: Distribuição Empírica Total - *Simulated Annealing* - Método 3

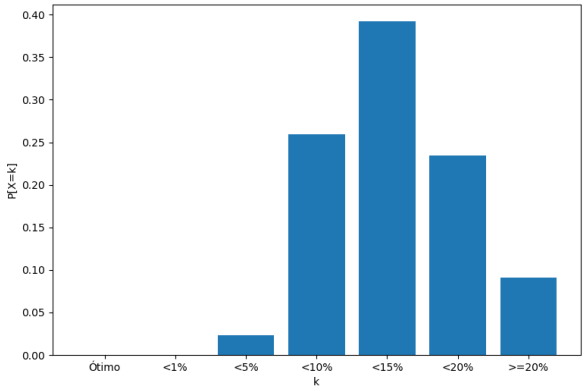


Figura 5.3: Distribuição Empírica Total - Algoritmo Genético

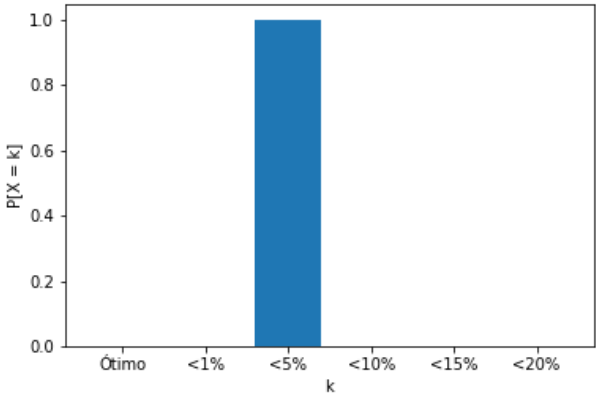


Figura 5.4: Distribuição Empírica - *S.A.*. - Método 2 - Melhor instância euc-80-50

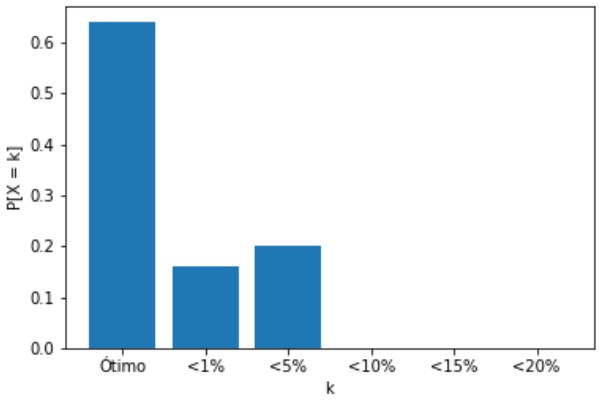


Figura 5.5: Distribuição Empírica - *S.A.*. - Método 3 - Melhor instância euc-50-100

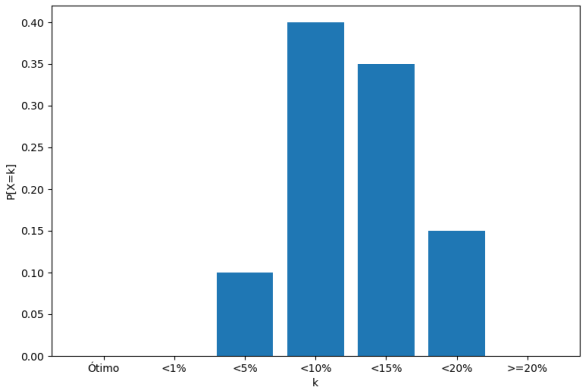


Figura 5.6: Distribuição Empírica - Algoritmo Genético - Melhor instância ran-bi-50-100

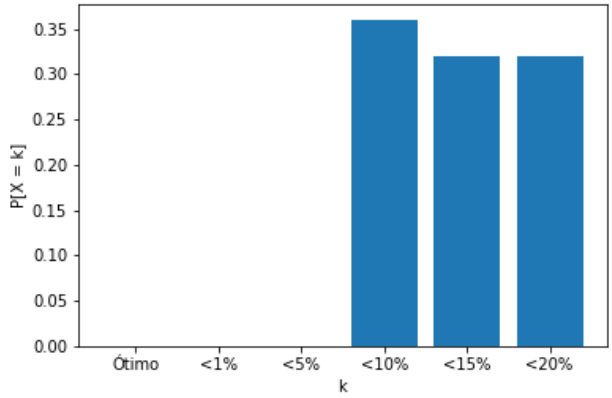


Figura 5.7: Distribuição Empírica - *S.A.*. - Método 2 - Pior instância ran-50-75

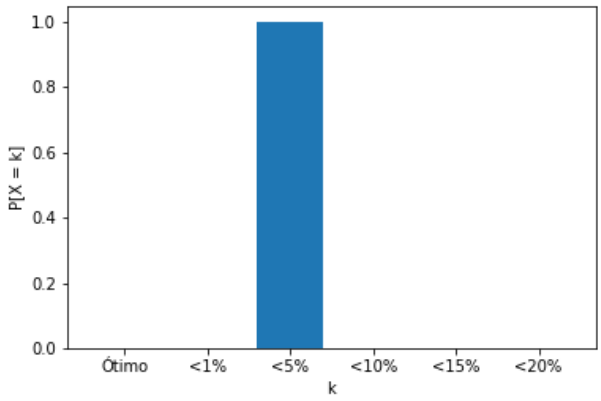


Figura 5.8: Distribuição Empírica - *S.A.*. - Método 3 - Pior instância euc-bi-80-100

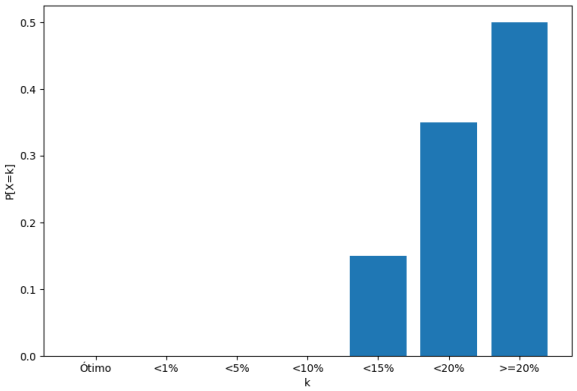


Figura 5.9: Distribuição Empírica - Algoritmo Genético - Pior instância ran-80-100

Podemos observar que os valores obtidos na distribuição empírica total se aproximam com os da tabela 5.2. Para instâncias euclidianas os algoritmos se mostraram mais eficientes. O *Simulated Annealing* utilizando o método de transição 3 se mostrou superior a todos, já que foi o único entre eles a encontrar resultados ótimos. Em certas instâncias euclidianas ele obteve o ótimo em mais de 50% das execuções, apesar disso, no geral, para todas as instâncias, esse valor cai para aproximadamente 15% das vezes. Por sua vez o algoritmo genético, para os parâmetros escolhidos, apresenta uma convergência prematura, dado que seus valores raramente ficam a menos de 5% do ótimo.

