UNIVERSIDADE FEDERAL FLUMINENSE

CRISTIANE MARIA SANTOS FERREIRA

Algoritmos para o Problema da Árvore Geradora Mínima Generalizado

NITERÓI

2007

UNIVERSIDADE FEDERAL FLUMINENSE

CRISTIANE MARIA SANTOS FERREIRA

Algoritmos para o Problema da Árvore Geradora Mínima Generalizado

Dissertação de Mestrado submetida ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense como requisito parcial para a obtenção do título de Mestre. Área de concentração: Otimização Combinatória e Inteligência Artificial.

Orientador:
Luiz Satoru Ochi

Co-orientador:
Elder Magalhães Macambira

NITERÓI

2007

Algoritmos para o Problema da Árvore Geradora Mínima Generalizado

Cristiane Maria Santos Ferreira

Dissertação de Mestrado submetida ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense como requisito parcial para a obtenção do título de Mestre.

Aprovada por:

Prof. Luiz Satoru Ochi, D.Sc / IC-UFF (Orientador)

Prof. Celso da Cruz Carneiro Ribeiro, Dr. Hab. / IC-UFF

Prof. Eduardo Uchoa Barboza, D.Sc / EP-UFF

Prof. Marcus Vinicius Soledade Poggi de Aragão, Ph.D. / INF-PUC

Prof. Geraldo Robson Mateus, D.Sc / DCC-UFMG

Niterói, 04 de Maio de 2007.

Ao meu amado pai.

Agradecimentos

Não é facil agradecer todas as pessoas responsáveis pela conclusão deste trabalho. Em primeiro lugar, agradeço a Deus por todas as oportunidades que me foram dadas: minha saúde, os estudos e a possibilidade de ingressar em curso de pós-graduação.

As pessoas que primeiro vêm à minha mente são meus pais, que sempre colocaram minha educação e a de meus irmãos em primeiro plano. Quero agradecer tambem ao Cris e à Libi, pelo apoio que deram desde o primeiro momento em que decidi morar no Rio de Janeiro.

Alguns dos queridos amigos da Universidade Federal de Alagoas estão indiretamente presentes neste trabalho, apesar do pouco contato que tivemos nos últimos dois anos. Juntos aprendemos os primeiros ensinamentos da Ciência da Computação e com certeza muito do que hoje sei a eles devo. Vale mencionar tambem o professor Jaime Evaristo, por seu notável talento para ensinar os primeiros passos na arte de programar.

De maneira especial quero deixar registrada minha gratidão por todo o apoio e carinho que sempre recebi da "gaiola": Dani, Renathinha e Vivi. Sem vocês com certeza tudo teria sido muito mais difícil. Não posso deixar de agradecer também aos amigos do curso de pós-graduação em computação da UFF. Para citar alguns: Jacques, Tiago (Facada), Stênio, Luciana, Haroldo, Luciano, Rodrigo, Idalmis, Jonivan, Tiago (Jovem), Luís, Guto, Toca, Satan, Diego Brandao, Diego Leal, Marcel, entre muitos outros. Cada um de vocês me trouxe, além de grandes alegrias em momento de festa, inspiração e conhecimento em momentos de dúvida.

Agradeço ao meu orientador, professor Luiz Satoru Ochi, pelo apoio constante nos últimos dois anos e ao professor Eduardo Uchoa pela orientação importante ao final do trabalho.

Agradeço tambem à Capes, por ter financiado meus estudos.

Resumo

Dado um grafo G cujos vértices estão divididos em grupos, o Problema da Árvore Geradora Mínima Generalizado consiste em encontrar uma árvore que cubra um vértice de cada grupo de G, de forma a minimizar a soma dos custos das arestas. As principais aplicações desse problema são encontradas na área de síntese de redes de telecomunicações.

Neste trabalho, são propostas versões da meta-heurística GRASP que utilizam mais de um algoritmo construtivo de forma adaptativa, além de mecanismos adicionais de aprimoramento, como reconexão de caminhos e busca local iterada. Testes comparativos com instâncias apresentadas na literatura indicaram que o uso adaptativo de diferentes algoritmos construtivos é promissor. Também foi verificado que as versões GRASP que utilizam mecanismos adicionais apresentam melhores resultados que as demais. Os algoritmos propostos resultaram em soluções melhores que algumas das melhores soluções conhecidas, em um tempo computacional razoavelmente baixo.

Também foi implementado um algoritmo de geração de cortes baseado em uma formulação para o Problema de Steiner em Grafos Direcionado. Com esse algoritmo, foi possível encontrar limites duais para 82 instâncias em aberto.

São apresentadas ainda regras para o pré-processamento de instâncias euclideanas, baseadas no conceito de Distância *Bottleneck*. Em média, tais regras propiciaram a redução das instâncias para 14% do número de arestas em relação aos grafos originais.

Palavras-chave: Meta-heurísticas, Programação Inteira, Otimização em Redes

Abstract

Given a graph G whose vertices are divided into clusters, the Generalized Minimum Spanning Tree Problem consists of finding a tree T spanning exactly one vertex of each cluster of G, in such a way that minimizes the total length of T. The main applications of this problem arise in telecommunications network design.

In this work, we propose some versions of GRASP that uses more than one constructive algorithm in an adaptive way, together with additional improvement mechanisms, such as path relinking and iterated local search. Comparative tests with instances presented in literature have showed that the adaptive use of different constructive algorithms is promising. We also verifyed that the GRASP versions using additional mechanisms present better results than the others. The proposed algorithms resulted in better solutions compared to the best known ones, in a reasonable computational time.

We also implemented a cut generation algorithm based on a directed Steiner problem formulation. With this algorithm, it was possible to find dual limits for 82 open problems.

Futhermore, rules for preprocessing euclidean instances based on a bottleneck distance concept are presented. In average, such rules resulted on the reduction of the instances to 14% of their original size.

Keywords: Metaheuristics, Integer Programming, Network Optimization

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Esta dissertação aborda o Problema da Árvore Geradora Mínima Generalizado (PAGMG), que, como o nome sugere, é uma generalização do Problema da Árvore Geradora Mínima. O PAGMG pode ser definido sobre um grafo não-direcionado $G(V, E)$ onde V representa o conjunto de vértices e E o conjunto de arestas. Cada aresta $e \in E$ possui associado um custo $c_e \in R^+$. Neste problema, o conjunto de vértices V é particionado em m grupos disjuntos e não vazios $V_1, ..., V_m$ e o objetivo é determinar uma árvore de custo mínimo que cubra exatamente um vértice de cada grupo, como introduzido em [23].

Outra versão encontrada na literatura e diretamente relacionado ao PAGMG é o de encontrar uma árvore que cubra *pelo menos* um vértice de cada grupo. Tais generalizações seguem a idéia do já conhecido Problema do Caixeiro Viajante Generalizado, em que as cidades estão divididas por regiões e o caixeiro só precisa passar por uma cidade de cada região [12].

Aplicações para o PAGMG podem ser encontradas nas áreas de síntese de redes de telecomunicação, onde redes locais precisam ser interconectadas para formar redes maiores [8], na localização de instalações e mesmo na descrição de processos físicos que compõem redes [19].

Apesar de sua clara importância prática, a maior parte dos principais trabalhos relacionados é bastante recente. Dentre eles, há o de Feremans et al. [8], que apresentam e comparam várias formulações matemáticas, além de proporem um algoritmo *branch-and-cut* [7] que resolveu instâncias com até 200 vértices. Golden et al. [15] também trouxeram contruibuições importantes para o problema, dentre elas duas heurísticas que encontram as soluções ótimas de quase todas as instâncias introduzidas por Feremans. Mais recentemente, Oncan et al. [24] apresentaram mais um procedimento para o cálculo

de limites duais e um algoritmo de busca tabu que superou os resultados de Golden, além de apresentar instâncias novas e de dimensões maiores.

O presente trabalho tem como objetivo propor algumas versões da heurística GRASP para o PAGMG. Foram implementados vários algoritmos construtivos, dentre eles alguns propostos neste trabalho e alguns da literatura. Testes preliminares indicaram que não deve existir um algoritmo com desempenho superior aos demais em todas as instâncias. Tais resultados motivaram a implementação de versões GRASP que utilizam vários algoritmos construtivos de forma adaptativa. O objetivo é fazer com que o algoritmo possa utilizar o melhor construtivo para cada instância.

Também foram incorporados ao GRASP mecanismos de aprimoramento adicionais como Reconexão de Caminhos e Busca Local Iterada. Foi verificado que tais mecanismos propiciam melhoras significativas sobre o desempenho dos algoritmos. As heurísticas aqui propostas foram comparadas com os resultados anteriores obtidos por Oncan et al [24].

Este trabalho apresenta ainda um algoritmo de geração de cortes para o cálculo de limites duais. Tal algoritmo utiliza uma formulação do Problema de Steiner em grafos direcionado. Também são propostas regras de redução das instâncias, baseadas no conceito de distância *bottleneck*.

O trabalho está organizado como segue: o Capítulo 2 traz uma descrição mais detalhada do problema e uma revisão dos trabalhos da literatura; o Capítulo 3 apresenta os algoritmos propostos; no Capítulo 4 encontram-se os resultados computacionais e, por fim, no Capítulo 5 são apresentadas as conclusões e algumas sugestões de trabalhos futuros.

Capítulo 2

Problema da Árvore Geradora Mínima Generalizado

Neste capítulo será feita uma descrição do Problema da Árvore Geradora Mínima Generalizado (PAGMG) e dos trabalhos sobre ele relatados na literatura. Além disso, também serão traçados alguns comentários sobre problemas que se tornam importantes no estudo do PAGMG, como o Problema da Árvore Geradora Mínima e o Problema de Steiner em Grafos.

O Problema da Árvore Geradora Mínima Generalizado é definido sobre um grafo não direcionado $G(V, E)$ cujos vértices estão particionados em m grupos $\{V_1, V_2, ..., V_m\}$. Dado que $|V| = n$, tem-se que $V = V_1 \cup V_2 \cup ... \cup V_m$ e $V_l \cap V_k = \emptyset \ \forall l, k \in \{1, ..., m\}, l \neq k$. Assume-se que as arestas estão definidas apenas entre vértices de grupos diferentes e cada aresta e possui um custo associado $c_e \in R^+$. O objetivo do problema é determinar uma árvore de custo mínimo que cubra exatamente um vértice de cada grupo. Tal problema foi introduzido em 1995 por Myung [23] e também é denomidado *Equality Generalized Minimum Spanning Tree Problem* (E-GMSTP).

Diferentemente do Problema da Árvore Geradora Mínima, vastamente abordado na literatura e resolvido de maneira eficiente em tempo polinomial [20], o PAGMG é classificado como $\mathcal{NP}$-difícil [23] e há um número restrito de trabalhos relacionados de que se tem conhecimento.

Uma outra generalização do problema da Árvore Geradora Mínima, também abordada na literatura, é o problema de encontrar uma árvore que cubra pelo menos um vértice de cada grupo. Muitos autores vêm demoninando essa variante de L-GMSTP (L vem de *at Least*).

A Figura 2.1 ilustra o exemplo de uma solução para o Problema da Árvore Geradora Mínima Generalizado em um grafo cujo conjunto de vértices está particionado em quatro grupos. No exemplo, as arestas (3,5), (3,7) e (7,11) compoem a solução, também dita Árvore Geradora Generalizada.

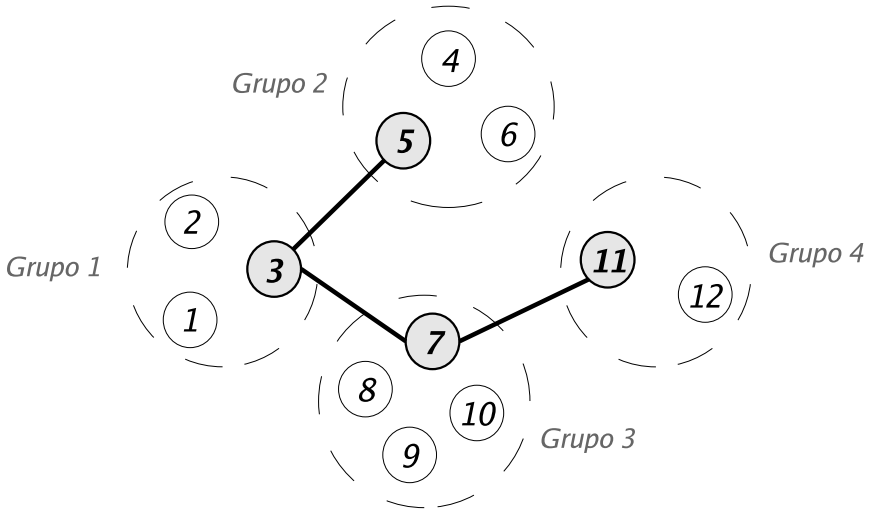


Figura 2.1: Uma solução do PAGMG.

Quando as instâncias são definidas sobre grafos completos e respeitam a desigualdade triangular, as soluções ótimas do L-GMSTP sempre terão apenas um vértice em cada grupo. Assim, nesses casos as duas generalizações do Problema da Árvore Geradora Mínima terão a mesma solução ótima. Uma conclusão direta disso é que o PAGMG é um caso particular do L-GMSTP [9].

Aplicações dos dois problemas podem ser encontradas, por exemplo, na área de telecomunicações, onde as redes regionais precisam ser interconectadas por uma árvore que contenha uma conexão para cada sub-rede. Para essa interconexão, um vértice de cada sub-rede deve ser selecionado como *gateway* [23].

O PAGMG também se aplica a vários problemas relacionados à localização de instalações que precisam estar conectadas por meio de rodovias ou enlaces de comunicação. Um exemplo acontece quando uma empresa quer estabelecer centros regionais de distribuição para suas lojas e precisa selecionar um ponto em cada região a fim de construir uma rede de comunicação interconectando esses centros [33].

Dror et al. [4] descreve um problema semelhante, na área de irrigação agrícola. Dadas m regiões que possuem uma fonte comum de água, o problema consiste em desenhar uma rede de irrigação de comprimento mínimo passando por pelo menos um vértice de cada região. Modelando o problema em um grafo, as arestas seriam as intersecções entre as

regiões e cada região corresponde a um grupo de vértices. Nesse caso, os grupos não são necessariamente disjuntos, mas Dror [4] mostra que é possível transformar as instâncias desse problema em instâncias do L-GMSTP através da inserção de vértices fictícios.

2.1 Problema da Árvore Geradora Mínima

O Problema da Árvore Geradora Mínima é considerado um dos mais importantes em otimização combinatória. Trata-se de, dado um grafo $G(V, E)$ onde cada aresta $e \in E$ possui um custo $c_e \in R^+$, encontrar uma árvore T que cubra cada vértice em V de forma a minimizar $\sum_{e \in T} c_e$.

Esse problema tem uma grande variedade de aplicações reais nas mais diversas áreas e, em muitos casos, surge como sub-problema para resolução de problemas mais complexos [1]. Dentre essas aplicações, pode-se citar o síntese de redes de transporte, energia, computadores e telecomunicações; agrupamento de dados; planejamento de produção; problemas de roteamento em geral; biologia computacional, etc.

Três algoritmos se destacam na resolução do problema: Kruskal [20], Prim [28] e Sollin [1]. Além de apresentarem complexidade de pior caso polinomial, tais algoritmos na prática são bastante eficientes, motivando com isso a aplicação do Problema da Árvore Geradora Mínima também como parte da resolução de vários problemas de otimização em grafos. A seguir, tem-se uma descrição dos dois primeiros algoritmos, mais comumente abordados na literatura.

O algoritmo de Kruskal primeiramente lista as arestas em ordem não-decrescente de seus custos. A solução T é composta de num componentes conexas, inicialmente $num = |V|$ e a T não possui arestas. A cada etapa, uma aresta e é analisada no intuito de verificar se sua inserção irá acarretar um ciclo em T. Caso isso ocorra, e é descartada; caso contrário, e é inserida em T. A cada inserção, o número de componentes conexas diminui uma unidade. Assim, o processo pára quando $num = 1$, ou, colocado de outra forma, quando $|T| = |V| - 1$. O pseudo-código está apresentado no Algoritmo 1.

Em uma implementação que siga o modelo original, a complexidade de pior caso do algoritmo é da ordem de $O(|V||E|)$, supondo tempo $O(|E|log|E|)$ para ordenação das arestas e $O(|V|)$ para a detecção de ciclos. Esse tempo pode ser melhorado para $O(|E| + |V|log|V|)$ somado ao tempo de ordenação, se forem utilizadas estruturas de dados mais apropriadas para as chamadas operações *union-find* de detecção de ciclos.

Algoritmo 1 Kruskal

```
ordenar(E);
T ← ∅;
num ← |V|;
enquanto num > 1 faça
  e(u, v) ← proxima_aresta(E);
  se !gera_ciclos(e) então
    T ← T ∪ e;
    num ← num − 1;
  fim se
fim enquanto
```

Na Figura 2.2 tem-se um exemplo da execução do algoritmo, apresentado por Ahuja et al. [1]. No exemplo, a lista de arestas segue a ordem (2,4),(3,5),(3,4),(2,3),(4,5),(2,1) e (3,1). O algoritmo adiciona (2,4),(3,5) e (3,4) em T. Nas duas iterações seguintes, (2,3) e (4,5) são descartadas, evitando a formação de ciclos. A seguir, a aresta (2,1) é inserida e o algoritmo termina sua execução.

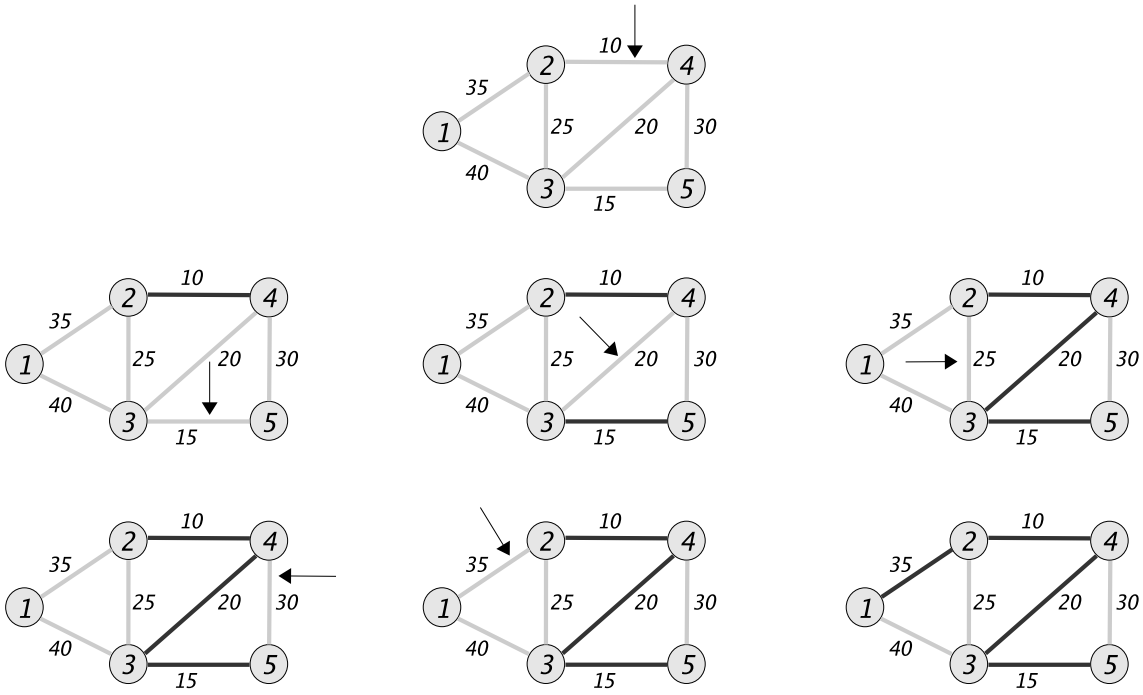


Figura 2.2: Ilustrando o funcionamento do algoritmo de Kruskal.

O algoritmo de Prim mantém uma árvore cobrindo um sub-conjunto S de V e adiciona a cada passo o vértice $v_i \in V \setminus S$ mais próximo de algum vértice $v_j \in S$. A cada iteração v_i é selecionado através da identificação da aresta e de menor custo do corte $[S, V \setminus S]$. Assim, e é inserido na solução T e v_i é adicionado a S. O algoritmo termina quando

$S = V$.

Prim executa $|V|-1$ operações de consulta à aresta de custo mínimo no corte $[S, V \setminus S]$, o que resulta em um tempo de pior caso de $O(|V||E|)$, se todas as arestas forem analisadas a cada iteração. Utilizando uma *heap* para encontrar o vértice com menor distância ao seu predecessor, a complexidade do algoritmo se reduz para $O(|E|log|V|)$.

O Algoritmo 2 apresenta o pseudo-código de Prim. O vetor δ armazena o predecessor de cada vértice na árvore. A função `proximo_vertice()` retorna o vértice $v \in V \setminus S$ com menor valor de $c_{(v,\delta_v)}$. O conjunto $E(v)$ contém todas as arestas que partem de v. Considera-se que $c_{(v,v)} = 0$ e $c_{(v,0)} = \infty$.

Algoritmo 2 Prim

para todo $i \in \{2, ..., |V|\}$ **faça**
  $\delta_i \leftarrow 0$;
fim para
$\delta_1 \leftarrow 1$;
$T \leftarrow \emptyset$; $S \leftarrow \emptyset$;
enquanto $|S| < |V|$ **faça**
  $v \leftarrow$ `proximo_vertice(`$V \setminus S$`, `δ`)`;
  $S \leftarrow S \cup v$;
  $T \leftarrow T \cup (v, \delta_v)$;
  para todo $e(u, v) \in E(v)$ **faça**
    se $c_e < c_{(u,\delta_u)}$ **então**
      $\delta_u \leftarrow v$;
    fim se
  fim para
fim enquanto

Para exemplificar o algoritmo, tomemos a Figura 2.3 que contém o mesmo grafo do exemplo anterior. Inicialmente, pode-se considerar $S = \{1\}$. O corte $[S, V \setminus S]$ contém então duas arestas: (1,2) e (1,3) e o algoritmo seleciona a primeira delas. Assim, S passa a ter dois elementos: $\{1, 2\}$ e o corte $[S, V \setminus S]$ contém as arestas (1,3),(2,3) e (2,4). A aresta (2,4) é selecionada por possuir o menor custo dentre as três. Nas próximas iterações são adicionadas as arestas (4,3) e (3,5). Como $S = \{1, 2, 3, 4, 5\}$, o algoritmo encerra.

Sollin pode ser entendido como uma mescla entre os algoritmos de Kruskal e Prim. O algoritmo inicia com uma floresta contendo $|V|$ árvores, que são passo a passo conectadas por meio das arestas de menor custo que as une. Ahuja [1] traz uma descrição detalhada dos três algoritmos, além de diversas áreas em que o problema vem sendo aplicado.

Para o cálculo de limites duais, as instâncias do PAGMG foram transformadas em instâncias do Problema de Steiner em Grafos, tendo em vista que o algoritmo de geração

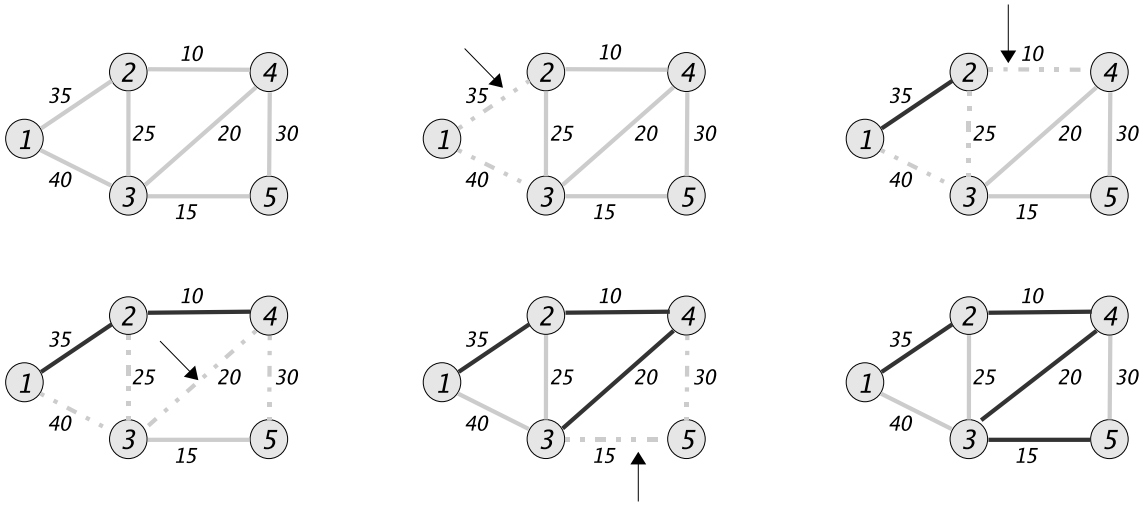


Figura 2.3: Ilustrando o funcionamento do algoritmo de Prim.

de cortes apresentado neste trabalho utiliza uma formulação de tal problema. Por esse motivo, será apresentada a seguir uma breve descrição do Problema de Steiner em Grafos.

2.2 Problema de Steiner em Grafos

Considere um grafo $G(V, E)$ onde V é o conjunto de vértices e E é o conjunto de arestas. Considere também um conjunto $T \subseteq V$ de vértices terminais. Cada aresta e possui um custo $c_e \in R^+$. O objetivo do Problema de Steiner em Grafos é encontrar uma árvore que cubra todos os vértices em T minimizando a soma dos custos de suas arestas, utilizando para isso quaisquer vértices de V.

A Figura 2.4 ilustra o exemplo de uma solução viável do problema. No exemplo, os quadrados representam os vértices terminais e os círculos representam os não-terminais - também chamados nós de Steiner, $|V| = 9$, $|T| = 4$ e $|E| = 16$. Observe que dois vértices não-terminais foram utilizados para compor a solução.

2.3 Trabalhos Relacionados

Contrariamente ao que acontece com o PAGM, existe apenas um número reduzido de trabalhos sobre o PAGMG na literatura. Nesta seção pretende-se apresentar os principais trabalhos de que se tem conhecimento sobre o PAGMG e sua versão "*at least*".

O PAGMG foi primeiramente abordado por Myung [23]. Através de uma redução

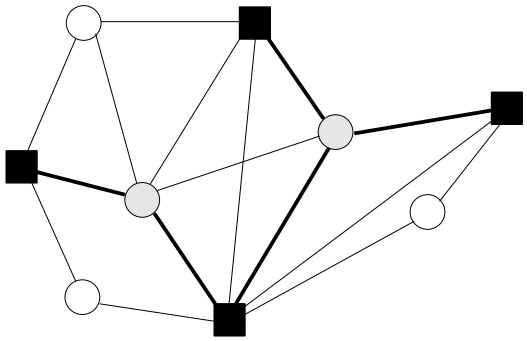


Figura 2.4: Solução viável do Problema de Steiner em Grafos.

ao Problema de Cobertura de Vértices, o autor demonstra que o PAGMG é fortemente $\mathcal{NP}$-difícil. Não é possível construir um algoritmo aproximativo com tempo de pior caso polinomial para o problema, a não ser que $\mathcal{P} = \mathcal{NP}$. Myung também propôs quatro formulações de programação inteira mista e um algoritmo *branch-and-bound* que resolveu para a otimalidade instâncias com até 100 vértices.

