Universidade Estadual de Campinas
Instituto de Computação

Pedro Olímpio Nogueira de Oliveira Pinheiro

Partição de Grafos Eulerianos em Circuitos

CAMPINAS
2021

Pedro Olímpio Nogueira de Oliveira Pinheiro

Partição de Grafos Eulerianos em Circuitos

Dissertação apresentada ao Instituto de Computação da Universidade Estadual de Campinas como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação.

Orientador: Prof. Dr. Zanoni Dias
Coorientador: Prof. Dr. Cid Carvalho de Souza

Este exemplar corresponde à versão final da Dissertação defendida por Pedro Olímpio Nogueira de Oliveira Pinheiro e orientada pelo Prof. Dr. Zanoni Dias.

CAMPINAS
2021

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

P655p

Pinheiro, Pedro Olímpio Nogueira de Oliveira, 1998-
Partição de grafos eulerianos em circuitos / Pedro Olímpio Nogueira de Oliveira Pinheiro. – Campinas, SP : [s.n.], 2021.

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

1. Otimização combinatória. 2. Programação linear inteira. 3. Heurística (Computação). I. Dias, Zanoni, 1975-. II. Souza, Cid Carvalho de, 1963-. III. Universidade Estadual de Campinas. Instituto de Computação. IV. Título.

Informações para Biblioteca Digital

Título em outro idioma: Partition of eulerian garphs in circuits
Palavras-chave em inglês:
Combinatorial optimization
Integer linear programming
Heuristic (Computer science)
Área de concentração: Ciência da Computação
Titulação: Mestre em Ciência da Computação
Banca examinadora:
Zanoni Dias [Orientador]
Kelly Cristina Poldi
Rafael Crivellari Saliba Schouery
Data de defesa: 08-10-2021
Programa de Pós-Graduação: Ciência da Computação

Identificação e informações acadêmicas do(a) aluno(a)
- ORCID do autor: https://orcid.org/0000-0002-8993-7451
- Currículo Lattes do autor: http://lattes.cnpq.br/0383874426039834

Universidade Estadual de Campinas
Instituto de Computação

Pedro Olímpio Nogueira de Oliveira Pinheiro

Partição de Grafos Eulerianos em Circuitos

Banca Examinadora:

- Prof. Dr. Zanoni Dias
  Instituto de Computação - Unicamp
- Profa. Dra. Kelly Cristina Poldi
  Instituto de Matemática, Estatística e Computação Científica - Unicamp
- Prof. Dr. Rafael Crivellari Saliba Schouery
  Instituto de Computação - Unicamp

A ata da defesa, assinada pelos membros da Comissão Examinadora, consta no SIGA/Sistema de Fluxo de Dissertação/Tese e na Secretaria do Programa da Unidade.

Campinas, 08 de outubro de 2021

Agradecimentos

Agradeço aos meus pais, Gledson e Laura, pelo constante apoio e orientação, pelo suporte incondicional e por tudo que fizeram e fazem por mim.

Agradeço aos meus irmãos, Junior e Manoel, pela amizade sensacional e por estarem comigo em todos os momentos.

Agradeço aos meus irmãos, Ester e Theo, pela alegria que trouxeram e trazem.

Agradeço aos meus orientadores, Zanoni Dias e Cid C. de Souza, pela excelente orientação. Agradeço pela paciência e pelos conselhos. Agradeço pela oportunidade e auxílio durante essa jornada.

Agradeço aos membros da república Revolta da Cajuína, por todos os momentos que passamos juntos e todo o apoio.

Agradeço aos membros do LOCo e todos os companheiros de disciplinas, que ajudaram nesse percurso, em especial ao Alexsandro, André e Allan.

O presente trabalho foi realizado com apoio do Conselho Nacional de Desenvolvimento Científico e Tecnológico - Brasil (CNPq), processo #130980/2019-6.

O presente trabalho foi realizado com apoio da Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP), processo nº 2019/25410-3.

Agradeço a todos que me ajudaram nesse ciclo.

Resumo

Uma partição em circuitos de um grafo consiste em particionar o conjunto de arestas desse grafo em subconjuntos não vazios, de forma que o grafo induzido por cada subconjunto seja um circuito no grafo original. Todo grafo euleriano pode ser particionado em circuitos, porém, diversas partições de tamanhos diferentes podem ser viáveis. No problema da Máxima Partição em Circuitos, o objetivo é encontrar uma partição em circuitos de um grafo euleriano com a maior cardinalidade possível. Para esse problema, propomos heurísticas gulosas, heurísticas que resolvem o modelo de Programação Linear Inteira (`PLI`) proposto por Caprara, Panconesi e Rizzi [7], com apenas um subconjunto das variáveis e sem geração de colunas, e algoritmos exatos que resolvem o mesmo modelo `PLI` com geração de colunas. Para o modelo `PLI`, foi necessária a criação de um algoritmo para resolver o problema de *pricing*. Criamos um *benchmark* de instâncias e executamos diversos experimentos para avaliar esses algoritmos. Verificamos que a heurística com uso de `PLI` obtém os melhores resultados, obtendo solução ótima em quase 70% das instâncias utilizadas. Também abordamos o problema da Máxima Partição em Circuitos Alternados, em que as arestas do grafo possuem uma cor, entre preta e cinza, e os circuitos devem ser alternados, ou seja, arestas seguidas devem ter cores diferentes. Para esse problema, propomos heurísticas gulosas, uma heurística baseada na meta-heurística Busca Tabu e algoritmos exatos com uso de `PLI`. Criamos outro *benchmark* de instâncias, executamos experimentos e comparamos os resultados obtidos com resultados de algoritmos encontrados na literatura. Executar o algoritmo exato usando as variáveis e limitantes obtidos pelas heurísticas foi o método que mostrou melhores resultados, superando os algoritmos da literatura e obtendo o resultado ótimo em 96.4% das instâncias usadas. Além disso, usamos as soluções do problema da Máxima Partição em Circuitos Alternados para resolver problemas de Rearranjo de Genomas e os nossos métodos também apresentaram os melhores resultados.

Abstract

A cycle partition of a graph consists of partitioning the set of edges of that graph into non-empty subsets, so that the graph induced by each subset is a cycle in the original graph. Every Eulerian graph can be partitioned into cycles, however, several partitions of different sizes may be viable. In the Maximum Eulerian Cycle Decomposition problem, the objective is to find a cycle partition of an Eulerian graph with the greatest possible cardinality. For this problem, we propose greedy heuristics, heuristics that solve the Integer Linear Programming (`ILP`) model proposed by Caprara, Panconesi and Rizzi [7], with only a subset of the variables and no column generation, and exact algorithms that solve the same `ILP` model with column generation. For the `ILP` model, it was necessary to create an algorithm to solve the pricing problem. We create a *benchmark* of instances and run several experiments to evaluate these algorithms. We verified that the `ILP` based heuristic obtains the best results, obtaining an optimal solution in almost 70% of the instances used. We also approached the problem of Maximum Alternating-Cycle Decomposition, in which the edges of the graph have a color, between black and gray, and the cycle must be alternated, that is, consecutive edges must have different colors. For this problem, we propose greedy heuristics, a heuristic based on the Tabu Search metaheuristic and exact algorithms using `ILP`. We create another *benchmark* of instances, run experiments and compare the results with results from algorithms found in the literature. Running the exact algorithm using the variables and bounds obtained by the heuristics was the method that showed the best results, surpassing the algorithms in the literature and obtaining the optimal result in 96.4% of the instances used. Furthermore, we use Maximum Alternating-Cycle Decomposition solutions to solve Genome Rearrangement problems and our methods also showed the best results.

Lista de Figuras

Lista de Tabelas

Lista de Algoritmos

Sumário

Capítulo 1

Introdução

O problema de particionar o conjunto de arestas de um grafo em uma classe específica de subgrafos é um problema recorrente na literatura [12, 20, 25]. Neste trabalho tratamos da partição de um grafo em circuitos, ou seja, da separação das arestas do grafo em uma coleção de subconjuntos disjuntos, cuja união é igual ao conjunto de arestas do grafo e cada um deles corresponde a um circuito.

Diversos trabalhos encontrados na literatura, como os de Mynhardt e van Bommel [20] e Rodger [25], visam encontrar cópias de um mesmo grafo dentro de um outro grafo maior, porém, nos problemas que abordamos, visamos encontrar circuitos com qualquer comprimento. No entanto, o objetivo dos problemas que tratamos não é somente particionar as arestas do grafo, mas encontrar essa partição de forma que ela tenha cardinalidade máxima.

Para um grafo admitir uma partição em circuitos, é necessário que ele seja euleriano ou que cada uma de suas componentes conexas seja. Portanto, neste trabalho consideraremos somente grafos eulerianos, já que a solução ótima para um grafo desconexo é a união das soluções ótimas para suas componentes conexas. Chamamos o problema de particionar as arestas de um grafo euleriano no maior número de circuitos de problema da Máxima Partição em Circuitos (*Maximum Eulerian Cycle Decomposition* - `MAX-ECD`).

Outro problema que abordamos neste trabalho foi uma variação do `MAX-ECD`: o problema da Máxima Partição em Circuitos Alternados (*Maximum Alternating-Cycle Decomposition* - `MAX-ACD`). Para o `MAX-ACD`, as arestas do grafo possuem uma cor, entre preta e cinza, e os circuitos devem ser alternados, isto é, arestas consecutivas devem ter cores diferentes. Para um grafo admitir uma partição em circuitos alternados, além de ser um grafo euleriano, é necessário que o número de arestas pretas e cinza incidentes em cada vértice sejam iguais.

Caprara [5] mostrou que o `MAX-ECD` e o `MAX-ACD` estão na classe NP-difícil. Chen [8] apresentou algoritmos para o problema da Ordenação por Rearranjos que dependem de soluções do `MAX-ACD`. Algoritmos de Ordenação por Rearranjos são frequentemente usados no estudo da distância evolucionária entre espécies.

O problema da Distância de Rearranjos é um problema que visa encontrar a menor sequência de rearranjos necessária para transformar o genoma de uma espécie no genoma de outra. Nesse problema, o genoma é geralmente representado como uma permutação, onde cada elemento da permutação corresponde a um gene. Usamos os n primeiros in-

teiros positivos para representar um genoma com n genes. Como podemos representar um dos genomas como uma permutação ordenada, o problema da Distância de Rearranjos pode pode ser resolvido através do problema de Ordenação por Rearranjos em uma permutação [22].

Os principais rearranjos estudados são os de reversão, transposição e *Double cut-and-join* (`DCJ`). O problema de ordenar por qualquer um desses rearranjos é NP-difícil [4, 5, 8]. O problema de Ordenação por Reversão e Transposição também é NP-difícil [21].

Bafna e Pevzner [2] apresentaram um algoritmo de aproximação com fator 7/4 para o problema de Ordenação por Reversão. Esse fator foi melhorado por Christie [9] para 3/2 e por Lin e Jiang [19] para $1.4193 + \epsilon$. Para a Ordenação por `DCJ`, Chen [8] apresentou um algoritmo com fator de aproximação de $1.4167 + \epsilon$ e Jiang *et al.* [17] apresentaram um algoritmo FPT randomizado com fator de aproximação de $4/3 + \epsilon$.

Caprara, Lancia e See-Kiong Ng [6] apresentaram um modelo de Programação Linear Inteira (`PLI`) com o método *Branch-and-Price* para resolver o `MAX-ACD` em grafos de *breakpoint*, que é a classe de grafos utilizada nos algoritmos de Ordenação por Rearranjos.

Caprara, Panconesi e Rizzi [7] apresentaram um algoritmo de aproximação para o `MAX-ECD` com fator de aproximação $O(\log n)$. Krivelevich *et al.* [18] mostraram que esse mesmo algoritmo tem, na verdade, fator de aproximação $\Theta(\sqrt{\log n})$.

Neste trabalho, propomos algoritmos exatos e heurísticos para o `MAX-ECD` e para o `MAX-ACD`. Para o `MAX-ECD`, usamos heurísticas gulosas, heurísticas que resolvem parcialmente o modelo `PLI` e algoritmos exatos que resolvem o modelo `PLI` completo. Por fim, avaliamos o desempenho desses algoritmos entre si usando um conjunto de instâncias que geramos. Para o `MAX-ACD`, usamos heurísticas gulosas, uma heurística baseada na metaheurística Busca Tabu e algoritmos exatos que resolvem o modelo `PLI` completo, para qualquer classe de grafos, e avaliamos o desempenho desses algoritmos, comparando com algoritmos encontrados na literatura, mais uma vez, usando um conjunto de instância que geramos.

Este trabalho está organizado da seguinte forma. No Capítulo 2 apresentamos os principais conceitos, definições e notações usadas. No Capítulo 3 apresentamos o problema da Máxima Partição em Circuitos, assim como os algoritmos utilizados e os resultados obtidos. No Capítulo 4 descrevemos o problema da Máxima Partição em Circuitos Alternados, os algoritmos utilizados e os resultados obtidos. Por fim, no Capítulo 5 fazemos nossas considerações finais e discutimos trabalhos futuros.

Capítulo 2

Fundamentação Teórica

Neste capítulo descrevemos os principais conceitos utilizados neste trabalho.

2.1 Definições de Grafos

As definições relacionadas a grafos são baseadas no livro de Bondy e Murty [3], a menos quando dito contrário.

Um **grafo** G é um par ordenado (V, E), onde V é um conjunto de vértices e E é um conjunto de arestas (pares não ordenados de vértices). Se dois vértices $u, v \in V$ possuem uma aresta entre si, ou seja, $(u, v) \in E$, dizemos que u e v são vizinhos ou adjacentes. Os vértices em que uma aresta incide são chamados seus extremos. Neste trabalho vamos considerar que todos os grafos são simples, ou seja, o extremo de uma aresta é sempre diferente do outro extremo e não existem duas arestas com os mesmos extremos. A Figura 2.1 mostra um exemplo de um grafo com quatro vértices $\{a, b, c, d\}$ e seis arestas $\{(a, b), (b, c), (c, d), (d, a), (a, c), (b, d)\}$.

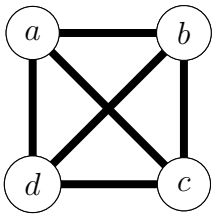


Figura 2.1: Exemplo de um grafo com quatro vértices e seis arestas.

O **grau** de um vértice é o número de arestas que incidem nesse vértice. O grau de um vértice v é denotado por $d(v)$.

Um grafo $G' = (V', E')$ é um **subgrafo** de um grafo $G = (V, E)$ se $V' \subseteq V$ e $E' \subseteq E$ e as extremidades das arestas de E' estão em V'. Ou seja, G' pode ser obtido a partir de G removendo vértices e arestas. O **subgrafo induzido** por um conjunto de arestas $E' \subseteq E$ de um grafo $G = (V, E)$ é o subgrafo de G cujo conjunto de vértices são os extremos das arestas em E' e o conjunto de arestas é E'.

Uma **trilha** em um grafo é uma sequência de vértices e arestas $(v_0, e_0, v_1, \ldots, v_{l-1}, e_{l-1}, v_l)$, em que v_i e v_{i+1} são os vértices extremos da aresta e_i. Note que pode haver repetição de vértices, mas não de arestas.

Um **caminho** em um grafo é uma trilha sem repetição de vértices. Um **circuito** em um grafo é uma trilha em que o primeiro vértice é igual ao último. O comprimento de um caminho (ou circuito) é a quantidade de arestas desse caminho (ou circuito). Um **ciclo** é um circuito em que todos os vértices são diferentes, exceto o primeiro e o último.

Os grafos da Figura 2.2 ilustram um caminho e um circuito no grafo da Figura 2.1, onde as arestas vermelhas representam as arestas do caminho e do circuito.

(a) (b)

Figura 2.2: Exemplo de (a) caminho e de (b) circuito ou ciclo.

Um grafo $G = (V, E)$ é **conexo** quando existe um caminho em G, que começa em u e termina em v, para qualquer par de vértices $u, v \in V$. Uma **componente conexa** de um grafo é um subgrafo conexo maximal de G.

Uma **trilha euleriana** é uma trilha que passa por cada aresta do grafo exatamente uma vez. Um **circuito euleriano** é um circuito que passa por cada aresta do grafo exatamente uma vez. Um **grafo euleriano** é um grafo em que existe um circuito euleriano.

Uma **partição** das arestas de um grafo $G = (V, E)$ é uma coleção de subconjuntos $\{E_1, E_2, \dots\}$ não vazios de E em que a interseção de qualquer dois subconjuntos é vazia e a união de todos os subconjuntos é igual a E. Uma **partição em circuitos** de um grafo $G = (V, E)$ é uma partição de E em subconjuntos $\{E_1, E_2, \dots\}$ tal que o subgrafo induzido por cada E_i é um circuito. A Figura 2.3 mostra um grafo e sua partição em dois circuitos, um com arestas vermelhas e outro com arestas azuis. Um grafo admite uma partição em circuitos se e somente se for um grafo euleriano ou se cada uma de suas componentes conexas for um subgrafo euleriano.

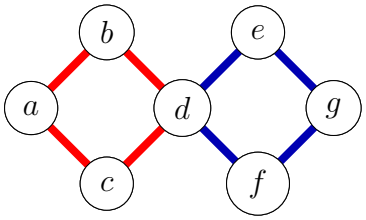


Figura 2.3: Exemplo de partições em circuitos.

Chamaremos de **grafos bicoloridos** os grafos cujas arestas são divididas em dois conjuntos, pretas (P) e cinza (C). Um **caminho alternado** em um grafo bicolorido é um caminho em que duas arestas consecutivas possuem cores diferentes. Um **circuito alternado** é um circuito em que duas arestas consecutivas possuem cores diferentes. Na Figura 2.4, um exemplo de circuito alternado é o circuito que contém as arestas $(1, 3), (3, 4), (4, 2)$ e $(2, 1)$. Diremos que $d_P(v)$ é o número de arestas pretas que incidem no vértice v e que $d_C(v)$ é o número de arestas cinza que incidem no vértice v.

Um **grafo de breakpoint** [2] é um grafo bicolorido $G(\pi)$, construído com base em uma permutação dos $n+2$ primeiros números naturais, que serão os vértices do grafo. Descreveremos a seguir o processo de construção de um grafo de *breakpoint*. Seja $\pi = (\pi_1, \pi_2, \ldots, \pi_{n-1}, \pi_n)$ uma permutação dos n primeiros inteiros positivos. Adicione o elemento $\pi_0 = 0$ no início da permutação e o elemento $\pi_{n+1} = n+1$ no final. Os vértices do grafo serão os $n+2$ números da permutação estendida. Para cada $i \in \{0, 1, \ldots, n\}$, se $|\pi_i - \pi_{i+1}| \neq 1$ adicione uma aresta preta de π_i para π_{i+1}. Seja π_i^{-1} a posição que o número i está na permutação π, de modo que $\pi_{\pi_i}^{-1} = i$. Para cada $i \in \{0, 1, \ldots, n\}$, se $|\pi_i^{-1} - \pi_{i+1}^{-1}| \neq 1$ adicione uma aresta cinza de π_i^{-1} para π_{i+1}^{-1}. Por exemplo, a Figura 2.4 representa o grafo de *breakpoint* para a permutação $\pi = (1, 3, 6, 2, 4, 5)$.

