UNIVERSIDADE FEDERAL FLUMINENSE

BRUNO JOSÉ DA SILVA BARROS

Algoritmos Heurísticos para o Problema da Árvore Geradora Mínima com Restrições de Conflitos

NITERÓI

2019

UNIVERSIDADE FEDERAL FLUMINENSE

BRUNO JOSÉ DA SILVA BARROS

Algoritmos Heurísticos para o Problema da Árvore Geradora Mínima com Restrições de Conflitos

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense como requisito parcial para obtenção do Grau de Mestre em Computação. Área de concentração: Algoritmos e Otimização.

Orientador:
Luiz Satoru Ochi

Coorientador:
Rian Gabriel Santos Pinheiro

NITERÓI
2019

Ficha catalográfica automática - SDC/BEE
Gerada com informações fornecidas pelo autor

B277a Barros, Bruno José da Silva
Algoritmos Heurísticos para o Problema da Árvore Geradora Mínima com Restrições de Conflitos / Bruno José da Silva Barros ; Luiz Satoru Ochi, orientador ; Rian Gabriel Santos Pinheiro, coorientador. Niterói, 2019.
50 f. : il.

Dissertação (mestrado)-Universidade Federal Fluminense, Niterói, 2019.

DOI: http://dx.doi.org/10.22409/PGC.2019.m.08969397400

1. Árvore Geradora Mínima. 2. Restrições de Conflitos. 3. GRASP. 4. Produção intelectual. I. Ochi, Luiz Satoru, orientador. II. Pinheiro, Rian Gabriel Santos, coorientador. III. Universidade Federal Fluminense. Instituto de Computação. IV. Título.

CDD -

Bibliotecária responsável: Fabiana Menezes Santos da Silva - CRB7/5274

Algoritmos Heurísticos para o Problema da Árvore Geradora Mínima com Restrições de Conflitos

Bruno José da Silva Barros

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense como requisito parcial para obtenção do Grau de Mestre em Computação. Área de concentração: Algoritmos e Otimização.

Aprovada por:

Prof. Luiz Satoru Ochi / IC-UFF (Presidente)

Prof. Rian Gabriel Santos Pinheiro / IC-UFAL

Prof. Uéverton dos Santos Souza / IC-UFF

Profª. Loana Tito Nogueira / IC-UFF

Prof. Igor Machado Coelho / IME-UERJ

Niterói, 15 de agosto de 2019 .

“Nós precisamos saber, e nós saberemos.”

David Hilbert

Para minha mãe, Aureci (in memorian), que um dia ousou acreditar em mim.

Agradecimentos

A minha grande família por acreditar, incentivar, ajudar e fazer sacrifícios que, de inicio, acreditei serem somente meus.

Aos professores do IC-UFF com quem tive o prazer de estudar.

Ao meu orientador, Prof. Satoru, por estar, de várias formas, acompanhando e guiando o trabalho.

Aos meus amigos com quem dividi tempo e conversas durante esta jornada acadêmica, sejam eles de graduação ou pós-graduação.

Ao Prof. Rian Gabriel do IC-UFAL pelos ensinamentos, incentivos, orientações, parceria, amizade e infinitas correções para que este trabalho pudesse acontecer.

A CAPES pelo apoio financeiro.

Bruno Barros

Resumo

Dados um grafo não direcionado com pesos nas arestas e um conjunto de pares de arestas conflitantes, a Árvore de Geradora Mínima com Restrições de Conflito (AGMRC) consiste em encontrar uma árvore geradora com no máximo uma aresta de cada par conflitante e custo mínimo. O problema é $\mathcal{NP}$-difícil e foi introduzido recentemente na literatura. Este trabalho propõe uma nova abordagem para a solução do AGMRC, o método é baseado na meta-heurística GRASP com uso de uma memória adaptativa. A heurística proposta foi avaliada nas instâncias da literatura com milhares de conflitos. Os resultados indicam que o algoritmo proposto foi capaz de encontrar todas as soluções ótimas conhecidas e superar as heurísticas da literatura para o AGMRC.

Palavras-chave: Árvore Geradora Mínima. Restrições de Conflitos. GRASP

Abstract

Given an undirected graph with weights on the edges and a set of conflicting edge pairs, the Minimum Spanning Tree Under Conflict Constraints (MSTCC) consists in finding minimum spanning tree with at most one edge of each conflicting pair and minimum cost. The problem is $\mathcal{NP}$-hard and was introduces recently at literature. This paper proposes a new approach for solving MSTCC, our method is based on the GRASP metaheuristic coupled with adaptive memory programming. We test the proposed heuristic on literature's instances with thousands of conflicts. The results indicate that the proposed algorithm was able to find all known optimal solutions and outperform the best-known heuristics for the MSTCC.

Keywords: Spanning Tree Problem. Conflict Constraints. GRASP.

Sumário

Lista de Figuras viii

Lista de Tabelas ix

1 Introdução 1
- 1.1 Motivação . . . 1
- 1.2 Aplicações para o Problema . . . 4
- 1.3 Trabalhos Relacionados . . . 4
- 1.4 Contribuições da Dissertação . . . 5
- 1.5 Estrutura do Trabalho . . . 6

2 Fundamentos 7
- 2.1 Definições . . . 7
- 2.2 Heurísticas . . . 8
  - 2.2.1 Heurísticas Construtivas . . . 9
  - 2.2.2 Busca Local . . . 9
  - 2.2.3 Meta-heurísticas . . . 10
  - 2.2.4 GRASP . . . 11
  - 2.2.5 ILS . . . 14
  - 2.2.6 Algoritmos Genéticos . . . 15
- 2.3 Métodos Exatos . . . 15
  - 2.3.1 Formulação Matemática . . . 16
  - 2.3.2 Algoritmos para Problemas de Programação Linear Inteira . . . 16

3 Algoritmos para AGMRC 17
- 3.1 GRASP . . . 17
- 3.2 Memória Adaptativa . . . 19
- 3.3 Método Construtivo . . . 20
- 3.4 Busca Local . . . 21
- 3.5 Detalhes de Implementação . . . 21
  - 3.5.1 Cálculo da Função Objetivo . . . 21
  - 3.5.2 Estrutura de Dados Auxiliar . . . 22
  - 3.5.3 Garantia de Soluções Acíclicas . . . 24

4 Resultados Computacionais 26
- 4.1 Ferramentas . . . 26
- 4.2 GRASP Enviesado com MA × GRASP Tradicional . . . 27
- 4.3 GRASP Enviesado com MA × GRASP Enviesado com Função Linear . . . 29
- 4.4 GRASP Enviesado com MA × ILS . . . 30

4.5 GRASP Enviedado com MA × MEGA 31

5 Conclusões e Trabalhos Futuros **34**

Bibliografia **38**

Lista de Figuras

1.1 Instância do AGMRC 2
1.2 AGM obtida pelo algoritmo de Kruskal com custo igual a 12 3
1.3 Árvore geradora mínima sob restrições de conflitos com custo igual a 24 3
3.1 Exemplo da estrutura de conflitos 24
3.2 Marcação dos vértices com a DFS 25
4.1 Gráfico *Time to Target* das instâncias I_1 e I_5 32

Lista de Tabelas

4.1 Instâncias `type1` 27
4.2 Comparação com GRASP tradicional 27
4.3 Comparação do tempo médio dos GRASPs com LRC 28
4.4 Comparação com viés linear 29
4.5 Comparação do ILS com o GRASP com MA 30
4.6 Comparação do tempo de execução médio do ILS com o GRASP com MA . . 31
4.7 Comparação do MEGA com o GRASP utilizando MA 31

Capítulo 1

Introdução

Neste capítulo é descrito o problema estudado neste trabalho: o problema da árvore geradora mínima com restrição de conflitos (AGMRC). Primeiramente, é descrita a motivação de se estudar este problema. Em seguida, descrevem-se algumas aplicações para o problema, os principais trabalhos relacionados e as contribuições deste trabalho. Por fim, a estrutura do trabalho é apresentada.

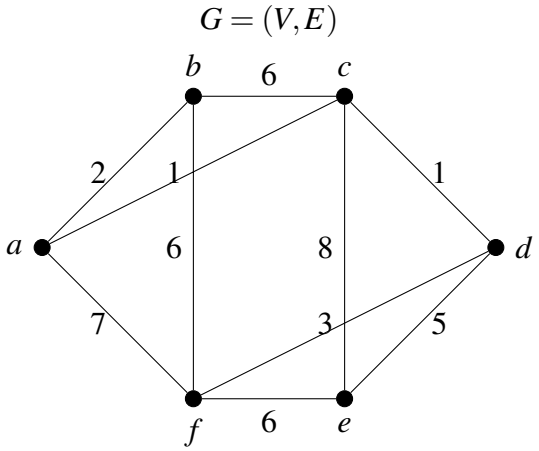
1.1 Motivação

Como já é conhecido, o problema da Árvore Geradora Mínima, (AGM, *Spanning Tree Problem* em inglês) é um problema de muita importância na área de otimização combinatória e inteligência computacional. Encontrar uma AGM de um grafo é um problema que pode ser resolvido em tempo polinomial pelo algoritmo de Kruskal [28], ou pelo algoritmo de Prim [34]. Esses dois algoritmos são gulosos, e encontram uma árvore geradora de custo mínimo de um grafo com arestas ponderadas. Apesar disso, algumas variações deste problema são $\mathcal{NP}$-difíceis. Este trabalho foca na variante conhecida como o problema da Árvore Geradora Mínima com Restrições de Conflitos (AGMRC, *Spanning Tree Problem under Coflict Constraints* em inglês). Recentemente, alguns problemas clássicos foram considerados sob uma visão que considera restrições de conflitos. Um exemplo que ganhou esta abordagem é o *Bin Packing* com restrições de conflitos [9, 10], ou mesmo os problemas de caminho mínimo ou emparelhamento com conflitos [17].

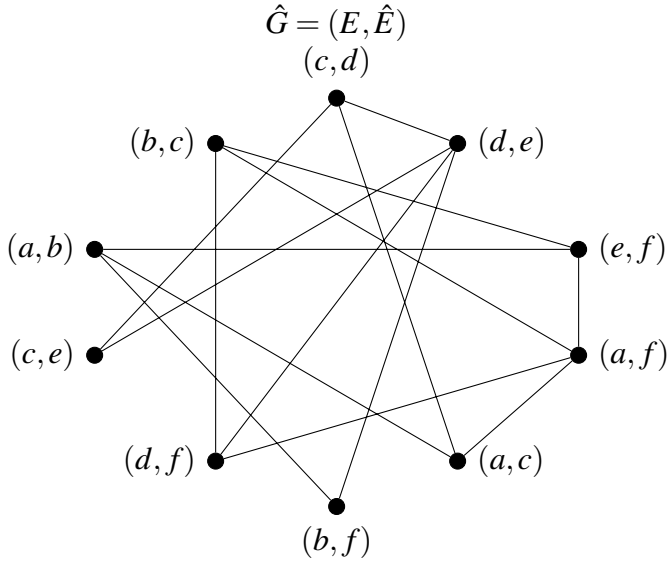
Definição do AGMRC: Dados um grafo $G = (V,E)$, uma função de peso ω que atribui a cada aresta $e \in E$ um peso $\omega(e)$ e um grafo de conflitos $\hat{G} = (E,\hat{E})$ em que os vértices de $\hat{G}$ são as arestas de G e $\hat{E}$ representa os pares de arestas conflitantes. O problema da AGMRC consiste em encontrar uma árvore geradora T de G com custo mínimo, tal que, se duas arestas $(e_i, e_j) \in \hat{E}$, então $(e_i, e_j) \notin T$, isto quer dizer, T é livre de conflitos.

A Figura 1.1 mostra um exemplo de uma instância para o problema AGMRC, a Figura 1.1(a) mostra o grafo G e a Figura 1.1(b) mostra o grafo de conflitos $\hat{G}$ associado a G.

Se for considerado apenas o grafo G, uma AGM de G (obtida pelo algoritmo de Kruskal, por exemplo) pode ser observada na Figura 1.2, o custo desta solução é 12. Note que a AGM apresenta alguns conflitos que não permitidos pelo problema da AGMRC, como por exemplo as arestas (a,b) e (a,c). Desta forma, quando as restrições de conflito são consideradas, — conflitos indicados pelo grafo $\hat{G}$ da Figura 1.1(b)—, a solução para o AGMRC passa a ser a árvore representada na Figura 1.3 e o custo é de 24. É fácil notar que o valor da AGMRC é maior ou igual ao valor da AGM.



(a) Exemplo de Grafo



(b) Grafo de Conflitos

Figura 1.1: Instância do AGMRC

É possível notar que uma solução livre conflitos para o problema da AGMRC é uma árvore em G que induz um conjunto independente em $\hat{G}$. Na Figura 1.3, o conjunto de arestas selecionadas, $\{(a,c),(b,f),(c,e),(e,f),(e,f)\}$ formam um conjunto independente em $\hat{G}$ da Figura 1.1(b).