Ihler et al. [18] introduziram em 1999 o primeiro trabalho sobre o L-GMSTP, com uma outra denominação, *Problema de Steiner em Grupos*. Foi provado que o problema é um caso particular do Problema de Steiner em Grafos Generalizado [32].

Dror et al. [4] foram uns dos poucos a propor métodos heurísticos para o L-GMSTP. Os autores propuseram duas formulações de programação inteira e um algoritmo para encontrar limites inferiores baseado em relaxação lagrangeana. Nesse mesmo trabalho, também são propostas quatro heurísticas relativamente simples e um algoritmo genético.

A primeira heurística (H1) é uma adaptação do algoritmo de Prim para o Problema da Árvore Geradora Mínima. Considerando uma solução inicialmente com apenas um grupo V_i, deve-se passo a passo inserir o grupo mais próximo da solução parcialmente construída. Por "grupo mais próximo da solução" entenda-se o grupo que possui o vértice mais próximo (com menor caminho mínimo) a um dos vértices da solução. O processo se repete para $i = 1, ..., m$ e a heurística retorna a melhor dentre as m soluções construídas.

A segunda heurística (H2) constrói uma árvore geradora mínima sobre G como solução inicial e depois remove dessa árvore as arestas "redudantes", em ordem não crescente de seus pesos preservando a sua viabilidade. Uma aresta é dita "redundante" se sua remoção não afetará a viabilidade da árvore enquanto solução do L-GMSTP.

A heurística H3 é um algoritmo de busca local. Começa a partir de uma solução viável e remove cada aresta por vez, seguindo uma ordem não crescente dos pesos e

reconectando as duas componentes restantes pelo caminho mínimo entre os vértices que compoem a aresta removida. Caso uma solução melhor seja encontrada, o algoritmo reordena as arestas e recomeça a remoção. Caso todas as arestas tenham sido verificadas sem que haja melhorias, o algoritmo pára.

A última heurística (H4) constrói n soluções e retorna a melhor dentre elas. Para cada vértice $v_i \in V$, o algoritmo constrói uma árvore a partir da união entre v_i e o vértice mais próximo de v_i de cada grupo. A união entre os vértices v_i e v_j é feita através da inserção do caminho mínimo entre v_i e v_j na solução.

O algoritmo genético é bastante simples. A solução é representada por um string "binária" X de comprimento n, em que $x_i = 1$ representa a participação do vértice i na solução. O operador de cruzamento sobre as soluções X_1 e X_2 permuta dois pedaços das duas soluções. A mutação é a inversão de um valor x_i e ocorre com probabilidade de 1%. O algoritmo considera soluções inviáveis aplicando uma penalidade proporcional ao número de grupos não cobertos pela solução.

Os testes foram realizados em 20 instâncias geradas pelos autores, com número de vértices variando entre 25 e 500, número de grupos entre 4 e 50 e arestas com pesos aleatórios. Os resultados indicaram um desempenho bem melhor da meta-heurística em relação às outras heurísticas, apesar de seu tempo de execução ser também bem maior. Mais tarde, Feremans [7] em 2001 comparou os resultados deste algoritmo genético com os valores ótimos das instâncias e demonstrou que as soluções se encontraram, em média, a $6,53\%$ da otimalidade.

Apesar de terem sido propostos e testados em instâncias do L-GMSTP, é fácil perceber que tais algoritmos podem ser aplicados a quaisquer instâncias do PAGMG, com exceção da heurística H3, que não é aplicável em instâncias com distâncias euclideanas.

Pop et al. [27] propuseram uma formulação de programação inteira, alternativa à de Myung [23], utilizada para resolver instâncias com até 75 vértices. Esse trabalho também descreveu um algoritmo de relaxação que apresentou bons resultados preliminares. Os mesmos autores [26] demonstraram através de um algoritmo polinomial que o PAGMG pode ser aproximado por um fator de 2ρ se o número de elementos dos grupos for limitado por ρ. Tal algoritmo é baseado no método descrito por Magnanti e Wolsey [22] para o *Vertex Weighted Steiner Tree Problem*.

Feremans é uma das autoras que mais trouxe contribuições ao Problema da Árvore Generadora Mínima Generalizado. Juntamente com Laporte e Labbé [9], a autora mostrou

que, como o problema se trata de um caso particular do L-GMSTP, algumas demonstrações de Dror et al. [4] derivam diretamente do trabalho de Myung [23]. No mesmo trabalho, os autores demonstraram que três das quatro formulações propostas por Dror estavam incorretas e propuseram mais uma formulação válida para o L-GMSTP.

Feremans et al. [10] fizeram uma análise comparativa de oito formulações do problema, algumas delas propostas por Myung. Os modelos são os seguintes: duas formulações baseadas em propridades de árvore e propostas por Myung (*Undirected Cutset* e *Undirected Subpacking*); suas correspondentes para o caso direcionado, sendo uma de Myung (*Directed cutset*) e uma proposta pelos próprios autores (*Directed subpacking*); uma formulação baseada em controle de fluxo (*Multiflow Formulation*), de Myung, e sua projeção em um espaço mais restrito (*Flow Cut Formulation*); e, por último, mais uma formulação de Myung (*Undirected Cluster Subpacking*) e sua correspondente para o caso direcionado (*Directed cluster subpacking*).

Ficou provado que quatro dessas formulações são equivalentes no sentido de que o poliedro associado às suas relaxações lineares são iguais e possuem os mesmos valores ótimos. Essas quatro também provêem limites duais melhores em relação às outras quatro, e a formulação *Undirected Cluster Subpacking* é a mais compacta em termos de variáveis. Tal formulação será apresentada a seguir.

Considere as variáveis binárias

$$x_e = \begin{cases} 1 & \text{se } e = \{i,j\} \in E \text{ estiver na solução} \\ 0 & \text{caso contrário} \end{cases}$$

$$y_i = \begin{cases} 1 & \text{se } i \in V \text{ estiver na solução} \\ 0 & \text{caso contrário,} \end{cases}$$

considere também a seguinte notação: $x(E') = \sum_{e \in E'} x_e$ para $E' \subseteq E$ e $y(V') = \sum_{i \in V'} y_i$ para $V' \subseteq V$. Dado que $S \subseteq V$, denotamos por $E(S) = \{e = \{i,j\} \in E : i,j \in S\}$ o sub-conjunto de arestas cujos vértices pertencem a S. Por fim, $\mu(S) = |\{k : V_k \subseteq S\}|$ é o número de grupos inclusos em S. Assim, a formulação é a que segue:

Minimizar

$$\sum_{e \in E} c_e \times x_e$$

Sujeito a

$$y(V_k) = 1 \qquad \forall k \in K, \tag{2.1}$$

$$x(E) = |K| - 1, \tag{2.2}$$

$$x((E(S)) \leqslant y(S) - 1 \qquad \forall S \subset V, 2 \leqslant |S| \leqslant |V| - 1, \mu(S) \neq 0, \tag{2.3}$$

$$x_e \in \{0, 1\} \qquad \forall e \in E \tag{2.4}$$

$$y_i \in \{0, 1\} \qquad \forall i \in V \tag{2.5}$$

As restrições 2.1 e 2.2 garantem que a árvore cobre exatamente um vértice de cada grupo. O conjunto de restrições 2.3 diz respeito à eliminação de sub-ciclos e é denominado *Generalized Subtour Elimination Constraints* (GSEC). As restrições 2.4 e 2.5 descrevem a integralidade das variáveis x e y, respectivamente.

Em sua tese de doutorado [7], Feremans fez uma investigação da estrutura poliédrica do PAGMG. A tese também propôs um algoritmo *branch-and-cut* e quatro famílias de desigualdades válidas: desigualdades de Hammock, de ciclos ímpares, de casamento de ciclos ímpares e de "buracos" ímpares.

Com a análise poliédrica, pôde-se provar que as desigualdades $x_e \geq 0$, $y_i \geq 0$, $y(V_k) \geq 1$ e as de Hammock definem facetas da envoltória convexa do problema.

O *branch-and-cut* utiliza a formulação *Undirected Cluster Subpacking*, descrita anteriormente, e possui basicamente sete passos:

1. Inicialização, com a resolução do problema linear sem as GSEC's
2. Seleção do sub-problema, seguindo um critério estabelecido
3. Resolução do sub-problema, que incorpora procedimento de busca local e de arredondamento
4. Primeiro procedimento de separação, que busca algumas violações das GSEC's específicas
5. Segundo procedimento de separação, buscando mais violações das GSEC's para introduzir ao sub-problema
6. Terceiro procedimento de separação, que busca violações das desigualidades de Hammock e de ciclos ímpares
7. *Branching*, que cria os dois sub-problemas sobre a restrição 2.2

Para gerar uma solução inteira factível como dado de entrada do *branch-and-cut*, Feremans propôs um algoritmo de Busca Tabu. Como heurística construtiva, foram testadas adaptações dos algoritmos de Kruskal, Prim e Sollin e o primeiro foi escolhido por demonstrar um melhor desempenho. A busca local do item 3 basicamente seleciona um grupo V_i e tenta reconstruir a solução através da adaptação de Kruskal, considerando todos os vértices do grupo V_i e fixando na solução os vértices dos outros grupos que estavam na solução corrente.

Feremans testou seu algoritmo em instâncias geradas aleatoriamente (euclideanas e não-euclideanas) com $24 \leq n \leq 200$ e $8 \leq m \leq 20$. O algoritmo também foi testado em 169 instâncias adaptadas do respositório de instâncias do Problema do Caixeiro Viajante (TSPLIB [30]) com $48 \leq n \leq 226$. A adaptação foi feita por Fischetti et al. [12] para o Problema do Caixeiro Viajante Generalizado e será explicada com mais detalhes na Seção 4.1.

O *branch-and-cut* conseguiu resolver todas as instâncias euclideanas com até 160 vértices e todas as instâncias com custos aleatórios com até 200 vértices. Também resolveu 150 das 169 instâncias do TSPLIB que foram testadas, considerando um tempo limite de duas horas. Comparando com o algoritmo de Myung [23], o algoritmo de Feremans apresentou limites melhores e conseguiu resolver instâncias bem maiores.

A tese de Feremans também propõe uma formulação para o L-GMSTP. Em comparação com a formulação *Undirected Cluster Subpacking*, há a evidente substituição do conjunto de restrições $y(V_k) = 1$ por $y(V_k) \geq 1$. As restrições GSEC's também foram substituídas, pelo fato de não serem válidas nesta versão do problema. Em seu lugar entram restrições de corte mínimo. Segundo a autora, essa formulação é reconhecidamente "fraca" em termos de relaxação linear, dedução que pode ser feita por analogia ao trabalho de Magnanti e Wolsey [35]. Mas as restrições de corte tornam o procedimento de separação fácil de ser resolvido por meio do problema de fluxo máximo.

Feremans apresenta ainda uma forma de transformar instâncias do L-GMSTP em instâncias do PAGMG. A transformação é feita inserindo uma aresta entre cada par de vértices de grupos diferentes que estão conectados por algum caminho na instância original. Também são retiradas arestas entre vértices que pertencem ao mesmo grupo. Foi provado que a solução ótima da instância gerada por tal transformação pode ser convertida em uma solução viável do L-GMSTP. Assim, tal solução é um limite superior para a solução ótima do L-GMSTP.

O *branch-and-cut* proposto foi testado também em instâncias do L-GMSTP, utili-

zando a formulação descrita acima e a transformação das instâncias. Os resultados foram comparados com o algoritmo genético de Dror et al. [4] e o *branch-and-cut* encontrou a solução ótima de todas as instâncias, em um tempo computacional razoável se comparado ao tempo de execução da meta-heurística.

Kansal e Torquato [19] apresentaram mais uma aplicação para o L-GMSTP, dessa vez em Física. Eles desenvolveram um método genérico com base no problema, através do qual as informações extraídas de uma rede podem ser utilizadas para a compreensão dos processos físicos que guiam a formação dessa rede.

Em 2002, Raghavan [29] demonstrou que o PAGMG pode ser modelado como um Problema de Steiner em Grafos com a inserção de vértices fictícios em cada grupo. Segundo o autor, a relaxação linear da formulação resultante é equivalente às quatro formulações apresentadas por Feremans et al. [10].

Shyu et al. [33] em 2003 apresentam uma meta-heurística baseada no método colônia de formigas para o L-GMSTP. O algoritmo foi denominado *Ant-tree* e foi desenvolvido como o passo inicial de um estudo dos métodos de colônia de formigas para problemas de otimização em árvores. O método demonstrou-se eficiente em produzir boas soluções aproximadas.

Em 2004, Pop [25] demonstrou que, além de ser $\mathcal{NP}$-difícil, o PAGMG não pode ser resolvido em tempo polinomial nem se o grafo for uma árvore, diferentemente do que acontece com vários problemas $\mathcal{NP}$-Completos [13].

Além disso, a autora apresenta um algoritmo exato baseado na relaxação de uma formulação de Myung [23], denominado *Rooting procedure*. De forma resumida, a relaxação é obtida através da seleção aleatória de um grupo V_k para ser fixado como raiz da árvore. Se a solução ótima de tal relaxação for uma árvore geradora generalizada, o problema está resolvido. Caso contrário, outro grupo é escolhido e o processo continua até que se encontre uma solução viável do PAGMG.

O algoritmo foi testado em instâncias com custos aleatórios apresentadas por Myung [23] e comparado com seu trabalho e com o de Feremans [7]. O *Rooting procedure* conseguiu encontrar a solução ótima de todas as instâncias testadas, com até 240 vértices, algumas delas ainda em aberto. É interessante apontar que o algoritmo fez no máximo 4 escolhas de grupos até encontrar as soluções ótimas.

Em 2005, Feremans et al. [11] discutiram a relação entre o PAGMG e o L-GMSTP e apresentaram melhorias no *branch-and-cut* proposto anteriormente. Segue uma descrição

suscinta das modificações apresentadas.

O procedimento de separação foi modificado, retirando-se as desigualdades de ciclos ímpares e de "buracos" ímpares. Em seu lugar, passaram a ser utilizados os cortes de Chvátal-Gomory [21], que, segundo os autores, vêm sendo re-descobertos por muitos pesquisadores. A estratégia de *branching* também foi modificada, o critério passou a ser as variáveis y com valor próximo de $0,5$. Algumas mudanças foram introduzidas especificamente para instâncias do L-GMSTP, como a heurística para arredondamento – baseada na adaptação do algoritmo de Prim – e políticas de separação mais prudentes para as GSEC's.

Os testes foram realizados no sentido de comparar quatro versões do *branch-and-cut* tanto para o PAGMG quanto para o L-GMSTP. As instâncias utilizadas foram as mesmas de Dror et al. [4]. Os resultados indicaram que as mudanças trouxeram melhoras efetivas no comportamento do *branch-and-cut*, especialmente em termos de tempo computacional. Quanto às instâncias adaptadas do TSPLIB, devido aos cortes de Chvátal-Gomory, os limites duais no nó raiz foram significativamente melhorados.

Ainda em 2005, Golden et al. [15] apresentaram um algoritmo para o cálculo de limites duais do PAGMG, além de uma comparação minuciosa de alguns algoritmos construtivos e duas heurísticas.

O cálculo de limites duais é bastante simples. Listadas em ordem crescente de seus pesos, as arestas são inseridas uma a uma, com a condição de não gerar ciclos na solução. O processo pára quando $m-1$ arestas forem inseridas. Observe que esse algoritmo pode gerar uma árvore desconexa e não necessariamente cobre todos os grupos. Nos testes com as 169 instâncias adaptadas do TSPLIB, os limites encontrados ficaram a $35,94\%$ dos valores ótimos, em média.

Os construtivos comparados são adaptações dos clássicos algoritmos de Kruskal, Prim e Sollin para o Problema da Árvore Geradora Mínima. As adaptações de Kruskal e Sollin apresentaram resultados melhores que Prim, e dentre os dois, há uma ligeira vantagem para o algoritmo baseado em Kruskal.

As heurísticas propostas foram uma busca local bastante simples, porém eficiente, e um algoritmo genético. As soluções iniciais são construídas da seguinte forma: seleciona-se um vértice de cada grupo de forma aleatória e constrói-se uma árvore geradora mínima com os vértices selecionados – através do algoritmo de Kruskal ou Prim.

Para a análise da vizinhança, define-se uma ordem em que os grupos serão visitados.

Considera-se um grupo de cada vez, seguindo a ordem estabelecida. Assim, durante a visita ao grupo V_k, o vértice $v_i \in V_k$ presente na solução é substituído por cada vértice $v_j \in V_k$ e a árvore geradora mínima é recomputada. A melhor dentre as $|V_k|$ soluções geradas passa a ser a solução corrente. A busca pára quando m grupos forem visitados sem que haja uma atualização da solução corrente. Como esse procedimento é repetido 500 vezes, garante-se uma boa varredura no espaço de busca, já que a solução inicial é construída aleatoriamente.

No algoritmo genético, cada indivíduo é representado por um vetor de tamanho m, contendo o vértice de cada grupo que está presente na solução. O operador de cruzamento seleciona um ponto de dois indivíduos e combina as duas soluções gerando dois filhos. O operador *tree separator* remove uma aresta da solução e a partir de cada sub-árvore gerada, constrói duas novas soluções utilizando a adaptação do algoritmo de Prim. Na mutação, seleciona-se um gene aleatoriamente e substitui-se o vértice referente por outro vértice do mesmo grupo. Com uma pequena probabilidade, aplica-se o procedimento de busca local descrito anteriormente sobre os indivíduos da população corrente. A população da próxima geração é composta em 10% pelos melhores indivíduos e em 90% por indíviduos selecionados por meio de roleta.

Os autores comparam os algoritmos nas instâncias adaptadas do TSPLIB, das quais 150 possuem o valor ótimo conhecido. As soluções do procedimento de busca local e do algoritmo genético ficaram, em média, a $0,07\%$ e $0,01\%$ da otimidalidade, respectivamente. O algoritmo genético também apresentou um desempenho ligeiramente melhor nos testes com instâncias cujo ótimo não é conhecido e com instâncias aleatórias propostas pelos autores. Vale colocar que tal algoritmo consome aproximadamente o dobro do tempo da busca local.

Em 2006, Haouari e Chaouachi [17] propuseram e compararam vários algoritmos para o cálculo de limites duais para o L-GMTP, todos tomando como base formulações para o Problema de Steiner. Os autores também propuseram uma adaptação do método PROGRES, de mesma autoria, e melhorias em um algoritmo genético proposto anteriormente [4]. O algoritmo foi comparado com os trabalhos de Dror et al. [4] e Shyu et al. [33], utilizando as instâncias apresentadas por Dror e instâncias dos próprios autores.

Recentemente, Oncan et al. [24] (em um trabalho ainda não publicado) propuseram um procedimento para o cálculo de limites duais e um algoritmo de busca tabu para as duas generalizações. Nesse mesmo trabalho também são apresentadas instâncias novas e de maior porte para PAGMG. Tais instâncias foram geradas a partir das instâncias do

TSPLIB, utilizando as mesmas técnicas de agrupamento de Fischetti [12].

O cálculo dos limites inferiores é feito com base na relaxação linear de uma formulação proposta por Myung [23]. Com ele, os autores encontraram limites bem melhores que os de Golden et al. [15], mas não foi possível encontrar limites para as instâncias que foram apresentadas então.

O algoritmo de busca tabu é dito *attributed-based*, por considerar os atributos da solução como critério tabu. Os atributos de uma solução s, no caso, são os vértices a compõem. Toda vez que um vértice v_i deixa de participar da solução corrente, só poderá ser re-inserido após θ iterações ou se o custo da solução que estiver sendo avaliada for melhor que γ_i. Assim, associados a cada vértice v_i há um valor β_i que indica até que iteração v_i é tabu e um critério de aspiração γ_i. A cada iteração, o critério de aspiração de cada vértice em s é atualizado para $min\{\gamma_i, c(s)\}$, onde $c(s)$ é o custo de s.

A vizinhança $N^c(s)$ de uma solução s é uma generalização da vizinhança proposta por Golden. A idéia é visitar c grupos por vez. Para cada grupo $V_p, 1 \leq p \leq c$, sustitui-se o vértice i_p presente na solução por um vértice j_p, $i \neq j$. Após isso é construída uma árvore geradora mínima com os vértices da solução. Assim, a vizinhança $N^1(s)$ abrange $|V_p| - 1$ possíveis soluções, a vizinhança $N^2(s)$ abrange $(|V_p| - 1) \times (|Vq| - 1)$ soluções, onde V_p e V_q são dois grupos selecionados.

É fácil perceber que quanto maior o número de grupos selecionados, maior será o espaço de busca. Procurando um *trade-off* entre tempo e qualidade de solução, a primeira vizinhança (mais rápida) executa um número maior de iterações e a última, por poucas iterações. O algoritmo tabu explora as vizinhanças $1 \leq c \leq 3$ sequencialmente, $N^1(s)$, $N^2(s)$, $N^3(s)$, $N^1(s)$, etc.

A solução inicial é construída aleatoriamente, como faz Golden [15]. Na busca local, a escolha dos c grupos em cada iteração é feita por roleta, onde cada grupo possui uma probabilidade de ser escolhido inversamente proporcional à quantidade de vezes que já foi selecionado. A vizinhança é percorrida exaustivamente e a melhor solução encontrada passa a ser a solução corrente. A fim de diversificação, as soluções são avaliadas com uma penalização que leva em conta o número de iterações em que cada vértice já esteve na solução corrente. O critério de parada estabelecido foi o número de iterações, de acordo com o tamanho de cada instância.

Os testes de comparação com a literatura foram realizados com 101 instâncias propostas pelos autores. O algoritmo tabu foi comparado com os dois algoritmos de Golden

descritos anteriormente [15] e um versão melhorada do algoritmo genético, récem implementada. Em todas as instâncias, a proposta de Oncan atingiu resultados melhores ou iguais aos de Golden.

O algoritmo descrito também foi adaptado para o L-GMSTP. Essa adaptação foi comparada com o trabalho de Chaouachi [17] sobre as instâncias propostas por Dror. O algoritmo de busca tabu não superou os resultados do genético, mas os autores consideraram os resultados promissores.

Este capítulo apresentou uma descrição do Problema da Árvore Geradora Mínima Generalizado e de problemas relacionados. Também foi feita uma revisão suscinta dos trabalhos sobre o tema encontrados na literatura. O capítulo seguinte traz os algoritmos propostos e implementados no presente trabalho para o PAGMG.

Capítulo 3

Algoritmos para o PAGMG

Este capítulo apresenta os algoritmos implementados para o Problema da Árvore Geradora Mínima Generalizado. Dentre eles, há algumas versões da heurística GRASP, que diferem entre si pelos algoritmos construtivos e pela utilização ou não de mecanimos adicionais de aprimoramento. Também foi implementado um algoritmo para o cálculo de limites duais, baseado na relaxação linear de uma formulação para o Problema de Steiner em Grafos. Por fim, são descritas as regras de redução utilizadas no pré-processamento das instâncias.

3.1 GRASP

GRASP (*Greed Randomized Adaptive Search Procedure*) é uma meta-heurística multi-partida, proposta por Feo e Resende [6], em que cada iteração é composta de uma fase de construção e de uma fase de aprimoramento. Durante o processo, a solução incubente é armazenada e atualizada sempre que a fase de aprimoramento resulta em uma solução melhor. Ao final de um número estabelecido de iterações, o algoritmo retorna a melhor solução encontrada.

A fase de construção gera uma solução viável para o problema através de um procedimento parcialmente guloso e parcialmente aleatório. A cada estapa da construção, a seleção dos elementos que vão compor a solução é feita aleatoriamente em um subconjunto do melhores elementos candidatos, denominada Lista de Candidatos Restrita. A fase de aprimoramento é uma busca local.

O Algoritmo 3 apresenta o pseudo-código de uma versão tradicional do GRASP para um problema de minimização. A função `custo(`s`)` retorna o custo da solução s e s^* é a

solução incubente.

Algoritmo 3 GRASP

```
s* ← ∅;
para todo i ∈ {1, ..., It_max} faça
  s_0 ← constroi_solucao(α);
  s ← busca_local(s_0);
  se custo(s) < custo(s*) então
    s* ← s;
  fim se
fim para
retornar s*;
```

No pseudo-código apresentado, cada iteração é um procedimento independente, não havendo nenhum tipo de memória entre as iterações no decorrer da execução.

Porém muitas técnicas podem ser agregadas ao GRASP tradicional, no intuito de aproveitar de alguma forma resultados de iterações anteriores para tentar encontrar melhores soluções. Dentre essas técnicas, pode-se citar reconexão de caminhos, mineração de dados e quaisquer procedimentos de aprimoramento sobre soluções já encontradas [31]. Além disso, outros elementos podem ser incorporados para aprimorar o desempenho do GRASP, como o uso de diferentes algoritmos de construção ou mesmo outras meta-heurísticas: busca em vizinhança variada (ou VNS), busca tabu, busca local iterada, etc.

Algumas versões propostas neste trabalho utilizam mais de um algoritmo construtivo, além de mecanismos adicionais, como reconexão de caminhos e busca local iterada. A seguir serão descritos todos esses procedimentos. Para o entendimento dos algoritmos, considere o PAGMG descrito em um grafo $G(V, E)$ em que $V = V_1 \cup V_2 \cup ... \cup V_m$ e $|V| = n$. Considere também um vetor γ em que, dada uma solução T, $\gamma[i] \in V$ é o vértice do grupo i selecionado para compor T.

3.1.1 Algoritmos de Construção

Nesta subseção serão descritas as heurísticas utilizadas na fase de construção do GRASP. Foram implementados oito algoritmos construtivos, alguns propostos na literatura e outros no presente trabalho. Entretanto, testes preliminares indicaram que cinco dessas heurísticas resultam soluções significativamente melhores que as outras três.

Dentre os três construtivos com desempenho inferior estão duas adaptações das heurísticas H2 e H4 de Dror et al. [4]. O terceiro construtivo inicia pela construção de uma permutação dos grupos, que procura respeitar a proximidade entre os mesmos. Em se-

guida, calcula o caminho mínimo entre o primeiro e o último grupo da permutação. Esse cálculo é feito de forma que o resultado seja uma solução válida para o PAGMG. Em vista de seus resultados, essa e as outras duas heurísticas não foram utilizadas nos testes finais.

Segue uma descrição detalhada do funcionamento das heurísticas que são de fato utilizadas nas abordagens propostas.

3.1.1.1 C1 - Construção Aleatória

Essa heurística foi primeiro utilizada por Golden [15] e constrói a solução de maneira trival. Seleciona-se um vértice de cada grupo aleatoriamente. Em seguida, através dos algoritmo de Kruskal, calcula-se a árvore geradora mínima dos m vértices selecionados.