Por convenção, desenhamos os grafos de *breakpoint* com os vértices alinhados horizontalmente, da esquerda para a direita na ordem da permutação, com as arestas pretas como retas na horizontal e com as arestas cinza formando arcos.

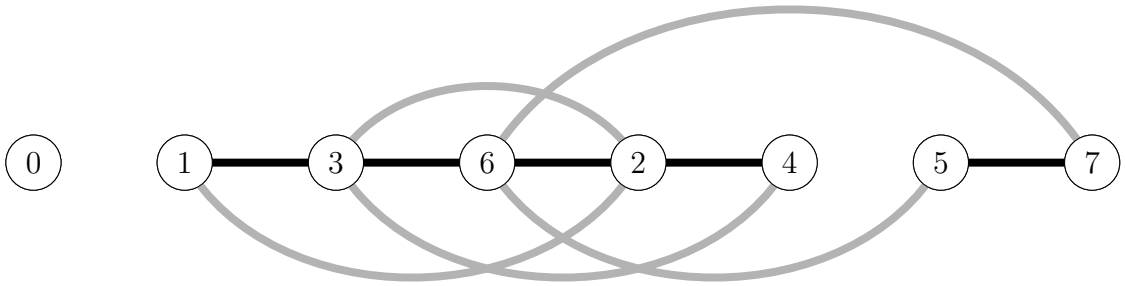


Figura 2.4: Exemplo de grafo de *breakpoint* para a permutação $\pi = (1, 3, 6, 2, 4, 5)$.

Um **grafo ponderado** é um grafo $G = (V, E)$, em que cada aresta $e \in E$ está associada com um número real $w(e)$, que chamamos de peso dessa aresta.

2.2 Busca Tabu

Busca Tabu é uma estratégia para resolver problemas de otimização combinatória. É um algoritmo capaz de utilizar muitos outros métodos, como algoritmos de programação linear e heurísticas especializadas, para superar ótimos locais [13, 14].

Para descrever a Busca Tabu, Glover [13, 14] utiliza o seguinte problema genérico: dado um conjunto X de soluções factíveis, encontrar um elemento $x \in X$ que minimize uma função $c(x)$ (possivelmente não linear).

Em seguida, define uma operação s, chamada **movimento**, que dado uma solução $x \in X$ gera uma nova solução $s(x) = x' \in X$. O conjunto de todos os movimentos possíveis é dado por S. O subconjunto de movimentos que podem ser aplicados em uma solução x é chamado $S(x)$.

O Algoritmo 1 descreve a heurística *Hill Climbing* que Glover [13] mostra antes de detalhar Busca Tabu.

Algoritmo 1 Heurística Hill Climbing.

1: Escolha uma solução inicial $x \in X$.
2: Escolha algum $s \in S(x)$ de tal modo que

$$c(s(x)) < c(x). \tag{2.1}$$

Se não existe tal s, então x é um ótimo local. Retorne x.
3: Faça $x \leftarrow s(x)$ e volte ao passo 2.

A principal limitação da heurística *Hill Climbing* é que um ótimo local encontrado pode não ser um ótimo global. A Busca Tabu continua a busca mesmo após encontrar um ótimo local, permitindo movimentos que não melhoram a função objetivo e evitando ótimos locais já visitados [13].

Para isso, um subconjunto T de S é criado, chamado de lista tabu. O conjunto T contém movimentos que foram executados em iterações recentes da heurística e que não são desejados que ocorram novamente ou que sejam desfeitos na iteração atual.

Inicialmente T é um conjunto vazio. A cada iteração, T é atualizado. Os movimentos que estão na lista a muitas iterações saem e o movimento realizado na última iteração é adicionado.

O Algoritmo 2 descreve a meta-heurística Busca Tabu de Glover [13].

Algoritmo 2 Meta-heurística Busca Tabu.

1: Escolha uma solução inicial $x \in X$ e faça $x^* \leftarrow x$, onde x^* é a melhor solução encontrada até o momento. Inicie o contador de iterações $k \leftarrow 0$ e a lista tabu $T = \emptyset$.
2: Se $S(x) \setminus T = \emptyset$, então vá para o passo 4. Caso contrário, faça $k \leftarrow k + 1$ e escolha $s_k(x) \in S(x) \setminus T$ de tal forma que

$$s_k(x) = \arg \min_{s \in S(x) \setminus T} \{c(s(x))\}. \tag{2.2}$$

3: Faça $x \leftarrow s_k(x)$. Se $c(x) < c(x^*)$, faça $x^* \leftarrow x$.
4: Se o critério de parada for satisfeito ou se $S(x) \setminus T = \emptyset$, pare o algoritmo e retorne x^*. Caso contrário, atualize T e continue do passo 2.

2.3 Programação Linear

Muitos problemas podem ser modelados como a otimização (maximização ou minimização) de uma função objetivo, dados recursos limitados ou um conjunto de restrições. Se essa função objetivo puder ser descrita como uma função linear de certas variáveis e as restrições puderem ser descritas como equações lineares ou inequações lineares dessas variáveis, então temos um problema de **Programação Linear** (`PL`) [10].

Chamamos de n o número de variáveis do problema e de m o número de restrições. As n variáveis são chamadas x_j, para cada $1 \leq j \leq n$. Para escrever um problema de programação linear são necessárias algumas constantes. Primeiro, n números reais c_j,

para cada $1 \leq j \leq n$, que são os pesos das variáveis na função objetivo. Em seguida, m números reais b_i, para cada $1 \leq i \leq m$, que são os números no lado direito das restrições. Por fim, mn números reais a_{ij}, para cada $1 \leq i \leq m, 1 \leq j \leq n$, que são os coeficientes das variáveis no lado esquerdo das restrições.

Na **forma canônica** consideramos que a função objetivo é de maximização, que todas as m restrições são de menor ou igual e que existem n restrições de não-negatividade, isto é, restrições que garantem que cada variável x_j é maior ou igual a 0. Desejamos encontrar valores reais para as variáveis x_j que maximize a Função Objetivo (2.3) e satisfaça as Restrições (2.4) e (2.5).

$$\max \sum_{j=1}^{n} c_j x_j \tag{2.3}$$

sujeito a:

$$\sum_{j=1}^{n} a_{ij} x_j \leq b_i, \qquad \forall i \in \{1, 2, \ldots m\} \tag{2.4}$$

$$x_j \geq 0, \qquad \forall j \in \{1, 2, \ldots, n\} \tag{2.5}$$

Todos os problemas de PL podem ser escritos na forma canônica. Uma solução $x' = (x'_1, x'_2, \ldots, x'_n)$ é factível se satisfaz todas as Restrições (2.4) e (2.5) e é infactível caso contrário. Chamamos de $\mathbb{X}$ o conjunto de todas as soluções factíveis para um problema de PL. Caso $\mathbb{X} = \emptyset$, dizemos que esse problema é infactível. Uma **solução ótima** x^* de um problema de PL é uma solução que pertence ao conjunto $\mathbb{X}$ cujo valor da função objetivo $\sum_{j=1}^{n} c_j x_j^*$ é maior ou igual ao de qualquer outra solução $x' \in \mathbb{X}$.

Por exemplo, para o problema PL:

$$\max x_1 + 2x_2 \tag{2.6}$$

sujeito a:

$$-2x_1 + x_2 \leq 2 \tag{2.7}$$

$$2x_1 + x_2 \leq 8 \tag{2.8}$$

$$x_1 \leq 3 \tag{2.9}$$

$$x_1 \geq 0 \tag{2.10}$$

$$x_2 \geq 0 \tag{2.11}$$

a solução ótima é $x_1 = 1.5$ e $x_2 = 5$, com valor 11.5 na função objetivo. A Figura 2.5 mostra a projeção no plano cartesiano do problema de PL descrito. As retas representam as restrições e as setas nas extremidades das retas indicam qual região do plano é válida para aquela restrição. A região verde representa as soluções factíveis, que chamaremos de **região viável**. A seta vermelha indica a direção de crescimento da função objetivo e o ponto vermelho é a solução ótima.

Para cada problema de PL existe um outro problema de PL associado, chamado **dual**.

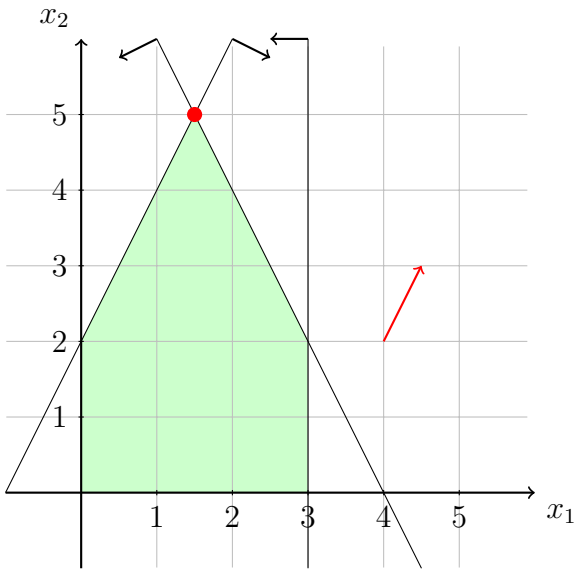


Figura 2.5: Projeção de um problema de programação linear no plano cartesiano, exibindo restrições, região viável, direção de crescimento da função objetivo e solução ótima.

O problema original é chamado de **primal**. Existem fortes relações entre um problema primal e seu dual [11].

O dual de um problema dual é igual ao problema primal. O sentido da função objetivo do problema dual sempre é oposto ao do problema primal, isto é, se o problema primal for de maximização, o dual é de minimização e, se o problema primal for de minimização, o problema dual é de maximização. Se ambos os problemas (primal e dual) tiverem ao menos uma solução factível, a solução ótima do problema primal terá valor na função objetivo igual ao valor na função objetivo da solução ótima do problema dual. Isso significa que, para um problema primal de maximização, qualquer solução x' factível do primal terá valor na função objetivo $\sum_{j=0}^{m} c_j x_j$ menor ou igual ao valor na função objetivo do problema dual de qualquer solução factível do dual. Além disso, a solução ótima do problema primal pode ser rapidamente obtida se a solução ótima do problema dual for conhecida, portanto, para alguns problemas é melhor resolver o problema dual do que o primal.

Para um problema primal de maximização, um limitante inferior, isto é, um valor que é menor ou igual ao ótimo do problema, é chamado de **limitante primal**. O valor na função objetivo de qualquer solução factível do problema é um limitante primal. Já um limitante superior, isto é, um valor que é maior ou igual ao ótimo do problema, é chamado de **limitante dual**. O valor na função objetivo do dual de qualquer solução factível do problema dual é um limitante dual do problema primal.

O problema dual é escrito usando as mesmas constantes a_{ij}, b_i e c_j do problema primal. No dual existirão m variáveis y_i, para cada $1 \leq i \leq m$, e n restrições, além das m restrições de não negatividade. As constantes b_i passam a ser o peso das variáveis na função objetivo e as constantes c_j viram o lado direito das restrições. As constantes a_{ij} continuam sendo os coeficientes das variáveis nas restrições, porém, ocorrem em uma ordem diferente do primal. No dual, os coeficientes da primeira restrição são $a_{11}, a_{21}, \ldots, a_{m1}$, da segunda restrição são $a_{12}, a_{22}, \ldots, a_{m2}$ e assim em diante. Se o problema primal estiver na forma

canônica (2.3)-(2.5), o problema dual será:

$$\min \sum_{i=1}^{m} b_i y_i \tag{2.12}$$

sujeito a:

$$\sum_{i=1}^{m} a_{ij} y_i \geq c_j, \qquad \forall j \in \{1, 2, \ldots n\} \tag{2.13}$$

$$y_i \geq 0, \qquad \forall i \in \{1, 2, \ldots, m\} \tag{2.14}$$

Por exemplo, o dual do problema primal (2.6)-(2.11) é

$$\min 2y_1 + 8y_2 + 3y_3 \tag{2.15}$$

sujeito a:

$$-2y_1 + 2y_2 + y_3 \geq 1 \tag{2.16}$$

$$y_1 + y_2 \geq 2 \tag{2.17}$$

$$y_1 \geq 0 \tag{2.18}$$

$$y_2 \geq 0 \tag{2.19}$$

$$y_3 \geq 0 \tag{2.20}$$

A solução ótima desse problema dual é $y_1 = 0.75$, $y_2 = 1.25$ e $y_3 = 0$, com valor 11.5 na função objetivo.

2.3.1 Método Simplex

Existem vários métodos para resolver um problema de PL. Alguns desses métodos executam em tempo polinomial no tamanho do problema. Porém, o método mais usado é o algoritmo **simplex**, que é um algoritmo que não executa em tempo polinomial no pior caso. O *simplex* é o método mais usado porque, na prática, resolve a maioria dos problemas de PL rapidamente [10].

A interpretação geométrica do método *simplex* é bem simples. Ele começa em algum vértice da região viável e executa uma sequência de iterações. Em cada iteração, ele se move ao longo de uma aresta da região viável, saindo do vértice atual e parando em um vértice vizinho cujo valor na função objetivo seja maior ou igual do que o do vértice atual (e geralmente é maior). O algoritmo *simplex* termina quando atinge um máximo local, que é um vértice do qual todos os vértices vizinhos têm um valor na função objetivo menor ou igual. O algoritmo também para se identificar que o problema é ilimitado, isto é, sempre existe um vértice com valor na função objetivo maior do que o valor do vértice da iteração corrente. Como a região viável é convexa e a função objetivo é linear, esse ótimo local é na verdade um ótimo global [10].

No exemplo do problema de PL (2.6)-(2.11), suponha que o algoritmo *simplex* inicia

no vértice $(0, 0)$, com valor 0 na função objetivo, que ocorre quando $x_1 = 0$ e $x_2 = 0$, em seguida vai para o vértice $(0, 2)$, com valor 4 na função objetivo, com $x_1 = 0$ e $x_2 = 2$, e por fim vai para o vértice $(1.5, 5)$, que é a solução ótima $x_1 = 1.5, x_2 = 5$, com valor 11.5 na função objetivo. Na Figura 2.6 cada iteração desse exemplo foi ilustrada com o vértice atual marcado com um ponto.

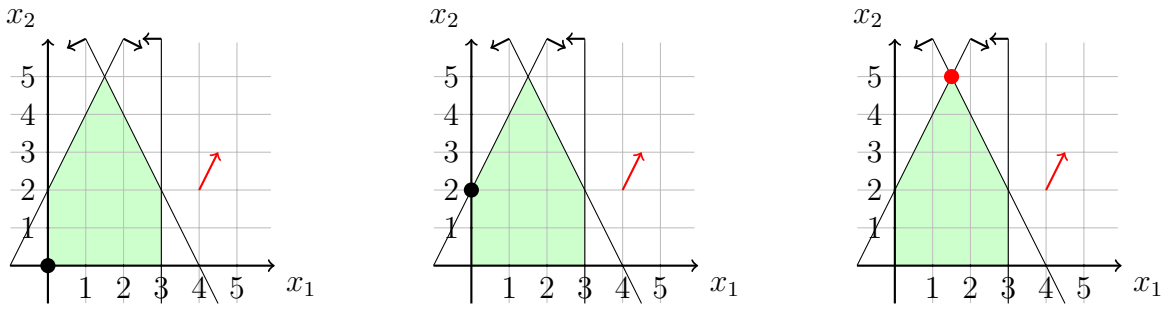


Figura 2.6: Iterações do algoritmo simplex. (a) Iteração no vértice $(0, 0)$, (b) iteração no vértice $(0, 2)$, (c) iteração final no vértice $(1.5, 5)$.

Apesar de ser intuitiva, a interpretação geométrica do algoritmo do *simplex* não é boa para ser computada. Por isso, a interpretação algébrica do *simplex* é importante para sua implementação. A seguir, descrevemos brevemente como o *simplex* funciona algebricamente.

Inicialmente é preciso descrever a **forma padrão** de um problema de PL. Essa forma é semelhante à forma canônica, porém todas as desigualdades são transformadas em igualdades. Para isso, m variáveis de folga s_i, para todo $1 \leq i \leq m$, são adicionadas no modelo. Essas variáveis têm coeficiente 0 na função objetivo e indicam o quanto o lado esquerdo das restrições está menor do que o lado direito. A forma padrão de um problema genérico de PL é a seguinte:

$$\max \sum_{j=1}^{n} c_j x_j \tag{2.21}$$

sujeito a:

$$\sum_{j=1}^{n} a_{ij} x_j + s_i = b_i, \qquad \forall i \in \{1, 2, \ldots m\} \tag{2.22}$$

$$x_j \geq 0, \qquad \forall j \in \{1, 2, \ldots, n\} \tag{2.23}$$

$$s_i \geq 0, \qquad \forall i \in \{1, 2, \ldots, m\} \tag{2.24}$$

O *simplex* particiona o conjunto de variáveis em dois conjuntos: **variáveis básicas** (B) e **variáveis não-básicas** (N). O tamanho de B é sempre m e de N é sempre $n - m$. Inicialmente as variáveis básicas serão as variáveis de folga. A função objetivo é reescrita como $\max z = \sum_{j=1}^{n} c_j x_j$. Em seguida a função objetivo e as variáveis básicas são escritas

em função das variáveis não-básicas:

$$z = \sum_{j \in N} c_j x_j \tag{2.25}$$

$$s_i = b_i - \sum_{j \in N} a_{ij} x_j, \qquad \forall i \in \{1, 2, \ldots, n\} \tag{2.26}$$

No *simplex*, as variáveis não-básicas sempre são fixadas em 0. Então, esse ponto inicial representa na projeção cartesiana a origem do hiperplano.

A cada iteração do *simplex* o coeficiente das variáveis não-básicas no lado direito da equação de z é verificado. Esse valor indica em quanto o valor da função objetivo seria alterado para cada unidade acrescida no valor daquela variável. Esse valor é chamado de **custo reduzido** da variável. O próximo passo é escolher uma das variáveis não-básicas com custo reduzido positivo para entrar na base. Essa variável é chamada de **variável de entrada**. Aumentar o valor de uma variável não-básica faz o valor de algumas das variáveis básicas diminuir, para manter as Equações (2.26) válidas. Para manter as restrições de não-negatividade (2.23) e (2.24) também válidas, a variável de entrada só aumenta o valor até o ponto em que a alguma variável básica vale 0. Essa variável básica que passa a ter valor 0 é chamada de **variável de saída**. Na Equação (2.26) em que a variável de saída estava do lado esquerdo, a variável de saída passa para o lado direito e a variável de entrada é isolada do lado esquerdo. Em seguida um pivoteamento é realizado para substituir a variável de saída pela variável de entrada nas equações das outras variáveis básicas e na função objetivo. Após esse processo a variável de entrada passa a ser uma variável básica e a de saída passa a ser uma variável não básica.

