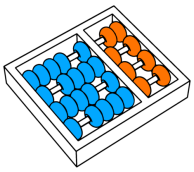


Igor Ribeiro de Assis

"O Problema do Recorte com Custo nas Conversões"

CAMPINAS
2013

Universidade Estadual de Campinas
Instituto de Computação

Igor Ribeiro de Assis

"O Problema do Recorte com Custo nas Conversões"

Orientador(a): **Cid Carvalho de Souza**

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Computação da Universidade Estadual de Campinas para obtenção do título de Mestre em Ciência da Computação.

Este exemplar corresponde à versão final da Dissertação defendida por Igor Ribeiro de Assis, sob orientação de Cid Carvalho de Souza.

Assinatura do Orientador(a)

CAMPINAS
2013

Ficha catalográfica
Universidade Estadual de Campinas
Biblioteca do Instituto de Matemática, Estatística e Computação Científica
Ana Regina Machado - CRB 8/5467

As76p

Assis, Igor Ribeiro de, 1987-
O problema do recorte com custo nas conversões / Igor Ribeiro de Assis. – Campinas, SP : [s.n.], 2013.

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

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

Informações para Biblioteca Digital

Título em outro idioma: Milling tour with turn costs
Palavras-chave em inglês:
Heuristic
Algorithms
Integer programming
Combinatorial optimization
Área de concentração: Ciência da Computação
Titulação: Mestre em Ciência da Computação
Banca examinadora:
Cid Carvalho de Souza [Orientador]
Marcus Vinicius Soledade Poggi de Aragão
Fábio Luiz Usberti
Data de defesa: 24-09-2013
Programa de Pós-Graduação: Ciência da Computação

TERMO DE APROVAÇÃO

Dissertação Defendida e Aprovada em 24 de setembro de 2013, pela Banca examinadora composta pelos Professores Doutores:

Prof. Dr. Marcus Vinicius Soledade Poggi de Aragão
DI / PUC-Rio

Prof. Dr. Fábio Luiz Usberti
IC / UNICAMP

Prof. Dr. Cid Carvalho de Souza
IC / UNICAMP

Instituto de Computação
Universidade Estadual de Campinas

O Problema do Recorte com Custo nas Conversões

Igor Ribeiro de Assis[1]

24 de setembro de 2013

Banca Examinadora:

- Cid Carvalho de Souza (*Orientador*)
- Marcus Vinicius Soledade Poggi de Aragão
  Departamento de Informática / PUC-Rio
- Fábio Luiz Usberti
  Instituto de Computação / UNICAMP
- Pedro Jussieu de Rezende
  Instituto de Computação / UNICAMP (Suplente)
- Cláudio Nogueira de Menezes
  Centro de Matemática, Computação e Cognição / UFABC (Suplente)

[1]Suporte financeiro de: Bolsa do CNPq 2010–2012

Resumo

No problema do recorte geométrico com custo nas conversões é dado um polígono ortogonal de entrada P que pode ou não conter buracos. Queremos encontrar uma curva poligonal fechada formada por segmentos verticais e horizontais que quando percorrida por um quadrado unitário, a área coberta é exatamente a de P. Custos são atribuídos às mudanças de direção e o objetivo é minimizar a soma dos custos. Aplicações desse problema incluem: máquina de controle numérico, inspeção automática e roteamento.

Esta dissertação estuda soluções para o problema do recorte. Propomos um modelo de programação linear inteira e a partir deste desenvolvemos um algoritmo exato. Descrevemos um algoritmo 3.75 aproximado da literatura e propomos algumas melhorias heurísticas para o mesmo. Abandonando as garantias teóricas, projetamos duas heurísticas: a primeira utiliza uma ideia gulosa bastante simples complementada por técnicas da meta-heurística GRASP, especificamente aleatorização e busca local; e a segunda resolve um problema de cobertura mais simples e cuja sua solução pode ser adaptada para o problema original. Por fim propomos uma série de vizinhanças executadas em uma fase de busca local, que dada uma solução, faz mudanças locais que reduzem o custo do *tour*.

As implementações dos algoritmos descritos são analisadas experimentalmente utilizando um *benchmark* de instâncias que construímos e deixamos público para comparações futuras.

Abstract

In the orthogonal milling with turn costs problem is given an orthogonal polygon P that may contain holes. Our goal is to find a closed polygonal curve made of horizontal and vertical segments which when traversed by a unit square, the covered area is exactly P. Turn costs are assigned to direction changes and the objective is to minimize the sum of turn costs. This problem arises in many applications including: numerically controlled (NC) machining, automatic inspection and routing.

This dissertation studies solutions for the milling problem. We propose an integer linear programming model and from which an exact algorithm is proposed. A 3.75-approximate algorithm from the literature is described for which heuristic improvements are proposed. Lifting the theoretical guarantees we project two heuristics: the first is based in a simple greedy idea supplemented with techniques of the GRASP meta-heuristic, specifically, randomization and local search; the latter solves a simpler covering problem whose its solution is adapted for the original problem. Finally, we propose a series of neighborhoods run in a local search step where, local changes are made to the solution such that the tour cost is reduced.

All described algorithms are implemented and evaluated experimentally using a benchmark of instances that we built and made public for future comparisons.

Agradecimentos

Primeiramente gostaria de agradecer ao meu orientador Cid C. de Souza, pelas ideias, dicas, sugestões e discussões que tiveram um papel fundamental na conclusão deste trabalho. Também agradeço por despertar meu interesse pela área de Otimização Combinatória, em particular pelas boas aulas, que foram motivantes e em alguns momentos desafiadoras.

Sou grato ao professor Orlando Lee pela ajuda quando nosso conhecimento sobre Teoria dos Grafos se mostrou insuficiente e ao Bruno Crepaldi que a dedicação e o interesse permitiu que colaborássemos no desenvolvimento de uma bela heurística para o problema.

Por fim, gostaria de agradecer a minha família e amigos que contribuíram indiretamente e incondicionalmente na realização deste trabalho.

Sumário

Lista de Tabelas

Lista de Figuras

Capítulo 1

O Problema do Recorte

Considere a seguinte situação. Um jardineiro utiliza um cortador de grama para cortar a grama de uma região cuja área é a de um quadrado de lado l. No gramado existem canteiros de flores, construções e fontes de água inacessíveis pelo cortador de grama. A maior parte do tempo gasto para cortar a grama é consumido nos momentos que o jardineiro precisa mudar a direção que está cortando e, portanto, é do seu interesse encontrar o caminho que ele deve percorrer de modo que o número de vezes que precisa mudar a direção é mínimo.

O leitor interessado em tentar resolver o problema do jardineiro interativamente pode acessar a página `www.ic.unicamp.br/~cid/Problem-instances/Milling/game/`, onde, para diversas entradas ele pode tentar encontrar a melhor forma da grama ser cortada, e ter seus resultados comparados com as melhores soluções conhecidas.

O problema do jardineiro que acabamos de descrever pode ser descrito de forma mais genérica como se segue. Um polígono ortogonal com buracos deve ser percorrido por uma ferramenta cuja área é a de um quadrado. O custo do percurso é dado pelo número de mudanças de direção que a ferramenta deve realizar ao segui-lo. Posto sob este formato, precebe-se que o problema é recorrente em diversas aplicações, conforme se vê pelos exemplos abaixo:

- **máquina de controle numérico:** na usinagem de peças temos uma placa metálica que precisa ser cortada por uma broca de modo a assumir uma forma específica. Em tais operações é comum que a máquina deve parar de se movimentar sempre que necessitar mudar a direção de corte e, portanto, procura-se minimizar o número de conversões de modo a reduzir o tempo total de usinagem.

- **pintura *spray*:** neste caso a ferramenta corresponde à cabeça do *spray* e a área a ser coberta é aquela que precisa ser pintada, os buracos e obstáculos são as regiões que não devem receber a tinta. Note que uma variação possível neste cenário poderia

considerar desligar o spray de tinta ao atravessar os buracos. Contudo esta variação não é estudada nessa dissertação.

- **inspeção automática:** a região a ser coberta corresponde àquela que precisa ser inspecionada por uma câmera (ou robô) e existem os obstáculos físicos onde a câmera não tem acesso.

Além destas aplicações, no artigo [6] e na revisão bibliográfica são apresentadas outras aplicações como: origami matemático, problemas de roteamento e de limpeza de regiões cobertas por neve.

Nessa dissertação serão estudadas soluções algorítmicas para o problema descrito acima, chamado de **Problema do Recorte Ortogonal com Custo das Conversões (MTWTC)** o qual sabe-se estar em $\mathcal{NP}$-difícil. As principais contribuições desse trabalho incluem: o desenvolvimento de um algoritmo exato e de heurísticas para o problema, além de suas respectivas implementações. Até onde foi possível verificar, esta é a primeira vez que métodos exatos e heurísticos foram propostos para o MTWTC. Além disso, também foi implementado o melhor algoritmo aproximado encontrado na literatura, e criado um *benchmark* público de instâncias, o que permitiu a realização de um intenso estudo experimental comparativo dos algoritmos em questão.

1.1 Organização do Texto

Essa dissertação está dividida em cinco capítulos. No primeiro capítulo apresentamos os fundamentos teóricos básicos principais para o entendimento do texto, que incluem *Teoria dos Grafos* e *Álgebra Linear*. O tratamento desses temas básicos tem como intuito tornar clara a nomenclatura que utilizaremos pois alguns termos podem estar definidos de maneiras diferentes na literatura. Em seguida, ainda no capítulo 1, enunciamos o problema formalmente, discutimos uma discretização do mesmo, um resultado com relação ao custo da solução e fazemos uma revisão bibliográfica.

No capítulo 2, após uma fundamentação teórica de Programação Linear e Programação Linear Inteira, propomos um modelo matemático para o problema e, a partir deste, desenvolvemos um algoritmo exato e de complexidade exponencial. Abandonando a obrigatoriedade de se obter soluções ótimas e levando em consideração os aspectos práticos e experimentais, apresentamos no capítulo 3 um algoritmo com fator de aproximação 3.75, desenvolvido originalmente por *Arkin et. al* [6]. Também são descritas modificações deste algoritmo. Ainda no capítulo 3, propomos duas heurísticas construtivas e uma heurística de busca local que utilizaremos para melhorar as soluções encontradas pelos algoritmos não-exatos descritos nessa dissertação.

Os capítulos 4 e 5 são compostos por uma análise experimental de todos os algoritmos descritos na dissertação e seu objetivo principal é avaliar e comparar os algoritmos levando em consideração a eficiência computacional e a qualidade das soluções obtidas. No capítulo 4 exibimos gráficos e tabelas relativos às execuções dos algoritmos assim como fornecemos uma descrição detalhada dos conjuntos de instâncias de teste e das implementações. Após a exposição dos dados obtidos experimentalmente, no capítulo 5 analisamos os mesmos e concluímos a dissertação resumindo os resultados nela alcançados. Em seguida fazemos sugestões de trabalhos futuros sobre o problema, que se mostrou bastante desafiador tanto do ponto de vista de teórico, como do ponto de vista experimental e heurístico.

1.2 Fundamentação Teórica

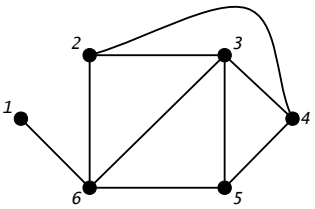
Nesta seção apresentamos a fundamentação teórica e definimos a nomenclatura básica utilizada no decorrer da dissertação. Outras definições e conceitos mais específicos serão apresentados ao longo do texto conforme necessário.

1.2.1 Teoria dos Grafos

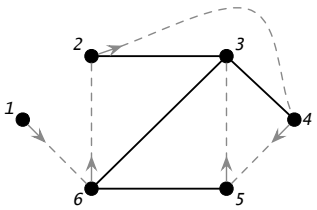
Um **grafo** é uma entidade abstrata que representa objetos e suas relações. Usualmente um grafo é denotado por $G = (V, E)$, em que V é o conjunto de **vértices**, cada um associado a um objeto, e E o conjunto de **arestas** correspondendo às relações entre estes objetos. Cada aresta é um par não-ordenado $e = \{u, v\}$ de vértices $u, v \in V$, chamados de **extremos** de e. Dizemos que e é **incidente** em u e v e que os vértices u e v são **adjacentes**. O **grau** de um vértice $v \in V$ é o número de arestas incidentes em v. Uma sequência de vértices $P = (u_1, \ldots, u_k)$ tal que $\{u_i, u_{i+1}\} \in E$ para $i = 1, \ldots, k$ é chamada de **passeio**. Se $u_1 = u_k$ dizemos que o passeio é **fechado** e, se somente o primeiro e o último vértices são iguais, então P é um **ciclo**. Quando todos os vértices da sequência são distintos chamamos de **caminho**. O predecessor e o sucessor do vértice u em um passeio P são chamados de **vizinhos de u em P**. Um **subgrafo** de um grafo $G = (V, E)$ é um grafo $G' = (V', E')$ tal que $V' \subseteq V$, $E' \subseteq E$ e se $\{u, v\} \in E'$ então $u, v \in V'$. Um grafo $G = (V, E)$ é dito **conexo** se existe um caminho ligando todo par de vértices $u, v \in V$. O **tamanho** de um grafo é dado por $|V| + |E|$.

Um conjunto M de arestas tal que nenhum par de arestas incide sobre o mesmo vértice é chamado de **emparelhamento**. Um emparelhamento M é **perfeito** se todo vértice do grafo é extremo de uma aresta de M. Uma **cobertura por vértices** é um conjunto de vértices K tal que toda aresta do grafo é incidente a pelo menos um vértice de K.

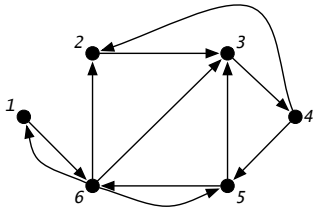
Um grafo $G = (V, E)$ é **bipartido** se seus vértices podem ser particionados em dois



(a) Grafo com seis vértices e nove arestas.



(b) Caminho do vértice 1 ao 3.



(c) Grafo orientado com seis vértices e onze arcos.

Figura 1.1: Conceitos básicos de Teoria dos Grafos.

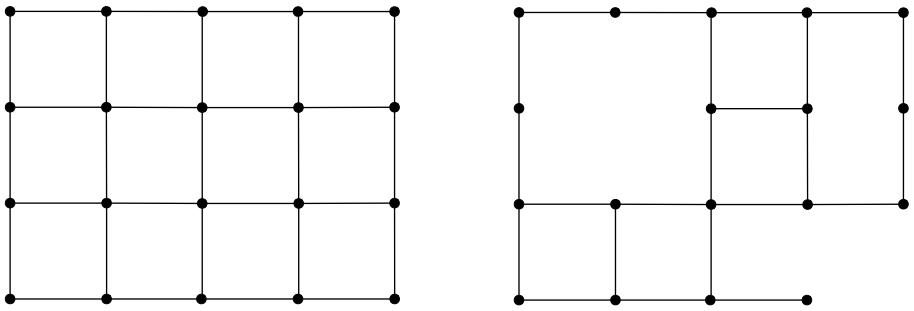
conjuntos V_1 e V_2 de forma que toda aresta incide em um vértice $u \in V_1$ e em um vértice $v \in V_2$.

Um **grafo orientado** é representado por $D = (V, A)$ em que V é um conjunto de vértices e A uma coleção de **arcos**. Um arco é um par ordenado $a = (u, v)$ de vértices $u, v \in V$. Dada uma aresta $\{u, v\}$ de um grafo G, os pares ordenados (u, v) e (v, u) representam as duas possíveis **orientações** da aresta. Os conceitos de passeio, caminho e ciclo, existem também na versão **orientada**, e a definição continua a mesma substituindo-se arestas por arcos.

Na figura 1.1 estão representados graficamente os conceitos e definições de grafos.

Dizemos que um grafo é uma **grade** $n \times m$, se os vértices correspondem aos pontos do plano com coordenadas inteiras (x, y) tal que $1 \leq x \leq n$ e $1 \leq y \leq m$ e existe uma aresta entre dois vértices, se e somente se, a distância no plano entre eles é de uma unidade. Um subgrafo de um grafo de grade é chamado de **grafo de grade parcial**. Na figura 1.2 temos alguns exemplos de grafos de grade e grade parcial.

Um grafo G conexo em que o grau de todo vértice é par é chamado de **grafo Euleriano** e um **passeio/tour Euleriano** é um passeio de G percorre cada aresta exatamente uma vez.



(a) Grafo de grade 4×5. (b) Grafo de grade parcial.

Figura 1.2: Grafos de grade e de grade parcial.

1.2.2 Álgebra Linear

Denotamos por $\mathbb{R}$ o conjunto dos números reais e por $\mathbb{Z}$ o conjunto dos números inteiros. Utilizamos $\mathbb{R}^n$ ($\mathbb{Z}^n$) para representar o conjunto dos vetores de n componentes reais (inteiras). Utilizamos x_i com $i = 1, \ldots, n$ para nos referir a i-ésima componente do vetor x.

O conjunto de todas as matrizes de m linhas e n colunas com elementos reais (inteiros) é denotado por $\mathbb{R}^{m \times n}$ ($\mathbb{Z}^{m \times n}$). A notação $A = (a_{ij})$ representa uma matriz A em que o elemento da linha i e coluna j para $i = 1, \ldots, n$ e $j = 1, \ldots, m$ é dado por a_{ij}.

Sejam a um vetor de coeficientes reais, x um vetor de variáveis reais, ambos com n componentes e b um número real. Uma equação da forma,

$$ax \leq b \equiv \sum_{i=1}^{n} a_i x_i \leq b$$

é chamada de **desigualdade linear**. Podemos escrever m desigualdades lineares descritas pelos vetores $a_1, \ldots, a_m$, x de n componentes e b um vetor de coeficientes com m componentes como $Ax \leq b$, onde A é uma matriz $m \times n$ em que o elemento a_{ij} é igual ao j-ésimo elemento do vetor a_i. Neste caso,

$$Ax \leq b \equiv \begin{cases} a_1 x \leq b_1 \\ a_i x \leq b_i \\ \quad \vdots \\ a_m x \leq b_m. \end{cases}$$

1.3 Definição do Problema

Em todos os problemas e situações discutidas no início deste capítulo temos como elemento comum, uma região que precisa ser percorrida por um objeto de área não nula. O custo do percurso está associado às mudanças de direção feitas durante o trajeto. A seguir formalizamos o problema que queremos tratar.

Seja P um polígono ortogonal com buracos (que também são polígonos ortogonais, inclusive aqueles com área zero), chamado de **pocket** e Q um quadrado unitário denotado por **cortador**, que se **move** sobre o *pocket* quando sai da posição (x, y) para a (x', y'). Dizemos que o cortador Q **cobre** a região correspondente à sua área. Além disso, o cortador tem seu movimento restrito às direções horizontal e vertical e em nenhum ponto do trajeto está fora do *pocket*. Entre dois movimentos consecutivos, dizemos que o cortador realizou uma **conversão**, que pode ser de três tipos:

- Colinear, quando não há mudança de direção nem de sentido.
- Simples, quando há mudança de direção.
- Meia-volta[1], quando há mudança de sentido.

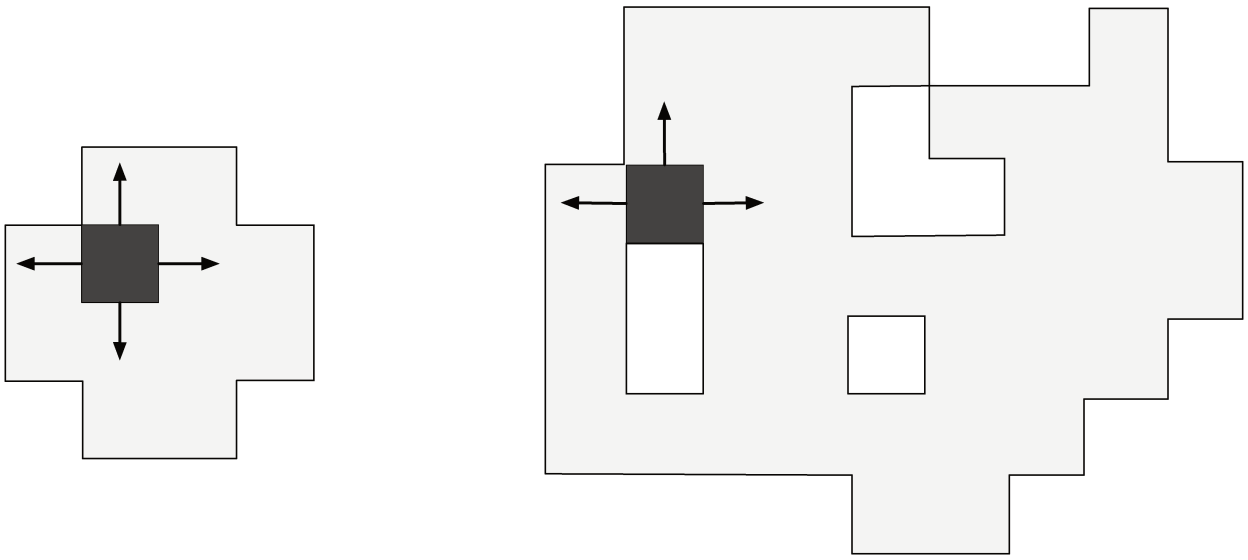
Associamos os custos zero, um e dois às conversões colinear, simples e meia-volta respectivamente.

Queremos encontrar uma sequência de movimentos consecutivos para o cortador, que comece e termine na mesma posição, tal que a área total coberta por ele seja exatamente a do *pocket* P e cuja a soma do custo das conversões seja mínima. Na figura 1.3 ilustramos esses conceitos, em 1.3a temos dois *pockets* um sem buraco e outro com três buracos e também o cortador com flechas representando os possíveis movimentos que o mesmo pode realizar a partir da posição onde se encontra. Os três tipos de conversão estão ilustrados na figura 1.3b e em 1.3c são dadas duas soluções possíveis para uma mesma instância.

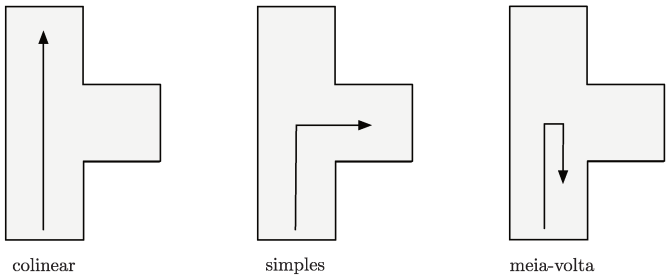
O Grafo de Subdivisão

Considere a situação em que podemos subdividir o pocket em N quadrados unitários disjuntos, chamados de **pixels**. Neste caso, o **grafo de subdivisão** $G = (V, E)$ é definido do seguinte modo. A cada pixel associamos um vértice de G e existe uma aresta entre dois vértices se os pixels correspondentes compartilham um lado, e além disso, o segmento de reta entre os seus centróides destes está inteiramente contido no pocket. Na figura 1.4 temos o grafo de subdivisão para uma instância do problema do recorte ortogonal. Observe que o grafo de subdivisão é um grafo de grade parcial.

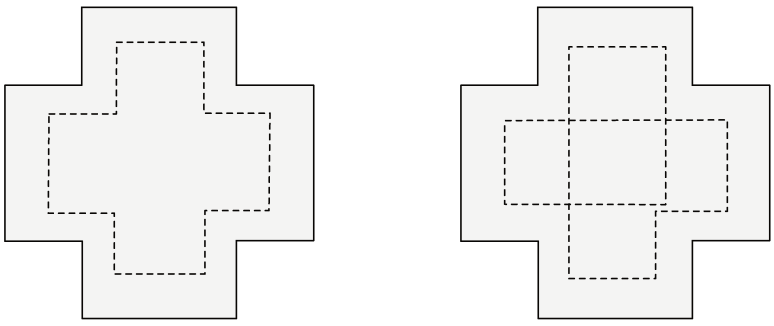
[1] Também chamada de conversão em U.



(a) Exemplos de *pocket* e cortador.

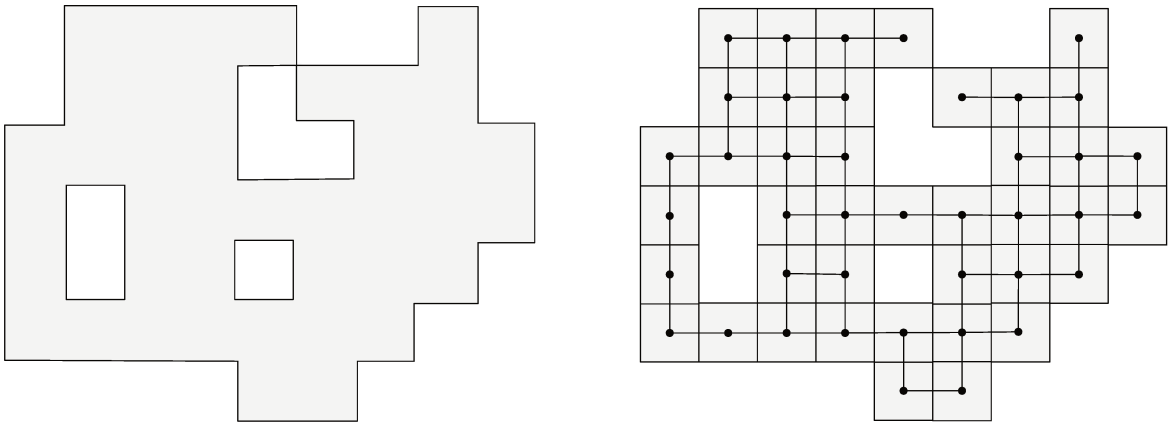


(b) Os três tipos de conversões.



(c) Duas possíveis soluções, a primeira com custo 12 e a segunda 10.

Figura 1.3: Definição do problema do recorte.



(a) Pocket do recorte ortogonal. (b) Grafo de subdivisão.

Figura 1.4: Grafo de subdivisão do pocket de uma instância do recorte ortogonal.

Com esta definição podemos enunciar o problema do recorte ortogonal, que é geométrico por natureza, como um problema em grafos. Assim, seja $G = (V, E)$ um grafo de subdivisão. Uma tripla de vértices (u, v, w) é uma conversão se u é adjacente a v e v é adjacente a w. O custo de uma conversão $c(u, v, w)$ é 0 se u e w são colineares, 1 se u e w são ortogonais e 2 se $u = w$. Note-se que essa definição de conversão é equivalente à definição geométrica vista na seção 1.3.

Um passeio fechado $W = (u_1, \ldots, u_r)$ que visita todos os vértices de G é chamado de **milling tour**. O custo de um *milling tour* é dado como:

$$\text{custo}(W) = c(u_{r-1}, u_r, u_1) + \sum_{i=1}^{r-2} c(u_i, u_{i+1}, u_{i+2}). \tag{1.1}$$

No **Problema do Recorte Ortogonal Discreto (MTWTC)**[2], queremos encontrar um *milling tour* de custo mínimo.

Nesta seção, associamos a uma entrada do problema do recorte ortogonal, dada por um *pocket* P e um cortador Q, o grafo de subdivisão $G = (V, E)$. Com isto, os *pixels* da discretização e os vértices de G são equivalentes e, portanto, intercambiáveis. No decorrer do texto, em um abuso de linguagem, usaremos essa equivalência implicitamente em frases como "um passeio fechado que visita todos os *pixels* do *pocket*" ou "os *pixels* adjacentes a um vértice".

Podemos representar uma solução do MTWTC de duas formas. Na primeira, como um passeio fechado do grafo de subdivisão, que corresponde à sequência de visitação dos vértices. Na segunda maneira, chamada de **lista de segmentos**, representa-se a

[2]Sigla derivada da denominação em língua inglesa, *Milling Tour with Turn Costs*.

solução de maneira mais compacta como uma sequência de segmentos ortogonais $W = (u_1, u'_1), \ldots, (u_k, u'_k)$ com $u_{i+1} = u'_i$ e $u_1 = u'_k$ (em uma conversão do tipo meia-volta é usado um segmento de comprimento zero). Essa última representação é útil quando queremos ignorar as conversões do tipo colinear e simplificar algumas demonstrações.

Utilizando os custos das conversões como sendo 0 para colinear, 1 simples e 2 para meia-volta, e restringindo-nos a polígonos ortogonais e a movimentos nas direções horizontal e vertical como no MTWTC, podemos derivar o seguinte resultado.

Teorema 1. *O custo de um milling tour para uma instância do MTWTC é sempre par.*

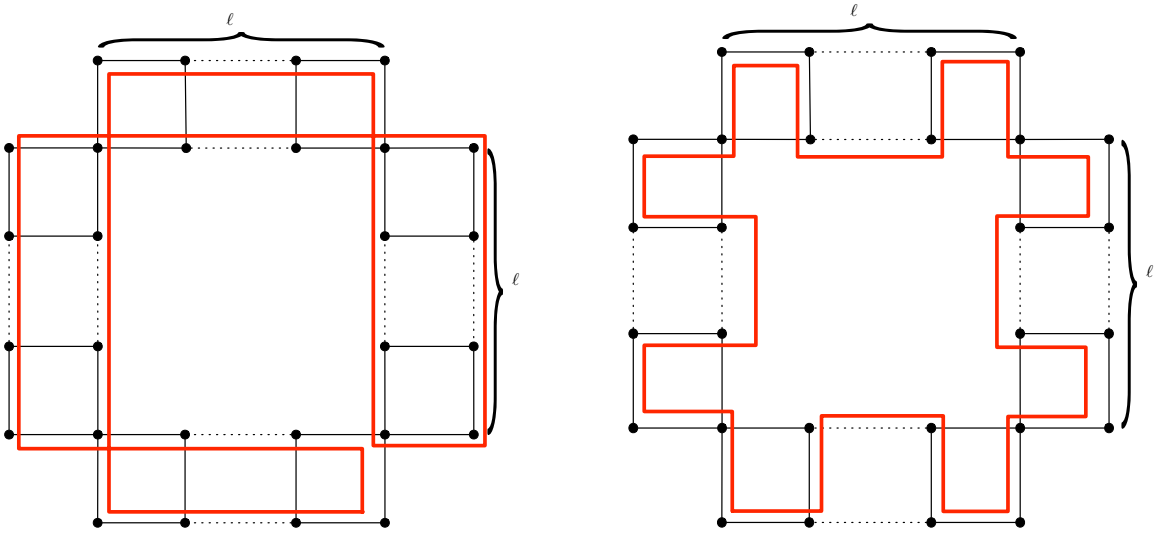
Demonstração. Seja S uma solução na forma de lista de segmentos. Podemos supor que no pixel inicial a direção não muda, isso implica que a sequência começa e termina com segmentos na mesma direção. Como a direção dos segmentos de S alternam entre horizontal e vertical, o número de conversões simples entre quaisquers dois segmentos na mesma direção é par. Em particular, isto vale para o primero e para o último segmento, o que nos conduz ao resultado desejado. □

A principal conseqüência desse resultado se dá na verificação de otimalidade de soluções. Seja S uma solução com valor s, suponha que pelas propriedades do problema conheçamos um limite inferior $\underline{s} \leq s$. A princípio, como os custos das conversões são inteiros, podemos concluir que S é uma solução ótima se $s - \underline{s} < 1$. Porém, com o teorema 1, podemos melhorar essa condição para $s - \underline{s} < 2$. No capítulo 2 desenvolvemos um algoritmo exato para o MTWTC que, a cada iteração, mantém um limitante inferior para a solução e, portanto, podemos aplicar esse resultado como critério de parada, melhorando sua eficiência computacional.

Problemas Semelhantes

A semelhança deste problema com outros bem estudados na literatura, nos faz questionar se soluções para problemas similares também são boas para o MTWTC. Claramente, no caso em que o grafo de subdivisão possui um **ciclo Hamiltoniano**, isto é, um ciclo que visita cada vértice exatamente uma vez, esse ciclo é uma solução válida. O problema do ciclo Hamiltoniano e sua versão com custo nas arestas, conhecida como **Problema do Caixeiro Viajante (TSP)**[3], são muito bem estudados. Na literatura encontramos algoritmos aproximados e heurísticas de boa qualidade para o TSP [29]. No entanto, mesmo que encontremos o ciclo Hamiltoniano com custo mínimo em número de conversões, está solução pode ser tão ruim quanto se queria, como ilustrado no exemplo da figura 1.5. Para esta classe de instâncias, a solução ótima do MTWTC, dada na figura 1.5a independe

[3]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*.



(a) Solução ótima do MTWTC, com custo fixo igual a 10.

(b) Ciclo Hamiltoniano, para ℓ ímpar, o custo é dado por $8\ell + 4$. Se ℓ for par, o grafo não possui um ciclo Hamiltoniano.

Figura 1.5: Um ciclo Hamiltoniano pode não ser uma boa aproximação para o MTWTC.

do lado ℓ e tem valor fixo 10. Já a solução com ciclo Hamiltoniano, da figura 1.5b, tem custo mínimo $n = 8\ell + 4$ que corresponde ao número de vértices do grafo.

Esse e outros problemas semelhantes, apresentados na próxima seção nos incentivam a buscar algoritmos específicos para o MTWTC.

1.4 Revisão Bibliográfica

Nesta seção será feita uma revisão dos trabalhos anteriores que discutem problemas relacionados ao MTWTC e que propõem métodos voltados para resolução dos mesmos.

O artigo [6] é de grande importância pois é o primeiro a fazer o estudo algorítmico do problema MTWTC. Nele é demonstrado que o problema é $\mathcal{NP}$-difícil, mesmo no caso mais simples onde P é um polígono **ortogonal estreito**, i.e., que não contém um quadrado 2×2. Nesse mesmo artigo também são dados algoritmos aproximados para diversas variações do problema. Para o recorte discreto é dado um algoritmo $(2\Delta + \rho + 2)$-aproximado, onde Δ é o grau máximo do grafo e ρ o número máximo de direções de movimentação do cortador. No caso de recorte ortogonal se tem um fator de aproximação constante de 6.25 e se, além disso, restringirmos os polígonos aqueles de coordenadas inteiras, o chamado **recorte ortogonal inteiro**, este fator é reduzido para 3.75.

