MO417 - Complexidade de Algoritmos I

Prof. Pedro J. de Rezende

1º Semestre de 2002

Versão de 16 de junho de 2002

Algoritmos em Grafos*

1 Introdução

Nesse tópico iremos estudar um pouco sobre algoritmos em grafos. Inicialmente começaremos apresentando conceitos básicos e definições sobre grafos, bem como maneiras de representá-los. A partir disso, então, começaremos a ver os algoritmos propriamente ditos, abordando os problemas referentes a grafos eulerianos, percurso em grafos, ordenação topológica, caminhos mais curtos a partir de um vértice, árvore geradora de custo mínimo, emparelhamento, circuitos hamiltonianos, todos os caminhos mais curtos e, por fim, fluxo em redes.

2 Definições

Antes de iniciarmos o estudo de algoritmos em grafos, vamos a algumas definições básicas:

Um **grafo dirigido** é um par (V, E), onde V é um conjunto finito de elementos chamados vértices e E é um conjunto de pares ordenados de vértices, chamados arestas. Da mesma forma, um **grafo não dirigido** é um conjunto $G = (V, E)$ sendo que E consiste de pares não ordenados de vértices.

Dizemos que um determinado vértice é **adjacente** a outro se houver uma aresta que os una. O **grau** de um vértice é o número de arestas incidentes no vértice. Nessa definição, temos que o **grau de entrada** (g_{in}) de um vértice é o número de arestas que chegam ao vértice, enquanto que o **grau de saída** (g_{out}) do mesmo é o número de arestas que partem do vértice.

Um **caminho** entre dois vértices é uma seqüência de vértices e arestas que une esses dois vértices. Dessa forma, dizemos que um vértice é **alcançável** a partir de outro se houver um caminho levando o último ao primeiro. Um caminho é **simples** se todos os vértices que o compõem forem distintos.

Chamamos de **subgrafo** de um grafo $G = (V, E)$, o grafo $G' = (V', E')$, tal que $V' \subseteq V$ e $E' \subseteq E$, ou seja, $G' \subseteq G$.

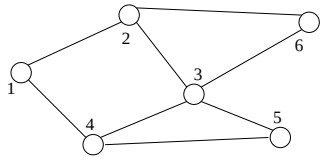
Um grafo é considerado **conexo** se cada par de vértices nele estiver ligado por um caminho e é **cíclico** se apresentar um **ciclo**, ou seja, se apresentar uma seqüência de vértices $v_1, v_2, \ldots, v_k$ tal que $v_k = v_1$.

Por fim, uma **árvore** é um grafo conexo, acíclico.

*Escriba: Norton Trevisan Roman.

3 Representação de Grafos

Existem várias maneiras de representar grafos. Aqui falaremos das duas mais importantes: lista de adjacências e matriz de adjacências. Considere o grafo:



A representação com lista de adjacências de um grafo $G = (V, E)$ é um vetor de $|V|$ listas, uma para cada vértice em V de modo que, para cada $u \in V$, a lista ligada a u contém os vértices v tais que $(u, v) \in V$. No caso do exemplo acima, a lista é:

1	$\rightarrow$	2	$\rightarrow$	4				
2	$\rightarrow$	1	$\rightarrow$	3	$\rightarrow$	6		
3	$\rightarrow$	2	$\rightarrow$	4	$\rightarrow$	5	$\rightarrow$	6
4	$\rightarrow$	1	$\rightarrow$	3	$\rightarrow$	5		
5	$\rightarrow$	3	$\rightarrow$	4				
6	$\rightarrow$	2	$\rightarrow$	3				

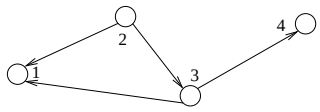
É fácil verificar que a quantidade de memória necessária para uma lista de adjacências é $O(|V| + |E|)$.

A representação com matrizes, por sua vez, de um grafo $G = (V, E)$ é uma matriz $|V| \times |V|$ tal que $a_{i,j} = 1$ se existir aresta entre v_i e v_j, e 0 se não existir tal aresta. No exemplo acima a matriz é:

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array} \begin{array}{c} \begin{array}{cccccc} 1 & 2 & 3 & 4 & 5 & 6 \end{array} \\ \begin{pmatrix} 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

Também é fácil verificar que a quantidade de memória necessária para essa representação é $\Theta(|V|^2)$.

Mas, e se o grafo for dirigido? Nada muda nas definições. Por exemplo, considere o grafo:



Sua representação em lista de adjacências será:

1	→	∅		
2	→	1	→	3
3	→	1	→	4
4	→	∅		

e em matriz será:

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

4 Grafos Eulerianos

4.1 Histórico e Definição

Antes de definir um grafo euleriano, vamos dar uma olhada em como eles surgiram: a história se passa no século XVIII, em Königsberg[2], na Prússia. Königsberg era cortada por um rio que formava uma ilha no interior da cidade, havendo pontes conectando essa ilha ao resto da cidade.

Por essa época, havia uma controvérsia entre os moradores locais que chegou aos ouvidos do matemático Leonhard Euler[3].

Euler descreveu a controvérsia na seguinte carta [7]:

> "O problema, que eu entendo ser bem conhecido, é descrito como segue: Na cidade de Königsberg, na Prússia, há uma ilha chamada Kneiphhof, com os dois braços do rio Pregel fluindo em volta dela. Há 7 pontes – a, b, c, d, e, f e g – cruzando estes dois braços." (Figura 1).
>
> "A questão é se uma pessoa pode planejar uma caminhada de modo que ela cruze cada uma destas pontes uma única vez, e não mais que isso..."

Vamos , então, reconstruir os passos de Euler na formulação do problema. Primeiro, olhemos mais atentamente para as pontes e o rio na Figura 2.

Retiremos, agora, as distrações, chegando à Figura 3 e simplificando o esquema mais uma vez, temos o grafo da Figura 4.

Analizando, então, este grafo, Euler resolveu a questão provando que uma caminhada assim é possível se e somente se o grafo for conexo e todos os seus vértices tiverem grau par.

[2] hoje a cidade russa de Caliningrado.

[3] Nascido em 15/04/1707 em Basel, Suíça e falecido em 18/09/1783 em São Petesburgo, Rússia.

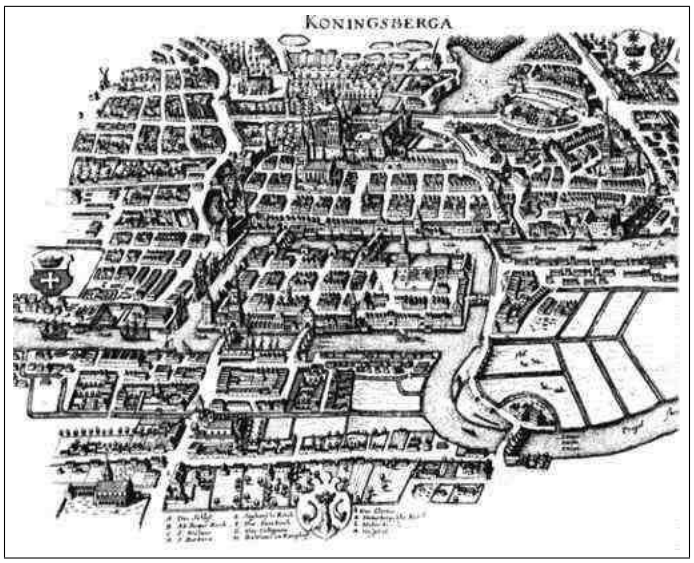


Figura 1: A cidade de Königsberg.

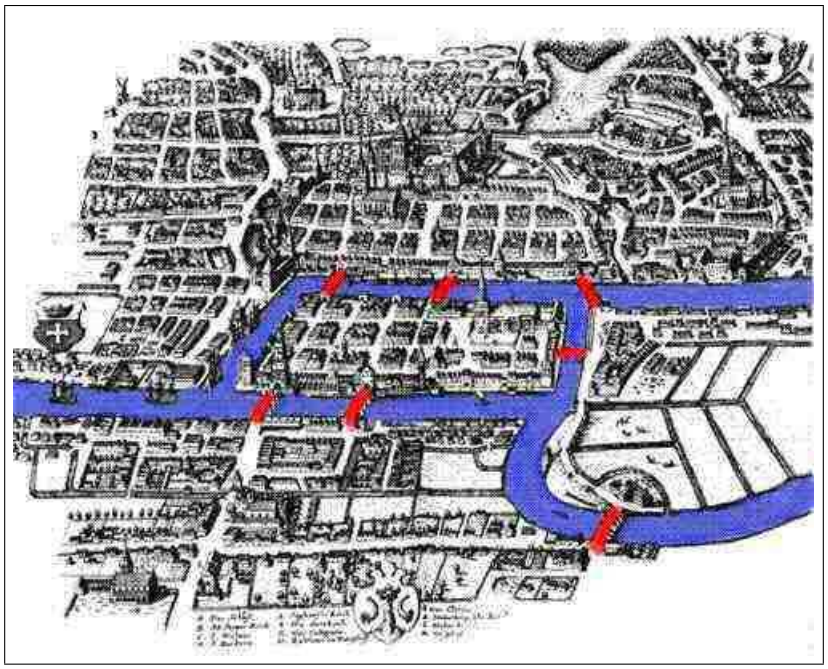


Figura 2: A cidade de Königsberg, com as 7 pontes e o rio em evidência.

Assim, Euler mostrou que, uma vez que o grafo da figura 4 tem vértices

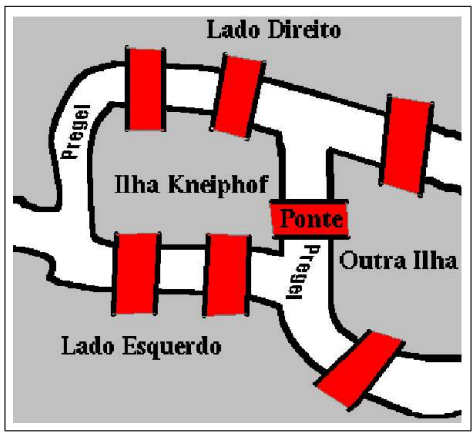


Figura 3: Esquema da cidade de Königsberg, com as 7 pontes e o rio em evidência.

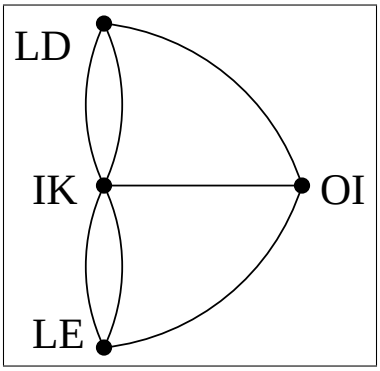


Figura 4: Grafo das 7 pontes e o rio.

de grau ímpar, a resposta ao problema de Königsberg era que tal caminhada era impossível. Desde então todo grafo conexo cujos vértices têm grau par é chamado de grafo euleriano, e um caminho fechado em um grafo que passe por cada aresta deste exatamente uma vez é chamado de circuito (ou ciclo) euleriano.

Agora vamos dar uma olhada em uma pova por indução para o teorema de Euler [10]. Vale notar que essa prova constitui também um algoritmo para a construção de um circuito euleriano em um grafo. Uma prova alternativa pode ser encontrada em [4].

O problema de Euler pode ser formulado da seguinte maneira: "Dado um grafo conexo não dirigido $G = (V, E)$, tal que todos os seus vértices têm grau par, quero encontrar um caminho fechado P tal que cada aresta de E aparece em P exatamente uma vez."

Agora, será que a restrição de todos os vértices de G terem grau par é necessária? Vejamos, em um caminho fechado, saímos de um vértice, v_1, cami-

nhamos pelo grafo e voltamos a v_1. Isso significa que toda vez que entramos em um vértice $v_i \neq v_1$ temos que sair dele, e assim percorremos todas as arestas. Como, ao percorrermos todas as arestas do grafo entramos e saímos de cada vértice o mesmo número de vezes e podemos usar cada aresta somente uma vez, o número de arestas adjacentes a cada vértice deve ser par. Então, para haver tal caminho, essa condição é necessária.

Vamos, então, tentar resolver o problema formulado acima por meio de uma prova por indução. O primeiro passo é decidir em que será feita a indução. Se fizermos indução simples no número de vértices ou arestas, no passo poderemos, ao retirar um vértice ou aresta, perder as propriedades de grau par dos vértices do grafo.

Assim, removemos um ciclo do grafo. Note que ao removermos o conjunto de arestas do grafo que forma um ciclo, mantemos a propriedade de que o grau dos vértices do grafo deve ser par. Então, assim fica prova:

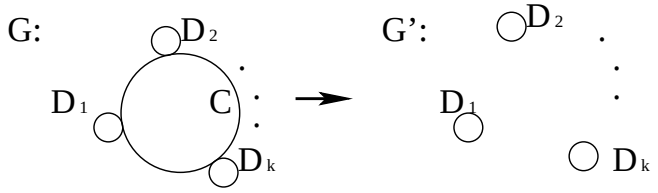
Prova: *(por indução forte no número m de arestas do grafo)*

Base: $|E| = 0 \Rightarrow |V| = 1$, pois é conexo. Sei achar o ciclo.

Hipótese de Indução: Sou capaz de determinar um ciclo euleriano num grafo conexo com $|E| < m$ arestas, cujos vértices têm grau par.

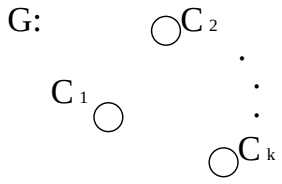
Passo: Seja $G = (V, E)$ um grafo conexo com $|E| = m$. Como o grau dos vértices de G é par, então G contém pelo menos um ciclo, C.

Seja G' o grafo obtido a partir de G retirando-se todas as arestas de C. G' pode ser desconexo. Sejam $D_1, D_2, \ldots, D_k$ as componentes conexas de G'.

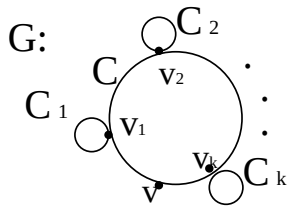


O grau de cada vértice em G' deve ser par, uma vez que o número de arestas adjacentes a cada vértice é par. Então, o grau dos vértices de cada D_i, $1 \leq i \leq k$, é par.

Como o número de arestas em D_i é menor que m, pela hipótese de indução sou capaz de determinar, para cada componente, um ciclo euleriano. Denote esses ciclos por $C_1, C_2, \ldots, C_k$:



Agora, recoloco C em G':

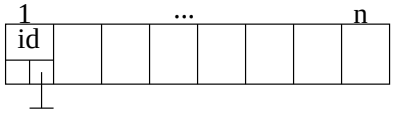


Comecemos, então, em um vértice em C, digamos, v. Vamos percorrendo C e, ao encontrar um vértice que pertence a uma das componentes D_i, atravessamos C_i, e assim prosseguimos até voltar a v, criando um caminho euleriano que, nesse caso é $v \to v_1 \overset{viaC_1}{\to} v_1 \to v_2 \overset{viaC_2}{\to} v_2 \to \ldots \to v_k \overset{viaC_k}{\to} v_k \to v$.

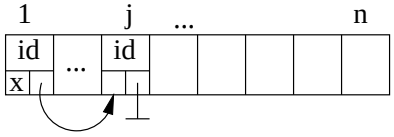
Mas note que, apesar da prova estar completa, ela possui uma falha como algoritmo: não diz como faço para encontrar um ciclo em G. Então vamos produzir um algoritmo para tal.