Se todas as variáveis não-básicas tiverem custo reduzido menor ou igual a 0, então a solução ótima foi encontrada e o algoritmo para.

Em seguida mostramos um exemplo do simplex sendo executado no problema de `PL` (2.6)-(2.11).

Assim fica o problema escrito em função das variáveis não-básicas:

$$z = x_1 + 2x_2 \tag{2.27}$$

$$s_1 = 2 + 2x_1 - x_2 \tag{2.28}$$

$$s_2 = 8 - 2x_1 - x_2 \tag{2.29}$$

$$s_3 = 3 - x_1 \tag{2.30}$$

Ambas as variáveis x_1 e x_2 têm custo reduzido positivo. Por ter custo reduzido maior, escolhemos a variável x_2 para ser a variável de entrada. A variável de saída é s_1. Após o pivoteamento, o problema fica:

$$z = 4 + 5x_1 - 2s_1 \tag{2.31}$$

$$x_2 = 2 + 2x_1 - s_1 \tag{2.32}$$

$$s_2 = 6 - 4x_1 + s_1 \tag{2.33}$$

$$s_3 = 3 - x_1 \tag{2.34}$$

Somente a variável x_1 tem custo reduzido positivo, então ela será a variável de entrada. A variável de saída será s_2. Após o pivoteamento, o problema resultante é:

$$z = \frac{23}{2} - \frac{3}{4}s_1 - \frac{5}{4}s_2 \tag{2.35}$$

$$x_2 = 5 - \frac{1}{2}s_1 - \frac{1}{2}s_2 \tag{2.36}$$

$$x_1 = \frac{3}{2} + \frac{1}{4}s_1 - \frac{1}{4}s_2 \tag{2.37}$$

$$s_3 = \frac{3}{2} - \frac{1}{4}s_1 + \frac{1}{4}s_2 \tag{2.38}$$

Como o custo reduzido de todas as variáveis não-básicas está negativo, a solução ótima foi encontrada. Substituindo os valores de s_1 e s_2 por 0, pois são variáveis não-básicas, é possível obter o valor da função objetivo e das variáveis nessa solução.

$$z = \frac{23}{2} = 11.5 \tag{2.39}$$

$$x_2 = 5 \tag{2.40}$$

$$x_1 = \frac{3}{2} = 1.5 \tag{2.41}$$

$$s_3 = \frac{3}{2} = 1.5 \tag{2.42}$$

2.3.2 Geração de Colunas

Quando um problema de PL tem muitas variáveis, pode ser inviável armazenar todo o modelo em memória e demanda muito tempo para gerar todas as variáveis, verificar o custo reduzido de cada uma e efetuar diversas outras operações. **Geração de Colunas** é um método para resolver problemas de PL com uma grande quantidade de variáveis.

Com essa técnica, o algoritmo resolvedor do PL, geralmente o *simplex*, inicia com apenas um subconjunto das variáveis. Ao finalizar a execução, ou seja, quando uma solução ótima do PL com apenas o subconjunto de variáveis for encontrado, um outro algoritmo verifica se existe alguma variável fora desse subconjunto que melhoraria a solução se fosse adicionada. Esse algoritmo deve ser capaz de encontrar alguma variável com custo reduzido positivo, no caso de um problema de maximização, ou determinar se nenhuma variável que satisfaça tal propriedade existe. Esse problema é denominado de *pricing*.

Uma maneira muito comum de resolver o problema de *pricing* é verificar no problema dual se existe alguma restrição violada. Assim como no primal algumas variáveis eram representadas implicitamente, no dual algumas restrições serão representadas implicitamente. Se alguma restrição é violada no dual, significa que essa restrição é necessária para resolver o problema dual, portanto, a variável equivalente é necessária para resolver o problema primal.

2.3.3 Programação Linear Inteira

Diversos problemas de PL exigem que suas variáveis assumam somente valores inteiros. Dizemos que esses são problemas de **Programação Linear Inteira** (PLI). Para problemas de PLI adicionamos as Restrições (2.43), que forçam as variáveis a pertencerem ao conjunto de números inteiros:

$$x_j \in \mathbb{Z}^+, \forall j \in \{1, 2, \ldots, n\} \tag{2.43}$$

Como os métodos de resolução de PL, inclusive o *simplex*, retornam vértices da região viável como solução ótima e os vértices da região viável podem não ser compostos por coordenadas inteiras, as soluções encontradas por esses métodos podem atribuir valores não inteiros para as variáveis. Portanto, esses métodos não resolvem problemas de PLI. Chamamos de relaxação linear o problema de PL que resulta da remoção das restrições de integralidade do problema de PLI.

No exemplo do PL (2.6)-(2.11), a solução ótima, $x_1 = 1.5$ e $x_2 = 5$, não é inteira, pois o valor de x_1 não pertence ao conjunto dos inteiros. Os valores de (x_1, x_2) que formam soluções factíveis para o problema PLI são $(0, 0)$, $(1, 0)$, $(2, 0)$, $(3, 0)$, $(0, 1)$, $(1, 1)$, $(2, 1)$, $(3, 1)$, $(0, 2)$, $(1, 2)$, $(2, 2)$, $(3, 2)$, $(1, 3)$, $(2, 3)$, $(1, 4)$ e $(2, 4)$. A Figura 2.7 ilustra no plano cartesiano as soluções factíveis para o problema PLI, onde as soluções estão marcadas por pontos.

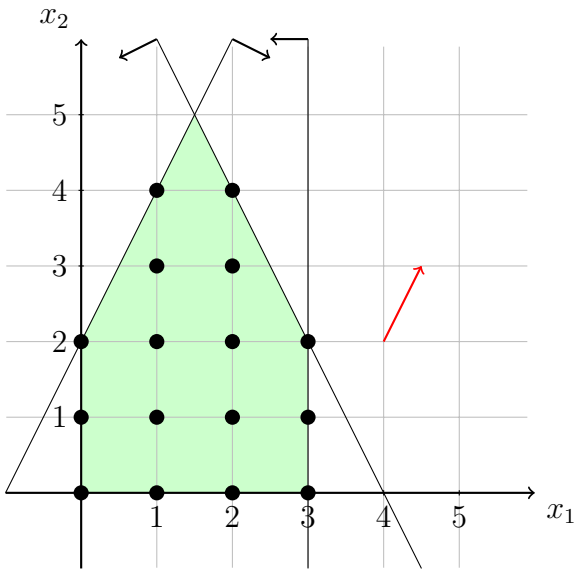


Figura 2.7: Projeção de um problema de programação linear inteira no plano cartesiano, exibindo restrições, soluções factíveis e direção de crescimento da função objetivo.

Como todas as soluções factíveis do PLI são factíveis para sua relaxação linear, o valor na função objetivo da solução ótima da relaxação linear nunca é pior do que o da solução ótima do PLI, ou seja, a solução da relaxação linear é um limitante dual para o PLI.

O método mais utilizado para resolver problemas de PLI é o *Branch-and-Bound* (B&B). O algoritmo de B&B usa o paradigma de divisão e conquista para dividir o problema em diversos subproblemas mais fáceis, resolver os subproblemas e usar as soluções desses para construir uma solução para o problema original [26].

No caso dos problemas de PLI, a divisão em subproblemas consiste em particionar o conjunto de solução factíveis. Também é desejável que a solução ótima da relaxação linear do problema original seja excluída na relaxação linear dos subproblemas.

Para isso, considere x^* como uma solução ótima da relaxação linear do problema de PLI original. Se os valores de todas as variáveis em x^* forem inteiros, então a solução x^* é uma solução factível para o problema PLI e o problema está resolvido. Caso contrário, deve ser escolhida alguma variável x_j cujo valor de $x_j^* \notin \mathbb{Z}$ para efetuar uma **ramificação** (*branch*). A ramificação na variável x_j consiste na criação de dois subproblemas, que são cópias do problema original com adição de uma restrição. No primeiro subproblema é adicionada a restrição:

$$x_j \leq \lfloor x_j^* \rfloor, \tag{2.44}$$

e no segundo, a restrição:

$$x_j \geq \lceil x_j^* \rceil. \tag{2.45}$$

Dessa forma, a solução x^* não é factível em nenhum subproblema, pois a variável x_j não pode valer x_j^*, e o conjunto de soluções factíveis foi particionado nos dois subproblemas. Por fim, a solução do problema original é a melhor solução dentre os dois subproblemas, já que nenhuma solução factível do PLI foi removida.

No exemplo do problema de PL (2.6)-(2.11), só existe uma variável na solução ótima $x^* = (1.5, 5)$ que não é inteira. Essa variável é $x_1 = 1.5$. Portanto, no primeiro subproblema a restrição a ser adicionada é

$$x_1 \leq 1, \tag{2.46}$$

e no segundo é

$$x_1 \geq 2. \tag{2.47}$$

No primeiro subproblema a solução ótima da relaxação linear é $x_1 = 1$ e $x_2 = 4$, com valor na função objetivo 9. No segundo subproblema a solução ótima da relaxação linear é $x_1 = 2$ e $x_2 = 4$, com valor na função objetivo 10. Como ambas são solução factíveis para o PLI, a solução ótima para o problema original é a melhor dentre as duas, ou seja, $x_1 = 2$ e $x_2 = 4$. A Figura 2.8 mostra a projeção no plano cartesiano dos dois subproblemas. A restrição em azul e a região em azul são respectivas ao primeiro subproblema e a restrição em vermelho e região em vermelho são respectivas ao segundo subproblema.

Uma maneira típica de representar essa abordagem de divisão e conquista é por meio de uma árvore de enumeração [26]. Na árvore de enumeração, o nó raiz conterá o problema de PLI original e os outros nós conterão subproblemas resultantes da ramificação dos seus antecessores.

Essa árvore de enumeração pode ficar muito grande, pois pode ter uma quantidade exponencial de nós, em função da quantidade de variáveis do problema de PLI original. Portanto, é usada uma enumeração implícita para gerenciar essa árvore.

O algoritmo de B&B inicia só com o nó raiz da árvore, que contém o problema original. Os outros nós serão adicionados através da ramificação de nós já existentes. Porém, existem condições que permitem a **poda** de um nó da árvore, isto é, não existe a necessidade de ramificar esse nó.

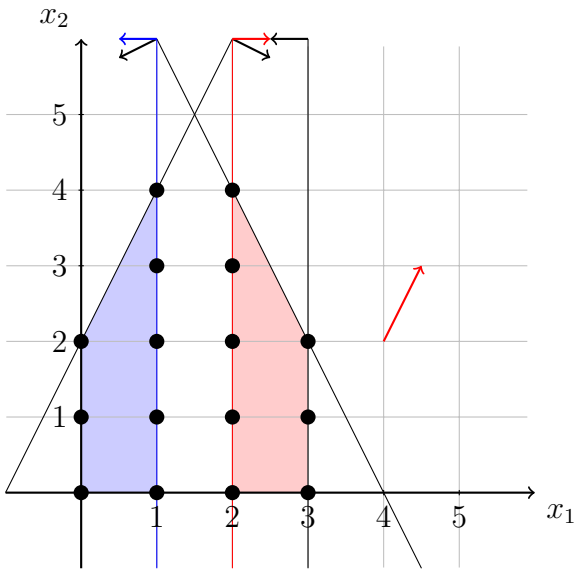


Figura 2.8: Projeção no plano cartesiano dos dois subproblemas resultantes da ramificação na variável x_1.

A primeira é a **poda por otimalidade** e ocorre quando a solução do PL é inteira, ou seja, a solução da relaxação linear é viável para o problema PLI e esse subproblema está resolvido.

A segunda é a **poda por infactibilidade** e ocorre quando a relaxação linear do subproblema é infactível, pois isso implica que o PLI também é infactível. Já que adicionar restrição só pode manter ou diminuir a quantidade de soluções factíveis, todos os nós descendentes desse nó também serão infactíveis.

A terceira é a **poda por limitante** e ocorre quando o limitante dual do nó, obtido resolvendo a relaxação linear, não é melhor do que algum limitante primal já conhecido. Em um problema de maximização, isso acontece quando o valor na função objetivo da solução ótima da relaxação linear é menor ou igual ao de alguma solução já conhecida. Como adicionar restrições reduz o conjunto de soluções factíveis, esse nó e seus descendentes não podem gerar uma solução melhor do que a que já é conhecida.

A árvore de enumeração do B&B para o problema PL (2.6)-(2.11) está ilustrada na Figura 2.9: Note que os nós 1 e 2 não criam mais ramificações, já que são podados por

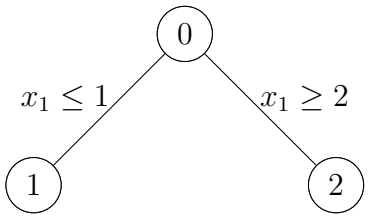


Figura 2.9: Exemplo da árvore de enumeração do B&B para o problema PL (2.6)-(2.11).

otimalidade. Porém, supondo que no nó 1 o valor de x_2 seja um valor não inteiro x_2^*, a árvore de enumeração cresce como na Figura 2.10.

Em diversos modelos PLI, inclusive os apresentados neste trabalho, as varáveis só podem assumir os valores 0 ou 1. Nesse caso, as restrições adicionadas nas ramificações

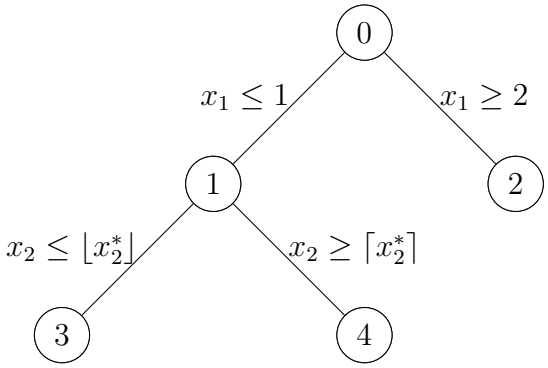


Figura 2.10: Exemplo da árvore de enumeração do `B&B`.

fixam o valor das variáveis. Por exemplo, se a ramificação for feita em uma variável x_j, em uma das subárvores será válido $x_j = 0$ e na outra $x_j = 1$.

Capítulo 3

Máxima Partição em Circuitos

Neste capítulo apresentamos o problema da Máxima Partição em Circuitos (*Maximum Eulerian Cycle Decomposition* - `MAX-ECD`). Na Seção 3.1, descrevemos o problema. Na Seção 3.2, detalhamos as abordagens que utilizamos para resolver o problema. Na Seção 3.3, descrevemos os experimentos realizados e discutimos os resultados obtidos. Na Seção 3.4, apresentamos as conclusões.

3.1 Introdução

O `MAX-ECD` é definido como: dado um grafo euleriano $G = (V, E)$, encontrar a partição em circuitos de maior cardinalidade, ou seja, particionar E no maior número possível de conjuntos que induzem circuitos em G. Caprara [5] mostrou que o `MAX-ECD` é um problema da classe NP-difícil.

A Figura 3.1 ilustra dois exemplos de Partição em Circuitos. A primeira partição tem cardinalidade 2, onde o primeiro circuito é composto pelas arestas azuis e o segundo pelas arestas vermelhas, e a segunda tem cardinalidade 4, e é composta pelos circuitos com as arestas vermelhas, azuis, verdes e magentas.

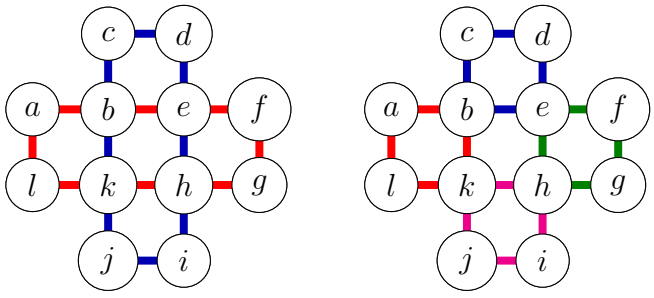
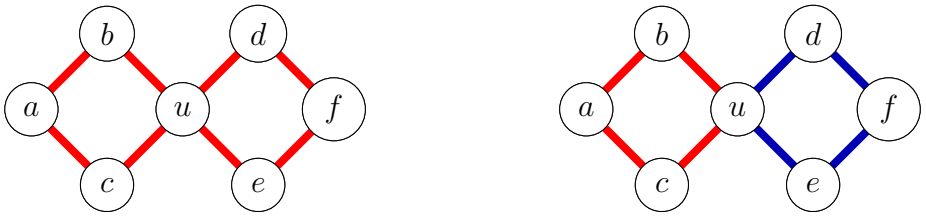


Figura 3.1: Exemplos de partições em circuitos de um grafo.

Note que, por se tratar de um problema de partição, todas as arestas precisam estar em algum circuito e não podem estar em mais de um circuito. Note também que, como a entrada é sempre um grafo euleriano, existe sempre uma solução trivial de cardinalidade 1: colocar todas as arestas do grafo em um mesmo circuito, formando um circuito euleriano.

Qualquer solução do `MAX-ECD` que possua um circuito que passe mais de uma vez pelo mesmo vértice u pode ser melhorada substituindo-se esse circuito por ciclos que passam

apenas uma vez por u.Por exemplo, na Figura 3.2, em (a), as arestas vermelhas formam um circuito euleriano, ou seja, uma solução para o MAX-ECD de cardinalidade 1, já em (b), a solução tem cardinalidade 2, um ciclo composto pelas arestas vermelhas e outro pelas arestas azuis.



(a) (b)

Figura 3.2: Exemplo de melhora na solução do MAX-ECD substituindo um circuito com repetição de vértices em (a) por dois ciclos em (b).

Lema 3.1.1. *Existe um limitante dual desse problema válido para qualquer grafo. Esse limitante é* $\lfloor \frac{|E|}{3} \rfloor$.

Demonstração. Chamaremos de triângulo os ciclos de comprimento 3. Esses são os menores ciclos possíveis, já que não existem ciclos de comprimento 1 ou 2 em grafos simples. Suponha, para efeitos de contradição que, existe uma partição $\mathcal{H}$, em que $|\mathcal{H}| > \lfloor \frac{|E|}{3} \rfloor$. Note que, caso $\frac{|E|}{3} \notin \mathbb{Z}$, temos que $|\mathcal{H}| \geq \frac{|E|}{3} + 1$, logo $|\mathcal{H}| > \frac{|E|}{3}$. Portanto, temos que $|\mathcal{H}| > \frac{|E|}{3}$. Logo, o comprimento médio dos ciclos $\frac{|E|}{|\mathcal{H}|} < 3$. Como os ciclos têm comprimento inteiro, temos pelo menos um ciclo com comprimento 2 ou menor, o que é uma contradição em grafos simples. □

3.2 Metodologia

Utilizamos abordagens exatas e heurísticas para resolver o problema MAX-ECD. Na Seção 3.2.1 descrevemos um algoritmo exato para o problema. Nas seções 3.2.2 e 3.2.3 descrevemos algoritmos heurísticos para o problema.

