UNIVERSIDADE DE SÃO PAULO
INSTITUTO DE MATEMÁTICA E ESTATÍSTICA

Giovana Gomes Delfino

EMPARELHAMENTOS EM GRAFOS: ALGORITMOS E IMPLEMENTAÇÕES

SÃO PAULO
2017

GIOVANA GOMES DELFINO

EMPARELHAMENTOS EM GRAFOS: ALGORITMOS E IMPLEMENTAÇÕES

Trabalho de Conclusão de Curso submetido à disciplina "MAC0499 - Trabalho de Formatura Supervisionado", sob a orientação do Prof. Carlos Eduardo Ferreira

São Paulo, dezembro de 2017

All women can do wonders if they're put to the test.

Wonder Woman

Resumo

DELFINO, G. D. **Emparelhamentos em Grafos: Algoritmos e Implementações**. Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2017.

Emparelhamentos são úteis em escalonamento de processos e atribuição de tarefas. É possível encontrar um emparelhamento máximo em um grafo qualquer em tempo polinomial. A chave para encontrar um emparelhamento máximo encontra-se no teorema de Berge[2] que propõe a ideia de que basta encontrar caminhos de aumento até que não existam mais. Caminhos de aumento são caminhos que conseguem aumentar um emparelhamento. O que difere um algoritmo do outro é a forma como esses caminhos são encontrados.

Palavras-chave: Grafos, Emparelhamentos, Caminhos de Aumento, Método Húngaro, Hopcroft Karp, Edmonds, Blossom

Lista de Figuras

*

Sumário

Capítulo 1

Introdução

O problema do emparelhamento vem sendo estudado há muito tempo e embora tenha começado com uma motivação puramente acadêmica, tornou-se interessante por poder ser utilizado para resolver problemas comuns do cotidiano, como atribuição de tarefas pessoais, escalonamento de processos, seleção de adversários em competições esportivas e diversas outras categorias.

Grandes matemáticos se interessaram pelo problema e fizeram contribuições bastante significativas, como Claude Berge e o que ficou conhecido como Teorema de Berge que define a base para a resolução do problema, John Hopcroft e Richard Karp que dentre as inúmeras contribuições para o campo de teoria de algoritmos foram responsáveis pela criação de um bastante eficiente para encontrar emparelhamento máximo em grafos bipartidos, Jack Edmonds, estudioso bastante premiado e dentre muitos tópicos foi responsável pela criação do algoritmo para encontrar emparelhamentos máximos em grafos genéricos e muitos outros matemáticos brilhantes direta ou indiretamente fizeram suas contribuições.

O problema se torna ainda mais interessante não só pela enorme quantidade de estudiosos que trabalharam nele, mas por não ser fácil encontrar material em português que una a teoria à prática, ou seja, que explique os conceitos por trás dos algoritmos e mostre implementações práticas dos mesmos, explicando em detalhes e ilustrando sempre que possível.

1.1 Objetivos

Neste trabalho apresentamos algoritmos relacionados ao problema do emparelhamento máximo desde os mais clássicos e simples como o aplicado a grafos bipartidos até os mais complexos e não tão conhecidos como o algoritmo de Edmonds-Blossom.

Diferentemente da maior parte dos livros e outras ferramentas de estudo as implementações aqui apresentadas serão bastante discutidas e a teoria necessária para chegar em tal implementação também será paralelamente introduzida. Todas as ideias apresentadas são acompanhadas de figuras que tentam ilustrar o argumento apresentado e às vezes de figuras que simulam o código em questão. Conforme os algoritmos são apresentados as complexidades são provadas da maneira mais clara e intuitiva possível.

1.2 Estrutura do texto

O capítulo 2 explica o problema do emparelhamento, dando exemplos e motivações, aqui também são especificados as notações básicas que serão usadas no decorrer do trabalho.

O capítulo 3 foca no problema do emparelhamento **máximo** explicando e ilustrando a ideia envolvida para encontrar um. Nesse capítulo também é apresentado o teorema de Berge[2] que é essencial para a resolução do problema, ao final um algoritmo simples que encontra um emparelhamento máximo em grafos bipartidos é apresentado.

Os capítulo 4 e 5 são focados unicamente nos algoritmos de, respectivamente, Hopcroft-Karp e Edmonds-Blossom. Em ambos são apresentandos o passo a passo para deduzir o algoritmo e cada etapa é ilustrada. Para facilitar o entendimento o algoritmo é dividido em várias funções auxiliares que são explicadas separadamente.

O capítulo 6 explica o problema de emparelhamento máximo de peso máximo, apresentando aplicações e motivações, no entanto, a maior parte do capítulo é focada na construção do algoritmo conhecido como Método Húngaro, que utiliza fortemente conceitos de programação linear, no entanto, por não ser o foco do trabalho a teoria de programação linear utilizada não é detalhada a fundo.

Capítulo 2

Emparelhamentos

2.1 Motivação

Suponhamos que você é diretor de uma escola e está tendo dificuldades para definir qual matéria será designada a cada professor. Como as aulas devem ser dadas todas ao mesmo tempo você sabe que não é possível que um professor seja responsável por mais que uma matéria, você também tem em mente que cada professor tem suas limitações, ou seja, cada professor tem conhecimento para ministrar um conjunto específico de matérias. Para melhor visualização do problema, vamos representar essas restrições usando um grafo.

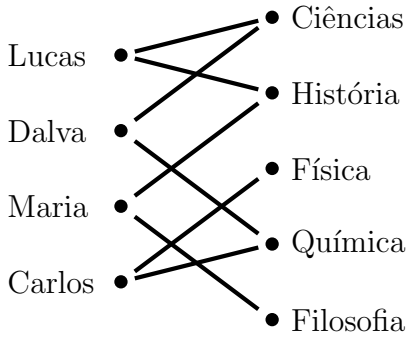

Figura 2.1.1: Arestas indicam que o professor pode dar a matéria a qual está ligado.

Agora com o modelo acima, vamos escolher algumas arestas para indicar que selecionamos um determinado professor para ministrar determinada matéria. Observe a escolha abaixo que não viola nenhuma restrição do problema.

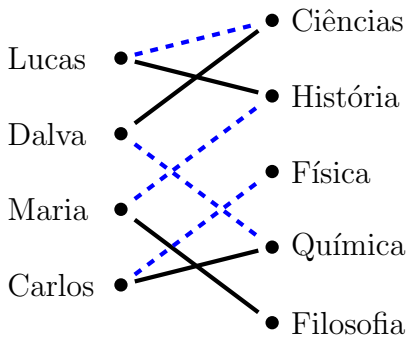

Figura 2.1.2: Arestas tracejadas indicam que o professor de uma de suas pontas dará a disciplina de sua outra ponta.

Definimos a disciplina de alguns professores ao selecionar algumas arestas mostradas no grafo acima, mas temos como justificar que esse foi o melhor jeito de fazer essa distribuição? Existe uma distribuição que permite que todas as matérias sejam oferecidas? Se não existe, qual o número máximo de matérias que podem ser dadas?

Essas perguntas motivam o conteúdo das próximas seções, nas quais estudaremos emparelhamentos e teremos o conhecimento necessário para resolver diversos problemas, como esse que acabamos de ver.

2.2 Definição

Um emparelhamento em um dado grafo é um subgrafo onde não existem arestas que possuem vértices em comum. Neste trabalho consideraremos o emparelhamento apenas como o conjunto de arestas desse subgrafo. Sendo assim, olharemos para um emparelhamento como um subconjunto M de arestas, tais que duas a duas não são adjacentes.

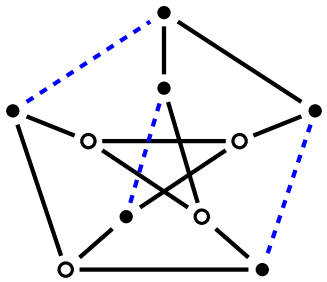

Figura 2.2.1: Arestas tracejadas formam um emparelhamento, mas arestas contínuas não.

Os vértices que são extremos de arestas que estão no emparelhamento são chamados emparelhados, enquanto que os que não estão são chamados livres. Por exemplo, para o emparelhamento formado pelas arestas tracejadas na figura 2.2.1 vértices brancos são livres, enquanto que os pretos são emparelhados.

É fácil notar que um conjunto de arestas $M = \emptyset$ é um emparelhamento, por isso, não temos problemas em encontrar um emparelhamento mínimo. O desafio é encontrar um emparelhamento de cardinalidade máxima, ao qual chamaremos apenas de emparelhamento máximo. Por exemplo, as arestas tracejadas da figura 2.2.1 formam um emparelhamento, mas não um emparelhamento máximo.

2.2.1 Caminhos alternantes

Dado um grafo G e um emparelhamento M definimos um caminho alternante como um caminho onde as arestas se alternam entre arestas que estão em M e as que não estão, isso equivale a dizer que dadas quaisquer duas arestas consecutivas de um caminho alternante, uma está em M e a outra não.

Figura 2.2.2: Exemplo de caminho alternante.

2.2.2 Caminhos de aumento

Ademais, definimos um caminho de aumento ou aumentante como um caminho alternante que começa e termina com vértices livres, o que implica que eles comecem e terminem com arestas que estão fora do emparelhamento.

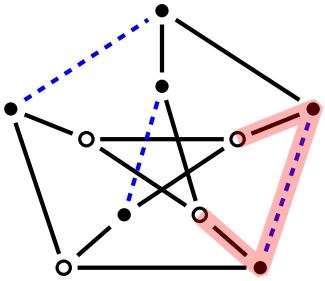

Figura 2.2.3: Exemplo de caminho de aumento.

2.2.3 Emparelhamentos perfeitos

Emparelhamentos perfeitos são aqueles onde todos os vértices estão emparelhados.

