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

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 et al., 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 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) $w : E \to R$ está associada às arestas. Formalmente: $G = (V, E, w)$, onde $w : E \to R$. 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; 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 calsses 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 caad 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 $w : E \to \mathbb{R}$. O peso de um caminho $p = \langle v_0, v_1 \ldots, v_k \rangle$ 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 camino 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:

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

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 $O(E + VlogV)$, o que é crucial para grafos densos.

TODO: pseudocódigo

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 $O(VE)$.

TODO: pseudocódigo

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 $1, 2, \ldots, k$ como intermediários. A recorrência é dada por:

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

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.

TODO: pseudocodigo

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 $c(u, v) \geq o$, que limita a quantidade de fluxo que pode atravessar aquela aresta. Um **fluxo** em G é uma função $f : E \to \mathbb{R}^+$ que satisfaz as seguintes propriedades (CORMEN et al., 2009):

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

$$0 \leq f(u, v) \leq c(u, v), \qquad \forall (u, v) \in E \tag{4}$$

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

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

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):

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

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 caminhhos 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:

$$c_f(u, v) = c(u, v) - f(u, v) \tag{7}$$

Quando a aresta (u, v) possui fluxo igual à sua capacidade, $c_f(u, v) = 0$ 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 1: Ford-Fulkerson Algorithm

```
(1)  Procedure FORD-FULKERSON(G, s, t)
(2)      foreach edge (u, v) ∈ G.E do
(3)          (u, v).f ← 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) ← min{c_f(u, v) : (u, v) is in p}
(7)          foreach edge (u, v) ∈ p do
(8)              if (u, v) ∈ E then
(9)                  (u, v).f ← (u, v).f + c_f(p)
(10)             else
(11)                 (v, u).f ← (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 one 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:

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

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

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

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 resiual $G(f)$ 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 4.2.3.3.

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 SAYAN GHOSH; SLOTA, 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 JULIAN J. MCAULEY; SMOLA, 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 JULIAN J. MCAULEY; SMOLA, 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 Taxonomia dos problemas

No contexto de emparelhamento em grafos, é possível classificar os problemas em diferentes categorias com base nas características dos grafos e nos objetivos específicos de emparelhamento.

3.1 Classificação Baseada na Estrutura do Grafo e Tipo de Atribuição

A classificação inicial dos problemas de emparelhamento é frequentemente determinada pela estrutura do grafo subjacente e pelas restrições de atribuição impostas:

- **Grafos Bipartidos:** É um caso restrito, mas crucial no estudo do problema de emparelhamento. Pode ser resolvido de forma exata, em um ”ambiente”contido, utilizando algoritmos como o algoritmo Hungaro e algoritmos de fluxo em redes. (COUR; SRINIVASAN; SHI, 2006) (LAWLER, 1976)
- **Grafos Gerais:** O emparelhamento em grafos não bipartidos é uma generalização mais complexa do problema. Encontrar uma solução exata para este tipo de problema pode ser computacionalmente desafiador, e muitas vezes são utilizados algoritmos aproximados ou heurísticas para obter soluções de forma viável.

- **Multigrafos:** Alguns problemas de emparelhamento envolvem multigrafos, onde múltiplas arestas podem existir entre o mesmo par de vértices. Como mencionado anteriormente, este trabalho não abordará esse tipo de problema.

4 Problemas de emparelhamento em grafos bipartidos

4.1 Emparelhamento de cardinalidade máxima

4.1.1 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).

4.1.2 Propriedades

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

4.1.2.1 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).

4.1.2.2 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)

4.1.2.3 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.

4.1.3 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:

4.1.3.1 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 maximo 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 2: 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] ← 1
(7)      for each r ∈ AdjList[l] do
(8)          if match[r] = −1 or Aug(match[r]) then
(9)              match[r] ← 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 ← 0
(16)     match ← −1 (initialize for all V)
(17)     for l ← 0 to n − 1 do
(18)         vis ← 0 (reset before each recursion)
(19)         MCBM ← 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).

4.1.3.2 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.

4.1.3.3 Algoritmo de Edmonds-Karp

O algoritmo de **Edmonds-Karp** é p 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 camino aumentante mais curto em número de arestas. Tem complexidade de tempo $O(VE^2)$:

Algorithm 3: 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 ← minEdge
(5)          return
(6)      else
(7)          if p[v] ≠ −1 then
(8)              Augment(p[v], min(minEdge, res_{p[v],v}))
(9)              res_{p[v],v} ← res_{p[v],v} − f
(10)             res_{v,p[v]} ← res_{v,p[v]} + f
(11)         end
(12)     end

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

4.1.3.4 Algoritmo de Dinic

TODO

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.

4.1.3.5 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 4: 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 ≠ ∅ do
(5)  |  Construct_layered_graph;
(6)  |  Find_set_ΔM;
(7)  |  M := M ⊖ ΔM;
(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 ≠ ∅ do
(13) |  |  for each i ∈ L_k do
(14) |  |  |  N(i) := {j : [i, j] ∈ E \ M, j ∉ L_1 ∪ L_3 ∪ ··· ∪ L_{k−1}};
(15) |  |  end
(16) |  |  L_{k+1} := ⋃_{i∈L_k} N(i);
(17) |  |  if L_{k+1} = ∅ then
(18) |  |  |  stop [comment: M is a maximum matching]
(19) |  |  end
(20) |  |  if L_{k+1} ∩ V_0 ≠ ∅ then
(21) |  |  |  k* := k + 1;
(22) |  |  |  L_{k+2} := ∅
(23) |  |  else
(24) |  |  |  L_{k+2} := {ī : [ī, j] ∈ M, j ∈ L_{k+1}};
(25) |  |  end
(26) |  |  k := k + 2
(27) |  end

(28) Procedure Find_set_ΔM
(29) |  [comment: find a maximal set ΔM of disjoint augmenting paths]
(30) |  delete all vertices in L_{k*} \ V_0;
(31) |  mark all remaining vertices as unscanned;
(32) |  k := 1; [comment: path counter]
(33) |  while L_0 ≠ ∅ do
(34) |  |  choose x_0 := i ∈ L_0 and delete it in L_0;
(35) |  |  ℓ := 0;
(36) |  |  while ℓ ≥ 0 do
(37) |  |  |  while x_ℓ has an unscanned neighbor in L_{ℓ+1} do
(38) |  |  |  |  choose unscanned neighbor x_{ℓ+1};
(39) |  |  |  |  mark x_{ℓ+1} as scanned;
(40) |  |  |  |  ℓ := ℓ + 1;
(41) |  |  |  |  if ℓ = k* then
(42) |  |  |  |  |  P_k := (x_0, x_1, ..., x_{k*});
(43) |  |  |  |  |  k := k + 1
(44) |  |  |  |  end
(45) |  |  |  end
(46) |  |  |  if ℓ < k* then
(47) |  |  |  |  ℓ := ℓ − 1
(48) |  |  |  else
(49) |  |  |  |  ℓ := −1
(50) |  |  |  end
(51) |  |  end
(52) |  end
(53) |  return ΔM := (P_1, P_2, ..., P_{k−1})
```