3.2.1 Modelo PLI

Caprara, Panconesi e Rizzi [7] apresentaram um modelo de Programação Linear Inteira (PLI) para o MAX-ECD. As notações a seguir serão utilizadas para descrever esse modelo. Seja $\mathcal{C}$ o conjunto de todos os ciclos de G. Para cada ciclo $C \in \mathcal{C}$, x_C é uma variável binária que vale 1 se C faz parte da solução e 0 caso contrário. O conjunto de todos os ciclos que contém uma aresta e é chamado $\mathcal{C}_e$. Segue o modelo:

$$\max \sum_{C \in \mathcal{C}} x_C \tag{3.1}$$

sujeito a:

$$\sum_{C \in \mathcal{C}_e} x_C \leq 1, \qquad \forall e \in E \tag{3.2}$$

$$x_C \in \{0, 1\}, \qquad \forall C \in \mathcal{C} \tag{3.3}$$

As Restrições (3.2) garantem que os ciclos da solução não possuem arestas em comum e as Restrições (3.3) garantem que as variáveis x_C são binárias.

Note que um conjunto $\mathcal{H}$ de ciclos que não cobre todas as arestas pode formar uma solução factível desse modelo. Porém, essa solução não seria ótima, já que a Função Objetivo (3.1) é de maximização e neste caso existiria pelo menos um ciclo em G que não tem nenhuma aresta em comum com os ciclos de $\mathcal{H}$.

Para provar isso, suponha, para efeitos de contradição, que existe um conjunto $\mathcal{H}$ de ciclos que não cobrem todas as arestas de G e que constituem uma solução ótima do modelo PLI (3.1)-(3.3). Seja G' um subgrafo de G obtido após a remoção de todas as arestas dos ciclos de $\mathcal{H}$ e de todos os vértices que ficaram com grau 0 após remover essas arestas. Se todos os vértices de G' tiverem grau pelo menos 2, então existe um ciclo C' em G'. Portanto o conjunto $\mathcal{H} \cup \{C'\}$ constitui uma solução melhor do que a ótima. Não é possível existir um vértice de grau 1 em G', pois qualquer vértice u tinha grau par em G, por ser um grafo euleriano, e teve seu grau subtraído por um valor par (redução de duas unidades para cada ciclo em $\mathcal{H}$ que contém u), resultando em um grau par em G'. Portanto, se existir pelo menos uma aresta em G' (aresta não coberta em G), então existe um ciclo em G', que é uma contradição com a premissa de que o conjunto $\mathcal{H}$ é uma solução ótima.

Para permitir apenas partições no modelo seria necessário substituir as Restrições (3.2) pelas Restrições (3.4):

$$\sum_{C \in \mathcal{C}_e} x_C = 1, \qquad \forall e \in E. \tag{3.4}$$

Porém, no modelo com as Restrições (3.2) é mais fácil encontrar limitantes primais, já que qualquer conjunto de ciclos disjuntos forma uma solução factível para esse modelo. Encontrar limitantes primais pode ser vantajoso para o algoritmo de `B&B`, pois a poda por limitante necessita de um limitante primal.

A relaxação linear do modelo PLI (3.1)-(3.3) é obtida substituindo as Restrições (3.3) pelas Restrições (3.5):

$$x_C \geq 0, \qquad \forall C \in \mathcal{C}. \tag{3.5}$$

Como o tamanho de $\mathcal{C}$ cresce exponencialmente em função do tamanho de E, não é viável criar todas as variáveis do modelo para instâncias grandes. Portanto, recorremos às técnicas de geração de colunas e *Branch and Price* (`B&P`). No início do algoritmo `B&B`, o modelo é alimentado com um pequeno subconjunto de variáveis x_C e mais variáveis são adicionadas conforme necessário até obter uma solução ótima. A decisão se uma variável é trazida para o modelo ou não é feita resolvendo o subproblema de *pricing*. Para isso,

considere o dual da relaxação linear do modelo (3.1)-(3.3):

$$\min \sum_{e \in E} y_e \tag{3.6}$$

sujeito a:

$$\sum_{e \in C} y_e \geq 1, \qquad \forall C \in \mathcal{C} \tag{3.7}$$

$$y_e \geq 0, \qquad \forall e \in E \tag{3.8}$$

O subproblema de *pricing* consiste em decidir se existe um ciclo C cuja restrição associada em (3.7) é violada pelas variáveis duais de uma solução y^*. Como o tamanho de $\mathcal{C}$ é exponencial, é impraticável verificar todas essas restrições. Em vez disso, usamos um algoritmo combinatório descrito abaixo para resolver o subproblema de *pricing*.

Para cada ciclo $C \in \mathcal{C}$, a Restrição (3.7) exige que a soma das variáveis y_e^*, com $e \in C$, seja maior ou igual a 1. Portanto, dado um ciclo C, se $\sum_{e \in C} y_e^* < 1$, a restrição associada a C está violada. Então, para o algoritmo de separação, criamos um grafo ponderado G' com os mesmos vértices e arestas de G, em que cada aresta $e \in E$ tem peso y_e^*. Uma restrição do modelo dual está violada se e somente se existir um ciclo em G' cujo seu peso, isto é, a soma do peso de suas arestas, é menor do que 1. Logo, o algoritmo precisa encontrar o ciclo de peso mínimo C^* em G' e, se seu peso for menor do que 1, adicionar a variável x_{C^*} no modelo primal, pois a restrição associada a C^* está violada. Se C^* tem peso maior ou igual a 1, então as Restrições (3.7) estão satisfeitas para todos os ciclos e nenhuma nova variável precisa ser adicionada, ou seja, a solução ótima da relaxação pode ser obtida usando somente as variáveis já criadas.

Assim, o subproblema de *pricing* é encontrar um ciclo de peso mínimo C^* no grafo G', sem considerar os ciclos relacionados às variáveis x_C que foram fixadas em 0 pelo algoritmo `B&B`, uma vez que queremos encontrar novos ciclos a serem considerados no modelo primal. Quando nenhuma variável x_C está fixada em 0, podemos usar o algoritmo `Ciclo-Mínimo` descrito no Algoritmo 3. Para cada aresta (u, v), o algoritmo encontra um caminho de peso mínimo P de u para v, que não contém a aresta (u, v), e cria o ciclo $P \cup \{(u, v)\}$. Observe que este é o ciclo de peso mínimo que contém a aresta (u, v). Em seguida, o algoritmo retorna o ciclo gerado com peso mínimo.

O algoritmo de *pricing* como descrito por Caprara, Panconesi e Rizzi [7] não resolve o problema para nós da árvore de `B&B` que não são a raiz. Então, propomos alterações nesse algoritmo para resolver o *pricing* em qualquer nó da árvore.

Quando as variáveis $\{x_{C_1}, x_{C_2}, \ldots, x_{C_k}\}$ estão fixadas em 0, precisamos proibir esses ciclos no subproblema de *pricing*. Para isso, criamos cópias $G_{z_1,z_2,\ldots,z_k}$ de G', de modo que $1 \leq z_i \leq |C_i|$, para cada $1 \leq i \leq k$. Em cada cópia $G_{z_1,z_2,\ldots,z_k}$, proibimos os ciclos C_1 a C_k atualizando o peso da z_i-ésima aresta de C_i para infinito. O algoritmo `Ciclo-Mínimo` é executado para cada cópia alterada de G' e o ciclo de peso mínimo encontrado em todas essas execuções é a solução do problema de *pricing*.

Esse algoritmo de *pricing* torna viável a ramificação descrita na Seção 2.3.3, que divide o problema `PLI` em dois subproblemas fixando uma das variáveis em 0 em um dos

Algoritmo 3 Algoritmo para encontrar ciclo de peso mínimo.

```
1: função Ciclo-Mínimo(G' = (V, E), pesos)
2:     peso_mínimo ← ∞
3:     ciclo_mínimo ← ∅
4:     para cada e = (u, v) ∈ E faça
5:         pesos' ← pesos
6:         pesos'[e] ← ∞
7:         (w, P) ← Dijkstra(u, v, G', pesos')
8:         w ← w + pesos[e]
9:         P ← P ∪ {e}
10:        se w < peso_mínimo então
11:            peso_mínimo ← w
12:            ciclo_mínimo ← P
13:        fim se
14:    fim para
15:    devolve (peso_mínimo, ciclo_mínimo)
16: fim função
```

subproblemas e em 1 no outro subproblema. Quando uma variável x_C está fixada em 1, o problema de *pricing* se torna mais fácil porque, para cada aresta $e \in C$, a variável do problema dual y_e terá valor 1 na solução do problema dual e nenhum ciclo C' que contém e terá sua restrição correspondente no problema dual $\sum_{e' \in C'} y_{e'} \geq 1$ violada, então essas arestas podem ser desconsideradas na busca do ciclo mínimo. Porém, no subproblema em que a variável x_C é fixada em 0, o problema de *pricing* se torna mais difícil, pois é necessário executar o algoritmo de `Ciclo-Mínimo` múltiplas vezes (uma para cada aresta de C). Se uma outra variável também é fixada em zero, o número de vezes que o algoritmo de `Ciclo-Mínimo` precisa ser executado é multiplicado novamente. Isso faz que o algoritmo de *pricing* seja ineficiente, principalmente após fixar muitas variáveis em 0, ou seja, quando muitas ramificações são feitas.

3.2.2 Heurística Gulosa

Descrevemos a seguir o algoritmo guloso desenvolvido para encontrar soluções válidas para o `MAX-ECD`.

O algoritmo tem como entrada o grafo $G = (V, E)$ e retorna uma partição em ciclos desse grafo. Inicialmente, um conjunto $\mathcal{H}$ vazio de ciclos é criado. A cada iteração do algoritmo um novo ciclo é adicionado em $\mathcal{H}$ e o grafo G é alterado para a próxima iteração. Por fim, quando não existir mais arestas no grafo, o conjunto $\mathcal{H}$ é retornado pelo algoritmo.

Uma iteração funciona da seguinte forma: um vértice v com grau maior do que 0 é escolhido; em seguida, usando um algoritmo de busca em largura, é encontrado um ciclo C, de comprimento mínimo, que contém v. Esse ciclo C é adicionado no conjunto $\mathcal{H}$ e as arestas de C são removidas de E.

Como todas as arestas de um ciclos são removidas do grafo antes da próxima iteração, todos os ciclos do conjunto $\mathcal{H}$ são disjuntos. Note que, após remover as arestas de um ciclo

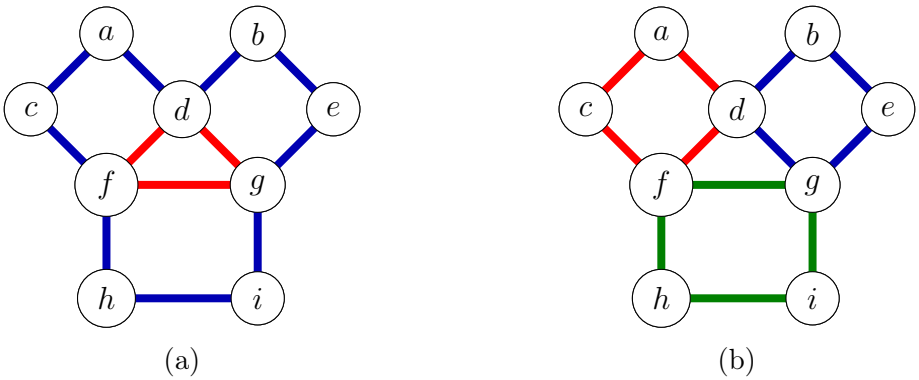
do grafo, o grau de todos os vértices continua par. Portanto, qualquer vértice com grau maior do que 0 está contido em algum ciclo, pois está em uma componente conexa com grau mínimo maior ou igual a 2. Logo, em qualquer iteração sempre haverá algum ciclo para ser adicionado na solução e removido do grafo. Então, em algum momento, todas as arestas serão removidas do grafo e estarão em algum ciclo da solução. Isso significa que o algoritmo sempre retorna uma partição em ciclos. Essa heurística tem complexidade de tempo $O(|E|^2)$.

O Algoritmo 4 descreve o essa heurística.

Algoritmo 4 Heurística gulosa para o `MAX-ECD`

```
1:  função HEURÍSTICA-GULOSA(G = (V, E))
2:      H ← ∅
3:      enquanto |E| > 0 faça
4:          escolha um vértice v
5:          C ← Ciclo-Mínimo(G, v)
6:          H ← H ∪ {C}
7:          E ← E \ {C}
8:      fim enquanto
9:      devolve H
10: fim função
```

Note que a forma como o vértice v é escolhido na Linha 4 do Algoritmo 4 pode influenciar no resultado do algoritmo. A Figura 3.3 mostra duas soluções diferentes do algoritmo para o mesmo grafo, que são diferentes porque o vértice escolhido foi diferente. Caso o vértice escolhido na primeira iteração da heurística seja o d, o ciclo de comprimento mínimo que contém d seria o que contém as arestas (d, g), (g, f) e (f, d). Na segunda iteração só restaria um ciclo no grafo com todas as arestas restantes. Essa cenário está ilustrado na Figura 3.3(a). Caso o vértice escolhido na primeira iteração da heurística seja a, o ciclo de comprimento mínimo que contém a seria o que contém as arestas (a, d), (d, f), (f, c) e (c, a). Só restam dois ciclos no grafo para a segunda e a terceira iteração. Esse cenário está ilustrado na Figura 3.3(b). Observe que no segundo caso o resultado tem um ciclo a mais.



(a) (b)

Figura 3.3: Soluções distintas da heurística gulosa para o `MAX-ECD`.

Escolhendo aleatoriamente o vértice da Linha 4, adicionamos um passo não determinístico no algoritmo, o que significa que o resultado pode ser diferente para diferentes

execuções. Com base nisso, uma nova heurística foi desenvolvida, que consiste em executar diversas vezes a heurística apresentada e devolver o melhor resultado obtido dentre essas execuções. Essa técnica de executar a heurística várias vezes também será usada na heurística descrita na próxima subseção.

3.2.3 Heurística PLI

Desenvolvemos uma heurística para o `MAX-ECD` que utiliza `PLI` para obter bons limitantes primais. Essa heurística consiste em executar o modelo `PLI` descrito na Subseção 3.2.1 utilizando somente as variáveis associadas a um subconjunto de ciclos $\mathcal{C}' \subset \mathcal{C}$.

Como usamos apenas parte do conjunto de ciclos, podemos escolher o subconjunto $\mathcal{C}'$ de forma que seja possível manter em memória todas as variáveis simultaneamente. Assim não é mais necessário resolver o problema de *pricing*, pois nenhuma nova variável é adicionada.

Porém, o algoritmo pode não encontrar mais uma solução ótima para o problema, já que não contém todas as variáveis do problema original. Por exemplo, se, para um determinado conjunto $\mathcal{C}'$ escolhido, todas as soluções ótimas de uma instância precisarem de algum ciclo do subconjunto $\mathcal{C} \setminus \mathcal{C}'$, então a heurística não é capaz de encontrar a solução ótima para essa instância.

Usamos a heurística descrita na Subseção 3.2.2 para escolher os ciclos do conjunto $\mathcal{C}'$. Primeiro executamos a heurística gulosa diversas vezes, com escolha aleatória do vértice na Linha 4 do Algoritmo 4. Como resultado obtemos diversas partições provavelmente distintas. O conjunto $\mathcal{C}'$ é formado pela união dessas partições. Note que a solução ótima da heurística `PLI` tem cardinalidade maior ou igual à melhor solução dentre as execuções da heurística gulosa, já que os ciclos dessa solução estão contidos em $\mathcal{C}'$.

Essa heurística pode retornar uma solução que não é uma partição, ou seja, nem todas as arestas estão em algum ciclo da solução. Isso ocorre quando uma solução que não cobre todas as arestas tem uma cardinalidade maior ou igual do que as partições possíveis com o conjunto $\mathcal{C}'$. Por exemplo, considere o grafo da Figura 3.4. Suponha que $\mathcal{C}'$ é formado por 5 ciclos, que são os ciclos com arestas vermelhas, azuis, laranjas, verdes e magentas. O ciclo com arestas pretas não está em $\mathcal{C}'$. A única partição possível é usar os ciclos azul e vermelho, formando uma solução com cardinalidade 2. Porém, existe uma solução com cardinalidade 3 que não cobre todas as arestas do grafo, usando os ciclos laranja, verde e magenta.

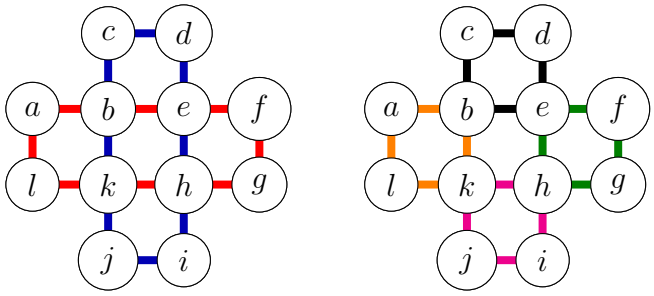


Figura 3.4: Ciclos do conjunto $\mathcal{C}'$ da heurística `PLI`.

Porém, remover do grafo as arestas dos ciclos de uma solução resulta em um grafo euleriano ou em um grafo desconexo cujo as componentes conexas são subgrafos eulerianos. Portanto é possível aplicar um pós-processamento para melhorar a solução encontrada, caso ela não seja uma partição. Como o objetivo da heurística é encontrar os melhores limitantes primais possíveis, executar essa heurística permitindo soluções que não são partições e executar o pós-processamento em seguida é melhor do que permitir somente soluções que são partições.

3.3 Experimentos Computacionais

Nesta seção descrevemos os resultados dos experimentos computacionais executados, assim como o ambiente em que os experimentos foram realizados e como foram criadas as instâncias.

3.3.1 Ambiente Computacional

Todos os experimentos foram executados em uma máquina com processador Intel Core i7-8700T CPU com 12 núcleos de 2.40 GHz, memória RAM de 8 GB e sistema operacional Ubuntu 18.04.3. O tempo limite de execução em todos os experimentos foi de 1800 segundos.

O resolvedor de `PL` usado foi o Gurobi 8.1.1 [15]. Usamos dois resolvedores de `PLI`. Quando não foi necessário aplicar as técnicas de geração de colunas e `B&P` usamos novamente o Gurobi 8.1.1. Para as implementações do `PLI` com geração de colunas e `B&P` usamos o resolvedor SCIP 6.0.2 [1]. O resolvedor SCIP foi usado porque fornece ferramentas para implementar geração de colunas e `B&P` de forma eficiente.

3.3.2 Geração de Instâncias

Os experimentos foram realizados em diversos conjuntos de instâncias geradas aleatoriamente, cada um agrupando grafos com mesmo número de vértices e arestas. Para gerar essas instâncias usamos uma variação do algoritmo apresentado por Hakimi [16] descrito a seguir.