Vamos representar $G = (V, E), |v| = n$, como uma matriz de adjacências. Esse problema, então é resolvido nos moldes do *union find*, apresentado em [10]:

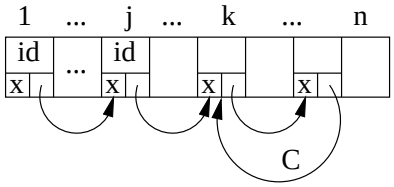
- Crio um vetor, V, de n elementos, onde cada elemento possui uma identificação, um campo para que seja "marcado" (ver mais adiante) e um ponteiro para outro elemento.



- Visito a adjacência de $V[1]$ na matriz de G (linha 1) e vejo em que coluna há um "1", indicando que os vértices têm aresta em comum. Suponha que seja na coluna j, então marco $V[1]$ e o faço apontar para $V[j]$.



- Visito, então, a adjacência de $V[j]$ na matriz (linha j) e repito a operação até encontrar um vértice já marcado, $V[k]$.



Nesse ponto, achei um ciclo, C, que inclui $V[k]$.

Exercícios:

1. Qual a complexidade do algoritmo acima?
2. Será que há um algoritmo representando o grafo como uma lista de adjacências que seja mais eficiente que os algoritmos que representam esse grafo como uma matriz? Se sim, apresente tal algoritmo.

Um outro método para achar um ciclo em grafos usa multiplicação de matrizes: analizemos a matriz $M^2 = M \times M$, onde M reresenta o grafo onde buscamos um ciclo. Um elemento $a_{k,l} \in M^2$ será positivo somente se houver i tal que $a_{k,i}$ e $a_{i.l} \in M$ sejam positivos, pois $a_{k,l} = \sum_{i=1}^{n} a_{k,i} \cdot a_{i,l}$. Mas isso é um caminho de k a l de comprimento 2.

Da mesma forma podemos verificar que, se $a_{k,l} > 0$, $a_{k,l} \in M^3$, então teremos um caminho de k a l de tamanho 3.

Assim, M^n dará os caminhos possíveis de tamanho n no grafo representado por M. Então vemos que, se $a_{i,j} = 0$, $a_{i,j} \in M^n$, então não há caminho de i a j em G, ou seja, i e j não pertencem à mesma componente conexa. Assim, para cada uma dessas matrizes, verificamos se $a_{i,j} > 0$, o que indicaria um caminho de a_i a $a_j > 0$, um ciclo.

Este é um algoritmo muito ineficiente, principalmente para grafos com grande número de vértices. Um algoritmo alternativo para achar ciclos em um grafo é o seguinte: partindo de um vértice, v, arbitrário, do grafo, marco esse vértice e todos os vértices adjacentes a ele. Vou repetindo a operação para cada vértice na adjacência de v. Com isso estou marcando uma componente conexa de G e se, durante o processo, eu encontrar um vértice já marcado, então achei um ciclo.

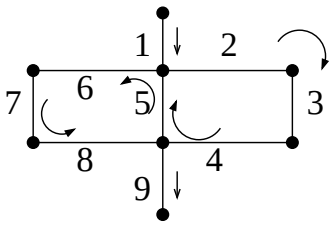
Exercício: Qual a complexidade dos dois algoritmos acima (matrizes e o último, de marcar vértices)?

Um corolário importante para o teorema de Euler é o seguinte:

Corolário 1 *Um grafo conexo tem um caminho euleriano se tiver no máximo 2 vértices de grau ímpar.*

A prova deste corolário pode ser vista em [4]. Naturalmente, esse caminho não será um ciclo, mas ainda assim terá a propriedade de percorrer todas as arestas do grafo uma única vez. Nesse caso, para achar o caminho, começamos

de um vértice de grau ímpar, passamos pelos de grau par, terminando no outro vértice de grau ímpar:



4.2 O problema do carteiro chinês

Esse é um problema de aplicação do teorema de Euler [4]: em seu trabalho, um carteiro pega as cartas no escritório do correio, as entrega e, então, volta ao escritório. Ele deve cobrir cada rua na sua área pelo menos uma vez e, sujeito a essa condição, deseja escolher uma rota de modo que ande o mínimo possível. Este problema foi proposto por um matemático chinês [9] e, por isso recebe esse nome.

Representemos as ruas na área coberta pelo carteiro por arestas em um grafo, G, sendo os vértices as interseções entre as ruas. Se G for euleriano, qualquer ciclo euleriano servirá. Se não for, então o problema é o de encontrar um caminho C que contenha a menor distância possível a ser caminhada pelo carteiro nas condições estabelecidas (uma resposta pode ser encontrada em [4]).

5 Percurso em Grafos

Existem dois métodos fundamentais de percurso em grafos: DFS e BFS.

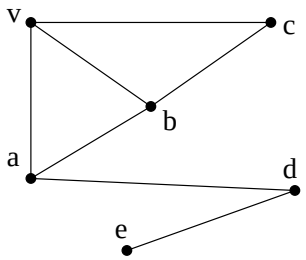
5.1 DFS - Depth First Search (Percurso em Profundidade)

A idéia básica da DFS é buscar "mais a fundo" no grafo quando possível [6]. Assim, a partir de um vértice v, as arestas ainda não exploradas o são e, ao final, a busca retorna ao vértice w, que levou ao descobrimento de v pela aresta (w, v) e explora suas arestas ainda não visitadas. Assim a busca continua até que todos os vértices sejam descobertos.

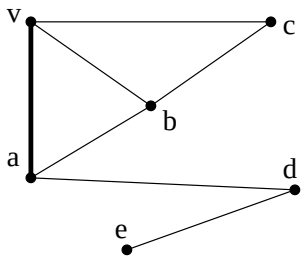
A seguir veremos o uso da DFS para grafos dirigidos e não dirigidos.

5.1.1 Grafos não dirigidos

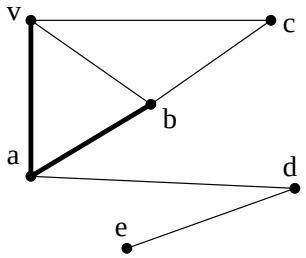
Considere o grafo a seguir:



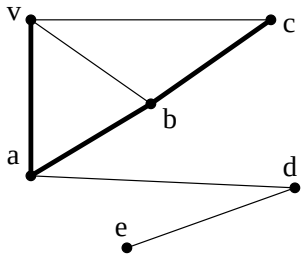
Vamos começar a busca a partir de um vértice arbitrário, v, chamado de raiz da DFS. Temos três caminhos para seguir: (v, a), (v, b) e (v, c). Vamos escolher (v, a) e percorrer:



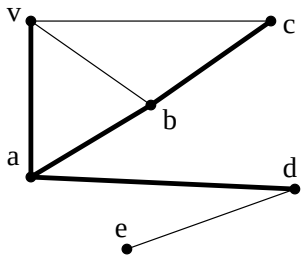
Novamente, temos algumas escolhas: (a, b), (a, v) e (a, d). Como já conheço v, devo escolher entre (a, b) e (a, d). Escolhamos (a, b):



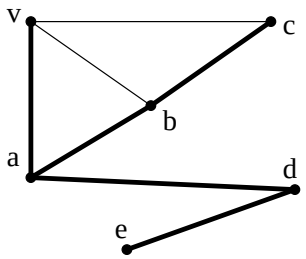
De b temos 3 escolhas: (b, a), (b, v) e (b, c). Mas como já conheço v e a, sobrou apenas uma escolha: (b, c).



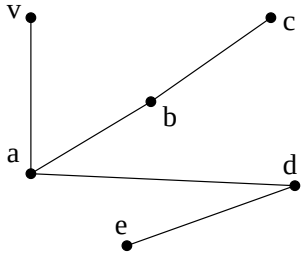
Chegando em c há 2 possibilidades: (c, v) e (c, b). Mas já conheço v e b, então não tenho para onde aprofundar, e sei que há vértices ainda não descobertos. Nesse caso volto pelo caminho que fiz até o vértice b (essa volta é também chamada de *backtracking*) e verifico se há algum caminho que me leve a um vértice não visitado a partir de b. No caso não há, então volto novamente pelo caminho por onde vim, parando em a. Em a, noto que (a, d) me leva a um vértice que não conheço. Então sigo por este caminho:



Em d, tenho 2 ecolhas, mas a única que me leva a um vértice não descoberto é (d, e):



De e não temos mais para onde ir e já descobrimos todos os vértices, então o percurso foi:



Que é uma árvore, a chamada árvore DFS ou árvore de percurso em profundidade.

O algoritmo geral para essa busca é mostrado a seguir. Nele, *prework* e *postwork* dependem da aplicação da DFS e definem ações que devem ser tomadas, respectivamente, quando um vértice é marcado e quando temos que voltar

por uma aresta ou quando vemos que a aresta leva a um vértice já marcado [10]:

Algoritmo DFS(G,v):
Entrada: $G = (V, E)$, um grafo não dirigido, e $v \in V$.
Saída: depende da aplicação.

início
    marque v;
    execute *prework* em v;
    para todas as arestas (v, w) faça
        se w não estiver marcado então DFS(G,w);
        execute *postwork* para (v, w);
fim

Figura 5: Algoritmo DFS.

Em [10] pode ser encontrada uma demonstração de que, se o grafo for conexo, então todos os seus vértices serão marcados pelo algoritmo acima, e todas as suas arestas serão visitadas pelo menos uma vez durante a execução do algoritmo.

5.1.2 Componentes Conexas Múltiplas

Vale notar que o algoritmo acima funciona somente para grafos conexos. Mas, e se nosso grafo não for conexo? Nesse caso, usamos o algoritmo acima e, ao final deste, teremos marcado os vértices de uma componente conexa. Recomeçamos, então, de um vértice arbitrário ainda não marcado e usamos DFS novamente. Fazendo isso para todos os vértices teremos várias árvores DFS representando as componentes conexas do grafo.

O algoritmo para efetuar DFS em grafos desconexos é dado abaixo (uma versão não recursiva é apresentada em [10]):

Algoritmo Componentes_Conexas(G,v):
Entrada: $G = (V, E)$, um grafo não dirigido.
Saída: v.Componente terá o número da componente contendo v.

início
    Numero_da_componentes := 1;
    enquanto houver um vértice não marcado faça
        DFS(G,v);
        {usando o *prework*:
            v.Componente := Numero_da_componente;}
        Numero_da_componente := Numero_da_componente + 1;
fim

Figura 6: Algoritmo DFS para múltiplas componentes conexas.

Com o *prework* e *postwork* adicionados à chamada do algoritmo DFS feita no algoritmo acima consigo contar o número de componentes conexas do grafo e rotular cada componente com esse número.

Agora vamos calcular a complexidade de nosso algoritmo (calcularemos a complexidade do algoritmo para grafos não conexos, por ser mais geral): cada aresta, pelo algoritmo, é vista 2 vezes (uma de cada vértice em seus extremos). Então o tempo de execução é $O(|E|)$. Contudo, uma vez que todos os vértices devem ser marcados (inclusive os desconexos), o tempo de execução total será $O(|V| + |E|)$.

Exercício: Qual o efeito sobre essa complexidade se uso uma matriz ou uma lista de adjacências para representar o grafo?

5.1.3 Numeração de Vértices e Construção da Árvore DFS

Durante a execução do algoritmo DFS podemos fazer algumas coisas a mais, como numerar os vértices do grafo e obter uma árvore geradora de G, a árvore DFS. Para tal basta incluir algumas ações no *prework* e *postwork* do algoritmo.

Modificando o *prework* do algoritmo de DFS da seguinte maneira:

Algoritmo NumeracaoDFS(G,v):
Entrada: $G = (V, E)$, um grafo não dirigido, e $v \in V$.
Saída: v.DFS terá o número do vértice.

Inicialmente Numero_DFS := 1;
Use DFS com o seguinte *prework*:
    v.DFS := Numero_DFS;
    Numero_DFS := Numero_DFS + 1;

Figura 7: Algoritmo para numeração de vértices em DFS.

teremos um algoritmo para contar os vértices do grafo, rotulando cada um. Para obter uma árvore DFS modificamos *postwork* desse modo:

Algoritmo ArvoreDFS(G,v):
Entrada: $G = (V, E)$, um grafo não dirigido, e $v \in V$.
Saída: T, árvore DFS de G.

Use DFS com o seguinte *postwork*:
    se w não estava marcado então inclua a aresta (v, w) em T;
    {a linha acima pode ser incluída no if da linha 4 do algoritmo DFS}

Figura 8: Algoritmo para construção da árvore DFS.

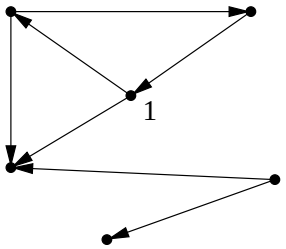
É possível mostrar que, dado um grafo não dirigido conexo, G, e uma árvore DFS de G, T, construída com o algoritmo acima, então toda aresta de G ou

pertence a T ou conecta 2 vértices em G, um dos quais é o ancestral[4] do outro em T (a prova pode ser vista em [10]). Essa é a principal propriedade de árvores DFS não dirigidas.

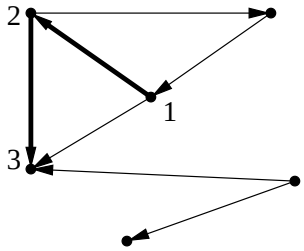
5.1.4 Grafos Dirigidos

Vamos analizar, agora, como fica a DFS para grafos dirigidos. A árvore DFS é obtida usando-se o mesmo algoritmo para detecção de múltiplas componentes conexas, apresentado acima.

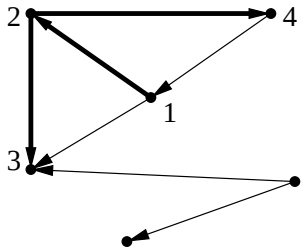
Considere, por exemplo, o grafo abaixo [1]. Primeiro escolhemos um vértice inicial:



Agora nos aprofundamos no grafo a partir desse vértice:

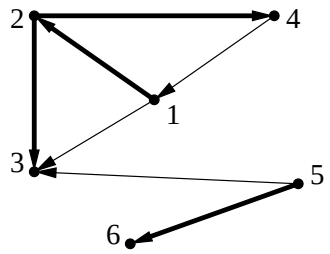


Então voltamos até encontrarmos um caminho não explorado:

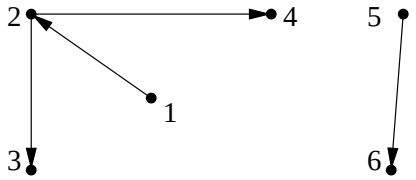


Continuamos esse processo enquanto houverem vértices não marcados

[4]Um vértice v é ancestral de um vértice w em uma árvore T com raiz r se v estiver no caminho único de r a w em T.



Gerando as árvores:



A propriedade principal de árvores DFS dirigidas afirma que, em um grafo dirigido, se (v, w) é uma aresta do grafo tal que v foi marcado antes de w, então w é um descendente[5] de v na árvore DFS de G (a demosntração pode ser encontrada em [10]).