Tabelas de resultados

As tabelas abaixo contêm os dados completos das 50 iterações.

Instâncias	r_1	r_2	Ótimo	$T_{Médio}(s)$	min	max	Média	Desvio Padrão
euc-50-100	1	19	2569	9.349	2664	2774	2771.38	15.439
euc-50-25	1	41	4559	7.715	4667	4929	4891.78	61.036
euc-50-50	1	5	3112	8.375	3231	3733	3627.68	142.477
euc-50-75	1	25	2870	9.048	3012	3134	3114.54	31.620
euc-80-100	1	2	3054	17.747	3182	3196	3195.72	1.960
euc-80-25	1	54	5222	13.148	5371	5484	5471.78	28.111
euc-80-50	1	22	4151	13.964	4336	4336	4336.00	0.000
euc-80-75	1	67	3275	16.143	3475	3483	3482.68	1.568
euc-bi-50-100	1	24	2983	7.245	3238	3282	3281.12	6.160
euc-bi-50-25	1	22	8732	6.162	9280	9628	9524.66	105.520
euc-bi-50-50	1	8	3983	6.772	4144	4324	4299.62	37.863
euc-bi-50-75	1	41	4033	6.988	4112	4513	4485.00	75.468
euc-bi-80-100	1	59	3896	13.707	4052	4083	4082.38	4.340
euc-bi-80-25	1	15	8943	11.936	9563	9707	9694.60	36.529
euc-bi-80-50	1	54	5873	12.889	5984	6055	6049.10	16.017
euc-bi-80-75	1	75	4368	13.165	4548	4559	4558.78	1.540
ran-50-100	1	33	516	6.917	547	547	547.00	0.000
ran-50-25	1	31	2220	6.624	2321	2321	2321.00	0.000
ran-50-50	1	30	1049	6.889	1100	1215	1206.14	18.941
ran-50-75	1	19	715	7.023	773	796	795.06	3.906
ran-80-100	1	26	508	13.210	522	530	529.54	1.486
ran-80-25	1	65	2197	12.480	2379	2427	2422.70	11.510
ran-80-50	1	29	1186	13.198	1241	1244	1243.94	0.420
ran-80-75	1	67	696	13.161	735	751	750.08	2.599
ran-bi-50-100	1	10	1104	6.870	1121	1176	1173.30	11.234
ran-bi-50-25	1	4	1257	6.198	1362	1676	1537.50	78.975
ran-bi-50-50	1	8	1277	6.607	1329	1354	1352.66	4.479
ran-bi-50-75	1	10	1480	6.870	1544	1708	1648.52	53.643
ran-bi-80-100	1	29	997	12.885	1034	1071	1066.98	8.627
ran-bi-80-25	1	79	1092	11.883	1122	1142	1140.48	4.187
ran-bi-80-50	1	10	1174	12.528	1195	1236	1231.10	10.374
ran-bi-80-75	1	41	1280	12.715	1367	1394	1391.64	6.878
							Gap Médio	7.94%

Tabela 5.4: Estatísticas para o *Simulated Annealing* - Método de transição 2

Instâncias	r_1	r_2	Ótimo	$T_{Médio}(s)$	min	max	Média	Desvio padrão
euc-50-100	1	19	2569	11.883	2569	2653	2582.78	23.137
euc-50-25	1	41	4559	10.537	4559	4692	4623.92	30.222
euc-50-50	1	5	3112	12.030	3121	3227	3180.50	17.831
euc-50-75	1	25	2870	11.148	2870	2963	2910.84	33.616
euc-80-100	1	2	3054	31.666	3054	3196	3125.22	37.197
euc-80-25	1	54	5222	25.695	5249	5369	5315.04	28.636
euc-80-50	1	22	4151	27.164	4181	4301	4244.36	28.810
euc-80-75	1	67	3275	26.874	3275	3398	3319.74	41.455
euc-bi-50-100	1	24	2983	11.640	2983	3129	3043.30	32.817
euc-bi-50-25	1	22	8732	8.559	8732	8886	8834.36	46.702
euc-bi-50-50	1	8	3983	10.125	3983	4080	4017.98	38.577
euc-bi-50-75	1	41	4033	9.923	4033	4094	4059.04	18.116
euc-bi-80-100	1	59	3896	26.835	3896	4003	3951.90	30.829
euc-bi-80-25	1	15	8943	21.416	8968	9273	9135.68	69.143
euc-bi-80-50	1	54	5873	24.213	5873	6052	5978.22	39.829
euc-bi-80-75	1	75	4368	26.059	4424	4537	4480.64	31.635
ran-50-100	1	33	516	9.869	516	533	523.58	3.790
ran-50-25	1	31	2220	8.780	2220	2310	2244.82	29.978
ran-50-50	1	30	1049	8.366	1049	1099	1069.80	15.106
ran-50-75	1	19	715	8.486	715	741	731.70	6.413
ran-80-100	1	26	508	18.910	508	519	514.06	3.597
ran-80-25	1	65	2197	17.550	2197	2271	2208.34	21.335
ran-80-50	1	29	1186	18.331	1190	1220	1201.32	8.441
ran-80-75	1	67	696	18.463	702	721	710.60	4.508
ran-bi-50-100	1	10	1104	7.195	1104	1126	1113.40	4.142
ran-bi-50-25	1	4	1257	6.353	1257	1286	1265.40	11.200
ran-bi-50-50	1	8	1277	7.305	1290	1321	1306.86	7.242
ran-bi-50-75	1	10	1480	7.353	1480	1513	1495.90	7.998
ran-bi-80-100	1	29	997	19.472	997	1041	1020.20	8.431
ran-bi-80-25	1	79	1092	16.523	1092	1124	1101.44	6.989
ran-bi-80-50	1	10	1174	18.151	1180	1220	1201.78	7.801
ran-bi-80-75	1	41	1280	20.110	1280	1326	1298.76	9.526
							Gap Médio	1.56%