O algoritmo de Hakimi recebe uma sequência S de n inteiros positivos e devolve um grafo G com n vértices, em que o grau de cada vértice de G é igual a um número em S, ou determina que não existe tal grafo. Esse algoritmo inicia o grafo G com n vértices e 0 arestas. Em seguida, a cada iteração, um número aleatório i é escolhido satisfazendo $1 \leq i \leq n$ e $S_i \geq 0$. Então, ele adiciona em G arestas entre o vértice i e os S_i vértices $\{j_1, j_2, \ldots, j_{S_i}\}$ com maiores valores em S (em caso de empate, o vértice que será vizinho de i é escolhido aleatoriamente entre os empatados). Por fim, ele atribui valor 0 para S_i e subtrai 1 dos valores em $\{S_{j_1}, S_{j_2}, \ldots, S_{j_{S_i}}\}$. Esse processo é repetido até que S esteja completamente zerado. Caso, em alguma iteração, S_i seja maior do que a quantidade de elementos não zero em S, então não é possível montar um grafo com essa sequência de números.

Como desejamos apenas grafos eulerianos, a sequência de números deve conter apenas números pares. Além disso, desejamos que todos os vértices do grafo sejam relevantes para o MAX-ECD, então nenhum número da sequência deve ser 0. Desenvolvemos um algoritmo que, dado o número de vértices e arestas do grafo desejado, retorna uma sequência que satisfaz as propriedades desejadas. O algoritmo inicia com uma lista de tamanho n com valor 2 em todas as posições da lista. Em seguida, $m - n$ iterações são realizadas e em cada iteração algum elemento aleatório da lista terá seu valor acrescido em 2. Por fim, a lista é retornada como a sequência de graus. O Algoritmo 5 mostra o pseudocódigo do procedimento descrito anteriormente.

Algoritmo 5 Sequência de Graus

```
1: função SEQUÊNCIA-DE-GRAUS(n, m)
2:     S ← uma lista de n elementos com valor 2 em todas as posições.
3:     m' ← m − n
4:     enquanto m' > 0 faça
5:         i ← Aleatório(1, n)
6:         S_i ← S_i + 2
7:         m' ← m' − 1
8:     fim enquanto
9:     devolve S
10: fim função
```

O algoritmo de Hakimi pode devolver um grafo desconexo. Em um grafo desconexo o MAX-ECD pode ser resolvido em cada componente conexa e depois suas soluções podem ser unidas para formar uma solução para o problema original. Porém, desejamos que as instâncias sejam difíceis, então executamos um pós-processamento para tornar os grafos conexos. Esse pós-processamento consiste em escolher duas arestas (u, v) e (x, y) que estejam em componentes conexas diferente e substituí-las pelas arestas (u, x) e (v, y), diminuindo o número de componentes conexas em 1. Esse processo é repetido até o grafo se tornar conexo.

Para cada combinação dos parâmetros $n \in \{10, 20, \ldots, 90, 100\}$ e $d \in \{0.1, 0.2, 0.3, 0.4, 0.5\}$, criamos um conjunto contendo 20 grafos com n vértices e $m = \lfloor d\binom{n}{2} \rfloor$ arestas. Como um grafo precisa de pelo menos n arestas para ser conexo, caso $\lfloor d\binom{n}{2} \rfloor < n$, é atribuído o valor n para m.

3.3.3 Resultados dos Experimentos

Os primeiros experimentos feitos foram usando o modelo PLI exato, isto é, utilizando geração de colunas e B&P para obter soluções ótimas do problema. Nesses experimentos, nenhuma variável x_C está no conjunto inicial de variáveis, porém uma base do algoritmo *simplex* pode ser formada usando as variáveis de folga. Esse método mostrou bons resultados para instâncias pequenas, porém, ao aumentar o tamanho das instâncias o algoritmo parou de conseguir encontrar a solução ótima dentro do tempo limite. Na Tabela 3.1 mostramos a proporção de instâncias em que a solução ótima foi encontrada por esse método, variando a quantidade de vértices (n) nas linhas e a densidade (d) nas colunas.

n \ d	10%	20%	30%	40%	50%
10	100%	100%	100%	100%	100%
20	100%	100%	100%	85%	100%
30	100%	95%	80%	80%	60%
40	95%	65%	50%	50%	35%
50	80%	20%	20%	10%	0%
60	40%	5%	0%	0%	0%
70	15%	0%	0%	0%	0%
80	0%	0%	0%	0%	0%
90	0%	0%	0%	0%	0%
100	0%	0%	0%	0%	0%

Tabela 3.1: Porcentagem de instâncias resolvidas usando `PLI` com `B&P`.

No total, 37.7% das instâncias foram resolvidas com resultado ótimo. Para grafos com $n = 10$, todas as instâncias foram resolvidas e para grafos com $n = 20$, só três instâncias não foram resolvidas. Porém, para grafos com $n \geq 80$, nenhuma instância foi resolvida, com $n = 70$ só três instâncias foram resolvidas (todas com densidade 10) e com $n = 60$ foram resolvidas nove instâncias (uma com densidade 20 e as demais com densidade 10). O aumento da densidade também dificulta o problema, por exemplo, com densidade pelo menos 30% nenhuma instância com mais de 50 vértices foi resolvida.

Além de não encontrar a solução ótima para o problema, em 88 instâncias com $n \geq 80$, nem mesmo a relaxação linear do nó raiz foi resolvida dentro do tempo limite. Isso significa que nenhum limitante dual foi encontrado, portanto, nenhum primal também. Nessas instâncias, em média, aproximadamente 74% do tempo de execução foi gasto executando o algoritmo de *pricing*. Esse tempo pode ser reduzido iniciando o modelo com um conjunto não vazio de variáveis, visando reduzir o número de chamadas ao algoritmo de *pricing*. Além disso, houve instâncias em que, mesmo encontrando limitantes duais, nenhum limitante primal foi encontrado, porque é necessária uma solução com valores inteiros em todas as variáveis para ser uma solução factível do problema de `PLI`.

Utilizamos também a heurística gulosa para escolher quais ciclos farão parte do conjunto inicial de variáveis. Para cada instância, executamos a heurística gulosa 100 vezes (alterando a semente de aleatoriedade em cada execução) e escolhemos como conjunto inicial de variáveis a união das 100 partições encontradas pela heurística. Após adicionar essas variáveis no início da execução, a relaxação linear do nó raiz foi resolvida dentro do tempo limite em todas as instâncias.

Também usamos esse mesmo conjunto de variáveis para executar a heurística `PLI`. O pós-processamento usado para completar as soluções da heurística `PLI` foi o modelo `PLI` com `B&P`, já que restam poucas arestas no grafo e o resolvedor consegue concluir a execução dentro do tempo limite (todas as instâncias foram resolvidas em menos que um décimo de segundo). Nos gráficos da Figura 3.5, comparamos os limitantes primais encontrados pela heurística gulosa com aqueles encontrados pela heurística `PLI`.

Como a heurística `PLI` usa os mesmo ciclos da gulosa, a solução ótima da heurística não poderia ser pior do que a solução da heurística gulosa. Porém, o resultado poderia não melhorar ou até mesmo piorar, caso o tempo limite fosse excedido antes de encontrar

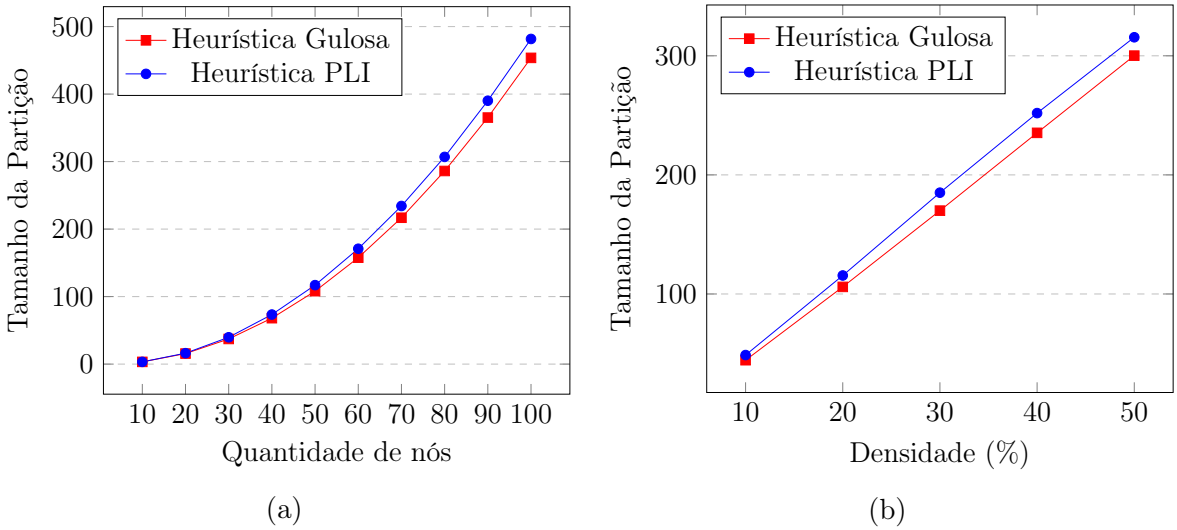


Figura 3.5: Crescimento dos limitantes primais das heurísticas em relação à (a) quantidade de nós e à (b) densidade da instância.

a solução ótima. Em nenhum instância o resultado piorou e a quantidade de ciclos nas soluções melhoraram 7%, em média.

Usando os dados obtidos pelas heurísticas, executamos novamente o modelo `PLI` exato, iniciando com as variáveis retornadas pela heurística gulosa (para diminuir o tempo gasto no *pricing*) e com os limitante primais retornados pela heurística `PLI` (visando diminuir o tamanho da árvore de `B&B`). Em nenhuma instância o `PLI` com `B&P` encontrou um limitante primal melhor do que o encontrado pela heurística `PLI`, ou seja, os limitantes primais da heurística `PLI` são iguais aos limitantes primais do `PLI` com `B&P` usando as variáveis da heurística gulosa como variáveis iniciais e o limitante primal da heurística `PLI`.

As Tabelas 3.2 e 3.3 mostram os tamanhos médios das soluções e do tempo gasto pelos algoritmos. As Tabelas 3.2a-3.2e mostram as médias das soluções agrupadas por densidade e a Tabela 3.2f mostra esses resultados considerando instâncias de todas as densidades. De forma semelhante, as Tabelas 3.3a-3.3e mostram o tempo médio de execução, agrupando instâncias com densidades iguais, e a Tabela 3.3f mostra esses resultados considerando todas as instâncias.

Nas Tabelas 3.2, a coluna `n` indica a quantidade de vértices das instâncias consideradas em cada linha, os valores nas colunas `Média` e `Máximo` abaixo de `Guloso` correspondem à média do tamanho das 100 decomposições retornadas pela heurística gulosa e a maior entre elas, a coluna `Primal` abaixo de `PLI` mostra os limitantes primais encontrados utilizando o modelo `PLI` com `B&P` e a coluna `Ótimo` à direita mostra a proporção de instâncias que foram provadas ótimas, e por fim, as colunas abaixo de `PLI melhorado` mostram os mesmos dados, porém, utilizando o modelo `PLI` com as variáveis iniciais fornecidas pela heurística gulosa e o limitante primal fornecido pela heurística `PLI`. Os valores nas tabelas preenchidos por "-" significam que nenhum limitante primal foi encontrado para as instâncias daquele cenário, portanto, 0% dos resultados são ótimos.

Nas Tabelas 3.3, a coluna `Guloso` mostra o tempo médio para gerar as 100 partições usando a heurística gulosa, a coluna `H_PLI` mostra o tempo médio de execução da heurística `PLI`, a coluna `PLI` mostra o tempo médio de execução do resolvedor para o modelo

PLI com B&P e a coluna PLI 2 mostra o tempo médio do resolvedor para o modelo PLI com B&P, usando as variáveis inicias da heurística gulosa e o limitante primal da heurística PLI.

	Guloso		PLI		PLI melhorado	
n	Média	Máximo	Primal	Ótimo	Primal	Ótimo
10	1.00	1.00	1.00	100%	1.00	100%
20	1.00	1.00	1.00	100%	1.00	100%
30	6.83	7.60	7.60	100%	7.60	100%
40	14.46	16.55	16.85	95%	16.85	100%
50	24.39	27.20	28.80	80%	28.85	100%
60	37.48	40.95	43.90	40%	44.15	90%
70	52.51	56.15	61.25	15%	61.85	40%
80	71.19	75.25	81.95	0%	83.35	10%
90	92.22	97.00	-	-	107.80	0%
100	116.43	121.20	-	-	134.75	0%

(a) Resultados com densidade 10%.

	Guloso		PLI		PLI melhorado	
n	Média	Máximo	Primal	Ótimo	Primal	Ótimo
10	1.00	1.00	1.00	100%	1.00	100%
20	8.36	9.10	9.10	100%	9.10	100%
30	20.90	22.90	23.85	95%	23.85	100%
40	39.01	41.80	44.65	65%	44.75	90%
50	63.25	66.60	71.95	20%	72.80	80%
60	93.49	97.05	105.35	5%	107.20	60%
70	130.26	134.45	-	-	148.40	0%
80	172.83	177.45	-	-	195.50	0%
90	221.70	227.05	-	-	248.50	0%
100	276.52	282.20	-	-	304.25	0%

(b) Resultados com densidade 20%.

	Guloso		PLI		PLI melhorado	
n	Média	Máximo	Primal	Ótimo	Primal	Ótimo
10	2.89	3.00	3.00	100%	3.00	100%
20	14.63	16.00	16.35	100%	16.35	100%
30	34.87	37.15	39.55	80%	39.65	100%
40	64.74	67.75	73.65	50%	73.80	90%
50	103.20	107.10	115.55	20%	117.00	75%
60	151.98	156.35	-	-	172.65	60%
70	209.63	215.05	-	-	237.15	30%
80	277.00	283.35	-	-	308.80	0%
90	354.29	363.20	-	-	394.65	0%
100	441.10	450.90	-	-	487.75	5%

(c) Resultados com densidade 30%.

	Guloso		PLI		PLI melhorado	
n	Média	Máximo	Primal	Ótimo	Primal	Ótimo
10	4.83	5.05	5.05	100%	5.05	100%
20	21.04	22.70	23.65	85%	23.70	100%
30	49.51	52.00	55.95	80%	56.10	100%
40	90.54	93.75	101.30	50%	102.05	100%
50	144.30	148.65	-	-	162.35	100%
60	210.99	216.65	-	-	235.45	100%
70	290.41	297.95	-	-	321.75	100%
80	382.82	393.40	-	-	420.95	95%
90	487.99	500.85	-	-	533.25	55%
100	606.16	622.10	-	-	658.35	10%

(d) Resultados com densidade 40%.

	Guloso		PLI		PLI melhorado	
n	Média	Máximo	Primal	Ótimo	Primal	Ótimo
10	6.05	6.40	6.40	100%	6.40	100%
20	27.39	29.20	30.75	100%	30.75	100%
30	63.88	66.50	71.20	60%	71.75	100%
40	116.56	120.30	128.55	35%	129.70	100%
50	185.03	190.50	-	-	203.95	100%
60	269.97	277.30	-	-	295.00	100%
70	370.75	380.45	-	-	402.00	100%
80	488.34	500.95	-	-	526.00	100%
90	621.70	637.95	-	-	666.75	75%
100	771.67	791.85	-	-	823.25	30%

(e) Resultados com densidade 50%.

	Guloso		PLI		PLI melhorado	
n	Média	Máximo	Primal	Ótimo	Primal	Ótimo
10	3.15	3.29	3.29	100%	3.29	100%
20	14.48	15.60	16.17	97%	16.18	100%
30	35.20	37.23	39.63	83%	39.79	100%
40	65.06	68.03	73.00	59%	73.43	95%
50	104.03	108.01	43.26	26%	116.99	91%
60	152.78	157.66	29.85	9%	170.89	82%
70	210.71	216.81	12.25	3%	234.23	53%
80	278.44	286.08	16.39	0%	306.92	41%
90	355.58	365.21	-	-	390.19	26%
100	442.37	453.65	-	-	481.67	9%

(f) Resultados para todas as densidades.

Tabela 3.2: Número de ciclos nas soluções obtidas para o problema MAX-ECD.

Na maioria dos casos, a heurística PLI retornou resultados melhores do que a heurística gulosa e do que o PLI com B&P. A heurística PLI encontrou uma solução ótima em 69.7% das instâncias. Nessas instâncias, o PLI com B&P provou a otimalidade da solução em 12.6 segundos, em média, e só demorou mais do que 60 segundos em 1.5% dos casos. Nas 30.3% instâncias restantes, o PLI com B&P não melhorou os limitantes primais nem provou a otimalidade da solução dentro do tempo limite.

Pelas Tabelas 3.2a-3.2e podemos observar que nas instâncias com maior densidade a solução ótima foi encontrada em mais casos do que nas instâncias com menor densidade. Por exemplo, para densidade 30% foram resolvidas 55% das instâncias, enquanto para densidade 50% foram resolvidas 90.5% das instâncias. Isso ocorre porque, quanto mais denso é o grafo, maiores são as chances de existir muitos triângulos nesse grafo, que são os menores ciclos possíveis em grafos simples. A Tabela 3.4 mostra a proporção de triângulos

nas soluções retornadas pela heurística PLI. Nas soluções das instâncias de densidade 10%, os triângulos compõem 29.96% dos ciclos, enquanto das instâncias de densidade 50%, os triângulos são 95.40% dos ciclos.

3.4 Conclusões

Neste capítulo, focamos no problema da Máxima Partição em Circuitos (MAX-ECD). Propusemos abordagens exatas e heurísticas. Implementamos o modelo PLI proposto por Caprara, Panconesi e Rizzi [7]. Desenvolvemos uma heurística gulosa e uma heurística que resolve parcialmente o modelo PLI. Realizamos experimentos com todas as abordagens e comparamos os resultados obtidos.

No melhor do nosso conhecimento, não existem experimentos computacionais para o problema MAX-ECD reportados na literatura. Então, criamos um conjunto de instâncias que usamos nos experimentos. Os grafos dessas instâncias foram gerados aleatoriamente, com as características de serem eulerianos e conexos.

O modelo PLI de Caprara, Panconesi e Rizzi [7] possui um número exponencial de variáveis, portanto, implementamos um algoritmo para resolver o problema de *pricing*, possibilitando a utilização de técnicas de *B&P*.

Executamos experimentos utilizando o modelo PLI proposto, porém, por causa do aumento da quantidade de vértices e da densidade das instâncias, o resolvedor não conseguiu resolver grande parte das instâncias dentro do tempo limite de execução.

Para a heurística gulosa proposta, adicionamos um passo não determinístico, que possibilitou obter resultados diferentes para uma mesma instância. Além de possivelmente gerar soluções melhores, usamos os ciclos dessas diversas soluções para executar uma outra heurística que usa o modelo PLI com apenas um subconjunto das variáveis.

Experimentos mostraram que essa heurística PLI melhora as soluções obtidas pela heurística gulosa e pelo modelo PLI completo em diversas instâncias e não piora em nenhuma.