Outra propriedade interessante da DFS é que, enquanto efetuamos o percurso, podemos classificar as arestas do grafo em 4 tipos [6]:

- Arestas da Árvore (*tree edges*): arestas na floresta DFS.
- Arestas de Retorno (*back edges*): arestas que conectam um determinado vértice a algum de seus ancestrais em uma árvore DFS.
- Arestas Progressivas (*forward edges*): conectam um vértice a algum de seus descendentes, sem pertencerem à árvore.
- Arestas de Cruzamento (*cross edges*): as restantes. Conectam vértices "não relacionados" na árvore DFS.

DFS pode, também, ser usada para determinar se um grafo contém um ciclo. Considere o seguinte algoritmo:

[5] um vértice w é descendente de um vértice v em uma árvore T se v for ancestral de w em T.

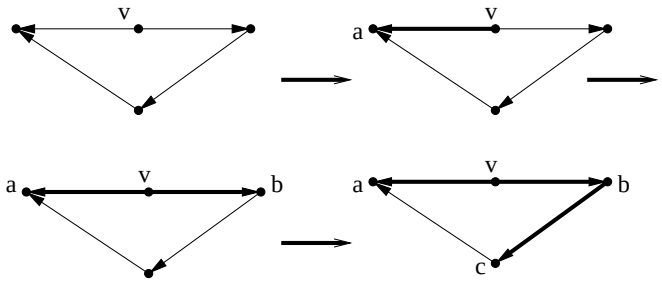
Algoritmo CicloDFS(G,v):
Entrada: $G = (V, E)$, um grafo não dirigido e v o vértice inicial
{Inicialmente, v_i.no_caminho := falso para todo $v_i \in V$.}
Saída: 'verdadeiro' se houver ciclo e 'falso' se não.

início
    marque v;
    para todas as arestas (v, w) faça
        se w não estiver marcado então
            v.no_caminho := verdadeiro;
            DFS(G,w);
        se w.no_caminho então
            CicloDFS := verdadeiro;
            termine;
        se w for o último vértice na lista de v então v.no_caminho := falso;
fim

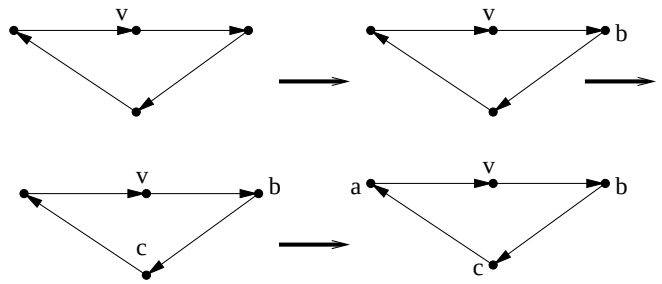
Figura 9: Algoritmo para a obtenção de um ciclo usando DFS.

A idéia por trás deste algoritmo é que, ao passar pelos vértices, eu os marco e se no decorrer do percurso eu encontrar um vértice já marcado, eis o ciclo. Naturalmente, ao efetuar um retorno (*backtracking*), devo retirar a marca do vértice.

Por exemplo, façamos o algoritmo rodar no seguinte grafo, iniciando em v (os vértices são nomeados à medida em que são descobertos):



E o algoritmo retorna falso. Já se o grafo for:

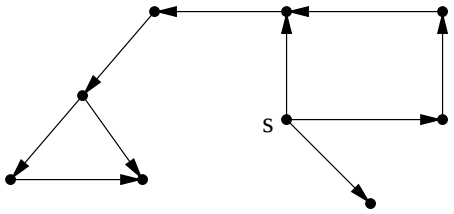


o algoritmo retorna verdadeiro.

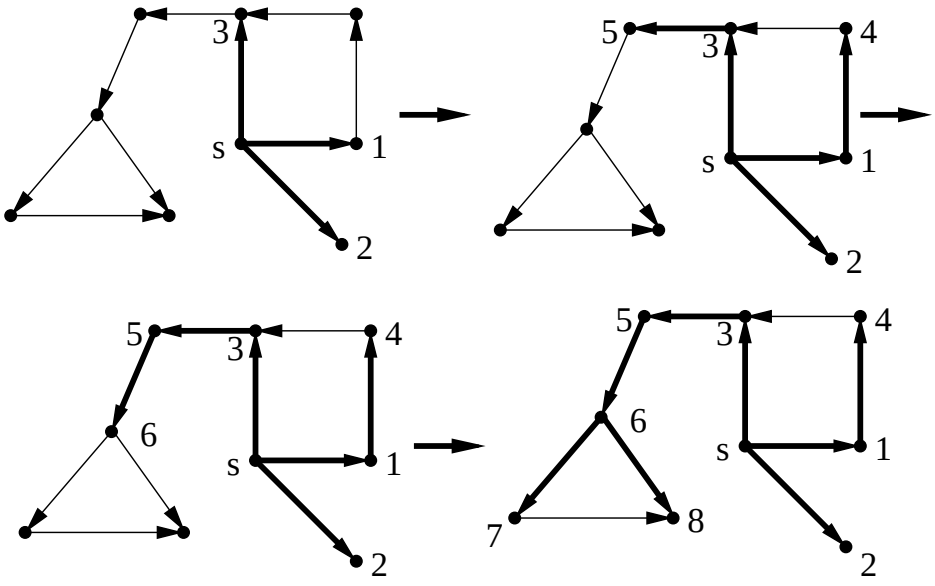
5.2 BFS - Breadth First Search (Percurso em Largura)

A idéia da busca em largura é bastante simples: os vértices do grafo são visitados nível a nível, ou seja, todos os vértices a uma distância k do vértice inicial[6] são visitados antes de qualquer vértice a uma distância $k+1$ do inicial.

Considere o seguinte exemplo (o vértice inicial foi rotulado s) [2]:

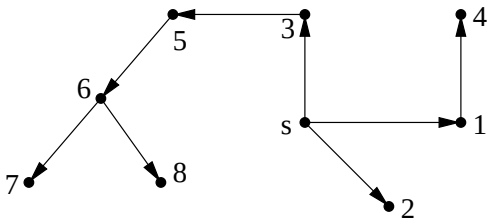


Começamos, então, a visita a todos os vértices adjacentes a s e assim por diante (note que não há *backtracking*).



Gerando a árvore:

[6] Vértice de onde foi iniciada a busca.



Abaixo podemos ver o algoritmo usado para efetuar esse percurso:

Algoritmo BFS(G,v):
Entrada: $G = (V, E)$, um grafo não dirigido, e $v \in V$.
Saída: depende da aplicação.
início
    marque v;
    coloque v em uma fila;
    enquanto a fila não estiver vazia faça
        remova o primeiro vértice w da fila;
        execute *prework* em w
        para todas as arestas (w, x) tais que x não está marcado faça
            marque x;
            inclua (w, x) na árvore T;
            coloque x na fila;
fim

Figura 10: Algoritmo para BFS.

Vale notar que, no algoritmo, usamos uma fila como estrutura auxiliar para podermos visitar os vértices do modo determinado pelo percurso. Assim, quando um vértice é visitado ele é retirado da fila e seus vértices adjacentes ainda não visitados são colocados no final desta. Ou seja, os vértices de um nível sempre estarão na frente dos vértices do nível seguinte.

A figura 11 mostra, usando o grafo do exemplo anterior, a movimentação dos vértices na fila enquanto o algoritmo BFS roda.

Além disso, vale notar que o algoritmo, à medida que é executado, também constrói a árvore BFS.

Uma observação importante sobre essa árvore é que ela é tal que a distância entre 2 vértices na árvore é a menor distância entre eles no grafo, considerando-se a distância entre 2 vértices como sendo o número de arestas em um caminho que leve de um vértice a outro. Dessa forma, o caminho entre 2 vértices na árvore BFS representa o menor caminho, em número de arestas, entre esses vértices no grafo (a demonstração para essa afirmação pode ser vista em [6]).

Vale também notar que, como na DFS, o algoritmo para a BFS pode ser modificado para encontrar ciclos e para numerar as componentes conexas em grafos não dirigidos.

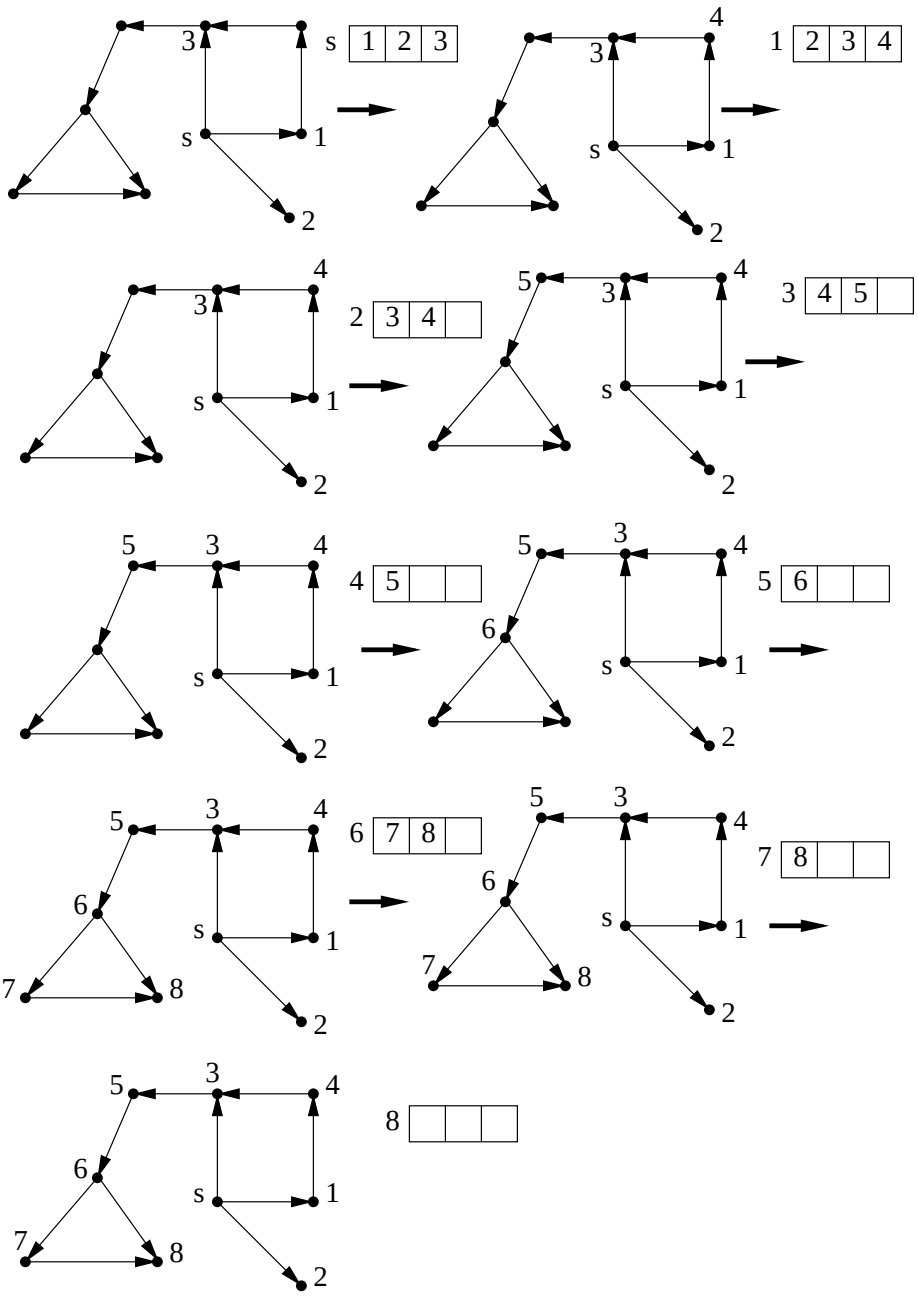
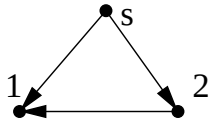


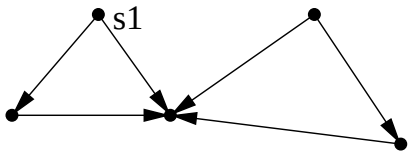
Figura 11: Movimentação dos vértices na fila enquanto o algoritmo da figura 10 roda.

No primeiro caso, é só verificar se, ao visitar um vértice, algum vértice de sua adjacência já foi marcado. Se foi, eis nosso ciclo, pois isso significa que já passamos por ele. Note que esse algoritmo funciona somente para grafos não dirigidos. No grafo abaixo ele falha:

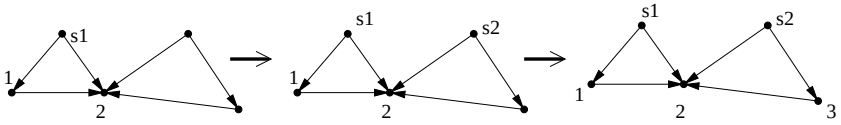


Nesse caso, ele acusaria erroneamente o ciclo, ao considerar $(2, 1)$.

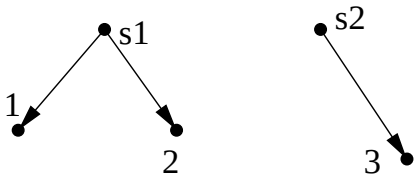
Da mesma forma, para enumerar as componentes conexas do grafo, basta repetirmos a BFS enquanto houver vértices não marcados. Assim, o percurso será efeturado em cada componente conexa. Novamente, em grafos dirigidos esse algoritmo apresenta problemas. Considere o seguinte grafo conexo:



Rodando BFS com esse grafo teremos;



O que nos dá as árvores:



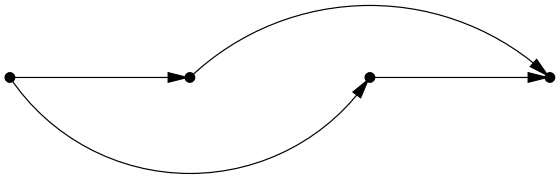
O algoritmo indicaria que o grafo é desconexo, o que não é verdade.

Agora vamos analizar a complexidade do algoritmo para a BFS [6]. Segundo o algoritmo, cada vértice entra e sai da fila uma vez, gerando um tempo $O(|V|)$. Mas, para cada vértice, sua lista de adjacências é verificada, um tempo $O(|E|)$ é gasto nisso. Então, o tempo de execução total do algoritmo é $O(|V| + |E|)$.

6 Ordenação Topológica

A ordenação topológica recebe um grafo dirigido acíclico e retorna uma lista ordenada dos vértices de modo que, se houver uma aresta (v, u) no grafo, então v aparecerá antes de u na lista [3].

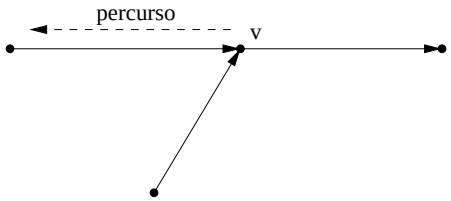
Ou seja, a ordenação topológica busca um alinhamento dos vértices do grafo de modo que as arestas apontem da esquerda para a direita:



Mas a questão agora é: como acho essa ordenação? Vejamos o seguinte lema:

Lema 1 *Todo grafo dirigido acíclico G tem um vértice com grau de entrada zero.*

Demonstração: pego um vértice qualquer de G, v. Se v tiver grau de entrada 0 está provado. Senão, escolho uma aresta qualquer que chegue a v e a percorro em sentido contrário:



Como o grafo é finito e acíclico, necessariamente chegarei a um vértice a partir do qual não terei como continuar com esse procedimento, esse é um vértice de grau de entrada 0.

