Universidade de São Paulo
Instituto de Matemática e Estatística
Bacharelado em Ciência da Computação

Animação de algoritmos sobre grafos cobertos por emparelhamentos

Guillermo Enrique Junchaya Heredia

Monografia Final

mac 499 — Trabalho de Formatura Supervisionado

Supervisor: Prof. Dr. Cláudio Leonardo Lucchesi

São Paulo
2023

Pra não dizer que não falei das flores.

— Geraldo Vandré

Resumo

Guillermo Enrique Junchaya Heredia. **Animação de algoritmos sobre grafos cobertos por emparelhamentos**. Monografia (Bacharelado). Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2023.

Grafos cobertos por emparelhamentos são grafos conexos tais que cada uma das suas arestas pertence a algum emparelhamento perfeito desse grafo. O algoritmo mais eficiente conhecido para decidir se um grafo é coberto por emparelhamentos foi descoberto por Carvalho e Cheriyan. Esse algoritmo é baseado no algoritmo de Edmonds, um algoritmo eficiente para achar um emparelhamento máximo em um grafo qualquer. Neste trabalho, apresentamos implementações do algoritmo de Edmonds e do algoritmo de Carvalho e Cheriyan. Essas implementações vêm acompanhadas de uma biblioteca gráfica, o que permite gerar animações desses algoritmos.

Palavras-chave: emparelhamentos, grafos cobertos por emparelhamentos, animação de algoritmos.

Abstract

Guillermo Enrique Junchaya Heredia. **Matching covered graphs algorithms animation**. Capstone Project Report (Bachelor). Institute of Mathematics and Statistics, University of São Paulo, São Paulo, 2023.

Matching covered graphs are connected graphs such that each of its edges belong to a perfect matching of this graph. The most efficient algorithm known for deciding if a graph is matching covered was discovered by Carvalho and Cheriyan. This algorithm is based upon the algorithm of Edmonds, an algorithm for finding a maximum matching in an arbitrary graph. In this study, we present implementations of the algorithms of Edmonds, and of Carvalho and Cheriyan. These implementations were made with a graphic library, which allows the generation of animations for these algorithms.

Keywords: matchings, matching covered graphs, algorithm animations.

Sumário

Capítulo 1

Introdução

A teoria de grafos teve seu início em 1736, quando Euler utilizou uma estrutura para resolver o "Problema das Pontes de Königsberg" [6]. Em 1891, Julius Petersen, famoso pelo grafo do mesmo nome, publicou "*Die Theorie der regulären graphs*" [18], o primeiro artigo com resultados fundamentais da teoria de grafos escritos explicitamente na linguagem da teoria dos grafos. Certamente, alguns resultados anteriores foram descritos sem usar claramente a definição de grafo. Por exemplo, os importantes resultados em teoria de grafos publicados pelo físico alemão Kirchoff estavam escritos em termos de redes elétricas.

No "*Die Theorie der regulären graphs*", Petersen enunciou o teorema que leva o seu nome: todo grafo cúbico sem pontes tem um emparelhamento perfeito. Em 1931, Kőnig provou o importante teorema *minimax* conhecido como Teorema de Kőnig. Esse teorema enuncia que, em um grafo bipartido, um conjunto mínimo de vértices que cobre o grafo tem a mesma cardinalidade que um emparelhamento máximo do mesmo. Também em 1931, Egerváry, outro matemático húngaro, provou uma generalização do Teorema de Kőnig para grafos bipartidos ponderados, mas não proporcionou um algoritmo.

Baseado nos trabalhos de Kőnig e Egerváry, Kuhn, em 1956, descobriu um algoritmo eficiente para resolver o Problema de Atribuição (a versão com pesos do problema do emparelhamento bipartido). Kuhn batizou esse algoritmo como Método Húngaro devido as contribuições fundamentais das ideias do Kőnig e do Egerváry. Contudo, ele não foi o primeiro em descobrir esse algoritmo. No século 19, muito antes dos trabalhos de Kőnig e Egerváry, Jacobi descreveu um algoritmo equivalente ao Método Húngaro para resolver o Problema de Atribuição. Em "A tale of three eras: The discovery and rediscovery of the Hungarian Method" [11], escrito pelo próprio Kuhn, o leitor pode encontrar a história da descoberta desse algoritmo.

Um algoritmo eficiente para resolver o problema do emparelhamento em grafos não bipartidos foi desenvolvido só uma década depois. Em 1965, Edmonds generalizou o Método Húngaro para grafos qualquer no famoso artigo "*Paths, trees and flowers*" [5]. Nesse algoritmo, Edmonds empregou a contração de certos subgrafos aos quais chamou de *blossoms* (corolas). Por tal motivo, esse algoritmo é também conhecido como Algoritmo Blossom. Esse artigo, e um outro publicado por Cobham [3], são considerados pelos cientistas da

computação como contribuições fundamentais para a teoria de Complexidade de Algoritmos devido à introdução da classe $\mathcal{P}$, a classe de problemas que admitem algoritmos polinomiais. Outros algoritmos mais eficientes para o problema do emparelhamento máximo foram achados posteriormente, como o algoritmo de Hopcroft e Karp [8] ou o algoritmo de Micali e Vazirani [13]. Para ler uma narrativa histórica mais ampla e detalhada da teoria de emparelhamentos, consulte o artigo de Michael Plummer [19].

A teoria de grafos cobertos por emparelhamentos, uma área particular da teoria de emparelhamentos, é especialmente interessante. Um grafo conexo é dito *coberto por emparelhamentos* se tem pelo menos uma aresta e todas as suas arestas pertencem a um emparelhamento perfeito do grafo. A primeira aparição da noção de grafos cobertos por emparelhamentos está no artigo "*The Factorization of Linear graphs*" de Tutte, de 1947 [22]. Nesse artigo, Tutte enunciou o famoso teorema conhecido como Teorema de Tutte. Esse teorema é uma generalização do Teorema de Hall para grafos não necessariamente bipartidos. Como corolário, Tutte provou uma versão mais forte do Teorema de Petersen: todo grafo cúbico 2-conexo é coberto por emparelhamentos.

É fácil pensar em algoritmos eficientes para decidir se um grafo conexo é coberto por emparelhamentos pois basta verificar, para cada aresta, se pertence a um emparelhamento perfeito. O algoritmo para resolver esse problema mais eficiente conhecido é devido a Carvalho e Cheriyan [2]. Esse algoritmo usa de forma simples várias repetições do algoritmo de Edmonds e resultados da teoria de emparelhamentos para conseguir seu objetivo. No artigo deles, utilizam esse algoritmo como subrotina de um outro algoritmo para decompor grafos cobertos por emparelhamentos em orelhas. O leitor pode encontrar no prefácio do livro de Lucchesi e Murty [12] um percurso detalhado pela história da teoria de grafos cobertos por emparelhamentos.

1.1 Preliminares

Nesta seção, estabelecemos as definições necessárias para entender os problemas e os algoritmos que serão descritos ao longo do texto. Ademais, formalizamos algumas das definições já feitas na seção anterior.

1.1.1 Definições

Um *grafo* $G = (V, E)$ é um conjunto de *vértices* V e *arestas* E tal que cada aresta de E incide em exatamente dois vértices de V, os *extremos* dessa aresta.

Se um grafo contém mais de uma aresta com os mesmos extremos, dizemos que o grafo tem *arestas múltiplas*. Um *laço* é uma aresta cujos extremos são o mesmo vértice. Chamamos de *grafo simples* a um grafo livre de arestas múltiplas e laços.

Seja $G = (V, E)$ um grafo.

O grafo $H = (V', E')$ é um *subgrafo* de G se $V' \subseteq V$ e $E' \subseteq E$.

Um *passeio* $W = (w_1, w_2, \ldots, w_k)$ de um grafo G é uma sequência de vértices em V e é tal que, para todo i, $1 \leq i < k$, existe uma aresta em E cujos extremos são w_i e w_{i+1}. O vértice w_1 é chamado de *origem* do passeio e o vértice w_k é chamado de *término* do passeio.

Um *caminho* é um passeio cujos vértices são todos distintos. Um ciclo $C = (w_1, w_2, \dots, w_1)$ é um passeio cujos vértices são todos distintos, à exceção do primeiro e do último vértices da sequência que são, necessariamente, iguais.

Uma partição $\{A, B\}$ dos vértices de G é uma *bipartição de* G se cada aresta de E tem um extremo em A e o outro em B. Um grafo é *bipartido* se admite uma bipartição. É bem conhecido que um grafo é bipartido se e somente se não contém ciclos ímpares.

Dizemos que G é *conexo* se, para cada par de vértices de G, existe um passeio que os contém. Uma *componente conexa* (ou, simplesmente, *componente*) de G é um subgrafo conexo maximal de G. Em outras palavras, não é possível adicionar mais vértices nem arestas de G à componente sem quebrar a condição de conexidade. Assim, todo grafo é formado por uma ou mais componentes conexas.

O grafo G é uma *árvore* se é conexo e $|E| = |V| - 1$. O grafo G é uma *floresta* se suas componentes são todas árvores.

Um subgrafo de G que é uma árvore é chamado de *subárvore* de G. Um subgrafo de G que é uma floresta é chamado de *subfloresta* de G.

Um *emparelhamento* de G é um conjunto $M \subseteq E$ tal que, em cada vértice em V, incide no máximo uma aresta de M. O conjunto M é um *emparelhamento máximo* se não existe nenhum outro emparelhamento de maior cardinalidade em G. Um emparelhamento M de G é *perfeito* se todo vértice de G é extremo de uma aresta de M. Uma aresta e é *emparelhável* se existe um emparelhamento perfeito em G que contém e. Um grafo é *coberto por emparelhamentos* se é conexo, tem pelo menos uma aresta, e todas as suas arestas são emparelháveis.

Problema 1.1 (O problema do emparelhamento bipartido).
Determine um emparelhamento máximo de um grafo bipartido dado.

Problema 1.2 (O problema do emparelhamento máximo).
Determine um emparelhamento máximo de um grafo dado qualquer.

Problema 1.3 (O problema dos grafos cobertos por emparelhamentos).
Determine se um grafo dado qualquer é coberto por emparelhamentos.

1.1.2 Notação

Usamos notações usuais de conjuntos. Contudo, por simplicidade, é comum escrever $A - B$ para a diferença de conjuntos em vez de $A \setminus B$. No caso de conjuntos unitários, escrevemos $A - x$ em vez de $A - \{x\}$ e $A + x$ em vez de $A \cup \{x\}$.

Uma operação importante de conjuntos que usamos é a *diferença simétrica*, representada pelo símbolo $\triangle$. Definimos a diferença simétrica de dois conjuntos A e B como $A \triangle B = (A - B) \cup (B - A)$. Ou seja, a diferença simétrica de dois conjuntos é a união de todos os elementos que pertencem a exatamente um dos dois conjuntos.

Sejam G um grafo e X um conjunto de vértices de G. O *subgrafo de G induzido por X*, denotado por $G[X]$, é o grafo H tal que $V(H) = X$ e $E(H)$ é o conjunto de arestas de G que têm ambos os extremos em X.

Para um grafo G e um subconjunto X dos vértices de G, denotamos por $\partial(X)$ ao conjunto de arestas de G tais que um dos seus extremos é um vértice de X e o outro extremo é um vértice que não está em X. Quando o conjunto X é formado por apenas um vértice, digamos v, escrevemos $\partial(v)$ em vez de $\partial(\{v\})$.

Sejam $G = (V, E)$ e $B \subseteq V$. Definimos $H = G/(B \longrightarrow b)$ como o grafo resultante da *contração do conjunto de vértices* B em um único vértice, b, removidas as arestas com ambos os extremos em B. O vértice b é chamado de *vértice da contração de* H. Quando o nome do vértice de contração não é relevante, escrevemos simplesmente G/B.

