



UNIVERSIDADE FEDERAL DE SANTA CATARINA  
CENTRO TECNOLÓGICO  
BACHAREL EM CIÊNCIAS DA COMPUTAÇÃO

Tális Breda

**ESTUDO DA FAMÍLIA DE EMPARELHAMENTO EM GRAFOS**

Florianópolis  
2024

Tális Breda

## **ESTUDO DA FAMÍLIA DE EMPARELHAMENTO EM GRAFOS**

Trabalho de Conclusão do Curso de Graduação em Ciências da Computação do Centro Tecnológico da Universidade Federal de Santa Catarina para a obtenção do título de Bacharel em Ciências da Computação.

Orientador: Prof. Dr. Rafael de Santiago

Florianópolis  
2024

# RESUMO

Problemas de emparelhamento em grafos, definidos como uma seleção de conjuntos de arestas sem vértices em comum, possuem ampla relevância em áreas da computação, como visão computacional, e também fora dela, como na biologia. São bastante importantes em tarefas que envolvem correspondência de elementos, como a comparação e correspondência de elementos em conjuntos distintos, por exemplo em análise de dados estruturados. Devido à complexidade computacional frequentemente NP-completa desses problemas, são necessárias abordagens que conciliem precisão e eficiência. Considerando a quantidade de algoritmos e métodos presentes na literatura, torna-se difícil encontrar a maneira mais eficiente de resolver um problema específico. Este trabalho busca levantar, comparar e classificar métodos computacionais, destacando suas aplicações e limitações, com o objetivo de fornecer insights úteis para pesquisadores e profissionais na escolha de soluções eficazes para cenários práticos.

**Palavras-chave:** Emparelhamento em grafos, grafos bipartidos, visão computacional

# ABSTRACT

Graph matching problems, defined as the selection of edge sets with no common vertices, are highly relevant in fields of computer science such as computer vision, as well as in areas outside of it, like biology. They are particularly important for tasks involving element matching, such as comparing and matching elements in distinct sets, for example, in structured data analysis. Due to the often NP-complete computational complexity of these problems, approaches that balance precision and efficiency are necessary. Given the large number of algorithms and methods in the literature, it is challenging to determine the most efficient way to solve a specific problem. This work aims to survey, compare, and classify computational methods, highlighting their applications and limitations, with the goal of providing useful insights for researchers and professionals in selecting effective solutions for practical scenarios.

**Keywords:** Graph matching, bipartite graphs, computer vision

# Conteúdo

<b>1 Introdução</b>	7
1.1 Objetivos	7
1.1.1 Objetivo geral	7
1.1.2 Objetivos específicos	7
1.2 Delimitação do estudo	8
<b>2 Fundamentação teórica</b>	8
2.1 Definições sobre grafos	8
2.1.1 Grafo direcionado	8
2.1.2 Grafo ponderado	8
2.1.3 Grafo bipartido	8
2.2 Busca em grafos	9
2.2.1 Representação computacional	9
2.2.2 Busca em largura	9
2.2.3 Busca em profundidade	9
2.3 Caminhos mais curtos	10
2.3.1 Princípio do relaxamento e subestrutura ótima	10
2.3.2 Algoritmo de Dijkstra	10
2.3.3 Algoritmo de Bellman-Ford	11
2.3.4 Algoritmo de Floyd-Warshall	11
2.4 Redes de fluxo em grafos	12
2.4.1 Definições básicas	12
2.4.2 O problema do fluxo máximo	12
2.4.2.1 Corte	12
2.4.2.2 Redes residuais	13
2.4.2.3 Caminhos aumentantes	13
2.4.3 Fluxo de custo mínimo	13
2.4.3.1 Condições de otimalidade e ciclos negativos	14
2.4.3.2 Potenciais de vértices e custos reduzidos	14
2.5 Problema Geral do Emparelhamento	14
2.5.1 Matching em grafos bipartidos	15
2.5.2 Formulação geral (emparelhamento e otimização)	15
<b>3 Problemas de emparelhamento em grafos bipartidos</b>	15
3.1 Emparelhamento de cardinalidade máxima	15
3.1.1 Descrição do problema	15
3.1.2 Propriedades	15
3.1.2.1 Emparelhamento máximo vs maximal	15
3.1.2.2 Caminhos aumentantes	15
3.1.2.3 Teorema de König	16
3.1.3 Algoritmos exatos	16
3.1.3.1 Algoritmo do caminho aumentante	16
3.1.3.2 Redução ao problema de fluxo máximo	17
3.1.3.3 Algoritmo de Edmonds-Karp	17
3.1.3.4 Algoritmo de Dinic	18
3.1.3.5 Algoritmo de Hopcroft-Karp	19
3.1.4 Heurísticas	21
3.1.4.1 Heurística gulosa simples	21
3.1.4.2 Heurística de Karp-Sipser	21
3.2 Problema de atribuição (Assignment Problem)	21
3.2.1 Descrição do problema	21
3.2.2 Propriedades	22

3.2.3	Algoritmos exatos . . . . .	22
3.2.3.1	Método Húngaro . . . . .	22
3.2.3.2	Jonker-Volgenant . . . . .	24
3.2.3.3	Redução para problema de fluxo de custo mínimo . . . . .	24
3.2.4	Heurísticas . . . . .	25
3.2.4.1	Heurística gulosa ordenada . . . . .	25
3.2.4.2	Algoritmo de leilão com $\epsilon$ -scaling . . . . .	26
3.3	Problema de atribuição gargalo . . . . .	27
3.3.1	Descrição do problema . . . . .	27
3.3.2	Soluções exatas . . . . .	27
3.3.2.1	Métodos de threshold . . . . .	27
3.3.2.2	Método dual . . . . .	28
3.3.2.3	Método de caminhos aumentantes . . . . .	29
3.3.3	Heurísticas . . . . .	31
3.3.3.1	Heurística Construtiva Gulosa . . . . .	31
3.4	Problema do emparelhamento estável . . . . .	31
3.4.1	Descrição do problema . . . . .	31
3.4.2	Propriedades . . . . .	32
3.4.3	Algoritmo de Gale-Shapley . . . . .	32
<b>4</b>	<b>Problemas de emparelhamento em grafos gerais . . . . .</b>	<b>33</b>
4.1	Emparelhamento de cardinalidade máxima . . . . .	33
4.1.1	Descrição . . . . .	33
4.1.2	Propriedades . . . . .	33
4.1.2.1	Ciclos ímpares . . . . .	33
4.1.2.2	Flor (Blossom) . . . . .	34
4.1.3	O algoritmo de Edmonds . . . . .	34
4.1.3.1	Teorema de Tutte-Berge . . . . .	37
4.1.4	Heurísticas . . . . .	37
4.1.4.1	Algoritmo de crescimento de caminho (Path Growing) . . . . .	37
4.2	Emparelhamento ponderado máximo . . . . .	38
4.2.1	Formulação poliédrica e Restrições de Blossom . . . . .	38
4.2.2	A abordagem Primal-Dual . . . . .	38
4.2.3	O algoritmo de Edmonds Ponderado . . . . .	39
4.2.4	Otimizações . . . . .	39
4.2.5	Heurísticas . . . . .	39
4.2.5.1	Algoritmo de Crescimento de Caminho (Path Growing) . . . . .	39
4.3	Emparelhamento induzido máximo . . . . .	40
4.3.1	Propriedades . . . . .	40
4.3.2	Complexidade computacional . . . . .	40
4.3.3	Redução ao Problema do Conjunto Independente . . . . .	40
4.3.4	Abordagens de solução e heurísticas . . . . .	41
4.3.4.1	Heurísticas gulosas . . . . .	41
4.3.4.2	Metaheurísticas . . . . .	41
4.4	Outros problemas . . . . .	41
<b>5</b>	<b>Trabalhos futuros . . . . .</b>	<b>42</b>

# 1 Introdução

Um emparelhamento em um grafo não-dirigido é definido como um conjunto de arestas sem pontas em comum. Em outras palavras, um emparelhamento é um conjunto  $M$  de arestas que satisfaz a seguinte propriedade: o grau de cada vértice no subconjunto  $M$  é no máximo 1 (FEOFILOFF, 2019). Os principais problemas relacionados a emparelhamentos geralmente envolvem grafos bipartidos, aqueles que podem ser particionados em dois conjuntos independentes  $U$  e  $V$ , onde todas as arestas conectam vértices de  $U$  com vértices de  $V$ .

Os problemas de emparelhamento têm aplicações significativas em várias áreas, como biologia computacional, onde podem ser empregados na diferenciação e classificação de estruturas proteicas (TAYLOR, 2002); redes sociais, para identificação de comunidades ou grupos (FAN, 2012); e visão computacional, para correspondências entre elementos de conjuntos distintos, como pontos em imagens ou vértices em malhas tridimensionais (HALLER et al., 2022).

Resolver problemas de emparelhamento em grafos é um desafio devido à sua complexidade computacional, frequentemente NP-completa, exigindo métodos inovadores para equilibrar precisão e eficiência. Estratégias clássicas, como o Problema de Atribuição Quadrática (QAP), modelam o emparelhamento como uma questão de otimização combinatória, empregando relaxações espectrais e heurísticas para encontrar soluções aproximadas de forma eficiente (YAN; YANG; HANCOCK, 2020). Por outro lado, avanços em transporte ótimo duplamente estocástico, como o algoritmo GOAT, têm demonstrado melhorias em robustez e velocidade, especialmente em grafos maiores (SAAD-ELDIN et al., 2021).

Estudos como esses ilustram a importância de investigar e comparar métodos computacionais que abordem a complexidade inerente ao problema de emparelhamento em grafos, contribuindo para aplicações como aprendizado de máquina e visão computacional. Essa abordagem torna-se essencial para desenvolver soluções adaptadas a diferentes cenários e com impacto direto na melhoria de algoritmos existentes.

Diante desse cenário, este trabalho visa sistematizar os principais problemas e métodos computacionais associados ao emparelhamento em grafos. Ele busca oferecer uma comparação detalhada entre suas aplicações e limitações, contribuindo para o entendimento aprofundado das ferramentas disponíveis e identificando oportunidades para o desenvolvimento de novas soluções. Essa abordagem torna-se ainda mais relevante à luz da importância prática desses algoritmos para resolver problemas reais em grande escala, conforme evidenciado por estudos recentes (SUSSMAN; LI, 2021).

## 1.1 Objetivos

### 1.1.1 Objetivo geral

Este trabalho tem como objetivo principal levantar, organizar e comparar problemas de emparelhamento em grafos para diversos casos de uso, além de comparar diferentes métodos computacionais presentes na literatura para a solução desses problemas.

### 1.1.2 Objetivos específicos

- Mapear as variações e características de diferentes problemas de emparelhamento presentes na literatura.
- Investigar e classificar métodos computacionais presentes na literatura de acordo com suas vantagens, limitações e casos de uso.
- Criar um ambiente de testes unificado a fim de reduzir variabilidade nos resultados e proporcionar comparações consistentes
- Implementar e testar os algoritmos descritos, a fim de comparar desempenho e eficiência em diferentes situações
- Organizar os resultados em um formato que facilite o acesso e a compreensão dos tópicos abordados por pesquisadores e profissionais.

## 1.2 Delimitação do estudo

Os problemas de emparelhamento em grafos podem ser classificados de diversas maneiras, como sendo em grafos bipartidos ou não-bipartidos, ponderados ou não-ponderados, direcionados ou não-direcionados, problemas em multigrafos, problemas em hipergrafos, entre outros. Este trabalho focará em problemas que envolvem grafos simples, isto é, excluindo multigrafos e hipergrafos.

# 2 Fundamentação teórica

## 2.1 Definições sobre grafos

Um **grafo**  $G$  é uma estrutura definida por um par  $G = (V, E)$ , consistindo de um conjunto finito  $V$  de elementos chamados **vértices** (ou **nós** ou **pontos**) e um conjunto (ou família)  $E$  de pares não ordenados de vértices, chamados de **arestas** (JUNGNICKEL, 2008). O conjunto  $V$  é o conjunto de vértices de  $G(VG)$  e  $E$  é a família de arestas de  $G(EG)$ .

No contexto deste projeto, assumimos que as arestas  $e = u, v$  são pares não ordenados de vértices distintos. Se uma aresta  $e$  conecta os vértices  $a$  e  $b$ , diz-se que  $a$  e  $b$  são **adjacentes** ou **vizinhos**, e que a aresta  $e$  é **incidente** a  $a$  e  $b$  (SCHRIJVER, 2004).

O conceito de grafo é amplamente utilizado como uma representação abstrata concisa para estruturas complexas e para codificar relações binárias entre um conjunto de objetos.

### 2.1.1 Grafo direcionado

Se as relações entre os vértices forem assimétricas, utilizamos um **grafo direcionado** ou **dígrafo**,  $D = (V, A)$ . Neste caso,  $A$  é uma família de pares ordenados de vértices, chamados **arcos** (ou arestas direcionadas). Para um arco  $e = (u, v)$ ,  $u$  é chamado o vértice inicial ou **cauda** (tail), e  $v$  é o vértice final ou **cabeça** (head) (MANBER, 1989).

### 2.1.2 Grafo ponderado

Um grafo  $G = (V, E)$  é chamado **grafo ponderado** (ou com pesos) se uma **função de peso** (ou função de custo, ou função de comprimento)  $w : E \rightarrow R$  está associada às arestas. Formalmente:  $G = (V, E, w)$ , onde  $w : E \rightarrow R$ . Geralmente, o peso  $w(e)$  de uma aresta  $e$  é um valor real não negativo que representa o custo, comprimento ou capacidade associada àquela aresta (ROSEN; KREHER; STINSON, 1998).

Grafos ponderados podem ser representados por matrizes de adjacência de valor real  $A$ , onde a entrada  $A_{ij}$  é o peso  $w_{ij}$  da aresta (ZASLAVSKIY; BACH; VERT, 2009).

### 2.1.3 Grafo bipartido

Um grafo bipartido é um grafo não direcionado que pode ser facilmente colorido com apenas duas cores (MANBER, 1989).

Formalmente, um grafo  $G = (V, E)$  é chamado **grafo bipartido** se o conjunto de vértices  $V$  puder ser particionado em dois subconjuntos disjuntos  $V_1$  e  $V_2$ , chamados classes de cor, tais que:

$$V = V_1 \cup V_2 \quad e \quad V_1 \cap V_2 = \emptyset \tag{1}$$

e todas as arestas  $e \in E$  conectam um vértice em  $V_1$  a um vértice em  $V_2$ . Ou seja, não existem arestas que conectem vértices dentro do mesmo subconjunto (DASGUPTA; VAZIRANI, 2006).

Observações:

- Um grafo é bipartido se e somente se não contém ciclos de comprimento ímpar (SCHRIJVER, 2004)
- O grafo bipartido completo  $K_{m,n}$  possui  $m$  vértices em  $V_1$  e  $n$  vértices em  $V_2$ , e contém todas as arestas possíveis entre  $V_1$  e  $V_2$  (JUNGNICKEL, 2008)

## 2.2 Busca em grafos

A exploração sistemática dos vértices e arestas de um grafo é uma sub-rotina fundamental para a maioria dos algoritmos em redes, incluindo aqueles utilizados para encontrar emparelhamentos, fluxo máximo e componentes conexos. A busca em grafos consiste em seguir as arestas do grafo para visitar os vértices, permitindo descobrir a estrutura da rede e propriedades de conectividade (CORMEN et al., 2009).

### 2.2.1 Representação computacional

Antes de discutir os algoritmos de busca, é essencial definir como o grafo  $G = (V, E)$  é representado computacionalmente, pois isso impacta diretamente a complexidade temporal das operações. Destacam-se duas representações primárias (AHO; HOPCROFT; ULLMAN, 1974):

- **Matriz de adjacência:** Uma matriz  $A$  de dimensão  $|V| \times |V|$ , onde  $A_{ij} = 1$  se existe uma aresta  $(i, j) \in E$ , e 0 caso contrário. Embora permita verificação de existência de arestas em tempo constante  $O(1)$ , ela consome espaço  $\theta(V^2)$ , sendo ineficiente para grafos esparsos.
- **Matriz de adjacência:** Consiste em um array de  $|V|$  e listas, onde cada lista  $Adj[u]$  contém todos os vértices  $v$  tal que  $(u, v) \in E$ . Esta representação é preferível para grafos esparsos, pois consome espaço  $\theta(V + E)$  (CORMEN et al., 2009).