2.3 Notações básicas

Usaremos a operação de diferença simétrica $\triangle$ entre dois conjuntos quaisquer X e Y como sendo $X \triangle Y = (X \cup Y) \setminus (X \cap Y)$.

Capítulo 3

Emparelhamento máximo

Nesta seção vamos dar mais atenção para caminhos de aumento e estudar como usá-los para obter um emparelhamento máximo.

Considere o seguinte caminho de aumento:

Figura 3.0.1: Caminho de aumento.

Com esse caminho de aumento se removermos as arestas tracejadas e adicionarmos as contínuas ao nosso emparelhamento, obtemos:

Figura 3.0.2: Caminho alternante resultante da aplicação do algoritmo descrito logo acima.

Perceba que removemos 2 arestas porém adicionamos 3, ou seja, acabamos de descrever uma operação que dado um caminho de aumento conseguimos aumentar a cardinalidade do emprelhamento.

Caminhos de aumento recebem esse nome pois com eles é possível aumentar o número de arestas de um emparelhamento, ou seja, obter um emparelhamento M' que possui mais arestas que M. Vamos formalizar essa operação que acabamos de descrever.

Seja P o conjunto de arestas de um caminho de aumento, então podemos obter um emparelhamento maior fazendo $M' = M \triangle P$. Usando nosso exemplo acima, P equivale a todas as arestas da figura 3.0.1 e M contém as arestas tracejadas da figura 3.0.1. Ao aplicar a operação de diferença simétrica obtemos M' que contém as arestas da figura 3.0.2.

Até aqui podemos ver que se um grafo ainda possui caminho de aumento para um emparelhamento, então não é máximo (já que acabamos de descrever um método que usa um caminho de aumento para aumentar a cardinalidade do emparelhamento). Mas isso não é tudo o que sabemos, o teorema abaixo nos garante a recíproca, ou seja, que se um grafo não possui caminhos de aumento, então o emparelhamento é máximo.

Teorema 3.0.1 (Berge, 1957)**.** [2] Seja G um grafo. M é um emparelhamento máximo em G se e somente se G não possui caminhos de aumento com relação a M.

Demonstração. É fácil ver que se G possui caminhos de aumento então M não é máximo, pois poderíamos aumentar M usando a ideia descrita acima. Sendo assim, nos resta provar que G não possuir caminhos de aumento implica em M ser um emparelhamento máximo.

Seja M^* um emparelhamento máximo e seja M um emparelhamento que não possui caminhos de aumento. Consideremos agora $H = M^* \cup M$. Vemos que H possui circuitos e caminhos alternantes com relação a M.

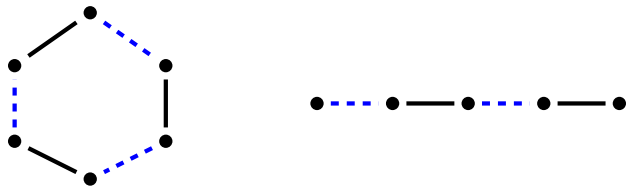

Figura 3.0.3: Arestas tracejadas estão em M e contínuas em M^*.

G não pode possuir circuitos ímpares, pois isso implicaria em duas arestas do mesmo emparelhamento que são adjacentes, logo, os circuitos têm comprimento par, e por isso M e M^* tem a mesma quantidade de arestas nesses circuitos. Os caminhos alternantes de H não podem começar e terminar com arestas de M^* pois senão M^* não seria máximo, ao mesmo tempo que não podem começar e terminar com arestas de M pois sabemos que não existem caminhos de aumento em G. Logo os caminhos de H tem comprimento par e portanto M possui a mesma quantidade de arestas que M^*. Isso implica que $|M| = |M^*|$. □

3.1 Implementação

Com o resultado do teorema 3.0.1 podemos pensar no seguinte algoritmo. Começamos com um emparelhamento $M = \emptyset$ e enquanto existir caminho de aumento P fazemos $M = M \triangle P$.

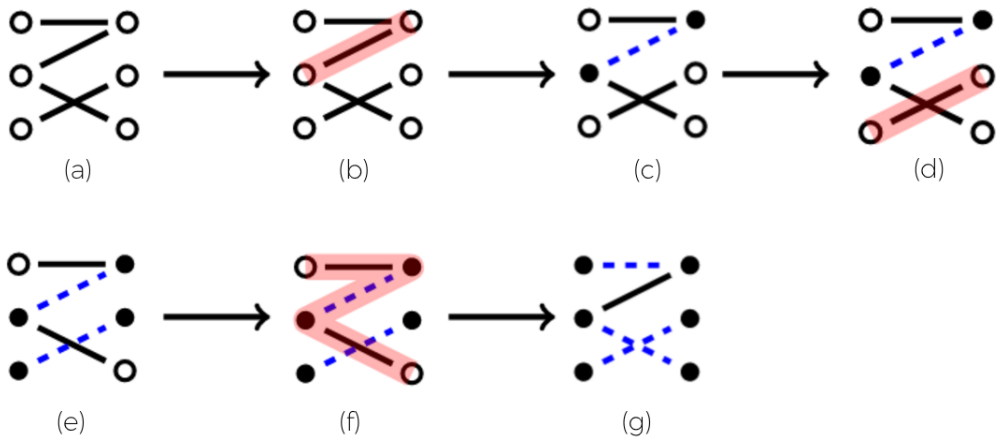

Figura 3.1.1: Visualização do algoritmo descrito acima.

Ao iniciar o algoritmo em (a) temos $M = \emptyset$, encontramos um caminho de aumento P em (b) correspondente a aresta em destaque, aplicamos então a operação de diferença simétrica e conseguimos aumentar M, resultando em (c), o mesmo processo ocorre até (g) onde não há mais caminhos de aumento e podemos, portanto, encerrar o algoritmo.

3.1.1 Grafos bipartidos

Seja G um grafo bipartido com duas partições A e B e um conjunto E de arestas. Para encontrar caminhos de aumento basta fazer uma busca em profundidade (DFS) começando por algum vértice livre. Perceba que não precisamos fazer buscas começando por vértices livres de A e de B, pois como caminhos de aumento têm sempre comprimento ímpar eles começam em uma componente e terminam em outra, ou seja, se começam em vértices livres de A terminam em vértices livres de B e vice-versa. Logo, fazer a busca começando apenas pelos vértices de A, por exemplo, é suficiente.

É importante perceber que quando estamos lidando com grafos bipartidos existe um limitante óbvio para o tamanho do emparelhamento máximo M^*, perceba que $|M^*| = min(|A|, |B|)$, pois como cada vértice só aparece uma vez no emparelhamento e cada aresta do mesmo possui um vértice de A e outro de B é fácil perceber que o emparelhamento máximo tem no máximo a cardinalidade da menor partição de G.

Sabendo disso, já temos o modelo de um algoritmo. Começamos a busca com um vértice u livre de A, visitamos um vizinho v de u, se v for livre temos um caminho de aumento e podemos então aumentar o emparelhamento, senão v está emparelhado e podemos continuar nossa busca a partir do vértice com que v está emparelhado.

Vamos representar o emparelhamento com dois vetores mA e mB onde $(u, mA[u])$ e $(v, mB[v])$ são arestas emparelhadas para todo $u \in A$ e $v \in B$. Pensando na implementação, como um vértice nunca pode estar emparelhado com ele mesmo e como começamos com $M = \emptyset$ é bastante plausível começar mA e mB com $mA[u] = u$ e $mB[v] = v$ para todo

$u \in A$ e todo $v \in B$, respectivamente. Dessa forma, para saber se um vértice $u \in A$ é livre basta checar se $mA[u] = u$.

No pseudocódigo abaixo, a função achaCaminhoAumento recebe um vértice $u \in A$ o grafo bipartido $G = (A, B, E)$, os vetores mA e mB descritos acima e além disso, um vetor $seen$ declarado globalmente no qual $seen[u] = true$ se o vértice u já foi visitado pela busca e $false$ caso contrário, é fácil perceber que $seen$ deve ser inicializado com $false$, já que no início da busca nenhum vértice foi visitado. A função devolve `false` se não encontra um caminho de aumento a partir de u e `true` caso contrário, e nesse caso os vetores mA e mB são atualizados.

```
1: function achaCaminhoAumento(u, G, mA, mB)
2:     for v ∈ adj[u] and seen[v] = false :
3:         seen[v] = true
4:         if mB[v] = v or achaCaminhoAumento(mB[v], G, mA, mB) = true :
5:             mA[u] ← v
6:             mB[v] ← u
7:             return true
8:     return false
```

Para encontrar um emparelhamento máximo, começamos com um emparelhamento qualquer (vazio ou não) e chamamos achaCaminhoAumento para cada vértice livre de A. Perceba que se não encontrarmos um caminho de aumento a partir de um vértice $u \in A$ não faz sentido tentar outra vez mais tarde, pois diferentemente das arestas, os vértices que estão emparelhados nunca deixam de estar emparelhados.

Note também que, ao contrário dos vértices de B, não precisamos marcar os vértices de A que são visitados na busca, pois com exceção da primeira chamada, a função achaCaminhoAumento só é chamada para vértices de A que estão emparelhados.

Agora vamos entender por que o algoritmo sempre para. A cada chamada de achaCaminhoAumento podemos ter dois resultados:

- o algoritmo retorna $false$ o que significa já temos um emparelhamento máximo e portanto terminamos.

- o algoritmo retorna $true$ o que significa que aumentamos o emparelhamento e portanto 2 novos vértices agora estão emparelhados, somando isso ao fato de que vértices do emparelhamento nunca deixam de estar emparelhados e de que temos uma quantidade finita de vértices é fácil ver que o algoritmo sempre para. E pelo item acima se o algoritmo para significa que não temos mais caminhos de aumento e portanto, pelo teorema 3.0.1 temos um emparelhamento máximo.

Logo, o algoritmo funciona.