A maioria dos estudos sobre problemas relacionados ao MTWTC tem como objetivo

minimizar o comprimento do *tour*. Em [7, 8] é demonstrado que o problema nessa versão também é $\mathcal{NP}$-difícil e são dados algoritmos aproximados com fator constante, tanto para os problemas de recorte quanto para o *lawn mowing problem*, uma variação do recorte em que o cortador não é restrito a deslocar-se dentro do *p*ocket P. Essa variação também é conhecida como *traveling cameraman problem* e foi estudada por [21]. Neste último caso a região P não precisa ser conexa.

Para o caso do recorte ortogonal inteiro, temos o problema semelhante de encontrar um ciclo Hamiltoniano em um grafo do tipo grade, que é polinomial se o grafo é **sólido**, i.e., planar e todas as suas faces têm tamanho quatro, exceto possivelmente uma, conforme [33].

Em Geometria Computacional temos o problema do *watchman route* em que um vigia deve cobrir uma região e queremos encontrar a rota de comprimento mínimo que ele deve percorrer para tal. A região coberta pelo vigia é determinada pelo seu campo de visão e, no caso em que esse é especificado por uma área pré-determinada, podemos fazer uma analogia entre o vigia e o seu campo de visão com o cortador e a área ocupada por ele. Algoritmos polinomiais existem no caso da região ser um polígono simples e o campo de visão ser ilimitado. No entanto, o problema é $\mathcal{NP}$-completo para o caso geral de polígonos com buracos. Para uma revisão desse e de outros problemas relacionados é possível consultar a referência [27].

Na comunidade de Pesquisa Operacional diversos trabalhos foram feitos em problemas de roteamento semelhantes ao problema do recorte discreto. Em [4] é estudada a variação do TSP em que se deseja minimizar a soma dos ângulos do *tour*, sendo estes últimos medidos entre duas arestas consecutivas do *tour*. Os resultados principais do artigos são a prova de que essa variação do TSP é $\mathcal{NP}$-difícil e a obtenção de um algoritmo $O(\log\ n)$-aproximado para resolvê-lo.

Outra variação do TSP similar ao problema do recorte é o *minimum bends TSP*, em que dado um conjunto de pontos no plano, o objetivo é encontrar um *tour* que minimiza o número de conversões. Algoritmos aproximados para este problema e suas variações foram estudadas por [31]. Uma nuance interessante deste artigo, é que o caixeiro não precisa necessariamente "andar" somente sobre as arestas.

Em [12, 10], os autores estudam heurísticas e algoritmos exatos para o problema do *carteiro rural*. Nesse problema, dado um grafo misto $G = (V, E, A)$, um conjunto de arestas obrigatórias R e um conjunto de arcos obrigatórios A_R, deseja-se encontrar um *tour* de custo mínimo que visite todas as arestas de R e todos os arcos de A_R pelo menos uma vez. A função objetivo visa a minimização do comprimento do *tour* com penalidades ao fazer conversões. Os algoritmos exatos são baseados na redução desse problema para o problema do TSP assimétrico.

No artigo [18], o problema do *menor caminho com conversões* é estudado e um al-

goritmo que utiliza técnicas de pré-processamento e re-uso das tabelas de custos para melhorar a eficiência computacional é apresentado.

1.5 Objetivos

Tendo em vista a revisão bibliográfica feita na seção anterior, constatamos que já existem alguns algoritmos aproximados específicos para o MTWTC, apresentado no capítulo 3. No entanto, os resultados da literatura são somente teóricos, já que em nenhum dos trabalhos uma implementação foi feita para avaliar experimentalmente estes algoritmos. Ademais, não foi possível encontrar nenhum algoritmo exato ou mesmo heurístico para tratar o problema. Sendo assim, as metas estabelecidas para esta dissertação foram as seguintes:

1. Criar um *benchmark* de instâncias de teste para o MTWTC, e colocá-las em domínio público, de forma que possam ser utilizadas na avaliação dos algoritmos desenvolvidos nesta dissertação e trabalhos posteriores.

2. Desenvolver e implementar um algoritmo exato para o MTWTC, que será usado como referência nas comparações e análises experimentais.

3. Implementar o algoritmo aproximado proposto em [6], avaliar seu comportamento experimental e comparar com os resultados teóricos já conhecidos.

4. Investigar a adoção de heurísticas e meta-heurísticas para o MTWTC, de forma a encontrar uma solução de compromisso entre resultados teóricos e eficiência prática.

5. Utilizar o *benchmark* do item 1 para fazer uma análise experimental e comparativa dos algoritmos apresentados nessa dissertação.

Como será visto ao longo deste documento, tais metas foram alcançadas, fazendo com que as principais contribuições dessa dissertação fossem: (1) o desenvolvimento e implementação do primeiro algoritmo exato para o MTWTC que se tem notícia; (2) a disponibilização do primeiro conjunto de instâncias do MTWTC em domínio público; (3) o desenvolvimento das primeiras heurísticas para o problema e (4) a descrição e análises comparativas inéditas de extensivos testes experimentais realizados com os algoritmos propostos para o MTWTC.

Capítulo 2

Resolução Exata do MTWTC

Neste capítulo desenvolveremos um método exato para resolução do MTWTC. Para tanto, o problema será modelado como um *Programa Linear Inteiro*, o qual deve ser resolvido usando os algoritmos existentes para problemas dessa categoria.

Na seção seguinte apresentamos de forma bastante sucinta alguns conceitos relacionados à *Programação Linear* e *Programação Linear Inteira* que consideramos importantes para o bom entendimento do trabalho que foi feito nesta dissertação.. Existem várias obras que podem ser consultadas pelo leitor interessado em um tratamento mais aprofundado destes assuntos, dentre as quais recomendamos os livros [9], [28] e [34]. Na seção 2.2 dedicamo-nos à modelagem do MTWTC usando Programação Linear Inteira e às questões algorítmicas ligadas à sua resolução.

2.1 Conceitos Básicos de Programação Linear

Não raro nos confrontamos com problemas em que devemos tomar decisões com relação ao uso de recursos ou produção de bens sujeitos a diversas restrições que muitas vezes competem entre si. Normalmente nosso objetivo é encontrar a solução de menor custo ou maior ganho.

Para uma classe bastante abrangente de problemas de otimização, podemos utilizar um conjunto de técnicas de modelagem e métodos de resolução que definem a **Programação Linear (PL)**. Essa classe é composta por problemas sobre variáveis contínuas em que as restrições e a função objetivo podem ser descritas por equações e desigualdades lineares.

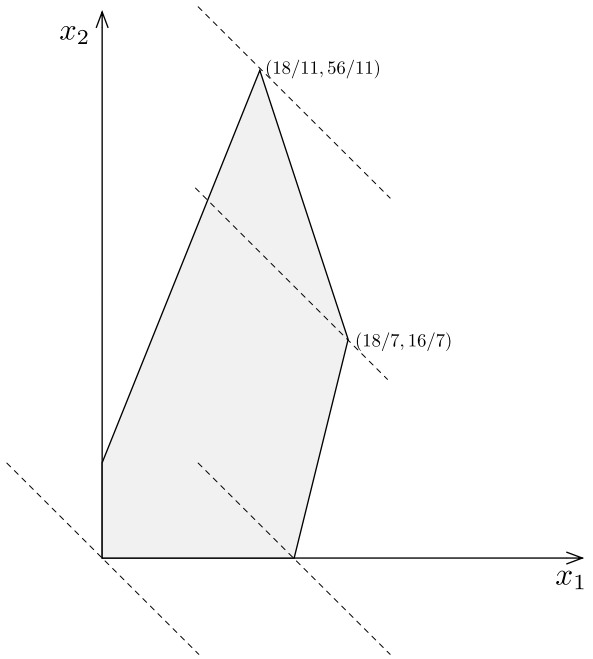


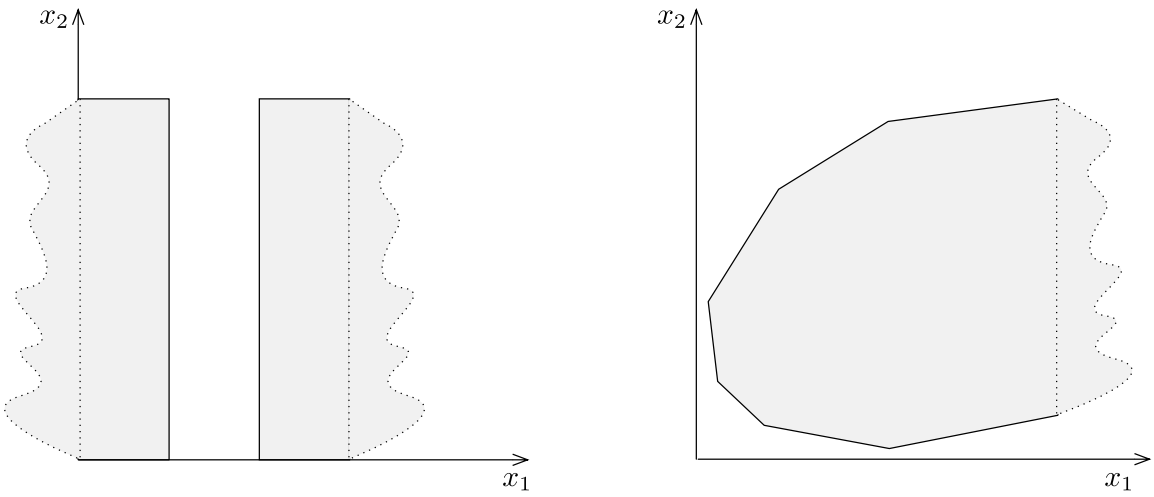
Figura 2.1: Região factível de um problema de **PL**.

Abaixo temos um exemplo de um problema de **PL** sobre as variáveis x_1 e x_2:

$$\begin{aligned} \max \quad & x_1 + x_2 && (2.1)\\ \text{s.a} \quad & 4x_1 - x_2 \leq 8 && (2.2)\\ & 2x_1 + x_2 \leq 10 && (2.3)\\ & 5x_1 - 2x_2 \geq -2 && (2.4)\\ & x_1, x_2 \geq 0. && (2.5) \end{aligned}$$

Na figura 2.1 estão representados graficamente os pontos da região (escura) que satisfazem as restrições (2.2) a (2.5), a chamada **região factível**. A função que queremos maximizar é chamada de **função objetivo** e o seu valor em um ponto específico é o **valor objetivo** deste ponto. Para uma solução que otimiza (maximiza ou minimiza) a função objetivo, denominamos esta quantidade por **valor ótimo** ou, simplesmente, **ótimo**.

Podemos resolver esse **PL** com duas variáveis graficamente da seguinte maneira. Considere a reta $x_1+x_2 = 0$, representada na figura 2.1 e cuja intersecção com a região factível – no exemplo, é composta somente pelo ponto $(0, 0)$ – corresponde as soluções do modelo cuja função objetivo tem valor 0. De maneira geral, a intersecção da reta $x_1 + x_2 = z$ com a região factível corresponde as soluções com valor objetivo z. Como a região do exemplo



(a) Problema infactível. (b) Problema com valor objetivo ilimitado.

Figura 2.2: Dois outros casos para a solução de um problema de **PL**.

é limitada, existe um valor máximo para z tal que a intersecção seja não vazia, qualquer ponto nessa intersecção é uma solução ótima do **PL** em questão. No exemplo, a solução ótima é o ponto $\left(\frac{18}{11}, \frac{56}{11}\right)$ cujo valor objetivo é, $z = \frac{74}{11}$.

As seguintes observações podem ser feitas do exemplo em duas dimensões, e também são válidas para **PL** em geral: a região factível é convexa e, se o valor ótimo é finito, existe um ponto extremo correspondendo a uma solução é ótima. Este último fato será formalizado mais adiante (ver Teorema 2).

Duas outras situações podem ocorrer: o problema é **infactível**, isto é, não tem solução viável ou **ilimitado**, quando o valor ótimo não é finito. A figura 2.2 ilustra estes casos. Em 2.2a temos um problema infactível, pois as restrições $x_1 \geq 2$ e $x_1 \leq 1$ não podem ser satisfeitas simultaneamente já em 2.2b podemos aumentar o valor objetivo o quanto quisermos.

2.1.1 Resolução de PL pelo Algoritmo SIMPLEX

Para a descrição dos métodos de resolução para modelos de **PL** gerais, é conveniente definir uma forma canônica de descrever o modelo. Na **forma canônica** temos n números reais $c_1, \ldots, c_n$, m números reais $b_1, \ldots, b_m$; e $m \times n$ números reais a_{ij} com $i = 1, \ldots, m$ e $j = 1, \ldots, n$. Além disso, temos n variáveis reais $x_1, \ldots, x_n$, e queremos resolver o

problema:

$$\max \quad \sum_{j=1}^{n} c_j x_j \tag{2.6}$$

$$\text{s.a} \quad \sum_{j=1}^{n} a_{ij} x_j \leq b_i \quad i = 1, \ldots, m \tag{2.7}$$

$$x_j \geq 0 \quad j = 1, \ldots, n. \tag{2.8}$$

O mesmo problema pode ser escrito na forma matricial. Seja $A = (a_{ij})$ uma matriz $m \times n$, $c = (c_i)$ e $x = (x_i)$ vetores de n componentes e $b = (b_i)$ um vetor de m componentes, temos:

$$\max \quad cx \tag{2.9}$$

$$\text{s.a} \quad Ax \leq b \tag{2.10}$$

$$x \geq 0 \tag{2.11}$$

De uma maneira compacta, um problema de **PL** pode ser denotado simplesmente por: $\max\{cx : Ax \leq b, x \geq 0\}$.

O conjunto de pontos que satisfazem um sistema de desigualdades lineares é chamado de **poliedro**. Portanto, a região factível de um **PL** é um poliedro. Os vértices, ou **pontos extremos** do poliedro associado a região factível de um **PL**, são importantes para sua otimização.

Considere novamente o exemplo da figura 2.1. A solução ótima daquele **PL** é um ponto extremo da sua região viável. Na verdade, isto não é uma coincidência mas sim um resultado básico de Programação Linear que generaliza o que ocorre neste exemplo de duas variáveis para poliedros definidos sobre mais variáveis.

Teorema 2. *Se um* **PL** *tem uma solução ótima com valor finito, então existe um ponto extremo do poliedro associado a região factível com valor ótimo.*

Demonstração. Ver [9, 13]. □

A partir do teorema 2, podemos esboçar um algoritmo que avalia o custo da solução nos pontos extremos e determina o de maior custo como sendo o ótimo. O algoritmo **SIMPLEX** utiliza esta ideia, e consiste em a partir de um ponto extremo (solução) inicial P_0 visitar os outros, construindo uma sequência de pontos extremos $P_0, P_1, \ldots, P_k$ tal que o valor objetivo em P_i é menor igual ao de P_{i+1} para $i = 0, \ldots, k-1$ e P_k é uma solução ótima.

Para o algoritmo do SIMPLEX ficar bem definido é necessário descrever precisamente os seguintes passos:

1. Como encontrar uma solução inicial factível.

2. Determinar se um ponto extremo P_i é ótimo.

3. Dado um ponto extremo P_i gerar P_{i+1}.

O passo 1, em muitos casos é trivial, e caso não seja, é usado um programa linear auxiliar derivado do problema original, com solução inicial trivial e cuja solução ótima também é factível para o problema original.

Os passos 2 e 3 envolvem operações algébricas feitas utilizando a forma canônica de um modelo de **PL** e o conceito de *dualidade*. Para uma descrição completa do algoritmo SIMPLEX assim como uma discussão de sua implementação, recomendamos o livro [9].

Na mesma referência pode ser encontrada a análise da complexidade do SIMPLEX. Neste sentido, o resultado principal é que o algoritmo não é polinomial e existem instâncias que, no pior caso, exigem um número exponencial de passos [24] para serem resolvidas. Porém, na prática, o algoritmo é extremamente eficiente e continua sendo usado como um dos métodos de resolução de **PL** em grande parte das aplicações.

A questão se existem algoritmos polinomiais para Programação Linear ficou em aberto durante vários anos e foi respondida positivamente com o advento do método dos elipsóides e o algoritmo de Karmarkar [9, 23].

2.1.2 Programação Linear Inteira (PLI)

Em alguns problemas de **PL** queremos modelar variáveis que só podem receber valores discretos. É o caso, por exemplo, quando precisamos especificar a quantidade de ônibus necessária para transportar um certo número de passageiros ou ainda decidir quais elementos de um conjunto finito serão usados para compor a solução de um problema. Para isto, utilizamos variáveis inteiras e, as vezes, binárias. Seguindo as definições utilizadas para **PL**, um problema de **PLI** com n variáveis é escrito como:

$$IP = \max\{cx : Ax \leq b, x \geq 0, x \in \mathbb{Z}^n\}, \tag{2.12}$$

e chamamos o problema linear obtido ao relaxar as restrições de integralidade, ou seja,

$$LP = \max\{cx : Ax \leq b, x \geq 0, x \in \mathbb{R}^n\} \tag{2.13}$$

de **relaxação linear** de IP.

Graficamente, em um modelo de **PLI** as soluções são os pontos de coordenadas inteiras dentro da região factível da relaxação linear. A figura 2.3 ilustra esta e uma outra característica importante de **PLI**, a de que o mesmo conjunto de soluções pode ser representado por modelos distintos, também chamados de **formulações**. Dentre as infinitas

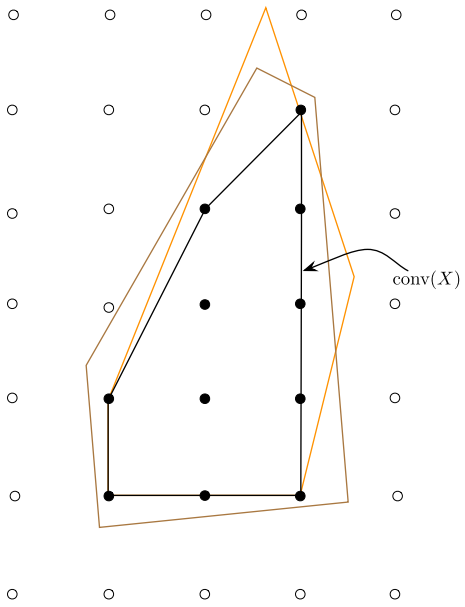


Figura 2.3: Três formulações distintas para o mesmo problema de **PLI**.

formulações possíveis, uma delas merece atenção especial, e está indicada na figura. É a chamada **envoltória convexa** de um conjunto de pontos $X \subset \mathbb{R}^n$, denotada por $\text{conv}(X)$ e definida como o menor poliedro que contém todos os pontos de X. Em particular, estamos interessados no caso em que X é o conjunto de pontos que correspondem às soluções do **PLI** que se quer resolver. Na figura 2.3, o conjunto X é dado pelos pontos pretos que representam as soluções do **PLI** ilustrado.

A importância de conhecermos um sistema linear que descreve a envoltória convexa de X é que, a rigor, isso permite resolver um problema de **PLI** usando um algoritmo de **PL**. Isso porque, supondo que o ótimo é finito, sabemos pelo Teorema 2, que existe uma solução ótima que é um ponto extremo de $\text{conv}(X)$, logo, inteira. Em teoria, esta redução do **PLI** para um **PL** com restrições definidas pela envoltória convexa de X parece ser atraente, uma vez que **PL** pertence à classe $\mathcal{P}$ enquanto **PLI** está em $\mathcal{NP}$-difícil. Na prática, o que ocorre é que o sistema linear que descreve $\text{conv}(X)$ contém um número de restrições exponencialmente grande no número de variáveis (i.e., na dimensão do espaço onde $\text{conv}(X)$ está imerso). O uso de um algoritmo polinomial para **PL** ainda iria requerer um tempo exponencial no número de variáveis para poder ser resolvido. Mesmo assim, como será visto a seguir, é computacionalmente interessante procurar por formulações de um **PLI** que se aproximem da envoltória convexa de suas soluções viáveis.

Algoritmos de Branch-and-Bound e de Branch-and-Cut

Como **PLI** pertence à classe $\mathcal{NP}$-difícil e não se sabe se $\mathcal{P}$=$\mathcal{NP}$, a complexidade dos algoritmos existentes atualmente para calcular modelos de **PLI** são exponenciais. Porém, dada a natureza combinatória dos problemas de **PLI**, os algoritmos de *força bruta* baseados em enumeração completa do espaço de soluções são impraticáveis. Neste caso, aplicam-se estratégias que permitem uma *enumeração implícita* deste espaço, o que alcançado por meio de algoritmos que se valem do paradigma de *divisão e conquista*. Um deles é o chamado algoritmo de *branch-and-bound*, o qual é explicado a seguir para o contexto específico de **PLI**.

Queremos resolver o problema de **PLI**:

$$P = \max\{cx : x \in S\} \tag{2.14}$$

onde,

$$S = \{x : Ax \leq b, x \geq 0, x \text{ inteiro}\}. \tag{2.15}$$

O algoritmo de **Branch-and-Bound**, é usado para resolver problemas dessa forma e consiste na construção de uma **árvore de enumeração** aliada a poda de nós utilizando limitantes.

Os nós da árvore são os problemas $P^i = \max\{cx : x \in S_i\}$ onde S_i é um subconjunto de S e $S_0 = S$. O **nó raiz** é $P^0 = P$. Seja P^i um nó qualquer da árvore e x^i a solução da relaxação linear de P^i. Geramos dois nós filhos P^{2i+1} e P^{2i+2} de P^i da seguinte forma, escolha uma variável x^i_j com valor fracionário e faça:

$$S_{2i+1} = S_i \cap \{x : x_j \leq \lfloor x^i_j \rfloor\} \tag{2.16}$$

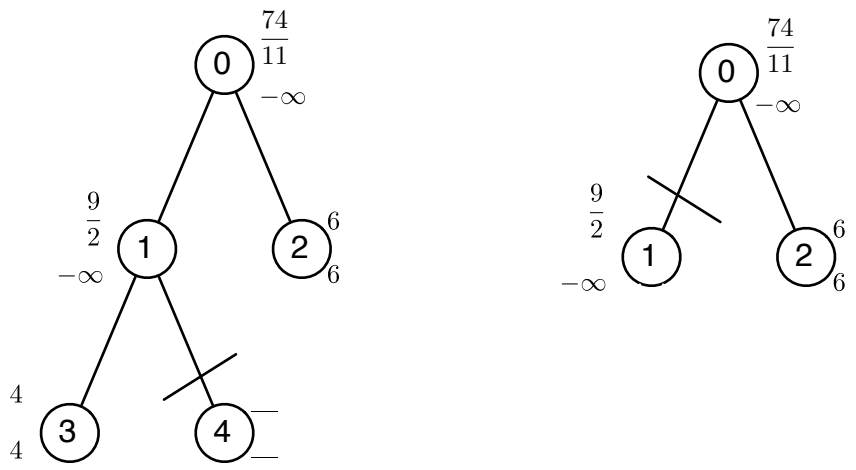
$$S_{2i+2} = S_i \cap \{x : x_j \geq \lceil x^i_j \rceil\} \tag{2.17}$$

Note que, $S_i = S_{2i+1} \cup S_{2i+2}$, $S_{2i+1} \cap S_{2i+2} = \emptyset$ e a solução de P^i é a melhor entre P^{2i+1} e P^{2i+2}. Além disso, x^i não é uma solução válida para as relaxações lineares de P^{2i+1} ou P^{2i+2}.

Para cada nó P_i, são atribuídos um limitante superior $\overline{z}^i$ e um limitante inferior $\underline{z}^i$. Os limitantes inferiores correspondem ao valor da solução inteira associado a um nó (quando é possível obter uma solução inteira, e $\underline{z}^i = -\infty$ caso contrário) e os superiores são iguais ao valor objetivo da relaxação linear do nó. Além disso mantemos um limitante superior (inferior) global $\overline{z}$ ($\underline{z}$) que é definido como sendo o maior limitante superior (inferior) dentre todos aqueles obtidos nos nós gerados.

Durante a construção da árvore de enumeração, os limitantes são usados para a poda de nós da árvore. Existem três razões pelas quais um nó pode ser podado

1) **poda por otimalidade**, o nó P^i foi resolvido.



(a) Árvore de **B&B**, explorando os nós na ordem: $0, 1, 3, 4, 2$.

(b) Árvore de **B&B**, explorando os nós na ordem: $0, 2, 1$.

Figura 2.4: Árvores de enumeração para um problema de **PLI**.

2) **poda por limitante**, $\overline{z}^i \leq \underline{z}$.

3) **poda por infactibilidade**, $S_i = \emptyset$.

A figura 2.4 ilustra a construção da árvore para o problema da figura 2.1 adicionada à exigência das variáveis serem inteiras, assim como os três casos de poda que podem ocorrer. Em 2.4a, após resolver o problema do nó raiz, arbitrariamente o nó 1 é escolhido para ser explorado e são gerados os filhos 3 e 4. Explorando o nó 3, o problema é resolvido e, portanto, tem-se uma poda por otimalidade, e o limitante inferior global, $\underline{z}$, é atualizado para 4. A poda do nó 4 é por infactibilidade. Resta agora o nó 2, que é resolvido na otimalidade com valor 6 e este é o novo limitante inferior. Como não há mais nós a serem explorados e o limitante inferior é igual ao superior, a solução do **PLI** tem valor objetivo 6.

No caso 2.4b, após resolver o problema da raiz, o nó 2 é escolhido e, como visto em 2.4a, resolvido na otimalidade com valor 6, que se torna o novo limitante inferior. O limitante superior do nó 1 é $\frac{9}{2} < 6$, logo, o nó 1 é podado por limitante e o problema é resolvido com valor objetivo 6.

O algoritmo de **Branch-and-Bound** descrito acima é capaz de resolver uma grande quantidade de problemas de **PLI**. Entretanto, seu sucesso é altamente dependente da qualidade dos limitantes obtidos a partir do modelo **PLI** empregado. Isto porque a eficácia do método está pautada essencialmente na capacidade de se efetuar as podas descritas na página 19. Estas, por sua vez, serão tão mais bem sucedidas quanto mais *apertados* forem

os limitantes superiores e inferiores calculados nos nós, ou seja, quanto mais próximos eles forem do valor ótimo referente àquele nó. Fica claro então, a importância de termos uma formulação que seja próxima da envoltória convexa das soluções inteiras, tendo em vista que, para esta última, os valores dos limitantes e o ótimo se igualam.

A partir da observação anterior, estamos aptos a entender o intuito de se aplicar a técnica de **planos de corte** na resolução de um **PLI**, que consiste em adicionar desigualdades à formulação do problema de forma a obter melhores limitantes da relaxação linear. Em outras palavras, tentamos iterativamente fazer com que a formulação corrente se aproxime cada vez mais do conv(X), resolvendo-se uma seqüência de **PL**s até que, idealmente, se obtenha uma solução em X (inteira), a qual deve ser ótima para o **PLI**. Convém notar que a técnica de planos de corte não exige sequer que as soluções inteiras do sistema linear inicialmente usado para aproximar o conv(X) estejam em X. Isso porque, na verdade, se em uma iteração chega-se a uma solução que não está em X, em teoria, a mesma pode ser descartada do conjunto de soluções da formulação da próxima iteração. Voltaremos a comentar sobre este tema mais para frente, após termos introduzido alguns conceitos relevantes para esta discussão. Contudo, já podemos adiantar que a situação descrita acima se aplica ao modelo para o MTWTC que desenvolvemos na próxima seção.

Para compreendermos o funcionamento do algoritmo de planos de corte, considere o problema, $IP = \max\{cx : x \in X\}$ onde $X = \{x \in \mathbb{R}^n : Ax \leq b, x \geq 0, x \text{ inteiro}\}$. Uma desigualdade linear $\pi x \leq \pi_0$ será *válida* para X se ela for satisfeita por todos pontos de X e, consequentemente, por todos pontos de conv(X). A *face* definida por essa desigualdade em conv(X) é dada pelos pontos deste poliedro para os quais $\pi x = \pi_0$. Se a dimensão desta face for igual a de conv(X) subtraída de uma unidade, diz-se que face definida pela desigualdade é uma *faceta*. Juntamente com um sistema minimal de equações lineares que descrevem os hiperplanos contendo conv(X), as desigualdades definidoras de facetas são necessárias e suficientes para descrever este poliedro. Assim, a princípio, poder-se trabalhar apenas com desigualdades definidoras de facetas. Porém, na prática, muitas vezes é mais eficiente limitar-se à utilização de *desigualdades válidas fortes* (aquelas que definem faces de alta dimensão, mas não necessariamente facetas), as quais, não raro, podem ser obtidas de desigualdades mais fracas por meio de operações conhecidas como *lifting*.

Neste ponto cabe notar que o conhecimento de desigualdades válidas por si só não permite que as mesmas sejam empregadas em um algoritmo de planos de corte. Para tanto, torna-se necessária a resolução do chamado *problema de separação* relativo a elas. Dada uma família $\mathcal{F}$ de desigualdades válidas para X e uma solução $x^\star$ para a relaxação linear de IP, o **problema da separação** de $\mathcal{F}$ relativo a X é definido por: *encontrar uma desigualdade $\pi x \leq \pi_0$ de $\mathcal{F}$ que é violada por $x^\star$ ou determinar que tal desigualdade não existe*. Um **algoritmo de separação** para $\mathcal{F}$ referente a X, é um algoritmo que

responde a tal problema.

Com as definições anteriores, podemos elencar os principais passos de um algoritmo de planos de corte para IP. Para tanto, em um primeiro momento, iremos supor que $\mathcal{F}$ contém todas as desigualdades válidas para X e que se conheçe um algoritmo que resolve o problema de separação associado. Assim, inicialmente, obtém-se um sistema linear que é satisfeito por todos pontos de X (e, eventualmente, por outros pontos inteiros fora deste conjunto). Em seguida, repetem-se iterativamente os seguintes passos: resolve-se a relaxação linear corrente, obtendo-se a solução $x^\star$; resolve-se o problema de separação para $\mathcal{F}$ e, se nenhuma desigualdade é retornada, declara-se $x^\star$ como sendo ótima, do contrário, insere-se a desigualdade $\pi x \leq \pi_0$, retornada pelo algoritmo de separação e, consequentemente, violada por $x^\star$, na formulação corrente, repetindo-se os passos anteriores. Algumas considerações se fazem necessárias quando se trata de implementar um algoritmo de planos de corte.

Primeiramente, vale destacar que, o *limitante dual* (que, no caso de um problema de maximização como o que é usado nesta apresentação, corresponde a um limite superior) decresce monotonamente ao longo das iterações do algoritmo de planos de corte. Com isto, a interrupção do mesmo, ainda que após uma quantidade reduzida de iterações, já poderá levar a limitantes duais de muito boa qualidade. Outro fato relevante é que, nesta exposição, partimos da hipótese de que $\mathcal{F}$ contém todas as desigualdades válidas para X. Na teoria de **PLI**, é possível encontrar famílias que satisfazem a esta propriedade como, por exemplo, os **cortes de Chvàtal-Gomory**. Na prática, o uso dessas desigualdades de âmbito geral levam o algoritmo a uma convergência muito lenta, inviabilizando o seu uso. No entanto, a literatura especializada é rica em exemplos de que o estudo de um modelo linear para um problema específico permite em muitos casos encontrar famílias de desigualdades válidas que, mesmo não contendo todas as desigualdade válidas possíveis, quando empregadas em um algoritmo de planos de corte resultam em limitantes duais bastante fortes. É claro que, para isso, é preciso que o problema de separação, muitas vezes também pertencente à classe $\mathcal{NP}$-difícil, possa ser resolvido de forma eficaz na prática.

Surge, assim, a ideia de combinar os algoritmos de planos de corte com algoritmo de *branch-and-bound*, com a esperança de que os melhores limitantes duais gerados pelo primeiro possam tornar mais efetivas as podas realizadas por este último. Neste contexto, os algoritmos de planos de corte normalmente têm sua execução interrompida de forma prematura, ou seja, antes que seja provada a otimalidade da solução. Esta combinação dos dois algoritmos dá origem ao algoritmo de **Branch-and-Cut**.

Recapitulando, este tipo de algoritmo consiste em estender o algoritmo de **B&B** com um passo de melhoria do modelo através da adição de planos de corte. De forma mais detalhada, seja x^i a solução da relaxação linear do nó P_i. Se existe uma desigualdade

$\pi x \leq \pi_0$ violada por x^i (obtida resolvendo o problema da separação para uma família $\mathcal{F}$), adicione esta restrição ao modelo de P_i e resolva a relaxação linear novamente. Caso não existam mais desigualdades violadas, os passos usuais do **B&B** são executados: atualização de limitantes e geração de novos nós.

2.2 Modelo PLI para o MTWTC

Após uma breve introdução a **PL** e **PLI** já temos os resultados necessários para descrever e argumentar a corretude de um algoritmo exato para o MTWTC. Antes de darmos a formulação matemática do modelo vamos apresentar algumas definições auxiliares.

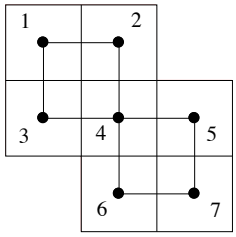
O Grafo de Conversões G'

Seja $G = (V, E)$ o grafo de subdivisão correspondente a uma instância do MTWTC, definido na seção 1.3. Definimos o grafo orientado $G' = (V', A)$ onde para cada aresta $e = \{i, j\}$ de G existem dois vértices ij e ji em V', um para cada orientação de e. Além disso, existe um arco entre dois vértices $u = ij$ e $v = xy$ de G' se e somente se $j = x$. Note que, deste modo cada arco de A representa uma conversão orientada em G, sendo o custo c_{uv} atribuído ao arco uv igual ao custo da conversão $(i, (j = x), y)$. O grafo G' é chamado de **grafo de conversões de** G. Na figura 2.5 são ilustrados, em 2.5a uma instância do MTWTC e o grafo de subdivisão correspondente. Em 2.5b temos os vértices e arcos de G' correspondentes as arestas 12 e 13 de G e, por fim, em 2.5c o grafo de conversões G'. A partir da descrição do parágrafo anterior, nota-se que a necessidade de definir o grafo de conversões decorre principalmente da maior facilidade que ele provê para a modelagem dos custos de conversão.