1.2 Estrutura do texto

No Capítulo 2, estudamos algoritmos para resolver os Problemas 1.1 e 1.2 além da teoria necessária para entendê-los. Detalharemos, principalmente, o algoritmo de Edmonds.

No Capítulo 3, estudamos dois algoritmos para resolver o Problema 1.3. Nos referimos ao primeiro desses algoritmos como algoritmo ingênuo pela sua simplicidade que, embora elegante, não consegue atingir uma eficiência muito boa. Já o segundo algoritmo, mais eficiente, é o descoberto por Carvalho e Cheriyan e a maior parte do capítulo é dedicada ao seu estudo.

O foco do Capítulo 4 está na implementação do software desenvolvido neste trabalho. Neste capítulo, principalmente, explica-se como instalar e usar as diferentes funcionalidades do programa. Além disso, são discutidos alguns detalhes da implementação dos algoritmos e suas animações, o desenvolvimento de testes e também são documentadas as animações geradas pelo programa.

Finalmente, no Capítulo 5 comentamos sobre a possibilidade de trabalhos futuros que possam expandir este projeto.

Capítulo 2

Emparelhamentos máximos

O objetivo deste capítulo é apresentar ao leitor o algoritmo de Edmonds. Por tal motivo, apresentamos, primeiramente, a teoria necessária para entender tal algoritmo.

2.1 O teorema de Berge

Seja $G = (V, E)$ um grafo e $M \subseteq E$ um emparelhamento de G. Um caminho (ou ciclo) *M-alternante* de G é um caminho (ou ciclo) que não tem duas arestas consecutivas fora de M. Ou seja, alterna uma aresta em M com uma que não está em M. Quando o emparelhamento está implícito, escrevemos apenas caminho (ou ciclo) alternante. Às vezes temos dois emparelhamentos, M e N, e as arestas de um caminho (ou ciclo) estão alternadamente em M e em N. Nesse caso, dizemos que o caminho (ou ciclo) é *(M, N)-alternante*.

Os vértices de G que são extremos de alguma aresta de M são vértices *cobertos* (por M). Os demais vértices são *expostos* (por M). Vamos denotar por $V(M)$ os vértices cobertos por M. Assim, $V(G) - V(M)$ é o conjunto dos vértices expostos por M em G.

Um caminho *M-aumentante* de G é um caminho M-alternante cujos extremos são vértices expostos.

Seja P um caminho M-aumentante. Note que $M' = M \triangle E(P)$ é um emparelhamento de G e que $|M'| = |M| + 1$. É precisamente por tal motivo que tais caminhos são chamados de aumentantes.

Podemos aumentar o tamanho de um emparelhamento através de diferenças simétricas com o conjunto de arestas de caminhos aumentantes. Intuitivamente, o emparelhamento conseguido com esse processo não deveria ser necessariamente máximo, apenas maximal. Contudo, essa intuição está errada como podemos ver no seguinte teorema.

Teorema 2.1 (Berge, [1]). *Um emparelhamento M de um grafo G é máximo se e somente se não existem caminhos M-aumentantes em G.* □

Apesar desse teorema ser usualmente atribuído a Berge, Petersen já tinha observado essa propriedade dos emparelhamentos máximos mais de 60 anos antes [14].

É imediato construir um algoritmo para o problema do emparelhamento máximo a partir deste teorema. Contudo, não é trivial, apenas usando esse teorema, desenvolver um algoritmo eficiente, pois o número de caminhos em um grafo pode ser exponencial em relação ao número de vértices.

2.2 Deficiência de emparelhamentos

Denotamos por $O(G)$ o conjunto de componentes ímpares (ou seja, com um número ímpar de vértices) do grafo G e denotamos por $o(G)$ a cardinalidade do conjunto $O(G)$.

Teorema 2.2 (Tutte, [22]). *Um grafo G é emparelhável se e somente se a desigualdade $o(G - S) \leq |S|$ vale para todo conjunto S de vértices de G.* □

A *deficiência* de um emparelhamento M de um grafo G, escrita como $\text{def}_G(M)$, é o número de vértices expostos de G expostos por M. Se G estiver subentendido então simplesmente escrevemos $\text{def}(M)$.

Proposição 2.3. *Seja M um emparelhamento do grafo G e seja S um conjunto de vértices de G tal que $\text{def}(M) = o(G - S) - |S|$. Então M é um emparelhamento máximo e todos os vértices de S são cobertos por M.*

Demonstração. Seja K uma componente de $O(G - S)$. Como $|V(K)|$ é ímpar, é impossível emparelhar todos os vértices de K usando apenas arestas de K. Para emparelhar todos os vértices de K, é necessário que pelo menos um desses vértices esteja emparelhado com um vértice de S. Como isso vale para todas as componentes de $O(G - S)$, então temos que, para qualquer emparelhamento, temos no mínimo $o(G - S) - |S|$ vértices expostos.

Suponha que $\text{def}(M) = o(G - S) - |S|$ e seja N um emparelhamento qualquer de G. Como foi visto, $\text{def}(N) \geq o(G - S) - |S| = \text{def}(M)$. Assim, $\text{def}(N) \geq \text{def}(M)$. Esta conclusão vale para todo emparelhamento N de G. Logo, M é máximo. □

Desta forma, um conjunto S de vértices que satisfaz as condições da Proposição 2.3 para um emparelhamento M é um *certificado* da otimalidade de M. Ainda mais, podemos verificar se um subconjunto cumpre essas condições eficientemente. Apesar disso, a proposição por si só não nos ajuda a encontrar um conjunto com tais características. Formalmente, a Proposição 2.3 implica que o problema do emparelhamento máximo está em $\mathcal{NP}$, mas não afirma nada sobre se está em $\mathcal{P}$ ou não. Da mesma forma, o Teorema 2.1 indica que o problema do emparelhamento máximo está em co-$\mathcal{NP}$. Assim, o problema está em $\mathcal{NP} \cap$ co-$\mathcal{NP}$. No restante do capítulo, vamos verificar que o problema de fato admite um algoritmo polinomial.

2.3 Vértices supérfluos

Um vértice v de um grafo G é *supérfluo* se G tem um emparelhamento máximo que deixa v exposto. Assim, se v é supérfluo então G tem um emparelhamento máximo que também é um emparelhamento (máximo) de $G - v$.

Proposição 2.4. *Seja M um emparelhamento máximo de um grafo G. Um vértice v de G é supérfluo se e somente se G tem um caminho M-alternante de comprimento par, cuja origem é um vértice exposto por M e cujo término é v.*

Demonstração.
($\Longrightarrow$) Suponha que G tem um caminho P de r a v, que é M-alternante, tem comprimento par e cuja origem r é exposta por M. Então $M \triangle E(P)$ é um emparelhamento máximo de G que deixa v exposto. Portanto, v é supérfluo.

($\Longleftarrow$) Suponha que v é um vértice supérfluo de G. Se v é exposto por M, então (v) é um caminho trivialmente M-alternante de comprimento par cuja origem é exposta por M e cujo término é v. Podemos então supor que v é coberto por M. Dado que v é supérfluo, deduzimos que G tem um emparelhamento máximo N que deixa v exposto. O conjunto de arestas $M \triangle N$ induz uma coleção $\mathcal{C}$ de caminhos e ciclos (M, N)-alternantes. Dado que M e N são ambos máximos, nenhum caminho em $\mathcal{C}$ tem comprimento ímpar, pois qualquer caminho de comprimento ímpar de $\mathcal{C}$ seria ou M-aumentante ou N-aumentante. Dado que v é exposto por N e coberto por M, então v é o término de algum caminho (M, N)-alternante (de comprimento par). A origem desse caminho é exposta por M (e coberta por N). □

Proposição 2.5. *Seja M um emparelhamento máximo de um grafo G e seja S um certificado da otimalidade de M. Todo vértice supérfluo de G pertence a alguma componente ímpar do grafo $G - S$.*

Demonstração. Seja v um vértice de G que não pertence a nenhuma componente ímpar de $G - S$, seja $H := G - v$ e seja F um emparelhamento de H. Vamos demonstrar que $\text{def}_H(F) > \text{def}_G(M)$. Seja $T := S - v$.

Se v está em uma componente K de $G - S$, e K é par, então, $T = S$ e o grafo $K - v$ contém um número ímpar de componentes ímpares; assim, temos que $O(G-S) \subset O(H-T)$ e, portanto,

$$o(H - T) - |T| \geq o(G - S) + 1 - |S|.$$

Por outro lado, se $v \in S$, então $O(H - T) = O(G - S)$ e $|T| = |S| - 1$. Portanto,

$$o(H - T) - |T| \geq o(G - S) - |S| + 1.$$

Em ambos os casos, $o(H - T) - |T| > o(G - S) - |S|$. Pela Proposição 2.3,

$$\text{def}_H(F) \geq o(H - T) - |T| > o(G - S) - |S| = \text{def}(M).$$

Concluímos que $\text{def}_H(F) > \text{def}(M)$. Assim, F não é um emparelhamento máximo de G. Esta conclusão vale para todo emparelhamento perfeito F de H. Portanto, v não é supérfluo. □

A recíproca da Proposição 2.5 não é verdadeira. Por exemplo, vamos considerar um quadrilátero $Q := C_4$. Seja (A, B) a bipartição de Q. Certamente nenhum vértice de Q é

supérfluo, pois Q é emparelhável, Entretanto B é um certificado da otimalidade de qualquer emparelhamento máximo de Q e os dois vértices de A induzem as duas componentes ímpares de $Q - B$.

Contudo, como veremos na Seção 2.5, o algoritmo de Edmonds obtém um emparelhamento máximo M de um grafo dado G e um certificado S da otimalidade de M em G com a seguinte propriedade adicional: um vértice é supérfluo em G se e somente se pertence a alguma componente ímpar de $G - S$. Esse certificado é dito *especial*. Por exemplo, se um grafo G é emparelhável então o conjunto vazio é um certificado especial de qualquer emparelhamento perfeito de G. No caso de um ciclo ímpar, o vazio também é um certificado especial.

2.4 O algoritmo de Kőnig

O algoritmo de Kőnig acha um emparelhamento máximo de G, quando G é bipartido. Alguns autores chamam esse algoritmo de algoritmo de Egerváry, de algoritmo de Kuhn ou de Método Húngaro. Contudo, os trabalhos de Egerváry e de Kuhn estavam focados na versão ponderada do problema de emparelhamento bipartido. Além disso, como podemos ver na tradução de Szárnyas [21] do artigo "Graphok és matrixok" do Kőnig [10], já a prova original do Teorema do Kőnig é construtiva e, dessa prova, pode-se derivar o algoritmo descrito nesta seção. Por tais motivos, no presente trabalho, nos referimos a este algoritmo como algoritmo de Kőnig.

O algoritmo de Kőnig trabalha em fases. No começo de cada fase, temos um emparelhamento M de G e vamos buscar um caminho M-aumentante P. Se encontrarmos tal caminho, começamos uma nova fase com o emparelhamento $M \triangle P$. Caso contrário, temos que M é um emparelhamento máximo.

Durante a execução do algoritmo, vamos manter uma *floresta M-alternante* de G. Seja T uma subárvore de G e r um vértice de T, chamado de *raiz* de T. Dizemos que T é uma *árvore M-alternante* se todos os vértices de T, com exceção de r, são cobertos por $M \cap E(T)$. Assim, a raiz é exposta por M em G. Veja a Figura 2.1. Em particular, se r é um vértice de G exposto por M então o grafo vértice induzido por r é uma árvore M-alternante. Para cada vértice v de T, denotamos por $P_T(v)$ o caminho (único) de r a v em T. Uma subfloresta de G é M-alternante se cada uma de suas árvores é M-alternante.