Cada chamada de achaCaminhoAumento tem complexidade $O(|A|+|B|+|E|)$, já que se trata de uma simples DFS. No pior caso, em cada chamada aumentamos em 1 o número

de vértices de A que estão emparelhados o que nos obriga a chamar a função no máximo $|A|$ vezes. Logo, o algoritmo tem complexidade $O(|A| * (|A| + |B| + |E|))$.

Capítulo 4

Hopcroft-Karp

4.1 Ideia

O método conhecido como Hopcroft-Karp é um algoritmo que encontra um emparelhamento máximo em um dado grafo. Recebe esse nome pois foi apresentado pela primeira vez em 1973[7] por John Hopcroft e Richard Karp.

Assim como o algoritmo citado na seção anterior Hopcroft-Karp também aumenta o tamanho do emparelhamento parcial através de caminhos de aumento, porém ao invés de achar um único caminho de aumento por vez ele encontra um conjunto de caminhos de aumento de tamanho mínimo, ou seja, suponhamos que o menor caminho de aumento do grafo tenha comprimento k, então a cada iteração o algoritmo encontra um conjunto maximal de caminhos de aumento de comprimento k.

A ideia principal é usar uma busca em largura (BFS) de forma a dividir o grafo em camadas que alternam entre arestas que estão no emparelhamento e as que não estão. Por exemplo, considere o seguinte grafo bipartido com um emparelhamento parcial M:

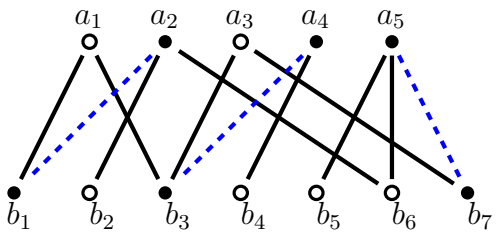

Figura 4.1.1: Exemplo de grafo bipartido no qual as arestas tracejadas formam um emparelhamento parcial, vértices brancos são livres e pretos emparelhados.

Chamemos de A a partição do grafo acima que contém os vértices a_1, a_2, a_3, a_4, a_5 e de B a que contém $b_1, b_2, b_3, b_4, b_5, b_6, b_7$.

O algoritmo de Hopcroft-Karp começa a BFS a partir dos vértices livres de uma das partições e constrói uma ou mais árvores alternantes 6.2 enraizadas em vértices livres de

uma das partições, sem perda de generalidade, vamos escolher a partição A. Assim em uma iteração o algoritmo nos dá a informação aqui representada:

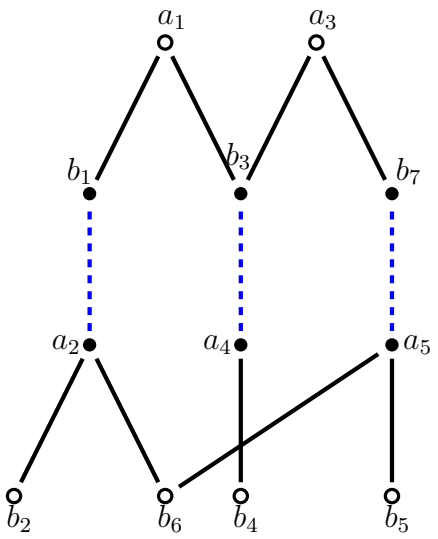

Figura 4.1.2: Exemplo de execução da fase BFS do algoritmo de Hopcroft-Karp.

Vemos então, que existem caminhos de aumento que terminam nos vértices b_2, b_6, b_4, b_5, é nesse ponto que executamos uma DFS a partir de cada um desses vértices, de forma a encontrar um conjunto maximal de caminhos de aumento que sejam disjuntos, sem perda de generalidade, suponhamos que os vértices foram encontrados da esquerda para a direita, assim, nessa etapa do algoritmo, a DFS nos retornaria 2 caminhos de aumento, como ilustrado a seguir.

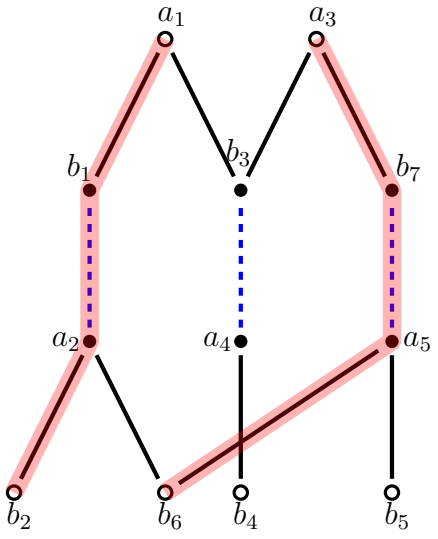

Figura 4.1.3: Exemplo de execução da fase DFS do algoritmo de Hopcroft-Karp.

Perceba que os caminhos destacados formam um conjunto maximal de caminhos de aumento disjuntos, isso significa, que não é possível adicionar um terceiro caminhos que seja de aumento e que seja disjunto desses dois.

Por fim, usamos a operação de diferença simétrica definida anteriormente para aumentar o emparelhamento parcial, resultando em:

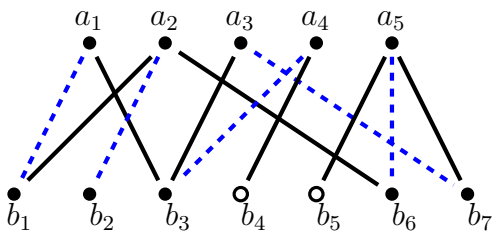

Figura 4.1.4: Exemplo de grafo bipartido no qual as arestas tracejadas formam um emparelhamento parcial, vértices brancos são livres e pretos emparelhados.

4.2 Implementação

Agora que temos a ideia de como o algoritmo funciona, podemos dividir cada iteração do mesmo em 3 partes principais

- 1. BFS a partir dos vértices livres de A para encontrar caminhos de aumentos de comprimento mínimo.

- 2. DFS para encontrar um conjunto maximal de caminhos de aumento a partir dos já encontrados no item anterior.

- 3. Com o resultado do item 2 aumentar o emparelhamento parcial.

Para facilitar o entendimento, vamos dividir a implementção seguindo os três itens acima, para o item 1 vamos fazer uma função ACHACAMINHOSDEAUMENTO que recebe o grafo G e os vetores mA e mB representando o emparelhamento onde $(u, mA[u])$ e $(v, mB[v])$ são arestas emparelhadas para todo $u \in A$ e $v \in B$, caso um vértice não esteja emparelhado representaremos $mA[u] = u$ para todo u livre. A função retorna TRUE caso um caminho de aumento seja encontrado e FALSE caso contrário. No caso de um caminho de aumento ser encontrado a função altera um vetor global d de distâncias.

```
1:  function achaCaminhosDeAumento(G, mA, mB)
2:      queue ← ∅
3:      k ← ∞
4:      for u ∈ A :
5:          if mA[u] == u :
6:              d[u] ← 0
7:              Enqueue(queue, u)
8:          else
9:              d[u] ← ∞
10:     while Empty(queue) == false :
11:         u ← Dequeue(queue)
12:         if d[u] < k :
13:             for v ∈ adj[u] :
14:                 if v == mB[v] and k == ∞ :
15:                     k ← d[u] + 1
16:                 else
17:                     if d[mB[v]] == ∞ :
18:                         d[mB[v]] ← d[u] + 1
19:                         Enqueue(queue, mB[v])
20:     return k != ∞
```

Perceba que a condição da linha 12 garante que não encontraremos caminhos de aumento de tamanho maior que o mínimo encontrado. O restante da função é basicamente uma BFS comum.

Agora com os caminhos de aumento encontrados e com as árvores representadas através do vetor d, podemos a partir de um vértice livre qualquer de A aumentar o emparelhamento através da função AUMENTAEMPARELHAMENTO que recebe um grafo G, os vetores mA e mB representando o emparelhamento e um vértice v de A e aumenta o emparelhamento caso exista um caminho de aumento que começa em v e que seja disjunto aos demais caminhos de aumento já encontrados.

```
1: function AUMENTAEMPARELHAMENTO(G, mA, mB, v)
2:     for u ∈ adj[v] :
3:         if mB[u] == u :
4:             mA[v] ← u
5:             mB[u] ← v
6:             return true
7:         if d[mB[u]] == d[v] + 1 :
8:             if aumentaEmparelhamento(G, mA, mB, mB[u]) == true :
9:                 mA[v] ← u
10:                mB[u] ← v
11:                return true
12:    d[v] ← ∞
13:    return false
```

Nessa etapa do algoritmo estamos usando os caminhos de aumento já encontrados para aumentar o emparelhamento parcial. Perceba que a condição da linha 5 nos garante que nenhum caminho de comprimento maior que o mínimo k será encontrado, pois sabemos que os valores do vetor d são todos menores ou igual a k ou ∞, sendo assim, dado um vértice v tal que $d[v] = k$ não existe vértice u tal que $d[u] == d[v] + 1$, portanto o algoritmo sempre para quando atinge comprimento k.

Outra observação importante é que nenhum vértice é explorado mais que uma vez, para isso temos que analisar 2 casos, o primeiro em que não foi encontrado nenhum caminho de aumento a partir do vértice v e o segundo que foi encontrado.

Primeiro caso: Seja um vértice $v \in A$ para o qual não foi encontrado um caminho de aumento a partir dele, a linha 10 seta $d[v]$ para ∞ e portanto a condição da linha 5 se torna falsa para futuras chamadas.

Segundo caso: Seja um vértice $v \in A$ pelo qual foi encontrado um caminho de aumento então existe $u \in B$ tal que $d[mB[u]] = d[v] + 1$ **antes** da aplicação da diferença simétrica, isso nos mostra que v está um nível acima de u, com a aplicação da diferença simétrica sabemos que $mB[u]$ passa a ser v, portanto qualquer outro vértice w que seja adjacente a u está no mesmo nível que v ou seja, no mesmo nível que $mB[u]$ e portanto a condição da linha 5 nunca é verdadeira.