Tabela 5.5: Estatísticas para o *Simulated Annealing* - Método de transição 3

Instâncias	r_1	r_2	Ótimo	$T_{Médio}(s)$	min	max	Média	Desvio padrão
euc-50-100	1	19	2569	17.241	2760	3025	2836.4	71.312
euc-50-25	1	41	4559	11.745	4722	5130	4927.7	120.803
euc-50-50	1	5	3112	13.415	3112	3613	3406.7	146.272
euc-50-75	1	25	2870	15.869	3065	3359	3195.7	83.612
euc-80-100	1	2	3054	30.560	3414	3633	3499.0	74.757
euc-80-25	1	54	5222	23.898	5693	6064	5837.5	120.324
euc-80-50	1	22	4151	24.741	4755	4967	4836.9	57.429
euc-80-75	1	67	3275	29.184	3631	4165	3888.3	147.490
euc-bi-50-100	1	24	2983	12.888	3200	3384	3284.7	62.801
euc-bi-50-25	1	22	8732	9.160	9066	9655	9355.2	162.318
euc-bi-50-50	1	8	3983	11.962	4180	4518	4361.7	98.552
euc-bi-50-75	1	41	4033	12.740	4391	4762	4513.2	108.441
euc-bi-80-100	1	59	3896	24.126	4259	4649	4437.3	133.919
euc-bi-80-25	1	15	8943	16.946	9384	10141	9838.5	211.521
euc-bi-80-50	1	54	5873	22.867	6243	6703	6450.0	159.378
euc-bi-80-75	1	75	4368	23.723	4656	5282	4980.9	165.606
ran-50-100	1	33	516	11.598	540	663	588.1	31.782
ran-50-25	1	31	2220	11.409	2251	2686	2466.4	128.198
ran-50-50	1	30	1049	11.401	1070	1341	1202.0	81.855
ran-50-75	1	19	715	12.384	786	844	811.1	16.580
ran-80-100	1	26	508	23.672	602	659	632.0	20.722
ran-80-25	1	65	2197	22.038	2457	2714	2597.0	82.140
ran-80-50	1	29	1186	24.061	1331	1447	1377.2	31.899
ran-80-75	1	67	696	24.411	798	893	834.8	27.455
ran-bi-50-100	1	10	1104	12.350	1161	1335	1216.5	51.210
ran-bi-50-25	1	4	1257	9.183	1334	1572	1452.3	63.607
ran-bi-50-50	1	8	1277	10.942	1392	1485	1435.8	26.389
ran-bi-50-75	1	10	1480	12.726	1527	1747	1643.3	63.859
ran-bi-80-100	1	29	997	24.073	1106	1266	1183.8	43.609
ran-bi-80-25	1	79	1092	16.747	1217	1433	1315.0	66.699
ran-bi-80-50	1	10	1174	22.904	1337	1492	1398.7	43.580
ran-bi-80-75	1	41	1280	21.917	1411	1655	1538.7	81.095
							Gap Médio	13.96%

Tabela 5.6: Estatísticas para o Algoritmo Genético

Analisando as tabelas 5.4, 5.5, 5.6 acima, podemos perceber que o Gap médio aumenta em relação a primeira tabela 5.2 apresentada anteriormente, o que é aceitável já que estamos tratando das instâncias que obtiveram os piores resultados na primeira execução. O algoritmo genético obteve uma melhora significativa, foi o único a conseguir reduzir o Gap médio em relação ao inicial.

Algumas referências, como por exemplo YAMADA *et al.* [3] e GEN e ZHOU [11], propõem soluções similares a algumas propostas por este trabalho, entretanto, não possuem resultados computacionais explícitos. Como suas propostas não foram reproduzidas por esse trabalho, não foi possível realizar comparações.

Capítulo 6

Conclusão e Trabalhos Futuros

Neste capítulo serão feitas a conclusão do trabalho e apresentados possíveis trabalhos futuros.

6.1 Conclusão

Neste trabalho foram vistas algumas abordagens possíveis para resolução do problema *K-MMSFP* apresentado. As soluções se mostraram eficientes de diferentes maneiras, as heurísticas em questão de tempo e as meta-heurísticas em questão de solução, dando um destaque para o *Simulated Annealing* que obteve excelentes resultados.

O modo de como é recalculado o custo da solução e a necessidade de ter que trabalhar com o grafos completo para garantir que todas as transições sejam válidas teve um grande impacto no tempo durante a execução do algoritmo, por isso, ele peca um pouco em questão de velocidade.

O algoritmo genético não se mostrou muito eficiente, entretanto, é um possível método a ser utilizado para resolução de instâncias desse problema onde os outros comentados não apresentam uma boa solução. Além disso, é possível que haja maneiras melhores de realizar as suas etapas de forma a melhorar seu desempenho.

O principal objetivo, que era propor heurísticas e meta-heurísticas que fossem eficientes para resolução do problema *K-MMSFP*, foi alcançado visto que os algoritmos propostos demonstraram bons resultados, dentro das limitações que ocorreram devido às escolhas de implementação. O objetivo secundário, que era analisar casos para mais raízes e instâncias, foi parcialmente realizado, os resultados encontrados pelos algoritmos propostos utilizando mais raízes para algumas das instâncias utilizadas nesse trabalho estão disponíveis no apêndice.