Executamos novamente experimentos usando o modelo PLI com todas as variáveis, porém, usando os ciclos das soluções da heurística gulosa como conjunto inicial de variáveis e as soluções da heurística PLI como limitantes primais. Nenhuma solução melhor foi encontrada, porém, foi provada a otimalidade de 69.7% das soluções.

n	Guloso	H_PLI	PLI	PLI 2
10	0.00	0.00	0.00	0.00
20	0.01	0.00	0.00	0.00
30	0.21	0.00	0.00	0.00
40	1.02	0.02	90.05	0.06
50	3.89	0.23	391.90	0.20
60	6.97	1.55	1082.40	218.03
70	11.42	6.20	1629.85	1189.97
80	17.28	82.59	1800.00	1620.33
90	35.56	752.49	1800.00	1800.00
100	48.12	1800.00	1800.00	1800.00

(a) Resultados com densidade 10%.

n	Guloso	H_PLI	PLI	PLI 2
10	0.00	0.00	0.00	0.00
20	0.08	0.00	0.00	0.00
30	0.55	0.06	116.90	0.03
40	1.94	0.63	635.40	230.08
50	5.31	4.57	1525.60	369.83
60	12.09	113.80	1786.70	722.72
70	23.06	826.49	1800.00	1800.00
80	41.02	1709.79	1800.00	1800.00
90	68.08	1800.00	1800.00	1800.00
100	107.77	1800.00	1800.00	1800.00

(b) Resultados com densidade 20%.

n	Guloso	H_PLI	PLI	PLI 2
10	0.01	0.00	0.00	0.00
20	0.15	0.01	0.00	0.01
30	0.83	0.22	373.80	62.96
40	3.13	2.24	1071.05	360.14
50	9.21	24.01	1558.15	494.85
60	20.90	594.01	1800.00	906.69
70	42.67	1432.95	1800.00	1261.47
80	79.32	1800.00	1800.00	1800.00
90	139.11	1800.00	1800.00	1800.00
100	230.98	1749.13	1800.00	1710.90

(c) Resultados com densidade 30%.

n	Guloso	H_PLI	PLI	PLI 2
10	0.01	0.00	0.00	0.00
20	0.21	0.02	270.00	3.84
30	1.31	0.69	415.10	28.32
40	5.31	5.41	1054.95	0.29
50	15.63	19.72	1697.00	0.98
60	37.36	146.00	1800.00	20.71
70	77.82	107.12	1800.00	4.51
80	150.90	620.56	1800.00	99.14
90	268.44	998.60	1800.00	822.75
100	454.00	1686.91	1800.00	1623.69

(d) Resultados com densidade 40%.

n	Guloso	H_PLI	PLI	PLI 2
10	0.01	0.00	0.00	0.00
20	0.30	0.03	4.75	0.02
30	2.05	0.29	742.80	0.10
40	8.61	3.37	1243.40	0.37
50	25.35	17.44	1800.00	1.14
60	62.28	69.24	1800.00	2.65
70	134.90	325.33	1800.00	6.01
80	265.77	1266.17	1800.00	14.29
90	487.29	1428.34	1800.00	474.75
100	902.33	1665.81	1800.00	1273.16

(e) Resultados com densidade 50%.

n	Guloso	H_PLI	PLI	PLI 2
10	0.01	0.00	0.00	0.00
20	0.15	0.02	54.95	0.77
30	0.99	0.25	329.72	18.28
40	4.00	2.33	818.97	118.19
50	11.88	13.20	1394.53	173.40
60	27.92	184.92	1653.82	374.16
70	57.97	539.62	1765.97	852.39
80	110.86	1095.82	1800.00	1066.75
90	199.70	1355.89	1800.00	1339.50
100	348.64	1740.37	1800.00	1641.55

(f) Resultados para todas as densidades.

Tabela 3.3: Tempo de execução médio para obter as soluções para o problema MAX-ECD.

n	10%	20%	30%	40%	50%
10	0.00	0.00	31.67	59.41	67.19
20	0.00	36.81	63.30	81.43	91.87
30	24.34	53.46	75.13	89.19	97.63
40	27.89	60.65	83.40	94.27	99.31
50	34.09	67.29	86.40	98.18	99.93
60	37.41	72.42	92.56	99.41	100.00
70	38.54	75.57	94.89	99.77	99.75
80	42.74	77.85	95.34	99.30	99.33
90	46.10	81.34	97.37	99.45	99.45
100	48.53	83.44	98.45	99.45	99.50

Tabela 3.4: Proporção de triângulos das soluções do `MAX-ECD` retornadas pela heurística PLI.

Capítulo 4

Máxima Partição em Circuitos Alternados

Neste capítulo apresentamos o problema da Máxima Partição em Circuitos Alternados (*Maximum Alternating-Cycle Decomposition* - `MAX-ACD`). Na Seção 4.1, descrevemos o problema. Na Seção 4.2, detalhamos as abordagens que utilizamos para resolver o problema. Na Seção 4.3, descrevemos os experimentos realizados e discutimos os resultados obtidos. Na Seção 4.4, apresentamos as conclusões.

4.1 Introdução

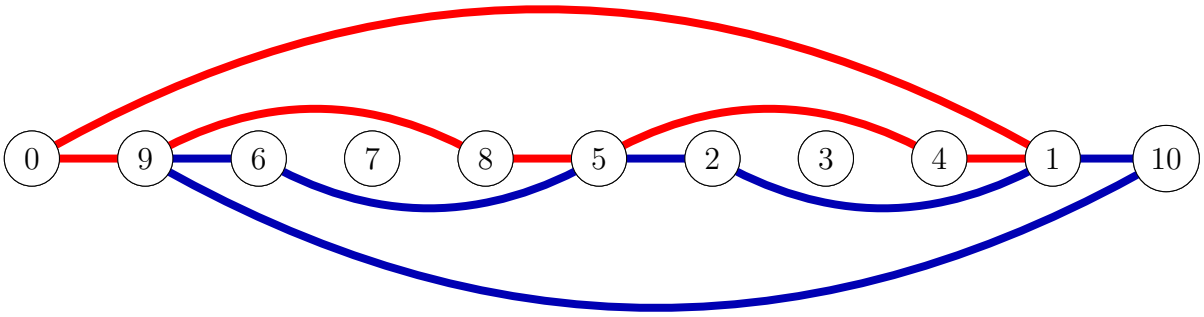
O `MAX-ACD` é definido como: dado um grafo euleriano bicolorido $G = (V, P \cup C)$, tal que P é o conjunto de arestas coloridas com a cor preta e C é o conjunto de arestas colorida com a cinza, que admite uma partição em circuitos alternados, encontrar a partição em circuitos alternados de maior cardinalidade.

A Figura 4.1 ilustra dois exemplos de Partição em Circuitos Alternados para o mesmo grafo. A primeira partição tem cardinalidade 2, onde o primeiro circuito é composto pelas arestas indicadas em azul e o segundo pelas arestas indicadas em vermelho, e a segunda partição tem cardinalidade 3 e é composta pelos circuitos com as arestas indicadas em vermelho, azul e verde. Vale lembrar que, como foi descrito na Seção 2.1, esses grafos são desenhados com as arestas pretas na horizontal e com as arestas cinza formando arcos.

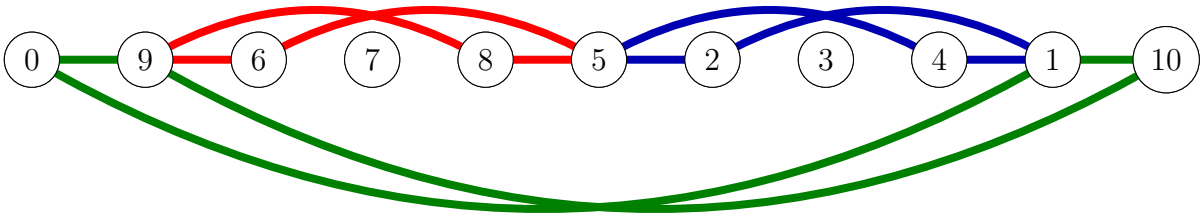
Lema 4.1.1. *Um grafo bicolorido $G = (V, P \cup C)$ admite uma partição em circuitos alternados se e somente se, para cada vértice u do grafo, $d_P(u) = d_C(u)$, isto é, a quantidade de arestas pretas incidentes no vértice u é igual à de arestas cinza.*

Demonstração. Dividimos a demonstração dessa afirmação em duas partes. Inicialmente mostramos a primeira parte da equivalência, isto é, a afirmação se G admite uma partição em circuitos alternados, então $d_P(u) = d_C(u), \forall\ u \in V$. Por fim provamos que se $d_P(u) = d_C(u), \forall\ u \in V$, então G admite uma partição em circuitos alternados.

A primeira parte é facilmente demonstrada, já que cada vértice tem quantidades iguais de arestas pretas e cinza incidente para cada circuito da partição, logo também é válido para a soma da quantidade de arestas em todos os circuitos.



(a)



(b)

Figura 4.1: Exemplos de partições em circuitos alternados de um grafo bicolorido.

A segunda parte é provada usando indução no número de circuitos alternados no grafo. O caso base da prova é quando temos um grafo vazio (sem arestas), que admite uma partição de cardinalidade 0. Como hipótese de indução temos que, se um grafo bicolorido $G' = (V', P' \cup C')$ possui a propriedade $d_{P'}(u) = d_{C'}(u), \forall\ u \in V'$, então G' possui uma partição em circuitos alternados. Para o passo indutivo, considere um grafo bicolorido $G = (V, P \cup C)$ com $d_P(u) = d_C(u), \forall\ u \in V$. Iremos demonstrar que sempre existe ao menos um circuito alternado nesse grafo. Seja $p = (u, v)$ uma aresta preta qualquer de P. Como $d_P(u) > 0$, então $d_C(u) > 0$. Seja $c = (u, w)$ uma aresta cinza que incide em u. Como $d_C(w) > 0$, então $d_P(w) > 0$. Em seguida, alguma aresta preta incidente em w poderia ser escolhida. Esse processo pode ser repetido até que uma aresta cinza incidente em v seja encontrada. Nesse momento essas arestas formam um circuito alternado A. O subgrafo G' será o grafo G após remover as arestas do circuito A e terá a propriedade $d_{P'}(u) = d_{C'}(u), \forall\ u \in V$, pois foi removida uma quantidade de arestas pretas de cada vértice igual à de arestas cinza. □

Note que, ao contrário do `MAX-ECD`, no `MAX-ACD` um circuito que não é um ciclo (um circuito que passa por um vértice mais de uma vez) pode ser necessário para uma solução ótima. No exemplo da Figura 4.2, a única solução para o `MAX-ACD` contém um circuito com essa característica, que é o circuito $(0, 1, 3, 4, 2, 3, 0)$.

Lema 4.1.2. *Um circuito que passa mais de duas vezes em um mesmo vértice não pode estar em uma solução ótima.*

Demonstração. Isso acontece porque um circuito que passa três ou mais vezes em uma mesmo vértice pode ser substituído por pelo menos dois circuitos na solução. Segue a

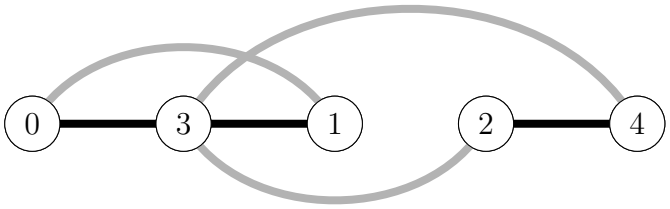


Figura 4.2: Exemplo de um grafo bicolorido que exige um circuito para particionar as arestas.

prova dessa afirmação. Todo circuito alternado possui comprimento par, já que tem uma quantidade igual de arestas pretas e cinza. Suponha, para efeitos de contradição, que exista uma solução ótima para o problema `MAX-ACD` que contenha um circuito C que passa três ou mais vezes no vértice u. Seja $\mathcal{P} = \{P_1, P_2, \ldots, P_k\}$ o conjunto de trilhas distintas que começam e terminam em u no circuito C.

Caso alguma trilha P_i tenha comprimento par, essa trilha P_i é um circuito alternado, já que tem arestas alternadas (porque faz parte do circuito alternado C) e as arestas do inicio e fim da trilha incidem no vértice u e são de cores distintas (pois a trilha tem comprimento par). O circuito C após remoção da trilha P continua sendo um circuito alternado (o argumento da trilha P_i também vale para o circuito C removido da trilha P_i). Logo, a solução pode ser melhorada substituindo o circuito C pelos dois circuitos descritos.

Caso todas as trilhas $P_1, P_2, \ldots, P_k$ tenham comprimento ímpar, o tamanho do conjunto $\mathcal{P}$ é par, pois a soma dos comprimentos das trilhas deve ser par. Note que as arestas do inicio e fim de uma trilha alternada de comprimento ímpar têm cores iguais. Sejam P_i e P_j duas trilhas consecutivas em C. A cor das arestas inicial e final de P_i é diferente das de P_j. A união de P_i com P_j forma um circuito alternado no grafo, já que têm arestas alternadas (porque faz parte do circuito alternado C) e incidem no vértice u com arestas de cores diferentes. Existem ao menos duas trilhas em $\mathcal{P} \setminus \{P_i, P_j\}$, pois $\mathcal{P}$ tem tamanho par maior do que dois. O argumento anterior vale para cada par de trilhas consecutivas em $\mathcal{P} \setminus \{P_i, P_j\}$. Portanto, a solução pode ser melhorada substituindo o circuito C por dois ou mais circuitos construídos da maneira descrita. □

Caprara [5] mostrou como o `MAX-ECD` e o `MAX-ACD` podem ser reduzidos em tempo polinomial um para o outro. Portanto, o `MAX-ACD` também pertence à classe NP-difícil. A redução do `MAX-ECD` para o `MAX-ACD` mantém fatores de aproximação. Porém, a redução do `MAX-ACD` para o `MAX-ECD` não mantém.

Usando `MAX-ACD` em grafos de *breakpoint* como subproblema, Lin e Jiang [19] desenvolveram um algoritmo de aproximação com fator $1.4193 + \epsilon$ para o problema da mínima ordenação por reversão (*Minimum Sorting by Reversals* - `MIN-SBR`). Nesse algoritmo, a solução do `MAX-ACD` no grafo de *breakpoint* influencia na solução do `MIN-SBR`.

O `MIN-SBR` consiste em, dado uma permutação $\pi = (\pi_1, \pi_2, \ldots, \pi_{n-1}, \pi_n)$ dos n primeiros inteiros positivos e uma operação de reversão ρ no intervalo (i, j) que inverte a

ordem dos elementos entre i e j (incluindo ambos), ou seja,

$$\pi\rho(i,j) = (\pi_1, \pi_2, \ldots, \pi_{i-1}, \underline{\pi_j, \pi_{j-1}, \ldots, \pi_{i+1}, \pi_i}, \pi_{j+1}, \ldots, \pi_n), \tag{4.1}$$

encontrar a menor sequência de reversões $\rho_1, \ldots, \rho_k$ que ordene a permutação π em ordem crescente, ou seja, $\pi\rho_1 \ldots \rho_k = (1, 2, \ldots, n-1, n)$. Chamamos essa permutação ordenada de ι_n.

Outras operações muito estudadas para o problema de ordenação, além das reversões, são transposição e *Double cut-and-join* (`DCJ`) [22]. Uma operação de transposição τ nas posições (i, j, k) da permutação troca os valores nos intervalos $(i, j-1)$ e $(j, k-1)$, ou seja,

$$\pi\tau(i,j,k) = (\pi_1, \pi_2, \ldots, \pi_{i-1}, \underline{\pi_j, \ldots, \pi_{k-1}}, \underline{\pi_i, \ldots, \pi_{j-1}}, \pi_k, \ldots, \pi_n). \tag{4.2}$$

Permitir operações diferentes pode alterar a quantidade de operações para ordenar a permutação, por exemplo, a permutação $\pi = (3, 4, 1, 2)$, para o problema `MIN-SBR`, pode ser ordenada por 3 reversões:

$$\pi = (\underline{3, 4, 1, 2}) \tag{4.3}$$
$$\pi\rho(1,4) = (\underline{2, 1}, 4, 3) \tag{4.4}$$
$$\pi\rho(1,4)\rho(1,2) = (1, 2, \underline{4, 3}) \tag{4.5}$$
$$\pi\rho(1,4)\rho(1,2)\rho(3,4) = (1, 2, 3, 4), \tag{4.6}$$

e na ordenação por reversão e transposição (*Minimum Sorting by Reversals and Transpositions* - `MIN-SRT`) pode ser ordenada por uma transposição:

$$\pi = (\underline{3, 4}, \underline{1, 2}) \tag{4.7}$$
$$\pi\tau(1,3,5) = (1, 2, 3, 4). \tag{4.8}$$

Essas soluções são ótimas para os problemas. Chamamos de Ordenação por Rearranjos o problema de ordenar uma permutação usando um subconjunto das operações mencionadas anteriormente.

4.2 Metodologia

Utilizamos abordagens heurísticas e exatas para resolver o problema `MAX-ACD`. Nas Seções 4.2.1 e 4.2.2 descrevemos algoritmos heurísticos para o problema. Na Seção 4.2.3 descrevemos o algoritmo exato.

4.2.1 Heurística Gulosa

Descreveremos a seguir o algoritmo guloso desenvolvido para encontrar soluções válidas para o `MAX-ACD`.

O algoritmo recebe como entrada o grafo bicolorido $G = (V, E)$ e retorna uma partição em circuitos alternados desse grafo. Essa heurística é semelhante à heurística apresentada na Seção 3.2.2, em que, inicialmente um conjunto $\mathcal{H}$ vazio de circuitos é criado e a cada

iteração um novo circuito alternado é adicionado em $\mathcal{H}$ e o grafo G é alterado para a próxima iteração. Por fim, quando não existir mais arestas no grafo, o conjunto $\mathcal{H}$ será retornado pelo algoritmo. A diferença consiste no algoritmo para encontrar o circuito mínimo que passa por um vértice específico, que no `MAX-ACD` precisa ser um circuito alternado.

O Algoritmo 6 descreve o funcionamento dessa heurística.

Algoritmo 6 Heurística gulosa para o `MAX-ACD`

```
1: função HEURÍSTICA-GULOSA(G = (V, E))
2:     H ← ∅
3:     enquanto |E| > 0 faça
4:         escolha um vértice v
5:         C ← Circuito-Alternado-Mínimo(G, v)
6:         H ← H ∪ {C}
7:         E ← E \ {C}
8:     fim enquanto
9:     devolve H
10: fim função
```

Note que a forma como o vértice v é escolhido na Linha 4 do Algoritmo 6 pode influenciar no resultado do algoritmo. Dessa forma, definimos 4 variações do algoritmo guloso:

- Variação `FIRST`, que consiste em escolher o vértice mais a esquerda, dada uma ordem específica dos vértices, que, no caso dos grafos de *breakpoint*, é a ordem da permutação que o originou.
- Variação `RANDOM`, que consiste em escolher aleatoriamente o vértice (semelhante ao usado na Seção 3.2.2).
- Variação `MAX`, que consiste em executar a variação `RANDOM` diversas vezes (pois cada execução pode retornar resultados diferentes) e retornar a melhor partição.
- Variação `ALL`, que consiste em encontrar o circuito alternado mínimo para cada vértice do grafo e escolher o menor dentre os encontrados.

