Instituto de Computação
Universidade Estadual de Campinas

O Problema do Corredor de Comprimento Mínimo: Algoritmos Exatos, Aproximativos e Heurísticos

Lucas de Oliveira

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Lucas de Oliveira e aprovada pela Banca Examinadora.

Campinas, 23 de Maio de 2012.

Cid Carvalho de Souza
Instituto de Computação - UNICAMP
(Orientador)

Dissertação apresentada ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

FICHA CATALOGRÁFICA ELABORADA POR
MARIA FABIANA BEZERRA MULLER - CRB8/6162
BIBLIOTECA DO INSTITUTO DE MATEMÁTICA, ESTATÍSTICA E
COMPUTAÇÃO CIENTÍFICA - UNICAMP

Oliveira, Lucas de, 1987-
OL4p O problema do corredor de comprimento mínimo: algoritmos exatos, aproximativos e heurísticos / Lucas de Oliveira. – Campinas, SP : [s.n.], 2012.

Orientador: Cid Carvalho de Souza.
Dissertação (mestrado) – Universidade Estadual de Campinas, Instituto de Computação.

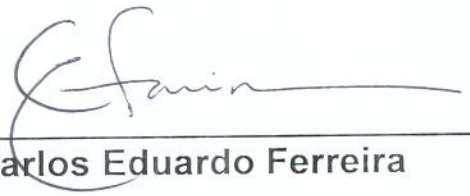
1. Geometria computacional. 2. Algoritmos. 3. Programação inteira. 4. Heurística. 5. Otimização combinatória. I. Souza, Cid Carvalho de, 1963-. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Informações para Biblioteca Digital

Título em inglês: The minimum length corridor problem: exact, approximative and heuristic algorithms
Palavras-chave em inglês:
Computational geometry
Algorithms
Integer programming
Heuristic
Combinatorial optimization
Área de concentração: Ciência da Computação
Titulação: Mestre em Ciência da Computação
Banca examinadora:
Cid Carvalho de Souza [Orientador]
Carlos Eduardo Ferreira
Pedro Jussieu de Rezende
Data de defesa: 23-05-2012
Programa de Pós-Graduação: Ciência da Computação

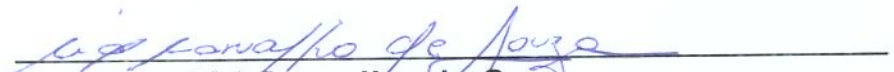
TERMO DE APROVAÇÃO

Dissertação Defendida e Aprovada em 23 de Maio de 2012, pela Banca examinadora composta pelos Professores Doutores:



Prof. Dr. Carlos Eduardo Ferreira
IME / USP

Prof. Dr. Pedro Jussieu de Rezende
IC / UNICAMP



Prof. Dr. Cid Carvalho de Souza
IC / UNICAMP

Instituto de Computação
Universidade Estadual de Campinas

O Problema do Corredor de Comprimento Mínimo: Algoritmos Exatos, Aproximativos e Heurísticos

Lucas de Oliveira[1]

Maio de 2012

Banca Examinadora:

- Cid Carvalho de Souza
  Instituto de Computação - UNICAMP (Orientador)
- Carlos Eduardo Ferreira
  Instituto de Matemática e Estatística - USP
- Pedro Jussieu de Rezende
  Instituto de Computação - UNICAMP
- Fabio Luiz Usberti
  Faculdade de Engenharia Elétrica e de Computação - UNICAMP (Suplente)
- Orlando Lee
  Instituto de Computação - UNICAMP (Suplente)

[1]Suporte financeiro de: Bolsa do CNPq (processo 132185/2010-5) 03/2010–07/2010, Projeto Fapesp (processo 2010/06720-7) 08/2010–01/2012.

Resumo

Esta dissertação tem como foco a investigação experimental de algoritmos exatos, aproximativos e heurísticos aplicados na resolução do chamado *problema do corredor de comprimento mínimo* (PCCM). No PCCM recebemos um polígono retilinear P e um conjunto de polígonos retilineares menores formando uma subdivisão S planar conexa de P. Uma solução para este problema, também chamada de *corredor*, é formada por um conjunto conexo de arestas de S, e tal que cada face interna em S possui pelo menos um ponto em sua borda que pertence a alguma aresta deste conjunto. O objetivo então é encontrar um corredor tal que a soma total dos comprimentos das arestas seja a menor possível.

Trata-se de um problema $\mathcal{NP}$-difícil com aplicações em áreas diversas, tais como telecomunicações, engenharia civil e projeto de circuitos VLSI. O PCCM pode ser reduzido polinomialmente a um problema em grafos denominado problema da árvore de Steiner com grupos (PASG). Considerando esta transformação, estudamos e implementamos dois métodos aproximativos, um método exato de *branch-and-cut*, e um método heurístico baseado na metaheurística GRASP combinada com um *evolutionary path relinking* (GRASP+EPR). Além disso, propomos três heurísticas de busca local que visam melhorar a qualidade de soluções do PASG.

Instâncias do PCCM foram geradas aleatoriamente, nas quais aplicamos os métodos implementados. Analisamos os resultados, e apresentamos as situações onde é interessante utilizar cada método. Verificamos que o método *branch-and-cut* foi capaz de encontrar soluções ótimas para instâncias que julgamos ser de grande porte em tempos computacionalmente aceitáveis. O melhor algoritmo aproximativo obteve corredores que na média têm comprimento 17% maior que o comprimento ótimo. Se combinarmos este algoritmo com as heurísticas de melhoria propostas este percentual cai para a média de 3,5%. Finalmente, o GRASP+EPR consome mais tempo que este algoritmo aproximativo, entretanto, o comprimento dos corredores obtidos por ele é em média 0,9% maior que o comprimento ótimo.

Abstract

This dissertation focuses on the experimental investigation of exact, approximation and heuristic algorithms applied to solve the so-called *minimum length corridor problem* (MLCP). In the MLCP we receive a rectilinear polygon P and a set of minor rectilinear polygons forming a connected planar subdivision S of P. A solution for this problem, also called corridor, is formed by a set of connected edges of S, and such that each inner face of S has at least one point on its your border which belongs to an edge in this set. The goal is to find a corridor such that the sum of lengths of the edges is as small as possible.

This is an $\mathcal{NP}$-hard problem with applications in several areas such as telecommunications, civil engineering and design of VLSI circuits. The MLCP can be polynomially reduced to a graph problem known as group Steiner tree problem (GSTP). Based on this transformation, we studied and implemented two approximation methods, an exact branch-and-cut method, and a heuristic method based on the metaheuristic GRASP combined with an evolutionary path relinking (GRASP+EPR). Furthermore, we propose three local search heuristics to improve the quality of GSTP solutions.

MLCP instances were randomly generated, in which we apply the methods implemented. We analyzed the results, and present situations where it is interesting to use each method. We found that the branch-and-cut has been able to find optimal solutions for instances that we consider to be large in acceptable computational times. The best approximation algorithm obtained corridors having average length 17% higher than the optimum length. If we combine this algorithm with the improvement heuristics proposed this percentage drops to an average of 3.5%. Finally, the GRASP+EPR spent more time than this approximation algorithm, however, the length of the corridors obtained by the method is, on average, 0.9% higher than the optimum length.

Dedicatória

Dedico esta dissertação aos meus pais, Maria e Wilson, em especial ao meu irmão Tarciso e minha cunhada Mônica, às minhas irmãs, Lucimar e Vanderlúcia, e aos meus cunhados, Áriston e José Aloísio, por sempre estarem presentes em todos os momentos da minha vida, e por terem me dado a chance de realizar esse sonho. Amo todos vocês!

Agradecimentos

Primeiramente, agradeço à minha família por todo apoio e amor incondicional. Por serem tão solidários e afetuosos, por sempre acreditarem em mim e estarem presentes em cada momento dessa conquista.

Ao meu orientador Cid, por ser paciente e por me mostrar o caminho para que eu me tornasse cada vez melhor tanto nos estudos quanto como pessoa.

Aos meus amigos Ednaldo, Rilson e Tatiane que me proporcionam ótimos momentos que sempre terei como lembranças.

Aos meus sobrinhos que vivem pegando no meu pé, criando situações engraçadas que trazem mais alegria para minha vida mesmo nas horas mais difíceis.

À família da minha cunhada Mônica, por terem me recebido de braços abertos fazendo com que eu me sentisse em casa todo este tempo que estive morando em Sumaré.

Aos professores e funcionários do Instituto da Computação da UNICAMP.

Aos membros da banca examinadora Carlos, Pedro e Fabio que contribuíram para o aperfeiçoamento desta dissertação.

Ao CNPq e Fapesp pelo suporte financeiro.

Enfim, agradeço de coração a todos que participaram desta conquista.

Muito obrigado!

Sumário

Resumo **vii**

Abstract **ix**

Dedicatória **xi**

Agradecimentos **xiii**

1 Introdução **1**

2 Notação e Preliminares **5**
2.1 Preliminares gerais 5
2.2 Geometria 6
2.3 Grafos 7
2.4 Programação Matemática 9

3 O Problema e Revisão Bibliográfica **13**
3.1 O Problema 13
3.2 Revisão Bibliográfica 17

4 Modelos Matemáticos **21**
4.1 Modelo de Cobertura por Vértices ($\mathcal{M}_{CV}$) 21
4.2 Modelo de Árvore Orientada ($\mathcal{M}_{AO}$) 24

5 Algoritmos Aproximativos **27**
5.1 Aproximação por Árvore Geradora Mínima 28
5.2 Aproximação por Caminhos Mínimos 30

6 Heurísticas de Melhoria **35**
6.1 Remoção de Folhas Desnecessárias 36
6.2 Troca de Folhas de Cobertura Única 38

6.3 Reconexão da Árvore 43

7 Uma Heurística GRASP com *Evolutionary Path Relinking* 47
7.1 Metaheurística GRASP com *Evolutionary Path Relinking* 47
7.2 A Heurística Desenvolvida 49

8 Algoritmo Exato 55
8.1 Funcionamento de um Algoritmo *Branch-and-Cut* Geral 55
8.2 O Algoritmo *Branch-and-Cut* Desenvolvido 60
8.2.1 Modelo Matemático 60
8.2.2 Heurística Primal 62

9 Resultados Computacionais 65
9.1 Geração de Instâncias 65
9.2 Análise dos Resultados 69
9.2.1 Análise dos Algoritmos Aproximativos e Heurísticas de Melhoria . . 70
9.2.2 Análise da Heurística GRASP com *Evolutionary Path Relinking* . . 79
9.2.3 Análise do Algoritmo Exato 83
9.2.4 Comparação entre os Algoritmos Exato, Aproximativos e Heurístico 85

10 Conclusões 91

Lista de Figuras

1.1 Uma instância do PCCM e um corredor. 2

3.1 Quatro diferentes conjuntos de segmentos para uma mesma instância, sendo que apenas os conjuntos em (a) e (b) são considerados viáveis (corredores) para o PCCM. 14
3.2 Ilustração das diferenças entre o PCCM e o PCCM-RE. 15
3.3 Ilustração da idéia de uma redução do PCCM-RE ao PCCM. 16
3.4 Instância do PCCM com um corredor em destaque e sua representação em grafos destacando a árvore que corresponde ao corredor. 17

5.1 Formato de instâncias do PCCM e PCCM-RE, nas quais o fator de aproximação do algoritmo ACM é atingindo assintoticamente. 32
5.2 Exemplo do formato apresentado na figura anterior com $w = 5$ e $z = 1$. . . 33

6.1 Exemplo que ilustra os pontos onde é possível melhorar uma solução obtida pelo algoritmo ACM. 36
6.2 Exemplo que mostra o impacto da escolha de qual caminho desnecessário que será removido. 37
6.3 Exemplo que mostra o impacto da ordem na qual alteramos os caminhos de cobertura única em um solução. 41

8.1 Exemplo de uma árvore de problemas gerada na execução do *branch-and-bound* aplicado a um problema de programação inteira com três variáveis inteiras. 58

9.1 Exemplo das etapas do processo de geração de instâncias proposto. 66
9.2 Exemplo de duas subdivisões com estruturas inadequadas. Em (a) é apresentada uma instância com segmentos que podem ser removidos sem alterar suas soluções ótimas, e em (b) uma subdivisão desconexa que não admite corredores. 66

9.3 Exemplo de uma instância que pode ser resolvida como duas instâncias menores (em termos da quantidade de segmentos). 67
9.4 Outro exemplo de uma instância que pode ser resolvida como duas instâncias menores (em termos da quantidade de segmentos). 67
9.5 Exemplo dos diferentes tipos de instâncias. 69
9.6 Comparação dos tempos de execução médios dos algoritmos AAGM, ACM e *branch-and-cut* (B&C) em função da quantidade de salas nas instâncias. 71
9.7 Comparação da qualidade das soluções obtidas pelos algoritmos AAGM e ACM em função da quantidade de salas nas instâncias. 72
9.8 Comparação das versões *First Fit* e *Best Fit* das heurísticas HD e HC. . . 73
9.9 Comparação das versões *First Fit* e *Best Fit* da heurística HR. 73
9.10 Comparação das combinações das heurísticas considerando a versão *First Fit* da heurística HR′. 75
9.11 Comparação das combinações das heurísticas considerando a versão *Best Fit* da heurística HR′. 75
9.12 Comparação do DRMCC do algoritmo ACM utilizando as combinações de heurísticas selecionadas com o algoritmo ACM puro em função da quantidade de salas. 77
9.13 Comparação do PRM do algoritmo ACM sem utilizar e utilizando as combinações de heurísticas selecionadas em função da quantidade de salas. . . 77
9.14 Comparação do TEM do algoritmo ACM sem utilizar e utilizando as combinações de heurísticas selecionadas em função da quantidade de salas. . . 78
9.15 Comparação do percentual de execuções em que cada um dos algoritmos, empregados na fase de construção do GRASP, encontrou a melhor solução, em função da quantidade de salas. 80
9.16 Comparação do percentual de execuções que nenhum (GRASP), ambos (GRASP+PRA+PRI), ou apenas um dos métodos de *path relinking* encontrou uma solução melhor que a melhor solução encontrada antes de sua execução, em função da quantidade de salas. 80
9.17 Comparação do DRMCC da solução encontrada pelo método com a melhor solução encontrada antes da execução do método em função da quantidade de salas. 81
9.18 Tempo médio gasto na execução da heurística em função da quantidade de salas. 82
9.19 Percentual do tempo total gasto em cada uma das etapas da heurística em função da quantidade de salas. 82

9.20 Tempo de execução médio do algoritmo *branch-and-cut*, considerando separadamente as instâncias do tipo 4 e as demais instâncias, em função da quantidade de salas. 84
9.21 Percentual de instâncias que foram resolvidas enumerando apenas um nó no algoritmo *branch-and-cut* em função da quantidade de salas. 84
9.22 Comparação do DRMCC dos algoritmos aproximativos ACM com e sem heurísticas e da heurística GRASP+EPR com o melhor limitante dual obtido pelo *branch-and-cut* em função da quantidade de salas. 86
9.23 Percentual de execuções onde a heurística GRASP+EPR encontrou o ótimo em função da quantidade de salas. 86
9.24 Comparação do tempo de execução médio dos algoritmos exato, aproximativo e heurístico em função da quantidade de salas (nas instâncias que não são do tipo 4). 88
9.25 Comparação do tempo de execução médio dos algoritmos exato, aproximativo e heurístico em função da quantidade de salas (nas instâncias do tipo 4). 88

Capítulo 1

Introdução

A Geometria Computacional é a área da Ciência da Computação dedicada ao estudo de algoritmos para a resolução de problemas geométricos. Os problemas geométricos estão presentes nas mais diversas áreas, tais como telecomunicações, computação gráfica, transporte, projeto de circuitos VLSI *(Very Large Scale Integration)* e robótica. Uma vez que boa parte destes problemas apresentam um elevado grau de dificuldade, torna-se importante o estudo e desenvolvimento de algoritmos capazes de resolvê-los em um tempo computacional aceitável.

Dentro da Geometria Computacional temos a classe de problemas denominada de *problemas geométricos generalizados*. Os problemas desta classe são genericamente enunciados da seguinte forma: dado um conjunto de objetos no plano, encontre uma rede que conecta cada um dos objetos satisfazendo certas propriedades. Em particular, nesta classe estamos interessados no problema denominado problema do corredor de comprimento mínimo (PCCM). No PCCM é dado um polígono retilinear subdivido em polígonos retilineares menores, formando uma subdivisão S planar conexa, e o que se deseja é encontrar um corredor de comprimento mínimo. Neste contexto, entende-se como sendo um corredor um conjunto conexo de arestas de S que conecta todas as faces internas de S. A figura 1.1 ilustra uma instância para o problema com um corredor em destaque.

O PCCM é $\mathcal{NP}$-difícil e o seu estudo é motivado por sua aplicação nas áreas de telecomunicações, construção civil e projeto VLSI. Tomando como exemplo a área de telecomunicações, imagine a seguinte situação: uma empresa deseja construir um *backbone* que deve atender diversos blocos de uma região metropolitana. Considere que os dutos por onde passam os cabeamentos nesta região formam figuras retilineares, e que a empresa deseja gastar a menor quantidade possível de cabeamento, reduzindo custos e facilitando a manutenção. Desta forma, não é difícil perceber que este problema pode ser convertido e resolvido como um PCCM. De forma análoga, na construção civil temos o problema de distribuir cabos em uma construção com a finalidade de atender todos os cômodos, e em

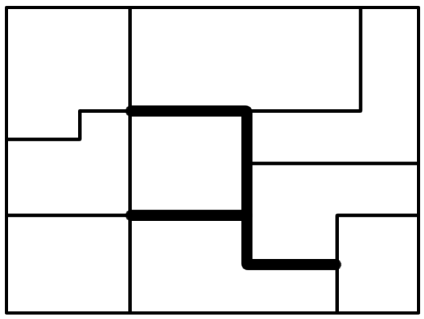


Figura 1.1: Uma instância do PCCM e um corredor.

projetos VLSI, o problema de distribuir fios conectando os componentes em uma placa.

Neste trabalho, realizamos uma revisão bibliográfica, onde constatamos que existem algoritmos exatos e aproximativos para o PCCM. Todavia todos os resultados encontrados são apenas teóricos, ou seja, em nenhum trabalho anterior foi de fato implementada e avaliada sistematicamente a aplicação de algum dos métodos propostos em um conjunto grande de instâncias. Desta forma, o comportamento deste algoritmos na prática é desconhecido. Provavelmente, um fator que contribui para dificultar a avaliação experimental destes métodos, até onde sabemos, é a inexistência de instâncias de domínio público para este problema. Além disso, não existem abordagens utilizando programação linear inteira, nem mesmo heurísticas específicas para o PCCM.

Com base nestes argumentos, decidimos estudar e implementar alguns algoritmos exatos, aproximativos e heurísticos para o PCCM. Inicialmente, a proposta era implementar algoritmos específicos para o problema. Entretanto, optamos por primeiro estudar e implementar algoritmos voltados para um problema mais geral, que tem uma relação bem estreita com o PCCM, denominado problema da árvore de Steiner com grupos (PASG). Constatamos que os resultados obtidos por estes algoritmos são bastante satisfatórios, por isso decidimos continuar a implementação de algoritmos para o PASG. Também, para que fosse possível avaliar experimentalmente estes métodos em instâncias do PCCM, estudamos uma forma de gerar tais instâncias. Desenvolvemos um gerador, com o qual criamos um conjunto de instâncias que será de domínio público, e nas quais aplicamos e avaliamos os métodos implementados.

O texto deste trabalho está estruturado da seguinte forma. O capítulo seguinte introduz notações e conceitos utilizados nesta dissertação. No capítulo 3 apresenta-se a definição formal do problema-alvo estudado e uma revisão da bibliografia relevante, de acordo com os estudos que foram feitos. O capítulo 4 descreve modelagens matemáticas que serão utilizadas pelos métodos apresentados nas seções seguintes. O capítulo 5 apresenta os métodos aproximativos, e no capítulo 6 são descritas as heurísticas propostas para melhorar os resultados destes métodos. No capítulo 7 descreve-se o método heurístico

proposto e que é baseado na combinação da metaheurística GRASP com o método *evolutionary path relinking*. O capítulo 8 apresenta o método exato desenvolvido. No capítulo 9 é feita uma análise experimental comparando os resultados obtidos pelos métodos desenvolvidos. Finalmente, o capítulo 10 elenca as principais contribuições desta dissertação e sugere trabalhos futuros.

Capítulo 2

Notação e Preliminares

Neste capítulo vamos introduzir notações e conceitos que são necessários para o entendimento do texto desta dissertação. Ao leitor que já possui algum conhecimento na área, encorajamos que faça uma leitura superficial deste conteúdo. Qualquer dúvida sobre notação ou conceito poderá ser esclarecida com uma breve consulta aqui.

2.1 Preliminares gerais

Sejam $\mathbb{Z}$ e $\mathbb{R}$, respectivamente, o conjunto dos números inteiros e o conjunto dos números reais. Um conjunto com os elementos e_1, e_2, ... , e_n é denotado por $\{e_1, e_2, ..., e_n\}$. A quantidade de elementos (cardinalidade) de um conjunto A é expressada por $|A|$.

Uma tupla é uma sequência de elementos denotada por $(e_1, e_2, ..., e_n)$. Também a chamamos de n-tupla para explicitar que ela possui n elementos. Um par ordenado é uma 2-tupla (dupla) cujos elementos geralmente são números reais ou inteiros, muito embora, os elementos em um tupla podem ser de domínios (tipos) distintos.

Consideramos que um vetor com n elementos é uma n-tupla. As posições dos elementos em um vetor (tupla) v podem ser indexadas por elementos de um conjunto A. Esta relação de indexação deve ser bijetora, ou seja, cada elemento em A faz referência a (indexa) apenas uma posição em v e, cada posição em v é referenciada (indexada) por apenas um elemento em A. Denotamos o elemento de v cuja posição é indexada pelo elemento $a \in A$ por v_a. Utilizamos a notação A^B para representar o conjunto dos vetores cujas posições são indexadas por B e, tal que os elementos destes vetores são pertencentes ao conjunto A.

Seja x um número, o valor da expressão $\lfloor x \rfloor$ corresponde ao maior número inteiro menor ou igual a x. De forma análoga, o valor da expressão $\lceil x \rceil$ corresponde ao menor número inteiro maior ou igual a x.

A classe de funções O, muito utilizada na análise de algoritmos, é definda da seguinte

forma. Dadas duas funções $f(n)$ e $g(n)$, dizemos que $g(n)$ é $O(f(n))$ se existirem m e c tal que, para todo $n \geq m$, a inequação $g(n) \leq cf(n)$ é satisfeita. Quando $g(n)$ é uma constante dizemos que $g(n)$ é $O(1)$. Um ótimo estudo sobre crescimento de funções, apresentando as classes de funções o, O, ω, Ω e Θ, se encontra no capítulo 3 de Cormen et al. [12].

2.2 Geometria

Nesta seção serão apresentados alguns conceitos básicos da geometria euclidiana. Vamos definir alguns elementos geométricos descritos no plano euclidiano bidimensional.

O plano euclidiano bidimensional é composto pelos pares ordenados (x,y), sendo que x e y pertencem a $\mathbb{R}$. Cada par ordenado (x,y) é chamado de ponto e, x e y são suas coordenadas. A distância entre dois pontos (x_1, y_1) e (x_2, y_2) é um valor real calculado pela fórmula $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$. Esta medida de distância é conhecida como métrica euclidiana.

Uma reta neste plano pode ser descrita por dois parâmetros a e b que são, respectivamente, seu coeficiente angular e linear. Todo ponto (x, y) que pertence a ela satisfaz a equação $y = ax + b$. Três ou mais pontos são colineares se eles pertencerem a uma mesma reta. Sejam p_1 e p_2 dois pontos distintos. Existe uma única reta no plano que passa por eles. Denotamos esta reta por $\overleftrightarrow{p_1p_2}$. O segmento de reta $\overline{p_1p_2}$ é composto por todos os pontos na reta $\overleftrightarrow{p_1p_2}$ que estão entre p_1 e p_2. Estes dois pontos são as extremidades do segmento. O comprimento de um segmento de reta é a distância entre suas extremidades.

A interseção entre duas retas, quando ocorre, corresponde ao conjunto não vazio de pontos que pertencem a ambas. Uma reta é retilinear se ela for paralela aos eixos das abscissas ou ordenadas que forma o plano bidimensional. Estas definições também valem para segmentos de reta. Um conjunto de segmentos é conexo se for possível definir uma sequência incluindo todos os segmentos deste conjunto de tal forma que cada par de segmentos consecutivos na sequência se intersectam. Dizemos que um conjunto de segmentos contém um ponto se este ponto pertencer a algum segmento deste conjunto.

Dada uma sequência com $n \geq 3$ pontos $p_1, p_2, ..., p_{n-1}, p_n$, todos distintos, onde três pontos consecutivos não são colineares, considerando-se consecutivos p_{n-1}, p_n e p_1, assim como p_n, p_1 e p_2, um polígono é composto pelos segmentos $\overline{p_1p_2}$, $\overline{p_2p_3}$, ..., $\overline{p_{n-1}p_n}$, $\overline{p_n, p_1}$. Os pontos na sequência são chamados de vértices, e os segmentos formados por eles são as arestas. Duas arestas são adjacentes se elas compartilham uma mesma extremidade (vértice). Um polígono é dito ser simples se não existir interseção entre suas arestas, exceto nas extremidades de arestas adjacentes. Se todas as arestas do polígono forem segmentos retilineares dizemos que ele é retilinear.

Seja P um polígono retilinear e S um conjunto de segmentos de reta retilineares tendo

ambas as extremidades em P e que, dois a dois, não se interceptem exceto possivelmente em seus extremos. A união de S com as arestas de P define uma subdivisão planar retilinear deste polígono. Os segmentos de reta deste conjunto são chamados de arestas da subdivisão enquanto que os seus extremos compõem o conjunto dos vértices da subdivisão. As faces internas da subdivisão são formadas pelos conjuntos maximais de pontos de P satisfazedo à seguinte propriedade. Seja F um destes conjuntos maximais. Então, dados dois pontos quaisquer p e q em F, existe um conjunto conexo de segmentos inteiramente contidos em P que não intersectam nenhuma aresta da subdivisão e que contém p e q. A chamada face externa da subdivisão é única e formada por todos os pontos do plano externos ao polígono P. Uma subdivisão de P é dita ser conexa se o conjunto contendo todas as suas arestas for um conjunto conexo de segmentos. Neste texto, ao mencionarmos a *borda externa* de uma subdivisão de P, estaremos nos referindo às arestas da subdivisão que correspodem às arestas de P.

2.3 Grafos

Um grafo $G = (V, E)$ consiste de dois conjuntos V e E, cujos elementos chamamos, respectivamente, de vértices (ou nós) e arestas. Uma aresta em E é um par de vértices em V. As arestas representam os relacionamentos entre os vértices do grafo.

Um grafo é não orientado quando as suas arestas são não orientadas, sendo representadas por pares não ordenados, ou seja, consideramos arestas (u,v) e (v,u) iguais. Já em um grafo orientado, também conhecido como dígrafo, as arestas são orientadas e representadas por pares ordenados ($(u,v) \neq (v,u)$). Neste caso, chamamos as arestas orientadas de arcos. Seja um arco (u,v), dizemos que ele parte (sai) de u e termina (entra) em v. Também dizemos que u é o início do arco e v é o término. Uma aresta não orientada (u,v) incide nos vértices u e v, já um arco (u,v) incide apenas no vértice v. Um vértice v é adjacente a um vértice u em um grafo se nele existir a aresta, ou arco, (u,v).

O grau de um vértice v em um grafo G não orientado é a quantidade de arestas que incidem nele. Em um dígrafo D temos o grau de entrada e saída de um vértice v, que corresponde à quantidade de arcos que entram e saem do vértice, respectivamente.

Qualquer grafo pode ser representado através de uma figura no plano. Para isto, basta associar um ponto no plano a cada vértice e traçar uma linha ligando os vértices de cada aresta no grafo. Os arcos são representados através de uma seta que parte do ponto que representa o seu início e vai até o ponto que representa o seu término. Um grafo é planar se ele puder ser desenhado no plano sem que nenhuma de suas arestas (arcos) intercepte outra, exceto, possivelmente, nas suas extremidades.

Um subgrafo de $G = (V, E)$ é um grafo $G' = (V', E')$ com $V' \subseteq V$ e $E' \subseteq E$, sendo $(u, v) \in E'$ com $u, v \in V'$. Um subgrafo de $G = (V, E)$ induzido por um conjunto de

vértices $U \subseteq V$ é um grafo $G' = (U, E')$ onde $E' = \{(u, v) \in E : u, v \in U\}$.

Um caminho em um grafo é uma sequência de vértices v_1,v_2,...,v_n tal que existe no grafo a aresta, ou arco, (v_i,v_{i+1}) para $1 \leq i \leq n-1$. Além disso, cada vértice pode aparecer no máximo uma vez na sequência. Os vértices v_1 e v_n são os extremos do caminho, e os demais são os vértices internos. Em um dígrafo o caminho é orientado. Um s-t caminho é aquele onde o primeiro vértice na sequência é s e o último é t. Seja u um vértice que não está em um s-t caminho, podemos aumentar este caminho adicionando u no início do caminho se s é adjacente a u, ou adicionando u no fim do caminho se u é adjacente a t. Dados dois caminhos dizemos que eles são disjuntos nas arestas se não houver uma mesma aresta em ambos, e disjuntos nos vértices se não houver um mesmo vértice em ambos.

Um grafo não orientado é conexo se para cada par de vértices u e v no grafo existir um u-v caminho. Uma árvore é um grafo não orientado onde existe um único caminho entre cada par de vértices. Em uma árvore, os vértices com grau um são chamados de folhas e os demais de vértices internos. Uma árvore é enraizada quando elegemos um vértice nela para ser sua raiz. Uma arborescência é um dígrafo onde temos um vértice raiz r, e para cada vértice v temos um único r-v caminho no grafo. Quando não houver perigo de ambiguidade, nos permitimos um abuso da linguagem e usamos indistintamente os termos arborescência e árvore.

Dados um grafo G=(V,E) e um conjunto $U \subseteq V$. Se G for não orientado, um corte nele, denotado por $\delta_G(U)$, é o subconjunto de E definido como $\{(u, v) \in E : u \in U, v \in V \setminus U\}$. Ou seja, $\delta_G(U)$ é o conjunto das arestas que incidem em um vértice em U e em um vértice fora de U. Mas se G for orientado, temos dois tipos de corte. O corte de entrada, denotado por $\delta^-_G(U)$ com $U \subseteq V$, é o conjunto dos arcos $\{(v, u) \in E : v \in V \setminus U, u \in U\}$. O corte de saída, denotado por $\delta^+_G(U)$, é o conjunto dos arcos $\{(u, v) \in E : u \in U, v \in V \setminus U\}$. Quando não houver risco de ambiguidade, não se utiliza o subscrito G ao denotar um corte deste grafo.

Em diversas situações é necessária a associação de pesos (ou custos) às arestas (arcos) de um grafo. Isto é feito através de uma função cujo domínio é o conjunto de arestas (arcos) e o contradomínio geralmente é o conjunto $\mathbb{R}$ ou $\mathbb{Z}$. O peso (custo) de um grafo, em termos desta função, é a soma dos pesos das arestas contidas no grafo.

Em teoria dos grafos, existem dois problemas de minimização muito recorrentes em diversos estudos. São eles o problema do s-t caminho mínimo e o problema do s-t corte mínimo. No problema do s-t caminho mínimo, dado um grafo, uma função peso e os vértices s e t, queremos encontrar um s-t caminho no grafo que tem peso menor ou igual a qualquer outro s-t caminho no grafo. O problema do s-t corte mínimo é defindo de forma semelhante, mas neste problema queremos encontrar um s-t corte no grafo que tem peso menor ou igual a qualquer outro s-t corte no grafo. Para o entendimento desta

dissertação será necessário ter um conhecimento básico sobre estes problemas. Para tal, consulte os capítulos 24 e 26 de Cormen et al. [12] onde são realizados estudos detalhados sobre eles e descritos algoritmos para resolvê-los.