Então agora podemos definir nosso problema:

"Dado um grafo dirigido acíclico com n vértices, queremos rotular os vértices de 1 a n tal que, se v for rotulado com k, então todos os vértices alcançáveis a partir de v por um caminho dirigido são rotulados com números $> k$.

Base: $|V| = 1$. Trivial.

Hipótese de Indução: Sei como rotular um grafo dirigido acíclico com $< n$ vértices.

Passo: Seja $G = (V, E)$, dirigido, acíclico, $|V| = n$, Acho um vértice de G, v, que seja uma fonte para G. Ou seja, um vértice com grau de entrada 0. O lema acima nos mostra como achar esse vértice. Retiro esse vértice do grafo e, por hipótese de indução, sei resolver o problema para $G' = G \setminus \{v\}$. Após

rotulado G' basta incluir v no grafo. Ele deverá ser rotulado com um número menor que o menor rótulo de G' por ser fonte.

Agora vamos ver um algoritmo que faz exatamente o que nossa demonstração mostra, mas não recursivamente (figura 12). No algoritmo, em cada vértice há o registro de seu grau de entrada. Então, ao iniciar, o algoritmo retira os vértices de grau 0, v_i, e os coloca em uma fila. Para cada vértice w_j tal que a aresta $(v_i, w_j) \in G$, decremento o grau de entrada de w_j em 1. Novamente retiro os vértices de grau 0, colocando na fila e executando o procedimento acima. O algoritmo termina quando todos os vértices de G tiverem sido rotulados (estiverem na fila).

```
Algoritmo OrdTop(G):
Entrada: G = (V, E), um grafo dirigido acíclico.
Saída: O campo 'rotulo' indica uma ordenação topológica de G.
início
    inicialize v.g_in para todos os vértices (com DFS);
    rotulo_G := 0;
    para i:=1 até n faça
        se v_i.g_in = 0 então coloque v_i na fila;
    repita
        remova um vértice v da fila;
        rotulo_G := rotulo_G + 1;
        v.rotulo := rotulo_G;
        para toda aresta (v, w) faça
            w.g_in := w.g_in - 1;
            se w.g_in = 0 então coloque w na fila;
    até que a fila esteja vazia
fim
```

Figura 12: Algoritmo para Ordenação Topológica.

(Em [6] pode ser encontrado um algoritmo para ordenação topológica que usa DFS para ordenar os vértices).

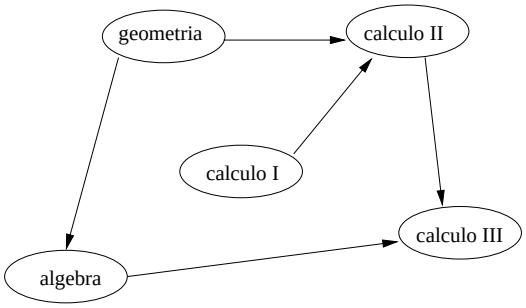
A complexidade do algoritmo acima é $O(|V| + |E|)$, pois esse é o tempo que leva para inicializar o grau de entrada dos vértices do grafo. Achar um vértice de grau 0 pode levar $O(|V|)$ (linha 5). Como cada aresta do grafo é considerada uma vez só, o número de vezes que os graus de entrada devem ser atualizados é $O(|E|)$. Então a complexidade total é $O(|V| + |E|)$. Mas vale notar que isso supõe que o grafo é representado por uma lista de adjacências. Se for por uma matriz, a complexidade sobe para $O(|V|^2)$, pois esse é o tempo para inicializar os graus de entrada no grafo, usando uma matriz.

Exercício: Verifique a complexidade de $O(|V|^2)$ para matrizes no algoritmo acima.

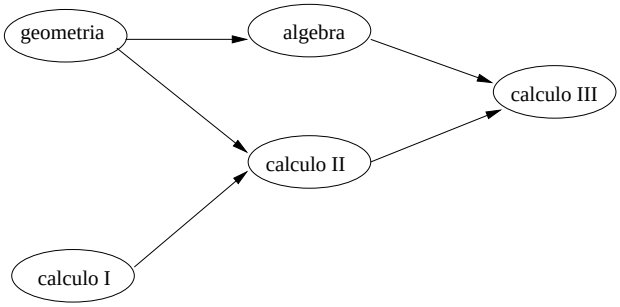
6.1 Exemplos:

Vamos observar 2 exemplos de aplicação da ordenação topológica:

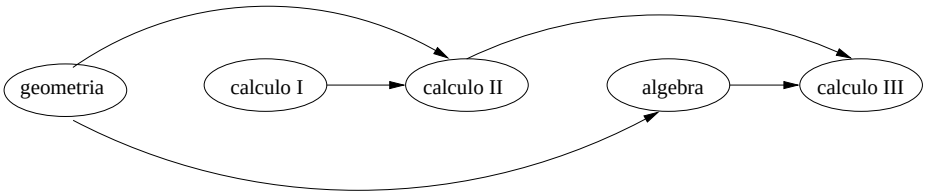
Árvore de Pré-requisitos: suponha que você quer organizar seu horário na faculdade e, para isso, quer saber que matérias podem ser feitas a cada semestre. O grafo das matérias é mostrado na figura abaixo (uma aresta de a para b indica que a é pré-requisito para b):



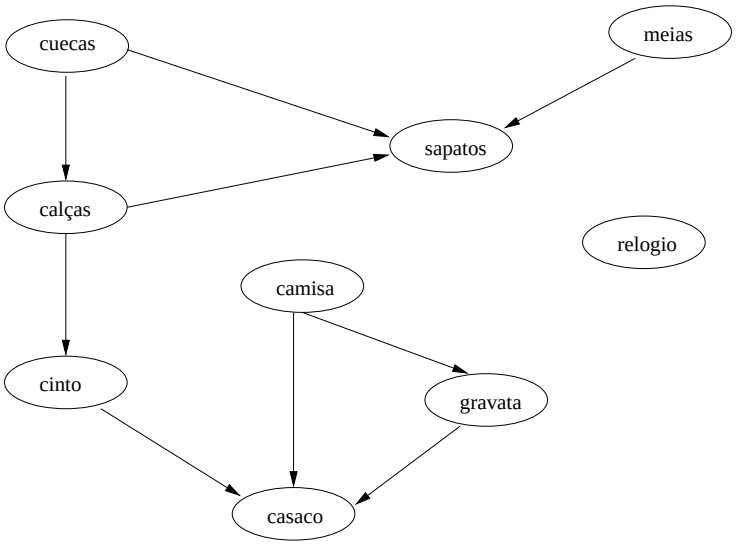
Então, executando uma ordenação topológica, um resultado pode ser



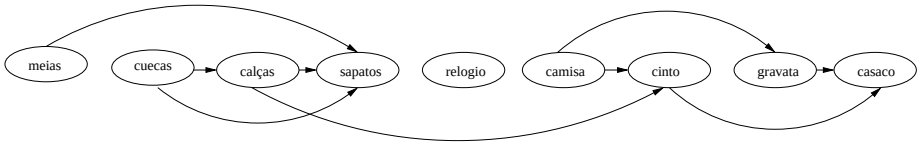
Ou



Vestir-se pela manhã [6]: suponha que sua sonolência impede seu processamento cerebral pela manhã e você está casado de sair de casa vestindo as cuecas por fora das calças. Então você resolve fazer uma lista do que deve vestir e em que ordem. Primeiro você começa fazendo um grafo com as roupas:



Efetuando-se a ordenação, conforme o algoritmo apresentado, teremos a seguinte lista(podem haver variações, já que há mais de um vértice de grau de entrada 0):



Agora sim você pode sair tanqüilo de casa.

7 Caminhos Mais Curtos

O problema de caminhos mais curtos trata de, dado um vértice inicial em um grafo, encontrar os caminhos mais curtos entre esse vértice e qualquer outro do grafo.

Vale notar que esse problema difere da árvore BSD, que dá o caminho mais curto em número de arestas, por exigir o caminho mais curto em comprimento de arestas. Ou seja, dado um grafo dirigido G com pesos positivos nas arestas, queremos encontrar os caminhos com menor peso entre um determinado vértice e todos os outros de G. Nesse caso, o peso de um caminho é definido como a soma do peso de cada aresta que compõe esse caminho.

Apesar de termos enunciado o problema para grafos dirigidos, ele também pode ser usado para grafo não dirigidos, pois tais grafos podem ser vistos como grafos dirigidos tais que cada aresta não dirigida corresponde a 2 arestas dirigidas com o mesmo comprimento, porém com direções opostas.

Assim, vamos enunciar o problema mais formalmente:

"Dado um grafo dirigido $G = (V, E)$ e um vértice v, quero encontrar os caminhos mais curtos de v a todos os outros vétices de G".

Vamos analizar o problema, primeiramente, com grafos acíclicos. Seja $G = (V, E)$ um grafo dirigido acíclico.

Prova: *(por indução simples no número n de vértices do grafo)*

Base: $|v| = 1$. Trivial.

Hipótese de Indução: Dada uma ordenação topológica, sei como encontrar os comprimentos dos caminhos mais curtos de v aos $n-1$ primeiros vértices.

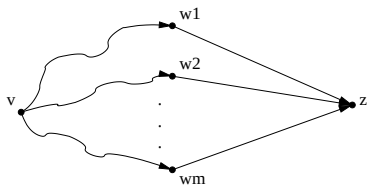
Passo: Seja G um grafo dirigido com n vértices em ordem topológica e um vértice v. Removamos o nº vértice, z, e, por hipótese de indução, calculo os comprimentos de v a todos os outros vértices. Então devolvo v e suas arestas a G. Sejam w_i, $1 \leq i \leq m$, os vértices de G tais que $(w_i, z) \in E$. Então, para achar o caminho mais curto de v a z tomo o menor dos caminhos $v \rightarrow w_i + (w_i, z)$.

Abaixo podemos ver o algoritmo:

Algoritmo CCAciclico(G,v,n):
Entrada: $G = (V, E)$, um grafo acíclico, um vértice v e o número de vértices n.
Saída: Para cada vértice $w \in V$, w.CC é o comprimento do caminho mais curto de v para w.
início
    seja z o vértice de rótulo n na ordenação topológica;
    se $z \neq v$ então
        CCAciclico($G \setminus \{z\}$,v,$n-1$);
        para todo w tal que $(w, z) \in E$ faça
            se w.CC +$comp(w, z) < z$.CC então
                z.CC := w.CC +$comp(w, z)$;
            senão v.CC := 0
fim

Figura 13: Algoritmo para menores caminhos em grafo acíclico.

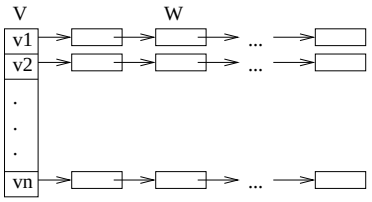
Esse algoritmo, então, funciona da seguinte maneira: antes de mais nada, ordeno G topologicamente. Se o rótulo de v (o vértice a partir do qual busco os caminhos mais curtos) for k, então os vértices com rótulo $< k$ não precisam ser considerados, pois não há aresta de v a esses vértices. Pegamos, então, um vértice com rótulo n, z, sorvedouro de G. Pela indução, conheço os caminhos mínimos a partir de v em $G \setminus \{z\}$:



Então, para achar o caminho mínimo de v a z, basta tomar o menor valor dos caminhos mínimos de v a $w_i + (w_i, z)$. Como z é o último vértice na ordenação topológica, nenhum outro vértice é alcançável a partir de z, garantindo que nenhum outro caminho é afetado pela inclusão de (w_i, z).

Uma observação interessante diz respeito à estrutura de dados usada. Em vez de usarmos a tradicional lista de adjacências para representar o grafo, é melhor usarmos uma lista transposta.

Na lista de adjacências coloco no vetor V os vértices v_i de G e nas listas W os vértices w_j tais que $(v_i, w_j) \in G$.



Na lista de adjacências transposta, coloco no vetor V os vértices v_i de G e nas listas W os vértices w_j tais que $(w_j, v_i) \in G$ (note que na lista de adjacências, em W íam (v_i, w_i)).

Ou seja, enquanto a lista de adjacências registra, para cada vértice no vetor V, os vértices ligados a v_i por uma aresta que sai de v_i, a lista transposta registra os vértices ligados a v_i por arestas que chegam a v_i, o que é útil para o algoritmo acima.

Dessa forma, o algoritmo indutivo pode ser representado pela relação de recorrência $T(n) = T(n-1) + g_{in}(z_n)$, que é $O(|E|)$. O algoritmo leva $O(|E|)$ para transformar de lista de adjacências para lista transposta e $O(|V|)$ para retirar os vértices. Então o tempo total de execução do algoritmo é $O(|V| + |E| + |E|) = O(|V| + |E|)$.

Mas agora surge uma questão: como evito a ordenação topológica feita no início? Ou melhor, como fazer com que a ordenação seja encontrada ao mesmo tempo que os caminhos mais curtos?

Para tal, achamos um vértice $v_1 \in G$ tal que v_1 seria o primeiro na ordenação topológica. O segundo vértice será um que receba aresta somente de v_1, e assim por diante. Naturalmente, a partir do terceiro vértice teremos que considerar os caminhos já descobertos até os vértices que levam a esse último por meio de uma aresta, como no algoritmo dado anteriormente. Ou seja, no nº vértice já achamos todos os caminhos de v_1 a w_i, tal que $(w_i, v_n) \in G$ e tomamos o menor dos caminhos de v_1 a $w_i + (w_i, v_n)$. Esse algoritmo é mostrado na figura 14.

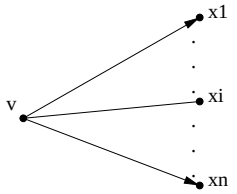
Tendo resolvido o problema dos caminhos mais curtos para grafos acíclicos, vamos a um caso geral, o de grafos que podem conter ciclos.

Nesse caso, não podemos usar a ordenação topológica, então vamos considerar os vértices na ordem imposta pelo tamanho dos seus caminhos mais curtos ao vértice inicial v.

Suponha o grafo abaixo, dirigido, podendo ter ciclos, com pesos positivos:

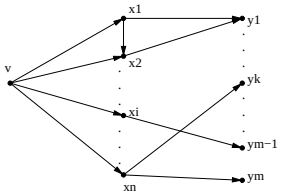
Algoritmo CCAciclico2(G,v):
Entrada: $G = (V, E)$, um grafo acíclico, um vértice v
Saída: Para cada vértice $w \in V$, w.CC é o comprimento do caminho mais curto de v para w.
início
    para todo vértice w faça w.CC := ∞;
    inicializa $v.g_{in}$ para todos os vértices
    para $i := 1$ até n faça
        se $v_i.g_{in} = 0$ então coloque v_i na fila;
    v.CC := 0;
    repita
        remova w da fila;
        para toda aresta (w, z) faça
            se w.CC +$comp(w, z) < z$.CC então
                z.CC := w.CC +$comp(w, z)$;
            $z.g_{in} := z.g_{in} - 1$;
            se $z.g_{in} = 0$ então coloque z na fila;
    até que a fila esteja vazia
fim

Figura 14: Algoritmo melhorado para menores caminhos em grafo acíclico.



Primeiro verificamos as arestas que saem de v e pegamos a de menor peso dentre elas, (v, x_i). Como todos os pesos são positivos, este é o caminho mais curto de v a x_i.