Para exemplificar, considere o grafo bicolorido da Figura 4.3.

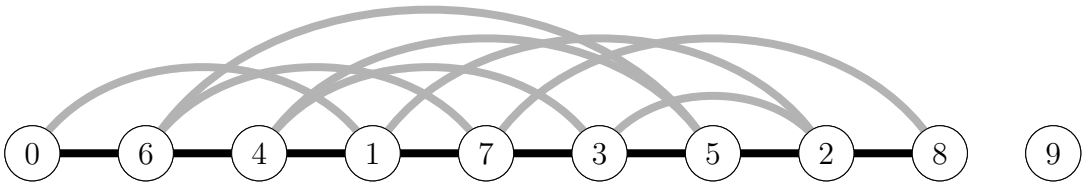


Figura 4.3: Exemplo um grafo bicolorido.

Caso a variação usada seja a `FIRST`, o resultado será uma partição de tamanho 4, ilustrada na Figura 4.4, em que cada circuito possui arestas com cores diferentes das

arestas dos outros circuitos. Na primeira iteração, o vértice escolhido seria o 0 e o circuito encontrado seria o circuito verde. Na segunda, o vértice 6 e o circuito vermelho. Na terceira, o vértice 4 e o circuito magenta. Como o vértice 1 não teria arestas incidentes restantes, na quarta iteração o vértice escolhido seria o 7, resultando no circuito azul.

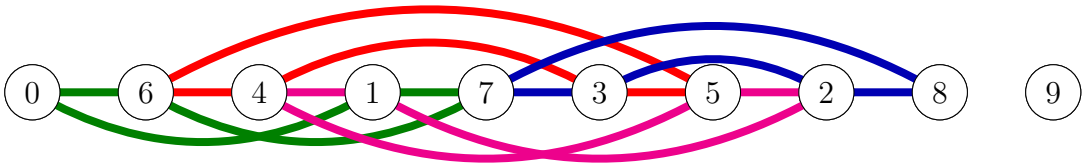


Figura 4.4: Solução para o `MAX-ACD` obtida pelo algoritmo guloso nas variações `FIRST`, `MAX` e `ALL`.

Caso a variação usada seja a `RANDOM` e os vértices escolhidos aleatoriamente sejam 6, 1 e 0, nessa ordem, então o algoritmo resultaria em uma partição de tamanho 3, ilutrada na Figura 4.5. Na primeira iteração, o circuito encontrado seria o vermelho. Na segunda, o azul. Na terceira iteração, só existe um circuito restante no grafo, que é o circuito verde. Logo, esse é o circuito adicionado na solução nessa iteração.

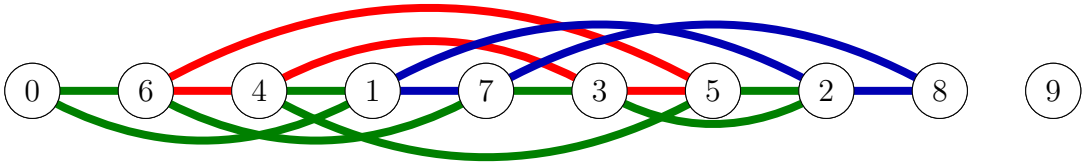


Figura 4.5: Solução para o `MAX-ACD` obtida pelo algoritmo guloso na variação `RANDOM`.

Para exemplificar a variação `MAX`, considere que a variação `RANDOM` foi executada duas vezes, na primeira, os vértices escolhidos foram 6, 1 e 0, nessa ordem, e na segunda foram 0, 6, 4 e 7, também nessa ordem. Os resultados seriam iguais os das Figuras 4.5 e 4.4. Portanto, o resultado final seria igual ao da Figura 4.4.

A partição da Figura 4.4 também seria o resultado da variação `ALL`, já que todos os circuitos possuem comprimento 4, que é o comprimento mínimo de um circuito alternado.

Nenhuma das variações do algoritmo guloso pode garantir um fator de aproximação, porém, é possível adaptar esses algoritmos para terem o mesmo fator de aproximação do algoritmo de Lin e Jiang [19]. Vale ressaltar que o algoritmo de Lin e Jiang não possui uma aproximação para o `MAX-ACD`, mas para o `MIN-SBR`. Essa alteração consiste em incluir no conjunto inicial de circuitos $\mathcal{H}$ do algoritmo guloso os circuitos de comprimento 6 ou menores usados pelo algoritmo de Lin e Jiang e remover as arestas desses circuitos do grafo [22].

4.2.2 Busca Tabu

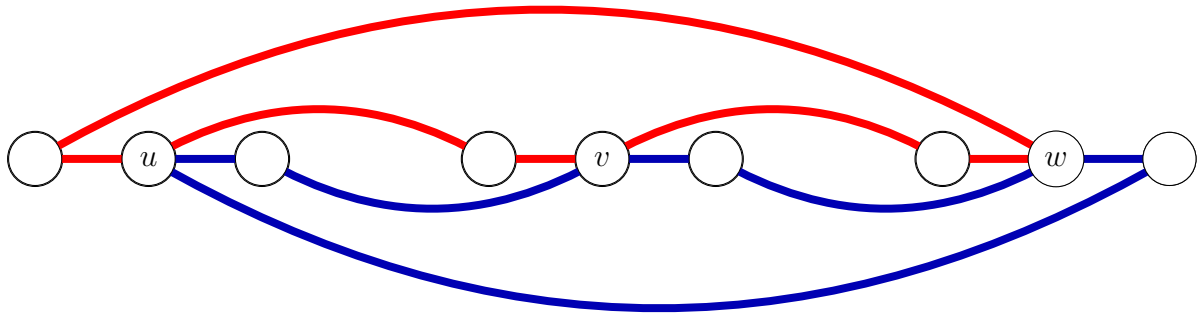
Propusemos uma heurística para o `MAX-ACD` baseada na meta-heurística Busca Tabu. Usamos os algoritmos da Seção 4.2.1 para obter uma solução inicial e os dois movimentos descritos a seguir para executar a busca local.

O primeiro movimento é aplicado em dois circuitos C_1 e C_2 e troca esses dois circuitos por três circuitos que usam as mesmas arestas.

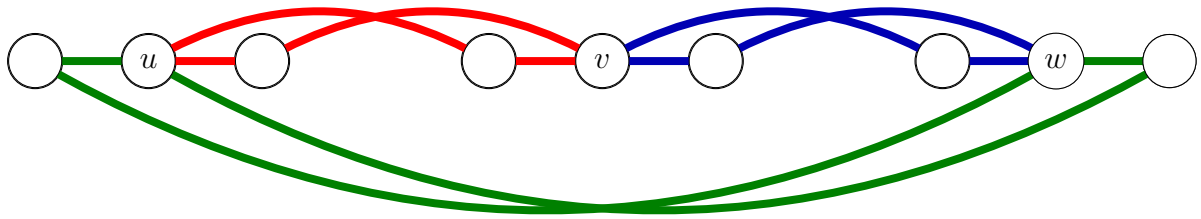
Para efetuar esse movimento, algumas condições precisam ser satisfeitas. Primeiramente, os dois circuitos precisam ter pelo menos três vértices em comum, digamos u, v e w. Dado dois vértices $x, y \in \{u, v, w\}$, P^1_{xy} e P^2_{xy} são os caminhos de u para v em C_1 e C_2, respectivamente, que não passam pelo terceiro vértice em comum. A próxima condição é que P^1_{uv} e P^2_{uv} tenham comprimentos com paridades iguais e a cor da primeira aresta desses caminhos precisa ser diferente. Por fim, o mesmo deve ser válido entre os caminhos P^1_{vw} e P^2_{vw} e entre os caminhos P^1_{wu} e P^2_{wu}.

Quando essas condições são satisfeitas, uma partição com um circuito a mais é criada, substituindo os circuitos C_1 e C_2 pelos circuitos C'_1, C'_2 e C'_3, onde C'_1 é composto pela união dos caminhos P^1_{uv} e P^2_{uv}, C'_2 é a união dos caminhos P^1_{vw} e P^2_{vw} e C'_3 é a união dos caminhos P^1_{wu} e P^2_{wu}. Note que as condições necessárias para executar o movimento garantem que C'_1, C'_2 e C'_3 são circuitos alternados.

A Figura 4.6 mostra um exemplo da aplicação desse movimento. Os circuitos C_1 e C_2 seriam os circuitos com arestas azuis e vermelhas em (a) e os circuitos C'_1, C'_2 e C'_3 são os circuitos com arestas vermelhas, azuis e verdes em (b).



(a)



(b)

Figura 4.6: Exemplo de aplicação do primeiro movimento da Busca Tabu.

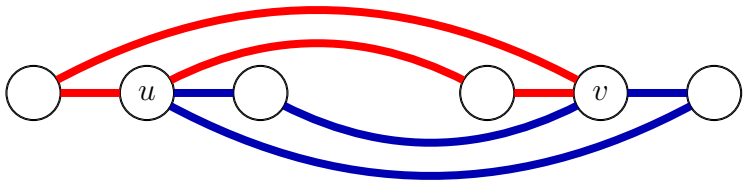
O segundo movimento também é aplicado em dois circuitos C_1 e C_2, porém, esses circuitos são substituídos por apenas dois circuitos distintos.

As condições para aplicar esse movimentos são ter dois vértices u e v em comum nos dois circuitos e a condição sobre os caminhos que descreveremos a seguir. Sejam P^1_{uv} e Q^1_{uv} os dois caminhos distintos de u para v em C_1, de forma que, P^1_{uv} começa em u com

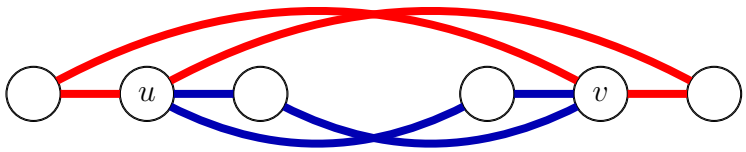
uma aresta preta e Q^1_{uv} começa em v (com aresta de qualquer cor). Sejam P^2_{uv} e Q^2_{uv} os caminhos definidos de forma análoga em C_2. Note que Q^1_{uv} e Q^2_{uv} terminam em arestas cinza. A condição é que P^1_{uv} e P^2_{uv} tenham comprimentos com paridades iguais.

Quando aplicado, o segundo movimento substitui os circuitos C_1 e C_2 pelos circuitos C'_1 e C'_2, onde C'_1 é composto pela união dos caminhos P^1_{uv} e Q^2_{uv} e C'_2 é composto pela união dos caminhos P^2_{uv} e Q^1_{uv}.

A Figura 4.7 mostra um exemplo da aplicação desse movimento. Os circuitos C_1 e C_2 seriam os circuitos em (a) e os circuitos C'_1 e C'_2 seriam os circuitos em (b).



(a)



(b)

Figura 4.7: Exemplo de aplicação do segundo movimento da Busca Tabu.

O primeiro movimento melhora a solução sempre que é aplicado. O segundo movimento não melhora a solução, mas permite continuar a busca após encontrar um ótimo local, pois, nesse caso, nenhum par de circuitos irá satisfazer as condições do primeiro movimento e o segundo movimento modifica esses circuitos, podendo possibilitar o uso do primeiro movimento nessa nova solução.

Porém, esse movimento precisa ser limitado, caso contrário a busca pode entrar em um ciclo infinito, aplicando o segundo movimento para substituir os circuitos C_1 e C_2 pelos circuitos C'_1 e C'_2 e, na próxima iteração, aplicando novamente o movimento nos circuitos C'_1 e C'_2, retornando a solução anterior. Para isso, após aplicar o movimento, o par de circuitos C'_1 e C'_2 entra na Lista Tabu, o que impede a aplicação do segundo movimento nesses circuitos nas próximas $|V|$ iterações.

4.2.3 Modelo PLI

O algoritmo exato que desenvolvemos usa PLI e *B&P* para resolver o problema. O modelo PLI foi proposto por Caprara, Lancia e See-Kiong Ng [6]. O algoritmo que implementamos é semelhante ao algoritmo descrito na Seção 3.2.1, com diferenças somente no conjunto de variáveis, já que os circuitos devem ser alternados, então devemos alterar o algoritmo de encontrar os circuitos mínimos.

Para isso, usamos um grafo bicolorido direcionado descrito a seguir. Dado um grafo bicolorido $G = (V, P \cup C)$, criaremos o grafo bicolorido direcionado $G' = (V', P' \cup C')$.

Para cada $v \in V$, o conjunto V' terá dois vértices, v e v'. Para cada aresta $p = (u, v) \in P$, o conjunto P' possui os arcos (u, v') e (v, u'). Para cada aresta $c = (u, v) \in C$, o conjunto C' possui os arcos (u', v) e (v', u). Dessa forma, para todos os vértices de V', os arcos de entrada terão cor diferente dos arcos de saída, portanto, todos os circuitos em G' são alternados. Assim, podemos usar o Algoritmo 3 para encontrar circuitos alternados mínimos. Por fim, para converter o circuito do grafo G' em um circuito no grafo G, os vértices v' são substituídos pelos vértices v, os arcos $p' = (u, v')$ são substituídos pelas arestas $p = (u, v)$ e os arcos $c' = (u', v)$ são substituídos pelas arestas $c = (u, v)$. Na Figura 4.8 mostramos em (a) um grafo bicolorido e em (b) o grafo bicolorido direcionado utilizado no *pricing*.

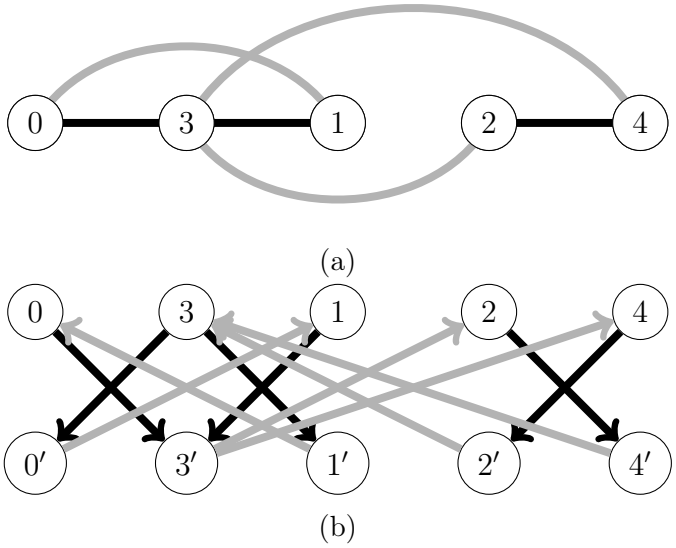


Figura 4.8: Exemplo de grafo bicolorido adaptado para *pricing*.

4.3 Experimentos Computacionais

Nesta seção descrevemos os resultados dos experimentos computacionais executados, assim como o ambiente em que os experimentos foram realizados e como foram criadas as instâncias.

O computador utilizado possui um processador Intel Core i7-8700T CPU com 12 núcleos de 2.40 GHz, memória RAM de 8 GB e sistema operacional Ubuntu 18.04.3. O tempo limite de execução em todos os experimentos foi de 1800 segundos. O resolvedor `PL` e `PLI` (sem *B&P*) foi o Gurobi 8.1.1 [15]. O resolvedor `PLI` com `B&P` foi o SCIP 6.0.2 [1].

As instâncias usadas nos experimentos foram grafos de *breakpoint* gerados a partir de permutações aleatórias. Para cada $n \in \{10, 20, \ldots, 90, 100\}$ foram geradas 100 permutações. Nessas permutações, dois números consecutivos diferem em pelo menos 2 unidades. Assim, o grafo de *breakpoint* terá o número máximo de arestas.

Primeiramente, realizamos experimentos para determinar o melhor valor k, que é a quantidade de vezes que o algoritmo guloso na variação `MAX` deve executar a variação `RANDOM`. A Tabela 4.1 mostra, em média, para cada tamanho de instância, o tamanho das

partições encontradas pela variação `MAX` com diversos valores de k em função do tamanho n da permutação. As linhas da tabela variam para os diversos tamanhos n de instâncias, e as colunas variam entre as opções de k em função de n.

Tamanho	n	$n \log n$	$n\sqrt{n}$	1000	n^2
10	**4.14**	**4.14**	**4.14**	**4.14**	**4.14**
20	6.88	**6.89**	**6.89**	**6.89**	**6.89**
30	9.20	**9.22**	**9.22**	**9.22**	**9.22**
40	11.39	**11.40**	**11.40**	**11.40**	**11.40**
50	13.39	13.55	**13.58**	**13.58**	**13.58**
60	15.29	15.51	15.57	15.59	**15.61**
70	17.05	17.32	17.41	17.44	**17.48**
80	18.96	19.28	19.43	19.50	**19.68**
90	20.34	20.84	21.05	21.06	**21.30**
100	22.19	22.63	22.81	22.81	**23.19**
Todos	13.88	14.08	14.15	14.16	**14.25**

Tabela 4.1: Tamanho médio das partições para diversos números de iterações da variação `MAX` da heurística gulosa.

A melhor relação entre qualidade da solução obtida e tempo de processamento necessário, foi quando $k = n\sqrt{n}$, pois gera os mesmos resultados que $k = n^2$ para instâncias de tamanho até 50 e resultados semelhantes para o restante, com uma diferença de apenas 0.09 unidades quando consideradas em média de todas as instâncias.

Para avaliar o desempenho das heurísticas descritas, executamos 4 experimentos e comparamos os resultados. O primeiro consiste em comparar as diversas variações do algoritmo guloso com o algoritmo que Lin e Jiang [19] usaram para resolver o `MAX-ACD` (`L&J`). O segundo, em repetir esse experimento, porém, impondo no algoritmo guloso os circuitos curtos do `L&J`. Os dois últimos cenários são semelhantes aos dois primeiros, porém, acrescido da Busca Tabu ao final da execução do guloso (e do `L&J`). Os resultados da variação `RANDOM` mostram a média de 100 execuções.

A Tabela 4.2 mostra o número médio de circuitos retornados pelo `L&J` e as quatro variações do algoritmo guloso (`FIRST`, `ALL`, `RANDOM` e `MAX`). Podemos ver que, exceto para permutações de tamanho até 20, todas as variações do algoritmo guloso retornaram partições com um número maior de circuitos em média em comparação com aquelas retornadas pelo `L&J`. Além disso, as partições obtidas pelo `MAX` têm em média mais circuitos do que L&J para todas as permutações de tamanho maior ou igual a 20. Para permutações de tamanho maior ou igual a 60, a heurística `MAX` retornou partições cujo número de circuitos é pelo menos 50% maior, em média, do que as partições de circuito retornadas pelo L&J. Os resultados do Experimento 2 mostram que usando o mesmo conjunto de circuitos curtos de L&J na partição do circuito, as variações `FIRST`, `ALL` e `RANDOM` retornaram partições com mais circuitos em comparação com seus resultados no Experimento 1. Porém, a variação `MAX`, que produz os melhores resultados em média, retornou partições com um número menor de circuitos, em média, do que as retornadas pelo Experimento 1 (usando a mesma variação) para permutações com tamanho entre 20 e 80.