2.4 Programação Matemática

Nesta seção vamos introduzir a formulação de modelos matemáticos lineares de otimização e apresentar alguns algoritmos para resolvê-los. Caso o leitor tenha interesse, os livros de Wolsey [47], Nemhauser e Wolsey [34] e Bertsimas e Tsitsiklis [8] cobrem um amplo conteúdo sobre este tema.

Um modelo matemático linear é formado por um conjunto de variáveis e um conjunto de equações lineares que relacionam as variáveis. Os valores que queremos determinar no problema são representados no modelo pelas variáveis. Já as equações modelam as restrições do problema, impondo quais valores as variváveis podem assumir. Pode-se utilizar equações na forma de igualdade, desigualdade de maior ou igual e desigualdade de menor ou igual. Além das restrições do problema, também temos que especificar as restrições de domínio das variáveis.

O domínio das variáveis dita qual é o tipo do modelo e, como veremos mais a frente, também dita a complexidade de sua resolução. Quando todas as variáveis têm domínio real temos um modelo linear contínuo e, quando todas têm domínio inteiro temos um modelo linear inteiro. Se houverem ambos os tipos de variáveis temos um modelo linear inteiro misto.

Em problemas de otimização, como estamos otimizando um valor (custo ou peso) associado a cada solução, incluimos no modelo uma expressão chamada de *função objetivo*. Esta expressão calcula o valor que estamos otimizando em função das variáveis.

Abaixo é dado um modelo matemático linear contínuo genérico com n variáveis e m igualdades (restrições) para um problema de minimização. Nele, x_i são as variáveis e a_{ij}, b_j e c_i os coeficientes das equações, sendo $1 \leq i \leq n$ e $1 \leq j \leq m$. Note que poderiamos ter utilizado desigualdades de maior ou igual ($\geq$) e menor ou igual ($\leq$), mas nos restringimos

a igualdades neste exemplo.

$$\text{Minimizar} \qquad c_1x_1 + c_2x_2 + ... + c_{n-1}x_{n-1} + c_nx_n \tag{2.1}$$

Sujeito a:

$$a_{1,1}x_1 + a_{1,2}x_2 + ... + a_{1,n-1}x_{n-1} + a_{1,n}x_n = b_1 \tag{2.2}$$

$$a_{2,1}x_1 + a_{2,2}x_2 + ... + a_{2,n-1}x_{n-1} + a_{2,n}x_n = b_2 \tag{2.3}$$

$$\vdots \tag{2.4}$$

$$a_{m-1,1}x_1 + a_{m-1,2}x_2 + ... + a_{m-1,n-1}x_{n-1} + a_{m-1,n}x_n = b_{m-1} \tag{2.5}$$

$$a_{m,1}x_1 + a_{m,2}x_2 + ... + a_{m,n-1}x_{n-1} + a_{m,n}x_n = b_m \tag{2.6}$$

$$x_1, x_2, ..., x_{n-1}, x_n \in \mathbb{R} \tag{2.7}$$

Também é possível apresentar um modelo de uma forma mais compacta através do uso de matrizes. Para tal, considere que A é uma matriz contendo os coeficientes $a_{i,j}$ onde i é o índice da linha e j da coluna. Também considere que c, b e x são matrizes coluna (vetores). Desta forma a função objetivo seria cx, as restrições seriam descritas por $Ax = b$ e o domínio das variáveis por $x \in \mathbb{R}^n$.

Existem também formas genéricas de se apresentar um modelo matemático. Este tipo de modelo descreve de forma analítica as restrições de qualquer instância do problema. Por exemplo, considere o problema da cobertura de vértices mínima enunciado da seguinte forma. Dado um grafo G=(V,E) encontre um subconjunto de vértices $C \subseteq V$ tal que cada aresta em E incide em pelo menos um vértice em C, e a cardinalidade de C é a menor possível.

Neste problema temos que decidir quais vértices estarão contidos em C. Desta forma, criamos uma variável x_v para cada $v \in V$. Se o valor de x_v for igual a um consideramos que v está em C, e se for igual a zero consideramos que v não está em C. Sendo assim, o domínio de x será o subconjunto $\{0, 1\} \subset \mathbb{Z}$. A única restrição imposta no problema é que para cada aresta $(u, v) \in E$, pelo menos um dos dois vértices adjacentes a ela deve estar em C, ou seja, a soma $x_u + x_v$ deve ser maior ou igual a um. Finalmente, a função objetivo é a cardinalidade de C que é dada pela soma das variáveis x. Com isso, concluímos a formulação do problema como um modelo linear inteiro, que é dado logo abaixo.

$$\text{Minimizar} \qquad \sum_{v \in V} x_v \tag{2.8}$$

Sujeito a:

$$x_u + x_v \geq 1 \qquad \forall \quad (u, v) \in E, \tag{2.9}$$

$$x_v \in \{0, 1\} \qquad \forall \quad v \in V. \tag{2.10}$$

A resolução de um modelo linear depende do domínio de suas variáveis. Para os modelos lineares contínuos existem algoritmos que os resolvem em tempo polinomial como,

por exemplo, o método dos elipsóides (veja capítulo 8 de Bertsimas e Tsitsiklis [8]). Outro método muito empregado na resolução destes modelos é método simplex (veja capítulo 3 de Bertsimas e Tsitsiklis [8]). Este método gasta tempo exponencial no pior caso mas, na prática, atinge bons tempos de computação em muitos casos. Já os modelos que possuem variáveis inteiras são de difícil resolução e, até agora, não se sabe se é possível resolvê-los em tempo polinomial. Todos algoritmos conhecidos atualmente para estes modelos tem complexidade de tempo exponencial. No capítulo 8 descreveremos dois métodos para resolver modelos lineares inteiros, que são os métodos *branch-and-bound* e *branch-and-cut*.

É importante ressaltarmos também uma situação que ocorrerá nos modelos apresentados no capítulo 4. Existem alguns modelos matemáticos que possuem uma quantidade exponencial de equações. Neste caso, a princípio, parece ser impossível resolvê-los gastando tempo polinomial, pois a entrada possui tamanho exponencial. Contudo, é possível solucioná-los sem considerar explicitamente todas as suas equações.

Isto pode ser feito através de um método chamado *planos de corte*. Mas para explicar este método, antes será necessário entender o chamado *problema da separação*. Neste problema, são dados um vetor x e um conjunto $\mathcal{F}$ de restrições relativas às variáveis deste vetor. O objetivo é verificar se x viola alguma restrição em $\mathcal{F}$ e, em caso afirmativo, identificar pelo menos uma restrição violada.

Com isso, o método de planos de corte para solucionar tal modelo funciona da seguinte forma. Considere que $\mathcal{M}$ é o modelo que queremos resolver e $\mathcal{F}$ é o conjunto exponencial de restrições neste modelo. Inicialmente, otimize o modelo $\mathcal{M}'$ formado pelas restrição de $\mathcal{M}$ que não estão em $\mathcal{F}$. Seja x^* a solução encontrada, resolva o problema da separação para x^* e $\mathcal{F}$. Se for encontrada uma restrição violada em $\mathcal{F}$, a acrescentamos em $\mathcal{M}'$, reotimizamos o modelo e novamente resolvemos o problema da separação. Este processo é repetido até x^* satisfazer todas as restrições em $\mathcal{F}$. Quando isso ocorrer o método termina, pois x^* é uma solução para $\mathcal{M}$.

A restrição que é inserida em $\mathcal{M}'$ a cada iteração é chamada de *corte*. Este nome foi dado porque ela elimina a solução x^* atual do conjunto de soluções. A rigor, na pior situação o método descrito acima vai inserir todos os cortes de $\mathcal{F}$ em $\mathcal{M}'$ e terminar sua execução. Mas ainda assim gastaríamos pelo menos tempo exponencial. Entretanto, com base nos resultados apresentados em Grötschel et al. [25], sabemos que se for gasto tempo polinomial na resolução do problema de separação, o método de planos de corte consumirá um tempo total que é polinomial no tamanho da instância do problema.

Capítulo 3

O Problema e Revisão Bibliográfica

Neste capítulo são apresentados formalmente o problema alvo desta dissertação, e os problemas relacionados, que serão abordados nos próximos capítulos. Por fim, é feita uma breve revisão bibliográfica destes problemas.

3.1 O Problema

O problema do corredor de comprimento mínimo (PCCM) é definido do seguinte modo. Considere um polígono retilinear subdividido em polígonos retilineares menores não sobrepostos, formando uma subdivisão planar S retilinear e conexa. Sejam V, E e F, respectivamente, o conjunto de vértices, arestas e faces de S. Considere que a face externa (ilimitada) de S não faz parte de F. Um *corredor* em S é um subconjunto C de segmentos de E, tal que *(i)* dados dois segmentos e e f quaisquer de C, saindo de uma extremidade qualquer de e, é possível chegar a qualquer das extremidade de f percorrendo-se apenas segmentos de C e *(ii)* para toda face f em F, existe pelo menos um ponto na borda de f que é intersectado por um segmento de C.

A cada um dos segmentos e pertencentes a E está associado um valor não-negativo c_e que corresponde ao seu comprimento euclidiano. Com isto, define-se o *comprimento* $c(C)$ de um corredor C como sendo a soma total dos comprimentos euclidianos dos seus segmentos, ou seja, $c(C) = \sum_{e \in C} c_e$. Assim, no PCCM, dado S, o objetivo é encontrar um corredor de comprimento mínimo.

Vale notar que, na literatura, é usual que as faces de S sejam chamadas de "salas". Isto porque pode-se imaginar que S represente a subdivisão em salas de um andar de um prédio. Os segmentos separando duas salas podem ser vistos como áreas de circulação, então nesta situação um corredor de *comprimento mínimo* representa a menor forma de conectar todas as salas entre si utilizando estas áreas.

A título de ilustração, considere a figura 3.1 onde são apresentados quatro corredores

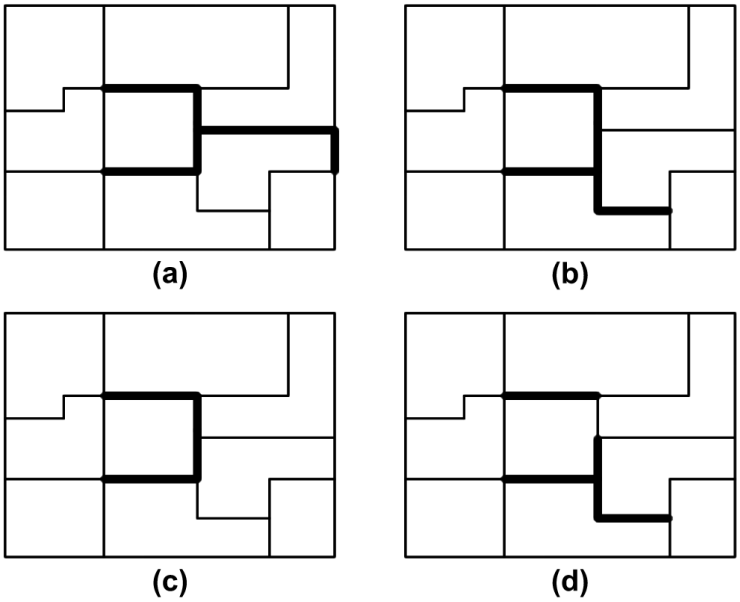


Figura 3.1: Quatro diferentes conjuntos de segmentos para uma mesma instância, sendo que apenas os conjuntos em (a) e (b) são considerados viáveis (corredores) para o PCCM.

para uma mesma instância. Nestas figuras os segmentos dos corredores estão destacados em negrito. Na figura 3.1(a) é apresentado um corredor conectando todas as salas. A figura 3.1(b) também apresenta um corredor conectando todas as salas, porém este corredor possui um comprimento menor que o anterior. A figura 3.1(c) apresenta um conjunto de segmentos com comprimento total menor que os anteriores, no entanto existe uma sala onde nenhum ponto está presente nestes segmentos. Desta forma, este conjunto de segmentos não pode ser considerado como uma solução (corredor) para o problema. A figura 3.1(d) apresenta um conjunto de segmentos que intersectam pontos em todas as salas. Entretanto, os segmentos não são conexos e, por isso, este conjunto de segmentos também não pode ser considerado como um corredor.

Segundo Demaine e O'Rourke [14], Naoki Katoh foi o primeiro a propor o PCCM e o fato ocorreu durante a 12ª Conferência Canadense de Geometria Computacional (CCCG) em 2000. A complexidade do problema permaneceu desconhecida durante alguns anos até que Gonzalez-Gutierrez e Gonzalez [24] e Bodlaender et al. [9] provaram simultaneamente, e de forma independente, que o problema é $\mathcal{NP}$-difícil.

É importante ressaltar que existem diferenças entre os problemas tratados por Gonzalez-Gutierrez e Gonzalez [24] e Bodlaender et al. [9]. Neste capítulo foi descrita a versão do PCCM adotada por Bodlaender et al. [9]. No PCCM abordado por Gonzalez-Gutierrez e Gonzalez [24], denominado daqui em diante como PCCM-RE, são acrescentadas as seguintes restrições ao problema: *(i)* a borda externa de S deve ter formato retangular e

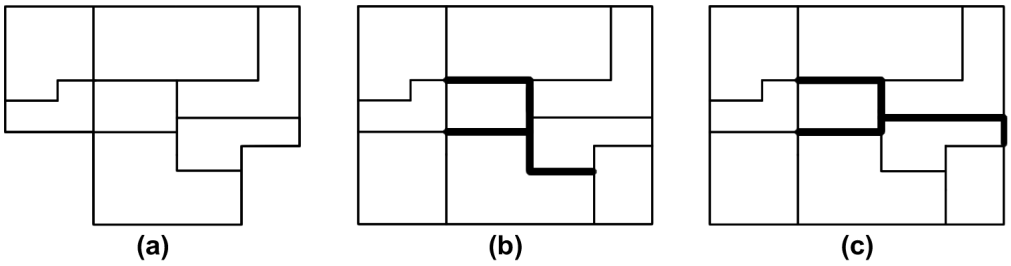


Figura 3.2: Ilustração das diferenças entre o PCCM e o PCCM-RE.

(ii) as soluções do problema devem ser corredores que, além de conectar todas as salas, também devem estar conectados a um ponto da borda externa de S. A justificativa para conectar o corredor à borda externa se deve ao fato de permitir a entrada e saída da subdivisão.

Para que fique mais claro, ilustramos as diferenças entre o PCCM e o PCCM-RE na figura 3.2. Na figura 3.2 (a) é apresentada uma instância do PCCM que não é uma instância do PCCM-RE, pois a borda externa dela não é retangular. Já na figura 3.2 (b) temos uma instância de ambos problemas, onde os segmentos destacados em negrito constituem um corredor para o PCCM que não está ligado à borda externa, portanto sendo inviável para o PCCM-RE. Finalmente, na figura 3.2 (c) é destacado em negrito um corredor para ambos problemas.

Uma vez que toda instância do PCCM-RE também é uma instância do PCCM, é natural nos questionarmos se é possível reduzir o PCCM-RE ao PCCM. Para isto, ao resolver uma instância do PCCM-RE com um algoritmo do PCCM, precisamos encontrar uma maneira de garantir que os corredores sempre estejam conectados à borda de S. A seguir vamos apresentar uma forma bem simples de efetuar esta redução, e que até onde sabemos ainda não foi descrita na literatura.

A idéia desta redução é ilustrada através do exemplo apresentado na figura 3.3. Nesta figura é dada uma instância do PCCM-RE constituída pelos segmentos não tracejados. Além dela, criamos também uma nova sala acrescentando os segmentos tracejados, formando uma instância do PCCM. A criação desta nova sala pode ser feita de forma genérica da seguinte forma. Selecione uma aresta de S situada em sua borda externa, e crie uma nova sala, seguindo a idéia ilustrada na figura, envolvendo os demais segmentos da borda externa de S. Ao criar esta sala, qualquer corredor estará ligado à borda de S para conectar a nova sala às demais. É fácil ver também que os segmentos tracejados nunca estarão presentes em uma solução ótima. Com isso, qualquer solução da instância do PCCM-RE é uma solução para instância do PCCM, e qualquer solução da instância do PCCM que não possui segmentos tracejados é uma solução para instância do PCCM-RE. Finalmente, para qualquer sistema de codificação razoável para uma instância de entrada, é possível

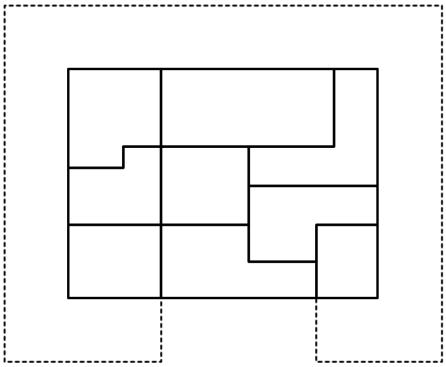


Figura 3.3: Ilustração da idéia de uma redução do PCCM-RE ao PCCM.

acrescentar esta nova sala em tempo polinomial. Sendo assim, fica claro que a redução está correta.

Através desta redução fica evidente que os métodos de resolução para o PCCM podem ser facilmente adaptados para solucionar instâncias do PCCM-RE. Por este motivo, nesta dissertação nosso foco será a investigação de métodos para resolução do PCCM. Decidimos abordá-lo como um problema em grafos e, para isso, vamos reduzi-lo ao chamado problema da árvore de Steiner com grupos (PASG). O PASG é enunciado da seguinte forma. Dados um grafo G com pesos nas arestas e uma coleção $\Re$ de conjuntos de vértices chamados de *grupos*, o objetivo deste problema é encontrar uma árvore de peso mínimo em G que contém pelo menos um vértice de cada grupo. No decorrer deste e dos próximos capítulos, para todo $R \in \Re$, dizemos que um vértice cobre o grupo R somente se ele pertencer ao grupo. Dizemos também que uma árvore em G cobre $\Re$ somente se houver pelo menos um vértice de cada grupo na árvore.

Para efetuar a redução do PCCM ao PASG, primeiramente, vamos converter a subdivisão S em um grafo. Não é difícil ver que S dá origem a um grafo planar $G = (V, E)$ com os vértices representando os pontos de V e cujas arestas correspondem aos segmentos em E (veja figura 3.4). Vamos associar pesos p_e não-negativos a cada aresta $e \in E$ correspondendo ao comprimento c_e do segmento que ela representa. A segunda etapa da redução consiste em criar um grupo para cada face f em F contendo todos os vértices em V que representam um ponto na borda de f. Note que um mesmo vértice pode estar contido em mais de um grupo. Feito isso, terminamos a redução obtendo uma instância do PASG. A transformação de uma solução do PCCM para uma solução do PASG, e vice versa, é imediata, e seus respectivos comprimento e peso são iguais. Portanto, isto mostra que a redução está correta. A figura 3.4 ilustra uma instância do PCCM com um corredor em destaque, e sua respectiva representação na forma de um grafo destacando a árvore que corresponde ao corredor.

As instâncias do PASG, obtidas da redução de instâncias do PCCM, podem ser trans-

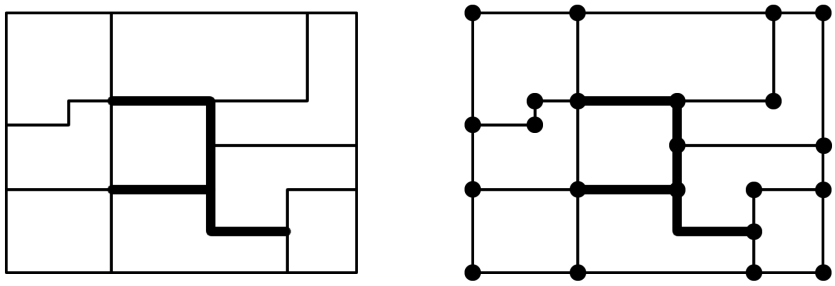


Figura 3.4: Instância do PCCM com um corredor em destaque e sua representação em grafos destacando a árvore que corresponde ao corredor.

formadas de modo a se eliminar todos os vértices de grau dois. Isso é possível porque os grupos cobertos por um vértice com grau dois também são cobertos pelos vértices adjacentes a ele. Portanto, dada uma solução qualquer para uma instância, se houver uma folha na árvore com grau dois, pode-se remover a aresta ligada a esta folha sem que nenhum grupo deixe de ser coberto. Fazendo isso, obtemos uma solução de peso menor. Então, tais soluções nunca serão ótimas, e podem ser desconsideradas. Uma vez que qualquer vértice de grau dois, se estiver presente em uma solução ótima, sempre será um vértice interno, podemos aplicar o seguinte procedimento que o remove do grafo e transforma as duas arestas ligadas a ele em uma só. Seja v um vértice com grau dois em G e $e_1 = (u,v)$ e $e_2 = (v,w)$ as arestas que incidem em v. Remova do grafo v, e_1, e_2 e acrescente uma nova aresta (u,w) com peso igual a $p_{e_1} + p_{e_2}$. Este processo pode ser repetido até que não haja mais vértices com grau dois no grafo. Note que a cada vez que esta operação é feita vamos obter um grafo com um vértice e uma aresta a menos. Isso diminui o tamanho da instância e possivelmente deixa sua resolução mais rápida. Ferreira e Oliveira [19] desenvolveram outras formas de se remover vértices e arestas do grafos e vértices dos grupos. Os autores também realizaram experimentos computacionais mostrando a efetividade das remoções em conjuntos de instâncias do PASG.

Como veremos na seção seguinte, existem na literatura diversos problemas relacionados ao PCCM e também abordagens voltadas para sua resolução.

3.2 Revisão Bibliográfica

Nesta seção será feita uma revisão dos trabalhos anteriores que apresentam problemas relacionados ao PCCM e propõem métodos voltados para a resolução dos mesmos. Esta revisão foi baseada naquela realizada por Arturo Gonzalez-Gutierrez em sua tese de doutorado Gonzalez-Gutierrez [23]. Após esta tese, até agora, foi encontrado apenas o artigo de Bodlaender et al. [10].

São conhecidas diversas variações do PCCM-RE, sendo que aqui vamos descrever apenas as mais interessantes para esta dissertação. Uma delas foi proposta por Eppstein [17] que acrescentou uma condição que determina que as salas devem ter o formato retangular. Esta variação do problema é denominada PCCM-RE-R. Outra variação deste problema, denominada PCCM-RE$_k$, consiste em restringir o formato das salas em k-gonos retilineares, ou seja, cada sala deve ter no máximo k lados.

Todo ponto que está situado na borda externa de S é considerado um ponto de acesso, pois ele pode ser visto como uma forma de conectar um corredor com o exterior de S. Gonzalez-Gutierrez [23] apresenta uma variação dos problemas PCCM-RE e PCCM-RE-R, denominada p-PCCM-RE e p-PCCM-RE-R, respectivamente, impõe que toda solução deve conter um determinado ponto de acesso p. Quando o ponto de acesso considerado nos problemas p-PCCM-RE e p-PCCM-RE-R está situado no canto superior mais à direita da borda de S, estes problemas também são chamados de TRA-PCCM-RE e TRA-PCCM-RE-R, respectivamente.

Ainda em Gonzalez-Gutierrez [23], o autor demonstrou que as variações do PCCM-RE descritas acima também fazem parte da classe dos problemas $\mathcal{NP}$-completos. Ele também propõe um algoritmo de aproximação com fator constante de 30 para o PCCM-RE-R. Isto significa que as soluções obtidas por esse algoritmo são no máximo 30 vezes piores que a solução ótima para qualquer instância do problema. Esse mesmo algoritmo pode ser adaptado para o PCCM-RE$_k$, todavia o fator de aproximação depende do parâmetro k. Quando o parâmetro k é constante, ou seja, o número de lados das salas é limitado por uma constante, o fator de aproximação também é constante. São feitas também outras adaptações deste algoritmo para a resolução dos problemas *group* TSP[1] e *rectangular group* TSP. Descrições mais detalhadas sobre estes dois últimos problemas podem ser encontradas nos trabalhos Dumitrescu e Mitchell [15], Safra e Schwartz [41] e Elbassioni et al. [16].

Em Bodlaender et al. [10] os autores propõem algoritmos de aproximação para versões mais restritas do PCCM. Considere a seguinte variação do PCCM. Dada uma instância do problema, define-se q como sendo o comprimento do lado do quadrado que pode ser inscrito em qualquer sala da instância. O perímetro de cada sala deve ser no máximo $c \cdot q$, onde c é constante e maior ou igual a 4. Segundo Gonzalez-Gutierrez [23] essa variação é também denominada problema de clusterização geográfica (*geografic clustering problem*). Para esta variação do PCCM, Bodlaender et al. [10] propõem um esquema de aproximação polinomial (PTAS) baseado nos algoritmos apresentados nos trabalhos de Arora [2] e Arora [3]. Este algoritmo possui complexidade temporal de $n^{O(\frac{1}{\varepsilon})}$ e fator de aproximação de $(1 + \varepsilon)$, sendo n o número de vértices de S. Uma segunda variação abordada por Bodlaender et al. [10] considera que cada sala P_i tem tamanho p_i definido

[1] A sigla TSP é usada aqui para denotar o problema do caixeiro viajante, de acordo com a sua denominação em língua inglesa *Traveling Salesman Problem*.

pelo comprimento do lado do menor quadrado que inscreve a sala e perímetro de no máximo $4 \cdot p_i$. Outra restrição a ser satisfeita é que todas as salas devem ser α-*fat*. Uma sala P_i é dita ser α-*fat*, se para qualquer quadrado Q que faz interseção com a sala e cujo centro está contido nela, a área de interseção de P_i e Q é no mínimo $\frac{\alpha}{4}$ vezes a área de Q. Para salas quadradas α é igual a 1 e para salas retangulares α tende a zero. O algoritmo de aproximação proposto por Bodlaender et al. [10] tem fator de aproximação $\frac{16}{\alpha} - 1$. Quando o problema é restrito a salas quadradas, esse algoritmo passa a ter um fator constante de 15, ou seja, metade do fator que o algoritmo proposto por Gonzalez-Gutierrez [23] atinge. Além dos algoritmos de aproximação, Bodlaender et al. [10] também propuseram um algoritmo exato para o PCCM e para o problema denominado *tree face cover problem*. O algoritmo possui complexidade temporal $O(n^3 + 2^{9.5539p}n)$ para grafos p-exoplanares e $O(2^{10.1335\sqrt{n}})$ para grafos planares, onde n representa a quantidade de vértices no grafo.

Uma variação mais geral do PCCM-RE consiste em permitir que exista mais de um corredor conectando todas as salas aos pontos de acesso situados na borda de S. Esta versão é denominada PCCM-RE$_f$. Quando os pontos de acesso são restritos a um subconjunto de pontos situados na borda de S, o problema passa a ser chamado de MA-PCCM-RE$_f$. Gonzalez-Gutierrez [23] estabelece que estas variações também fazem parte da classe de problemas $\mathcal{NP}$-completo.

O PCCM-RE também pode ser generalizado para um problema de grafo denominado *network* PCCM-RE ou N-PCCM-RE. Dado um grafo $G(V, E, p)$ não-dirigido onde $p : E \to R^+$ é uma função que define os pesos de cada aresta do grafo, o objetivo deste problema é encontrar uma árvore de peso mínimo que possui pelo menos um vértice de cada ciclo presente em G. Segundo Gonzalez-Gutierrez [23] o problema N-PCCM-RE também é denominado *tree feedback node set* (TFNS). Outro problema descrito em Gonzalez-Gutierrez [23], semelhante ao TFNS, é o *tree vertex cover* (TVC). Este problema possui um algoritmo de aproximação com taxa constante que pode ser adaptado para o PCCM-RE-R [4, 6, 11]. Todavia essa adaptação em geral não obtém boas soluções para o problema PCCM-RE-R e suas variações [23].

O PCCM também é visto como um caso particular do problema da árvore de Steiner com grupos (PASG). Este problema faz parte da classe de problemas $\mathcal{NP}$-completo e foi proposto por Reich e Widmayer [36]. O PASG foi definido na seção anterior. Algumas variações do problema PASG também são generalizações do PCCM, como por exemplo a variação citada em Bodlaender et al. [10] denominada *generalized geometric Steiner tree problem*.

Diversos estudos abordando o problema PASG são encontrados na literatura. Uma excelente referência sobre o problema é Oliveira [35]. Nela são apresentados algoritmos aproximativos para o problema, e testes de redução de instâncias. Também é realizada uma investigação poliédrica propondo algumas formulações matemáticas para o PASG.

Por fim, é proposto e avaliado um algoritmo *branch and cut* que foi capaz de encontrar soluções ótimas para diversas instâncias encontradas na literatura. Demais estudos do problema podem ser encontrados em Ihler [28], Bateman et al. [5], Helvig et al. [27], Slavik [42] e Slavik [43].

Capítulo 4

Modelos Matemáticos

Neste capítulo serão apresentados modelos matemáticos de programação linear inteira (PLI) para o problema de árvore de Steiner com grupos (PASG). Como visto anteriormente, o PCCM é um caso particular deste problema, portanto tais modelos podem ser utilizados para resolvê-lo. Estes modelos serão de importância nos próximos capítulos onde serão introduzidos algoritmos que os utilizam de alguma forma.

Durante este capítulo estamos supondo que o leitor já esteja familiarizado com modelos matemáticos de programação linear e programação linear inteira, e com as técnicas que permitem formular problemas de otimização combinatória com o auxílio destas ferramentas (para tal, consulte Wolsey [47]).

No que se segue, considere as instâncias do PASG dadas pela tupla $(G,\Re)$, onde $G = (V,E)$ é um grafo não orientado com pesos associados às arestas definidos pela função $p : E \to \mathbb{R}_+$, e $\Re$ é uma coleção de conjuntos de vértices (grupos) contidos em V.

4.1 Modelo de Cobertura por Vértices ($\mathcal{M}_{CV}$)

O modelo apresentado nesta seção é encontrado no trabalho de Slavik [43]. Nele são utilizadas as variavéis $x \in \{0,1\}^E$ e $y \in \{0,1\}^V$ para representar, respectivamente, as arestas e os vértices do grafo G. Entenda por $x_e \in \{0,1\}$ a variável que representa a aresta $e \in E$, e $y_v \in \{0,1\}$ a variável que representa o vértice $v \in V$. Uma aresta ou vértice faz parte da solução do modelo quando o valor de sua respectiva variável for igual a um, e não faz parte quando for igual a zero.

Neste modelo vamos fixar um vértice v_0 que fará parte de qualquer solução. Este artifício é necessário para formular a restrição que garante que a solução obtida seja conexa. Esta fixação somente é possível para instâncias que possuem ao menos um grupo com apenas um vértice, pois neste caso podemos afirmar que o vértice deste grupo fará parte de qualquer solução. Entretanto, dada uma instância qualquer do PASG, é possível

adaptá-la de forma a modificar um grupo qualquer em $\Re$ para ter apenas um vértice.

A adaptação é feita da seguinte forma. Dada uma instância do PASG, acrescente em G um novo vértice r. Escolha um grupo R' qualquer em $\Re$. Para cada $v \in R'$ acrescente em E a aresta (r,v), e após isso, substitua o grupo R' pelo grupo $\hat{R}$ que tem apenas o vértice r ($\hat{R} = \{r\}$). Vamos denominar estas arestas adicionadas de *artificiais*. O peso associado às arestas artificiais deve ser escolhido de forma apropriada para não obtermos uma solução ótima para a instância adaptada que será infactível para a instância original. Para isto, este peso é ajustado de modo que ele seja maior que o peso dos caminhos mínimos, em termos de p, de todos os pares de vértices pertencentes a R'. Isto é suficiente para que, em qualquer solução ótima desta instância, exista apenas uma aresta artificial. Não é dificil perceber que removendo o vértice r e a aresta artificial de uma solução ótima da instância adaptada teremos uma solução ótima para a instância original. Com este artifício é possível utilizar o modelo abaixo para resolver qualquer instância do PASG.