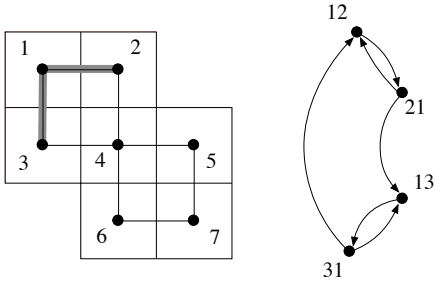
Façamos agora uma breve discussão sobre o tamanho do grafo de conversões G' com relação a G. Seja $|V| = n$ o número de vértices de G e $|E| = m$ o número de arestas de G. Temos que para cada aresta de G são criados dois vértices em G, logo $|V'| = n' = 2m$. Com relação ao número de arcos, seja $i \in V$ um vértice de G e d_i o grau de i. Todo par de arestas incidentes em i corresponde a duas conversões orientadas, que são dois arcos em G'.Além disso, também podemos realizar U-turns nos vértices num total de d_i arcos. Logo, para cada vértice i temos $2\binom{d_i}{2}$ arcos. Usando o fato de que o grafo de subdivisão é um grafo de grade parcial (grau máximo quatro) temos, no total,

$$\sum_{i \in V}(2\binom{d_i}{2} + d_i) \leq 2\sum_{i=1}^{n}(\binom{4}{2} + 2) \leq 12n + 4n \leq 16n$$

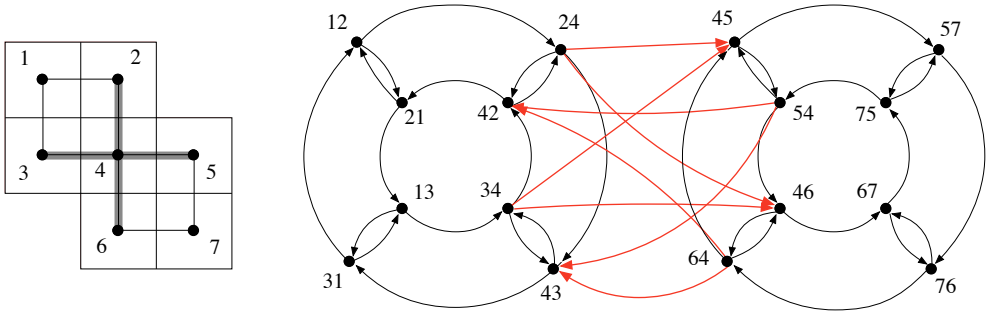
arcos e, portanto, o grafo de conversões G' tem tamanho linear com relação ao grafo de subdivisão G.



(a) Grafo de subdivisão de uma instância.



(b) Grafo de conversões parcial, das arestas $\{1,2\}$ e $\{1,3\}$.



(c) Grafo de conversões com destaque para os arcos das arestas incidentes no vértice 4.

Figura 2.5: Ilustração dos grafos de subdivisão e de conversões.

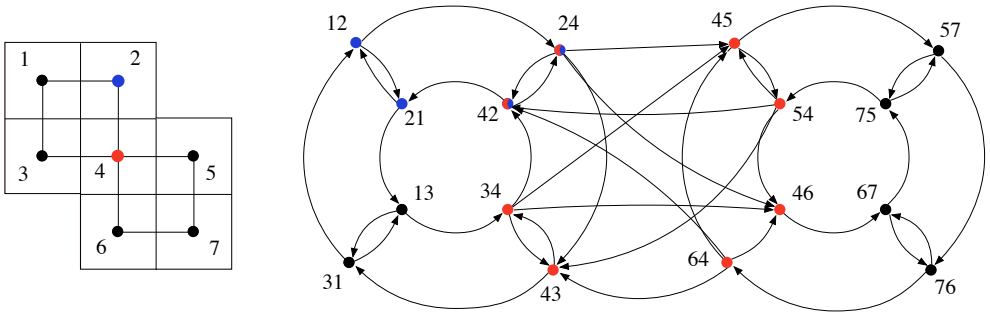


Figura 2.6: Conjunto dos representantes de G em G' para os vértices 2 e 4.

Representantes de G em G'

Para cada vértice $i \in V$ defina o conjunto C_i como sendo formado pelos vértices de G' que correspondem a arestas de G incidentes em i, ou seja, vértices da forma ix ou xi. Os conjuntos C_i são chamados de **representantes de G em G'**. Note que cada vértice ij de G' pertence a exatamente dois representantes, especificamente, C_i e C_j. Na figura 2.6 temos um grafo de subdivisão G, o grafo de conversões correspondente G' e os representantes dos vértices i e j destacados.

MTWTC como um Problema de Otimização em G'

Mostraremos a seguir que existe um problema equivalente ao MTWTC no grafo de conversões. Seja P uma solução para o MTWTC no grafo de subdivisão, ou seja, um passeio fechado que visita todos os vértices de G pelo menos uma vez. Na figura 2.7 está representando o passeio P em G e o passeio equivalente P' em G'. Percebe-se pelo exemplo que todo milling tour em G corresponde a um ciclo em G' que contém pelo menos um elemento representante de cada vértice $i \in G$. O custo do milling tour é a soma do custo dos arcos deste ciclo. Portanto em G', o MTWTC se torna o problema de encontrar um ciclo de custo mínimo que contenha pelo menos um elemento representante de cada vértice.

2.2.1 Algoritmo Exato para o MTWTC

Já enunciamos o MTWTC como um problema de otimização em $G' = (V', E')$. Vejamos agora um modelo de programação linear inteira para este problema. No modelo a seguir x_{uv} são variáveis inteiras não-negativas que contam o número de vezes que o arco $(u, v) \in E'$ é percorrido pela solução. Além disso, para cada vértice $w \in V'$ associamos a variável binária y_w, que vale 1 se o vértice está na solução e 0 caso contrário.

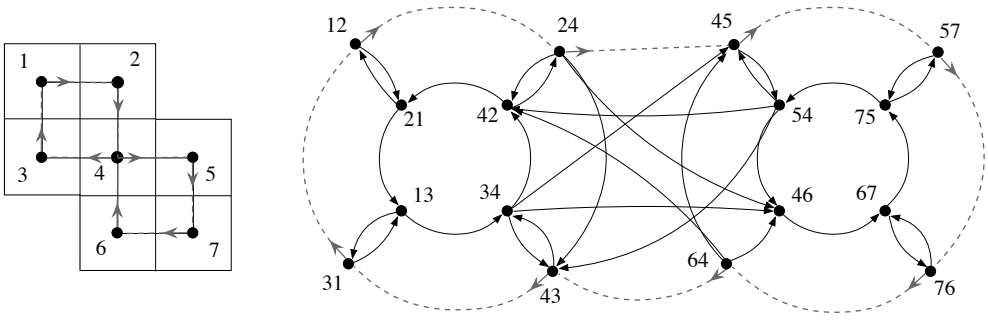


Figura 2.7: Correspondência entre soluções do G no G'.

A função objetivo é minimizar a soma total dos custos das conversões, isto é:

$$\min \sum_{(u,v)\in A} c_{uv} x_{uv} \tag{2.18}$$

O número de vezes que se entra em um vértice é igual ao número de vezes que se sai:

$$\sum_{(u,v)\in A} x_{uv} = \sum_{(v,u)\in A} x_{vu} \ , \ \forall \ u \in V'. \tag{2.19}$$

Todo vértice do grafo de subdivisão G é visitado no mínimo uma vez, isso implica que pelo menos dois vértices de cada conjunto C_i estão na solução:

$$\sum_{u\in C_i} y_u \geq 2 \ , \ \forall \ i \in V. \tag{2.20}$$

A restrição a seguir, impede que a solução contenha um arco saindo de um vértice que não está na solução:

$$x_{uv} \leq n y_u \ , \ \forall \ u \in V', \forall \ (u,v) \in A. \tag{2.21}$$

Note que, da forma como foi escrita, a restrição limita em $n = |V|$ o número máximo de vezes que um arco pode estar na solução, que é suficiente e não elimina soluções ótimas.

Isto porque, se uma solução percorre um arco mais do que n vezes, significa que, em algum momento, ela passa pelo arco porém não alcança nenhum vértice que ainda não foi visitado. Contudo, quando isso ocorre, existe uma solução de custo menor ou igual do que a solução corrente e que percorre o arco no máximo n vezes.

Somente essas restrições não são suficientes para a corretude do modelo, pois permitem soluções compostas por múltiplos passeios (*subtours*), como ilustrado na figura 2.8. Para eliminar esse tipo de solução, precisamos impor a conectividade à solução. Para isso, exigimos que para todo $U \subset V'$ com um vértice na solução e cuja intersecção com algum C_i é vazia, exista pelo menos uma aresta de U para $V - U$ na solução, ou seja,

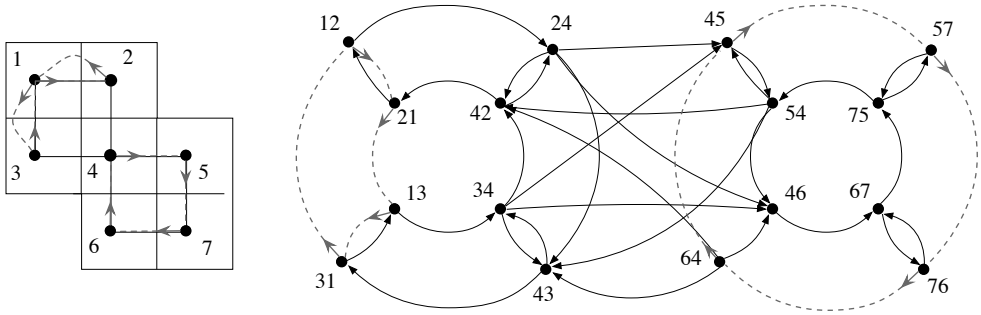


Figura 2.8: Solução inválida com dois passeios disjuntos: $1, 2, 1, 3, 1$ e $4, 5, 7, 6, 4$

$$\sum_{u \in U, v \notin U} x_{uv} \geq y_w \ , \forall \ U \subset V' : \exists \ i \in V : U \cap C_i = \emptyset, \ w \in U. \tag{2.22}$$

Por fim temos as restrições de domínio das variáveis:

$$x_{uv} \in \mathbb{Z}_+, \ y_w \in \mathbb{B} \ , \forall \ (u, v) \in A, \ w \in V'. \tag{2.23}$$

A dificuldade agora é que o número de restrições do tipo (2.22) para eliminar *sub-tours* é exponencial no tamanho do grafo de conversões G'. Logo, se tentarmos utilizar diretamente o algoritmo de **B&B** o tempo de construção da matriz de coeficientes e a quantidade de memória necessária já tornam o método inviável. Este fato nos leva à próxima escolha: empregar um algoritmo de **B&C**, no qual as desigualdades (2.22) são usadas como cortes.

Da seção 2.1.2 temos que dado um problema $IP = \min\{cx : Ax \leq b, x \geq 0, x \in \mathbb{Z}^n\}$ e uma família de desigualdades $\mathcal{F}$. No problema de separação de $\mathcal{F}$, dada uma solução $x^\star$ de IP, queremos encontrar uma desigualdade de $\mathcal{F}$ violada por $x^\star$, ou concluir que ela não existe.

Seja então $(x^\star, y^\star)$ uma solução para a relaxação linear do modelo contendo apenas um subconjunto das restrições (2.22). Se $(x^\star, y^\star)$ não satisfaz uma restrição (2.22), ausente no modelo corrente, então existe um $U \subset V'$ cuja intersecção com algum C_i é vazia e um vértice $w \in U$ tal que,

$$\sum_{u \in U, v \notin U} x^\star_{uv} < y^\star_w. \tag{2.24}$$

Deste modo, para resolver o problema de separação referente as restrições (2.22), queremos um algoritmo eficiente para encontrar um subconjunto U e um vértice w que satisfaz essas condições ou concluir que os mesmos não existem.

Para finalizar, o modelo completo e com todas as restrições é dado a seguir.

$$\min \quad \sum_{(u,v)\in E'} c_{uv}x_{uv} \tag{2.18}$$

$$\text{s.a} \quad \sum_{(u,v)\in E'} x_{uv} = \sum_{(v,u)\in E'} x_{vu} \quad \forall\ u \in V', \tag{2.19}$$

$$\sum_{u\in C_i} y_u \geq 2 \qquad \forall\ i \in \{1,\ldots,n\}, \tag{2.20}$$

$$\sum_{u\in U, v\notin U} x_{uv} \geq y_w \qquad \forall\ U \subset V' : \exists\ i \in \{1,\ldots,n\} : U \cap C_i = \emptyset,\ w \in U \tag{2.22}$$

$$x_{uv} \leq n y_u \qquad \forall\ u \in V', \forall\ (u,v) \in E', \tag{2.21}$$

$$x_{uv} \in \mathbb{Z},\ y_w \in \mathbb{B} \qquad \forall\ (u,v) \in E',\ w \in V'.$$

O Problema do Corte Mínimo

Seja $D = (V_D, A_D)$ um grafo orientado em que a cada arco $(u,v) \in A_D$ é atribuída uma capacidade c_{uv}. Sejam ainda S e T dois subconjuntos não-vazios e disjuntos de V_D. Da teoria dos grafos temos as seguintes definições. Um (S,T)-**corte** de D é uma partição de V_D em dois conjuntos S' e $T' = V - S'$, tal que $S \subseteq S'$ e $T \subseteq T'$. Dizemos que um (S,T)-corte **separa** S de T. Ademais, a capacidade de um (S,T)-corte é dada por:

$$c(S,T) = \sum_{u\in S', v\in T'} c_{uv}. \tag{2.25}$$

O problema do **corte mínimo** consiste em encontrar um (S,T)-corte cuja capacidade é mínima dentre todos os (S,T)-cortes de D. Um caso especial do problema do corte mínimo é quando S e T possuem exatamente um vértice. Essa versão foi estudada de maneira extensa pela literatura (cf [5] para ver os principais algoritmos e resultados sobre este problema).

Vejamos agora como o problema do corte mínimo geral pode ser reduzido para o caso com um vértice. Sejam $D = (V_D, E_D)$ um grafo orientado, $S \subset V_D$ e $T \subset V_D$ subconjuntos não-vazios e disjuntos de V_D. Para fazer a redução construímos um grafo como o da figura 2.9, ou seja, criamos dois vértices artificiais s,t em D e adicionamos os arcos (s,v) para $v \in S$ e (v,t) para $v \in T$. A capacidade dos novos arcos é definida como sendo infinita.

Podemos agora re-escrever o problema de separação das restrições de eliminação de *subtours* do MTWTC como uma série de problemas de corte mínimo no grafo de conversões com a capacidade dos arcos dada por $x^\star$. Em pseudo-código, a rotina de separação é dada

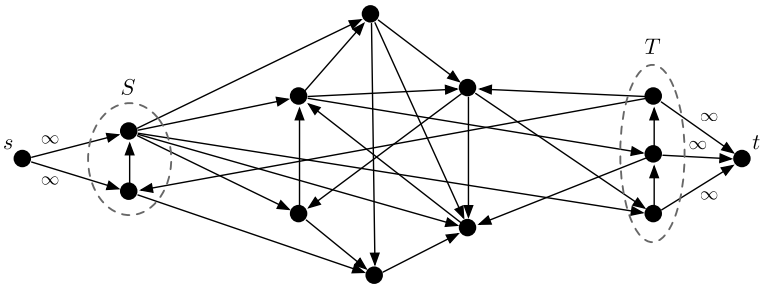


Figura 2.9: Redução do (S, T)-corte para a versão com somente um vértice em S e T.

no algoritmo 2.1. A função CORTE-MINIMO(S, T, D, f) retorna o (S, T)-corte mínimo no grafo $D = (V_D, A_D)$ com as capacidades dadas por f[1].

Vejamos em detalhes porque a rotina funciona. Seja $\mathcal{W} = \{w : w \in V \wedge y^\star_w > 0\}$, note que se $y_w = 0$ a restrição (2.22) é sempre satisfeita. Se encontrarmos um $(\{w\}, C_i)$-corte, para algum $w \in \mathcal{W}$ e $i \in V$ tal que $c(\{w\}, C_i) < y^\star_w$, então existe $S \supseteq \{w\}$ e $T \supseteq C_i$ para os quais a restrição (2.22) com $U = S$ não é satisfeita por $(x^\star, y^\star)$. Suponha agora que $(x^\star, y^\star)$ viola (2.22) para alguma tripla U, v, k. A rotina de separação em algum momento encontra um $(\{v\}, C_k)$-corte mínimo definido por $S \subset V'$, com custo $c(S, V' - U) \leq c(U, V' - U)$ e, portanto, também viola (2.22).

Algoritmo 2.1 Separação exata de restrições do tipo 2.22.

```
    function SEPARACAO-EXATA(x*, y*)
        for w ∈ V' do
3:          if y*_w > 0 then
                for i ∈ V do
                    if w ∉ C_i then
6:                      (δ, S') ← MINCUT({w}, C_i, G', x*)
                        if δ < y*_w then
                            { Restrição violada para um U = S' e w.}
9:                      end if
                    end if
                end for
12:         end if
        end for
    end function
```

A complexidade do algoritmo de separação depende do algoritmo utilizado para resolver o problema do corte mínimo. Seja $D = (V_D, A_D)$ um grafo orientado e $O(\text{MC}(V_D, A_D))$

[1]Em outras palavras, $f : A_D \to \mathbb{R}$

a complexidade de encontrar o corte mínimo de D. No caso deste trabalho esta complexidade corresponde àquela da implementação do *Network SIMPLEX*, constante das bibliotecas do pacote CPLEX [3].

Seja $G = (V, E)$ um grafo de subdivisão, $G' = (V', A)$ o grafo de conversões de G, $n = |V|$ e $m = |E|$. O algoritmo 2.22 verifica a condição da linha (3) para cada vértice w do grafo de conversões G'. Se verdadeira, para cada C_i com $i \in V$ que não contém w é executada uma rotina do corte mínimo sobre G'. Como cada vértice de G' está em exatamente dois representantes, a rotina de corte mínimo é executada $n - 2$ vezes para cada w. Portanto, a complexidade final é $O(mn\mathrm{MC}(m, m))$.

Um importante resultado da teoria de **PLI**, relaciona as complexidades computacionas dos problemas de otimização e separação. De maneira informal, um problema de otimização da forma $\max\{cx : x \in \mathrm{conv}(X)\}$ pode ser resolvido em tempo polinomial se e somente se o problema de separação para $\mathrm{conv}(X)$ pode ser resolvido em tempo polinomial. Esse resultado é demonstrado em [20, 19].

A rotina de separação para as restrições do tipo (2.22) no modelo para o MTWTC é polinomial. Sendo assim, pelo resultado descrito acima, temos que a relaxação linear do modelo **PLI** é resolvida em tempo polinomial, mesmo com o número de restrições sendo exponencial.

Capítulo 3

Algoritmos Aproximados e Heurísticas

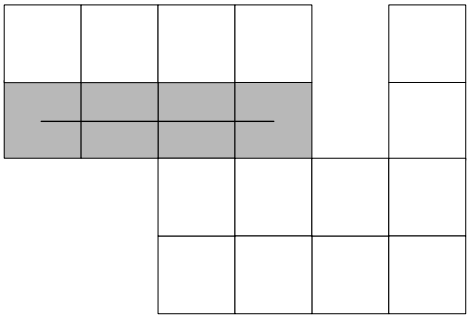
Vimos no capítulo 1 que os resultados mais importantes na resolução do MTWTC derivam dos algoritmos aproximados propostos em [6] para diferentes problemas de *milling*. Neste capítulo detalharemos o algoritmo aproximado específico para o MTWTC e descreveremos diversas abordagens heurísticas adotadas com o objetivo de melhorar os resultados práticos alcançados por ele.

Algumas das heurísticas são aperfeiçoamentos do algoritmo aproximado, na tentativa de explorar características do problema ignoradas pelo algoritmo original. Outras, são abordagens diferentes para a obtenção de soluções. Para cada uma delas vamos argumentar o porque ela foi desenvolvida, qual dificuldade ela tenta superar e, se for o caso, a qual fase do algoritmo aproximado ela se aplica.

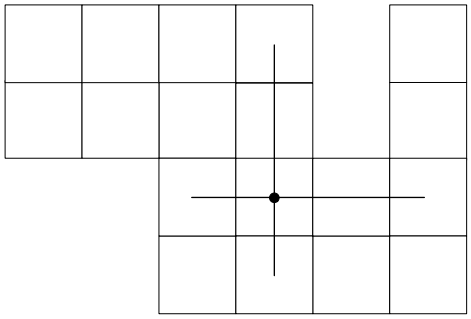
Para um bom entendimento do capítulo, recomendamos a leitura da seção 1.2.1 que contém definições e conceitos utilizados aqui.

3.1 Algoritmo Aproximado para o MTWTC

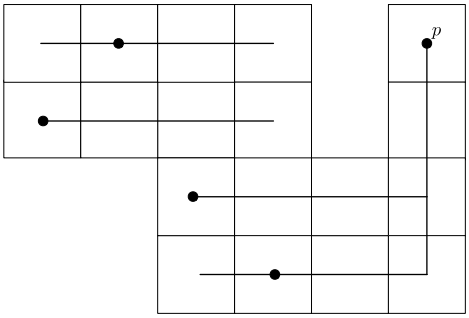
Nesta seção detalhamos um algoritmo aproximado (APX) para o MTWTC, descrito originalmente em [6]. Em termos gerais, o algoritmo pode ser dividido em três fases que são executadas em sequência, sendo a saída de uma fase a entrada da seguinte. São elas: 1) Cobertura por Faixas; 2) Cobertura por Ciclos e 3) União de Ciclos. Estes procedimentos são detalhados a seguir.



(a) Segmento ligando dois centróides e a faixa correspondente.



(b) Torre posicionada e os pixels atacados por ela.



(c) Cobertura por faixas e posicionamento de torres associado.

Figura 3.1: Ilustração dos conceitos de faixa, torre e a relação entre cobertura por faixas e posicionamento de torres.

Cobertura por Faixas - Fase 1

Um segmento de reta vertical ou horizontal maximal ligando os centróides de dois pixels e inteiramente contido no pocket define uma **faixa**. Uma faixa é composta por todos os pixels cujos centróides pertencem ao segmento que a define. Dizemos que a faixa **cobre** os pixels que a compõe. Um conjunto $\mathcal{S}$ de faixas é uma **cobertura** se a união da área coberta pelas faixas em $\mathcal{S}$ é exatamente a do pocket. Uma **torre** posicionada em um dado pixel p do pocket **ataca** o pixel $p' \neq p$ se existe um segmento horizontal ou vertical inteiramente contido no pocket que contém como extremidades os centróides dos pixels p e p'. Um **posicionamento de torres** é um conjunto de torres posicionadas de forma que nenhum par de torres se ataca. A figura 3.1 ilustra estes conceitos. Com essas definições podemos dar um resultado fundamental usado pelo APX.

Teorema 3. *Uma cobertura por faixas de cardinalidade mínima e um posicionamento de torres máximo tem o mesmo tamanho.*

Demonstração. Seja $B = (V_1, V_2, E)$ um grafo bipartido onde V_1 é o conjunto das faixas horizontais e V_2 das faixas verticais. Existe uma aresta $\{v_1, v_2\} \in E$ para $v_1 \in V_1$ e $v_2 \in V_2$ se as faixas tem um pixel em comum. É fácil ver que um emparelhamento máximo M em B corresponde a um posicionamento de torres máximo e uma cobertura por vértices mínima K corresponde a uma cobertura por faixas mínima. Portanto, pelo teorema de König-Egerváry (cf, [11]) M e K, tem a mesma cardinalidade. □

A primeira fase do APX consiste em encontrar uma cobertura por faixas mínima. Pelo teorema 3 isso pode ser feito em tempo polinomial com um algoritmo de emparelhamento máximo em grafo bipartido [11].

O resultado a seguir relaciona o valor ótimo do MTWTC com a cardinalidade de uma cobertura por faixas mínima. Para compreendê-lo, é conveniente que um *milling tour* seja visto como sendo a união de um conjunto de segmentos de reta horizontais e verticais maximais que descrevem de maneira única a trajetória do cortador. Vale notar que, nesta representação, supõe-se que uma meia-volta é representada por um segmento degenerado de comprimento nulo e que vai do ponto onde ocorre a meia-volta até ele próprio.

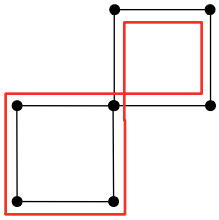
Teorema 4. *Seja $G = (V, E)$ uma instância do MTWTC, K uma cobertura por faixas mínima e $W^\star$ um* milling tour *ótimo de G. Ainda, seja S o conjunto de segmentos de reta horizontais e verticais maximais correspondendo a $W^\star$. Então, a cardinalidade de S é um limite inferior para o custo de $W^\star$.*

Demonstração. Seja $S = \{s_1, \ldots, s_k\}$ a lista de segmentos maximais correspondentes ao *milling tour* ótimo $W^\star$. A cada segmento $s_i \in S$ pode ser associada uma faixa f_i, o conjunto de faixas $F = \{f_1, \ldots, f_k\}$ é uma cobertura por faixas de G. Como K é uma cobertura mínima, $|K| \leq |F|$. Mas, como $|F| \leq \text{custo}(W^\star)$, chega-se ao resultado procurado. □

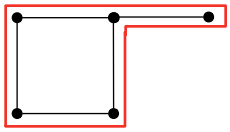
Observação: O exemplo da figura 3.2 ilustra o significado da última desigualdade da prova anterior. Na figura 3.2a, vê-se uma solução onde só existem conversões simples. Note que, neste caso, a quantidade de faixas correspondente ao tour é exatamente igual ao número de conversões, ou seja, ao custo. Na presença de U-turns, como é o caso da figura 3.2b, a quantidade de faixas é inferior ao número de conversões que, por sua vez é ainda menor do que o custo.

Cobertura por Ciclos - Fase 2

Uma **cobertura por ciclos** de um grafo $G = (V, E)$ é um conjunto de ciclos, tal que a união dos vértices visitados pelos ciclos é igual a V. A segunda fase do algoritmo APX consiste em transformar a cobertura por faixas da primeira fase em uma cobertura por



(a) Solução com somente conversões simples e custo igual ao número de faixas.



(b) Solução com $U-$ *turns* e custo maior que o número de faixas.

Figura 3.2: Explicação da prova do Teorema 4.

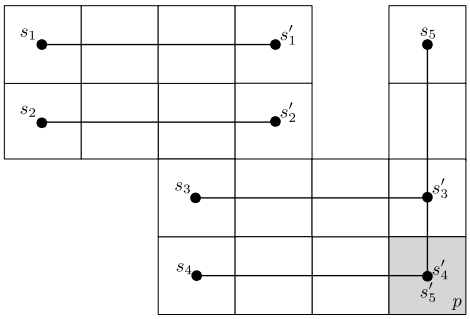
ciclos do grafo de subdivisão. Uma **cobertura por ciclos mínima** é aquela cuja soma do custo de conversão dos ciclos é a menor possível.

Seja $\mathcal{S}$ uma cobertura por faixas encontrada na fase 1 e $H = (V_H, E_H)$ um grafo completo onde V_H é o conjunto das extremidades das faixas em $\mathcal{S}$. Note que usamos vértices diferentes para extremidades representando o mesmo pixel em p. Também permitimos a existência de faixas degeneradas, em que ambas as extremidades são dadas pelo mesmo vértice, como pode ser observado no pixel p da figura 3.3a. Como existem exatamente dois vértices por faixa, a cardinalidade de V_H é par. A cada aresta $\{u, v\}$ de E_H atribuímos um peso w_{uv} cujo valor é igual ao tamanho do caminho mínimo (em custo das conversões) de u até v, exceto quando u e v são as extremidades de uma mesma faixa, situação esta em que atribui-se custo infinito à aresta (u, v).

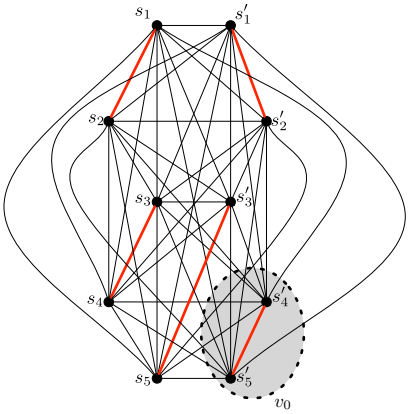
O próximo passo consiste em encontrar uma emparelhamento perfeito de custo mínimo em H, que sempre existe pois $|V_H|$ é par e H é um grafo completo. Seja M_H um emparelhamento perfeito de H, uma cobertura por ciclos pode ser construída fazendo a união das faixas de $\mathcal{S}$ com os caminhos representados pelas arestas de M_H. Na figura 3.3a temos uma cobertura por faixas com cinco faixas e em 3.3b o grafo H correspondente com o emparelhamento $M_H = \{(s_1, s_2), (s'_1, s'_2), (s_3, s_4), (s'_4, s'_5), (s'_3, s_5)\}$ e, em destaque, o pixel p representado duas vezes em H (vértices s'_4 e s'_5). A cobertura por ciclos, formada por dois ciclos e custo total 10 é dada na figura 3.3b. Note a necessidade de uma conversão do tipo meia-volta e que no ciclo final o pixel p é visitado exatamente uma vez.

Para encontrar um emparelhamento perfeito podemos utilizar o algoritmo de Edmonds [16] e construímos cobertura por ciclos a partir de M_H e $\mathcal{S}$ de maneira trivial, simplesmente percorrendo os vértices da faixa e do caminho representado pela aresta do emparelhamento.

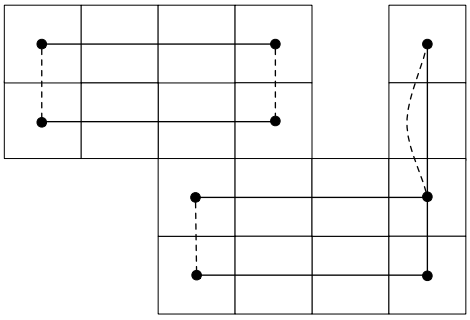
Em [6] é mostrado que todo milling tour T, pode ser descrito como a união de dois emparelhamentos perfeitos distintos M_1 e M_2 de um grafo $\mathcal{H}$ obtido de forma bastante



(a) Cobertura por faixas e pixel p.



(b) Grafo H da cobertura e emparelhamento perfeito.



(c) Cobertura por ciclos com dois ciclos e custo 10.

Figura 3.3: Cobertura por ciclos a partir de uma cobertura por faixas.

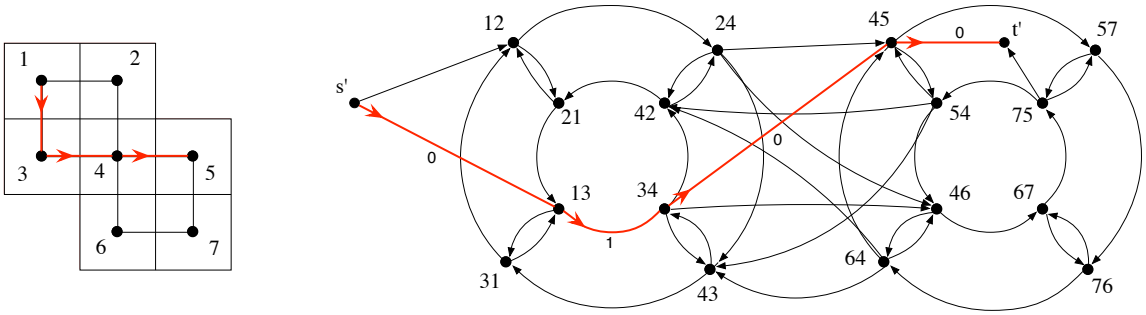


Figura 3.4: Caminho de custo mínimo em conversões do pixel 1 ao 5.

semelhante àquela que usamos para definir o grafo H anteriormente. Em $\mathcal{H}$ os vértices também estão associados a extremos de faixas e o peso de uma aresta equivale à distância mínima entre os pontos correspondentes, computada em termos de custos de conversões, conforme detalhado no parágrafo a seguir. Um resultado fundamental daquele artigo e que iremos usar na prova do Teorema 5 diz que, se $c_H(M_1)$ é o custo de M_1 em H e $c_H(M_2)$ o de M_2, então, $c(M_1) + c(M_2) \leq c(T)$. Aléms disso, o custo do emparelhamento perfeito de peso mínimo em H e $\mathcal{H}$ são idênticos.

Vejamos então como calcular o caminho mínimo em custo de conversões entre duas extremidades de faixas, o que permite atribuir o peso das arestas em H (e $\mathcal{H}$) e, consequentemente, definirmos o peso de um emparelhamento. Vamos utilizar para isto o grafo de conversões, definido no capítulo 2. Sabemos que no grafo de conversões G' para cada aresta $\{i, x\}$ incidente no vértice i do grafo de subdivisão G existem dois vértices ix e xi, um para cada orientação da aresta. Além disso, definimos C_i, o conjunto dos vértices de G' da forma ix e xi, chamado de representante de i. Sejam s e t vértices de G, o menor caminho em G corresponde, em G', ao menor caminho de um vértice da forma sx à um da forma xt. Podemos reduzir este problema, ao problema de caminhos mínimos clássico com uma fonte e um destino, bastante estudado na literatura e resolvido tradicionalmente com o algoritmo de Dijkstra [13]. Na redução criamos dois vértices s' e t', ligamos s' a todo vértice da forma sx e t' a todo vértice da forma xt. Os arcos criados têm custo associado igual a zero. Para ilustrar usaremos o grafo de conversões da figura 2.5. Na figura 3.4 adicionamos os vértices s' e t' ao grafo de conversões G' e destacamos o caminho mínimo do vértice 1 ao 5.

Teorema 5. *A cobertura por ciclos encontrada é uma* 2.5*-aproximação para o problema de encontrar cobertura por ciclos mínima.*

Demonstração. Seja S uma cobertura por faixas mínima e $z^\star$ o valor objetivo de um *milling tour* ótimo. Pelo teorema 4, $z^\star \geq |S|$.

Como as faixas são maximais, os extremos são *pixels* da borda do *pocket* e, portanto, qualquer *tour* T, realiza uma conversão ou cruza o pixel ortogonalmente à faixa do qual ele é extremo. Em ambos os casos, há um segmento ortogonal à faixa na solução.