O pseudo-código é apresentado no Algoritmo 4. A função `rand(`V_i`)` seleciona aleatoriamente um vértice do grupo V_i. A árvore geradora mínima é calculada na função `kruskal()`, utilizando apenas arestas que saem e chegam de vértices em γ. Optou-se por utilizar Kruskal em vez Prim pelo fato do primeiro ser, em média, mais eficiente para as instâncias utilizadas neste trabalho.

Algoritmo 4 C1 - Construção aleatória

```
para todo i ∈ {1, .., m} faça
  γ[i] ← rand(V_i);
fim para
T ← kruskal(γ, E(γ));
retornar T;
```

Apesar de sua simplicidade, esse algoritmo gera soluções de qualidade aceitável pelo fato de retornar sempre uma árvore geradora mínima, além de possuir um pequeno tempo de execução. Outra característica a favor do algoritmo é que as soluções são bem diferentes entre si. Logo, considerando uma grande quantidade de execuções, o espaço de busca é razoalvemente bem percorrido.

3.1.1.2 C2 - Adaptação de Kruskal

Como o nome sugere, esse construtivo é baseado no algoritmo de Kruskal apresentado na Seção 2.1 para o Problema da Árvore Geradora Mínima. A modificação consiste em assegurar que apenas um vértice de cada grupo esteja presente na solução. Feremans et al. [7] foram os primeiros autores a citar a possibilidade dessa adaptação.

As arestas também são inseridas por ordem de seus pesos. Como o objetivo é cobrir

exatamente um vértice de cada grupo, a inclusão da aresta $e(v_1, v_2)$ determina a seleção dos vértices $v_1 \in V_i$ e $v_2 \in V_j$. Assim, arestas que partem de $v_3 \in V_k$ não podem ser inseridas caso $k = i$ e $v_3 \neq v_1$ ou $k = j$ e $v_3 \neq v_2$. Logo, além de evitar arestas que resultem na formação de ciclos, o algoritmo também descarta arestas que implicariam na seleção de um segundo vértice para algum dos grupos.

A Figura 3.1 ilustra um exemplo da execução da heurística. O grafo é o mesmo utilizado no exemplo da Figura 2.1. A princípio, nenhum grupo possui vértices selecionados, ou seja, $\gamma[i] = 0, \forall i \in \{1, ..., m\}$. As arestas são ordenadas, gerando a seguinte lista: (3,5), (6,11), (3,7), (5,7), (7,11), (3,8), (6,7)... (Figura 3.1(a)) Na primeira iteração a aresta (3,5) é inserida, assim $\gamma[1] = 3$ e $\gamma[2] = 5$ (Figura 3.1(b)). A seguir, (6,11) é descartada pois $\gamma[2] \neq 6$ (Figura 3.1(c)). A aresta (3,7) entra na solução e $\gamma[3] = 7$ (Figura 3.1(d)). Evitando a formação de ciclos, descarta-se (5,7). Por fim, analisando a inserção da aresta (7, 11), verifica-se que $\gamma[3] = 7$ e $\gamma[4] = 0$ (Figura 3.1(e)). Por fim, (7,11) passa a fazer parte da solução e o algoritmo termina (Figura 3.1(f)).

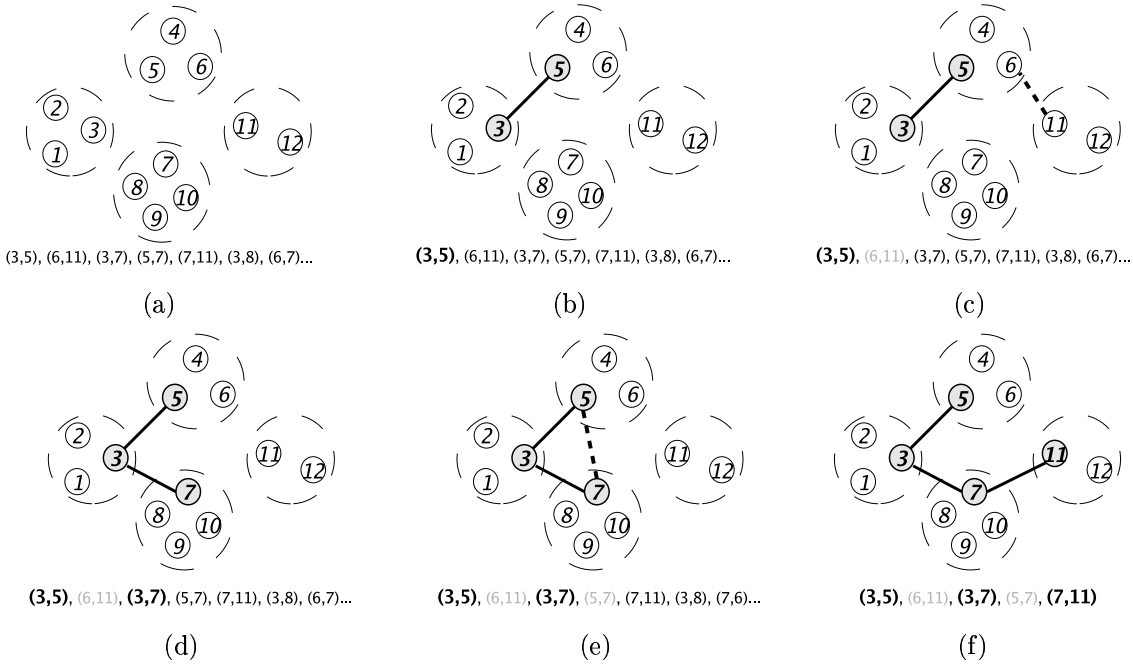


Figura 3.1: Ilustrando a heurística construtiva C2.

Pode-se perceber que C2 é um algoritmo guloso, pois a cada iteração a "melhor" aresta candidata é analisada. Assim, foi implementada também uma versão randomizada de C2, que seleciona cada aresta aleatoriamente a partir de um sub-conjunto de E. Esse sub-conjunto é chamado de Lista de Candidatos Restrita (LCR). A LCR é criada a cada iteração, antes da seleção da próxima aresta.

Chamemos de c_{min} e c_{max} respectivamente o menor e o maior custo das arestas ainda não avaliadas durante o processo de construção. A LCR é composta de todas as arestas cujo custo seja menor ou igual a $c_{min} + \alpha(c_{max} - c_{min})$. O valor de α é estabelecido no intervalo [0,1] e indica o quão gulosa será a seleção das arestas. É fácil perceber que se $\alpha = 0$, o algoritmo procederá da mesma forma que a adaptação de Kruskal descrita anteriormente.

O pseudo-código da versão randomizada, denotada por RC2, está apresentado no Algoritmo 5. O número de componentes conexas em T é indicado por num. O primeiro laço inicializa o valor de γ para todos os grupos. A LCR está indicada por L e é construída pela função `constroi_LCR()`. A função `rand()` seleciona um elemento de L aleatoriamente. No decorrer do algoritmo, uma aresta $e(u, v)$ é analisada a cada iteração, e se for incluída na solução, num decrementa uma unidade. A variável g_v indica o índice do grupo a que o vértice v pertence.

Observe que a principal diferença em relação ao Algoritmo 1 (Kruskal) é a presença de mais um critério na avaliação de $e(u, v)$. Além da detecção de ciclos, é preciso verificar se os grupos g_u e g_v possuem vértices diferentes de u e v na solução. O algoritmo termina quando T possui apenas uma componente conexa.

Algoritmo 5 RC2 - Adaptação de Kruskal para o PAGMG

```
ordenar(E);
T ← ∅;
num ← m;
para todo i ∈ {1, ..., m} faça
  γ[i] ← 0;
fim para
enquanto num > 1 faça
  L ← constroi_LCR(E, α);
  e(u, v) ← rand(L);
  se !gera_ciclos(e) e (γ[g_u]= u ou γ[g_u]= 0) e (γ[g_v]= v ou γ[g_v]= 0) então
    T ← T ∪ e;
    num ← num − 1;
    se γ[g_u] = 0 então
      γ[g_u] ← u;
    fim se
    se γ[g_v] = 0 então
      γ[g_v] ← v;
    fim se
  fim se
fim enquanto
retornar T;
```

3.1.1.3 C3 - Adaptação de Prim

A adaptação de Prim pouco difere do original. Em sua versão para o PAGMG, o algoritmo procede com os grupos de forma semelhante com que procede com os vértices na versão para o PAGM. Considera-se como distância entre dois grupos o custo da menor aresta que une seus vértices.

Ao inicío do algoritmo, a solução T é composta por apenas um grupo V_i. A cada passo, uma aresta é inserida em T, agregando à solução um grupo V_j ainda não coberto. Na primeira iteração são analisadas todas as arestas entre os vértices de V_i e os outros grupos. Nas iterações seguintes, considera-se apenas as arestas que unem os vértices presentes em T e os vértices de grupos que ainda não fazem parte de T. O procedimento termina quando todos os grupos estão cobertos pela árvore. Como a escolha de V_i influencia a solução a ser gerada, executa-se o mesmo procedimento para $i = \{1, ..., m\}$.

Para entender melhor o funcionamento dessa heurística, tomemos a Figura 3.2. O grafo é o mesmo que vem sendo utilizado em exemplos anteriores. Ao início da execução, considere $T = \emptyset$. No exemplo dado, $i = 1$, ou seja, o objetivo inicial é encontrar a aresta de menor custo que una V_1 ao resto do grafo (Figura 3.2(a)). Na Figura 3.2(b), todas as arestas que partem de V_1 são analisadas e (3,5) é selecionada.

Na segunda iteração, tem-se $T = \{(3, 5)\}$ e procura-se a menor aresta que una um grupo ainda não coberto (3 ou 4) a T. São analisadas todas as arestas $e(u, v)$, tal que $u \in \{V_3 \cup V_4\}$ e $v = 3$ ou $v = 5$ (Figura 3.2(c)). A aresta (3,7) é selecionada e na Figura 3.2(d) $T = \{(3, 5), (3, 7)\}$. O algoritmo procura o vértice de V_4 mais próximo de T e adiciona (7,11) à solução. O algoritmo retorna a árvore $\{(3, 5), (3, 7), (7, 11)\}$ ilustrada na Figura 3.2(e).

Também foi implementada uma versão randomizada da adaptação de Prim, RC3. Neste caso, a seleção dos vértices a cada iteração é feita com base na Lista de Candidatos Restrita, construída de forma semelhante ao que acontece com a adaptação de Kruskal. A lista é composta por todos os vértices cuja distância a T é menor que $d_{max} = c_{min} + \alpha_3(c_{max} - c_{min})$. Neste caso c_{min} e c_{max} são, respectivamente, a menor e a maior distância entre T e os vértices de grupos ainda não cobertos. O parâmetro α_3 tem a mesma função que α_2.

O pseudo-código de RC3 está apresentada no Algoritmo 6. Para cada grupo V_i uma solução T é construída. O conjunto K contém todos os vértices que podem ser selecionados na iteração corrente. A função `constroi_LCR()` insere na LCR os vértices de K que estão

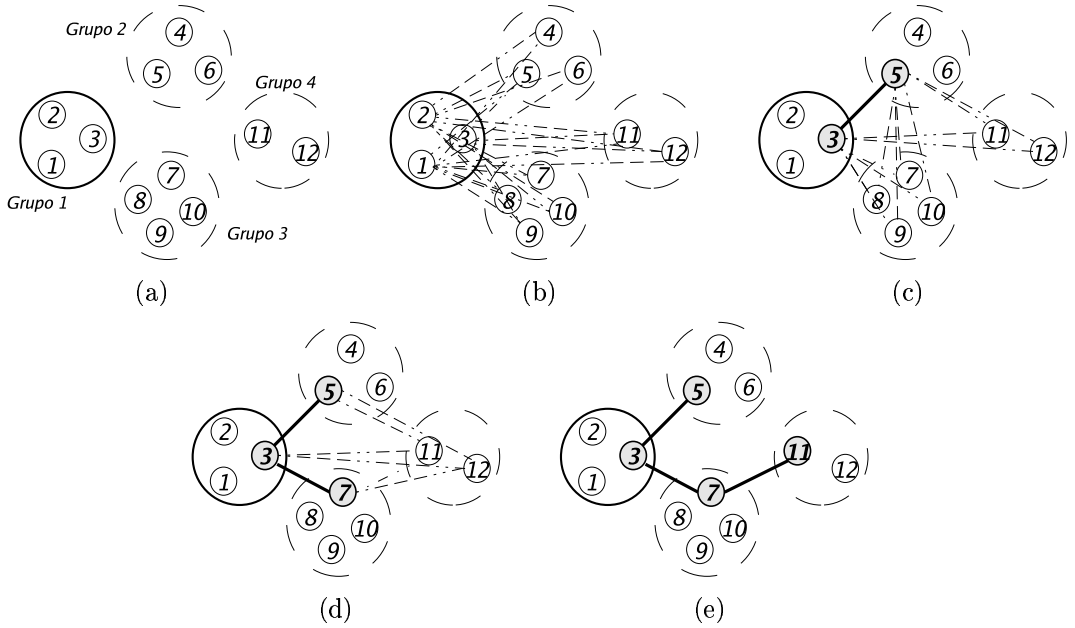


Figura 3.2: Ilustrando a heurística construtiva C3.

à distância máxima d_{max} de algum vértice de T. A função `rand()` escolhe aleatoriamente uma aresta $e(u, v) \in L$, em que u não é coberto por T e v o é. A partir da inserção de e em T, o grupo g_u passa a ser coberto pela solução, dado que g_u indica o índice do grupo de que u faz parte. Cada árvore T é concluída quando K não possui mais vértices, ou seja, quando $|T| = m - 1$. A função `c(`T`)` calcula $\sum_{e \in T} c_e$. O algoritmo retorna a melhor dentre as m soluções construídas.

Algoritmo 6 RC3 - Adaptação de Prim para o PAGMG

```
T* ← ∅;
para todo i ∈ {1, ..., m} faça
  T ← ∅;
  K ← V \ V_i;
  enquanto K ≠ ∅ faça
    L ← constroi_LCR(K,T, α_3);
    e(u, v) ← rand(L);
    T ← T ∪ e;
    K ← K \ V_{g_u};
  fim enquanto
  se c(T) < c(T*) então
    T* ← T;
  fim se
fim para
retornar T*;
```

3.1.1.4 C4 - Cálculo de Distância Média

O objetivo do algoritmo construtivo C4 é selecionar um vértice $\gamma[i]$ em cada grupo V_i levando em consideração os grupos a que V_i poderá estar conectado na solução. A seleção de $\gamma[i]$ é feita com base em sua distância média a um conjunto $S_i \subset \{V \setminus V_i\}$, dado que S_i é composto por vértices de grupos que se encontram a uma distância máxima D_{max} do grupo V_i.

Ao início da construção, define-se aleatoriamente uma ordem para percorrer os grupos. Em seguida, a cada grupo V_i percorrido, constrói-se o conjunto S_i e calcula-se a distância de cada vértice $v \in V_i$ aos elementos de S_i. A distância d_v de um vértice v a S_i é o custo médio das arestas entre v e os vértices presentes em S_i, ou seja, $d_v = \frac{\sum_{u \in S_i} c_{(u,v)}}{|S_i|}$. Em seguida, é selecionado o vértice $\gamma[i]$que minimiza d_v.

Na construção de S_i, considera-se todos os grupos cuja distância até V_i seja no máximo D_{max}. Nos grupos que já foram percorridos, integrarão S_i apenas os vértices que foram selecionados para participar da solução.

O funcionamento do algoritmo C4 é ilustrado na Figura 3.3. Considere a seguinte ordem em que os grupos são percorridos: 3, 4, 1, 2. Considerando também que os grupos 3 e 4 já foram percorridos, tem-se que $\gamma[3] = 7$ e $\gamma[4] = 11$. Na Figura 3.3(a), o grupo 1 é o próximo a ser percorrido. O conjunto S_1 é composto pelos vértices do grupo 2 e por $\gamma[3]$ (O grupo 4 não participa por estar a uma distância maior que D_{max} do grupo 1). Na Figura 3.3(c), calcula-se a distância de cada vértice de V_1 a S_1. O vértice 3 é selecionado porque $d_3 < d_1$ e $d_3 < d_2$.

Seguindo a ordem estabelecida, deve-se agora selecionar o vértice do grupo 2 que irá participar da solução. Os grupos que estão a D_{max} são 1 e 4. Como já estão estabelecidos os vértices $\gamma[1]$ e $\gamma[4]$, o conjunto S_2 será formado por apenas dois vértices (Figura 3.3(d)). As distâncias dos vértices de V_2 a S_2 são calculadas (Figura 3.3(e)) e o vértice 5 é fixado na solução. Como todos os grupos já foram percorridos, calcula-se a árvore de custo mínimo que cobre os vértices $\{3, 5, 7, 11\}$ e o algoritmo encerra retornando a solução ilustrada na Figura 3.3(f).

Foi também implementadoa uma versão randomizada de C4, denominada RC4, que seleciona os vértices aleatoriamente a partir de uma lista de candidatos. Nesse caso, cada grupo percorrido V_i possui uma LCR, composta de todos os vértices cuja distância d_v é menor ou igual a $d_{min} + \alpha_4(d_{max} - d_{min})$, onde d_{min} e d_{max} são a menor e a maior distância média dos vértices de V_i, respectivamente. Observe que se $\alpha_4 = 1$, o comportamento de

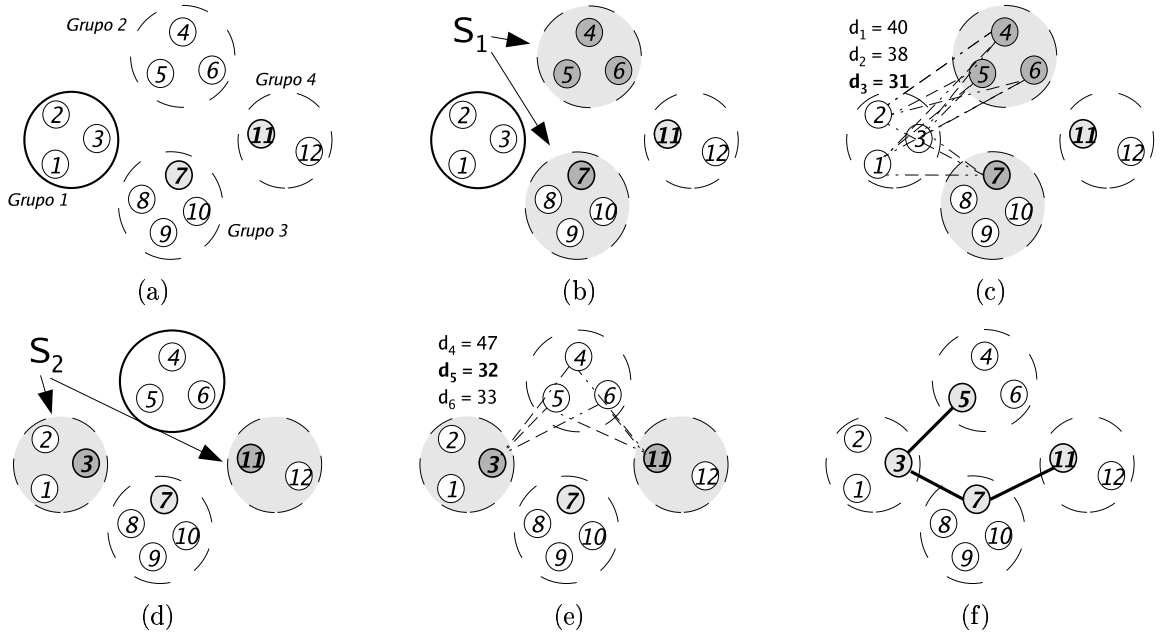


Figura 3.3: Ilustrando a heurística construtiva C4.

C4 será idêntico ao de C1.

O pseudo-código de RC4 está expresso no Algoritmo 7. Para cada grupo Vi, constrói-se o conjunto S_i e calcula-se a distância d_v de cada vértice. Em seguida, a lista de candidatos restrita é criada por `constroi_LCR()` com base nas distâncias mínima e máxima encontradas e no valor de α_4. O vértice $\gamma[i]$ é selecionado aleatoriamente pela função `rand()`. Ao final do algoritmo, computa-se a árvore geradora mínima T dos vértices presentes no vetor γ.

Algoritmo 7 RC4 - Cálculo de distâncias médias

para todo $i \in \{1, .., m\}$ **faça**
  $S_i \leftarrow$ `constroi_S(`V_i`,` D_{max}`)`;
  para todo $v \in V_i$ **faça**
    $d_v \leftarrow 0$;
    para todo $u \in S_i$ **faça**
      $d_v \leftarrow d_v + c_{(u,v)}$;
    fim para
    $d_v \leftarrow d_v/|S_i|$;
  fim para
  $L \leftarrow$ `constroi_LCR(`V_i`,` d`,` α_4`)`;
  $\gamma[i] \leftarrow$ `rand(`L`)`;;
fim para
$T \leftarrow$ `kruskal(`γ`,` $E(\gamma)$`)`;
retornar T;

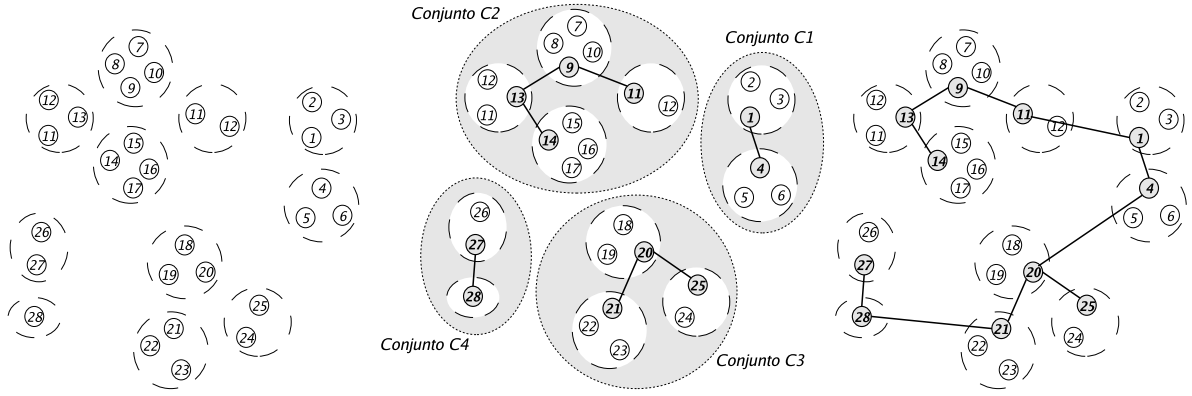
3.1.1.5 C5 - Método de Aglomeração

A heurística C5, baseada em conceitos de aglomeração, inicialmente divide os grupos em num_conj conjuntos e aplica o algoritmo construtivo C2 sobre cada um destes conjuntos. O problema é resolvido apenas dentro dos conjuntos, resultando em uma floresta com num_conj árvores. Após a aplicação de C2, o vértice que fará parte da solução já está definido em cada grupo. Assim, uma árvore geradora mínima é calculada sobre os vértices selecionados, gerando uma solução viável para o PAGMG. O valor de num_conj é tal que $1 < num\_conj < m$.

O conjuntos são construídos com base em num_conj grupos que atuam como sementes. Cada um dos $m - num\_conj$ grupos não-semente é associado à semente mais próxima.

A seleção das sementes é feita de duas formas, procurando selecionar os num_conj grupos mais distantes entre si (C5) e aleatoriamente (RC5). Em C5, o primeiro grupo θ_1 é escolhido aleatoriamente dentre os m grupos do grafo; o segundo grupo θ_2 é o mais distante de θ_1; o terceiro é aquele com maior distância média a θ_1 e θ_2, e assim sucessivamente. Em resumo, o grupo θ_i é o mais distante em média de $\{\theta_1, ..., \theta_{i-1}\}$ dentre os $m - i + 1$ grupos restantes. A distância entre dois grupos é o custo da menor aresta que os une.

A Figura 3.4 mostra um exemplo do funcionamento da heurística para $num\_conj = 4$. O grafo original, ilustrado em 3.4(a), contém 11 grupos que foram divididos em quatro conjuntos: $|C_1| = 2$, $|C_2| = 4$, $|C_3| = 3$ e $|C_4| = 2$. Aplicando a adaptação de Kruskal sobre o conjunto C_1 tem-se uma solução com os vértices $\{1, 4\}$; sobre C_2 tem-se $\{9, 11, 13, 14\}$; em C_3 a solução é composta por $\{20, 21, 25\}$; e em C_4 $\{27, 28\}$. O algoritmo resulta em uma árvore de custo mínimo que cobre todos esses vértices, apresentada em 3.4(c).



(a) Grafo original (b) Solução em cada conjunto (c) Solução final

Figura 3.4: Ilustrando a heurística construtiva C5.

O Algoritmo 8 apresenta o pseudo-código de RC5. A princípio, através da função `rand()`, seleciona-se as num_conj sementes aleatoriamente com o cuidado de não haver repetições. Em seguida, formam-se os conjuntos. A função `mais_proximo(`i`,` θ`)` retorna a semente mais próxima do grupo V_i. A adaptação de Kruskal é aplicada sobre cada conjunto, armazenando em γ_i o vértice selecionado em cada grupo V_i. Ao final, calcula-se uma árvore geradora mínima sobre os vértices em γ.

Algoritmo 8 RC5 - Método de Aglomeração

para todo $j \in \{1, .., num\_conj\}$ **faça**
  $\theta_j \leftarrow$ `rand(`m`)`;
  $C_j \leftarrow V_{\theta_j}$;
fim para
para todo $i \in \{1, .., m\}$ **faça**
  $j \leftarrow$ `mais_proximo(`i`,` θ`)`;
  $C_j \leftarrow C_j \cup V_i$;
fim para
para todo $j \in \{1, .., num\_conj\}$ **faça**
  `adapt_kruskal(`C_j`,` γ`)`;
fim para
$T \leftarrow$ `kruskal(`γ`,` $E(\gamma)$`)`;
retornar T;

3.1.2 Busca Local

A busca local implementada foi utilizada no PAGMG primeiramente por Golden et al. [15]. Dada uma solução inicial T, o procedimento pode ser descrito basicamente nas três etapas abaixo:

1. Aleatoriamente define-se uma ordem em que os grupos serão visitados
2. A cada visita a um grupo V_i, calcula-se todos os vizinhos de T em relação a V_i. A solução corrente passa a ser o melhor vizinho T', caso este seja melhor que T.
3. Repete-se o passo 2 até que m grupos sejam visitados sem que haja melhoras na solução corrente.

Sabendo que γ é o vetor de vértices associado à solução T e que γ' é o vetor associado à solução T', a vizinhança de T em relação ao grupo V_i são todas as soluções T' em que $\gamma[i]' \neq \gamma[i]$ e $\gamma[j]' = \gamma[j], \forall j \in \{1, .., m\} \setminus i$. Ou seja, em relação ao grupo V_i, T possui $|V_i| - 1$ vizinhos. Cada solução T' é construída a partir da árvore geradora mínima que cobre os vértices em γ'.