A formulação do modelo $\mathcal{M}_{CV}$ é dada pelas equações (4.1)-(4.6). Considere $R_0 = \{v_0\}$ como sendo o grupo unitário escolhido para ter seu vértice v_0 fixado.

$$\text{Minimizar} \sum_{e \in E} p(e) x_e \tag{4.1}$$

Sujeito a:

$$y_{v_0} = 1, \tag{4.2}$$

$$\sum_{v \in R} y_v \geq 1, \qquad \forall \quad R \in \Re, R \neq R_0, \tag{4.3}$$

$$\sum_{e \in \delta_G(S)} x_e \geq y_v, \qquad \forall \quad S \subseteq V \setminus \{v_0\}, v \in S, \tag{4.4}$$

$$x_e \in \{0,1\}, \qquad \forall \quad e \in E, \tag{4.5}$$

$$y_v \in \{0,1\}, \qquad \forall \quad v \in V. \tag{4.6}$$

A equação (4.1) consiste na função objetivo que visa minimizar o peso da solução. A restrição (4.2) determina que o vértice v_0 esteja presente em qualquer solução obtida pelo modelo. A restrição (4.3) garante que cada grupo seja coberto. A restrição (4.4) impõe que os vértices em qualquer solução estejam conectados ao vértice v_0, garantindo que a solução seja conexa. Finalmente as restrições (4.5) e (4.6) definem o domínio das variáveis x e y.

Considere agora a relaxação linear deste modelo dada pela troca das restrições (4.5) e (4.6) pelas restrições (4.7) e (4.8).

$$0 \leq x_e \leq 1, \qquad \forall \quad e \in E, \tag{4.7}$$

$$0 \leq y_v \leq 1, \qquad \forall \quad v \in V. \tag{4.8}$$

Nos referimos a esta relaxação como modelo $\mathcal{M}_{CV}^R$. Como visto na seção 2.4, não é viável inserir todas as restrições no modelo $\mathcal{M}_{CV}^R$ e resolvê-lo através de um método de programação linear, pois a quantidade de restrições em (4.4) é exponencial. Portanto, visando resolver este modelo de uma forma mais eficiente, vamos analisar o problema de separação para estas restrições.

O problema da separação em questão pode ser enunciado da seguinte forma. Sejam $x^f \in \{0,1\}^E$ e $y^f\{0,1\}^V$ vetores que satisfazem todas as restrições do modelo $\mathcal{M}_{CV}^R$, à exceção, eventualmente, de algumas inequações da restrição (4.4). O que queremos é encontrar um conjunto $S \subseteq V \setminus \{v_0\}$ e um vértice $v \in S$ que correspondem a uma inequação em (4.4) violada por esta solução ou, então, afirmar que esta solução respeita todas as inequações da restrição (4.4). Este problema pode ser formalizado através da equação (4.9). Existe uma inequação violada em (4.4) se, e somente se, o valor ótimo de (4.9) for menor que zero. Se isto ocorrer, então um conjunto S e um vértice v que correspondem a uma solução ótima para este problema estão associados a uma inequação com a maior violação possível nesta família de restrições.

$$\text{Minimizar} \sum_{e \in \delta_G(S \cup \{v\})} x_e^f - y_v^f, \qquad S \subseteq V \setminus \{v_0\}, v \in V \tag{4.9}$$

Vamos apresentar uma maneira simples de resolver o problema da equação (4.9) em tempo polinomial. Ao fixar $v = v'$ onde v' é um vértice qualquer em $V \setminus \{v_0\}$, chega-se ao problema dado pela equação (4.10). Note que, agora, $y_{v'}^f$ é uma constante. Portanto, podemos resolver o problema de minimização do somatório separadamente e posteriormente subtrair o valor desta constante do resultado obtido.

$$\text{Minimizar} \sum_{e \in \delta_G(S \cup \{v'\})} x_e^f - y_{v'}^f, \qquad S \subseteq V \setminus \{v_0\} \tag{4.10}$$

Considere a função de capacidade $c' : E \to \mathbb{R}_+$ definida como $c'(e) = x_e^f$. Não é difícil perceber que o problema de minimização do somatório da equação (4.10) reduz-se ao problema de s-t corte de capacidade mínima em G onde a capacidade das arestas é dada por c', e os vértices s e t são, respectivamente, iguais a v' e v_0.

Então, solucionando o problema da equação (4.10) para todo vértice em $V \setminus \{v_0\}$ e tomando o valor mínimo dentre os ótimos assim encontrados, chega-se ao valor ótimo do problema dado pela equação (4.9). Desta forma, resolver este problema equivale a encontrar $|V| - 1$ s-t cortes de capacidade mínima. Como se sabe, é possível calcular um s-t corte de capacidade mínima em tempo polinomial (veja Cormen et al. [12], seção 26), logo, a solução do problema da separação para a família de restrições (4.4) pode ser obtida em tempo polinomial. Sendo assim, conclui-se que é possível otimizar o modelo $\mathcal{M}_{CV}^R$ em tempo polinomial através de um método de planos de corte, o qual foi visto na seção 2.4.

4.2 Modelo de Árvore Orientada ($\mathcal{M}_{AO}$)

O modelo apresentado nesta seção foi proposto por Oliveira [35]. Ele foi obtido através do estudo das formulações matemáticas propostas para o problema da árvore de Steiner, o qual consiste em um caso particular do PASG.

O primeiro passo para entender esta nova modelagem é considerar a versão orientada do grafo G. Assim, dado um grafo $G = (V,\ E)$, denota-se por $D = (V,\ A)$ a versão orientada de G, onde os arcos (u,v) e (v,u) estão em A se existir a aresta (u,v) em E. A função $p : E \to \mathbb{R}_+$ é redefinida para $p : A \to \mathbb{R}_+$ onde cada arco passa a ter o mesmo peso que estava associado à aresta que o gerou.

A seguinte relação é a chave para entender a idéia por trás do modelo desta seção. Qualquer árvore em G pode ser transformada em uma árvore enraizada em D que liga os mesmos vértices e tem o mesmo peso, basta apenas escolher um vértice qualquer da árvore em G para ser a raiz da árvore em D. Em contrapartida, qualquer árvore enraizada em D pode ser convertida em uma árvore em G que liga os mesmos vértices e que também tem peso igual.

Portanto, as soluções do PASG serão representadas no dígrafo D como árvores enraizadas. Primeiramente é preciso determinar uma raiz para a solução. Para isso, escolhe-se um grupo qualquer pertencente a $\Re$ para ser considerado como grupo raiz R_0. Ao menos um vértice deste grupo fará parte de qualquer solução, então um vértice dentre os vértices do grupo pode ser escolhido para ser a raiz da árvore.

São utilizadas no modelo as variáveis $x \in \{0,1\}^A$ e $y \in \{0,1\}^{R_0}$ para representar, respectivamente, os arcos e a raiz da árvore. Denota-se por x_a a variável que representa o arco $a \in A$, e y_v a variável que representa o vértice raiz $v \in R_0$. De forma análoga ao modelo anterior, um arco ou vértice raiz pertence à solução do modelo somente se o valor de sua respectiva variável for um, e não pertence se for zero. É importante ressaltar que não será necessária nenhuma adaptação nas instâncias como foi feito antes, já que qualquer grupo em $\Re$ pode ser escolhido para ser o grupo raiz.

A formulação do modelo $\mathcal{M}_{AO}$ é dada pelas equações (4.11)-(4.15).

$$\text{Minimizar} \sum_{a \in A} p(a) x_a \tag{4.11}$$

Sujeito a:

$$\sum_{v \in R_0} y_v = 1, \tag{4.12}$$

$$\sum_{a \in \delta_D^+(S)} x_a \geq \sum_{v \in S \cap R_0} y_v, \quad \forall \quad R \in \Re \setminus \{R_0\}, S \subseteq V \setminus R, \tag{4.13}$$

$$x_a \in \{0,1\}, \quad \forall \quad a \in A, \tag{4.14}$$

$$y_v \in \{0,1\}, \quad \forall \quad v \in R_0. \tag{4.15}$$

A equação (4.11) consiste na função objetivo que visa minimizar o peso da solução. A restrição (4.12) impõe a escolha de um vértice em R_0 para ser a raiz da árvore e cobrir este grupo. A restrição (4.13) garante a existência de um caminho que liga o vértice raiz a pelo menos um vértice de cada um dos demais grupos. Desta forma todos os grupos serão cobertos e a solução será conexa. As restrições (4.14) e (4.15) definem o domínio das variáveis x e y.

De forma análoga ao modelo anterior, considere a relaxação linear da formulação acima dada pela substituição das restrições (4.14) e (4.15) pelas restrições abaixo:

$$0 \leq x_a \leq 1, \qquad \forall \quad a \in A, \tag{4.16}$$

$$0 \leq y_v \leq 1, \qquad \forall \quad v \in R_0. \tag{4.17}$$

Nos referimos a esta relaxação como modelo $\mathcal{M}_{AO}^R$. Novamente estamos diante de um programa linear com uma quantidade exponencial de restrições. Por este motivo, vamos apresentar como resolver o problema da separação para a família de restrições (4.13).

Sejam $x^f \in \{0,1\}^A$ e $y^f \in \{0,1\}^{R_0}$ vetores que respeitam as restrições do modelo $\mathcal{M}_{AO}^R$, mas podendo não satifazer todas as inequações da restrição (4.13). O problema da separação para a família de restrições (4.13) é dado pela equação (4.18). Existe uma equação violada em (4.13) se, e somente se, o valor ótimo do problema da equação (4.18) for menor que zero.

$$\text{Minimizar} \sum_{a \in \delta_D^+(S)} x_a^f - \sum_{v \in S \cap R_0} y_v^f, \qquad R \in \Re \setminus \{R_0\}, S \subseteq V \setminus R \tag{4.18}$$

Oliveira [35] propôs resolver este problema da seguinte forma. Primeiramente considere o dígrafo $D' = (V', A')$, inicializado como sendo igual a D, e a função $c' : A' \to \mathbb{R}_+$ definida como $c'(a) = x_a^f$, onde $a \in A'$. Adicione em D' um vértice s e, para cada $v \in R_0$, um arco $a = (s,v)$ com $c'(a) = y_v^f$.

Seja $R' \in \Re \setminus \{R_0\}$. Observe que um s-R' corte $\delta_{D'}^+(S \cup \{s\})$ satisfaz a condição $S \subseteq V \setminus R'$. Considere c' como uma função capacidade. Então, a capacidade de um s-R' corte é definida pela equação:

$$\sum_{a \in \delta_{D'}^+(S \cup \{s\})} c'(a) = \sum_{a \in \delta_D^+(S)} x_a^f + \sum_{v \in R_0 \setminus S} y_v^f \tag{4.19}$$

$$\tag{4.20}$$

Subtraindo a constante $\sum_{v \in R_0} y_v^f$ em ambos os lados da equação acima chega-se na equação:

$$\sum_{a \in \delta_{D'}^+(S \cup \{s\})} c'(a) - \sum_{v \in R_0} y_v^f = \sum_{a \in \delta_D^+(S)} x_a^f - \sum_{v \in S \cap R_0} y_v^f \tag{4.21}$$

Portanto, ao resolver um problema de s-R' corte de capacidade mínima e subtrair a constante $\sum_{v \in R_0} y_v^f$, chega-se a solução da equação (4.18) para $R = R'$. Desta forma, o problema da separação para a família de restrições (4.13) se resume a encontrar um s-R corte de capacidade mínima para cada $R \in \Re \setminus \{R_0\}$ e, dentre estes, escolher aquele de menor capacidade. Seja $\delta_G^+(X)$ o s-R corte de menor capacidade escolhido. Então $X \setminus \{s\}$ induz uma restrição violada em (4.13) se, e somente se, o valor da capacidade deste corte menos $\sum_{v \in R_0} y_v^f$ for menor que zero. Caso contrário, não existe nenhuma inequação violada por x^f e y^f nesta família de restrições.

O problema de encontrar um s-R corte de capacidade mínima pode ser reduzido a um problema de encontrar um s-t corte de capacidade mínima. Para tanto, basta acrescentar um vértice t no dígrafo e ligar cada vértice $v \in R$ ao vértice t por meio de um arco de capacidade infinita. Deste modo, o problema da separação para a família de restrições (4.13) consiste em resolver $|\Re| - 1$ problemas de s-t corte de capacidade mínima. Isso deixa claro que este problema pode ser solucionado em tempo polinomial. Por isso, como vimos antes, a otimização do modelo $\mathcal{M}_{AO}^R$ também pode ser feita em tempo polinomal.

Por fim, verificamos que existe uma adaptação do algoritmo desenvolvido por Rao e Orlin para o problema de corte irrestrito mínimo[1] [26], que também pode ser empregada na resolução destes $|\Re| - 1$ problemas de s-t corte de capacidade mínima. A adaptação é apresentada em Cronholm et al. [13], onde o algoritmo modificado é utilizado em uma rotina de separação para um modelo do problema da árvore de Steiner. Os autores mostram experimentalmente que este algoritmo obtém melhores resultados quando comparado a um algoritmo que resolve múltiplos problemas de s-t corte de capacidade mínima separadamente. No nosso caso, vamos utilizar o algoritmo adaptado para encontrar um s-R corte de capacidade mínima. Para isso, primeiramente é preciso preparar o dígrafo. Para cada $R \in \Re \setminus \{R_0\}$ adicione um novo vértice w no dígrafo e ligue cada vértice v em R ao vértice adicionado por um arco de capacidade infinita (v,w). Feito isso, aplicamos o algoritmo adaptado no dígrafo modificado para encontrar um corte de capacidade mínima que contém s e não contém pelo menos um dos vértices adicionados, e assim obtemos um menor s-R corte de capacidade mínima.

[1] O problema de corte irrestrito mínimo é semelhante ao problema do s-t corte mínimo, diferindo apenas que no primeiro problema s e t podem ser quaisquer vértices do grafo.

Capítulo 5

Algoritmos Aproximativos

Na literatura pode-se encontrar algoritmos aproximativos voltados para a resolução de casos particulares do PCCM. Gonzalez-Gutierrez [23] propôs um algoritmo aproximativo para o PCCM-RE com fator de aproximação constante 30 quando as salas constituem polígonos retangulares (PCCM-RE-R). O autor também mostra que é possível extender este algoritmo para quando as salas tem formato de polígonos com no máximo k lados (PCCM-RE$_k$). Entretanto, o fator de aproximação para este algoritmo passa a ser $15k$, deixando de ser constante.

Bodlaender et al. [10] propôs algoritmos aproximativos para dois casos particulares do PCCM. Uma das variações do PCCM abordada por este autor é denominada problema de clusterização geográfica. Neste problema, seja q o comprimento do lado de um quadrado que pode ser inscrito em qualquer sala e, considere que o perímetro de cada sala é no máximo $c \cdot q$, sendo c uma constante maior ou igual a 4. Para este caso, o autor desenvolveu um esquema de aproximação polinomial com fator de aproximação $(1 + \varepsilon)$ e complexidade temporal $n^{O(\frac{1}{\varepsilon})}$. A outra variação abordada pelo autor deve respeitar as seguintes restrições. Seja p_i o tamanho de uma sala P_i definido pelo comprimento do lado do menor quadrado que inscreve a sala. Neste problema o perímetro de cada sala P_i deve ter comprimento total de no máximo $4 \cdot p_i$. Além disso todas as salas devem ser α-*fat*. Uma sala é dita ser α-*fat* se, para qualquer quadrado Q que faz interseção com a sala e esteja centrado no seu interior, a área de interseção de P_i e Q é no mínimo $\frac{\alpha}{4}$ vezes a área de Q. O algoritmo aproximativo proposto pelo autor para este problema tem fator de aproximação $\frac{16}{\alpha} - 1$.

Dentre todas estas variações do PCCM que possuem algoritmos aproximativos, o PCCM-RE$_k$ é a mais próxima de ser adaptável para tratar também o PCCM. Entretanto, esta versão faz uso de um algoritmo aproximativo para o PASG que poderia ser aplicado diretamente ao PCCM. Por este motivo, e também pelo fato de estarmos interessados no estudo do PCCM de forma geral, decidimos estudar dois algoritmos aproximativos para

o PASG.

Novamente, neste capítulo considere as instâncias do PASG dadas pela tupla $(G,\Re)$, onde $G = (V,E)$ é um grafo não orientado com pesos associados as arestas definidos pela função $p : E \to \mathbb{R}_+$, e $\Re$ é uma coleção de conjuntos de vértices (grupos) em V.

5.1 Aproximação por Árvore Geradora Mínima

O algoritmo aproximativo apresentado nesta seção foi proposto por Slavik [43]. A idéia deste algoritmo é extrair informações de uma solução ótima do modelo $\mathcal{M}^R_{CV}$, visto na seção 4.1. Baseado nos valores desta solução são determinados quais vértices de G são interessantes selecionar para cobrir $\Re$.

A partir destes vértices, é possível construir um subgrafo induzido G' de G, e obter uma árvore geradora mínima de G' que será uma solução para o PASG. Contudo, nem sempre G' será conexo. Para contornar este problema vamos calcular a árvore geradora do grafo induzido G'_k de G_k, onde G_k é o grafo completo obtido de G acrescentando as arestas que faltam para torná-lo completo da seguinte forma. Faça G_k igual a G. Seja (u,v) uma aresta que não está em G, onde $u, v \in V$. Encontre um u-v caminho mínimo, em termos de p, e adicione em G_k a aresta (u,v) associando a esta aresta o peso deste caminho mínimo. Note que dado um subgrafo de G_k é possível obter um subgrafo de G correspondente com peso menor ou igual substituindo cada aresta que não pertence a G pelo caminho que a gerou. Ao fazer esta substituição, pode-se ter um subgrafo com ciclos. Por isso, além de fazer a substituição, também deve ser feita uma busca em largura para verificar a existência de ciclos. Ao encontrar um ciclo, o mesmo é eliminado do subgrafo removendo-se dele a aresta de maior peso.

Considere por x^* e y^* uma solução ótima do modelo $\mathcal{M}^R_{CV}$. A decisão de quais vértices serão selecionados para cobrir os grupos em $\Re$ será feita utilizando os valores de y^*. Para isso, será construído um conjunto de vértices X que contém pelo menos um vértice de cada grupo da seguinte forma. Dado um grupo $R \in \Re$ qualquer, pela restrição (4.3) do modelo sabe-se que existe pelo menos um vértice v em R tal que $y^*_v \geq \frac{1}{|R|}$ pois, caso contrário, a restrição não poderia estar satisfeita. Sendo assim, seja ρ a maior cardinalidade de um grupo em $\Re$. Todo vértice v de V que pertence a algum grupo, e tal que $y^*_v \geq \frac{1}{\rho}$, é incluído em X. Isso garante que pelo menos um vértice de cada grupo estará contido em X.

O algoritmo 1 formaliza os passos discutidos acima e, doravante, será denotado por AAGM. Vamos verificar que ele possui complexidade de tempo polinomial. Como vimos na seção 4.1, a otimização do modelo $\mathcal{M}^R_{CV}$ pode ser realizada em tempo polinomial. Sabemos também que existem algoritmos que encontram uma árvore geradora mínima de um grafo completo gastando $O(|V|^2)$ operações (cf. Cormen et al. [12], capítulo 23). Desta forma nos resta apenas analisar os passos 2, 3 e 5. No passo 2 gasta-se $O(|V|)$

operações para verificar quais vértices estão em X. Para montar o subgrafo no passo 3, são gastas $O(|V|^2)$ operações, pois G_k é um grafo completo. Já no passo 5, também são gastas $O(|V|^2)$ operações na conversão das arestas de G_k para G e na eliminação de ciclos. Vale a pena ressaltar também que gerar o grafo G_k a partir de G é uma tarefa que pode ser realizada gastando $O(|V|^3)$ operações empregando-se o algoritmo de Floyd-Warshall (descrito em Cormen et al. [12], seção 25.2). Calcular ρ gasta-se $O(|\Re||V|)$ operações. Finalmente, podemos concluir que a complexidade total deste algoritmo é polinomial.

Entrada: Uma instância $I = (G = (V, E), \Re)$ do PASG.
Saída: Uma solução T para I.

Passo 1: Resolva o modelo $\mathcal{M}^R_{CV}$ para I obtendo uma solução ótima (x^*,y^*).
Passo 2: Para cada vértice $v \in V$, $v \in X$ se v pertencer a algum grupo em $\Re$ e $y^*_v \geq \frac{1}{\rho}$.
Passo 3: Construa o subgrafo G'_k de G_k induzido por X.
Passo 4: Encontre uma árvore geradora mínima T' para G'.
Passo 5: Converta T' para um subgrafo T de G e o retorne.

Algoritmo 1: Algoritmo aproximativo baseado no conceito de árvore geradora mínima (algoritmo AAGM).

Proposição 1. *Seja T uma solução obtida pelo algoritmo AAGM e T^* uma solução ótima de I. Se os pesos associados às arestas de G respeitarem a desigualdade triangular, então temos que $p(T) \leq p(T^*)(2 - \frac{2}{|X|})\rho$.*

A prova desta proposição é extensa, fazendo uso de diversos lemas e modelos matemáticos. Os detalhes da mesma podem ser encontrados em Slavik [43].

Com base na proposição 1, podemos afirmar que o algoritmo AAGM é um algoritmo aproximativo com fator de aproximação da ordem de ρ. Observe que este algoritmo se torna interessante para instâncias onde a quantidade de vértices nos grupos é limitada por uma constante, pois, nesta situação, ele terá um fator de aproximação constante. Foi com base nesta idéia que Gonzalez-Gutierrez [23] obteve um algoritmo de aproximação constante para o PCCM-RE-R. O autor mostrou uma forma de reduzir este problema ao PASG limitando a quantidade de vértices nos grupos com a garantia de um fator de aproximação constante. Todavia, esta demonstração vale apenas para o caso em que as salas possuem formato retangular.

5.2 Aproximação por Caminhos Mínimos

Nesta seção será apresentado um algoritmo aproximativo baseado na idéia de construir uma solução encontrando caminhos mínimos entre os vértices na solução parcial corrente e os vértices de grupos ainda não cobertos. Este algoritmo surgiu de uma modificação sugerida por Ihler [28] para a heurística de caminhos mínimos desenvolvida por Reich e Widmayer [36] para o PASG.

A heurística de caminhos mínimos é bastante simples e funciona da seguinte forma. Seja v_0 um vértice qualquer de algum grupo. Inicialize a árvore T composta apenas por v_0. Enquanto T não cobrir algum grupo em $\Re$, adicione em T um caminho mínimo, em termos de p, que liga um vértice em T a um vértice de um grupo que ainda não foi coberto. Ao cobrir todos os grupos, a heurística termina retornando T como uma solução.

Ihler [28] mostrou que escolher um vértice de forma arbitrária para ser o primeiro vértice em T pode fazer com que o peso da solução obtida por esta heurística seja tão alto quanto quisermos. Por isso, a seguinte modificação foi proposta por ele. Escolha um grupo $R_0 \in \Re$ qualquer, denominado grupo raiz. Para cada $v \in R_0$ aplique a heurística de caminhos mínimos fazendo v_0 igual a v. Os passos do algoritmo proposto por Ihler [28] são apresentados no algoritmo 2, denominado algoritmo ACM.

Entrada: Uma instância $I = (G = (V, E), \Re)$ do PASG.
Saída: Uma solução T para I.

Passo 1: Faça $T = \emptyset$ e $p(T) = \infty$.
Passo 2: Escolha o grupo raiz $R_0 \in \Re$ e faça $H = R_0$.
Passo 3: Retire um vértice v_0 de H.
Passo 4: Inicialize a árvore T' com v_0.
Passo 5: Se T' cobre todos os grupos em $\Re$ vá para o passo 7, caso contrário prossiga para o próximo passo.
Passo 6: Seja W o conjunto de vértices pertencentes a algum grupo não coberto, e P o conjunto de todos os caminhos que iniciam em um vértice em T' e terminam em um vértice em W. Encontre um caminho de menor peso, em termos de p, em P e o adicione em T' mantendo-a como uma árvore. Feito isso, volte ao passo 5.
Passo 7: Se $p(T') < p(T)$ faça $T = T'$.
Passo 8: Se H não estiver vazio volte ao passo 3, caso contrário prossiga para o próximo passo.
Passo 9: Retorne T.

Algoritmo 2: Algoritmo aproximativo baseado no conceito de caminhos mínimos (algoritmo ACM).

Na heurística de Reich e Widmayer procura-se por um caminho mínimo no máximo

$O(|\Re|)$ vezes. Para encontrar um caminho mínimo gasta-se $O(|E|\log|V|)$ operações (cf. Cormen et al. [12], capítulo 24). Com a modificação proposta por Ihler, esta heurística é executada no máximo ρ vezes, onde ρ corresponde a maior quantidade de vértices em um grupo. Desta forma, a complexidade de tempo do algoritmo ACM é $O(\rho|\Re||E|\log|V|)$.

Proposição 2. *Seja T uma solução obtida pelo algoritmo ACM e T^* uma solução ótima de I. Então temos que $p(T) \leq (|\Re| - 1)p(T^*)$.*

Demonstração. Vamos reproduzir aqui a prova realizada por Oliveira [35]. Seja r_0 um vértice em T^* que cobre o grupo raiz R_0 escolhido no passo 2 do algoritmo. Considere por $\hat{T}$ a solução T' que atinge o passo 7 do algoritmo quando v_0 for escolhido igual a r_0 no passo 3. Vamos verificar que $p(\hat{T}) \leq (|\Re| - 1)p(T^*)$.

Para isso, sejam as tuplas (t_1,v_1),...,(t_k,v_k) representando os caminhos que vão de um vértice t_i em $\hat{T}$ até um vértice de um grupo não coberto v_i escolhidos na i-ésima iteração do passo 6, tendo v_0 igual a r_0 e $i = 1,...,k$. Como T^* cobre $\Re$, então para cada v_i temos um vértice t_i^* em T^* que pertence ao mesmo grupo de v_i. Considere que $T^*(u,v)$ seja o caminho em T^* que sai de u e termina em v. Seja $p^*(u,v)$ o peso de um caminho mínimo, em termos de p, que liga os vértices u e v. Então na i-ésima execução do passo 6 temos a seguinte relação.

$$p^*(t_i, v_i) \leq p^*(v_0, v_i) \leq p(T^*(v_0, v_i)) \leq p(T^*) \tag{5.1}$$

Uma vez que o passo 6 é executado $k \leq (|\Re| - 1)$ vezes segue a relação.

$$p(\hat{T}) = \sum_{i=1}^{k} p^*(t_i, v_i) \leq \sum_{i=1}^{k} p(T^*) = kp(T^*) \leq (|\Re| - 1)p(T^*) \tag{5.2}$$

O peso da solução T obtida pelo algoritmo é menor ou igual ao peso da solução $\hat{T}$ e, portanto, menor ou igual a $(|\Re| - 1)p(T^*)$.

□

Com base neste resultado, tem-se que o algoritmo ACM possui fator de aproximação $(|\Re| - 1)$. Oliveira [35] demonstrou que não é possível melhorar este fator de aproximação assintoticamente. Como estamos lidando com um caso particular do PASG, vamos provar que para o PCCM, e para o PCCM-RE, também não é possível melhorar assintoticamente este fator de aproximação.

Para tal, vamos construir uma instância do PCCM, e do PCCM-RE, com $w + 3$ salas, respeitando o formato dado na figura 5.1. Neste formato cria-se uma estrutura aninhando $w \geq 1$ salas que geram um segmento vermelho na esquerda e um segmento azul na direita. Os segmentos destacados em vermelho, verde e azul possuem comprimentos, respectivamente, iguais a w, $w + 3z$ e z, com $z > 0$. A sala destacada como R_0 é um

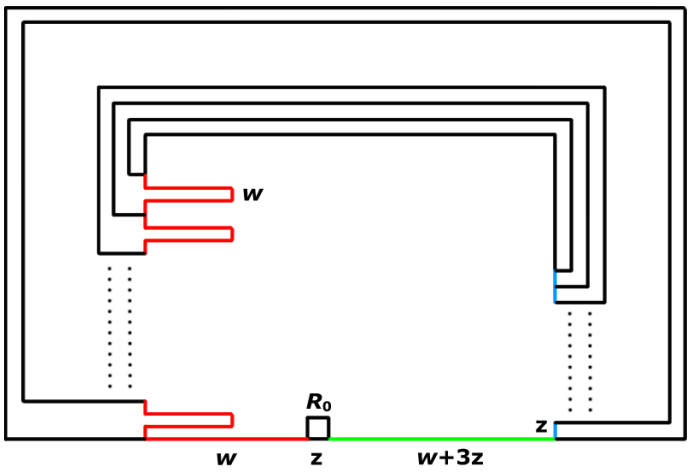


Figura 5.1: Formato de instâncias do PCCM e PCCM-RE, nas quais o fator de aproximação do algoritmo ACM é atingindo assintoticamente.

quadrado de lado com comprimento igual a z. Um exemplo para w igual 5 e z igual a 1 é dado na figura 5.2.

O corredor mínimo para uma instância que segue este formato é composto pelos segmentos azuis e o verde, e tem comprimento igual $(w + 3z) + wz = w + z(w + 3) = w + \varepsilon$, sendo $\varepsilon = z(w+3)$. Ao transformar a mesma instância em uma instância do PASG, e considerar a sala R_0 como grupo raiz, a solução obtida pelo algoritmo ACM será composta pelas arestas que correspondem aos segmentos vermelhos. Portanto, ela corresponde a uma solução do PCCM com comprimento igual $(w + 1)w$. Desta forma, o fator de aproximação da solução encontrada pelo algoritmo ACM é $((w + 1)w)/(w + \varepsilon)$. Então, para $\varepsilon = 1$ temos o fator $|\Re| - 3$, e na medida que ε tende a zero, o fator tende a $|\Re| - 2$, uma unidade a menos que o fator obtido por Oliveira [35] para as instâncias do PASG. Isso deixa evidente que o fator de aproximação deste algoritmo é atingido assintoticamente. Logo, também não é possível melhorá-lo assintoticamente para as instâncias do PCCM e do PCCM-RE.

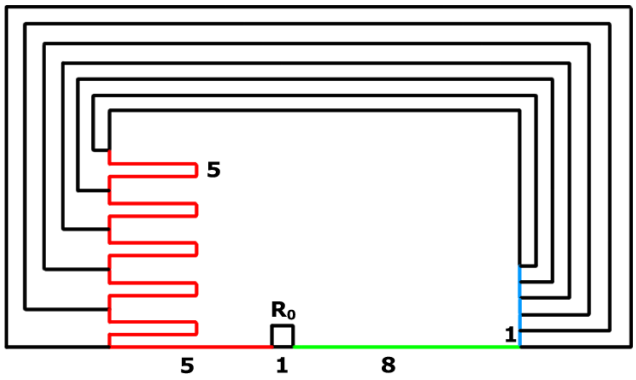


Figura 5.2: Exemplo do formato apresentado na figura anterior com $w = 5$ e $z = 1$.

Capítulo 6

Heurísticas de Melhoria

No capítulo anterior foram apresentados algoritmos que possuem um fator de aproximação para a qualidade da solução gerada por eles. Ou seja, existe uma garantia de quantas vezes o valor de uma solução obtida por algum destes algoritmos será pior que o valor da solução ótima. Neste capítulo são apresentadas três heurísticas que têm por objetivo tomar uma solução factível do PASG e tentar diminuir seu peso. Estas heurísticas não garantem a melhoria da solução em todos os casos, entretanto, na prática, podem conseguir bons resultados.

Durante a construção de uma solução através de um método heurístico ou aproximado são tomadas decisões locais que podem ser revistas posteriormente. Ao analisar as soluções obtidas pelos algoritmos aproximativos, constatamos que elas apresentavam alguns aspectos que poderiam ser melhorados. Isso nos levou a criar as heurísticas que apresentamos nas próximas seções. Mais adiante, na seção 9.2, estudamos experimentalmente qual a melhor forma de combiná-las.