Para os algoritmos de busca descritos a seguir, assume-se geralmente o uso de listas de adjacência, resultando em complexidades lineares em relação ao tamanho do grafo.

### 2.2.2 Busca em largura

A **busca em largura**, ou Breadth-First Search (BFS) é um dos algoritmos mais simples e fundamentais para a exploração de grafos. A BFS explora o grafo em "camadas" ou níveis. Dado um vértice fonte  $s$ , o algoritmo visita primeiro todos os vizinhos de  $s$  (camada  $L_1$ ), depois os vizinhos desses vizinhos (camada  $L_2$ ), e assim sucessivamente (KLEINBERG; TARDOS, 2005).

O procedimento utiliza uma estrutura de dados do tipo Fila (FIFO) para gerenciar a fronteira de exploração. Cormen et al. (2009) descrevem o uso de um sistema de "cores" para monitorar o estado dos vértices: brancos (não descobertos), cinzas (descobertos, mas com vizinhos ainda não explorados) e pretos (totalmente explorados).

**Propriedades e Complexidade:** A propriedade mais importante da BFS, destacada por Dasgupta e Vazirani (2006), é que ela calcula o caminho mais curto (em número de arestas) de  $s$  a todos os outros vértices alcançáveis em grafos não ponderados. A complexidade de tempo da BFS é  $O(V + E)$ , pois cada vértice é enfileirado no máximo uma vez e cada lista de adjacência é varrida uma única vez.

### 2.2.3 Busca em profundidade

A **busca em profundidade** ou Depth-First Search (DFS), ao contrário da estratégia por níveis da BFS, a busca em profundidade explora as arestas a partir do vértice mais recentemente descoberto que ainda possui arestas inexploradas. Conforme Manber (1989), a DFS "aprofunda-se" no grafo tanto quanto possível e, quando não há mais para onde ir, realiza o backtracking (retrocesso) para explorar outros caminhos.

Computacionalmente, a DFS pode ser implementada de forma recursiva ou utilizando uma Pilha (LIFO). Uma característica crucial da DFS é a classificação temporal dos eventos. Cormen et al. (2009) sugerem registrar dois carimbos de tempo para cada vértice  $u$ :

- $d[u]$ : Momento de descoberta (quando o vértice passa de branco para cinza).
- $f[u]$ : Momento de finalização (quando o vértice passa de cinza para preto, após toda sua lista de adjacência ser explorada).

**Classificação de arestas:** a execução da DFS induz uma estrutura de floresta (Floresta DFS) e permite classificar as arestas do grafo original em quatro tipos, fundamentais para entender ciclos e estrutura (CORMEN et al., 2009)

- **Arestas de Árvore:** Arestas  $(u, v)$  percorridas pela busca quando  $v$  é descoberto.
- **Arestas de Retorno (Back edges):** Conectam um vértice  $u$  a um ancestral  $v$  na árvore DFS. A existência destas arestas indica a presença de ciclos.
- **Arestas de Avanço (Forward edges):** Conectam  $u$  a um descendente  $v$  que não é seu filho direto.
- **Arestas de Cruzamento (Cross edges):** Conectam vértices sem relação de ancestralidade.

A complexidade da DFS, assim como a BFS, é  $\theta(V + E)$  para grafos representados por listas de adjacência.

## 2.3 Caminhos mais curtos

O problema de encontrar o caminho mais curto entre dois vértices em um grafo é um dos problemas de otimização combinatória mais estudados, servindo como sub-rotina para diversas aplicações, incluindo roteamento de redes e algoritmos de fluxo.

Seja  $G = (V, E)$  um grafo ponderado com uma função de peso  $w : E \rightarrow \mathbb{R}$ . O peso de um caminho  $p = \langle v_0, v_1 \dots, v_k \rangle$  é a soma dos pesos de suas arestas constituintes:  $w(p) = \sum_{i=1}^k w(v_{i-1}, v_i)$ . O **caminho mais curto** de um vértice  $u$  a um vértice  $v$  é definido como qualquer caminho  $p$  com peso mínimo  $\delta(u, v)$  (CORMEN et al., 2009).

### 2.3.1 Princípio do relaxamento e subestrutura ótima

Os algoritmos de caminho mínimo baseiam-se na propriedade de **subestrutura ótima**: subcaminhos de um caminho mais curto são, por si mesmos, caminhos mais curtos. Além disso, utilizam a técnica de **relaxamento** (KLEINBERG; TARDOS, 2005).

Para cada vértice  $v$ , o algoritmo mantém um atributo  $d[v]$ , que é um limite superior para o peso do caminho mais curto da fonte  $s$  até  $v$ . O processo de relaxar uma aresta  $(u, v)$  consiste em testar se é possível melhorar o caminho para  $v$  passando por  $u$ :

$$\text{Se } d[v] > d[u] + w(u, v), \text{ então } d[v] \leftarrow d[u] + w(u, v) \quad (2)$$

---

**Algorithm 1:** Método de relaxamento (CORMEN et al., 2009)

---

```

(1) Procedure Relax( $u, v, w$ )
(2)   if  $v.d > u.d + w(u, v)$  then
(3)      $v.d \leftarrow u.d + w(u, v)$ 
(4)      $v.\pi \leftarrow u$ 
(5)   end

```

---

A diferença entre os algoritmos reside na ordem e na frequência com que as arestas são relaxadas.

### 2.3.2 Algoritmo de Dijkstra

Para grafos onde os pesos das arestas são não-negativos ( $w(u, v) \geq 0$ ), o Algoritmo de Dijkstra é a abordagem mais eficiente. Ele utiliza uma estratégia gulosa, mantendo um conjunto de vértices cujos pesos finais dos caminhos mais curtos já foram determinados (AHO; HOPCROFT; ULLMAN, 1974).

O algoritmo seleciona repetidamente o vértice  $u$  com a menor estimativa de caminho mais curto  $d[u]$  de uma fila de prioridade, adiciona-o ao conjunto de visitados e relaxa todas as arestas que saem de  $u$ . Dasgupta e Vazirani (2006) destacam que a eficiência do Dijkstra depende da estrutura de dados utilizada. Com um Heap Binário, a complexidade é  $O((V + E) \log V)$ . Cormen et al. (2009) apontam que, ao utilizar um Heap de Fibonacci, a complexidade amortizada pode ser reduzida para  $O(E + V \log V)$ , o que é crucial para grafos densos.

---

**Algorithm 2:** Algoritmo de Dijkstra (CORMEN et al., 2009)

```
(1) Procedure Dijkstra(G, w, s)
(2)   Initialize-Single-Source(G, s)
(3)   S  $\leftarrow \emptyset$ 
(4)   Q  $\leftarrow G.V$ 
(5)   while Q  $\neq \emptyset$  do
(6)     u  $\leftarrow$  Extract-Min(Q)
(7)     S  $\leftarrow S \cup \{u\}$ 
(8)     foreach vertex v  $\in G.Adj[u]$  do
(9)       | Relax(u, v, w)
(10)      | end
(11)    end
```

---

### 2.3.3 Algoritmo de Bellman-Ford

Quando o grafo contém arestas com pesos negativos, o algoritmo de Dijkstra não garante a corretude. Nesses casos, utiliza-se o Algoritmo de Bellman-Ford. Segundo Manber (1989), este algoritmo baseia-se em programação dinâmica, relaxando todas as arestas do grafo  $|V| - 1$  vezes.

A propriedade fundamental do Bellman-Ford é sua capacidade de detectar ciclos negativos. Se, após  $|V| - 1$  iterações, ainda for possível relaxar alguma aresta, o grafo contém um ciclo com peso total negativo acessível a partir da fonte, o que implica que  $\delta(s, v) = -\infty$  para alguns vértices (KLEINBERG; TARDOS, 2005). Sua complexidade é  $O(VE)$ .

---

**Algorithm 3:** Algoritmo de Bellman-Ford (CORMEN et al., 2009)

```
(1) Procedure Bellman-Ford(G, w, s)
(2)   Initialize-Single-Source(G, s)
(3)   for i = 1 to  $|G.V| - 1$  do
(4)     foreach edge  $(u, v) \in G.E$  do
(5)       | Relax(u, v, w)
(6)     | end
(7)   | end
(8)   foreach edge  $(u, v) \in G.E$  do
(9)     | if v.d > u.d + w(u, v) then
(10)      | | return false
(11)      | | end
(12)    | end
(13)  | return true
```

---

### 2.3.4 Algoritmo de Floyd-Warshall

Enquanto Dijkstra e Bellman-Ford resolvem o problema de fonte única (Single-Source), o algoritmo de Floyd-Warshall resolve o problema de caminhos mais curtos entre todos os pares de vértices (All-Pairs).

Aho, Hopcroft e Ullman (1974) explicam que o algoritmo utiliza programação dinâmica baseada na numeração dos vértices. Seja  $d_{ij}^{(k)}$  o peso do caminho mais curto de *i* a *j* utilizando apenas vértices do conjunto  $1, 2, \dots, k$  como intermediários. A recorrência é dada por:

$$d_{ij}^{(k)} = \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}) \quad (3)$$

A complexidade temporal é  $\theta(V^3)$ . Jungnickel (2008) observa que, para grafos densos, este método é geralmente mais eficiente do que executar o algoritmo de Bellman-Ford a partir de cada vértice.

---

**Algorithm 4:** Algoritmo de Floyd-Warshall (CORMEN et al., 2009)

---

```

(1) Procedure Floyd-Warshall(W)
(2)   n  $\leftarrow$  W.rows
(3)   D(0)  $\leftarrow$  W
(4)   for k = 1 to n do
(5)     let D(k) = (dij(k)) be a new n  $\times$  n matrix
(6)     for i = 1 to n do
(7)       for j = 1 to n do
(8)         dij(k)  $\leftarrow$  min (dij(k-1), dik(k-1) + dkj(k-1))
(9)       end
(10)      end
(11)    end
(12)   return D(n)

```

---

## 2.4 Redes de fluxo em grafos

A teoria dos fluxos em redes é um ramo fundamental da otimização combinatória que modela problemas de transporte de recursos através de um sistema conectado. Esta seção define os conceitos preliminares e apresenta os teoremas centrais que sustentam os algoritmos utilizados neste trabalho.

### 2.4.1 Definições básicas

Uma rede de fluxo é definida por um grafo direcionado  $G = (V, E)$ , onde  $V$  é o conjunto de vértices e  $E$  é o conjunto de arestas. Distinguem-se dois vértices especiais: a **fonte**  $s \in V$ , que produz o fluxo, e o **sorvedouro**  $t \in V$ , que o consome. Para cada aresta  $(u, v) \in E$ , associa-se uma capacidade não-negativa  $c(u, v) \geq 0$ , que limita a quantidade de fluxo que pode atravessar aquela aresta. Um **fluxo** em  $G$  é uma função  $f : E \rightarrow \mathbb{R}^+$  que satisfaz as seguintes propriedades (CORMEN et al., 2009):

- **Restrição de capacidade:** O fluxo em uma aresta não pode exceder a sua capacidade.

$$0 \leq f(u, v) \leq c(u, v), \quad \forall (u, v) \in E \quad (4)$$

- **Conservação de fluxo:** para todo vértice  $v \in V - \{s, t\}$ , a quantidade total de fluxo que entra deve ser igual à que sai

$$\sum_{v \in V} f(v, u) = \sum_{v \in V} f(u, v) \quad (5)$$

### 2.4.2 O problema do fluxo máximo

O **problema do fluxo máximo** consiste em encontrar um fluxo  $f$  tal que  $|f|$  seja o maior possível. Para entender a limitação desse fluxo, precisamos entender os conceitos de corte, redes residuais e caminhos aumentantes.

#### 2.4.2.1 Corte

Um **corte**  $(S, T)$  é uma partição do conjunto de vértices  $V$  em dois subconjuntos disjuntos  $S$  e  $T$ , tal que  $s \in S$  e  $t \in T$ . A **capacidade do corte**, denotada por  $c(S, T)$ , é a soma das capacidades das arestas que partem de  $S$  para  $T$  (CORMEN et al., 2009):

$$c(S, T) = \sum_{u \in S, v \in T} c(u, v) \quad (6)$$

O **corte mínimo** de uma rede  $N$  é um corte cuja capacidade é a menor entre todos os cortes possíveis nessa rede (CORMEN et al., 2009). Isso nos leva ao **teorema do fluxo máximo e corte mínimo (Max-flow Min-cut)**

*O valor máximo do fluxo de uma rede  $N$  é igual à capacidade do corte mínimo em  $N$  (JUNGNICKEL, 2008)*

Este teorema é fundamental para a compreensão e corretudo dos algoritmos de caminhos aumentantes.

#### 2.4.2.2 Redes residuais

A maioria dos algoritmos de resolução de problemas de fluxo máximo se baseia no conceito de redes residuais. Uma **rede residual**  $G_f$  consiste em arestas com capacidades que representam como podemos alterar o fluxo das arestas em  $G$ . A capacidade das arestas da rede residual  $G_f$  corresponde à diferença entre a capacidade da aresta original de  $G$  e o fluxo  $f$  que está passando pela aresta (CORMEN et al., 2009). Ou seja:

$$c_f(u, v) = c(u, v) - f(u, v) \quad (7)$$

Quando a aresta  $(u, v)$  possui fluxo igual à sua capacidade, então  $c_f(u, v) = 0$  e a aresta não é incluída no grafo residual  $G_f$ .

#### 2.4.2.3 Caminhos aumentantes

Dado um grafo  $G = (V, E)$  e um fluxo  $f$ , um **caminho aumentante** é um caminho de  $s$  para  $t$  na rede residual  $G_f$ . Por definição, podemos aumentar o fluxo em uma aresta  $(u, v)$  de um caminho aumentante em até  $c_f(u, v)$  sem violar a capacidade da aresta original em  $G$  (CORMEN et al., 2009).

O método de Ford-Fulkerson, que é base dos principais algoritmos de resolução do problema do fluxo máximo, iterativamente encontra um caminho aumentante, calcula sua capacidade residual e adiciona esse fluxo à solução atual até que não existam mais caminhos aumentantes (AHUJA; MAGNANTI; ORLIN, 1993).

---

##### Algorithm 5: Ford-Fulkerson Algorithm

---

```

(1) Procedure FORD-FULKERSON( $G, s, t$ )
(2)   foreach edge  $(u, v) \in G.E$  do
(3)      $(u, v).f \leftarrow 0$ 
(4)   end
(5)   while there exists a path  $p$  from  $s$  to  $t$  in the residual network  $G_f$  do
(6)      $c_f(p) \leftarrow \min\{c_f(u, v) : (u, v) \text{ is in } p\}$ 
(7)     foreach edge  $(u, v) \in p$  do
(8)       if  $(u, v) \in E$  then
(9)          $(u, v).f \leftarrow (u, v).f + c_f(p)$ 
(10)      else
(11)         $(v, u).f \leftarrow (v, u).f - c_f(p)$ 
(12)      end
(13)    end
(14)  end

```

---

#### 2.4.3 Fluxo de custo mínimo

Uma outra variação dos problemas em redes de fluxo em grafos é o problema do fluxo de custo mínimo. Essa variação considera, além das capacidades disponíveis, uma dimensão econômica. Seja  $G = (V, E)$  uma rede direcionada onde cada aresta  $(u, v)$  possui uma capacidade  $c(u, v)$  e um custo unitário  $w(u, v)$ . O objetivo é transmitir uma quantidade de fluxo pré-determinada  $F$  da fonte  $s$  ao sorvedouro  $t$  com o menor custo total possível.

A formulação linear do problema é dada por (AHUJA; MAGNANTI; ORLIN, 1993). O primeiro objetivo é minimizar o custo total do fluxo, que segue a fórmula:

$$\text{Minimizar o custo do fluxo: } z(f) = \sum_{(u,v) \in A} w(u,v) \cdot f(u,v) \quad (8)$$

A função a seguir assegura que o nó fonte  $s$  gere exatamente a demanda  $F$ , e que o nó sorvedouro  $t$  absorva-a integralmente.

$$\sum_{i,j \in V} f(i,j) - \sum_{k,i \in V} f(k,i) = \begin{cases} F & \text{se } i = s \\ -F & \text{se } j = t \\ 0 & \text{caso contrário} \end{cases} \quad (9)$$

Além desta, é importante lembrar da equação 4, que define que o fluxo em uma aresta não pode exceder a sua capacidade.

