# Unidade II

## 5 ÁRVORE MÍNIMA

Árvore mínima (ou árvore geradora mínima) é um conceito em teoria dos grafos que se aplica a grafos não direcionados e conexos. Ela conecta todos os vértices do grafo e tem peso total mínimo, onde o peso é dado pela soma dos pesos das arestas da árvore.

**Observação**

Grafos não direcionados são aqueles em que as arestas não têm direção, ou seja, a relação de conexão é bidirecional. Isso significa o seguinte: se um vértice A está conectado a um vértice B, então o vértice B também está conectado ao vértice A.

Por sua vez, um grafo conexo é aquele em que existe pelo menos um caminho entre qualquer par de vértices. Ou seja, é possível partir de um vértice qualquer para outro vértice do grafo seguindo um caminho de arestas.

Um grafo pode ser tanto não direcionado quanto conexo, ou pode ter apenas uma dessas propriedades.

Uma árvore mínima é uma subestrutura do grafo original que conecta todos os vértices com um conjunto mínimo de arestas. Existem diferentes algoritmos para encontrar a árvore mínima de um grafo, sendo o algoritmo de Kruskal e o algoritmo de Prim os mais conhecidos.

Ambos os algoritmos possuem uma complexidade de tempo de O(E log E), onde E é o número de arestas do grafo. A árvore mínima tem várias aplicações em problemas de otimização, como em redes de comunicação, roteamento e planejamento de projetos.

Ela também é útil em problemas de transporte de energia elétrica, em que é necessário minimizar o custo de construção de novas linhas de transmissão. Além disso, ela pode ser usada em problemas de clusterização, quando o objetivo é dividir um conjunto de pontos em clusters de forma que a soma das distâncias dentro de cada cluster seja mínima.

Vale ressaltar que a árvore mínima pode não ser única, ou seja, existem vários conjuntos de arestas que conectam todos os vértices com peso total mínimo. No entanto, todas as árvores mínimas possuem o mesmo número de arestas e o mesmo peso total.

Por fim, é importante lembrar que o conceito de árvore mínima só se aplica a grafos não direcionados e conexos. Em grafos desconexos, é possível encontrar uma árvore mínima para cada componente conexo separadamente. Em grafos direcionados, é necessário utilizar conceitos similares, como a floresta mínima ou o caminho mínimo, dependendo do problema em questão.

**O problema da árvore geradora mínima (AGM)**

É um problema clássico em teoria dos grafos que busca encontrar a árvore que conecta todos os vértices de um grafo ponderado de forma que o peso total da árvore seja mínimo. Em outras palavras, dada uma rede de nós conectados por arestas ponderadas, o objetivo é encontrar um subconjunto de arestas que conecta todos os nós e que tenha o menor peso total possível.

O problema da AGM é importante porque é comum em muitas aplicações práticas, como a construção de redes de comunicação, a roteirização de veículos e a modelagem de sistemas de transporte e distribuição de energia. A solução ótima para o problema da AGM pode ser encontrada usando algoritmos, como o de Kruskal ou o de Prim.

Todavia, encontrar a solução ótima para o problema da AGM em grafos muito grandes pode ser computacionalmente caro e demorado, especialmente se o grafo for denso. Portanto, existem muitas heurísticas e algoritmos aproximados que podem ser usados para encontrar uma solução razoavelmente boa em um tempo mais curto.

Algumas dessas heurísticas e algoritmos aproximados incluem o algoritmo de Boruvka, o de Kruskal aleatório e o algoritmo de Prim com fila de prioridade limitada. Eles podem ser usados para obter uma aproximação da solução ótima do problema da AGM em um tempo razoável.

Além disso, o problema da AGM é conhecido por ser NP-completo, o que significa que não existe um algoritmo polinomial que possa encontrar a solução ótima em tempo polinomial para todos os casos. Portanto, para alguns casos de problema da AGM, pode ser necessário usar algoritmos de programação inteira ou heurísticas mais avançadas para obter uma solução aproximada.

Observe alguns pontos importantes a seguir:

- uma árvore geradora para um grafo conexo é uma árvore sem raiz, cujo conjunto de nós coincide com o conjunto de nós do grafo e cujos arcos são (alguns dos) arcos do grafo;

- um dos algoritmos para a construção da árvore geradora foi apresentado por Prim em 1957.

Para exemplificar, a computação do algoritmo fez uso de um grafo cuja matriz de adjacência modificada é como o ilustrado a seguir:

$$\begin{bmatrix} - & 6.7 & 5.2 & 2.6 & 5.6 & 3.6 \\ 6.7 & - & 5.7 & 7.3 & 5.1 & 3.2 \\ 5.2 & 5.7 & - & 3.4 & 8.5 & 4.0 \\ 2.8 & 7.3 & 3.4 & - & 8.0 & 4.4 \\ 5.6 & 5.1 & 8.5 & 8.0 & - & 4.6 \\ 3.6 & 3.2 & 4.0 & 4.4 & 4.6 & - \end{bmatrix}$$

Figura 45

Para ilustrar o algoritmo, empregar-se-á a mesma matriz de adjacência.

- Iniciar-se-á, arbitrariamente, no nó 1.
- IN = {1} (inicialização arbitrária no nó 1).
- Entre os nós adjacentes ao nó 1, aquele cuja aresta para alcançá-lo apresenta menor distância é o nó 4, que é selecionado: IN = {1,4}.
- Entre os nós adjacentes ao nó 4, aquele cuja aresta para alcançá-lo apresenta menor distância é o nó 3, que é selecionado: IN = {1,4,3}.
- Entre os nós adjacentes do nó 3, aquele cuja aresta para alcançá-lo apresenta menor custo é o nó 4. Uma vez que este foi inserido no conjunto IN, escolhe-se o nó 6: IN = {1,4,3,6}.
- Entre os nós adjacentes ao nó 6, aquele cuja aresta para alcançá-lo apresenta menor custo é o nó 2, que é selecionado: IN = {1,4,3,6,2}.
- Resta o nó 5: IN = {1,4,3,6,2,5}.

## 5.1 Algoritmo de Prim

É usado para encontrar a árvore mínima de um grafo não direcionado e conexo. Ele é um dos mais conhecidos e utilizados para resolver esse tipo de problema em teoria dos grafos.

O algoritmo de Prim começa com um vértice arbitrário do grafo e adiciona a aresta de menor peso que conecta esse vértice a um vértice ainda não incluído na árvore mínima. Em seguida, a aresta de menor peso que conecta qualquer vértice já incluído na árvore mínima a um vértice ainda não incluído é adicionada. O processo é repetido até que todos os vértices estejam incluídos na árvore mínima.

Ele utiliza uma estrutura de dados chamada heap (ou fila de prioridade) para manter as arestas disponíveis para serem adicionadas à árvore mínima. A cada passo do algoritmo, o vértice de menor distância é escolhido e suas arestas adjacentes são adicionadas ao heap. O heap é atualizado a cada iteração do algoritmo, de forma que as arestas de menor peso sempre fiquem no topo do heap.

Esse algoritmo possui uma complexidade de tempo de O(E log V), sendo E o número de arestas do grafo e V o número de vértices. Esse é o mesmo tempo de execução do algoritmo de Kruskal, outro algoritmo popular para encontrar a árvore mínima de um grafo. No entanto, o algoritmo de Prim é geralmente mais eficiente em grafos esparsos, ou seja, em grafos com poucas arestas em relação ao número de vértices.

Destaca-se que ele pode ser utilizado em uma variedade de aplicações em teoria dos grafos, como em redes de comunicação, roteamento, planejamento de projetos e clusterização. Ele é um algoritmo relativamente simples e fácil de implementar, o que o torna uma ferramenta útil para resolver problemas de otimização em diversas áreas.

O pseudocódigo para o algoritmo de Prim é o seguinte:

```
Prim(G, s):
        para cada vértice v em G:
                chave[v] = infinito
                pai[v] = nulo
        chave[s] = 0
        Q = conjunto de todos os vértices em G
        enquanto Q não estiver vazio:
                u = vértice em Q com a menor chave
                remover u de Q
  para cada vértice v adjacente a u:
        se v ainda estiver em Q e peso(u,v) < chave[v]:
          pai[v] = u
          chave[v] = peso(u,v)
  retornar a AGM representada pelo conjunto de pares (pai[v],v) para todos os
vértices v em G, exceto para o vértice raiz s
```

Nesse pseudocódigo, G é o grafo ponderado e conexo, s é o vértice inicial para começar a construir a AGM, e peso(u,v) é a função que retorna o peso da aresta entre u e v. O algoritmo mantém duas listas de tamanho n, uma para a chave e outra para o pai de cada vértice, onde n é o número de vértices do grafo. A lista-chave é inicializada com o valor infinito para todos os vértices, exceto o vértice inicial s, que é inicializado com chave[s] = 0. A lista Q é inicializada com todos os vértices de G.