6.2 Trabalhos Futuros

Existem diversos caminhos para evolução deste trabalho, talvez o mais imediado seja remodelar o método que é utilizado para o cálculo da função objetivo, de forma que não seja preciso recalcular todo custo das árvores presente numa determinada solução, diminuindo assim o tempo de processamento tanto no *Simulated Annealing* quanto no algoritmo genético. Testar outras formas de transição entre estados para o *Simulated Annealing* ou formas de reprodução e mutação para o algoritmo genético também podem ser excelentes ideias para trabalhos futuros, visto que essas etapas são essenciais para os algoritmos.

Outro possível trabalho seria analisar o desempenho das meta-heurísticas utilizando arestas em vez de vértices para modelar as soluções.

Referências Bibliográficas

[1] DA CUNHA, A. S., SIMONETTI, L., LUCENA, A. "Optimality cuts and a branch-and-cut algorithm for the K-rooted mini-max spanning forest problem", *European Journal of Operational Research*, v. 246, n. 2, pp. 392–399, 2015.

[2] TAKAHASHI, T. Y. H., KATAOKA, S. "A branch-and-bound algorithm for the mini-max spanning forest problem", *European Journal of Operational Research*, v. 101, pp. 93–103, 1997.

[3] YAMADA, T., TAKAHASHI, H., KATAOKA, S. "A heuristic algorithm for the mini-max spanning forest problem", *European Journal of Operational Research*, v. 91, n. 3, pp. 565–572, 1996.

[4] MEKKING, M., VOLGENANT, A. "Solving the 2-rooted mini-max spanning forest problem by branch-and-bound", *European Journal of Operational Research*, v. 203, n. 1, pp. 50–58, 2010.

[5] PRIM, R. C. "Shortest connection networks and some generalizations", *Bell system technical journal*, v. 36, n. 6, pp. 1389–1401, 1957.

[6] KRUSKAL, J. B. "On the shortest spanning subtree of a graph and the traveling salesman problem", *Proceedings of the American Mathematical society*, v. 7, n. 1, pp. 48–50, 1956.

[7] KLEINBERG, J., TARDOS, E. *Algorithm design*. Pearson Education India, 2006.

[8] HÄGGSTRÖM, O. *Finite Markov chains and algorithmic applications*, v. 52. Cambridge University Press, 2002.

[9] CHIB, S., GREENBERG, E. "Understanding the metropolis-hastings algorithm", *The american statistician*, v. 49, n. 4, pp. 327–335, 1995.

[10] AARTS, E., KORST, J. "Simulated annealing and Boltzmann machines", 1988.

[11] GEN, M., ZHOU, G. "A genetic algorithm for the mini-max spanning forest problem". In: *Proceedings of the 2nd Annual Conference on Genetic and Evolutionary Computation*, pp. 387–387. Morgan Kaufmann Publishers Inc., 2000.

Apêndice A

Código

Todo código implementado nesse trabalho pode ser encontrado no repositório do *Github* maasouza/K-MMSFP. Entretanto, as funções e classes mais relevantes estão disponíveis abaixo.