#### 2.4.3.1 Condições de otimalidade e ciclos negativos

O critério fundamental para verificar se uma solução é ótima é o **teorema dos ciclos negativos**:

Um fluxo  $f$  é uma solução viável para o problema do fluxo de custo mínimo se e somente se ele satisfaz a condição de otimalidade de ciclo negativo: a rede residual  $G(f)$  não contém um ciclo e custo negativo (AHUJA; MAGNANTI; ORLIN, 1993)

Este teorema é a base dos algoritmos de **cancelamento de ciclos** (Cycle Canceling), que iterativamente identificam ciclos negativos na rede residual e enviam fluxo através deles para reduzir o custo total da função objetivo até que nenhum ciclo negativo reste.

#### 2.4.3.2 Potenciais de vértices e custos reduzidos

Embora o cancelamento de ciclos seja teoricamente sólido, a detecção de ciclos negativos (via Bellman-Ford) pode ser computacionalmente custosa. Para permitir o uso de algoritmos mais eficientes como o de Dijkstra, a literatura introduz o conceito de **potenciais de vértices**, baseando-se na teoria de dualidade da programação linear.

A ideia é manter um **potencial** para cada vértice  $v$  do grafo, chamado de  $h(v)$ , com valor inicial 0. Definimos o **custo reduzido** de uma aresta  $(u,v)$  como  $\gamma'(u,v) = \gamma(u,v) + h(u) - h(v)$ . Se os potenciais forem escolhidos de forma adequada, todos os custos reduzidos podem ser garantidos como não-negativos, permitindo o uso do algoritmo de Dijkstra. Além disso, é garantido que os caminhos de custo mínimo no grafo original e no grafo com custos reduzidos são os mesmos. (JUNGNICKEL, 2008).

O conceito de potenciais e de custo reduzido será utilizado em algoritmos como o de caminhos sucessivos para resolução do problema de atribuição na seção 3.2.3.3.

## 2.5 Problema Geral do Emparelhamento

O problema de **matching** (emparelhamento) em grafos é um problema fundamental na otimização combinatória.

Um **matching**  $M$  em um grafo não-direcionado  $G = (V, E)$  é um subconjunto de arestas  $M \subseteq E$  tal que nenhum par de arestas em  $M$  compartilha um vértice comum (KLEINBERG; TARDOS, 2005). Em outras palavras, cada nó aparece em no máximo uma aresta de  $M$ .

- Um vértice é chamado **coberto** (*matched*) se for incidente a uma aresta em  $M$ ; caso contrário, é **descoberto** (*unmatched* ou *exposed*) (CORMEN et al., 2009)
- Um **matching de cardinalidade máxima** (Maximum Matching) é um matching com o maior número possível de arestas (CORMEN et al., 2009). A cardinalidade máxima de um matching é denotada por  $v(G)$  (SCHRIJVER, 2004)

- Um **matching perfeito** é um matching que cobre todos os vértices do grafo (KLEINBERG; TARDOS, 2005)
- O **problema de matching ponderado** (Weighted Matching Problem) envolve encontrar um matching para o qual a soma dos pesos das arestas é máxima. Em um grafo ponderado  $G = (V, E, w)$ , busca-se um  $M \subseteq E$  que maximize  $w(M)$  (MANDULAK et al., 2024)

### 2.5.1 Matching em grafos bipartidos

O problema de **Matching Bipartido** é o caso clássico de encontrar um matching de cardinalidade máxima em um grafo bipartido  $G = (V, E)$ , onde  $V = X \cup Y$  (KLEINBERG; TARDOS, 2005)

- O matching em grafos bipartidos pode modelar situações de atribuição, como associar empregos ( $X$ ) a máquinas ( $Y$ ), ou professores ( $X$ ) a cursos ( $Y$ ), onde uma aresta indica uma capacidade de atribuição (KLEINBERG; TARDOS, 2005)
- O problema de matching ponderado em grafos bipartidos é equivalente ao **problema de atribuição** (LAWLER, 1976), que historicamente motivou o desenvolvimento do **método Húngaro** (SCHRIJVER, 2004)

### 2.5.2 Formulação geral (emparelhamento e otimização)

Em um contexto mais amplo, o emparelhamento de grafos pode ser formalizado como um problema de otimização que busca maximizar a compatibilidade entre dois grafos  $G$  e  $G'$  (CAETANO et al., 2009).

- O problema de graph matching é frequentemente abordado como um **problema de atribuição quadrática (QAP)**. Essa formulação busca maximizar uma função objetivo que combina termos de compatibilidade unária (nó-nó,  $c_{ii'}$ ) e compatibilidade par a par (aresta-aresta,  $d_{ii'jj'}$ ), sujeito a restrições de atribuição binária ( $y_{ii'} \in \{0, 1\}$ ). O termo quadrático codifica a preservação das relações (arestas) entre os nós (CAETANO et al., 2009)
- Para grafos bipartidos, a determinação de um matching máximo pode ser resolvida de maneira eficiente e está intimamente ligada a problemas de *network flow* (fluxo em redes) (KLEINBERG; TARDOS, 2005)

## 3 Problemas de emparelhamento em grafos bipartidos

### 3.1 Emparelhamento de cardinalidade máxima

#### 3.1.1 Descrição do problema

Dado um grafo não-direcionado  $G = (V, E)$ , o problema de emparelhamento de cardinalidade máxima (MCM) busca encontrar um emparelhamento  $M \subseteq E$  tal que o número de arestas em  $M$  seja maximizado, ou seja,  $|M|$  é o maior entre todos os emparelhamentos possíveis em  $G$ .

Um caso especial, mas importante, do MCM é o **Emparelhamento perfeito**, que é um emparelhamento onde todos os vértices do grafo são cobertos por exatamente uma aresta do emparelhamento. Para que um emparelhamento perfeito exista, o grafo deve ter um número par de vértices e o emparelhamento máximo deve ter cardinalidade igual a  $|V|/2$  (JUNGNICKEL, 2008).

#### 3.1.2 Propriedades

Antes de continuar para os algoritmos e soluções, é necessário entender algumas propriedades importantes

### **3.1.2.1 Emparelhamento máximo vs maximal**

Um emparelhamento máximo é aquele que contém o maior número possível de arestas, enquanto um emparelhamento maximal é aquele que não pode ser aumentado adicionando mais arestas, mas não necessariamente é o maior possível (MANBER, 1989).

### **3.1.2.2 Caminhos aumentantes**

Um caminho aumentante  $P$  em um grafo  $G$  é chamado de caminho aumentante se  $P$  tem tamanho ímpar, começa e termina em vértices livres (não cobertos por nenhuma aresta do emparelhamento atual) e alterna entre arestas que pertencem a  $M$  e arestas que não pertencem a  $M$ . A existência de um caminho aumentante indica que o emparelhamento atual não é máximo (SCHRIJVER, 2004)

### **3.1.2.3 Teorema de Kőnig**

O teorema de Kőnig estabelece uma relação fundamental entre emparelhamentos e coberturas em grafos bipartidos. Ele afirma que, em qualquer grafo bipartido, o tamanho do maior emparelhamento é igual ao tamanho da menor cobertura de vértices (JUNGNICKEL, 2008).

A cobertura de vértices é um conjunto de vértices tal que cada aresta do grafo é incidente a pelo menos um vértice desse conjunto. O teorema de Kőnig é uma ferramenta poderosa para resolver problemas de emparelhamento, utilizado nos algoritmos de Hopcroft-Karp e no método Húngaro.

## **3.1.3 Algoritmos exatos**

Por ser em um ambiente mais restrito, o problema de emparelhamento de cardinalidade máxima tem diversas soluções em tempo polinomial. A seguir listadas algumas delas:

### **3.1.3.1 Algoritmo do caminho aumentante**

O algoritmo de caminho aumentante se baseia em um lema proposto por Claude Berge, que afirma que um grafo  $G$  é maxímo se e somente se não existe nenhum caminho aumentante em  $G$ . Este algoritmo é uma implementação direta deste lema, onde se busca repetidamente por caminhos aumentantes e se aumenta o emparelhamento até que nenhum caminho aumentante possa ser encontrado (LAWLER, 1976).

Começamos com um emparelhamento vazio  $M = \emptyset$ . Escolhemos um vértice livre  $u$  no lado esquerdo do grafo bipartido e tentamos encontrar um caminho aumentante  $P$  que começa em  $u$  e termina em um vértice livre no lado direito do grafo, respeitando a regra da alternância entre arestas em  $M$  e arestas fora de  $M$ . Se um caminho aumentante for encontrado, atualizamos o emparelhamento  $M$  invertendo as arestas ao longo do caminho  $P$ . Repetimos esse processo até que não seja possível encontrar mais caminhos aumentantes. O emparelhamento resultante será o emparelhamento de cardinalidade máxima (HALIM; HALIM, 2013).

Em seguida o algoritmo em pseudo-código:

---

**Algorithm 6:** Algoritmo de caminhos aumentantes (MCBM) (HALIM; HALIM, 2013)

(1) **Global:**  $match$  (vetor de emparelhamento),  $vis$  (vetor de visitados),  $AdjList$

```
(2) Procedure  $Aug(l)$ 
(3)   if  $vis[l] = 1$  then
(4)     return 0
(5)   end
(6)    $vis[l] \leftarrow 1$ 
(7)   for each  $r \in AdjList[l]$  do
(8)     if  $match[r] = -1$  or  $Aug(match[r])$  then
(9)        $match[r] \leftarrow l$ 
(10)      return 1 [comment: found 1 matching]
(11)    end
(12)   end
(13)   return 0 [comment: no matching]

(14) Procedure  $MCBM\_Algorithm(n)$ 
(15)    $MCBM \leftarrow 0$ 
(16)    $match \leftarrow -1$  (initialize for all  $V$ )
(17)   for  $l \leftarrow 0$  to  $n - 1$  do
(18)      $vis \leftarrow 0$  (reset before each recursion)
(19)      $MCBM \leftarrow MCBM + Aug(l)$ 
(20)   end
(21)   return  $MCBM$ 
```

---

Como o algoritmo repete a busca por caminhos aumentantes para cada vértice do lado esquerdo do grafo, o tempo de execução total do algoritmo é  $O(VE)$ , onde  $V$  é o número de vértices e  $E$  é o número de arestas no grafo bipartido (HALIM; HALIM, 2013).

### 3.1.3.2 Redução ao problema de fluxo máximo

O problema de emparelhamento de cardinalidade máxima em grafos bipartidos pode ser eficientemente resolvido através de uma redução ao problema de fluxo máximo em redes. A ideia central é construir uma rede de fluxo a partir do grafo bipartido original, onde cada aresta do grafo bipartido é convertida em uma aresta com capacidade unitária na rede de fluxo. (LAWLER, 1976)

Primeiro, adicionamos um vértice fonte  $s$  e um vértice sumidouro  $t$  à rede. Conectamos o vértice fonte  $s$  a todos os vértices do conjunto esquerdo do grafo bipartido com arestas de capacidade 1. Em seguida, conectamos todos os vértices do conjunto direito do grafo bipartido ao vértice sumidouro  $t$ , também com arestas de capacidade 1. As arestas entre os conjuntos esquerdo e direito do grafo bipartido são mantidas na rede de fluxo, cada uma com capacidade 1 (HALIM; HALIM, 2013).

Com isso, temos agora um grafo de fluxo onde o objetivo é encontrar o fluxo máximo do vértice fonte  $s$  para o vértice sumidouro  $t$ . O valor do fluxo máximo encontrado nesta rede corresponde ao tamanho do emparelhamento máximo no grafo bipartido original. (LAWLER, 1976). Para encontrar o fluxo máximo, podemos utilizar algoritmos clássicos como o de Edmonds-Karp.

### 3.1.3.3 Algoritmo de Edmonds-Karp

O algoritmo de **Edmonds-Karp** é o principal método para resolver esse tipo de problema. Ele é uma especialização do método de Ford-Fulkerson descrito na seção 2.4.2.3 que utiliza uma busca em largura (BFS) para encontrar sempre o caminho aumentante mais curto em número de arestas. Tem complexidade de tempo  $O(VE^2)$ :

---

**Algorithm 7:** Algoritmo de Edmonds-Karp (HALIM; HALIM, 2013)

(1) **Global:**  $res$  (matriz de capacidade residual),  $mf, f, s, t, p$  (vetor de pais da BFS)

```
(2) Procedure Augment( $v, minEdge$ )
(3)   if  $v = s$  then
(4)      $f \leftarrow minEdge$ 
(5)     return
(6)   else
(7)     if  $p[v] \neq -1$  then
(8)       Augment( $p[v], \min(minEdge, res_{p[v], v})$ )
(9)        $res_{p[v], v} \leftarrow res_{p[v], v} - f$ 
(10)       $res_{v, p[v]} \leftarrow res_{v, p[v]} + f$ 
(11)    end
(12)  end

(13) Procedure EdmondsKarp( $s, t$ )
(14)    $mf \leftarrow 0$ 
(15)   while true do
(16)      $f \leftarrow 0$ 
(17)      $dist \leftarrow \infty; dist[s] \leftarrow 0$ 
(18)      $Q \leftarrow \{s\}; p \leftarrow \text{nil (init vector with -1)}$ 
(19)     while  $Q \neq \emptyset$  do
(20)        $u \leftarrow Q.\text{front}(); Q.\text{pop}()$ 
(21)       if  $u = t$  then
(22)         break (stop BFS)
(23)       end
(24)       for each  $v \in V$  do
(25)         if  $res_{u, v} > 0$  and  $dist[v] = \infty$  then
(26)            $dist[v] \leftarrow dist[u] + 1$ 
(27)            $Q.\text{push}(v)$ 
(28)            $p[v] \leftarrow u$ 
(29)         end
(30)       end
(31)     end
(32)     Augment( $t, \infty$ ); [comment: find min edge weight  $f$  in path]
(33)     if  $f = 0$  then
(34)       break (terminate)
(35)     end
(36)      $mf \leftarrow mf + f$ 
(37)   end
(38)   return  $mf$ 
```

---

### 3.1.3.4 Algoritmo de Dinic

Outra opção para resolver o problema de fluxo máximo é o algoritmo de Dinic, que é mais eficiente em muitos casos práticos, especialmente em grafos densos. O algoritmo de Dinic utiliza uma combinação de buscas em largura (BFS) para construir níveis na rede residual e buscas em profundidade (DFS) para encontrar caminhos aumentantes dentro desses níveis. O tempo de execução do algoritmo de Dinic é  $O(E\sqrt{V})$  para grafos gerais, tornando-o uma escolha preferida para muitos problemas de fluxo máximo (HALIM; HALIM, 2013).

Este trabalho não tem uma implementação do algoritmo de Dinic. Ao invés disso, será apresentado o algoritmo de Hopcroft-Karp, que é uma variação especializada do algoritmo de Dinic para o problema de

emparelhamento em grafos bipartidos, oferecendo uma solução mais eficiente para este caso específico.

### 3.1.3.5 Algoritmo de Hopcroft-Karp

O algoritmo de Hopcroft-Karp é um algoritmo eficiente para encontrar o emparelhamento máximo em grafos bipartidos. Ele melhora o desempenho do algoritmo de caminhos aumentantes ao encontrar múltiplos caminhos aumentantes em cada iteração, em vez de apenas um. O tempo de execução do algoritmo de Hopcroft-Karp é  $O(E\sqrt{V})$ , tornando-o significativamente mais rápido do que o algoritmo de caminhos aumentantes simples, especialmente em grafos grandes (HALIM; HALIM, 2013).

O algoritmo consiste em duas fases principais: a fase de construção de níveis e a fase de busca de caminhos aumentantes. Na fase de construção de níveis, uma busca em largura (BFS) é realizada a partir dos vértices livres no lado esquerdo do grafo bipartido para construir uma camada de níveis que ajuda a identificar os caminhos aumentantes mais curtos. Na fase de busca de caminhos aumentantes, uma busca em profundidade (DFS) é realizada para encontrar todos os caminhos aumentantes possíveis dentro da camada de níveis construída na fase anterior. Esses caminhos são então usados para aumentar o emparelhamento (HALIM; HALIM, 2013).