Ao longo deste capítulo vamos utilizar o exemplo, que consiste em uma instância do PCCM convertida em um instância do PASG dada pela figura 6.1, para ilustrar os casos onde é possível melhorar uma solução. Na figura, os pesos das arestas correspondem ao seu comprimento euclidiano. Nela temos uma solução cujas arestas são destacadas nas cores azul, verde, amarelo e vermelha. Observe que esta solução é obtida pelo algoritmo ACM ao se escolher a sala situada no canto inferior esquerdo como grupo R_0 raiz. Além das arestas nesta solução, também destacamos três outras arestas numeradas que vamos usar nos exemplos.

Na discussão que se segue usamos a mesma notação dos capítulo anteriores. Assim, as instâncias do PASG são dadas pela tupla $(G,\Re)$, onde $G = (V,E)$ é um grafo não orientado com pesos associados às arestas definidos pela função $p : E \to \mathbb{R}_+$, e $\Re$ é uma coleção de conjuntos de vértices (grupos) em V. Ao computar árvores ou caminhos de peso mínimo no grafo de uma instância do PASG, estaremos supondo sempre que os pesos

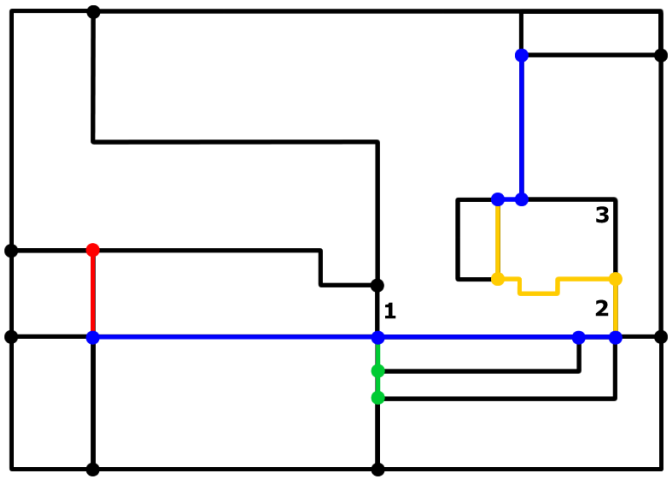


Figura 6.1: Exemplo que ilustra os pontos onde é possível melhorar uma solução obtida pelo algoritmo ACM.

das arestas considerados nos cálculos são aqueles atribuídos pela função peso p definida na instância. Denotaremos por ω a maior quantidade de grupos que um vértice cobre e, durante este capítulo, vamos analisar a complexidade de tempo das heurísticas em função de ω e da quantidade de vértices ($|V|$) e arestas ($|E|$) no grafo da instância.

6.1 Remoção de Folhas Desnecessárias

Dependendo da forma de como é construída uma solução do PASG, podem-se ter caminhos que, se removidos, ainda levem a uma outra solução viável. Por exemplo, observe a solução apresentada na figura 6.1. O caminho destacado em verde está cobrindo grupos que já foram cobertos por vértices azuis. Se este caminho for removido, ainda continuamos com uma solução viável, porém, com peso menor que a anterior.

Com base nesta observação, vamos definir um tipo de caminho que pode ser removido de uma solução do PASG sem que ela deixe de ser viável. Dada uma solução do PASG, dizemos que um s-t caminho composto por arestas desta solução é um *caminho desnecessário* se ele respeitar as seguintes condições: *(i)* deve haver pelo menos uma aresta no caminho, *(ii)* o vértice t possui grau um na solução, *(iii)* todos os vértices no interior do caminho tem grau dois na solução e *(iv)* para cada grupo R que um vértice deste caminho cobre, com exceção do vértice s, existe outro vértice na solução que não está no caminho e cobre R, ou s cobre R. Um caminho desnecessário é maximal se não for possível aumentá-lo sem que ele deixe de respeitar pelo menos uma das restrições anteriores. Um s-t caminho que faça parte de uma solução e não atenda a uma das condições de *(i)* a *(iv)* é dito ser *necessário*.

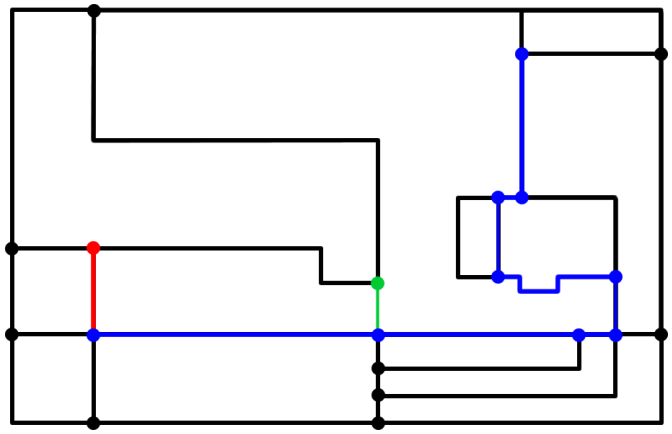


Figura 6.2: Exemplo que mostra o impacto da escolha de qual caminho desnecessário que será removido.

Ao encontrar um caminho desnecessário em uma solução pode-se removê-lo pois, pelas condições acima, a solução permanecerá conexa e ainda cobrirá todos os grupos. Na heurística proposta os caminhos desnecessários removidos serão sempre maximais. Para encontrá-los basta olhar para os vértices folha na solução e proceder da seguinte forma. Seja um caminho composto apenas por uma aresta incidente a um vértice folha. Se ele for desnecessário, aumente-o até torná-lo maximal. Note que esta é uma tarefa simples de se realizar dado que os vértices no interior do caminho podem ter apenas grau dois na solução.

O processo de remoção destes caminhos deve ser feito iterativamente pois, a cada remoção, a solução é alterada, o que pode fazer com que alguns dos demais caminhos desnecessários ainda não removidos deixem de respeitar a restrição *(iv)*. Mais que isso, a ordem na qual os caminhos são removidos pode impactar na melhoria da qualidade da solução. Estas situações são ilustradas pela figura 6.2. Considere que o peso de cada aresta corresponde ao seu comprimento euclidiano. Nesta figura são apresentados dois caminhos desnecessários maximais destacados em vermelho e verde. Quando um deles for removido o outro passa a ser necessário, não podendo mais ser removido da solução. Portanto, pode ser interessante olhar para todos os caminhos desnecessários e remover aquele que leva à melhor solução no momento, ou seja, o caminho de maior peso. No exemplo, seria escolhido o caminho destacado na cor vermelha para ser removido.

Isso nos leva a considerar uma heurística de caminhos desnecessários (HD) com duas versões. A primeira versão chamada de heurística HD *First Fit* é dada pelo algoritmo 3. Nela, os caminhos desnecessários maximais são removidos assim que são encontrados na solução. A segunda versão chamada de heurística HD *Best Fit* é dada pelo algoritmo 4. Nela, primeiramente são encontrados todos os caminhos desnecessários maximais na

solução e, após isso, é escolhido aquele de maior peso para ser removido. Esse processo é repetido até que não exista mais nenhum destes caminhos na solução.

A seguir, vamos verificar a complexidade temporal no pior caso destes dois algoritmos. Observe que, no pior caso, T tem $O(|V|)$ vértices e $O(|E|)$ arestas. Na heurística HD *First Fit*, ao percorrer os vértices folha gasta-se $O(|V|)$ operações. No processo de encontrar um caminho desnecessário, torná-lo maximal, e removê-lo da solução, percorremos no máximo todas as arestas da árvore uma única vez e, ao percorrê-las, verificamos os grupos cobertos por algum de seus extremos, logo, gastamos $O(\omega|E|)$ operações. As duas análises anteriores correspondem aos passos 3-5 do algoritmo. Nos demais passos gasta-se no máximo $(|E|)$ operações. Por isso, a complexidade temporal no pior caso deste algoritmo é $O(\omega|E|)$.

Na análise da heurística HD *Best Fit* consideramos que T não é um caminho, pois esta situação não representa um pior caso. Sendo assim, podemos afirmar que todos os caminhos desnecessários maximais em T são disjuntos nas arestas. Portanto, a cada vez que enumeramos todos eles, percorremos cada aresta apenas uma vez verificando os grupos que seus extremos cobrem, totalizando $O(\omega|E|)$ operações. Como fazemos isso no máximo $O(|V|)$ vezes, gastamos no máximo $O(\omega|V||E|)$ operações. Nos passos 1-4, 6 e 8 não é difícil ver que são gastas no máximo $O(|V||E|)$ operações. Finalmente, conclui-se que a complexidade temporal no pior caso deste algoritmo é $O(\omega|V||E|)$.

6.2 Troca de Folhas de Cobertura Única

Ao observar soluções do PASG, é possível encontrar determinados caminhos que estão presentes na árvore para cobrir apenas um grupo. Em geral, alguns destes caminhos podem ser removidos individualmente da solução de tal forma que a árvore resultante deixe de cobrir apenas um grupo. Desta forma, para que esta árvore volte a ser uma solução novamente, deve-se adicionar a ela um caminho que liga um de seus vértices com algum vértice do grupo que deixou de ser coberto. Se for possível encontrar um caminho com peso menor que o daquele que foi removido, então, será gerada uma nova solução com peso menor que o da solução anterior. Na figura 6.1, ao remover o caminho destacado em vermelho deixa-se de cobrir a sala que fica localizada no canto superior esquerdo. O menor caminho que liga um vértice da árvore resultante a algum vértice da sala que deixou de ser coberta é composto pela aresta número 1. Este caminho tem peso menor do que aquele que foi removido. Portanto, foi obtida uma solução melhor que a anterior. A partir desta idéia propomos a chamada heurística HC.

Inicialmente, vamos caracterizar os caminhos de interesse nesta heurística. Um s-t caminho na solução é dito ser de *cobertura única para um grupo* R, se ele respeita as seguintes condições: *(i)* o caminho possui pelo menos uma aresta, *(ii)* o vértice t tem

Entrada: Uma instância I=($G = (V, E)$,$\Re$) e uma solução T do PASG.
Saída: Uma solução T' para I.

Passo 1: Faça $T' = T$.
Passo 2: Faça V' o conjunto com todos os vértices folha em T'.
Passo 3: Se $V' \neq \emptyset$ retire um vértice v de V'. Caso contrário, vá para o passo 6.
Passo 4: Seja e a aresta em T' que incide em v. Se e for um caminho desnecessário, então construa um caminho desnecessário P maximal em T' a partir dele. Caso contrário, volte ao passo 3.
Passo 5: Remova P de T' e volte ao passo 3.
Passo 6: Retorne T'.

Algoritmo 3: Heurística de remoção de caminhos desnecessários versão *First Fit* (heurística HD *First Fit*).

Entrada: Uma instância I=($G = (V, E)$,$\Re$) e uma solução T do PASG.
Saída: Uma solução T' para I.

Passo 1: Faça $T' = T$.
Passo 2: Faça V' o conjunto com todos os vértices folha em T'.
Passo 3: Faça $P' = \emptyset$ e $p(P') = \infty$.
Passo 4: Se $V' \neq \emptyset$ retire um vértice v de V'. Caso contrário, vá para o passo 7.
Passo 5: Seja e a aresta em T' que incide em v. Se e for um caminho desnecessário, então construa um caminho desnecessário P maximal em T' a partir dele. Caso contrário, volte ao passo 4.
Passo 6: Se $p(P) < p(P')$ faça $P' = P$. Volte ao passo 4.
Passo 7: Se $P' \neq \emptyset$, remova P' de T' e volte ao passo 2. Caso contrário, vá para o próximo passo.
Passo 8: Retorne T'.

Algoritmo 4: Heurística de remoção de caminhos desnecessários versão *Best Fit* (heurística HD *Best Fit*).

grau um na solução, *(iii)* todos os vértices no interior do caminho possuem grau dois na solução, *(iv)* para cada grupo $R' \neq R$ que um vértice deste caminho cobre, com exceção do vértice s, existe outro vértice na solução que não está no caminho e cobre R', ou s cobre R' e *(v)* R só é coberto por vértices da solução que estão no caminho. Um caminho de cobertura única é dito ser maximal se não for possível aumentá-lo sem que ele deixe de respeitar pelo menos uma das condições anteriores.

É possível notar uma certa semelhança entre os caminhos desnecessários e os caminhos de cobertura única. De fato, se desconsiderarmos a condição *(v)*, por definição um caminho desnecessário seria um caminho de cobertura única para qualquer grupo. Entretanto, estes caminhos constituem diferentes estruturas em uma solução. Os caminhos desnecessários são estruturas que não contribuem para a cobertura dos grupos e, por isso, podem ser removidos. Já os caminhos de cobertura única têm a função de cobrir apenas um grupo. Portanto, podemos tentar alterá-los visando a redução do peso da solução. Por este motivo, optamos por lidar com estes diferentes tipos de caminhos em heurísticas separadas.

Assim como na heurística anterior, esta busca pelos caminhos de cobertura única também é feita partindo dos vértices folha. Desta forma, dada uma aresta que incide em algum vértice folha na solução, se ao removê-la deixamos de cobrir apenas um grupo R, então esta aresta é um caminho de cobertura única para R. Com este caminho em mãos, aumente-o até que se torne maximal. Ao remover este caminho da solução, agora é preciso encontrar um caminho que parte de um dos vértices na árvore resultante e termina em dos vértices do grupo R que deixou de ser coberto. Como estamos visando diminuir o peso da solução, este caminho deve ser mínimo. Se o caminho mínimo encontrado for menor que o caminho removido, incluímos ele na solução, caso contrário, incluímos de volta na solução o caminho removido.

Para encontrar um caminho mínimo que liga um vértice na árvore a um vértice de um grupo R não coberto faça o seguinte. Seja G o grafo da instância do PASG. Crie um grafo G' cópia de G. Inclua dois vértices s e t em G'. Para cada vértice v na árvore acrescente em G' a aresta (s,v) com peso zero, e para cada vértice u em R acrescente em G' a aresta (u,t) também com peso zero. Encontre um s-t caminho mínimo em G'. Não é difícil perceber que ao remover os vértices s e t do caminho encontrado será obtido um caminho mínimo em G que liga a árvore ao grupo R.

Este processo de substituição de caminhos nos leva a uma heurística de caminhos de cobertura única (HC). Nela também deve-se considerar a ordem na qual são alterados os caminhos de cobertura única, pois a solução que se tem no momento é alterada. Para ilustrar tal situação veja a figura 6.3. Considere que os pesos das arestas correspondem ao seu comprimento euclidiano. Na figura 6.3 (a) temos uma solução com dois caminhos de cobertura única maximais, destacados nas cores verde e vermelha. Na figura 6.3 (b) temos a solução que resulta se primeiro for alterado o caminho de cor vermelha, o que faz

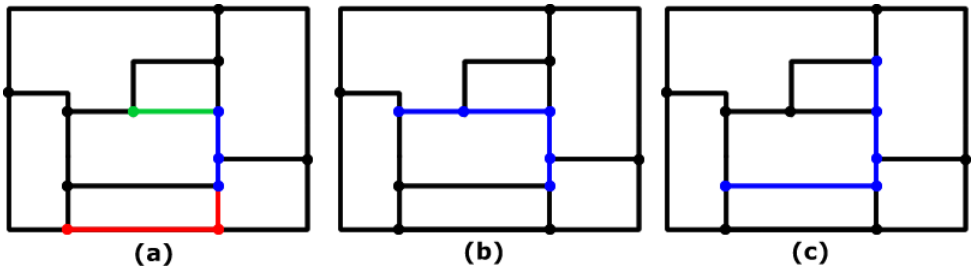


Figura 6.3: Exemplo que mostra o impacto da ordem na qual alteramos os caminhos de cobertura única em um solução.

com que o caminho na cor verde deixe de ser um caminho de cobertura única. Na figura 6.3 (c) primeiro é alterado o caminho de cor verde e depois o caminho de cor vermelha, resultando em uma solução de peso maior que a solução obtida pela ordem inversa.

Por isso, nesta heurística também consideramos as versões *First Fit* e *Best Fit*, dadas, respectivamente, pelos algoritmos 5 e 6. Na versão *First Fit*, os caminhos de cobertura única maximais são encontrados percorrendo os vértices folha em uma ordem cíclica arbitrária. Ao encontrar cada caminho, é verificado se a alteração deste caminho melhora a solução atual, caso melhore a alteração é realizada e o processo continua. A heurística termina quando todos os vértices são percorridos e nenhuma alteração na solução atual for feita. Na versão *Best Fit*, são encontrados todos os caminhos de cobertura única maximais sem alterar a solução atual. Determina-se qual destes caminhos que, ao ser alterado, leva à maior redução do peso da solução atual. Se houver tal caminho, ele é alterado na solução atual. Este processo é repetido até que não se encontrem mais caminhos que melhoram a solução atual.

Nestas duas heurísticas, não é difícil perceber que os procedimentos que dominam a complexidade de tempo são a computação do caminho de cobertura única maximal e a busca pelo caminho que reconecta o grupo não coberto à árvore (passos 5 e 6 nos algoritmos). No procedimento de encontrar o caminho de cobertura única gasta-se $O(\omega|E|)$ operações. No outro procedimento podemos empregar um algoritmo de caminhos mínimos que tem complexidade de tempo no pior caso igual a $O((|V| + |E|) \log |V|)$ (consulte Cormen et al. [12], capítulo 24). Assim, vamos analisar a complexidade de tempo destas heurísticas em função da quantidade destas operações. Também, ambos os algoritmos dependem da quantidade de vezes que atualizamos a solução corrente, denotada por Δ. Esta quantidade pode ser estimada, no pior caso, pelo maior valor de uma solução da instância. Como parece ser complexo obter boas estimativas para este parâmetro, e já que ele pode variar em diferentes situações, vamos dar a complexidade de tempo em função dele.

Pelas condições *(ii)* e *(iii)*, sabemos que existem no máximo $|V|$ caminhos de cobertura única maximais diferentes. Na heurística HC *First Fit*, o pior caso seria percorrer todos

Entrada: Uma instância I=($G = (V, E)$,$\Re$) e uma solução T do PASG.
Saída: Uma solução T' para I.

Passo 1: Faça $T' = T$.
Passo 2: Faça $F = 0$.
Passo 3: Faça V' ser o conjunto com todos os vértices folha em T'.
Passo 4: Se $V' \neq \emptyset$ retire um vértice v de V'. Caso contrário, vá para o passo 8.
Passo 5: Seja e a aresta em T' incidente em v. Se e for um caminho de cobertura única. então construa um caminho de cobertura única P maximal em T' a partir dela. Caso contrário, volte ao passo 4.
Passo 6: Remova P de T' e encontre um caminho mínimo P^* em G que liga um vértice de T' a um vértice do grupo R que deixou de ser coberto.
Passo 7: Se $p(P^*) < p(P)$ inclua P^* em T' e faça $F = 1$, caso contrário, apenas inclua P de volta. Volte ao passo 4.
Passo 8: Se F for igual a 1 volte ao passo 2.
Passo 9: Retorne T'.

Algoritmo 5: Heurística de alteração de caminhos de cobertura única versão *First Fit* (heurística HC *First Fit*).

Entrada: Uma instância I=($G = (V, E)$,$\Re$) e uma solução T do PASG.
Saída: Uma solução T' para I.

Passo 1: Faça $T' = T$.
Passo 2: Faça V' ser o conjunto com todos os vértices folha em T'.
Passo 3: Faça $P' = \hat{P} = \emptyset$ e $p(P') = p(\hat{P}) = \infty$.
Passo 4: Se $V' \neq \emptyset$ retire um vértice v de V'. Caso contrário, vá para o passo 8.
Passo 5: Seja e a aresta em T' incidente em v. Se e for um caminho de cobertura única, então construa um caminho de cobertura única P maximal em T' a partir dela. Caso contrário, volte ao passo 4.
Passo 6: Remova P de T' e encontre um caminho mínimo P^* em G que liga um vértice de T' a um vértice do grupo R que deixou de ser coberto.
Passo 7: Se $p(P)$ - $p(P^*) > p(P')$ - $p(\hat{P})$ faça $P' = P$ e $\hat{P} = P^*$. Inclua P de volta em T' e volte ao passo 4.
Passo 8: Se $P' \neq \emptyset$, então remova P' de T', inclua $\hat{P}$ e volte ao passo 2. Caso contrário, prossiga para o próximo passo.
Passo 9: Retorne T'.

Algoritmo 6: Heurística de alteração de caminhos de cobertura única versão *Best Fit* (heurística HC *Best Fit*).

os caminhos de cobertura única maximais de uma mesma solução, e alterar apenas o último. O mesmo ocorre na heurística HC *Best Fit*, onde enumeramos todos os caminhos e escolhemos a melhor alteração na solução. Portanto, a complexidade de tempo no pior caso destes algoritmos é $O(\Delta|V|(\ \omega|E| + (|V| + |E|)\log|V|\ ))$.

6.3 Reconexão da Árvore

A terceira heurística proposta, denominada heurística HR, é uma extensão da busca local denominada *key-path-based local search* proposta por Verhoeven et al. [45] para o problema da árvore de Steiner. Ela se baseia na propriedade de que é possível dividir uma solução do PASG em duas árvores disjuntas nos vértices onde pelo menos um vértice de cada grupo está em alguma delas, todavia, ao realizar esta operação deixa-se de ter uma solução pois ela não é conexa. Por exemplo, ao retirar uma aresta qualquer da árvore, nós a dividimos em duas subárvores sem remover nenhum vértice. Portanto, nenhum grupo deixa de ser coberto.

Indo além, ao invés de retirar apenas uma aresta, é possível retirar um caminho da solução e ainda obter duas subárvores que cobrem todos grupos. Mais formalmente, um caminho é dito ser *de ligação* se ele satisfaz as seguintes restrições: *(i)* o caminho possui pelo menos uma aresta, *(ii)* todos os vértices no interior do caminho devem ter grau dois na solução e *(iii)* para cada grupo R coberto por um vértice interno neste caminho, deve existir outro vértice na solução que não está no interior do caminho e também cobre R. Um caminho de ligação é dito ser maximal se ele não puder ser aumentado sem deixar de respeitar pelo menos uma das restrições anteriores.

Pelas restrições definidas, ao removermos de uma solução um caminho de ligação maximal, nós a dividimos em duas árvores que juntas cobrem todos os grupos. Portanto, para obter novamente uma solução é preciso apenas religar estas duas árvores. A melhor forma de fazer isso é procurar por um caminho mínimo que parte de um vértice de uma árvore e termina na outra. Se este caminho tiver peso menor que o caminho que foi removido, então será obtida uma nova solução com peso menor que o peso da solução que tínhamos antes. Para entender esse procedimento, observe a solução dada pela figura 6.1, considerando que as arestas verdes não fazem parte da solução. Retirando desta solução o caminho destacado em amarelo, tem-se duas árvores que cobrem todos os grupos. O menor caminho que religa as duas árvores é composto pelas arestas 2 e 3. Este caminho pesa menos que o caminho em amarelo. Logo, ao reconectar as árvores por este caminho através da inclusão das arestas 2 e 3 chega-se a uma solução melhor.

Encontrar caminhos de ligação em uma solução é uma tarefa simples. Isto porque, por definição, qualquer aresta é um caminho de ligação. Se este caminho não for maximal, é possível aumentá-lo acrescentando-se a ele alguma das arestas que incidem nos vértices

extremos do caminho, desde que sejam respeitadas as restrições impostas pela definição. Ao acrescentar as arestas para tornar o caminho maximal, se houver mais de uma opção, escolhemos a aresta de maior peso. Como vamos remover este caminho da solução, a escolha é feita desta forma visando maximizar o peso do caminho. Este processo de inclusão de arestas é repetido até que o caminho se torne maximal.

Seja G o grafo da instância do PASG. Encontrar um caminho mínimo que liga duas árvores T' e T'' em G pode ser feito de forma semelhante à que foi feita na heurística anterior. Crie um grafo G' cópia de G. Inclua dois vértices s e t em G'. Para cada vértice v em T' acrescente em G' a aresta (s,v) com peso zero, e para cada vértice u em T'' acrescente em G' a aresta (u,t) também com peso zero. Encontre um s-t caminho mínimo em G'. Ao remover os vértices s e t do caminho encontrado será obtido um caminho mínimo em G que liga as duas árvores.

Com base nos caminhos de ligação propomos a heurística de reconexão da árvore (HR). A qualidade da solução obtida por esta heurística também depende da forma escolhida para atualizar a solução corrente. Sendo assim, também consideramos para esta heurística as versões *First Fit* e *Best Fit* dadas pelos respectivos algoritmos 7 e 8. Na versão *First Fit*, as arestas na solução são percorridas em uma ordem cíclica arbitrária. É construído um caminho de ligação maximal a partir de cada aresta da forma descrita acima. Encontra-se um caminho mínimo que liga as duas árvores obtidas quando o caminho de ligação que se tem no momento é removido. Se o peso do caminho de ligação for maior que o peso do caminho mínimo, o substituímos na solução atual. A heurística termina quando todas as arestas são percorridas e a solução não é alterada. Na versão *Best Fit*, todas as arestas são percorridas construindo os caminhos de ligação maximais. Na medida em que os caminhos de ligação maximais são encontrados, encontra-se também um caminho mínimo que liga as duas árvores obtidas pela remoção de cada caminho de ligação, sem alterar a solução atual. Ao mesmo tempo, é guardado o caminho de ligação que substituído pelo caminho mínimo correspondente leva à maior diminuição do peso da solução atual. Após isso, se houver tal caminho, ele é substituído na solução atual e o processo é repetido, caso contrário, a heurística termina sua execução.

A análise da complexidade de tempo no pior caso destes algoritmos é feita de forma análoga aquela feita na seção anterior. O procedimento que constrói o caminho de ligação maximal e o procedimento que liga as duas subárvores por um caminho mínimo (passo 5 e 6 dos algoritmos) são os que dominam a complexidade de tempo das heurísticas. No primeiro, gasta-se $O(\omega|E|)$ operações. No segundo, empregamos um algoritmo de caminhos mínimos cuja complexidade de tempo no pior caso, como visto antes, é $O((|V| + |E|) \log |V|)$. Assim como na seção anterior, vamos dar a complexidade destas heurísticas em função de Δ que corresponde a quantidade de vezes que alteramos a solução corrente.

Na heurística HR *First Fit*, o pior caso corresponde a percorrer $O(|E|)$ caminhos de

Entrada: Uma instância I=(G,$\Re$) e uma solução T do PASG.
Saída: Uma solução T' para I.

Passo 1: Faça $T' = T$.
Passo 2: Faça $F = 0$.
Passo 3: Faça E' ser o conjunto com todas as arestas em T'.
Passo 4: Se $E' \neq \emptyset$ retire uma aresta e de E'. Caso contrário, vá para o passo 8.
Passo 5: Construa um caminho de ligação P maximal em T' a partir de e.
Passo 6: Encontre um caminho mínimo P^* em G que liga as duas árvores obtidas ao remover P de T'.
Passo 7: Se $p(P^*) < p(P)$ remova P de T', inclua P^* e faça $F = 1$. Volte ao passo 4.
Passo 8: Se F for igual a 1, volte ao passo 2.
Passo 9: Retorne T'.

Algoritmo 7: Heurística de reconexão da árvore versão *First Fit* (heurística HR *First Fit*).

Entrada: Uma instância I=(G,$\Re$) e uma solução T do PASG.
Saída: Uma solução T' para I.

Passo 1: Faça $T' = T$.
Passo 2: Faça E' ser o conjunto com todas as arestas em T'.
Passo 3: Faça $P' = \hat{P} = \emptyset$ e $p(P') = p(\hat{P}) = \infty$..
Passo 4: Se $E' \neq \emptyset$ retire uma aresta e de E'. Caso contrário, vá para o passo 8.
Passo 5: Construa um caminho de ligação P maximal em T' a partir de e.
Passo 6: Encontre um caminho mínimo P^* em G que liga as duas árvores obtidas ao remover P de T'.
Passo 7: Se $p(P) - p(P^*) > p(P') - p(\hat{P})$, faça $P' = P$ e $\hat{P} = P^*$. Volte ao passo 4.
Passo 8: Se $P' \neq \emptyset$, remova P' de T', inclua $\hat{P}$ e volte ao passo 2. Caso contrário, vá para o próximo passo.
Passo 9: Retorne T'.

Algoritmo 8: Heurística de reconexão da árvore versão *Best Fit* (heurística HR *Best Fit*).

ligação maximal de uma mesma solução, e efetuar a alteração na solução ao encontrar o último deles. Na versão *Best Fit*, ocorre a mesma situação, pois são calculados $O(|E|)$ caminhos para escolher a melhor forma de alterar a solução. Portanto, em ambas as heurísticas, a complexidade de tempo no pior caso é $O(\Delta|E|(\ \omega|E| + (|V| + |E|)\log|V|\ ))$.

Capítulo 7

Uma Heurística GRASP com *Evolutionary Path Relinking*

Neste capítulo vamos apresentar a heurística proposta neste trabaho, baseada em uma combinação de um GRASP com *evolutionary path relinking*. Na seção seguinte introduzimos a idéia geral do método, e em seguida são apresentados os detalhes da heurística desenvolvida.

7.1 Metaheurística GRASP com *Evolutionary Path Relinking*

Uma metaheurística consiste em uma heurística mais geral, independente do problema em que será aplicada, que guia a aplicação de outras heurísticas específicas para o problema que se está abordando. Em linhas gerais, a metaheurística GRASP segue a filosofia de construir uma solução através de um método guloso aleatorizado, e em seguida é aplicada uma busca local na solução obtida. Esta metaheurística foi proposta em Feo e Resende [18], e desde então vem sendo aplicada com sucesso em diversos problemas da literatura. Várias referências sobre esta metaheurística podem ser encontradas nos trabalhos de Festa e Resende [21], Resende e Ribeiro [39], Resende [37] e Festa e Resende [20].

A versão mais básica do GRASP constitui um método iterativo. Este método executa um número pré-fixado de iterações. Cada iteração é divida em duas fases que são, repectivamente, a fase de construção e a fase de busca local. Primeiramente é executada a fase de construção onde se emprega um algoritmo *semi-guloso* para construir uma solução do problema em questão. Um algoritmo semi-guloso utiliza a mesma idéia de um algoritmo guloso, entretanto, a cada iteração, ao invés de escolher sempre o melhor elemento para adicionar na solução em construção, a escolha agora passa a ser feita

entre os melhores elementos. Desta forma, a cada iteração do algoritmo semi-guloso é feita uma lista restrita de candidatos (LRC), onde são armazenados os elementos que são melhores ou iguais a todos os elementos que podem ser escolhidos. Só então um elemento desta lista é selecionado aleatoriamente. É utilizado um parâmetro $\alpha \in [0, 1]$ para determinar quais elementos podem entrar na lista. Considere que estamos tratando um problema de minimização. Seja C o conjunto de elementos que podem ser escolhidos em um dado momento, f a função que determina a qualidade dos elementos em C, $f_{min} = \min\{\ f(e) \mid e \in C\ \}$ e $f_{max} = \max\{\ f(e) \mid e \in C\ \}$. Um elemento e está na lista somente se $f(e) \leq f_{min} + \alpha(f_{max} - f_{min})$. Note que se α for igual a 0 vamos ter um algoritmo guloso, e se for igual a 1 teremos um algoritmo totalmente aleatório. Então, este parâmetro é o que regula o quão guloso ou aleatório será o algoritmo semi-guloso.

Além da aplicação do algoritmo semi-guloso, existem na literatura diversas outras formas de se construir soluções no GRASP. Em Resende e Ribeiro [38] são apresentadas outras seis diferentes fases de construção. Dentre elas, vamos utilizar uma conhecida como construção aleatória-gulosa (em inglês, *random plus greedy*). Nesta construção, primeiramente parte da solução é construída de forma totalmente aleatória e, em seguida, o restante da solução é construído de forma gulosa.