A.0.1 Grafo e Heurísticas

code/grafo.py

```
1  import queue as Q
2  import numpy as np
3  import math
4  from util import *
5
6  class Graph:
7      def __init__(self, file = None, numVertex = 0):
8          self.V = numVertex
9          self.E = 0
10         self.maxWeight = 0
11         self.file = file
12         self.edges = {}
13         self.adj_matrix = [[0 for x in range(self.V)] for y in
               range(self.V)]
14         self.adj_list = {}
15         if(file != None):
16             self.read_file(file)
17         self.subtrees = {}
18
19     def read_file(self, file):
20         with open(file,'r') as file:
21             self.V, self.E = map(int,file.readline().split())
22             self.adj_matrix = [[0 for x in range(self.V)] for y in
                   range(self.V)]
23             for i in range(self.E):
```

```
24                  i, j, c = map(int, file.readline().split())
25                  self.addEdge(i-1, j-1, c)
26                  self.addEdge(j-1, i-1, c)
27                  self.maxWeight += c
28
29      def addEdge(self, i, j, c):
30          '''Adiciona ou altera o valor da aresta i,j'''
31          self.edges[(i,j)] = c
32          self.edges[(j,i)] = c
33          self.adj_list.setdefault(i,{})[j] = c
34          self.adj_list.setdefault(j,{})[i] = c
35          self.adj_matrix[i][j] = c
36          self.adj_matrix[j][i] = c
37
38      def removeEdge(self, i, j):
39          self.edges.pop((i,j),None)
40          self.edges.pop((j,i),None)
41          self.adj_list[i].pop(j,None)
42          self.adj_list[j].pop(i,None)
43          self.adj_matrix[i][j] = 0
44          self.adj_matrix[j][i] = 0
45
46      def mst(self, v=0, roots=[], marked = None  ):
47          st_root = v
48          if(len(roots)>1):
49              st_root = roots[0]
50              v = roots[0]
51          tree_cost = 0
52          used = 0
53          if(marked != None):
54              inMST = marked
55              used = sum(marked)
56          else:
57              inMST = [False]*self.V
58          key = {}
59          tree = {}
60          heap = Q.PriorityQueue()
61          for i in range(self.V):
62              tree[i] = -1
63              key[i] = math.inf
64          key[v] = 0
65          heap.put((key[v], v))
66          while used != self.V:
67              h = heap.get()
68              u = h[1]
69              w = h[0]
70              if(inMST[u] == True):
```

```
71                  continue
72              inMST[u] = True
73              used += 1
74              tree_cost+=w
75              for adj in self.adj_list[u].items():
76                  v = adj[0]
77                  w = adj[1]
78                  if (inMST[v] == False) and (key[v]>w):
79                      key[v] = w
80                      heap.put((key[v],v))
81                      tree[v] = u
82
83          spanning_tree = {
84              'root': st_root,
85              'cost': tree_cost,
86              'edges': set()
87              }
88
89          for v in roots:
90              tree[v] = -1
91
92          for edge in tree.items():
93              if(edge[1]==-1):
94                  continue
95              spanning_tree['edges'].add((edge[1],edge[0]))
96          return spanning_tree
97
98      def rooted_trees(self, roots):
99          self.subtrees = {}
100         rootEdges = []
101         for i in range(len(roots)-1):
102             for j in range(i+1,len(roots)):
103                 rootEdges.append((roots[i], roots[j]))
104                 rootEdges.append((roots[j], roots[i]))
105                 self.addEdge(roots[i], roots[j], 0)
106         bottleneck_tree = self.mst(roots = roots)
107
108         for root in roots:
109             self.subtrees[root] = {
110                 'cost': 0,
111                 'edges': set(),
112             }
113             q = Q.Queue()
114             q.put(root)
115             while not q.empty():
116                 v = q.get()
117                 for edge in bottleneck_tree['edges']:
```

```
118                     if(v in edge) and   (edge not in
                            self.subtrees[root]['edges'] ):
119                         self.subtrees[root]['edges'].add(edge)
120                         self.subtrees[root]['cost'] +=
                                self.adj_matrix[edge[0]][edge[1]]
121                         if(v == edge[0]):
122                             q.put(edge[1])
123                         else:
124                             q.put(edge[0])
125         for i in range(len(roots)-1):
126             for j in range(i+1,len(roots)):
127                 self.removeEdge(roots[i], roots[j])
128         return self.subtrees, bottleneck_tree
129
130     def greedy_random_sf(self, roots):
131         self.subtrees = {}
132         heapDict = {}
133         inST = [False]*self.V
134         used = len(roots)
135         key = {}
136         cur_tree = roots[0]
137         for root in roots:
138             inST[root] = True
139             key[root] = [math.inf]*self.V
140             heapDict[root] = Q.PriorityQueue()
141             self.subtrees[root] = {
142                 'cost':0,
143                 'edges': set()
144             }
145             for adj in self.adj_list[root].items():
146                 # item = (cost, (v, parent[v])
147                 key[root][adj[0]] = adj[1]
148                 item = (adj[1],(adj[0],root))
149                 heapDict[root].put(item)
150
151         while used != self.V:
152             item = heapDict[cur_tree].get()
153             c = item[0]
154             v,parent = item[1]
155             if(inST[v]):
156                 continue
157             inST[v] = True
158             used+=1
159             self.subtrees[cur_tree]['cost']+=c
160             self.subtrees[cur_tree]['edges'].add((parent, v))
161             for adj in self.adj_list[v].items():
162                 i =  adj[0]
```

```
163                 c = adj[1]
164                 if (not inST[i]) and (key[cur_tree][i]>c):
165                     key[cur_tree][i] = c
166                     item = (c,(i,v))
167                     heapDict[cur_tree].put(item)
168             for root in roots:
169                 cost = self.subtrees[root]['cost']
170                 if(cost<= self.subtrees[cur_tree]['cost']):
171                     cur_tree = root
172         return self.subtrees
173 
174     def improved_random_sf(self, roots):
175         self.subtrees = {}
176         heapDict = {}
177         inST = [False]*self.V
178         used = len(roots)
179         key = {}
180         cur_tree = roots[0]
181         target = 0
182         for root in roots:
183             inST[root] = True
184             key[root] = [math.inf]*self.V
185             heapDict[root] = Q.PriorityQueue()
186             self.subtrees[root] = {
187                 'cost':0,
188                 'edges': set()
189             }
190             for adj in self.adj_list[root].items():
191                 # item = (cost, (v, root))
192                 key[root][adj[0]] = adj[1]
193                 item = (adj[1],(adj[0],root))
194                 heapDict[root].put(item)
195         #selecting first tree
196         possible_trees = set()
197         for root in roots:
198             if(len(heapDict[root].queue) > 0):
199                 min_vertex = heapDict[root].queue[0]
200                 item = (root, min_vertex[0])
201                 possible_trees.add(item)
202         cur_tree = min_set(possible_trees)[0]
203         possible_trees.clear()
204         while used != self.V:
205 
206             item = heapDict[cur_tree].get()
207             c = item[0]
208             v,parent = item[1]
209 
```

```
210             if(inST[v]):
211                 continue
212             inST[v] = True
213             used+=1
214 
215             self.subtrees[cur_tree]['cost']+=c
216             self.subtrees[cur_tree]['edges'].add((parent, v))
217 
218             for adj in self.adj_list[v].items():
219                 i =  adj[0]
220                 c = adj[1]
221                 if (not inST[i]) and (key[cur_tree][i]>c):
222                     key[cur_tree][i] = c
223                     item = (c,(i,v))
224                     heapDict[cur_tree].put(item)
225 
226             for root in roots:
227                 if(len(heapDict[root].queue) > 0):
228                     min_vertex = heapDict[root].queue[0]
229                     item =
                            (root,self.subtrees[root]['cost']+min_vertex[0])
230                     possible_trees.add(item)
231 
232             cur_tree = min_set(possible_trees)[0]
233             possible_trees.clear()
234 
235         return self.subtrees
236 
237     def get_prob(self, roots):
238         probs = []
239         for i in range(self.V):
240             dist_sum = 0
241             prob = [0]*len(roots)
242             if(i in roots):
243                 prob[roots.index(i)] = 1
244                 probs.append(prob)
245                 continue
246             for idx, r in enumerate(roots):
247                 dist_sum +=  self.adj_list[i].setdefault(r,0)
248                 prob[idx] = self.adj_list[i].setdefault(r,0)
249 
250             for idx in range(len(prob)):
251                 if(prob[idx]!=0):
252                     prob[idx] = (dist_sum -
                            prob[idx])/((len(prob)-1)*dist_sum)
253             probs.append(prob)
254         return probs
```

```
255
256     def mst_from_probs(self, roots, probs):
257         vertex_list = {}
258         spanning_tree = {}
259         for root in roots:
260             spanning_tree[root] = None
261             vertex_list[root] = [True]*self.V
262
263         for idx, prob in enumerate(probs):
264             tree = np.random.choice(roots, p = prob)
265             vertex_list[tree][idx] = False
266
267         for root in roots:
268             spanning_tree[root] = self.mst(root, marked =
                vertex_list[root])
269
270         return spanning_tree
271
272     def make_complete(self):
273         for i in range(self.V-1):
274             for j in range(i+1,self.V):
275                 if((i, j) not in self.edges and (j,i) not in
                    self.edges):
276                     self.addEdge(i, j, self.maxWeight+1)
277
278     def make_uncomplete(self):
279         for i in range(self.V-1):
280             for j in range(i+1,self.V):
281                 if(self.adj_matrix[i][j]==self.maxWeight+1):
282                     self.removeEdge(i, j)
283
284     def __repr__(self):
285         ret = 'Grafo\n\t |V| => '+str(self.V)+'\n\t |E| =>
            '+str(self.E)+'\n Matrix de adjacencia\n'
286         for i in  self.adj_matrix:
287             ret += str(i)+'\n'
288         return ret
```