A seguir, o pseudo-código do algoritmo de Hopcroft-Karp:

---

**Algorithm 8:** Algoritmo de Hopcroft-Karp (BURKARD; DELL'AMICO; MARTELLO, 2009)

---

(1) **let**  $G = (U, V; E)$  be a bipartite graph with initial (possibly empty) matching  $M$ ;  
(2) **let**  $U_0$  contain all unmatched vertices of  $U$ ;  
(3) **let**  $V_0$  contain all unmatched vertices of  $V$ ;  
(4) **while**  $U_0 \neq \emptyset$  **do**  
    (5)     Construct\_layered\_graph;  
    (6)     Find\_set $\Delta M$ ;  
    (7)      $M := M \ominus \Delta M$ ;  
    (8)     update sets  $U_0, V_0$  of unmatched vertices  
(9) **end**

(10) **Procedure** Construct\_layered\_graph  
(11)      $L_0 := U_0, k^* := k := 0$ ;  
(12)     **while**  $L_k \neq \emptyset$  **do**  
        (13)         **for each**  $i \in L_k$  **do**  
        (14)              $N(i) := \{j : [i, j] \in E \setminus M, j \notin L_1 \cup L_3 \cup \dots \cup L_{k-1}\}$ ;  
        (15)             **end**  
        (16)              $L_{k+1} := \bigcup_{i \in L_k} N(i)$ ;  
        (17)             **if**  $L_{k+1} = \emptyset$  **then**  
        (18)                 **stop** [**comment:**  $M$  is a maximum matching]  
        (19)             **end**  
        (20)             **if**  $L_{k+1} \cap V_0 \neq \emptyset$  **then**  
        (21)                  $k^* := k + 1$ ;  
        (22)                  $L_{k+2} := \emptyset$   
        (23)             **else**  
        (24)                  $L_{k+2} := \{\bar{i} : [\bar{i}, j] \in M, j \in L_{k+1}\}$ ;  
        (25)             **end**  
        (26)              $k := k + 2$   
        (27)     **end**

(28) **Procedure** Find\_set $\Delta M$   
(29)     **[comment: find a maximal set  $\Delta M$  of disjoint augmenting paths]**  
(30)     delete all vertices in  $L_{k^*} \setminus V_0$ ;  
(31)     mark all remaining vertices as unscanned;  
(32)      $k := 1$ ; **[comment: path counter]**  
(33)     **while**  $L_0 \neq \emptyset$  **do**  
        (34)         choose  $x_0 := i \in L_0$  and delete it in  $L_0$ ;  
        (35)          $\ell := 0$ ;  
        (36)         **while**  $\ell \geq 0$  **do**  
           (37)             **while**  $x_\ell$  has an unscanned neighbor in  $L_{\ell+1}$  **do**  
               (38)                 choose unscanned neighbor  $x_{\ell+1}$ ;  
               (39)                 mark  $x_{\ell+1}$  as scanned;  
               (40)                  $\ell := \ell + 1$ ;  
               (41)                 **if**  $\ell = k^*$  **then**  
                   (42)                      $P_k := (x_0, x_1, \dots, x_{k^*})$ ;  
                   (43)                      $k := k + 1$   
               (44)                 **end**  
           (45)             **end**  
           (46)             **if**  $\ell < k^*$  **then**  
               (47)                  $\ell := \ell - 1$   
           (48)             **else**  
               (49)                  $\ell := -1$   
           (50)             **end**  
           (51)         **end**  
        (52)     **end**  
        (53)     **return**  $\Delta M := (P_1, P_2, \dots, P_{k-1})$

---

### 3.1.4 Heurísticas

Por mais que o algoritmo de Hopcroft-Karp descrito em 8 seja considerado relativamente eficiente, com complexidade  $O(E\sqrt{V})$ , ele ainda não é tão viável para instâncias muito grandes do problema de emparelhamento de cardinalidade máxima. Por isso, foram propostas alguns métodos aproximados (heurísticas) com o objetivo de resolver o problema com complexidade linear  $O(E)$ .

#### 3.1.4.1 Heurística gulosa simples

Um método que itera sobre as arestas do grafo em ordem arbitrária. Se a aresta  $(u, v)$  conecta dois vértices livres, adicione essa aresta ao emparelhamento e remova os vértices  $u$  e  $v$ . Tem complexidade de tempo linear  $O(E)$  e garante um emparelhamento maximal, que conforme visto na seção 3.1.2.1, é um emparelhamento ao qual não é possível adicionar mais arestas, mas que não é necessariamente o de maior cardinalidade. A solução também garante um emparelhamento de pelo menos  $1/2$  (metade) da cardinalidade máxima (VAZIRANI, 2001).

#### 3.1.4.2 Heurística de Karp-Sipser

Heurística descrita por Karp e Sipser (1981) para grafos esparsos que toma decisões com base na topologia do grafo ao invés de decisões aleatórias, tornando essas decisões mais seguras.

O algoritmo funciona reduzindo dinamicamente o grafo  $G = (V, E)$  através da aplicação repetida de duas regras, nesta ordem de prioridade:

1. **Redução de folhas de grau 1:** Se existir algum vértice  $v$  no grafo atual com **grau 1** (apenas um vizinho  $u$ ), a aresta  $(u, v)$  é selecionada imediatamente para o emparelhamento. Isso é seguro pois, se  $u$  é o único vizinho de  $v$ , ele também é única opção para emparelhar  $v$ . Se isso não for feito,  $v$  ficará isolado.
2. **Escolha Aleatória:** Se **não** existirem vértices de grau 1 (ou seja, todos os vértices têm grau  $\geq 2$ ), o algoritmo escolhe uma aresta  $(u, v)$  aleatoriamente, adiciona ao emparelhamento e remove  $u$  e  $v$  do grafo.

Em grafos esparsos, isso tende a quebrar ciclos e formar novas folhas (vértices de grau 1), permitindo que o algoritmo volte a usar a regra 1. Porém, em grafos densos, essa escolha aleatória pode comprometer a precisão da solução.

Para grafos esparsos, a regra 2 é raramente aplicada, pois a remoção de vértices na regra 1 frequentemente cria novas folhas, provocando um efeito cascata, fazendo com que a maior parte do grafo seja resolvida de maneira ótima. A complexidade do algoritmo é linear ( $O(E)$ ).

## 3.2 Problema de atribuição (Assignment Problem)

### 3.2.1 Descrição do problema

O problema de atribuição, também conhecido como problema do emparelhamento ponderado, consiste em encontrar uma combinação ótima de atribuições entre dois conjuntos disjuntos, minimizando o custo total associado a essas atribuições. Exemplo: Considere  $N$  trabalhadores e  $N$  tarefas, onde cada trabalhador pode ser designado a exatamente uma tarefa, e cada tarefa deve ser atribuída a exatamente um trabalhador. O custo de atribuir o trabalhador  $i$  à tarefa  $j$  é representado por uma matriz de custos  $W = [w_{ij}]$  (LAWLER, 1976). O objetivo é encontrar um conjunto de atribuições que minimize o custo total.

### **3.2.2 Propriedades**

### **3.2.3 Algoritmos exatos**

#### **3.2.3.1 Método Húngaro**

O método Húngaro é um algoritmo eficiente para resolver o problema de atribuição em tempo polinomial. Ele foi desenvolvido por Harold Kuhn em 1955 e é baseado no trabalho de Dénes König e Jenő Egerváry sobre emparelhamentos em grafos bipartidos (JUNGNICKEL, 2008).

O método se baseia em manter um conjunto de potenciais para os vértices dos dois conjuntos do grafo bipartido, sendo  $u_i$  o potencial do vértice  $i$  no conjunto esquerdo e  $v_j$  o potencial do vértice  $j$  no conjunto direito. A regra que o algoritmo mantém é que para cada aresta  $(i, j)$ , a soma dos potenciais deve ser maior ou igual ao custo da aresta, ou seja,  $u_i + v_j \geq w_{ij}$ .

Cria-se um subgrafo de igualdade  $H_{u,v}$  contendo apenas as arestas onde a soma dos potenciais é igual ao custo da aresta, ou seja,  $u_i + v_j = w_{ij}$ . O algoritmo então tenta encontrar um emparelhamento máximo neste subgrafo de igualdade. Se o emparelhamento encontrado cobre todos os vértices do conjunto esquerdo, então ele é ótimo para o problema de atribuição original. Caso contrário, o algoritmo ajusta os potenciais para criar novas arestas de igualdade e repete o processo até encontrar um emparelhamento máximo que cubra todos os vértices do conjunto esquerdo (JUNGNICKEL, 2008).

Em seguida o pseudo-código do método Húngaro:

---

**Algorithm 9:** Algoritmo Húngaro (JUNGNICKEL, 2008)

---

```

(1) Procedure HUNGARIAN( $n, w; mate$ )
(2)   for  $v \in V$  do  $mate(v) \leftarrow 0$ 
(3)   for  $i = 1$  to  $n$  do  $u_i \leftarrow \max\{w_{ij} : j = 1, \dots, n\}; v_i \leftarrow 0$ 
(4)    $nrex \leftarrow n$ 
(5)   while  $nrex \neq 0$  do
(6)     for  $i = 1$  to  $n$  do  $m(i) \leftarrow \text{false}; p(i) \leftarrow 0; \delta_i \leftarrow \infty$ 
(7)      $aug \leftarrow \text{false}; Q \leftarrow \{i \in S : mate(i) = 0\}$ 
(8)     repeat
(9)       remove an arbitrary vertex  $i$  from  $Q$ ;  $m(i) \leftarrow \text{true}; j \leftarrow 1$ 
(10)      while  $aug = \text{false}$  and  $j \leq n$  do
(11)        if  $mate(i) \neq j'$  then
(12)          if  $u_i + v_j - w_{ij} < \delta_j$  then
(13)             $\delta_j \leftarrow u_i + v_j - w_{ij}; p(j) \leftarrow i$ 
(14)            if  $\delta_j = 0$  then
(15)              if  $mate(j') = 0$  then
(16)                AUGMENT( $mate, p, j'; mate$ )
(17)                 $aug \leftarrow \text{true}; nrex \leftarrow nrex - 1$ 
(18)              else
(19)                 $Q \leftarrow Q \cup \{\text{mate}(j')\}$ 
(20)              end
(21)            end
(22)          end
(23)        end
(24)         $j \leftarrow j + 1$ 
(25)      end
(26)      if  $aug = \text{false}$  and  $Q = \emptyset$  then
(27)         $J \leftarrow \{i \in S : m(i) = \text{true}\}; K \leftarrow \{j' \in T : \delta_j = 0\}$ 
(28)         $\delta \leftarrow \min\{\delta_j : j' \in T \setminus K\}$ 
(29)        for  $i \in J$  do  $u_i \leftarrow u_i - \delta$ 
(30)        for  $j' \in K$  do  $v_j \leftarrow v_j + \delta$ 
(31)        for  $j' \in T \setminus K$  do  $\delta_j \leftarrow \delta_j - \delta$ 
(32)         $X \leftarrow \{j' \in T \setminus K : \delta_j = 0\}$ 
(33)        if  $mate(j') \neq 0$  for all  $j' \in X$  then
(34)          for  $j' \in X$  do  $Q \leftarrow Q \cup \{\text{mate}(j')\}$ 
(35)        else
(36)          choose  $j' \in X$  with  $mate(j') = 0$ 
(37)          AUGMENT( $mate, p, j'; mate$ )
(38)           $aug \leftarrow \text{true}; nrex \leftarrow nrex - 1$ 
(39)        end
(40)      end
(41)    until  $aug = \text{true}$ 
(42)  end

```

---

**Algorithm 10:** Método AUGMENT (JUNGNICKEL, 2008)

---

```

(1) Procedure AUGMENT( $mate, p, j'; mate$ )
(2)   repeat
(3)      $i \leftarrow p(j); mate(j') \leftarrow i; next \leftarrow mate(i); mate(i) \leftarrow j'$ 
(4)     if  $next \neq 0$  then
(5)        $j' \leftarrow next$ 
(6)     end
(7)   until  $next = 0$ 

```

---

### 3.2.3.2 Jonker-Volgenant

O algoritmo de Jonker-Volgenant é uma evolução otimizada para a resolução do problema de atribuição linear, proposto por Roy Jonker e Anton Volgenant em 1987. Ele foi desenvolvido para superar o desempenho prático do método Húngaro, especialmente em grafos densos, combinando a teoria de dualidade com estratégias de inicialização e busca mais eficientes (JONKER; VOLGENANT, 1987).

Assim como o método Húngaro, este algoritmo baseia-se na manutenção de potenciais duais ( $u_i$  e  $v_j$ ) e respeita as condições de folga complementar. No entanto, sua principal distinção conceitual é o uso da estratégia de Caminho Aumentante Mais Curto (Shortest Augmenting Path). Ao invés de construir emparelhamentos máximos em fases distintas, o algoritmo foca em encontrar o caminho de custo mínimo que conecta uma linha não atribuída a uma coluna livre.

O processo inicia com heurísticas avançadas (conhecidas como redução de colunas e transferência de redução) para resolver rapidamente as atribuições triviais. Para os vértices restantes, o algoritmo realiza uma busca similar ao algoritmo de Dijkstra: ele explora o grafo para encontrar o caminho aumentante mais curto, atualizando os potenciais duais simultaneamente durante a busca. Isso garante que, a cada iteração, o emparelhamento seja aumentado pelo menor custo possível até que a solução ótima completa seja atingida.

A implementação deste algoritmo é bastante complexa, e portanto não será detalhada aqui. No entanto, sua eficiência prática o torna uma escolha preferida para muitos problemas de atribuição em aplicações reais, especialmente quando comparado ao método Húngaro tradicional (JUNGNICKEL, 2008).

### 3.2.3.3 Redução para problema de fluxo de custo mínimo

Similarmente ao que foi feito para o problema de emparelhamento de cardinalidade máxima, o problema de atribuição pode ser resolvido através de uma redução ao problema de fluxo de custo mínimo. Assim como para o problema anterior, essa redução é feita criando dois novos vértices: um vértice fonte  $s$  e um vértice sorvedouro  $t$ . Em seguida, são adicionadas arestas do vértice fonte  $s$  para cada vértice de um dos conjuntos do grafo bipartido, com capacidade 1 e custo 0. Também são adicionadas arestas de cada vértice do outro conjunto para o vértice sorvedouro  $t$ , também com capacidade 1 e custo 0. Finalmente, As arestas do grafo bipartido original são adicionadas com capacidade 1 e custo igual ao custo de atribuição correspondente.

Para resolver problemas de fluxo podemos utilizar algoritmos como o de caminhos de custo mínimo sucessivos (Successive Shortest Paths) ou o algoritmo de cancelamento de ciclos (Cycle canceling) (SCHRIJVER, 2004).

O primeiro algoritmo, o de caminhos de custo mínimo sucessivos, foi inicialmente proposto por Jewell, Busacker e Gowen e é uma adaptação do algoritmo de Ford-Fulkerson para incorporar o custo das arestas. Em poucas palavras, ele encontra o caminho de  $s$  para  $t$  de custo mínimo, envia o máximo fluxo possível ao longo desse caminho, e repete esse processo até que não seja mais possível aumentar o fluxo sem violar as capacidades das arestas (SCHRIJVER, 2004).