Já mencionamos anteriormente que um tour pode ser entendido como a união de dois emparelhamentos perfeitos distintos M_1 e M_2 em $\mathcal{H}$. Portanto, se $c(M)$ é o custo do emparelhamento perfeito de custo mínimo de H, então, $c(M) \leq z^\star/2$.

Agora, seja M o emparelhamento perfeito de custo mínimo e S a cobertura por faixas mínima do grafo de subdivisão. Quando construímos a cobertura por ciclos a partir de M e S, o custo aumenta em no máximo 2 por faixa, portanto, $2|S| + c(M) \leq 2z^\star + z^\star/2 = 2.5z^\star$. □

União de Ciclos - Fase 3

O passo final do APX consiste em unir os ciclos da cobertura encontrada no passo anterior em um único ciclo, que corresponde a um *milling tour*. Para evitar confusões, convém chamar a atenção do leitor para o fato de que, aqui, estamos nos referindo aos ciclos no grafo de subdivisão resultantes da cobertura de ciclos no grafo H que foi construída na fase anterior. Em outras palavras, estamos supondo que já foram computados ciclos no grafo de conversões que correspondam aos ciclos obtidos em H na fase 2. Assim, mostraremos a seguir, como podemos unir dois ciclos do grafo de subdivisão, C_1 com custo em conversões t_1 e C_2 com custo t_2, em um único ciclo C de forma que o custo de C seja no máximo $t_1 + t_2 + 2$.

Diremos que dois ciclos se **intersectam** se eles possuem um vértice em comum e que são **adjacentes** se não se intersectam e possuem dois vértices adjacentes. Consideremos inicialmente o caso de dois ciclos C_1 e C_2 que se intersectam em um vértice w. Sejam u, u' os vizinhos de w em C_1 e v, v' os vizinhos de w em C_2. Construa o ciclo final C ligando u a v' através de w e v a u' também através de w, conforme ilustrado na figura 3.5. Note que cada uma dessas ligações gera no pior caso uma nova conversão simples.

Vejamos agora o casos de ciclos adjacentes. Sejam C_1 e C_2 ciclos adjacentes, w_1 o pixel de C_1 adjacente ao pixel w_2 de C_2. Sem perda de generalidade, podemos assumir que w_2 está abaixo de w_1 e é o pixel mais a esquerda. Por causa disso, existem somente três maneiras que o pixel w_2 pode ser visitado pelo ciclo C_2. Na figura 3.6 ilustramos essas três maneiras e como o ciclo pode ser quebrado em w_2 e ligado a w_1 formando um ciclo único juntamente com C_1 e aumentando o custo em no máximo dois.

O tratamento dessas duas situações, ciclos que se intersectam e ciclos adjacentes, é suficiente pois, se não existe nenhum par de ciclos com essas características, então a cobertura por ciclos deixou de visitar algum vértice, o que contraria a definição.

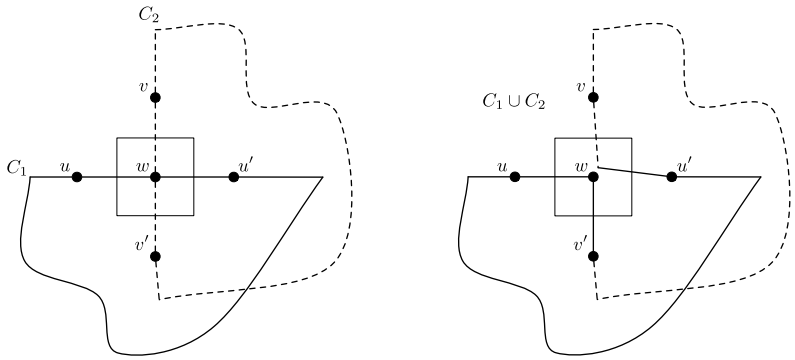


Figura 3.5: União de dois ciclos que se intersectam no pixel w.

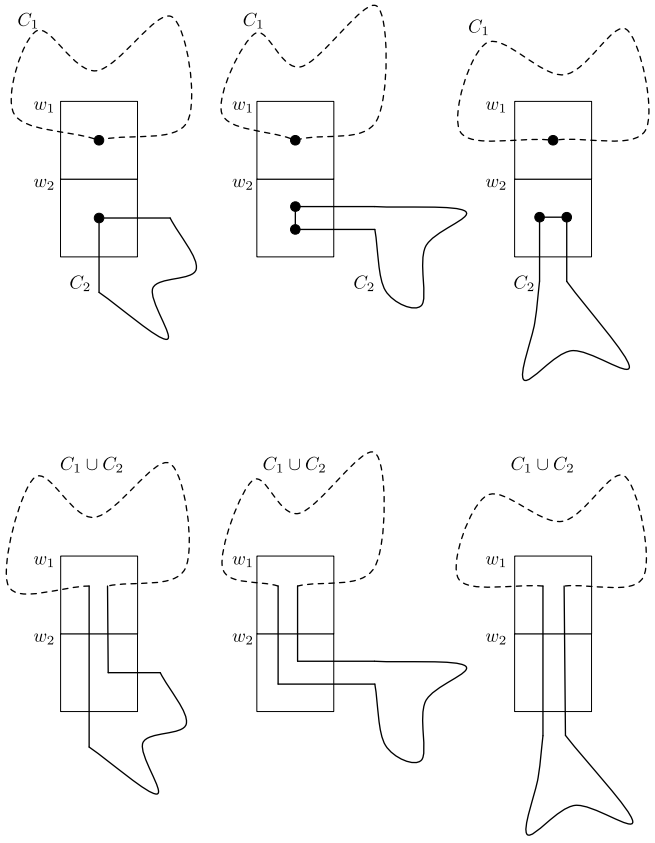


Figura 3.6: União de dois ciclos adjacentes.

Teorema 6. *Seja $\mathcal{C}$ uma cobertura por c ciclos com custo em conversões total igual a t. É possível construir um milling tour a partir de $\mathcal{C}$ com custo no máximo $t + 2(c-1)$.*

Demonstração. A prova é por indução no número de ciclos c. Se temos um ciclo, $c = 1$ e $t = t + 2(1-1)$ é trivialmente satisfeita. Para $c > 1$, suponha que o teorema é válido para $c - 1$. Seja $P \in C$, $C' = C \backslash P$, e t' o custo em conversões de C'. Pela hipótese indutiva, conseguimos fazer a união de $c-1$ ciclos em um ciclo W com custo $t' + 2(c-2)$. Suponha que P tenha custo em conversões r, logo $t = t' + r$. Como o grafo de subdivisão é conexo e $W \cup P$ cobre todos os vértices então W e P se intersectam ou são adjacentes. Nesses casos conseguimos unir W e P com um acréscimo no custo de no máximo duas unidades. Portanto, $T = W \cup P$ tem custo $t \leq t' + 2(c-2) + r + 2 \leq t + 2(c-1)$. □

Observe que pelo fato do grafo de subdivisão ser de grade parcial, cada ciclo tem no mínimo quatro conversões e, assim, $c \leq t/4$. Substituindo esta expressão no resultado do teorema 6, temos que o custo do milling tour é no máximo $t + 2(\frac{t}{4} - 1) \leq \frac{3}{2}t$.

Teorema 7. *O milling tour encontrado quando o grafo de subdivisão é de grade parcial é uma* 3.75*-aproximação do ótimo.*

Demonstração. Seja t o custo da cobertura por ciclos e $t^\star$ o custo da cobertura por ciclos ótima. Usando teorema 6 e o fato de que o grafo de subdivisão é de grade parcial, encontramos um milling tour com custo $z \leq \frac{3}{2}t$. O teorema 5 nos diz que $t \leq 2.5t^\star$ e, portanto, $z \leq \frac{3}{2}2.5t^\star = 3.75t^\star$. Seja $z^\star$ o custo do milling tour ótimo. Como um milling tour também é um cobertura por ciclos, temos que $t^\star \leq z^\star$ e, deste modo, $z \leq 3.75z^\star$. □

3.2 Melhorando o Desempenho do APX

No capítulo 4 é apresentada uma ferramenta de visualização de soluções e instâncias do MTWTC. Com o auxílio desta ferramenta, notamos alguns padrões sub-ótimos nas soluções geradas pelo algoritmo aproximado. Nas seções seguintes descrevemos em detalhes esses padrões e propomos algumas modificações heurísticas ao algoritmo aproximado com o objetivo de evitá-los.

Alteração no Custo do Emparelhamento

Na segunda fase do APX construímos um grafo completo ligando as extremidades das faixas escolhidas na primeira fase, definindo o custo das arestas como o custo do caminho mínimo em número de conversões entre as extremidades. Em seguida, encontramos um emparelhamento perfeito de custo mínimo neste grafo.

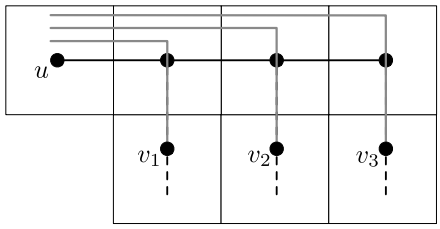


Figura 3.7: Ligando u a v_1, v_2 e v_3 com custo em conversões igual mas distância diferente.

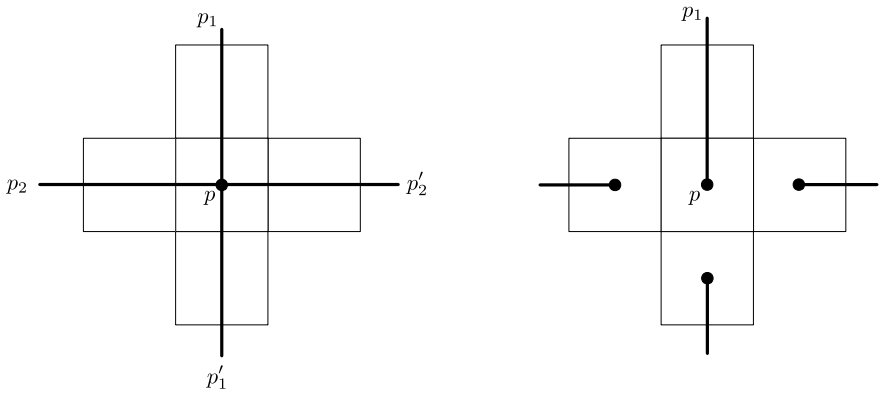
Por construção, a cobertura por faixas já é suficiente para visitar todos os vértices. Deste modo, esta fase do algoritmo simplesmente liga as faixas, sem que seja necessário cobrir novos vértices. Considerando o custo de conversões, as ligações ilustradas na figura 3.7 tem o mesmo custo, no entanto, a distância percorrida é diferente. Como o nosso objetivo é minimizar o custo em conversões, a princípio, esta diferença na distância é irrelevante. Porém, intuitivamente podemos perceber que a utilização do vértice mais próximo, deixa a solução visualmente mais simples, podendo até levar a uma redução no custo.

Para isto, utilizamos a seguinte fórmula que modifica o custo das arestas do grafo completo da segunda fase do algoritmo aproximado. Sejam u e v extremos de faixas que correspondem a vértices de H. Considere que a distância em conversões de u a w é t e a distância em número de arestas é d. O custo da aresta ligando u a v em H é dado por $t \times T + d$, onde T é a soma das distâncias em conversões para todo par $\{u, v\}$ de extremos de faixas. Claramente, desta forma uma solução S_1 com custo de conversões menor que uma outra solução S_2 sempre irá ter valor objetivo melhor, mesmo que o comprimento total dos seus segmentos seja maior.

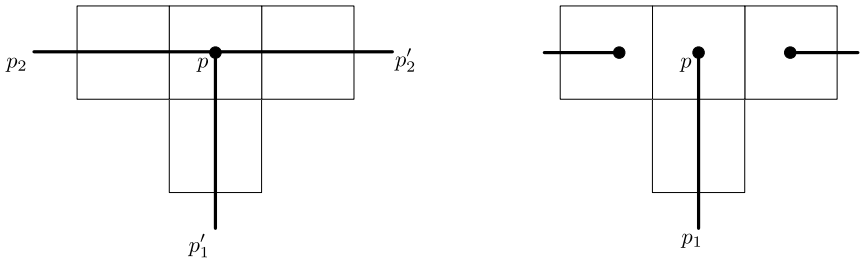
Partição ao invés de Cobertura

A primeira fase do APX consiste em encontrar uma cobertura por faixas do pocket. Podemos deixar o problema mais restrito e exigir que cada pixel seja coberto por exatamente uma faixa, isto é, queremos encontrar uma **partição** em faixas. Uma partição pode ser gerada a partir de uma cobertura através de dois procedimentos aplicados em sequência no conjunto de faixas.

Seja p um pixel coberto por duas faixas S_1 e S_2, onde p_1, p_1' são as extremidades de S_1 e p_2, p_2' as extremidades de S_2. Se p não é extremidade de S_1 então substitua a faixa por duas novas faixas com extremidades (p_1, p) e (p, p_1'). Faça o mesmo para S_2. Este processo é chamado de **quebra**, e é feito até que todo pixel seja coberto por exatamente uma faixa ou é extremidade de toda faixa que o cobre.



(a) Quebra e redução em um pixel não-extremo.



(b) Quebra e redução em um pixel extremo de umas das faixas.

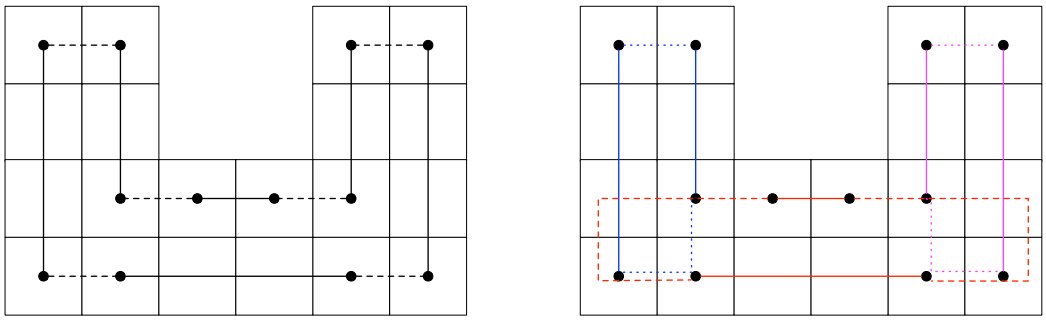
Figura 3.8: Os possíveis casos de quebra e redução num pixel p.

Após a quebra, realizamos o processo de **redução**. Seja p um pixel coberto múltiplas vezes e $(p_1, p), \ldots, (p_k, p)$ as extremidades das faixas que o cobrem. Para $i = 2, \ldots, k$, substitua (p_i, p) por (p_i, p'_i) onde p'_i é o pixel adjacente à p coberto pela faixa i. Este processo é repetido até que todo pixel seja coberto por exatamente uma faixa. A figura 3.8 ilustra os processos de quebra e redução.

Uma partição construída pelos processos de quebra e redução tem a vantagem de gerar novos pontos de ligação entre extremos de faixas (útil na próxima fase do APX) sem modificar o custo dos caminhos da cobertura e, possivelmente, gerando soluções melhores. Na figura 3.9 ilustramos duas coberturas por ciclos construídas a partir de uma partição. A da figura 3.9b corresponde àquela que seria obtida caso a heurística de partição por faixas não fosse executada.

União de Ciclos Gulosa

A última fase do APX consiste em transformar uma cobertura por ciclos em um único ciclo, que corresponde a uma solução do problema. Na seção 3.1 é descrita uma maneira



(a) Cobertura por ciclos, que só pode ser gerada a partir de uma partição.

(b) Cobertura por ciclos gerada por uma cobertura por faixas.

Figura 3.9: Coberturas por ciclos de uma partição em faixas.

de unir dois ciclos de forma que o custo aumente em no máximo dois. No entanto, ao analisar instâncias específicas de união de ciclos, percebemos que em alguns casos existem diversas maneiras com custos distintos de unir dois ciclos. Alguns padrões de união foram identificados e a rotina de união de dois ciclos foi modificada de forma a fazê-lo de maneira a minimizar o custo final da união. Na figura 3.10 estão ilustrados os padrões de união identificados e o incremento no custo de cada um. Note que em alguns casos o custo final é reduzido.

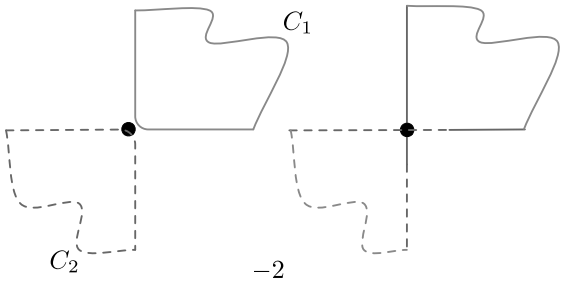
É possível minimizar o custo final por uma escolha apropriada da sequência de união de ciclos dois a dois de forma a minimizar o custo final. Utilizaremos um algoritmo heurístico que faz uma escolha gulosa de qual par unir a cada iteração. Assim, unem-se os dois ciclos cujo incremento no custo seja mínimo. Em caso de empate, a escolha é feita de forma arbitrária entre os melhores pares.

União de Ciclos via Emparelhamento

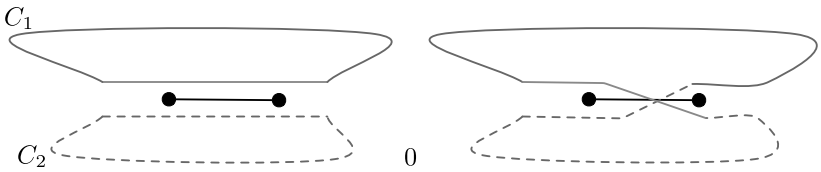
Uma outra maneira de fazer a união de r ciclos dado que sabemos unir os ciclos dois a dois é através de um algoritmo de emparelhamento.

Seja $G = (V, E)$ um grafo de subdivisão e $\mathcal{C} = \{C_1, \ldots, C_r\}$ um conjunto de ciclos que cobre G. Construa o grafo completo $H = (V_h, E_H)$ onde $V_H = \{1, \ldots, r\}$ e o custo da aresta $\{i, j\} \in E_H$ é igual ao custo(C_i, C_j), i.e, o incremento no custo para fazer a união dos ciclos C_i e C_j. Encontre um emparelhamento perfeito M_H de custo mínimo neste grafo e defina $\delta = \min\{\text{custo}(C_i, C_j) : \{i, j\} \in M_H\}$. Para cada aresta $\{i, j\} \in M_H$ faça a união dos ciclos se o custo da união é igual a δ e substitua C_i e C_j pela união em $\mathcal{C}$. Após este processo, a cardinalidade de $\mathcal{C}$ é reduzida em pelo menos um.

A união de ciclos via emparelhamento consiste em repetir o procedimento acima até



(a) Conversões simples opostas no mesmo vértice. Variação no custo: -2.



(b) Ciclos compartilham arestas. Variação no custo: *zero*.

Figura 3.10: Padrões especiais para a união de ciclos.

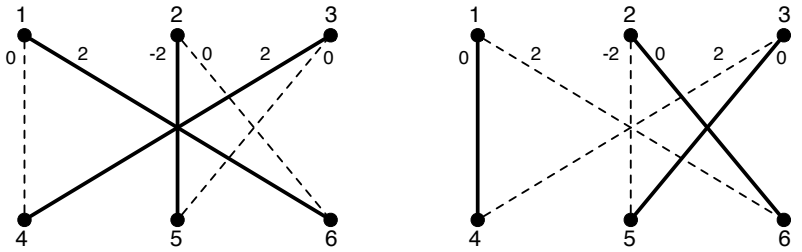
que o número de ciclos em $\mathcal{C}$ seja igual a um. A figura 3.11 exemplifica um caso em que a união por este algoritmo é melhor que a do guloso. No caso desta figura, o grafo representa uma solução onde $\mathcal{C}$ é constituída de seis ciclos e as arestas não desenhadas têm custo infinito. A união de C_2 com C_5 reduz o custo em 2 e é a primeira a ser escolhida pelo algoritmo guloso. No entanto, isto força duas uniões de custo 2 e, assim, o custo final é 2. No emparelhamento perfeito, a aresta $\{2, 5\}$ não é escolhida, e são feitas três uniões com custo zero.

3.3 Outras Heurísticas

Foram descritas quatro tentativas de melhoria do algoritmo aproximado APX, os resultados obtidos com cada uma delas serão descritos no capítulo 4. Nesta seção apresentamos duas heurísticas que não se enquadram como simples mudanças em algum passo específico do APX e, como consequência, não mantém a garantia de aproximação.

3.3.1 Heurística do Jardineiro

Nesta seção apresentamos uma heurística simples porém eficiente (dentro de suas limitações, discutidas no capítulo 5), chamada de **Heurística do Jardineiro**, pois foi



(a) União gulosa faz união 3 com 4 forçando custo dois.

(b) União via emparelhamento, custo da união é zero.

Figura 3.11: Comparação da união de ciclos gulosa com a via emparelhamento.

inspirada na forma gulosa que um humano tentaria resolver o problema, aqui imaginado no contexto da aplicação do corte de grama.

A entrada da heurística é o polígono de subdivisão P, e um pixel v_0 pertencente a borda do polígono, chamado de **origem**. Seja p o pixel onde o cortador está posicionado quando se inicia uma iteração do algoritmo (na primeira iteração, o cortador se encontra na origem). A partir dessa posição o cortador se move no sentido e direção em que o maior número de vértices não cobertos possam ser visitados em um único movimento, sempre terminando em um pixel não coberto. Caso não seja possível se mover dessa forma, o cortador se move até o pixel não coberto mais próximo (em custo de conversões) ou a origem caso todos os pixels tenham sido cobertos.

Na figura 3.12 são ilustradas quatro iterações da heurística. Inicialmente o cortador se move da origem v_0 para a posição v_1, cobrindo quatro vértices. Em seguida se move para v_2, v_3 até a posição corrente p. A partir de p o cortador precisa executar um movimento do segundo tipo, até o vértice mais próximo em conversões. A princípio, o algoritmo escolhe arbitrariamente um pixel entre v ou v' (a rigor também poderia ser selecionado o pixel imediatamente acima de v').

Para melhorar o eficiência da heurística, utilizamos técnicas da meta-heurística GRASP [30], especificamente, aleatorização e múltiplos reinícios. Em nossa implementacão múltiplos reinícios são feitas através da escolha de origens distintas. Empregamos aleatorização em dois passos do algoritmo: na escolha do movimento e na escolha do próximo pixel a ser visitado quando nenhum movimento atinge um pixel não coberto. Em ambos os casos a escolha é feita sobre uma lista restrita de elementos. No primeiro caso, como o número máximo de movimentos é quatro, a lista tem um tamanho fixo com os dois movimentos que cobrem mais pixels. No segundo, a lista tem os $\lceil 0.01n \rceil$ pixels não cobertos mais próximos do atual, onde n é o total não cobertos. Fazemos a escolha de maneira uniforme nos dois casos.

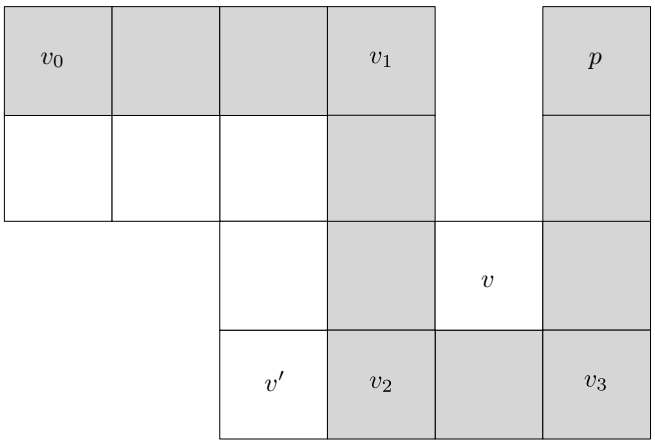


Figura 3.12: Quatro iterações da Heurística do Jardineiro, p é o pixel atual e v ou v' é o próximo pixel a ser visitado, através do caminho de custo mínimo em conversões, considerando pixels já visitados.

Note que não descrevemos nenhuma busca local, outro elemento importante e bastante utilizado pelo GRASP. Embora não tenhamos desenvolvido uma busca local específica para esta heurística, na Seção 3.4 serão descritas três vizinhanças de busca local.

3.3.2 Heurística de Cobertura

Descrevemos a seguir uma heurística desenvolvida em conjunto com o graduando Bruno Crepaldi em seu trabalho de Iniciação Científica, publicado posteriormente [15]. O principal ingrediente do algoritmo é o uso dos mesmos passos básicos do APX combinados com a resolução de modelos de **PLI**, o que permite classificá-lo como uma *matheuristic*, termo da língua inglesa usado para cunhar as heurísticas com estas características.

A motivação dessa heurística partiu da observação visual de que as soluções do APX frequentemente têm sua qualidade deteriorada pela presença excessiva de meias-voltas (U-turns). Esta dificuldade aparece na transição da fase 1 para a fase 2 do algoritmo aproximado quando a cobertura por faixas é transformada em uma cobertura de ciclos no grafo de subdivisão G. Nossas análises nos conduziram à conclusão de que soluções melhores poderiam ser alcançadas se a construção dos ciclos fosse feita diretamente, dispensando a execução da fase 1 do APX original. A alternativa encontrada foi utilizar a **PLI** para computar diretamente uma cobertura de ciclos de G que, de alguma forma, levasse a uniões de ciclos menos custosas na fase 3 do APX. Sendo assim, a heurística da cobertura pode ser vista como uma modificação do APX em que favorecemos o desempenho do algoritmo na prática, abrindo mão da prova teórica de aproximação. A seguir detalhamos o modelo **PLI** empregado.

O pocket P de uma instância do MTWTC pode ser visto como a união de N pixels definidos por quadrados unitários com vértices de coordenadas inteiras e lados paralelos aos eixos. Dessa forma, podemos representar P como uma matriz booleana A de dimensão $m \times n$, onde m é a altura e n a largura do menor retângulo que contém P.

Seja p_{ij} as coordenadas do centróide do pixel que corresponde a célula (i, j) da matriz A e $r_{p_{ab},p_{cd}}$ o **retângulo** cujo vértice superior esquerdo é dado por p_{ab} e o vértice inferior direito por p_{cd}. Um retângulo é dito **próprio** se $a < c$ e $b < d$. Quando $a = c$ ou $b = d$ o retângulo corresponde a uma **faixa** (não necessariamente maximal) e é dito ser *degenerado*.

O conjunto $V(r_{r_{ab},p_{cd}})$ de pixels que pertence a $r_{p_{ab},p_{cd}}$ é dado por:

$$\begin{aligned} V(r_{p_{ab},p_{cd}}) = \{&p_{ab}, p_{a(b+1)}, \ldots, p_{a(d-1)}, \\ &p_{ad}, p_{(a+1)d}, \ldots, p_{(c-1)d}, \\ &p_{cd}, p_{c(d-1)}, \ldots, p_{c(b+1)}, \\ &p_{cb}, p_{(c-1)b}, \ldots, p_{(a+1)b}\}, \end{aligned}$$

i.e, o retângulo é composto pelos pixels que pertencem a sua borda.

Dizemos que um retângulo $r_{p_{ab},p_{cd}}$, $r_{p_{ab},p_{cd}} \subseteq P$ se e somente se $\forall\ p_{ij} \in V(r_{p_{ab},p_{cd}})$, $p_{ij} \in P$. Note que uma faixa $r_{p_{ab},p_{cd}}$ é **maximal** se ela pertence a P e não existe uma faixa $r_{p_{a'b'},p_{c'd'}}$ que pertence a P e $V(r_{p_{ab},p_{cd}}) \subset V(r_{p_{a'b'},p_{c'd'}})$.

Seja $\mathcal{R} = \{r_{p_{ab},p_{cd}} \mid r_{p_{ab},p_{cd}} \subseteq P,\ a < c,\ b < d \text{ e } a, b, c, d \in \mathbb{N}\}$ o conjunto de retângulos próprios contidos em P e $\mathcal{S}$ o conjunto de faixas maximais de P.

Um conjunto de retângulos $R \subseteq \mathcal{R} \cup \mathcal{S}$ onde todo pixel $p \in P$ está em pelo menos um retângulo R é chamado de **cobertura por retângulos**. Uma cobertura por retângulos de cardinalidade mínima é chamada de **cobertura por retângulos mínima**. Podemos encontrar uma cobertura por retângulos mínima com o seguinte modelo de **PLI**. A cada retângulo $r \in \mathcal{R} \cup \mathcal{S}$ associamos uma variável binária x_r tal que $x_r = 1$ se o retângulo r está na cobertura e $x_r = 0$ caso contrário. Para cada par $p \in P$ e $r \in \mathcal{R} \cup \mathcal{S}$ definimos uma entrada na matriz de coeficientes $A = a_{pr}$ de forma que $a_{pr} = 1$ se o pixel p está contido no retângulo r e $a_{pr} = 0$ caso contrário.

A partir daí, obtemos o seguinte modelo matemático para o problema de cobertura de P por retângulos:

$$\begin{aligned} \min \quad & \sum_{i \in \mathcal{R} \cup \mathcal{S}} x_i & & \\ \text{s.a} \quad & \sum_{i \in \mathcal{R} \cup \mathcal{S}} a_{pi} x_i \geq 1 & \forall\ p \in P, & \qquad (3.1) \\ & x_i \in \mathbb{B} & \forall i, j \in \mathcal{R} \cup \mathcal{S}. & \end{aligned}$$

A restrição (3.1) exige que cada pixel P seja esteja em pelo menos um retângulo da solução. Este é um modelo clássico do problema de cobertura de conjuntos (*set covering*), bastante estudado na literatura [34] e que em geral são resolvidos rapidamente pelos modernos resolvedores de **PLI**.

Em geral, a solução encontrada por este modelo não é válida para o MTWTC. Porém, utilizando os algoritmos descritos anteriormente para efetuar a união de ciclos (fase 3 do APX), podemos obter um *milling tour*. Isto ocorre porque os retângulos encontrados pelo **PLI** nada mais são do que ciclos no grafo de subdivisão. Vale observar que, dada uma cobertura de retângulos R com $|R| = c$, pelo Teorema 6, podemos transformá-la em um *milling tour* com custo de no máximo $4c + 2(c-1) = 6c-2$, já que cada ciclo (retângulo) tem custo 4.

A formulação **PLI** anterior constitui-se no nosso modelo básico para substituir as fases 1 e 2 do APX original. Contudo, observamos em testes preliminares que os retângulos oriundos da solução deste modelo nem sempre correspondem àqueles que podem ser melhor aproveitados na fase 3. Assim, modificamos este modelo de modo a fazer com que ele minimize o custo de conversões da cobertura e priorize retângulos promissores para serem unidos na fase posterior. A seguir explicamos como isso foi feito.

Primeiramente criamos novas variáveis binárias. Dados dois pares de retângulos distintos i e j de $\mathcal{R} \cup \mathcal{S}$ associamos uma variável binária y_{ij} tal que $y_{ij} = 1$ se é possível unir i e j reduzindo o custo. A situação típica em que isso é possível ocorre quanto dois retângulos r_1 e r_2 compartilham um mesmo vértice, digamos x, em que os ângulos retos associados às conversões de r_1 e r_2 em x são opostos, como ilustrado na Figura 3.10a. Neste caso, o custo dos dois retângulos antes da operação de união era oito, sendo decrementada para seis depois da realização da *transformação em cruz* (ver Seção 3.4.2).

Considere o conjunto $\mathcal{C}$ de possíveis posições para um vértice extremo de um retângulo. Ou seja, defina $\mathcal{C} = \{SE, SD, ID, IE\}$ onde cada elemento corresponde aos mnemônicos (Superior Esquerdo, Superior Direito, Inferior Direito, Inferior Esquerdo), conforme ilustrado na figura 3.13. Para qualquer par de retângulos $i, j \in \mathcal{R} \cup \mathcal{S}$ e $s \in \mathcal{C}$, associamos a constante b_{ij}^s com valor 1 se e somente se é possível unir i com j utilizando o pixel do extremo s do retângulo i.

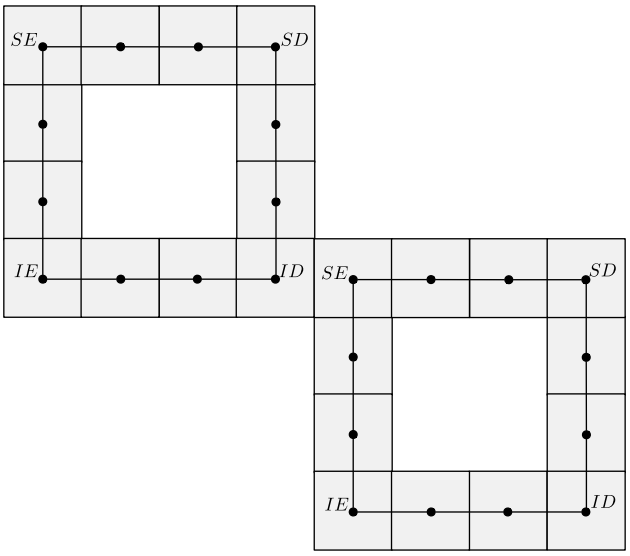


Figura 3.13: Ilustração dos extremos de um retângulo e como dois retângulos se unem pelos extremos.