Por simplicidade, costuma-se começar o algoritmo com um emparelhamento vazio, mas também é possível começar com um emparelhamento qualquer. Em cada fase, executamos os seguintes passos. Vamos denotar por (A, B) uma bipartição de G, tal que $|A| \geq |B|$.

1. Suponhamos que todos os vértices de A são cobertos por M. Nesse caso, como $|A| \geq |B|$, deduzimos que o grafo é emparelhável e portanto o conjunto vazio é um certificado especial de M.

2. Uma invariante do algoritmo é uma subfloresta M-alternante F de G em que as raízes de todas as árvores estão em A. Inicialmente, temos $F := (A_0, \emptyset)$, onde A_0 é o conjunto de vértices de A que são expostos por M. O algoritmo mantém também um conjunto de *arestas visitadas*, que é vazio inicialmente.

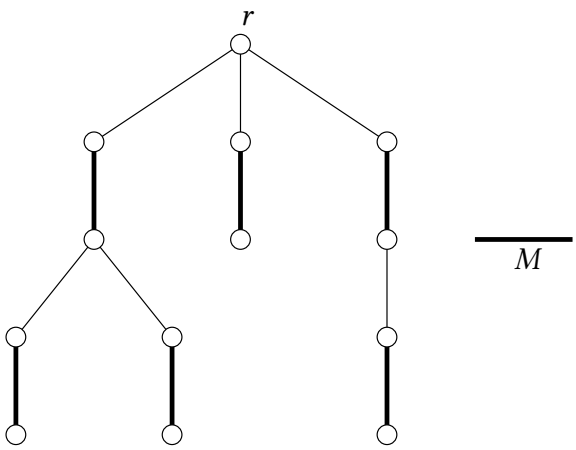


Figura 2.1: *Uma árvore M-alternante com raiz r.*

3. Seja u um vértice em A de uma árvore T de F e $e := uv$ uma aresta incidente em u que não foi visitada. A aresta então é adicionada ao conjunto das arestas visitadas. Temos os seguintes casos:

   (a) O vértice v pertence a F. Nesse caso, ignoramos a aresta e e repetimos o passo 3.

   (b) O vértice v não pertence a F mas é coberto por M. Como $u \in A$, então ou u é exposto em G por M ou u incide em uma aresta de $M \cap E(F)$. Em ambos os casos, $e \notin M$. Portanto, existe uma aresta $f := vw \in M$ tal que v é um dos seus extremos. O vértice w não pertence a F, pois todos os vértices de F cobertos por M são cobertos por arestas de $M \cap E(F)$.

   Acrescentamos então os vértices v e w e as arestas e e f a à árvore F. Adicionamos a aresta f ao conjunto de arestas visitadas. Note que $v \in B$ e $w \in A$. Repetimos o passo 3.

   (c) O vértice v não pertence a F mas é exposto por M. Seja T a árvore de F que contém o vértice u e seja r a raiz de T. Lembramos que $P_T(u)$ denota o caminho em T de r a u. Obtemos o caminho M-aumentante P adicionado a $P_T(u)$ a aresta e. Aumentamos o emparelhamento fazendo $M := M \triangle P$ e começamos uma nova fase, no passo 1.

4. Se todas as arestas incidentes nos vértices de F em A foram visitadas então M é máximo. De fato, seja $S := (B \cap V(F)) \cup (A - V(F))$. Veja a Figura 2.2. Então,

   (a) nenhuma aresta de G liga um vértice de $A \cap V(F)$ a um vértice de $B - V(F)$, portanto todos os vértices de $G - S$ são isolados em $G - S$ e então $o(G - S) = |V(G) - S|$;

   (b) todos os vértices de $B \cap V(F)$ são cobertos por arestas de $M \cap E(F)$;

   (c) todos os vértices de $A - V(F)$ são cobertos por arestas de $M - E(F)$;

   (d) logo, todo vértice de S está emparelhado com um vértice de $G - S$ por M;

   (e) portanto, há $|V(G) - S| - |S|$ vértices expostos por M em G e então o conjunto S

é um certificado da otimalidade de M. Veja a Proposição 2.3. Note, porém, que esse certificado não é necessariamente especial.

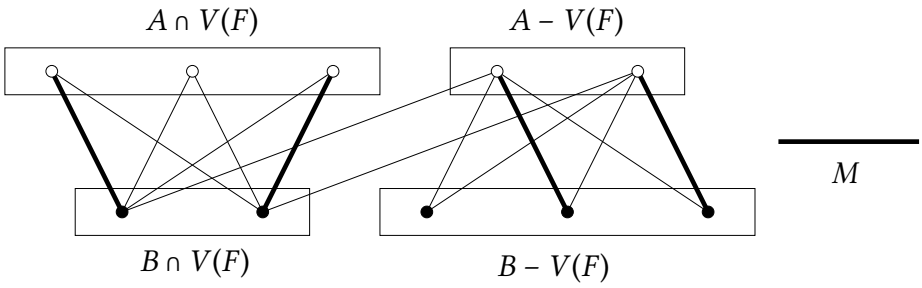


Figura 2.2: *Os vértices de F no algoritmo de Kőnig.*

Sejam n o número de vértices e m o número de arestas do grafo. Como em cada fase do algoritmo o emparelhamento aumenta de uma aresta (à exceção da última fase), serão executadas no máximo $O(n)$ fases. Em cada fase, iteramos pelas arestas do grafo até visitar todas as arestas de G incidentes em $A \cap V(F)$ ou encontrar um caminho aumentante. Assim, em cada fase são executadas $O(m)$ operações simples. Portanto, a complexidade do algoritmo de Kőnig no pior caso é $O(nm)$. Supondo que G seja simples, a complexidade é $O(n^3)$.

2.5 O algoritmo de Edmonds

O algoritmo de Edmonds é similar ao algoritmo de Kőnig. Note que no algoritmo de Kőnig sempre estendemos a floresta alternante construída a partir de vértices que estão a distância par das raízes das suas árvores. Contudo, no caso geral, o grafo não é bipartido e, dependendo da ordem em que as arestas forem percorridas, um mesmo vértice pode estar a distância par ou ímpar da raiz da sua árvore. A Figura 2.3 é um exemplo de como isso pode acontecer. O caminho $(v_1, v_2, v_3, v_6, v_7, v_8)$ do grafo representado nessa figura é aumentante. Contudo, o caminho $(v_1, v_2, v_3, v_4, v_5, v_7, v_6)$ não é aumentante e nem é extensível a um caminho aumentante.

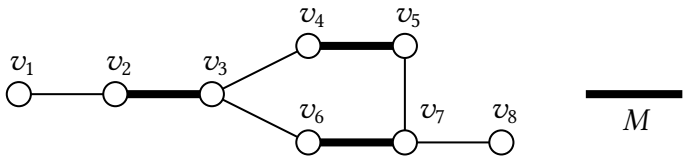


Figura 2.3: *O caminho $(v_1, v_2, v_3, v_6, v_7, v_8)$ é M-aumentante.*

Os ciclos ímpares, como podemos ver, dificultam o problema. Por tal motivo, o algoritmo de Edmonds apareceu só mais de 30 anos depois do algoritmo de Kőnig. Como os ciclos ímpares têm um papel importante no algoritmo de Edmonds, temos as seguintes definições para um grafo G e um emparelhamento M de G.

Um *pedúnculo* é um caminho M-alternante de comprimento par cuja origem é um vértice exposto por M. Uma *corola* C é um ciclo ímpar M-alternante cuja origem é exposta por $M \cap E(C)$. Uma *flor* é a concatenação de um pedúnculo e uma corola, de forma que o

término do pedúnculo é a origem da corola. Na Figura 2.4, temos uma flor cujo pedúnculo é o caminho (v_0, v_1, v_2) e cuja corola é o ciclo C. Note que o pedúnculo de uma flor pode ter comprimento zero se a origem da corola for exposta por M.

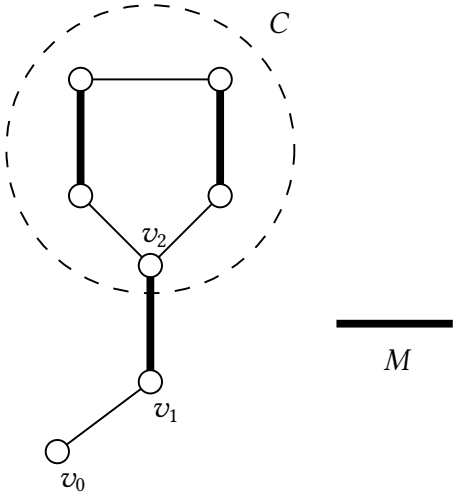


Figura 2.4: *Uma flor.*

Um dos procedimentos do algoritmo, que chamamos de *algoritmo básico*, é semelhante ao algoritmo de Kőnig. Dado um grafo G e um emparelhamento M de G, é feita uma busca, produzindo uma floresta M-alternante, com três possíveis retornos:

1. Ou o algoritmo básico determina um certificado especial da otimalidade de M,
2. ou o algoritmo básico determina um caminho M-aumentante,
3. ou o algoritmo básico determina uma flor.

Um segundo procedimento, chamado de *algoritmo intermediário*, também tem como parâmetros o grafo G e o emparelhamento M. Utiliza o algoritmo básico e tem dois possíveis retornos:

1. Ou o algoritmo intermediário determina um certificado especial da otimalidade de M,
2. ou o algoritmo intermediário determina um caminho M-aumentante.

A iteração do algoritmo intermediário permite então obter um emparelhamento máximo e um certificado de sua otimalidade, de forma análoga à que foi utilizada no algoritmo de Kőnig. Entretanto, no caso do Algoritmo de Edmonds garantimos que o certificado da otimalidade é especial.

2.5.1 O algoritmo básico

1. Se todos os vértices de G são cobertos por M, então certamente M, um emparelhamento perfeito, é máximo. Nesse caso, o algoritmo retorna o conjunto vazio, que é um certificado especial da otimalidade de M (Proposições 2.3 e 2.5). Suponhamos então que G tem vértices expostos por M.

2. Uma invariante do algoritmo é uma subfloresta M-alternante F de G. A floresta F tem uma bipartição (A, B), onde as raízes das árvores de F estão em A e são os vértices expostos de G por M. Inicialmente, temos $F := (V_0, \emptyset)$, onde V_0 é o conjunto de vértices de G que são expostos. O algoritmo mantém um conjunto de *arestas visitadas*, inicialmente vazio.

3. Seja u um vértice em A de uma árvore T de F e $e := uv$ uma aresta incidente em u que não foi visitada. A aresta então é adicionada ao conjunto das arestas visitadas. Temos os seguintes casos:

   (a) O vértice v não pertence a F. Por definição de F, todos os vértices expostos de G por M são raízes de árvores de F. Portanto, v está coberto por M. Seja $f := vw$ a aresta de M que incide em v. O vértice w não pertence a F, pois todos os vértices de F que são cobertos por M são na verdade cobertos por arestas de $M \cap E(F)$. Acrescentamos então os vértices v e w e as arestas e e f à árvore T. Acrescentamos o vértice v a B e o vértice w a A. Adicionamos a aresta f ao conjunto de arestas visitadas. Repetimos o passo 3.

   (b) O vértice v pertence a $V(F) \cap B$. Nesse caso, ignoramos a aresta e e repetimos o passo 3.

   (c) O vértice v pertence a $(V(F) - V(T)) \cap A$. Seja U a árvore de F, distinta de T, que contém o vértice v. Seja $P_T(u)$ o caminho em T da raiz de T a u e seja $P_U(v)$ o caminho em U da sua raiz a v. Seja $\text{rev}(P_U(v))$ o reverso de $P_U(v)$. O caminho $P := P_T(u) \cdot (u, v) \cdot \text{rev}(P_U(v))$ é M-aumentante. Nesse caso, o algoritmo retorna o caminho P.