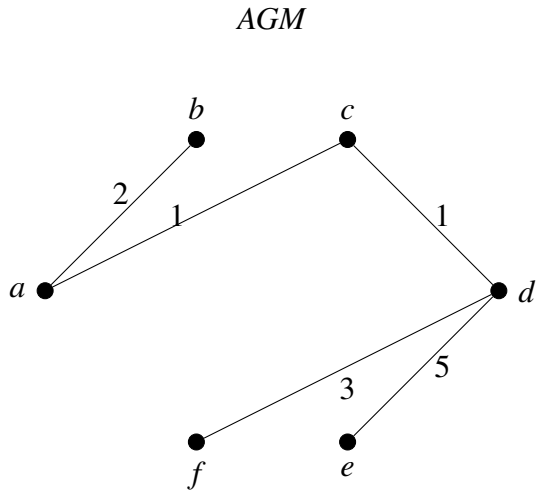


Figura 1.2: AGM obtida pelo algoritmo de Kruskal com custo igual a 12

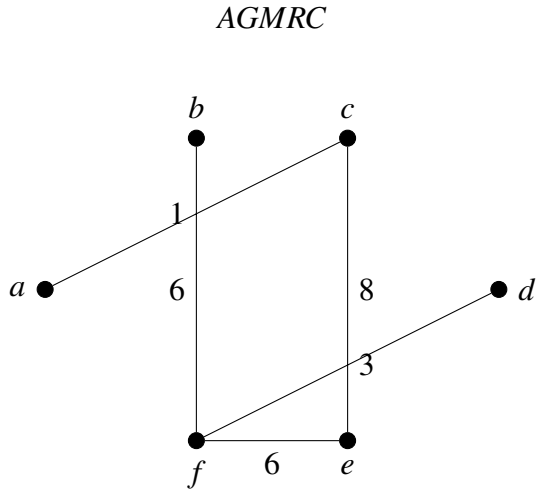


Figura 1.3: Árvore geradora mínima sob restrições de conflitos com custo igual a 24

1.2 Aplicações para o Problema

Muitas aplicações utilizam a AGM, como por exemplo: projetos de redes de computadores, transportes, transmissões em redes de computadores [15], projetos de circuitos [33] ou taxonomia bacteriana [38]. Alguns trabalhos também estudam a AGM em grafos representando as relações financeiras para analisar o mercado financeiro [30, 18]. Em muitas dessas aplicações, alguns conflitos podem surgir, impossibilitando que alguns pares de relacionamentos sejam utilizados. Em Darmann et al. [16], foi descrita uma aplicação bem específica do AGMRC, que é a instalação de um oleoduto com o objetivo de conectar vários países. A forma mais barata de construir o sistema é utilizando a árvore geradora mínima, mas por razões políticas, técnicas ou por questões internas das várias empresas, podem surgir muitos conflitos. Por exemplo, uma empresa pode não estar disposta a colaborar como outra. Nesse caso, o problema clássico da AGM não representa uma solução viável, já que é necessário respeitar os conflitos.

1.3 Trabalhos Relacionados

O AGMRC foi introduzido por Darmann et al. [16, 17] juntamente com sua demonstração de $\mathcal{NP}$-dificuldade e uma formulação matemática. Esses dois primeiros trabalhos tiveram um caráter bastante teórico, por exemplo, os autores mostraram que este problema é fortemente $\mathcal{NP}$-difícil mesmo se o grafo de conflitos seja um conjunto de caminhos de tamanho igual a dois. E que é polinomial se for um conjunto de caminhos de tamanho igual a um. Outras propriedades do problema foram estudadas por Zhang et al. [42]. Além de disponibilizarem um conjunto de instâncias para o problema, os autores mostraram que, se o grafo de conflitos é uma coleção de cliques disjuntas, o problema pode ser resolvido em tempo polinomial, na verdade, esse resultado de clique é uma generalização do resultado sobre caminhos de tamanho igual a um, pois caminhos de tamanho um são coleções de cliques. Além disso, provaram que se G é um grafo cacto e $\hat{G}$ é um grafo qualquer, então, é possível encontrar uma solução viável (ou provar não existir solução viável) em tempo polinomial, entretanto, a versão de otimização mantém-se $\mathcal{NP}$-difícil. Um cacto é um grafo conexo em que todo bloco é um ciclo ou uma aresta, ou seja, dois ciclos têm no máximo um vértice me comum.

Zhang et al. [42] ainda propuseram: um algoritmo construtivo, um método de busca local, duas meta-heurísticas (sendo uma dela um *Tabu Search* e a outra um *Tabu Thresholding*), e uma formulação de programação linear inteira. Vale destacar que foram os primeiros algoritmos heurísticos para o AGMRC.

Uma outra contribuição desse mesmo trabalho foi desenvolver testes de inviabilidades que foram apresentados com dois lemas e um corolário. O resultados serão apresentados a seguir, porém, as demonstrações podem ser encontradas em Zhang et al. [42].

Lema 1. *Seja* α *é um limitante superior para o conjunto dominante no grafo de conflitos, se*

$\alpha < |V(G)| - 1$ *então o AGMRC é inviável.*

Lema 2. *Seja* β *um limitante inferior para a cobertura mínima por vértices de* $\hat{G}$ *e* $|E(G)| - \beta < |V(G)| - 1$*, então AGMRC é inviável. Além disso, se* $D_1, D_2, \ldots, D_r$ *é uma partição de cliques de* $\hat{G}$ *e* $|V)G)| - 1 > |E(G)| + r - \sum_{i=1}^{r} |D_i|$*, então AGMRC é inviável.*

Corolário 1. *Seja M o emparelhamento de cardinalidade máxima de* $\hat{G}$*. Se* $|V(G)| - 1 > |E(G)| - |M|$*, então AGMRC é inviável.*

Posteriormente, Samer & Urrutia [36, 37] propuseram um algoritmo *branch and cut* (B&C) que melhorara os resultados da literatura na época, porém os *gaps* obtidos ainda permaneceram altos para muitas instâncias. Bittencourt et al. [6] apresentaram uma nova formulação matemática baseada em restrições *MTZ* (Miller, Tucker e Zemlin). No trabalho de Barbosa & Delbem [2], foi apresentado um algoritmo ILS (*Iterated Local Search*) e uma nova busca local. Apesar da melhora dos resultados da literatura, ainda existem instâncias com soluções viáveis que não se conhece o valor ótimo.

A seguir é apresentada a formulação para o AGMRC proposta por Darmann et al. [16]:

$$(\text{AGMRC}) \qquad \min \sum_{e \in E} \omega(e) x_e \tag{1.1}$$

$$\text{s.a:} \qquad \sum_{e \in E} x_e = n - 1 \tag{1.2}$$

$$\sum_{e \in E(S)} x_e \leq |S| - 1 \qquad \forall S \subseteq V, |S| \geq 3 \tag{1.3}$$

$$x_e + x_f \leq 1 \qquad \forall (e, f) \in \hat{E} \tag{1.4}$$

$$x_e \in \{0, 1\} \qquad \forall e \in E \tag{1.5}$$

Nesta formulação, as variáveis de decisão x_e valem 1 se e somente se a aresta e pertence à solução ótima. A Equação (1.1) é a função objetivo do problema, a Restrição (1.2) garante que $(n-1)$ arestas sejam escolhidas, as Restrições (1.3) são as restrições de eliminação de subciclo, as Restrições (1.4) eliminam pares de arestas conflitantes, e por ultimo as Restrições (1.5) garantem a integralidade das variáveis.

Recentemente, Carrabs et al. [11] propuseram um algoritmo genético para a AGMRC. Em outro trabalho, Carrabs et al. [12] propuseram novas equações válidas para a AGMRC e implementaram um novo algoritmo B&C, além disso geraram um novo conjunto de instâncias para o problema.

1.4 Contribuições da Dissertação

As contribuições deste trabalho podem ser resumidas como:

- algoritmo GRASP para o AGMRC;

- aprimoramento das busca locais encontradas na literatura;
- nova variante de GRASP enviesado com memória adaptativa;

1.5 Estrutura do Trabalho

A continuação deste trabalho está organizado da seguinte maneira:

- Capítulo 2: apresenta os fundamentos teóricos sobre os métodos heurísticos;
- Capítulo 3: contém a proposta de um algoritmo GRASP enviesado com memória adaptativa;
- Capítulo 4: exibe e discute os resultados computacionais;
- Capítulo 5: apresenta conclusões e trabalhos futuros.

Capítulo 2

Fundamentos

Neste capítulo são apresentados os principais fundamentos para o entendimento do trabalho. Primeiramente serão apresentadas algumas definições básicas, em seguida, alguns conceitos sobre os algoritmos heurísticos.

2.1 Definições

Um *problema de otimização P* é definido como um conjunto de restrições e um conjunto de instâncias no qual procura-se uma solução que seja melhor que qualquer outra. Em que uma *instância* de P é definida como uma entrada válida para o problema P e as restrições são um conjunto de condições que devem ser atendidas para garantir uma solução para o problema. Se todas as restrições são satisfeitas por uma solução s, então s é dita viável.

Um Problema de Otimização tanbém pode ser formulado da seguinte forma:

$$\begin{aligned} \max \quad & f(x) \\ \text{s. a} \quad & g_i(x) \leq 0, i = 1, \ldots, m. \end{aligned}$$

em que, $f : \mathbb{R}^n \to \mathbb{R}$ e $g_i : \mathbb{R}^n \to \mathbb{R}, (i = 1, \ldots, m)$. Note que neste caso, todas as funções são contínuas. Uma solução é dita *viável* quando se ela satisfaz todas as restrições, ou seja, todas as inequações $g_i(x) \geq 0$ são válidas. A função objetivo f associa para toda solução viável (as vezes inviável) um número real que indica o valor ou a qualidade da solução.

O problemas em que todas as restrições g_i, juntamente com a função objetivo f são funções lineares, recebem o nome de problemas de programação linear. Além disso, se o domínio de x for restringido ao conjunto dos inteiros, dá-se o nome de problema de programação linear inteira (PPI).

Define-se uma *estrutura de vizinhança* $N(s)$ de uma solução s para o problema P como uma função $N : S \to 2^S$, que atribui a cada solução $s \in S$ um o conjunto de soluções $N(s) \subseteq S$. Uma solução $s' \in N(s)$ em uma vizinhança de s é denotada vizinha de s, e é obtida através de um

movimento m que transforma s em s'. Uma solução s é dita ser um *ótimo local* em relação à estrutura de vizinhança N, se $\forall s' \in N(s), f(s) \leq f(s')$, em que f é a avaliação de uma solução e o problema P é de minimização. Sem perda de generalidade, para um problema de minimização, S^* é dito *ótimo global* de P se $\forall s \in S, f(s^*) \leq f(s)$, em que S é o conjunto de todas as soluções viáveis da instância i para o problema P.

Dada a definição de um problema de otimização, existem técnicas já consagradas de busca de soluções. Estas técnicas são estudadas em larga escala e podem ser utilizadas e facilmente adaptadas para diferentes problemas. A busca consiste em, partindo da definição e da instância de um problema, definir um estado inicial e ir construindo e testando outros estados afim de chegar àquele que caracteriza uma solução. Esse tipo de busca é conhecido como *heurística construtiva* e foca apenas em construir uma solução para o problema. Outras técnicas também são bem conhecidas e visam, além de determinar uma solução, encontram uma solução de boa qualidade, como é o caso dos algoritmos exatos e das meta-heurísticas.

Para implementar algoritmos que achem essas boas soluções, é necessária uma forma de avaliar uma solução, para isso, utiliza-se uma função de custo C, que possibilita comparar duas soluções. Por exemplo, no problema clássico do caixeiro viajante (PCV), uma função de custo seria uma função que calcula a soma total das distâncias do trajeto escolhido, dessa forma, pode-se determinar a melhor entre duas soluções s_1 e s_2.

2.2 Heurísticas

Existem vários tipos de algoritmos que não garantem a otimalidade da solução de um problema de otimização. Basicamente, esse algoritmos são divididos em: algoritmos de aproximação e heurísticas.

Os algoritmos de aproximação oferecem garantias com relação à qualidade das soluções encontradas, ou seja, garantem que a solução encontrada não fica a mais de uma constante do valor ótimo. Para isso, eles são, geralmente, desenvolvidos especificamente para o problema tratado. No entanto, na maioria das vezes, a qualidade das aproximações fica distante da qualidade das soluções ótimas [40].