Em seguida, o algoritmo itera até que a lista Q esteja vazia. A cada iteração, o algoritmo escolhe o vértice u em Q com a menor chave e o remove da lista. Em seguida, o algoritmo percorre todos os vértices v adjacentes a u e atualiza a chave[v] se o peso(u,v) é menor que a chave[v]. O pai[v] é atualizado para u se a chave[v] for atualizada.

Ao final do algoritmo, a AGM é representada pelo conjunto de pares (pai[v],v) para todos os vértices v em G, exceto para o vértice raiz s.

Esse conjunto de pares representa as arestas da AGM. Esta é construída adicionando essas arestas na ordem dada pelo conjunto de pares. O algoritmo de Prim garante que a AGM encontrada é a árvore geradora mínima do grafo G, ou seja, a árvore com o menor custo total que conecta todos os vértices do grafo. O tempo de execução do algoritmo de Prim é de O(E log V), onde E é o número de arestas e V é o número de vértices do grafo.

Agora observe um exemplo de aplicação do algoritmo de Prim.

## Exemplo de aplicação

Considerando o grafo a seguir, no primeiro momento temos:

Arestas incluídas no subgrafo: {}

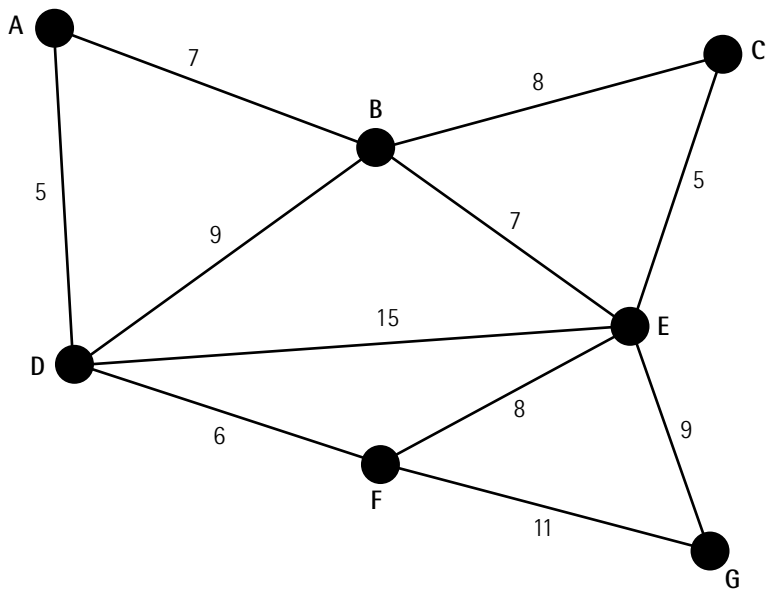



Figura 46

Disponível em: https://bit.ly/400FGEp. Acesso em: 24 abr. 2023.

Arestas incluídas no subgrafo: {DA}

O vértice D foi escolhido como ponto inicial do algoritmo. Vértices A, B, E e F estão conectados com D através de uma única aresta. A é o vértice mais próximo de D e, portanto, a aresta AD será escolhida para formar o subgrafo.

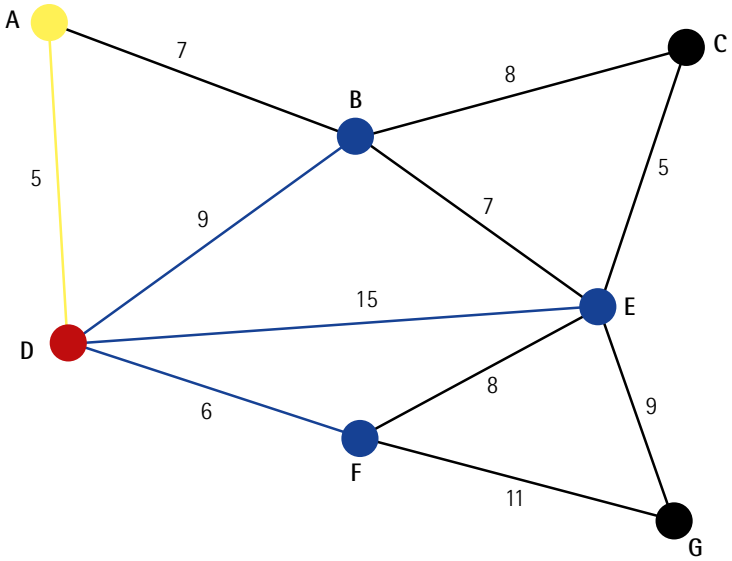



Figura 47

Disponível em: https://bit.ly/3AqRYI0. Acesso em: 24 abr. 2023.

Arestas incluídas no subgrafo: {DA, DF}

O próximo vértice escolhido é o mais próximo de D ou A. B está a uma distância 9 de D, E em uma distância 15 e F em uma distância 6. E A está a uma distância de 7 de B. Logo, devemos escolher a aresta DF, pois é o menor peso.

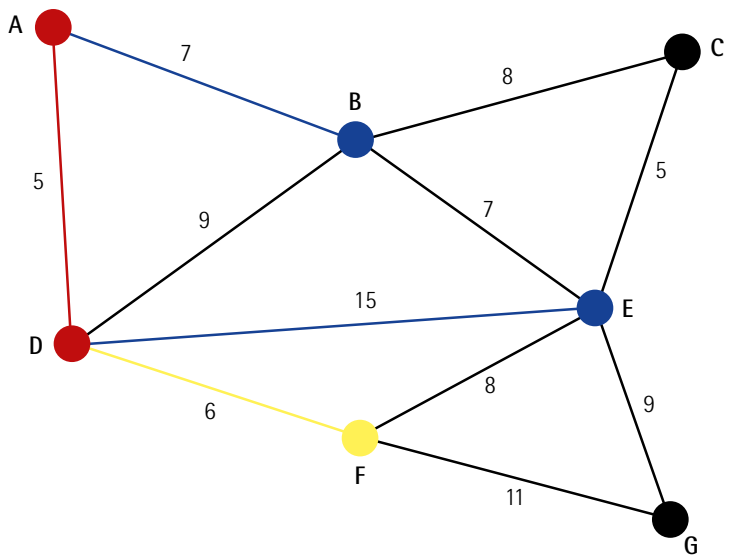



Figura 48

Disponível em: https://bit.ly/40AQm9s. Acesso em: 24 abr. 2023.

Arestas incluídas no subgrafo: {DA, DF, AB}

Agora devemos escolher o vértice mais próximo dos vértices A, D ou F. A aresta em questão é a aresta AB.

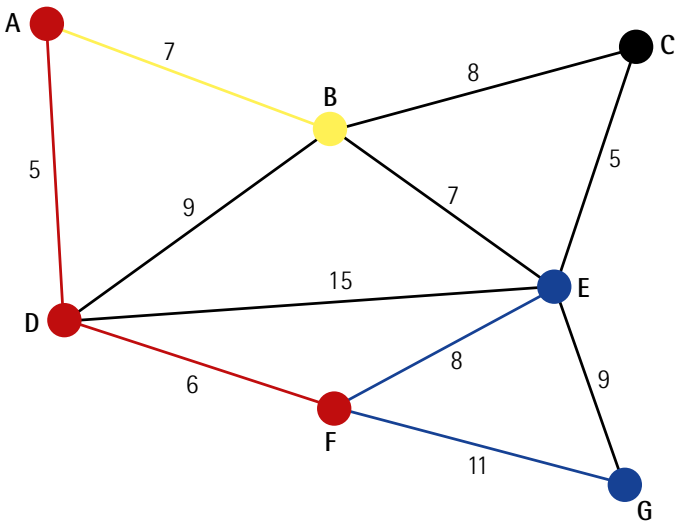



Figura 49

Disponível em: https://bit.ly/3V9712H. Acesso em: 24 abr. 2023.