Os demais detalhes da função são bastante parecidos ao procedimento já descrito em 3.1.1.

Com a função ACHACAMINHOSDEAUMENTO e AUMENTAEMPARELHAMENTO já explicadas, finalizar o algoritmo de Hopcroft-Karp é bastante simples, enquanto ACHACAMINHOSDEAUMENTO retornar **true** usamos AUMENTAEMPARELHAMENTO para aumentar nosso emparelhamento. O algoritmo HOPCROFTKARP é descrito com mais detalhes logo abaixo.

```
1: function HopcroftKarp(G)
2:     for v ∈ A :
3:         mA[v] ← v
4:     for u ∈ B :
5:         mB[u] ← u
6:     matching ← 0
7:     while achaCaminhosDeAumento(G, mA, mB) == true :
8:         for v ∈ A :
9:             if mA[v] == v :
10:                if aumentaEmparelhamento(G, mA, mB, v) == true :
11:                    matching ← mathcing + 1
12:    return matching
```

A função retorna o tamanho do emparelhamento máximo e altera os vetores mA e mB de forma a representar o emparelhamento máximo encontrado.

4.3 Análise de complexidade

O argumento para mostrar que o algoritmo sempre termina é bastante semelhante com o usado na seção anterior. Sabemos que em cada iteração pelo menos um caminho de aumento é encontrado e portanto pelo menos uma aresta é adicionada ao emparelhamento já existente, como estamos considerando apenas grafos com um conjunto finito de vértices e arestas é fácil ver que o algoritmo sempre para. Não tão fácil é perceber a complexidade do algoritmo.

Cada iteração do algoritmo executa apenas uma BFS e uma DFS, assim em um grafo de $|V|$ vértices e $|E|$ arestas, a complexidade de uma iteração é $O(|E| + |V|)$. Seguindo essa ideia, a complexidade das $\sqrt{|V|}$ primeiras iterações é $O(|E|\sqrt{|V|})$.

O algoritmo sempre seleciona um conjunto máximo de caminhos de aumento de comprimento mínimo, sendo assim, uma vez que as $\sqrt{|V|}$ primeiras iterações estejam feitas o comprimento do menor caminho de aumento é de pelo menos $\sqrt{|V|}$.

Similarmente ao argumento usado na demonstração do teorema 3.0.1 chamemos de M^* um emparelhamento máximo e de M o emparelhamento parcial resultante das $\sqrt{|V|}$ primeiras fases do algoritmo. Agora consideramos o conjunto de caminhos e ciclos formado por $S = M^* \Delta M$. Para melhor visualização vamos colorir as arestas de M^* de azuis e as de M de amarelo. É fácil ver que S tem $d = |M^*| - |M|$ arestas azuis a mais que amarelas e portanto existem d caminhos em S que começam e terminam com arestas azuis e esses d caminhos são portanto caminhos de aumento com relação a M, no entanto, na iteração $\sqrt{|V|}$ sabemos que o menor caminho de aumento tem tamanho pelo menos $\sqrt{|V|}$, sendo assim, podem existir no máximo $\sqrt{|V|}$ caminhos de aumento, ou seja, $d \leq \sqrt{|V|}$.

Com o argumento do parágrafo anterior temos que após $\sqrt{|V|}$ iterações, existem no máximo $\sqrt{|V|}$ caminhos de aumento a serem descobertos, ou seja, no máximo o algoritmo

precisará de mais $\sqrt{|V|}$ para finalizar. Isso nos mostra que no total Hopcroft-Karp tem no máximo $2\sqrt{|V|}$ iterações e portanto tem complexidade total de $O(2\sqrt{|V|}(|V| + |E|))$, ou seja, $O(\sqrt{|V|}(|V| + |E|))$.

Capítulo 5

Algoritmo de Edmonds

O algoritmo Blossom de Edmonds consegue encontrar um emparelhamento máximo em um dado grafo genérico em tempo polinomial $O(|V|^2|E|)$. Ele recebe este nome pois foi proposto pela primeira vez por Jack Edmonds em 1961 e publicado em 1965. O algoritmo é também conhecido como Blossom-Edmonds pois usa a ideia de contração de Blossom que será explicada mais adiante.

O algoritmo usa várias ideias mencionadas no algoritmo de Hopcroft-Karp, portanto é fortemente recomendado que a seção 4 seja lida primeiro.

5.1 Ideia e implementação

Vimos nas seções anteriores diferentes formas de encontrar um emparelhamento máximo para grafos bipartidos. O que difere um grafo qualquer G de um bipartido é que G pode conter ciclos ímpares. Por isso, a intuição do algoritmo é tratá-los de uma maneira diferente contraindo-os em um único vértice e assim como o algoritmo da seção 3.1 o algoritmo de Edmonds busca encontrar caminhos de aumento a fim de aumentar o emparelhamento parcial em 1 unidade.

Sendo assim, a base do algoritmo de Edmonds está descrito na função Edmonds-Blossom que recebe um grafo G e um emparelhamento parcial M e retorna um emparelhamento máximo.

```
1: function Edmonds-Blossom(G, M)
2:     P ← AchaCaminhoDeAumento(G, M)
3:     if P.empty() == true :
4:         return M
5:     else
6:         M ← M Δ P
7:         return Edmonds-Blossom(G, M)
```

Embora a função pareça simples o mais complicado está na subfunção AchaCaminho-

DeAumento, que levanta a questão: como achar caminhos de aumento de forma eficiente em um grafo qualquer?

Pelo teorema de Berge 3.0.1 sabemos que se M não é um emparelhamento máximo então existem caminhos de aumentam que ainda não foram encontrados. Assim como no algoritmo de Hopcroft-Karp, Edmonds contrói uma ou mais árvores alternantes (definição 6.2) enraizadas nos vértices livres de G, um conjunto de árvores é denominado floresta, sendo assim a função AchaCaminhoDeAumento recebe o grafo G e um emparelhamento parcial e contrói uma floresta alternante, usada para encontrar os caminhos de aumento.

```
1: function AchaCaminhoAumento(G, M)
2:     F ← ∅
3:     E ← ∅
4:     for v ∈ G :
5:         if v ∉ M :
6:             F.add(v)
7:             raiz[v] ← v
8:             dist[v] ← 0
9:     for e = (v, w) ∈ G :
10:        E.add(e)
11:    for e = (v, w) ∈ M :
12:        E.remove(e)
13:    for v ∈ F :
14:        for  e = (v, w) ∈ E :
15:            if w ∉ F :
16:                AdicionaNaFloresta(M, F, v, w)
17:                dist[w] ← dist[v] + 1
18:            else
19:                if dist[w] % 2 == 0 :
20:                    if raiz[v] != raiz[w] :
21:                        P ←RetornaCaminhoAumento(F, v, w, raiz[])
22:                    else
23:                        P ← RecursaoBlossom(G, M, F, v, w)
24:                    return P
25:            E.remove(e)
26:    return ∅
```

Na função AchaCaminhoDeAumento são explorados os vértices livres v de G. Como explicado anteriormente, esses vértices se tornam raizes de árvores alternantes(definição 6.2) dentro da floresta. E todos os vértices são explorados de forma a sempre adicionar um par de arestas (uma que está e outra que não está no emparelhamento) e assim construir caminhos alternantes, como as arestas são sempre adicionadas em pares são explorados vértices que

estão a uma distância par da raiz.

Um ponto importante a ser percebido é que uma vez que o algoritmo está analisando uma aresta vw 3 casos podem acontecer.

1. A aresta vw não pertence a floresta e portanto deve ser incluída.
2. A aresta vw liga duas árvores alternantes distintas, existe portanto um caminho de aumento que começa no vértice livre da raiz da árvore onde se encontra v e a raiz da árvore onde se encontra w e esse caminho deve ser portanto retornado.
3. A aresta vw liga dois vértices de uma mesma árvore alternante T e portanto a função $T + vw$ forma um ciclo ímpar que precisa ser tratado.

O caso 1 é tratado na linha 15 e utiliza a função auxiliar AdicionaNaFloresta que recebe o emparelhamento parcial M representado de tal forma que $M[v] = w$ se v está emparelhado com w, a floresta F representada por listas de adjacencias tal que $F[u]$ contém todos os vértices ligados a u com a operação $F.add(u)$ que acrescenta a lista de adjacência referente a um novo vértice u inicialmente vazia e os vértices v e w e insere a aresta vw e a aresta com que w está emparelhado na floresta F.

```
1: function AdicionaNaFloresta(M, F, v, w)
2:     x ← M[w]
3:     F[v].add(w)
4:     F.add(w)
5:     F[w].add(x)
6:     raiz[w] = raiz[v]
7:     raiz[x] = raiz[v]
```

No caso 2 temos uma aresta que liga duas árvores alternantes distintas que garantimos com a condição da linha 20.

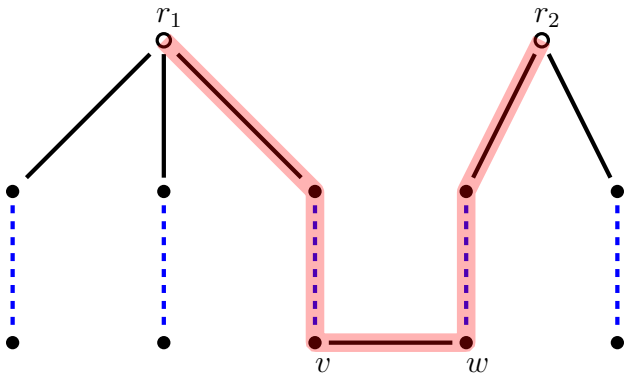

Figura 5.1.1: Representação de como usar a aresta vw para encontrar um caminho de aumento, representado aqui pelo caminho em destaque.