Uma vez definidas as variáveis y e as constantes b, temos o novo modelo dado por:

$$\begin{aligned}
\min \quad & \sum_{i \in \mathcal{R} \cup \mathcal{S}} \left(4x_i - \sum_{j \in \mathcal{R} \cup \mathcal{S}} y_{ij} \right) & & \\
\text{s.a} \quad & \sum_{i \in \mathcal{R} \cup \mathcal{S}} a_{pi} x_i \geq 1 & \forall\ p \in P, & \quad (3.2) \\
& \sum_{j \in \mathcal{R} \cup \mathcal{S}} b_{ij}^{s} y_{ij} \leq 1 & \forall\ i \in \mathcal{R} \cup \mathcal{S} \cup \mathcal{S}, s \in \mathcal{C} & \quad (3.3) \\
& b_{ij}^{SE} y_{ij} = b_{ji}^{ID} y_{ji} & \forall\ i, j \in \mathcal{R} \cup \mathcal{S} \cup \mathcal{S} & \quad (3.4) \\
& b_{ij}^{SD} y_{ij} = b_{ji}^{IE} y_{ji} & \forall\ i, j \in \mathcal{R} \cup \mathcal{S} \cup \mathcal{S} & \quad (3.5) \\
& y_{ij} \leq \left(\sum_{s \in S} b_{ij}^{s} \right) x_i & \forall\ i, j \in \mathcal{R} \cup \mathcal{S} \cup \mathcal{S} & \quad (3.6) \\
& y_{ij} \leq \left(\sum_{s \in S} b_{ij}^{s} \right) x_j & \forall\ i, j \in \mathcal{R} \cup \mathcal{S} \cup \mathcal{S} & \quad (3.7) \\
& x_i, y_{ij} \in \mathbb{B} & \forall i, j \in \mathcal{R} \cup \mathcal{S}, s \in \mathcal{C}. &
\end{aligned}$$

As restrições (3.3) garantem que cada extremo de um retângulo seja usado em no máximo uma união. As restrições (3.4) e (3.5) garantem que ambos os extremos utilizados por uma união ij sejam marcados corretamente. Restrições (3.6) e (3.7) garantem que a união de i e j só ocorre se ela é possível e ambos os retângulos fazem parte da solução. A nova função objetivo reflete o fato de que um retângulo tem custo quatro enquanto que

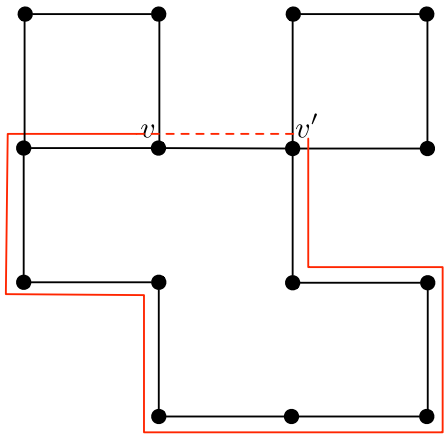


Figura 3.14: Exemplo de ciclo de ponte construído a partir de dois vértices adjacentes v e v'.

uma união de dois retângulos com um canto em comum representando conversões opostas permite uma redução de duas unidades no custo.

Podemos melhorar o modelo ainda mais se utilizarmos um terceiro conjunto de ciclos além dos retângulos ($\mathcal{R}$) e das faixas ($\mathcal{S}$). O uso desse conjunto adicional de ciclos se deve à observação de que havia uma perda de qualidade na solução heurística sempre que o grafo de subdivisão continha muitas arestas que não pertenciam aos retâgulos em $\mathcal{R}$. Note que, nesse caso, o único jeito de cobrir tais arestas é por meio de faixas maximais o que, como dito antes, pode levar à criação de muitas meia-voltas no *milling tour* final, deteriorando o seu custo. Os ciclos adicionais foram então criados como descrito a seguir.

Sejam $p_i, p_j \in P$ dois pixels adjacentes de P. Encontre o caminho de menor custo de conversões entre p_i e p_j que não use a aresta entre p_i e p_j. Agora, adicione essa aresta ao caminho calculado para obter o ciclo C. Se C não é um retângulo então ele é chamado de **ciclo de ponte**. A figura 3.14 exemplifica um ciclo de ponte. Seja $\mathcal{B}$ o conjunto de todos os ciclos de ponte de P.

Podemos incluir o conjunto $\mathcal{B}$ em nosso modelo **PLI** anterior, adicionando variáveis x_i para cada $i \in \mathcal{B}$ e constantes a_{ip} para cada para $p \in P$ e $i \in \mathcal{B}$. Atualizamos a função objetivo e as constantes a_{ip} de modo a refletir a inclusão das novas variáveis. Em nossas implementações decidimos não gerar restrições que envolvem uniões entre ciclos de ponte com os demais ciclos. Essa decisão foi motivada pela observação de que, ao fazer isso, obtínhamos um modelo mais difícil de ser resolvido e sem que nenhum ganho de qualidade fosse notado.

Resultados experimentais preliminares mostraram que a heurística da Cobertura provê melhores resultados quando comparada à heurística do Jardineiro. No entanto, a partir do

momento que aplicamos as rotinas de busca local (explicadas na seção seguinte) a todas soluções geradas pela heurística do Jardineiro, e não só a melhor delas, os resultados se inverteram. Concluímos com isto que não existe uma relação direta entre o custo da solução antes de realizar a busca local e o custo da solução obtida depois de sua aplicação. A vantagem da heurística do Jardineiro parecia ser decorrente do fato de que ela gerava um grande conjunto de soluções.

Inspirados nessa observação desenvolvemos uma extensão da heurística da Cobertura cuja ideia básica é, utilizando modelos de **PLI** distintos, obter múltiplas soluções e realizar a busca local em cada uma delas.

Seja T o número de soluções que queremos obter e IP^k o modelo da iteração k para $k = 1, \ldots, T$. No modelo IP^k o domínio das variáveis x_i é substituído por $x_i \in \mathbb{B}, \forall i, j \in \mathcal{R}^k \cup \mathcal{B}^k \cup \mathcal{S}$. Os conjuntos $\mathcal{R}^k \subset \mathcal{R}$ e $\mathcal{B}^k \subset \mathcal{B}$ são compostos por elementos escolhidos aleatoriamente de forma que $|\mathcal{R}^k| = \alpha\mathcal{R}$ e $|\mathcal{B}^k| = \beta\mathcal{B}$ onde α e β são constantes reais entre 0 e 1. O conjunto de faixas maximais $\mathcal{S}$ é sempre utilizado por completo de forma a garantir que a relaxação linear de IP^k tenha pelo menos uma solução básica.

Para esta versão da heurística, utilizamos o modelo, constituído unicamente pelas restrições (3.1) pois, por ser mais simples, seu custo computacional é menor. Com isto, pode-se resolver mais **PLI**s, i.e., obter mais soluções sobre as quais se aplicar as heurísticas de busca local. No próximo capítulo, onde relatamos os nossos experimentos, veremos que esta estratégia foi bem sucedida.

3.4 Heurísticas de Busca Local

Seja $G = (V, E)$ um grafo de subdivisão de uma instância para o MTWTC e W um milling tour de G. Chamamos de **vizinhança** uma função aplicada a W, que retorna um tour $f(W)$ com custo menor ou igual ao de W, ou o próprio W se a não é possível reduzir o custo segundo os critérios da função. Ao aplicarmos consecutivamente uma operação qualquer f sobre um tour W até que o *tour* corrente não seja alterado pela função, obtemos a sequência $\mathbf{W} = (W_1, \ldots, W_t)$ onde $W_0 = W$, $W_i = f(W_{i-1})$ para $i = 1, \ldots, t$ e $W_t = W_{t-1}$. Por definição, custo$(W_i) \leq$ custo(W_j) se $i \geq j$ e W_t é um mínimo local (ou **tour minimal**) com relação a operação f. Essa sequência é chamada **sequência de** f.

A seguir, definimos três vizinhanças que compõem o que chamamos de rotina de busca local, que é executada nas soluções dos algoritmos aproximado e heurísticos com o objetivo de melhorar a solução através de modificações locais no tour.

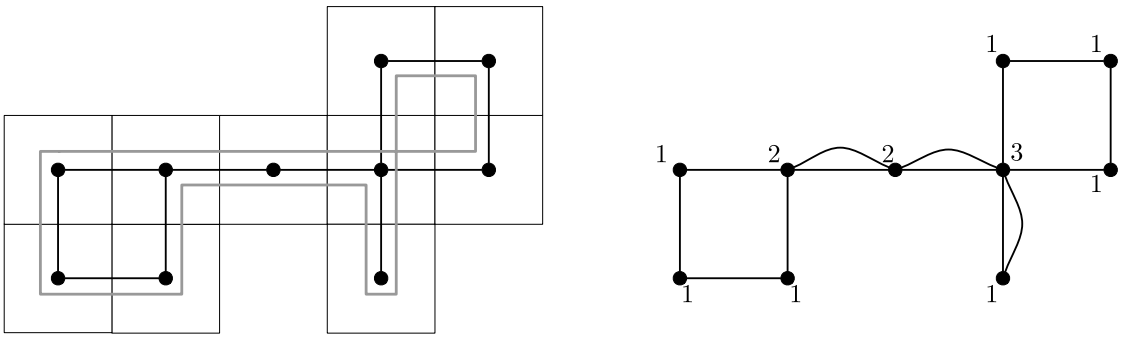


Figura 3.15: Milling tour W, o grafo da solução correspondente e o valor de cobertura de cada vértice.

3.4.1 Vizinhança do Passeio Euleriano

Dado um grafo de subdivisão $G = (V, E)$, seja $W = (v_1, \ldots, v_r)$ um milling tour de G e $G' = (V', E')$ o grafo de conversões associado. Defina $H = (V, E_H)$ como o subgrafo de G gerado a partir do tour W em que $E_H = \{\{v_i, v_{i+1}\} : 1 \leq i < r\, v_i = v_{r+1}\}$ e corresponde à coleção das arestas visitadas por W. O valor c_v associado a todo $v \in V$ é chamado de **valor de cobertura** de v é definido como sendo o número de vezes que o vértice v é visitado pelo tour W. O grafo H é chamado **grafo da solução** de W. Na figura 3.15 temos um exemplo de milling tour, o grafo de solução correspondente e o valor de cobertura dos vértices.

Note que H é conexo, possivelmente com arestas múltiplas, e Euleriano pois foi gerado a partir de uma solução para o MTWTC. Assim como, o tour W corresponde a um passeio Euleriano em H.

Também é verdade que para todo passeio Euleriano em H existe um milling tour correspondente em G com custo dependente do passeio. A figura 3.16 ilustra essa afirmação. Com base nisso, podemos definir o problema de, dado o grafo de solução, encontrar o passeio Euleriano cujo milling tour correspondente é de custo mínimo.

Seja $W = (u_1, \ldots, u_s)$ o milling tour correspondente a um passeio Euleriano qualquer de H e $R = (u_i, \ldots, u_j)$ com $u_i = u_j$ uma subsequência consecutiva de W em que $c_{u_k} > 1$ para $k = i, \ldots, j$ e, para todo par (i_1, i_2) com $i < i_1, i_2 < j$ e $i_1 \neq i_2$, temos $u_{i_1} \neq u_{i_2}$. Podemos substituir R por u_i para construir um novo tour W' cujo custo é menor que o de W. A operação acima é chamada de E*uler*. Note que a escolha de R não é única e que escolhas diferentes podem levar a resultados distintos. Em nossa implementação, elegemos a subsequência com menor i_1 e, em caso de empate, decidimos pela de i_2 mínimo.

A figura 3.17 ilustra um tour inicial $W = (1, 2, 6, 2, 3, 7, 3, 4, 8, 7, 10, 9, 6, 5, 1)$, um possível tour E*uler*(W) onde a subsequência $(2, 6, 2)$ de W é substituída por 2 e um tour

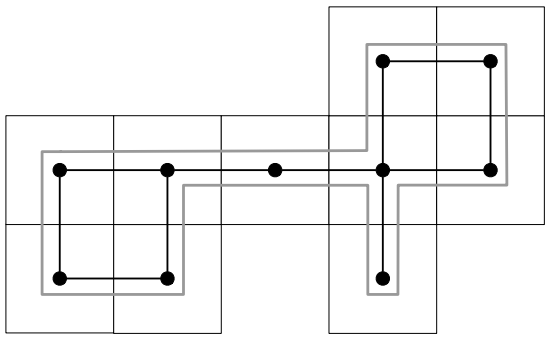


Figura 3.16: Outra opção de milling tour do grafo de solução da figura 3.15.

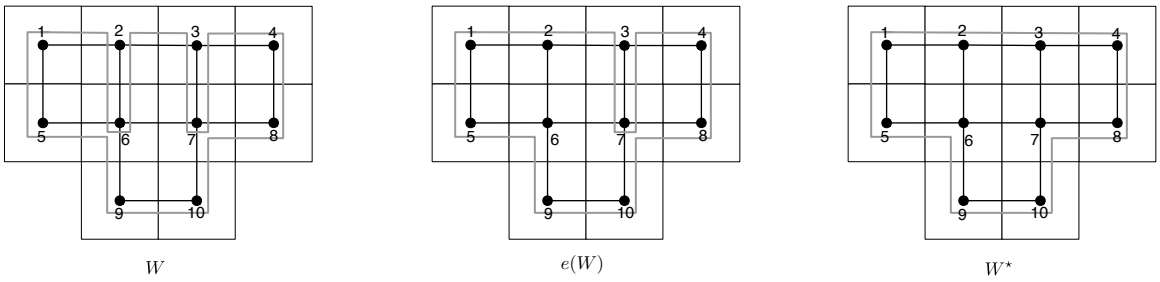


Figura 3.17: Tour inicial W, o tour E$uler(W)$ e o tour minimal $W^\star$.

minimal $W^\star$ após a substituição de $(3, 7, 3)$ por 7 em E$uler(W)$.

A **Vizinhança do Passeio Euleriano** é uma vizinhança que, dado um tour inicial W, constrói o grafo da solução, H, de W e gera um conjunto de tour, $\mathcal{W}$, a partir de um subconjunto dos tour Eulerianos de H escolhidos aleatoriamente. Para cada tour $W' \in \mathcal{W}$, a heurística encontra o tour minimal da sequência de E$uler$ para W' e, dentre todos eles, retorna o de menor custo. Essa operação é chamada de E$uler'$.

Tendo em vista a vizinhança acima, uma questão interessante é decidir, dado um tour W e o grafo de solução correspondente H, quando a aplicação da heurística acima é irrelevante, ou seja, o custo da solução não irá melhorar independente dos passeios em $\mathcal{W}$. Note que o número de passeios de H é exponencial, portanto, com exceção das instâncias pequenas, $\mathcal{W}$ será um subconjunto próprio do conjunto de passeios.

No pseudo-código 3.1 temos uma implementação para encontrar o tour minimal da sequência da operação E$uler$ para um dado milling tour $W = (u_1, \ldots, u_s)$.

A entrada do algoritmo 3.1 é um milling tour W representado como uma lista ligada de vértices. As variáveis locais $cobertura(u_i)$ e $\pi(u_i)$ correspondem, respectivamente, ao número de vezes que o vértice u_i é visitado pelo tour W e a última posição visitada que contém o vértice u_i.

Algoritmo 3.1 Rotina para encontrar o tour minimal da operação E*uler*.

```
1: function TOUR-MINIMAL-DE-EULER(W = (u_1, ..., u_s))
2:     for i = 1..., s do
3:         cobertura(u_i) ← 0
4:     end for
5:     for i = 1..., s do
6:         cobertura(u_i) ← cobertura(u_i) + 1
7:         π(u_i) ← nil
8:     end for
9:     for i = 1, ..., s do
10:        if cobertura(u_i) > 1 ∧ π(u_i) = nil then
11:            π(u_i) ← i
12:        else if π(u_i) ≠ nil then
13:            { Remove a subsequência e atualiza π e cobertura.}
14:            W ← W\(u_{π(u_i)}, ..., u_{i-1})
15:            for j = π(u_i) + 1, ..., i do
16:                π(u_j) ← nil
17:                cobertura(u_j) ← cobertura(u_j) − 1
18:            end for
19:            π(u_i) ← i
20:        end if
21:        { O vértice só é coberto uma vez}
22:        if cobertura(u_i) = 1 then
23:            j ← i − 1
24:            while π(u_j) ≠ nil do
25:                π(u_j) ← nil
26:                j ← j − 1
27:            end while
28:        end if
29:    end for
30: end function
```

Nas linhas $9-31$ os vértices são percorridos seguindo a ordem dada pela lista ligada de entrada W e três condições são verificadas. Na primeira, na linha 9, o vértice foi visitado pelo menos duas vezes pelo tour e não faz parte da subsequência corrente de vértices repetidos. Portanto, é um candidato à remoção.

No segundo caso, linhas $12-20$ um ciclo $u_{\pi(u_i)}, \ldots, u_i$ de vértices coberto múltiplas vezes (subsequência repetida) foi encontrado e é removido. Vale notar que, representando W como uma lista ligada, esse passo pode ser feito em tempo constante por vértice removido, ou seja, num total de $O(s)$ operações.

Da linha 22 à 28 é considerado o caso em que o vértice é coberto exatamente uma vez e, assim, nenhuma subsequência que o contém pode ser removida. Portanto, uma nova subsequência é iniciada e os valores correntes em $\pi(u_i)$ para todo u_i pertencente a subsequência corrente são invalidados. No total esta operação de atualização de π não pode ser feita mais do que $O(s)$ vezes.

A saída do algoritmo é um milling tour com custo menor ou igual ao original e, pela discussão acima, conclui-se que a complexidade é linear no número de vértices do tour de entrada W.

3.4.2 Vizinhança da Cruz

Apresentamos a seguir outra vizinhança de busca local que podemos fazer em um milling tour de forma a reduzir o custo da solução. Essa vizinhança reduz o custo do tour em dois sempre que aplicada, e é similar a união de ciclos com conversões opostas.

Seja x um vértice de grau quatro em que são feitas duas conversões simples e opostas, como ilustrado na figura 3.10a da união de ciclos. Da mesma forma como naquela situação, as duas conversões simples são transformadas em conversões colineares, reduzindo, assim, o custo em dois. Note que, com esta substituição das conversões, o tour também é percorrido em uma ordem diferente.

Observe no entanto que, como a vizinhança é aplicada em um único tour, existem casos em não é possível evitar as duas conversões simples, já que a aplicação da transformação pode quebrar o ciclo em dois, o que é uma solução inválida para o MTWTC. A figura 3.18 exemplifica esta situação.

Como consequência da afirmação anterior, precisamos definir precisamente as condições para que esta vizinhança possa ser aplicada sem que um ciclo seja quebrado. Seja $G = (V, E)$ um grafo de subdivisão, W um milling tour de G. Suponha que exista um vértice x onde ocorrem duas conversões simples e opostas: (u, x, v) e (u', x, v'). Sem perda de generalidade, supomos que $W = w_0, \ldots, w_i, u, x, v, w_{i+1}, \ldots, w_j, u', x, v', w_{j+1}, \ldots, u_0$. Como as conversões são opostas, os vértices u e u' são colineares assim como v e v'. Se for possível eliminar as duas conversões em x, então podemos obter o tour $\textsc{cruz}(W) =$

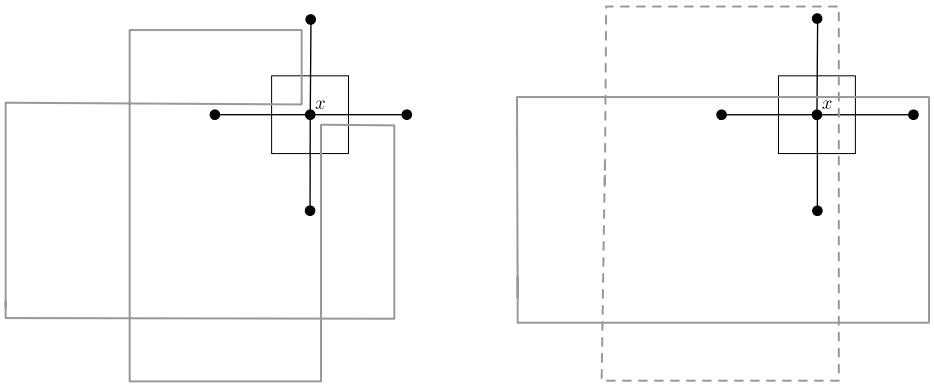


Figura 3.18: Transformação de cruz feita no vértice x desconecta o milling tour gerando dois tours, tornando a solução inválida.

$(w_0, \ldots, w_i, u, x, u', \underline{w_j, \ldots, w_{i+1}, v}, x, v', w_{j+1}, \ldots, u_0)$, com a parte sublinhada indicando que em $\textsc{cruz}(W)$ está subsequência está em ordem contrária a de W. Se não existe um vértice x com essa propriedade então $g(W) = W$. A **Vizinhança da Cruz** é a vizinhança $\textsc{cruz}'$, que retorna o tour minimal da sequência de $\textsc{cruz}$, ou seja, aplica a vizinhança $\textsc{cruz}$ consecutivamente até que o tour não se modifique.

O pseudo-código 3.2 é uma implementação eficiente para a vizinhança da cruz. A entrada do algoritmo, assim como na rotina 3.1, é um milling tour representado como uma lista ligada de vértices.

Inicialmente, para cada vértice x, construímos a lista de conversões simples que ocorrem em x. Nas linhas 2–6, para cada conversão (u_i, u_{i+1}, u_{i+2}) construímos uma lista ligada $T[(u_i, u_{i+1}, u_{i+2})]$ com as posições em que esta conversão ocorre na lista ligada W que representa um milling tour. A variável T é um mapa de triplas de vértices para a lista ligada de vértices. Em seguida, iteramos novamente sobre o tour, e para cada conversão simples $t = (u_i, u_{i+1}, u_{i+2})$ verificamos se existe uma conversão oposta a t e fazemos a substituição pelas conversões colineares em 18. No pseudo-código assumimos a existência de algumas funções para executar subrotinas auxiliares:

- INSERE-LISTA(L, e), insere o elemento e no fim da lista L;
- OPOSTO(u, v), retorna o vértice u' que é colinear a u e adjacente a v;
- RETORNA-PRIMEIRO(L), remove o primeiro elemento da lista L e o retorna.

A complexidade do algoritmo depende daquela do mapa T, que tem no máximo s elementos. Se utilizarmos uma árvore binária de busca [13], as operações de busca e inserção tem complexidade $O(\log s)$. Da linha 7 à 22 iteramos sobre cada conversão de W, ou seja $s-2$ vezes, enquanto que o laço das linhas 13 até 15 é repetido no máximo $s-2$

pois este é o valor que limita o número de conversões simples. Portanto, a complexidade do algoritmo no pior caso é $O(s^2 \log s)$.

Algoritmo 3.2 Rotina para encontrar o tour minimal da operação C*ruz*.

```
1: function TOUR-MINIMAL-DE-CRUZ(W = (u_1, ..., u_s))
2:     for i = 1, ..., s − 2 do
3:         if (u_i, u_{i+1}, u_{i+2}) é simples then
4:             INSERE-LISTA(T[u_i, u_{i+1}, u_{i+2}], i)
5:         end if
6:     end for
7:     for i = 1, ..., s − 2 do
8:         if (u_i, u_{i+1}, u_{i+2}) é simples then
9:             u' ← OPOSTO(u_i, u_{i+1})
10:            v' ← OPOSTO(u_{i+2}, u_{i+1})
11:            if T[u', u_{i+1}, v'] ≠ ∅ then
12:                j ← i + 1
13:                repeat
14:                    j ← RETORNA-PRIMEIRO(T[u', u_j, v'])
15:                until j > i + 1 ∧ T[u', u_j, v'] ≠ ∅
16:                { As conversões que ocorrem antes de i + 1 já foram consideradas.}
17:                if j > i + 1 then
18:                    W ← (u_1, ..., u_i, u_{i+1}, u_{j−1}, u_{j−2}, ..., u_{i+2}, u_j, u_{j+1}, ..., u_s)
19:                end if
20:            end if
21:        end if
22:    end for
23: end function
```

3.4.3 Vizinhança da Corda

Um outro padrão que encontramos nas soluções retornadas pelo APX e pelas heurísticas e que permite a redução do custo, consiste em encontrar uma aresta que une dois vértices cujo caminho que os liga na solução original é pelo menos, duplamente coberto. Uma aresta com essa propriedade é chamada de **corda**. A vizinhança da corda, consiste em substituir o caminho pela sua corda. Seja $W = (v_0, \ldots, u, u_1, \ldots, u_s, v, \ldots, v_k)$ um milling tour onde u, v são vértices adjacentes no grafo de subdivisão. A vizinhança da corda é definida pela vizinhança C*orda* que, quando aplicada no milling tour W, gera o milling tour $corda(W) = (v_0, \ldots, u, v, \ldots, v_k)$ com custo menor ou igual o de W. A redução no custo depende do custo do caminho que liga os vértices da corda em W. A figura 3.19 ilustra uma aplicação de C*orda* sobre W onde há uma redução de duas unidades no custo.

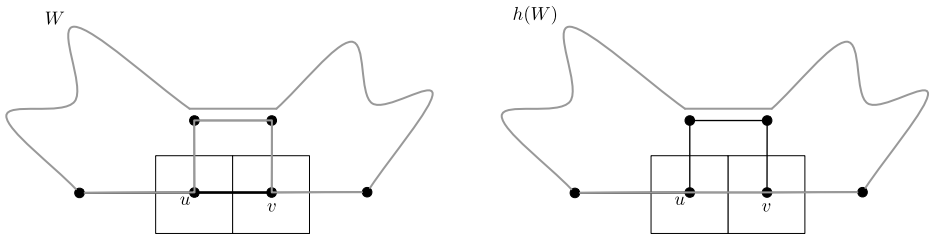


Figura 3.19: Transformação da corda aplicada no milling tour W, usando a corda $\{u, v\}$.

De maneira análoga à vizinhança da cruz, a **Vizinhança da Corda** é a vizinhança C$orda'$ que retorna o tour minimal da sequência de C$orda$, isto é, o tour obtido ao se aplicar a operação C$orda$ consecutivamente até que o tour não se modifique.

No pseudo-código 3.3 implementamos a vizinhança C$orda'$ utilizando ideias similares àquelas usadas para a vizinhança do passeio Euleriano. A entrada do algoritmo é o tour W e o grafo de subdivisão $G = (V, E)$ e a saída um milling tour W' que é minimal com relação a vizinhança C$orda$. O vetor $cobertura(v)$ contém o número de vezes que o vértices $v \in G$ é percorrido na solução W. Da linha 12 à 33 iteramos sobre o milling tour W mantendo a invariante de que $(u_i, \ldots, u_j)$ é um caminho de u_i à u_j e a cobertura dos vértices é maior ou igual a dois, exceto possivelmente a de u_i ou u_j. O elemento $\pi(u)$ contém a posição do vértice u no caminho que está sendo considerado e é utilizada na detecção de ciclos. Na linha 18, quando uma corda entre u_i e u_j é encontrada, a subsequência que representa o caminho ligando esses dois vértices é eliminada do tour.

Fazendo-se uma análise de complexidade similar àquela realizada para o algoritmo 3.1 referente à vizinhança do passeio Euleriano, chega-se à conclusão que o algoritmo 3.3 tem comlexidade linear em s.

Combinando as três vizinhanças anteriores construímos a vizinhança $\mathcal{P}$ que consiste em aplicar as três vizinhanças, do passeio Euleriano, da cruz e da corda, em sequência, ao tour W, ou seja, $\mathcal{P}(W) =$ C$orda'$(Cruz'(E$uler'(W)$)). A **heurística de busca local** recebe como entrada um milling tour W e retorna o tour minimal da sequência de $\mathcal{P}$.

No próximo capítulo é feito um estudo experimental de todos os algoritmos que foram discutidos até aqui.

Algoritmo 3.3 Rotina para encontrar o tour minimal da operação C*orda*.

```
1: function TOUR-MINIMAL-DE-CORDA(W = (u_1, ..., u_s), G = (V, E)))
2:     { Inicializa os vetores cobertura e π.}
3:     for i = 1..., s do
4:         cobertura(u_i) ← 0
5:     end for
6:     for i = 1..., s do
7:         cobertura(u_i) ← cobertura(u_i) + 1
8:         π(u_i) ← nil
9:     end for
10:    i ← 1
11:    π(u_1) ← 1
12:    for j = 2, ..., s do
13:        if {u_i, u_j} ∈ E then
14:            for k = i + 1, ..., j − 1 do
15:                cobertura(u_k) ← cobertura(u_k) − 1
16:                π(u_k) ← nil
17:            end for{ Elimina o caminho duplamente coberto (u_{i+1}, ..., u_{j−1}).}
18:            W ← (u_1, ..., u_i, u_j, u_{j+1}, u_s)
19:        else if π(u_j) ≠ nil then
20:            for k = i, ..., π(u_j) − 1 do
21:                π(u_k) ... nil
22:            end for
23:            i ← π(u_j) + 1
24:            π(u_j) ← nil
25:        end if
26:        if cobertura(u_j) = 1 then
27:            for k = i, ..., j do
28:                π(u_k) ← nil
29:            end for
30:            i ← j
31:        end if
32:        π(u_j) ← j
33:    end for
34: end function
```

Capítulo 4

Avaliação Experimental

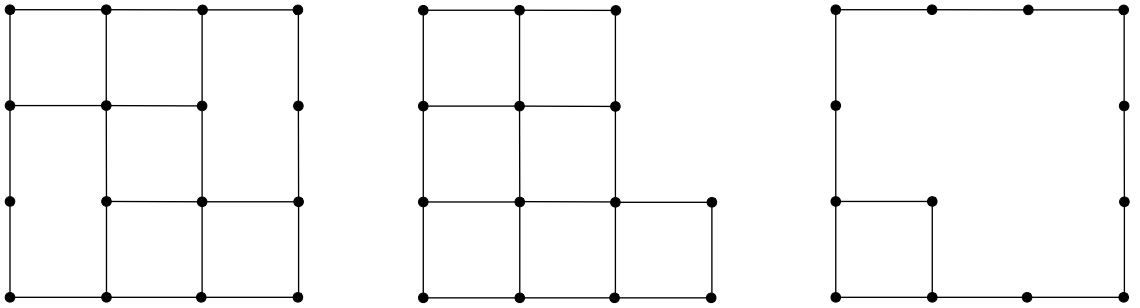
Os capítulos anteriores foram dedicados ao estudo e desenvolvimento de algoritmos para o MTWTC. Especificamente, projetamos um algoritmo exato usando **PLI**, descrevemos um algoritmo aproximado, para o qual propusemos modificações e, por fim, criamos três heurísticas e três vizinhanças que compõe uma busca local. A seguir, os algoritmos serão identificados por suas siglas, conforme descrito abaixo:

- **EXATO**: algoritmo de **B&C** que resolve o problema à otimalidade (ver Capítulo 2).
- **APX**: algoritmo aproximado com fator 3.75 desenvolvido por *Arkin et. al* e descrito pela primeira vez em [6] (ver Seção 3.1).
- **GARDENER**: heurística inspirada em uma maneira gulosa de abordar o problema (ver Seção 3.3.1), incrementada com técnicas da meta-heurística GRASP, como aleatorização e múltiplos-reinícios.
- **HEUR**: heurística que faz uso de um modelo **PLI** para resolver um problema mais simples e obter uma cobertura por ciclos que explora características específicas da rotina de união de ciclos (Seção 3.3.2).
- **MHEUR**: extensão da heurística **HEUR** que implementa múltiplos-inícios (resolução de vários modelos **PLI**) conforme descrito no final da Seção 3.3.2.

Neste capítulo, faremos um estudo experimental destes algoritmos com o objetivo de avaliar o comportamento dos mesmos e obter intuição para melhorias.

4.1 Instâncias de Teste

O primeiro passo de uma análise experimental consiste em definir os conjuntos de instâncias a serem utilizados nos testes. Na revisão bibliográfica constatamos que não existem



(a) Densidade de $\frac{20}{24} = 0.91$. (b) Densidade de $\frac{20}{24} = 0.84$. (c) Densidade de $\frac{14}{24} = 0.58$.

Figura 4.1: Exemplos de instâncias e as respectivas densidades com relação a grade 4×4.

instâncias em domínio público e, portanto, foi necessário desenvolver o nosso próprio conjunto de instâncias.

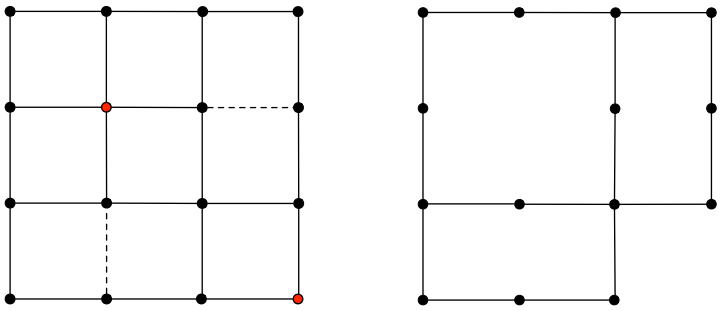
Seja $G = (V, E)$ o grafo de subdivisão de uma instância gerada a partir de um quadrado de lado s, chamado aqui de **tamanho** da instância. A **densidade** de uma instância é dada por:

$$\text{densidade}(G) = \frac{|E|}{2s(s-1)}. \tag{4.1}$$

Na figura 4.1 são dados três exemplos de instâncias, geradas a partir de uma grade 4×4 e suas respectivas densidades. Observe como as instâncias 4.1a e 4.1b tem a mesma densidade porém as áreas de cobertura são diferentes, ilustrando o fato da densidade, como foi definida, considerar também as maneiras de como a região do pocket pode ser coberta.

A rotina de geração de instâncias recebe como parâmetros, o tamanho s e a densidade d. Inicialmente geramos um quadrado de lado s e o grafo de subdivisão $G = (V, E)$ deste, que corresponde a um grafo de grade $s \times s$. Em seguida iteramos nas arestas e vértices do grafo. A cada iteração uma aresta ou um vértice é escolhido de modo aleatório a fim de ter sua remoção avaliada. Se a remoção da aresta não desconecta o grafo e não diminui o grau de algum vértice para menos que dois a aresta é removida. Já um vértice (e as arestas incidentes) é removido se esta operação não desconecta o grafo e não deixa um outro vértice com grau menor que dois. A razão para desconsiderar vértices de grau um é que toda solução ótima necessariamente faz um único U-turn neste vértice. As iterações ocorrem enquanto a densidade do grafo corrente for maior que d. No final, se a diferença da densidade corrente com d for maior que uma tolerância de 0.01, a instância é desconsiderada e o procedimento se repete.

O conjunto de testes `DENSIDADE` é composto por 2080 instâncias geradas a partir da grade de lado 8 e agrupadas por densidade. Cada grupo contém exatamente 160 instâncias



(a) As arestas tracejadas e os vértices destacados são removidos.

(b) Instância final após a remoção de arestas e vértices.

Figura 4.2: Instância gerada com parâmetros $s = 4$ e $d = \frac{2}{3}$.

e as densidades variam de 0.44 até 0.96, em incrementos de 0.04. Ou seja, o primeiro grupo contém instâncias de densidade $[0.44, 0.48)$, o segundo de $[0.48, 0.52)$ e assim por diante, até o último grupo com densidades no intervalo $[0.92, 0.96)$.