Arestas incluídas no subgrafo: {DA, DF, AB, BE}

Agora podemos escolher entre os vértices C, E e G. C está a uma distância de 8 de B, E está a uma distância 7 de B e G está a 11 de F. E é o mais próximo do subgrafo e, portanto, escolhemos a aresta BE.

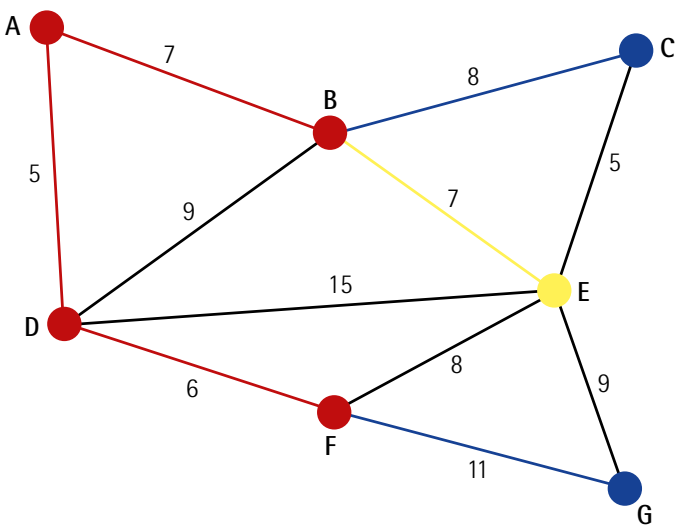



Figura 50

Disponível em: https://bit.ly/3H8GhJO. Acesso em: 24 abr. 2023.

Arestas incluídas no subgrafo: {DA, DF, AB, BE, EC}

Restam somente os vértices C e G. C está a uma distância 5 de E e de G a E 9. C é escolhido, então a aresta EC entra no subgrafo construído.

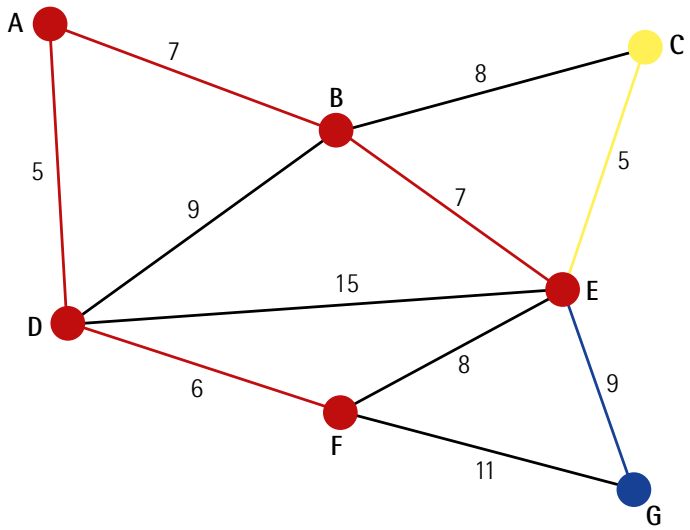



Figura 51

Disponível em: https://bit.ly/3Ls1GQZ. Acesso em: 24 abr. 2023.

Arestas incluídas no subgrafo: {DA, DF, AB, BE, EC, EG}

Agora só resta o vértice G. Ele está a uma distância 11 de F e 9 de E. E é o mais próximo, então G entra no subgrafo conectado pela aresta EG.

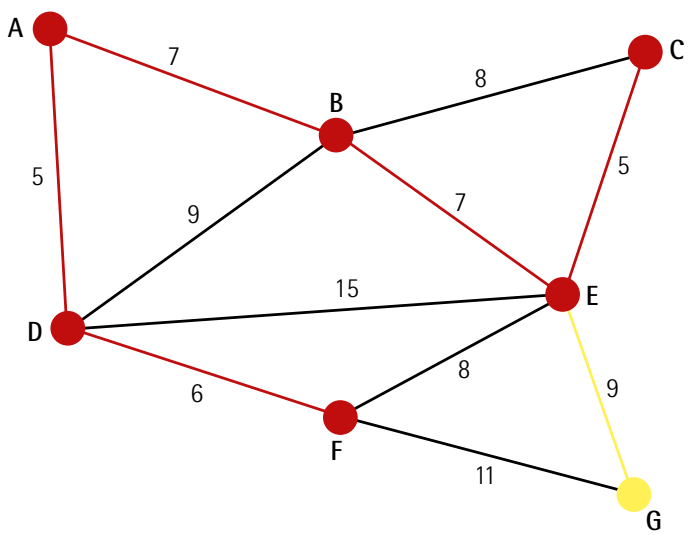



Figura 52

Disponível em: https://bit.ly/3n1A1N6. Acesso em: 24 abr. 2023.

Arestas incluídas no subgrafo: {DA, DF, AB, BE, EC, EG}

Aqui está o fim do algoritmo. Já o subgrafo formado pelas arestas em vermelho representam a árvore geradora mínima. Nesse caso, essa árvore indica como a soma de todas as suas arestas o número 39.

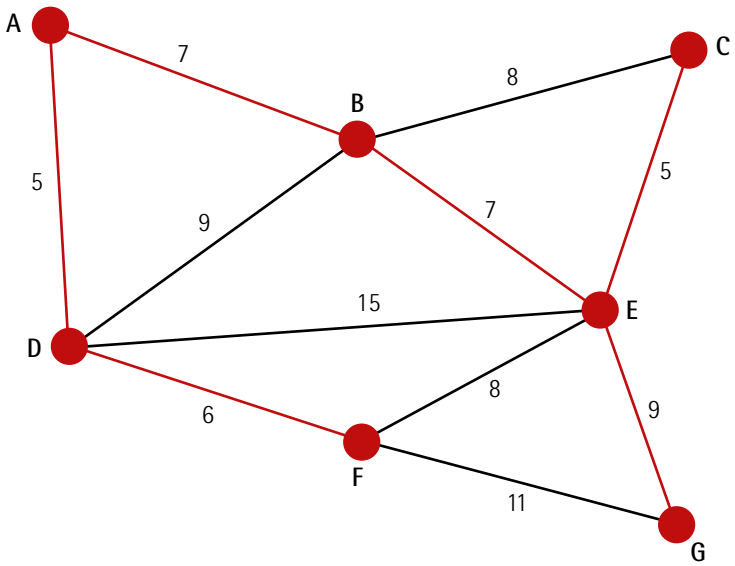



Figura 53

Disponível em: https://bit.ly/41Wb9FB. Acesso em: 24 abr. 2023.

## 5.2 Algoritmo de Kruskal

É usado para encontrar a árvore mínima de um grafo não direcionado e conexo. Ele é um dos algoritmos mais conhecidos e utilizados para resolver esse tipo de problema em teoria dos grafos.

O algoritmo de Kruskal começa com uma floresta que contém todos os vértices do grafo como árvores individuais. Em seguida, as arestas são ordenadas por peso crescente e processadas em ordem crescente. Para cada aresta, se os vértices que ela conecta pertencem a árvores diferentes, a aresta é adicionada à floresta e as árvores são unidas em uma única árvore. O processo é repetido até que todas as arestas tenham sido processadas ou até que haja apenas uma árvore na floresta, que será a árvore mínima.

Ele usa uma estrutura de dados chamada union-find (união-busca) para manter as árvores individuais e encontrar rapidamente a árvore a que um determinado vértice pertence. A cada iteração do algoritmo, a estrutura union-find é atualizada para unir as árvores que contêm os vértices conectados pela aresta processada.

Esse algoritmo possui uma complexidade de tempo de O(E log E), onde E é o número de arestas do grafo. Essa é a mesma complexidade de tempo do algoritmo de Prim, outro algoritmo popular para encontrar a árvore mínima de um grafo. No entanto, o algoritmo de Kruskal é geralmente mais eficiente em grafos densos, ou seja, em grafos com muitas arestas em relação ao número de vértices.

Ele pode ser utilizado em uma variedade de aplicações em teoria dos grafos, como em redes de comunicação, roteamento, planejamento de projetos e clusterização. Ele é um algoritmo relativamente simples e fácil de implementar, o que o torna uma ferramenta útil para resolver problemas de otimização em diversas áreas.

Observe a seguir alguns pontos importantes:

- o algoritmo de Kruskal destina-se a encontrar uma árvore geradora mínima (AGM) em um grafo conexo;
- o algoritmo de Kruskal inclui os arcos em ordem crescente de distância onde quer que estejam no grafo;
- a única restrição é que um arco não é incluído se a sua inclusão criar um ciclo;
- o algoritmo termina quando todos os nós estiverem incorporados em uma estrutura conexa.