O pseudo-código está ilustrado no Algoritmo 9. Considere T uma solução inicial para a busca. A função `define_ordem()` define a ordem em que os grupos serão visitados. A cada iteração, `proximo_grupo()` retorna o próximo grupo a ser visitado segundo a ordem estabelecida. Calcula-se os vizinhos de T através da função `vizinho()` que retorna uma solução diferente de T apenas pelo vértice v, conforme o modelo de vizinhança já explicado. Para cada grupo visitado, a melhor solução T' é fixada na solução corrente, caso seja melhor que T.

Algoritmo 9 Busca Local

```
define_ordem();
iter ← 1;
enquanto iter ≤ m faça
  V_i ← proximo_grupo();
  para todo v ∈ V_i faça
    T' ← vizinho(T, v);
    se c(T') < c(T) então
      T ← T';
      iter ← 0;
    fim se
  fim para
  iter ← iter +1;
fim enquanto
retornar T';
```

3.1.3 Reconexão de Caminhos

O mecanismo de Reconexão de Caminhos, proposto originalmente para a busca tabu e *scatter search* por Glover [14] tem como objetivo encontrar soluções intermediárias entre duas boas soluções. O algoritmo parte de uma solução T_0, e passo-a-passo a transforma em outra solução T_d. Nesse trajeto, entende-se que pode ser encontrada uma solução melhor que T_0 e T_d. A solução T_0, de onde o procedimento inicia, é dita solução base e s_d, solução a que se pretende chegar, é a solução guia.

Se T_0 e T_d são duas soluções com d vértices diferentes entre si, um movimento de T_0 para T_d é a substituição de um vértice em T_0 por um vértice de T_d. Ou seja, tomando uma solução base uma solução T_0 e uma solução T_5, se há cinco vértices diferentes entre as duas soluções, há cinco possibilidades para a solução intermediária T_1.

A reconexão de caminhos procede da seguinte forma: partindo de T_0, um movimento para T_d é gerado e tem-se uma solução intermediária T_1; a seguir mais um movimento é gerado, de T_1 a T_d; e assim por diante até que se chegue a uma solução T_{d-1}, após $d-1$

movimentos realizados.

A Figura 3.5 ilustra um exemplo do funcionamento do mecanismo, nesse caso T_d é o grafo ilustrado na Figura 2.1. Como T_0 é composta pelos vértices $\{3, 6, 8, 12\}$ e a solução guia T_d pelos vértices $\{3, 5, 7, 11\}$, há três diferenças entre as duas soluções. Há, portanto, três candidatos para gerar T_1 (os vetores de vértices associados são $\{3, 6, 8, 11\}$, $\{3, 5, 8, 12\}$ e $\{3, 6, 7, 12\}$). A melhor opção é a troca do vértice 6 por 5. Na próxima iteração, os candidatos são T_1^1 (composta pelos vértices $\{3, 5, 8, 11\}$) e T_1^2 (composta por $\{3, 5, 7, 12\}$). A primeira opção é selecionada para ser T_2. Para o movimento seguinte, a única possibilidade é a troca do vértice 8 por 7. Com essa mudança, tem-se a solução guia T_3 e o procedimento termina.

As versões do GRASP que utilizam o mecanismo de reconexão de caminhos mantêm um conjunto elite CE contendo as CE_{max} melhores soluções distintas encontradas durante a execução. Esse conjunto é atualizado a cada iteração, caso a solução gerada pela busca local seja melhor que a pior solução em CE. Sempre o algoritmo atinge It_{max} iterações sem atualização do conjunto elite, são removidas todas as suas soluções, com exceção da melhor.

A reconexão de caminhos é ativada a partir da iteração It_{min}, e ocorre sempre que a solução T, resultante da busca local, for no máximo $p\%$ pior que a melhor solução do conjunto elite. Aplica-se o procedimento entre a T e a solução de CE que maximize d. A solução base é sempre a melhor dentre as duas. Como estratégia de intensificação, o valor de p é proporcional ao número de iterações sem atualização de CE, ou seja, quanto mais difícil for encontrar uma "boa"solução, mais próvavel ocorrer reconexão de caminhos.

Como esse procedimento é sempre aplicado entre soluções com um nível mínimo de qualidade, entende-se que as soluções intermediárias geradas no processo têm grande chance de serem boas soluções. Assim, toda solução intermediária T_i é comparada com as soluções em CE e se T_i for melhor que a pior solução elite, é-lhe aplicado uma busca local e CE é atualizado. Também as soluções candidatas passam por uma busca local. Isso ocorre independetemente de sua qualidade, com uma probabilidade de $q\%$, dado que q é a razão entre o número de iterações sem atualização do conjunto elite e It_{max}, ou seja, $q \leq 1$.

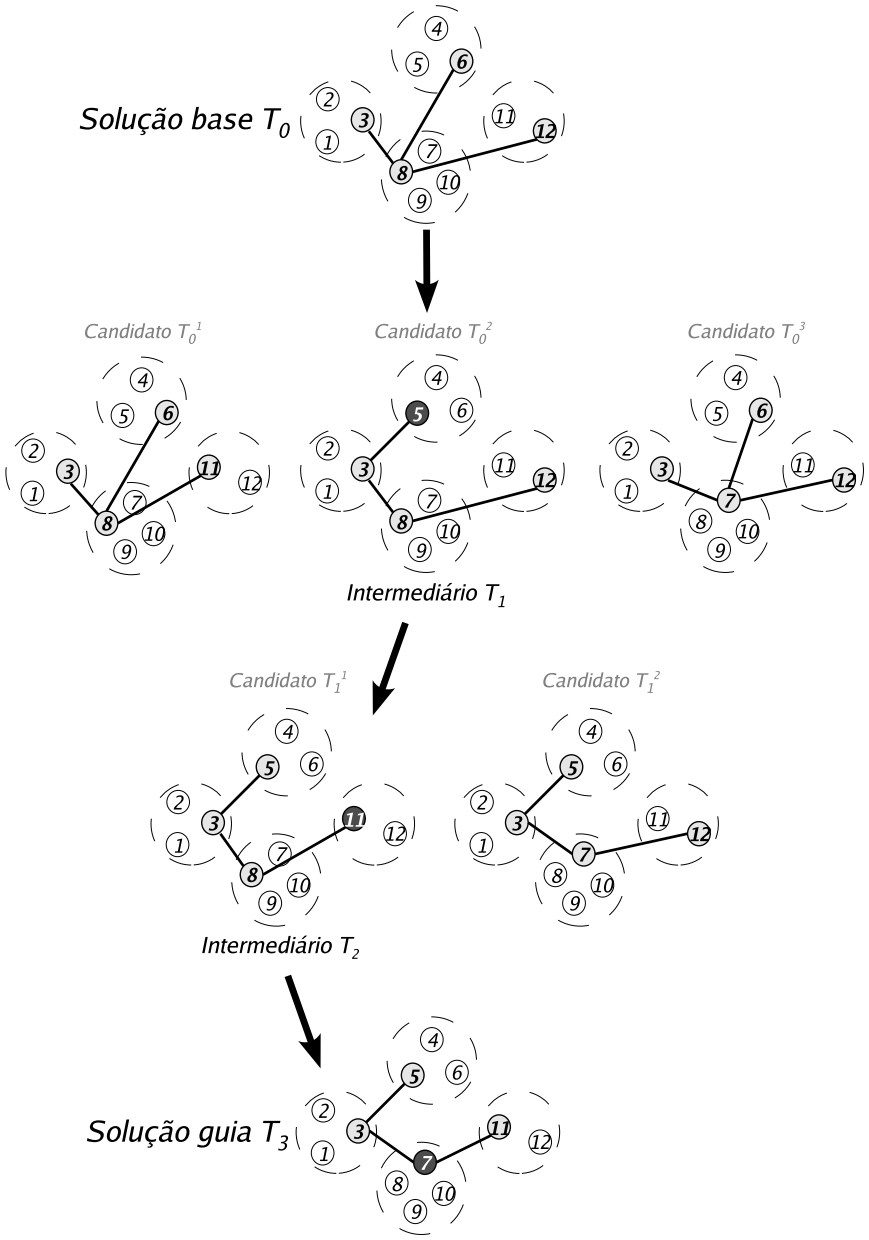


Figura 3.5: Ilustrando o mecanismo de reconexão de caminhos.

3.1.4 Busca Local Iterada

A idéia básica da Busca Local Iterada (*Iterated Local Search* - ILS) é aplicar pertubações sobre a solução corrente sempre que a mesma esbarrar em um ótimo local.

Besten et al. [3] citam quatro componentes básicos necessários para a implementação de um ILS: uma solução inicial, um procedimento de busca local, uma estratégia de perturbação e um critério de aceitação. A idéia básica será descrita a seguir: Aplica-se a busca local em uma solução inicial. A solução s gerada é tida como solução corrente. A seguir, s passa por uma perturbação seguida de uma busca local, gerando uma nova

solução s'. Se o critério de aceitação for satisfeito, s' passa a ser a solução corrente. O processo se repete até que um critério de parada seja atingido.

Percebe-se que a natureza e o tamanho da perturbação sobre s são fundamentais para o bom funcionamento da busca. Se for aplicada uma perturbação pequena, a busca local possivelmente retornará ao mesmo ótimo local s. Em oposto, se a perturbação for muito grande, corre-se o risco de se estar reiniciando a busca.

A estratégia de perturbação adotada neste trabalho tem como base um grupo V_i. Dado que $\gamma[i]$ é o vértice de V_i presente na solução corrente T, a perturbação seleciona aleatoriamente um vértice $v \in \{V_i \backslash \gamma[i]\}$ para compor a solução no lugar de $\gamma[i]$. O mesmo acontece com todos os grupos que se encontrem a uma distância máxima D_{max} de V_i. A solução proveniente dessa perturbação passa por uma busca local, gerando uma solução T'. O critério de aceitação é bastante simples: T' passa a ser a solução corrente se seu custo for menor que o de T. A busca local é a mesma descrita na Subseção 3.1.2.

O Algoritmo 10 apresenta o pseudo-código do procedimento. Considere uma solução T como dado de entrada. Ao início, `define_ordem()` define uma ordem de visita aos grupos. Dada essa ordem, a perturbação é aplicada em T por `pertubacao()` tomando como base o grupo V_i. A função `c(`T`)` retorna o custo de T. Se uma solução melhor for encontrada, T é atualizado. O processo se repete até que sejam atingidas m iterações sem melhora.

Algoritmo 10 Busca Local Iterada

```
define_ordem();
iter ← 1;
enquanto iter ≤ m faça
  V_i ← proximo_grupo();
  T' ← pertubacao(T, V_i, D_max);
  se c(T') < c(T) então
    T ← T';
    iter ← 0;
  fim se
  iter ← iter +1;
fim enquanto
```

Neste trabalho, a busca local iterada é aplicada sempre que o algoritmo atinge It_{max} iterações sem atualizar o conjunto elite. Sempre que uma solução melhor que a pior solução elite for encontrada, o conjunto elite é atualizado. No GRASP com busca local iterada, CE só é reiniciado se não for atualizado durante o ILS.

3.1.5 Versões GRASP Implementadas

Várias versões do algoritmo GRASP foram implementadas, com o objetivo de verificar a contribuição dos mecanismos adicionais de aprimoramento sobre a qualidade das soluções finais geradas. Essas versões diferem entre si pela utilização ou não de reconexão de caminhos e busca local iterada e pelo algoritmo construtivo utilizado.

Como RC4 apresentou melhor desempenho em relação às demais heurísticas, algumas versões GRASP utilizam apenas esse algoritmo na fase de construção. Porém, dentre os construtivos apresentados, não há um que predomine em todas as instâncias do PAGMG. Tal observação motivou a pesquisa para a implementação de versões GRASP que utilizem várias heurísticas para construir as soluções.

Essas versões, chamadas adaptativas, utilizam todos os algoritmos não gulosos descritos na Subseção 3.1.1 durante as iterações GRASP: C1, RC2, RC3, RC4 e RC5. O termo adaptativo é utilizado no sentido de que os algoritmos procuram se adequar às características da instância durante a execução. Essa adaptação será explicada a seguir.

Neste trabalho, as versões GRASP adaptativas têm como objetivo atribuir a cada construtivo uma quantidade de iterações proporcional ao seu desempenho. Portanto, cada heurística construtiva i é executada por um número β_i de iterações GRASP consecutivas, como uma fila circular. Se durante as β_i iterações de uma heurística o conjunto elite for atualizado, o valor de β_i é incrementado em uma unidade. No decorrer da execução, haverá uma tendência das heurísticas que propiciarem melhores soluções executarem por mais iterações. Nas versões que utilizam reconexão de caminhos, o valor de β_i é reiniciado para todas as heurísticas sempre que o conjunto elite é reiniciado. Isso é feito como uma forma de propiciar diversificação à busca.

As versões adaptativas também utilizam os algoritmos construtivos gulosos descritos na Subseção 3.1.1. Ao início da execução, cada construtivo guloso gera uma solução s_0, sobre a qual é aplicada uma busca local $m/10$ vezes, onde m é o número de grupos em que os vértices de G estão particionados. Como a busca parte de uma permutação dos grupos construída aleatoriamente, diferentes soluções podem ser geradas a partir de T_0.

A Tabela 3.1 apresenta as seis versões GRASP propostas no presente trabalho. G1 utiliza apenas o construtivo RC4 e nenhum outro mecanismo de aprimoramento, além da busca local. G2 e G3 utilizam o construtivo RC4 e reconexão de caminhos, diferindo pela utilização ou não de busca local iterada. Os algoritmos ditos adaptativos são G4, G5 e G6. G4 não tem mecanismos adicionais, G5 tem apenas reconexão de caminhos e G6

Versão	Construtivo	Reconexão de Caminhos	Busca Local Iterada
G1	RC4	Não	Não
G2	RC4	Sim	Não
G3	RC4	Sim	Sim
G4	Todos	Não	Não
G5	Todos	Sim	Não
G6	Todos	Sim	Sim

Tabela 3.1: Algoritmos GRASP propostos.

utiliza reconexão de caminhos e busca local iterada.

Para ilustrar o funcionamento das versões propostas, será tomado como exemplo o algoritmo G3. O pseudo-código está apresentado no Algoritmo 11. O algoritmo executa enquanto o critério de parada não for satisfeito. Cada iteração é composta por basicamente três fases: construção (através da heurística RC2), aprimoramento (busca local) e reconexão de caminhos. Após a busca local, a função `atualiza()` verifica se T pode fazer parte do conjunto elite, e se for o caso, CE é atualizado. A reconexão de caminhos só ocorre após a It_{min}-ésima iteração, e se a condição de qualidade de s em relação a CE for satisfeita. Quando o algoritmo atinge It_{max} iterações sem atualização, é aplicada uma busca local iterada (`ils()`) sobre cada solução do conjunto elite. A função `reiniciar(`CE`)` remove todas as soluções do conjunto elite, se o mesmo não for atualizado durante o ILS. O GRASP retorna a melhor solução em CE.

3.2 Geração de Cortes

Nesta seção será descrito o algoritmo de geração de cortes implementado para o cálculo de limites duais das instâncias do PAGMG. Esse algoritmo tem como base uma formulação para o Problema de Steiner em Grafos Direcionado. A seguir será explicado como as instâncias do Problema da Árvore Geradora Mínima Generalizado foram convertidas para instâncias do Problema de Steiner em Grafos.

3.2.1 Transformação das Instâncias

O Problema de Steiner em Grafos Direcionado difere do descrito na Seção 2.2 por ser definido sobre um grafo direcionado $G(V, A)$. Além disso, um vértice em T é fixado como raiz r da árvore. O objetivo então é encontrar uma árvore *com raiz em* r que cubra todos os vértices em T, de forma a minimizar a soma dos custos dos arcos.

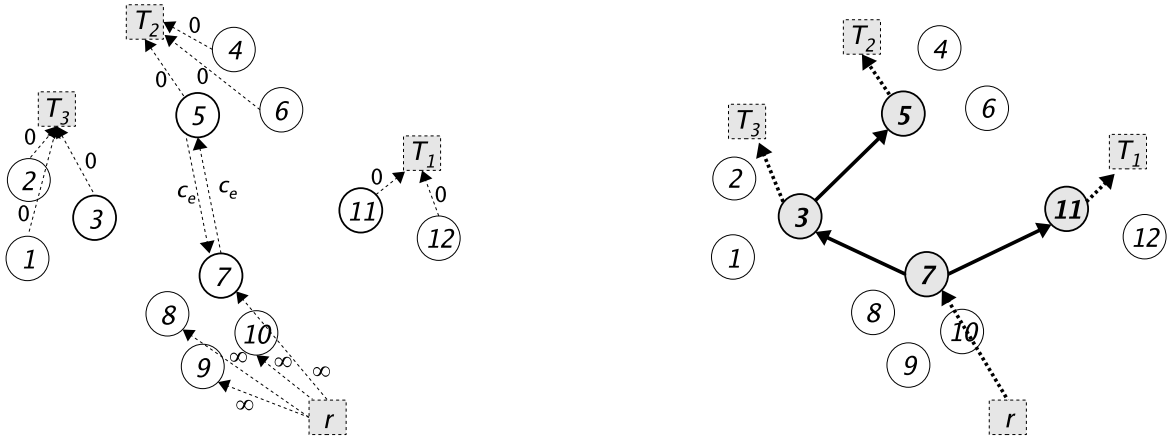
Algoritmo 11 GRASP G3

```
iter ← 1;
enquanto nao condicao_parada() faça
  T_0 ← constroi_solucao(RC4);
  T ← busca_local(T_0);
  se atualiza(CE, T) então
    iter ← 0;
  fim se
  se (iter > It_min) e (qualidade_minima(CE, T)) então
    reconexao_caminhos(CE, T);
  fim se
  se iter > It_max então
    para todo T ∈ CE faça
      ils(T);
    fim para
    reiniciar(CE);
  fim se
  iter ← iter +1;
fim enquanto
retornar melhor_solucao(CE);
```

A tranformação das instâncias é a mesma adotada nos trabalhos [17] e [5]. Nesses trabalhos, provou-se que as instâncias do L-GMSTP podem ser convertidas para instâncias do problema de Steiner, garantindo-se que a solução ótima dessas últimas são equivalentes às soluções ótimas das primeiras.

Considere uma instância do Problema da Árvore Geradora Mínima Generalizado, $G(V, E)$. Na transformação, um vértice terminal é inserido para cada grupo V_i. O terminal referente ao grupo com mais vértices será tido como raiz r. Além disso, um arco é inserido partindo de r para cada vértice no grupo referente, esses arcos terão custo infinito. Nos outros grupos, insere-se arcos que partem de cada vértice não-terminal e chegam ao terminal correspondente, dessa vez com custo zero. As arestas originais são substituídas por arcos correspondentes nas duas direções.

A Figura 3.6(a) ilustra um exemplo da transformação aplicada com base no grafo apresentado na Figura 2.1. Foram inseridos quatro vértices fictícios, que serão vértices terminais na instância do PSTGD. Por uma questão de clareza, apenas os arcos (5,7) e (7,5) estão ilustrados. O terminal do grupo $\{7, 8, 9, 10\}$ é definido como raiz. Através da Figura 3.6(b), é possível perceber como uma solução para o PSTGD pode ser convertida para uma solução do PAGMG.



(a) Instância (b) Solução

Figura 3.6: Instância do Problema de Steiner em Grafos Direcionado gerada a partir de uma instância do PAGMG.

3.2.2 Algoritmo de Geração de Cortes

O algoritmo é baseado na formulação *Directed Cut* do Problema de Steiner. Dado um conjunto de vértices $W \subset V$ tal que $r \notin W$ e $T \cap W \neq \emptyset$, a formulação será explicada a seguir.

$$\text{Minimizar} \sum_{a \in A} c_a \times x_a$$

Sujeito a:

$$\sum_{a(u,v):v \in W} x_a \geq 1, \quad \forall W \tag{3.1}$$

$$x_a \in \{0, 1\} \tag{3.2}$$

O conjunto de restrições 3.1 garante que, para todo corte $[W, V \setminus W]$, existe ao menos uma aresta na solução. As restrições 3.2 dizem respeito à integralidade das variáveis x. A função objetivo procura minimizar a soma dos custos das arestas presentes na solução.

Dada uma solução X para o Problema de Steiner em Grafos, uma forma de verificar a viabilidade de X é encontrar um conjunto W em que alguma das restrições em 3.1 esteja violada. Como W é um conjunto de vértices que possui ao menos um terminal, com exceção da raiz, se não existir uma aresta do corte $[W, V \setminus W]$ presente na solução, X está desconexo ou não possui algum terminal. A solução X é viável se e somente se $\forall W$ exista ao menos uma aresta do corte $[W, V \setminus W]$ presente na solução.

É fácil perceber que a quantidade de conjuntos W cresce exponencialmente com o aumento de $|V|$. Isso dificulta a resolução da relaxação linear dessa formulação se forem consideradas todas as restrições do conjunto 3.1. Por esse motivo, foi implementado um algoritmo de geração de cortes, que resolve problemas lineares com um pequeno conjunto de restrições, e em seguida procura restrições violadas pela solução gerada.

Ao início da execução, resolve-se o problema linear P referente à formulação apresentada, considerando apenas um conjunto trivial de restrições. A seguir, a partir da solução X_0 gerada por essa relaxação, é resolvido um problema de separação, procurando-se restrições não presentes em P e que são violadas por X_0. Tais restrições são incluídas e P é novamente resolvido, gerando uma nova solução X_1. O processo se repete até que não haja mais restrições violadas por X.

Na primeira iteração, são consideradas apenas as restrições referentes a $W = \{t\}$, $\forall t \in \{T \setminus r\}$. Logo, o primeiro problema linear possui $|T| - 1$ restrições.

A resolução do problema de separação é um dos elementos fundamentais para a eficiência do algoritmo. O tempo de conversão pode ser bastante reduzido se a cada iteração forem inseridos os cortes ideais. Logo, o algoritmo procura as restrições mais violadas pela solução corrente, ou seja, procura-se o conjunto W que maximize $1 - \sum_{a(u,v):v \in W} x_a$.

O problema de encontrar um conjunto W que corresponda às restrições mais violadas corresponde a um Problema de Corte Mínimo, que no algoritmo implementado é resolvido por meio do seu problema dual, o Problema de Fluxo Máximo. A resolução é feita através do algoritmo proposto por Orlin et al. [16]. Tal algoritmo foi escolhido porque encontra, em uma execução, todos os cortes referentes a cada vértice terminal.

O pseudo-código do algoritmo de geração de cortes está apresentado no Algoritmo 12. A função `resolver()` resolve o problema linear em P, que retorna uma solução possivelmente não viável X. Um problema de corte mínimo é gerado a partir de X e resolvido na função `separacao()`. O conjunto R contém os conjuntos W que correspondem aos cortes mais violados. As restrições correspondentes a R são adicionadas ao problema linear por meio do procedimento `adicionar()`. Um novo problema é gerado e resolvido. Esses passos se repetem até que não haja mais restrições violadas. O programa retorna a solução linear viável resultante.

Algoritmo 12 Geração de Cortes

```
X ← resolver(P);
repita
  R ← separacao(X);
  adicionar(P, R);
  X ← resolver(P);
até |R| = 0
retornar X;
```

3.3 Pré-processamento das Instâncias

O pré-processamento das instâncias é realizado por meio de regras de redução que tomam como base o conceito de Distância *Bottleneck* [34], utilizado para o Problema de Steiner em Grafos. Tais regras foram aplicadas com o objetivo de diminuir o tamanho das instâncias, removendo arestas que comprovadamente não fazem parte de ao menos uma solução ótima.

3.3.1 Distância *Bottleneck*

Dada uma instância do Problema de Steiner em Grafos, a Distância de Steiner $SD(P)$ de um caminho P é o comprimento do seu maior sub-caminho entre dois terminais. Por comprimento de um caminho, entende-se a soma dos custos de suas arestas. Assim, a distância *Bottleneck* entre dois vértices u e v é tida pela expressão:

$$B(u,v) = min\{SD(P)|P \in \mathcal{P}(u,v)\},$$

onde $\mathcal{P}(u,v)$ é o conjunto de todos os caminhos entre u e v. Esse valor pode ser interpretado como o "gargalo" das distâncias entre qualquer par de terminais em um caminho de u a v.

A Figura 3.7 ilustra parte de uma instância do Problema de Steiner. No exemplo, $\mathcal{P}(1,9)$ é composto pelos caminhos $\{(1,2),(2,5),(5,7),(7,10),(10,9)\}$, $\{(1,9)\}$ e $\{(1,3),(3,4),(4,8),(8,9)\}$, cujas distâncias de Steiner são, respectivamente, 50, 100 e 70. Logo, $B(1,9) = min\{50,100,70\} = 50$.

A distância *Bottleneck* $B(u,v)^{-(u,v)}$ entre os vértices u e v *que não passa pela aresta* (u,v) é definida como:

$$B(u,v)^{-(u,v)} = min\{SD(P), P \in \mathcal{P}(u,v); (u,v) \notin P\}$$

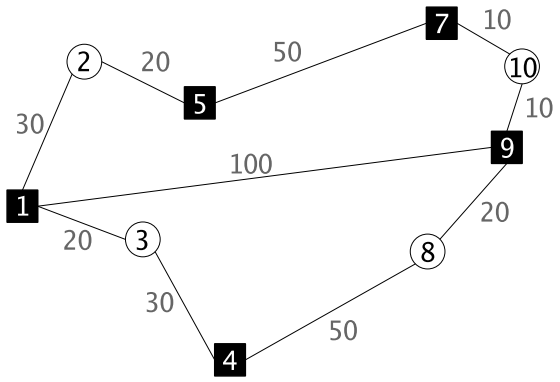


Figura 3.7: Distância *Bottleneck* em uma instância do Problema de Steiner em Grafos.

Se $B(u,v)^{-(u,v)} \leq c_{(u,v)}$, então (u,v) é dita redundante. Ou seja, existe ao menos uma solução ótima em que (u,v) não está presente. No exemplo da Figura 3.7, $B(1,9)^{-(1,9)} = 50$ e (1,9) é redudante. Por outro lado, dentre os caminhos de 5 a 7 que não passam pela aresta $(5,7)$, a menor Distância de *Steiner* é 70 (custo do sub-caminho $\{(4,8),(8,9)\}$). Logo $B(5,7)^{-(5,7)} = 70 > c_{(5,7)}$, conseqüentemente a aresta (5,7) não pode ser removida. Se $\{(u,v)\}$ é o único caminho de u a v, então $B(u,v)^{-(u,v)} = \infty$.

3.3.2 Distância *Bottleneck* no PAGMG

O conceito de distância *Bottleneck* foi redefinido para que pudesse ser aplicado na redução de instâncias do PAGMG. Dada uma instância do problema, considere $\mathcal{P}(u,v)$ o conjunto de todos os caminhos de u a v compostos apenas por vértices presentes na solução ótima (além de u e v). A distância $B(u,v)^{-(u,v)}$ pode então ser redefinida para a expressão:

$$B(u,v)^{-(u,v)} = min\{C(P), P \in \mathcal{P}(u,v); (u,v) \notin P\},$$

onde $C(P)$ é o custo da maior aresta do caminho P.