As heurísticas não oferecem nenhuma garantia com relação à qualidade das soluções. Geralmente, divide-se as heurísticas em: heurísticas específicas e meta-heurísticas. Como o nome diz, a primeira classe é desenvolvida especialmente para o problema abordado, já as meta-heurísticas podem ser vistas como *frameworks* genéricos que podem ser adaptados para diferentes problemas, no geral, são um conjunto de técnicas de como guiar uma heurística até uma solução de boa qualidade. Entretanto, as meta-heurísticas não possuem nenhum mecanismo capaz de garantir o quão próximo de uma valor ótimo a solução está.

2.2.1 Heurísticas Construtivas

Os algoritmos heurísticos utilizam as características específicas de um problema para construir uma solução. Nesta classe, se encontram as heurísticas gulosas. As heurísticas gulosas fazem sempre a escolha do melhor estado, dentre os estados disponíveis para escolha. Note que, essa estratégia a cada iteração adiciona à solução corrente apenas um candidato. Exatamente o que tem a melhor avaliação por iteração. Esses algoritmos descrevem-se também como míopes, pois só consideram o campo de visão atual para fazer sua escolha. Em outras palavras, se a escolha de um candidato acarretar que a próxima escolha seja muito ruim, o algoritmo não considerará este fato, apenas vai escolher o melhor dentre os disponíveis a cada iteração [14, 39, 23].

Uma adaptação dos algoritmos gulosos que também é bem popular, são os algoritmos gulosos aleatorizados. Nesse caso, para tomar a decisão de qual candidato escolher, são atribuídas probabilidades a cada opção de acordo com a sua qualidade. Por exemplo, considere o problema do caixeiro viajante, em que uma dada cidade x, com opção de 4 cidades (a, b, c e d) como escolha de cidade seguinte. Cujas distâncias para cada cidade a partir de x são: $4, 2, 1$ e 1 quilômetros respectivamente. Uma forma de selecionar a próxima cidade na solução é atribuir as probabilidades $\frac{1}{11}, \frac{2}{11}, \frac{4}{11}$ e $\frac{4}{11}$ a cada uma delas, respectivamente. Nesse caso, as cidades c e d têm, cada uma, 4 vezes mais chances de serem selecionadas do que a cidade a. A probabilidade de escolha de qual cidade seguir pode ser definida de várias formas, como por exemplo, de acordo com ordenação segundo um critério guloso. Esse tipo de seleção aleatória pode ser facilmente implementado simulando-se uma roleta.

As heurísticas gulosas são, em geral, facilmente implementadas para qualquer problema. Mas existem várias outras heurísticas construtivas que são implementadas seguindo as características de cada problema.

2.2.2 Busca Local

As buscas locais são um tipo de heurísticas de melhoramento, são algoritmos nos quais, dado problema P, uma instância i, uma estrutura de vizinhança $N(\cdot)$ e uma solução particular s, então usa-se alguma estratégia predefinida para tentar melhorar a solução já existente.

As buscas locais basicamente são algoritmos iterativos nos quais, definido um critério de modificação para melhoramento na solução, ficam tentando melhorá-la até que não seja mais possível [39, 23].

Podemos dividir essas buscas em dois principais tipos: *first improvement* e *best improvement*, entretanto existem outros tipos que não se enquadram em nenhum desses dois tipos. No *first improvement* a busca na vizinhança é executada até que um melhor vizinho seja encontrado. A busca, nesse momento, pode parar ou adotar essa nova solução para continuar a busca. Já no *best improvement*, toda a vizinhança é avaliada e a escolha do elemento é feita na direção do melhor vizinho. Caso a vizinhança seja muito densa, algumas técnicas podem ser utilizadas

para evitar avaliá-la por completo. O Algoritmo 2.1 apresenta um *template* de uma busca local com *first improvemet*.

Algoritmo 2.1 Exemplo de *first improvemet*

```
1: função BUSCA LOCAL(s_0, f(·), N(·))
2:     s ← s_0
3:     para s' ∈ N(s) faça
4:         se f(s') < f(s) então
5:             retorne s'
6:         fim se
7:     fim para
8: fim função
```

2.2.3 Meta-heurísticas

Diferentemente das heurísticas específicas, as meta-heurísticas são aplicadas a problemas gerais e provêm de mecanismos de escape de ótimos locais de baixa qualidade. Dentre, as meta-heurísticas mais utilizadas destacam-se: algoritmos genéticos, *simulated annealing*, colônia de formigas, VNS, ILS, GRASP e busca tabu. Cada método tem sua peculiaridade e uma forma diferente de escape de ótimos locais. Mais detalhes podem ser encontrados nos trabalhos de Talbi [40] e Gendreau & Potvin [22].

As meta-heurísticas GRASP (sigla para *Greedy Randomized Adaptive Search Procedure*) e ILS (siglaGRAS para *Iterated Local Search*) são altamente difundidas para resolver problemas combinatórios em grafos. Existem ainda os Algoritmos Genéticos (AG) que se baseiam na seleção natural descrita por Darwin e várias outras classes.

Das teorias que fundamentam o uso de meta-heurísticas há dois termos bem recorrentes que devem ser balanceados a fim de se obter sucesso na aplicação desse método: diversificação e intensificação. Diversificação consiste em investigar uma porção maior no espaço de busca, ou seja, aumentar as chances de encontrar outras soluções promissoras que ainda precisam ser refinadas. Isso facilita a fuga de ótimos locais. Já a intensificação consiste em concentrar a busca em uma região menor e limitada, mas promissora, do espaço de busca, assim, as chances de melhorar uma solução já existente são grandes. Um método de busca local é uma forma de intensificação [3].

As estratégias de busca de diferentes meta-heurísticas são altamente dependentes do seu funcionamento, da sua filosofia em si. Assim, a principal diferença entre as meta-heurísticas está relacionada a como cada uma lida com esse balanceamento entre diversificação e intensificação [5].

Dessa forma, Birattari et al. [5] definem algumas propriedades para a classificação de meta-heurísticas, tais como:

- Solução única × Baseada em população — diz respeito ao uso de um único ponto de busca (uma única solução candidata por vez), como exemplo de meta-heurística de solução única, pode-se citar *Simulated Annealing*, *Itered Local Search (ILS)*, *Variable Neighborhood Search (VNS)*, Busca tabu, etc. Em contrapartida, há os métodos que mantêm e melhoram múltiplas soluções candidatas usando características populacionais para guiar a busca, para esse caso destacam-se *Ant Colony Optimization (ACO)*, Algoritmos Genéticos, entre outros.

- Uso da memória × métodos sem memória — outro ponto é o uso da memória das soluções antigas para guiar o rumo do algoritmo para novas direções. Como exemplo de meta-heurísticas que não utilizam as soluções antigas tem-se o *GRASP* e *Simulated Annealing*, em contrapartida, Busca Tabu e *ACO* são exemplos que utilizam de algum tipo de memória para se guiarem.

- Inspiração na natureza — vários métodos são na verdade inspirados em algum fenômeno da natureza e tentam tomar vantagem desse fenômeno para criar soluções inteligentes para problemas de otimização combinatória. Destacam-se citar aqui *ACO*, Algoritmos Genéticos e *Simulated Annealing*

Uma outra técnica bastante comum é hibridizar meta-heurísticas para resolver problemas. Um exemplo bem difundido de hibridização é o GRASP com reconexão de caminhos [20], onde o conceito de população elite é incorporado ao GRASP para fazer reconexão de caminhos. Além de combinar as ideias de diferentes meta-heurísticas, a hibridização também pode combinar técnicas de meta-heurísticas com métodos exatos. Este último é conhecido como *matheuristic* [7].

2.2.4 GRASP

O GRASP (*Greedy Randomized Adaptive Search Procedure*) é uma meta-heurística em que cada iteração consiste basicamente de duas fases: construção gulosa aleatória e busca local [20]. O construtivo da solução do GRASP é um passo muito importante, no geral, o construtivo é responsável pela diversificação das soluções. Então a busca local se preocupa apenas com a etapa de intensificação.

O GRASP é um tipo de meta-heurística que foca na importância da diversificação de soluções e, é uma das poucas meta-heurísticas que indicam como o construtivo deve funcionar quanto a escolha de candidatos para entrar na solução.

A construção da solução, a cada iteração, é guiada por uma Lista Restrita de Candidatos (LRC). A LRC guarda, a cada iteração do construtivo, os candidatos que podem entrar na solução parcial. Além disso, o GARSP conta com uma função (gulosa) de avaliação dos candidatos c que informa o custo de adicionar um elemento e à solução s. Todos os candidatos na LRC têm a mesma probabilidade de entrar na solução. O GRASP também utiliza um parâmetro $\alpha \in [0,1]$

para controlar o quão aleatório ou guloso o construtivo será. Funciona da seguinte forma: se todos os elementos em LRC podem ser selecionados com igual probabilidade, então o construtivo é exatamente o aleatório, para evitar isso a LRC é limitada a apenas alguns candidatos. Seja $c(e_{min})$ o candidato com o menor custo (considerando um problema de minimização, para um problema de maximização o raciocínio se aplica de forma análoga) e $c(e_{max})$ o de maior custo dentre todos em LRC, apenas podem entrar na solução os candidatos que satisfazem a seguinte condição:

$$e \in LRC \Leftrightarrow c(e_{min}) \leq c(e) \leq c(e_{min}) + \alpha(c(e_{max}) - c(e_{min})) \tag{2.1}$$

A cada iteração do construtivo, a LRC é reconstruída e os elementos reavaliados antes do próximo candidato ser escolhido. Sendo assim, se $\alpha = 0$ o construtivo é exatamente o guloso, e se $\alpha = 1$ é exatamente o aleatório. O Algoritmo 2.2 apresenta a fase construtiva do GRASP.

Algoritmo 2.2 Algoritmo construtivo guloso aleatório

```
1:  função SOLUCAO_GULOSA_ALEATORIA(α)
2:      s ← ∅
3:      Avalie os custos de todos os elementos candidatos
4:      enquanto Solução incompleta faça
5:          Crie a LRC de acordo com o parâmetro α
6:          e ←SeleçãoAleatória(LRC)
7:          s ← s ∪ {e}
8:      fim enquanto
9:      retorne s
10: fim função
```

Depois de terminada a construção a busca local é ativada. A solução geral é atualizada sempre que a fase de busca encontra uma solução melhor que todas as outras já encontradas. O processo é repetido até um certo número de iterações ou outro critério de parada seja satisfeito. O Algoritmo 2.3 apresenta um pseudocódigo da meta-heurística.

Algoritmo 2.3 Pseudo-Código do GRASP

```
1: função GRASP(problema P, instância i)
2:     f(s*) ← ∞
3:     enquanto Condição de parada não for satisfeita faça
4:         s ← solucao_gulosa_aleatoria(α)
5:         s ←busca_local(s)
6:         se f(s) < f(s*) então
7:             s* ← s
8:         fim se
9:     fim enquanto
10:    retorne s*
11: fim função
```

No Algoritmo 2.3 a variável s é a solução de cada iteração, s^* é a melhor solução até o momento e a função f é a função que calcula o custo de uma solução. A cada iteração uma solução gulosa aleatória é criada e então é melhorada com a busca local. O passo seguinte é avaliar a nova solução e se for melhor que o *best* (s^*) então essa nova solução é salva.

GRASP com Enviesamento

O GRASP com enviesamento consiste de uma mudança na fase de construção do GRASP. Na Seção 2.2.4 foi explicado como a LRC é utilizada para construir uma solução no GRASP. No GRASP com enviesamento, a construção funciona de forma diferente. Uma função de distribuição de probabilidade é utilizada para enviesar a LRC. Inicialmente, os elemento de LRC são avaliados e ordenados de acordo com um critério guloso, com a ordenação, é obtido a classificação (*rank*) de cada elemento, esse *rank* será utilizado para enviesar a LRC. A princípio, como explicado em Bresina [8], qualquer função de distribuição de probabilidade pode ser utilizada para fazer o enviesamento em um algoritmo guloso aleatório, mas algumas foram tratadas com destaque.

Seja $e \in RLC$, e $r(e)$ a classificação do elemento e. O viés (*bias*) de e pode ser calculado de acordo com as seguintes funções:

- *Aleatório*: $bias(r(e)) = 1$;
- *Logarítmica*: $bias(r(e)) = \log^{-1}(r(e))$;
- *Polinomial de ordem n*: $bias(r(e)) = r(e)^{-n}$;
- *Linear:* $bias(r(e)) = 1/r(e)$.

A probabilidade, $P(e)$, do elemento e ser escolhido é dado por:

$$P(e) = \frac{bias(r(e))}{\sum_{i \in RLC} bias(r(i))} \tag{2.2}$$

Inicialmente, o critério de enviesamento de Bresina [8] não foi proposto especificamente para o GRASP. Posteriormente, Binato et al. [4] aplicaram o método de Bresina [8] na etapa construtiva do GRASP. Outros trabalhos, como Expósito et al. [19], também aplicaram a mesma estratégia com sucesso.