As fases de construção do GRASP tem-se mostrado capazes de obter soluções de boa qualidade. Apesar disso, em geral, não há garantia de que a solução obtida nesta fase seja um mínimo local. Por isso, em seguida, executa-se a fase de melhoria, onde é empregada alguma busca local específica para o problema que se está tratando. Com isso, ao término de cada iteração do GRASP é obtido um mínimo local e, no fim da execução do algoritmo, uma solução cujo custo corresponde ao melhor mínimo local encontrado é retornada.

A partir do GRASP básico surgiram diversas variantes deste método. Neste trabalho, vamos adotar uma variante híbrida, combinando o GRASP com um método chamado *evolutionary path relinking*, muito utilizado em diversos trabalhos na literatura, que comprovam a sua capacidade de obtenção de bons resultados. Alguns dos trabalhos que fazem uso deste método são Andrade e Resende [1], Villegas et al. [46] e Usberti et al. [44].

Antes de introduzir o *evolutionary path relinking* precisamos conhecer o método *path relinking*. O *path relinking* é um método que toma duas soluções, e traça no espaço de busca um caminho, composto por soluções intermediárias, entre estas duas soluções. Isso é feito da seguinte forma. Primeiramente, uma das soluções é escolhida para ser a solução *origem* e a outra passa a ser a solução *destino*. Então, executa-se um processo iterativo que começa com a solução origem, e a modifica até que ela se torne a solução destino. Para isso, primeiramente determinamos a diferença simétrica entre estas duas soluções, ou seja, encontramos os elementos que estão em uma solução mas não estão na outra. Dentre estes elementos, escolhemos entre remover um elemento que está na solução que estamos modificando, ou acrescentar um elemento que não está nela. Geralmente, levando-se em

conta que estamos tratando um problema de minimização, é escolhida a alteração que leva à maior diminuição do valor da função objetivo ou, caso isso não seja possível, aquela que leva ao menor acréscimo do valor. Este processo é repetido até que a solução corrente se torne igual à solução destino. A cada vez que alteramos a solução corrente, não há garantia de que esta seja um mínimo local. Sendo assim, aplicamos uma busca local em uma cópia dela para não alterar a solução original. Ao aplicar o *path relinking* em um par de soluções, temos a esperança de encontrar soluções melhores que estas duas. Caso isso ocorra, a melhor solução encontrada neste processo é retornada.

O *evolutionary path relinking* é um método que evolui uma *população* de soluções através da aplicação do *path relinking*. A cada iteração deste método, o *path relinking* é aplicado nos pares de soluções na população atual, gerando novas soluções. No fim, temos as soluções da população e as geradas. Como a população normalmente tem tamanho fixo, são escohidas dentre estas soluções as melhores, que apresentam um certo grau de dissimilaridade, para compor a população da próxima iteração. Este critério de dissimilaridade é adotado para que o *path relinking* possa se beneficiar de uma boa diversidade da população. No final de cada iteração, se a melhor solução obtida tiver peso menor que a melhor solução da população anterior, o método vai para a próxima iteração. Caso contrário, o método termina sua execução e retorna a melhor solução encontrada.

O GRASP combinado com o *evolutionary path relinking* trabalha da seguinte forma. Durante a execução do GRASP, é criado um conjunto de soluções elitistas, chamado de *pool*, selecionadas dentre os mínimos locais encontrados ao longo das iterações do algoritmo. Nesta seleção também leva-se em conta o grau de dissimilaridade entre as soluções, assim como no *evolutionary path relinking*. Após a execução do GRASP, o *evolutionary path relinking* é executado tomando o *pool* como população inicial. A melhor solução encontrada neste processo é retornada.

7.2 A Heurística Desenvolvida

No GRASP desenvolvido, optamos por utilizar os algoritmos aproximativos AAGM e ACM na fase de construção. Para isso, modificamos estes algoritmos de tal forma a torná-los aleatórios. A cada iteração do GRASP, é feito um sorteio, com chances iguais para ambos algoritmos, determinando qual deles será executado.

A modificação do algoritmo AAGM foi feita da seguinte forma. Neste algoritmo, nos passos 1 e 2 (página 29), decide-se quais vértices irão cobrir os grupos através de informações fornecidas por um modelo matemático. Modificamos estes passos para tornar esta escolha aleatória, onde para cada grupo é feito um sorteio aleatório com distribuição uniforme para determinar qual vértice irá cobri-lo. Os demais passos do algoritmo permancem inalterados. Esta modificação faz com que este algoritmo seja mais rápido, pois

resolver o modelo matemático tomaria muito tempo, sendo inviável a sua utilização na heurística. Note que esta é uma fase de construção aleatória-gulosa, onde primeiro selecionamos os vértices de forma aleatória e depois as arestas são escolhidas seguindo um critério guloso.

No algoritmo ACM foram feitas duas modificações. O passo 6 (página 30) deste algoritmo procura pelo menor caminho que liga um vértice da solução em construção a um vértice de um grupo não coberto, e o acrescenta na solução. A primeira modificação proposta consiste em permitir que o algoritmo escolha, de forma aleatória e uniformemente distribuída, um dentre cinco caminhos que possuem peso menor ou igual que os demais, ao invés de selecionar sempre o menor. Isto torna este algoritmo semi-guloso. Todavia, não utilizamos o parâmetro α para determinar quem está na lista restrita de candidatos. Simplesmente atribuímos um tamanho fixo para ela. Outro aspecto do algoritmo ACM é a criação de diversas soluções, onde cada uma é construída a partir de um vértice inicial diferente. Isto pode deixar o algoritmo lento, causando desperdício de tempo na heurística. Portanto, decidimos que o algoritmo efetuará a construção de apenas uma solução. O vértice escolhido para ser o vértice inicial será aquele que esteve presente mais vezes nos mínimos locais gerados pelo GRASP. Caso haja mais de um vértice nesta situação, é feito um sorteio com chances iguais para escolher um dentre eles.

A fase de busca local do GRASP consiste na aplicação de uma combinação das heurísticas propostas no capítulo anterior. Na seção 9.2.1, é feito um estudo experimental visando encontrar a melhor forma de efetuar tal combinação. Como visto anteriormente, estas heurísticas trabalham percorrendo vértices ou arestas da solução. No intuito de dotá-las de uma componente aleatória, decidimos que a cada execução da heurística, ela irá realizar esse percurso, nos vértices ou arestas, em uma ordem definida de forma aleatória. Além disso, visando evitar o despedício de tempo nesta fase, a busca local é aplicada somente às soluções que têm um valor que é no máximo 20% pior que o valor da melhor solução encontrada até o momento. Esta decisão foi tomada com base nos resultados apresentados na seção 9.2.1, que mostram que o percentual de melhora das buscas locais no valor de uma solução é menor que 20% na média.

Como vamos ter uma fase de aplicação de um *evolutionary path relinking* após a execução do GRASP, é preciso manter o *pool* com as melhores soluções encontradas pelo GRASP. Determinamos que o *pool* terá no máximo 6 soluções elitistas. Esta restrição de tamanho foi imposta para não tornar a fase do *evolutionary path relinking* lenta, já que vamos aplicar este método em cada par de soluções na população, sendo que a busca local será aplicada nas soluções intermediárias encontradas durante este processo. Uma solução pode ser inserida no *pool* somente se ela for melhor que a pior solução contida nele. Além disso, também é preciso verificar um critério de similaridade. Sejam E_1 e E_2 os conjuntos das arestas pertecentes a duas soluções que queremos comparar. Dizemos

que estas soluções são similares se a restrição da equação abaixo for atendida.

$$\frac{|E_1 \cap E_2|}{\min\{|E_1|, |E_2|\}} > 0.95 \tag{7.1}$$

Uma solução será inserida no *pool* se esta for melhor que todas as soluções elitistas similares a ela e, quando isso ocorre, todas as soluções similares são removidas do *pool*. Caso o *pool* esteja cheio e não haja nenhuma solução similar nele, a solução será inserida se possuir pesor menor que a pior solução no *pool*. Quando isto ocorre, a pior solução é removida para que o *pool* não aumente de tamanho.

Para realizar o *path relinking* entre duas soluções, adotamos dois diferentes métodos. Primeiramente vamos discutir como estes métodos funcionam, e logo após apresentamos como eles vão atuar em conjunto no *evolutionary path relinking*.

O primeiro método, chamado de PRA (*path relinking* pelas arestas), é baseado na idéia de inserção e remoção de arestas, e funciona da seguinte forma. Dadas duas soluções, a solução origem é aquela que tiver menor peso entre as duas. A cada iteração deste método, determina-se a diferença simétrica das arestas nas soluções. Após isso, como estamos interessados em melhorar a solução corrente, primeiramente verificamos se é possível remover alguma aresta nesta solução que está na diferença simétrica. Uma aresta na solução corrente pode ser removida quando ela estiver ligada a um vértice folha, onde os grupos cobertos por este vértice estão cobertos por outros vértices na árvore. Caso não exista nenhuma aresta que possa ser removida, procuramos pela aresta de menor peso na diferença simétrica que pode ser inserida na solução corrente. Uma aresta pode ser inserida quando ela é adjacente a algum vértice na solução. Ao inserir uma aresta na solução, é possível que seja criado um ciclo. Se isto ocorrer, a aresta de maior peso deste ciclo, contida na diferença simétrica, será removida da solução. Desta forma as arestas vão sendo inseridas e removidas até que a solução corrente se torne igual a solução destino. Note que se as soluções não possuírem um vértice em comum, é impossível que a solução origem se torne igual a solução destino utilizando este processo. Quando temos duas soluções onde isso ocorre, não aplicamos este método.

O outro *path relinking* utilizado, chamado de PRI (*path relinking* intermediário), foi inspirado no método proposto em Ribeiro et al. [40], que também foi combinado com um GRASP aplicado ao problema da árvore de Steiner. Os autores desta referência consideraram este procedimento de intensificação como um *path relinking*. Contudo, o procedimento não gera várias soluções intermediárias, produzindo apenas uma solução que tende a herdar características de ambas as soluções. Para isso, ao selecionar as duas soluções para aplicar o procedimento, primeiramente é feita uma alteração dos pesos do grafo associado à instância da seguinte forma. Para cada aresta que não está em nenhuma das soluções, o seu peso é multiplicado por 10^6. Cada aresta que está em apenas uma das

soluções tem seu peso multiplicado por um valor sorteado aleatoriamente entre 50 e 100. As arestas em comum entre as soluções permanecem com seu peso inalterado. Finalmente, o algoritmo aproximativo ACM é aplicado considerando este grafo alterado, obtendo uma solução intermediária. Contudo, neste passo o algoritmo ACM também é modificado para evitar a construção de várias soluções a partir de vértices iniciais diferentes, assim como foi feito na fase de construção do GRASP. Mas, neste caso, diferentemente do GRASP, o algoritmo continua selecionando sempre o melhor caminho. O vértice escolhido como inicial neste algoritmo é um que está nas duas soluções e, esteve presente mais vezes nos mínimos locais gerados pelo GRASP e nas soluções obtidas pelo PRI. Se não houver um vértice em comum entre as duas soluções, fazemos da mesma forma, mas neste caso, escolhemos um vértice que está em apenas uma das soluções. Quando temos mais de uma opção para escolha é feito um sorteio aleatório com chances iguais para todas.

A fase do *evolutionary path relinking* é executada da seguinte forma. Como temos dois *path relinking* diferentes, adotamos a estratégia de executar o *evolutionary path relinking* duas vezes. Na primeira execução, a evolução é feita utilizando o PRA, e na segunda, separadamente, a evolução é realizada utilizando o PRI. A população inicial na segunda evolução é igual ao *pool* retornado pelo GRASP, e distinta da população evoluída no passo anterior. Antes de começar o segundo processo de evolução, tentamos inserir na população inicial a melhor solução encontrada pela evolução anterior. Tanto esta inserção, quanto aquelas realizadas durante o processo de evolução da população, são feitas segundo os critérios adotados pelo GRASP nas inserções das soluções no *pool*. Como visto antes, o processo de evolução de cada população é feito através de gerações. Desta forma, a cada geração, temos uma população que corresponde à geração atual, e criamos uma cópia dela para ser a população da próxima geração. Aplicamos o método de *path relinking* para cada par de soluções na população da geração atual. Na medida em que as soluções intermediárias são encontradas, aplicamos a busca local nelas, e tentamos inseri-las na população da próxima geração. A busca local que é aplicada nas soluções intermediárias também é uma combinação das heurísticas de melhoria determinada experimentalmente na seção 9.2.1. Ao terminar a execução do *path relinking* nos pares de soluções, a população da próxima geração se torna a população da geração atual, e o processo se repete. O processo termina quando a melhor solução da próxima geração não for melhor que a melhor solução da geração atual.

Finalmente, resumindo a heurística discutida nesta seção, apresentamos seus passos principais no algoritmo 9.

Entrada: Uma instância I do PASG.
Saída: Uma solução T para I.

Passo 1: Faça $n = 1$ e seja $P = \emptyset$ o *pool* de melhores soluções do GRASP.
Passo 2: Faça um sorteio aleatório escolhendo entre a versão aleatória do algoritmo AAGM ou ACM. Aplique o algoritmo escolhido e construa uma solução T para I.
Passo 3: Execute a busca local em T.
Passo 4: Verifique se é possível inserir T em P e, caso seja, efetue a inserção.
Passo 5: Se $n < IterMax$ faça $n = n + 1$ e volte ao passo 2. Caso contrário, prossiga para o próximo passo.
Passo 6: Faça $P' = P$.
Passo 7: Faça a evolução de P utilizando o PRA.
Passo 8: Seja T' a melhor solução encontrada no passo anterior. Tente inserí-la em P', da mesma forma como deve ser feito no passo 4.
Passo 9: Faça a evolução de P' utilizando o PRI.
Passo 10: Retorne a melhor solução em P'.

Algoritmo 9: Heurística GRASP com *evolutionary path relinking*.

Capítulo 8

Algoritmo Exato

Nesta seção será apresentado um método exato destinado à resolução do PASG, portanto ele também é capaz de resolver o PCCM. Em linhas gerais, este método resolve um problema de programação linear inteira através de um método *branch-and-cut*. O desenvolvimento do método *branch-and-cut* foi feito utilizando a biblioteca *Callable Library* do ILOG CPLEX 12.1. Inicialmente é feita uma breve introdução na qual descreve-se como um método *branch-and-cut* resolve um problema de programação linear inteira. Em seguida são descritos os detalhes do método desenvolvido para o problema em questão.

8.1 Funcionamento de um Algoritmo *Branch-and-Cut* Geral

O *branch-and-cut* consiste de um método *branch-and-bound*[1] modificado para adicionar cortes durante sua execução. Desta forma, a seguir vamos introduzir o *branch-and-bound* e, posteriormente, explicamos como funciona o *branch-and-cut*.

O *branch-and-bound* é um método que utiliza o paradigma de divisão e conquista para enumerar implicitamente as soluções no espaço de busca de um problema, visando encontrar a solução ótima contida nele. Sabe-se que enumerar todo o espaço de busca de determinados problemas é uma tarefa que nem sempre pode ser realizada em um tempo computacional aceitável. Portanto, este método faz uso de duas estimativas que permitem descartar a enumeração de porções do espaço de busca. As estimativas usadas por ele são chamadas de limitantes primais e duais. A definição dessas estimativas varia se o problema em questão é um problema de minimização ou maximização. Como o foco

[1] O *branch-and-bound* é um método um tanto quanto genérico, podendo ser aplicado a diversos tipos de problemas. Entretanto, neste capítulo vamos sempre nos referir a versão deste método específica para problemas de programação linear inteira.

deste trabalho são problemas de minimização, as definições apresentadas aqui estarão de acordo com esse tipo de problema.

Formalizando, seja P um problema de programação linear inteira definido genericamente pelas equações (8.1)-(8.3). Denotamos por z^* o valor ótimo de P.

$$\text{Minimizar} \quad z = cx \tag{8.1}$$

$$\text{Sujeito a:}$$

$$Ax = b, \tag{8.2}$$

$$x \in \mathbb{Z}^n. \tag{8.3}$$

Considere a versão relaxada de P como sendo o problema obtido ao se relaxar a restrição de integralidade, dada pela equação (8.3), substituindo-a por $x \in \mathbb{R}^n$. Note que nesta relaxação estão incluídas as soluções de P, mais as soluções que satifazem o sistema com x sendo fracionário. Por esta razão, fica claro que o ótimo do problema relaxado é sempre menor ou igual a z^*. Logo, este é um limitante dual de P. Em contrapartida, o valor de qualquer solução de P é maior ou igual a z^*, portanto, nos fornece um limitante primal. Note que quanto menor (maior) for o valor do limitante primal (dual) melhor ele será, pois assim ele estará mais próximo do valor ótimo z^*. Observe também que uma solução com valor igual ao valor de um limitante dual é uma solução ótima pois, pela definição deste limitante, não pode existir outra solução com valor menor que esta.

O *branch-and-bound* funciona da seguinte forma. Ao se tomar o modelo matemático de P, é calculado o valor do limitante dual, denotado por $\underline{z}$, através da resolução do problema relaxado de P. Note que o problema relaxado é um problema de programação linear, por isso, pode ser resolvido em tempo polinomial. Também computa-se um limitante primal, denotado por $\overline{z}$. Normalmente, para isto emprega-se um método heurístico que gera uma solução para P. Observe ainda que a solução do problema relaxado pode ter todos os valores inteiros, neste caso fornecendo uma solução para P. O leitor interessado pode se reportar à dissertação de Berthold [7] onde é feito um estudo sobre heurísticas para gerar limitantes primais que se aplicam a qualquer modelo de programação linear inteira mista. Diversas técnicas são apresentadas pelo autor e um estudo experimental é realizado para comparar a qualidade dos limitantes obtidos. Quando um limitante primal não puder ser calculado, utiliza-se um valor superestimado para ele. Após calcular os limitantes, verificamos se foi encontrado o valor ótimo. Se $\underline{z} = \overline{z}$ o ótimo foi encontrado[2] e o método termina a execução. Caso contrário, prosseguimos com a otimização divindo P em dois subproblemas.

Para efetuar a divisão de P, primeiramente escolhemos uma variável em x, digamos x_f, que tem valor fracionário λ. Sejam P_1 e P_2 os subproblemas que estamos criando. De-

[2]O teste também pode comparar se $\overline{z} - \underline{z}$ está abaixo de um valor que assegure que o valor primal é ótimo.

finimos P_1 como sendo igual a P incluindo a restrição $x_f \leq \lfloor \lambda \rfloor$, e P_2 igual a P incluindo a restrição $x_f \geq \lceil \lambda \rceil$. Note que com esta divisão não perdemos nenhuma solução inteira, portanto, o ótimo de P está em P_1 ou em P_2. Então, para encontrá-lo resolvemos P_1 e P_2 recursivamente, de forma semelhante a que estamos resolvendo P, diferindo apenas em algumas situações onde é possível descartar a resolução de subproblemas. São duas as situações onde isto ocorre. Primeiro, ao inserirmos uma nova restrição em um subproblema, este pode não admitir soluções, tornando-se infactível. Quando isto for identificado, por exemplo na resolução do problema de programação linear, descarta-se este subproblema. No segundo caso, ao obter um limitante dual de um subproblema, verificamos se este é maior ou igual ao melhor limitante primal de P encontrado. Se isto for verdade não será necessário resolvê-lo, pois o seu ótimo é maior ou igual ao limitante primal de P, logo, maior ou igual ao ótimo de P. Por este motivo, ao resolvê-lo, na melhor das hipóteses, encontramos um valor igual ao que já temos. Note que quanto menor for o limitante primal de P, mais chances temos de eliminar um subproblema. Observe também que os limitantes primais de cada subproblema, da forma como foi sugerida, também são limitantes primais do problema P. Então, quando fazemos esta comparação com o limitante dual, consideramos sempre o menor limintante primal encontrado até então. Ao evitar resolver estes subproblemas, diminuimos a enumeração do espaço de busca, e conseqüentemente o tempo de resolução de P.

Este processo recursivo de resolução gera uma árvore de subproblemas. O nó raiz da árvore corresponde ao problema P, e os nós internos correspondem aos subproblemas gerados ao resolvê-lo. Na figura 8.1 é ilustrada uma árvore gerada para um problema com três variáveis inteiras $0 \leq x_1, x_2, x_3 \leq 1$. Os valores de $\overline{z}$ e $\underline{z}$ são posicionados próximo ao subproblema correspondente. Em cada aresta é dada a restrição acrescentada no subproblema (nó filho). Considere que os subproblemas na árvore foram resolvidos na ordem em que eles foram numerados, então, é nesta ordem que o melhor limitante primal foi atualizado. Neste percurso, os nós destacados na cor cinza foram descartados com base nos valores dos limitantes. Com isso, foi evitada a enumeração de 8 nós, ou seja, mais da metade caso todo o espaço de busca fosse enumerado.

O algoritmo 10 sumariza os passos do método discutido acima. Neste algoritmo, uma lista L, cujos itens são conjuntos de restrições, é utilizada para armazenar os subproblemas que são criados durante a execução do método e cujos valores ótimos ainda possam ser ótimos para o problema P original. Representamos, assim, um subproblema com o conjunto que contém exatamente todas as suas restrições que subdividem o problema P original, ou seja, aquelas que não fazem parte do modelo $\mathcal{M}$ inicial. Os passo 1 e 2 do algoritmo são inicializações. No passo 3 escolhemos um problema para ser resolvido. No passo 4 calcula-se o limitante dual e também é feita a verificação de infactibilidade. O passo 5 verifica se foi obtida uma solução de P e o limitante primal é atualizado quando

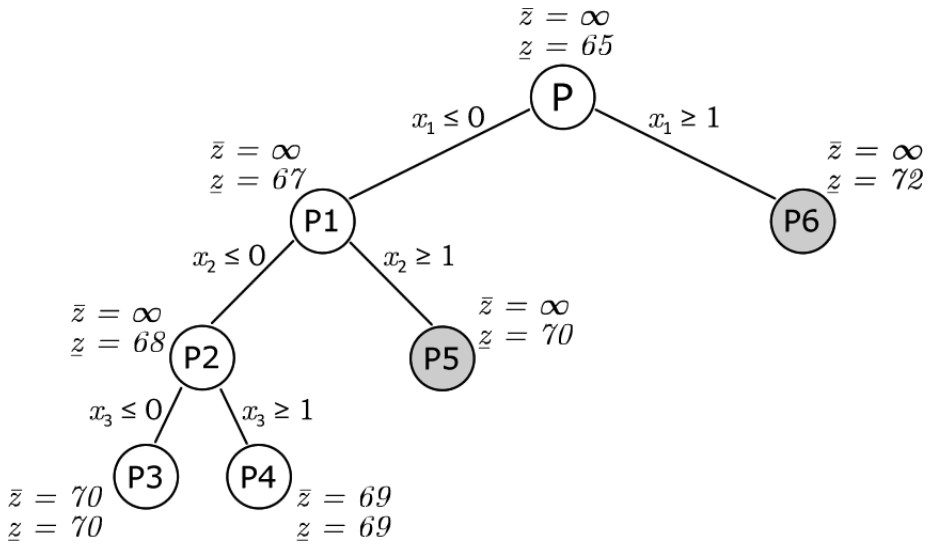


Figura 8.1: Exemplo de uma árvore de problemas gerada na execução do *branch-and-bound* aplicado a um problema de programação inteira com três variáveis inteiras.

Entrada: Um modelo matemático de programação linear inteira $\mathcal{M}$.
Saída: Uma solução ótima x^* para o modelo $\mathcal{M}$, ou um certificado de que o modelo não possui solução ($x^* = \emptyset$).

Passo 1: Faça $x^* = \emptyset$ e $\overline{z} = \infty$.
Passo 2: Crie uma lista L de conjuntos de restrições, iniciamente com um conjunto vazio (sem restrições).
Passo 3: Se $L \neq \emptyset$ retire um item S de L. Caso contrário vá para o passo 8.
Passo 4: Resolva a relaxação do problema correspondente a união de $\mathcal{M}$ com as restrições em S. Caso o modelo admita solução, denota-se por x a solução ótima encontrada e $\underline{z}$ seu valor. Caso contrário, volte para passo 3.
Passo 5: Se x satisfaz as restrições de integralidade de $\mathcal{M}$, então faça $x^* = x$ e $\overline{z} = \underline{z}$.
Passo 6: Se $\overline{z} \leq \underline{z}$ então volte ao passo 3.
Passo 7: Escolha uma variável x_f cujo valor, denotado por λ, não é inteiro. Crie duas cópias S_1 e S_2 de S. Adicione a restrição $y_f \leq \lfloor \lambda \rfloor$ em S_1, e a restrição $y_2 \geq \lceil \lambda \rceil$ em S_2. Insira S_1 e S_2 em L e volte ao passo 3.
Passo 8: Retorne x^*.

Algoritmo 10: Algoritmo *branch-and-bound* para problemas de programação linear inteira.

necessário. O passo 6 é o responsável por eliminar os subproblemas com base nos limitantes. A criação dos subproblemas é efetuada no passo 7. O algoritmo termina quando todos os subproblemas são resolvidos, ou seja, quando L estiver vazia.

Vimos que a qualidade dos limitantes gerados durante a execução do *branch-and-bound* dita a quantidade de nós que serão enumerados pelo método. Isto porque, quanto melhores eles forem mais chances se tem de descartar a resolução de subproblemas. Portanto, em muitos casos vale a pena gastar mais tempo no cálculo destes limitantes visando sua melhoria. Foi mostrado anteriormente que é possível utilizar heurísticas para tentar gerar bons limitantes primais. Na tentativa de melhorar os limitantes duais obtidos, pode-se adicionar restrições válidas (cortes) nos problemas relaxados, assim como é feito no método de planos de corte. Restrições válidas são aquelas que quando adicionadas ao modelo não eliminam nenhuma solução do problema P, mas podem eliminar soluções do seu problema relaxado. Ao fazer isso, as novas soluções obtidas no problema relaxado terão valores pelo menos tão bons quanto aquele obtido sem os cortes.

O *branch-and-cut* é um método que faz uso dessa idéia. Basicamente este método é um *branch-and-bound* que adiciona cortes durante a resolução de cada subproblema. Sendo assim, neste método adicionamos outras restrições que não estão no modelo corrente, na expectativa de melhorar o limitante dual obtido. Estas novas restrições podem fazer parte de uma família de restrições conhecidas *a priori* para o problema em questão, ou também podem ser restrições genéricas, tal como os cortes de Chvátal-Gomory (cf. Wolsey [47], seção 8.3.2) que se aplicam a qualquer modelo de programação linear inteira.

Outro caso em que se deve aplicar um método *branch-and-cut* no lugar de um método *branch-and-bound*, é quando temos um modelo matemático para um problema com uma quantidade exponencial de restrições em relação ao tamanho da entrada. Sabemos que, exceto para pequenas instâncias, não é viável carregar e resolver a versão relaxada de tal modelo em um computador. Mas como foi apresentado na seção 2.4, em certos casos ainda é possível resolver este tipo de modelo em tempo polinomial através de um método de planos de corte. Isso nos leva a um método *branch-and-bound* que adiciona cortes através de um método de planos de corte sendo executado em cada subproblema, ou seja, temos um método *branch-and-cut*.

Na literatura podem ser encontradas diversas referências sobre o *branch-and-bound* e o *branch-and-cut*. Uma boa introdução a respeito de cada um dos métodos é encontrada em Wolsey [47]. Mais referências a respeito podem ser encontradas em Nemhauser e Wolsey [34] e Mitchell [32, 33].

8.2 O Algoritmo *Branch-and-Cut* Desenvolvido

Desenvolvemos o algoritmo *branch-and-cut* utilizando a biblioteca *Callable Library* do ILOG CPLEX 12.1. Esta biblioteca disponibiliza rotinas para carregar um modelo matemático em memória. Com ele em memória é possível executar um método *branch-and-bound*, implementado na biblioteca, que se encarrega de otimizá-lo. Além disso, também é possivel implementar rotinas específicas que são chamadas em determinados momentos durante o processo de otimização. Dentre estas rotinas, implementamos uma para gerar cortes e uma outra para gerar soluções primais de forma heurística. Elas serão apresentadas com mais detalhes no decorrer desta seção.

Como será visto mais a frente, vamos trabalhar com instâncias cujas soluções possuem apenas valores inteiros. Então, quando a diferença do limitante primal para o limitante dual é inferior a uma unidade podemos afirmar que o valor do limitante primal é ótimo. Neste caso, encerramos a execução do algoritmo retornando a solução que temos no momento. É importante ressaltar também que, na nossa implementação, as rotinas de pré-processamento, corte e heurística primal implementadas pela própria biblioteca foram desabilitadas. As demais configurações disponibilizadas não foram alteradas.

8.2.1 Modelo Matemático

Dentre os modelos apresentados no capítulo 4, optamos por utilizar o modelo $\mathcal{M}_{AO}$ descrito na seção 4.2. A justificativa para esta escolha está no fato de que no trabalho de Oliveira [35] foi realizado um estudo poliédrico mostrando que esta é uma formulação forte para o problema. Além disso, também foi desenvolvido um algoritmo *branch-and-cut* que obteve bons resultados em instâncias do PASG para uma aplicação em problemas de roteamento em circuitos. Estas instâncias são constituídas de grafos planares com pesos nas arestas que respeitam a métrica retilinear. Desta forma, esperamos que este algoritmo também obtenha bons resultados para o PCCM, uma vez que, as instâncias do PCCM, quando convertidas em instâncias do PASG, possuem estas mesmas características.

Para este modelo, é preciso implementar a rotina de separação descrita na seção 4.2. Esta rotina é ajustada para ser invocada toda vez que o método *branch-and-bound* obtém uma solução fracionária em um nó. Quando isso ocorre, a rotina procura por uma inequação violada e, caso a encontre, adiciona-a ao modelo. Caso não seja encontrada nenhuma inequação violada, o método de *branch-and-bound* prossegue a otimização na sua forma usual. A rotina de separação consiste na resolução de uma sequência de problemas de corte mínimo. Porém, como visto na seção 4.2, também podemos utilizar a adaptação do algoritmo de Hao e Orlin para o problema de corte irrestrito mínimo proposta por Cronholm et al. [13]. Assim como o algoritmo de Hao e Orlin, esta adaptação também é uma modificação do algoritmo de fluxo máximo baseado na idéia de pré-fluxo, proposto

por Goldberg e Tarjan [22]. Portanto, implementamos este algoritmo de fluxo máximo e o modificamos da forma como é descrita em Hao e Orlin [26] e Cronholm et al. [13]. Com isso, a rotina de separação em questão foi desenvolvida fazendo uso deste algoritmo, assim como foi apresentada na seção 4.2. Além disso, também empregamos nesta rotina a estratégia denominada *Creep Flow* apresentada em Koch e Martin [31]. Nesta estratégia definimos um valor mínimo para as capacidades que atribuímos no grafo onde calculamos o corte mínimo. Isto é feito para que a rotina encontre cortes com uma menor quantidade de arcos, levando a desigualdades mais fortes. Então, ao atribuirmos uma capacidade a um arco do grafo, esta deverá ser escolhida como sendo o máximo entre o valor da variável correspondente na solução fracionária e um milionésimo.