4.2 Problema de atribuição (Assignment Problem)

4.2.1 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.

4.2.2 Propriedades

4.2.3 Algoritmos exatos

4.2.3.1 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 Kőnig 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 5: Algoritmo Húngaro (JUNGNICKEL, 2008)

```
(1)  Procedure HUNGARIAN(n, w; mate)
(2)      for v ∈ V do mate(v) ← 0
(3)      for i = 1 to n do u_i ← max{w_ij : j = 1, ..., n}; v_i ← 0
(4)      nrex ← n
(5)      while nrex ≠ 0 do
(6)          for i = 1 to n do m(i) ← false; p(i) ← 0; δ_i ← ∞
(7)          aug ← false; Q ← {i ∈ S : mate(i) = 0}
(8)          repeat
(9)              remove an arbitrary vertex i from Q; m(i) ← true; j ← 1
(10)             while aug = false and j ≤ n do
(11)                 if mate(i) ≠ j' then
(12)                     if u_i + v_j − w_ij < δ_j then
(13)                         δ_j ← u_i + v_j − w_ij; p(j) ← i
(14)                         if δ_j = 0 then
(15)                             if mate(j') = 0 then
(16)                                 AUGMENT(mate, p, j'; mate)
(17)                                 aug ← true; nrex ← nrex − 1
(18)                             else
(19)                                 Q ← Q ∪ {mate(j')}
(20)                             end
(21)                         end
(22)                     end
(23)                 end
(24)                 j ← j + 1
(25)             end
(26)             if aug = false and Q = ∅ then
(27)                 J ← {i ∈ S : m(i) = true}; K ← {j' ∈ T : δ_j = 0}
(28)                 δ ← min{δ_j : j' ∈ T \ K}
(29)                 for i ∈ J do u_i ← u_i − δ
(30)                 for j' ∈ K do v_j ← v_j + δ
(31)                 for j' ∈ T \ K do δ_j ← δ_j − δ
(32)                 X ← {j' ∈ T \ K : δ_j = 0}
(33)                 if mate(j') ≠ 0 for all j' ∈ X then
(34)                     for j' ∈ X do Q ← Q ∪ {mate(j')}
(35)                 else
(36)                     choose j' ∈ X with mate(j') = 0
(37)                     AUGMENT(mate, p, j'; mate)
(38)                     aug ← true; nrex ← nrex − 1
(39)                 end
(40)             end
(41)         until aug = true
(42)     end
```

Algorithm 6: Método AUGMENT (JUNGNICKEL, 2008)

```
(1)  Procedure AUGMENT(mate, p, j'; mate)
(2)      repeat
(3)          i ← p(j); mate(j') ← i; next ← mate(i); mate(i) ← j'
(4)          if next ≠ 0 then
(5)              j' ← next
(6)          end
(7)      until next = 0
```

4.2.3.2 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).