### Algoritmo AGM Kruskal

```
Outra AGM (matriz n x n; coleção de arcos T):
// Algoritmo de Kruskal para encontrar uma árvore geradora mínima;
// inicialmente, T é vazio: ao final, T = árvore geradora mínima.
Ordene os arcos em G por distância em ordem crescente:
        Repita:
                Se o próximo arco na ordem não completa um ciclo:
                Então, inclua esse arco em T.
                Fim do Se.
        Até T ser conexo e conter todos os nós em G.
Fim de OutraAGM.
```

Observe a seguir um exemplo de aplicação do algoritmo de Kruskal.

## Exemplo de aplicação

Considerando o grafo a seguir, no primeiro momento temos:

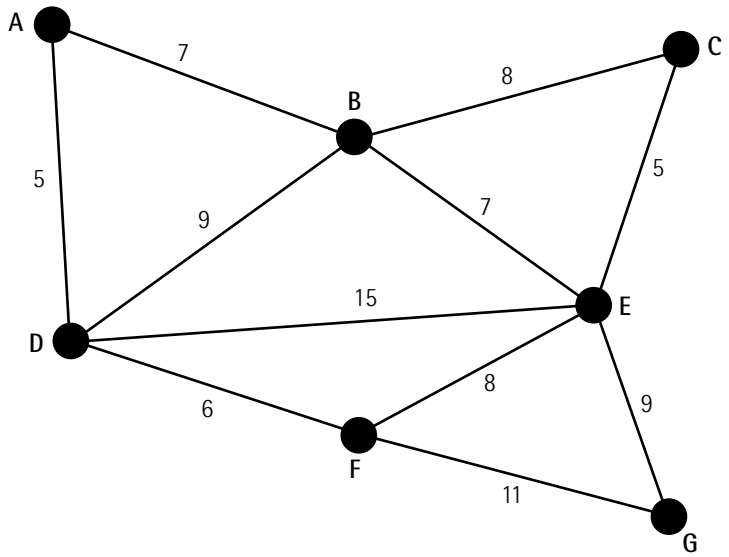



Figura 54

Disponível em: https://bit.ly/3Ap54Wd. Acesso em: 24 abr. 2023.

As arestas AD e CE são as mais leves do grafo e ambas podem ser selecionadas. Escolhe-se ao acaso AD.

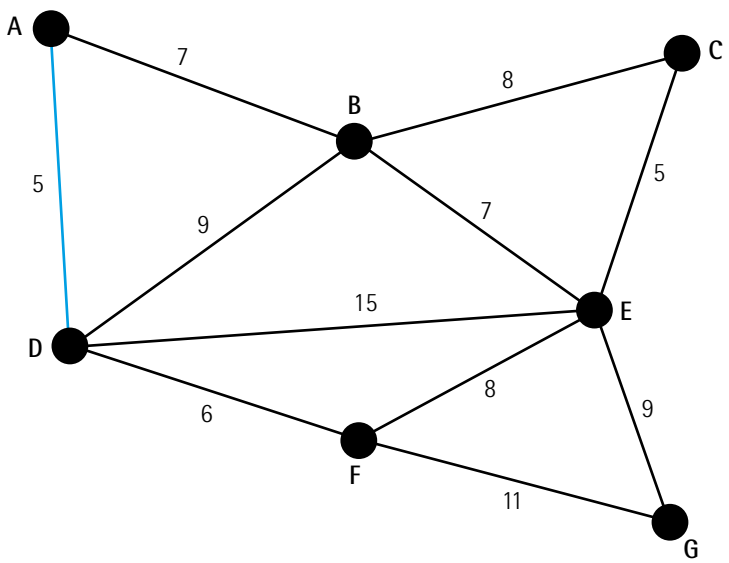



Figura 55

Disponível em: https://bit.ly/3L3WoJL. Acesso em: 24 abr. 2023.

Agora a aresta CE é a mais leve. Já que ela não forma um laço com AD, ela é selecionada.

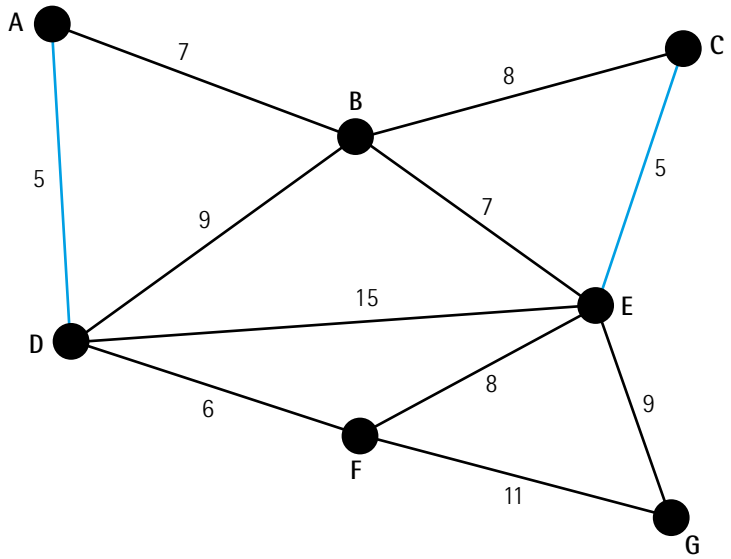



Figura 56

Disponível em: https://bit.ly/3L3WiSp. Acesso em: 24 abr. 2023.

A próxima aresta é a DF com peso 6. Ela não forma um laço com as arestas já selecionadas, então ela é selecionada.

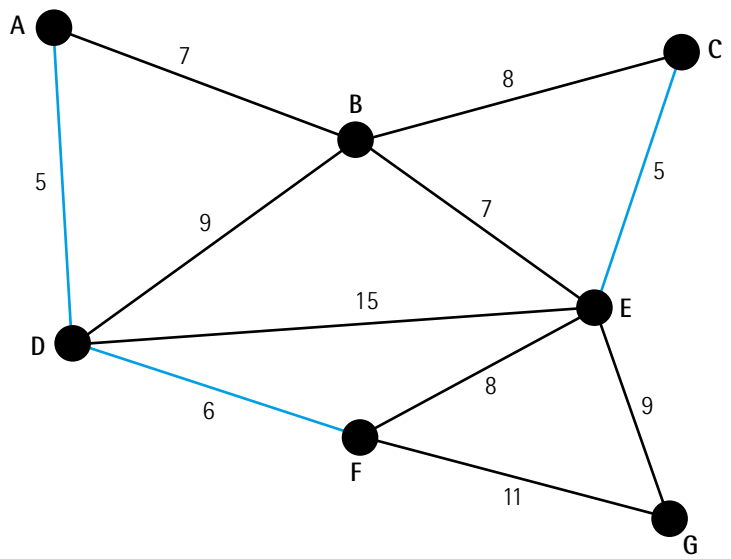



Figura 57

Disponível em: https://bit.ly/3otWS4k. Acesso em: 24 abr. 2023.

Assim, duas arestas com peso 7 podem ser selecionadas e uma é escolhida ao acaso. A aresta BD é marcada porque forma um laço com as outras arestas já selecionadas.

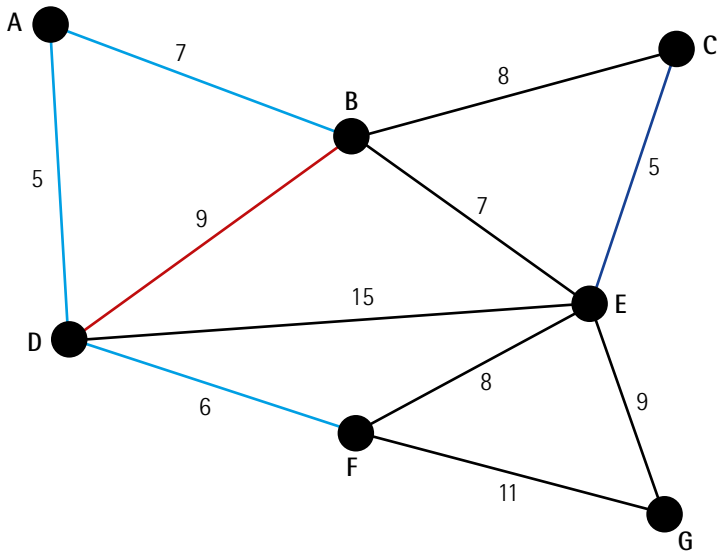



Figura 58

Disponível em: https://bit.ly/3V4ebp6. Acesso em: 24 abr. 2023.