   (d) O vértice v pertence a $V(T) \cap A$. Vamos denotar por r a raiz de T. Seja $P_T(u) := (r = w_0, w_1, \ldots, w_k = u)$, $k \geq 0$, o caminho M-alternante em T de r a u. Analogamente, seja $P_T(v) := (r = x_0, x_1, \ldots, x_\ell = v)$, $\ell \geq 0$, o caminho M-alternante em T de r a v. Os vértices u e v pertencem ambos a A. Portanto, k e ℓ são ambos pares. A raiz r é origem de ambos os caminhos $P_T(u)$ e $P_T(v)$. Seja i o maior índice, $0 \leq i \leq k$, tal que $w_i \in V(P_T(v))$.

   Podemos então escrever $P_T(u) = P'_u \cdot P''_u$, onde o término de P'_u (e a origem de P''_u) é o vértice w_i. Analogamente, podemos escrever $P_T(v) = P'_v \cdot P''_v$, onde w_i é o término de P'_v (e a origem de P''_v). Seja $\text{rev}(P''_v)$ o reverso do caminho P''_v. A unicidade dos caminhos em T implica que $P'_u = P'_v$. A definição de i implica que w_i é o único vértice em comum de P''_u e P''_v. Portanto, $C := P''_u \cdot (u, v) \cdot \text{rev}(P''_v)$ é um ciclo. O vértice w_i está em A. Portanto, ou $w_i = r$ ou a última aresta de P'_u pertence a M. Assim, C é ímpar, M-alternante, e sua origem não é coberta por $M \cap E(C)$. Finalmente, o passeio $Z := P'_u \cdot C$ é uma flor, onde P'_u é o pedúnculo e o ciclo C é a corola. O algoritmo retorna a flor Z.

4. Se todas as arestas incidentes nos vértices de F em A foram visitadas então M é máximo e B é um certificado especial da otimalidade de M. De fato,

   (a) toda aresta de G incidente em um vértice de A é incidente em B, portanto todos os vértices de $F \cap A$ são isolados em $G - B$;

   (b) todos os vértices de B são cobertos por arestas de $M \cap E(F)$;

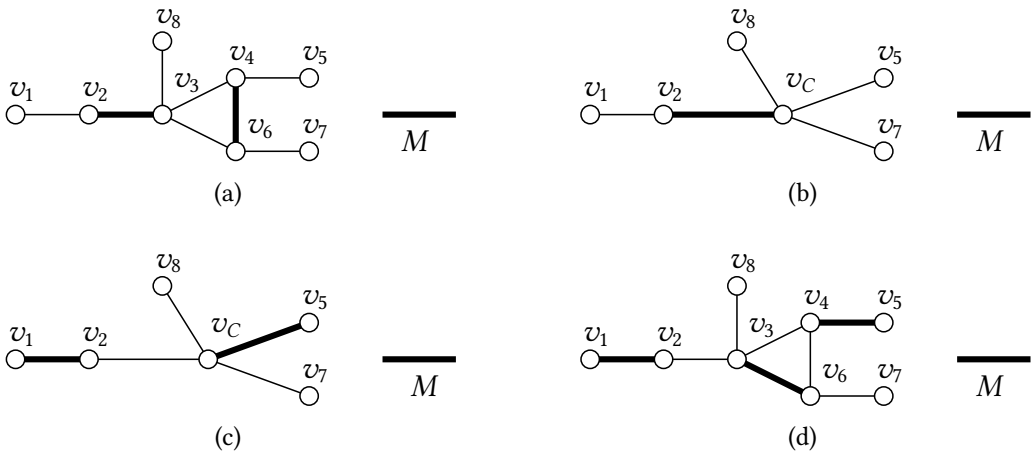


Figura 2.5: *As corolas contraídas podem esconder caminhos aumentantes.*

(c) o único caminho que liga uma raiz de uma árvore de F a um vértice de $F \cap A$ é um caminho M-alternante de comprimento par e, portanto, todos os vértices de $F \cap A$ são supérfluos (veja a Proposição 2.4);

(d) logo, os $|B| + \text{def}(M)$ vértices de A são todos isolados em $G - B$. O algoritmo retorna, então, o conjunto B, um certificado especial da otimalidade de M (veja a Proposição 2.3).

2.5.2 O algoritmo intermediário

O algoritmo intermediário utiliza o algoritmo básico. Quando é achada uma flor, o algoritmo contrai sua corola e continua a busca de caminhos aumentantes. Ao aumentar o emparelhamento, algumas dessas corolas contraídas precisam ser expandidas. Veja na Figura 2.5 o porquê da necessidade de aplicar tais expansões. A Figura 2.5(a) mostra uma corola $C = (v_3, v_4, v_6, v_3)$. A Figura 2.5(b) mostra a contração da corola C no vértice v_C. Note que o caminho $P := (v_1, v_2, v_C, v_5)$ é aumentante. A Figura 2.5(c) mostra o resultado de aumentar o emparelhamento usando o caminho P. A Figura 2.5(d) mostra a expansão de v_C de volta em C. Note que, no grafo representado por essa figura, temos o caminho aumentante $P' := (v_8, v_3, v_6, v_7)$, mas no grafo da Figura 2.5(c) não há caminhos aumentantes. Após aumentar um emparelhamento com um caminho aumentante, as corolas contraídas que estão nesse caminho podem esconder outros caminhos aumentantes. Por tal motivo, essas corolas precisam em algum momento ser expandidas. As demais corolas poderiam ficar contraídas.

O algoritmo intermediário utiliza o algoritmo básico. Vamos fazer uma descrição de uma versão recursiva do algoritmo.

1. Como primeiro passo, dado o grafo G e o emparelhamento M, o algoritmo executa o algoritmo básico, utilizando como argumentos o grafo G e o emparelhamento M.
2. Se for obtido um certificado especial S da otimalidade de M, o algoritmo retorna S.
3. Se for obtido um caminho M-aumentante P, o algoritmo retorna o emparelhamento

$M \triangle E(P)$.

4. Suponhamos então que foi obtida uma flor Z, cujo pedúnculo é P_Z e cuja corola é um ciclo C_Z. O algoritmo contrai a corola C_Z a um único vértice, v_C, gerando o novo grafo $H := G/(C_Z \longrightarrow v_C)$. O emparelhamento associado a H será então $N := M - E(C_Z)$. Note que

$$V(N) = \begin{cases} [V(M) - V(C_Z)] + v_c, & \text{se } M \cap \partial(V(C_Z)) \neq \emptyset, \\ V(M) - V(C_Z), & \text{caso contrário.} \end{cases} \tag{2.1}$$

Portanto,

$$\text{def}_H(N) = \text{def}_G(M). \tag{2.2}$$

Em seguida, o algoritmo determina, de forma recursiva, o resultado da sua execução, com H e N fazendo o papel de G e de M, respectivamente. Assim, obtém-se ou um certificado da otimalidade de N em H ou um emparelhamento de H que tem mais uma aresta do que N.

Antes de continuar a descrição do algoritmo, enunciaremos duas propriedades cujas demonstrações são elementares e que serão úteis na análise posterior.

Proposição 2.6. *Seja N' um emparelhamento de H tal que $v_C \notin V(N')$. Dado um vértice x de C_Z, seja Q o emparelhamento de C_Z tal que $V(C_Z) - x = V(Q)$. Então, $M' := N' \cup Q$ é um emparelhamento de G. Ademais,*

$$V(M') = V(N') \cup [V(C_Z) - x], \text{ portanto } \text{def}_G(M') = \text{def}_H(N'). \qquad \square$$

Proposição 2.7. *Seja N' um emparelhamento de H tal que $v_C \in V(N')$. Seja e a aresta de $N' \cap \partial(v_C)$, seja x o extremo da aresta e que, em G, pertence a $V(C_Z)$ e seja Q o emparelhamento de C_Z tal que $V(C_Z) - x = V(Q)$. Então, $M' := N' \cup Q$ é um emparelhamento de G. Ademais,*

$$V(M') = [V(N') - v_C] \cup V(C_Z), \text{ portanto } \text{def}_G(M') = \text{def}_H(N'). \qquad \square$$

Caso: o emparelhamento N não é máximo. A recursão retorna um emparelhamento N' tal que $|N'| > |N|$.

Pelas Proposições 2.6 e 2.7, G tem um emparelhamento M' que satisfaz a igualdade $\text{def}_G(M') = \text{def}_H(N')$. Em vista de (2.2),

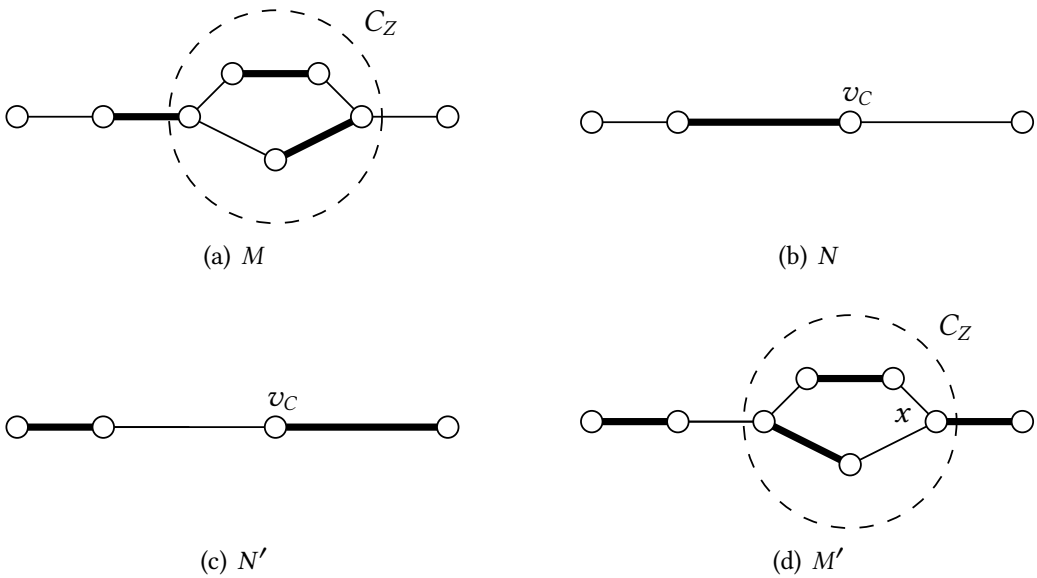
$$\text{def}_G(M') = \text{def}_H(N') < \text{def}_H(N) = \text{def}_G(M).$$

Portanto, M' é um emparelhamento de G com mais arestas do que M. Portanto, M não é máximo e o algoritmo retorna o emparelhamento M'.

Esse caso está ilustrado na Figura 2.6.

Caso: o emparelhamento N é máximo. A recursão retorna um certificado especial T da otimalidade de N.

Nesse caso, o algoritmo retorna T, conforme justificado pelo resultado a seguir.



(a) M (b) N

(c) N' (d) M'

Figura 2.6: *O caso em que a recursão retorna o emparelhamento N' de maior cardinalidade do que o emparelhamento N.*

Proposição 2.8. *O conjunto T é um certificado especial da otimalidade do emparelhamento M.*

Demonstração. Vamos, inicialmente, mostrar que o vértice de contração v_C pertence a uma componente ímpar de $H - T$. O pedúnculo P_Z de Z pode ser considerado como um caminho em H, se substituirmos seu término por v_C (e sua origem também, se P_Z tiver comprimento zero). Assim, P_Z é um caminho M_H-alternante de comprimento par, cuja origem é um vértice exposto por M_H em H e cujo término é v_C. Pela Proposição 2.4, v_C é supérfluo em H. Pela Proposição 2.5, v_C pertence a uma componente ímpar de $H - T$.