4.2.3.3 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 7: Algoritmo de caminhos de custo mínimo sucessivos (JUNGNICKEL, 2008)

```
(1)  Procedure OPTFLOW(G, c, s, t, γ, v; f, sol)
(2)      for e ∈ E do f(e) ← 0
(3)      sol ← true, val ← 0
(4)      while sol = true and val < v do
(5)          construct the auxiliary network N' = (G', c', s, t) with cost function γ'
(6)          if t is not accessible from s in G' then
(7)              sol ← false
(8)          else
(9)              determine a shortest path P from s to t in (G', γ')
(10)             δ ← min{c'(e) : e ∈ P}; δ' ← min(δ, v − val); val ← val + δ'
(11)             augment f along P by δ'
(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 7 utilizando a técnica de potenciais:

Algorithm 8: Adaptação do algoritmo 7 utilizando Dijkstra com técnica de potenciais

Procedure *OPTFLOW_DIJKSTRA*($G, c, s, t, \gamma, v; f$, *sol*)
  for $e \in E$ **do** $f(e) \leftarrow 0$
  for $x \in V$ **do** $\pi(x) \leftarrow 0$
  sol $\leftarrow$ true, val $\leftarrow 0$
  while *sol* $=$ *true* **and** *val* $< v$ **do**
    construct the auxiliary network $N' = (G', c', s, t)$
    for $e = (u, v) \in E(G')$ **do**
      define reduced cost: $w(e) \leftarrow \gamma(e) + \pi(u) - \pi(v)$
    end
    Run Dijkstra on N' using weights w to find shortest path distances $d(\cdot)$ from s
    if t *is not accessible from* s *(i.e.,* $d(t) = \infty$*)* **then**
      sol $\leftarrow$ false
    else
      for $x \in V$ *such that* $d(x) < \infty$ **do** $\pi(x) \leftarrow \pi(x) + d(x)$
      Identify shortest path P from s to t based on $d(\cdot)$
      $\delta \leftarrow \min\{c'(e) : e \in P\}; \delta' \leftarrow \min(\delta, v - \text{val}); \text{val} \leftarrow \text{val} + \delta'$
      augment f along P by δ'
    end
  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.

4.3 Problema de atribuição gargalo

4.3.1 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. Para resolver este problema, podemos utilizar o método de threshold (LAWLER, 1976).

4.3.2 Propriedades

4.3.3 Soluções exatas

4.3.3.1 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 $c\star$. Na segunda fase, constrói-se um grafo bipartido com todas as arestas cujo peso é menor ou igual a $c\star$. Em seguida, verifica-se se existe um emparelhamento perfeito nesse grafo. O menor valor de $c\star$ 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 $c\star$, podemos utilizar uma busca binária entre o menor e o maior peso das arestas do grafo original. A cada iteração, ajustamos o valor de $c\star$ com base na existência ou não de um emparelhamento perfeito no grafo construído. Esse processo continua até que o valor ótimo seja encontrado. Burkard (BURKARD; DELL'AMICO; 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 no capítulo 4.1.3, como por exemplo o algoritmo de Hopcroft-Karp.

Lawler (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 no capítulo 4.3.3.3.

Algorithm 9: 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** $\{1, 2, \ldots, n\}$ **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**

4.3.3.2 Método dual

Assim como os métodos de threshold, o método dual também define um valor limite $c\star$. No entanto, ao invés de 'chutar' um valor usando busca binária, este método define o valor inicial de $c\star$ 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:

$$c\star = \max\left\{\max_i(\min_j c_{ik}), \max_j(\min_i c_{kj})\right\} \tag{10}$$

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 $c\star$, usando os mesmos algoritmos de antes. Se um emparelhamento perfeito for encontrado, então $c\star$ é a solução ótima. Caso contrário, é necessário aumentar o valor de $c\star$, e para isso nos baseamos no teorema de Kőnig.

Sabemos que todas as arestas $c_{ij} > c\star$ 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 $c\star$, 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 $c\star$:

$$Novoc\star = \min_{i\notin I, j\notin J} c_{ij} \tag{11}$$

A partir daqui, o processo se repete. Com o novo valor de $c\star$ 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 $c\star$ será a solução ótima para o problema de atribuição gargalo.

Algorithm 10: Dual_LBAP. Dual algorithm for the LBAP (BURKARD; DELL'AMICO; MARTELLO, 2009).