**Algorithm 11:** Algoritmo de caminhos de custo mínimo sucessivos (JUNGNICKEL, 2008)

---

```

(1) Procedure OPTFLOW( $G, c, s, t, \gamma, v; f, sol$ )
(2)   for  $e \in E$  do  $f(e) \leftarrow 0$ 
(3)    $sol \leftarrow true$ ,  $val \leftarrow 0$ 
(4)   while  $sol = true$  and  $val < v$  do
(5)     construct the auxiliary network  $N' = (G', c', s, t)$  with cost function  $\gamma'$ 
(6)     if  $t$  is not accessible from  $s$  in  $G'$  then
(7)       |  $sol \leftarrow false$ 
(8)     else
(9)       | determine a shortest path  $P$  from  $s$  to  $t$  in  $(G', \gamma')$ 
(10)      |  $\delta \leftarrow \min\{c'(e) : e \in P\}$ ;  $\delta' \leftarrow \min(\delta, v - val)$ ;  $val \leftarrow val + \delta'$ 
(11)      | augment  $f$  along  $P$  by  $\delta'$ 
(12)    end
(13)  end

```

---

A complexidade do algoritmo é dependente do método utilizado para encontrar o caminho  $P$ . A implementação padrão utilizaria o algoritmo de Dijkstra, que tem uma complexidade de  $O(m + n \log n)$

em uma implementação com filas de prioridade, onde  $n$  é o número de vértices e  $m$  o número de arestas. Porém, como o grafo residual pode conter arestas com custos negativos, essa implementação não funcionaria corretamente. Para contornar esse problema, temos duas opções principais:

- **Utilizar Bellman-Ford:** A solução mais simples e direta é utilizar, ao invés do algoritmo de Dijkstra, o algoritmo de Bellman-Ford para encontrar o caminho de custo mínimo  $P$ . O Bellman-Ford é mais lento, com uma complexidade de  $O(nm)$ , mas lida corretamente com arestas de custo negativo.
- **Utilizar a técnica de Potenciais:** Permite que utilizemos o algoritmo de Dijkstra mesmo na presença de arestas de custo negativo.

A cada iteração do loop, encontra-se o caminho de custo mínimo  $P$  utilizando Dijkstra com os custos reduzidos. Após encontrar o caminho, os potenciais são atualizados para todos os vértices  $v$  alcançáveis a partir de  $s$  no grafo residual, ajustando-os conforme a distância mínima encontrada pelo Dijkstra, de acordo com a fórmula  $h(v) \leftarrow h(v) + d(s, v)$ , onde  $d(s, v)$  é a distância mínima de  $s$  até  $v$  no grafo residual com custos reduzidos.

Abaixo uma adaptação do algoritmo 11 utilizando a técnica de potenciais:

---

**Algorithm 12:** Adaptação do algoritmo 11 utilizando Dijkstra com técnica de potenciais

---

```

(1) Procedure OPTFLOW_DIJKSTRA( $G, c, s, t, \gamma, v; f, sol$ )
(2)   for  $e \in E$  do  $f(e) \leftarrow 0$ 
(3)   for  $x \in V$  do  $\pi(x) \leftarrow 0$ 
(4)    $sol \leftarrow true, val \leftarrow 0$ 
(5)   while  $sol = true$  and  $val < v$  do
(6)     construct the auxiliary network  $N' = (G', c', s, t)$ 
(7)     for  $e = (u, v) \in E(G')$  do
(8)       | define reduced cost:  $w(e) \leftarrow \gamma(e) + \pi(u) - \pi(v)$ 
(9)       | end
(10)      Run Dijkstra on  $N'$  using weights  $w$  to find shortest path distances  $d(\cdot)$  from  $s$ 
(11)      if  $t$  is not accessible from  $s$  (i.e.,  $d(t) = \infty$ ) then
(12)        |  $sol \leftarrow false$ 
(13)      else
(14)        | for  $x \in V$  such that  $d(x) < \infty$  do  $\pi(x) \leftarrow \pi(x) + d(x)$ 
(15)        | Identify shortest path  $P$  from  $s$  to  $t$  based on  $d(\cdot)$ 
(16)        |  $\delta \leftarrow \min\{c'(e) : e \in P\}; \delta' \leftarrow \min(\delta, v - val); val \leftarrow val + \delta'$ 
(17)        | augment  $f$  along  $P$  by  $\delta'$ 
(18)      end
(19)    end

```

---

A complexidade do algoritmo utilizando Dijkstra com a técnica de potenciais é  $O(n(m + n \log n))$ , onde  $n$  é o número de vértices e  $m$  o número de arestas do grafo. Isso ocorre porque, em cada iteração do loop principal, executamos o algoritmo de Dijkstra, que tem complexidade  $O(m + n \log n)$ , e o número máximo de iterações é limitado por  $n$ , o número de vértices no grafo bipartido.

### 3.2.4 Heurísticas

Assim como no problema de emparelhamento de cardinalidade máxima, as soluções exatas, apesar de apresentarem tempo polinomial, são inviáveis para instâncias muito grandes. A seguir algumas heurísticas que objetivam resolver o problema de atribuição em tempo até mesmo linear ( $O(E)$ ).

#### 3.2.4.1 Heurística gulosa ordenada

Similar à heurística gulosa simples descrita na seção 3.1.4.1, com a diferença de que a ordem das arestas não é mais arbitrária, e sim decrescente por peso. Tem a mesma garantia de metade do peso ótimo, mas tem complexidade  $O(E \log E)$  por conta da ordenação.

### 3.2.4.2 Algoritmo de leilão com $\epsilon$ -scaling

O Algoritmo de Leilão (**Auction Algorithm**), descrito em Bertsekas et al. (1992) e Bertsekas (1988), resolve o problema da atribuição maximizando o benefício total através de um processo iterativo de lances e aumentos de preços, análogo a um leilão do mundo real.

Considere  $N$  pessoas e  $N$  objetos.

- $a_{ij}$ : O valor (benefício) que a pessoa  $i$  atribui ao objeto  $j$ .
- $p_j$ : O preço atual do objeto  $j$ .

O objetivo de cada pessoa  $i$  é maximizar seu próprio **lucro líquido**:

$$\text{Lucro}_{ij} = a_{ij} - p_j \quad (10)$$

Em um equilíbrio de mercado (solução ótima), cada pessoa deve estar designada a um objeto que maximize seu lucro pessoal, e os preços devem ser tais que todos estejam satisfeitos. Isso corresponde, na teoria de dualidade, à condição de **Folga Complementar**.

Se exigirmos que cada pessoa pegue "estritamente" o melhor objeto, o algoritmo pode entrar em um loop infinito de "guerra de preços" com incrementos nulos ou infinitesimais. Para garantir a convergência, Bertsekas introduziu o conceito de  $\epsilon$ -CS (Folga Complementar Epsilon).

Uma atribuição é  $\epsilon$ -ótima se cada pessoa  $i$  designada ao objeto  $j$  estiver a uma distância  $\epsilon$  de seu lucro máximo:

$$(a_{ij} - p_j) \geq \max_k (a_{ik} - p_k) - \epsilon \quad (11)$$

O Passo do Leilão:

1. Uma pessoa não designada  $i$  escolhe o objeto  $j^*$  que oferece o maior lucro.
2. Ela calcula a diferença entre o lucro do melhor objeto e o lucro do segundo melhor objeto ( $\gamma$ ).
3. Ela faz um lance para o objeto  $j^*$ , aumentando seu preço  $p_{j^*}$  em:

$$\Delta p = \gamma + \epsilon$$

4. A pessoa  $i$  ganha o objeto, e quem estava com ele (se houvesse alguém) torna-se não designado e volta à fila para dar lances.

O termo  $\epsilon$  é crucial: ele garante que o preço suba pelo menos um valor fixo a cada lance, impedindo loops infinitos e forçando o algoritmo a terminar. A escolha do valor de  $\epsilon$  é muito importante pois define a eficiência do algoritmo:

- Se  $\epsilon$  for **grande**: O algoritmo termina muito rápido (poucos lances para resolver conflitos), mas a solução é de baixa qualidade (distante do ótimo).
- Se  $\epsilon$  for **pequeno**: A solução é muito próxima ou igual à ótima, mas o algoritmo é lento (muitos incrementos pequenos de preço necessários).

O conceito de  $\epsilon$ -scaling consiste em um refinamento sucessivo do valor de  $\epsilon$ :

1. Comece com um  $\epsilon$  grande (ex: metade do maior peso da aresta).
2. Rode o leilão até convergir (todos emparelhados).
3. Reduza o valor de  $\epsilon$  (ex:  $\epsilon \leftarrow \epsilon/4$ ).
4. Mantenha os preços ( $p_j$ ) e a atribuição da fase anterior. Reinicie o leilão com o novo  $\epsilon$ .

Ao reutilizar os preços, as fases subsequentes convergem muito mais rápido do que se começassem do zero, pois os preços já estão "quase" certos. O processo repete-se até que  $\epsilon < 1/N$ , momento em que a solução obtida é comprovadamente a ótima inteira (para pesos inteiros).

A complexidade do algoritmo (usando  $\epsilon$ -scaling) é de  $O(N \cdot E \log(C))$ , onde  $C$  é o maior custo da aresta. Uma melhoria significativa dos algoritmos  $O(N^3)$ , como o Método Húngaro. Além disso, o algoritmo é paralelizável, pois permite que as pessoas façam lances simultaneamente, sendo ideal para aplicações onde é possível utilizar GPUs ou clusters distribuídos para o processamento.

### 3.3 Problema de atribuição gargalo

#### 3.3.1 Descrição do problema

O problema de atribuição gargalo, também conhecido como **Problema de Emparelhamento Min-Max** ou **Bottleneck Assignment problem**, é uma variação do problema de atribuição tradicional. O objetivo deste problema é encontrar, em um grafo bipartido ponderado, um emparelhamento de cardinalidade máxima (maior número possível de arestas) no qual o mínimo dos pesos das arestas do emparelhamento seja maximizado (LAWLER, 1976).

Por exemplo. Considere  $N$  trabalhadores e  $N$  estações de trabalho.  $w_{ij}$  representa a eficiência do trabalhador  $i$  na estação de trabalho  $j$ . A eficiência total da produção é limitada pela eficiência do trabalhador menos eficiente. O objetivo é atribuir trabalhadores às estações de trabalho de forma a maximizar a eficiência mínima entre todas as atribuições. (LAWLER, 1976).

#### 3.3.2 Soluções exatas

##### 3.3.2.1 Métodos de threshold

Métodos de threshold funcionam alternando entre duas fases. A primeira fase consiste em selecionar um valor limite, chamando de **threshold**, descrito por  $c^*$ . Na segunda fase, constrói-se um grafo bipartido com todas as arestas cujo peso é menor ou igual a  $c^*$ . Em seguida, verifica-se se existe um emparelhamento perfeito nesse grafo. O menor valor de  $c^*$  que permite um emparelhamento perfeito é a solução ótima para o problema de atribuição gargalo (BURKARD; DELL'AMICO; MARTELLO, 2009).

Para selecionar o valor limite  $c^*$ , podemos utilizar uma busca binária entre o menor e o maior peso das arestas do grafo original. A cada iteração, ajustamos o valor de  $c^*$  com base na existência ou não de um emparelhamento perfeito no grafo construído. Esse processo continua até que o valor ótimo seja encontrado. Burkard, Dell'Amico e Martello (2009), que descreve esse método, utiliza um algoritmo baseado em matrizes para decidir se um emparelhamento é perfeito ou não, porém podemos utilizar qualquer algoritmo clássico de emparelhamento em grafos bipartidos, como os definidos na seção 3.1.3, como por exemplo o algoritmo de Hopcroft-Karp. Também segundo o autor, o algoritmo tem uma complexidade de tempo de  $O(n^{2.5} / \sqrt{\log n})$ .

Lawler (1976) menciona um método de threshold para solução do problema de atribuição gargalo, porém a definição de threshold dele é levemente diferente, referindo-se ao valor dinâmico do rótulo de um vértice durante busca de caminhos. O algoritmo que ele descreve se assemelha mais aos algoritmos de caminhos aumentantes, que serão discutidos na seção 3.3.2.3.

---

**Algorithm 13:** Threshold algorithm for the LBAP (BURKARD; DELL'AMICO; MARTELLO, 2009)

---

```

(1) let  $C = (c_{ij})$  be a given  $n \times n$  cost matrix;
(2)  $c_0^* := \min_{ij} \{c_{ij}\}$ ,  $c_1^* := \max_{ij} \{c_{ij}\}$ 
(3) if  $c_0^* = c_1^*$  then
(4)   |  $z := c_0^*$  [comment: any permutation of  $\{1, 2, \dots, n\}$  is optimal]
(5) else
(6)   | while  $C^* = \{c_{ij} : c_0^* < c_{ij} < c_1^*\} \neq \emptyset$  do
(7)     |   [comment: find the median,  $c^*$ , of  $C^*$ ]
(8)     |    $c^* := \min\{c \in C^* : |\{c_{ij} \in C^* : c_{ij} \leq c\}| \geq |C^*|/2\}$ ;
(9)     |   Feasibility_check( $c^*, c_0^*, c_1^*$ )
(10)    | end
(11)   | if Feasibility_check has not yet been executed for  $c_0^*$  then
(12)     |   | Feasibility_check( $c_0^*, c_0^*, c_1^*$ )
(13)   | end
(14)   | find a perfect matching in the current bipartite graph  $G[c_1^*]$ 
(15)   |  $z := c_1^*$  [comment: value of the optimal matching]
(16) end

(17) Procedure Feasibility_check( $c^*, c_0^*, c_1^*$ )
(18)   define the current graph  $G[c^*]$ 
(19)   if  $G[c^*]$  contains a perfect matching then
(20)     |   |  $c_1^* := c^*$ 
(21)   else
(22)     |   |  $c_0^* := c^*$ 
(23)   end

```

---

### 3.3.2.2 Método dual

Assim como os métodos de threshold, o método dual também define um valor limite  $c^*$ . No entanto, ao invés de 'chutar' um valor usando busca binária, este método define o valor inicial de  $c^*$  como sendo o menor valor possível que ele pode assumir. Olhando para a matriz de custos, esse valor é o máximo entre os mínimos de cada linha e cada coluna da matriz:

$$c^* = \max \left\{ \max_i (\min_j c_{ik}), \max_j (\min_i c_{kj}) \right\} \quad (12)$$

Com o limite definido, verificamos se é possível encontrar um emparelhamento perfeito no grafo bipartido formado pelas arestas cujo peso é menor ou igual a  $c^*$ , usando os mesmos algoritmos de antes. Se um emparelhamento perfeito for encontrado, então  $c^*$  é a solução ótima. Caso contrário, é necessário aumentar o valor de  $c^*$ , e para isso nos baseamos no teorema de König, definido na seção 3.1.2.3.

Sabemos que todas as arestas  $c_{ij} > c^*$  estão conectadas a um conjunto de linhas  $I$  e um conjunto de colunas  $J$ . Essas linhas e colunas "cobrem" todas as arestas do grafo atual. Para aumentar o valor de  $c^*$ , precisamos adicionar uma nova aresta que não esteja coberta por esses conjuntos. Procuramos o menor peso entre as arestas não cobertas, ou seja, aquelas que não pertencem às linhas em  $I$  nem às colunas em  $J$ . Esse valor se torna o novo  $c^*$ :

$$Novoc^* = \min_{i \notin I, j \notin J} c_{ij} \quad (13)$$

A partir daqui, o processo se repete. Com o novo valor de  $c^*$  e a nova aresta adicionada ao grafo, verificamos novamente a existência de um emparelhamento perfeito. Esse ciclo continua até que um emparelhamento perfeito seja encontrado, momento em que o valor atual de  $c^*$  será a solução ótima para o problema de atribuição gargalo.