Sabemos que r_1 e r_2 são vértices livres de G pois começamos o algoritmo na linha 5 apenas com vértices livres e exploramos os demais a partir deles. Sendo assim, a aresta vw forma um caminho aumentador que leva de r_1 a r_2, ou seja, de um vértice livre a outro. Na figura 5.1.5 o caminho em destaque representa esse caminho de aumento. A função cumpre então seu objetivo e pode retornar o caminho de aumento encontrado.

Note que se w estiver em F e a uma distância ímpar da raiz a afirmação do parágrafo anterior não á válida, pois o caminho de w até a raiz de árvore de w começaria com uma aresta fora do emparelhamento, no entanto a aresta vw também não está no emparelhamento, quebrando assim a hipótese de ser um caminho alternante.

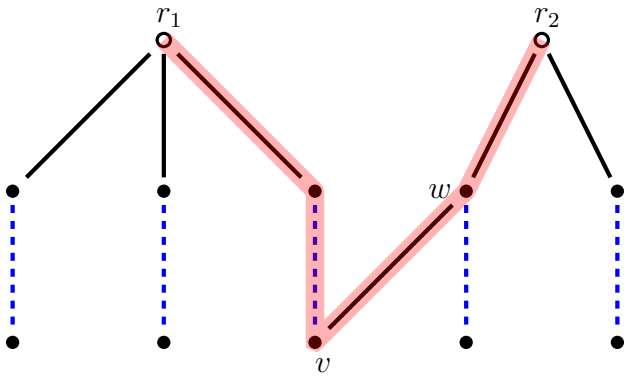

Figura 5.1.2: Na imagem, como w tem distância ímpar com a raiz veja como o caminho em destaque não forma um caminho de aumento.

Por isso essa situação é eliminada pela condição da linha 19 que considera apenas os w com distância par até a raiz, por isso, o caso 2 é verdadeiro na linha 20 e usa a função auxiliar RetornaCaminhoAumento, que recebe a floresta F, os vértices v e w e o vetor $raiz$ tal que $raiz[v]$ contém a raiz da árvore que contém o vértice v.

```
1: function RetornaCaminhoAumento(F, v, w, raiz[])
2:     P1 ← PegaCaminho(raiz[v], v)
3:     P2 ← PegaCaminho(w, raiz[w])
4:     return P1 + P2
```

A função usa como função auxiliar PegaCaminho(v, w), que recebe um vértice v e um vértice w retorna o menor caminho de v a w, ou seja, a função é basicamente uma BFS que retorna um caminho.

```
1: function PegaCaminho(v, w)
2:     for u ∈ G :
3:         seen[u] ← false
4:     pai[v] ← v
5:     vertices.push(v, v)
6:     while vertices.size() > 0 :
7:         (x, pai-de-x) ← vertices.pop()
8:         if seen[x] == false :
9:             pai[x] ← pai-de-x
10:            seen[x] ← true
11:            if x == w :
12:                while x != v :
13:                    P.add(x)
14:                    x ← pai[x]
15:                P.add(x)
16:                return P.reverse()
17:            for u ∈ adj[x] :
18:                if seen[u] == false :
19:                    vertices.add(u, x)
```

O caso mais interessante é sem dúvida o caso 3 no qual a aresta vw liga dois vértices de uma mesma árvore, e como v está a uma distância par de sua raiz pela forma que foi contruído e w também está a uma distância par da sua raiz, temos um cíclo ímpar de caminhos alternantes, a esse tipo de ciclo damos o nome blossom.

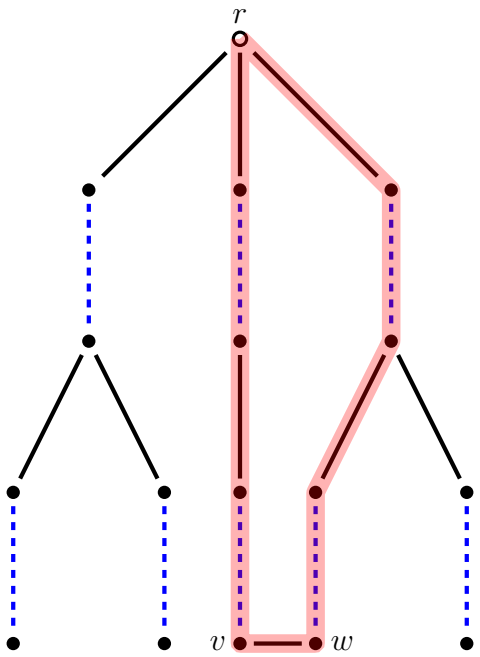

Figura 5.1.3: O caminho destacado forma um blossom.

É importante perceber que o algoritmo detecta blossoms corretamente, pois se AchaCaminhoDeAumento atinge a linha 23 então sabemos que o vértice v está na lista dos vértices com distância par da raiz e o vértice w também pertence à floresta (pela linha 15), além disso, os dois pertencem à mesma árvore (linha 20) e estão a distancia par da raiz. Como ambos estão a uma distância par da raiz existe um caminho alternante de comprimento par entre v e w, isso é garantido pelo fato de que as arestas são sempre adicionadas em pares na floresta. Por isso, adicionar a aresta vw que não está emparelhada fecha esse caminho par e cria um ciclo ímpar. No ciclo, existe um vértice u que será extremidade de duas arestas que não estão no emparelhamento (pois o ciclo é impar), enquanto todos os outros vértices terão uma extremidade em uma aresta que está no emparelhamento e outra em uma que não está.

Note que se w tivesse uma distância ímpar até a raiz o ciclo teria três arestas consecutivas que não estão no emparelhamento e portanto esse ciclo não seria um blossom. No entanto, eliminamos essa possibilidade pela condição da linha 19.

Sabendo disso, podemos detalhar a função auxiliar usada para detectar blossoms na linha 23. A função RecursaoBlossom recebe o grafo G, o emparelhamento parcial M, a floresta F e os vértices v e w que satisfazem as condições citadas anterioremente e retorna um caminho de aumento considerando o grafo com o blossom comprimido.

```
1: function RecursaoBlossom(G, M, F, v, w)
2:     B ← PegaCaminho(v, w) + v
3:     G' ← ContraiBlossomDoGrafo(G, B, w)
4:     M' ← ContraiBlossomDoEmparelhamento(M, B, w)
5:     P' ← AchaCaminhoAumento(G', M')
6:     if w ∈ P' :
7:         P ← Lift(B, P')
8:         return P
9:     else
10:        return P'
```

A função usa como auxiliar ContraiBlossomDoGrafo(G, B, w) que consiste em basicamente copiar o grafo G porém substituindo os vértices de B por w resultando em G'.

```
1: function ContraiBlossomDoGrafo(G, B, w)
2:     for v ∈ G :
3:         for u ∈ adj[v] :
4:             if u ∈ B :
5:                 u ← w
6:             if v ∈ B :
7:                 v ← w
8:             if v != u  and u ∉ adj[v] :
9:                 adj[v].add(u)
```

A outra função auxiliar ContraiBlossomDoEmparelhamento(M, B, w) segue a mesma ideia do algoritmo acima só que aplicado ao emparelhamento M.

A idéia é imaginar G' como um grafo cujo blossom em questão foi eliminado e substituído pelo vértice w. Por exemplo:

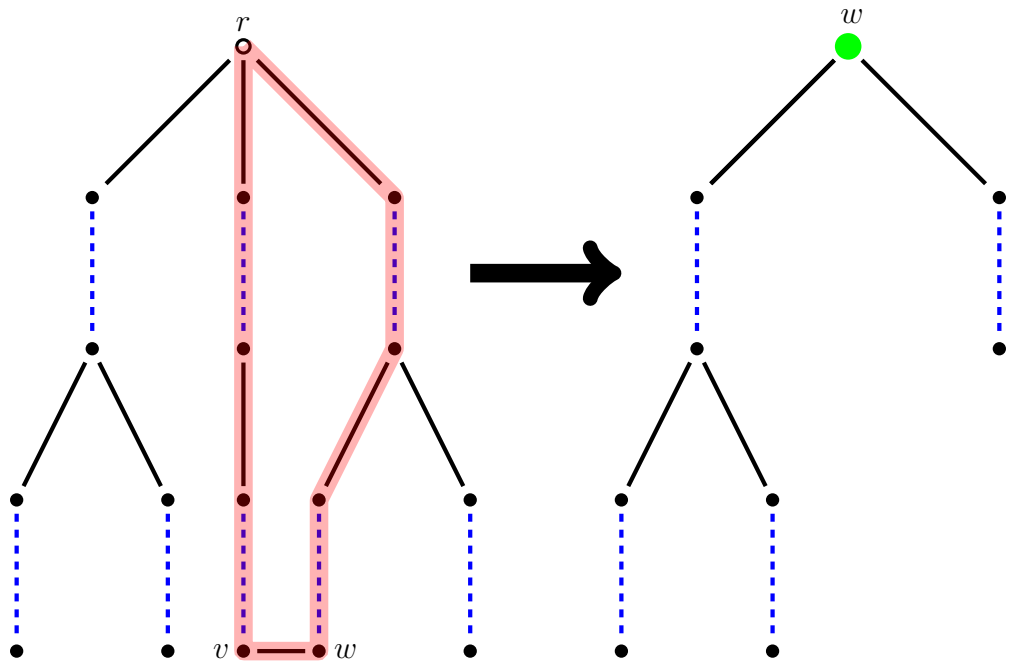

Figura 5.1.4: A imagem mostra o processo de compressao de um blossom em um único vértice w representado pelo vértice em destaque na árvore da direita.

Para melhor explicar o processo de lifting, precisamos da definição de base do blossom. A **base do Blossom** é o vértice v tal que é o único vértice do blossom que possui incidência de duas arestas que estão fora do emparelhamento. Na figura 5.1.3 a base do blossom é a raiz r.