2.2.5 ILS

O ILS (*Iterated Local Search*) ou algoritmo de Busca Local Iterada, é um algoritmo que foca na busca local para obter soluções cada vez melhores. Funciona da seguinte forma: quando a busca local é aplicada à uma solução e não é mais possível melhorá-la mediante apenas a busca local, o ILS usa a estratégia de fazer com que as soluções saltem para regiões diferentes no espaço de solução. Inicialmente o ILS utiliza, em muitos casos, uma heurística gulosa para criar uma solução de partida, e entra em *loop* até que uma condição de parada seja satisfeita. A cada iteração do algoritmo, (*i*) é aplicada uma função de perturbação; (*ii*) em seguida realiza-se a busca local; (*iii*) e então são verificados os critérios de aceitação que podem aceitar ou rejeitar a nova solução [29].

Por sua natureza focada na busca local, o ILS é uma meta-heurística em que o principal elemento é a intensificação. Uma vez definida uma solução inicial, toda a intensificação da busca acontece em torno dessa solução inicial. Já a diversificação ocorre através da pertubação.

Uma função de perturbação é uma forma de fazer um algoritmo escapar de algum *ótimo local* de de baixa qualidade [39]. Geralmente, quando se está fazendo alguma melhoria na solução de um problema (utilizando uma busca local), o algoritmo fica preso ótimo local ou alguma área de planície. Quando isso acontece, uma estratégia bastante usada é perturbar a solução afim outras melhorias possam tentar ser feitas. Sendo assim, a perturbação age diversificando uma solução, de forma que, os algoritmos possam fugir dos ótimos locais. Funções de perturbação são geralmente do tipo estocástico [39, 23]. Entretanto, a perturbação deve ser equilibrada, se for muito drástica a busca local dificilmente conseguirá bons resultados, e se for muito fraca não será suficiente para escapar do ótimo local. Em muitos caso, a perturbação pode ser simplesmente a troca aleatória de candidatos que não estão na solução por candidatos que estão.

O ILS é um algoritmo que utiliza uma estrutura de vizinhança, e sua busca por novas soluções mais eficazes baseia-se em direcionar a busca sempre para o melhor vizinho. O pseudocódigo descrito no Algoritmo 2.4, é um exemplo do ILS.

Algoritmo 2.4 Pseudo-Código do ILS

```
1: função ILS(problema P, instância i)
2:     s* ← inicializa_solucao()
3:     s* ← busca_local(s*)
4:     enquanto Condição de parada não for satisfeita faça
5:         s ← perturbacao(s*)
6:         s ←busca_local(s)
7:         s* ←criterio_aceitacao(s, s*)
8:     fim enquanto
9:     retorne s*
10: fim função
```

Como mostrado no pseudocódigo do Algoritmo 2.4, uma solução inicial é criada, enquanto nenhuma condição de parada for satisfeita a solução é perturbada e depois melhorada com a busca local. Finalmente, os critérios de aceitação da solução são verificados.

2.2.6 Algoritmos Genéticos

Os algoritmos genéticos, são algoritmos que se baseiam na teoria evolutiva de Darwin [39]. A execução destes algoritmos funciona assim: primeiro é criada uma população inicial de indivíduos[1]; cada individuo sofre uma mutação e a nova geração de indivíduos é formada combinando dois estados pais [24]. A cada geração, uma função de adaptação é executada para verificar se os indivíduos se adaptaram, excluindo indivíduos não promissores de gerar descendentes, que provavelmente, não seriam bons. Como os algoritmos genéticos utilizam elementos aleatórios, como a mutação por exemplo, são considerados um tipo de busca estocástica [39, 23]. Nos algoritmos genéticos também é comum encontrar técnicas de outros algoritmos, como por exemplo utilizar uma busca local na população para refinar a busca.

2.3 Métodos Exatos

Métodos exatos são métodos que retornam a melhor solução garantindo sua otimalidade. Geralmente estes métodos, para muitas classes de problemas, podem demorar muito para executar, como por exemplo para problemas $\mathcal{NP}$-completos, chegando, em casos extremos, a passar até anos executando e ainda não retornar a solução ótima procurada. Mesmo assim esses métodos são de extrema importância para a resolução de muitos problemas. São úteis para atestar a qualidade de métodos heurísticos e, nos casos em que o tempo não for prioridade, são uma

[1] Nesse caso, os indivíduos são soluções que podem ser geradas aleatoriamente por exemplo, ou seja a população é um conjunto de soluções.

excelente forma de resolver problemas, além disso, certos algoritmos exatos são implementados com algumas técnicas que aceleram a execução.

Existem diversas formas de se resolver um problema utilizando algoritmos exatos; uma delas é por formulação matemática, onde depois de pronta a formulação é preciso utilizar algum *software* que resolva, poupando assim esforço do programador. Esse *software* normalmente implementa algoritmos do tipo *branch-and-bound*, que utilizam enumeração para resolver o problema, ou seja, tenta verificar todas as soluções possíveis. No entanto, o *branch-and-bound* possui formas de evitar que todas as soluções possíveis sejam avaliadas através de podas.

2.3.1 Formulação Matemática

A formulação matemática descreve, usando uma sintaxe matemática, uma forma de encontrar a solução ótima de um problema. Cada problema exige uma formulação própria, pois para tal é necessário utilizar suas características. Com a formulação matemática pronta é necessário utilizar algum *solver*[2] para executá-la.

2.3.2 Algoritmos para Problemas de Programação Linear Inteira

Uma das formas de resolver um PPI é por *Branch-and-Bound* (B&B) [41]. O B&B encontra a solução ótima do PPI dividindo o problema em subproblemas e resolvendo cada um separadamente. Para dividir um problema é resolvida a relaxação linear, após a resolução, se existir alguma variável fracionária, uma pode ser escolhida uma para ramificação com valores inteiros. E os subproblemas são resolvidos novamente. A cada resolução é possível calcular limites que permites descartar a ramificação de alguns nós da árvore do B&B.

Uma outra forma muito utilizada para resolver e estudar um PPI é com planos de corte [41]. Os planos de cortes são inequações válidas que cortam uma solução fracionária. Quando a relaxação linear de um PPI for resolvida e a solução não for inteira, é possível adicionar inequações que invalidem essa solução. Uma família de cortes muito conhecida são os cortes de Chvátal-Gomory [25, 13].

A junção do B&B com planos de cortes é conhecida como *Branch-and-Cut*(B&C). Basicamente o B&C é um B&B onde a cada iteração novos cortes são adicionados.

[2]Software utilizado para rodar a formulação matemática que já possui uma sintaxe própria para implementação da formulação, por exemplo, o *CPLEX*.

Capítulo 3

Algoritmos para AGMRC

Neste capítulo é apresentada uma meta-heurística GRASP (*Greedy Randomized Adaptive Search Procedure*) com memória adaptativa para o AGMRC. Algoritmos similares foram utilizados com êxito na resolução de diversos problemas como: o Problema de Recobrimento de Rotas [31] e o Problema de Roteamento de Veículos com Múltiplas Origens [32], outro trabalho que também faz uso de memória adaptativa é Gonçalves et al. [26], entretanto estes trabalhos usam a memória adaptativa apenas para calibrar o valor de α de acordo com a execução do algoritmo. Com a técnica da memória adaptativa, a resolução se apoia na ideia de deixar o algoritmo aprender a resolver cada instância de teste, isto é, não engessar o algoritmo exageradamente, dessa forma, o algoritmo torna-se flexível às características intrínsecas da instância.

3.1 GRASP

Greedy Randomized Adaptive Search Procedure (GRASP) é uma meta-heurística iterativa em que cada iteração consiste de duas fases: uma para a construção de uma solução inicial e outra que busca melhores soluções a partir desta [21]. A melhor solução (*best solution*) é atualizada sempre que for encontrada uma solução melhor que todas as anteriores. O processo é repetido até um certo número de iterações ou até um outro critério de parada estabelecido ser satisfeito. Nos últimos anos, o GRASP tem sido aplicado com sucesso em uma grande variedade de problemas de otimização [35].

A meta-heurística GRASP, proposta por Feo & Resende [20], consiste em um método guloso aleatório seguido de uma busca local. Dentre as meta-heurísticas existentes, o GRASP tem se mostrado uma das mais competitivas. No entanto, assim como outros *frameworks* dedicados a resolver problemas $\mathcal{NP}$-completos, o método pode não conseguir resolver bem um problema específico caso não conte com adaptações para se adequar às características do problema que se pretende resolver. O GRASP possui várias adaptações conhecidas, muitas delas podem ser encontradas em Resende & Ribeiro [35]. Um dos mecanismos que é muito utilizado é o uso de memória.

No contexto de meta-heurísticas, vários tipos de memória podem ser utilizados. Na meta-heurística Busca Tabu por exemplo, utiliza-se uma memória de curto prazo para se evitar um retorno rápido às soluções já verificadas [27]. Existem também meta-heurísticas que fazem uso de um conjunto elite de solução, como por exemplo o GRASP com reconexão de caminhos [35]. Outro tipo de memória, denominada memória adaptativa ou memória flexível, permite que a meta-heurística aprenda no decorrer de suas iterações. No caso do GRASP, o conhecimento desta memória de longo prazo é utilizado na fase de construção.

O Algoritmo 3.1 apresenta um pseudocódigo do GRASP proposto. No algoritmo, a variável T_b representa a melhor solução encontrada até o momento e a T' a solução corrente. Nesta proposta, optou-se por trabalhar com soluções inviáveis. Desta forma, é utilizada uma estratégia de penalização na função objetivo (*value*) dos conflitos que aparecem na solução. Seja T uma árvore geradora de G, o valor da função objetivo para o AGMRC é dado por:

$$evalue(T) = \sum_{e \in T} \omega(e) + \Delta |(T \times T) \cap \hat{E}| \tag{3.1}$$

Onde Δ é a penalidade dos conflitos na FO e ω é função de custo das arestas. O Δ foi escolhido de forma à garantir que nenhuma solução livre de conflitos tivesse uma avaliação pior que alguma solução com conflitos. A memória adaptativa é representada pelo vetor ρ e é explicada na Seção 3.2. A criação da solução através do método construtivo é detalhada na Seção 3.3, já a busca local é apresentada na Seção 3.4. Por fim, os detalhes de implementação são abordados na Seção 3.5.

A seguir, serão descritas as atividade executadas em cada linha do algoritmo proposto. Primeiramente, as linhas 2 e 3 calculam as variáveis que são utilizadas nos pesos da memória adaptativa. A linha 4 produz uma solução inicial completamente aleatória (algoritmo de Kruskal). A memória adaptativa é inicializada na linha 7 com um vetor unitário de dimensão m. A linha 8 representa a condição de parada. O *loop* principal se inicia com a criação de uma solução gulosa aleatória na linha 9, em seguida a busca local é executada na linha 10 e a melhor solução atualizada na linha 12. Por fim, os pesos da memória adaptativa são calculados na linha 13 e a memória atualizada na linha 15.

Algoritmo 3.1 Pseudo-código do GRASP com memória adaptativa

1: **função** GRASPMA(grafo $G = (V,E)$, grafo de conflitos $\hat{G} = (E,\hat{E})$)
2: $\beta \leftarrow |E|/10$ ▷ β e ε, variáveis auxiliares da memória adaptativa
3: $\varepsilon \leftarrow (|\hat{E}|/|V|)/5$
4: $T' \leftarrow$ `RandomSolution`$(G,\hat{G})$
5: $T' \leftarrow$ `localSearch`$(T',G,\hat{G})$
6: $T_b \leftarrow T'$
7: $\rho[e] \leftarrow 1 \quad \forall e \in E$ ▷ Memória adaptativa, ver Seção 3.2
8: **enquanto** Stop condition **faça**
9: $T' \leftarrow$ `RandomGreedySolution`$(G,\hat{G},\rho)$ ▷ Seção 3.3
10: $T' \leftarrow$ `localSearch`$(T',G,\hat{G})$ ▷ Seção 3.4
11: **se** $value(T') < value(T_b)$ **então**
12: $T_b \leftarrow T'$
13: $\beta \leftarrow \beta + \varepsilon$
14: **fim se**
15: $\rho \leftarrow update(\rho,\beta)$ ▷ Atualização da memória adaptativa
16: **fim enquanto**
17: **retorne** T_b
18: **fim função**

3.2 Memória Adaptativa

A memória adaptativa tem como objetivo guardar informação sobre cada elemento que pode compor uma solução. Com essas informações em mãos, o algoritmo pode a cada iteração construir uma nova solução com base no conhecimento prévio. No contexto do AGMRC, cada solução S corresponde a um conjunto de arestas, assim, a memória adaptativa irá associar um peso a cada aresta de acordo com sua utilização nas iterações passadas do algoritmo. Inicialmente, todas as arestas tem as mesmas chances, mas a partir de algumas iterações isso muda. Em outras palavras, arestas que fizeram parte de boas soluções terão pesos maiores que arestas que só apareceram em soluções ruins.