A figura 4.2 ilustra o procedimento de geração de instância para um caso específico. Em 4.2a, as tracejadas e os vértices em cinza foram escolhidos para remoção. A instância final é dada em 4.2b.

O conjunto `DENSIDADE` visa avaliar os algoritmos com relação à variação de densidade. No entanto, também é interessante avaliá-los com relação ao tamanho da instância. Para isto, foi criado o conjunto `TAMANHO` que contém instâncias com densidade no intervalo $[0.75, 0.90]$ dividido em seis grupos com vinte instâncias cada. No primeiro grupo as instâncias foram geradas com $s = 5$, no segundo $s = 6$ e assim por diante até $s = 10$. Na seção 4.5 esclarecemos com mais detalhes por que classificamos as instâncias com relação à densidade e por que restringimos a densidade das instâncias no conjunto `TAMANHO` a um dado intervalo.

4.2 Ambiente de Testes e Desenvolvimento

Os algoritmos descritos nessa dissertação foram implementados utilizando a linguagem *C++* [32] e compilados utilizando *GNU Compiler Collection (gcc)* [2] na versão 4.4.3 distribuída com o sistema operacional *Linux*, especificamente, *Ubuntu 10.04.3 LTS* [1]. Algumas bibliotecas e códigos de terceiros que não são padrão da linguagem C++ foram utilizados, são eles: *Boost 1.47.0* [14] para funções e classes auxiliares, *Apache Thrift 0.7.0* [17] para a comunicação inter-processos entre a interface gráfica e as implementações dos algoritmos e *Blossom V 2.0.2* [25] para resolver o problema de emparelhamento per-

feito de custo mínimo no APX. O *download* dos *softwares* e bibliotecas citados – inclusive o código-fonte – podem ser feitos nos respectivos sítios.

Também utilizamos o pacote comercial *IBM ILOG CPLEX Optimizer 12* [3] na versão 12.2.0.0. Em particular, utilizamos o resolvedor de **PLI** do pacote na implementação do algoritmo EXATO descrito no capítulo 2 e na heurística HEUR do capítulo 3. A tabela 4.1 tem o valor dos parâmetros do CPLEX que foram modificados pela implementação. A descrição detalhada do parâmetro e os possíveis valores podem ser encontrados no manual do pacote.

Tabela 4.1: Valor dos parâmetros do CPLEX modificados pela implementação.

Parâmetro	Valor
CPX_PARAM_TILIM	$1.80000000000000e+03$
CPX_PARAM_EPAGAP	$1.9990000000000e+00$

Os experimentos foram feitos em um máquina com processador *Intel®Xeon®X*3430 *2.40GHz*, $8MB$ de memória cache e $8GB$ de memória RAM dedicada exclusivamente para a realização os testes desta dissertação.

4.3 Ferramenta de Visualização

Quando estudamos problemas de origem geométrica, a visualização das instâncias e principalmente das soluções tem uma grande importância no entendimento das particularidades do problema. Com esta motivação em mente, um dos objetivos deste trabalho de mestrado consistiu no desenvolvimento de uma ferramenta de visualização e criação de instâncias para o MTWTC. A seguir descreveremos as principais características do visualizador desenvolvido.

A ferramenta tem dois modos de uso: construção de instâncias e visualização de soluções. No primeiro modo, iniciamos com um grafo de grade completo $s \times s$ que representa o grafo de subdivisão de uma instância e removemos vértices e arestas conforme desejado. Informações auxiliares como o número de vértices, arestas e densidade são recalculadas a cada modificação da instância, assim como o conjunto de vértices e arestas que podem ser removidos. As instâncias construídas podem ser salvas, carregadas e resolvidas usando qualquer um dos algoritmos desta dissertação diretamente a partir da interface.

Já o segundo modo, permite a visualização das soluções e informações extras dependentes do algoritmo. Para o algoritmo exato, também é dado o valor do melhor limitante inferior encontrado e, para o APX, também é possível visualizar quais foram as faixas escolhidas na fase de cobertura por faixas e qual a cobertura por ciclos do fim da fase dois. Em termos de execução dos algoritmos, as facilidades disponibilizadas pela interface

permitem a alteração das opções dos algoritmos de forma que a comparação entre soluções possa ser feita de maneira mais interativa.

Na figura 4.3 temos imagens da interface, mostrando os dois modos uso. Em um primeiro momento construímos uma instância e em seguida a executamos usando o APX com a rotina de busca local habilitado.

Com relação à implementação, a ferramenta foi desenvolvida utilizando a linguagem *Python* versão 2.7 e a biblioteca *Qt4* com o auxilio do *PyQt* que permite a interoperabilidade entre o Python e a Qt.

4.4 Detalhes de Implementação

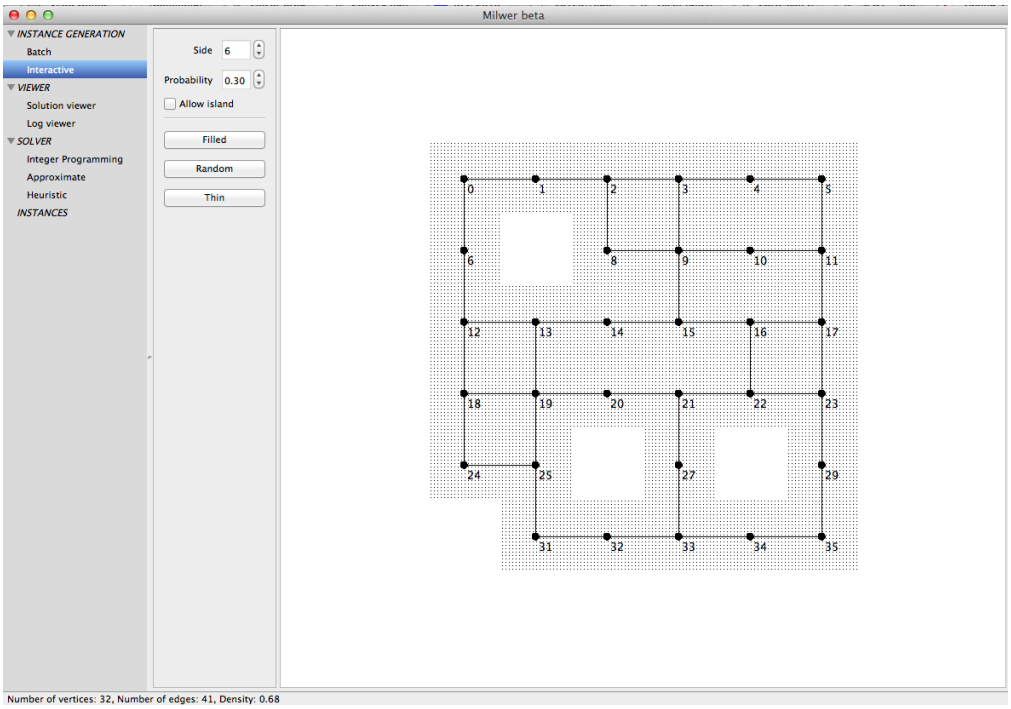
Vejamos agora alguns aspectos relevantes da implementação, incluindo as estruturas de dados utilizadas para representar os elementos principais do problema e a complexidade dos algoritmos mais importantes. Também descreveremos em detalhes o formato das instâncias de entrada e das respectivas soluções.

A principal estrutura de dados é dada pelo tipo **TGraph**, que representa um grafo orientado incrementado com conversões. Nessa estrutura, um grafo orientado $G = (V, A)$ tem seus vértices representados por inteiros de $0 \ldots |V| - 1$ e os arcos identificados por inteiros de $0 \ldots |A| - 1$. São definidos três vetores: **src**, **dst** e **cost** indexados pelos arcos e com valores iguais a origem, o destino e o custo do arco. A cada vértice v é associado um vetor de inteiros $\mathbf{adj}[v]$ indexado por inteiros de $0 \ldots N(v) - 1$ onde $N(v)$ é o número de arcos incidentes em v e cujo valor é o identificador do arco.

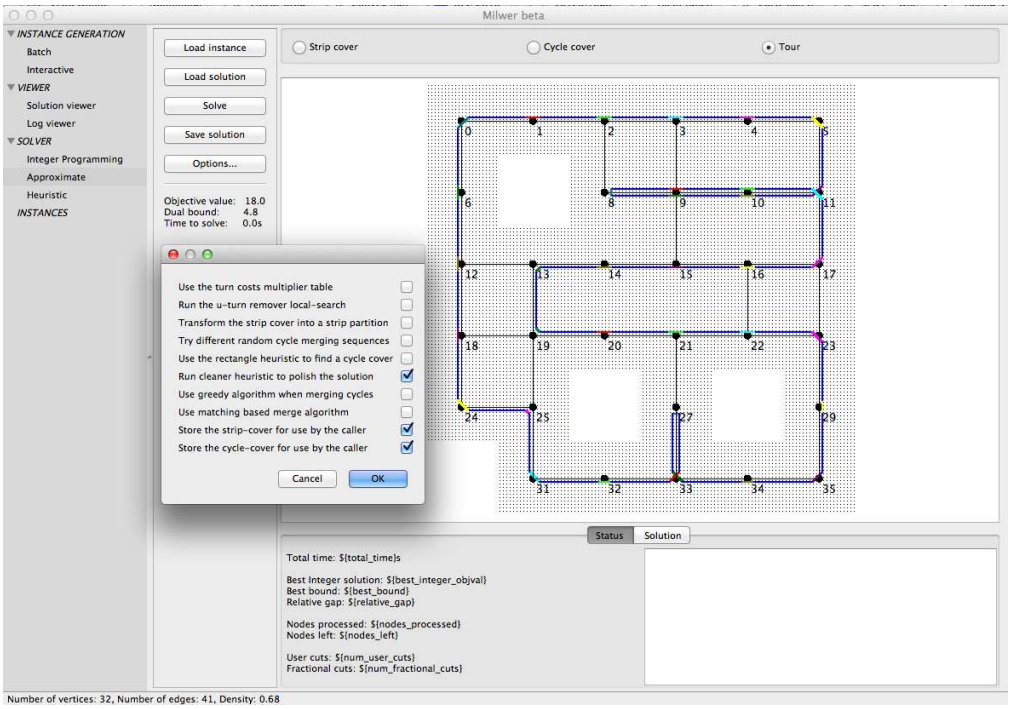
Essa estrutura é incrementada com uma representação explícita das conversões existentes no grafo. Para isso, a cada arco $a = (u, v)$ são associados dois vetores $\mathbf{trn}[a]$ e $\mathbf{trn\_cost}[a]$ indexados por inteiros de $0 \ldots N(a)$ onde $N(a)$ é o número de arcos cuja origem é o vértice v e cujos valores são, o identificador de um arco e o custo da conversão, respectivamente. Exemplificando, seja (u, v, w) um conversão orientada simples, i.e, com mudança de direção, então existem os arcos $a = (u, v)$, $b = (v, w)$ e um índice i tal que trn$[a][i] = b$ e trn_cost[a][i] $= 1$.

Quanto à complexidade das operações, dado o identificador do vértice v, com complexidade $O(N(v))$ podemos iterar sobre os vértices adjacentes a v, e com o identificador do arco a, em tempo $O(1)$ encontramos as extremidades, o custo do arco e podemos iterar sobre as conversões que utilizam o arco a. Com relação ao espaço, se o inteiro for representado por 4 *bytes* de 8 bits cada, então, um grafo orientado $G = (V, A)$ com $|T|$ conversões utiliza espaço de $4(4|E| + 2|T|)$ *bytes*[1].

[1]Nesta conta, estamos desconsiderando questões da arquitetura como alinhamento de memória e o *overhead* da representação dos vetores.



(a) Construção de instâncias.



(b) Execução e visualização de soluções.

Figura 4.3: Modos da ferramenta de visualização de instâncias e soluções.

Uma observação importante sobre esta representação é que, ao utilizarmos grafos não-orientados, adicionamos dois arcos por aresta e duas conversões orientadas por conversão (u, v, w). Além disso, a remoção de vértices (arestas) sem a invalidação dos identificadores é uma tarefa não trivial e, nestes casos, a solução adotada é utilizar vetores auxiliares como marcadores que indicam se o vértice (aresta) foi removido.

A próxima estrutura de dados fundamental de nossa implementação, é aquela que descreve uma solução, ou seja, é um passeio fechado do grafo de subdivisão. O tipo **walk_t** representa uma solução e corresponde a um vetor de vértices. No entanto, em algumas situações o vetor é convertido para uma lista ligada, pois esta permite que algumas operações sejam realizadas de uma forma mais eficiente. Em particular, nas rotinas de busca local, os algoritmos descritos no capítulo 3 assim como as respectivas análises de complexidade supõem que uma lista ligada esteja sendo utilizada.

A representação de listas ligadas mantém implicitamente uma ordem de como os vértices são percorridos. Contudo, a descrição do problema não define uma ordem de visitação dos vértices e, ao impor essa restrição na implementação, exige-se que se trate separadamente a ordem de percurso dos vértices e a circularidade da sequência. Para entender este aspecto, considere uma sequência de vértices $W = (u_1, \ldots, u_s)$. A partir daí, defina a sequência reversa $W' = (u_s, u_{s-1}, \ldots, u_1)$ e a rotacionada $W^R = (u_s, u_1, \ldots, u_{s-1})$. Aplicando os algoritmos sobre W, W' e W^R e combinando os resultados, tratamos as simetrias de ordem e circularidade.

Vejamos agora um resumo das características de alguns algoritmos utilizados como sub-rotinas de outro algoritmo descrito na dissertação. Começando pelo EXATO, da seção 4.2, sabemos que a implementação do **B&C** é aquela do pacote CPLEX. A interação com o CPLEX é feita através do uso de *callbacks*, que são funções definidas pelo usuário para serem chamadas em pontos específicos do algoritmo. A lista completa de callbacks podem ser encontradas no manual que acompanha o pacote.

A principal callback que utilizamos é chamada após a resolução da relaxação linear e é responsável por adicionar planos de corte ao modelo[2]. Nossa implementação é a rotina de separação dada pelo algoritmo 2.1. Dessa rotina, um detalhe importante é implementação do algoritmo de (S, T)-corte. Embora na literatura existam diversos algoritmos combinatórios decidimos utilizar o *Network SIMPLEX* – implementado pelo CPLEX – que é uma variação do algoritmo SIMPLEX do capítulo 2 desenvolvida especialmente para problemas de *Fluxo de Custo Mínimo* do qual o (S, T)-corte é um caso particular.

O APX também faz uso de diversos algoritmos conhecidos. Na primeira fase, precisamos encontrar um emparelhamento de cardinalidade máxima em um grafo bipartido. Nossa implementação utiliza o algoritmo de caminhos aumentantes [5] e tem complexidade $O(|V||E|)$.

[2]Na nomenclatura do CPLEX, é uma *CutCallback*.

A fase 2 do APX, é realizada com auxílio de dois algoritmos clássicos. Em um primeiro momento precisamos calcular o caminho de custo mínimo entre dois vértices no grafo de conversões. Para tanto, nossa implementação utiliza o algoritmo de Dijkstra com um heap binário. Uma vez que a estrutura TGraph permite iterar sobre os vizinhos de um vértice em tempo proporcional ao número de vizinhos e que o grafo de conversões sobre o qual o caminho é calculado tem tamanho proporcional a $|V|$, a complexidade final para encontrar o caminho é de $O(|V| \log |V|)$. A mesma análise é válida para as heurísticas HEUR e GARDENER onde também precisamos encontrar o menor caminho em conversões entre pares de vértices.

Os custos e caminhos computados pelo algoritmo de Dijsktra, correspondem aos custos das arestas na construção de uma cobertura por ciclos a partir da cobertura por faixas da fase 1. Essa cobertura é calculada a partir das arestas do emparelhamento perfeito de custo mínimo entre os extremos de faixas. Para encontrar o emparelhamento tiramos proveito de uma implementação do algoritmo de Edmonds [16] conhecida como *Blossom V* [26], que tem complexidade no pior caso de $O(|V|^2|E|)$, mas é extremamente eficiente na prática.

4.5 Experimentos

Queremos com esta análise experimental avaliar, comparar e classificar os algoritmos desta dissertação com relação a dois aspectos:

1. Qualidade dos resultados.
2. Eficiência computacional.

Com relação à qualidade, nosso foco é entender para quais classes de instâncias os algoritmos se comportam melhor. Definiremos a seguir as métricas empregadas por nós:

Seja $G = (V, E)$ uma instância do MTWTC, $\mathcal{S}$ uma solução desta e valor($\mathcal{S}$) o valor da função objetivo para a solução $\mathcal{S}$. Definimos a distância entre duas soluções $\mathcal{S}_1$ e $\mathcal{S}_2$ para a mesma instância G como sendo:

$$\text{dist}(\mathcal{S}_1, \mathcal{S}_2) = |\text{valor}(\mathcal{S}_1) - \text{valor}(\mathcal{S}_2)|. \tag{4.2}$$

Derivamos da métrica anterior a medida normalizada de distância, aqui denominada por *gap* e dada por:

$$\text{gap}(\mathcal{S}_1, \mathcal{S}_2) = \frac{\text{dist}(\mathcal{S}_1, \mathcal{S}_2)}{\text{valor}(\mathcal{S}_2)} \tag{4.3}$$

Chamamos de **gap dual** o gap calculado utilizando uma solução hipotética cujo valor é igual ao limitante dual (limitante inferior) encontrado pelo algoritmo EXATO.

Tabela 4.2: Resumo dos resultados do algoritmo EXATO.

Conjunto de Instâncias	total	# de ótimos	gap médio não-ótimos
`DENSIDADE`	2080	1976	$8\% \pm 3\%$
`TAMANHO`	160	122	$8\% \pm 3\%$

Avaliação da Solução Exata

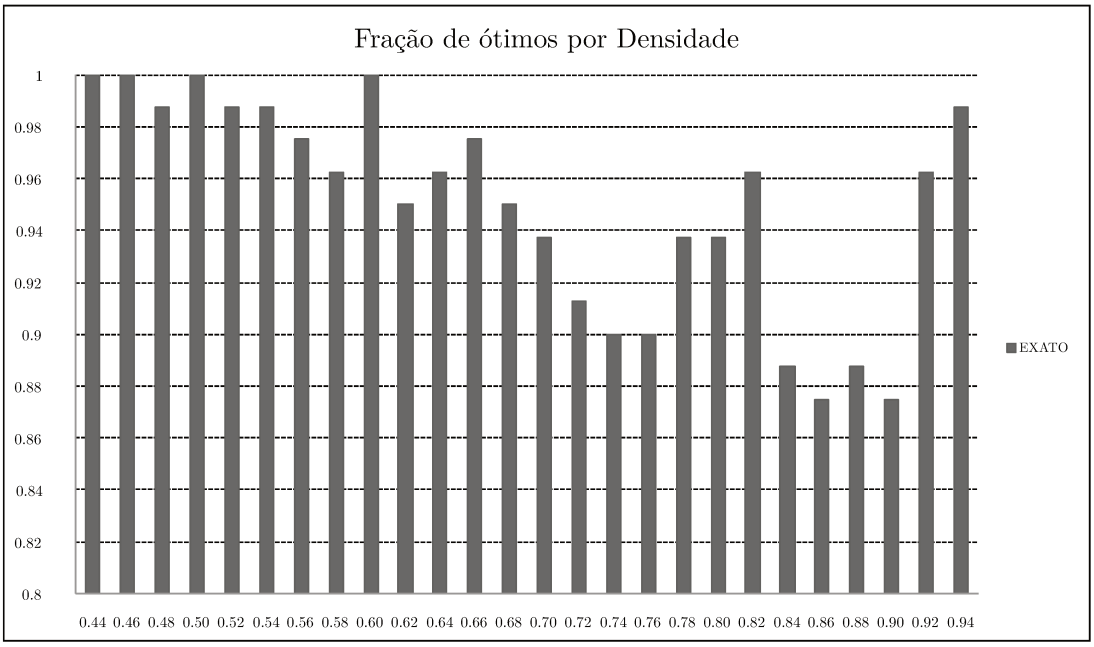
Quando se tem um algoritmo de complexidade exponencial (situação atual para os problemas $\mathcal{NP}$-dificíl) e, em especial, quando esse algoritmo usa técnicas de **PLI**, é necessária uma análise experimental para se obter informações úteis sobre o potencial real de aplicação do algoritmo com relação ao tamanho das instâncias.

Nesta seção iremos avaliar comportamento do algoritmo EXATO, desenvolvido no capítulo 2, para os conjuntos de instâncias `TAMANHO` e `DENSIDADE`.

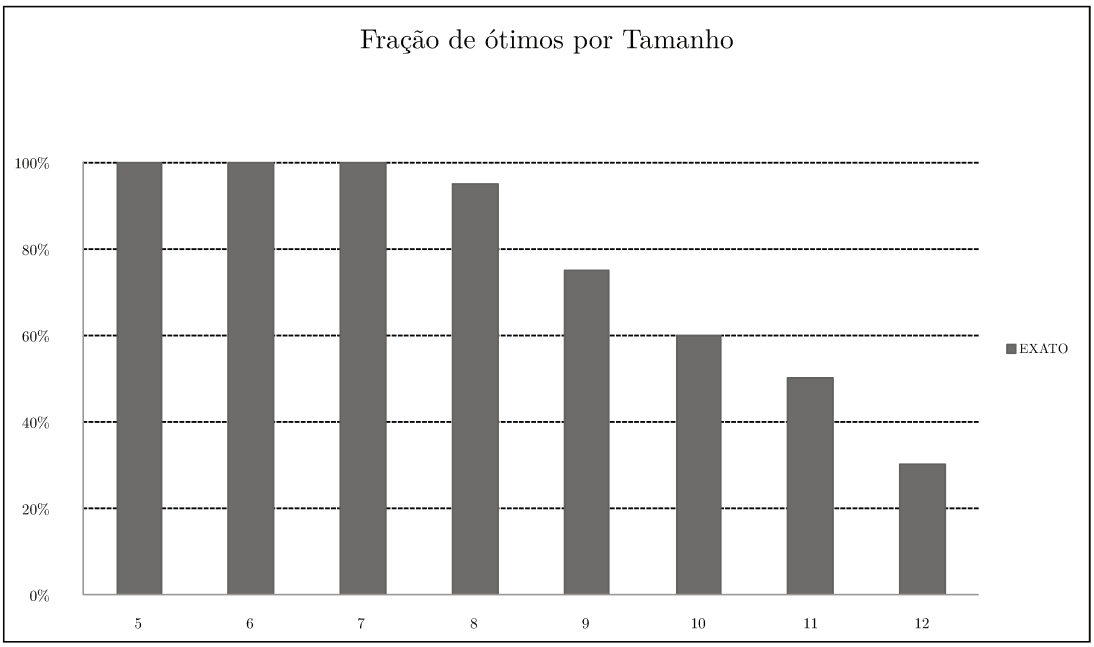
Para cada instância o algoritmo EXATO foi executado por no máximo 30. Determinamos o número de instâncias cujo valor encontrado é ótimo, e quando este último não é encontrado, utilizamos como medida o gap entre o valor do limitante dual e primal. Na tabela 4.2 estão resumidos os resultados de cada um dos conjuntos de teste.

Complementando os resultados da tabela 4.2, na figura 4.4 temos a variação da fração de ótimos por densidade e tamanho. Observando o gráfico da figura 4.4a temos que nos extremos, i.e, com instâncias de densidade maior que 0.94 ou menor que 0.7, mais de 94% delas foram resolvidas na otimalidade em menos de meia-hora. A região crítica se encontra no intervalo de 0.75 à 0.90 onde há uma queda de aproximadamente 6% com relação ao número de ótimos. No entanto, pela tabela 4.2 sabemos que o valor médio do gap dual é 8%. O gráfico da figura 4.4b mostra que para instâncias em que o grafo de subdivisão tem mais de 160 arestas, o algoritmo exato baseado em **PLI** se torna impraticável e soluções alternativas devem ser procuradas.

Combinando os resultados do gap dual por densidade com o gráfico de tempo de execução da figura 4.5 podemos concluir que há uma relação entre a densidade e o tempo de execução e que o intervalo de densidade entre 0.75 e 0.9 contém as instâncias mais difíceis. Observe como, independente da densidade, $\frac{3}{4}$ das instâncias são resolvidas em menos de 5 minutos para grades 8×8, mas os piores casos se encontram nas densidades de 0.8 à 0.9. Depois disso, para instâncias com densidades acima de 0.92, o tempo começa a reduzir tanto na média quanto no pior caso, indicando que o aumento no tempo não foi devido somente ao tamanho do modelo. Foi justamente esta análise que nos levou a construir instâncias de teste classificadas de acordo com a densidade.



(a) Conjunto `DENSIDADE`.



(b) Conjunto `TAMANHO`.

Figura 4.4: Fração de ótimos encontrados por conjunto de teste para o EXATO.

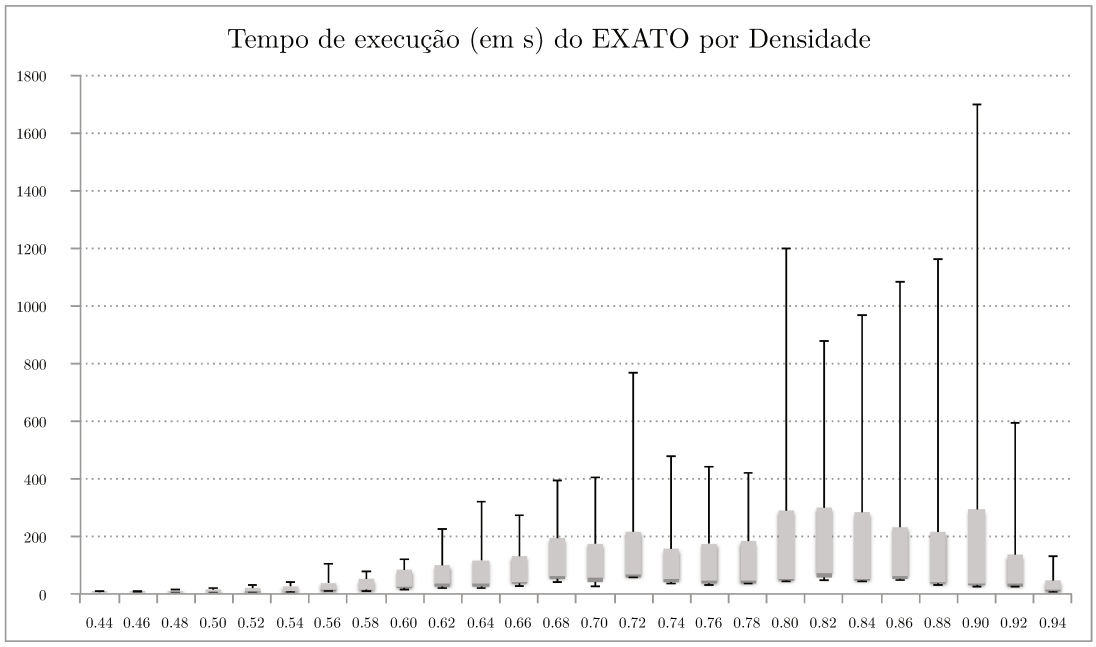


Figura 4.5: Tempo de execução do exato, para cada densidade são apresentados cinco percentis: 10%, 25%, 50% (mediana), 75% e 90%.

Avaliação e Evolução do Algoritmo Aproximado

Da teoria vista no capítulo 3, temos que o APX é um algoritmo com fator de aproximação 3.75. Este é um resultado de importância teórica. No entanto, uma outra informação interessante é saber como o algoritmo se comporta na prática e qual o fator de aproximação observado experimentalmente. Vale salientar que esta informação também pode ser aproveitada do ponto de vista teórico, pois nos dá uma noção mais precisa de quão justo é o fator de 3.75, levando a duas possíveis alternativas: tentar encontrar um fator menor ou garantir que o atual é justo.

Além disso, estamos interessados em conhecer o efeito das modificações heurísticas feitas no capítulo 3, são elas: considerar a distância no custo do caminho na fase 2, substituir a cobertura por faixas por uma partição em faixas e realizar a união de ciclos de forma gulosa ou via algoritmos de emparelhamento em grafos.

Para facilitar a identificação de quais modificações estão sendo aplicadas nos gráficos e tabelas a seguir o nome do algoritmo será dado da seguinte forma: APX_x^y, quando $x = d$ significa que os custos do menor caminho foram modificados para considerar a distância e quando $x = p$, além da modificação no custo, a cobertura por faixas foi transformada em um partição em faixas. Com relação ao sobrescrito y, quando $y = g$ o algoritmo guloso para união de ciclos foi utilizado e quando $y = m$, a união via emparelhamento (do inglês *matching*) foi usada. Para finalizar, usaremos o sufixo "+", para indicar que a rotina de

busca local foi aplicada a solução. Por exemplo, APX_p^m+ denota a versão mais completa do APX, com as duas modificações aplicadas, união de ciclos via emparelhamento e busca local aplicada na solução.

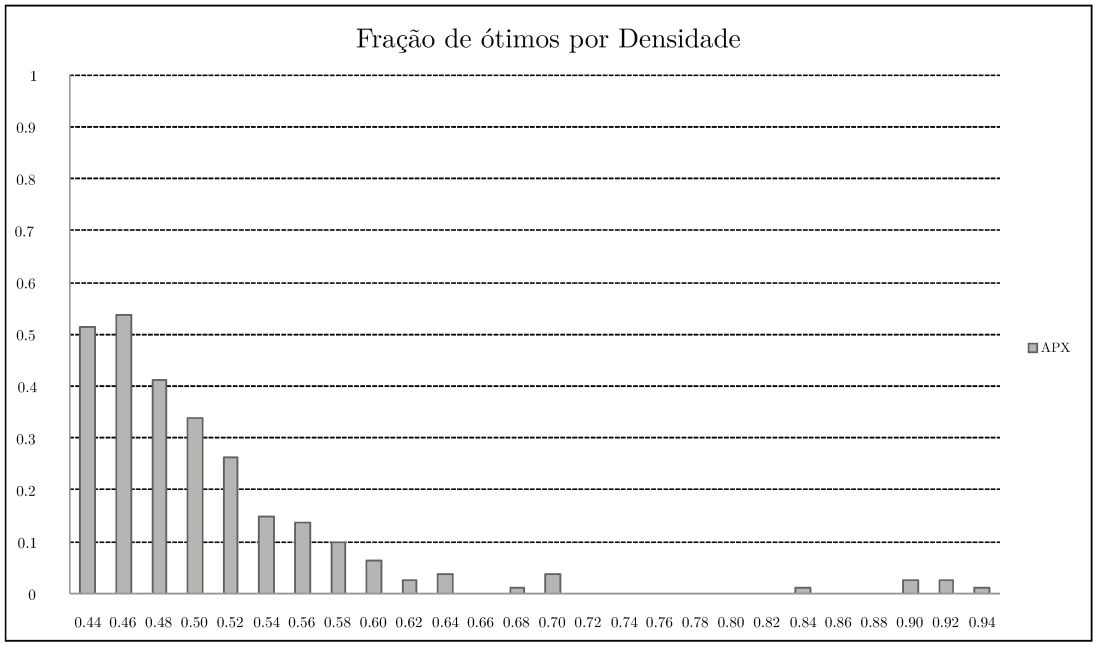
Na figura 4.6 temos os gráficos de fração de ótimos por densidade e gap dual por densidade para o APX. Neste caso a fração de ótimos refere-se à porcentagem de soluções ótimas encontradas em relação ao número de instâncias para os quais o ótimo é conhecido. Observe que conforme a densidade aumenta, o gap também aumenta variando de 5% para as densidades mais baixas e aproximadamente 35% nas densidades mais altas. Note que a curva com a fração de ótimos acompanha a densidade de maneira inversa, saindo de 50%, decrescendo rapidamente a menos de 1% e continuando assim.

Esses dois gráficos, da fração de ótimos e da média dos gaps duais, definem a base da nossa comparação entre os algoritmos. Vejamos qual o impacto das modificações no APX. Na figura 4.7a comparamos os gaps duais alcançados pelo APX_d e pelo APX. Embora haja uma redução do gap nas densidades baixas ao usar o APX_d, o maior impacto se encontra na faixa crítica, i.e, com densidades mais altas, onde há uma redução média de aproximadamente 3% sendo que, em alguns casos, como para a densidade 0.88, a redução chega a 8%.

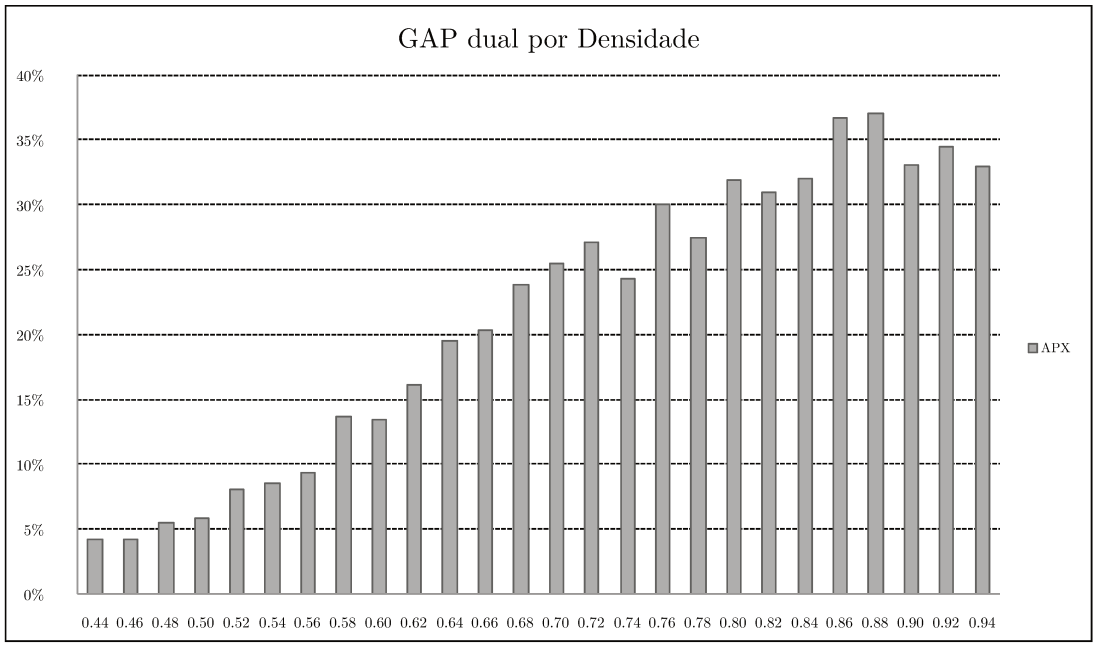
Continuando com as mudanças, a próxima versão é o APX_p que, como discutimos no capítulo 3, não poderia dar resultados piores que o APX_d. Sendo assim, o que falta é quantificar o ganho proporcionado pelo uso da partição no lugar da cobertura. Pelo gráfico da figura 4.7b, onde comparamos APX, APX_d e APX_p nota-se que, para densidades altas, a diferença de gap é mínima quando comparamos APX_p com o APX_d. No entanto, se considerarmos o intervalo $[0.58, 0.82]$, encontramos exemplos em que a redução é considerável. No geral, essas duas modificações reduziram o gap de 35% para 25% se considerarmos o intervalo de densidade $[0.44, 0.80]$, de 38% para aproximadamente 32% em todo o espaço, e como esperado, o APX_p foi estritamente melhor que o APX_d para todas as densidades.