Agora escolhemos o segundo vértice mais próximo a v. Para tal temos que considerar as outras arestas, (v, x_j), $1 \leq j \leq n$, $j \neq i$ ou caminhos vindos de v contendo 2 arestas (mostrados abaixo): uma aresta (v, x_j) e outra (x_j, x_k) ou (x_j, y_l), $1 \leq j \leq n$, $1 \leq l \leq m$, escolhendo o mínimo do *comprimento*(v, x_j), $x_j \neq x_i$ ou *comprimento*(v, x_i) + *comprimento*(x_i, y_l), $y_l \neq v$ ou *comprimento*(v, x_i) + *comprimento*(x_i, x_j), $x_i \neq x_j$, $x_j \neq v$.



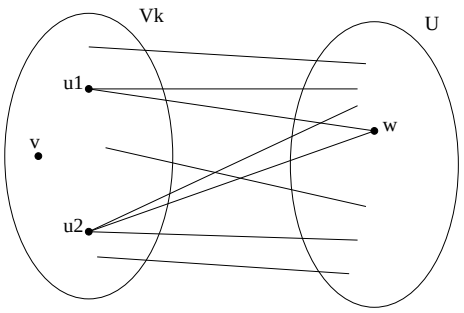
Então, podemos definir o algoritmo através da seguinte indução:

Prova: *(por indução simples no número de vértices cujo caminho mais curto já foi calculado)*

Hipótese de Indução: Dado um grafo G e um vértice v, conhecemos os k vértices que estão mais próximos de v e os comprimentos dos caminhos mais curtos a eles.

Base: $k = 1$. Sei como achar o vértice mais próximo de v, basta pegar a menor aresta que sai de v.

Passo: Seja V_k o conjunto contendo v e seus k vértices mais próximos. Podemos dividir G da seguinte maneira:



Queremos encontrar $w \in U = G \setminus V_k$ que seja o mais próximo a v dentre os vértices que não estão em V_n.

Por hipótese de indução tenho os caminhos mais curtos de v a todos os vértices em V_k. Então o próximo vértice a ser adicionado a V_k será o vértice $w \in U$ que recebe aresta de algum $u \in V_k$ tal que o caminho de v a $u+(u,w)$ seja mínimo. Ou seja, w será tal que $\min_{u \in V_k} (caminhoMC(u)^7 + (u,w))$ seja minimal sobre todos os $w \in U$. w é, então, o $(k+1)^{\underline{o}}$ vértice mais próximo a v, estendendo a hipótese de indução. Passo w para V_k e repito a operação.

7.1 Algoritmo de Dijkstra

O algoritmo acima pode ser melhorado, como veremos agora. Da mesma forma que o algoritmo anterior, o algoritmo de Dijkstra mantém um conjunto V_k de vértices cujos caminhos mais curtos a partir de v foram determinados. O algoritmo repetidamente seleciona vértices $w \in U$ com o menor caminho mais curto estimado, insere w em V_k e atualiza V_k.

Ou seja, como incluí w em V_k, posso ter mudado alguns caminhos mais curtos em V_k e U[8]. Naturalmente, não preciso atualizar todo G, apenas os vértices $u_i \in G$ que possuem um caminho de v a u_i passando por w. Então precisamos checar todas as arestas que saem de w, verificar o tamanho do $caminhoMC(v,w) + (w,z)$, $z \in G$, e atualizar $caminhoMC(v,z)$ se necessário. Assim, cada iteração

[7] $caminhoMC(u)$ é o caminho mais curto de v a u.

[8] Os vértices de U inicialmente possuem $caminhoMC = \infty$.

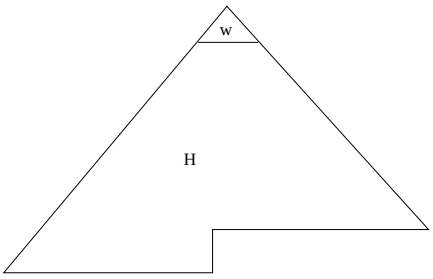
requer a descoberta de um vértice com um valor de *caminhoMC* mínimo e a atualização dos *caminhoMC*s de alguns dos vértices restantes (note que os vértices atualizados podem pertencer tanto a V_k quanto a U). Para maiores detalhes sobre o algoritmo de Dijkstra consulte [6].

Implementação: vamos, agora, implementar o algoritmo acima.

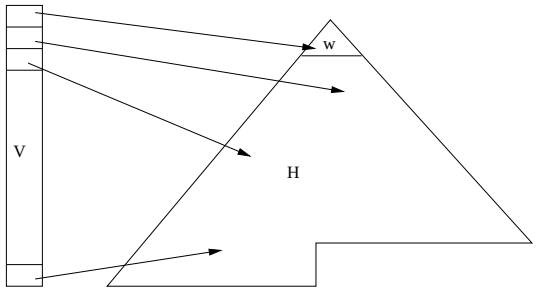
Algoritmo Dijkstra(G,v):
Entrada: $G = (U, E)$, um grafo acíclico, um vértice v
Saída: Para cada vértice $w \in U$, w.CC é o comprimento do caminho mais curto de v para w.
início
    para todo vértice w faça
        w.marca := falso;
        w.CC := ∞;
    v.CC := 0;
    enquanto existir um vértice não marcado faça
        seja w um vértice não marcado tal que w.CC é minimal;
        w.marca := verd;
        para toda aresta (w, z) tal que z não está marcado faça
            se w.CC + $comp(w, z) < z$.CC então
                z.CC := w.CC + $comp(w, z)$;
fim

Figura 15: Algoritmo de Dijkstra.

Nesse algoritmo, usamos um *heap* para armazenar os vértices de U, pois temos que encontrar o vértice cujo caminho a v tem o menor comprimento. A chave de cada vértice no *heap* é seu caminho mais curto, que, inicialmente, é ∞ para todos, menos para v, que é 0.



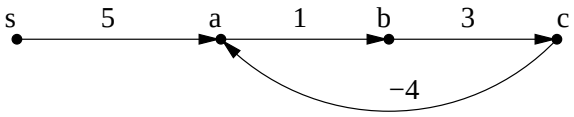
Retiro w e verifico todas as arestas (w, u), atualizando os caminhos em H, atualizando a ordem em H. Mas devo atualizar somente vértices cujo caminho a v passa por w. Então temos que conhecer a posição de cada vértice no *heap*. Para isso colocamos os vértices de G em um vetor com ponteiros para sua localização no *heap* e, assim, para achar um vértice no *heap* basta acessar o vetor.



Agora vamos verificar a complexidade do algoritmo. Temos $|E|$ atualizações no *heap*, levando a $O(|E|log|V|)$ comparações nele. Também temos $|V|$ remoções seguidas de $log|V|$ atualizações no *heap*, gastando no total $O(|V|log|V|)$. Assim, o tempo total de execução é $O((|V| + |E|)log|V|)$.

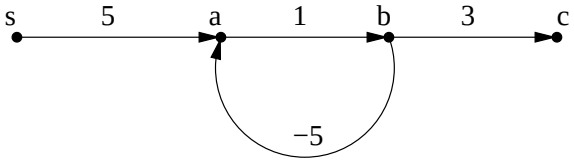
Vale notar que os grafos aqui analisados possuem pesos positivos em suas arestas. Mas, e se nosso grafo possuir pesos negativos? Nesse caso a coisa muda de figura, e calcular o menor caminho fica mais difícil.

Considere, por exemplo, o seguinte grafo:



Temos ∞ caminhos mais curtos de s a c: *sabc* (tamanho 8), *sabcabc* (tamanho 8), *sabcabcabc* (tamanho 8) etc.

Nesse caso não temos como dizer qual o caminho mais curto, apenas podemos dar um mais curto. A situação é ainda pior se houver um ciclo de comprimento negativo:



Nesse caso não temos como dar o caminho mais curto de s a c, pois teríamos que percorrer o ciclo $a \rightarrow b$ ∞ vezes, gerando um caminho de comprimento $-\infty$.

Em [6], você vai encontrar o algoritmo de Bellman-Ford (que não será descrito aqui), o qual trata de grafos com pesos negativos. O algoritmo verifica se existe algum ciclo de peso negativo alcançável a partir da fonte e, se existir tal ciclo, ele indica que não existe solução para o problema. Já se não existir o ciclo, o algoritmo produz os caminhos mais curtos e seus pesos.

Uma outra observação cuja demonstração será deixada como exercício é que o resultado final do algoritmo para caminhos mais curtos é uma árvore contendo

esses caminhos e com v como raiz. Essa árvore recebe o nome de árvore de caminhos mais curtos.

8 AGCM – Árvore Geradora de Custo Mínimo (*Minimum Spanning Tree*)

Uma árvore geradora de um grafo G é uma árvore contendo todos os vértices de G e algumas de suas arestas (o suficiente para conectar esses vértices na árvore).

Desse modo, uma árvore geradora de custo mínimo de um grafo G é um subgrafo conexo de G, uma árvore, contendo todos os vértices de G, de modo que a soma dos pesos das arestas no subgrafo seja mínima (novamente, estaremos trabalhando com pesos positivos).

O problema de encontrar uma AGCM de G, então, pode ser descrito assim:

"Dado um grafo não dirigido conexo com pesos nas arestas $G = (V, E)$, quero encontrar uma árvore geradora de G, T, de custo total mínimo."

Para resolver esse problema, antes, vamos verificar o seguinte lema:

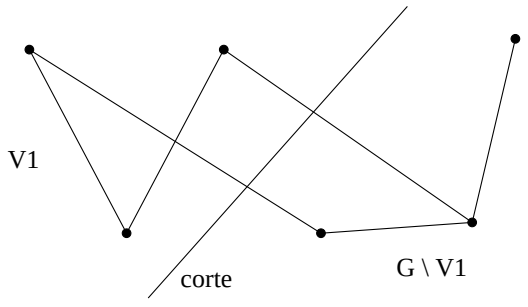
Lema 1 *Seja T uma AGCM de $G = (V, E)$ e seja $e = (u, v) \in E$ a aresta de menor custo em G. Então $e \in T$.*

Prova: *(Contradição)* Suponha que $e \notin T$. Seja $H = T \cup \{e\}$. Então H tem um ciclo C, pois e une 2 vértices, u e v, que já estão unidos por algum caminho em T. Seja $e' \in C \setminus \{e\}$. Como e é a aresta de menor custo em G, então $custo(e') > custo(e)$ e, nesse caso, $T' = H \setminus \{e'\}$ é uma árvore cujo custo é menor que o custo de T. Contradição, pois, nesse caso, T não é AGCM. Portanto, a aresta de menor custo de G deve estar em T.

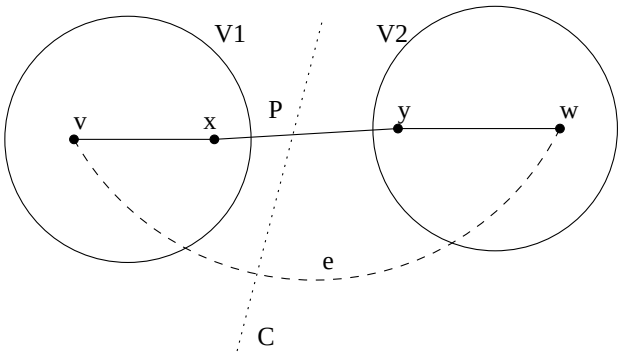
Além do lema acima, vamos verificar outro lema que será útil mais adiante:

Lema 2 *Sejam $G = (V, E, w)$, V_1, $V_2 \subseteq V$, $V_1 \cup V_2 = V$, $V_1 \cap V_2 = \varnothing$. Então a aresta do corte de V_1 e V_2 de menor custo pertence à AGCM de G.*

Prova: *(Contradição):* antes de mais nada, vamos definir um corte de G: um corte em um grafo G, $(V_1, G \setminus V_1)$ é uma partição de V (ver figura abaixo)



Agora sim, vamos à prova. Seja $e = (v, w)$ a aresta do corte de V_1 e V_2 de menor custo e seja T a AGCM de G. Suponha que $e \notin T$.



Seja $T' = T \setminus \{(x, y)\} \cup \{e\}$. Então T' é uma AG de G e, como e é a aresta de menor custo em C, então $custo(e) < custo((x, y))$ e, nesse caso, $custo(T') < custo(T)$, ou seja, T não é AGCM. Contradição.

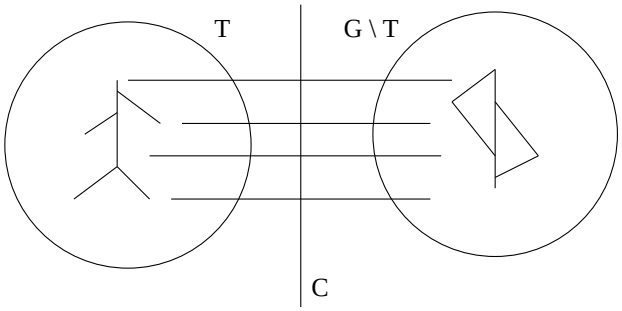
Agora sim podemos ver uma demonstração por indução de como construir uma AGCM de G:

Prova: *(por indução forte no número de arestas de T, subgrafo da AGCM de G)*

Hipótese de Indução: Dado $G = (V, E, w)$ sou capaz de encontrar um subgrafo $T \subseteq G$ com k arestas, $k < |V| - 1$, tal que T é uma árvore e T é um subgrafo da AGCM de G.

Base: $T = \{v\}$. Faço um corte em torno de v e pego a aresta mais leve no corte, (v, u). Com certeza essa é a menor aresta que conecta v a $G \setminus \{v\}$.

Passo: Seja $G = (V, E, w)$. Tome, por hipótese de indução, $T \subseteq G$ tal que T é árvore e subgrafo da AGCM de G. Façamos um corte C em G, $(G, G \setminus T)$:



Então, pelo lema 2, a aresta cruzando C com menor custo pertence à AGCM de G. Seja e essa aresta. Estendamos T, fazendo $T' = T \cup \{e\}$. T' é uma árvore, pois une um vértice $\in T$ a um $\in G \setminus T$ por um único caminho e, pelo lema 2, T'

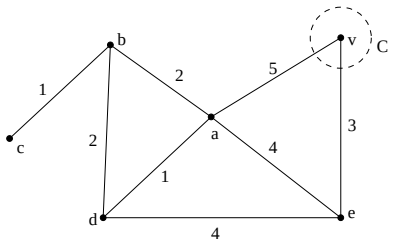
é subgrafo da AGCM de G. Dessa forma podemos estender T até que ela cubra todos os vértices de G, sendo uma AGCM de G.

Esse algoritmo é conhecido como algoritmo de Prim e é mostrado abaixo:

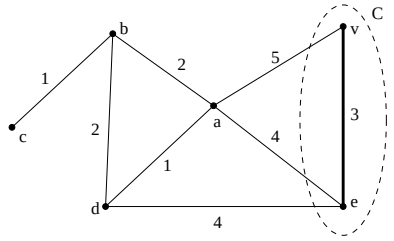
Algoritmo AGCM-Prim(G,r,w):
Entrada: $G = (V, E)$, um grafo acíclico, um vértice r e os pesos w.
Saída: A árvore geradora de custo mínimo de G
início
    $Q := V$;
    para ccada $u \in Q$ faça $chave[u] := \infty$;
    $chave[r] := 0$;
    $pai[r] :=$ NIL;
    enquanto $Q \neq \varnothing$ faça
        $u :=$ valor mínimo de Q;
        para cada $v \in Adj[u]$ faça
            se $v \in Q$ e $w(u, v) < chave[v]$ então
                $pai[v] := u$;
                $chave[v] := w(u, v)$
fim

Figura 16: Algoritmo de Prim para AGCM.