Teorema 1 *Dada uma instância do Problema da Árvore Geradora Mínima Generalizado $G = (V,E)$, cuja solução ótima é uma árvore geradora mínima $T = (V',E')$, se $B(u,v)^{-(u,v)} \leq c_{(u,v)}$ então $(u,v) \notin E'$.*

Prova Suponha-se que $B(u,v)^{-(u,v)} \leq c_{(u,v)}$ e $(u,v) \in E'$. Como $B(u,v)^{-(u,v)} \leq c_{(u,v)}$, existe uma aresta (w,z) em um caminho entre u e v, tal que $c_{(w,z)} \leq c_{(u,v)}$ e $w,z \in V'$. Logo, substituindo (u,z) por (w,z) em E', tem-se uma árvore com custo menor que T, o que é absurdo. Conclui-se portanto que se $B(u,v)^{-(u,v)} \leq c_{(u,v)}$, então $(u,v) \notin T$. □

Na redução de instâncias deste trabalho, a remoção de cada aresta (u, v) foi realizada por meio da constatação da existência de ao menos um caminho $P \in \mathcal{P}(u, v)$ tal que $C(P) \leq c_{(u,v)}$. Se P existe, (u, v) é redundante pois $B(u, v)^{-(u,v)} \leq C(P) \leq c_{(u,v)}$.

Na Figura 3.8, é ilustrada a aresta (u, v) e o conjunto de arestas E'', composto por todas as arestas entre u e V_1, entre V_k e v e entre os vértices de cada par de grupos $\{V_i, V_{i+1}\}$, $1 \leq i \leq k-1$. Como em cada grupo há um vértice que faz parte da solução ótima, se $c_e \leq c_{(u,v)} \forall e \in E''$, então $\exists P$ tal que $C(P) \leq c_{(u,v)}$ e (u, v) é redundante. O mesmo é válido para $1 \leq k \leq m-2$.

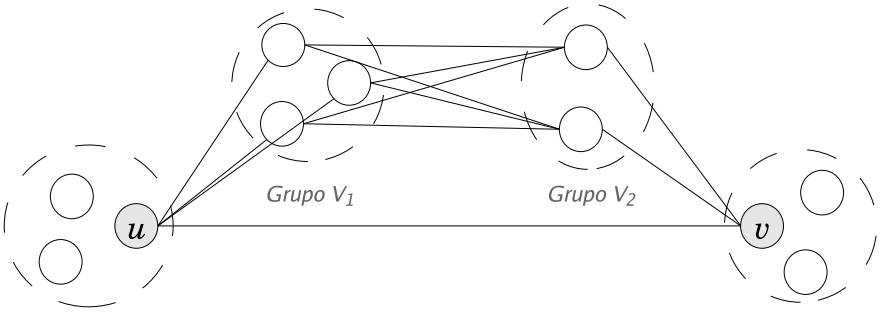


Figura 3.8: Conjunto de arestas analisadas no pré-processamento das instâncias.

O tempo para a detecção de P cresce exponencialmente em relação ao valor de k. Por esse motivo, no pré-processamento de instâncias realizado neste trabalho, o algoritmo de redução considera apenas os casos em que $k = 1$ e $k = 2$. Primeiramente são aplicadas as regras referentes a $k = 1$. Cada aresta (u, v) é tida como redundante se houver um grupo $V_i, i \in \{1, ..., m\}$ em que todas arestas (u, v_i) e (v_i, v) possuam custo menor ou igual a $c_{(u,v)}$, $\forall v_i \in V_i$.

A partir da instância já reduzida com a primeira fase, aplicam-se as regras referentes a $k = 2$. Nesse caso, uma aresta (u, v) é dita redundante se houver dois grupos V_i e V_j tal que $c_{(u,v)}$ é maior ou igual ao custo de cada aresta contida em $\{(u, v_i), (v_i, v_j), (v_j, v)\}$, $\forall v_i \in V_i, \forall v_j \in V_j$.

Neste capítulo foram apresentados os algoritmos implementados para o Problema da Árvore Geradora Mínima Generalizado. Como propostas heurísticas, foram implementados algoritmos construtivos propostos neste trabalho e na literatura. Foram também implementados procedimentos de melhoramento, como busca local, reconexão de caminhos e busca local iterada. A partir desses algoritmos, foram desenhadas seis versões da heurística GRASP. Também foi implementado um algoritmo de geração de cortes que tem como base uma formulação para o Problema de Steiner em Grafos e um procedimento de

redução de arestas utilizado como pré-processamento.

Capítulo 4

Resultados Computacionais

Neste capítulo serão apresentados os resultados experimentais obtidos com os algoritmos apresentados no Capítulo 3. Primeiramente serão descritas as instâncias utilizadas nos testes e a redução obtida por meio do seu pré-processamento; em seguida, apresentam-se os resultados do algoritmo de geração de cortes; por fim, seguem os testes realizados com as seis versões GRASP propostas.

Os testes computacionais foram realizados em um computador Pentium IV, com 3.2GHz e 1GB de memória principal, rodando um sistema operacional Linux versão 2.6.8-1.521. Os programas foram implementados na linguagem de programação C++ e compilados com g++ versão 4.0.2 utilizando as opções de compilação -o3 e march=pentium4.

4.1 Instâncias

A fim de facilitar o entendimento do trabalho, as instâncias foram divididas em dois grupos:

Grupo 1 Composto por 169 instâncias com $48 \leq |V| \leq 226$. Foram geradas por Fischetti et al. [12] para o Problema do Caixeiro Viajante Generalizado e introduzidas no PAGMG por Feremans et al. [8].

Cinco instâncias desse grupo (15spain47, 27europ47, 50gr96africa, 35gr137america e 34gr202europe) representam a disposição geográfica real de cidades e seu agrupamento segue a divisão natural das cidades em regiões. As demais instâncias do Grupo 1 foram adaptadas do repositório do Problema do Caixeiro Viajante, TSPLIB [30].

Dentre as 169 instâncias, 150 possuem valor ótimo conhecido. O melhor limite dual

para as 19 restantes foi apresentado por Oncan et al. [24].

Grupo 2 Composto por 101 instâncias com $229 \leq |V| \leq 783$, introduzidas por Oncan et al. [24]. Tais instâncias também são adaptadas do TSPLIB. Não há limites duais na literatura para essas instâncias.

Nas instâncias geradas a partir do TSPLIB, o agrupamento dos vértices foi realizado por meio de duas técnicas: *Cluster Centering* e *Grid Clusterization*, descritas a seguir.

Em *Cluster Centering*, os vértices são agrupados em $m = \lceil |V|/5 \rceil$ grupos. Primeiramente m vértices são selecionados como centro de cada grupo. Para tanto, o primeiro centro é selecionado aleatoriamente, o segundo é o mais distante do primeiro, o terceiro é o mais distante dos outros dois, e assim sucessivamente. Em seguida, cada um dos vértices restantes é associado ao centro mais próximo.

A técnica *Grid Clusterization* foi aplicada apenas em instâncias cujas coordenadas são conhecidas. O plano cartesiano é dividido de forma a gerar uma grade de dimensões $NG \times NG$, onde cada célula corresponde a um grupo de vértices. O número NG é o menor inteiro tal que os grupos contenham no mínimo um e no máximo $|V|/\mu$ vértices. O parâmetro μ expressa o número aproximado de vértices por grupo da instância.

A Tabela 4.1 apresenta o número de instâncias geradas por cada técnica de agrupamento. A segunda coluna é referente às instâncias cujo agrupamento segue a disposição geográfica dos vértices.

Grupo	Geo	*Cluster Centering*	*Grid Clusterization*				Total
			$\mu = 3$	$\mu = 5$	$\mu = 7$	$\mu = 10$	
1	5	36	32	32	32	32	169
2	-	21	20	20	20	20	101
Total	5	57	52	52	52	52	270

Tabela 4.1: Número de instâncias geradas por cada técnica de agrupamento.

4.1.1 Testes de Redução

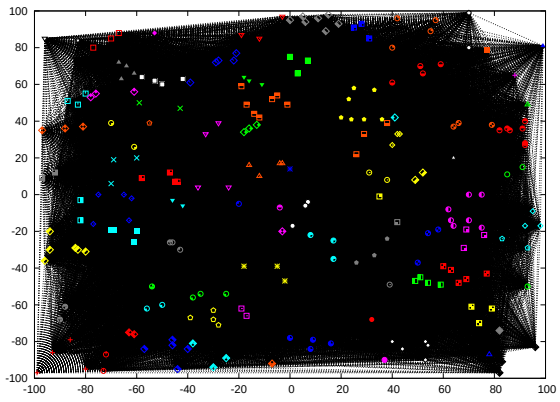
Os testes de redução foram aplicados em todas as instâncias consideradas neste trabalho, e propiciaram a redução do número de arestas a 14,71% do original, em média. A aplicação das regras considerando o parâmetro $k = 1$ resultou na remoção de 84,73% das arestas. Com a aplicação seguinte, com $k = 2$, foi possível remover 0,56% das arestas em relação à quantidade original.

Das 270 instâncias, 115 reduziram sua quantidade de arestas a menos de 10% do

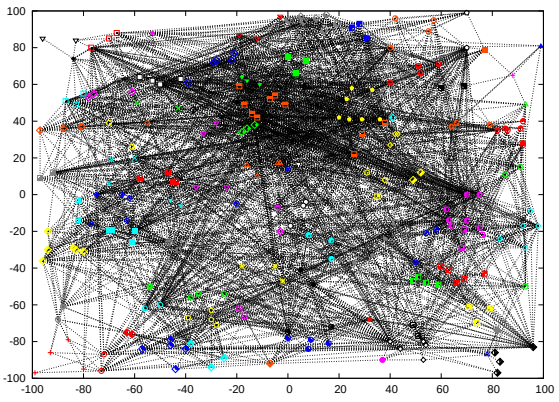
original e apenas 13 delas permaneceram com mais de 40% de suas arestas. O tempo de execução do pré-processamento é de 2,04s em média, e chegou a no máximo 20,45s.

Foi observado que os testes obtiveram maior impacto nas instâncias com maior discrepância entre as distâncias entre seus vértices. Ou seja, as arestas redundantes podem ser mais facilmente detectadas quando as instâncias possuem alto desvio padrão em relação ao custo das arestas.

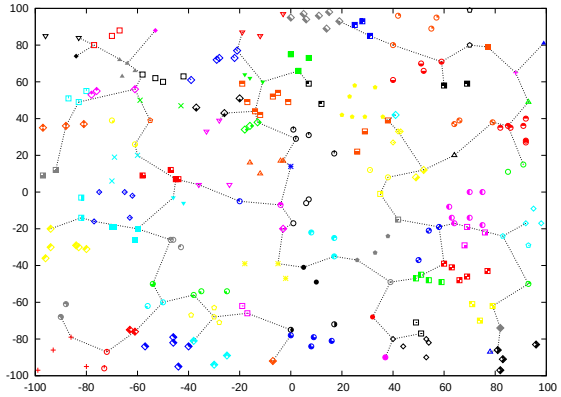
Através da Figura 4.1, é possível perceber o impacto da redução sobre as instância 95gil262, cujos vértices foram agrupados pela técnica *Grid Clusterization* com $\mu = 3$. As Figuras 4.1(a) e 4.1(b) apresentam a instância antes e depois do pré-processamento, respectivamente. O pré-processamento propiciou a redução do número de arestas a $3,55\%$ do original. A solução ótima é apresentada na Figura 4.1(c).



(a) Sem pré-processamento (100% das arestas)



(b) Com pré-processamento (3,55% das arestas)



(c) Solução ótima

Figura 4.1: Instância 95gil262 gerada por *Grid Clusterization* com $\mu = 3$.

As Tabelas A.1 a A.10 no Apêndice A apresentam as características das instâncias utilizadas neste trabalho. A primeira coluna indentifica as instâncias, no formato $mXXXn$, onde m é o número de grupos da instância, XXX é o seu nome e $n = |V|$;

as três colunas seguintes trazem o número de vértices, arestas e grupos, respectivamente; na quinta coluna tem-se o número de arestas resultante do pré-processamento, seguido da porcentagem de arestas restantes em relação ao número original e do tempo gasto com a redução; as duas últimas colunas apresentam os resultados do algoritmo de geração de cortes.

4.2 Geração de Cortes

O algoritmo de geração de cortes descrito na Seção 3.2 foi implementado em linguagem C++, utiliza a ferramenta CPLEX versão 10.1 como resolvedor linear e foi compilado com g++ versão 4.0.2. Os testes aqui apresentados foram realizados sobre as instâncias que passaram pelo pré-processamento descrito anteriormente, em uma máquina Pentium IV 3,2 GHz com 2GB de memória principal.

Os resultados estão apresentados nas Tabelas A.1 a A.10, em anexo. As duas últimas colunas de cada tabela indicam o limite encontrado e o tempo de execução do algoritmo para cada instância.

Dentre as instâncias do primeiro grupo, o algoritmo encontrou a solução ótima das 150 instâncias cujo ótimo é conhecido, em um tempo de no máximo 170s. Tais resultados foram previamente encontrados pelo algoritmo *branch-and-cut* de Feremas et al. [7], entretanto o tempo de execução não pode ser diretamente comparado pelo fato de que as máquinas possuem arquiteturas diferentes. O *branch-and-cut* executou em uma máquina Sun sparc com 360MHz e seu pior tempo foi de 5254s para a instância 40kroa200.

A Tabela 4.2 apresenta os resultados para as 19 instâncias do Grupo 1 e que se conhecia apenas o limite inferior. A primeira coluna apresenta o nome das instâncias, seguido do número de vértices, arestas e grupos, respectivamente; a quarta coluna indica o limite encontrado no trabalho de Oncan et al. [24]; as duas últimas colunas trazem o limit encontrado pela geração de cortes e seu tempo de execução. Pode-se observar que os limites são iguais para quase todas as instâncias, com exceção da instância 45ts225, em que se observa uma pequena melhora do algoritmo aqui proposto. Não é conhecido o tempo de execução do algoritmo de Oncan.

O desvio percentual relativo (gap) entre o melhor custo conhecido para as instâncias 45ts225 (gerada por *Cluster Centering*), 45ts225 (gerada por *Grid Clusterization*) e 31gr202 em relação aos limites encontrados é de 0,04%, 0,02% e 0,5%, respectivamente. O teto do limite da instância 68gr202 resulta em um valor igual ao custo da sua melhor

solução conhecida. Nas demais instâncias, o gap entre o custo da melhor solução conhecida e o limite dual apresentado é zero. Ou seja, para 16 das 19 instâncias, o custo da solução ótima foi obtido por meio do limite dual encontrado.

Instância	$\|V\|$	$\|E\|$	m	LB (Oncan)	Geração de Cortes	
					LB	Tempo (s)
Cluster centering						
40d198	198	2371	40	**7044**	**7044**	21,53
41gr202	202	2595	41	**242**	**242**	26,18
45ts225	225	1834	45	**62246**	**62246**	66,26
46pr226	226	1522	46	**55515**	**55515**	165,55
Grid Clusterization com $\mu = 3$						
67d198	198	1375	67	**8283**	**8283**	10,25
68gr202	202	1926	68	**292,5**	**292,5**	41,65
75ts225	225	900	75	**79019**	**79019**	3,79
84pr226	226	891	84	**62527**	**62527**	9,7
Grid Clusterization com $\mu = 5$						
40d198	198	2675	40	**7098**	**7098**	15,67
41gr202	202	6300	41	**232**	**232**	176,59
45ts225	225	2945	45	60578	***60578,7**	269,2
50pr226	226	1947	50	**56721**	**56721**	40,81
Grid Clusterization com $\mu = 7$						
32d198	198	3222	32	**6501**	**6501**	44,41
31gr202	202	5184	31	**202**	**202**	49,76
35ts225	225	2649	35	**50813**	**50813**	55,45
33pr226	226	1890	33	**48249**	**48249**	220,32
Grid Clusterization com $\mu = 10$						
25d198	198	4003	25	**6185**	**6185**	246,9
21gr202	202	9052	21	**177**	**177**	194,38
25ts225	225	4180	25	**40339**	**40339**	152,51

Tabela 4.2: Comparação entre os limites duais encontrados com os da literatura.

O algoritmo de geração de cortes encontrou limites duais para 82 dentre as 101 instâncias do segundo grupo. Entretanto, o tempo necessário para esse resultado foi bem acima do tempo das instâncias do primeiro grupo. Em 65 instâncias, o limite encontrado equivale ao custo da melhor solução conhecida. O gap médio do melhor custo conhecido das 82 instâncias em relação aos seus limites duais é de 0,013%.

4.3 GRASP

Nesta seção estão descritos os resultados de testes realizados com as seis versões da heurística GRASP propostas neste trabalho. Os testes possuem dois objetivos: comparação

entre as propostas e comparação com algoritmos da literatura.

Nos testes de comparação entre as versões propostas, primeiro foi feita uma comparação do desempenho dos algoritmos construtivos. Foi também realizada uma comparação direta entre resultados das versões GRASP, além de uma análise probabilística dessas versões considerando um conjunto restrito de instâncias. Os resultados das versões GRASP são referentes a instâncias que passaram pelo pré-processamento descrito na Seção 3.3. O tempo de pré-processamento foi contabilizado no tempo de execução dos algoritmos.

A comparação com os algoritmos da literatura focou em comparar a versão GRASP que apresentou melhores resultados, G6, com o algoritmo de busca tabu proposto por Oncan et al. [24], que mantém os melhores resultados da literatura para as instâncias consideradas neste trabalho.

Para a análise probabilística e comparação com a literatura foram utilizadas apenas as instâncias do Grupo 2, por permitirem uma melhor análise do comportamento dos algoritmos. A seguir, será apresentada a calibragem dos parâmetros dos algoritmos propostos.

4.3.1 Calibragem de Parâmetros

Todos os testes para calibragem de parâmetros foram realizados com 10 instâncias do Grupo 2, duas para cada tipo de agrupamento. As instâncias foram selecionadas de forma que houvesse diversidade em termos de número de vértices e grau de dificuldade.

O valor do parâmetro α das heurísticas construtivas RC2, RC3 e RC4 foi definido por meio de 100 execuções para valores entre 0,05 e 0,9. No construtivo RC2, o valor $0,05$ apresentou melhores resultados para praticamente todas as instâncias. Em relação aos demais construtivos, não houve predominância de um valor específico, os melhores resultados, em média, foram obtidos para $0,05 \leq \alpha \leq 0,3$. Portanto, em todas as versões do GRASP, o valor de α é selecionado aleatoriamente no intervalo de [0,05; 0,3] antes de cada execução de RC3 e RC4.

Vale ressaltar que o valor pequeno de α em RC2 se deve à natureza dos elementos de sua LCR. Por ser composta por arestas, a lista tende a ficar demasiadamente grande com o aumento do valor de α, o que descaracteriza o comportamento do algoritmo em relação ao construtivo guloso C2.

A distância D_{max} para os construtivos C4 e RC4 foi testada entre os valores $\overline{c}$, $\overline{c}/2$,

$\overline{c}/3$ e $\overline{c}/4$, onde $\overline{c}$ é o custo médio das arestas da instância. O valor do parâmetro foi fixado como $\overline{c}/3$, por apresentar melhores resultados para quase todas as instâncias.

O número de conjuntos das heurísticas C5 e RC5 foi determinado também por 100 execuções para cada instância. Os valores testados foram $m/3$, $m/5$, $m/7$ e $m/10$, onde m é o número de grupos da instância. As melhores soluções foram obtidas com num_conj= $m/7$ em quase todos os casos. O valor de num_conj é então selecionado aleatoriamente no intervalo $[\lceil m/7 \rceil - 2; \lceil m/7 \rceil + 2]$ antes de cada execução da heurística.

Os parâmetros It_{min}, CE_{max}, It_{max} e p foram estabelecidos por meio de execuções do G5 com um tempo de execução máximo estabelecido.

O tamanho do conjunto elite CE_{max} foi definido por testes com valores entre 4 e 8. Para cada valor foram realizadas 30 execuções. Os resultados com os valores 4 e 5 superaram levemente os demais. Também foi verificado que quanto menor o tamanho do conjunto elite, o algoritmo tende a executar mais rápido. Assim, foi estabelecido $CE_{max} = 4$.

Foi verificado que, em geral, o conjunto elite começa a comportar boas soluções a partir da vigésima iteração. O valor de It_{min} foi então estabelecido como 20. Entretanto, é importante salientar que o melhor valor para It_{min} depende muito das características da instância, pois a qualidade média das soluções geradas pela busca local varia muito de instância para instância.

A quantidade de atualizações do conjunto elite tende a descrecer seguindo uma função logarítmica no decorrer do algoritmo. A partir de 50 iterações sem atualização, o conjunto elite começa a estagnar e as atualizações praticamente deixam de existir. Por esse motivo, foi estabelecido que $It_{max} = 50$.

Em instâncias em que os algoritmos facilmente encontram soluções de boa qualidade, a reconexão de caminhos tende a ocorrer excessivamente, prejudicando a eficiência do algoritmo. Por outro lado, em instâncias em que a qualidade média das soluções é baixa, a reconexão de caminhos raramente é aplicada. Por esse motivo, foi determinando que o valor de p deve ser proporcional a `iter`, o número de iterações sem atualização do conjunto elite. Ficou estabecelecido que $p = 1 +$ `iter`, assim a aplicação da reconexão de caminhos se torna mais provável quando o algoritmo tem dificuldades em atualizar o conjunto elite.

A Tabela 4.3 apresenta a relação dos valores fixados para cada parâmetro dos algoritmos.

Parâmetro	Valor
α (RC2)	0,05
α (RC3 e RC4)	[0,05;0,3]
D_{max}	$\overline{c}/3$
num_conj	$[\lceil m/7 \rceil - 2; \lceil m/7 \rceil + 2]$
It_{min}	20
CE_{max}	4
It_{max}	50
p	$1 +$ `iter`

Tabela 4.3: Configuração dos parâmetros do algoritmos.

4.3.2 Heurísticas Construtivas

A bateria de testes apresentada nesta subseção teve como objetivo comparar o desempenho dos algoritmos construtivos não-gulosos. Através desses testes também foi possível verificar o impacto da busca local sobre as soluções geradas por cada heurística.

Os testes foram realizados em duas etapas. Na primeira etapa, cada algoritmo construtivo foi executado 100 vezes para cada instância. Na segunda, foi aplicada uma busca local sobre cada uma das soluções geradas na etapa anterior. Nas duas etapas foram consideradas todas as instâncias dos dois grupos.

A Tabela 4.4 apresenta a comparação entre os algoritmos e está dividida em duas partes: na primeira estão os resultados das soluções geradas pelos construtivos; e na segunda estão os resultados da aplicação da busca local sobre aquelas soluções. Na coluna 1, tem-se o nome do algoritmo; a segunda coluna apresenta a média do custo médio obtido para cada instância; na terceira coluna tem-se o gap médio do custo das soluções em relação ao melhor custo conhecido; a coluna 4 apresenta a soma do tempo médio gasto com cada instância; por fim, a última coluna indica a porcentagem de instâncias em que cada algoritmo conseguiu o melhor custo médio.

Na primeira parte da tabela, é possível perceber que o algoritmo RC2 obtém melhores soluções que os demais para boa parte das instâncias. Como esperado, C1 gera soluções com qualidade muito inferior, e não atingiu o melhor custo médio para nennhuma instância. O tempo de execução das adaptações de Kruskal e Prim é bem maior que o dos demais algoritmos. O menor tempo de execução é o do construtivo C1.

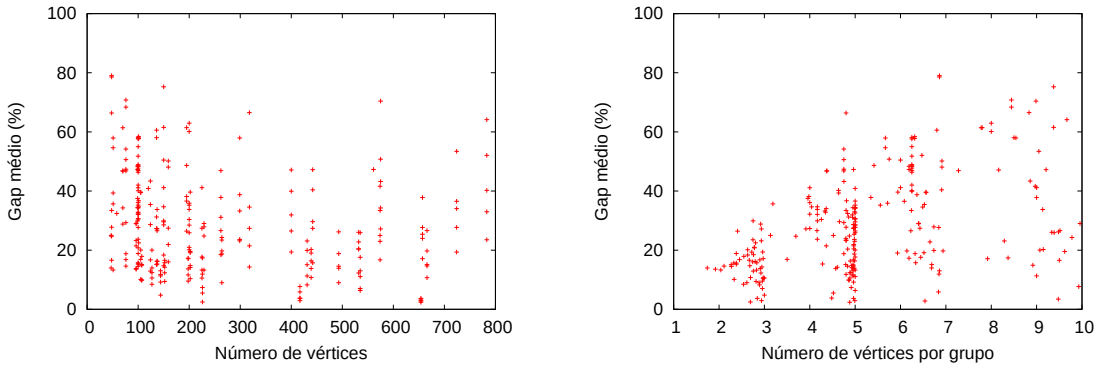
Não foram identificadas diferenças no comportamento do algoritmos entre as instâncias do Grupo 1 e do Grupo 2. Em geral, os melhores construtivos para instâncias com poucos vértices também o são para as instâncias maiores.

Heurística	Custo Médio	Gap Médio	Tempo Total	Instâncias (%)
C1	24509,09	28,47	0,01	0
RC2	**21930,4**	**9,86**	**26,83**	**71,11**
RC3	22392,24	12,12	25,46	18,8
RC4	23013,92	16,21	0,1	2,96
RC5	22216,49	11,94	0,07	7,03
C1 + BL	20798,45	2,27	15,97	8,88
RC2 + BL	20788,64	2,26	41,64	29,25
RC3 + BL	20773,00	2,04	41,24	24,81
RC4 + BL	**20769,45**	**1,81**	**15**	**29,62**
RC5 + BL	20804,17	2,27	14,56	13,33

Tabela 4.4: Comparação entre os algoritmos construtivos.

Com a aplicação da busca local sobre as soluções, percebe-se que a diferença entre o desempenho dos algoritmos diminui. Os construtivos RC2, RC3 e RC5 apresentam melhores custos para boa parte das instâncias e o gap médio do construtivo RC4 é levemente superior aos demais. Os tempos de execução também se tornam mais próximos, mas os algoritmos RC2 e RC3 continuam com os maiores tempos.

Foi verificado que a forma de agrupamento dos vértices exerce mais influência sobre a qualidade das soluções que o número de vértices propriamente dito. O gap do custo das soluções geradas em relação ao melhor custo conhecido tende a ser maior nas instâncias que apresentam mais vértices por grupo. Por esse motivo, os algoritmos construtivos geram soluções melhores para as instâncias geradas por *Grid Clusterization* com $\mu = 3$.



(a) (b)

Figura 4.2: Relação entre características das instâncias e qualidade das soluções geradas pelo construtivo C1.

Esse fato pode ser verificado pelos gráficos da Figura 4.2. Nos gráficos, cada ponto representa uma das 270 instâncias consideradas neste trabalho. O eixo Y apresenta o gap médio das soluções geradas pelo construtivo C1. Na Figura 4.2(a), o eixo X indica

o número de vértices e na Figura 4.2(b) tem-se o número de vértices por grupo das instâncias.

Pode-se perceber que não há uma relação clara entre o número de vértices e o gap médio de cada instância. Entretanto, esse gap tende a crescer com o aumento do número de vértices por grupo. Em instâncias com menos de três vértices por grupo, o gap não ultrapassa 40%. O comportamento é muito parecido para os outros construtivos testados.