Vamos agora mostrar que T é um certificado da otimalidade de M. Seja L a componente ímpar de $H - T$ que contém o vértice v_C. Se expandirmos em L o vértice v_C de volta para o ciclo C_Z, obteremos uma componente ímpar, K, de $G - T$. Ademais, as outras componentes de $G - T$ coincidem com as outras componentes de $H - T$. Então, $o(G - T) = o(H - T)$. Da igualdade (2.2) temos que

$$\mathrm{def}_G(M) = \mathrm{def}_H(N) = o(H - T) - |T| = o(G - T) - |T|.$$

Concluímos que T é um certificado da otimalidade de M.

Resta agora provar que T é especial. Seja w um vértice de alguma componente ímpar de $G - T$.

2.8.1. *O grafo G tem um emparelhamento máximo que deixa w exposto e, portanto, w é supérfluo em G.*

Demonstração. Vamos considerar, inicialmente, o caso em que $w \in V(C_Z)$. Como mencionado no começo desta demostração, o vértice v_C é supérfluo em H. Assim,

H tem um emparelhamento máximo, digamos, N', que deixa v_C exposto. Pela Proposição 2.6, G tem um emparelhamento M' que deixa w exposto. Ademais, em vista de (2.2), $\text{def}_G(M') = \text{def}_H(N') = \text{def}_H(N) = \text{def}_G(M)$. Portanto, M' é máximo em G.

Podemos então supor que $w \notin V(C_Z)$. Assim, w é um vértice que pertence a uma componente ímpar de $H - T$ e é distinto de v_C. Vimos que T é uma certidão especial da otimalidade de H. Seja N' um emparelhamento máximo de H que deixa w exposto. Se $v_C \notin V(N')$ então, pela Proposição 2.6, G tem um emparelhamento M' que deixa w exposto e tal que $\text{def}_G(M') = \text{def}_H(N')$. A mesma conclusão vale se $v_C \in V(N')$, pela Proposição 2.7‘. Em ambos os casos, em vista da igualdade (2.2), $\text{def}_G(M') = \text{def}_H(N') = \text{def}_H(N) = \text{def}_G(M)$. Portanto, M' é máximo em G. □

Concluímos que todo vértice que pertence a uma componente ímpar de $G - T$ é supérfluo. De fato, T é uma certidão especial da otimalidade de M. □

2.5.3 Descrição do algoritmo de Edmonds

Dado um grafo G, o algoritmo de Edmonds determina um emparelhamento máximo de G e um certificado especial da otimalidade do emparelhamento achado. Para isso, o algoritmo faz várias chamadas sucessivas ao algoritmo intermediário.

1. O algoritmo de Edmonds mantém, a cada iteração, um emparelhamento M de G. Inicialmente, o emparelhamento M pode ser vazio.
2. Executa-se o algoritmo intermediário com argumentos G e M. Temos os seguintes dois casos:
   (a) O algoritmo intermediário retorna um emparelhamento M' com mais arestas do que M. Repetimos o passo 2 com M' no lugar de M.
   (b) O algoritmo intermediário retorna um certificado especial S da otimalidade de M. Então, M é um emparelhamento máximo. Nesse caso, o algoritmo de Edmonds retorna o emparelhamento máximo M e o certificado especial S da otimalidade de M.

Sejam n o número de vértices e m o número de arestas de G. O algoritmo básico é linear, pois o número de operações elementares é $O(m + n)$. O algoritmo intermediário obtém $O(n)$ flores antes de obter um caminho aumentante ou um certificado da otimalidade de M. Portanto, em tempo $O((m + n)n)$, se obtém um caminho M-aumentante ou um certificado da otimalidade de M. Assim, esta implementação do algoritmo de Edmonds tem complexidade $O((m + n)n^2)$. Supondo que o grafo não tenha arestas múltiplas, a complexidade então é $O(n^4)$.

O gargalo do tempo de execução está na contração das corolas. É possível melhorar a complexidade de tempo modificando o algoritmo para que não seja necessário fazer as contrações de forma explícita. Uma implementação com essa abordagem foi feita por Gabow [7]. Lovász e Plummer, no famoso livro “*Matching Theory*”, também apresentam uma versão do algoritmo em que as contrações são feitas apenas implicitamente [20]. Essas duas versões do algoritmo têm complexidade de tempo $O(n^3)$.

Capítulo 3

Grafos cobertos por emparelhamentos

Neste capítulo, apresentamos algoritmos para resolver o problema dos grafos cobertos por emparelhamentos. O primeiro algoritmo que mostramos é simples e eficiente, mas não é o melhor conhecido. Já o segundo algoritmo, mais sofisticado, usa resultados da teoria de grafos cobertos por emparelhamentos para atingir uma melhor complexidade.

Lembre que um grafo é coberto por emparelhamentos se tem pelo menos uma aresta, é conexo e todas suas arestas são emparelháveis. Verificar conexidade é trivial e já vimos no capítulo anterior como obter um emparelhamento máximo de um grafo. Por tal motivo, os algoritmos deste capítulo pressupõem que o grafo da entrada é conexo, emparelhável e tem pelo menos uma aresta. Assim, o foco dos algoritmos descritos a seguir consiste em decidir se todas as arestas do grafo são emparelháveis.

3.1 O algoritmo ingênuo

O algoritmo ingênuo, descrito nesta seção, é uma aplicação direta da seguinte proposição.

Proposição 3.1. *Uma aresta $e := uv$ de um grafo G é emparelhável se e somente se o grafo $G - u - v$ é emparelhável.*

Demonstração.
($\Longrightarrow$) Se e é emparelhável, então existe um emparelhamento perfeito M de G tal que $e \in M$. Logo, $M - e$ é um emparelhamento perfeito de $G - u - v$. Portanto, o grafo $G - u - v$ é emparelhável.

($\Longleftarrow$) Se $G - u - v$ é emparelhável, então existe um emparelhamento perfeito M de $G - u - v$. Logo, $M + e$ é um emparelhamento perfeito de G. Portanto, a aresta e é emparelhável. □

O algoritmo tem como entrada um grafo G conexo, emparelhável e com pelo menos uma aresta. Utiliza, como procedimento, outro algoritmo para determinar emparelhamentos máximos em subgrafos de G e tem dois possíveis retornos:

1. Ou o algoritmo determina uma aresta e de G não emparelhável e, portanto, decide que G não é coberto por emparelhamentos,
2. ou o algoritmo decide que G é coberto por emparelhamentos.

Vamos assumir que o algoritmo de emparelhamento máximo utilizado pelo algoritmo ingênuo é o algoritmo de Edmonds, como descrito na seção 2.5.3. Contudo, qualquer outro algoritmo de emparelhamento máximo também funcionaria. A seguir, vejamos uma descrição do algoritmo.

1. O algoritmo ingênuo mantém um conjunto de *arestas visitadas*, inicialmente vazio.
2. Se não há mais arestas não visitadas, então o algoritmo continua no passo 3. Caso contrário, seja $e := uv$ uma aresta não visitada. O algoritmo ingênuo executa o algoritmo de Edmonds com o grafo $G - u - v$. Seja M o emparelhamento obtido. Para decidir se M é um emparelhamento perfeito em $G - u - v$, basta verificar se todos os vértices de $G - u - v$ são cobertos por M. Temos os seguintes dois casos.
   (a) O emparelhamento M é perfeito em $G - u - v$. Nesse caso, o algoritmo adiciona as arestas de $M + e$ no conjunto de arestas visitadas. Depois, repete o passo 2.
   (b) O emparelhamento M não é perfeito em $G - u - v$. Pela Proposição 3.1, sabemos que e não é emparelhável. Portanto, o algoritmo decide que G não é coberto por emparelhamentos.
3. Todas as arestas foram visitadas. Assim, todas as arestas são emparelháveis. Portanto, o algoritmo decide que o grafo G é coberto por emparelhamentos.

Sejam n o número de vértices e m o número de arestas de G. São feitas $O(m)$ execuções do algoritmo de Edmonds. Assim, assumindo que o grafo é simples, a complexidade total do algoritmo ingênuo é $O(n^6)$.

Note que a complexidade desse algoritmo pode melhorar usando um algoritmo de emparelhamento máximo mais eficiente.

3.2 O algoritmo de Carvalho e Cheriyan

Um algoritmo mais sofisticado para resolver o problema dos grafos cobertos por emparelhamentos foi descrito por Carvalho e Cheriyan [2]. Eles usaram esse algoritmo como um procedimento de um algoritmo para decomposição em orelhas de grafos cobertos por emparelhamentos.

O algoritmo de Carvalho e Cheriyan utiliza o algoritmo de Edmonds e algumas propriedades da teoria de grafos cobertos por emparelhamentos para conseguir seu objetivo. A seguir, apresentamos uma descrição desse algoritmo, com uma prova de corretude distinta da do artigo original.

O algoritmo de Carvalho e Cheriyan tem como entrada um grafo G conexo e um emparelhamento perfeito M de G. Este algoritmo tem os mesmos possíveis retornos que o algoritmo ingênuo.

1. O algoritmo mantém um conjunto de *vértices visitados*, inicialmente vazio.

2. Se não há mais vértices não visitados, então o algoritmo continua no passo 3. Caso contrário, sejam v um vértice não visitado e $uv \in M$ a aresta do emparelhamento M incidente nesse vértice. O algoritmo executa o algoritmo intermediário do algoritmo de Edmonds (descrito na seção 2.5.2) com o grafo $G' := G - v$ e o emparelhamento $M' := M - uv$ de G'. Note que u é o único vértice não coberto por M' em G'. Assim, o emparelhamento M' não pode ser aumentado em G'. Logo, o algoritmo intermediário retorna um certificado especial, digamos S', da otimalidade de M' no grafo G'.

   O algoritmo decide se as arestas incidentes em v são emparelháveis em G usando o resultado a seguir.

   Proposição 3.2. *Uma aresta $vw \in \partial(v)$ é emparelhável em G se e somente se o vértice w está em uma componente ímpar de $G' - S'$.*

   Demonstração. Pela Proposição 3.1, temos que vw é emparelhável se e somente se $G' - w$ é emparelhável. Como a deficiência de G' é igual a 1, temos que a aresta vw é emparelhável se e somente se o vértice w é supérfluo em G'. Como S' é um certificado especial, concluímos que a aresta vw é emparelhável em G se e somente se o vértice w está em uma componente ímpar de $G' - S'$. □

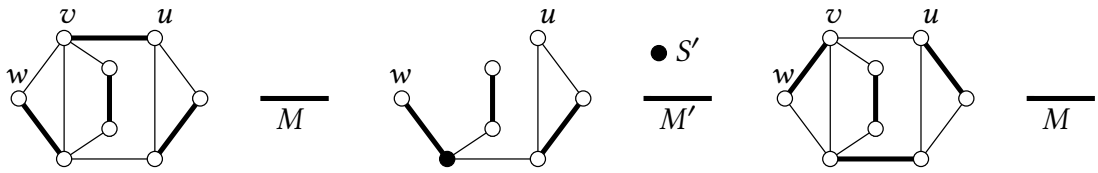
   Assim, podemos distinguir dois casos:

   (a) Existe uma aresta $vw \in \partial(v)$ tal que o vértice w não está em uma componente ímpar de $G' - S'$. Nesse caso, a aresta vw não é emparelhável e o algoritmo decide que o grafo G não é coberto por emparelhamentos.

   (b) Para toda aresta $vw \in \partial(v)$, o vértice w está em uma componente ímpar de $G' - S'$. Nesse caso, o algoritmo adiciona o vértice v no conjunto de vértices visitados e repete o passo 2.

3. Se todos os vértices foram visitados, então todas as arestas de G são emparelháveis. Portanto, o algoritmo decide que o grafo é coberto por emparelhamentos.

A Figura 3.1 mostra um exemplo da execução do algoritmo de Carvalho e Cheriyan. Na Figura 3.1(b) temos o resultado da execução do algoritmo intermediário no grafo $G' := G - v$. Note que os vértices u e w estão em componentes ímpares de $G' - S'$ e que portanto as arestas vu y vw são emparelháveis. De fato, na Figura 3.1(c), podemos ver um emparelhamento perfeito de G que usa a aresta vw.