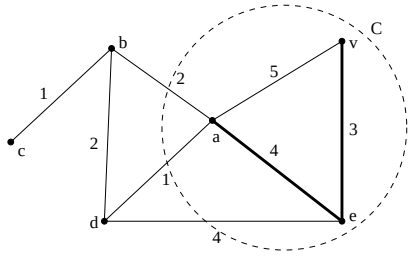
A idéia básica do algoritmo é a seguinte: tenho um grafo não dirigido com pesos nas arestas, G. Escolho um vértice qualquer de G, v, e faço um corte em G, $(\{v\}, G \setminus \{v\})$ com respeito a v. Inicialmente faço $T = \{v\}$.



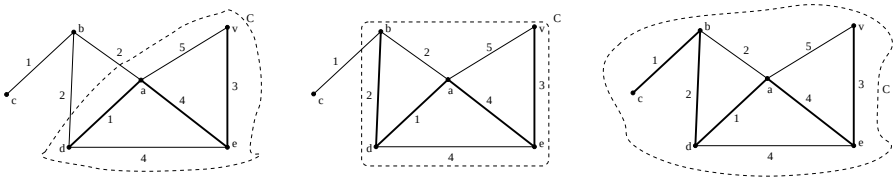
Pego, então, a aresta de menor peso desse corte, e, e a incluo em T ($T = T \cup \{e\}$). Faço um novo corte em G, $C = (T, G \setminus T)$.



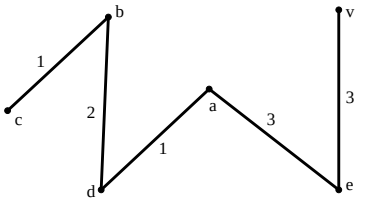
Novamente tomo a aresta, e, de menor peso nesse corte e faço $T = T \cup \{e\}$, fazendo um novo corte $C = (T, G \setminus T)$ em G.



Assim prossigo até cobrir todos os vértices de G:



Gerando a árvore



Que é uma AGCM de G.

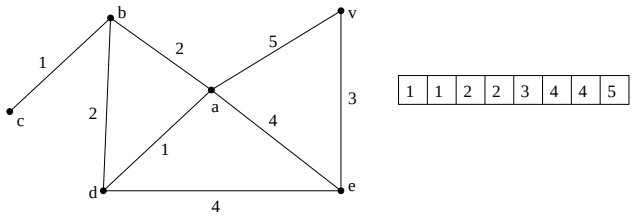
Vamos analisar, agora, a complexidade do algoritmo de Prim. Se o *queue*, Q, for implementado como um *heap* binário, sua inicialização gasta $O(|V|)$. O laço é executado $|V|$ vezes e , como cada operação de extração do valor mínimo no *heap* leva $O(log|V|)$, o tempo total é $O(|V|log|V|)$. O laço for é executado $O(|E|)$ vezes e a atribuição na última linha requer atualização no *heap* em tempo $O(log|V|)$. Então o tempo total do algoritmo é $O(|V|log|V| + |E|log|V|) = O(|E|log|V|)$.

Uma alternativa ao algoritmo de Prim é o algoritmo de Kruskal, apresentado na figura 17.

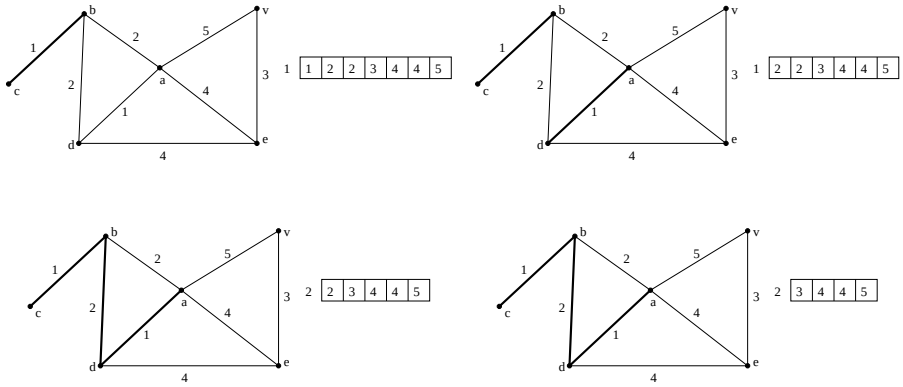
O algoritmo funciona da seguinte maneira: primeiro ordeno as arestas de G. Considero, então, cada vértice de G como pertencendo a uma árvore em G, ou seja, $v_i \in T_i$, $1 \leq i \leq |V|$.

Algoritmo AGCM-Kruskal(G,w):
Entrada: $G = (V, E)$, um grafo acíclico e os pesos w.
Saída: A árvore geradora de custo mínimo de G
início
    $A := \varnothing$;
    para ccada $v \in V$ faça $Construa\_Conjunto(v)$;
    ordene as arestas de E por ordem crescente do peso w;
    para cada aresta $(u, v) \in E$, em ordem crescente, faça
        se $Conjunto(u) \neq Conjunto(v)$ então
            $A := A \cup \{(u, v)\}$;
            $Une\_arvores(u, v)$;
    retorne A;
fim

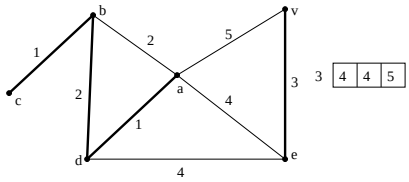
Figura 17: Algoritmo de Kruskal.



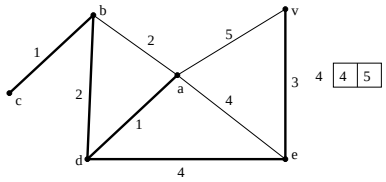
Pego a menor aresta de G e vejo se ela ume 2 vértices pertencentes a árvores diferentes. Se sim, unifico as 2 árvores e repito a operação. Se não, passo à próxima aresta na lista ordenada e repito a operação.



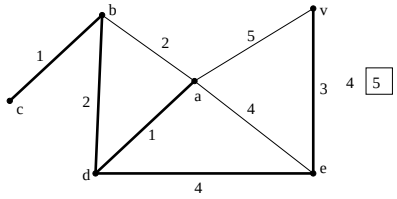
Note que, nesse último passo mostrado, a aresta (a, b) foi ignorada, por unir 2 vértices pertencentes à mesma árvore. O algoritmo cotinua assim até unir todas as sub-árvores em G, ou seja, todos os vértices.



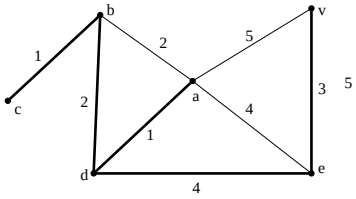
3 | 4 | 4 | 5



4 | 4 | 5

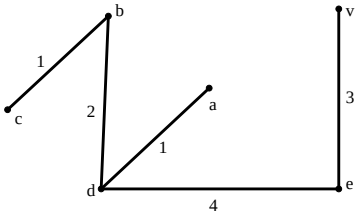


4 | 5



5

e a árvore é



Agora vamos analisar a complexidade do algoritmo de Kruskal. A demonstração de sua funcionalidade pode ser vista em [6].

O algoritmo gasta tempo $|V|$ para criar as $|V|$ árvores iniciais. Então ordena as arestas, gastando $O(|E|log|E|)$ para tal. Cada aresta é verificada uma vez por união das sub-árvores, gastando $O(|E|)$. Então o tempo do algoritmo é $O(|E|log|E|)$.

Uma observação importante sobre esses dois algoritmos é sobre sua estratégia de aquisição de novas arestas. Os algoritmos usam uma estratégia gulosa que, a cada passo toma o elemento que é localmente ótimo, mostrando que isso leva à resposta ótima. Alguns autores, no entanto, não consideram esses algoritmos puramente gulosos. Para maiores informações sobre o método guloso consulte [5]. A estratégia gulosa também pode ser usada como heurística para muitos problemas, sendo muito usada em aplicações de inteligência artificial. Para mais informações sobre a heurística gulosa consulte [5] e [11].

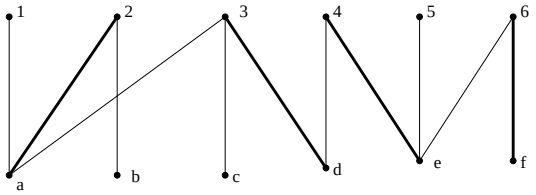
9 Emparelhamento

Um emparelhamento em um grafo G é um conjunto de arestas e vértices tais que nenhum par de arestas tem vértices em comum. Assim um emparelhamento perfeito é um emparelhamento de todos os vértices de G.

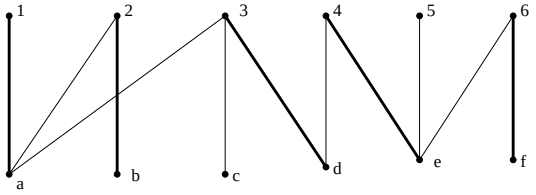
9.1 Emparelhamento em Grafos Bipartidos

Nesse caso, o problema é encontrar o maior emparelhamento possível num grafo bipartido, que é um grafo $G = (V, E)$ tal que $V = V_1 \cup V_2$ e $\forall e = (u, v) \in E \Rightarrow u \in V_1$ e $v \in V_2$ ou vice-versa.

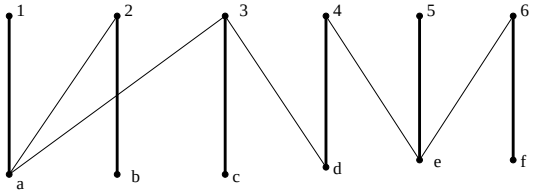
Para resolvê-lo, suponha que já temos um emparelhamento maximal em G, ou seja, um emparelhamento que não pode ser estendido pela adição de uma aresta, e vamos tentar melhorá-lo. Considere a figura abaixo e seu emparelhamento:



Podemos melhorar o emparelhamento trocando a aresta $(2, a)$ por $(1, a)$ e $(2, b)$:



e $(3, d)$ e $(4, e)$ por $(3, c)$, $(4, d)$ e $(5, e)$:



Note que cada mudança começou com um vértice não emparelhado (1, no primeiro exemplo), passou por um emparelhamento – $(a, 2)$ – e terminou num vértice não emparelhado (b). Então trocamos o caminho $1a2b$ por $(1, a)$ e $(2, b)$.

A idéia, então, é tomar um vértice não emparelhado v e escolher outro vértice adjacente a v, u, que pertença a um emparelhamento (com w, por exemplo). Então emparelhamos v com u e quebramos o emparelhamento de u com w. Agora temos que achar um emparelhamento para w. Se w estiver conectado a um vértice não emparelhado, achamos. Senão, podemos repetir o procedimento acima.

Esse foi o procedimento adotado nas figuras acima, onde o vértice c foi emparelhado a 3, quebrando $(3, d)$. O vértice d foi emparelhado a 4, quebrando $(4, e)$ e, por fim, o vértice e foi emparelhado ao 5, que não estava emparelhado com ninguém.

Vamos, agora, tentar formalizar essa discussão. Um caminho P no grafo que possua uma aresta que não pertence a um emparelhamento M, uma que $\in M$, e assim por diante até a última aresta de P, que $\notin M$, é chamado de caminho alternante, por possuir arestas que se alternam entre pertencer ou não a M. O número de arestas deve ser ímpar, pois o caminho começa e termina com arestas que $\notin M$, havendo uma aresta a mais que $\notin M$

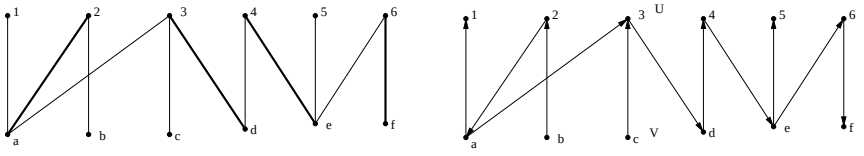
Mas veja que o que fizemos no exemplo acima foi encontrar caminhos alternantes e trocar, neles, as arestas que $\in M$ pelas que $\notin M$. No caminho $1a2b$ trocamos $a2$ por $1a$ e $2b$ e no caminho $c3d4e5$ trocamos $3d$ e $4e$ por $c3$, $d4$ e $e5$.

Assim, vamos ao seguinte Teorema do Caminho Alternante (cuja demonstração pode ser vista em [10]):

Teorema 1 *Um emparelhamento M num grafo bipartido G é máximo se e somente se não admitir caminho alternante.*

Com base nesse teorema, então, podemos construir um algoritmo para achar um emparelhamento máximo em um grafo bipartido. A idéia básica é, dado o grafo, encontrar um emparelhamento maximal e, então, buscar caminhos alternantes, modificando o emparelhamento até que não possamos mais encontrar tais caminhos e, nesse caso, o emparelhamento será máximo.

Para tal, transformo o grafo não dirigido em grafo dirigido, colocando as arestas $\in M$ em uma direção e as $\notin M$ em direção oposta:



Um caminho alternante é formado, assim, por um caminho dirigido que parte de um vértice não emparelhado em V, chegando a um vértice não emparelhado em U, o que pode ser descoberto com uma simples busca em profundidade começando em um vértice não emparelhado.

O tempo gasto na DFS é $O(|V| + |E|)$, e pode ser repetido até $\binom{|V|}{2}$ vezes (pois o caminho alternante estende o emparelhamento uma aresta por vez e há no máximo $\binom{|V|}{2}$ arestas em cada emparelhamento). Então a complexidade do algoritmo é $O(\binom{|V|}{2} \cdot (|V| + |E|)) = O(|V| \cdot (|V| + |E|))$.

Esse tempo pode ser melhorado se, enquanto procuramos um caminho alternante, percorrermos o grafo todo achando vários caminhos alternantes vértice disjuntos. Como esses caminhos modificam vértices diferentes, eles podem ser analisados paralelamente. Esse algoritmo tem um tempo de execução $O((|V| + |E|)\sqrt{|V|})$ no pior caso e é descrito com mais detalhes em [10].

9.2 Emparelhamento Perfeito em Grafos Muito Densos

Veremos aqui uma interessante aplicação de demonstração por indução reversa para um domínio finito.

Em [10] há uma demonstração alternativa do resultado que apresentamos abaixo, a qual é também por indução, mas que, além de ser em outro parâmetro, consiste de indução direta.

Vamos ao problema:

"Seja $G = (V, E)$ um grafo com $|V| = 2n$. Se todos os seus vértices têm grau pelo menos n, então G tem um emparelhamento perfeito."

Perceba que dentre todos os grafos que satisfazem as condições do enunciado do problema, o mais denso em arestas é o grafo completo. Este é, portanto, um bom caso para base da indução reversa.

Prova: *(por indução reversa no número m de arestas de G)*

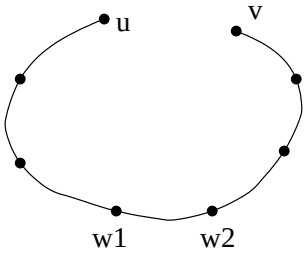
Base: Um grafo completo de $2n$ vértices certamente tem emparelhamentos perfeitos. Basta dividir o conjunto de vértices em pares arbitrários.