A.0.2 Simulated Annealing

code/sa.py

```
1 def simulatedAnnealing(self, initialT, epsilon, coolingStrategy, beta):
2         self.restartInstance()
3         def format_e(n):
4             #format large number in scientific notation
```

```
5           a = '%E' % n
6           return a.split('E')[0].rstrip('0').rstrip('.') + 'E' +
                a.split('E')[1]
7       print("Start optimal value:", self.bestSolution.objFunction)
8       name = 'Simulated Annealing T = ' + str(format_e(initialT)) +
            ' b = ' + str(beta) + ' ' + coolingStrategy.__name__
9       temperature = initialT
10      t = 0   # steps
11      start = time.time()
12      found_time = [0]
13      ret = [self.currentSolution.objFunction]
14      while temperature > epsilon:
15          t += 1
16          currentChanged = False
17          newState = State()
18          while len(newState.items) == 0:
19              index = random.randint(0, self.graph.V-1)
20              root = np.random.choice(self.roots)
21              newState = self.newStateFor(index, root).copy()
22
23          if(newState in self.savedStates):
24              newState = self.savedStates[newState].copy()
25          else:
26              newState.update(self.graph)
27              self.savedStates[newState] = newState.copy()
28
29          delta =
                newState.objFunction-self.currentSolution.objFunction
30          if delta < 0:
31              self.currentSolution = newState.copy()
32              currentChanged = True
33          else:
34              if random.random() < self.boltzman(delta, temperature,
                    1, 1):
35                  self.currentSolution = newState.copy()
36                  currentChanged = True
37
38          if currentChanged and
                self.isBetterSolution(self.currentSolution):
39              print("Itt - ",t," - ",
                    self.currentSolution.objFunction)
40
41          ret.append(self.bestSolution.objFunction)
42          found_time.append(time.time() - start)
43          temperature = coolingStrategy(initialT, beta, t, delta,
                temperature)
44      return name, ret, found_time
```

```
45
46
47
48      def simulatedAnnealingSubtree(self, initialT, epsilon,
            coolingStrategy, beta):
49          self.restartInstance()
50          def format_e(n):
51              #format large number in scientific notation
52              a = '%E' % n
53              return a.split('E')[0].rstrip('0').rstrip('.') + 'E' +
                    a.split('E')[1]
54          print("Start optimal value:", self.bestSolution.objFunction)
55          name = 'Simulated Annealing T = ' + str(format_e(initialT)) +
                ' b = ' + str(beta) + ' ' + coolingStrategy.__name__
56          temperature = initialT
57          t = 0   # steps
58          start = time.time()
59          ret = [self.currentSolution.objFunction]
60          found_time = [0]
61          while temperature > epsilon:
62              t += 1
63              currentChanged = False
64              newState = State()
65              while len(newState.items) == 0:
66                  index = random.randint(0, self.graph.V-1)
67                  root = np.random.choice(self.roots)
68                  newState = self.newStateForSubtree(index, root).copy()
69
70              if(newState in self.savedStates):
71                  newState = self.savedStates[newState].copy()
72              else:
73                  newState.update(self.graph)
74                  self.savedStates[newState] = newState.copy()
75              delta =
                    newState.objFunction-self.currentSolution.objFunction
76
77              if delta < 0:
78                  self.currentSolution = newState.copy()
79                  currentChanged = True
80              else:
81                  if random.random() < self.boltzman(delta, temperature,
                        1, 1):
82                      self.currentSolution = newState.copy()
83                      currentChanged = True
84
85              if currentChanged and
                    self.isBetterSolution(self.currentSolution):
```

```
86                  print("Itt - ",t," - ",
                        self.currentSolution.objFunction)
87
88              ret.append(self.bestSolution.objFunction)
89              found_time.append(time.time()-start)
90              if currentChanged and
                    self.isBetterSolution(self.currentSolution):
91                  print("Itt - ",t," - ",
                        self.currentSolution.objFunction)
92              elif(random.random()<0.005):
93                  continue
94              temperature = coolingStrategy(initialT, beta, t, delta,
                    temperature)
95          return name, ret, found_time
```