---

**Algorithm 14:** Dual\_LBAP. Dual algorithm for the LBAP (BURKARD; DELL'AMICO; MARTELLO, 2009).

---

```

(1) let  $C = (c_{ij})$  be a given  $n \times n$  cost matrix;
(2)  $c^* := \max_{k=1,2,\dots,n} (\min_{i=1,2,\dots,n} c_{ik}, \min_{j=1,2,\dots,n} c_{kj})$ 
(3)  $M := \emptyset$ 
(4) while  $|M| < n$  do
(5)   define the bipartite graph  $G[c^*]$ 
(6)   find a maximum matching  $M$  in  $G[c^*]$ 
(7)   if  $|M| < n$  then
(8)     let  $I \subseteq U$  and  $J \subseteq V$  be vertex sets in a minimum vertex cover of  $G[c^*]$ 
(9)      $c^* := \min_{i \notin I, j \notin J} c_{ij}$ 
(10)    end
(11)  end

```

---

### 3.3.2.3 Método de caminhos aumentantes

Ao contrário do método dual, que ajusta um limiar global para todo o grafo, o método de caminhos aumentantes é construtivo. Ele inicia com um emparelhamento vazio e busca aumentar sua cardinalidade iterativamente, adicionando um par de vértices de cada vez. O objetivo central em cada passo é encontrar um caminho aumentante  $P$  que conecte uma linha livre a uma coluna livre, de tal forma que a capacidade desse caminho seja mínima. A capacidade aqui é definida pelo peso da aresta mais cara contida no caminho:

$$cap(P) = \max_{(i,j) \in P} c_{ij} \quad (14)$$

Para encontrar tal caminho, utiliza-se uma adaptação do algoritmo de Dijkstra. Em vez de somar os pesos das arestas, mantemos para cada vértice um valor de rótulo que representa o "gargalo" mínimo necessário para alcançá-lo. Durante a busca, ao tentar estender o caminho de um nó  $u$  (com rótulo  $d_u$ ) para um vizinho  $v$  através da aresta  $(u,v)$ , o custo não é somado, mas sim comparado. O novo custo potencial para atingir  $v$  é o máximo entre o gargalo já existente até  $u$  e o peso da nova aresta:

$$NovoCusto_v = \max(d_u, c_{uv}) \quad (15)$$

O algoritmo seleciona, a cada passo, o nó não visitado com o menor rótulo atual, garantindo uma exploração gulosa que minimiza o gargalo. Assim que uma coluna livre é alcançada por essa busca, o caminho aumentante é identificado e o emparelhamento é expandido, invertendo-se as arestas ao longo do caminho. Esse processo se repete (reiniciando as distâncias) até que um emparelhamento perfeito seja obtido. A solução do problema será determinada pela maior capacidade encontrada entre todos os caminhos aumentantes utilizados na construção.

---

**Algorithm 15:** LBAP usando caminhos aumentantes (BURKARD; DELL'AMICO; MARTELLO, 2009)

---

(1) let  $G = (U, V; E)$  be a bipartite graph with edge lengths  $c_{ij}$  for  $[i, j] \in E$ ;  
(2) find a lower bound  $c^*$  for the optimum objective function value;  
(3) [comment: e.g.,  $c^* := \max(\max_{i \in U} \min_{j \in V} c_{ij}, \max_{j \in V} \min_{i \in U} c_{ij})$ ]  
(4) find a maximum matching  $M$  in the graph  $G[c^*]$  having an edge  $[i, j]$  iff  $c_{ij} \leq c^*$ ;  
(5) if  $|M| = |U|$  then  
    (6)     stop [comment:  $M$  is an optimum matching of cost  $c^*$ ]  
    (7) else  
        (8)     let  $L$  be the set of unmatched vertices of  $U$   
        (9) end  
    (10) while  $L$  is nonempty do  
        (11)     choose an arbitrary vertex  $i \in L$ ;  
        (12)      $L := L \setminus \{i\}$ ;  
        (13)     Dijkstra( $i$ ); [comment: the procedure returns a path  $P$  starting in  $i$ ]  
        (14)     if  $P \neq \text{nil}$  then  
            (15)          $M := M \oplus P$ ;  
            (16)          $c^* := \max(c^*, \ell(P))$   
        (17)     end  
    (18) end  
    (19) [comment:  $M$  is a maximum matching with minimum cost  $c^*$ .]

---

---

**Algorithm 16:** Dijkstra modificado para caminhos aumentantes em LBAP (BURKARD; DELL'AMICO; MARTELLO, 2009)

---

```

(1) Procedure Dijkstra(i)
(2)   label all vertices  $j \in V$  by  $(\alpha(j), \beta(j)) := (\infty, \text{nil})$ ;
(3)    $R := V$ ; [comment:  $R$  contains the unscanned vertices of  $V$ ]
(4)    $\bar{\alpha}(i) := c^*$ ,  $P := \text{nil}$ ;
(5)   Label(i);
(6)   while  $R \neq \emptyset$  do
(7)     find a vertex  $j_1 \in R$  with minimum  $\alpha(j_1)$ ;
(8)     if  $\alpha(j_1) = \infty$  then
(9)        $R := \emptyset$ 
(10)    else
(11)      if  $j_1$  is unmatched then
(12)        find the path  $P$  induced by the vertex sequence  $(i, \dots, \bar{\beta}(\beta(j_1)), \beta(j_1), j_1)$ ;
(13)         $\ell(P) := \alpha(j_1)$ ;
(14)         $R := \emptyset$ 
(15)      else
(16)        let  $[i_1, j_1]$  be the matching edge;
(17)        label  $i_1$  with  $(\bar{\alpha}(i_1), \bar{\beta}(i_1)) = (\alpha(j_1), j_1)$ ;
(18)         $R := R \setminus \{j_1\}$ ;
(19)        Label( $i_1$ );
(20)      end
(21)    end
(22)  end

(23) Procedure Label(i)
(24)   for each neighbor  $j \in R$  of i do
(25)     if  $\alpha(j) > \max(\bar{\alpha}(i), c_{ij})$  then
(26)        $\alpha(j) := \max(\bar{\alpha}(i), c_{ij})$ ;
(27)        $\beta(j) := i$ 
(28)     end
(29)   end

```

---

### 3.3.3 Heurísticas

#### 3.3.3.1 Heurística Construtiva Gulosa

Diferente do problema de atribuição, onde o guloso simples funciona bem, no problema de gargalo o guloso deve ser adaptado para evitar criar "bicos de garrafa" muito altos no final do processo.

O algoritmo seleciona arestas iterativamente. Para cada vértice livre, ele verifica não apenas a aresta de menor peso incidente, mas também o "custo de oportunidade" de não escolhê-la.

Se não escolhermos a aresta  $(u, v)$  de peso 10, a próxima melhor opção para  $u$  tem peso 100? Se sim, a prioridade de  $(u, v)$  aumenta drasticamente (MARTELLO; TOTH, 1987).

## 3.4 Problema do emparelhamento estável

### 3.4.1 Descrição do problema

O problema do emparelhamento estável, também conhecido como *Stable Marriage Problem* (SMP), é um problema clássico na teoria dos grafos e na ciência da computação, proposto inicialmente por Gale e Shapley em 1962 (KLEINBERG; TARDOS, 2005). Ele envolve a combinação de dois conjuntos distintos de elementos e, apesar de semelhante aos problemas de emparelhamento em grafos bipartidos estudados anteriormente, possui características e objetivos diferentes.

Imagine dois conjuntos distintos de mesmo tamanho,  $N$  homens e  $N$  mulheres. Cada indivíduo em ambos os conjuntos tem uma lista de preferências ordenadas para todos os membros do outro conjunto. O objetivo do SMP é encontrar um emparelhamento entre homens e mulheres de tal forma que não existam dois indivíduos, um homem e uma mulher, que prefeririam estar juntos em vez de com seus parceiros atuais. Se tal par existir, o emparelhamento é considerado instável.

O problema pode ser resolvido utilizando o algoritmo de Gale-Shapley, que usa um processo iterativo de propostas e rejeições para garantir que o emparelhamento final seja estável.

### 3.4.2 Propriedades

- **Estabilidade:** Um emparelhamento é estável se não houver dois indivíduos que prefeririam estar juntos em vez de com seus parceiros atuais. A estabilidade é a característica central do problema (LAWLER, 1976).
- **Existência de solução:** Gale e Shapley provaram que sempre existe pelo menos um emparelhamento estável para qualquer conjunto de preferências (AHUJA; MAGNANTI; ORLIN, 1993).
- **Otimalidade para quem propõe:** O emparelhamento resultante do algoritmo de Gale-Shapley é ótimo para o conjunto que faz as propostas (por exemplo, os homens), significando que cada indivíduo nesse conjunto recebe o melhor parceiro possível dentro dos emparelhamentos estáveis (AHUJA; MAGNANTI; ORLIN, 1993).
- **Pessimismo para quem recebe propostas:** Por outro lado, o emparelhamento é o pior possível para o conjunto que recebe as propostas (por exemplo, as mulheres), significando que cada indivíduo nesse conjunto recebe o pior parceiro possível dentro dos emparelhamentos estáveis (AHUJA; MAGNANTI; ORLIN, 1993).
- **Não-unicidade:** Pode haver múltiplos emparelhamentos estáveis para um dado conjunto de preferências, dependendo das listas de preferências dos indivíduos (KLEINBERG; TARDOS, 2005).

### 3.4.3 Algoritmo de Gale-Shapley

A solução clássica para o problema do emparelhamento estável é o algoritmo de Gale-Shapley, também conhecido como o algoritmo de casamento estável. O algoritmo funciona sob a lógica de propostas e rejeições, onde um dos conjuntos (por exemplo, os homens) faz propostas às mulheres com base em suas listas de preferências.

O algoritmo pode ser descrito de forma simples da seguinte maneira:

Enquanto existir um homem  $h$  que ainda não propôs a todas as mulheres em sua lista:

1. O homem  $h$  propõe à primeira mulher  $m$  da sua lista de preferências a quem ele ainda não propôs.
2. A decisão da mulher  $m$  é tomada com base em suas preferências:
  - Se  $m$  não estiver comprometida, ela aceita a proposta de  $h$ . Eles ficam "noivos".
  - Se  $m$  já estiver comprometida com outro homem  $h'$ , ela compara  $h$  e  $h'$ . Se  $m$  preferir  $h$  a  $h'$ , ela rejeita  $h'$  (que fica livre) e aceita a proposta de  $h$ . Caso contrário, ela rejeita  $h$  (que continua livre e tenta a próxima da sua lista).

O algoritmo continua até que todos os homens estejam comprometidos. O resultado final é um emparelhamento estável, onde não existem pares instáveis (KLEINBERG; TARDOS, 2005).

Segue um exemplo em pseudocódigo do algoritmo:

---

**Algorithm 17:** Algoritmo de Gale-Shapley para o problema do emparelhamento estável (KLEIN-BERG; TARDOS, 2005)

---

- (1) Initially all  $m \in M$  and  $w \in W$  are free
  - (2) **while** there is a man  $m$  who is free and hasn't proposed to every woman **do**
  - (3)     Choose such a man  $m$
  - (4)     Let  $w$  be the highest-ranked woman in  $m$ 's preference list to whom  $m$  has not yet proposed
  - (5)     **if**  $w$  is free **then**
  - (6)          $(m, w)$  become engaged
  - (7)     **else**
  - (8)          $w$  is currently engaged to  $m'$
  - (9)         **if**  $w$  prefers  $m'$  to  $m$  **then**
  - (10)              $m$  remains free
  - (11)         **else**
  - (12)              $(m, w)$  become engaged
  - (13)              $m'$  becomes free
  - (14)         **end**
  - (15)     **end**
  - (16) **end**
  - (17) Return the set  $S$  of engaged pairs
- 

## 4 Problemas de emparelhamento em grafos gerais

Ao contrário dos problemas em grafos bipartidos, problemas de emparelhamento em grafos gerais podem ser consideravelmente mais complexos. Alguns dos exemplos apresentados nesta seção são resolvíveis por algoritmo exatos em tempo polinomial, apesar de apresentarem uma complexidade de tempo considerada alta. Outros problemas pertencem à classe NP, o que significa que não foi proposto um algoritmo que pudesse resolver esses problemas em tempo poliomial.

### 4.1 Emparelhamento de cardinalidade máxima

#### 4.1.1 Descrição

A definição do problema de emparelhamento de cardinalidade máxima é, fundamentalmente, a mesma da sua contraparte bipartida, descrita na seção 3.1. No entanto, a solução deste problema é significativamente mais complexa que a versão bipartida, mesmo ainda sendo em tempo polinomial.

#### 4.1.2 Propriedades

##### 4.1.2.1 Ciclos ímpares

Previamente, na seção 3.1.2.2, vimos a definição de caminhos aumentantes, que vieram a ser fundamentais para a solução de grande parte dos problemas de emparelhamento em grafos bipartidos. Em grafos gerais, a base da solução ainda é a mesma, se baseando no lema de Berge, descrito por Jungnickel (2008) como:

Um emparelhamento  $M$  em um grafo  $G$  é máximo se, e somente se, não existe nenhum caminho aumentante em  $G$  relativo a  $M$ .

Em grafos bipartidos, ciclos sempre têm tamanho par. Isso garante que, ao realizar uma busca (BFS ou DFS) a partir de um vértice livre, nunca encontraremos uma aresta que conecte dois vértices que estejam à mesma distância ("paridade") da raiz na árvore de busca alternante (AHUJA; MAGNANTI; ORLIN, 1993).

No entanto, em grafos gerais, a presença de ciclos de comprimento ímpar quebra essa lógica. Schrijver (2004) explica que, ao explorar o grafo, o algoritmo pode encontrar uma aresta conectando dois vértices que

possuem paridade "par" em relação à raiz da busca. Essa estrutura forma um **ciclo ímpar** que "confunde" a identificação de um caminho aumentante simples, pois um vértice poderia ser alcançado por dois caminhos alternantes de paridades diferentes.

#### 4.1.2.2 Flor (Blossom)

A solução para lidar com círculos ímpares foi proposta por Edmonds (1965), e consiste em uma estrutura denominada **Blossom**.

Uma Blossom  $B$  é definida como um ciclo de comprimento ímpar  $2k + 1$ , onde  $k$  arestas pertencem ao emparelhamento  $M$ , de tal forma que existe apenas um vértice  $v_B$  no ciclo (a base da flor) que não é coberto por uma aresta de  $M$  dentro do próprio ciclo. A base  $v_B$  é conectada ao resto do grafo por um caminho alternante de comprimento par (o **caule** ou **stem**).

Edmonds provou que uma *blossom* pode ser contraída em um único vértice, com base no seguinte teorema:

Existe um caminho aumentante no grafo original  $G$  se, e somente se, existe um caminho aumentante no grafo contraído  $G' = G/B$ , onde a flor  $B$  foi reduzida a um único vértice. (SCHRIJVER, 2004)

#### 4.1.3 O algoritmo de Edmonds

Baseando-se na propriedade de Blossom e no teorema da contração, Lawler (1976) descreve o algoritmo para encontrar o emparelhamento máximo em grafos gerais da seguinte forma:

1. **Busca:** Inicie uma busca (geralmente BFS) a partir dos vértices livres para encontrar um caminho aumentante, construindo uma Floresta de Árvores Alternantes.
2. **Detectação:** Se a busca encontrar uma aresta que fecha um ciclo ímpar, uma **blossom** foi identificada.
3. **Contração:** Contraia todos os vértices da **blossom** em um único super-vértice e modifique o grafo temporariamente.
4. **Recursão:** Continue a busca no grafo contraído.
5. **Expansão:** Se um caminho aumentante for encontrado no grafo contraído passando pelo super-vértice, "abra"(expanda) a flor novamente para traduzir esse caminho para o grafo original. O caminho dentro da flor será roteado de forma a manter a alternância correta.
6. **Aumento:** Use o caminho encontrado para aumentar o emparelhamento através da diferença simétrica (inversão das arestas).

Schrijver (2004) detalha que, a depender da forma como o algoritmo é implementado, principalmente no que trata da contração, a complexidade do algoritmo ingênuo pode chegar a  $O(V^3)$ . No entanto, melhorias subsequentes nesse algoritmo culminaram em uma implementação em  $O(E\sqrt{V})$ , descrita por Micali e Vazirani (1980). No entanto, por conta da complexidade de implementação da solução mais rápida, a maioria das implementações práticas do algoritmo utilizam variantes com complexidade próxima a  $O(V^3)$ .