Depois de comprimir o blossom em um único vértice w a função procura por um caminho de aumento P', quando encontrado existem 2 casos:

1. P' não contém w e portanto pode ser retornado.

2. P' contém w e portanto w precisa ser descomprimido para que os vértices que formam o blossom que w representa apareçam no caminho.

O caso 1 não tem muito segredo, a condição da linha 6 checa se w está no caminho encontrado, e se não estiver apenas retorna P'.

O caso 2 é um tanto mais complicado, como já explicado, se w está em P' precisamos descomprimir w para expandir o caminho, chamamos esse processo de $lifting$. Para conseguir fazer esse processo, precisamos analisar 2 casos principais:

1. w está em um dos extremos de P'.

2. w está em algum lugar do meio de P'.

A partir de agora, para facilitar a explicação precisamos de uma nova definição, chamaremos as arestas que incidem em w de *stems*. Além disso, para fazer o processo de lifting, precisamos analisar a quais vértices do blossom os *stems* são incidentes. Para isso considere o lemma abaixo.

Lemma 5.1.1. Se w não é um extremo de P' então um dos dois *stems* incide na base.

Demonstração. Por hipótese w não é um extremo de P', perceba então que pelo menos um dos *stems* pertence ao emparelhamento M, já que eles fazem parte de um caminho de aumento em G'. O *stem* que não está em M pode ser incidente a qualquer vértice do grafo original G, no entanto, o *stem* emparelhado deve obrigatoriamente ser incidente a base, pois caso contrário o vértice com o qual o *stem* estivesse ligado estaria conectado a duas arestas de M, o que contradiz a hipótese de M ser um emparelhamento.

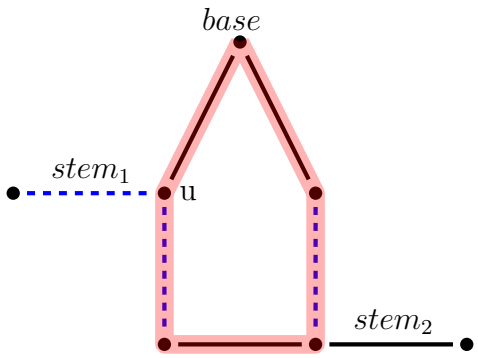

Figura 5.1.5: Note como $stem_1$ só pode estar ligado a base, caso contrário, o vértice u seria incidido por duas arestas do emparelhamento e por definição M não seria um emparelhamento.

□

No caso 2 temos uma aresta que liga duas árvores alternantes distintas, ou seja, temos a presença de dois *stems*, e portanto, fazer o lift do blossom significa substituir w pela porção do blossom que está entre os dois *stems*. Como o blossom tem comprimento ímpar, perceba que que existem dois jeitos de fazer isso, pegar o caminho par ou o ímpar entre os dois *stems*. Como queremos que o caminho de aumento resultante permaneça um caminho de aumento temos que inserir a P' o caminho par do blossom, pois os *stems* alternam entre emparelhado e não emparelhado. Esse processo transforma um caminho de aumento em G' em um caminho de aumento expandido em G.

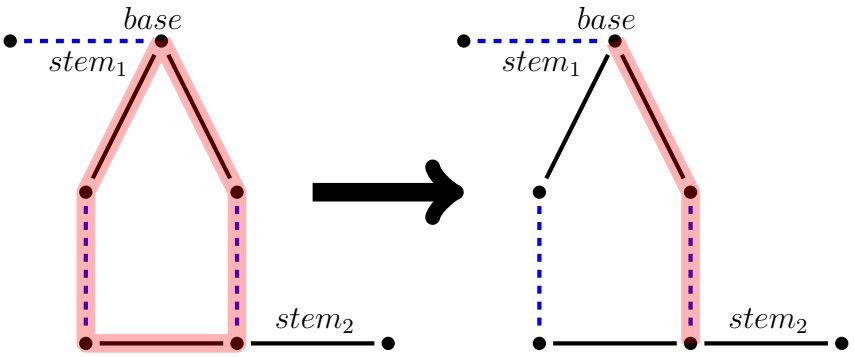

Figura 5.1.6: O lado esquerdo mostra em destaque o blossom, o lado direito mostra em destaque o pedaço do blossom que deve ser incluído em P' conforme o procedimento mencionado acima no caso 2.

No caso 1, w é um extremo do caminho de aumento P', ou seja, w possui apenas um *stem* que obrigatoriamente é não emparelhado, pois arestas extremas de um caminho de aumento são sempre não emparelhadas. Nesse caso então percorremos o blossom seguindo a aresta emparelhada que segue o *stem* e continuamos fazendo isso até encontrar a base, assim garantimos que a última aresta do caminho de aumento expandido continua sendo não emparelhada, e portanto, continua sendo um caminho de aumento em G. É importante notar que se o *stem* é incidente a base, não fazemos nada.

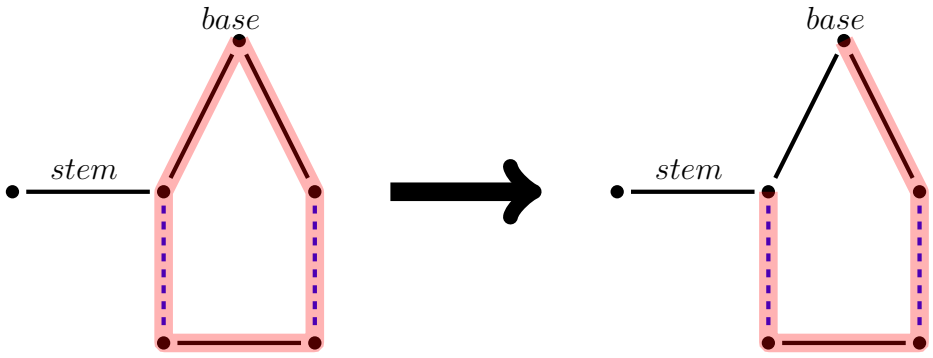

Figura 5.1.7: O lado esquerdo mostra em destaque o blossom, o lado direito mostra em destaque o pedaço do blossom que deve ser incluído em P' conforme o procedimento mencionado acima no caso 1.

5.2 Análise de complexidade

Para começar vamos observar que o algoritmo base EDMONDS-BLOSSOM(G, M) executa no máximo $\lfloor \frac{|V|}{2} \rfloor$ chamadas da função ACHACAMINHOAUMENTO(G, M), pois o número máximo de arestas no emparelhamento de um grafo com $|V|$ vértices é $\lfloor \frac{|V|}{2} \rfloor$, além disso,

sabemos que a cada chamada de AchaCaminhoAumento(G, M) adiciona exatamente uma aresta no emparelharemento, sendo assim, começando com um emparelhamento vazio a função é chamada no máximo $\lfloor \frac{|V|}{2} \rfloor$ vezes.

A cada iteração de AchaCaminhoAumento percorremos no máximo todos os vértices do grafo, e para cada vértice, exploramos apenas as arestas ainda não visitadas, sendo assim, conforme já explicado, para cada vértice v e uma aresta não visitada vw 3 situações podem acontecer:

1. **Adiciona na Floresta:** Para cada aresta vw, se existir $wx \in M$ adicionamos vw e wm na floresta, o que tem custo $O(1)$ e pode ser visto na função AdicionaNaFloresta. No pior caso, fazemos essa operação para todas as arestas, logo, o custo total é $O(|E|)$.

2. **Retorna Caminho Aumento:** Essa função é chamada apenas quando um caminho de aumento foi encontrado, e após executada ela retorna ao algoritmo base Edmonds-Blossom. Sendo assim, dentre todas as chamadas de AchaCaminhoAumento a função RetornaCaminhoAumento pode ser chamada apenas uma vez. Como para retornar o caminho de aumento realizamos apenas uma busca simples no grafo no pior caso percorremos o grafo todo, portanto, a função tem custo $O(|V|)$.

3. **Recursão Blossom:** Para analisar esse caso, precisamos fazer duas observações importantes.

   Observação 1: precisamos saber quantas contrações de blossom são feitas para cada chamada de AchaCaminhoAumento. Como blossoms tem tamanho ímpar, sabemos que cada contração de blossom reduz em no mínimo 2 o número de vértices do grafo, ou seja, no máximo podemos ter um total de $\lfloor \frac{|V|}{2} \rfloor$ contrações de blossom. Isso significa que cada iteração de AchaCaminhoAumento não pode gastar mais de $O(|V|)$ com contrações de blossom. Por exemplo, observe a figura 5.2.1.

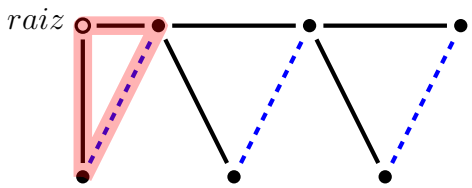

Figura 5.2.1: Imagem mostra uma sequência de blossoms a serem encontrados, o primeiro deles em destaque.

   No primeiro passo o primeiro blossom de tamanho 3 será contraído e se tornará a base do segundo blossom, que será recursivamente contraído e assim por diante até que o grafo todo esteja contraído, e nesse ponto o processo de lifting começa, "desmembrando" os blossoms um por um até o caminho estar completo, perceba como a soma dos custos de todos os processos de lifting dão no máximo $O(|V|)$, já que os blossoms

não se intersectam e cada processo de lifting percorre apenas os vértices do blossom em questão.

Observação 2: Também precisamos ter em mente que cada contração percorremos todos os vértices e arestas do grafo e renomeamos para o identificador do blossom, percorrer todas os vértices e arestas do grafo e também o emparelhamento custa um total de $O(|V| + |E|)$, ou seja, $O(|E|)$.