A.0.3 Algoritmo Genético

code/genetico.py

```
1  import numpy as np
2  class AG:
3
4      def __init__(self, population, generations, bornRate, deathRate,
           mutationRate, grafo, roots):
5          self.roots = roots
6          self.grafo = grafo
7          self.grafo.make_complete()
8          self.generations = generations
9          self.bornRate = bornRate
10         self.deathRate =deathRate
11         # self.crossoverType = crossoverType
12         self.prob = []
13         self.all_cost = 0
14         self.mutationRate = mutationRate
15         self.population = []
16         self.initialPop = population
17         for _ in range(population):
18             person = []
19             for i in range(grafo.V):
20                 person.append(np.random.choice(roots))
21             for root in roots:
22                 person[root] = root
23             self.population.append([person,None,0])
24         self.evaluate()
25
26     def resetInstance(self):
```

```
27          self.population = []
28          for _ in range(self.initialPop):
29              person = []
30              for i in range(self.grafo.V):
31                  person.append(np.random.choice(self.roots))
32              for root in self.roots:
33                  person[root] = root
34              self.population.append([person,None,0])
35          self.evaluate()
36
37      def evaluate(self):
38          self.prob = []
39          self.all_cost = 0
40          for idx, person in enumerate(self.population):
41              cost = []
42              objFunction = 0
43              if(person[2]==0):
44                  for root in self.roots:
45                      used = [0 if root == gene else 1 for gene in
                            person[0]]
46                      cost.append(self.grafo.mst(root, marked =
                            used)['cost'])
47                  objFunction = max(cost)
48              else:
49                  objFunction = person[1]
50              self.population[idx][1] = objFunction
51              self.population[idx][2] = 1
52              self.prob.append(objFunction)
53              self.all_cost += objFunction
54
55          for idx, _ in enumerate(self.prob):
56              self.prob[idx] =  (self.all_cost - self.prob[idx]) / ((
                    len(self.prob)-1)*self.all_cost)
57
58      def crossover(self):
59          number_of_sons = round(self.initialPop*self.bornRate)
60          number_of_deaths = round(self.initialPop*self.deathRate)
61          filhos = []
62          n_pais = len(self.roots)
63          for _ in range(number_of_sons):
64              pais = np.random.choice(len(self.population), 2 ,
                        p=self.prob)
65              pais = [self.population[i] for i in pais]
66              mutate = np.random.random()
67              filho = []
68              if(np.random.random() < self.bornRate):
69                  for i in range(self.grafo.V):
```

```
70                      if(mutate < self.mutationRate and i not in
                            self.roots):
71                          filho.append(np.random.choice(self.roots))
72                      else:
73                          p = self.calc_prob(pais)
74                          idx = np.random.choice(len(pais),p = p)
75                          filho.append(pais[idx][0][i])
76                  filhos.append([filho,None,0])
77
78          deaths = 0
79          mean = self.all_cost/len(self.population)
80          will_die = []
81          for idx, person in enumerate(self.population):
82              if(person[1] > mean):
83                  del self.population[idx]
84                  deaths+=1
85                  if(deaths == number_of_deaths):
86                      break
87          self.population.extend(filhos[:])
88          delta = self.initialPop - len(self.population)
89          for i in range(delta):
90              self.population.append([self.create_random(), None, 0])
91
92      def create_random(self):
93          person = []
94          for i in range(self.grafo.V):
95              person.append(np.random.choice(self.roots))
96          for root in self.roots:
97              person[root] = root
98          return person
99
100     def calc_prob(self, v):
101         cost = 0
102         n = len(v)-1
103         p = []
104         for i in v:
105             cost += i[1]
106             p.append(i[1])
107         for i,_ in enumerate(p):
108             p[i] = (cost - p[i])/(n*cost)
109         return p
110
111
112     def bestSolution(self):
113         return sorted(self.population, key = lambda x:x[1])[0]
114
115     def run(self):
```

```
116         t = 0
117         self.resetInstance()
118         while t<self.generations:
119             t+=1
120             self.crossover()
121             self.evaluate()
122         return self.bestSolution()[1]
```

Apêndice B

Resultados para Instâncias maiores

A tabela B abaixo apresenta soluções para instâncias maiores que não foram utilizadas para comparações, pois possuem soluções ótimas desconhecidas. Estas instâncias estão disponíveis no repositório citado no apêndice A.

Instância	# Vértices	r1	r2	Solução	Tempo (s)
$bb15x15_1$	225	1	21	3281	10.717
$bb15x15_2$	225	1	9	3132	10.041
$bb45x5_1$	225	1	20	3636	9.033
$bb45x5_2$	225	1	6	5668	8.869
euc-100-100-1	100	1	20	3372	4.782
euc-100-25-1	100	1	5	6717	3.131
euc-100-50-1	100	1	12	4618	3.532
euc-100-75-1	100	1	19	3944	4.128
euc-bi-100-100-1	100	1	21	4597	3.804
euc-bi-100-50-1	100	1	19	6633	3.156
euc-bi-100-75-1	100	1	21	5041	3.439
g400-4-01	400	1	9	5767	30.441
g400-4-05	400	1	10	5785	32.679
$le450_15a$	450	1	9	1055	47.121
ran-100-100-1	100	1	10	613	3.990
ran-100-50-1	100	1	11	1082	4.054
ran-100-75-1	100	1	5	809	3.202
ran-bi-100-100-1	100	1	17	1092	3.509
ran-bi-100-50-1	100	1	11	1340	3.909
ran-bi-100-75-1	100	1	2	1132	3.491
steinc15	500	1	9	3034	49.778
steind15	1000	1	15	6310	200.447

Tabela B.1: Soluções para Instâncias maiores

Apêndice C

Resultados para mais raízes

Foram escolhidas 3, 4 e 5 raízes aleatórias para gerar soluções das instâncias utilizadas no trabalho com os algoritmos propostos. As melhores soluções encontradas estão disponíveis nas tabelas abaixo. Estas instâncias também estão disponíveis no repositório citado no apêndice A.

Instância	r1	r2	r3	Solução	Tempo
euc-100-50.dat	6	12	18	3797	5.857610
euc-100-75.dat	2	9	18	2899	8.009215
euc-50-100.dat	3	8	18	1987	3.096931
euc-50-50.dat	4	14	15	2359	3.091413
euc-50-75.dat	4	14	20	1895	3.485354
euc-80-100.dat	6	11	17	2325	6.080927
euc-80-50.dat	2	13	20	2812	3.976978
euc-80-75.dat	1	13	17	2399	4.906374
euc-bi-100-100.dat	4	12	19	3166	5.563767
euc-bi-100-50.dat	4	11	18	4891	5.222727
euc-bi-100-75.dat	4	13	18	3461	5.211197
euc-bi-50-100.dat	1	8	16	2258	1.792185
euc-bi-50-50.dat	1	8	16	2810	1.557565
euc-bi-50-75.dat	6	9	16	3126	1.651675
euc-bi-80-100.dat	4	10	17	3086	3.460178
euc-bi-80-50.dat	3	9	19	4069	3.831787
euc-bi-80-75.dat	6	10	17	3359	3.633622
g400-4-01.dat.dat	4	13	20	3870	40.381317