---

**Algorithm 18:** Procedimento MAXMATCH (JUNGNICKEL, 2008)

---

```

(1) Procedure MAXMATCH( $G; mate, nrex$ )
(2)   INMATCH( $G; mate, nrex$ )
(3)    $r \leftarrow 0$ 
(4)   while  $nrex \geq 2$  and  $r < n$  do
(5)      $r \leftarrow r + 1$ 
(6)     if  $mate(r) = 0$  then
(7)        $Q \leftarrow \emptyset, aug \leftarrow \text{false}, m \leftarrow 0$ 
(8)       for  $v \in V$  do
(9)          $p(v) \leftarrow 0, d(v) \leftarrow -1, a(v) \leftarrow \text{true}$ 
(10)         $CA(v) \leftarrow A_v$ 
(11)      end
(12)       $d(r) \leftarrow 0$ ; append  $r$  to  $Q$ 
(13)      while  $aug = \text{false}$  and  $Q \neq \emptyset$  do
(14)        remove the first vertex  $x$  of  $Q$ 
(15)        if  $a(x) = \text{true}$  then
(16)           $cont \leftarrow \text{false}$ 
(17)          for  $y \in CA(x)$  do
(18)             $u(y) \leftarrow \text{false}$ 
(19)          end
(20)          repeat
(21)            choose  $y \in CA(x)$  with  $u(y) = \text{false}; u(y) \leftarrow \text{true}$ 
(22)            if  $a(y) = \text{true}$  then
(23)              if  $d(y) \equiv 0 \pmod{2}$  then
(24)                 $m \leftarrow m + 1$ 
(25)                BLOSSOM( $x, y; B(m), w(m)$ )
(26)                CONTRACT( $B(m), m, w$ )
(27)              end
(28)              else if  $d(y) = -1$  then
(29)                if  $mate(y) = 0$  then
(30)                  AUGMENT( $x, y$ )
(31)                else
(32)                   $z \leftarrow mate(y)$ 
(33)                   $p(y) \leftarrow x; d(y) \leftarrow d(x) + 1$ 
(34)                   $p(z) \leftarrow y; d(z) \leftarrow d(y) + 1$ 
(35)                  insert  $z$  with priority  $d(z)$  into  $Q$ 
(36)                end
(37)              end
(38)            end
(39)            until  $u(y) = \text{true}$  for all  $y \in CA(v)$  or  $aug = \text{true}$  or  $cont = \text{true}$ 
(40)          end
(41)        end
(42)      end
(43)    end

```

---

---

**Algorithm 19:** Algoritmo de BLOSSOM (JUNGNICKEL, 2008)

---

```
(1) Procedure BLOSSOM( $x, y; B, w$ )
(2)   |  $P \leftarrow \{x\}; P' \leftarrow \{y\}; u \leftarrow x; v \leftarrow y$ 
(3)   | repeat
(4)   |   |  $P \leftarrow P \cup \{p(u)\}; u \leftarrow p(u)$ 
(5)   | until  $p(u) = r$ 
(6)   | repeat
(7)   |   |  $P' \leftarrow P' \cup \{p(v)\}; v \leftarrow p(v)$ 
(8)   | until  $v = r$ 
(9)   |  $S \leftarrow P \cap P'$ 
(10)  | let  $w$  be the element of  $S$  for which  $d(w) \geq d(z)$  for all  $z \in S$ 
(11)  |  $B \leftarrow ((P \cup P') \setminus S) \cup \{w\}$ 
```

---

---

**Algorithm 20:** Procedimento CONTRACT (JUNGNICKEL, 2008)

---

```
(1) Procedure CONTRACT( $B, m, w$ )
(2)   |  $b \leftarrow n + m; a(b) \leftarrow \text{true}$ 
(3)   |  $p(b) \leftarrow p(w); d(b) \leftarrow d(w); mate(b) \leftarrow mate(w)$ 
(4)   | insert  $b$  into  $Q$  with priority  $d(b)$ 
(5)   |  $CA(b) \leftarrow \bigcup_{z \in B} CA(z)$ 
(6)   | for  $z \in CA(b)$  do
(7)   |   |  $CA(z) \leftarrow CA(z) \cup \{b\}$ 
(8)   | end
(9)   | for  $z \in B$  do
(10)  |   |  $a(z) \leftarrow \text{false}$ 
(11)  | end
(12)  | for  $z \in CA(b)$  do
(13)  |   | if  $a(z) = \text{true}$  and  $p(z) \in B$  then
(14)  |   |   |  $d(z) \leftarrow d(b) + 1; p(z) \leftarrow b$ 
(15)  |   |   |  $d(mate(z)) \leftarrow d(z) + 1$ 
(16)  |   | end
(17)  | end
(18)  |  $cont \leftarrow \text{true}$ 
```

---

---

**Algorithm 21:** Procedimento AUGMENT (JUNGNICKEL, 2008)

---

```

(1) Procedure AUGMENT( $x, y$ )
(2)    $P \leftarrow \{y, x\}; v \leftarrow x$ 
(3)   while  $p(v) \neq 0$  do
(4)      $| P \leftarrow P \cup \{p(v)\}; v \leftarrow p(v)$ 
(5)   end
(6)   while there exists  $b \in P$  with  $b > n$  do
(7)     choose the largest  $b \in P$  with  $b > n$ 
(8)      $B \leftarrow B(b - n); w \leftarrow w(b - n); z \leftarrow mate(w)$ 
(9)     let  $q$  be the neighbor of  $b$  on  $P$  which is different from  $z$ 
(10)    choose some  $q' \in B \cap CA(q)$ 
(11)    determine the alternating path  $B'$  of even length in  $B$  from  $w$  to  $q'$ 
(12)    replace  $b$  by  $w$  in  $P$ 
(13)    insert  $B'$  into  $P$  between  $w$  and  $q$ 
(14)  end
(15)   $u \leftarrow y; v \leftarrow x$ 
(16)  while  $v \neq r$  do
(17)     $| z \leftarrow mate(v); mate(v) \leftarrow u; mate(u) \leftarrow v$ 
(18)     $| u \leftarrow z; \text{let } v \text{ be the successor of } z \text{ on } P$ 
(19)  end
(20)   $mate(v) \leftarrow u; mate(u) \leftarrow v$ 
(21)   $nrex \leftarrow nrex - 2; aug \leftarrow \text{true}$ 

```

---

#### 4.1.3.1 Teorema de Tutte-Berge

O teorema de Tutte-Berge, apesar de não aparecer explicitamente na descrição do algoritmo, é fundamental para sua corretude, pois ele fornece um certificado de otimalidade. Por exemplo, digamos que a etapa de busca do algoritmo de Blossom não encontrou nenhum caminho aumentante. Como sabemos que o emparelhamento atual é máximo? Para descobrir isso, utilizamos a fórmula de Tutte-Berge, descrita por Jungnickel (2008) da seguinte forma:

$$v(G) = \frac{1}{2} \min_{U \subseteq V} (|V| + |U| - \text{odd}(G - U)) \quad (16)$$

Onde  $v(G)$  é o tamanho do emparelhamento máximo, e  $\text{odd}(G - U)$  é o número de componentes conexos com número ímpar de vértices no grafo resultante após a remoção de  $U$ . Se o grafo possui um emparelhamento perfeito, o mínimo desta equação é  $|V|/2$ , o que ocorre se e somente se  $\text{odd}(G - U) \leq |U|$  para todo  $U$ .

#### 4.1.4 Heurísticas

Algumas das heurísticas que resolvem este problema são as mesmas que vimos na seção 3.1, como a Heurística Gulosa Simples e a Heurística de Karp-Sipser, que também funcionam para grafos gerais. Mas, além dessas, existem outras heurísticas que podem ser utilizadas neste problema.

##### 4.1.4.1 Algoritmo de crescimento de caminho (Path Growing)

Uma variação inteligente do método guloso, descrita por Drake e Hougardy (2003), que usa DFS e tenta evitar a "miopia" de escolher uma aresta isolada.

1. Escolha um vértice livre inicial  $v$ .
2. Inicie uma busca em profundidade (DFS) para construir um caminho simples o mais longo possível.

3. Ao longo desse caminho, selecione arestas alternadas para o emparelhamento (a 1<sup>a</sup> sim, a 2<sup>a</sup> não, a 3<sup>a</sup> sim...).
4. Remova os vértices usados e repita.

Ao contrário do guloso simples (que escolhe arestas desconexas), o Path Growing tende a cobrir cadeias longas de vértices, o que é estruturalmente vantajoso.

## 4.2 Emparelhamento ponderado máximo

Enquanto o problema de cardinalidade máxima busca o maior número de arestas, o problema do **Emparelhamento de Peso Máximo** (Maximum Weight Matching) introduz uma função de peso  $w : E \rightarrow \mathbb{R}$  nas arestas, e o objetivo torna-se encontrar um emparelhamento  $M$  que maximize  $\sum_{e \in M} w(e)$ . Este problema é a generalização natural do Problema da Atribuição para grafos não-bipartidos, descrito em 3.2.

A complexidade deste problema em grafos gerais decorre, assim como no problema de emparelhamento de cardinalidade máxima descrito em 4.1, da presença de ciclos ímpares. Se tentarmos modelar o problema usando Programação Linear (PL) apenas com restrições de grau ( $\sum x_e \leq 1$ ), a solução ótima do sistema relaxado pode ser fracionária (ex:  $x_e = 0.5$  para as três arestas de um triângulo), o que não corresponde a um emparelhamento válido.

### 4.2.1 Formulação poliédrica e Restrições de Blossom

Conforme detalhado por Schrijver (2004), para descrever o **Poliedro do Emparelhamento** (Matching Polytope) em grafos gerais, Edmonds (1965) provou que é necessário adicionar uma família exponencial de restrições conhecidas como **Desigualdades de Blossom** (ou restrições de subconjuntos ímpares):

$$\sum_{e \in E(U)} x_e \leq \frac{|U| - 1}{2}, \quad \forall U \subseteq V, |U| \text{ ímpar}, |U| \geq 3 \quad (17)$$

Onde  $E(U)$  é o conjunto de arestas com ambas as extremidades em  $U$ . Essas inequações garantem que, em qualquer subconjunto ímpar de vértices, o número de arestas selecionadas não excede a capacidade máxima inteira possível.

### 4.2.2 A abordagem Primal-Dual

Diferente do caso de cardinalidade, onde buscamos apenas caminhos aumentantes, o algoritmo para o caso ponderado é um método **Primal-Dual**. Ele é uma generalização do Método Húngaro, mas adaptado para lidar com as blossoms.

Segundo Lawler (1976), o algoritmo mantém variáveis duais não apenas para os vértices ( $y_v$ ), mas também para cada subconjunto ímpar de vértices ( $z_B$ ) que forma uma blossom contraída. A condição de otimalidade baseia-se na folga complementar. Uma aresta  $(u, v)$  é considerada "admissível" ou "justa" (*tight*) se a soma dos duais correspondentes iguala o peso da aresta. Simplificadamente:

$$y_u + y_v + \sum_{B: (u, v) \in B} z_B = w_{uv} \quad (18)$$

O algoritmo opera iterativamente em dois estágios, descritos por Jungnickel (2008) e Schrijver (2004):

1. **Estágio Primal (Busca):** Restringindo-se apenas às arestas admissíveis (onde a equação acima é satisfeita), o algoritmo busca um caminho aumentante usando a técnica de contração de blossoms de Edmonds. Se encontrado, o emparelhamento é aumentado.
2. **Estágio Dual (Ajuste de Preços):** Se a busca falha em encontrar um caminho aumentante no grafo de arestas admissíveis, o algoritmo não termina (ao contrário do caso de cardinalidade). Em vez disso, ele realiza uma atualização das variáveis duais  $y$  e  $z$ .

- O algoritmo calcula um valor  $\delta$  (delta), que é a menor "folga" entre as arestas que conectam a árvore de busca atual a vértices externos.
- Os duais dos vértices na árvore são ajustados por  $\pm\delta$  e os duais das blossoms ( $z_B$ ) também são ajustados.
- Esse ajuste tem o efeito de "destravar" novas arestas (tornando-as admissíveis) ou permitir a expansão/contração de blossoms, possibilitando que a busca prossiga.

### 4.2.3 O algoritmo de Edmonds Ponderado

O procedimento resultante é frequentemente chamado de **Algoritmo de Blossom Ponderado** ou simplesmente Algoritmo de Edmonds para Emparelhamento Ponderado.

A implementação original de Edmonds era polinomial,  $O(V^4)$ . Lawler (1976) apresentou uma implementação  $O(V^3)$ , que por muito tempo foi o padrão em livros didáticos.

A variante  $O(V^3)$  descrita por Lawler (1976) para o emparelhamento ponderado utiliza uma abordagem primal-dual rigorosa. O algoritmo mantém um conjunto de arestas admissíveis (custo reduzido nulo) baseadas em variáveis duais de vértices ( $y_i$ ) e de subconjuntos ímpares ( $z_B$ ). A cada iteração onde a busca por caminhos aumentantes falha, o algoritmo calcula um escalar crítico  $\delta$  baseado nas folgas das arestas incidentes à árvore alternante atual (conjunto  $S$ ). A atualização dual subsequente ( $y_i \leftarrow y_i \pm \delta$ ,  $z_B \leftarrow z_B \pm 2\delta$ ) garante o surgimento de novas arestas admissíveis ou a expansão de \*blossoms\* com dual nulo, mantendo a complexidade cúbica através do gerenciamento eficiente dos valores de folga mínima (\*slack\*) para vértices fora da árvore.

Lawler também destaca que a manipulação das variáveis duais das blossoms ( $z_B$ ) é o que permite ao algoritmo "pagar" o custo de usar arestas dentro de estruturas cíclicas ímpares. Quando uma blossom é formada durante a busca, ela recebe um valor dual  $z_B > 0$ ; quando ela precisa ser desfeita (expandida) para encontrar um caminho, seu valor dual retorna a zero.

### 4.2.4 Otimizações

Desenvolvimentos posteriores por Galil, Micali e Gabow (1986) e Gabow (1990) melhoraram a estrutura de dados para gerenciar a prioridade das atualizações duais e a contração de blossoms, atingindo complexidades próximas a  $O(EV \log V)$  ou  $O(E\sqrt{V}\alpha(E, V))$  em implementações de escalonamento (SCHRIJVER, 2004).

Apesar de ser polinomial (classe P), a implementação de emparelhamento ponderado em grafos gerais é considerada uma das rotinas mais complexas da otimização combinatória clássica, sendo significativamente mais difícil de implementar corretamente do que algoritmos para grafos bipartidos ou fluxo máximo.

### 4.2.5 Heurísticas

Aqui também podemos utilizar a mesma heurística que usamos para a versão do problema para grafos bipartidos: a Heurística Gulosa Ordenada, descrita na seção 3.2.4.1. Mas além dessas, temos:

#### 4.2.5.1 Algoritmo de Crescimento de Caminho (Path Growing)

Similar à heurística descrita na seção 4.1.4.1, esta também faz uso de Path Growing para encontrar um emparelhamento. A vantagem desta é que não é necessário ordenação, tornando possível a execução em tempo linear ( $O(E)$ ). Foi descrito por Drake e Hougardy (2003):

1. O algoritmo constrói caminhos simples disjuntos no grafo.
2. Comece em um vértice livre  $v$  qualquer. Estenda o caminho escolhendo a aresta de maior peso conectada a  $v$  que leve a um vértice ainda não visitado neste caminho.
3. Repita até que o caminho não possa crescer mais.
4. Dado um caminho  $P = (e_1, e_2, e_3, \dots, e_k)$ , temos dois emparelhamentos possíveis formados por arestas alternadas:

5.
  - $M_1 = \{e_1, e_3, e_5, \dots\}$  (Arestas ímpares)
  - $M_2 = \{e_2, e_4, e_6, \dots\}$  (Arestas pares)
6. Compare o peso total de  $M_1$  e  $M_2$ . Adicione o conjunto mais pesado à solução final e remova os vértices do grafo.

A diferença para a versão de emparelhamento de cardinalidade máxima é o passo 5, onde os pesos são somados para escolher o melhor conjunto. Garante emparelhamento de pelo menos  $1/2$  da cardinalidade máxima (DRAKE; HOGARDY, 2003).

## 4.3 Emparelhamento induzido máximo

Diferentemente dos problemas de emparelhamento em grafos gerais vistos até o momento, que pertencem à classe de complexidade P, o problema do Emparelhamento Induzido Máximo (MIM - *Maximum Induced Matching*), frequentemente referido na literatura como **Emparelhamento Forte** (*Strong Matching*), pertence à classe os problemas NP-difíceis. Apesar de ter uma variação estruturalmente simples quando comparado aos problemas clássicos, esse problema impõe uma restrição topológica severa sobre quais arestas podem coexistir na solução.

### 4.3.1 Propriedades

Seja  $G = (V, E)$  um grafo geral. Um subconjunto de arestas  $M \subseteq E$  é um **emparelhamento induzido** se satisfaz duas condições:

1. **Condição de Emparelhamento:** Nenhuma aresta em  $M$  compartilha um vértice (são disjuntas par a par).
2. **Condição Induzida:** O subgrafo induzido pelos vértices saturados por  $M$ , denotado por  $G[V(M)]$ , contém **apenas** as arestas de  $M$ .