Vejamos agora os resultados da mudança do algoritmo de união de ciclos, i.e, a terceira fase do algoritmo aproximado. No gráfico da figura 4.8c, temos o APX_p^m, que usa a união via emparelhamento. Note que não precisamos explicitamente comparar com a união gulosa, pois ela é dominada pela união via emparelhamento, sua vantagem sendo apenas sua menor complexidade. Observa-se que a redução no custo foi pequena, entre 0% e 2.2%, o que resultou em um teto de 30% no gap para todas as densidades.

A última versão que verificamos é o APX_p^m+ que aplica a rotina de busca local nas soluções. O gráfico comparativo do gap dual entre as versões é dado na figura 4.8d. Na figura 4.8, temos novamente um gráfico comparativo mas, dessa vez exibindo a porcentagem de ótimos encontrados em relação ao total de instâncias com otimalidade comprovada pelo algoritmo exato. A redução de gap causada pelo busca local é visível para todas as

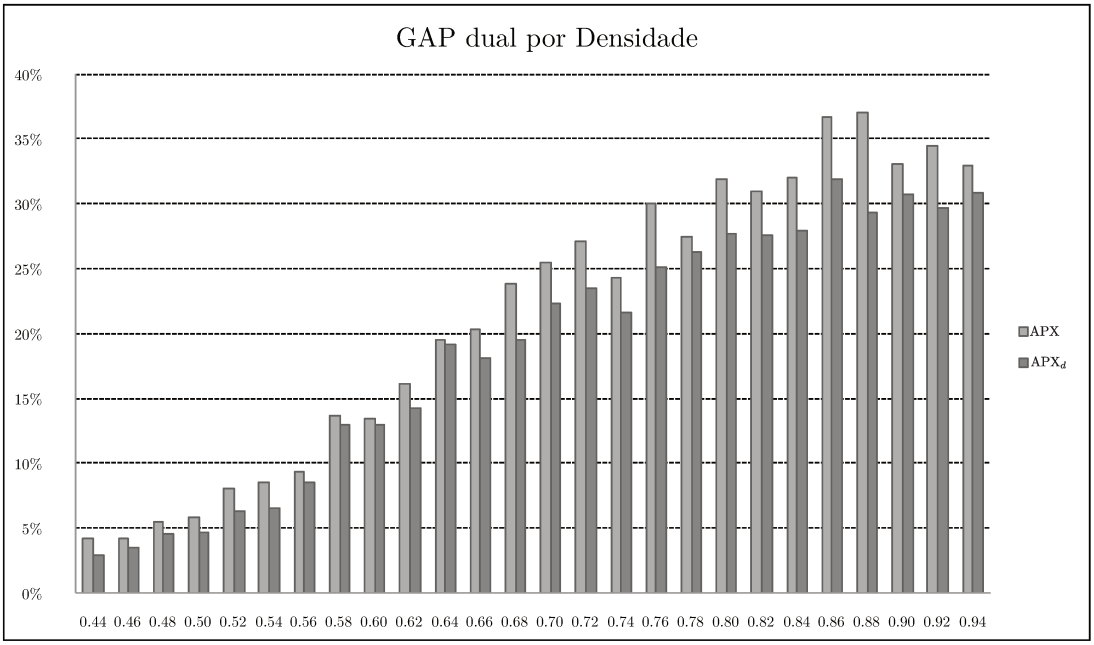


(a) Fração de ótimos.

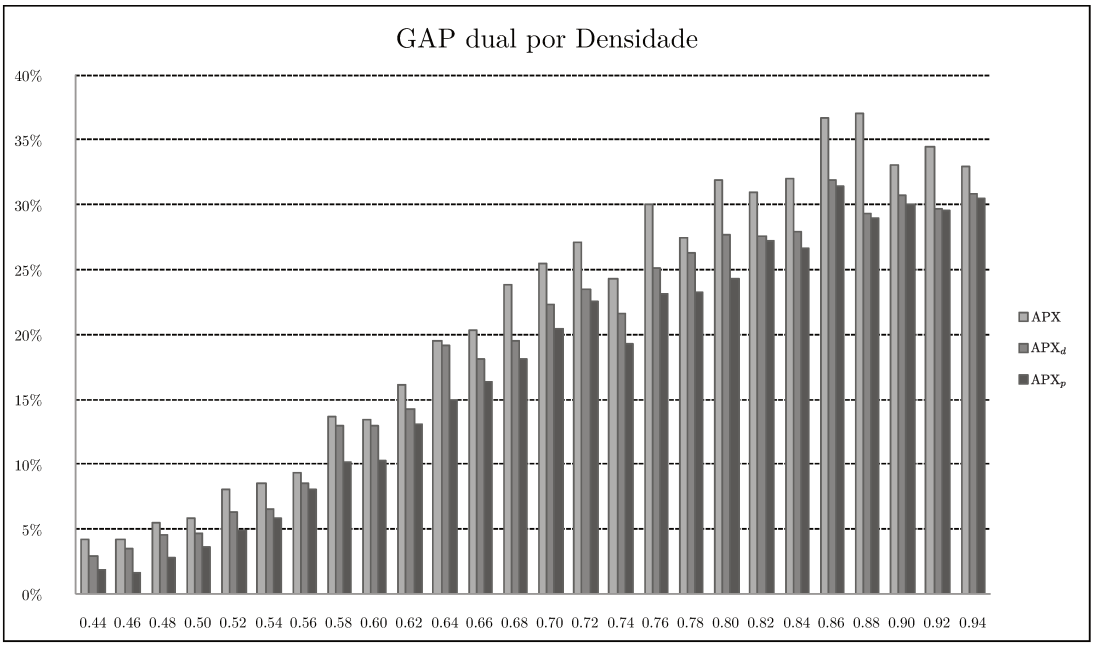


(b) Gap dual.

Figura 4.6: Comparação do APX com o EXATO.

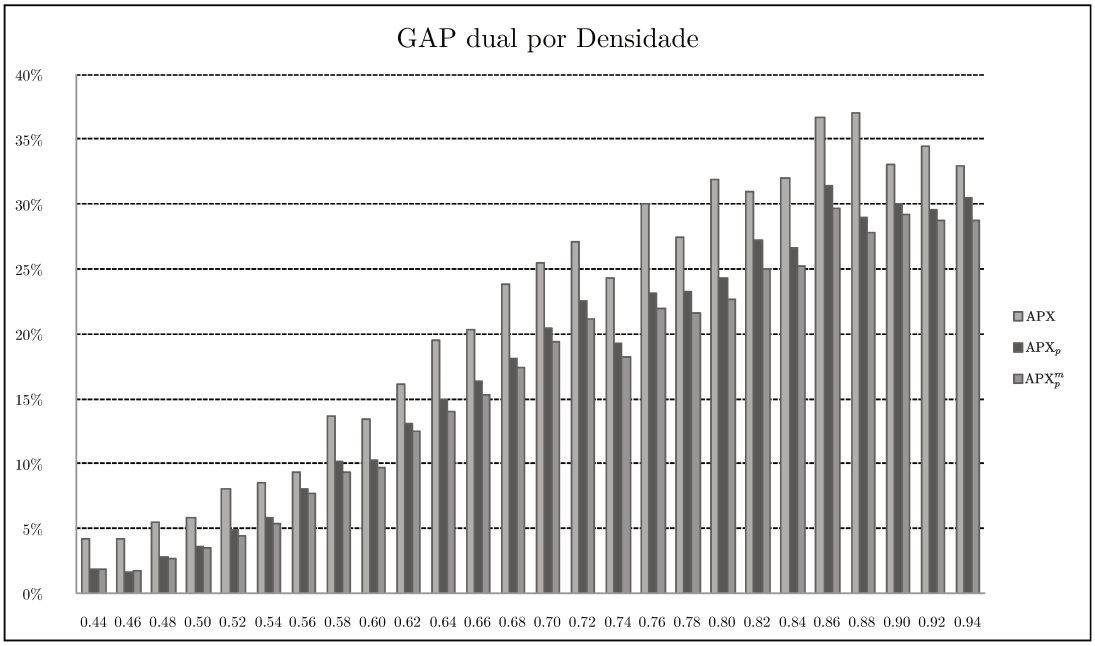


(a) APX × APX$_d$

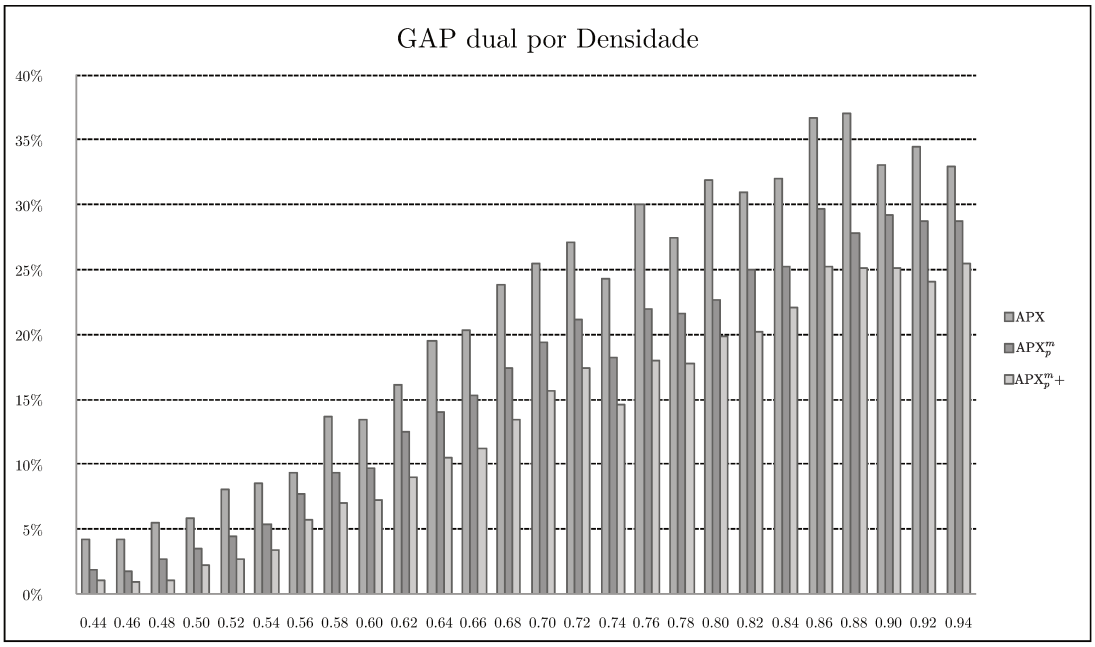


(b) APX × APX$_d$ × APX$_p$

Figura 4.7: Evolução do gap com relação ao dual conforme o APX é melhorado.



(c) APX × APX$_p$ × APX$_p^m$



(d) APX × APX$_p^m$ × APX$_p^m$+

Figura 4.7: Evolução do gap com relação ao dual conforme o APX é melhorado.

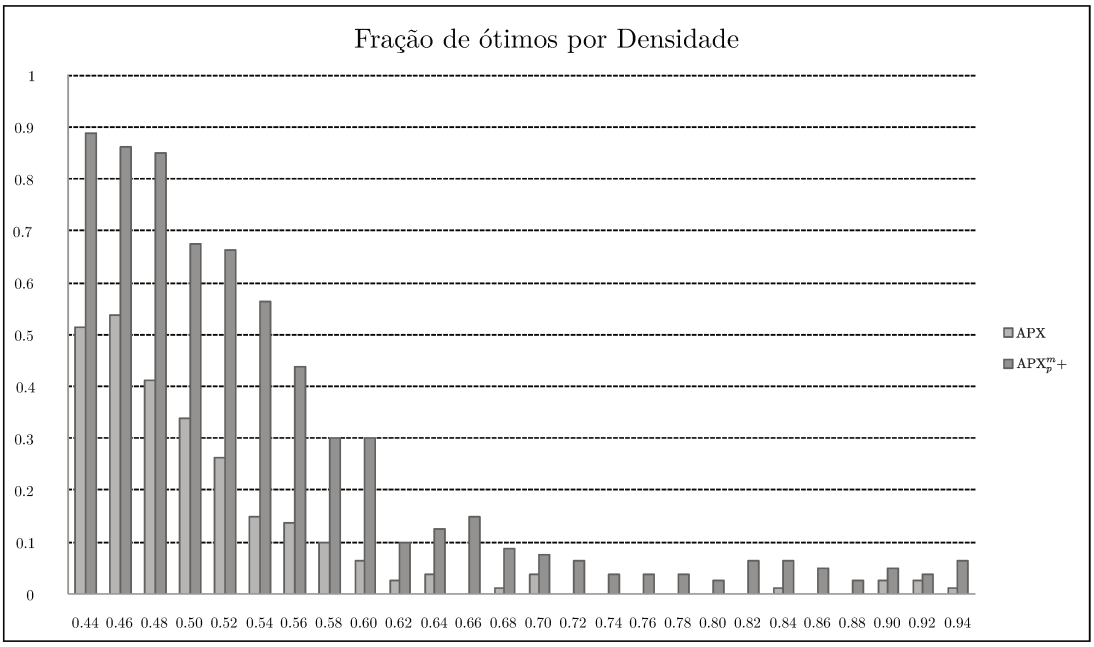


Figura 4.8: Fração de ótimos encontrados pelo APX$_p^m$+.

densidades, exceto para densidades muito baixas, onde o gap já era pequeno. Nos outros casos a redução com relação ao APX é, em média, de 5%. Portanto, considerando a versão original do algoritmo aproximado, APX, com a melhor versão obtida nesta dissertação, APX$_p^m$+, temos uma redução de aproximadamente 10% do gap na região crítica e para densidades baixas o gap ficou máximo em 5%. Já o número de ótimos encontrados no intervalo de densidade de $[0.44, 0.60]$ também aumentou sensivelmente. Contudo, para densidades maiores, esse valor continua próximo de zero.

Com relação ao tempo de execução, pelo gráfico da figura 4.9 que exibe o tempo de execução do APX$_p^m$+, vê-se que ele é baixo (menos de um segundo) e praticamente constante com relação à densidade.

O próximo passo é analisar as heurísticas, principalmente para as instâncias com densidade maior que 0.84 que correspondem àquelas em que o APX apresentou resultados insatisfatórios.

Avaliação das Heurísticas

Após estudar o comportamento do APX e suas modificações, continuaremos a avaliação experimental com as heurísticas: GARDENER e HEUR. Na comparação com o algoritmo aproximado, usaremos a versão APX_p^m+, que corresponde ao APX com todos os aperfeiçoamentos (i.e., união de ciclos via emparelhamento e busca local) e, de acordo com o exposto na seção anterior, obteve resultados superiores aos de outras variantes do

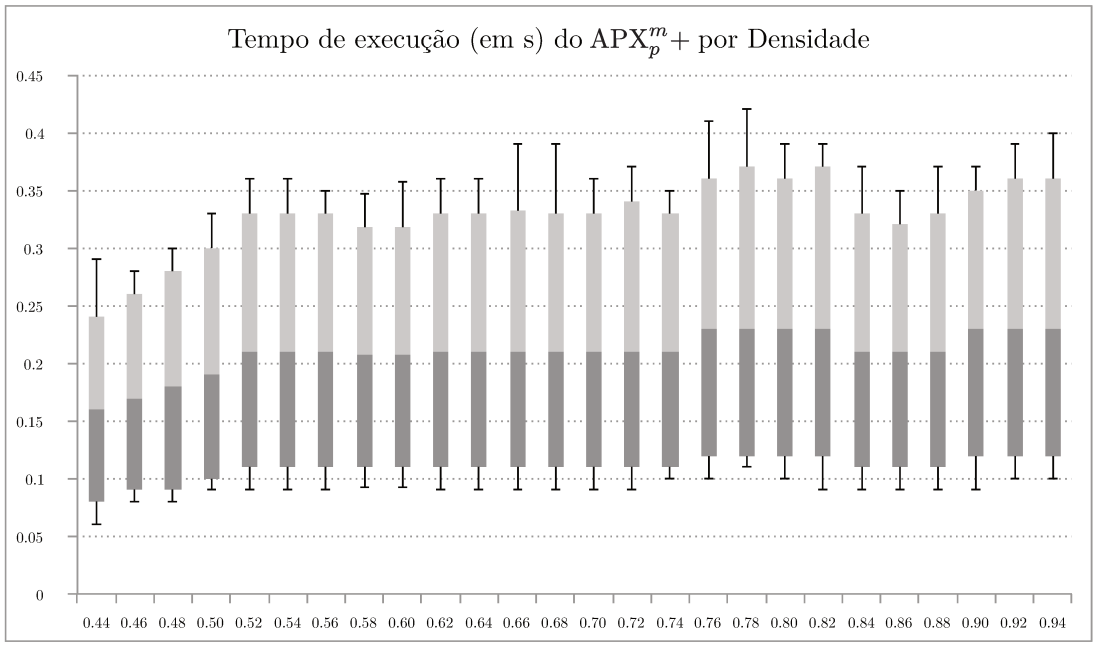


Figura 4.9: Tempo de execução do APX, para cada densidade são apresentados três percentis: 25%, 50% (mediana), 75%, e os valores máximo e mínimo.

Tabela 4.3: Valores médios (e desvios) do *gap* do GARDENER para diferentes números de iterações.

Grupo	1024 iter	max	2048 iter	max	4096 iter	max	8192 iter	max
5	0.000 ± 0.000	0.000	0.000 ± 0.000	0.000	0.000 ± 0.000	0.000	0.000 ± 0.000	0.000
6	0.006 ± 0.029	0.167	0.005 ± 0.026	0.167	0.005 ± 0.026	0.167	0.004 ± 0.024	0.167
7	0.021 ± 0.047	0.182	0.014 ± 0.039	0.167	0.011 ± 0.034	0.167	0.009 ± 0.032	0.167
8	0.058 ± 0.058	0.250	0.041 ± 0.050	0.154	0.031 ± 0.046	0.154	0.022 ± 0.041	0.125
9	0.123 ± 0.068	0.250	0.102 ± 0.067	0.250	0.082 ± 0.060	0.250	0.068 ± 0.062	0.250
10	0.136 ± 0.075	0.300	0.131 ± 0.070	0.300	0.097 ± 0.050	0.167	0.079 ± 0.055	0.167

APX. Também faremos a busca local nas soluções obtidas pelas heurísticas e, portanto, as versões dos algoritmos usadas nessa seção, seguindo a nomenclatura adotada aqui, são: GARDENER+, HEUR+. Em todas as execuções, o número de iterações do GARDENER foi pré-definido em 2048. A escolha dessa constante é baseada nos dados da tabela 4.3 em que utilizamos o conjunto de instâncias `TAMANHO` para avaliar o comportamento do GARDENER à medida que aumentamos o número de iterações. É possível notar que o gap diminiu conforme o número de iterações aumenta mas a partir de um certo ponto a melhora é insignificante.

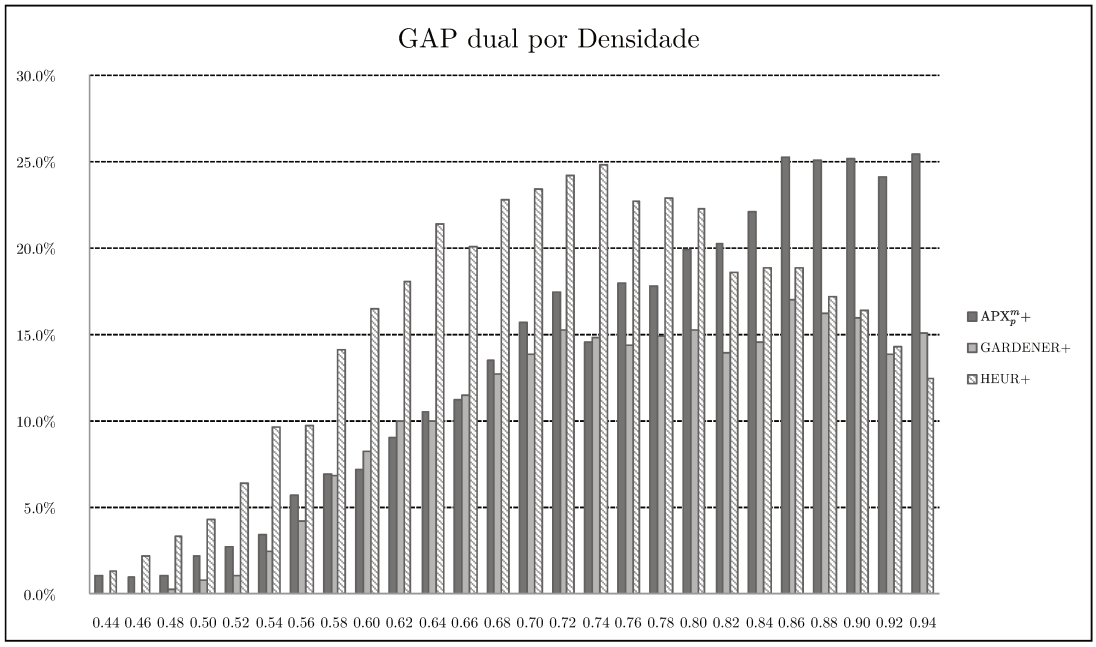
Na figura 4.10 temos os gráficos dessa comparação. Em 4.10a, temos o gráfico do gap dual para o conjunto `DENSIDADE`. Inicialmente vamos desconsiderar o GARDENER+. No intervalo de $[0.44, 82)$ o algoritmo aproximado tem os melhores resultados com um gap

máximo de 20%. Para as instâncias com densidade maior que 0.82, a heurística HEUR+ obtém os melhores resultados. Combinando os resultados de APX_p^m+ com os de HEUR+ conseguimos manter o gap menor que 20% para todas as instâncias. Para nossa surpresa, o GARDENER+, mesmo sendo uma heurística gulosa bastante simples, obtém resultados melhores que o APX e HEUR+ em quase todos os grupos, sendo pior que HEUR+ somente nas instâncias muito densas (maior 0.94) e que o APX na faixa de $[0.60, 0.64)$, com diferenças de no máximo 3% em ambos os casos. No gráfico da figura 4.10b, reduzimos o número de pontos no eixo x, agrupando as instâncias em intervalos de densidade de 0.04 em 0.04, o que facilita a visualização do comportamento dos algoritmos com a variação da densidade. Considerando os três algoritmos o gap é mantido a menos de 16%.

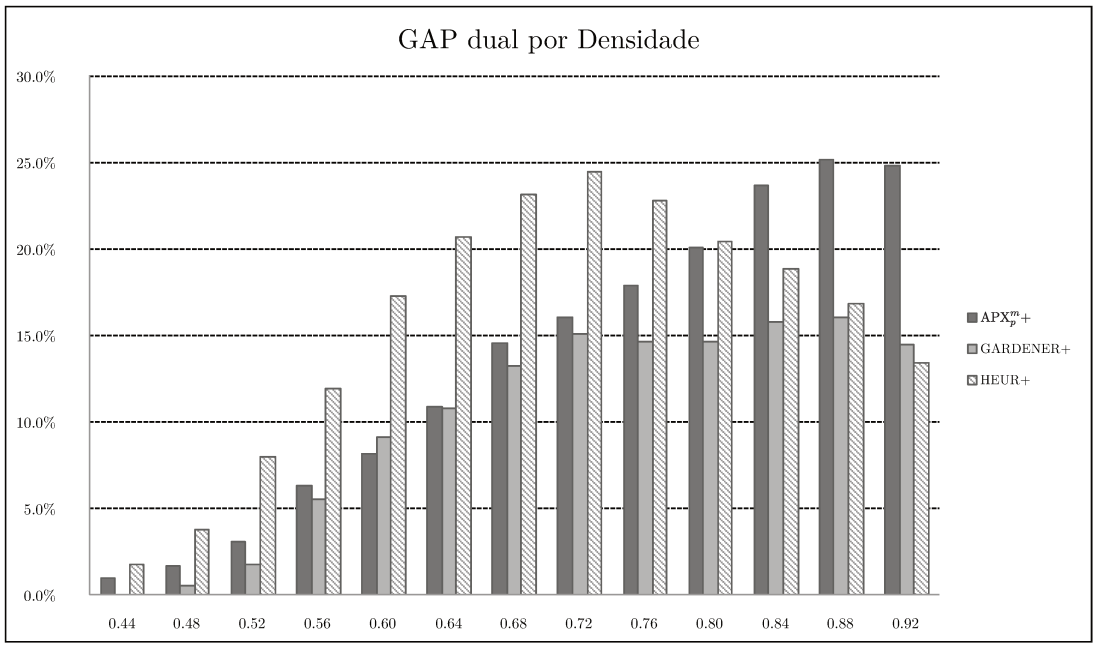
O gráfico para o conjunto de testes `TAMANHO` é dado na figura 4.10c. Observe como para instâncias pequenas o GARDENER+ encontra a solução ótima, mas a qualidade começa a degradar conforme aumentamos o tamanho das instâncias e o gap dual varia de 0% até 27%. Enquanto para HEUR+ e APX_p^m a variação do gap é menor e se mantém entre 16% e 23% para o HEUR+ e 17% e 29% para o APX_p^m.

O gráfico com o tempo de execução para as heurísticas é dado na figura 4.9. O gráfico do tempo de execução do GARDENER+ é dado na figura 4.10a. A variação é pequena com todas as instâncias resolvidas com tempos entre 2 e 10 segundos. Podemos observar que as instâncias menos densas – com grafo de subdivisão menor – em média demoram mais que as mais densas. Existem dois motivos principais para isto, o primeiro é o fato de o cortador na instância mais esparsa precisa calcular o caminho mínimo entre vértices mais vezes. O segundo, é o fato do laço da rotina de busca local repetir mais vezes nas instâncias de baixa densidade, pois, como veremos na próxima seção, a eficiência da busca local é bem maior em instâncias de baixa densidade para o GARDENER. O gráfico 4.10b mostra os tempos do GARDENER, i.e, sem a rotina de busca local. Observe que em média há uma redução de aproximadamente 5 segundos para qualquer densidade. No entanto, as instâncias de baixa densidade continuam com tempo de execução maior que as mais densas em média.

Na figura 4.10c temos o tempo de execução para a heurística HEUR+. Em média o tempo se mantém a menos de 1 segundo. No entanto este cresce com a densidade podendo demorar até 25 segundos no pior caso. A explicação para isto é que, conforme a densidade aumenta, maior o modelo de **PLI** que a heurística precisa resolver. Esta é uma indicação de que, para problemas muito grandes, essa heurística pode ser pouco atrativa. A análise do tempo de execução da rotina de busca local é irrelevante nesse caso, pois o processamento é dominado pela fase de resolução do **PLI**.

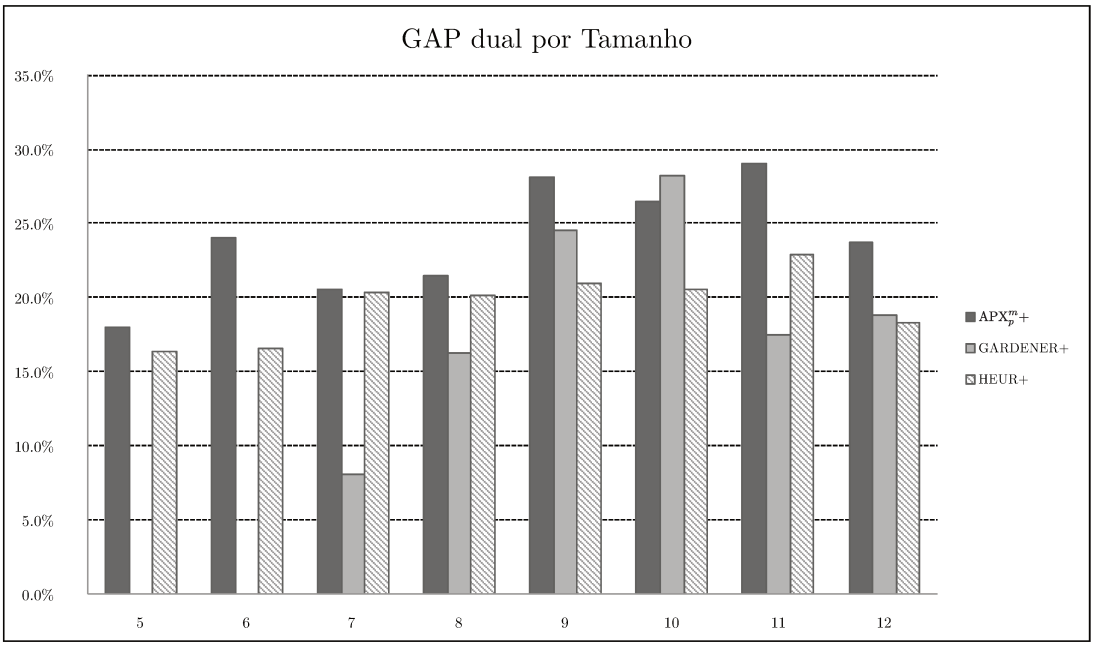


(a) Conjunto `DENSIDADE`.



(b) Conjunto `DENSIDADE` agrupado de 0.04 em 0.04.

Figura 4.10: Comparação das heurísticas com relação ao gap dual.



(c) Conjunto `TAMANHO`.

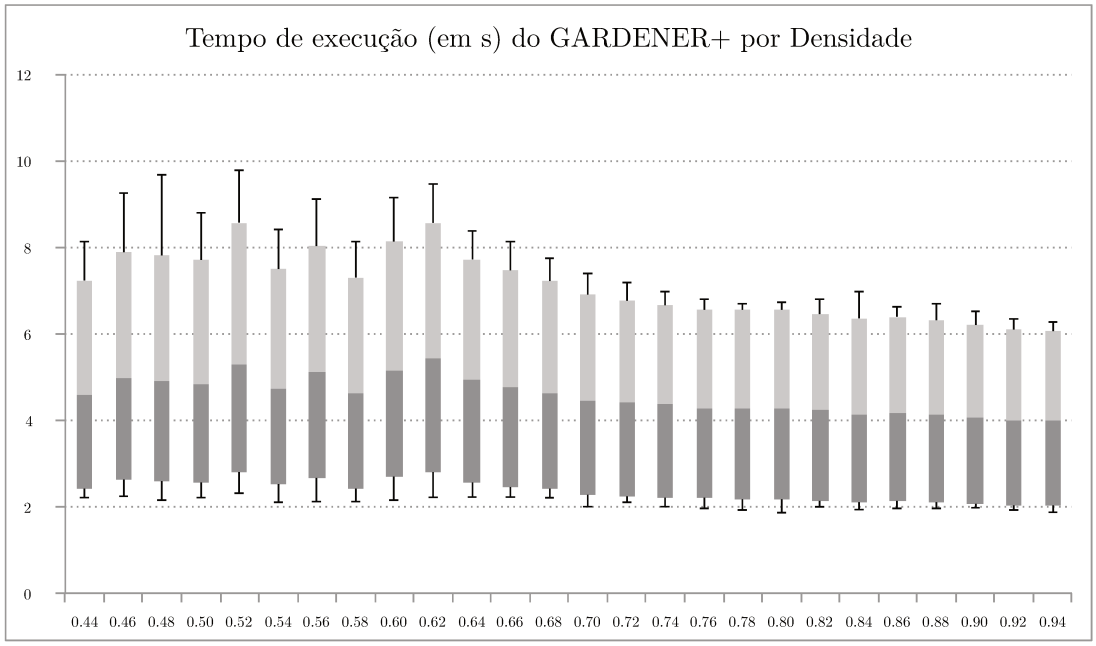
Figura 4.9: Comparação das heurísticas para o conjunto `TAMANHO`.