No Algoritmo 3.1, a memória adaptativa é representada pelo vetor ρ. Para calcular os pesos atribuídos a ρ são utilizadas variáveis auxiliares β e ε. Tais variáveis são utilizadas para controlar a reconstrução da nova solução a cada iteração, e são calculadas com os dados de entrada e atualizados durante a execução. A variável β representa os pesos atribuídos às arestas presentes em uma nova melhor solução, enquanto que ε é a taxa de crescimento de β durante o algoritmo.

Durante a execução do GRASP, o vetor ρ precisa ser atualizado (função *update* no Algoritmo 3.1), além dele, a variável β também sofre atualização durante a execução. A atualização funciona da seguinte forma, inicialmente $\rho[i] \leftarrow 1$, para todo $e_i \in E$. Em seguida, no decorrer

do algoritmo, as arestas que aparecem na solução corrente passam a ter mais chances de aparecerem nas soluções futuras. Uma solução simples seria incrementar em uma unidade o peso de todas presentes nessa solução, ou seja, se $e \in T'$ então $\rho[e] \leftarrow \rho[e] + 1$. Entretanto, essa atualização não consegue ser eficiente em todas as instâncias, pois se o GRASP passar muitas iterações preso em um ótimo local e depois encontrar uma solução melhor, o vetor ρ fará com que as próximas iterações produzam soluções semelhantes à solução antiga e não à nova.

Para tratar deste problema — já que o objetivo é obter soluções mais semelhantes a T_b —, toda vez que T_b for atualizado, todas as entradas de ρ serão fixadas com o valor 1, com exceção das entradas referentes às arestas que aparecem em T_b que receberão o valor β. A medida que T_b é atualizado, o fator de atualização de pesos β também se torna mais agressivo como mostrado na linha 13 do pseudocódigo do GRASP. Vale lembrar que essa atualização mais intensa só é ativada quando T_b é atualizada. Nas iterações em que T_b não for atualizada, as arestas que aparecem naquela solução serão atualizadas com um fator de atualização menor, apenas uma unidade será incrementada nas posições da memória referente a solução corrente.

3.3 Método Construtivo

Na Seção 2.2.4 foi apresentada uma explicação do funcionamento do construtivo do GRASP, também foi descrito funções de enviesamento que podem ser utilizadas para não permitir que a escolha do candidato a entrar na solução seja completamente aleatória.

Normalmente, todos candidatos na RCL possuem a mesma probabilidade de ser escolhidos. Bresina [8] propôs um método de enviesamento baseado na ordenação dos candidatos de acordo com o critério guloso, em outras palavras, o i-ésimo melhor candidato teria como enviesamento uma função de i.

Neste trabalho, diferente da estratégia de Bresina [8], o enviesamento não será calculado em função do ordenamento, e sim, em função da memória adaptativa. Com relação ao tamanho da RCL, será utilizado um valor $\alpha = 1$, ou seja, todos os elementos farão parte da RCL, no entanto, tal estratégia não será completamente aleatória por conta do enviesamento utilizado.

Para calcular o enviesamento, o método construtivo aleatório do GRASP proposto utiliza o conhecimento da memória adaptativa para gerar uma nova solução a cada iteração. A memória adaptativa implementa uma função de enviesamento para escolha das arestas. Seja e_j uma aresta do grafo G, e $\rho(e_j)$ o peso da memória contabilizado à aresta e_j. A probabilidade de e_j aparecer na solução é dada por:

$$P(e_j) = \frac{\rho(e_j)}{\sum_{i=1}^{|E|} \rho(e_i)} \tag{3.2}$$

Com as probabilidades calculadas, a construção de uma nova solução é iniciada. Para isso, foi utilizada uma variação aleatória do algoritmo de *Kruskal*, em que as arestas são escolhidas em um método de roleta de acordo com suas probabilidades. Note que, a primeira solução

construída na linha 4 do Algoritmo 3.1 é uma exceção à construção gulosa aleatória. Essa solução é construída de forma totalmente aleatória.

Dessa forma, a memória adaptativa do GRASP aqui proposto entra no algoritmo para controlar o construtivo. Essa é a real diferença com o GRASP clássico.

3.4 Busca Local

A busca local (BL) utilizada neste trabalho foi proposta por Barbosa & Delbem [2] e é denominada *BL4ex*. Ela consiste em trocar as arestas da solução corrente duas a duas por outras arestas que não estão na solução e que reconectam a árvore sem formar ciclos. Dada uma árvore geradora T de G, a estrutura de vizinhança é definida como:

$$N(T) = \{(T \setminus \{e_i, e_j\}) \cup \{e_k, e_l\} \mid e_i, e_j \in T \text{ e } e_k, e_l \in E \setminus T\}.$$

A busca local, como explicado na Seção 2.2.2, pode ser utilizada de mais de uma forma diferente quanto aos critérios de aceitação. Neste trabalho foi utilizado a busca local do tipo *first improvement*.

Outra vizinhança, proposta por Zhang et al. [42], faz a troca de uma única aresta por vez, foi testada, porém, o seu uso, a não ser nas instâncias menores, faz com que a heurística fique presa facilmente em ótimos locais. Dessa forma, optou-se por trabalhar apenas com a BL4ex, que apesar de ser mais pesada, apresenta resultados muito melhores. Além disso, na prática, é possível implementá-la de forma eficiente. A BL4ex tem complexidade $O(n^2m^2)$.

3.5 Detalhes de Implementação

Como a BL4ex é um pouco pesada, algumas técnicas se fazem necessárias para fazer com que seja possível acelerar o algoritmo.

3.5.1 Cálculo da Função Objetivo

Em Barbosa & Delbem [2], os autores explicam que para acelerar a execução da BL4ex, antes de testar a combinação das arestas para reconectar a árvore, eles dividem as arestas em três grupos, essa parte do código será chamada de fase de separação. Ao remover duas arestas da solução corrente T, serão formadas três componentes conexas V_A, V_B e V_C. Com isso, considerando o conjunto das arestas que não produzem ciclo, pode-se particionar este conjunto nas partes E_{AB}, E_{AC} e E_{BC}, de acordo com as componentes conexas que as arestas conectam, assim, as arestas de E_{AB} são as arestas viávesi que conectam V_A a V_B; as arestas de E_{AC} conectam V_A a V_C e E_{BC} conectam V_B a V_C.

Tal particionamento realmente faz muita diferença, porém o detalhe que mais impactou no tempo computacional foi a exclusão das arestas com custo maior do que a soma dos custos das duas arestas já retiradas da solução. E para fazer isso, não é necessário um novo pré-processamento, basta excluir as arestas na fase de separação, lembrando que o custo na verdade é o custo da aresta mais as penalidades causadas pela adição à solução.

Para calcular a função objetivo (FO) é utilizada a forma incremental descrita por Zhang et al. [42]. Com isso, durante a verificação das novas arestas não é preciso calcular o valor da FO, basta computar o acréscimo das duas novas arestas e só no final computar o valor completo da FO. Para fazer isso rapidamente, é necessário manter uma estrutura auxiliar que guarda as informações da solução corrente. A estrutura precisa manter o número de conflitos de cada aresta na solução corrente, atualizando-as a cada modificação. Assim, a cada iteração da BL4ex é possível verificar em $O(1)$ se uma aresta causa conflitos na solução e o custo de adicioná-la na solução.

Vale ressaltar que as informações da solução corrente não são restritas apenas às arestas que estão na solução. As arestas que estão fora da solução também guardam as informações da solução. Ou seja, se uma aresta $e \notin T'$, é possível verificar rapidamente quantos conflitos essa aresta causa na solução, caso seja adicionada, e quais arestas que estão na solução são conflitantes com ela.

3.5.2 Estrutura de Dados Auxiliar

Para que as operações sejam executadas com uma menor complexidade, algumas estruturas auxiliares foram adicionadas ao algoritmo. Seja T a solução corrente do algoritmo, as estruturas guardam as seguintes informações:

- `numero_conflitos`[e]: vetor que guarda a quantidade de conflitos que a aresta e possui com as arestas em T, é o mesmo que a vizinhança de e no grafo de conflitos induzido por T;
- `solucao`[e]: vetor binário que permite verificar em $O(1)$ se uma aresta está ou não na solução;
- `arvore`: grafo da solução corrente representado com lista de adjacência. É utilizado para fazer a busca em profundidade.

Todas essa informações mantidas sobre as arestas precisam ser atualizada a cada inserção ou remoção.

Estrutura auxiliar: A estrutura auxiliar que permite essas trocas serem feitas rapidamente e acelera o cálculo da função objetivo é na verdade um subgrafo de $\hat{G}$, essa estrutura será chamada de $\mathcal{C}$. Nesse subgrafo as informações são inseridas o removidas durante toda execução. Sendo assim, a atualização é da ordem de $O(m)$, pois uma aresta pode ter no máximo $(m-1)$ conflitos,

mas na prática esse número é bem menor, pois se uma aresta tiver mais de $(m-n+2)$ conflitos ela não pode fazer parte de nenhuma solução viável.

Inserção: Quando uma nova aresta é inserida na solução a lista de conflitos da aresta em $\hat{G}$ é acessada. Então, na estrutura auxiliar $\mathcal{C}$, essas arestas serão atualizadas com o número de conflitos sendo incrementado (`numero_conflito`[e]++). A única informação atualizada sobre a aresta inserida é que ela passa à pertencer a solução (`solucao`[e]=1), suas arestas conflitantes que serão realmente atualizadas. A estrutura `arvore` também precisa ser atualizada. Para esta, basta apenas inserir a aresta.

Remoção: A atualização na remoção é análoga, quando uma aresta é removida da solução, todas as arestas conflitantes do grafo original terão seu número de conflitos decrementado. O vetor binário recebe 0 na posição referente à aresta e na `arvore` será removida a aresta.

A Figura 3.1 exemplifica como essa estrutura se comporta, nesse exemplo os conflitos que aparecem são os conflitos referentes a AGM obtido pelo algoritmo de Kruskal da Figura 1.2. Note que uma aresta (conflito) só está na estrutura se pelo menos uma aresta do conflito está na solução, isso quer dizer que o conjunto de aresta considerado por essa estrutura são justamente as arestas que são cobertas pelos vértices da solução (vértice do grafo de conflitos é uma aresta em G). A solução corrente foi marcada pelos vértices em vermelho, esta é a condição a ser verificada para saber se uma aresta faz parte da solução. Arestas que ligam dois vértices vermelhos são conflitos que aparecem na solução, já a vizinhança de um vértice (u,v) em $\mathcal{C}$ é o conjunto de arestas na solução que são conflitantes com (u,v). Para isso os conflitos são orientados, a aresta (b,c), por exemplo, tem a aresta (d,f) como conflito, mas o contrário não acontece, isso porque (d,f) está na solução e (b,c) não está. No caso das duas arestas estarem na solução o conflito tem os dois sentidos. Se, para este exemplo, fosse considerado o grafo de conflitos referente as solução do AGMRC da Figura 1.3 ao invés do grafo da Figura 1.2 não haveriam arestas entre vértices vermelhos. Além disso não existem arestas entre dois vértices que não estão na solução.