4.3.3 Comparação de Versões

As versões da heurística GRASP propostas foram comparadas por meio de testes realizados com todas as instâncias dos dois grupos. O objetivo aqui é verificar qual das versões é mais propensa a resultar em boas soluções para um maior número de instâncias.

Nos testes com as instâncias do Grupo 1, como a grande maioria das soluções ótimas é conhecida, os algoritmos foram executados tomando como critério de parada um custo alvo ou um tempo limite de execução. O alvo fixado foi o custo da melhor solução conhecida de cada instância. Nos testes com as instâncias do Grupo 2, o critério de parada foi apenas um tempo limite de execução. Em todos os testes, o tempo limite foi estabelecido como 300s – tempo suficiente para G6 encontrar o melhor custo conhecido em mais de 95% das instâncias. Os resultados apresentados são referentes à média de dez execuções de cada algoritmo.

A Tabela 4.5 resume os resultados dos testes realizados com as instâncias do Grupo 1. A primeira coluna indica o nome do algoritmo; na segunda tem-se o custo médio de todas as instâncias; na terceira, o gap médio entre o custo das soluções e o alvo; a coluna 4 traz o tempo médio das execuções; a última coluna apresenta a porcentagem de instâncias em que cada algoritmo atingiu o alvo nas dez execuções.

Versão	Custo Médio	Gap Médio (%)	Tempo Médio (s)	Sucesso (%)
G1	14679,29	0,002	6,81	98,2
G2	14679,02	0,000	0,70	100,0
G3	14679,02	0,000	0,35	100,0
G4	14679,33	0,001	5,58	98,8
G5	14679,02	0,000	0,52	100,0
G6	14679,02	0,000	0,33	100,0

Tabela 4.5: Comparação entre as versões do GRASP sobre as instâncias do Grupo 1.

O desempenho de todos os algoritmos foi bem similar, possivelmente porque as instâncias são relativamente fáceis. Pode-se observar que todos os gaps são muito próximos

de zero. Todos os algoritmos conseguiram encontrar os alvos das instâncias cujo ótimo é conhecido.

Pode-se perceber que, como as versões que não possuem reconexão de caminhos (G1 e G4) tiveram mais dificuldade em encontrar os alvos, seu tempo de execução foi bem acima dos demais. Os algoritmos restantes encontraram os alvos de todas as instâncias em todas as execuções e os algoritmos que aplicam busca local iterada são levemente mais rápidos.

Na Tabela 4.6 os algoritmos são comparados por meio dos testes com as instâncias do Grupo 2, de dimensões maiores. Os resultados da tabela indicam valores correspondentes à média das dez execuções. A coluna 1 apresenta o nome de cada versão; na coluna 2 tem-se a média dos custos de todas instâncias; a coluna 3 apresenta o gap médio dos custos em relação ao melhor custo conhecido; por fim, as duas últimas colunas exibem a porcentagem de instâncias em que cada algoritmo encontrou o melhor custo e a porcentagem de instâncias em que cada algoritmo encontrou o melhor custo sozinho, respectivamente.

Versão	Custo Médio	Gap Médio	Instâncias (%)	Instâncias* (%)
G1	30321,07	0,244	30,69	0
G2	30292,92	0,028	65,34	1,98
G3	30288,91	0,014	82,17	7,92
G4	30315,21	0,209	48,51	0
G5	30290,16	0,032	70,29	0
G6	30289,31	0,017	84,15	8,91

Tabela 4.6: Comparação entre as versões do GRASP sobre as instâncias do Grupo 2.

Nas instâncias do Grupo 2, a diferença entre os algoritmos é mais perceptível. Nesse caso, os algoritmos com melhor desempenho foram as versões com aplicação de busca local iterada (G3 e G6) que conseguiram o menor custo para mais de 80% das instâncias. Os piores resultados são dos algoritmos que não possuem reconexão de caminhos. Com a última coluna, percebe-se que G1 e G4 não atingiram um custo médio superior ao dos demais algoritmos para nenhuma instância.

Comparando o uso de um ou vários construtivos, não há diferença considerável. O melhor gap é o da versão G3, que utiliza apenas o construtivo RC4, seguido de G6, que utiliza todos os construtivos. A diferença entre esses dois gaps é praticamente desprezível. Nas versões sem busca local iterada, os algoritmos que utilizam todos os construtivos conseguem melhores soluões para um número maior de instâncias que os algoritmos com apenas RC4. Mas essa diferença diminui entre G6 e G3.

O impacto gerado pelos mecanismo de reconexão de caminhos e busca local iterada

sobre o tempo de execução dos algoritmos pode ser observado analisando-se o tempo gasto com cada iteração GRASP entre as versões propostas. Para tanto, foram realizadas 10 execuções de cada algoritmo com quatro instâncias: 49rd400, 107ali535, 96d657 e 221d657, com características razoalvemente diferentes.

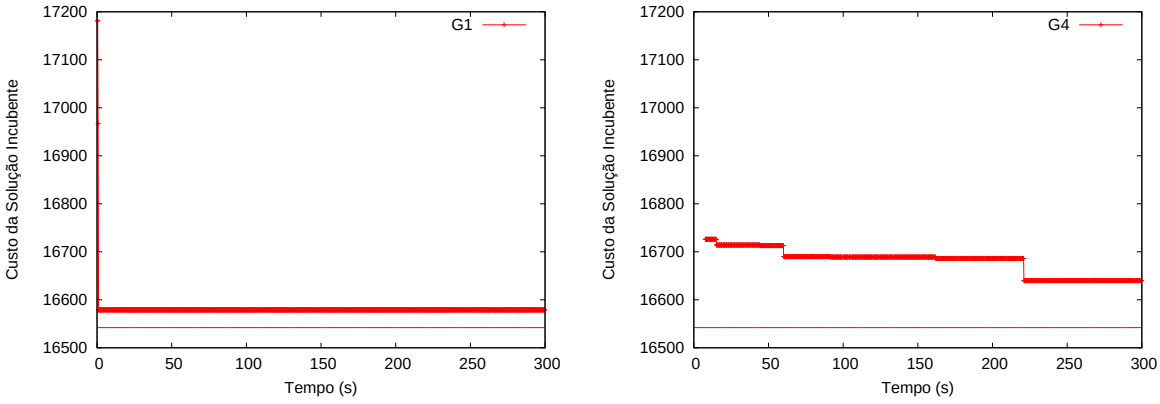
Como não foram identificadas grandes diferenças entre as execuções, e por questão de concisão, a análise será realizada tomando como base apenas uma execução de cada algoritmo com a instância 96d657, cujos resultados estão expressos na Figura 4.3. Cada gráfico ilustra o resultado de uma execução de uma versão GRASP com a instância 96d657, gerada por *Grid Clusterization* com $\mu = 7$. Cada ponto (x, y) dos gráficos indica o custo x da solução incubente no instante de tempo y. Os pontos apresentados correspondem ao fim de cada iteração GRASP. Além desses pontos, foi traçada uma reta indicando o custo ótimo da instância (16542).

Na Figura 4.3(a), referente ao GRASP G1, a distribuição homogênea dos pontos sobre a curva indica que não há grandes diferenças no tempo gasto em cada iteração. Possivelmente isso ocorre porque o algoritmo não aplica mecanismos de aprimoramento adicionais. Resultado semelhante pode ser observado na Figura 4.3(b), com o algoritmo G4. Uma observação é que G1 encontrou uma solução de baixo custo ao início da execução, e não conseguiu mais atualizar a solução incubente. G4, por sua vez, fez várias atualizações no decorrer da execução. Por esse motivo, as curvas dos dois algoritmos têm aspectos diferentes.

Nos algoritmos G2 e G5, a probabilidade p de ocorrência de busca local durante a reconexão de caminhos é proporcional ao número de iterações sem atualização do conjunto elite. Diante disso, o tempo gasto com a reconexão de caminhos aumenta consideravelmente à medida que o algoritmo tende a estagnar. Sempre que ocorre uma atualização do conjunto elite ou o algoritmo atinge o número máximo de iterações sem atualização, p retorna ao seu valor original. Isso pode ser observado pelos gráficos das Figuras 4.3(c) e 4.3(d). Percebe-se que periodicamente há um aumento progressivo do tempo gasto com cada iteração, seguido de uma queda brusca.

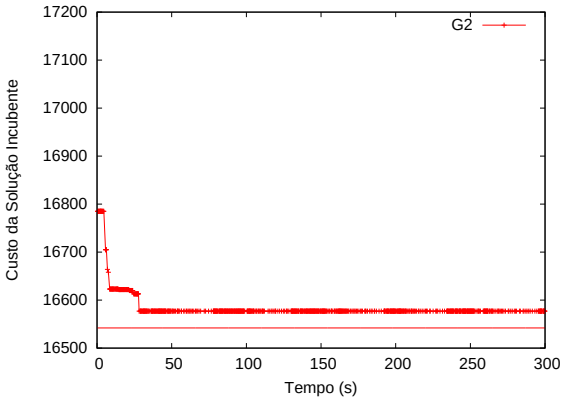
O tempo gasto com a busca local iterada pode ser percebido pela Figura 4.3(e), referente a G3. No gráfico, há dois intervalos entre os pontos: entre 48 e 115s e entre 166 e 280s. Esses intervalos correspondem ao tempo que o algoritmo interrompeu o processo tradicional construção-aprimoramento para aplicar a busca local iterada. O mesmo pode ser percebido pelo gráfico da Figura 4.3(f).

Comparando as versões que utilizam apenas o construtivo RC4 (G1, G2 e G3) com as

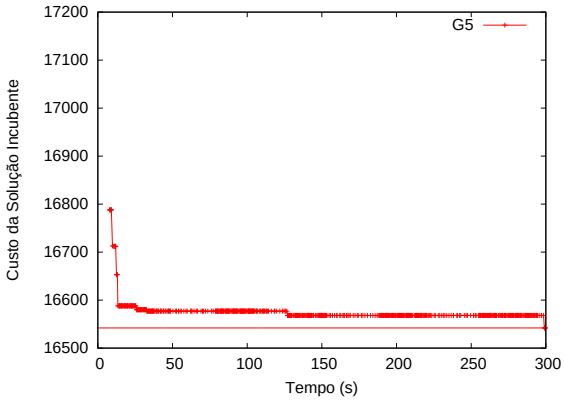


(a) G1

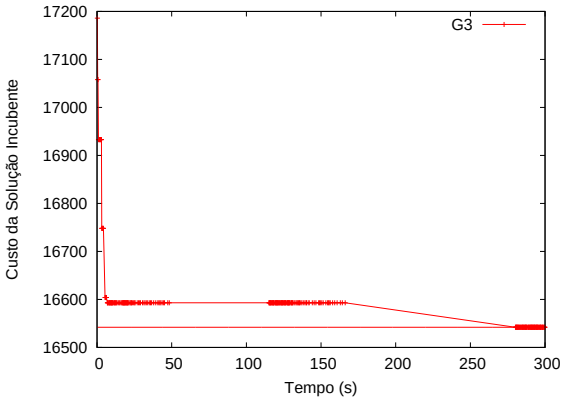
(b) G4



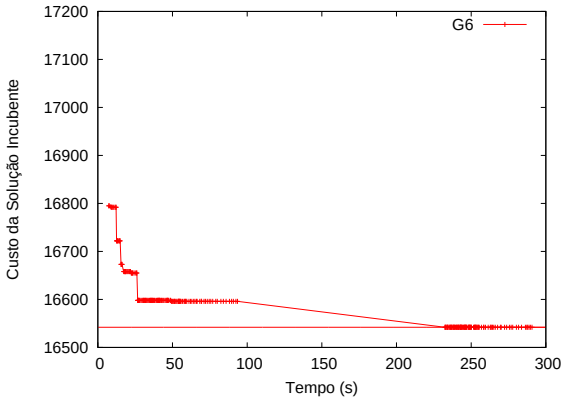
(c) G2



(d) G5



(e) G3



(f) G6

Figura 4.3: Análise de uma execução de cada versão GRASP com a instância 96d657

correspondentes que utilizam todos os construtivos (G4, G5 e G6), pode-se perceber que a primeira iteração dos algoritmos adaptativos inicia mais tarde. Isso pode ser explicado pelo tempo gasto com as iterações iniciais, em que se aplica uma busca local sobre soluções obtidas pelos construtivos gulosos.

4.3.4 Comparação com Resultados da Literatura

Como as instâncias do Grupo 1 são razoavelmente fáceis, torna-se difícil identificar diferenças nos desempenhos médios dos algoritmos. Por esse motivo, a comparação com trabalhos da literatura será feita com base nas instâncias do Grupo 2. Os testes foram realizados apenas com o GRASP G6, que apresentou bons resultados para um número maior de instâncias em comparação com as demais versões.

No trabalho de Oncan et al. [24], é feita uma comparação do seu algoritmo de busa tabu com os melhores algoritmos conhecidos até então. Tal comparação é baseada nas instâncias que compoem o Grupo 2. Os resultados estão resumidos na Tabela 4.7. As duas primeiras linhas são referentes à busca local e ao algoritmo genético proposto por Golden et al. [15]. A terceira linha apresenta os resultados para uma versão melhorada do algoritmo genético de Golden - ainda não há referência na literatura para esse algoritmo. A última linha traz os resultados da busca tabu de Oncan. As primeiras quatro colunas indicam o nome do algoritmos, custo, gap e tempo médio para todas as instâncias, respectivamente; as duas últimas apresentam, respectivamente, a porcentagem de instâncias em que cada algoritmo encontrou a melhor solução e em que cada algoritmo encontrou a melhor solução sozinho.

Algoritmo	Custo	Gap (%)	Tempo (s)	Instâncias (%)	Instâncias* (%)
LS	30328,11	0,33	548,14	33,33	0
GA1	30305,55	0,17	754,41	44,11	0
GA2	30292,88	0,04	578,12	67,64	0
Tabu	30289,49	0,01	736,85	100	24,5

Tabela 4.7: Comparação entre os principais algoritmos da literatura.

Apesar do tempo de execução da busca tabu não ser o menor entre todos, suas soluções mantêm o menor gap em relação às melhores soluções conhecidas. Outro fato marcante é que o algoritmo conseguiu os melhores resultados para todas as instâncias. Por esse motivo, os resultados do algoritmo G6 (aqui proposto) serão comparados com o trabalho de Oncan et al. [24].

A comparação com G6 será então feita com dois objetivos: analisar as soluções encontradas por G6 no mesmo tempo de execução do Tabu e verificar o tempo necessário para o G6 encontrar as soluções daquele algoritmo. Os resultados reportados sobre o trabalho de Oncan et al. são referentes a testes realizados pelos autores, com algoritmos implementados em C++ e executados em uma máquina Pentium IV 3GHz. A arquitetura difere da utilizada nos testes com G6, mas acredita-se que a diferença é praticamente desprezível.

Considerando o primeiro objetivo, G6 foi executado dez vezes com o tempo limite de execução igual ao tempo gasto pelo Tabu para cada instância. Os resultados estão apresentados nas tabelas 4.8, 4.9 e 4.10.

Em cada tabela, a primeira coluna apresenta o nome da instância; na segunda tem-se o tempo gasto pelo algoritmo de busca tabu – mesmo tempo utilizado para o GRASP; a terceira coluna apresenta o limite dual encontrado pelo algoritmo de geração de cortes (para as instâncias em que foi possível encontrá-lo); nas colunas quatro e cinco estão o custo e o gap das soluções do algoritmo tabu (não é reportado no trabalho de Oncan et. al [24] se tais resultados são referentes à média de várias execuções); as duas últimas colunas apresentam o custo e o gap médio das soluções encontradas pelo GRASP nas dez execuções. Em todas as tabelas, o gap foi calculado em relação ao limite dual apresentado.

Instância	Tempo (s)	Limite Dual	Tabu		G6	
			Custo	Gap (%)	Custo	Gap (%)
107ali535	683	114289	114303	0,01	114303	0,01
107att532	597	12001	12001	**0,00**	12001	**0,00**
99d493	587	16493	16493	**0,00**	16493	**0,00**
132d657	1056	19427	19427	**0,00**	19427	**0,00**
84fl417	233	7935	7935	**0,00**	7935	**0,00**
53gil262	74	887	887	**0,00**	887	**0,00**
87gr431	233	-	86885	-	86885	-
134gr666	1365	-	144756	-	*144737	-
64lin318	130	18471	18471	**0,00**	18471	**0,00**
131p654	1045	-	22208	-	*22207	-
113pa561	702	861	864	0,35	864	0,35
89pcb442	266	19571	19571	**0,00**	19571	**0,00**
53pr264	72	21872	21872	**0,00**	21872	**0,00**
60pr299	94	20290	20290	**0,00**	20290	**0,00**
88pr439	574	51749	51760	0,02	*51749	**0,00**
115rat575	762	2168,5	*2170	0,07	2170,4	0,09
157rat783	1916	3009	3017	0,27	*3012,4	0,11
80rd400	208	5868	5868	**0,00**	5868	**0,00**
107si535	573	12791	12791	**0,00**	12791	**0,00**
115u574	517	15027	15037	0,07	*15032,4	0,04
145u724	1290	15904	15905	0,01	*15904	**0,00**

Tabela 4.8: Comparação entre G6 e Tabu em instâncias geradas por *Cluster Centering*.

Instância	Tempo (s)	Limite Dual	Tabu		G6	
			Custo	Gap (%)	Custo	Gap (%)
Grid Clusterization com $\mu = 3$						
181ali535	1576	-	134362	-	134362	-
182att532	1569	15652	15652	**0,00**	15652	**0,00**
171d493	1104	20269	20269	**0,00**	20269	**0,00**
221d657	3027	25703	*25703	**0,00**	25704,2	0,00
142fl417	681	8353	8353	**0,00**	8353	**0,00**
95gil262	204	1255	1255	**0,00**	1255	**0,00**
81gr229	67	74792	74792	**0,00**	74792	**0,00**
149gr431	1697	-	103844	-	103844	-
224gr666	3105	-	174671	-	*174655	-
108lin318	206	24092	24092	**0,00**	24092	**0,00**
230p654	4127	23547	23553	0,03	*23547	**0,00**
156pcb442	810	27513	27513	**0,00**	27513	**0,00**
101pr264	102	29199	29199	**0,00**	29199	**0,00**
102pr299	399	23096	23096	**0,00**	23096	**0,00**
163pr439	932	64953	64953	**0,00**	64953	**0,00**
196rat575	2080	2880	2884	0,14	*2880	**0,00**
285rat783	7180	4203	4211	0,19	*4203	**0,00**
135rd400	548	7632	7632	**0,00**	7632	**0,00**
198u574	1340	19696	19699	0,02	*19696	**0,00**
266u724	5201	21739	21761	0,1	*21747	0,04
Grid Clusterization com $\mu = 5$						
108ali535	632	-	108201	-	108201	-
110att532	641	11896	11896	**0,00**	11896	**0,00**
102d493	725	16132	16132	**0,00**	16132	**0,00**
137d657	1249	-	19846	-	*19826	-
93fl417	230	7952	7952	**0,00**	7952	**0,00**
63gil262	63	984	*984	**0,00**	984,4	0,04
47gr229	43	54305	54305	**0,00**	54305	**0,00**
87gr431	244	81856	81856	**0,00**	81856	**0,00**
139gr666	2192	-	144077	-	144077	-
64lin318	116	17667	17667	**0,00**	17667	**0,00**
134p654	1316	22377	22379	0,01	*22377	**0,00**
95pcb442	357	19383	19411	0,14	*19383	**0,00**
55pr264	67	21351	21351	**0,00**	21351	**0,00**
69pr299	105	18582	18582	**0,00**	18582	**0,00**
96pr439	587	54230	54230	**0,00**	54230	**0,00**
121rat575	705	2014	2018	0,2	*2014	**0,00**
169rat783	2424	2823	2832	0,32	*2823	**0,00**
81rd400	284	5433,5	*5434	0,01	5434,8	0,02
127u574	1037	15240	15255	0,1	*15252	0,08
166u724	2064	15435	15458	0,15	*15435	**0,00**

Tabela 4.9: Comparação entre G6 e Tabu em instâncias geradas por *Grid Clusterization* com $\mu = 3$ e $\mu = 5$.

Instância	Tempo (s)	Limite Dual	Tabu		G6	
			Custo	Gap (%)	Custo	Gap (%)
Grid Clusterization com $\mu = 7$						
83ali535	1012	-	94149	-	94149	-
80att532	992	10204	10206	0,02	*10204	**0,00**
78d493	212	14152	14152	**0,00**	14152	**0,00**
96d657	440	-	*16542	-	16542,6	-
61fl417	99	7446	7446	**0,00**	7446	**0,00**
49gil262	63	802	*802	**0,00**	802,6	0,07
34gr229	13	45989	46049	0,13	46049	0,13
64gr431	111	-	71415	-	71415	-
96gr666	452	-	119271	-	119271	-
49lin318	44	14909	14909	**0,00**	14909	**0,00**
100p654	462	-	21770	-	21770	-
64pcb442	117	14639,5	*14644	0,03	14645,2	0,04
43pr264	28	20438	20438	**0,00**	20438	**0,00**
47pr299	37	15238	15238	**0,00**	15238	**0,00**
74pr439	140	47101	47101	**0,00**	47101	**0,00**
100rat575	601	1734,5	*1735	0,03	1735,6	0,06
121rat783	850	2228	2230	0,09	*2229	0,04
64rd400	97	4576	4581	0,11	4581	0,11
92u574	304	12186	12186	**0,00**	12186	**0,00**
119u724	710	-	*13101	-	13109	-
Grid Clusterization com $\mu = 10$						
57ali535	213	-	73359	-	73359	-
57att532	187	8494,84	8497	0,03	8497	0,03
52d493	180	11121	11121	**0,00**	11121	**0,00**
73d657	386	-	13495	-	13495	-
42fl417	75	6986	6986	**0,00**	6986	**0,00**
36gil262	33	639	639	**0,00**	639	**0,00**
23gr229	28	39793	39793	**0,00**	39793	**0,00**
52gr431	110	62722	62722	**0,00**	62722	**0,00**
70gr666	315	-	97198	-	97198	-
36lin318	31	10119	10119	**0,00**	10119	**0,00**
69p654	274	20739	20739	**0,00**	20739	**0,00**
48pcb442	135	11941	11941	**0,00**	11941	**0,00**
27pr264	28	16546	16546	**0,00**	16546	**0,00**
35pr299	63	11624	11624	**0,00**	11624	**0,00**
48pr439	117	40518	40518	**0,00**	40518	**0,00**
64rat575	284	1235	1235	**0,00**	1235	**0,00**
81rat783	468	-	1682	-	1682	-
49rd400	99	3825	3825	**0,00**	3825	**0,00**
64u574	234	9755	9755	**0,00**	9755	**0,00**
80u724	365	-	9608	-	9608	-

Tabela 4.10: Comparação entre G6 e Tabu em instâncias geradas por *Grid Clusterization* com $\mu = 7$ e $\mu = 10$.

Comparando a soluções dos dois algoritmos, percebe-se que em 72 instâncias os dois algoritmos empataram, em 21 o custo médio de G6 é melhor e em 8 instâncias G6 não conseguiu em média atingir o custo do algoritmo de busca tabu.

Considerando apenas as instâncias de que se conhece o limite inferior, as soluções do GRASP são em média mais próximas do limite dual que as soluções do algoritmo tabu: tem-se um gap médio de 0,016% para o GRASP enquanto o gap médio do tabu é de 0,031%.

Mesmo nas instâncias em que o GRASP não atingiu o custo das soluções do Tabu em média, esse custo sempre foi atingido em alguma das dez execuções. O gap médio das melhores soluções que o GRASP encontrou é de 0,013% em relação ao limite dual.

Outra observação é que o GRASP teve mais dificuldade em encontrar os alvos para instâncias com mais vértices por grupo. Nas instâncias geradas por *Grid Clusterization* com $\mu = 3$, o GRASP encontrou muitas soluções melhores que as do algoritmo de Oncan. Já nas instâncias geradas com $\mu = 7$, em muitos casos o tempo de execução do algoritmo de busca tabu não foi suficiente para que o GRASP encontrasse a melhor solução conhecida em todas as execuções.

Como foi dito anteriormente, a segunda bateria de testes apresentada nesta subseção teve como objetivo principal verificar a eficiência do GRASP em encontrar soluções iguais ou melhores que as soluções do algoritmo tomado como comparação. O critério de parada foi atingir o custo da solução encontrada pelo algoritmo tabu ou um tempo limite de 3 horas de execução. O tempo limite foi bem mais alto que em testes anteriores para que G6 sempre encontrasse as soluções alvo. Para cada instância, foram realizadas dez execuções de G6.

Visando a análise do impacto da redução das instâncias sobre o desempenho do algoritmo, também foram realizadas dez execuções de G6 sem a fase de pré-processamento.

O gráfico da Figura 4.4 ilustra o número de instâncias em que cada algoritmo atingiu o alvo por unidade de tempo. O impacto do pré-processamento sobre a eficiência do algoritmo fica evidente, pois percebe-se que o número de alvos encontrados pelo GRASP com redução de instâncias é sempre maior, em qualquer instante de tempo considerado. Em 10s de execução, por exemplo, o GRASP com redução de instâncias consegue encontrar os alvos de 54 instâncias, enquanto o GRASP sem redução encontra os alvos de apenas 37. O tempo máximo do GRASP com redução foi de 896s contra 4654s da versão sem a redução.

Um ponto importante a ser comentado é que ambas as versões atingem os alvos em um tempo máximo razoavelmente pequeno em relação aos tempos apresentados no trabalho de Oncan. Não se pretende com isso estabelecer uma comparação direta entre os algoritmos, pois não é conhecido o tempo necessário para o tabu encontrar aquelas soluções. Mas é possível perceber que o GRASP implementado não necessita de um tempo muito maior que o tempo de execução do algoritmo de busca tabu para encontrar soluções iguais ou melhores.

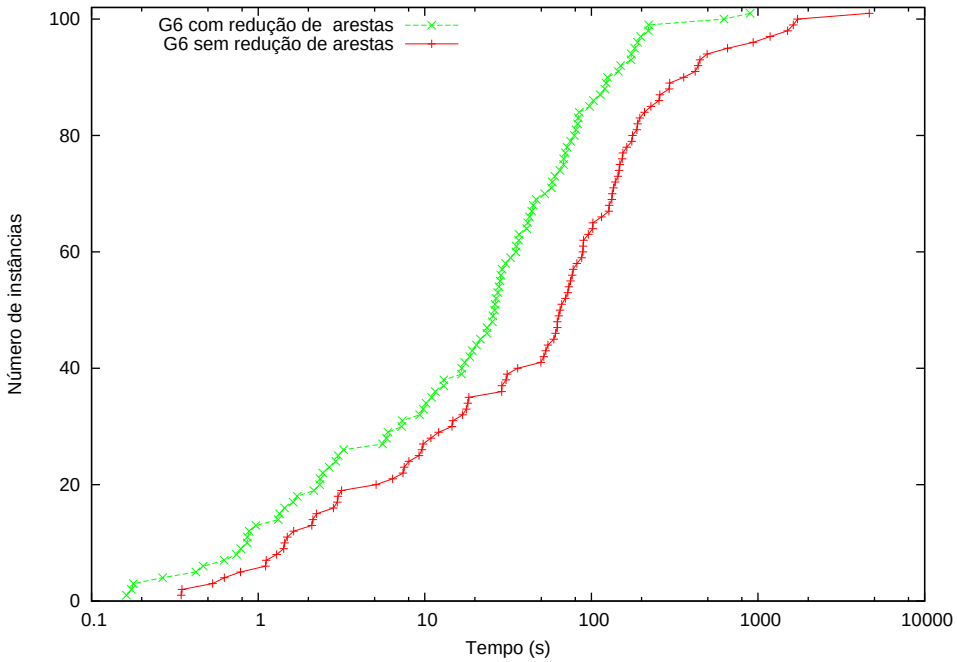


Figura 4.4: Número de alvos encontrados por G6 com e sem redução de arestas.