Além desta rotina de separação, também implementamos outra rotina de separação considerando as restrições de fluxo apresentadas em Koch e Martin [31]. As restrições nesta referência são específicas para o PAS. Elas modelam uma árvore de Steiner através de um conjunto de fluxos de uma unidade que saem do vértice terminal considerado como raiz da árvore e chegam em cada um dos demais terminais. Para poder utilizar estas restrições no nosso modelo, foi necessário adaptá-las para o PASG. Sendo assim, no nosso caso teremos um conjunto de fluxos de uma unidade que saem do vértice raiz e chegam em pelo menos um vértice de cada grupo. As restrições para o PASG são dadas pelas equações abaixo, considerando as variáveis do modelo $\mathcal{M}_{AO}$.

$$\sum_{a \in \delta_D^-(\{v\})} x_a \leq \begin{cases} 1 - y_v, & \forall \quad v \in R_0, \\ 1, & \forall \quad v \in V \setminus R_0, \end{cases} \tag{8.4}$$

$$\sum_{a \in \delta_D^-(R)} x_a \geq 1, \qquad \forall \quad R \in \Re \setminus \{R_0\}, \tag{8.5}$$

$$\sum_{a \in \delta_D^-(\{v\})} x_a \leq \sum_{a \in \delta_D^+(\{v\})} x_a, \qquad \forall \quad v \in V \setminus \bigcup_{R \in \Re} R, \tag{8.6}$$

$$\sum_{a' \in \delta_D^-(\{v\})} x_{a'} \geq \begin{cases} x_a - y_v, & \forall \quad v \in R_0, a \in \delta_D^+(\{v\}), \\ x_a, & \forall \quad v \in V \setminus R_0, a \in \delta_D^+(\{v\}). \end{cases} \tag{8.7}$$

No modelo $\mathcal{M}_{AO}$ determinamos qual será o nó raiz da árvore através das variáveis y. Desta forma, na restrição (8.4) estabelecemos que todos os vértices do grafo podem receber no máximo uma unidade de fluxo, à exceção do vértice raiz que pode apenas enviar fluxo. A restrição (8.5) determina que uma unidade de fluxo deve chegar a pelo menos um vértice de cada grupo que não é raiz. Segundo a restrição (8.6), a quantidade de fluxo que sai de um vértice que não pertecente a nenhum grupo deve ser no mínimo a mesma quantidade de fluxo que entra neste vértice. Observe que pode sair mais fluxo do que entra no vértice. Este desbalanceamento é necessário para representar as ramificações da árvore. Finalmente, a restrição (8.7) impõe que a quantidade de fluxo que passa por um

arco é no máximo a quantidade de fluxo que chega no vértice de onde este arco parte, salvo os arcos que partem do vértice raiz que podem enviar mais fluxo do que este vértice recebe.

Além destas restrições, incluímos também na rotina de separação a família de restrições dada pela equação abaixo.

$$x_a + x_{a'} \leq 1, \qquad \forall \quad u, v \in V, a = (u, v), a' = (v, u) \in A, \tag{8.8}$$

A finalidade desta restrição é impedir que os dois arcos que representam uma mesma aresta do grafo orientado estejam em uma solução. Observe que a princípio não necessitamos desta condição, pois toda instância com pesos positivos já a satisfaz. Todavia, a consideramos porque verificamos através de testes experimentais que ela ajuda a acelerar a execução do método de planos de corte.

Apesar de termos considerado as restrições (8.4)-(8.8) para a rotina de separação, não utilizamos as restrições (8.5) e (8.6) na rotina. Primeiramente, note que para o PCCM a restrição (8.6) corresponde a um conjunto vazio de inequações, pois sabemos que as instâncias do PASG que representam instâncias do PCCM têm todos os vértices em G pertencendo a pelo menos um grupo. Também verificamos experimentalmente que acrescentar a restrição (8.5) deixa a resolução do modelo relaxado muito lenta e não melhora a qualidade dos limitantes duais.

Durante o processo de otimização existem rotinas implementadas pela biblioteca *Callable Library* que se encarregam de fazer a manutenção do modelo. Tais rotinas procuram por restrições redundantes ou que já não estão mais justas e as removem do modelo. Entretanto, quando inserimos no modelo uma nova restrição da família dada pela equação (4.13), marcamo-la para nunca ser removida do modelo. Isso foi feito porque constatamos experimentalmente que impedir a remoção destas restrições levou o método a obter melhores resultados.

8.2.2 Heurística Primal

Utilizando a biblioteca *Callable Library*, é possível executar uma rotina para gerar uma solução do problema inteiro cada vez que uma nova solução fracionária é obtida em um nó. Todavia, por causa da adição de novas restrições no nó, a quantidade de soluções fracionárias obtidas tende a ser muito grande, o que pode nos levar a gastar muito tempo neste processo. Para contornar isso, executamos a rotina implementada com uma certa frequência pré-definida pelo parâmetro $HeuPrimalFreq = 10$. Este parâmetro fixa a quantidade de soluções fracionárias que devemos desconsiderar, até que a heurística seja chamada novamente. As soluções geradas são passadas para as rotinas da biblioteca que se encarregam de atualizar o limitante primal quando necessário.

A heurística primal implementada consiste em uma modificação do algoritmo aproximativo ACM descrito na seção 5.2. Isto é feito de modo a aproveitar a informação contida em uma solução fracionária obtida durante o processo de otimização. Assim, a alteração proposta consiste em utilizar esta informação no procedimento de escolha dos vértices que o algoritmo seleciona para cobrir os grupos que ainda não foram cobertos, enquanto a solução está sendo construída. Para isso, vamos selecionar um conjunto X que contém os vértices que julgamos ser os mais interessantes. Tal seleção é feita com base nos valores das variáveis x e y da solução fracionária corrente. Então, mudamos a definição do conjunto P no passo 6 do algoritmo 2 (página 30). Agora P passa a ser o conjunto de todos os caminhos que iniciam em um vértice em T' e terminam em um vértice em $W \cap X$.

A decisão de quais vértices vão pertencer a X será feita através de uma regra baseada na satisfação das restrições (8.9) e (8.10). Seja R_0 o grupo escolhido como raiz no modelo $\mathcal{M}_{AO}$, (x^f,y^f) uma solução fracionária obtida durante o processo de otimização e ρ a maior cardinalidade de um grupo em $\Re$. Então, se v for um vértice para o qual estamos decidindo pela pertinência ou não em X, temos que v será incluído em X somente se ele não pertencer a R_0 e satisfazer a restrição (8.9), ou então, se ele pertencer a R_0 e satifazer a restrição (8.10).

$$\sum_{a \in \delta_D^-(\{v\})} x_a^f \geq \frac{1}{\rho} \tag{8.9}$$

$$\sum_{a \in \delta_D^-(\{v\})} x_a^f + y_v^f \geq \frac{1}{\rho} \tag{8.10}$$

Esta regra é baseada no valor do grau de entrada de cada vértice na solução fracionária. São escolhidos apenas os vértices onde este valor esta acima do limiar $1/\rho$, seguindo a mesma lógica do algoritmo aproximativo apresentado na seção 5.1. É importante também observar que não é possível garantir que em X sempre vai existir pelo menos um vértice de cada grupo em $\Re$. Isto se deve ao fato de que, durante a execução do algoritmo de planos de corte, normalmente não estamos com o modelo $\mathcal{M}_{AO}^R$ completo. Portanto, é possível que não tenhamos uma solução que satisfaça todas as suas restrições. Assim, após criar X utilizando a regra descrita acima, verificamos para cada $R \in \Re$ se a interseção de X com R é vazia. Caso seja, acrescentamos todos os vértices de R em X. Com isso, garantimos que em X sempre vai existir pelo menos um vértices de cada grupo.

Após gerar uma solução utilizando esta rotina, não há garantias de que esta seja um mínimo local. Por isso, aplicamos-lhe uma combinação das heurísticas de melhoria do capítulo 6 visando encontrar uma solução ainda melhor. A combinação empregada em nossa implementação é descrita no capítulo de resultados computacionais, na seção 9.2.3.

Por último, é importante dizer que apesar deste algoritmo ser uma modificação de um algoritmo aproximativo, não podemos considerá-lo como tal. A prova do fator de aproximação do algoritmo original não é mais válida após a alteração do passo 6. Portanto, o algoritmo que usamos para gerar soluções deve ser considerado como uma heurística.

Capítulo 9

Resultados Computacionais

Neste capítulo serão apresentados resultados obtidos a partir da experimentação dos algoritmos exato, aproximativos e heurístico discutidos nos capítulos anteriores. Até onde pesquisamos, não foi possível encontrar um conjunto de instâncias teste para o PCCM na literatura. Por este motivo, primeiramente propomos uma maneira de construí-las e implementamos um gerador. Feito isso, estabelecemos um conjunto de testes ao qual submetemos os algoritmos. Finalmente, analisamos os resultados alcançados e apresentamos as conclusões finais.

9.1 Geração de Instâncias

Uma instância do PCCM é um polígono retilinear particionado em polígonos retilineares menores não sobrepostos. Propomos gerar este tipo de figura de modo aleatório e em duas etapas. Para tanto, iniciamos com um quadrado de um determinado tamanho. Em uma primeira fase, particionamos o interior desse quadrado construindo uma subdivisão formada por retângulos retilineares. Isso é feito sorteando um ponto de forma aleatória dentro do quadrado. A partir desse ponto podem sair quatro segmentos de reta, dois verticais e dois horizontais, que iniciam no ponto e terminam na primeira interseção com um segmento da subdivisão corrente. Sorteamos, com uma probabilidade de 95%, se vamos adicionar todos os quatros segmentos. Se a probabilidade não for atingida, escolhemos aleatoriamente três dos quatros segmentos para serem adicionados. A justificativa para adicionarmos de três a quatro segmentos por vez, com mais chances para quatro segmentos, é devida ao fato de que na próxima fase serão feitas remoções de segmentos. Portanto, em um primeiro momento, é interessante termos uma grande quantia deles. Ilustramos o procedimento desta etapa inicial na figura 9.1 (a).

É importante ressaltar que tanto os vértices do quadrado inicial quanto os pontos gerados aleatoriamente para efetuar as subdivisões têm coordenadas inteiras. Isto faz

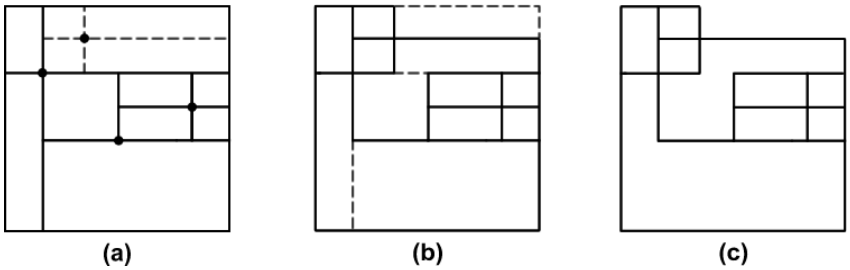


Figura 9.1: Exemplo das etapas do processo de geração de instâncias proposto.

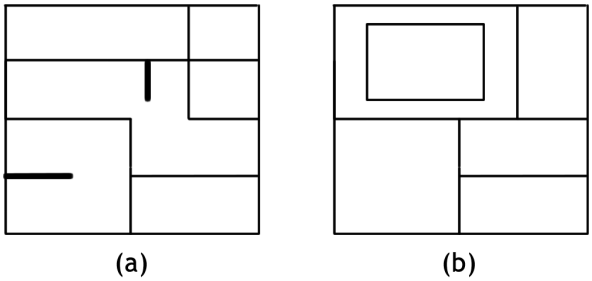


Figura 9.2: Exemplo de duas subdivisões com estruturas inadequadas. Em (a) é apresentada uma instância com segmentos que podem ser removidos sem alterar suas soluções ótimas, e em (b) uma subdivisão desconexa que não admite corredores.

com que os comprimentos dos segmentos sejam inteiros. As demais operações aplicadas no decorrer do processo de geração, e vistas a seguir, não alteram esta propriedade. Com isto, todas instâncias obtidas pelo procedimento possuem esta característica.

Na segunda fase (figura 9.1 (b)), removemos segmentos com determinadas características da subdivisão para unir duas faces adjacentes. Esse processo é aplicado com a finalidade de tornar a estrutura da subdivisão menos regular, pois sem esta etapa teríamos apenas faces retangulares. Depois que realizamos as remoções podemos ter faces bem irregulares como pode ser visto no exemplo da figura 9.1 (c). Quanto maior a quantidade de remoções mais irregular será a estrutura obtida.

A remoção dos segmentos nesta fase deve ser feita com algum cuidado para não criarmos estruturas indesejadas. Vamos apresentar, através de exemplos, quatro diferentes casos onde ocorrem estruturas inadequadas. A figura 9.2 (a) apresenta uma instância com segmentos destacados em negrito que não contribuem para a divisão da estrutura, podendo assim ser removidos sem alterar qualquer solução ótima da instância. Já a figura 9.2 (b) apresenta uma subdivisão que não admite corredores pois ela é desconexa.

Nas figuras 9.3 (a) e 9.4 (a) são dadas instâncias com pontos destacados em negrito que obrigatoriamente farão parte de qualquer corredor. Em razão disso, podemos decompô-las em instâncias menores (em termos da quantidade de segmentos). Todavia, esta decom-

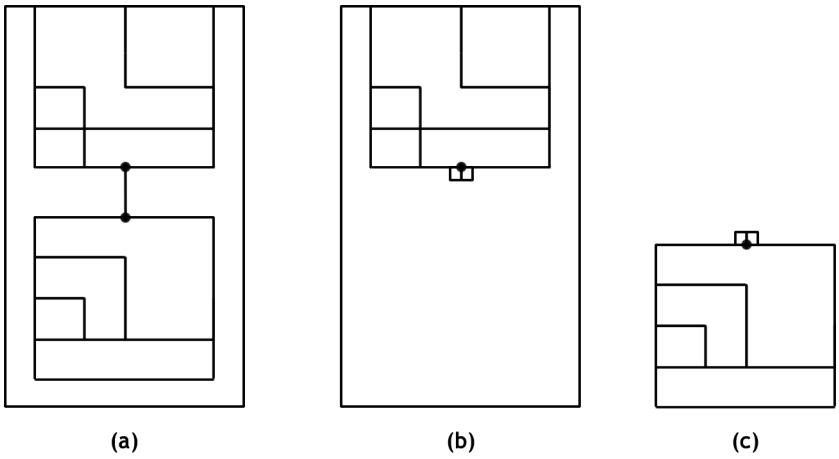


Figura 9.3: Exemplo de uma instância que pode ser resolvida como duas instâncias menores (em termos da quantidade de segmentos).

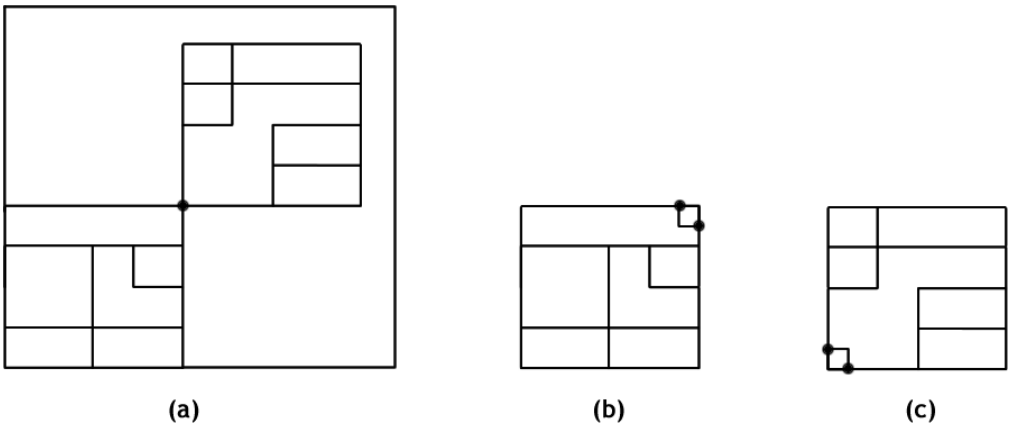


Figura 9.4: Outro exemplo de uma instância que pode ser resolvida como duas instâncias menores (em termos da quantidade de segmentos).

posição deve ser feita de forma apropriada para que os corredores nas instâncias menores estejam sempre conectados aos pontos destacados em negrito. Assim, é possível uni-los levando a uma solução ótima para a instância original. Para tal, nas figuras 9.3 (b) e (c) criamos uma estrutura com duas pequenas salas quadradas justapostas fazendo com que o corredor se conecte na junção entre elas. Nas figuras 9.4 (b) e (c) pelo fato do ponto estar em uma quina foi suficiente criar apenas uma sala onde o corredor estará ligado em um dos dois pontos destacados. Neste caso, para obter um corredor ótimo da instância original ligamos os dois corredores ótimos das instâncias menores ao ponto destacado na instância original através do menor caminho que vai dos corredores até ele.

Desta forma, antes de realizar a remoção de segmentos na subdivisão que estamos gerando, é preciso verificar se isso não nos leva a algum desses casos. Para isso, vamos trabalhar com o grafo induzido pela subdivisão. Neste grafo, vamos remover arestas pois assim estaremos evitando o problema apresentado na figura 9.2 (a). Agora, note que os vértices destacados nas figuras 9.3 (a) e 9.4 (a) constituem pontos de articulação (vértices de corte) no grafo. Sabemos que um grafo possui pontos de articulação se, e somente se, ele não for biconexo (veja Jungnickel [29], seção 8.3). Então, se garantirmos esta propriedade no grafo, evitamos estes casos, e obviamente também aquele apresentado na figura 9.2 (b). Observe que na primeira fase geramos uma subdivisão que induz um grafo biconexo. A partir daí vamos manter o grafo biconexo durante as remoções das arestas. Ao remover uma aresta, verificamos se foi criado um ponto de articulação no grafo através de uma busca em profundidade (encontra-se tal algoritmo na referência citada acima). Caso tenha sido criado, inserimos a aresta de volta no grafo. Enfim, com este procedimento garantimos que a instância não terá nenhuma das estruturas indesejáveis apresentadas.

Conhecendo as duas etapas de geração da subdivisão, podemos fazer alguns ajustes para gerar tipos diferentes de instâncias. Definimos cinco tipos de instâncias apresentados na figura 9.5. Para o tipo 1 executamos apenas a primeira fase, gerando uma subdivisão formada apenas por polígonos retangulares. O tipo 2 consiste na execução de poucas remoções na segunda etapa, e o tipo 3 são executadas muitas remoções. Os tipos 4 e 5 são, respectivamente, semelhantes aos tipos 2 e 3, entretanto alteramos a primeira fase para gerar uma grade regular, com todas as faces sendo quadrados de tamanho unitário, ao invés de gerar um polígono particionado aleatoriamente.

Após gerar uma instância do PCCM, esta é reduzida a uma instância do PASG, como descrito na seção 3.1. No grafo correspondente a esta instância aplicamos as remoções de vértices com grau dois. Isso faz com que ela deixe de respeitar a desigualdade triangular, pois podem haver arestas com peso maior ou igual ao peso de um caminho que liga os vértices extremos desta aresta. Esta propriedade é necessária para garantir o fator de aproximação do algoritmo aproximativo AAGM. Deste forma, para voltar a ter uma instância que respeita esta condição realizamos o seguinte procedimento. Para cada aresta

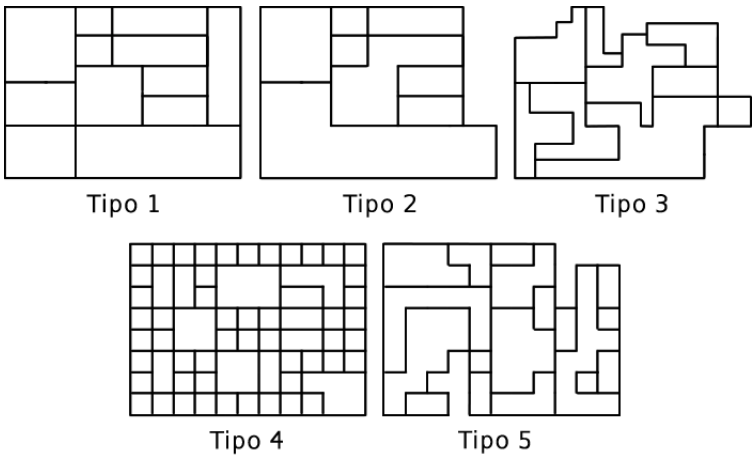


Figura 9.5: Exemplo dos diferentes tipos de instâncias.

do grafo, procuramos pelo menor caminho que liga os vértices extremos da aresta e não passa pela aresta. Se este caminho tiver peso menor ou igual ao peso da aresta, ela é removida. Note que ao fazer isso pelo menos uma solução ótima da instância será preservada.

A geração do conjunto de instâncias teste foi feita com base nos cinco tipos de instâncias apresentados e no número de faces (salas) presentes nas instâncias. Cabe ressaltar que a criação de uma instância com um número fixo de salas é uma tarefa simples bastando, para isso, controlar a quantidade de faces na primeira etapa e de remoções na segunda etapa. Deste modo, foram criadas instâncias com as seguintes quantidades de salas: 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000, 1100, 1200, 1300, 1400, 1500. Para cada combinação de tipo e quantidade de salas geramos cinco instâncias aleatoriamente. No total geramos um conjunto de teste com 375 diferentes instâncias que se encontram disponíveis para serem baixadas em `http://www.ic.unicamp.br/~cid/Problem-instances/MinCorridor`.

9.2 Análise dos Resultados

As implementações dos métodos apresentados foram feitas na linguagem de programação C++ e compiladas com o GNU GCC versão 4.5.1. Todos os testes foram realizados no sistema operacional Linux distribuição Fedora 14, em uma máquina equipada com um processador Intel Core i7-820QM e 8 gigabytes de memória RAM.

Durante esta seção são utilizadas algumas métricas para avaliar o desempenho dos algoritmos, tais como o tempo de execução médio (TEM) (medido em segundos), desvio relativo médio do comprimento do corredor (DRMCC), desvio relativo médio do tempo

de execução (DRMTE) e o percentual de resultados mínimos (PRM). Ressaltamos que o desvio relativo médio corresponde à média dos desvios relativos, e não ao desvio relativo das médias.

O desvio relativo de um valor z_1 para um valor z_2 é dado pela equação $100(z_1 - z_2)/z_2$. O DRMCC e o DRMTE nos gráficos apresentados são seguidos por hífen e o nome do algoritmo para o qual estamos efetuando o desvio. No cálculo do DRMTE, um desvio é considerado nulo quando temos o valor absoluto de $z_1 - z_2$ menor ou igual a um. Isso foi adotado porque constatamos pequenas variações no tempo de execução de um mesmo algoritmo em uma mesma instância. Então, quando não há uma diferença no tempo de execução maior que um segundo consideramos que os tempos de execução são iguais.

Dados um conjunto de algoritmos A e um conjunto de instâncias I, denote por z_{ai} o comprimento do corredor obtido por um algoritmo $a \in A$ em uma instância $i \in I$. O PRM de um algoritmo $a \in A$ neste contexto é dado pela equação (9.1).

$$100 \times \frac{\sum_{i \in I} m(a,i)}{|I|} \tag{9.1}$$

$$m(a,i) = \begin{cases} 1, & z_{ai} = \min_{a' \in A}\{z_{a'i}\} \\ 0, & z_{ai} \neq \min_{a' \in A}\{z_{a'i}\} \end{cases} \tag{9.2}$$

A seguir são apresentados resultados individuais de cada um dos algoritmos estudados nos capítulos anteriores, e após isso realizamos a comparação entre todos eles na seção 9.2.4. No decorrer deste capítulo, são apresentadas médias de resultados obtidos em grupos de instâncias com determinadas características, tais como quantidade de salas fixa, tipo da instância, etc. Estas médias sempre são calculadas considerando resultados de todas as instâncias no conjunto de teste com tal característica, salvo se o texto informar o contrário.

9.2.1 Análise dos Algoritmos Aproximativos e Heurísticas de Melhoria

Primeiramente vamos analisar os resultados obtidos pelos algoritmos aproximativos ACM e AAGM apresentados no capítulo 5, e o impacto nos resultados quando são aplicadas as heurísticas de melhoria HD, HC e HR, propostas no capítulo 6, nas soluções encontradas por estes algoritmos.

Mas antes disso, destacamos um resultado que mostra o quanto é importante a análise do comportamento de um algoritmo na prática. Na teoria diversos algoritmos apresentam bons resultados porém, é só com sua experimentação prática que os seus pontos fracos acabam por se revelar. Assim, o algoritmo aproximativo AAGM parecia ser uma boa

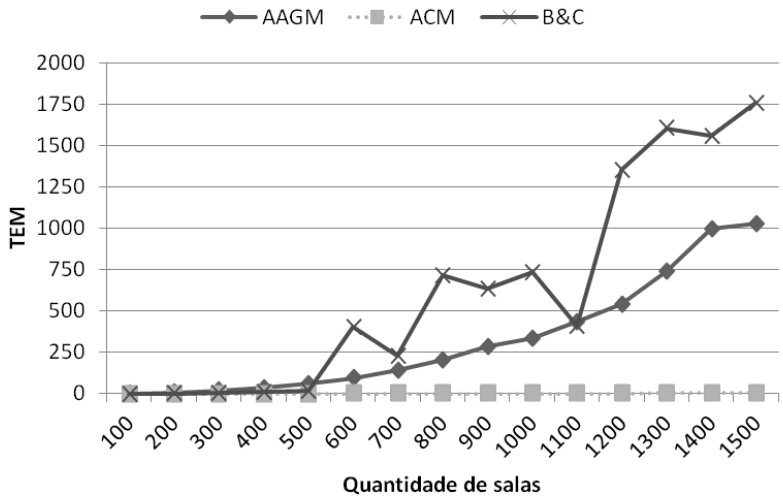


Figura 9.6: Comparação dos tempos de execução médios dos algoritmos AAGM, ACM e *branch-and-cut* (B&C) em função da quantidade de salas nas instâncias.

escolha na teoria, principalmente pelo fato de seu fator de aproximação depender da maior quantidade de pontos na borda de uma sala. Contudo, na prática, este algoritmo mostrou ser muito lento quando comparado com o algoritmo ACM, e nem sempre tão mais rápido quando comparado com o algoritmo exato de *branch-and-cut*, como pode ser visto no gráfico da figura 9.6 que confronta os tempos de execução médios dos três algoritmos. Na medida que a quantidade de salas nas instâncias aumenta o tempo de execução médio do algoritmo AAGM cresce consideravelmente. Isso acontece devido à resolução do modelo relaxado $\mathcal{M}^R_{CV}$, que experimentalmente mostrou ser lenta. Além disso, no gráfico da figura 9.7 apresentamos o DRMCC do algoritmo AAGM para o ACM. Neste gráfico observamos que os corredores obtidos pelo algoritmo AAGM são em média pelo menos 32% maiores que os corredores obtidos pelo algoritmo ACM. Como o AAGM é mais lento e apresenta soluções que em média são piores que aquelas fornecidas pelo ACM, ele foi descartado nas próximas análises.

Sendo assim nos resta analisar os resultados obtidos pelo algoritmo aproximativo ACM combinado com as heurísticas de melhoria. Como existem diferentes formas de combinar estas heurísticas, primeiro vamos avaliar o comportamento das versões *First Fit* e *Best Fit* de cada heurística separadamente e, após isso, investigamos como combiná-las. Os testes foram realizados da seguinte forma. Para cada uma das heurísticas e suas versões, executamos o algoritmo ACM uma vez em cada instância com mil ou mais salas e, em seguida, aplicamos a heurística na solução obtida pelo algoritmo. Decidimos realizar esses testes considerando apenas as instâncias com maior número de salas, porque são

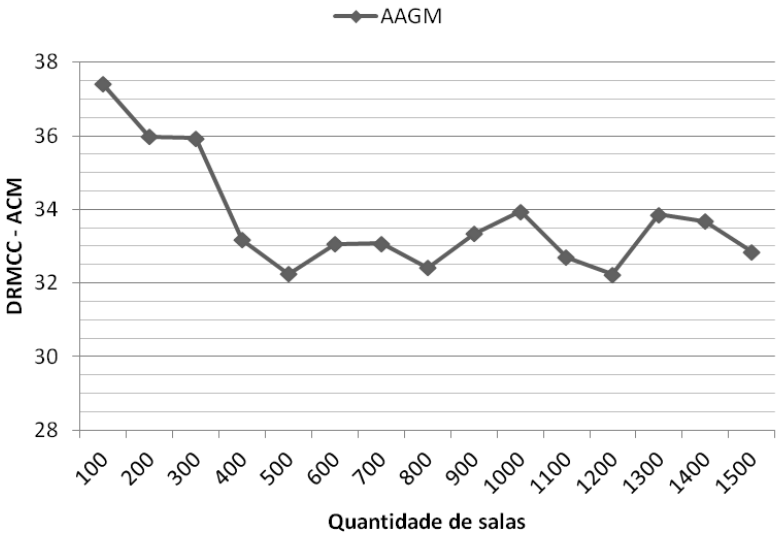


Figura 9.7: Comparação da qualidade das soluções obtidas pelos algoritmos AAGM e ACM em função da quantidade de salas nas instâncias.

nessas circunstâncias que a diferença entre os resultados é mais evidente. Denominamos o algoritmo ACM seguido da execução de uma heurística por ACM seguido de hífen e o nome da heurística.

Através dos gráficos apresentados nas figuras 9.8 e 9.9 comparamos os resultados obtidos pelo algoritmo ACM combinado com uma mesma heurística, mudando apenas sua versão. Estes gráficos comparam o DRMCC e o DRMTE do algoritmo ACM combinado com alguma das heurísticas com o algoritmo ACM sem utilizar nenhuma heurística, e o PRM entre os algoritmos que utilizam uma mesma heurística. O DRMCC teve seu sinal invertido nos gráficos para que ele pudesse ser melhor apresentado juntamente com o DRMTE e o PRM. O DRMTE no gráfico da figura 9.8 foi omitido pois todos os seus valores são nulos, ou seja, a aplicação destas heurísticas não aumentou substancialmente o tempo de execução. No gráfico da figura 9.9 o DRMTE foi dividido por cem para que seja exibido em uma escala que permite uma melhor visualização dos resultados.

A diferença do DRMCC entre os algoritmos utilizando uma mesma heurística não é muito significativa, mas este valor apresenta em média o quanto cada algoritmo melhora em relação a solução do algoritmo ACM sem utilizar nenhuma heurística. O algoritmo ACM-HR obteve na média os melhores valores de DRMC. Entretanto, esta heurística quando aplicada aumenta muito o tempo de execução médio do algoritmo, como pode ser visto pelo seu DRMTE. O PRM deixa mais claro quais versões das heurísticas obtêm os melhores resultados, mesmo que a diferença entre estes resultados não seja suficientemente grande para ser destacada no DRMCC. Através do PRM podemos concluir que o algoritmo

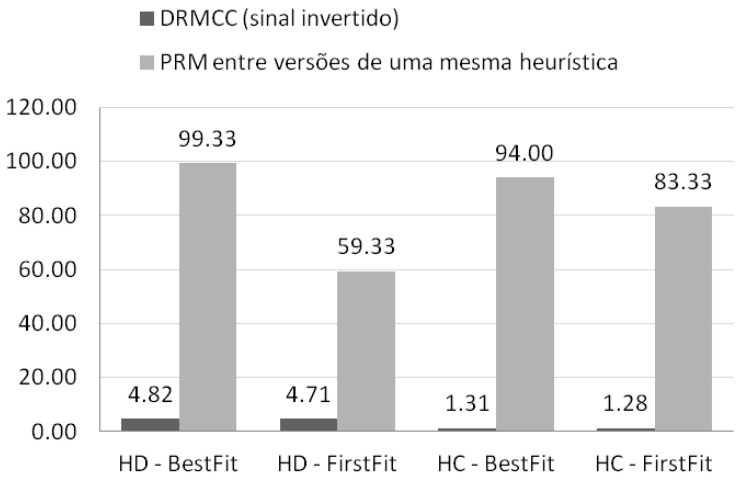


Figura 9.8: Comparação das versões *First Fit* e *Best Fit* das heurísticas HD e HC.