(a) G (b) G' (c) Emparelhamento perfeito de G que usa a aresta vw

Figura 3.1: *Execução do passo 2 do algoritmo de Carvalho e Cheriyan no vértice v.*

Sejam n o número de vértices e m o número de arestas de G. O algoritmo de Carvalho

e Cheriyan executa $O(n)$ vezes o passo 2. Em cada um desses passos, o algoritmo fixa um vértice v e executa o algoritmo intermediário em $G' := G - v$. O algoritmo intermediário, então, retorna um subconjunto de vértices S'. Depois, o algoritmos verifica se os vértices adjacentes a v estão em alguma componente ímpar de $G' - S'$. Essa verificação pode ser feita em $O(n + m)$ operações simples. Assim, o gargalo de tempo de execução de uma iteração do algoritmo está na execução do algoritmo intermediário. Então, assumindo que o grafo é simples, a complexidade do algoritmo de Carvalho e Cheriyan é $O(n^4)$.

Contudo, como visto no final do Capítulo 2, existem implementações mais eficientes do algoritmo de Edmonds em que as contrações das corolas são implícitas. Nesses algoritmos, o tempo de execução do algoritmo intermediário é $O(m)$. Assim, é possível implementar uma versão $O(n^3)$ do algoritmo de Carvalho e Cheriyan.

Capítulo 4

Implementação

A função principal do programa implementado para este projeto é gerar animações de algoritmos em grafos. Este programa pode ser baixado do repositório hospedado em https://github.com/Byakko97/matching_covered_graphs_animations. Todo o seu código fonte está na linguagem Python.

A interface do programa é de linha de comandos, mas as animações são executadas em uma janela de interface gráfica.

Os algoritmos suportados são o algoritmo de Edmonds (veja a seção 2.5) e o algoritmo de Carvalho e Cheriyan (veja a seção 3.2). Contudo, o programa é de código aberto e a sua estrutura facilita a adição de novos algoritmos, como se explica na seção 4.4.

4.1 Programa

Para poder usar o programa, primeiro tem que clonar o repositório indicado acima. Depois, tem que instalar o arcabouço para animações `graph-tool` [16]. Para isso, consulte a guia de instalação dessa ferramenta [15].

Uma vez instalado `graph-tool`, rode, na raiz do projeto, o comando

```
$ python3 -m pip install -e .
```

para instalar o programa.

Nesta seção, explicamos como usar as diferentes funcionalidades deste programa e também mostramos a estrutura do projeto.

4.1.1 Funcionalidades do programa

Todos os comandos suportados pelo programa requerem o argumento `algo`, o qual aceita os nomes dos algoritmos implementados. Como dito acima, os algoritmos implementados no programa são o algoritmo de Edmonds, representado pela cadeia `‘‘edmonds’’`, e o algoritmo de Carvalho e Cheriyan, representado pela cadeia `‘‘carvalho-cheriyan’’`.

A função principal do programa é a de gerar animações de algoritmos. Para gerar a animação de um algoritmo de nome ‘‘algoritmo’’ com o teste de entrada ‘‘entrada’’ rode, na raiz do projeto, o comando

```
$ make animate algo=algoritmo test=entrada
```

Por exemplo, para gerar uma animação do algoritmo de Edmonds com o grafo de Petersen, execute

```
$ make animate algo=edmonds test=petersen
```

Veja a seção 4.1.3 para saber mais sobre os arquivos de entrada para teste.

É possível modificar o funcionamento padrão do comando `make animate` passando argumentos opcionais a este. Adicionando o argumento `f=freq`, mudamos a frequência em que se atualiza o quadro de animação para `freq` milissegundos. A frequência padrão é de 1 segundo. Assim, para dobrar a velocidade da animação em relação à configuração padrão, rode

```
$ make animate algo=algoritmo test=entrada f=500
```

Também é possível executar a animação em modo manual, no qual, para passar de um quadro da animação ao seguinte, tem que se fazer um clique do mouse. Para executar a animação nesse modo, adicione o argumento `m=true` da seguinte forma:

```
$ make animate algo=algoritmo test=entrada m=true
```

Ademais, é possível salvar os quadros da animação. Ao se fazer isso, a animação não é executada na tela; apenas são salvos os quadros da animação na pasta `frames`, localizada na raiz do projeto. Para salvar os quadros da animação dessa forma, adicione o argumento `off=true`:

```
$ make animate algo=algoritmo test=entrada off=true
```

Às vezes pode ser útil rodar um algoritmo sem animação, apenas com saída no terminal. Por exemplo, para depurar a implementação de um algoritmo, essa opção pode ser vantajosa. O comando para executar, sem animação, o algoritmo ‘‘algoritmo’’ com o teste de entrada ‘‘entrada’’ é

```
$ make run algo=algoritmo test=entrada
```

Finalmente, também é possível testar se as implementações dos algoritmos estão corretas executando-as com todos os testes de entrada. Assim, para verificar a implementação do algoritmo ‘‘algoritmo’’, execute

```
$ make test algo=algoritmo
```

Obviamente, não é suficiente executar um algoritmo com vários testes para concluir que está correto, pois os testes poderiam não ser fortes o suficiente. Contudo, vários erros podem ser encontrados e corrigidos dessa forma.

4.1.2 Estrutura do programa

É importante conhecer a estrutura do programa, pois é feita considerando possíveis extensões da versão atual (principalmente, a adição de novos algoritmos).

O programa está organizado seguindo a seguinte estrutura de diretório de arquivos:

- `src`: todo o código fonte está nessa pasta
  - `algorithm`: pacote com os algoritmos implementados
  - `animation`: pacote para animação de grafos
  - `api`: pacote com funcionalidades executáveis pelos usuários
  - `data_structures`: pacote com estruturas de dados usadas nos algoritmos
- `tests` - arquivos de entrada de teste

O pacote `algorithm` contém as classes `EdmondsBlossom`, que implementa o algoritmo de Edmonds, e `CarvalhoCheriyan`, que implementa o algoritmo de Carvalho e Cheriyan. Ambas classes herdam da classe `AlgorithmBase`, a qual permite adicionar novos algoritmos facilmente, como se explica na seção 4.4.

O pacote `animation` contém a classe `GraphAnimation` que encapsula um objeto `Graph` do pacote `graph-tool` e implementa as funções de animação de grafos usadas nos algoritmos. Além disso, esse pacote contém classes com os distintos estilos usados no desenho de vértices e arestas. Essas classes contêm constantes relacionadas às formas, tamanhos, larguras, etc. dos objetos animados.

O pacote `api` contém as funcionalidades executáveis pelos usuários mostradas na seção 4.1.1. O arquivo `Makefile` encapsula a execução dessas funções, mas os usuários podem também executar as funções desse pacote diretamente.

O pacote `data_structures` contém classes que implementam grafos, vértices, arestas, corolas, entre outras estruturas de dados usadas nos algoritmos.

4.1.3 Arquivos de entrada para teste

A entrada para todos os algoritmos do programa é um grafo. Assim, cada arquivo de entrada é um arquivo de texto cujo conteúdo representa um grafo. A primeira linha de um arquivo de entrada contém dois números inteiros: o número de vértices n e o número de arestas m do grafo representado, nessa ordem. Os vértices do grafo estão numerados do 0 até o $n - 1$. As seguintes m linhas da entrada representam as arestas do grafo. Cada uma dessas linhas representa uma aresta com dois inteiros, os índices dos vértices nos seus extremos. Na Figura 4.1 temos um exemplo de um arquivo de entrada (na esquerda) e o grafo representado por dito arquivo (na direita).

Para adicionar um novo teste ao programa, basta colocar um novo arquivo de texto, que siga o formato descrito no parágrafo anterior, na pasta `tests`.

4.1.4 Animações

Como dito acima, o arcabouço para animações usado no programa é `graph-tool`. Esse arcabouço é um pacote eficiente de Python para a manipulação e a análise estatística de

```
5 5
0 1
0 2
1 3
1 4
3 4
```

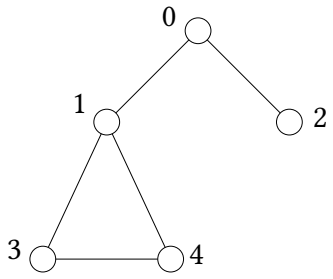


Figura 4.1: *Exemplo de arquivo de entrada e o grafo que representa.*

grafos [17]. Contudo, essa ferramenta também permite a visualização dos grafos e até a animação destes.

Existem outros arcabouços para animação de grafos, mas a escolha do `graph-tool` foi feita devido ao fato de ser um pacote de Python, a linguagem usada na implementação do programa, e também à detalhada documentação desse software.

O `graph-tool` implementa diferentes algoritmos de desenho de grafos e é possível decidir qual usar na hora de gerar as animações. Neste projeto, usamos o algoritmo sfdp (*Scalable Force-Directed Placement*) para desenhar os grafos. Esse algoritmo é descrito no artigo "*Efficient and High Quality Force-Directed Graph Drawing*" [9].

4.2 Algoritmo de Edmonds

Apesar de que a descrição do algoritmo na seção 2.5.3 é recursiva, implementamos uma versão iterativa do algoritmo. Nessa implementação, as expansões das corolas foram feitas apenas quando necessárias, ou seja, só quando o caminho aumentante achado contém uma corola comprimida.

Para a compressão das corolas, foi usada uma estrutura de dados para dar suporte a operações em conjuntos disjuntos. Desta forma, a compressão de corolas foi implementada como a união dos vértices da corola com um novo vértice, o vértice que representa a corola comprimida. Isso também nos permite saber em que vértice do grafo resultante se acham comprimidos os vértices do grafo anterior.

São bem conhecidas estruturas de dados eficientes que dão suporte a essas operações (união e busca) em conjuntos disjuntos. Por exemplo, veja o capítulo "*Data Structures for Disjoint Sets*" do famoso livro "*Introduction to algorithms*" de Cormen, Leiserson, Rivest e Stein [4]. Contudo, para a expansão de corolas, também foi necessário implementar a separação de conjuntos. Por tal motivo, a estrutura de dados implementada é tal que as operações de união e separação (de um elemento do resto do conjunto) têm complexidade de tempo $O(1)$. Contudo, para atingir essa eficiência, a implementação da operação de busca tem complexidade de tempo $O(n)$, onde n é o número de vértices do grafo.

4.2.1 Execução sem animação

Quando o algoritmo é executado sem animação com o comando `make run`, a saída é um emparelhamento máximo do grafo da entrada. Esse emparelhamento é mostrado no terminal como uma lista de arestas, as quais estão representadas da mesma forma que no arquivo de entrada.

4.2.2 Testes

Para verificar que a implementação do algoritmo de Edmonds está correta, primeiro verificamos que a saída do algoritmo seja, de fato, um emparelhamento. Para isso, basta verificar que não existe mais de uma aresta do emparelhamento incidente no mesmo vértice.

Depois, para verificar que o emparelhamento seja máximo, usamos a Proposição 2.3. Seja S o certificado devolvido pelo algoritmo. Verificamos que S cumpre com as condições dessa proposição. Ou seja, contamos o número de componentes ímpares em $G - S$ e comprovamos que seja igual à deficiência do emparelhamento achado mais a cardinalidade do conjunto S.

4.2.3 Animação

O emparelhamento que é mantido durante a execução do algoritmo de Edmonds é representado por arestas vermelhas e grossas. Veja a Figura 4.2.