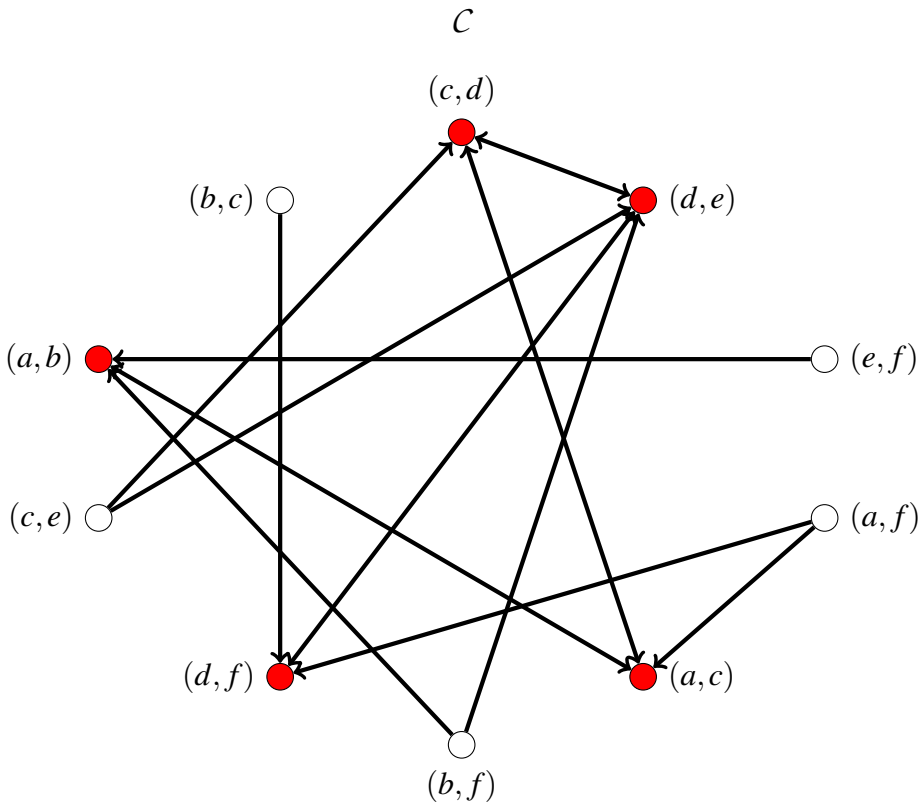


Figura 3.1: Exemplo da estrutura de conflitos

3.5.3 Garantia de Soluções Acíclicas

Ao reconstruir a solução durante o processo de busca local, é necessária a garantia de que as soluções sejam acíclicas. Existe a possibilidade de utilizar a estrutura *union-find* — normalmente implementada no algoritmo de Kruskal. Uma explicação do funcionamento desta estrutura pode ser encontrada em Cormen et al. [14]. No entanto, mesmo a estrutura *union-find* sendo muito rápida, em instâncias de grande porte essa estratégia se torna lenta, já que a cada iteração tudo precisa ser reconstruído, tornando assim, todo o processo lento. Isso justifica a utilização da estratégia abordada em Barbosa & Delbem [2]. Trata-se de: definir dois atributos para cada vértice; a partir de um vértice inicial, fazer uma busca em profundidade (DFS, *deep first search*); e marcar o tempo que cada vértice foi alcançado juntamente com o tempo que a busca fechou o vértice. O tempo que um vértice foi alcançado é o tempo que um vértice foi alcançado na DFS. Já o tempo que a DFS termina de visitar todos os vizinhos de um vértice é o tempo de fechamento.

Com isso, considere que a aresta $e = (u, v)$ foi removida da solução e seja $d(x)$ o tempo de descoberta do vértice x e $f(x)$ o seu tempo de fechamento. Para saber se x é descendente de u, basta verificar se $d(x) > d(u)$ e $f(x) < f(u)$. Por fim, a aresta $e = (x, y)$ reconectará a solução se x e y não possuírem descendentes em comum, ou seja, um será descendente de u e o outro não. Analogamente, esta verificação é estendida para a remoção de duas arestas.

Na solução proposta, a verificação de ciclo foi aprimorada. Foi utilizada a DFS depois de ser retirada uma aresta da solução. Em seguida, é executada a DFS a partir de cada um dos extremos da aresta retirada, marcando cada vértice descoberto com uma *flag* referente ao vértice em que a busca se iniciou. Assim, para verificar se $e = (x, y)$ reconecta a solução basta testar se $flag(x) \neq flag(y)$, isso quer dizer que x e y estão em subárvores diferentes.

Imaginando a remoção de apenas uma aresta, a Figura 3.2 exemplifica a execução da DFS a partir dos vértices u e v, onde (u, v) foi a aresta removida da árvore. Neste exemplo, cada cor marca uma *flag* diferente, assim, dois vértices da mesma cor estão na mesma componente conexa.

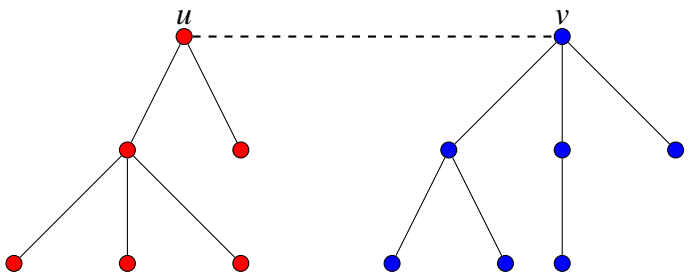


Figura 3.2: Marcação dos vértices com a DFS

Note que qualquer aresta com dois extremos da mesma cor formam um ciclo, já se a aresta possui cores diferentes é elegível a entrar na solução.

Para executar a DFS uma outra estrutura é mantida em paralelo, uma árvore. Até esse ponto, tudo pode ser executado apenas com os índices das arestas. Para não montar a árvore em toda execução da DFS basta manter essa aŕvore e realizar toda modificação na solução também nessa árvore. Assim, essa atualização pode ser executada junto com a atualização do subgrafo de $\hat{G}$.

Capítulo 4

Resultados Computacionais

Neste capítulo, os resultados computacionais referentes ao método GRASP enviesado com memória adaptativa são apresentados. Primeiramente, o GRASP proposto é comparado com a abordagem clássica do GRASP e com GRASP enviesado com uma função linear. Em seguida, o GRASP enviesado com memória adaptativa é comparado com as abordagens encontradas na literatura.

4.1 Ferramentas

Todas os métodos desenvolvidos foram implementados em C e executado em um computador com sistema operacional Ubuntu 18.04.02, com arquitetura de 64 bits, 16 GB de RAM e processador Intel Core i7-4770 3.40GHz.

Para realizar os testes e comparar com outros algoritmos da literatura, foram utilizadas as instâncias geradas por Zhang et al. [42]. Tais instâncias são divididas em dois grupos denominados `type1` e `type2`. As instâncias do grupo `type1` são difíceis de serem resolvidas, neste grupo, várias instâncias se mostraram inviáveis, outras apresentam soluções viáveis mas não se conhece o valor ótimo. Ainda neste grupo, apenas 8 instâncias possuem otimalidade comprovada, por outro lado, existe um subconjunto que não se sabe se existe solução viável. Em contrapartida, o grupo `type2` é formado por instâncias fáceis com todos os ótimos conhecidos. Para essas instâncias, as meta-heurísticas não têm dificuldade em resolvê-las. Por este motivo, os resultados apresentados aqui foram restringidos apenas para instâncias difíceis (`type1`) com viabilidade comprovada. Teste computacionais nas instâncias fáceis não acrescentam muitas informações, da mesma forma que a execução de testes em instâncias em que não é atingida uma solução viável, entretanto os testes foram executados nas instâncias com viabilidades desconhecida. Mas como os algoritmos da literatura, nenhuma solução viável foi encontrada, dessa forma os resultados dessas instâncias seriam apenas tabelas com tempo de execução. Além disso, mesmo não tendo sido encontrada solução viável para algumas instâncias do grupo type1, todas com viabilidade desconhecida passam no teste de inviabilidade do Lema 1.

A Tabela 4.1 mostra as instâncias `type1` utilizadas neste trabalho. A coluna `Instância` identifica a instância, a colunas $|V|$, $|E|$ e $|\hat{E}|$ apresentam o número de vértices, o número de arestas e o número de conflitos, respectivamente. A coluna L_i apresenta os limites inferiores (obtidos pelos B&C de Samer & Urrutia [36, 37] e de Bittencourt et al. [6]) e os limites superiores obtidos pelas heurísticas testadas são encontrados na coluna L_s. Note que, quando o valor dos limites são idênticos, significa que o valor é ótimo.

Instância	$\|V\|$	$\|E\|$	$\|\hat{E}\|$	L_i	L_s
I_1	50	200	199	**708**	**708**
I_2	50	200	398	**770**	**770**
I_3	50	200	597	**917**	**917**
I_4	50	200	995	**1324**	**1324**
I_5	100	300	448	**4041**	**4041**
I_6	100	300	897	**5658**	**5658**
I_7	100	500	1247	**4275**	**4275**
I_8	100	500	2495	**5997**	**5997**
I_9	100	500	3741	6707,8	7523
I_{10}	200	600	1797	13264	13815
I_{11}	200	800	3196	20941,5	21480

Tabela 4.1: Instâncias `type1`

4.2 GRASP Enviesado com MA × GRASP Tradicional

Para mostrar que utilizar o enviesamento de acordo com a execução da busca local é melhor que utilizar o GRASP tradicional descrito na Seção 2.2.4, foi feita uma comparação entre estes tipos de GRASP. O GRASP desenvolvido neste trabalho será chamado aqui de GRASP com MA e para diferenciar dos GRASPs com LRC, os que utilizarem LRC serão identificados com o valor do α utilizado pra controlar o construtivo.

Como já foi explicado, o α controla o quão guloso ou aleatório o GRASP será. Os resultados obtidos pelos diferentes valores de α são apresentados na Tabela 4.2.

I	GRASP com MA			GRASP–0,3			GRASP–0,6			GRASP–0,9		
	Média	#best	GAP	Média	#best	GAP	Média	#best	GAP	Média	#best	GAP
I_1	708	30	0,00%	708	30	0,00%	708	30	0,00%	708	30	0,00%
I_2	770	30	0,00%	770	30	0,00%	770	30	0,00%	770,1	29	0,00%
I_3	917	30	0,00%	935,7	4	0,00%	944,8	5	0,00%	943,2	2	0,00%
I_4	1324	30	0,00%	1343	3	0,00%	1344,4	3	0,00%	1337,7	10	0,00%
I_5	4041	30	0,00%	4046,5	3	0,00%	4048,6	3	0,00%	4049	4	0,00%
I_6	5706,2	5	0,00%	5768	0	0,31%	5783,1	0	0,44%	5771,6	0	0,44%
I_7	4275	30	0,00%	4279	14	0,00%	4278,9	11	0,00%	4281,1	7	0,00%
I_8	6043,9	3	0,00%	6350,6	0	3,11%	6385,9	0	3,36%	6370,2	0	2,90%
I_9	7631,5	7	0,00%	8389,7	0	4,65%	8494,3	0	7,56%	8449,9	0	7,13%
I_{10}	14144,2	1	0,00%	14966,9	0	4,08%	14926,5	0	4,56%	14902,8	0	4,67%
I_{11}	21721,4	2	0,00%	23164,9	0	5,04%	23159	0	4,64%	23158,4	0	5,91%

Tabela 4.2: Comparação com GRASP tradicional

Na Tabela 4.2 é apresentada a média das 30 execuções (coluna média), a quantidade de vezes que o melhor valor conhecido foi obtido (coluna #best) e o GAP do melhor valor obtido nas 30 execuções (coluna GAP), o GAP considera o melhor resultado da literatura. O GAP é calculado como:

$$\text{GAP} = 100 \times \frac{value(x) - value(L_s)}{value(L_s)} \tag{4.1}$$

A coluna GRASP equivale ao algoritmo proposto neste trabalho (enviesado com memória adaptativa) e as outras equivalem ao GRASP tradicional com diferentes valores para α.

Fica claro nessa tabela que o enviesamento é muito mais eficaz que a LRC mesmo que a taxa de aleatoriedade varie muito. Para nenhum valor de α o GRASP conseguiu encontrar todos os *bests*, em 5 instâncias não encontra nenhuma vez, e nas que encontra, com exceção das instâncias I_1 e I_2, encontra poucas vezes, além disso, a média das 30 execuções é bem mais alta. Esse problema não aparece na coluna do GRASP enviesado com MA. Na Tabela 4.2 os algoritmos fora executados 30 vezes, fazendo $|E|$ iterações cada execução e usando sementes de 1 a 30 para gerar números aleatórios. A única diferença entre os GRASPs é o construtivo, todo o resto do algoritmo executa de forma idêntica.

Instância	GRASP com MA	GRASP–0,3	GRASP–0,6	GRASP–0,9
I_1	0,182	0,226	0,229	0,231
I_2	0,186	0,23	0,235	0,235
I_3	0,183	0,243	0,246	0,246
I_4	0,179	0,235	0,243	0,243
I_5	1,723	2,256	2,256	2,277
I_6	1,533	2,369	2,389	2,398
I_7	4,321	5,293	4,905	5,311
I_8	3,946	5,278	5,209	5,301
I_9	3,784	5,361	5,366	5,365
I_{10}	27,28	42,807	42,809	43,071
I_{11}	44,166	63,142	63,274	63,297

Tabela 4.3: Comparação do tempo médio dos GRASPs com LRC

Apesar de serem muito semelhantes, nas duas últimas instâncias existe uma diferença significativa no tempo médio de execução entre as diferentes versões do GRASP, *a versão que utiliza a memória adaptativa executa mais rápido em todas instâncias*. Isso ocorre, provavelmente, devido ao construtivo com MA ser um pouco mais sofisticado, essa diferença no tempo pode ser verificada na Tabela 4.3. Para montar a LRC, é preciso avaliar todas as arestas em cada iteração do construtivo, essa avaliação leva um certo tempo. Já o construtivo com memória adaptativa não avalia arestas, apenas escolhe, independente do custo ou dos conflitos.