Com isso, para calcular o custo total, precisamos multiplicar o custo de cada procedimento pelo total de iterações. Assim temos:

$$Total = \underbrace{O(|V|)}_{\text{Iterações}} * \Big[\underbrace{O(|E|)}_{\text{Caso 3}} + \Big(\underbrace{O(|E|)}_{\text{Caso 1}} + \underbrace{O(|E|)}_{\text{Caso 2}} \Big) * \underbrace{O(|V|)}_{\text{Recursao Blossom}} \Big]$$

O que resulta em $O(|V|^2|E|)$.

Capítulo 6

Emparelhamento máximo de peso máximo

Imagine a seguinte situação, n medicamentos estão sendo pesquisados em um estudo clínico. Para observar os efeitos desses medicamentos em seres humanos, foram selecionados m voluntários onde cada voluntário possui uma certo limite para cada medicamento, ou seja, cada voluntário pode receber no máximo uma quantidade pré-determinada de mililitros de cada medicamento. Suponhamos que temos os voluntários A, B e C e os medicamentos X, Y, Z, e W e que os valores limites de cada voluntário estejam tabelados a seguir:

Voluntário	X	Y	Z	W
A	50	0	100	0
B	90	0	0	20
C	0	5	0	40

No entanto, a fim de evitar reações provocadas pela mistura de duas ou mais drogas queremos que cada paciente seja medicado com apenas um medicamento. De forma a tornar o experimento ótimo, queremos que a soma das quantidades injetadas em cada paciente seja a maior possível, sempre respeitando as restrições citadas acima.

Podemos modelar esse problema como um grafo onde existe aresta entre voluntário e medicamento se determinado voluntário pode receber determinada droga e mais, agora cada aresta terá como valor a quantidade limite de medicamento que cada voluntário pode receber da mesma.

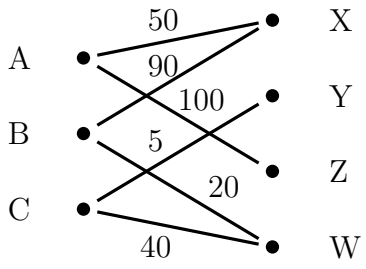

Figura 6.0.1: Grafo representando estudo clínico

Perceba que no grafo acima um emparelhamento máximo possível seria o seguinte.

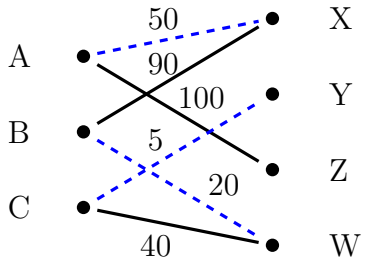

Figura 6.0.2: Arestas tracejadas formam um possivel emparelhamento máximo.

No entanto esse emparelhamento não dos dá soma máxima, a solução ótima nesse caso seria:

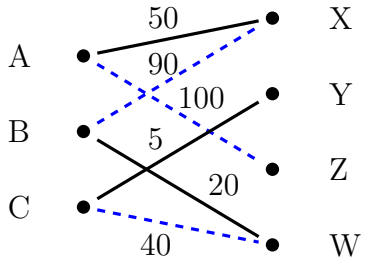

Figura 6.0.3: Arestas tracejadas formam um emparelhamento ótimo para o problema

Note que um emparelhamento máximo qualquer não é necessariamente um emparelhamento ótimo para o problema.

Da mesma forma como no exemplo acima, quando estamos trabalhando com grafos que possuem uma função peso $w\colon E \to \mathbb{R}$ nas arestas pode ser interessante buscar um emparelhamento cuja soma dos pesos de suas arestas seja máxima, a esse problema damos o nome de emparelhamento máximo de peso máximo.

Para nos ajudar a resolver esse problema, precisamos escrevê-lo como um problema de programação linear, para isso, usamos uma variável, x_{ij} tal que $x_{ij} = 1$ se a aresta ij está

em no emparelhamento M e 0 caso contrário. Assim, podemos fazer a seguinte formulação primal do problema:

$$\text{Maximizar} \quad \sum_{ij \in E} w(ij)x_{ij} \tag{6.1}$$

$$\text{sujeito a} \quad \sum_{j \in V} x_{ij} = 1, \text{ para todo } i \in V \tag{6.2}$$

$$\sum_{e \in E(S)} x_e \leq \frac{|S|-1}{2}, \ \forall \ S \ \subseteq V \ com \ |S| \ ímpar \tag{6.3}$$

$$x_{ij} \geq 0, \text{ para todo } ij \in E \tag{6.4}$$

Onde $E(S)$ é o conjunto de todas as arestas que tem ambas as pontas em vértices de S.

A equação 6.2 precisa existir pois temos que garantir que todo vértice de G possua no máximo uma aresta do emparelhamento incidente a ele. No entanto, perceba que se a equação 6.3 não existisse, poderíamos ter uma solução não inteira, considere o circuito da imagem abaixo, se considerarmos uma solução x_{ij} de cada aresta como $\frac{1}{2}$, claramente valem as restrições 6.2 e 6.4 e essa seria uma solução viável, o que não é o que procuramos.

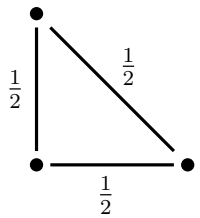

Figura 6.0.4: Grafo representando uma solução não inteira.

Agora, precisamos mostrar que o problema modelado acima de fato corresponde a um emparelhamento em um grafo qualquer G. Para isso, considere um emparelhamento qualquer M em um dado grafo G, agora considere o vetor x' associado a M, ou seja, um vetor onde x'_{ij} para todo $ij \in G$ que vale 1 se ij está em M e 0 caso contrário, como M é um emparelhamento, claramente x satisfaz 6.2, além disso, x possui apenas valores inteiros maiores ou iguais a 0, logo, x satisfaz 6.3 e 6.4. Assim x é uma solução viável do problema definido acima. Da mesma forma, agora considere x solução viável. Já foi provado que além de viável, existe solução inteira para o problema mas não vamos entrar em detalhes nessa demonstração, assim, assumimos x viável inteira, como x é inteiro e satisfaz a condição 6.2, x claramente corresponde a um emparelhamento. Logo, as restrições 6.2, 6.3 e 6.4 modelam perfeitamente um emparelhamento.

Um detalhe importante que torna a solução desse problema um tanto quanto inviável é que a quantidade de restrições geradas por 6.3 é exponencial, pois para cada circuito ímpar no grafo precisamos adicionar uma restrição que envolve os vértices desse circuito. Isso é

computacionalmente bastante ineficiente. A seguir veremos que ao trabalhar com um tipo específico de grafo esse processo se torna menos complicado.

6.1 Grafos bipartidos

Quando consideramos G bipartido (A, B, E) sem perda de generalidade, assumimos G completo e com $|A| = |B|$, caso o grafo não possua essas condições podemos satisfazê-las adicionando vértices e arestas com custo nulo.

Assim, o modelo primal do problema se torna mais interessante, uma vez que o grafo não possui ciclos ímpares e portanto a restrição 6.3 não é mais necessária. Além disso, ao resolvermos o problema de programação linear obtemos uma solução inteira. Podemos então reescrever o primal como:

$$\text{Maximizar} \quad \sum_{ij \in E} w(ij) x_{ij} \tag{6.5}$$

$$\text{sujeito a} \quad \sum_{j \in B} x_{ij} = 1, \text{ para todo } i \in A \tag{6.6}$$

$$\sum_{i \in A} x_{ij} = 1, \text{ para todo } j \in B \tag{6.7}$$

$$x_{ij} \geq 0, \text{ para todo } ij \in E \tag{6.8}$$

Seja $x \in \mathbb{R}^{|E|}$ um vetor cujos elementos são as variáveis x_{ij} para todo $ij \in E$. Então, x é uma solução viável se satisfaz 6.6, 6.7 e 6.8. Se x é viável e ainda maximiza 6.5 dizemos que x é solução ótima do primal.

Da forma como modelamos o primal, se escolhermos $M = \{ij : x_{ij} = 1\}$ para uma solução viável x vale que M é um emparelhamento perfeito e que se x é solução ótima então M é um emparelhamento perfeito de peso máximo.

Agora vamos escrever o dual desse problema. No dual, introduziremos uma variável y_i para cada $i \in A$ e z_j para cada $j \in B$. Assim temos:

$$\text{Minimizar} \quad \sum_{i \in A} y_i + \sum_{j \in B} z_j \tag{6.9}$$

$$\text{sujeito a} \quad y_i + z_j \geq w(ij), \text{ para todo } ij \in E \tag{6.10}$$

Da mesma forma feita anteriormente, vamos considerar um vetor $y \in \mathbb{R}^{|A|}$ e um vetor $z \in \mathbb{R}^{|B|}$ contendo as variáveis do dual. Sabemos que se (y, z) satisfazem 6.10 então (y, z) é uma solucão viável do dual.

Utilizando a teoria de programação linear, se temos x viável no primal e (y, z) viável no dual então

$$\sum_{ij \in E} w(ij)x_{ij} \leq \sum_{i \in A} y_i + \sum_{j \in B} z_j. \tag{6.11}$$

A inequação acima é conhecida como teorema da dualidade fraca e ela vale por igualdade se x é ótimo no primal e (y, z) é ótima no dual. Dado (y, z) viável no dual, vamos definir o subgrafo de G como $G_{yz} = (A, B, E_{yz})$ onde $E_{yz} = \{ij \in E \colon y_i + z_j = w(ij)\}$.

Teorema 6.1.1 (Folgas Complementares)**.** [6] Seja (y, z) viável no dual, se M é um emparelhamento perfeito em G_{yz} então M é emparelhamento de peso máximo em G.