Tabela C.1: Resultados para 3 raízes

Instância	r1	r2	r3	Solução	Tempo
Instancia-yamada.dat.dat	2	14	16	568	0.596029
ran-100-100.dat	3	14	15	429	4.834906
ran-100-50.dat	6	9	15	843	4.594186
ran-100-75.dat	3	14	17	569	6.827432
ran-50-100.dat	3	8	17	425	1.858517
ran-50-50.dat	6	13	15	880	1.763363
ran-50-75.dat	3	13	16	550	1.790337
ran-80-100.dat	2	9	20	367	3.444800
ran-80-50.dat	4	10	16	961	3.365869
ran-80-75.dat	7	12	19	487	3.334901
ran-bi-100-100.dat	7	8	16	750	4.456857
ran-bi-100-50.dat	4	11	15	912	4.719819
ran-bi-100-75.dat	7	14	18	840	4.358754
ran-bi-50-100.dat	2	13	16	850	1.691267
ran-bi-50-50.dat	7	11	15	871	1.581446
ran-bi-50-75.dat	6	14	17	1132	1.990772
ran-bi-80-100.dat	7	9	16	711	3.504813
ran-bi-80-50.dat	2	13	20	930	4.045240
ran-bi-80-75.dat	2	10	19	870	4.282752

Tabela C.2: Continução tabela C.1

Instância	r1	r2	r3	r4	Solução	Tempo
euc-100-100	1	7	14	18	2153	2.242911
euc-100-50	2	8	13	17	2569	1.735383
euc-100-75	6	10	15	18	2322	3.667583
euc-50-100	5	7	14	21	1470	1.089985
euc-50-50	1	10	13	17	2285	0.654383
euc-50-75	5	11	16	20	1663	0.751298
euc-80-100	6	10	16	17	2037	1.588522
euc-80-50	5	8	13	17	2439	1.253832
euc-80-75	4	9	12	20	1949	1.411682
euc-bi-100-100	2	10	13	20	2580	1.755368
euc-bi-100-50	1	11	14	18	3915	1.588520
euc-bi-100-75	2	11	15	17	2893	1.680434
euc-bi-50-100	1	8	12	18	1927	0.665380
euc-bi-50-50	2	10	15	21	2331	0.733318
euc-bi-50-75	5	10	15	21	2255	0.624420
euc-bi-80-100	2	10	13	20	2531	1.242842
euc-bi-80-50	1	8	16	19	3337	1.169911
euc-bi-80-75	3	7	15	18	2547	1.313774

Tabela C.3: Resultados para 4 raízes

Instância	r1	r2	r3	r4	Solução	Tempo
Instancia-yamada	1	9	14	19	456	0.282736
ran-100-100	1	8	15	20	357	1.896231
ran-100-50	6	7	14	19	553	1.700413
ran-100-75	1	8	14	19	456	1.681430
ran-50-100	3	7	12	18	458	0.669377
ran-50-50	1	7	15	18	714	0.664384
ran-50-75	3	9	14	21	428	0.680366
ran-80-100	3	8	12	18	336	1.380707
ran-80-50	3	7	14	17	775	1.220862
ran-80-75	1	11	16	17	382	1.219864
ran-bi-100-100	5	11	16	21	884	1.846281
ran-bi-100-50	1	7	16	17	668	2.089053
ran-bi-100-75	3	7	15	21	635	1.829287
ran-bi-50-100	2	10	12	19	606	0.660384
ran-bi-50-50	2	9	13	20	1121	0.701347
ran-bi-50-75	3	9	12	17	947	0.688357
ran-bi-80-100	6	7	12	18	703	1.246837
ran-bi-80-50	2	8	16	19	642	1.196887
ran-bi-80-75	2	10	12	18	774	1.225858

Tabela C.4: Continuação da Tabela C.3

Instância	r1	r2	r3	r4	r5	Solução	Tempo
euc-100-50.dat	4	5	10	16	20	2514	3.252967
euc-100-75.dat	2	8	12	16	17	1893	3.341887
euc-50-100.dat	2	5	9	14	17	1400	1.361733
euc-50-50.dat	2	7	10	14	20	1863	1.234849
euc-50-75.dat	2	5	9	15	18	1621	1.416680
euc-80-100.dat	1	6	10	16	19	1518	2.650528
euc-80-50.dat	1	8	11	13	18	2142	2.386774
euc-80-75.dat	1	8	12	13	18	1840	2.736452
euc-bi-100-100.dat	2	6	9	15	17	2327	3.351876
euc-bi-100-50.dat	1	5	10	15	20	2782	2.856342
euc-bi-100-75.dat	3	7	11	14	19	2588	3.184033
euc-bi-50-100.dat	1	6	9	15	17	1780	1.251833
euc-bi-50-50.dat	1	5	10	16	19	1855	1.130944
euc-bi-50-75.dat	3	6	11	14	19	1946	1.189891
euc-bi-80-100.dat	4	6	9	14	17	2096	2.324833
euc-bi-80-50.dat	1	8	12	15	19	2576	2.644536
euc-bi-80-75.dat	4	8	12	14	18	2269	2.389772
g400-4-01.dat.dat	1	5	9	14	20	2338	27.077773

Tabela C.5: Resultados para 5 raízes

Instância	r1	r2	r3	r4	r5	Solução	Tempo
Instancia-yamada.dat.dat	3	8	12	16	19	414	0.562476
ran-100-100.dat	3	6	9	14	20	325	4.248043
ran-100-50.dat	3	6	12	13	17	602	3.383846
ran-100-75.dat	1	7	11	13	18	383	3.713541
ran-50-100.dat	4	7	9	13	18	285	1.319767
ran-50-50.dat	3	8	9	16	17	668	1.485616
ran-50-75.dat	4	6	10	14	17	490	1.281805
ran-80-100.dat	4	7	11	13	19	226	2.487687
ran-80-50.dat	1	5	11	16	20	669	2.287866
ran-80-75.dat	3	6	10	13	18	334	2.313843
ran-bi-100-100.dat	3	7	10	14	20	526	3.007195
ran-bi-100-50.dat	4	5	12	15	17	572	2.918283
ran-bi-100-75.dat	3	5	9	14	18	561	4.743580
ran-bi-50-100.dat	4	5	10	14	20	517	1.625485
ran-bi-50-50.dat	4	6	10	14	20	650	1.237844
ran-bi-50-75.dat	3	6	12	15	20	826	1.177902
ran-bi-80-100.dat	4	5	9	16	20	707	2.662519
ran-bi-80-50.dat	1	8	12	15	18	502	2.231921
ran-bi-80-75.dat	4	5	11	13	18	661	2.163987

Tabela C.6: Continuação da Tabela C.5