A Heurística da Cobertura com Múltiplos-Inícios

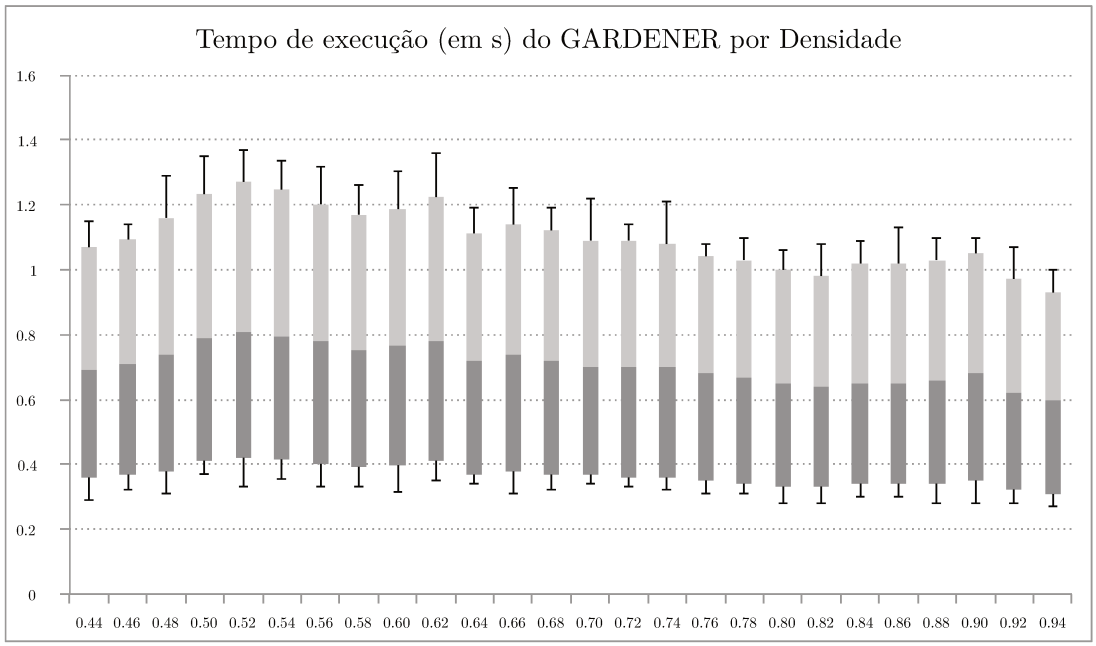
Vejamos agora os resultados obtidos com a heurística MHEUR que corresponde a HEUR combinada com elementos do GARDENER, especificamente, múltiplos-reinícios e execução da busca local sobre cada solução intermediária. A versão da heurística utilizada, conforme a nomenclatura já definida, é MHEUR+.

Os parâmetros da MHEUR+ foram fixados assim: a fração de retângulos do modelo (α) igual a 0.4 e a fração de ciclos por ponte (β) igual a 0.95. Para a escolha destes valores, resolvemos um subconjunto das instâncias de teste múltiplas vezes com diversos valores de α e de β e utilizamos o par que obteve as melhores soluções. O número de iterações da MHEUR+ foi pré-definido em 100, de forma que o tempo de execução fosse similar ao do GARDENER+.

O gráfico da figura 4.10 é análogo ao da figura 4.10a substituindo HEUR+ por MHEUR+, e compara o gap das três melhores versões dos algoritmos APX, GARDENER e MHEUR. Observe que, exceto para as instâncias esparsas, o algoritmo MHEUR+ obtém os melhores resultados e a vantagem deste aumenta conforme a densidade aumenta. Na média, o gap do GARDENER+ é aproximadamente 15% enquanto para a MHEUR+ este fica a 12% na faixa crítica e a menos de 10% nas outras densidades. Para nenhuma das densidades o APX obtém os melhores resultados e seu gap pode chegar a mais de 25% nas instâncias mais densas.

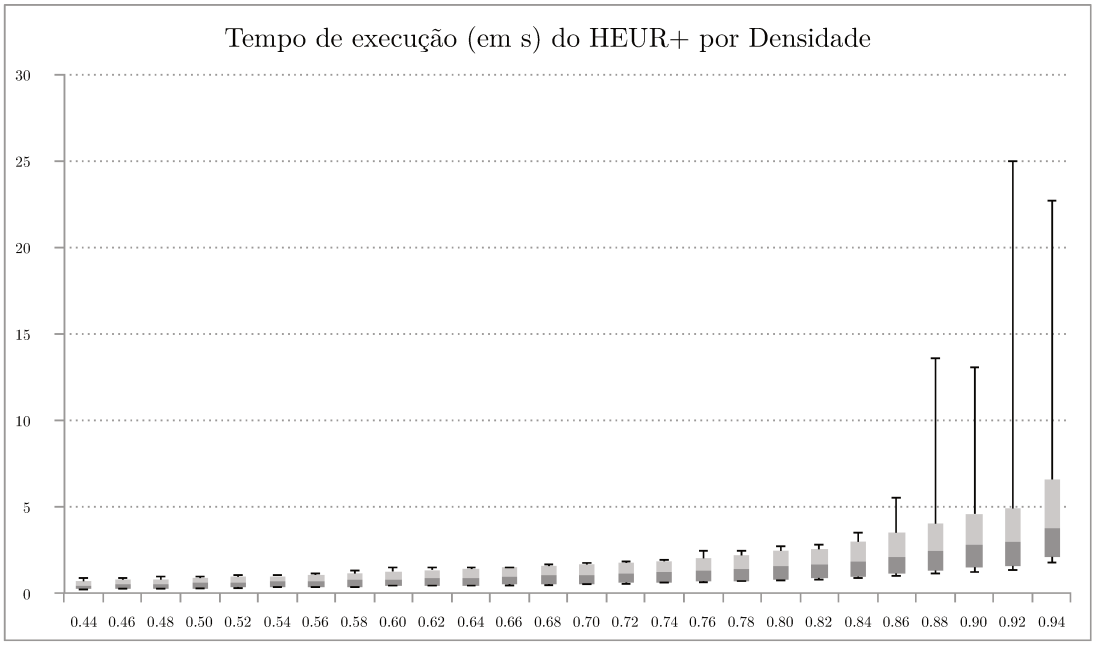


(a) Tempo de execução do GARDENER+.



(b) Tempo de execução do GARDENER.

Figura 4.10: Tempo de execução das heurísticas GARDENER+ e GARDENER, para cada densidade são apresentados três percentis: 25%, 50% (mediana), 75%, e os valores máximo e mínimo.



(c) Tempo de execução do HEUR+.

Figura 4.9: Tempo de execução da heurística HEUR+, para cada densidade são apresentados três percentis: 25%, 50% (mediana), 75%, e os valores máximo e mínimo.

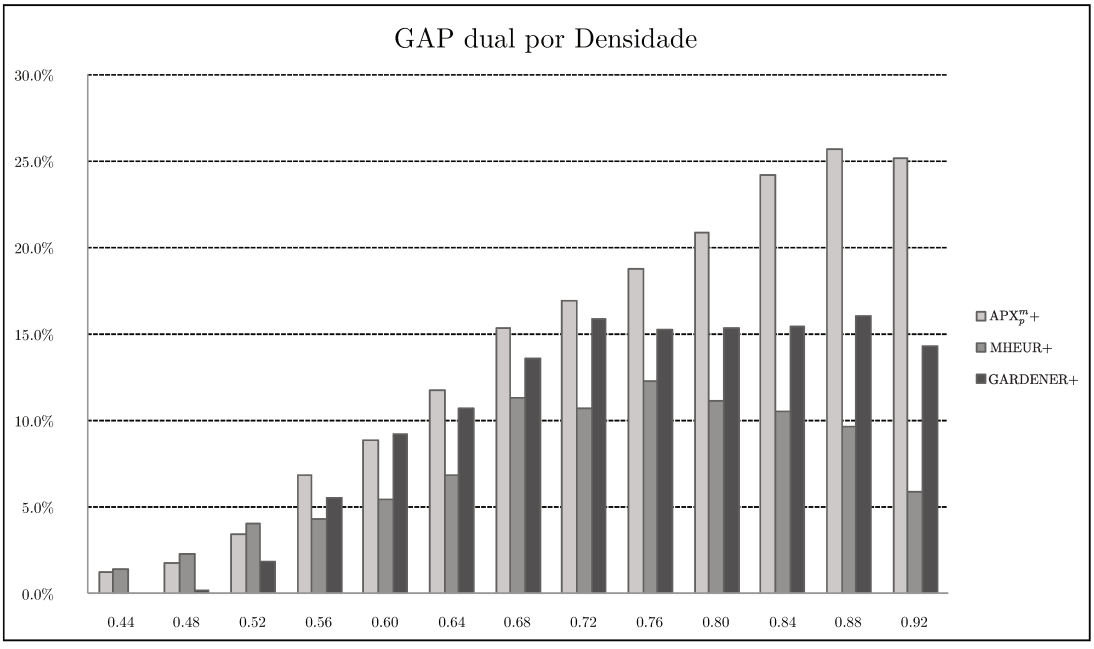


Figura 4.10: Comparação dos algoritmos APX_p^m+, GARDENER+, MHEUR+ com relação ao gap dual.

Esta análise mostra a eficiência das heurísticas, MHEUR e GARDENER. Além disso, sugere que, de fato, não parece haver uma relação direta entre a qualidade da solução da cobertura e a qualidade da solução final.

Concluímos nossa análise das heurísticas MHEUR e GARDENER com um comentário breve sobre a eficiência computacional e o tempo de execução dos algoritmos.

Concentramos a nossa atenção em duas questões: (*i*) as diferenças em tempo de execução entre MHEUR e o GARDENER e, (*ii*) o aumento no tempo computacional devido à rotina de busca local. Com relação a primeira questão, na média, o MHEUR executou por $3.6s$ com um máximo de $12.5s$. As instâncias mais densas são também as mais demoradas, provavelmente devido ao aumento no tamanho do modelo de **PLI** resolvido em cada iteração.

Em comparação, o GARDENER+ demorou em média $6.7s$ com um máximo de $11.7s$ em cada instância. No entanto, ao contrário do caso anterior, as instâncias mais esparsas foram as mais demoradas. Uma explicação para tal comportamento é que, embora as instâncias mais densas sejam as que tenham mais pixels, nas instâncias esparsas é necessário computar o caminho mínimo entre dois vértices, sendo essa, uma operação custosa.

Dissemos anteriormente que o número de iterações do MHEUR+ foi escolhido de forma que o tempo de execução fosse similar ao do GARDENER+. No entanto, as estatísticas sugerem que poderíamos aumentar esta quantidade ainda mais. Contudo, experimentos preliminares para estudar a influência desse parâmetro mostraram que, mesmo dobrando o número de iterações para 200, os resultados não mudariam de maneira significativa. Porém, como esperado, o tempo de execução seria mais próximo ao do GARDENER+.

Eficiência da Busca Local

No capítulo 3 descrevemos vizinhanças que são aplicadas às soluções na procura por padrões sub-ótimos, que quando encontrados, são tratados com vista a melhorar o custo da solução. Estas vizinhanças compõe uma busca local para o problema, recebendo como entrada uma instância do MTWTC e uma solução da mesma. Nesta seção vamos analisar a eficiência da busca local relativamente aos algoritmos GARDENER e HEUR. O objetivo deste estudo é determinar o quanto as soluções encontradas por estes algoritmos tendem a ser compostas por "padrões ruins" que podem ser eliminados pela busca local e, com isso, avaliar a necessidade ou não do gasto computacional da busca local. Para o APX esta análise já foi feita e vimos que há uma redução no gap de aproximadamente 5%.

Para os algoritmos HEUR e GARDENER, iremos comparar os gaps duais antes e após a execução da rotina de busca local para as instâncias do conjunto `DENSIDADE`. Na figura 4.11 temos o gráfico para a heurística, GARDENER (sem a rotina de busca local)

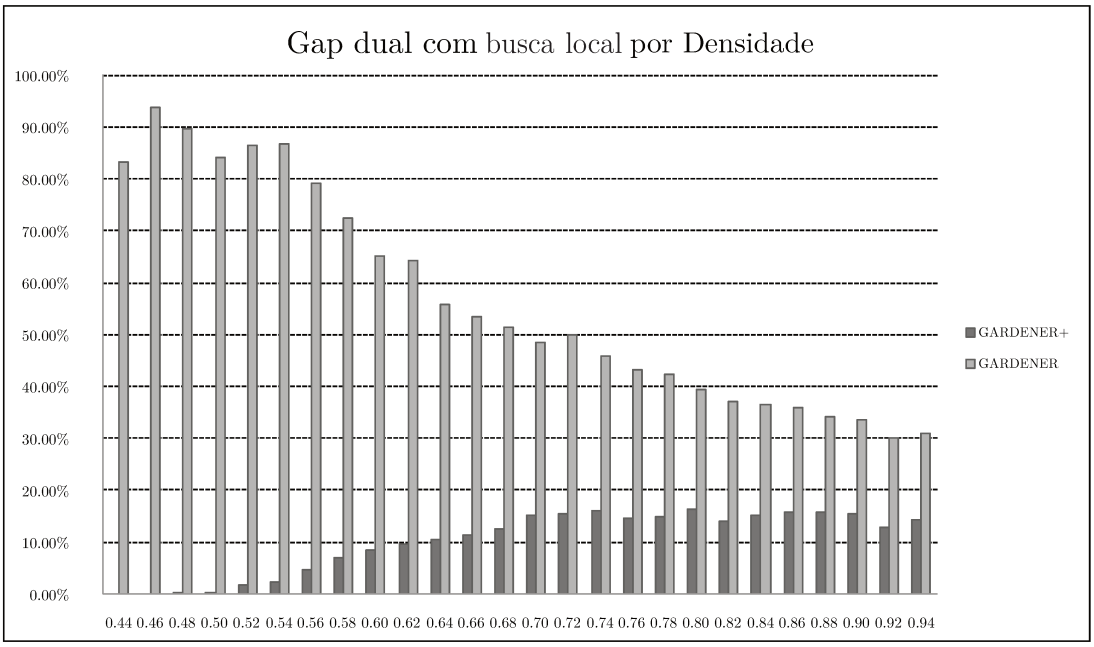
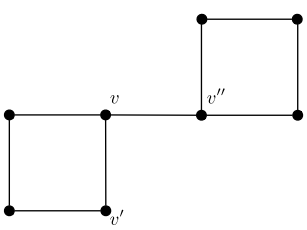


Figura 4.11: Eficiência da busca local no GARDENER.

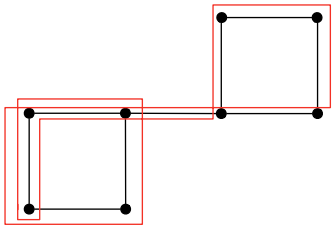
e GARDENER+ (após a aplicação da rotina). Como podemos ver é essencial a aplicação do busca local, com reduções médias do gap de 46.5%, com máximo de 93.7% e mínimo de 16.4%. Os melhores resultados ocorrem para densidades menores que 0.62. Em especial, no intervalo $[0.44, 0.48)$, o gap reduziu-se de 90% para aproximadamente 0%. Ou seja, encontramos as soluções ótimas em praticamente todas as instâncias.

Para densidades mais altas, o ganho é menor. Porém, neste caso, deve-se observar que as soluções, desconsiderando a busca local, também produzem gaps mais justos. A explicação para isso está no fato de que uma conversão sub-ótima em uma instância de baixa densidade é de difícil correção, visto que as possibilidades de movimentação do cortador são reduzidas. Por exemplo, na figura 4.12, suponha que o cortador parte do vértice inferior esquerdo, desloca-se na vertical e posteriormente na horizontal, chegando ao vértice v. Neste momento, existem duas opções de conversão. No primeiro caso, se o cortador for para o vértice v', o menor custo possível é 12, ou seja, 50% maior o custo ótimo, obtido se a conversão em v for feita para o vértice v''.

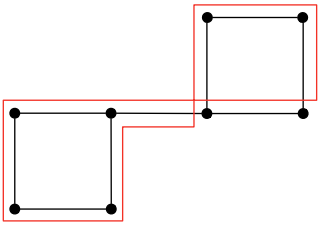
Na figura 4.13 temos a comparação entre a heurística da cobertura sem busca local (HEUR) e com busca local (HEUR+). Os ganhos foram modestos em comparação ao GARDENER, em média de 5.2% com máximo de 8.7% e mínimo de 1.70%. Novamente os ganhos maiores ocorrem nas densidades mais baixas. No entanto vale observar que embora os ganhos em altas densidades sejam menores, as soluções também são melhores, e a tendência do HEUR é mantida, ou seja, a curva é aproximadamente uma gaussiana,



(a) Duas possibilidades de conversão no vértice v.



(b) Custo mínimo 12 se conversão for em v'.



(c) Custo mínimo 8 se conversão for em v''.

Figura 4.12: Exemplo de por que soluções esparsas são difíceis de corrigir.

com máximo nas densidades entre 0.72 e 0.76.

Com essas análises encerramos a apresentação dos nossos resultados experimentais. No próximo capítulo vamos resumir quais foram as principais contribuições deste trabalho, exibindo as nossas conclusões e sugerindo direções futuras para a pesquisa sobre o MTWTC.

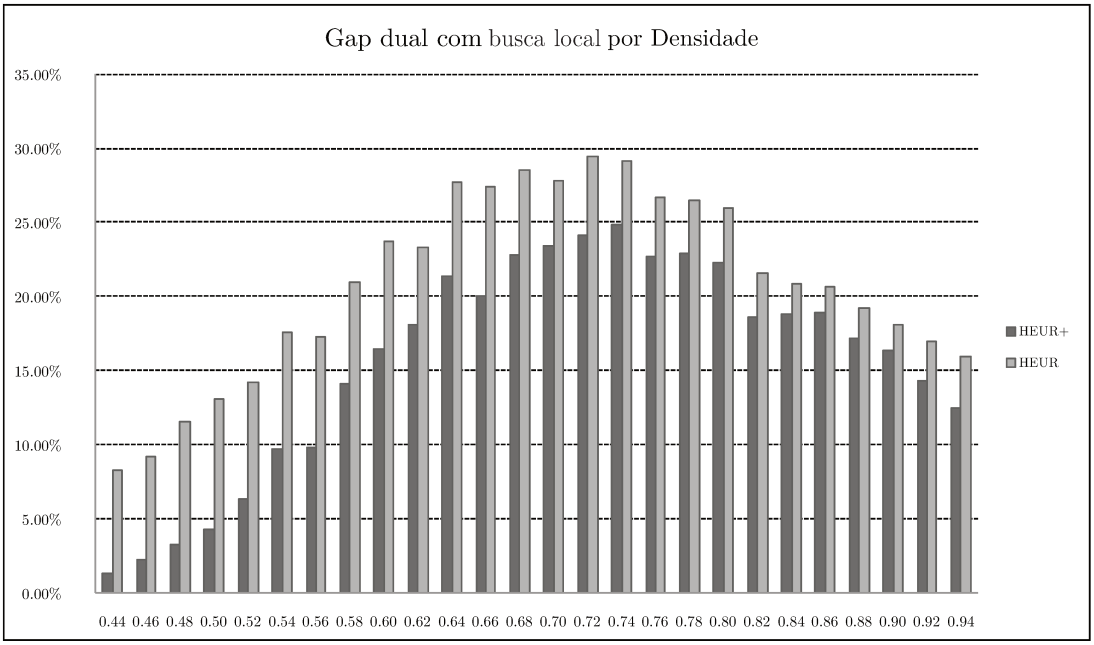


Figura 4.13: Eficiência da busca local no HEUR.

Capítulo 5

Conclusões e Trabalhos Futuros

Nessa dissertação estudamos o problema do recorte ortogonal com custo nas conversões (MTWTC). No capítulo 1 apresentamos o problema e algumas áreas de suas aplicações, que incluem: operação de máquinas de controle numérico, pintura *spray*, inspeção automática e o nosso exemplo didático do jardineiro que precisa cortar a grama de uma região com obstáculos. Ainda no capítulo 1 descrevemos uma discretização para o problema e o reduzimos para um problema em grafos, que consiste em encontrar um passeio que visita todos os vértices minimizando os custos das conversões. Em seguida fizemos uma revisão bibliográfica que incluia problemas de origem geométrica, e de teoria dos grafos e também. Demos exemplos de problemas da literatura que, embora semelhantes e com solução compatíveis, podem dar resultados ruins se aplicados diretamente ao MTWTC.

Nossa primeira tentativa de solução foi o desenvolvimento de um algoritmo exato, usando programação linear inteira (**PLI**). No capítulo 2 estudamos brevemente a teoria de **PLI** e, em seguida, descrevemos um modelo matemático e um algoritmo de *branch-and-cut* para o MTWTC. Chamamos esse algoritmo de EXATO e seus resultados foram utilizados como base de comparação em nossos experimentos. Convém destacar que, no melhor do nosso conhecimento, este é o primeiro método exato proposto na literatura para o problema.

No capítulo 3 abandonamos a obrigatoriedade de encontrar soluções ótimas e nosso foco foi em algoritmos aproximados e, heurísticas. A principal referência para o MTWTCé o artigo [6] em que ele é demonstrado ser $\mathcal{NP}$-difícil e é dado um algoritmo 3.75-aproximado para resolvê-lo, denotado nesta dissertação por APX. Propomos algumas melhorias para o APX e o implementamos com o objetivo analisá-lo experimentalmente já que o artigo original se limita a resultados teóricos. Desenvolvemos ainda duas heurísticas: a do jardineiro (GARDENER) e a da cobertura (HEUR). A primeira utiliza uma ideia bastante simples de "cortar a grama" de maneira gulosa, complementada com técnicas de aleato-

rização e múltiplos inícios. A segunda, mais elaborada, resolve um problema de cobertura por retângulos, utilizando **PLI**e alguns procedimentos do APX. Completando o capítulo 3, propomos uma rotina de busca local, composta por três vizinhanças que recebem uma solução e fazem mudanças locais de forma a reduzir o seu custo. Essa busca local se tornou de fundamental importância, principalmente quando combinado com a heurística do jardineiro, com a qual obtivemos os melhores resultados experimentais. Cabe observar que, até onde pudemos verificar, não existe na literatura atual para o MTWTC nenhuma proposta de algoritmo heurístico, seja construtivo ou de busca local. Como consequência, boa parte dos desenvolvimentos descritos no capítulo 3 constitui-se de trabalho original.

Após a construção do nosso arcabouço de algoritmos para o MTWTC, no capítulo 4 fizemos uma análise experimental e comparativa. O primeiro passo consistiu em criar conjuntos de instâncias para compor um *benchmark*. Para isso, definimos uma medida de densidade da instância que, intuitivamente, mede a liberdade de movimentação do cortador. Isto foi feito em virtude dos nossos experimentos terem mostrado que este é um indicador que espelha bem a dificuldade de se resolver uma instância. A partir dessa medida desenvolvemos um gerador de instâncias aleatórias parametrizado pela densidade e pelo lado de um quadrado que inscreve a região inicial. Criamos dois conjuntos de testes denominados `DENSIDADE` e `TAMANHO`. No primeiro grupo encontram-se instâncias de baixa até alta densidade para os quais o lado do quadrado foi fixado em oito. No segundo grupo variamos o tamanho do lado do quadrado, restringindo o intervalo de densidade a uma faixa associada às instâncias mais difíceis.

Com o *benchmark* de instâncias passamos para a avaliação experimental dos algoritmos. Para o EXATO, concluímos que as instâncias mais difíceis tem densidade entre 0.75 e 0.90 e que o algoritmo começa a se tornar inviável para instâncias cujo grafo de subdivisão possui mais de 160 arestas.

Em seguida, avaliamos o algoritmo aproximado APX e suas variantes propostas neste trabalho. O algoritmo APX original tem um gap de dualidade que varia de aproximadamente 5% para instâncias com densidades muito baixas, de 0.44 a 0.46. Observamos que esse gap cresce conforme aumentamos a densidade, até um máximo de 35% na média. O pior resultado prático foi um gap de 89%, portanto, muito abaixo do fator 3.75, o que levanta uma questão sobre o quão justo é o limite teórico. Aplicando as melhorias propostas, tivemos uma redução no gap, sem reverter contudo, a tendência de piorá-lo conforme a densidade aumenta. Na média, com as nossas alterações conseguimos uma redução de aproximadamente 10% no gap e um aumento no número de ótimos de 20% a 40% nas instâncias com densidade menor que 0.60, enquanto que, para densidades maiores, esse acréscimo não foi expressivo.

Por fim avaliamos as heurísticas GARDENER, HEUR e MHEUR, e comparamos os resultados com aqueles alcançados pela melhor versão do APX. De um modo geral, verificou-

se que o uso da estratégia de multiplos-reinícios, associado à resolução de formulações **PLI**, foi bastante eficaz para a obtenção de soluções de boa qualidade para o MTWTC. Tal fato ficou evidente pelo melhor desempenho do MHEUR quando comparado aos seus concorrentes, à exceção das instâncias com baixa densidade.

O estudo experimental do MTWTC como realizado nesta dissertação não tem precedentes na literatura. Além disso, tornamos de domínio público o primeiro *benchmark* de instâncias para o problema, incluindo os melhores limitantes duais e primais conhecidos.

Uma vez concluída a revisão do conteúdo desta dissertação, recapitulamos abaixo as principais contribuições do trabalho:

1. formulação matemática do MTWTC como um **PLI**;
2. desenvolvimento do primeiro algoritmo exato para o problema, o qual foi concebido a partir do modelo **PLI** e de um algoritmo de *branch-and-cut*;
3. proposta de modificações do melhor algoritmo aproximado conhecido para o problema, as quais, ainda mantendo o limite de aproximação, permitem a obtenção de resultados práticos de qualidade superior;
4. desenvolvimento das primeiras heurísticas de que se tem notícia para o MTWTC;
5. criação e disponibilização pública de um *benchmark* de testes para o problema;
6. realização de testes comparativos sem precedentes na literatura nas quais foram avaliados vários algoritmos propostos para o MTWTC.

Parte dessas constribuições deram origem a dois trabalhos em congressos internacionais, a saber:

- *Experimental Evaluation of Algorithms for the Orthogonal Milling Problem with Turn Costs.* de Assis, I. e de Souza, C. C. .Proceedings of the 10th International Symposium on Experimental Algorithms (SEA 2011), Lecture Notes in Computer Science, volume 6630, pp. 304–314.
- *Heuristics for the Orthogonal Discrete Milling Problem.* de Assis, I. , Crepaldi, B. e de Souza, C. C. . Fourth International Workshop on Model-Based Mathheuristics, Angra dos Reis, Brasil, 17 a 20 de setembro de 2012, 12 páginas. Anais não publicados (ver `www2.ic.uff.br/matheuristics2012`).

Para concluir esse documento, damos algumas sugestões de trabalhos futuros. Em termos heurísticos, acreditamos que ainda há espaço para melhora, principalmente se considerarmos as instâncias na faixa crítica, com densidades entre 0.75 e 0.90. A rotina

de busca local é uma indicação de que encontrar buscas locais eficazes para o problema, embora não pareça uma tarefa trivial, pode levar a resultados melhores que os atuais. Ainda sobre as rotinas de busca local, alguma questões teóricas interessantes são:

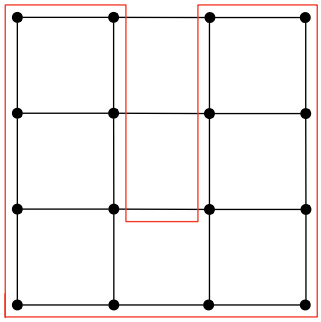
1. Dado o grafo de solução, qual o passeio Euleriano que resulta no melhor *milling tour*? Esse problema é $\mathcal{NP}$-difícil? O trabalho apresentado em [22] parece estar relacionado com esta questão e sugere que, de fato, o problema é difícil.

2. Existe uma condição necessária e suficiente de fácil verificação para determinar se estamos em um ótimo local com relação à vizinhança do passeio Euleriano?

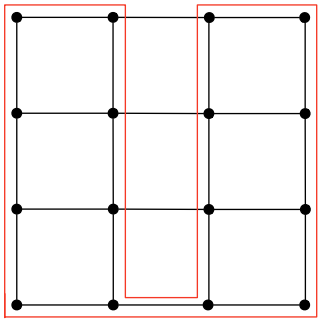
Durante a pesquisa, surgiram outros questionamentos não relacionados ao pós-processamento. Abaixo os mais relevantes.

Se o grafo de subdivisão tem uma ponte, i.e, uma aresta que quando removida desconecta o grafo, necessariamente esta deve ser percorrida pelo menos duas vezes em qualquer solução do MTWTC. No entanto, para todas as instâncias que resolvemos, existe uma solução ótima em que toda aresta do grafo de subdivisão é percorrida no máximo duas vezes em qualquer solução do MTWTC. Assim, uma questão interessante é determinar se esta é uma propriedade do problema. Com uma resposta positiva, imediatamente já teríamos um modelo de **PLI** melhor, substituindo a constante n da restrição (2.21) por 2. Ainda sobre o modelo de **PLI**, um trabalho interessante é fazer análise poliédrica para encontrar a dimensão do politopo correspondente a envoltória convexa das soluções inteiras e possivelmente novas desigualdades válidas, demonstrando que algumas delas definem facetas.

Nosso foco neste trabalho foi minimizar o custo em conversões no problema ortogonal. Diversas variações do problema, surgem ao permitimos o cortador se mover em outras direções e não restringirmos o *pocket* a polígonos ortogonais. Outra opção é alterar a função objetivo para considerar também a distância percorrida e não apenas o custo de conversões. Esse último caso é particularmente interessante pois duas soluções ótimas em custo de conversões podem percorrer distâncias diferentes (ver exemplo da figura 5.1), o que pode ser relevante em situações práticas.



(a) Custo em conversões 8 e distância percorrida 12.



(b) Custo em conversões 8 e distância percorrida 14.

Figura 5.1: Exemplo de soluções com o mesmo custo em conversões mas que percorrem distâncias distintas.

Referências Bibliográficas

[1] Canonical Ltd. Ubuntu for you. `http://www.ubuntu.com/ubuntu`. (página visitada em Março de 2012).

[2] Free Software Foundation Inc. GCC: the GNU compiler collection. `http://gcc.gnu.org`. (página visitada em Março de 2012).

[3] IBM ILOG CPLEX Optimizer. `http://www.ibm.com/software/integration/optimization/cplex-optimizer/`. (página visitada em Março de 2012).

[4] A. Aggarwal, D. Coppersmith, S. Khanna, R. Motwani, and B. Schieber. The angular-metric traveling salesman problem. In *SODA '97: Proceedings of the Eighth annual ACM-SIAM Symposium on Discrete Algorithms*, pages 221–229, New Orleans, LA, United States, 1997.

[5] R. Ahuja, T. Magnanti, and J. Orlin. *Network flows: theory, algorithms, and applications*. Prentice Hall, 1993.

[6] E. Arkin, M. Bender, E. Demaine, S. Fekete, J. Mitchell, and S. Sethia. Optimal covering tours with turn costs. *SIAM Journal on Computing*, 35(3):531–566, 2005.

[7] E. Arkin, S. Fekete, and J. Mitchell. The lawnmower problem. In *Proceedings of the Fifth Canadian Conference on Computational Geometry*, pages 461–466, Waterloo, ON, Canada, 1993.

[8] E. Arkin, S. Fekete, and J. Mitchell. Approximation algorithms for lawn mowing and milling. *Computational Geometry*, 17(1-2):25–50, 2000.

[9] M. Bazaraa, J. Jarvis, H. Sherali, and M. Bazaraa. *Linear programming and network flows*. Wiley Online Library, 2 edition, 1990.

[10] E. Benavent and D. Soler. The directed rural postman problem with turn penalties. *Transportation Science*, 33(4):408–418, 1999.

[11] J. Bondy and U. Murty. *Graph theory*. Graduate texts in mathematics. Springer, 2007.

[12] A. Corberán, R. Martí, E. Martínez, and D. Soler. The rural postman problem on mixed graphs with turn penalties. *Computers Operations Research*, 29(7):887–903, 2002.

[13] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to algorithms*. MIT Press, 3rd edition, 2009.

[14] B. Dawes, D. Abrahans, and R. Rivera. Boost C++ libraries. `http://www.boost.org`. (página visitada em Março de 2012).

[15] I. de Assis, C. de Souza, and B. Crepaldi. Heuristics for the orthogonal discrete milling problem. In *Matheuristics'2012*, 2012.

[16] J. Edmonds. Paths, trees, and flowers. *Canadian Journal of Mathematics*, 17(3):449–467, 1965.

[17] A. S. Foundation. Apache thrift. `http://thrift.apache.org`. (página visitada em Março de 2012).

[18] R. Geisberger and C. Vetter. Efficient routing in road networks with turn costs. In P. M. Pardalos and S. Rebennack, editors, *SEA*, volume 6630 of *Lecture Notes in Computer Science*, pages 100–111. Springer, 2011.

[19] M. Grötschel, L. Lovász, and A. Schrijver. Corrigendum to our paper "the ellipsoid method and its consequences in combinatorial optimization". *Combinatorica*, 4(4):291–295, 1984.

[20] M. Grötschel, L. Lovász, and A. Schrijver. The ellipsoid method and its consequences in combinatorial optimization. *Combinatorica*, 1(2):169–197, 1984.

[21] K. Iwano, P. Raghavan, and H. Tamaki. The traveling cameraman problem, with applications to automatic optical inspection. *Algorithms and Computation*, 834:29–37, 1994.

[22] B. Jackson. Removable cycles in 2-connected graphs of minimum degree at least four. *J. London Math. Soc.(2)*, 21:385–392, 1980.

[23] N. Karmarkar. A new polynomial-time algorithm for linear programming. *Combinatorica*, 4(4):373–396, 1984.

[24] V. Klee and G. J. Minty. How good is the simplex algorithm? In O. Shisha, editor, *Inequalities*, volume III, pages 159–175. Academic Press, New York, 1972.

[25] V. Kolmogorov. Blossom V software. `http://pub.ist.ac.at/ vnk/software.html`. (página visitada em Março de 2012).

[26] V. Kolmogorov. Blossom V: A new implementation of a minimum cost perfect matching algorithm. *Mathematical Programming Computation*, 1(1):43–67, 2009.

[27] J. Mitchell. *Geometric shortest paths and network optimization*. Elsevier Science, North-Holland, 2000.

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

[29] G. Reinelt. *The traveling salesman: computational solutions for TSP applications*. Springer-Verlag, Berlin, Heidelberg, 1994.

[30] M. G. Resende and C. C. Ribeiro. Greedy randomized adaptive search procedures: Advances, hybridizations, and applications. In M. Gendreau and J.-Y. Potvin, editors, *Handbook of Metaheuristics*, volume 146 of *International Series in Operations Research and Management Science*, pages 283–319. Springer US, 2010.

[31] C. Stein and D. P. Wagner. Approximation algorithms for the minimum bends traveling salesman problem. In K. Aardal and B. Gerards, editors, *IPCO*, volume 2081 of *Lecture Notes in Computer Science*, pages 406–422. Springer, 2001.

[32] B. Stroustrup. *The C++ programming language:*. Addison-Wesley, 2000.

[33] C. Umans and W. Lenhart. Hamiltonian cycles in solid grid graphs. In *Annual Symposium on Foundation of Computer Science*, pages 496–505, Miami Beach, FL, USA, 1997.

[34] L. Wolsey. *Integer programming*. Wiley-Interscience Series in Discrete Mathematics and Optimization. Wiley, 1st edition, 1998.