4.3 GRASP Enviesado com MA × GRASP Enviesado com Função Linear

Além das formas tradicionais de montar o GRASP com LRC, existem variantes que utilizam uma função de enviesamento para escolher as arestas [8], como descrito na Seção 2.2.4, Nesse caso foi escolhido uma função linear para realizar a comparação em relação a memória adaptativa. Se a LRC está ordenada decrescentemente da forma $LRC = \{e_1, e_2, \ldots, e_k\}$, para simular a função linear basta utilizar um vetor de probabilidades por ranqueamento de arestas (*bias*), onde $bias(e_i) = i$. Assim a probabilidade da aresta e_i aparecer na solução do construtivo em uma determinada iteração é dada por:

$$P(e_i) = \frac{i}{\sum_{j=1}^{k} j} \tag{4.2}$$

Se e_k é a aresta mais bem avaliada em uma dada iteração, então e_k possui a maior probabilidade de ser sorteada. A Tabela 4.4 mostra os resultados obtidos pelo GRASP enviesado com memória adaptativa e o GRASP com enviesamento linear.

Instância	GRASP com viés linear				GRASP enviesado com MA			
	Média	Best	GAP	Tempo	Média	Best	GAP	Tempo
I_1	708	708*	0,00%	0,21	708	708*	0,00%	0,182
I_2	770	770*	0,00%	0,217	770	770*	0,00%	0,186
I_3	942,1	917*	0,00%	0,234	917	917*	0,00%	0,183
I_4	1346	1324*	0,00%	0,229	1324	1324*	0,00%	0,179
I_5	4046,8	4041*	0,00%	2,011	4041	4041*	0,00%	1,723
I_6	5778,4	5658	0,00%	2,124	5706,2	5658*	0,00%	1,533
I_7	4281,4	4275*	0,00%	5,225	4275	4275*	0,00%	4,321
I_8	6378,6	6223	3,76%	5,2	6043,9	5997*	0,00%	3,946
I_9	8469,9	7945	5,60%	5,276	7631,5	7523	0,00%	3,784
I_{10}	14868,9	14239	3,06%	40,056	14144,2	13815	0,00%	27,28
I_{11}	23314,7	22556	5,00%	62,105	21721,4	21480	0,00%	44,166

Tabela 4.4: Comparação com viés linear

É possível notar na Tabela 4.4 que o enviesamento linear também não é uma forma eficiente para substituir o construtivo do GRASP. Os resultados apresentam um GAP muito mais alto e não encontram os *bests* em várias instâncias. As condições de comparação foram as mesmas do GRASP com LRC tradicional apresentada na Tabela 4.2 e mais uma vez, além de piorar os resultados, o tempo médio de execução também aumenta pois o construtivo com enviesamento linear necessita da avaliação e ordenação das arestas a cada iteração.

A seguir, serão apresentados os resultados dos outros experimentos computacionais. Foram realizadas comparações do GRASP proposto com os métodos da literatura: ILS e MEGA (Multiethnic Genetic Algorithm) e com um GRASP tradiconal.

4.4 GRASP Enviesado com MA × ILS

Nesta seção, o método proposto é comparado com o ILS de Barbosa & Delbem [2]. Ao conhecimento dos autores, essa foi a proposta algorítmica com melhores resultados na literatura. Os autores de Barbosa & Delbem [2], gentilmente, forneceram o código fonte com duas versões do ILS, no entanto, apenas a melhor versão será comparada.

Os dois algoritmos foram executados na mesma máquina e utilizando os mesmos critérios de parada. Cada algoritmo foi executado 30 vezes com limite de $m = |E|$ iterações. Entretanto, não foi possível obter os mesmos resultados divulgados pelos autores, já eles utilizaram sementes aleatórias na geração dos números aleatórios. Para facilitar a reprodução dos resultados, em ambos os algoritmos foram executados testes com as sementes inteiras de 1 até 30, uma para cada execução de cada instância.

Instância	ILS					GRASP com MA				
	Média	Best	GAP	#best	Tempo (s)	Média	Best	GAP	#best	Tempo (s)
I_1	**708**	**708***	0,00%	30	0,367	**708**	**708***	0,00%	30	0,182
I_2	**770**	**770***	0,00%	30	0,372	**770**	**770***	0,00%	30	0,186
I_3	919,3	**917***	0,00%	16	0,363	**917**	**917***	0,00%	30	0,183
I_4	1326,5	**1324***	0,00%	22	0,353	**1324**	**1324***	0,00%	30	0,179
I_5	4041,4	**4041***	0,00%	27	2,544	**4041**	**4041***	0,00%	30	1,723
I_6	5672,2	**5658***	0,00%	2	1,89	5706,2	**5658***	0,00%	5	1,533
I_7	4275,4	**4275***	0,00%	24	6,605	**4275**	**4275***	0,00%	30	4,321
I_8	6026	**5997***	0,00%	2	6,266	6043,9	**5997***	0,00%	3	3,946
I_9	7550,3	**7523**	0,00%	7	5,611	7631,5	**7523**	0,00%	7	3,784
I_{10}	13924,7	13820	0,04 %	0	22,338	14144,2	**13815**	0,00%	1	27,28
I_{11}	21571,8	**21480**	0,00%	1	46,964	21721,4	**21480**	0,00%	2	44,166

Tabela 4.5: Comparação do ILS com o GRASP com MA

Nos testes computacionais, o ILS não alcançou o melhor valor conhecido na instância I_{10}, mas nos resultados apresentados por Barbosa & Delbem [2] esse valor é alcançado. Para cada algoritmo, a coluna `Média` equivale à média dos custos obtidos nas 30 execuções, `Best` ao melhor valor, `GAP` ao *gap* da melhor solução para o melhor valor conhecido, `#best` à quantidade de vezes que o melhor valor conhecido foi obtido e `Tempo` ao tempo médio em segundos. O valores em negrito são os melhores valores já encontrados, e os marcados com (*) são valores ótimos.

De acordo com estes resultados, fica claro que o GRASP consegue chegar mais vezes nas melhores soluções, inclusive, em 6 instâncias encontra o ótimo nas 30 execuções enquanto o ILS faz isso apenas em 2 instâncias. Vale destacar que em uma instância o ILS não chega no melhor valor conhecido. Com relação ao tempo médio de cada algoritmo, os resultados da Tabela 4.6 mostram que o GRASP executa mais rapidamente que o ILS, com exceção em apenas uma instância.

Outros experimentos foram realizados por meio de gráficos `TTTPlot` [1], com estes gráficos, é possível verificar a probabilidade que cada algoritmo possui de alcançar alvos em função do tempo de execução. Neste trabalho, os alvos considerados foram os limites superiores. Dessa

Instância	ILS	GRASP com MA
I_1	0,367	0,182
I_2	0,372	0,186
I_3	0,363	0,183
I_4	0,353	0,179
I_5	2,544	1,723
I_6	1,89	1,533
I_7	6,605	4,321
I_8	6,266	3,946
I_9	5,611	3,784
I_{10}	22,338	27,28
I_{11}	46,964	44,166

Tabela 4.6: Comparação do tempo de execução médio do ILS com o GRASP com MA

forma, cada algoritmo foi executado 100 vezes nas instâncias I_1 e I_5 com critério de parar ao alcançar o alvo. O resultado pode ser verificado na Figura 4.1. Pode-se observar que o GRASP com MA apresenta uma convergência mais rápida aos alvos que o ILS.

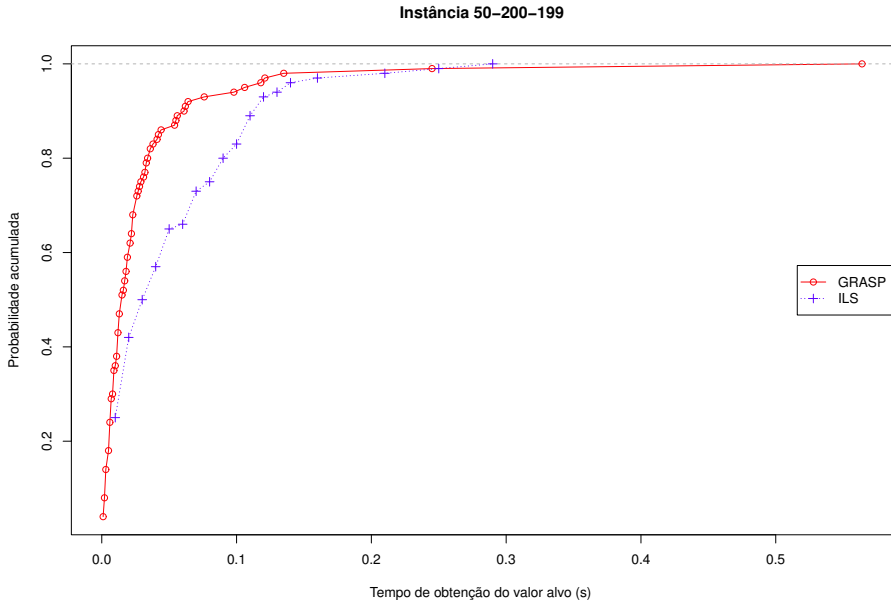
4.5 GRASP Enviedado com MA × MEGA

Carrabs et al. [11] desenvolveram um algoritmo genético denominado MEGA. Como o código-fonte não foi disponibilizado, não foi possível executar os testes sob as mesmas condições. A condição de parada utilizado no MEGA é bem diferente, já que o algoritmo utiliza mais de uma busca local e o número de iterações do MEGA pode variar de acordo com a busca local utilizada. Além disso, o número de iterações é decidido em tempo de execução, causando uma grande variabilidade nos tempos computacionais em cada execução.

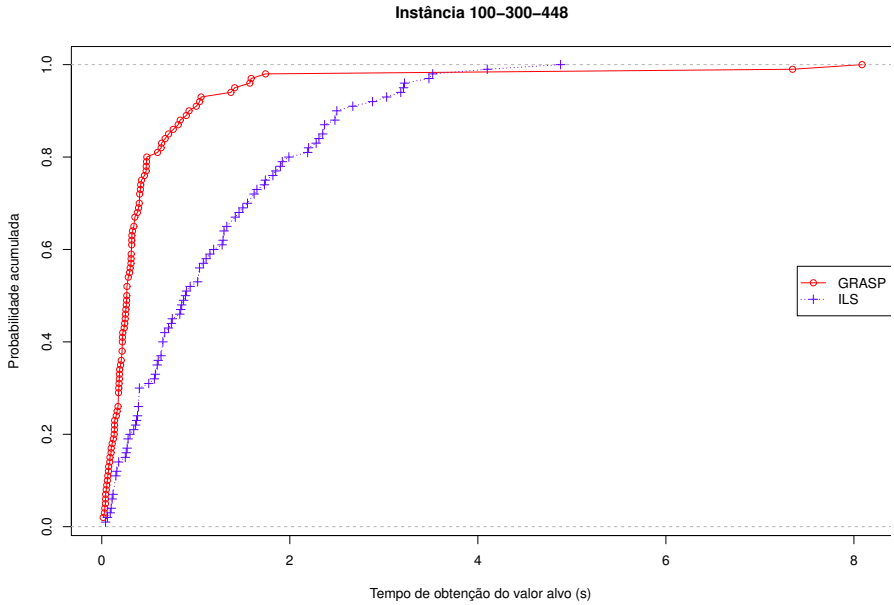
Instância	MEGA				GRASP com MA				
	Média	Best	GAP	Tempo (s)	Média	Best	GAP	Tempo (s)	Pior
I_1	708	**708***	0,00%	0,71	708	**708***	0,00%	0,182	708
I_2	770	**770***	0,00%	0,68	770	**770***	0,00%	0,186	770
I_3	917	**917***	0,00%	0,63	917	**917***	0,00%	0,183	917
I_4	1365,4	1336	0,91%	0,66	1324	**1324***	0,00%	0,179	1324
I_5	4099,2	4088	1,16%	2,39	4041	**4041***	0,00%	1,723	4041
I_6	-	6095	7,72%	1,81	5706,2	**5658***	0,00%	1,533	6091
I_7	4291,2	**4275***	0,00%	5,18	4275	**4275***	0,00%	4,321	4275
I_8	6325	6199	3,37%	5,12	6043,9	**5997***	0,00%	3,946	6131
I_9	7788	7665	1,89%	3,72	7631,5	**7523**	0,00%	3,784	8026
I_{10}	-	15029	8,79%	12,23	14144,2	**13815**	0,00%	27,28	14892
I_{11}	22350,8	22110	2,93%	23,42	21721,4	**21480**	0,00%	44,166	23220

Tabela 4.7: Comparação do MEGA com o GRASP utilizando MA

Sendo assim, não foi realizado outra bateria de testes, os mesmo valores do teste anterior são comparados com os valores reportados no artigo do MEGA. Mesmo com essas diferenças é possível notar que o GRASP com MA tem melhor performance como mostra a Tabela 4.7.