Hipótese: Consideremos que dado um grafo $G = (V, E)$ com $|V| = 2n$, $g(v) \geq n \ \forall v \in V$, e $|E| = m \leq \binom{n}{2}$ é possível determinar um emparelhamento perfeito em G.

Passo: Seja $G = (V, E)$ um grafo com $|V| = 2n$, $g(v) \geq n \ \forall v \in V$, e $|E| = m - 1 < \binom{n}{2}$.

Como G não é um grafo completo, sejam u e v dois vértices de V tais que $(u, v) \notin E$.

Tome o novo grafo $G' = (V, E')$ onde $E' = E \cup \{(u, v)\}$. G' certamente satisfaz as hipóteses do enunciado e tem uma aresta a mais que G. Portanto, podemos aplicar a hipótese de indução a G'.



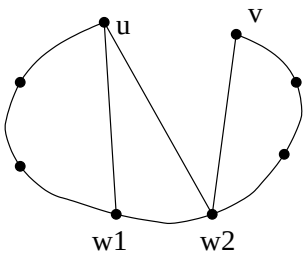
Seja, então, P um emparelhamento perfeito em G' dado pela hipótese de indução. Se P não inclui a aresta (u, v), a demonstração está completa.

Caso contrário, i.é., $(u, v) \in P$, basta mostrarmos que é possível encontrar um caminho alternante em G com relação ao emparelhamento parcial $P \setminus \{(u, v)\}$.

Se aos dois vértices de cada uma das arestas de $P \setminus \{(u, v)\}$ chegassem apenas duas arestas partindo dos vértices u, v, teríamos que $g(u) + g(v) = 2n - 2$ pois $P \setminus \{(u, v)\}$ tem apenas $n - 1$ arestas. Mas, em G, $g(u) + g(v) \geq 2n$.

Portanto, necessariamente existe alguma aresta, digamos (w_1, w_2), em $P \setminus \{(u, v)\}$ a cujos vértices chegam (pelo menos) três arestas partindo de u

e v. Suponha, sem perda de generalidade que $(u, w_1) \in E$, $(u, w_2) \in E$ e $(v, w_2) \in E$. Então, u, w_1, w_2, v é um caminho alternante em G com relação a $P \setminus \{(u, v)\}$. CQD



Exercício: Encontre o menor contra-exemplo para a afirmação:

"Seja $G = (V, E)$ um grafo com $|V| = 2n > 2$. Se todos os seus vértices têm grau $n - 1$, então G tem um emparelhamento perfeito."

Desafio: Prove ou apresente um contra-exemplo para a seguinte afirmação:

"Seja $G = (V, E)$ um grafo com $|V| = 2n$. Se dois dos vértices de G têm grau n e todos os seus outros vértices têm grau $n-1$, então G tem um emparelhamento perfeito."

10 Circuito Hamiltoniano

Um circuito hamiltoniano é um circuito que passa por todos os vértices de um grafo exatamente uma vez. Um grafo que apresente tal circuito é chamado de hamiltoniano.

De fato, esse nome foi dado a tais grafos após Hamilton ter descrito em uma carta a um amigo um jogo matemático no dodecaedro (figura 18), no qual uma pessoa coloca 5 marcas em 5 vértices quaisquer consecutivos e a outra deve completar o caminho até então criado formando um ciclo gerador, ou seja, um que contenha todos os vértices do dodecaedro.

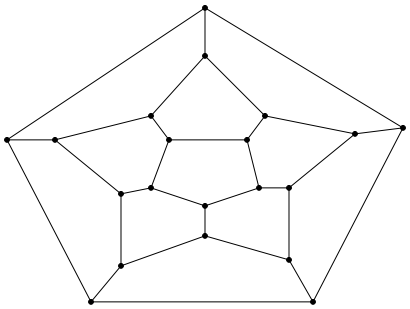


Figura 18: O dodecaedro

Como o problema de achar ciclos hamiltonianos é NP-completo, aqui será apresentado um método para achar um ciclo hamiltoniano somente em grafos muito densos.

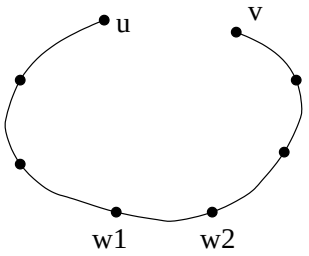
Teorema 1 *Seja $G = (V, E)$ um grafo não dirigido, conexo, com $g(u) + g(v) \geq n = |V|$, $\forall u, v \in V$ com $(u, v) \notin E$. Então G é um grafo hamiltoniano.*

Prova: *(por indução reversa em $|E|$).* Uma prova alternativa pode ser encontrada em [4].

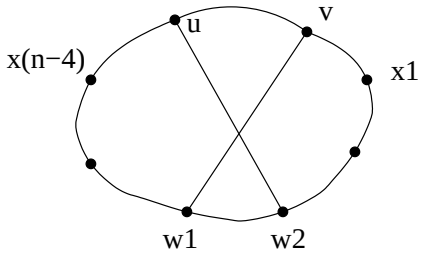
Base: G completo. Se $|V| \geq 3$, então tenho o ciclo para qualquer grafo completo, basta colocar os vértices em ordem arbitrária e conectá-los.

Hipótese de Indução: O teorema é verdadeiro para todo grafo com $|E| = m \leq \binom{n}{2}$.

Passo: Seja $G = (V, E)$ um grafo não dirigido, conexo, com $|E| = m \leq \binom{n}{2}$ e $g(u) + g(v) \geq n$, $|V| = n$, $\forall u, v \in V$, com $(u, v) \notin E$. Como G não é completo, sejam $u, v \in V$ tais que $(u, v) \notin E$:



Tome $G' = (V, E')$ com $E' = E \cup \{(u, v)\}$. Por hipótese de indução, G' é hamiltoniano. Seja C' um circuito hamiltoniano de G'. Se $(u, v) \notin C'$, está provado. Senão agimos da seguinte maneira: considere as arestas de G que saem de u e v. Como $g(u) + g(v) \geq n$, há pelo menos n delas. Mas, como G contém $n - 2$ vértices além de u e v, há 2 vértices, w_1 e w_2, tais que $(w_1, w_2) \in C'$, que se conectam a u e v (w_1 com v e w_2 com u):



Podemos, então, achar um novo ciclo hamiltoniano que não passa por (u, v): $v, w_1, \ldots, x_{n-4}, u, w_2, \ldots, x_1, v$.

O algoritmo que vem dessa prova começa com o grafo completo e muda uma aresta por vez. Para cada aresta que retiro, busco nos vértices um circuito que os cubra. Então o algoritmo é $O(|E| \cdot |V|)$.

10.1 Problema do Caixeiro Viajante

Esse é um problema que ilustra o uso do que foi discutido: um caixeiro viajante quer visitar um certo número de cidades e então voltar ao ponto de partida. A questão é: como ele deve planejar seu itinerário de modo a visitar cada cidade apenas uma vez no menor tempo possível?

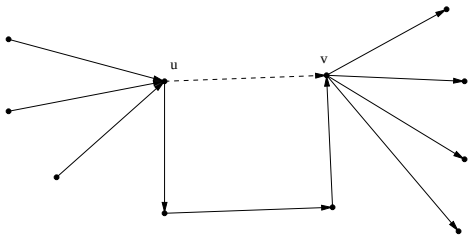
Basicamente o objetivo é encontrar um ciclo hamiltoniano de comprimento mínimo. A solução para um exemplo deste problema pode ser vista em [4]

11 Todos os Caminhos Mais Curtos

O problema de todos os caminhos mais curtos pode ser assim enunciado: dado um grafo $G = (V, E)$ com pesos positivos nas arestas, quero encontrar os caminhos de comprimento mínimo entre todos os pares de vértices de V.

Uma abordagem inicial pode ser a seguinte: para cada vértice $v \in V$ calculo a árvore de caminhos mínimos a partir de v. Como para calcular uma árvore gasto $O((|E|+|V|)log|V|)$, gasto $O(|V|(|V|+|E|)log|V|)$ nessa abordagem. É claro que se o grafo for não dirigido o trabalho é menor, pois nesse caso, $caminho(u, v) = caminho(v, u)$, mas ainda assim é $O(|V|(|E| + |V|)log|V|)$.

Vamos então tentar outra abordagem: construir o algoritmo por indução no número de arestas[9]. Suponha G e retire a aresta (u, v):

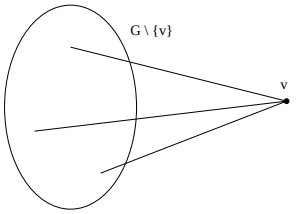


Calculo os caminhos mínimos e, então recoloco a aresta (u, v). No pior caso, terei que verificar, para cada par de vértices v_1 e v_2, se o caminho de v_1 a u + (u, v)+ caminho de v a v_2 é mais curto que o caminho mais curto já conhecido de v_1 a v_2. Ou seja, para cada aresta (u, v) que adiciono, se $comp(u, v) < caminhoMC(u, v)$ então devo atualizar $caminho(u, v)$, igualando-o a $comp(u, v)$ e rever o caminho entre cada par de vértices.

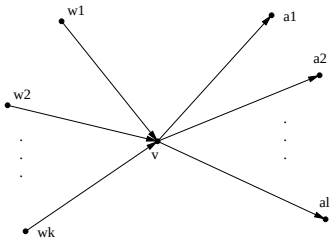
Como, para cada aresta, temos que fazer $O(|V|^2)$ comparações (entre todos os possíveis pares), no pior caso esse algoritmo é $O(|E|\cdot|V|^2)$ e, como o número de arestas pode ser tão grande quanto $O(|V|^2)$, o algoritmo pode chegar a $O(|V|^4)$.

Tentemos, agora, indução no número de vértices (novamente mostrarei informalmente somente o passo na demonstração). Suponha um grafo G e retire um vértice v:

[9]Estou mostrando informalmente o passo na demonstração por indução.



Pela hipótese de indução, sei os caminhos mais curtos entre qualquer par de vértices em $G \setminus \{v\}$. Recoloco então v:

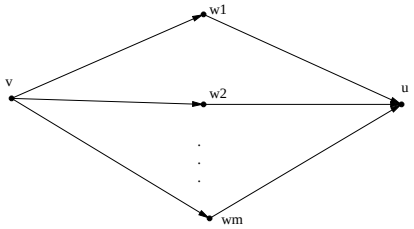


Sei achar o caminho mínimo de v aos outros vértices de G (árvore de caminhos mais curtos). Para achar o caminho mais curto dos outros vértices a v basta inverter o sentido de todas as arestas de G e achar os caminhos mais curtos de v a todos os outros vértices. Seja G' o grafo G com os sentidos das arestas invertido. Então um caminho mínimo em G' de u a v é um caminho mínimo de v a u em G, e um caminho mínimo de u a todo $v_i \in G'$ em G' é um caminho mínimo de todo $v_i \in G$ a u em G.

Mas isso não é tudo. Ainda temos que verificar, para cada par de vértices, se existe um caminho mais curto entre eles que passa por u, ou seja, para cada par v_1 e v_2, comparamos o caminho mais curto em $G \setminus \{v\}$ de v_1 a v_2 com o caminho mais curto entre v_1 e v + o caminho mais curto entre v e v_2.

Os caminhos mais curtos de v a todos os vértices e de todos os vértices a v gasta $O(2|E|log|V|) = O(|E|log|V|)$. A última verificação entre todos os pares de vértices gasta $O(|V|^2)$. Essas 2 operações são feitas para cada vértice, fazendo com que o tempo total do algoritmo seja $O(|V|(|E|log|V| + |V|^2))$, o que é melhor que o $O(|V|^4)$ da indução anterior.

Uma melhoria para essa solução pode ser obtida ao observarmos que, a cada vértice u adicionado, já calculei os caminhos de v a w_i:



Então não preciso recalculá-los a cada vez, basta olhar a adjacência de u. Isso faz com que o tempo $O(|E|log|V|)$ acima se transforme em $O(|V| \cdot g(u)) = O(|V|^2)$ e, então, o tempo total do algoritmo é $O(|V|(|V|^2 + |V|^2)) = O(|V|^3)$.

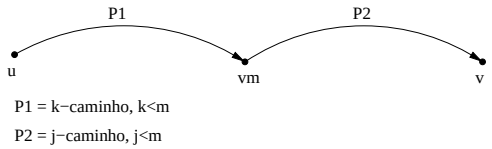
Mas vamos olhar um outro modo de resolver o problema: restringindo o tipo de caminho permitido em G. Definamos, para tal, um k-caminho de u a v como sendo um caminho de u a v que só passa por vértices de rótulo $\leq k$ (não levando em conta u e v). Note que um 0-caminho é uma aresta. Dessa forma, devo rotular os vértices de G arbitrariamente com números de 1 a $|V|$.

Vamos então à seguinte indução:

Hipótese de Indução: Seja o grafo $G = (V, E)$. Consigo determinar os k-caminhos mais curtos entre cada par de vértices de V, para $k < m$.

Base: $m = 1$. Então só posso considerar arestas diretas e a solução é trivial.

Passo: Seja $G = (V, E)$ um grafo no qual determinei os k-caminhos mais curtos, $k \leq m - 1$. Seja o vértice v_m, com rórulo m. Qualquer m-camiho mais curto deve incluir v_m uma única vez. Então qualquer m-caminho mais curto entre u e v é o k-caminhos mais curto ($k < m$) entre u e v_m (v_m pode pertencer aos extremos do k-caminho, não ao caminho em si) + o j-caminho mais curto ($j < m$) entre v_m e v.



Como, por hipótese de indução, conheço os k-caminhos de G para $k < m$, ao somar os dois caminhos acima tenho os k-caminhos para $k = m$.

O algoritmo dado por essa demonstração é mostrado abaixo e é conhecido como algoritmo de Floyd-Warshall (para maiores informações consulte [6]):

Algoritmo F-W(W):
Entrada: W. matriz de adjacências $n \times n$ representando o grafo.
$\{W[x, y]$ é o peso de (x, y), se existir, senão é ∞. $W[x, x] = 0, \forall x.\}$
Saída: W terá os caminhos maiscurtos
início
    para $m := 1$ até n faça
        para $x := 1$ até n faça
            para $y := 1$ até n faça
                se $W[x, m] + W[m, y] < W[x, y]$ então
                    $W[x, y] := W[x, m] + W[m, y]$;
fim

Figura 19: Algoritmo de Floyd-Warshall.

Nesse algoritmo, para cada m-caminho, $1 \leq m \leq n$, $n = |V|$, verifico, para cada par de vértices x e y, se $caminho(x, m) + caminho(m, y) < caminho(x, y)$

e, se for, atualizo $caminho(x, y)$. Assim, na verdade, estou verificando, para cada m, se $v_m \in$ ao caminho mais curto entre x e y e, se pertencer, atualizo esse caminho. Basicamente, o algoritmo caminha em cima da seguinte relação de recorrência:

$$C_{ij}^{(k)} = \begin{cases} (i,j) & \text{se } k = 0 \\ min(C_{ij}^{(k-1)}, C_{ik}^{(k-1)} + C_{kj}^{(k-1)}) & \text{se } k \geq 1 \end{cases}$$

Ou seja, como dito antes, para todo k, $1 \leq k \leq n$, o algoritmo atribui a $C_{ij}^{(k)}$ (o $caminho(x, y)$) o menor valor dentre $caminho(x, y)$ e $caminho(x, k) + caminho(k, y)$, calculados para $k-1$, comparando, assim, o caminho mais curto envolvendo v_k ($C_{ik}^{(k-1)} + C_{kj}^{(k-1)}$) com o caminho mais curto já calculado sem v_k ($C_{ij}^{(k-1)}$).