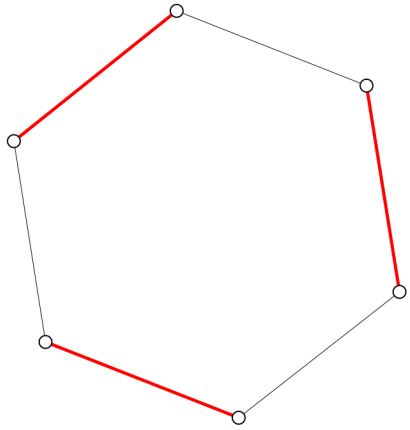


Figura 4.2: *Um emparelhamento máximo em um ciclo de 6 vértices.*

O algoritmo busca caminhos aumentantes até que o emparelhamento achado seja máximo. Quando um caminho aumentante é encontrado, as arestas desse caminho que não estão no emparelhamento são representadas por linhas azuis tracejadas. Veja a Figura 4.3.

Em vez de um caminho aumentante, o algoritmo pode também achar uma flor. As arestas de uma flor são representadas da mesma forma que as de um caminho aumentante. Por outro lado, os vértices da sua corola são representados por círculos duplos. Veja a Figura 4.4(a). Quando é achada uma flor, sua corola é comprimida. O vértice que

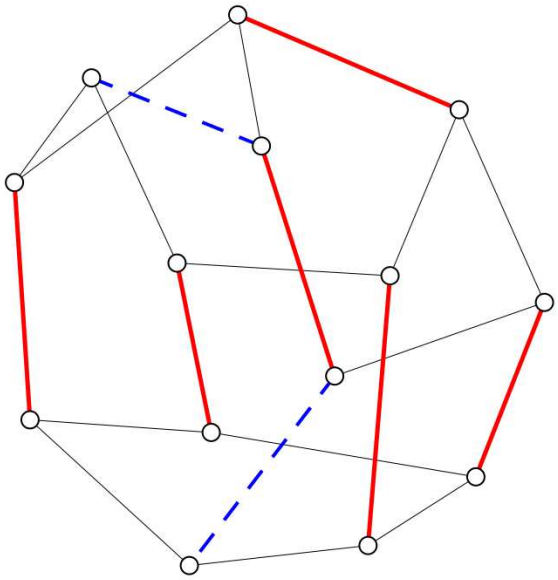


Figura 4.3: *Um caminho aumentante.*

resulta da compressão da corola é representado por um círculo preenchido de cinza. Veja a Figura 4.4(b).

Finalmente, é obtido um emparelhamento máximo do grafo e um certificado da otimalidade desse emparelhamento. Os vértices desse certificado são representados por quadrados preenchidos de azul. Veja a Figura 4.5.

4.3 Algoritmo de Carvalho e Cheriyan

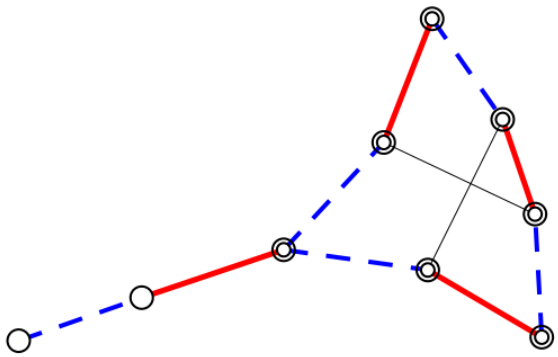
Diferente da descrição do algoritmo feita na seção 3.2, o algoritmo implementado não espera que a entrada seja um grafo conexo nem que tenha um emparelhamento perfeito. A implementação verifica, primeiro, se o grafo da entrada cumpre com essas condições antes de executar o resto do algoritmo.

No caso em que o grafo da entrada cumpre com as condições ditas acima, o algoritmo computa a lista de arestas não emparelháveis do grafo. Desta forma, podemos concluir que o grafo é coberto por emparelhamentos se essa lista for vazia.

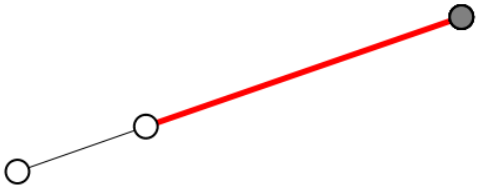
4.3.1 Execução sem animação

Quando o algoritmo é executado sem animação usando o comando `make run`, no terminal, o algoritmo indica se o grafo da entrada é coberto por emparelhamentos ou não. No caso de não ser coberto por emparelhamentos, a mensagem escrita no terminal explica o motivo. Assim, nesse caso, é mostrado na tela um dos seguintes motivos, nesta ordem de prioridade:

1. O grafo não tem nenhuma aresta;
2. o grafo não é conexo;



(a)



(b)

Figura 4.4: *Uma flor antes e depois de comprimir sua corola.*

3. o grafo não é emparelhável;

4. o grafo tem arestas que não são emparelháveis.

No caso em que o grafo tem arestas não emparelháveis, a lista de arestas não emparelháveis é também mostrada na tela. Essas arestas são representadas da mesma forma que as do arquivo de entrada.

4.3.2 Testes

Para verificar que a implementação do algoritmo de Carvalho e Cheriyan está correta, implementamos também o algoritmo ingênuo para decidir se um grafo é coberto por emparelhamentos (veja a seção 3.1). Nessa implementação, a saída de ambos algoritmos é o conjunto das arestas não emparelháveis do grafo de entrada. Assim, basta conferir que esses conjuntos sejam idênticos para comprovar que a implementação do algoritmo de Carvalho e Cheriyan está certa.

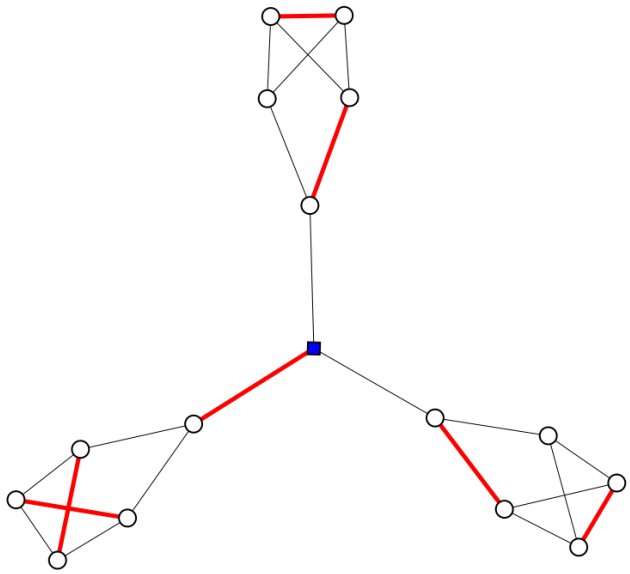


Figura 4.5: *Um emparelhamento máximo do grafo de Sylvester e um certificado de sua otimalidade.*

4.3.3 Animação

Vamos a descrever a animação do algoritmo em um grafo G.

Quando o grafo G não é conexo, apenas é mostrada sua representação na tela e o algoritmo não é executado. No caso em que não é emparelhável, é mostrado o resultado da execução do algoritmo de Edmonds nesse grafo: um emparelhamento máximo, que não é perfeito, e um certificado da otimalidade desse emparelhamento.

Assim, quando o grafo G é conexo e emparelhável, a animação começa com um emparelhamento perfeito M do grafo. Depois, o algoritmo itera por cada vértice do grafo. Quando o algoritmo está processando o vértice v, gera um novo grafo $G - v$. Na representação dessa operação, o vértice v e as arestas incidentes nesse vértice são coloridas de cinza. Além disso, essas arestas são representadas como linhas tracejadas. Veja a Figura 4.6.

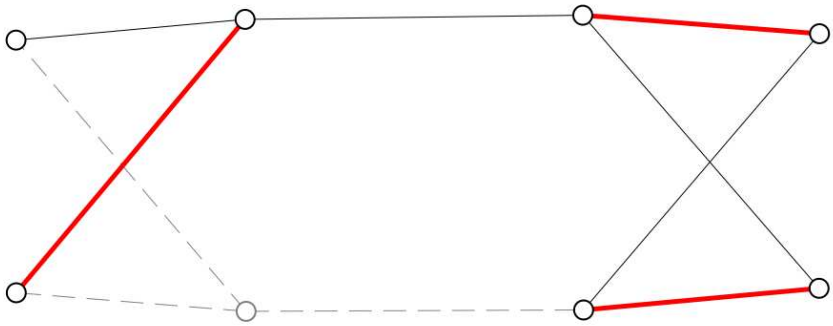


Figura 4.6: $G - v$.

Seja e a aresta de M que cobre o vértice v. O algoritmo então executa o algoritmo de Edmonds com o grafo $G - v$ e o emparelhamento $M - e$. Assim, é obtido um certificado S da otimalidade de $M - e$ no grafo $G - v$. Cada vértice u de $G - v$ é classificado em um de 3 tipos:

- O vértice u está no conjunto S. Nesse caso, o vértice é representado como um quadrado preenchido de azul.
- O vértice u está em uma componente ímpar de $G - S$. Nesse caso, o vértice é representado por um círculo duplo.
- O vértice u está em uma componente par de $G - S$. Nesse caso, o vértice é representado por um círculo preenchido de preto.

Após essa classificação, o algoritmo decide quais arestas incidentes em v no grafo G são emparelháveis. As arestas não emparelháveis incidentes em v são representadas por linhas azuis com uma barra no meio. Veja a Figura 4.7.

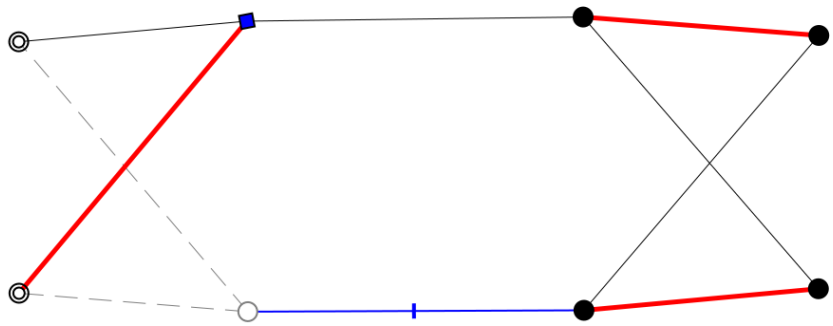


Figura 4.7: *Classificação das arestas incidentes em v.*

No fim do algoritmo, são destacadas todas as arestas não emparelháveis do grafo com a representação descrita. Veja a Figura 4.8. Assim, se G for coberto por emparelhamentos, nenhuma das arestas do grafo é destacada.

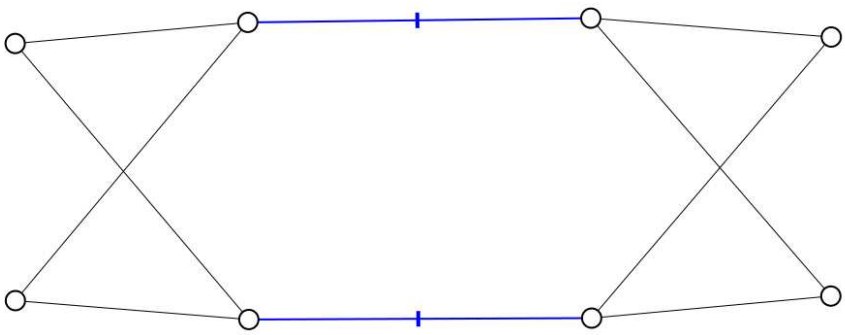


Figura 4.8: *O grafo G não é coberto por emparelhamentos.*

4.4 Adição de novos algoritmos

Para adicionar um novo algoritmo no programa, o primeiro que precisa ser feito é adicionar um novo arquivo na pasta `src/algorithm`. O arquivo adicionado precisa conter uma classe que implemente o algoritmo a ser adicionado. Além disso, é obrigatório que essa classe herde da classe `AlgorithmBase`, implementada no arquivo `src/algorithm/algorithm_base.py`.

```
1  from src.data_structures.graph import Graph
2
3
4  class AlgorithmBase():
5      def __init__(self, g: Graph):
6          self.g = g
7
8      def update_state(self, widget, event) -> bool:
9          self.g.update_animation_state()
10         return True
11
12     def run_algorithm(self) -> None:
13         while self.update_state(None, None):
14             pass
15
16     def animate(self, manual_mode: bool, frequence: int) -> None:
17         self.g.animation.animate(
18             self.update_state, manual_mode, frequence,
19         )
20
21     def run(self) -> None:
22         self.run_algorithm()
23
24     def test(self) -> bool:
25         self.run_algorithm()
26         return True
```

Programa 4.1: *Código fonte da classe `AlgorithmBase`*

Podemos ver no Programa 4.1 que todos os métodos dependem da função `update_state`. Essa função precisa ser sobrecarregada pelo novo algoritmo adicionado e é nessa função que é implementada a lógica do algoritmo em si. As funções `run_algorithm` e `animate` executam a função `update_state` enquanto retornar o valor `True`.