Agora a outra aresta de peso 7 é selecionada porque cobre todos os requisitos de seleção. Similarmente ao passo anterior, outras arestas são marcadas para não serem selecionadas, pois resultariam em um laço.

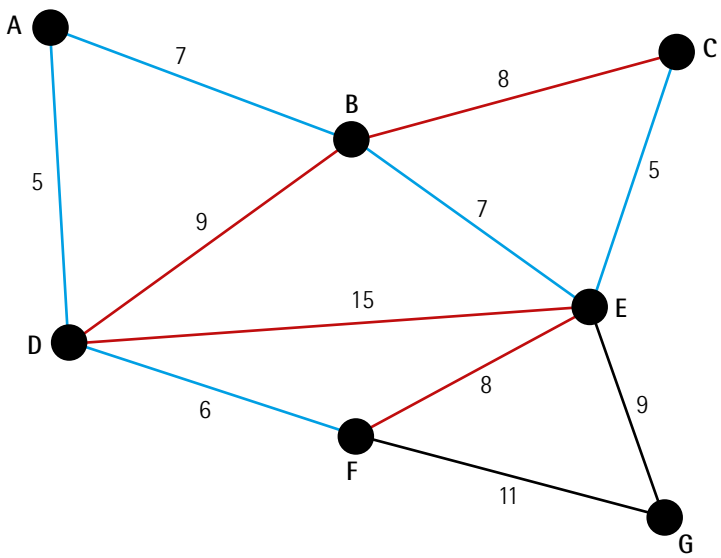



Figura 59

Disponível em: https://bit.ly/43VgLBl. Acesso em: 24 abr. 2023.

Para finalizar, é selecionada a aresta EG com peso 9 e FG é marcada. Já que agora todas as arestas disponíveis formariam um laço, chega-se ao final do algoritmo e a árvore geradora mínima é encontrada.

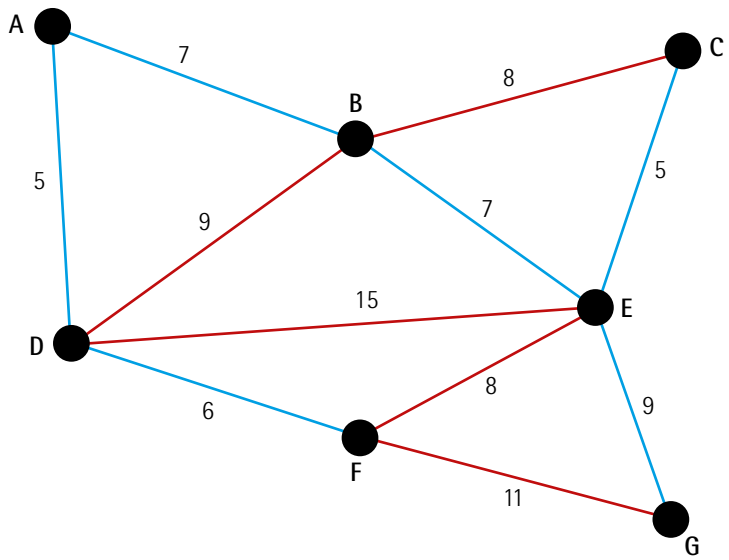



Figura 60

Disponível em: https://bit.ly/41zbNZA. Acesso em: 24 abr. 2023.

**Saiba mais**

Com o objetivo de entender mais a respeito dos algoritmos, leia a seguinte obra:

CORMEN, T. H. *et al. Algoritmos*: teoria e prática. Rio de Janeiro: Elsevier, 2012.

## 6 CAMINHOS MÍNIMOS

Em teoria dos grafos, caminhos mínimos se referem aos caminhos mais curtos entre dois vértices de um grafo ponderado. Esses caminhos podem ser encontrados utilizando algoritmos específicos, como o de Dijkstra e o de Bellman-Ford.

Em um grafo ponderado, cada aresta tem um peso ou custo associado, que representa o custo de percorrer a aresta. O objetivo dos algoritmos de caminhos mínimos é encontrar o caminho mais curto entre dois vértices, ou seja, o caminho com a soma mínima dos pesos das arestas.

O algoritmo de Dijkstra é um algoritmo de caminhos mínimos que funciona em grafos com pesos não negativos, enquanto o de Bellman-Ford funciona em grafos com pesos negativos. Ambos utilizam técnicas de programação dinâmica para encontrar o caminho mais curto.

Os algoritmos de caminhos mínimos têm diversas aplicações em teoria dos grafos e em outras áreas, como em redes de comunicação, roteamento, planejamento de trajetórias em robótica e jogos de estratégia. Eles permitem encontrar a rota mais eficiente para percorrer um grafo ponderado, o que pode ser útil em uma variedade de contextos.

Observe a seguir alguns pontos importantes.

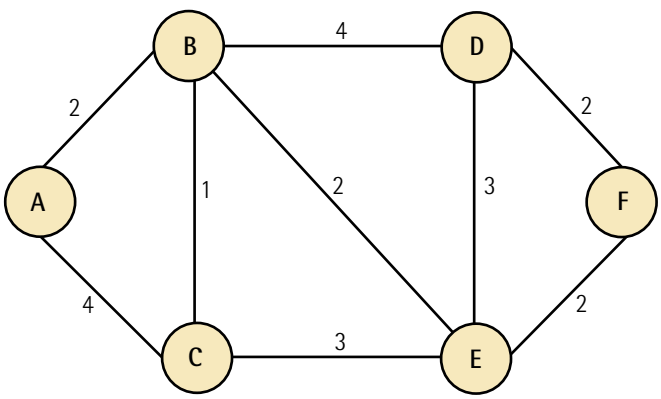
Seja um grafo simples e conexo, e com peso, onde os pesos são positivos:

- o peso representa, muitas vezes, a distância entre dois nós.
- se o grafo for conexo, então, existe, ao menos, um caminho entre dois nós quaisquer. Ainda, podem existir muitos caminhos.

A pergunta é: se existirem muitos caminhos, como encontrar um caminho com peso mínimo?

A seguir são apresentados algoritmos de caminho mínimo.

Considere o seguinte grafo e a matriz de adjacência modificada:




$$A = \begin{bmatrix} \infty & 2 & 4 & \infty & \infty & \infty \\ 2 & \infty & 1 & 4 & 2 & \infty \\ 4 & 1 & \infty & \infty & 3 & \infty \\ \infty & 4 & \infty & \infty & 3 & 2 \\ \infty & 2 & 3 & 3 & \infty & 2 \\ \infty & \infty & \infty & 2 & 2 & \infty \end{bmatrix}$$

Figura 61

Ao se ordenar as arestas, segundo os pesos, tem-se:

- d = 1, aresta: B-C;
- d = 2, arestas: B-A; B-E; E-F; F-D;
- d = 3, arestas E-D e E-C;
- d = 4: C-A.

Árvore geradora mínima:

- A-C;
- C-B;
- B-E;
- E-D;
- E-F.

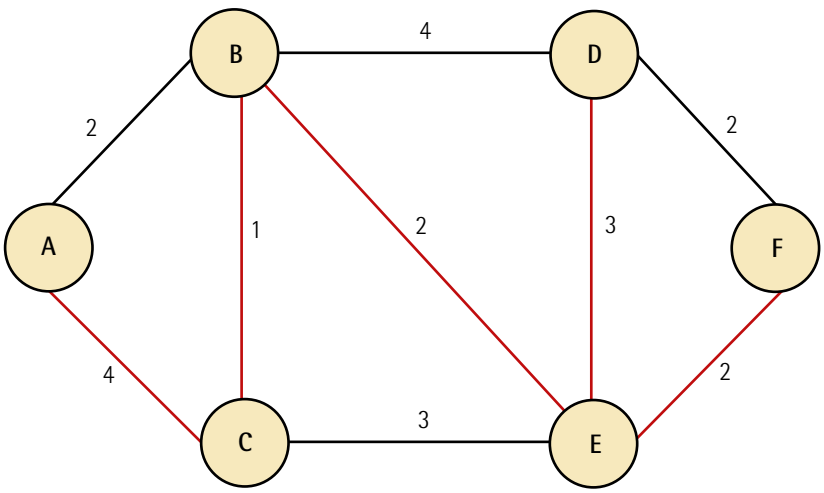



Figura 62

## 6.1 Caminho mínimo de fonte única

Em teoria dos grafos, é um problema em que se deseja encontrar o caminho mais curto entre um vértice de origem e todos os outros vértices do grafo. Ele é comumente referido como o problema do caminho mínimo de fonte única.