Vamos agora analisar a complexidade desse algoritmo. Para cada vértice v_k, $1 \leq k \leq |V|$, verifico o caminho entre todos os outros pares de vértices, passando por v_k. Então o tempo de execução do algoritmo é $O(|V|^2 \cdot |V|) = O(|V|^3)$.

Vale notar que, devido à sua simplicidade, esse algoritmo é melhor que os anteriormente descritos para grafos densos, onde $|E|$ é da ordem de $|V|^2$ e, nesse caso, $O(|V|(|E|log|V|+|V|^2)) = O(|V|^3 log|V|)$. Apesar disso, para grafos pouco densos, o uso dos algoritmos anteriores pode ser melhor.

12 Fluxo em Redes

Antes de mais nada vamos fazer algumas definições: seja $G = (V, E)$ um grafo dirigido conexo orientado com dois vértices distintos, s (a fonte), com $g_{in}(s) = 0$ e t (o sorvedouro), com $g_{out}(t) = 0$. A cada aresta de E associamos um valor positivo, $c(e)$, chamado de capacidade de e.

A capacidade da aresta é a quantidade de fluxo que pode passar por ela, e fluxo é uma função f nas arestas tal que:

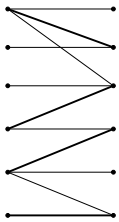
1. $0 \leq f(e) \leq c(e), \forall e \in E$. Ou seja, o fluxo em uma aresta não excede sua capacidade.

2. $\forall v \in V \setminus \{s, t\}, \sum_{u \in V} f(u, v) = \sum_{w \in V} f(v, w)$. Ou seja, o fluxo total que entra em v também sai dele (exceto para s e t).

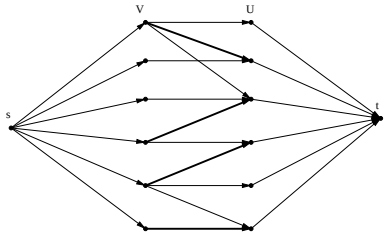
Essas 2 condições implicam que $\sum_{v \in V} f(s, v) = \sum_{v \in V} f(v, t)$. Ou seja, todo o fluxo que sai de s entra em t.

O problema então é maximizar esse fluxo. Inicialmente, vamos ver como o problema de emparelhamento em grafos bipartidos se reduz ao problema de fluxo em redes:

Seja $G = (V, E, U)$ um grafo bipartido no qual queremos achar um emparelhamento de cardinalidade máxima:



Adicionamos s e t a G, construindo $G' = G \cup \{s, t\}$, criando arestas de s a V e de U a t:



Direcionamos, então, as arestas de V para U e definimos o peso de todas as arestas (sua capacidade) como sendo 1.

Temos, assim, que um fluxo máximo em G' dá um emparelhamento máximo em G e vice-versa.

A primeira parte da afirmação é clara, pois se o fluxo é máximo e corresponde a um emparelhamento, então não pode haver emparelhamento maior, pois corresponderia a um fluxo maior.

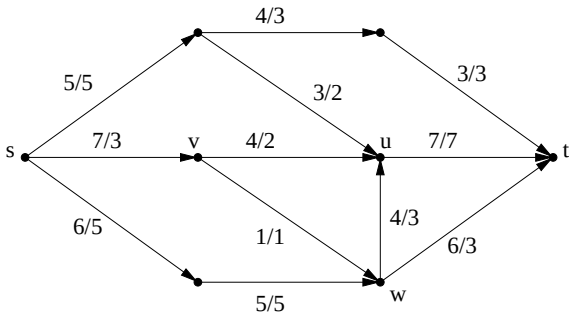
A segunda parte é mais difícil e requer o conceito de caminho aumentante. Um caminho aumentante, P, com respeito a um fluxo f é um caminho dirigido de s a t tal que, para toda aresta $(v, u) \in P$ exatamente uma das seguintes condições vale:

1. $(v, u) \in E$ e $f(v, u) < c(v, u)$. Ou seja, (v, u) está na mesma direção que em G e tem fluxo menor que sua capacidade. Esse tipo de aresta é chamado de *forward edge* (aresta progressiva).

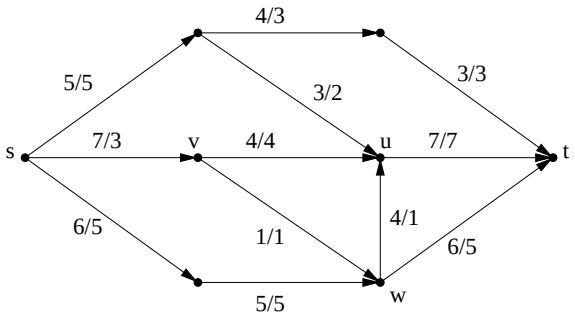
2. (v, u) é a reversa de $(u, v) \in E$ e $f(u, v) > 0$. Ou seja, (v, u) está na direção oposta que em G (que é (u, v)) e possui um fluxo. Esse tipo de aresta é chamado de *backward edge* (aresta de retorno).

Se f admitir um caminho aumentante (se houver um caminho aumentante com respeito a f), então f não é máximo (como a relação entre caminhos alternantes e emparelhamento) e sempre podemos aumentar o fluxo no caminho aumentante.

No caso do caminho conter somente arestas do tipo 1 acima, essa conclusão é direta. Se contiver arestas do tipo 2 é um pouco mais complicado. Considere o grafo abaixo (as arestas são marcadas com a/b, onde a é sua capacidade e b seu fluxo)



É fácil notar que não existem caminhos aumentantes contendo somente arestas do tipo 1, contudo, temos como aumentar o fluxo. Veja que o caminho $s \to v \to u \to w \to t$ possui arestas do tipo 1 e 2, logo, é um caminho aumentante. Um fluxo de valor 2 pode ser enviado de s a u (ocupando toda a aresta (v, u)). Dessa forma, o fluxo em u não está mais balanceado (o fluxo de entrada é o fluxo de saída +2). Então desviamos parte do fluxo que vem de w a u para t, fazendo $(w, u) = 4/1$ e $(w, t) = 6/5$.



Com isso mantemos todos os vértices balanceados e aumentamos em 2 o fluxo total em t.

Podemos pensar que o fluxo a mais que chegava em u foi desviado via $w \to u \to t$ pela "cotra-mão" em (w, u). Assim, vemos que se houver um caminho aumentante, então o fluxo não é máximo.

Agora considere o conjunto $A \subset V$ tal que $s \in A$ e $t \notin A$ e o conjunto $B = V \setminus A$. Um corte em G é um conjunto de arestas da forma $C = \{(v, w) \in E : v \in A, w \in B\}$. Então, o corte C é um conjunto de arestas que separa s de t. A capacidade de C é definida como a soma da capacidade de suas arestas, ou seja $c(C) = \sum_{e \in C} c(e)$.

Como nenhum fluxo pode exceder a capacidade de qualquer corte, temos o seguinte lema:

Lema 1 *Se C for um corte e f um fluxo, então $c(C) \geq f$.*

Ou seja, não há como o fluxo ser maior que a capacidade do corte, pois não há arestas por onde fluir. Agora, qual a conseqüência de encontrarmos um fluxo

f e um corte C tais que $f = c(C)$? Nesse caso, o fluxo será máximo e o corte será mínimo.

Vejamos o seguinte lema:

Lema 2 *Se f é um fluxo que não admite caminho aumentante, então existe um corte C tal que $f = c(C)$.*

Prova: seja f um fluxo que não admite um caminho aumentante de s a t. Seja $A = \{v \in V :$ existe caminho aumentante de s a $v\}$. Como f não admite caminho aumentante, então $s \in A$ e $t \notin A$ e, assim, A define um corte C. Temos então que:

1. Toda aresta (v, w) de C do tipo *forward edge* satisfaz $f(v, w) = c(v, w)$, pois $(v, w) \in C, v \in A$ e $w \in B = V \setminus A$. Como (v, w) está no corte, não pode admitir caminho aumentante nela, pois não há caminho aumentante de s a t.

2. Não existe *backward edge*, $e = (w, v)$ em C com $f(e) > 0$, pois ela poderia estender o caminho aumentante.

Então, o fluxo f é igual à capacidade do corte definido por A e é máximo. Ou seja $c(C) = \sum_{e \in C} c(e) = \sum_{e \in C} f(e) = f(C)$.

Da discussão acima temos o Teorema do Caminho Aumentante:

Teorema 1 *Um fluxo é máximo se e somente se não admite caminho aumentante.*

Prova: a primeira parte é direta, se o fluxo for máximo, não existe caminho aumentante pois, se existisse, o fluxo não seria máximo. A segunda parte, por sua vez, vem do lema 2, que afirma que se o fluxo não admite caminho aumentante, então existe corte C tal que $f(C) = c(C)$, ou seja, é máximo.

Vamos enunciar agora, sem provar, outro teorema, *Max Flow Min-Cut*:

Teorema 2 *O valor de um fluxo máximo é igual à menor capacidade do corte do grafo.*

Levando em conta os dois teoremas anteriores, chegamos a um terceiro, o Teorema do Fluxo Integral:

Teorema 3 *Se $c(e) \in Z^+, \forall e \in E$, então existe um fluxo máximo f tal que $f \in Z^+$.*

Prova: é direta do teorema do caminho aumentante. Qualquer algoritmo que use somente caminhos aumentantes terá fluxo inteiro se suas capacidades forem inteiras.

Vamos agora construir um algoritmo baseado nos caminhos aumentantes. Para tal vamos definir antes grafo residual: um grafo residual com respeito a $G = (V, E)$, $c : E \to R^+$ e um fluxo f é uma rede $R = (V, F)$, com s e t, $c' : F \to R^+$ (ou seja, com capacidades possivelmente diferentes das de G) tal que $(v, w) \in F$ se, ou

1. $(v, w) \in$ a um caminho aumentante, $c'(v, w) = c(v, w) - f(v, w) > 0$ como *forward edge*; ou

2. $(v, w) \in$ como *back edge* a um caminho aumentante $c'(v, w) = f(v, w)$.

Construir esse grafo residual requer $|E|$ passos, uma vez que cada aresta é verificada uma única vez. Então temos o seguinte algoritmo para a obtenção do fluxo máximo em G:

Algoritmo de Ford-Fulkerson
Entrada: $G = (V, E)$, um grafo não dirigido, $s, t \in V$, fonte e sorvedouro.
Saída: Grafo com fluxo máximo.

início
  Tome G
  Construa um fluxo, f, de s a t
  Enquanto houver caminho aumentante, P, aprimore f usando P.
fim

Figura 20: Algoritmo Ford-Fulkerson.

Ou, numa versão mais detalhada:

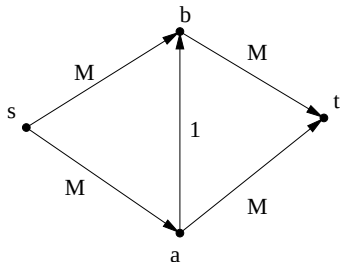
Algoritmo Ford-Fulkerson(G,s,t):
Entrada: $G = (V, E)$, um grafo não dirigido, $s, t \in V$, fonte e sorvedouro.
Saída: Grafo com fluxo máximo.

início
  para cada aresta $(u, v) \in E$ faça
    $f[u, v] \leftarrow 0$
    $f[v, u] \leftarrow 0$
  enquanto existir caminho P de s a t na rede residual G_f faça
    $c_f(P) \leftarrow min\{c_f(u, v) : (u, v$ está em $P\}$
    para cada aresta (u, v) em P faça
      $f[u, v] \leftarrow f[u, v] + c_f(P)$
      $f[v, u] \leftarrow -f[u, v]$
fim

Figura 21: Algoritmo Ford-Fulkerson mais detalhado.

Nesse algoritmo, nas linhas 1 a 3 o fluxo é inicializado com 0. Nas linhas 4 a 8 o algoritmo repetidamente encontra um caminho aumentante P em G_f e incrementa o fluxo f sobre P com a capacidade residual $c_f(P)$, que é definida como $c_f(u, v) = c(u, v) - f(u, v)$. Quando não existir mais caminho aumentante o fluxo é máximo.

Considere agora o grafo abaixo:



O fluxo máximo é obviamente 2M. Contudo, se iniciarmos com o caminho $s \rightarrow a \rightarrow b \rightarrow t(= 1)$, podemos tomar como caminho aumentante $s \rightarrow b \rightarrow a \rightarrow t$, o que aumenta o fluxo em somente 1. Esse processo pode ser repetido 2M vezes, com M podendo ser bem grande. Ou seja, o algoritmo é exponencial no tamanho da entrada, pois a entrada é $|V| + |E| + log|w|$ onde $log|w|$ é a representação binária dos pesos w, e o algoritmo é $O(2|E| \cdot |f|)$.

Nesse ponto vale uma dica de ouro: "Estude o assunto de aleatoriedade (*randomization*) para conhecer meios de evitar, ou minimizar as chances de encontrar, o terrível azar acima descrito".

Naturalmente esse não é o único algoritmo para o esse problema. Edmonds e Karp [6] sugerem que o próximo caminho aumentante seja obtido tomando-se o caminho aumentante com o menor número de arestas. Dessa forma, se fizermos o cálculo do caminho aumentante P na linha 4 do algoritmo acima usando BFS, faremos com que o caminho aumentante seja um caminho mais curto de s a t, em número de arestas, na rede residual, como pedem Edmonds e Karp. Usando essa mudança (dentre outras coisas) eles mostraram que o algoritmo pode rodar em $O(|V| \cdot |E|^2)$.

Além desses algoritmos, existem também alguns que conseguem resolver o problema de fluxo em tempo $O(|V|^3)$. Para maiores detalhes sobre os algoritmos aqui descritos e outros algoritmos sobre fluxo consulte [6] e [10]

Referências

[1] Algoritmos em Grafos. *http://www.cs.oberlin.edu/classes/dragn/labs/greedy/greedy0.html.*

[2] BFS. *http://www.cs.oberlin.edu/classes/labs/graphs/graphs5.html.*

[3] BFS(2). *http://www.cs.oberlin.edu/classes/dragn/labs/greedy/greedy1.html.*

[4] Bondy, J. (1976); Murty, U. *Graph Theory With Applications*, American Elsevier Publishing.

[5] Brassard, Gilles. (1955); Bratley, Paul. *Algorithmics: Theory and Practice*, Prentice-Hall.

[6] Cormen. Thomas (2000); Leiserson, Charles.; Rivest, Ronald. *Introduction to Algorithmics*, McGraw-Hill.

[7] Euler, Leonhard (1736). *Solutio Problematis ad Geometriam Situs Pertinentis*, em Commentarii Academiae Scientiarum Imperialis Petropolitanae, 8, 128 – 140.

[8] Grafos de Euler. *http://www.ctl.ua.edu/math103/euler/historic.htm.*

[9] Kuan, M. K. (1962). *Graphics Programming Using Odd or Even Points*, em Chinese Mathematics, 1, 273 – 277.

[10] Mamber, Udi (1989). *Algorithmics: A Creative Approach*, Addison-Wesley.

[11] Russel, Stuart. (1985); Norvig, Peter. *Artificial Intelligence: a Modern Approach*, Prentice Hall.