A Tabela 4.11 apresenta um resumo dos resultados do GRASP tendo com a solução do algoritmo tabu como alvo. A primeira coluna indica o algoritmo a que os dados se referem. A segunda apresenta o custo médio encontrado pelos algoritmos; a coluna 3 apresenta o gap médio de cada algoritmo em relação ao custo das melhores soluções conhecidas; a quarta e a quinta coluna trazem o melhor custo e o melhor gap dentre as dez execuções, respectivamente; a última coluna apresenta o tempo médio de execução de cada algoritmo.

Algoritmo	Custo	Gap (%)	Custo*	Gap* (%)	Tempo Médio (s)
Tabu	30289,49	0,0112	30289,49	0,0112	736,85
G6 - Sem Redução	30288,82	0,0076	30288,17	0,0025	207,25
G6 - Com Redução	30288,65	0,0062	30287,93	0,0007	59,13

Tabela 4.11: G6 com alvo igual à solução do algoritmo de busca tabu.

Com a comparação direta dos dois algoritmos, foi verificado que o G6, mesmo sem

redução de instâncias, consegue encontrar as mesmas soluções em um tempo três vezes menor que o tempo de execução do Tabu. A redução causa um impacto considerável, pois o algoritmo se torna, em média, quase quatro vezes mais rápido.

4.3.5 Análise Probabilística

A bateria de testes apresentada nesta subseção teve como objetivo analisar a função de distribuição de probabilidade dos algoritmos propostos atingirem uma determinada solução em relação ao tempo. Os testes foram realizados conforme o modelo proposto por Aiex et al. [2].

Foram selecionadas quatro instâncias para a realização dos experimentos. Procurou-se utilizar instâncias com características diferentes em relação ao número de vértices, técnica de agrupamento e grau de dificuldade. As seguintes instâncias foram utilizadas:

- **49rd400** - com 400 vértices, gerada por *Grid Clusterization* com $\mu = 10$, cuja solução ótima é atingida em menos de 10s por G6;
- **107ali535** - com 535 vértices, gerada por *Cluster Centering*, em que G6 encontrou a melhor solução conhecida em um tempo médio de 86s;
- **96d657** - com 657 vértices, gerada por *Grid Clusterization* com $\mu = 7$, cujo ótimo é encontrado em mais de 100s de execução, em média.
- **221d657** - com 657 vértices, gerada por *Grid Clusterization* com $\mu = 3$, em que os algoritmos encontram a solução ótima em um tempo médio maior que 200s.

Em cada experimento, cada algoritmo foi executado com critério de parada atingir um determinado custo alvo ou um tempo limite de execução. Foram computados os tempos de 100 execuções independentes para cada versão GRASP. O tempo limite estipulado foi 3600s, considerado suficiente para a análise do comportamento dos algoritmos.

Para cada instância, foram realizados testes com um alvo "fácil" e um alvo "difícil". Foi considerado como alvo fácil o custo da solução encontrada pelo algoritmo de busca local de Golden et al. [15] (reportado por Oncan et al. [24]) e como alvo difícil o custo da melhor solução conhecida de cada instância. A busca local de Golden et al. foi utilizada como critério para escolha do alvo fácil por ser o algoritmo da literatura que resulta em um pior custo médio para esse grupo de instâncias. Os algoritmos G1 e G4 não foram utilizados

nos experimentos com alvos difíceis, em vista de seu desempenho ser consideravelmente inferior ao dos demais algoritmos.

Em cada curva dos gráficos seguintes, dentre as 100 execuções, o i-ésimo menor tempo de execução t_i foi associado à probabilidade $p_i = (i - 0,5)/100$, gerando pontos (t_i, p_i) para $i = 1, ..., 100$. A análise dos gráficos pode ser feita considerando o alinhamento das curvas: a curva mais à esquerda indica o algoritmo que converge mais rápido para o valor alvo.

No gráfico da Figura 4.5, estão os resultados para testes com a instância 49rd400 utilizando um alvo fácil (3844). Percebe-se que até 1,2s, o comportamento de todos os algoritmos é bastante semelhante. Entretanto, os algoritmos que aplicam reconexão de caminhos (G2, G3, G5 e G6) atigem os alvos em 100% das execuções em um tempo máximo de 20s. Nesse mesmo tempo, G1 e G4 atingiram o alvo em apenas 40% das execuções. Outra observação é que não há diferenças visíveis entre G2, G3, G5 e G6.

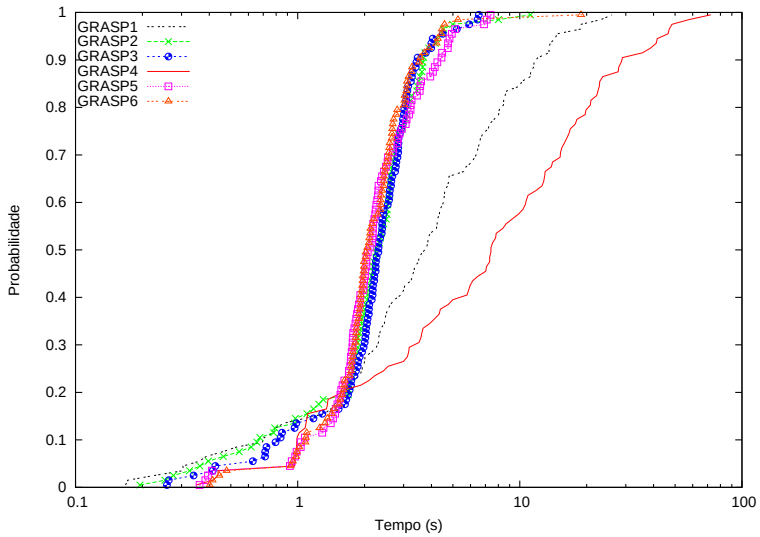


Figura 4.5: Função de distribuição de probabilidade dos algoritmos atingirem uma solução alvo fácil com a instância 49rd400.

Nos experimentos realizados com alvo difícil, há uma dinstinção bem mais clara entre as curvas geradas pelas versões GRASP que utilizam apenas um construtivo e pelas adaptativas. Com isso, é possível observar a influência do algoritmo construtivo utilizado sobre o desempenho dos algoritmos.

Na Figura 4.6, o alvo é o custo 3825. Pode-se perceber que as versões adaptativas, G5 e G6, apresentam melhor desempenho. Uma possível explicação para isso é o fato de tais algoritmos utilizarem algoritmos de construção que melhor se enquadram às características

da instância. Dentre G2 e G3, que utilizam apenas o construtivo RC4, há uma leve vantagem para o último algoritmo, possivelmente pela utilização do mecanismo de busca local iterada.

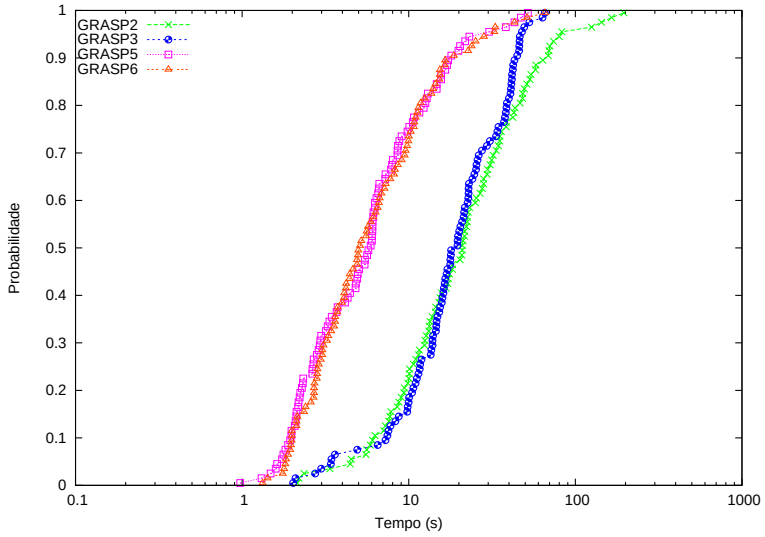


Figura 4.6: Função de distribuição de probabilidade dos algoritmos atingirem uma solução alvo difícil com a instância 49rd400.

Os próximos dois gráficos são referentes a experimentos realizados com a instância 107ali535. O gráfico da Figura 4.7 apresenta o comportamento dos algoritmos para um alvo fácil, 114362. É possível perceber que novamente as versões com reconexão de caminhos apresentam melhor desempenho. Neste caso, a diferença destas em relação às demais é mais evidente: enquanto tais algoritmos executam em pouco mais de 100s, G4 chega a executar em um tempo próximo de 1700s e G1 não atinge o valor alvo em 100% das execuções.

O formato das curvas permite verificar que as versões com reconexão de caminhos também possuem comportamento mais uniforme que as demais. Em todas as execuções, os algoritmos executam em tempos entre 1 e 111s, enquanto G1, por exemplo, chega a encontrar o alvo em 3s e em algumas execuções não é capaz de encontrá-lo, considerando o tempo limite de 3600s.

Considerando os testes realizados com alvo difícil, os algoritmos que aplicam busca local iterada (G3 e G6) apresentaram um desempenho melhor. Comparando G2 e G5, que diferem entre si pela utilização de um ou mais construtivos, a última versão tem um desempenho superior, convergindo mais rápido para o alvo. Entretanto, a busca local iterada tornou o resultados de G3 e G6 extremamente parecidos, não é possível fazer uma distinção entre as curvas dos dois algoritmos.

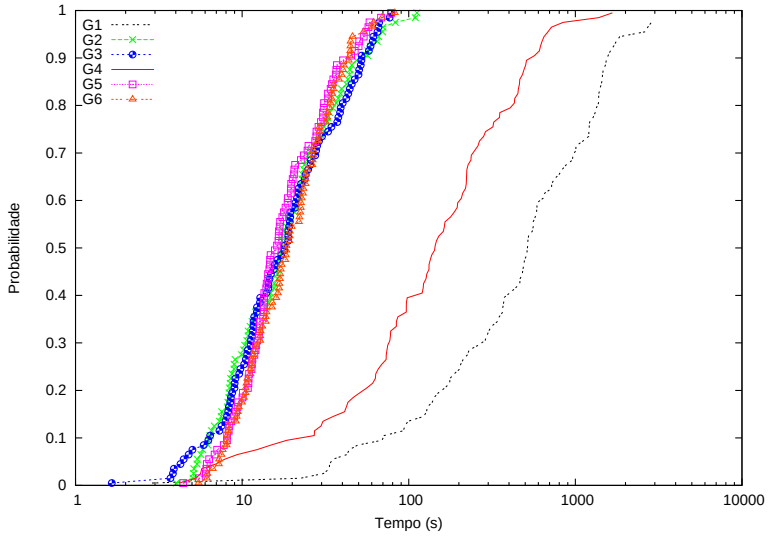


Figura 4.7: Função de distribuição de probabilidade dos algoritmos atingirem uma solução alvo fácil com a instância 107ali535.

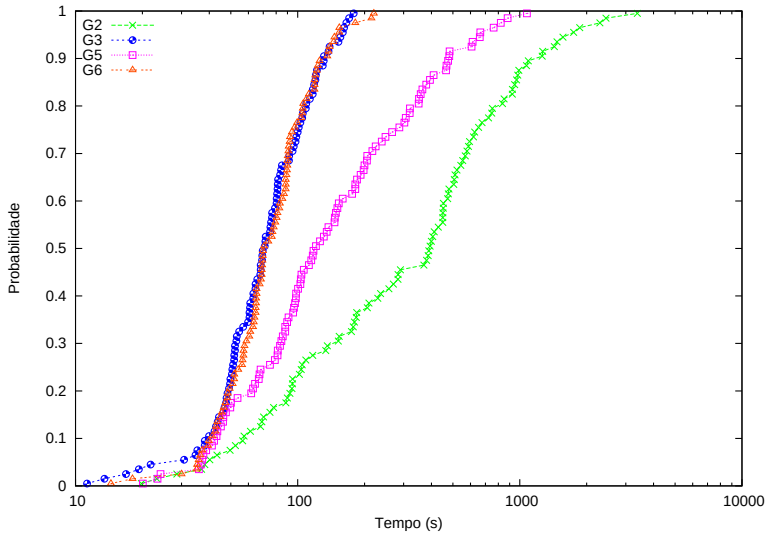


Figura 4.8: Função de distribuição de probabilidade dos algoritmos atingirem uma solução alvo difícil com a instância 107ali535.

Nos testes com a instância 96d657, o alvo fácil foi o custo 16663. O gráfico da Figura 4.9 apresenta os resultados. Mais uma vez as duas versões que não aplicam reconexão de caminhos, G1 e G4, apresentam um desempenho inferior. Entretanto, é possível perceber que os algoritmos G2 e G3 convergem para o alvo em um tempo menor que G5 e G6, diferentemente do que acontece com as instâncias apresentadas anteriormente. Mesmo a aplicação da busca local iterada não é suficiente para que o GRASP adaptativo G6 encontre o custo alvo tão rapidamente quanto G2, que não aplica tão mecanismo.

O gráfico da Figura 4.10 apresenta os resultados dos testes com alvo difícil, cujo valor

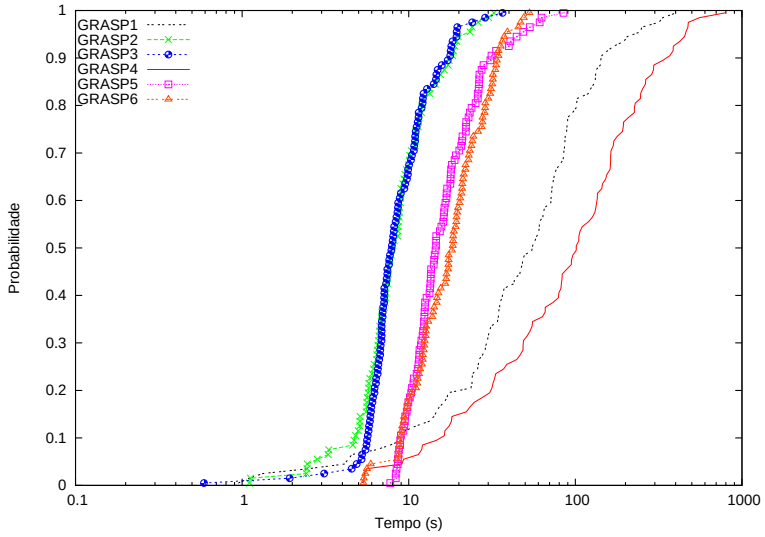


Figura 4.9: Função de distribuição de probabilidade dos algoritmos atingirem uma solução alvo fácil com a instância 96d657.

é 16542. Nenhum dos algoritmos conseguiu atingir o alvo em 100% dos casos. O algoritmo G6 apresenta o melhor resultado, converindo para o alvo mais rápido que os demais em quase todas as execuções. Apenas em uma execução G6 não chegou a encontrar o custo alvo. Apesar de não haver grandes discrepâncias entre os tempos dos algoritmos, G2 apresenta o pior resultado, convergindo para o alvo em apenas 85% dos casos.

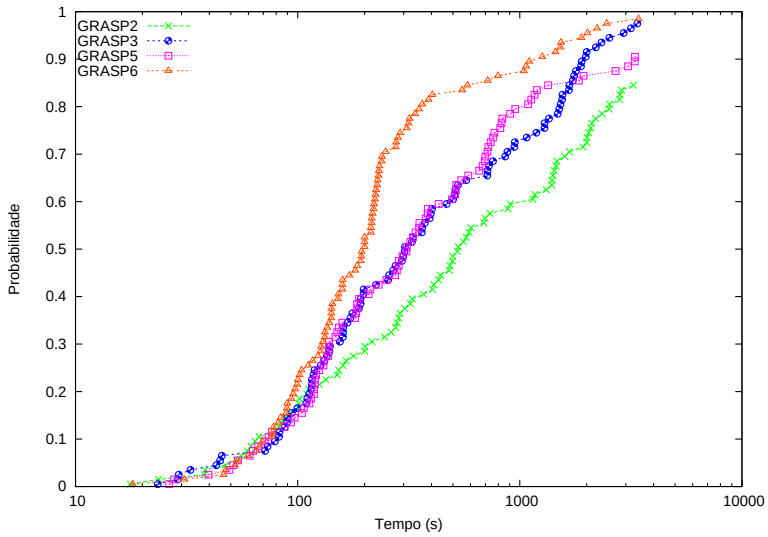


Figura 4.10: Função de distribuição de probabilidade dos algoritmos atingirem uma solução alvo difícil com a instância 96d657.

Dentre as instâncias consideradas nesta bateria de testes, a instância 221d657 é aquela em que os algoritmos necessitam de mais tempo para encontrar a solução ótima. O gráfico

da Figura 4.11 ilustra os resultados obtidos para os experimentos com alvo fácil, cujo valor é 25781.

Nesse caso, os algoritmos G2 e G3, que utilizam apenas o algoritmo RC4 como construtivo conseguiram convergir para o alvo em menos de 20s em 100% dos casos. Nesse mesmo tempo, as versões adaptativas não atingiram o alvo em nenhuma das execuções. Os algoritmos sem reconexão de caminhos novamente apresentam um desempenho inferior, mas o GRASP adaptativo precisa de mais de 1500s para garantir 100% de sucesso.

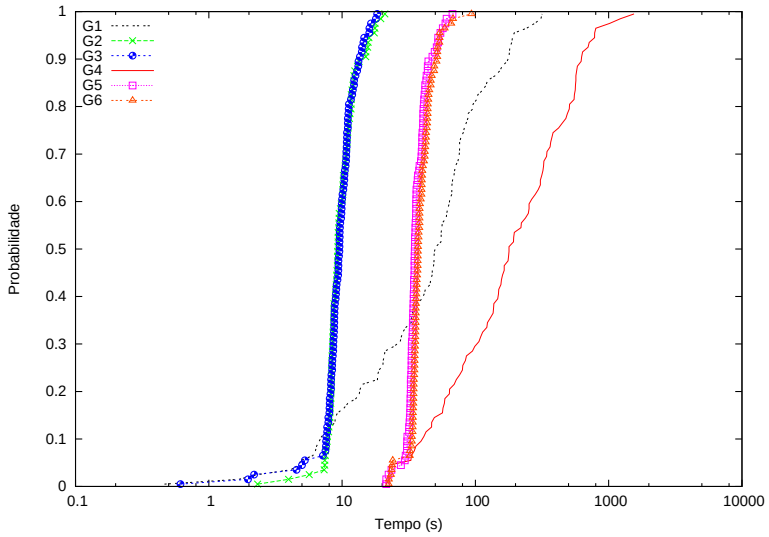


Figura 4.11: Função de distribuição de probabilidade dos algoritmos atingirem uma solução alvo fácil com a instância 221d657.

Os resultados para o alvo difícil estão expressos no gráfico da Figura 4.12. O alvo considerado é o custo da solução ótima, 25703. Nenhum dos algoritmos conseguiu encontrá-lo em 100% das execuções. Comparando as versões G2 e G5, que diferem pela utilização de um ou mais construtivos, a última versão apresenta um desempenho ligeiramente superior.

Entre as versões que aplicam busca local iterada, G3 e G6, a primeira consegue encontrar o alvo, em média, em um tempo menor que a versão adaptativa. Em torno de 150s, por exemplo, G3 encontrou o alvo em quase 60% das execuções, enquanto G6 em apenas 20%. Entretanto, a versão adaptativa chega ao alvo em 94%, enquanto a primeira em apenas 89% das execuções.

Com os resultados dos testes com a instância 221d657, percebe-se que a utilização do melhor algoritmo construtivo é mais eficiente para se atingir uma solução relativamente fácil. Percebe-se também que para essa instância o algoritmo G3 tende a encontrar a solução ótima mais rápido que os demais, entretanto, o algoritmo G6 atinger a solução

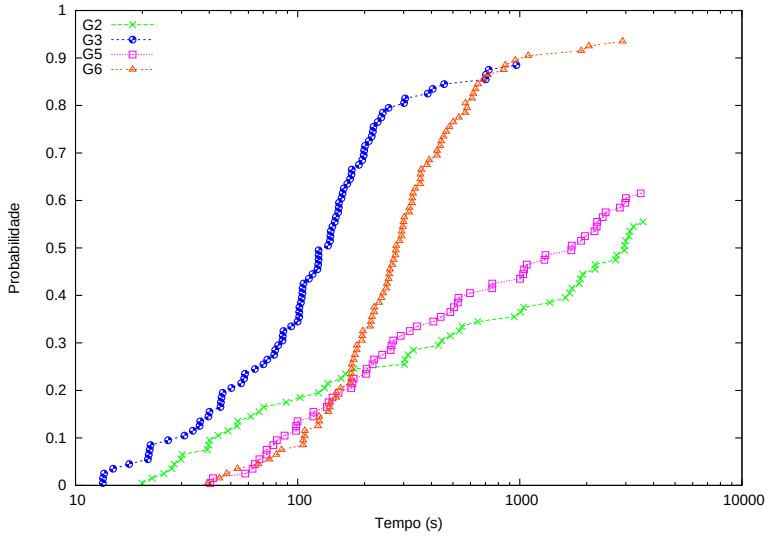


Figura 4.12: Função de distribuição de probabilidade dos algoritmos atingirem uma solução alvo difícil com a instância 221d657.

ótima com mais freqüência, o que o torna mais robusto que os demais na busca de soluções difíceis.

Este capítulo apresentou os resultados computacionais de testes realizados com os algoritmos propostos neste trabalho. Primeiramente descreveu-se o procedimento de pré-processamento para a redução das instâncias. Em seguida, foram apresentados os resultados do algoritmo de geração de cortes. Também foram apresentados e discutidos resultados de testes comparativos com as seis versões GRASP descritas no capítulo anterior, além dos resultados de testes realizados para comparação com resultados da literatura.

Capítulo 5

Conclusões e Trabalhos Futuros

Este trabalho abordou uma generalização do Problema da Árvore Geradora Mínima, denominado Problema da Árvore Geradora Mínima Generalizado.

Foram implementados algoritmos construtivos da literatura e alguns propostos neste trabalho. Também foi proposto um mecanismo de reconexão de caminhos e de busca local iterada. O desempenho de cada algoritmo construtivo foi verificado em instâncias apresentadas na literatura. A partir desses experimentos, seis versões GRASP foram propostas e comparadas por meio de testes experimentais.

Os resultados computacionais indicaram que a utilização de mecanismos adicionais de aprimoramento propicia uma melhora significativa do desempenho da heurística GRASP, independentemente do algoritmo de construção utilizado. As versões que não contêm nenhum desses mecanismos obtiveram resultados piores que as demais em praticamente todos os experimentos realizados. Além disso, as versões GRASP que aplicam busca local iterada encontram os melhores custos para mais de 80% das instâncias testadas.

A utilização de mais de um algoritmo de construção tende a fazer com que os algoritmos encontrem boas soluções para um número maior de instâncias. Apesar do tempo de execução ser ligeiramente maior, as versões adaptativas apresentam um comportamento mais uniforme em relação às demais.

Em comparação com algoritmos da literatura, o GRASP adaptativo com reconexão de caminhos e busca local iterada é capaz de encontrar custos melhores para 22 das 101 instâncias, enquanto em apenas 8 instâncias seu custo médio não atinge o melhor resultado da literatura, obtido por um algoritmo de busca tabu. Em relação às instâncias de que se conhece o limite dual, o gap médio das soluções do GRASP é de 0,031% contra 0,016% do algoritmo tabu. Foi constatado também que o tempo de execução do GRASP proposto é

compatível com os tempos apresentados em trabalhos relacionados.

Também foi proposto um algoritmo de geração de cortes para o cálculo de limites duais baseado em uma formulação para o Problema de Steiner em Grafos Direcionado. O limite encontrado pelo algoritmo corresponde ao valor ótimo de todas as instâncias cujo ótimo é conhecido. Em relação às 101 instâncias de que não se conhece um limite dual, o algoritmo encontrou o limite para 82 delas.

O conceito de Distância *Bottleneck* utilizado no pré-processamento de instâncias do Problema de Steiner em Grafos foi adaptado para o PAGMG. Com isso, foram produzidas regras que propiciaram a redução das instâncias para 14% do seu tamanho original, em média.

Como trabalhos futuros, propõe-se um estudo mais aprofundado de técnicas de memória adaptativa, para a implementação de algoritmos que possam ser configurados no decorrer da execução, propiciando uma calibração mais adequada às características de cada instância. Tal estudo envolve, por exemplo, a calibração do parâmetro α e de parâmetros específicos dos algoritmos apresentados.

Foram implementadas adaptações do algoritmos de Kruskal e Prim como algoritmos construtivos. Assim, como trabalhos futuros, propõe-se a implementação da adaptação de Sollin e comparação de seu desempenho com as demais heurísticas. Além disso, propõe-se também a estipulação de uma distância mínima entre os membros do conjunto elite e implementação de mecanimos mais sofisticados para a busca local, como por exemplo, a idéia de vizinhança variada (VNS).

Em relação ao algoritmo de geração de cortes, é interessante implementar um procedimento de remoção de cortes que não são relevantes para o problema linear corrente. Também é interessante a implementação de um algoritmo *branch-and-cut* com base nessa geração de cortes.

Referências

[1] Ahuja, R. K., Magnanti, T. L. *Network Flows: Theory, Algorithms and Applications.* Prentice-Hall, Englewood Cliffs, New Jersey, 1993.

[2] Aiex, R., Resende, M., Ribeiro, C. Probability distribution of solution time in GRASP: An experimental investigation. *Journal of Heuristics 8* (2002), 343–373.

[3] den Besten, M. L., Stützle, T., Dorigo, M. Design of iterated local search algorithms: An example application to the single machine total weighted tardiness problem. In *Applications of Evolutionary Computing: Proceedings of EvoWorkshops 2001* (Berlin, Germany, 2001), E. J. W. Boers, J. Gottlieb, P. L. Lanzi, R. E. Smith, S. Cagnoni, E. Hart, G. R. Raidl, and H. Tijink, Eds., vol. 2037 of *Lecture Notes in Computer Science*, Springer Verlag, p. 441–452.

[4] Dror, M., Haouari, M., Chaouachi, J. S. Generalized spanning trees. *European Journal of Operational Research 120* (2000), 583–592.

[5] Duin, C. W., Volgenanti, A., Voss, S. Solving group Steiner problems as Steiner problems. *European Journal of Operational Research 154* (2004), 323–329.