Quando o algoritmo é executado com animação pelo comando `make animate`, cada execução da função `update_state` gera um novo quadro da animação. Por tal motivo, a implementação do novo algoritmo precisa ser feita em passos bem definidos, divididos segundo o que se quer mostrar na animação.

Para diferenciar que a função `update_state` está sendo executada com animação, verifique que o objeto `self.g.animation` não seja `None`. Caso contrário, o algoritmo

está sendo executado sem animação, seja pelo comando `make run` ou seja pelo comando `make test`.

O procedimento `run` também deve ser sobrecarregado se se quer mostrar o resultado da execução na tela do terminal.

Quando o comando `make test` é executado, a função `test` do algoritmo é invocada com todas as entradas disponíveis. Essa função tem que devolver o valor `True` quando a verificação é exitosa. Assim, em princípio, essa função não precisa ser sobrecarregada porque já devolve o valor `True` quando a execução do algoritmo acaba sem problemas. Contudo, é interessante sobrecarregar essa função se se quer implementar uma melhor verificação da implementação, relacionada com o algoritmo em si.

```
1  from src.algorithm.edmonds_blossom import EdmondsBlossom
2  from src.algorithm.carvalho_cheriyan import CarvalhoCheriyan
3
4  algorithm_map = {
5     "edmonds": EdmondsBlossom,
6     "carvalho-cheriyan": CarvalhoCheriyan,
7  }
```

Programa 4.2: *Dicionário de algoritmos implementados*

Finalmente, depois de implementar o novo algoritmo, basta adicionar uma entrada no dicionário `algorithm_map` que está no arquivo `src/api/utils.py` (veja o Programa 4.2). Esse dicionário mapeia o nome que é usado para se referir a um algoritmo nos comandos do programa com a classe que implementa tal algoritmo.

Capítulo 5

Comentários finais

Os grafos são estruturas matemáticas que são facilmente representadas graficamente. Essas representações visuais ajudam a entender melhor o comportamento dessas estruturas. Assim, é comum acompanhar a explicação de algoritmos em grafos com desenhos ou outras representações visuais. Em particular, o algoritmo de Edmonds é extremamente interessante de se visualizar devido as compressões e expansões das corolas.

Por tal motivo, as animações que podem ser geradas pelo programa desenvolvido para esse trabalho, além de visualmente estimulantes, podem ser úteis no ensino de algoritmos em grafos. Apesar de que a versão atual do programa suporta apenas dois algoritmos, o código é aberto e a sua estrutura foi desenvolvida com a intenção de que seja simples adicionar novos algoritmos.

No sentido de usar o programa como apoio ao ensino de algoritmos em grafos, uma possível melhora interessante seria adicionar um texto, do lado da animação, que explique o que está acontecendo a cada passo do algoritmo executado.

Outra possível melhora seria desenvolver uma interface gráfica, mais amigável para o usuário que ter que usar a linha de comandos, para executar as animações. Isso também poderia facilitar a inclusão de novos grafos para teste, desenhando-os na tela. No mesmo sentido, podem ser buscadas melhores formas de distribuição do software ou, inclusive, hospedar a plataforma online.

Agradecimentos

O professor Cláudio L. Lucchesi me orientou em todos os aspectos relacionados com este trabalho. Na ideia inicial do projeto, nas implementações dos algoritmos, nas provas de corretude, no uso de LaTeX, na organização da escrita, na norma padrão da língua portuguesa, no apoio emocional, o professor Lucchesi contribuiu de forma significativa com sua ampla experiência e o grande repertório de habilidades que ele possui. Aprendi muitas coisas ao longo do desenvolvimento deste trabalho graças a ele. Estou imensamente agradecido pela orientação e a ajuda.

Também gostaria de agradecer às pessoas que trouxeram meu interesse para a área de teoria da computação. Quando eu era calouro, a professora Nancy Espinoza, com quem eu fiz a minha primeira disciplina formal de programação, me recomendou entrar no grupo de maratona de programação da minha universidade. Ela não sabe, mas mudou a minha vida e sou muito grato a ela por isso. Foi assim que eu entrei no grupo ICPC PUCP, onde descobri a minha paixão por algoritmos, grafos e teoria da computação em geral. Agradeço a todos os membros desse grupo que conheci ao longo dos anos que estive nele. Agradeço também aos professores Cristina G. Fernandes, José Coelho e Marcel K. Silva, cuja paixão e interesse genuínos nas matérias que eles ministram consolidaram meu interesse na área.

Gostaria também de agradecer aos membros do grupo de extensão MaratonUSP. Com alguns deles eu tive discussões sobre os algoritmos que implementei neste trabalho, as quais melhoraram meu entendimento desses algoritmos. Em particular, agradeço ao Nathan Martins, cujas dúvidas sobre o algoritmo de Edmonds conseguiram que eu percebesse que ainda não tinha entendido esse algoritmo.

Ademais, agradeço ao Gustavo M. Carlos, quem tinha feito o seu trabalho de conclusão de curso recentemente e cujas dicas me foram muito úteis. Por outro lado, agradeço à Ariana Quispe e à Diana Junchaya pelos diversos comentários, muito úteis, sobre as versões preliminares deste trabalho, especialmente das animações.

Por último, gostaria de agradecer aos meus pais. Quando eu tinha no máximo 7 anos, eu disse pra minha mãe, Patricia Heredia, que eu queria modificar um videogame que eu jogava bastante. Então, por algum motivo, ela decidiu que era uma boa ideia me ensinar `C`. Eu apenas lembro da tela cheia de palavras com cores. É claro que eu não aprendi `C`, mas alguns conceitos de lógica já tinham sido instalados na minha cabeça. Sou muito grato a ela por sempre estimular a minha curiosidade e não ter subestimado as minhas capacidades. Também agradeço ao meu pai, Rafael L. Junchaya, pelo suporte durante a graduação, não apenas financeiro, senão também me impulsionando a seguir os meus

sonhos, inclusive nos momentos em que eu tinha desistido deles. O meu pai insiste em que algum dia inventarei um algoritmo e, devo admitir, essa confiança em mim me enche de motivação para seguir em frente. Eu devo tudo aos meus pais.

Referências

[1] Claude Berge. "Two Theorems in Graph Theory". Em: *Proceedings of the National Academy of Sciences of the United States of America* 43.9 (1957), pp. 842–844. ISSN: 00278424. URL: http://www.jstor.org/stable/89875 (acesso em 06/12/2023).

[2] Marcelo Henriques de Carvalho e Joseph Cheriyan. "An $O(VE)$ algorithm for ear decompositions of matching-covered graphs". Em: *ACM Trans. Algorithms* 1.2 (2005), pp. 324–337. DOI: 10.1145/1103963.1103969. URL: https://doi.org/10.1145/1103963.1103969.

[3] Alan Cobham. "The Intrinsic Computational Difficulty of Functions". Em: *Logic, Methodology and Phylosophy of Science, Proceeding of the 1964 International Congress*. Ed. por Yehoshua Bar-Hillel. Studies in Logic and the Foundation of Mathematics. North-Holland, 1965, pp. 24–30.

[4] Thomas H. Cormen et al. *Introduction to Algorithms*. The MIT Press, 2001. ISBN: 0262032937.

[5] Jack Edmonds. "Paths, Trees, and Flowers". Em: *Canadian Journal of Mathematics* 17 (1965), pp. 449–467.

[6] Leonhard Euler. "Solutio problematis ad geometriam situs pertinentis". Em: *Commentarii academiae scientiarum Petropolitanae* 8 (1741), pp. 128–140.

[7] Harold N. Gabow. "An Efficient Implementation of Edmonds' Algorithm for Maximum Matching on Graphs". Em: *J. ACM* 23.2 (abr. de 1976), pp. 221–234. ISSN: 0004-5411.

[8] J. Hopcroft e R.M. Karp. "An $n^{5/2}$ Algorithm for Maximum Matching in Bipartite Graphs". Em: *SIAM Journal on Computing* 2 (1973), pp. 225–231.

[9] Yifan Hu. "Efficient and High Quality Force-Directed Graph Drawing". Em: *Mathematica Journal* 10 (jan. de 2005), pp. 37–71.

[10] Dénes Kőnig. "Graphok és matrixok". Em: *Mat. Fiz. Lapok* 38 (1931), pp. 116–119.

[11] Harold W. Kuhn. "A tale of three eras: The discovery and rediscovery of the Hungarian Method". Em: *European Journal of Operational Research* 219.3 (2012). Feature Clusters, pp. 641–651. ISSN: 0377-2217. DOI: https://doi.org/10.1016/j.ejor.2011.11.008.

[12] Cláudio L. Lucchesi e U.S.R. Murty. *Perfect Matchings: A Theory of Matching Covered Graphs*. Springer International Publishing AG, 2024.

[13] Silvio Micali e Vijay V. Vazirani. "An $O(\sqrt{|V||E|})$ algorithm for finding maximum matching in general graphs". Em: *21st Annual Symposium on Foundations of Computer Science (SFCS 1980)*. 1980, pp. 17–27.

[14] Henry Martyn Mulder. "Julius Petersen's theory of regular graphs". Em: *Discrete Mathematics* 100.1 (1992), pp. 157–175. ISSN: 0012-365X.

[15] Tiago P. Peixoto. *Installation instructions*. Guia de instalação do software `graph-tool`. 2017. URL: https://git.skewed.de/count0/graph-tool/-/wikis/installation-instructions (acesso em 20/11/2023).

[16] Tiago P. Peixoto. "The graph-tool python library". Em: *figshare* (2014). DOI: 10.6084/m9.figshare.1164194. URL: http://figshare.com/articles/graph_tool/1164194 (acesso em 20/11/2023).

[17] Tiago P. Peixoto. *Welcome to graph-tool's documentation!* c2023. URL: https://graph-tool.skewed.de/static/doc/index.html (acesso em 15/10/2023).

[18] Julius Petersen. "Die Theorie der regulären graphs". Em: *Acta Mathematica* 15 (1891), pp. 193–220.

[19] M. D. Plummer. "Matching Theory—a Sampler: from Dénes Kőnig to the present". Em: *Discrete Mathematics* 100.1 (1992), pp. 177–219. ISSN: 0012-365X.

[20] M.D. Plummer e L. Lovász. *Matching Theory*. ISSN. Elsevier Science, 1986. ISBN: 9780080872322.

[21] Gábor Szárnyas. "Graphs and matrices: A translation of "Graphok és matrixok"by Dénes Kőnig (1931)". Em: (2020). Tradução ao inglês do artigo de Kőnig [10]. arXiv: 2009.03780 [math.HO].

[22] W. T. Tutte. "The Factorization of Linear Graphs". Em: *Journal of the London Mathematical Society* 22.2 (1947), pp. 107–111. ISSN: 0024-6107.