Em um grafo ponderado, cada aresta tem um peso ou custo associado, que representa o custo de percorrer a aresta. O objetivo do problema do caminho mínimo de fonte única é encontrar o caminho mais curto entre um vértice de origem e todos os outros vértices do grafo, ou seja, o caminho com a soma dos pesos mínimos das arestas para cada vértice.

Existem vários algoritmos que podem ser usados para resolver o problema do caminho mínimo de fonte única, dependendo das características do grafo. Alguns exemplos incluem o algoritmo de Dijkstra, o de Bellman-Ford, o de A* etc.

O algoritmo de Dijkstra é amplamente usado para encontrar o caminho mínimo de fonte única em grafos com pesos não negativos. Ele funciona construindo um conjunto de vértices não visitados e um conjunto de vértices visitados, calculando as distâncias mínimas a partir do vértice de origem para todos os vértices não visitados. O algoritmo utiliza uma fila de prioridade para selecionar o vértice com a menor distância entre os vértices não visitados a cada iteração.

O algoritmo de Bellman-Ford é outro algoritmo que pode ser usado para encontrar o caminho mínimo de fonte única em grafos ponderados, inclusive em grafos com pesos negativos. Ele funciona atualizando as distâncias dos vértices iterativamente, até que não haja mais atualizações possíveis, indicando que as distâncias são as mínimas possíveis.

O problema do caminho mínimo de fonte única é importante em muitas áreas, como em roteamento de redes, planejamento de rotas em sistemas de transporte e roteamento de pacotes em sistemas de comunicação. Ele permite encontrar a rota mais eficiente para percorrer um grafo ponderado a partir de um vértice de origem, o que pode ser útil em muitas aplicações.

### 6.1.1 O caminho mínimo de Bellman-Ford

O algoritmo de Bellman-Ford resolve o problema do caminho mínimo de fonte única em grafos ponderados, inclusive em grafos com pesos negativos. Ele funciona atualizando as distâncias dos vértices iterativamente, até que não haja mais atualizações possíveis, indicando que as distâncias são as mínimas possíveis.

O algoritmo começa com a atribuição de uma distância infinita para todos os vértices, exceto o vértice de origem, que recebe distância zero. Em seguida, o algoritmo relaxa as arestas em ordem crescente de seus pesos. Relaxar uma aresta significa atualizar a distância do vértice destino se a distância do vértice de origem somada com o peso da aresta é menor do que a distância atual do vértice destino. Se houver uma atualização em uma iteração, o algoritmo continua a relaxar as arestas em ordem crescente de peso. O algoritmo repete esse processo V-1 vezes, onde V é o número de vértices no grafo, para garantir que todas as distâncias sejam atualizadas corretamente. Se ainda houver uma atualização após a iteração V-1, isso significa que há um ciclo negativo no grafo.

Um ciclo negativo é um ciclo em que a soma dos pesos das arestas é negativa. Quando o algoritmo de Bellman-Ford encontra um ciclo negativo, ele não pode determinar o caminho mínimo de fonte única, pois o peso do ciclo negativo pode ser reduzido indefinidamente, fazendo com que a distância

mínima não exista. Nesse caso, o algoritmo de Bellman-Ford retorna um sinal de erro, indicando a presença de um ciclo negativo.

O algoritmo de Bellman-Ford é mais lento do que o de Dijkstra para grafos com pesos não negativos, mas é capaz de lidar com grafos com pesos negativos. Ele é usado em várias aplicações, como em roteamento de pacotes em sistemas de comunicação, análise de redes de transporte e análise de fluxos em redes.

### Algoritmo de Bellman-Ford

Executa uma série de cálculos, procurando encontrar os caminhos mínimos, sucessivamente, de comprimento 1, depois de comprimento 2, e assim por diante, até o comprimento máximo n-1.

```
Algoritmo OutroCaminhoMinimo:
OutroCaminhoMinimo (matriz n x n A; no x; vetor de inteiros d, vetor de nós s[y];
// Algoritmo de Bellman-Ford. A é uma matriz de adjacência;
// modificada de um grafo simples e conexo com pesos positivos;
// x é um nó no grafo, quando o algoritmo terminar os nós;
// do Caminho Mínimo de x para um nó y são y, s[y], s[s[y]],...,x;
// a distância correspondente é d[y].

Variáveis locais:
Nós z, p;              // nós temporários;
Vetor de inteiros t;   // vetor de distâncias temporários.
// inicializa os vetores d e s; estabelece os caminhos mínimos de comprimento 1
a partir de x d[x] = 0.
Para todos os nós z diferentes de x, faça:
        d[z] = A[x, z]
        s[z] = x
Fim do para.
// encontrar os Caminhos Mínimos de comprimentos 2, 3 etc.:
Para i = 2, até n – 1, faça:
        t = d
        // modifica t para guardar os menores caminhos de;
        // comprimento i.
        Para todos os nós z diferentes de x, faça:
                // encontra o Caminho Mínimo com mais um arco;
                p = nó em G para o qual (d[p] + A[p,z]) é mínimo;
                t[z] = d[p] + A[p,z]
                Se p≠ z, então:
                        s[z] = p
                Fim do se.
        Fim do para.
        d = t;

Fim do para.
Fim do OutroCaminhoMinimo.
```

O exemplo ilustrado a seguir foi baseado em Gersting (2014).

**Exemplo de aplicação**

Deseja-se usar o algoritmo de Bellman-Ford para encontrar o caminho da origem do nó 2 para qualquer outro nó.

Após duas iterações, tem-se:

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| d | 3 | 0 | 2 | 3 | 3 | 4 | 1 | 2 |
| s | 2 | - | 2 | 3 | 8 | 1 | 2 | 7 |

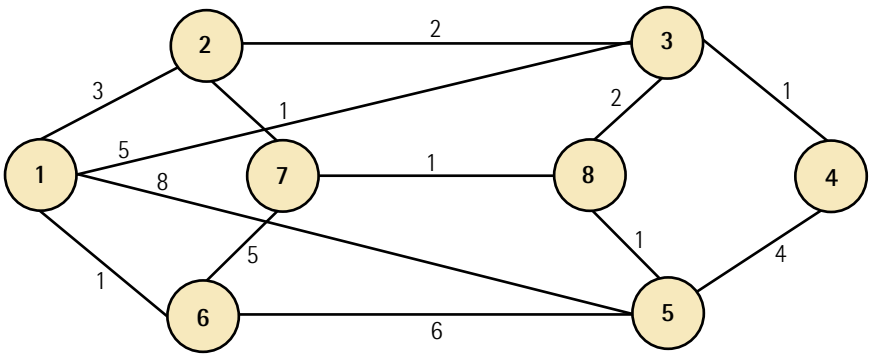



Figura 63

## 6.1.2 Algoritmo de Dijkstra

É um algoritmo de caminho mínimo de fonte única que resolve o problema de encontrar o caminho mais curto em um grafo ponderado com arestas não negativas. Seu objetivo é determinar o caminho mais curto a partir de um nó de origem para todos os outros nós do grafo.

O algoritmo de Dijkstra começa selecionando um nó inicial e definindo sua distância como zero. Em seguida, avalia todos os nós adjacentes a esse nó inicial, calculando suas distâncias até o nó inicial. Se a distância do nó adjacente é menor do que a distância atual armazenada, a distância é atualizada. Esse processo é repetido para todos os nós adjacentes, até que todos os nós sejam visitados.

Subsequentemente, o algoritmo continua expandindo a fronteira de nós visitados, selecionando sempre o nó com a menor distância conhecida e atualizando as distâncias dos seus nós adjacentes. O algoritmo termina quando todos os nós no grafo foram visitados ou quando o nó de destino é alcançado.

O algoritmo de Dijkstra usa uma fila de prioridade para armazenar os nós visitados e suas distâncias. Isso permite que o algoritmo encontre rapidamente o próximo nó com a menor distância conhecida e também permite que as distâncias sejam atualizadas de forma eficiente.