```
(1)  let C = (c_ij) be a given n × n cost matrix;
(2)  c* := max_{k=1,2,...,n}(min_{i=1,2,...,n} c_ik, min_{j=1,2,...,n} c_kj)
(3)  M := ∅
(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 ⊆ U and J ⊆ V be vertex sets in a minimum vertex cover of G[c*]
(9)  |   |   c* := min_{i∉I, j∉J} c_ij
(10) |   end
(11) end
```

4.3.3.3 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:

$$cap(P) = \max_{(i,j)\in P} c_{ij} \tag{12}$$

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:

$$NovoCusto_v = \max(d_u, c_{uv}) \tag{13}$$

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 11: 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) | | $M := M \ominus P$;
(16) | | $c^* := \max(c^*, \ell(P))$
(17) | **end**
(18) **end**
(19) **[comment:** M **is a maximum matching with minimum cost** c^***.]**

Algorithm 12: Dijkstra modificado para caminhos aumentantes em LBAP (BURKARD; DELL'AMICO; MARTELLO, 2009)

```
(1)  Procedure Dijkstra(i)
(2)      label all vertices j ∈ V by (α(j), β(j)) := (∞, nil);
(3)      R := V; [comment: R contains the unscanned vertices of V]
(4)      ᾱ(i) := c*, P := nil;
(5)      Label(i);
(6)      while R ≠ ∅ do
(7)          find a vertex j_1 ∈ R with minimum α(j_1);
(8)          if α(j_1) = ∞ then
(9)              R := ∅
(10)         else
(11)             if j_1 is unmatched then
(12)                 find the path P induced by the vertex sequence (i, ..., β̄(β(j_1)), β(j_1), j_1);
(13)                 ℓ(P) := α(j_1);
(14)                 R := ∅
(15)             else
(16)                 let [i_1, j_1] be the matching edge;
(17)                 label i_1 with (ᾱ(i_1), β̄(i_1)) = (α(j_1), j_1);
(18)                 R := R \ {j_1};
(19)                 Label(i_1);
(20)             end
(21)         end
(22)     end

(23) Procedure Label(i)
(24)     for each neighbor j ∈ R of i do
(25)         if α(j) > max(ᾱ(i), c_ij) then
(26)             α(j) := max(ᾱ(i), c_ij);
(27)             β(j) := i
(28)         end
(29)     end
```

4.4 Problema do emparelhamento estável

4.4.1 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.

4.4.2 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).

4.4.3 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 continue 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 13: Algoritmo de Gale-Shapley para o problema do emparelhamento estável (KLEINBERG; TARDOS, 2005)

```
(1)  Initially all m ∈ M and w ∈ 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
```

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.

BURKARD, R.; DELL'AMICO, M.; MARTELLO, S. *Assignment Problems*. USA: Society for Industrial and Applied Mathematics, 2009. ISBN 0898716632.

CAETANO JULIAN J. MCAULEY, L. C. Q. V. L. T. S.; SMOLA, A. J. 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.

CORMEN, T. H. et al. *Introduction to Algorithms*. 3rd. ed. Cambdrige, MA: The MIT Press, 2009. ISBN 978-0-262-53305-8.

COUR, T.; SRINIVASAN, P.; SHI, J. Balanced graph matching. In: SCHöLKOPF, B.; PLATT, J.; HOFFMAN, T. (Ed.). *Advances in Neural Information Processing Systems*. MIT Press, 2006. v. 19. Disponível em: ⟨https://proceedings.neurips.cc/paper_files/paper/2006/file/d1c373ab1570cfb9a7dbb53c186b37a2-Paper.pdf⟩.

DASGUPTA, C. H. P. S.; VAZIRANI, U. V. *Algorithms*. 1st. ed. New York, NY: McGraw-Hill Education, 2006. ISBN 978-0073523408.

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⟩.

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

HALLER, S. et al. *A Comparative Study of Graph Matching Algorithms in Computer Vision*. 2022. Disponível em: ⟨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⟩.

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

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 SAYAN GHOSH, S. M. F. M. H. M.; SLOTA, G. Efficient weighted graph matching on gpus. *SC24*, IEEE Computer Society, United States, November 2024. ISSN 0162-8828.

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⟩.

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

SUSSMAN, D. L. et al. *Overview of Graph Matching Challenges and Approaches*. 2021. Acessado em: 26 nov. 2024. Disponível em: ⟨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.

YAN, J.; YANG, S.; HANCOCK, E. Learning for graph matching and related combinatorial optimization problems. In: BESSIERE, 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, F. B. M.; 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.