(a) Instância I_1



(b) Instância I_5

Figura 4.1: Gráfico *Time to Target* das instâncias I_1 e I_5

Neste experimento, o MEGA foi executado apenas 5 vezes (resultados divulgados pelos autores) enquanto o GRASP foi executado 30 vezes. Os *hardwares* utilizados possuem configurações bem parecidas. Considerando o tempo médio de execução de cada algoritmo, o GRASP com MA executa por mais tempo em duas instâncias. O MEGA não chega nos melhores valores da literatura em várias instâncias e nas instâncias I_6 e I_{10} a ausência da média indica que provavelmente os algoritmo só encontrou solução viável em uma das 5 execuções, isso já mostra uma grande deficiência em relação ao GRASP com MA que em 30 execuções nunca retornou uma solução inviável. Os valores médios também são bem melhores na coluna do GRASP. Outro fator em favor do GRASP com MA está na coluna `Pior` do GRASP, esta coluna informa o pior valor da função objetivo das 30 execuções, nela é possível notar que a pior execução do GRASP é melhor ou igual a melhor execução do MEGA na maioria das instâncias, apenas em duas instâncias o *best* do MEGA é melhor que o pior valor encontrado pelo GRASP com MA. Com isso, fica evidente a superioridade do GRASP em relação ao MEGA. Acredita-se que o algoritmo genético (MEGA) teve um baixo desempenho pelo fato de que a combinação de duas boas soluções no MEGA, provavelmente, pode gerar filhos ruins.

Capítulo 5

Conclusões e Trabalhos Futuros

O trabalho mostra as técnicas abordadas na resolução do problema da Árvore Geradora Mínima com Restrições de Conflitos. A principal proposta foi um algoritmo baseado na metaheurística GRASP utilizando novos componentes como a memória adaptativa.

Com relação aos métodos heurísticos, vale ressaltar que *gap* do melhor resultado foi 0% em todos os casos — considerando o resultado exato ou o melhor resultado da literatura.

O GRASP enviesado com memória adaptativa consegue ganhar de todos os algoritmos encontrados na literatura, e ainda tem um desempenho superior a versões clássicas do GRASP. Ganhando em tempo e, principalmente, em qualidade de solução. Isso mostra o quão promissor é utilizar a memória adaptativa, pois com a memória adaptativa foi possível projetar um construtivo bem mais leve e ainda com uma capacidade bem maior de gerar boas soluções. Além disso, ao conhecimento dos autores, esse trabalho é o primeiro trabalho que utiliza a memória adaptativa na função de enviesamento do GRASP.

Dentre as contribuições deste trabalhos podemos citar:

- algoritmo GRASP para resolver o problema da AGMRC;
- aprimoramento da busca local BLex4 da literatura;
- nova variante de GRASP enviesado com memória adaptativa;

Para trabalhos futuros, serão pesquisadas novas variantes para o problema, em que os algoritmos propostos poderão ser aplicados. Assim como outros problemas com restrições de conflito. Também é interessante considerar a aplicação da BL4ex em outros problema. Como foi mostrado, é possível implementá-la de forma bastante eficaz. Um outro ponto bastante interessante que será abordado é estudar o problema do ponto de vista teórico. Por fim, planeja-se estudar algoritmos híbridos com métodos exatos para o problema da AGMRC.

Bibliografia

[1] AIEX, R. M. ; RESENDE, M. G. C. ; RIBEIRO, C. C. TTT plots: a perl program to create time-to-target plots. *Optimization Letters* 1, 4 (2007), 355–366.

[2] BARBOSA, M. A. L. ; DELBEM, A. C. B. Uma Busca Local Iterada para o Problema da Árvore Geradora Mínima sob Restrições de Conflitos. *Anais do XLVIII Simpósio Brasileiro de Pesquisa Operacional* (2017), 2533–2544.

[3] BIANCHI, L. ; DORIGO, M. ; GAMBARDELLA, L. M. ; GUTJAHR, W. J. A survey on metaheuristics for stochastic combinatorial optimization. *Natural Computing* 8, 2 (2009), 239–287.

[4] BINATO, S. ; J. HERY, W. ; M. LOEWENSTERN, D. A Grasp for Job Shop Scheduling. *Essays and Surveys on Metaheuristics* 15 (2000), 217–278.

[5] BIRATTARI, M. ; PAQUETE, L. ; STÜZTLE, T. ; VARRENTRAPP, K. Classification of Metaheuristics and Design of Experiments for the Analysis of Components / Technical report, Darmstadt University of Technology, Germany. 2001 (AIDA-01-05). – Forschungsbericht. – 12 S. – Texto Técnico.

[6] BITTENCOURT, Y. B. ; CAMPÊLO, M. ; DIAS, F. C. S. Formulaçao MTZ para o Problema da Árvore Geradora Mınima sob Restriçoes de Conflito. *Anais do XLVIII Simpósio de Pesquisa Operacional* (2016), 2441–2448.

[7] BOSCHETTI, M. A. ; MANIEZZO, V. ; ROFFILLI, M. ; RÖHLER, A. B. Matheuristics: Optimization, simulation and control. In *International Workshop on Hybrid Metaheuristics*, (2009), pp. 171–177.

[8] BRESINA, J. L. Heuristic-biased stochastic sampling. In *AAAI/IAAI, Vol. 1*, (1996), pp. 271–278.

[9] CAPUA, R. ; FROTA, Y. ; OCHI, L. S. ; VIDAL, T. A study on exponential-size neighborhoods for the bin packing problem with conflicts. *Journal of Heuristics* 24, 4 (2018), 667–695.

[10] CAPUA, R. ; FROTA, Y. ; VIDAL, T. ; OCHI, L. S. Um algoritmo heurıstico para o problema de bin packing com conflitos. *Anais do XLVII Simpósio Brasileiro de Pesquisa Operacional* (2015), 4252–4261.

[11] CARRABS, F. ; CERRONE, C. ; PENTANGELO, R. A multiethnic genetic approach for the minimum conflict weighted spanning tree problem. *Networks* (2019), 134-141.

[12] CARRABS, F. ; CERULLI, R. ; PENTANGELO, R. ; RAICONI, A. Minimum spanning tree with conflicting edge pairs: a branch-and-cut approach. *Annals of Operations Research* (2018), 1–14.

[13] CHVATAL, V. Edmonds polytopes and a hierarchy of combinatorial problems. *Discrete mathematics* 4, 4 (1973), 305–337.

[14] CORMEN, T. H. ; LEISERSON, C. E. ; RIVEST, R. L. ; STEIN, C. *Algoritmos: Teoria e Prática*. 2. ed. Elsevier, 2002. – ISBN 85-352-0926-3.

[15] DALAL, Y. K. ; METCALFE, R. M. Reverse path forwarding of broadcast packets. *Communications of the ACM* 21, 12 (1978), 1040–1048.

[16] DARMANN, A. ; PFERSCHY, U. ; SCHAUER, J. Determining a minimum spanning tree with disjunctive constraints. In *International Conference on Algorithmic DecisionTheory*, (2009), pp. 414–423.

[17] DARMANN, A. ; PFERSCHY, U. ; SCHAUER, J. ; WOEGINGER, G. J. Paths, trees and matchings under disjunctive constraints. *Discrete Applied Mathematics* 159, 16 (2011), 1726–1735.

[18] DJAUHARI, M. A. ; GAN, S. L. Optimality problem of network topology in stocks market analysis. *Physica A: Statistical Mechanics and its Applications* 419 (2015), 108–114.

[19] EXPÓSITO, A. ; BRITO, J. ; MORENO, J. A. A Heuristic-Biased GRASP for the Team Orienteering Problem. In *Conference of the Spanish Association for Artificial Intelligence*, (2016), pp. 428–437.

[20] FEO, T. A. ; RESENDE, M. G. C. Greedy randomized adaptive search procedures. *Journal of Global Optimization* 6 (1995), 109–133.

[21] FEO, T. ; RESENDE, M. Greedy randomized adaptive search procedures. *Journal of Global Optimization* 6 (1995), 109–133.

[22] GENDREAU, M. ; POTVIN, J.-Y. *Handbook of Metaheuristics*. 2. ed. Springer Publishing Company, Incorporated, 2010. – ISBN 1441916636, 9781441916631.

[23] GOLDBARG, M. C. ; GOLDBARG, E. G. ; LUNA, H. P. L. *Otimização Combinatória e Meta-Heurísticas: Algoritmos e Aplicações*. 1. ed. Elsevier, 2016. – ISBN 978-85-352-7812-5.

[24] GOLDBERG, D. E. ; HOLLAND, J. H. Genetic Algorithms and Machine Learning. *Machine Learning* 3, 2 (1988), 95–99.

[25] GOMORY, R. E. An algorithm for integer solutions to linear programs. *Recent advances in mathematical programming* 64, 260-302 (1963), 14.

[26] GONÇALVES, L. B. ; OCHI, L. S. ; MARTINS, S. L. A grasp with adaptive memory for a period vehicle routing problem. In *International Conference on Computational Intelligence for Modelling, Control and Automation and International Conference on Intelligent Agents, Web Technologies and Internet Commerce (CIMCA-IAWTIC'06)* 1, (2005), pp. 721–727.

[27] HERTZ, A. ; WERRAde ,. Using tabu search techniques for graph coloring. *Computing* 39, 4 (1987), 345–351.

[28] KRUSKAL, J. B. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical society* 7, 1 (1956), 48–50.

[29] LOURENÇO, H. R. ; MARTIN, O. C. ; STUTZLE, T. Iterated Local Search: Framework and Applications. Version: 2010. http://dx.doi.org/10.1007/978-1-4419-1665-5_12. In GLOVER, F. (Hrsg.) ; KOCHENBERGER, G. A. (Hrsg.): *Handbook of Metaheuristics, 2nd Edition* v.146. Kluwer Academic Publishers, 2010. – DOI 10.1007/978–1–4419–1665–5_12, 363–397.

[30] MANTEGNA, R. N. Hierarchical structure in financial markets. *The European Physical Journal B-Condensed Matter and Complex Systems* 11, 1 (1999), 193–197.

[31] MOTTA, L. C. ; OCHI, L. S. Metaheurísticas com memória adaptativa para o problema de recobrimento de rotas. *IX Congresso Brasileiro de Redes Neurais* (2009), 5.

[32] NEVES, T. A. ; OCHI, L. S. GRASP com Memória Adaptativa na solução de um Problema de Roteamento de Veículos com Múltiplas Origens. *Anais do XXXVIII Simpósio Brasileiro de Pesquisa Operacional* (2009).

[33] OHLSSON, H. ; GUSTAFSSON, O. ; WANHAMMAR, L. Implementation of low complexity FIR filters using a minimum spanning tree. In *Proceedings of the 12th IEEE Mediterranean Electrotechnical Conference (IEEE Cat. No. 04CH37521)* v.1, (2004), pp. 261–264.

[34] PRIM, R. C. Shortest connection networks and some generalizations. *The Bell System Technical Journal* 36, 6 (1957), 1389–1401.

[35] RESENDE, M. G. ; RIBEIRO, C. C. *Optimization by GRASP: Greedy Randomized Adaptive Search Procedures*. 1. ed. Springer-Verlag New York, 2016. http://dx.doi.org/10.1007/978-1-4939-6530-4. http://dx.doi.org/10.1007/978-1-4939-6530-4. – ISBN 978–1–4939–6528–1,978–1–4939–6530–4.

[36] SAMER, P. ; URRUTIA, S. Um algoritmo de branch and cut para árvores geradoras mınimas sob restriçoes de conflito. *Anais do XLV Simpósio Brasileiro de Pesquisa Operacional* (2013), 2521–2532.

[37] SAMER, P. ; URRUTIA, S. A branch and cut algorithm for minimum spanning trees under conflict constraints. *Optimization Letters* 9, 1 (2015), 41–55.

[38] SNEATH, P. H. The application of computers to taxonomy. *Microbiology* 17, 1 (1957), 201–226.

[39] STUART, R. ; PETER, N. *Inteligência Artificial*. 2. ed. Elsevier, 2004. – ISBN 85-352-1177-2.

[40] TALBI, E.-G. *Metaheuristics: From Design to Implementation*. v. 74. Wiley Publishing, 2009. – ISBN 0470278587, 9780470278581.

[41] WOLSEY, L. A. *Integer Programming*. Wiley-Interscience, Series in Discrete Mathematics Optimization, 1998.

[42] ZHANG, R. ; KABADI, S. N. ; PUNNEN, A. P. The minimum spanning tree problem with conflict constraints and its variations. *Discrete Optimization* 8, 2 (2011), 191–205.