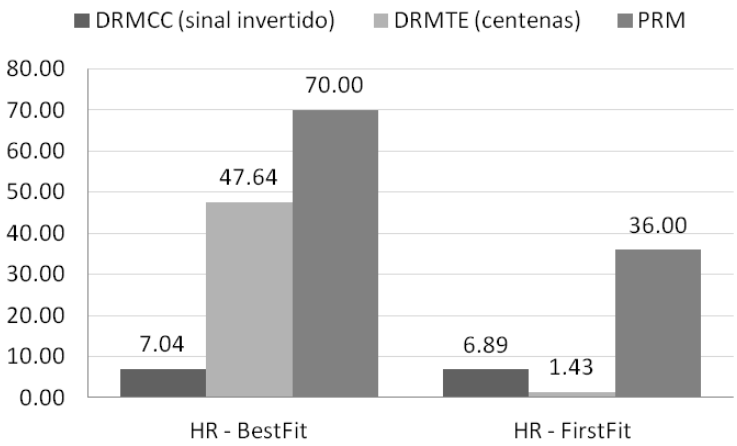


Figura 9.9: Comparação das versões *First Fit* e *Best Fit* da heurística HR.

ACM-HD na versão *Best Fit* encontra 40% a mais de corredores com comprimento menor que a versão *First Fit*. No algoritmo ACM-HR essa diferença é 34%. Contudo neste algoritmo existe um compromisso no tempo de execução como mostra a diferença do DRMTE de 4621%. A diferença do PRM entre os algoritmos que usam as versões da heurística HC não foi tão grande quanto os demais algoritmos, todavia, ainda é possível destacar a versão *Best Fit* como melhor opção. Através do PRM também fica evidente que a versão *Best Fit* de cada heurística, mesmo obtendo os melhores resultados na média, não foi capaz de obter o melhor resultado em todas as instâncias.

Como nas heurísticas HD e HC a versão *Best Fit* obteve os melhores resultados sem o acréscimo significativo do tempo de execução, vamos desconsiderar a versão *First Fit* delas. O mesmo não pode ser dito para a heurística HR que apresenta uma diferença substancial no tempo de execução entre as versões. Os próximos resultados que vamos apresentar mostram experimentalmente a melhor forma que encontramos de combinar estas heurísticas.

Através de testes preliminares concluímos que a heurística HD deve ser sempre executada na solução inicial das heurísticas HC e HR (antes do primeiro passo), e na melhor solução sempre que ela for atualizada nas heurísticas HC (durante o passo 7 do algoritmo 4) e HR (durante o passo 7 do algoritmo 5, e passo 8 do algoritmo 6). Ao fazer isso obtivemos bons resultados sem um grande acréscimo no tempo de execução. Distinguimos estas heurísticas HC e HR combinadas com a heurística HD, denominando-as repectivamente por HC′ e HR′.

Destes testes preliminares, selecionamos as melhores combinações das heurísticas. Com base neles, notamos que a melhor forma de combiná-las é aplicar primeiro HC′ e, em seguida HR′. Este processo pode ser repetido, intercalando a execução destas heurísticas, até que uma delas não consiga mais melhorar a solução. As combinações escolhidas foram nomeadas como CRC, CRCRC, e (CR)*. As letras C e R representam a aplicação de HC′ e HR′ respectivamente. Estas heurísticas são executadas na ordem correspondente ao nome da combinação. Por exemplo, CRC significa que executa-se primeiro HC′, depois HR′ e, em seguida, HC′ novamente, encerrando-se o processo. Na combinação que tem o nome delimitado por (e)*, significa que ela é repetida até que a solução não seja melhorada.

Na avaliação destas combinações, consideramos novamente cada versão da heurística HR′, executando o algoritmo ACM uma vez em cada instância com mil ou mais salas e, em seguida, aplicamos a combinação heurística na solução obtida pelo algoritmo. Denominamos o algoritmo ACM seguido da execução de uma combinação por ACM seguido de hífen e o nome da heurística combinada. As figuras 9.10 e 9.11 apresentam gráficos que comparam os algoritmos considerando, respectivamente, as versões *First Fit* e *Best Fit* da heurística HR′. Nestes gráficos comparamos os DRMCC e DRMTE do algoritmo

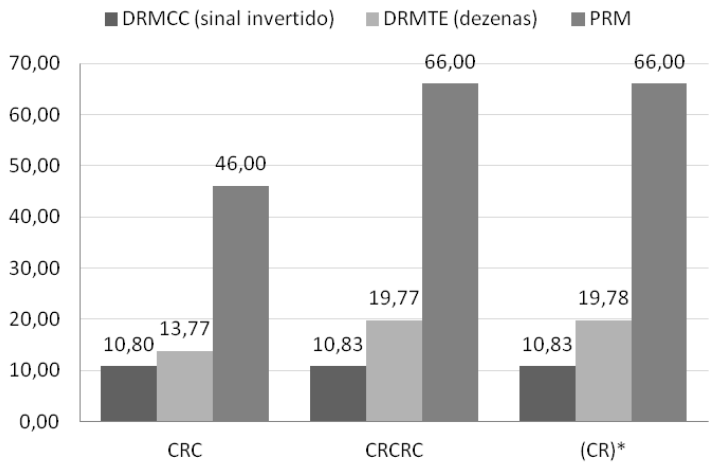


Figura 9.10: Comparação das combinações das heurísticas considerando a versão *First Fit* da heurística HR′.

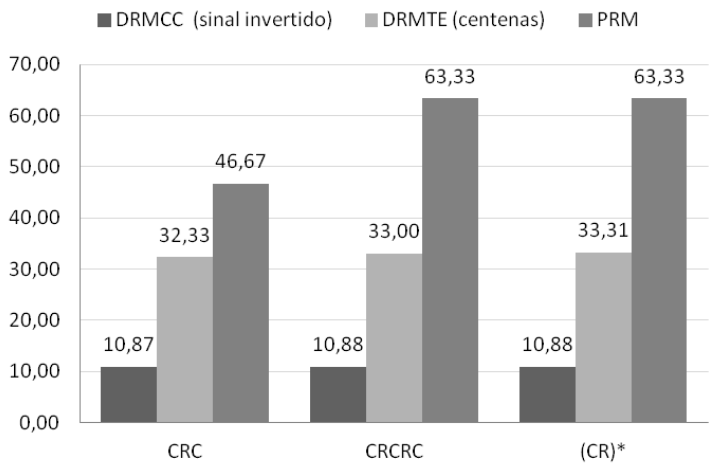


Figura 9.11: Comparação das combinações das heurísticas considerando a versão *Best Fit* da heurística HR′.

ACM usando uma heurística combinada com o algoritmo ACM puro. Também é comparado o PRM entre os algoritmos que utilizam uma mesma versão da heurística HR′. Novamente o DRMCC teve seu sinal invertido para que possa ser apresentado juntamente com o DRMTE e o PRM nos gráficos. O DRMTE também foi divido por dez no gráfico da figura 9.10, e por cem no gráfico da figura 9.11.

Ao analisar ambos os gráficos, comparando os resultados obtidos pelas heurísticas CRCRC e (CR)*, concluímos que não adianta executar a intercalação das heurísticas mais que a quantidade de vezes executada pela heurísticas CRCRC, pois os resultados não são melhorados. Mais uma vez o DRMCC é muito próximo entre as heurísticas, desta forma, recorremos novamente ao PRM para decidir quais são as melhores heurísticas. Observamos que a heurística CRCRC encontra 20% a mais de soluções com comprimentos menores que a heurística CRC considerando a versão *First Fit* da heurística HR′, e 16,66% a mais considerando a versão *Best Fit*. Contudo deve-se considerar um compromisso entre qualidade e tempo de execução, pois a heurística CRCRC gasta em média 60% a mais de tempo em sua execução que a heurística CRC considerando a versão *First Fit* da heurística HR′, e em média 67% a mais quando consideramos a versão *Best Fit*. Por fim, note que as heurísticas que utilizam a versão *Best Fit* da heurística HR′ consomem muito mais tempo em sua execução, tendo um DRMTE com pelo menos 3000% a mais.

Finalizamos esta análise do algoritmo ACM combinado com as heurísticas de melhoria comparando as duas versões da melhor heuristica não combinada, ou seja, HR com as quatro melhores heurísticas combinadas destacadas na análise dos gráficos 9.10 e 9.11, que são CRC e CRCRC, ambas considerando as duas versões da heurística HR′. Chamaremos de ACM-H1, ACM-H2, ACM-H3 e ACM-H4, respectivamente, o algoritmo ACM utilizando as combinações de heurísticas CRC com a versão *First Fit* de HR′, CRCRC com a versão *First Fit* de HR′, CRC com a versão *Best Fit* de HR′ e CRCRC com a versão *Best Fit* de HR′. A tabela abaixo resume o significado das heurísticas combinadas utilizadas com o algoritmo ACM.

Tabela 9.1: Significado das heurísticas combinadas H1, H2, H3 e H4.

Comb. / **HR′**	CRC	CRCRC
First Fit	H1	H2
Best Fit	H3	H4

As figuras 9.12, 9.13 e 9.14 apresentam gráficos que comparam, respectivamente, o DRMCC das heurísticas e de suas combinações com o algoritmo ACM puro, o PRM e o TEM obtidos por estes algoritmos, em função da quantidade de salas nas instâncias. No gráfico da figura 9.12 notamos que a melhoria, dada em termos do DRMCC, realizada

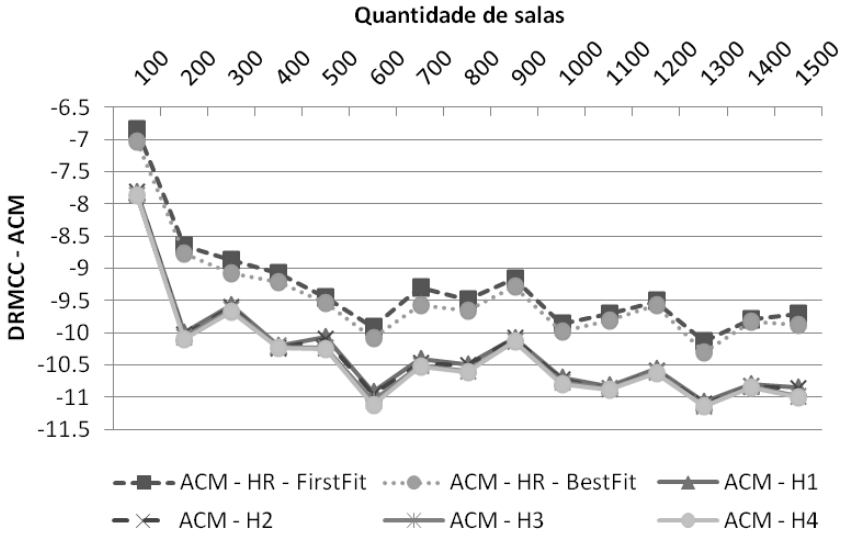


Figura 9.12: Comparação do DRMCC do algoritmo ACM utilizando as combinações de heurísticas selecionadas com o algoritmo ACM puro em função da quantidade de salas.

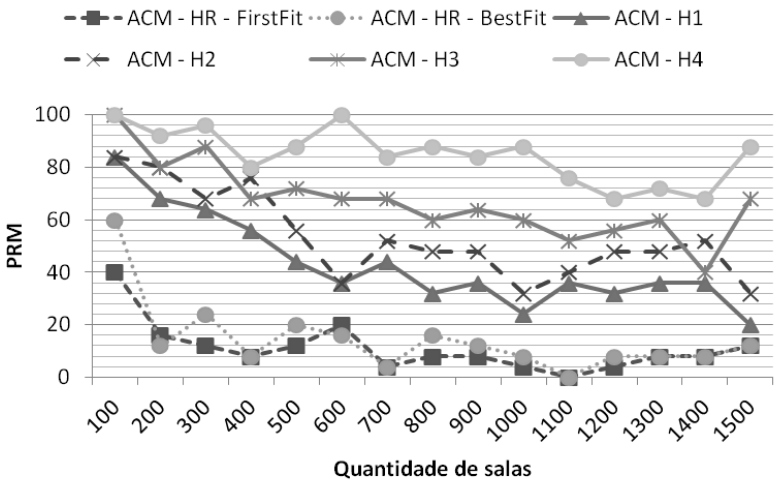


Figura 9.13: Comparação do PRM do algoritmo ACM sem utilizar e utilizando as combinações de heurísticas selecionadas em função da quantidade de salas.

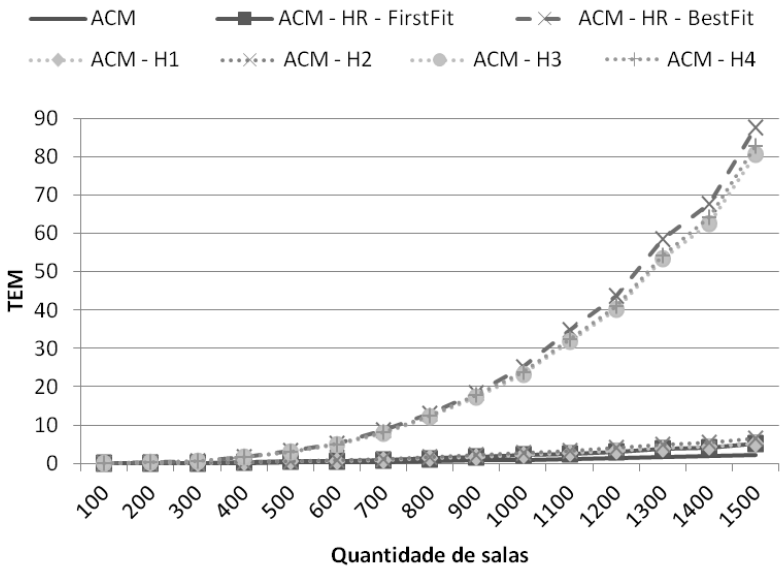


Figura 9.14: Comparação do TEM do algoritmo ACM sem utilizar e utilizando as combinações de heurísticas selecionadas em função da quantidade de salas.

pelas heurísticas e suas combinações tende a aumentar na medida em que a quantidade de salas aumenta, variando, em média, entre 6,83% e 10,3% nas heurísticas não combinadas, e entre 7,8% e 11,15% nas heurísticas combinadas. Além disso, a diferença entre o DRMCC das heurísticas combinadas e as heurísticas não combinadas varia entre 0,5% e 1,44% na média. Neste gráfico os algoritmos que fazem uso de alguma heurística combinada apresentam valores de DRMCC muito próximos. Desta forma mais uma vez recorremos ao PRM, apresentado na figura 9.13, para compará-los. Os algoritmos ACM-H3 e ACM-H4 apresentam, na maioria das vezes o maior PRM. Em seguida vêm os algoritmos ACM-H1 e ACM-H2, e finalmente os algoritmos ACM-HR *BestFit* e ACM-HR *FirstFit*. Nota-se que os algoritmos ACM-H3 e ACM-H4 também são aqueles que tem o maior TEM (figura 9.14), ou seja, existe um compromisso entre o PRM e o TEM. Já os demais algoritmos, exceto o ACM-HR *BestFit*, possuem TEM relativamente próximo. No caso do ACM-HR *BestFit*, observa-se que este gastou mais tempo que as heurísticas combinadas. Isto foi devido ao fato da combinação da heurística HC′ com HR′ diminuir o número de iterações de HR′. Ainda no gráfico da figura 9.14, observa-se que o algoritmo aproximativo ACM combinado com as heurísticas é de rápida execução, pois mesmo para instâncias com 1500 salas, sua versão mais lenta gastou em média pouco menos de 90 segundos para ser executada.

9.2.2 Análise da Heurística GRASP com *Evolutionary Path Relinking*

Como visto na seção 7.2, vamos utilizar combinações das heurísticas de melhoria como rotinas de busca local tanto no GRASP, quanto nos métodos de *path relinking*. Com base no estudo feito na seção anterior, que mostra o comportamento das combinações destas heurísticas aplicadas nas soluções do algoritmo aproximativo ACM, optamos por adotar a combinação H2, pelo fato desta ser capaz de obter bons resultados com um baixo consumo de tempo. Esta mesma rotina será utilizada como busca local em todas as etapas desta heurística. Contudo, existe uma diferença na execução dela junto ao PRA (*path relinking* pelas arestas). Como este método tende a obter diversas soluções, que se assemelham muito de uma iteração para outra, para cada solução visitada executamos apenas a heurística HD. Após isso, somente se ela for melhor que as soluções origem e destino, aplicamos a heurística combinada H2.

Com base em testes preliminares, ajustamos o GRASP para executar 200 iterações. Uma vez que esta heurística apresenta várias componentes aleatórias, seu resultado final depende da semente fornecida ao gerador aleatório antes de sua execução. Desta forma, a heurística foi rodada 5 vezes, com diferentes sementes, em cada uma das 375 instâncias.

A primeira análise feita investiga a eficiência dos métodos de construção do GRASP, em termos da obtenção do melhor resultado. O gráfico da figura 9.15 apresenta o percentual de execuções que cada um dos algoritmos, AAGM e ACM aleatorizados, em conjunto com a busca local, encontrou a melhor solução obtida pelo GRASP, em função da quantidade de salas. Na medida em que a quantidade de salas aumenta, o algoritmo AAGM tende a obter a maior quantidade de melhores soluções. Nas instâncias com mais de 500 salas este algoritmo obtém mais de 80% das melhores soluções. Todavia, cabe ressaltar que o algoritmo ACM ainda contribui para o método encontrando de 5% a 20% das melhores soluções.

De forma análoga à análise anterior, também investigamos qual das etapas da heurística é aquela que obtém o melhor resultado. Para isso, considere o gráfico da figura 9.16. Neste gráfico temos o percentual de execuções que nenhum (GRASP), ambos (GRASP+PRA+PRI), ou apenas um dos métodos de *evolutionary path relinking* encontrou uma solução melhor que a melhor solução encontrada antes de sua execução. Deste gráfico nota-se que as etapas do *evolutionary path relinking* são fundamentais na heurística, encontrando cada vez mais as melhores soluções na medida em que o número de salas aumenta. A partir de 500 salas, mais de 85% das melhores soluções são encontradas por algum dos métodos de *evolutionary path relinking*. A quantidade de execuções onde ambos os métodos de *evolutionary path relinking* encontram uma solução melhor aumenta de acordo com o crescimento da quantidade de salas, ultrapassando a quantidade onde apenas um dos

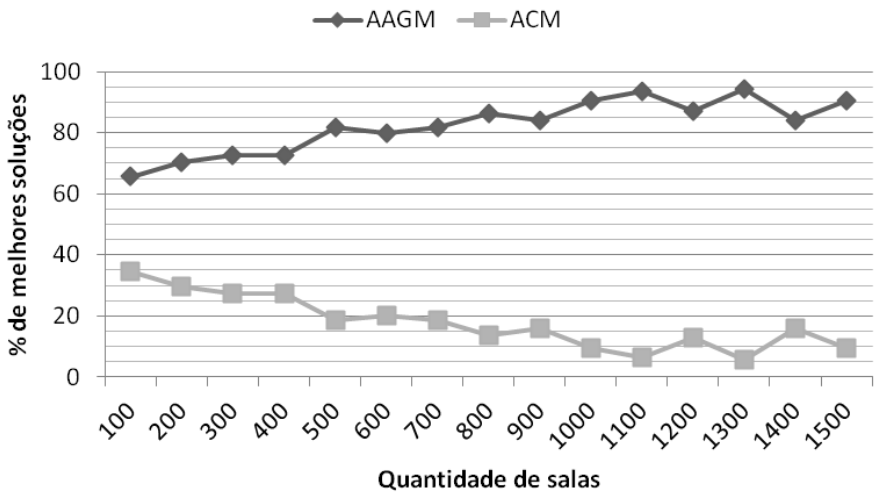


Figura 9.15: Comparação do percentual de execuções em que cada um dos algoritmos, empregados na fase de construção do GRASP, encontrou a melhor solução, em função da quantidade de salas.

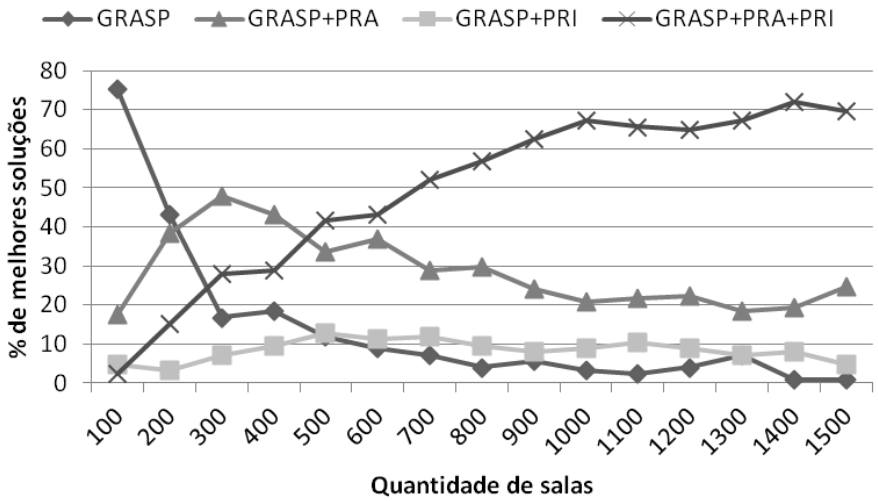


Figura 9.16: Comparação do percentual de execuções que nenhum (GRASP), ambos (GRASP+PRA+PRI), ou apenas um dos métodos de *path relinking* encontrou uma solução melhor que a melhor solução encontrada antes de sua execução, em função da quantidade de salas.

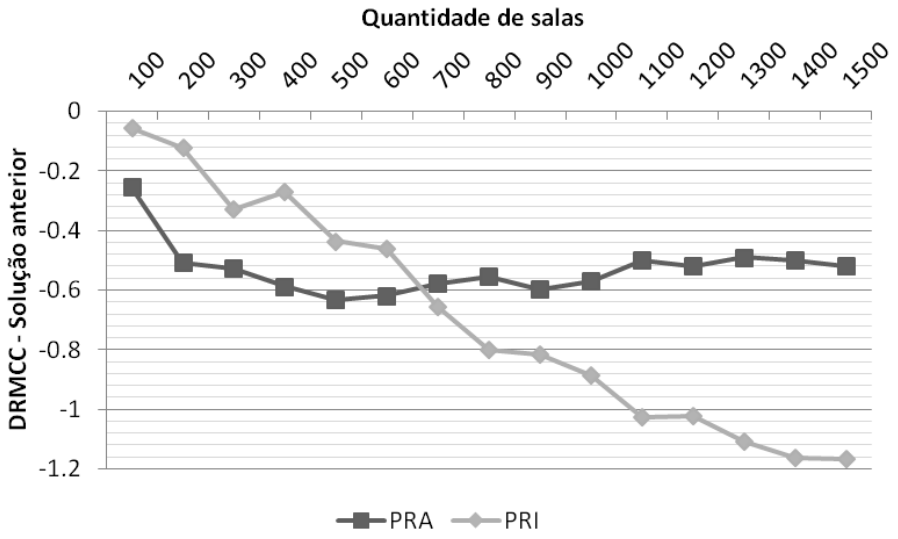


Figura 9.17: Comparação do DRMCC da solução encontrada pelo método com a melhor solução encontrada antes da execução do método em função da quantidade de salas.

métodos encontra uma solução melhor. Neste gráfico também observamos que pode existir uma dependência do PRI em relação ao PRA, pois o PRI sozinho melhora poucas vezes a solução, sendo no máximo em 15% da execuções.

Da análise anterior, sabemos que os métodos de *path relinking* encontram soluções melhores, mas ainda não sabemos em quanto elas são melhoradas. Desta forma, apresentamos no gráfico da figura 9.17 o DRMCC da solução encontrada pelo método para a melhor solução encontrada antes da execução do método. Em termos do DRMCC, o PRA consegue obter soluções melhores em até 0,65% na média, e o PRI em até 1,2% na média.

O tempo de execução gasto pela heurística é dado em função da quantidade de salas no gráfico mostrado na figura 9.18. Deste gráfico observa-se que o tempo de execução tende a aumentar gradativamente com o aumento de salas nas instâncias, mas mesmo para instâncias grandes, possuindo 1500 salas, a heurística gastou menos de 12 minutos para ser executada. A distribuição do tempo gasto em cada uma das etapas da heurística é dada pelo gráfico na figura 9.19. Neste gráfico observa-se que o GRASP consome a maior parte do tempo de execução da heurística, correspondendo, em média, a pelo menos 68% do total. Notamos também que o percentual de tempo consumido pelo PRI, apesar de ser baixo, é em média cerca de 5% a 20%, e tende a aumentar com a quantidade de salas, diminuindo o percentual de tempo consumido pelo GRASP. O PRA tende a manter um percentual que na média fica próximo a 15% do tempo total gasto pela heurística.

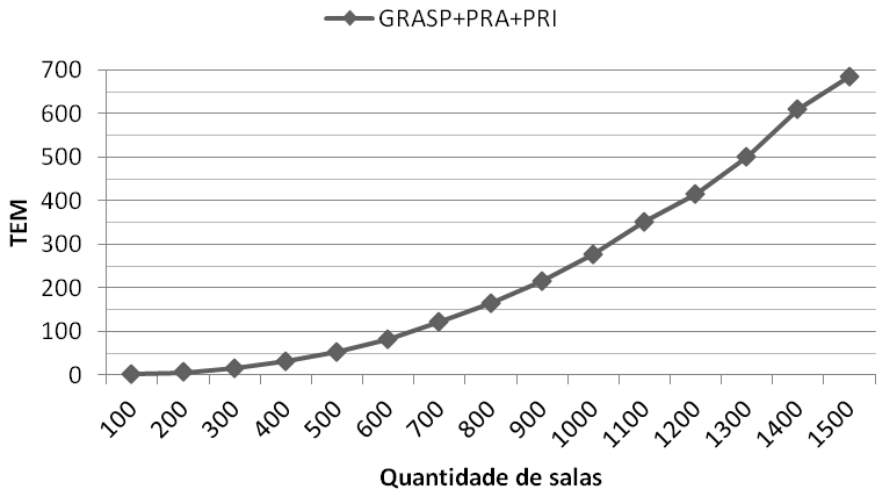


Figura 9.18: Tempo médio gasto na execução da heurística em função da quantidade de salas.

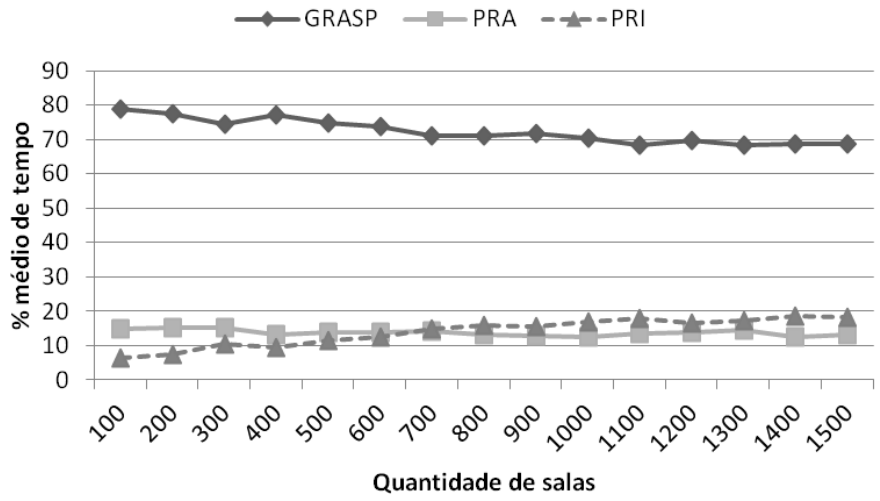


Figura 9.19: Percentual do tempo total gasto em cada uma das etapas da heurística em função da quantidade de salas.

9.2.3 Análise do Algoritmo Exato

Uma vez que o algoritmo *branch-and-cut* pode levar muito tempo para encontrar o ótimo de uma instância, limitamos a sua execução a no máximo duas horas. Caso ele atinja este limite temporal e termine sem encontrar o ótimo, consideramos a melhor solução (limitante primal) e a comparamos com o melhor limitante dual obtido para saber o quanto podemos estar distantes do ótimo. Assim como na seção anterior, na heurística primal do algoritmo *branch-and-cut* também adotamos como busca local a combinação H2.

O *branch-and-cut* foi capaz de encontrar soluções ótimas para grande parte do conjunto de instâncias, não resolvendo na otimalidade apenas 22 delas. Contudo, o maior *gap*[1] obtido nestas instâncias pelo algoritmo foi de 0,81%. Isso mostra que mesmo quando o algoritmo não é capaz de provar a otimalidade, os valores dos limitantes obtidos por ele, se não forem ótimos, estão muito próximos dele.

Nos testes realizados, observamos que o algoritmo exato se comporta de modo diferente dependendo do tipo da instância (ver seção 9.1) em que ele é aplicado. Notamos que as instâncias do tipo 4 apresentam um grau de dificuldade maior que as demais neste método, sendo que todas as instâncias não resolvidas na otimalidade são desse tipo. Então, nesta análise, diferentemente das anteriores, se faz necessário separar as instâncias por tipo por causa da discrepância dos valores obtidos. Isso fica mais evidente na figura 9.20, que apresenta o tempo de execução médio em função da quantidade de salas, considerando separadamente as instâncias do tipo 4 e as demais instâncias. Através deste gráfico nota-se que a dificuldade na resolução das instâncias do tipo 4 fica evidente quando elas passam a ter 600 ou mais salas, onde o tempo de execução médio passa a ser bem maior. As instâncias dos demais tipos são resolvidas de forma rápida, sendo que mesmo para instâncias grandes, em termos da quantidade de salas, são gastos por volta de 7 minutos em média. Ainda neste gráfico, nota-se também que o tempo de execução médio nem sempre é não-decrescente em função da quantidade de salas. Isso ocorre devido à geração aleatória das instâncias e à pouca quantidade de instâncias consideradas para o cálculo das médias (cinco instâncias). Todavia, ainda percebe-se a tendência de crescimento. O mesmo comportamento é visto no gráfico da figura 9.21, onde se aplica a mesma justificativa.

Finalizamos a análise do algoritmo *branch-and-cut* apresentando um comportamento interessante, que foi sua capacidade de resolver instâncias enumerando apenas um nó. Pouco mais de 60% das instâncias foram resolvidas desta forma. Isto quer dizer que, para estas instâncias, o modelo $\mathcal{M}_{AO}^{R}$ obteve uma solução inteira ou, então, a diferença do limitante primal para o limitante dual foi inferior a uma unidade. Nesta última situação, como trabalhamos com custos inteiros, podemos afirmar que o valor do limitante primal

[1] O gap corresponde ao desvio do valor do limitante primal para o valor do limitante dual.

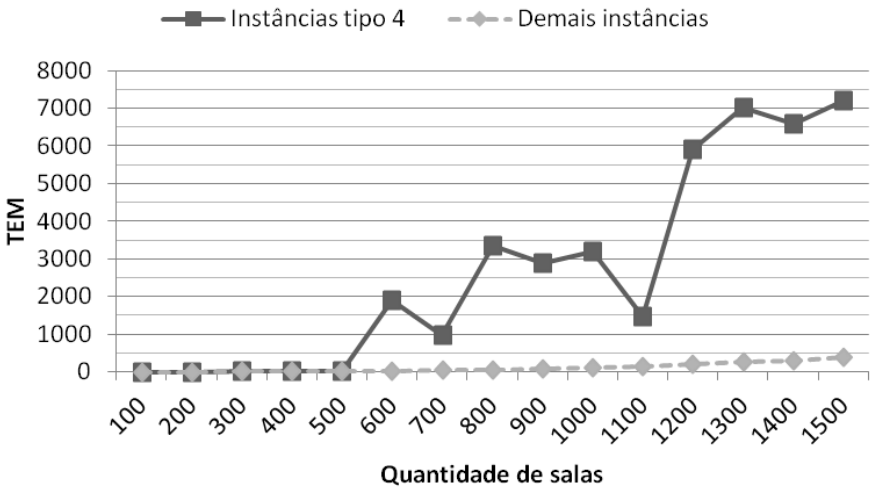


Figura 9.20: Tempo de execução médio do algoritmo *branch-and-cut*, considerando separadamente as instâncias do tipo 4 e as demais instâncias, em função da quantidade de salas.