Ele é amplamente utilizado em várias aplicações, como em sistemas de roteamento de rede, análise de redes sociais, sistemas de transporte e análise de dados geoespaciais. No entanto, ele só pode ser aplicado em grafos com arestas não negativas e tem um tempo de execução de O(E + V log V), onde E é o número de arestas e V é o número de nós no grafo.

O algoritmo de Dijkstra é conhecido por sua eficiência e simplicidade, mas sua limitação em lidar com arestas negativas pode ser contornada pelo algoritmo de Bellman-Ford, que é um algoritmo de caminho mínimo de fonte única mais genérico, capaz de lidar com arestas negativas.

Existem também variantes do algoritmo de Dijkstra que visam melhorar sua eficiência em certos cenários. Por exemplo, o algoritmo de Dijkstra bidirecional pode ser usado em grafos densos, onde a distância até o destino é pequena em relação ao número de nós no grafo. Nesse algoritmo, a busca é iniciada a partir do nó de origem e do nó de destino simultaneamente, expandindo as fronteiras de ambos os lados e parando quando as fronteiras se encontram.

Além disso, o algoritmo de A* é uma variante do algoritmo de Dijkstra que utiliza uma heurística para direcionar a busca em direção ao nó de destino, reduzindo o número de nós visitados e melhorando a eficiência do algoritmo.

Em resumo, o algoritmo de Dijkstra é uma ferramenta essencial na teoria dos grafos, permitindo encontrar caminhos mínimos de fonte única em grafos ponderados com arestas não negativas. Sua simplicidade e eficiência o tornam uma escolha popular para uma variedade de aplicações em diversas áreas, desde redes de transporte até análise de redes sociais.

O algoritmo proposto por E. W. Dijkstra para encontrar o caminho mínimo é apresentado por Gersting (2014), conforme acentuado a seguir.

```
Caminho Mínimo (matriz n x n; nós x, y):
// Algoritmo de Dijkstra. A é uma matriz de adjacência modificada;
// de um grafo simples e conexo com pesos positivos; x e y são nós;
// no grafo; o algoritmo escreve os nós do caminho mínimo de x;
// para y e a distância correspondente.

Variáveis locais:
Conjunto de nós IN;                    // nós cujo caminho mínimo de x é
conhecido.
Nós z; p // nós temporários.
Vetor de inteiros; d // para cada nó, distância de x usando  // nós em IN.
Vetor de nós s; // para cada nó, nó anterior no caminho                    //
mínimo.
Inteiro DistânciaAnterior; // distância para comparar.
// inicializa o conjunto IN e os vetores d e s.
IN = {x}
d[x] = 0
Para todos os nós z não pertencentes a IN, faça:
d[z] = A[x, z]
s[z] = x
Fim do para.
```

```
// coloca nós em IN
Enquanto y não pertence a IN, faça:
// adiciona o nó de distância mínima não                                    //
pertencente a IN
p = nó z não pertencente a IN com d[z] mínimo
IN = IN ∪ {p}
d[z] = min(d[z], d[p] + A[p, z]
        Se d[z] ≠ DistânciaAnterior, então:
                S[z] = p
        Fim do se.
Fim do para.
Fim do enquanto
// escreve os nós do caminho
                        Escreva ("Em ordem inversa, os nós do caminho são")
                        escreva (y)
                        Repita:
                                Escreva (s[z])
                                z = s[z]
                        Até z = x
// escreve a distância correspondente
Escreva ("A distância percorrida é: ", d[y])
Fim de CaminhoMínimo.
```

Observe com atenção exemplo seguinte, assim você entenderá como o algoritmo é encontrado.

## Exemplo de aplicação

Considere o grafo e a matriz de adjacência modificada apresentados a seguir. Deseja-se saber o caminho mínimo entre os nós A e F. A matriz de adjacência modificada considera para cada nó o peso daquela aresta que está incidindo sobre o nó. Então, no exemplo o nó A é adjacente do nó B com o peso 2, isto é, de A para B temos peso 2, do nó A para C temos peso 4, e assim sucessivamente. Para o nó de A até F não existe aresta de A para F, então anota-se infinito.

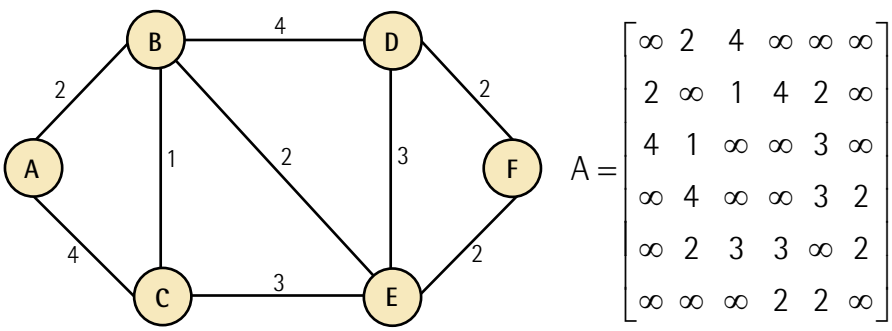



Figura 64

Seguidamente, deve-se somar d (B,x) a d e comparar o resultado com d (A,x). Entre os dois valores, seleciona-se o valor mínimo e insere-se no conjunto IN, o nó x, selecionado.

Repete-se, sucessivamente, até que todo o nó destino seja inserido em IN.

Assim, o algoritmo de Dijkstra prevê um conjunto IN, e nesse conjunto IN anota-se o nó origem. Vamos admitir nesse exemplo que se deseja calcular um caminho mínimo de um nó A para o nó F, então de A para A temos distância (d) igual a 0 e source (s), que significa origem no nó A. Dessa maneira, temos de A para B distância 2, de A para C distância 4, e assim sucessivamente. Em seguida, elege-se um próximo nó para o caminho que será aquele que apresenta menor distância no caso no B.

- IN = {A}.

| | A | B | C | D | E | F |
|---|---|---|---|---|---|---|
| d | 0 | 2 | 4 | ∞ | ∞ | ∞ |
| s | A | A | A | A | A | A |

| | A | B | C | D | E | F |
|---|---|---|---|---|---|---|
| d | 0 | 2 | 4 | ∞ | ∞ | ∞ |
| s | A | A | A | A | A | A |

Figura 65

- IN = {A,B}.
- d = 2.
- d[C]= min(4, 2 + A[B, C]) = min(4, 3) = 3.
- d[D]= min(∞, 2 + A[B, D]) = min(∞, 2 + 4) = 6.
- d[E]= min(∞, 2 + A[B, E]) = min(∞, 2 + 2) = 4.
- d[F]= min(∞, 2 + A[B, F]) = min(∞, 2 + ∞) = ∞.

| | A | B | C | D | E | F |
|---|---|---|---|---|---|---|
| d | 0 | 2 | 3 | 6 | 4 | ∞ |
| s | A | A | B | B | B | A |

| | A | B | C | D | E | F |
|---|---|---|---|---|---|---|
| d | 0 | 2 | 3 | 6 | 4 | ∞ |
| s | A | A | B | B | B | A |

Figura 66

- IN = {A, B, C}.
- d = 3.
- d[D]= min(6, 3 + A[C, D]) = min(6, 3 + 4) = 6.
- d[E]= min(4, 3 + A[C, E]) = min(4, 3 + 3) = 4.
- d[F]= min(∞, 3 + A[C, F]) = min(∞, 3 + ∞) = ∞.

| | A | B | C | D | E | F |
|---|---|---|---|---|---|---|
| d | 0 | 2 | 3 | 6 | 4 | ∞ |
| s | A | A | B | B | B | A |

| | A | B | C | D | E | F |
|---|---|---|---|---|---|---|
| d | 0 | 2 | 3 | 6 | 4 | ∞ |
| s | A | A | B | B | B | A |

Figura 67

- IN = {A, B, C, E}.
- d = 4.
- d[D]= min(6, 4 + A[E, D]) = min(6, 4 + 3) = 6.
- d[F]= min(∞, 4 + A[E, F]) = min(∞, 4 + 2) = 6.

Solução: caminho mínino = A-B-E-F.

| | A | B | C | D | E | F |
|---|---|---|---|---|---|---|
| d | 0 | 2 | 3 | 6 | 4 | ∞ |
| s | A | A | B | B | B | A |

| | A | B | C | D | E | F |
|---|---|---|---|---|---|---|
| d | 0 | 2 | 3 | 6 | 4 | 6 |
| s | A | A | B | B | B | E |

Figura 68

O caminho mínimo entre o nó A e o nó F tem custo 6.