Em termos mais simples, se tomarmos duas arestas quaisquer  $(u, v)$  e  $(x, y)$  pertencentes ao emparelhamento induzido, não pode existir no grafo original nenhuma aresta conectando um vértice do par  $\{u, v\}$  a um vértice do par  $\{x, y\}$ . Isso implica que as arestas do emparelhamento devem estar "topologicamente isoladas"umas das outras.

Cameron (1989) oferece uma caracterização alternativa baseada em distância: um emparelhamento induzido é um conjunto de arestas onde a distância entre quaisquer duas arestas distintas é de pelo menos 2 (ou seja, não há aresta de ligação entre elas).

### 4.3.2 Complexidade computacional

Diferentemente do emparelhamento clássico (cardinalidade ou ponderado), que possui algoritmos polinomiais eficientes, o problema de encontrar o emparelhamento induzido de cardinalidade máxima é **NP-Difícil**.

Este resultado foi estabelecido no trabalho fundamental de Stockmeyer e Vazirani (1982). Eles demonstraram que o problema permanece NP-Difícil mesmo quando restrito a grafos bipartidos com grau máximo 3. Isso é um fato surpreendente e crucial: a propriedade de ser bipartido, que geralmente torna problemas de emparelhamento fáceis (como visto no Teorema de König), **não** ajuda no caso do emparelhamento induzido.

### 4.3.3 Redução ao Problema do Conjunto Independente

A intratabilidade do MIM pode ser explicada através de uma transformação de grafos. O problema de encontrar um emparelhamento induzido em  $G$  é equivalente a encontrar um **Conjunto Independente Máximo** (Maximum Independent Set - MIS) em um grafo transformado.

A transformação ocorre em dois passos:

1. Construímos o **Grafo de Linha**  $L(G)$ , onde cada vértice representa uma aresta de  $G$  e dois vértices são adjacentes se as arestas correspondentes em  $G$  compartilham uma ponta. (Emparelhamento clássico em  $G \equiv$  Conjunto Independente em  $L(G)$ ).
2. Construímos o **Quadrado do Grafo de Linha**  $L(G)^2$ . Neste grafo, conectamos dois vértices se a distância entre eles em  $L(G)$  for 1 ou 2.

Encontrar um emparelhamento induzido máximo em  $G$  equivale exatamente a encontrar um Conjunto Independente Máximo em  $L(G)^2$  (CAMERON, 1989). Como o problema do Conjunto Independente é NP-Difícil para grafos gerais (GAREY; JOHNSON, 1990), essa relação estrutural justifica a dificuldade do MIM.

#### 4.3.4 Abordagens de solução e heurísticas

Dada a natureza NP-Difícil do problema e a ineficácia de métodos exatos para grafos densos ou de grande escala, a literatura concentra-se majoritariamente em algoritmos construtivos gulosos e metaheurísticas.

##### 4.3.4.1 Heurísticas gulosas

Dada a equivalência entre o problema do Emparelhamento Induzido Máximo em  $G$  e o Conjunto Independente Máximo no grafo transformado  $L(G)^2$ , as heurísticas gulosas clássicas da literatura de conjuntos independentes são naturalmente adaptadas para este contexto.

O procedimento geral consiste em selecionar iterativamente uma aresta  $e \in E$ , adicioná-la à solução e remover sua vizinhança induzida do grafo. As variantes diferem no critério de seleção da aresta:

- **Guloso Aleatório (Random Greedy):** Seleciona a próxima aresta uniformemente ao acaso dentre as disponíveis. Duckworth, Wormald e Zito (2002) analisaram o comportamento assintótico desta abordagem em grafos cúbicos aleatórios, demonstrando que, apesar de sua simplicidade, ela fornece uma linha de base (*baseline*) essencial para a avaliação de métodos mais sofisticados.
- **Guloso de Grau Mínimo (Min-Degree Greedy):** A heurística prioriza a seleção da aresta que possui o **menor grau induzido** (ou seja, a aresta cuja remoção implica a eliminação do menor número de outras arestas candidatas). Esta abordagem é uma adaptação direta da heurística *Min-Degree* (ou *Greedy*) para o problema do Conjunto Independente Máximo. Halldórsson e Radhakrishnan (1994) estabeleceram os fundamentos teóricos para essa estratégia, demonstrando que priorizar vértices de baixo grau fornece garantias de aproximação superiores em grafos esparsos e de grau limitado. No contexto do MIM, essa lógica traduz-se em escolher a aresta  $e = (u, v)$  que minimiza a soma dos graus de seus extremos ou o tamanho de sua vizinhança em  $L(G)^2$ .

##### 4.3.4.2 Metaheurísticas

Para superar os ótimos locais frequentemente encontrados pelos métodos gulosos, abordagens iterativas de busca local têm sido aplicadas.

Fürst, Leichter e Rautenbach (2017) investigaram formalmente o poder da busca local para o Emparelhamento Induzido Máximo. A estratégia analisada consiste em iniciar com um emparelhamento induzido qualquer (ou vazio) e tentar aumentá-lo iterativamente através de melhorias locais simples. Eles focaram na análise de grafos  $k$ -regulares e demonstraram que, embora a busca local possa ficar presa em ótimos locais longe da solução global em grafos gerais, ela oferece garantias de desempenho interessantes para classes específicas (como grafos livres de garras ou  $C_4$ -free). O algoritmo de busca local proposto tenta adicionar arestas que não violem a restrição induzida e, se necessário, remover um número pequeno de arestas existentes para acomodar a nova inclusão, um processo análogo às trocas (\* $k$ -swaps\*) em outros problemas combinatórios.

## **4.4 Outros problemas**

No futuro, serão inclusos neste trabalho outros problemas de emparelhamento em grafos gerais, como o de b-emparelhamento, emparelhamento em matroid, e o Stable Roommates Problem.

## **5 Trabalhos futuros**

As próximas etapas deste trabalho concentram-se no aprofundamento teórico dos N problemas de emparelhamento previamente delineados, bem como na inclusão de novos problemas e na investigação de M abordagens heurísticas para resolução dos mesmos. O objetivo central é estabelecer uma análise comparativa rigorosa entre estas heurísticas e os algoritmos exatos consagrados na literatura.

Para viabilizar essa análise, será desenvolvido um ambiente computacional padronizado (framework de testes), onde os algoritmos serão implementados e submetidos a um conjunto de X instâncias de teste com topologias e dimensões variadas. A avaliação final buscará classificar as soluções heurísticas considerando a relação entre eficiência computacional (tempo de execução) e a qualidade da solução (proximidade do ótimo), utilizando os resultados dos algoritmos exatos como referência.

## Referências

- AHO, A. V.; HOPCROFT, J. E.; ULLMAN, J. D. *The Design And Analysis of Computer Algorithms*. 1st. ed. Reading, MA: Addison-Wesley, 1974. ISBN 9780201000290.
- AHUJA, R. K.; MAGNANTI, T. L.; ORLIN, J. B. *Network flows: theory, algorithms, and applications*. USA: Prentice-Hall, Inc., 1993. ISBN 013617549X.
- BERTSEKAS, D. The auction algorithm: A distributed relaxation method for the assignment problem. *Annals of Operations Research*, v. 14, p. 105–123, 12 1988.
- BERTSEKAS, D. et al. *Auction Algorithms for Network Flow Problems: A Tutorial Introduction*. Massachusetts Institute of Technology, Laboratory for Information and Decision Systems, 1992. (CICS (Series)). Disponível em: <https://books.google.com.br/books?id=DCYhzwEACAAJ>.
- BURKARD, R.; DELL'AMICO, M.; MARTELLO, S. *Assignment Problems*. USA: Society for Industrial and Applied Mathematics, 2009. ISBN 0898716632.
- CAETANO, T. S. et al. Learning graph matching. *IEEE TRANSACTIONS ON PATTERN ANALYSIS AND MACHINE INTELLIGENCE*, IEEE Computer Society, United States, v. 31, n. X, December 2009. ISSN 0162-8828.
- CAMERON, K. Induced matchings. *Discrete Applied Mathematics*, v. 24, n. 1, p. 97–102, 1989. ISSN 0166-218X. Disponível em: <https://www.sciencedirect.com/science/article/pii/0166218X9290275F>.
- CORMEN, T. H. et al. *Introduction to Algorithms*. 3rd. ed. Cambdrige, MA: The MIT Press, 2009. ISBN 978-0-262-53305-8.
- DASGUPTA, C. H. P. S.; VAZIRANI, U. V. *Algorithms*. 1st. ed. New York, NY: McGraw-Hill Education, 2006. ISBN 978-0073523408.
- DRAKE, D. E.; HOUGARDY, S. A simple approximation algorithm for the weighted matching problem. *Information Processing Letters*, v. 85, n. 4, p. 211–213, 2003. ISSN 0020-0190. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0020019002003939>.
- DUCKWORTH, W.; WORMALD, N.; ZITO, M. Maximum induced matchings of random cubic graphs. *Journal of Computational and Applied Mathematics*, v. 142, n. 1, p. 39–50, 2002. ISSN 0377-0427. Probabilistic Methods in Combinatorics and Combinatorial Optimization. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0377042701004575>.
- EDMONDS, J. Paths, trees, and flowers. *Canadian Journal of Mathematics*, v. 17, p. 449–467, 1965.
- FAN, W. Graph pattern matching revised for social network analysis. In: *Proceedings of the 15th International Conference on Database Theory*. New York, NY, USA: Association for Computing Machinery, 2012. (ICDT '12), p. 8–21. ISBN 9781450307918. Disponível em: <https://doi.org/10.1145/2274576.2274578>.
- FEOFILOFF, P. *Emparelhamento em Grafos Bipartidos*. 2019. IME-USP. Disponível em: [https://www.ime.usp.br/~pf/algoritmos\\_para\\_grafos/aulas/matching-bipartite.html](https://www.ime.usp.br/~pf/algoritmos_para_grafos/aulas/matching-bipartite.html).
- FÜRST, M.; LEICHTER, M.; RAUTENBACH, D. *Locally Searching for Large Induced Matchings*. 2017. Disponível em: <https://arxiv.org/abs/1708.02028>.
- GABOW, H. N. Data structures for weighted matching and nearest common ancestors with linking. In: *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms*. USA: Society for Industrial and Applied Mathematics, 1990. (SODA '90), p. 434–443. ISBN 0898712513.
- GALIL, Z.; MICALI, S.; GABOW, H. An  $o(ev \log v)$  algorithm for finding a maximal weighted matching in general graphs. *SIAM J. Comput.*, v. 15, p. 120–130, 02 1986.

GAREY, M. R.; JOHNSON, D. S. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. USA: W. H. Freeman & Co., 1990. ISBN 0716710455.

HALIM, S.; HALIM, F. *Competitive Programming 3: The New Lower Bound of Programming Contests*. 3rd. ed. [S.I.]: Lulu.com, 2013. Self-published by the authors. ISBN 978-1482852483.

HALLDÓRSSON, M.; RADHAKRISHNAN, J. Greed is good: approximating independent sets in sparse and bounded-degree graphs. In: *Proceedings of the Twenty-Sixth Annual ACM Symposium on Theory of Computing*. New York, NY, USA: Association for Computing Machinery, 1994. (STOC '94), p. 439–448. ISBN 0897916638. Disponível em: [⟨https://doi.org/10.1145/195058.195221⟩](https://doi.org/10.1145/195058.195221).

HALLER, S. et al. *A Comparative Study of Graph Matching Algorithms in Computer Vision*. 2022. Disponível em: [⟨https://arxiv.org/abs/2207.00291⟩](https://arxiv.org/abs/2207.00291).

JONKER, R.; VOLGENANT, A. A shortest augmenting path algorithm for dense and sparse linear assignment problems. *Computing*, v. 38, n. 4, p. 325–340, 1987. Disponível em: [⟨https://doi.org/10.1007/BF02278710⟩](https://doi.org/10.1007/BF02278710).

JUNGNICKEL, D. *Graphs, Networks and Algorithms*. 3rd. ed. Berlin, Germany: Springer, 2008. ISBN 978-354727798.

KARP, R. M.; SIPSER, M. Maximum matching in sparse random graphs. In: *Proceedings of the 22nd Annual Symposium on Foundations of Computer Science*. USA: IEEE Computer Society, 1981. (SFCS '81), p. 364–375. Disponível em: [⟨https://doi.org/10.1109/SFCS.1981.21⟩](https://doi.org/10.1109/SFCS.1981.21).

KLEINBERG, J.; TARDOS Éva. *Algorithm Design*. 1st. ed. Boston, MA: Pearson, 2005. ISBN 978-0321295354.

LAWLER, E. L. *Combinatorial Optimization: Networks and Matroids*. New York, NY: Dover Publications, 1976. Reprint edition. ISBN 978-0486414539.

MANBER, U. *Introduction to Algorithms: A Creative Approach*. 1st. ed. Reading, MA: Addison-Wesley, 1989. ISBN 978-0201120370.

MANDULAK, M. et al. Efficient weighted graph matching on gpus. *SC24*, IEEE Computer Society, United States, November 2024. ISSN 0162-8828.

MARTELLO, S.; TOTH, P. Linear assignment problems. In: MARTELLO, S. et al. (Ed.). *Surveys in Combinatorial Optimization*. North-Holland, 1987, (North-Holland Mathematics Studies, v. 132). p. 259–282. Disponível em: [⟨https://www.sciencedirect.com/science/article/pii/S0304020808732389⟩](https://www.sciencedirect.com/science/article/pii/S0304020808732389).

MICALI, S.; VAZIRANI, V. An  $O(\sqrt{v}E)$  Algorithm for Finding Maximum Matching in General Graphs. In: . [S.I.: s.n.], 1980. p. 17–27.

ROSEN, K. H.; KREHER, D. L.; STINSON, D. R. *Discrete Mathematics and Its Applications*. 1st. ed. Boca Raton, FL: CRC Press, 1998. ISBN 978-0849339882.

SAAD-ELDIN, A. et al. *Graph Matching via Optimal Transport*. 2021. Disponível em: [⟨https://arxiv.org/abs/2111.05366⟩](https://arxiv.org/abs/2111.05366).

SCHRIJVER, A. *Combinatorial Optimization: Polyhedra and Efficiency*. 1st. ed. Berlin, Germany: Springer, 2004. ISBN 3540204563.

STOCKMEYER, L. J.; VAZIRANI, V. V. Np-completeness of some generalizations of the maximum matching problem. *Information Processing Letters*, v. 15, n. 1, p. 14–19, 1982. ISSN 0020-0190. Disponível em: [⟨https://www.sciencedirect.com/science/article/pii/0020019082900771⟩](https://www.sciencedirect.com/science/article/pii/0020019082900771).

SUSSMAN, D. L.; LI, L. *Overview of Graph Matching Challenges and Approaches*. 2021. Acessado em: 26 nov. 2024. Disponível em: [⟨https://www.ll.mit.edu/⟩](https://www.ll.mit.edu/).

TAYLOR, W. R. Protein structure comparison using bipartite graph matching and its application to protein structure classification. *Molecular and Cellular Proteomics*, American Society for Biochemistry and Molecular Biology, United States, v. 1, n. 4, April 2002. ISSN 1535-9476.

VAZIRANI, V. V. *Approximation algorithms*. Berlin, Heidelberg: Springer-Verlag, 2001. ISBN 3540653678.

YAN, J.; YANG, S.; HANCOCK, E. Learning for graph matching and related combinatorial optimization problems. In: BESSIÈRE, C. (Ed.). *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI-20*. International Joint Conferences on Artificial Intelligence Organization, 2020. p. 4988–4996. Survey track. Disponível em: <<https://doi.org/10.24963/ijcai.2020/694>>.

ZASLAVSKIY, M.; BACH, F.; VERT, J.-P. A path following algorithm for the graph matching problem. *IEEE TRANSACTIONS ON PATTERN ANALYSIS AND MACHINE INTELLIGENCE*, IEEE Computer Society, United States, v. 31, n. 12, December 2009. ISSN 0162-8828.