Demonstração. Seja M^* um emparelhamento perfeito de peso máximo em G.

$$\sum_{ij \in M^*} w(ij) \leq \sum_{i \in A} y_i + \sum_{j \in B} z_j = \sum_{ij \in M} y_i + z_j = \sum_{ij \in M} w(ij).$$

A primeira desigualdade segue de (y, z) ser viável e a primeira igualdade segue do fato de todo $i \in A$ e $j \in B$ estar em M^* já que o mesmo é perfeito. Assim, temos que M é um emparelhamento com mesmo peso que M^*, logo, é máximo. □

6.2 Método Húngaro

Nesta seção apresentamos o Método "Húngaro", um algoritmo que resolve o problema de encontrar um emparelhamento máximo de peso máximo em um grafo dado. O método foi proposto em 1955 por Harold Kuhn, que deu este nome pois seu método se baseava no trabalho anterior de Dénes Kőnig e Jenő Egerváry, dois brilhantes matemáticos húngaros.

O Método Húngaro é um algoritmo que, levando em consideração as definições feitas na seção anterior, começa com uma solução viável (y, z) para o dual e um emparelhamento em G_{yz}, e a partir disso, o algoritmo tenta encontrar um emparelhamento perfeito em G_{yz}, se não consegue, altera (y, z) de forma a mantê-lo como uma solução viável e ainda adicionar arestas em G_{yz} sem perder as que já existem e, então, repetir o procedimento.

Como solução (y, z) inicial, defininos $y_i = \max_{i \in A} w(ij)$ para todo $i \in A$ e $z_j = 0$ para todo $j \in B$. E como emparelhamento inicial definimos $M = \emptyset$.

Uma forma menos eficiente e mais simples de implementar o método húngaro é utilizar o algoritmo de caminhos de aumento descrito na seção 3.1.1. Basta garantir que o algoritmo percorra apenas as arestas de G_{yz}. Para cada aresta $ij \notin E_{yz}$ guardamos um valor associado $\delta_{ij} = y_i + z_j - w(ij)$, podemos ver δ_{ij} como o quanto falta subtrair de y_i ou z_j para que $y_i + z_j$ seja igual a $w(ij)$, ou seja, o quanto falta subtrair para que a aresta ij entre em E_{yz}. Para atualizar (y, z) definimos $\delta = \min_{ij \notin E_{yz}} \{\delta_{ij}\}$ e para cada $i \in A$ visitado pela busca, subtraímos δ de y_i, da mesma forma, para cada $j \in B$ visitado pela busca, adicionamos δ de z_j.

É fácil notar que as arestas que já estão em E_{yz} continuam existindo, pois a soma $y_i + z_j$ não sofre alteração. Além disso, o valor de δ é escolhido de forma a ser o menor valor capaz de adicionar uma nova aresta em G_{yz}.

Usando essa implementação, no pior caso, cada atualização de (y, z) adiciona apenas uma aresta no grafo. Portanto, pode ser necessário executar n^2 buscas resultando em complexidade $O(n^4)$.

Veremos a seguir, que é possível implementar o método húngaro em tempo $O(n^3)$, o segredo está em guardar resultados das buscas anteriores, para que não precisem ser feitas novamente. Para isso, precisamos introduzir o conceito de árvore alternante.

Uma árvore alternante é uma árvore enraizada em um vértice livre de A na qual qualquer caminho da raíz até as pontas é um caminho alternante. Vamos denotar S e T como o conjunto de vértices de A e de Bque estão na árvore. Veja um exemplo, na imagem a seguir.

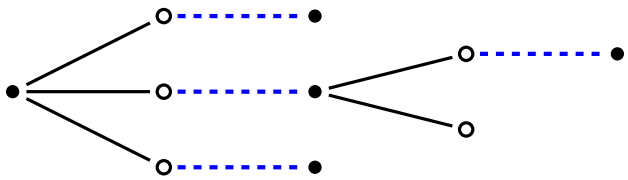

Figura 6.2.1: Árvore alternante.

Na imagem acima, arestas tracejadas estão em M, vértices pretos estão em S e brancos em T. Além de manter as informações pré-calculadas usando uma árvore alternante, com o intuito de deixar essa implementação mais eficiente vamos mudar a forma como (y, z) é atualizada. Considere os conjuntos S e T da árvore alternante que não possui caminhos de aumento, com $T \neq B$. Definimos

$$\delta = \min_{i \in S,\, j \in B \setminus T} (y_i + z_j - w_{ij}).$$

O novo vetor y' será

$$y'_i = \begin{cases} y_i - \delta, & \text{se } i \in S \\ y_i, & \text{se } i \in A \setminus S \end{cases}$$

Analogamente, z' será

$$z'_j = \begin{cases} z_j + \delta, & \text{se } j \in T \\ z_j, & \text{se } j \in B \setminus T \end{cases}$$

Assim temos uma nova solução (y', z').

Da mesma forma que a atualização anterior, é fácil perceber que $E_{yz} \subseteq E_{y'z'}$ e que $|E_{yz}| < |E_{y'z'}|$.

Pensando em facilitar a implementação, usaremos um vetor d onde $d_j = \min_{i \in S}(y_i + z_j - w(ij))$ para todo $j \in B \setminus T$. Da mesma forma, definimos $\delta = \min_{j \in B \setminus T} d_j$. Basta atualizar d toda vez que um novo vértice for inserido em S.

A função ATUALIZADUAL abaixo, recebe (y, z) soluções viáveis no dual, o vetor d e os conjuntos de vértices S e T e retorna os os vetores (y, z) e d atualizados.

```
1: function atualizaDual(y, z, d, S, T)
2:     δ ← min_{j∈B\T} d_j
3:     for all i ∈ S : y_i ← y_i − δ
4:     for all j ∈ B :
5:         if j ∈ T :
6:             z_j ← z_j + δ
7:         else
8:             z_j ← z_j − δ
9:     return (y, z) e d
```

O algoritmo abaixo recebe um grafo $G = (A, B, E)$ e o peso nas arestas w e devolve um emparelhamento de peso máximo e utiliza a notação de vizinhança como $N_{yz}(X)$ sendo o conjunto dos vértices de B que são adjacentes a algum vértice em X no grafo G_{yz}, sendo X um subconjunto de A.

```
1: function MÉTODOHÚNGARO(G, w)
2:     inicializa (y, z) e M
3:     while M não é perfeito em G_yz :
4:         i ← i ∈ A livre
5:         S ← {i}
6:         T ← {∅}
7:         atualiza d
8:         while Enquanto não encontrar caminho de aumento :
9:             if N_yz = T :
10:                y, z, d ← atualizaDual(y, z, d, S, T)
11:            j ← j ∈ N_yz(S) \ T
12:            if j é livre :
13:                P ← caminho de i a j
14:                M ← M △ P
15:                volte para linha 3
16:            else
17:                i' ← vértice ao qual j está emparelhado
18:                S ← S ∪ {i'}
19:                T ← T ∪ {j}
20:                atualiza d
21:    return M
```

Na linha 9 vale que não é possível aumentar a árvore alternante, e por isso, os valores de (y, z) e d devem ser atualizados, pois vamos adicionar mais vértices em S. Na linha 12 foi encontrado um caminho de aumento então basta aumentar nosso emparelhamento e para isso modificamos a raíz da árvore alternante, devemos então retornar a linha 3 para começar com uma nova. Como o algoritmo começa com um emparelhamento vazio, o loop da linha 3 é repetido exatamente $|A|$ vezes.

A atualização do vetor d nas linhas 7 e 20 consomem tempo $O(|A|)$, pois basta verificar dos vértices da componente em que está o novo vértice inserido. O loop da linha 8 é repetido no máximo $|A|$ vezes pois toda vez que não encontra um caminho de aumento, um novo vértice é adicionado a T na linha 19. Logo, a complexidade final de tempo do algoritmo é $O(|A|^3)$.

Capítulo 7

Códigos e aulas

Todos os algoritmos explicados nesse trabalho foram implementados em C++ e podem ser encontrados em https://github.com/gidelfino/MAC0499/tree/master/codigos e na página da disciplina https://linux.ime.usp.br/ gidelfino/mac0499/.

Além disso, para cada assunto foram gravadas aulas explicando cada conteúdo que estão disponíveis no youtube em:

Emparelhamento máximo: https://youtu.be/05TGeZWira0
Hopcroft Karp: https://youtu.be/DHk6yF-oNyI
Edmonds:https://youtu.be/3fYEGJaOJi4
Método Húngaro: https://youtu.be/VR9TXIG0MLA

O link para as aulas também pode ser encontrado na página da disciplina.

Capítulo 8

Conclusão

O desenvolvimento do presente estudo possibilitou conhecer, entender e implementar algoritmos eficientes que resolvem o problema do emparelhamento, permitindo a resolução de problemas como atribuição de tarefas pessoais, escalonamento de processos entre muitos outras aplicações práticas.

Mais que tudo, esta dissertação permite o entendimento dos conceitos básicos e essenciais para resolução do problema apresentado facilitando o entendimento de generalizações como o problema do b-emparelhamento.

Referências Bibliográficas

[1] Carlos E. Ferreira e Yoshiko Wakabayashi, *Combinatória Poliédrica e Planos-de-Corte Faciais*, , Instituto de Matemática e Estatística.

[2] Claude Berge, *Two theorems in graph theory*, Proceedings of the National Academy of Sciences of the United States of America.

[3] William J. Cook, William H. Cunningham, William R. Pulleyblank, Alexander Schrijver, *Combinatorial Optimization*

[4] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest e Clifford Stein *Introduction to Algorithms* 3a edição.

[5] L. Lovász e M. D. Plummer *Matching Theory* Annals of Discrete Mathematics

[6] Chvátal, V. (1983) *Linear Programming*

[7] John E. Hopcroft e Richard M. Karp *SIAM Journal on Computing* An $n^{\frac{5}{2}}$ algorithm for maximum matchings in bipartite graphs.