A Tabela 4.3 mostra o número médio de circuitos retornados pela Busca Tabu usando

n	Experimento 1					Experimento 2			
	L&J	FIRST	ALL	RANDOM	MAX	FIRST	ALL	RANDOM	MAX
10	**4.14**	3.99	4.09	3.98	**4.14**	**4.14**	**4.14**	**4.14**	**4.14**
20	6.58	6.23	6.63	6.35	**6.89**	6.75	6.75	6.75	6.75
30	8.01	8.04	8.75	8.27	**9.22**	8.89	8.90	8.88	8.91
40	9.12	9.98	10.70	10.06	**11.39**	10.81	10.89	10.84	10.97
50	9.70	11.39	12.75	11.67	**13.45**	12.61	12.74	12.62	12.97
60	10.20	13.06	14.38	13.20	**15.38**	14.25	14.47	14.27	14.88
70	10.97	14.45	16.36	14.73	**17.13**	16.09	16.33	16.02	16.84
80	11.52	15.69	18.16	16.30	**19.01**	17.77	18.24	17.77	18.95
90	11.73	17.27	19.51	17.60	20.44	19.14	19.63	19.13	**20.72**
100	12.31	18.35	21.35	19.03	22.17	20.67	21.27	20.71	**22.54**

Tabela 4.2: Número médio de circuitos nas partições retornadas nos experimentos 1 e 2.

a saída de L&J e as quatro variações do algoritmo guloso. Podemos ver que a Busca Tabu foi capaz de melhorar os resultados de todos os algoritmos, principalmente do L&J que teve uma grande melhoria (provavelmente porque retornou o menor número médio de circuitos, com um grande espaço para melhorias). A variação MAX permanece como a que retorna o maior número de circuitos em média, e o mesmo comportamento do Experimento 2 aconteceu aqui: em média, os resultados do Experimento 4 são ligeiramente piores que os resultados do Experimento 3, para tamanhos de permutações maiores que 10.

n	Experimento 3					Experimento 4			
	L&J	FIRST	ALL	RANDOM	MAX	FIRST	ALL	RANDOM	MAX
10	**4.14**	4.09	4.12	**4.14**	**4.14**	**4.14**	**4.14**	**4.14**	**4.14**
20	6.76	6.50	6.80	**6.89**	**6.89**	6.77	6.77	6.81	6.81
30	8.87	8.54	9.00	**9.22**	**9.22**	9.03	9.02	9.13	9.13
40	10.85	10.50	11.12	11.39	**11.40**	11.08	11.14	11.33	11.33
50	12.55	12.28	13.06	13.45	**13.58**	12.99	13.08	13.40	13.42
60	14.39	13.97	14.90	15.40	**15.61**	14.75	14.85	15.34	15.41
70	16.16	15.72	16.78	17.15	**17.48**	16.60	16.79	17.35	17.41
80	17.90	17.18	18.67	19.11	**19.63**	18.39	18.77	19.42	19.54
90	19.24	18.65	20.17	20.58	**21.33**	19.87	20.24	21.14	21.26
100	20.72	20.29	21.96	22.33	**23.27**	21.56	21.99	22.94	23.14

Tabela 4.3: Número médio de circuitos nas partições retornadas nos experimentos 3 e 4.

Usamos o algoritmo exato com PLI principalmente para obter limitantes duais, já que poucos limitantes primais melhores foram encontrados.

Usando o PLI sem *B&P* como heurística, assim como descrito na Seção 3.2.3, com os resultados do Experimento 3 para variáveis iniciais e para limitante primal inicial, fomos capazes de melhorar o limitante primal em apenas 19 instâncias das 1000, e em apenas 1 unidade. Repetimos o experimento usando os resultados do Experimento 4 como variáveis e limitantes iniciais, porém o resultado se manteve.

Esse mesmo resultado foi obtido quando executamos o PLI com *B&P*. Porém, agora obtemos limitantes duais válidos. Com esses limitantes duais, atestamos que 96.4% das instâncias foram resolvidas com valores ótimos.

4.3.1 Experimentos com Ordenação por Rearranjos

Algoritmos para o MAX-ACD são frequentemente usados como parte de algoritmos de aproximação para problemas de Ordenação por Rearranjos, como MIN-SBR, MIN-SRT ou ordenação por DCJ [8, 19, 24].

Usamos os resultados obtidos no Experimento 4 nos algoritmos de Ordenação por Rearranjos para avaliar como as diferentes partições retornadas afetam os resultados. Vale lembrar que, o objetivo é encontrar a menor sequência de rearranjos que ordena a permutação. Chamamos o tamanho dessa sequência de distância de rearranjo.

Para uma permutação π e uma partição $\mathcal{H}$ em circuitos alternados de $G(\pi)$, uma ordenação usando DCJ pode ser feita em $b(\pi) - |\mathcal{H}|$ operações [8, 27], onde $b(\pi)$ é o número de arestas pretas em $G(\pi)$. Como $b(\pi)$ é constante em uma instância, esse valor depende somente de $|\mathcal{H}|$, portanto, os resultados desse experimento serão semelhantes aos da Tabela 4.3. Esses resultados estão apresentados na Tabela 4.4.

n	L&J	FIRST	ALL	RANDOM	MAX
10	**6.86**	**6.86**	**6.86**	**6.86**	**6.86**
20	14.42	14.23	14.23	**14.19**	**14.19**
30	22.99	21.97	21.98	**21.87**	**21.87**
40	31.88	29.92	29.86	**29.67**	**29.67**
50	41.30	38.01	37.92	37.60	**37.58**
60	50.80	46.25	46.15	45.66	**45.59**
70	60.03	54.40	54.21	53.65	**53.59**
80	69.48	62.61	62.23	61.58	**61.46**
90	79.27	71.13	70.76	69.86	**69.74**
100	88.69	79.44	79.01	78.06	**77.86**

Tabela 4.4: Média das distâncias de rearranjo usando a operação de DCJ.

Para os experimentos com reversão e transposição, usamos o algoritmo apresentado por Rahman *et al.* [24], que usa como subproblema o MAX-ACD. Para uma permutação π e uma partição $\mathcal{H}$ de $G(\pi)$, o algoritmo retorna uma sequência de operações S que ordena a permutação, onde $\frac{b(\pi)-|\mathcal{H}|}{2} \leq |S| \leq b(\pi) - |\mathcal{H}|$. Quando consideramos as operações de reversão e transposição, algumas características dos circuitos da partições $\mathcal{H}$ podem influenciar na distância de rearranjo. Por esse motivo, uma partição $\mathcal{H}_1$ pode resultar em distância de rearranjo maior do que uma partição $\mathcal{H}_2$, mesmo se $|\mathcal{H}_1| < |\mathcal{H}_2|$ for válido. Esses resultados estão apresentados na Tabela 4.5.

Todos os algoritmos propostos na Seção 4.2 tiveram resultados significativamente melhores do que os do algoritmo que Lin e Jiang usaram para resolver o MAX-ACD. Apesar da variação FIRST apresentar resultados piores para todos os tamanhos de instâncias na Tabela 4.3, ela apresenta, em média, as menores distâncias de rearranjo com reversão e

n	L&J	FIRST	ALL	RANDOM	MAX
10	**6.02**	6.23	6.23	6.20	6.21
20	12.16	12.17	12.15	12.06	**11.91**
30	18.64	**17.70**	17.79	17.83	**17.70**
40	25.11	**23.17**	23.22	23.26	23.24
50	32.86	29.03	28.89	28.83	**28.79**
60	39.91	33.98	34.26	34.24	**33.68**
70	46.83	39.48	39.44	39.48	**39.04**
80	54.69	44.45	44.56	44.74	**44.29**
90	61.12	**49.55**	50.10	50.11	49.56
100	68.59	55.04	54.91	55.20	**54.13**

Tabela 4.5: Média das distâncias de rearranjo usando as operações de reversão e transposição.

transposição para permutações de tamanho 40 e 90. No entanto, no caso geral, a variação MAX continua apresentando os melhores resultados.

4.4 Conclusões

Neste capítulo focamos no problema da Máxima Partição em Circuitos Alternados (MAX-ACD) e em suas aplicações no problema da Ordenação por Rearranjos. Propusemos abordagens heurísticas e exatas. Implementamos o modelo PLI proposto por Caprara, Lancia e See-Kiong Ng [6]. Desenvolvemos heurísticas gulosas e uma heurística baseada na meta-heurística Busca Tabu. Realizamos experimentos com todas as abordagens e comparamos os resultados obtidos entre elas e com algoritmos encontrados na literatura. Avaliamos também o desempenho desses algoritmos quando aplicados no problema de Ordenação por Rearranjos.

Criamos um conjunto de instâncias para o MAX-ACD, que são grafos de *breakpoint* gerados a partir de permutações aleatórias. As permutações possuem características que fazem o grafos possuírem o número máximo de arestas possível.

O modelo PLI de Caprara, Lancia e See-Kiong Ng [6] possui um número exponencial de variáveis, portanto, implementamos um algoritmo para resolver o problema de *pricing*, possibilitando a utilização de técnicas de *B&P*.

Propusemos uma heurística gulosa e quatro variações dessa heurística, uma delas com passos não determinísticos, possibilitando obtenção de resultados diferentes para uma mesma instância, e outra delas consiste em executar essa variação não determinística diversas vezes. Na maioria dos experimentos, as heurísticas apresentadas tiveram melhores resultados que o algoritmo proposto por Lin e Jiang [19].

Usando os resultados obtidos pelas heurísticas gulosas, executamos uma Busca Tabu, que foi capaz de melhorar o resultado em grande parte das instâncias, até mesmo os resultados do algoritmo de Lin e Jiang.

Executamos o algoritmo PLI com e sem *B&P* usando um conjunto não vazio de variáveis inicias e limitantes primais fornecidos pelas heurísticas. Obtivemos uma melhora

pequena nos limitantes primais, porém, com os limitantes duais fornecidos pelo `PLI` provamos otimalidade de 96.4% das soluções.

Quando utilizados para resolver os problemas de Ordenação por Reversão, os algoritmos propostos continuam mostrando bom desempenho, superando o algoritmo de Lin e Jiang.

Capítulo 5

Considerações Finais

Neste trabalho, abordamos dois problemas de partição em grafos, o `MAX-ECD` e o `MAX-ACD`. Para ambos os problemas propomos algoritmos heurísticos e exatos.

Como apresentado no Capítulo 3, para o `MAX-ECD`, as heurísticas desenvolvidas foram um algoritmo guloso e um algoritmo para resolver o modelo `PLI` proposto por Caprara, Panconesi e Rizzi [7] com apenas um subconjunto das variáveis, tornando possível armazenar em memória todas as variáveis. Já o algoritmo exato, consiste em resolver esse mesmo modelo `PLI` com o conjunto de variáveis completo. Para isso, foi necessário a utilização de técnicas de *B&P*. Implementamos um algoritmo para o problema de *pricing* que viabilizou a utilização dessas técnicas. O artigo "*Algorithms for the Maximum Eulerian Cycle Decomposition Problem*" (Pedro O. Pinheiro, Alexsandro O. Alexandrino, André R. Oliveira, Cid C. de Souza, e Zanoni Dias) [23], que contém parte dos resultados desse capítulo, será apresentado no *LIII Simpósio Brasileiro de Pesquisa Operacional* (SBPO), em novembro de 2021.

Porém, o alto custo computacional do algoritmo de *pricing* fez com que o algoritmo exato proposto tivesse um desempenho ruim, pior do que o da heurística gulosa. Remover o tempo necessário para gerar variáveis do `PLI` fez que a heurística `PLI` tivesse bons resultados, melhorando de forma significativa os resultados da heurística gulosa. Executamos novamente o modelo exato, dessa vez usando os limitante e as variáveis encontradas pelas heurísticas, para produzir limitantes duais e possivelmente melhorar os limitantes primais. Nenhum limitante primal foi melhorado, mas foi provado que 69.7% das soluções já obtidas eram ótimas.

No Capítulo 4, abordamos o `MAX-ACD`. As heurísticas gulosas desenvolvidas foram algoritmos gulosos e um algoritmo baseado na meta-heurística Busca Tabu. Foi feita uma pequena adaptação no algoritmo exato do `MAX-ECD`, para resolver de forma exata o `MAX-ACD`. O artigo "*Heuristics for Breakpoint Graph Decomposition with Applications in Genome Rearrangement Problems*" (Pedro O. Pinheiro, Alexsandro O. Alexandrino, André R. Oliveira, Cid C. de Souza, e Zanoni Dias) [22], que contém parte dos resultados desse capítulo, foi apresentado no *XIII Brazilian Symposium on Bioinformatics (BSB)*, em novembro de 2020.

Os algoritmos gulosos propostos apresentaram resultados melhores que os do algoritmo de Lin e Jiang [19], e a Busca Tabu foi capaz de melhorar consistentemente esses resultados. Usando o algoritmo exato, junto com os limitantes primais e variáveis iniciais

encontrados pelas heurísticas, para obter melhores limitantes primais e limitantes duais, provamos a otimalidade de 96.4% das soluções. Também avaliamos o desempenho desses algoritmos quando aplicados para resolver o problema da Ordenação por Rearranjos. Os resultados dos algoritmos propostos foram melhores que aqueles do algoritmo de Lin e Jiang.

As instâncias utilizadas nos experimentos foram disponibilizadas num repositório público [1].

Um possível trabalho futuro seria o desenvolvimento de um algoritmo de *pricing* mais eficiente para o *B&P* dos modelos `PLI` utilizados, pois esse foi o principal gargalo desses modelos.

Como os algoritmos gulosos randomizados funcionaram bem, principalmente para o `MAX-ACD`, outro possível trabalho futuro seria desenvolver algoritmos baseados na meta-heurística `GRASP`. Da mesma forma, como a heurística `PLI` funcionou bem, principalmente para o `MAX-ECD`, mais uma sugestão de trabalho futuro é o desenvolvimento de algoritmos baseados na meta-heurística de Algoritmos Genéticos.

[1] `https://github.com/pedroonop/instances-for-cycle-decomposition`

Referências Bibliográficas

[1] T. Achterberg. SCIP: Solving Constraint Integer Programs. *Mathematical Programming Computation*, 1(1):1–41, 2009.

[2] V. Bafna and P. A. Pevzner. Genome rearrangements and sorting by reversals. *SIAM Journal on Computing*, 25(2):272–289, 1996.

[3] J. A. Bondy and U. S. R. Murty. *Graph Theory*. Graduate Texts in Mathematics. Springer, 2008.

[4] L. Bulteau, G. Fertin, and I. Rusu. Sorting by transpositions is difficult. *SIAM Journal on Discrete Mathematics*, 26(3):1148–1180, 2012.

[5] A. Caprara. Sorting permutations by reversals and Eulerian cycle decompositions. *SIAM Journal on Discrete Mathematics*, 12(1):91–110, 1999.

[6] A. Caprara, G. Lancia, and SK. Ng. Sorting permutations by reversals through branch-and-price. *INFORMS Journal on Computing*, 13(3):224–244, 2001.

[7] A. Caprara, A. Panconesi, and R. Rizzi. Packing cycles in undirected graphs. *Journal of Algorithms*, 48(1):239–256, 2003.

[8] X. Chen. On sorting unsigned permutations by double-cut-and-joins. *Journal of Combinatorial Optimization*, 25(3):339–351, 2013.

[9] D. A. Christie. A 3/2-approximation algorithm for sorting by reversals. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 244–252, 1998.

[10] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009.

[11] G. B. Dantzig. *Linear programming and extensions*, volume 48. Princeton University Press, 1998.

[12] S. Ganesamurthy and P. Paulraja. $2p$-cycle decompositions of some regular graphs and digraphs. *Discrete Mathematics*, 341(8):2197–2210, 2018.

[13] F. W. Glover. Tabu Search - Part I. *INFORMS Journal on Computing*, 1(3):190–206, 1989.

[14] F. W. Glover. Tabu Search - Part II. *INFORMS Journal on Computing*, 2(1):4–32, 1990.

[15] Incorporate Gurobi Optimization. Gurobi Optimizer Reference Manual. *URL* `http://www.gurobi.com`, 2015.

[16] S. L. Hakimi. On Realizability of a Set of Integers as Degrees of the Vertices of a Linear Graph. I. *Journal of the Society for Industrial and Applied Mathematics*, 10(3):496–506, 1962.

[17] H. Jiang, L. Pu, L. Qingge, D. Sankoff, and B. Zhu. A randomized fpt approximation algorithm for maximum alternating-cycle decomposition with applications. In *International Computing and Combinatorics Conference (COCOON)*, pages 26–38. Springer, 2018.

[18] M. Krivelevich, Z. Nutov, M. R. Salavatipour, J. Verstraete, and R. Yuster. Approximation algorithms and hardness results for cycle packing problems. *ACM Transactions on Algorithms*, 3(4):48–es, 2007.

[19] G. Lin and T. Jiang. A further improved approximation algorithm for breakpoint graph decomposition. *Journal of Combinatorial Optimization*, 8(2):183–194, 2004.

[20] C. M. Mynhardt and C. M. van Bommel. Triangle decompositions of planar graphs. *Discussiones Mathematicae Graph Theory*, 36(3):643–659, 2016.

[21] A. R. Oliveira, K. L. Brito, U. Dias, and Z. Dias. On the complexity of sorting by reversals and transpositions problems. *Journal of Computational Biology*, 26(11):1223–1229, 2019.

[22] P. O. Pinheiro, A. O. Alexandrino, A. R. Oliveira, C. C. de Souza, and Z. Dias. Heuristics for breakpoint graph decomposition with applications in genome rearrangement problems. In *Brazilian Symposium on Bioinformatics (BSB)*, pages 129–140. Springer, 2020.

[23] P. O. Pinheiro, A. O. Alexandrino, A. R. Oliveira, C. C. de Souza, and Z. Dias. Algorithms for the maximum eulerian cycle decomposition problem. In *Simpósio Brasileiro de Pesquisa Operacional (SBPO)*, pages 1–12, 2021.

[24] A. Rahman, S. Shatabda, and M. Hasan. An approximation algorithm for sorting by reversals and transpositions. *Journal of Discrete Algorithms*, 6(3):449–457, 2008.

[25] C. A. Rodger. Graph decomposition. *Le Matematiche*, 45(1):119–140, 1990.

[26] L. A. Wolsey. *Integer Programming*. Wiley Series in Discrete Mathematics and Optimization. Wiley, 1998.

[27] S. Yancopoulos, O. Attie, and R. Friedberg. Efficient sorting of genomic permutations by translocation, inversion and block interchange. *Bioinformatics*, 21(16):3340–3346, 2005.