[6] Feo, T., Resende, M., Smith, S. A greedy randomized adaptive search procedure for maximum independent set. *Operations Research 42* (1994), 860–878.

[7] Feremans, C. *Generalized Spanning Trees and Extensions.* PhD thesis, Université Libre de Bruxelles, 2001.

[8] Feremans, C., Labbé, M., Laporte, G. The generalized minimum spanning tree problem: Polyhedral analysis and branch-and-cut algorithm. Tech report, Université Libre de Bruxelles, Bruxelas, Bélgica, 2000. Technical Report.

[9] Feremans, C., Labbé, M., Laporte, G. On generalized minimum spanning trees. *European Journal of Operational Research 134* (2001), 457–458.

[10] Feremans, C., Labbé, M., Laporte, G. A comparative analysis of several formulations for the generalized minimum spanning tree problem. *Networks 39* (2002), 29–34.

[11] Feremans, C., Lodi, A., Toth, P., Tramotani, A. Improving on branch-and-cut algorithms for generalized minimum spanning trees. *Pacific Journal of Optimization 1* (2005), 491–508.

[12] Fischetti, M., Salazar, J. J., Toth, P. The symmetric generalized traveling salesman polytope. *Networks 26* (1995), 113–123.

[13] Garey, M. R., Johnson, D. S. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* Freeman, San Franscisco, CA, 1979.

[14] Glover, F., Laguna, M., Martí, R. Fundamentals of scatter-search and path relinking. *Control and Cybernetics 36* (2000), 653–684.

[15] Golden, B., Raghavan, S., Stanojević, D. Heuristic search for the generalized minimum spanning tree. *INFORMS Journal on Computing 17* (2005), 290–304.

[16] Hao, J., Orlin, J. B. A faster algorithm for finding the minimum cut in a graph. In *SODA '92: Proceedings of the third annual ACM-SIAM symposium on Discrete algorithms* (Philadelphia, PA, USA, 1992), Society for Industrial and Applied Mathematics, p. 165–174.

[17] Haouari, M., Chaouachi, J. S. Upper and lower bounding strategies for the generalized minimum spanning tree problem. *European Journal of Operational Research 171* (2006), 632–647.

[18] Ilher, E., Reich, G., Widmayer, P. Class Steiner trees and VLSI-design. *Discrete Applied Mathematics 90* (1999), 173–194.

[19] Kansal, A. R., Torquato, S. Globally and locally minimal weigth spanning tree networks. *Physica A 301* (2001), 601–619.

[20] Kruskal, J. B. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society 7*, 1 (fevereiro de 1956), 48–50.

[21] Letchford, A. N., Lodi, A. Strengthening Chvátal-Gomory cuts and fractional cuts. *Operations Research Letters 30*, 2 (2002), 74–82.

[22] Magnanti, T. L., Olsey, L. A. Optimal trees. In *Handbooks in Operations Research and Management Science*, C. L. Ball, C. L. Momma, and G. L. Nemhauser, Eds., vol. 7. Elsevier Science, Amsterdam, 1995, p. 503–615.

[23] Myung, Y. S., Lee, C. H., Tcha, D. W. On the generalized minimum spanning tree problem. *Networks 26* (1995), 231–241.

[24] Oncan, T., Cordeau, J.-F., Laporte, G. A tabu search heuristic for the generalized minimum spanning tree problem. Tech report, Center for Research on Transportation, HEC Montréal, Montréal, Canada, 2006. Technical Report.

[25] Pop, P. C. New models of the generalized minimum spanning tree problem. *Journal of Mathematical Modelling and Algorithms 3* (2004), 153–166.

[26] Pop, P. C., Kern, W., Still, G. J. An approximation algorithm for the generalized minimum spanning tree problem with boundend cluster size. Tech report, Department of Operations Research and Mathematical Programing - University of Twente, Twente, The Netherlands, 2001. Technical Report.

[27] Pop, P. C., Kern, W., Still, G. J. The generalized minimum spanning tree problem. Tech report, Department of Operations Research and Mathematical Programing - University of Twente, Twente, The Netherlands, 2001. Technical Report.

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

[29] Raghavan, S. On modeling the generalized minimum spanning tree. Tech report, The Robert Smith School od Business - University of Maryland, Maryland, USA, 2002. Technical Report.

[30] Reinelt, G. TSPLIB - a traveling salesman problem library. *ORSA Journal on Computing 3* (1991), 376–384.

[31] Resende, M., Ribeiro, C. GRASP and path-relinking: Recent advances and applications. In *Proceedings of the Fifth Metaheuristics International Conference (MIC2003)* (2003), T. Ibaraki and Y. Yoshitomi, Eds., p. T1–T6.

[32] Salazar, J. J. A note on the generalized Steiner tree polytope. *Discrete Applied Mathematics 100*, 1-2 (2000), 137–144.

[33] Shyu, S. J., Yin, P. Y., Lin, B. M. T., Haouari, M. Ant-tree: An ant colony optimization approach to the generalized minimum spanning tree problem. *Journal of Experimental and Theoretical Artificial Inteligence 15* (2003), 103–112.

[34] Uchoa, E. Reduction tests for the prize-collecting Steiner problem. *Operations Research Letters 34* (2006), 437–444.

[35] Wolsey, L. A. *Integer programming.* John Wiley & Sons Inc., New York, 1998. A Wiley-Interscience Publication.

APÊNDICE A – Instâncias

Instância	$\|V\|$	$\|E\|$	m	Redução			Geração de Cortes	
				E'	E'/E (%)	T (s)	Limite	T (s)
15spain47	47	985	15	131	13,3	0,01	2393	0,04
27europ47	47	1042	27	111	10,65	0,02	13085	0,05
50gr96africa	96	4463	50	193	4,32	0,08	306	0,22
35gr137america	137	8251	35	898	10,88	0,13	209	0,89
34gr202europe	202	19018	34	1506	7,92	0,71	135	11,13
10att48	48	1010	10	376	37,23	0,01	10923	0,13
10gr48	48	1017	10	418	41,1	0,02	1282	0,17
10hk48	48	995	10	450	45,23	0,02	4119	0,21
11eil51	51	1158	11	481	41,54	0,02	132	0,28
12brazil58	58	1464	12	573	39,14	0,03	9206	0,33
14st70	70	2248	14	638	28,38	0,04	233	0,67
16eil76	76	2660	16	893	33,57	0,06	186	2,04
16pr76	76	2661	16	820	30,82	0,05	46514	1,39
20gr96	96	4292	20	760	17,71	0,07	221	2,15
20rat99	99	4609	20	998	21,65	0,08	402	2,17
20kroa100	100	4727	20	858	18,15	0,08	7982	1,87
20krob100	100	4716	20	924	19,59	0,08	8111	2,05
20kroc100	100	4714	20	897	19,03	0,08	8041	1,2
20krod100	100	4716	20	939	19,91	0,09	7643	1,48
20kroe100	100	4702	20	856	18,21	0,08	8164	1,22
20rd100	100	4703	20	856	18,2	0,08	2779	1,06
21eil101	101	4776	21	1238	25,92	0,11	204	4,5
21lin105	105	5130	21	902	17,58	0,09	6728	4,21
22pr107	107	5441	22	841	15,46	0,37	20398	48,19
24gr120	120	6820	24	1331	19,52	0,13	2255	1,68
25pr124	124	7315	25	888	12,14	0,13	30174	4,96
26bier127	127	7191	26	2595	36,09	0,23	58150	45,15
28pr136	136	8879	28	1422	16,02	0,18	34104	4,37
28gr137	137	8892	28	1537	17,29	0,19	329	20,23
29pr144	144	9952	29	1050	10,55	0,19	40055	2,63
30kroa150	150	10809	30	1399	12,94	0,2	9815	6,2
30krob150	150	10807	30	1505	13,93	0,21	10048	9,57
31pr152	152	11080	31	1229	11,09	0,5	39109	12,36
32u159	159	12031	32	1432	11,9	0,21	18723	7,89
39rat195	195	18478	39	2280	12,34	0,4	751	53,38
40kroa200	200	19409	40	1842	9,49	0,36	11634	20,31
40krob200	200	19430	40	1975	10,16	0,37	11244	31,54
40d198	198	18841	40	2371	12,58	0,44	7044	21,53
41gr202	202	19532	41	2595	13,29	0,9	242	26,18
45ts225	225	24650	45	1834	7,44	0,57	62246	66,26
46pr226	226	24626	46	1522	6,18	0,47	55515	165,55

Tabela A.1: Instâncias do Grupo 1 geográficas e geradas por *cluster centering*.

Instância	$\|V\|$	$\|E\|$	m	Redução			Geração de Cortes	
				E'	E'/E (%)	T (s)	Limite	T (s)
18att48	48	1062	18	224	21,09	0,02	16521	0,11
25eil51	51	1235	25	139	11,26	0,02	242	0,06
24st70	70	2329	24	282	12,11	0,03	297	0,15
36eil76	76	2789	36	255	9,14	0,04	306	0,22
31pr76	76	2770	31	333	12,02	0,05	58038	0,42
33gr96	96	4413	33	418	9,47	0,07	298	0,46
36rat99	99	4753	36	488	10,27	0,09	521	0,28
43kroa100	100	4844	43	315	6,5	0,07	11914	0,31
44krob100	100	4854	44	341	7,03	0,07	12561	0,34
42kroc100	100	4847	42	305	6,29	0,07	12284	0,25
42krod100	100	4846	42	324	6,69	0,08	11827	0,26
42kroe100	100	4846	42	350	7,22	0,08	12292	0,29
36rd100	100	4801	36	395	8,23	0,07	3978	0,35
36eil101	101	4912	36	595	12,11	0,08	295	1,9
45lin105	105	5340	45	363	6,8	0,1	9280	0,42
42pr107	107	5542	42	279	5,03	0,09	23290	0,25
45pr124	124	7440	45	465	6,25	0,13	37837	0,61
50bier127	127	7729	50	1199	15,51	0,19	71221	8,23
60pr136	136	9064	60	432	4,77	0,18	52817	0,85
49gr137	137	9122	49	744	8,16	0,21	391	2,09
48pr144	144	10084	48	487	4,83	0,23	43725	3,69
57kroa150	150	11005	57	622	5,65	0,19	14050	0,89
56krob150	150	10982	56	572	5,21	0,18	13845	0,71
54pr152	152	11254	54	482	4,28	0,32	44253	2,24
58u159	159	12321	58	676	5,49	0,24	24214	1,06
81rat195	195	18745	81	786	4,19	0,5	1111	1,53
72kroa200	200	19636	72	870	4,43	0,35	14881	2,15
76krob200	200	19661	76	851	4,33	0,37	15320	2,44
67d198	198	19101	67	1375	7,2	0,5	8283	10,25
68gr202	202	19826	68	1926	9,71	1,1	292,5	41,65
75ts225	225	24900	75	900	3,61	0,59	79019	3,79
84pr226	226	25118	84	891	3,55	0,66	62527	9,7

Tabela A.2: Instâncias do Grupo 1 geradas por *grid clusterization* com $\mu = 3$.

Instância	$\|V\|$	$\|E\|$	m	Redução			Geração de Cortes	
				E'	E'/E (%)	T (s)	Limite	T (s)
13att48	48	1025	13	301	29,37	0,01	13189	0,16
16eil51	51	1214	16	303	24,96	0,02	158	0,07
16st70	70	2277	16	455	19,98	0,03	214	0,21
16eil76	76	2695	16	786	29,17	0,05	149	0,42
16pr76	76	2661	16	678	25,48	0,06	29788	0,65
22gr96	96	4315	22	692	16,04	0,07	234	1,69
25rat99	99	4692	25	727	15,49	0,09	410	0,48
23kroa100	100	4748	23	807	17	0,09	8054	1,44
25krob100	100	4743	25	627	13,22	0,08	7880	0,45
25kroc100	100	4762	25	755	15,85	0,08	8084	0,88
24krod100	100	4724	24	656	13,89	0,08	8741	2,04
25kroe100	100	4738	25	575	12,14	0,08	8401	0,56
24rd100	100	4742	24	594	12,53	0,08	3077	0,38
25eil101	101	4812	25	929	19,31	0,09	217	2,9
30lin105	105	5237	30	567	10,83	0,1	7410	1,2
22pr107	107	5438	22	1164	21,4	0,7	19877	70,04
25pr124	124	7219	25	785	10,87	0,11	27156	0,61
26bier127	127	7173	26	2277	31,74	0,2	58989	14,46
34pr136	136	8888	34	841	9,46	0,15	37735	2,37
32gr137	137	8908	32	943	10,59	0,18	338	12,07
30pr144	144	9928	30	880	8,86	0,18	36279	5,86
36kroa150	150	10868	36	1054	9,7	0,19	10101	2,69
36krob150	150	10870	36	1264	11,63	0,2	9780	2,13
33pr152	152	11083	33	1179	10,64	0,57	38143	45,06
32u159	159	12046	32	1386	11,51	0,22	17059	7,96
49rat195	195	18600	49	1794	9,65	0,46	795,5	7,61
47kroa200	200	19464	47	1484	7,62	0,36	11628	5,67
48krob200	200	19511	48	1574	8,07	0,38	11113	6,13
40d198	198	18772	40	2047	10,9	0,44	7098	15,67
41gr202	202	19303	41	4216	21,84	1,37	232	176,59
45ts225	225	24726	45	1932	7,81	0,56	60578,7	269,2
50pr226	226	24711	50	1420	5,75	0,61	56721	40,81

Tabela A.3: Instâncias do Grupo 1 geradas por *grid clusterization* com $\mu = 5$.

Instância	$\|V\|$	$\|E\|$	m	Redução			Geração de Cortes	
				E'	E'/E (%)	T (s)	Limite	T (s)
7att48	48	947	7	484	51,11	0,01	6667	0,19
9eil51	51	1143	9	509	44,53	0,02	100	0,3
16st70	70	2277	16	455	19,98	0,03	214	0,22
16eil76	76	2695	16	786	29,17	0,05	149	0,42
16pr76	76	2661	16	678	25,48	0,05	29788	0,63
15gr96	96	4170	15	1153	27,65	0,08	186	2,69
16rat99	99	4583	16	1385	30,22	0,12	308	1,11
16kroa100	100	4647	16	1181	25,41	0,1	5987	3,05
16krob100	100	4652	16	1179	25,34	0,1	6058	1,81
16kroc100	100	4651	16	999	21,48	0,08	5534	1,05
16krod100	100	4647	16	1086	23,37	0,1	5904	1,44
16kroe100	100	4607	16	1008	21,88	0,08	6450	1,99
16rd100	100	4606	16	1004	21,8	0,08	2287	0,66
16eil101	101	4757	16	1427	30	0,1	141	4,04
16lin105	105	5005	16	1248	24,94	0,1	4542	1,77
16pr107	107	5322	16	1401	26,32	0,66	17547	54,12
19pr124	124	7077	19	1037	14,65	0,13	23164	0,76
19bier127	127	7039	19	3439	48,86	0,27	52097	79,92
20pr136	136	8724	20	1665	19,09	0,2	22541	3,59
22gr137	137	8746	22	1604	18,34	0,19	264	29,64
21pr144	144	9762	21	1381	14,15	0,19	33947	14,62
25kroa150	150	10703	25	1721	16,08	0,23	7944	4,74
25krob150	150	10725	25	1756	16,37	0,2	7293	18,08
24pr152	152	11008	24	1352	12,28	0,37	35429	12,13
23u159	159	11896	23	1938	16,29	0,24	12659	5,13
36rat195	195	18454	36	2585	14,01	0,48	639	13,83
35kroa200	200	19335	35	2504	12,95	0,41	9640	20,9
36krob200	200	19335	36	2231	11,54	0,41	9742	38,03
32d198	198	18372	32	3222	17,54	0,48	6501	44,41
31gr202	202	18872	31	5184	27,47	1,36	202	49,76
35ts225	225	24544	35	2649	10,79	0,56	50813	55,45
33pr226	226	24355	33	1890	7,76	0,53	48249	220,32

Tabela A.4: Instâncias do Grupo 1 geradas por *grid clusterization* com $\mu = 7$.

Instância	$\|V\|$	$\|E\|$	m	Redução			Geração de Cortes	
				E'	E'/E (%)	T (s)	Limite	T (s)
7att48	48	947	7	484	51,11	0,01	6667	0,19
9eil51	51	1143	9	509	44,53	0,02	100	0,29
9st70	70	2160	9	925	42,82	0,05	146	0,59
9eil76	76	2545	9	1233	48,45	0,05	94	1,32
9pr76	76	2532	9	1226	48,42	0,06	20501	0,9
15gr96	96	4170	15	1153	27,65	0,08	186	2,69
16rat99	99	4583	16	1385	30,22	0,12	308	1,12
16kroa100	100	4647	16	1181	25,41	0,1	5987	3,02
16krob100	100	4652	16	1179	25,34	0,1	6058	1,8
16kroc100	100	4651	16	999	21,48	0,08	5534	1,06
16krod100	100	4647	16	1086	23,37	0,1	5904	1,41
16kroe100	100	4607	16	1008	21,88	0,08	6450	1,98
16rd100	100	4606	16	1004	21,8	0,08	2287	0,66
16eil101	101	4757	16	1427	30	0,1	141	4,04
16lin105	105	5005	16	1248	24,94	0,11	4542	1,79
12pr107	107	5196	12	1662	31,99	0,66	16754	169,66
14pr124	124	6896	14	1443	20,93	0,13	18554	3,01
14bier127	127	6240	14	3874	62,08	0,22	43778	22,64
16pr136	136	8548	16	1838	21,5	0,16	21732	2,61
15gr137	137	8504	15	2288	26,9	0,22	197	17,21
16pr144	144	9508	16	1375	14,46	0,17	32510	52,46
16kroa150	150	10506	16	2886	27,47	0,31	5229	24,66
16krob150	150	10455	16	2673	25,57	0,26	5494	21,05
16pr152	152	10828	16	2395	22,12	0,49	33340	19,98
23u159	159	11896	23	1938	16,29	0,24	12659	5,14
25rat195	195	18225	25	3996	21,93	0,69	482	34,68
25kroa200	200	19100	25	3757	19,67	0,55	6895	90,38
25krob200	200	19100	25	3517	18,41	0,57	6912	72,65
25d198	198	18149	25	4003	22,06	0,51	6185	246,9
21gr202	202	17904	21	9052	50,56	2,38	177	194,38
25ts225	225	24300	25	4180	17,2	0,63	40339	152,51
27pr226	226	24137	27	2047	8,48	0,46	43389	81,39

Tabela A.5: Instâncias do Grupo 1 geradas por *grid clusterization* com $\mu = 10$.

Instância	$\|V\|$	$\|E\|$	m	Redução			Geração de Cortes	
				E'	E'/E (%)	T (s)	Limite	T (s)
107ali535	535	137733	107	16369	11,88	6,19	114289	10107,89
107att532	532	139216	107	8805	6,32	3,89	12001	5630,01
99d493	493	118406	99	13655	11,53	5,7	16493	14077,1
132d657	657	213650	132	8031	3,76	5,08	19427	8669,22
84fl417	417	84841	84	3501	4,13	1,75	7935	7138,6
53gil262	262	33521	53	2804	8,36	0,85	887	185996,85
87gr431	431	88666	87	14087	15,89	4,08	-	-
134gr666	666	216964	134	18937	8,73	7,73	-	-
64lin318	318	49363	64	2727	5,52	0,98	18471	169474,98
131p654	654	210044	131	8191	3,9	5,79	-	-
113pa561	561	155544	113	8049	5,17	9,9	861	1213,75
89pcb442	442	96360	89	4760	4,94	2,58	19571	924625,58
53pr264	264	34028	53	2894	8,5	0,97	21872	89,01
60pr299	299	43786	60	3210	7,33	0,93	20290	80,48
88pr439	439	94585	88	6448	6,82	2,63	51749	2001,58
115rat575	575	163695	115	6993	4,27	5,32	2168,5	4886,08
157rat783	783	304384	157	9430	3,1	12,51	3009	20264,91
80rd400	400	78754	80	4471	5,68	1,71	5868	1102,01
107si535	535	140799	107	5468	3,88	5	12791	1463,38
115u574	574	163035	115	6477	3,97	3,79	15027	3284,82
145u724	724	259764	145	7913	3,05	6,31	15904	10007,71

Tabela A.6: Instâncias do Grupo 2 geradas por *cluster centering*

Instância	$\|V\|$	$\|E\|$	m	Redução			Geração de Cortes	
				E'	E'/E (%)	T (s)	Limite	T (s)
181ali535	535	140717	181	8689	6,17	5,13	-	-
182att532	532	140049	182	4717	3,37	4,08	15652	853311,08
171d493	493	119945	171	6571	5,48	5,76	20269	2028,67
221d657	657	214467	221	4264	1,99	6,5	25703	927,17
142fl417	417	85821	142	1910	2,23	2,21	8353	200955,21
95gil262	262	33845	95	1200	3,55	0,73	1255	11,98
81gr229	229	25584	81	1947	7,61	0,73	74792	8,35
149gr431	431	90783	149	8531	9,4	4,45	-	-
224gr666	666	218524	224	11236	5,14	9,31	-	-
108lin318	318	49821	108	1697	3,41	1,2	24092	30,55
230p654	654	211794	230	4249	2,01	7,3	23547	174,31
156pcb442	442	96904	156	2217	2,29	3,13	27513	38,44
101pr264	264	34357	101	1274	3,71	1,04	29199	15,22
102pr299	299	44114	102	1484	3,36	0,93	23096	8,18
163pr439	439	95444	163	3094	3,24	2,9	64953	100373,9
196rat575	575	164385	196	3431	2,09	8,1	2880	118,18
285rat783	783	305319	285	4160	1,36	20,45	4203	342,31
135rd400	400	79271	135	2262	2,85	1,65	7632	100,47
198u574	574	163647	198	3115	1,9	5,1	19696	156,18
266u724	724	260733	266	3645	1,4	10	21739	132,47

Tabela A.7: Instâncias do Grupo 2 geradas por *grid clusterization* com $\mu = 3$.

Instância	$\|V\|$	$\|E\|$	m	Redução			Geração de Cortes	
				E'	E'/E (%)	T (s)	Limite	T (s)
108ali535	535	137198	108	15604	11,37	6,14	-	-
110att532	532	138952	110	10094	7,26	4,83	11896	4652,02
102d493	493	118143	102	14335	12,13	6,23	16132	35916,33
137d657	657	213715	137	8058	3,77	6,08	-	-
93fl417	417	84944	93	3126	3,68	2,07	7952	592111,07
63gil262	262	33643	63	2214	6,58	0,8	984	198404,8
47gr229	229	25129	47	3678	14,64	0,77	54305	50,23
87gr431	431	88598	87	16394	18,5	5,7	81856	1964,21
139gr666	666	216025	139	19330	8,95	9,33	-	-
64lin318	318	49320	64	2840	5,76	1,05	17667	73,21
134p654	654	209256	134	7791	3,72	6,34	22377	51432,84
95pcb442	442	96383	95	4564	4,74	3,1	19383	443219,1
55pr264	264	34016	55	2463	7,24	1,01	21351	10,37
69pr299	299	43893	69	2869	6,54	0,98	18582	61,42
96pr439	439	94417	96	6875	7,28	2,89	54230	1272,43
121rat575	575	163862	121	7272	4,44	7,35	2014	893,04
169rat783	783	304584	169	9327	3,06	17,53	2823	2269,93
81rd400	400	78835	81	4294	5,45	1,69	5433,5	1284,7
127u574	574	163044	127	6046	3,71	4,6	15240	2504,63
166u724	724	260039	166	6923	2,66	9,62	15435	1301,3

Tabela A.8: Instâncias do Grupo 2 geradas por *grid clusterization* com $\mu = 5$.

Instância	$\|V\|$	$\|E\|$	m	Redução			Geração de Cortes	
				E'	E'/E (%)	T (s)	Limite	T (s)
83ali535	535	137048	83	20812	15,19	7,38	-	-
80att532	532	137788	80	13809	10,02	5,75	10204	10319,05
78d493	493	116770	78	19899	17,04	7,78	14152	28834,08
96d657	657	212843	96	12625	5,93	6,45	-	-
61fl417	417	83476	61	4355	5,22	1,86	7446	3851,73
49gil262	262	33487	49	2983	8,91	0,82	802	127865,82
34gr229	229	24595	34	4643	18,88	0,79	45989	343982,79
64gr431	431	86817	64	19594	22,57	6,05	-	-
96gr666	666	212798	96	24908	11,7	9,72	-	-
49lin318	318	49106	49	4213	8,58	1,17	14909	312129,17
100p654	654	207125	100	10420	5,03	6,1	-	-
64pcb442	442	95900	64	7614	7,94	3,03	14639,5	4133,11
43pr264	264	33481	43	2730	8,15	0,73	20438	32494,73
47pr299	299	43463	47	3801	8,75	0,96	15238	207047,96
74pr439	439	93659	74	10139	10,83	3,4	47101	2144,72
100rat575	575	163589	100	9914	6,06	7,99	1734,5	5556,01
121rat783	783	303899	121	14790	4,87	17,03	2228	13742,93
64rd400	400	78574	64	5865	7,46	1,72	4576	3075,77
92u574	574	162462	92	9208	5,67	4,95	12186	4035,59
119u724	724	259303	119	10568	4,08	9,23	-	-

Tabela A.9: Instâncias do Grupo 2 geradas por *grid clusterization* com $\mu = 7$.

Instância	$\|V\|$	$\|E\|$	m	Redução			Geração de Cortes	
				E'	E'/E (%)	T (s)	Limite	T (s)
57ali535	535	129989	57	18947	14,58	5,05	-	-
57att532	532	136796	57	20939	15,31	8,91	8494,84	71620,81
52d493	493	114177	52	27390	23,99	9,11	11121	28697,91
73d657	657	212014	73	18446	8,7	7,51	-	-
42fl417	417	82007	42	4760	5,8	1,78	6986	12366,98
36gil262	262	33208	36	4386	13,21	0,9	639	262836,9
23gr229	229	23761	23	7304	30,74	1,35	39793	820192,35
52gr431	431	84391	52	20108	23,83	5,8	62722	353097,8
70gr666	666	208810	70	31123	14,9	11,98	-	-
36lin318	318	48664	36	6461	13,28	1,42	10119	166287,42
69p654	654	203619	69	13132	6,45	5,67	20739	117635,67
48pcb442	442	95084	48	8690	9,14	3,3	11941	2069,57
27pr264	264	32814	27	3942	12,01	0,98	16546	22,28
35pr299	299	43054	35	5569	12,93	1,03	11624	169188,03
48pr439	439	90883	48	15644	17,21	4,37	40518	5042,72
64rat575	575	162675	64	16773	10,31	8,99	1235	4919,19
81rat783	783	302709	81	24809	8,2	19,27	-	-
49rd400	400	78196	49	7799	9,97	1,92	3825	5557,72
64u574	574	161567	64	14874	9,21	6	9755	5792,33
80u724	724	258292	80	18625	7,21	10,32	-	-

Tabela A.10: Instâncias do Grupo 2 geradas por *grid clusterization* com $\mu = 10$.