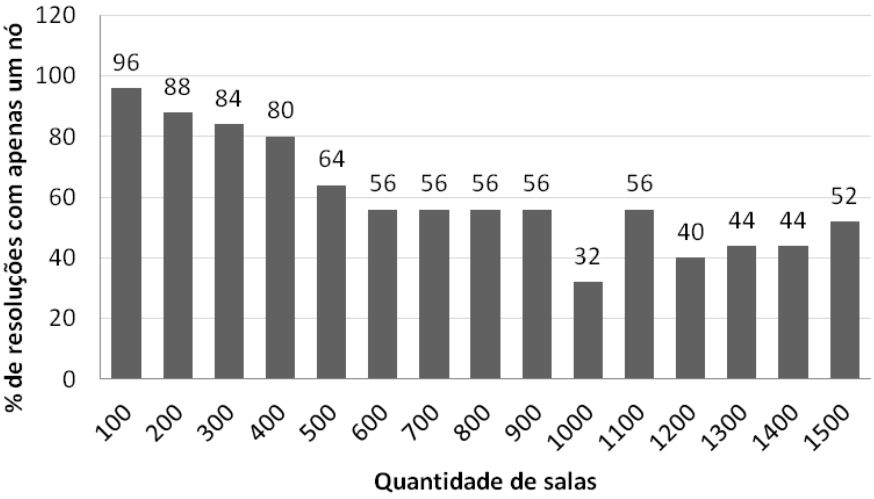


Figura 9.21: Percentual de instâncias que foram resolvidas enumerando apenas um nó no algoritmo *branch-and-cut* em função da quantidade de salas.

é ótimo. A rigor a resolução destas instâncias tomou um tempo polinomial já que a otimização do modelo $\mathcal{M}_{AO}^{R}$ gasta tempo polinomial. O gráfico da figura 9.21 apresenta como esse percentual de instâncias resolvidas no primeiro nó da árvore de enumeração fica distribuído em função da quantidade de salas. Observe que o percentual é elevado não apenas nas instâncias com poucas salas, uma vez que uma grande quantidade de instâncias com muitas salas foram resolvidas desta forma. Estes resultados confirmam na prática que este modelo é uma formulação forte para o PCCM, sendo capaz de fornecer excelentes limitantes duais.

9.2.4 Comparação entre os Algoritmos Exato, Aproximativos e Heurístico

Nesta seção vamos realizar análises comparando o algoritmo exato de *branch-and-cut* (B&C), o algoritmo aproximativo ACM com e sem as quatro heurísticas combinadas, e a heurística GRASP com *evolutionary path relinking* (GRASP+EPR).

Agora que conhecemos as soluções ótimas das instâncias, a exceção de 22 delas, nas quais ainda temos um limitante dual bem próximo do valor ótimo, podemos avaliar a qualidade das soluções obtidas pelos algoritmos aproximativos e a heurística GRASP+EPR. No gráfico da figura 9.22 apresentamos o DRMCC destes métodos relativos ao valor do melhor limitante dual encontrado pelo *branch-and-cut*. Dentre os algoritmos aproximativos que utilizam alguma heurística, apresentamos apenas o DRMCC do algoritmo ACM - H4, que obteve os melhores resultados. Os demais algoritmos combinados possuem DRMCC muito próximos, sendo no máximo 0,15% maior que o valor apresentado pelo ACM - H4, portanto, não haveria uma diferença visualmente perceptível na figura.

Neste gráfico, primeiramente observamos que ao utilizar alguma das heurísticas de melhoria em conjunto com o algoritmo aproximativo ACM, o DRMCC deste, que ficava entre 10% e 17%, passa a ficar em menos de 3,5%. Ou seja, a utilização de alguma das heurísticas de melhoria em conjunto com o algoritmo aproximativo ACM produz um impacto significativo na qualidade das soluções obtidas. Em função da quantidade de salas, o decréscimo no DRMCC varia de 8,8% a 12,97%. Nota-se também que as soluções obtidas pela heurística GRASP+EPR são muito próximas das soluções ótimas, tendo um DRMCC para o limitante dual de no máximo 0,9%.

O algoritmo aproximativo usando alguma heurística combinada encontrou o ótimo em apenas duas instâncias com 100 salas e uma instância com 200 salas, sendo que nas demais não foram encontrados os ótimos. Já o GRASP+EPR encontrou uma quantidade maior de ótimos. O gráfico dado na figura 9.23 apresenta o percentual de execuções onde o GRASP+EPR encontrou uma solução ótima em função da quantidade de salas. Este percentual diminui rapidamente em função da quantidade de salas, muito embora ele seja

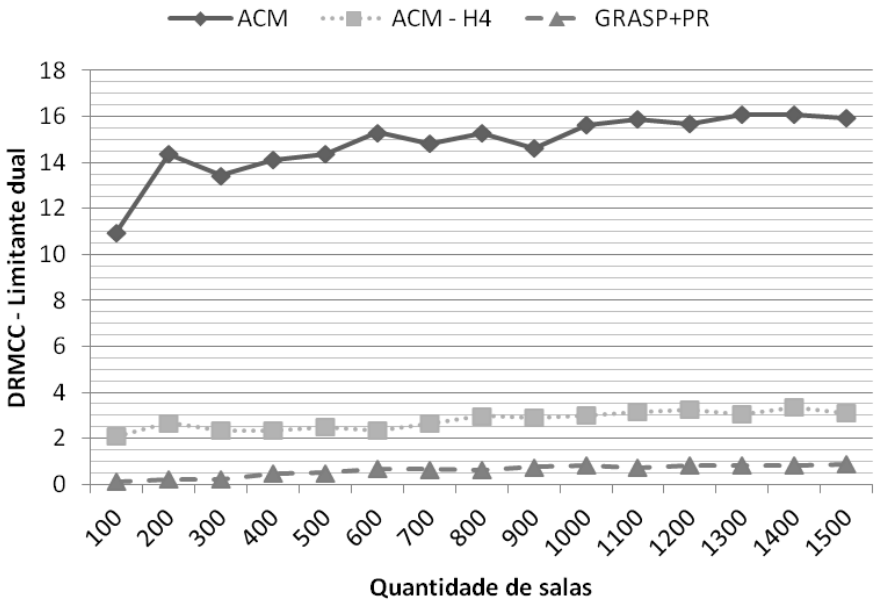


Figura 9.22: Comparação do DRMCC dos algoritmos aproximativos ACM com e sem heurísticas e da heurística GRASP+EPR com o melhor limitante dual obtido pelo *branch-and-cut* em função da quantidade de salas.

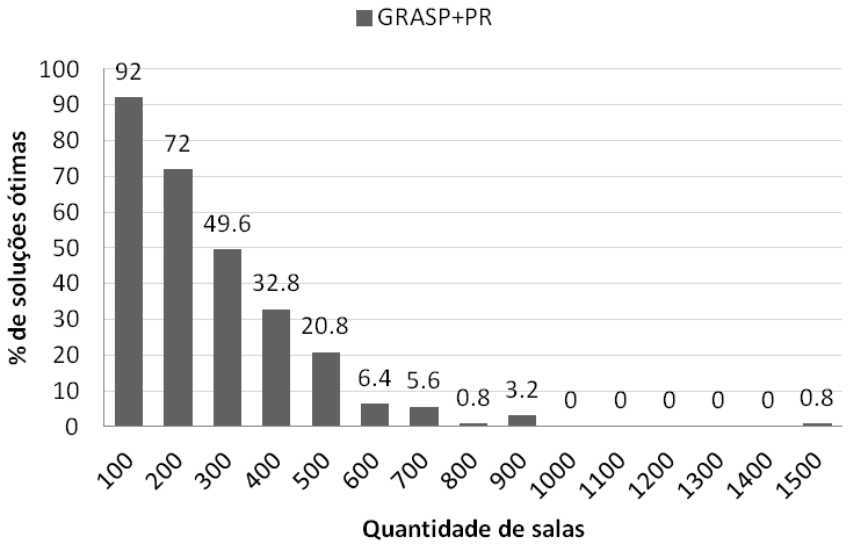


Figura 9.23: Percentual de execuções onde a heurística GRASP+EPR encontrou o ótimo em função da quantidade de salas.

considerável em instâncias com até 500 salas. Ainda assim, o GRASP+EPR encontrou uma solução ótima em uma execução de uma instância com 1500 salas.

Sabemos que o algoritmo ACM possui fator de aproximação teórico na ordem de $k-1$, onde k corresponde a quantidade de salas na instância. Voltando ao gráfico da figura 9.22, é possível estimar o fator de aproximação do algoritmo ACM atingido experimentalmente em função da quantidade de salas nas instâncias, bastando apenas dividir o valor no gráfico por 100 e somar um ao valor obtido. Portanto, o fator médio de aproximação do algoritmo ACM, na prática, para as instâncias testadas é no máximo 1,17. Quando as heurísticas são aplicadas o fator de aproximação médio fica abaixo de 1,035. Isso mostra que os fatores de aproximação destes algoritmos são muito bons na prática.

Na análise do algoritmo *branch-and-cut*, vimos que este gasta muito mais tempo na resolução das instâncias do tipo 4, comparado ao tempo gasto na resolução das demais instâncias. Por esse motivo, decidimos avaliar separadamente o TEM dos métodos, através dos gráficos apresentados nas figuras 9.24 e 9.25. Estes gráficos mostram o TEM dos algoritmos que estamos comparando em função da quantidade de salas nas instâncias. Achamos desnecessário incluir todos os algoritmos aproximativos, já que alguns destes não apresentam diferenças significativas nestes gráficos, desta forma incluímos apenas os algoritmos aproximativos ACM e ACM - H4, que correspondem ao método mais rápido e ao mais lento dentre os algoritmos aproximativos escolhidos.

Analisando o TEM nas instâncias que não são do tipo 4 (figura 9.24), nota-se que a heurística GRASP+EPR gasta em média mais tempo que o método *branch-and-cut*. Por isso, para este tipo de instâncias não faz sentido utilizar a heurística GRASP+EPR, pois gastamos mais tempo obtendo uma solução heurística, sendo que é possível obter em menos tempo uma solução ótima. Ainda nesta análise, observamos que a diferença do TEM entre o método *branch-and-cut* e os algoritmos aproximativos começa a apresentar uma diferença significativa quando as instâncias possuem 600 ou mais salas. Para instâncias menores, julgamos que o método *branch-and-cut* ainda é a melhor opção. E neste contexto, caso o tempo de execução não seja muito limitado, podemos considerar o método *branch-and-cut* até mesmo na resolução de instâncias com 1500 salas, onde ele gasta pouco menos de 7 minutos para ser executado.

Quando consideramos nesta análise as instâncias do tipo 4 (figura 9.25), ainda julgamos o método *branch-and-cut* como melhor opção na resolução de instâncias com até 500 salas, pois o TEM deste para os demais métodos é relativamente próximo. Já nas instâncias com 600 ou mais salas, a heurística GRASP+EPR e os algoritmos aproximativos consomem em média muito menos tempo que o método *branch-and-cut*. Então, nas situações onde o tempo de execução não é muito crítico, consideramos a heurística GRASP+EPR como melhor opção, pois esta gastou em média menos de 8 minutos para resolver instâncias com até 1500 salas, e conseguiu soluções melhores que as soluções obti-

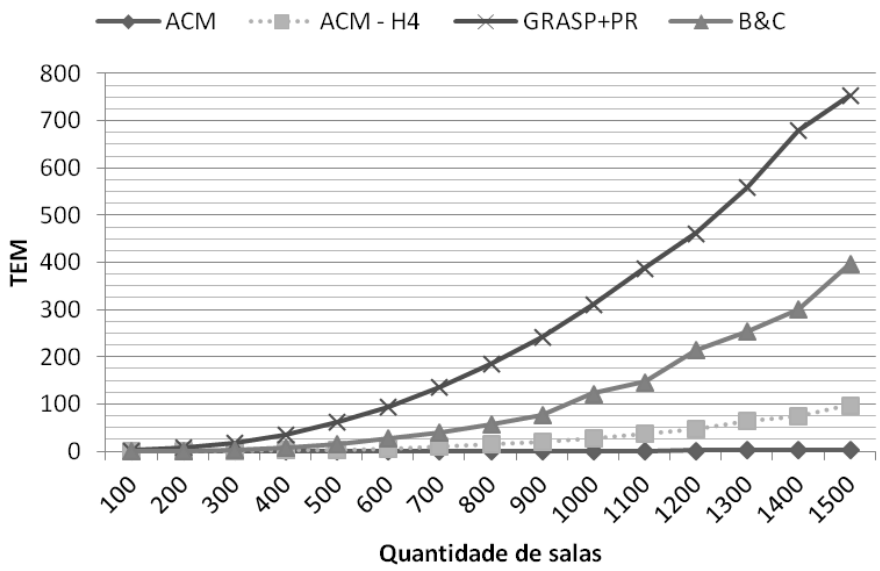


Figura 9.24: Comparação do tempo de execução médio dos algoritmos exato, aproximativo e heurístico em função da quantidade de salas (nas instâncias que não são do tipo 4).

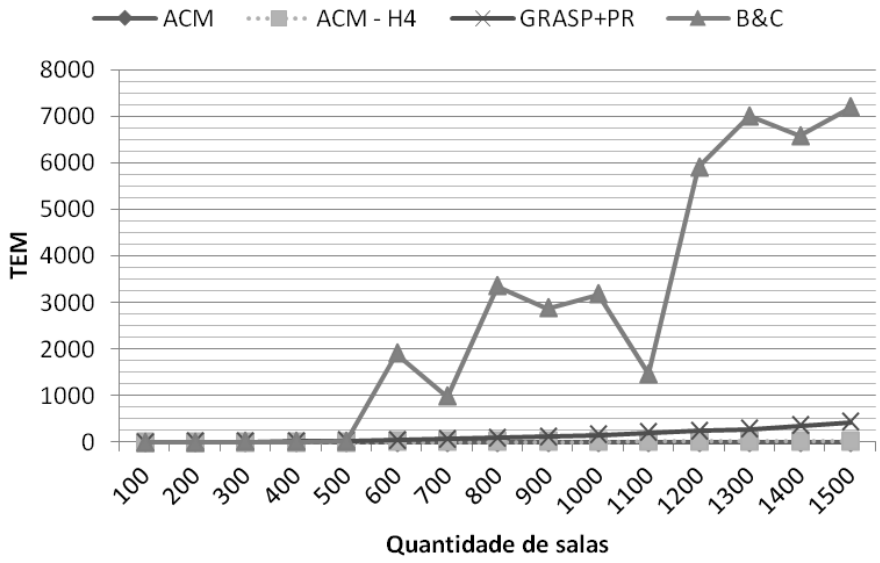


Figura 9.25: Comparação do tempo de execução médio dos algoritmos exato, aproximativo e heurístico em função da quantidade de salas (nas instâncias do tipo 4).

das pelos algoritmos aproximativos. Mas, quando o tempo de execução possuir um limite apertado, os algoritmos aproximativos passam a ser as melhores opções, pois o algoritmo ACM gasta em média menos de dois segundos para resolver instâncias com 1500 salas, e o algoritmo ACM - H4 gasta em média menos de 25 segundos.

Capítulo 10

Conclusões

Neste trabalho, estudamos o problema denominado problema do corredor de comprimento mínimo (PCCM). No PCCM recebemos um polígono retilinear subdividido em polígonos retilineares menores formando uma subdivisão S planar conexa do polígono dado. Uma solução para este problema, também chamada de corredor, é formada por um conjunto conexo de arestas de S, tal que cada face interna em S possui pelo menos um ponto em sua borda que pertence a alguma aresta deste conjunto. O objetivo então é encontrar uma solução tal que a soma total dos comprimentos das arestas seja a menor possível.

O PCCM pode ser reduzido polinomialmente a um problema em grafos denominado problema da árvore de Steiner com grupos (PASG). Através desta redução podemos aplicar os algoritmos do PASG conhecidos na literatura. Assim, foram estudados dois modelos matemáticos do PASG, que servem de base para um algoritmo aproximativo estudado e para o método exato *branch-and-cut* desenvolvido. Também estudamos outro algoritmo aproximativo baseado na construção de uma solução através de caminhos mínimos. Além destes algoritmos, desenvolvemos três heurísticas de busca local que recebem uma solução inicial e tentam encontrar soluções melhores. A partir destas heurísticas e dos algoritmos aproximativos estudados, propomos ainda uma heurística baseada na metaheurística GRASP combinada com um *evolutionary path relinking* (GRASP+EPR) que faz uso de dois diferentes métodos de *path relinking*.

Como não foram encontradas instâncias do PCCM disponíveis na literatura, desenvolvemos um gerador aleatório de instâncias para o problema. Com isto, criamos um conjunto com 375 instâncias, nas quais aplicamos os métodos estudados e desenvolvidos.

Feito isso, destacamos as principais contribuições desta dissertação:

- Propomos as primeiras heurísticas para o problema.
- Desenvolvemos a primeira abordagem exata utilizando programação linear inteira para o problema.

- Dois novos resultados teóricos foram obtidos. Apresentamos uma forma fácil de reduzir o PCCM-RE ao PCCM. Também mostramos que o fator de aproximação do algoritmo ACM, que é $k-1$, chega a pelo menos $k-2-\mu$ nas instâncias do PCCM, onde k é a quantidade de salas na instância e μ é uma constante muito pequena. Por isso, este fator não pode ser melhorado assintoticamente.

- Geramos o primeiro *benchmark* de instâncias para o problema, disponibilizado em `http://www.ic.unicamp.br/~cid/Problem-instances/MinCorridor`.

- Realizamos os primeiros experimentos computacionais comparando algoritmos exatos, aproximativos e heurísticos, fornecendo resultados para o *benchmark*.

- Apesar do foco ter sido o PCCM, todos os algoritmos desenvolvidos não se limitam a ele, sendo aplicáveis também ao PASG.

Das análises dos resultados obtidos nos experimentos computacionais chegamos às seguintes conclusões. Primeiramente, observamos que é viável resolver na otimalidade de forma rápida, empregando o método *branch-and-cut*, instâncias com até 500 salas. Se considerarmos que uma hora é um limite de tempo computacional aceitável, levando-se em conta a tecnologia atual, então podemos cogitar a resolução exata de instâncias com até 1100 salas. A análise em separado dos resultados do método *branch-and-cut*, com exceção das instâncias tipo 4, mostra que o método é capaz de resolver, em pouco tempo, instâncias ainda maiores, uma vez que em nossos experimentos foram resolvidas instâncias com 1500 salas gastando em média cerca de 7 minutos. Ao analisar a forma como geramos as instâncias do tipo 4, acreditamos que o fator complicador para a resolução das mesmas se deve à grande quantidade de arestas em S possuindo comprimentos iguais, ou muito próximos. Isso pode fazer com que a instância tenha muitas soluções ótimas diferentes, dificultando a busca por bons limitantes duais durante o *branch-and-cut*. Portanto, sugerimos que o método *branch-and-cut* deve ser avaliado como forma de resolução sempre que a instância não apresentar esta característica.

Para as instâncias difíceis de serem solucionadas de forma exata, pode-se utilizar a heurística GRASP+EPR e obter soluções bem próximas das soluções ótimas, uma vez que nos experimentos que fizemos, esta levou a um gap médio de no máximo 0,9% para o ótimo. Já o algoritmo aproximativo ACM, com ou sem heurística de melhoria, deve ser utilizado em aplicações onde o tempo é um recurso muito limitado, e é aceitável obter soluções com uma qualidade levemente inferior. Cabe observar que este algoritmo, quando utilizado com alguma das heurísticas de melhoria, foi capaz de obter soluções bastante satisfatórias nos testes realizados, com um gap médio de no máximo 3,5% para o ótimo.

Ainda destacamos os seguintes resultados obtidos em nossos experimentos:

- O algoritmo *branch-and-cut* foi capaz de encontrar o ótimo em 353 instâncias. Apenas 22 instâncias do tipo 4 não foram resolvidas na otimalidade, mas o maior gap obtido pelo algoritmo nestas instâncias foi de 0,81%.

- Em pouco mais de 60% das instâncias, o *branch-and-cut* encontrou a solução ótima resolvendo apenas um nó. Isso indica experimentalmente que a formulação do modelo adotado é forte para o PCCM.

- A heurística GRASP+EPR foi mais lenta que o *branch-and-cut* na resolução das instâncias que não são do tipo 4. Contudo, a heurística gastou no máximo 12 minutos em média para resolver cada instância, o que a torna interessante quando as instâncias são do tipo 4, onde o *branch-and-cut* na média leva mais tempo em instâncias com 600 ou mais salas.

- Os dois métodos *path relinking* incorporados na heurística GRASP melhoram as soluções em até 1,2% na média, em termos do DRMCC.

- O algoritmo aproximativo AAGM não é um bom algoritmo na prática. Ele consome muito tempo resolvendo um problema de programação linear. Além disso, o algoritmo ACM encontra soluções aproximadas melhores em um tempo muito menor e, em alguns casos, também é possível encontrar uma solução ótima em muito menos tempo.

- Destacamos quatro combinações das heurísticas apresentadas no capítulo 6. Estas heurísticas combinadas, quando aplicadas nas soluções obtidas pelo algoritmo ACM, melhoram substancialmente a qualidade das soluções. Ao utilizá-las, o DRMCC das soluções para o valor ótimo cai em média de 8,8% a 12,97%.

- O algoritmo ACM, utilizando a combinação de heurísticas mais lenta, ainda tem um tempo de execução relativamente pequeno, gastando em média pouco menos de 90 segundos para resolver instâncias com 1500 salas. Já o algoritmo ACM sem utilizar nenhuma heurística gasta em média pouco menos de 2 segundos para resolver as mesmas instâncias.

- O fator de aproximação do algoritmo ACM na prática foi no máximo 1,17 na média. Portanto, muito menor que o fator teórico de $k-1$, onde k corresponde a quantidade de salas na instância.

Finalizamos esta dissertação sugerindo um trabalho futuro que acreditamos que ser capaz de melhorar os resultados apresentados aqui. Encontram-se na literatura técnicas de pré-processamento para reduzir a quantidade de vértices e arestas nas instâncias do

problema da árvore de Steiner e também nas instâncias do problema da árvore de Steiner com grupos [19, 30]. Estas técnicas poderiam ser aplicadas às instâncias do PCCM, potencialmente levando a melhorias no tempo de execução dos métodos e, até mesmo, na qualidade das soluções obtidas, pois o espaço de busca será reduzido.

Referências Bibliográficas

[1] D. V. Andrade e M. G. C. Resende. GRASP with evolutionary path-relinking. Relatório Técnico TD-6XPTS7, AT&T Labs Research, Florham Park, NJ, USA, 2007.

[2] S. Arora. Nearly linear time approximation schemes for Euclidean TSP and other geometric problems. In *Proceedings of the 38th Annual Symposium on Foundations of Computer Science*, FOCS '97, pages 554–563, Washington, DC, USA, 1997. IEEE Computer Society.

[3] S. Arora. Approximation schemes for NP-hard geometric optimization problems: A survey. *Mathematical Programming*, 97(1-2):43–69, 2003.

[4] V. Bafna, P. Berman, e T. Fujito. Constant ratio approximations of the weighted feedback vertex set problem for undirected graphs. In *ISAAC '95: Proceedings of the 6th International Symposium on Algorithms and Computation*, pages 142–151, London, UK, 1995. Springer-Verlag.

[5] C. D. Bateman, C. S. Helvig, G. Robinsl, e A. Zelikovsky. Provably good routing tree construction with multi-port terminals. In *ISPD '97: Proceedings of the 1997 international symposium on physical design*, pages 96–102, New York, NY, USA, 1997. ACM.

[6] A. Becker e D. Geiger. Approximation algorithms for the loop cutset problem. In *Proceedings of the 10th. Conference on Uncertainty in Artificial Intelligence*, pages 60–68, San Francisco, CA, 1994.

[7] T. Berthold. Primal heuristics for mixed integer programs. Dissertação de mestrado, Technische Universität Berlin, 2006.

[8] D. Bertsimas e J. Tsitsiklis. *Introduction to linear optimization*. Athena Scientific, 1997.

[9] H. L. Bodlaender, C. Feremans, A. Grigoriev, E. Penninkx, R. Sitters, e T. Wolle. On the minimum corridor connection problem and other generalized geometric problems. *Lecture Notes in Computer Science*, 4368:69–82, 2007.

[10] H. L. Bodlaender, C. Feremans, A. Grigoriev, E. Penninkx, R. Sitters, e T. Wolle. On the minimum corridor connection problem and other generalized geometric problems. *Computational Geometry: Theory and Applications*, 42(9):939–951, November 2009.

[11] F. A. Chudak, M. X. Goemans, D. Hochbaum, e D. Williamson. A primal-dual interpretation of two 2-approximation algorithms for the feedback vertex set problem in undirected graphs. *Operations Research Letters*, 22:111–118, 1998.

[12] T. H. Cormen, C. E. Leiserson, R. L. Rivest, e C. Stein. *Introduction to algorithms*. The MIT Press, second edition, 2001.

[13] W. Cronholm, F. Ajili, e S. Panagiotidi. On the minimal Steiner tree subproblem and its application in branch-and-price. In Roman Barták e Michela Milano, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, volume 3524 of *Lecture Notes in Computer Science*, pages 816–817. Springer Berlin / Heidelberg, 2005.

[14] E. D. Demaine e J. O'Rourke. Open problems from CCCG 2000. In *Proceedings of the 13th Canadian Conference on Computational Geometry (CCCG 2001)*, pages 185–187, Waterloo, Ontario, Canada, August 2001.

[15] A. Dumitrescu e J. S. B. Mitchell. Approximation algorithms for TSP with neighborhoods in the plane. In *SODA '01: Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms*, pages 38–46, Philadelphia, PA, USA, 2001. Society for Industrial and Applied Mathematics.

[16] K. Elbassioni, A. Fishkin, N. H. Mustafa, e R. Sitters. Approximation algorithms for euclidean group TSP. In *Automata, Languages and Programming : 32nd International Colloquim, ICALP*, pages 1115–1126, Lisboa, Portugal, 2005.

[17] D. Eppstein. Some open problems in graph theory and computational geometry. `http://www.ics.uci.edu/~eppstein/200-f01.pdf`, 2001.

[18] T. A. Feo e M. G. C. Resende. A probabilistic heuristic for a computationally difficult set covering problem. *Operations Research Letters*, 8(2):67 – 71, 1989.

[19] C. E. Ferreira e F. M. Oliveira. New reduction techniques for the group Steiner tree problem. *SIAM Journal on Optimization*, 17(4):1176–1188, December 2006. ISSN 1052-6234.

[20] P. Festa e M. G. C. Resende. GRASP: basic components and enhancements. *Telecommunication Systems*, 46(3):253–271, 2011.

[21] P. Festa e M.G.C. Resende. GRASP: An annotated bibliography. In C.C. Ribeiro e P. Hansen, editors, *Essays and surveys in metaheuristics*, pages 325–367. Kluwer Academic Publishers, 2002.

[22] A. V. Goldberg e R. E. Tarjan. A new approach to the maximum flow problem. *Journal of the ACM*, 35:921–940, 1988.

[23] A. Gonzalez-Gutierrez. *Complexity of the minimum-length corridor problem*. Tese de doutorado, University of California, Santa Barbara, CA, USA, 2007.

[24] A. Gonzalez-Gutierrez e T. F. Gonzalez. Complexity of the minimum-length corridor problem. *Computational Geometry*, 37(2):72–103, 2007.

[25] M. Grötschel, L. Lovász, e A. Schrijver. *Geometric algorithms and combinatorial optimization*, volume 2 of *Algorithms and Combinatorics*. Springer, second corrected edition, 1993.

[26] J. Hao e J. B. Orlin. A faster algorithm for finding the minimum cut in a directed graph. *Journal Of Algorithms*, 17:424–446, 1994.

[27] C. S. Helvig, G. Robins, e A. Zelikovsky. An improved approximation scheme for the group Steiner problem. *Networks*, 37(1):8–20, 2001.

[28] E. Ihler. Bounds on the quality of approximate solutions to the group Steiner problem. In *WG '90: Proceedings of the 16th international workshop on Graph-theoretic concepts in computer science*, pages 109–118, New York, NY, USA, 1991. Springer-Verlag New York, Inc.

[29] D. Jungnickel. *Graphs, networks and algorithms*. Springer, third edition, 2008.

[30] J. H. Kingston e N. P. Sheppard. On reductions for the Steiner problem in graphs. *Journal of Discrete Algorithms*, 1(1):77 – 88, 2003.

[31] T. Koch e A. Martin. Solving Steiner tree problems in graphs to optimality. *Networks*, 32:207–232, 1998.

[32] J. E. Mitchell. Branch-and-cut algorithms for combinatorial optimization problems. In P.M. Pardalos e M.G.C. Resende, editors, *Handbook of Applied Optimization*, pages 65–77. Oxford University Press, 2002.

[33] J. E. Mitchell. Branch and cut. `http://www.rpi.edu/~mitchj/papers/EORMS_JEM_BnC.html`, 2010.

[34] G. L. Nemhauser e L. A. Wolsey. *Integer and combinatorial optimization*. Wiley-Interscience, New York, NY, USA, 1988.

[35] F. M. Oliveira. O problema de Steiner com grupos. Dissertação de mestrado, Instituto de Matemática e Estatística - USP, São Paulo, SP, Brasil, 2005.

[36] G. Reich e P. Widmayer. Beyond Steiner's problem: a VLSI oriented generalization. In *WG '89: Proceedings of the fifteenth international workshop on Graph-theoretic concepts in computer science*, pages 196–210, New York, NY, USA, 1990. Springer-Verlag New York, Inc.

[37] M. G. C. Resende. Greedy randomized adaptive search procedures. In Christodoulos A. Floudas e Panos M. Pardalos, editors, *Encyclopedia of Optimization*, pages 1460–1469. Springer, 2009.

[38] M. G. C. Resende e C. C. Ribeiro. Greedy randomized adaptive search procedures: Advances and applications. *Handbook of Metaheuristics*, pages 1–35, 2009.

[39] M.G.C. Resende e C.C. Ribeiro. GRASP and path-relinking: Recent advances and applications. In T. Ibaraki e Y. Yoshitomi, editors, *Proceedings of the Fifth Metaheuristics International Conference (MIC2003)*, pages T6–1 – T6–6, 2003.

[40] C. C. Ribeiro, E. Uchoa, e R. F. Werneck. A hybrid GRASP with perturbations for the Steiner problem in graphs. *INFORMS Journal on Computing*, 14:228–246, July 2002.

[41] S. Safra e O. Schwartz. On the complexity of approximating TSP with neighborhoods and related problems. In *Proc. of the 11th. Annual European Symposium on Algorithms*, pages 446–458, 2003.

[42] P. Slavik. The errand scheduling problem. Relatório técnico, Department of Computer Science and Engineering, University of New York, Buffalo, NY, USA, 1997.

[43] P. Slavik. *Approximation algorithms for set cover and related problems*. Tese de doutorado, University of New York, Buffalo, NY, USA, 1998.

[44] F. L. Usberti, P. M. França, e A. L. M. França. GRASP with evolutionary path-relinking for the capacitated arc routing problem. *Computers & Operations Research*, 2011.

[45] M. G. A. Verhoeven, M. E. M. Severens, e E. H. L. Aarts. Local search for Steiner trees in graphs. In R. J. Rayward-Smith, I. H. Osman, C. R. Reeves, e G. D. Smith, editors, *Modern Heuristics Search Methods*, pages 117–129. J. Wiley, 1996.

[46] J. G. Villegas, C. Prins, C. Prodhon, A. L. Medaglia, e N. Velasco. A GRASP with evolutionary path relinking for the truck and trailer routing problem. *Computers & Operations Research*, 2010.

[47] L. A. Wolsey. *Integer programming*. Wiley-Interscience, New York, NY, USA, 1998.