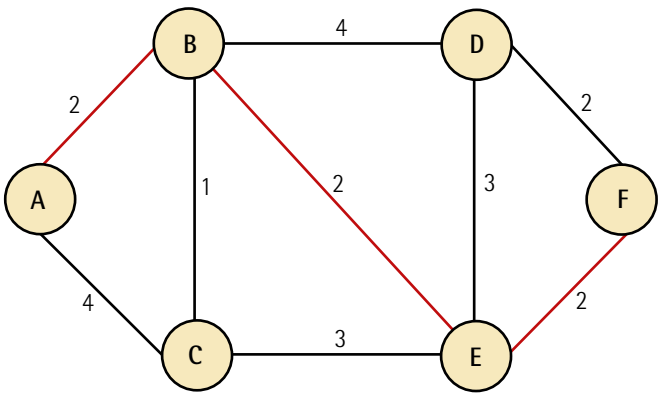



Figura 69

## 6.2 Caminho mínimo entre todos os pares

O caminho mínimo entre todos os pares (também conhecido como algoritmo de Floyd-Warshall) é um algoritmo clássico em teoria dos grafos que encontra o caminho mínimo entre todos os pares de nós em um grafo ponderado.

O algoritmo funciona construindo uma matriz de distâncias, na qual o valor da entrada (i, j) representa o caminho mínimo entre os nós i e j. A matriz é inicializada com as distâncias diretas entre os nós, e então o algoritmo itera sobre a matriz, atualizando as distâncias sempre que um caminho mais curto é encontrado através de outro nó.

O algoritmo é eficiente e tem complexidade de tempo O(n^3), sendo n o número de nós no grafo. Ele é capaz de lidar com grafos com arestas negativas, mas não é capaz de lidar com ciclos negativos. Um ciclo negativo é um ciclo no grafo cuja soma das arestas é negativa, o que torna impossível determinar o caminho mínimo entre os nós do ciclo.

**Saiba mais**

Com o objetivo de saber mais sobre complexidade de algoritmos, leia os seguintes capítulos da obra indicada: 1.4 Complexidade de algoritmos; 1.5 A notação O; 1.6 Algoritmos ótimos.

SZWARCFITER, J. L.; MARKENZON, L. *Estruturas de dados e seus algoritmos*. 3. ed. Rio de Janeiro: LTC, 2010.

Em resumo, o caminho mínimo entre todos os pares é uma ferramenta útil na teoria dos grafos para encontrar o caminho mínimo entre todos os pares de nós em um grafo ponderado. É um algoritmo eficiente e capaz de lidar com arestas negativas, mas é preciso ter cuidado com ciclos negativos. Ele é amplamente utilizado em diversas áreas, desde engenharia de transporte até análise de redes complexas.

Observe a seguir o algoritmo caminho mínimo entre todos os pares:

```
CaminhoMinimoEntreTodosOsPares (matriz n x n A): //------------O(n³)-----------
// Algoritmo de Floyd – calcula o Caminho Mínimo entre todos pares;
inicialmente, A é a matriz;
// de adjacência; ao final, A vai conter todas as distâncias dos Caminhos Mínimos.
        Para k = 1, até n, faça:
                Para = 1, até n, faça:
                        Para j = 1, até n, faça:
                                Se A[i,k] + A[k,j] < A[i,j], então:
                                        A[i,j]= A[i,k] + A[k,j]
                                Fim do se.
                        Fim do para.
                Fim do para.
        Fim do para.
Fim de CaminhoMinimoEntreTodosOsPares.
```

## 7 PROBLEMA DAS QUATRO CORES

Este famoso problema em teoria dos grafos consiste em determinar se é possível colorir os vértices de qualquer mapa planar com no máximo quatro cores, de forma que nenhum par de vértices adjacentes tenha a mesma cor.

O problema foi proposto inicialmente em 1852 por Francis Guthrie, que se perguntou se era possível colorir o mapa da região inglesa de counties (condados) usando apenas quatro cores. Apesar de parecer uma questão simples, o problema acabou se mostrando extremamente difícil de ser resolvido e levou mais de um século para ser solucionado.

Em 1976, Kenneth Appel e Wolfgang Haken finalmente demonstraram que é possível colorir qualquer mapa planar usando no máximo quatro cores. A prova envolveu o uso de computadores para analisar todos os possíveis casos, o que foi considerado controverso por alguns matemáticos.

Desde então, várias outras demonstrações do teorema foram propostas, algumas das quais usando técnicas mais abstratas, como a teoria dos grafos e a teoria das cores. Apesar disso, o problema das quatro cores ainda é considerado um dos mais desafiadores em matemática, e sua solução envolve uma grande quantidade de teorias e técnicas avançadas.

Esse teorema tem aplicações práticas em áreas como cartografia e design de mapas. Ele permite que mapas de regiões e países possam ser criados de forma mais eficiente e com menos erros, garantindo que as cores usadas sejam suficientes para representar todas as regiões, sem que haja cores repetidas em regiões adjacentes.

Além disso, o problema das quatro cores é um exemplo clássico de um problema de otimização combinatória, que busca encontrar a melhor solução para um problema dentro de um conjunto de possibilidades. Ele pode ser formulado como um problema de programação linear inteira, no qual cada cor é representada por uma variável binária, e a restrição é que cada par de vértices adjacentes não pode ter a mesma cor.

Ele também tem sido usado como uma fonte de inspiração para outros problemas de coloração, como o problema de coloração de grafos, que envolve a atribuição de cores aos vértices de um grafo de forma que vértices adjacentes tenham cores diferentes. A solução desse problema tem aplicações em áreas como redes de computadores, processamento de imagens e análise de dados.

Esse problema é uma questão de coloração de vértices, em que se deseja colorir os vértices de um grafo de modo que nenhum par de vértices adjacentes tenha a mesma cor e usando no máximo quatro cores.

Existem diversas abordagens para resolver esse problema, e uma delas é o algoritmo de Welsh-Powell:

1. Ordenar os vértices do grafo em ordem decrescente de grau.

2. Começar pelo primeiro vértice da lista ordenada e atribuir a ele a cor 1.

3. Para cada vértice seguinte na lista, atribuir a ele a menor cor possível que não tenha sido usada por seus vértices adjacentes.

4. Repetir o passo 3 até que todos os vértices tenham sido coloridos.

Esse algoritmo é razoavelmente eficiente e, em muitos casos, é capaz de encontrar uma solução ótima usando apenas quatro cores. No entanto, não há garantia de que ele sempre encontrará a solução ótima e existem casos em que ele pode falhar.

Outras abordagens mais complexas, como o algoritmo de coloração de grafos baseado em programação linear inteira, também podem ser usadas para resolver o problema das quatro cores de forma mais precisa. No entanto, esses métodos são mais complexos e podem ser mais lentos para serem executados.

Além disso, existem heurísticas e algoritmos que podem ser utilizados para resolver o problema das quatro cores, como o DSatur, o LDO e o algoritmo genético.

O DSatur é uma heurística que começa com um vértice qualquer colorido com a cor 1 e, em seguida, escolhe o vértice com o maior grau e com o maior número de cores diferentes em seus vizinhos para ser colorido com a menor cor possível. Esse processo é repetido até que todos os vértices estejam coloridos.

O LDO (Large Degree Order) é uma heurística que ordena os vértices do grafo em ordem decrescente de grau e, em seguida, escolhe cada vértice na ordem e atribui a ele a menor cor possível que não tenha sido usada por seus vértices adjacentes.

Por sua vez, o algoritmo genético é um método de otimização pautado em processos biológicos de evolução. Ele funciona criando uma população de soluções candidatas e aplicando operadores genéticos, como cruzamento e mutação, para gerar novas soluções. As soluções são avaliadas com base em uma função de aptidão que mede o quão bem elas resolvem o problema das quatro cores. As soluções mais aptas são selecionadas para a próxima geração, enquanto as menos aptas são descartadas. Esse processo é repetido até que uma solução adequada seja encontrada.

Associado a qualquer mapa existe um grafo, chamado de grafo dual do mapa. Cada região do mapa corresponde a um vértice no grafo, e um arco entre dois nós devem estar associados a cada duas regiões adjacentes.

Agora citemos um exemplo. Na figura ilustrada a seguir, pode-se observar que o mapa M tem como dual o grafo completo K4.

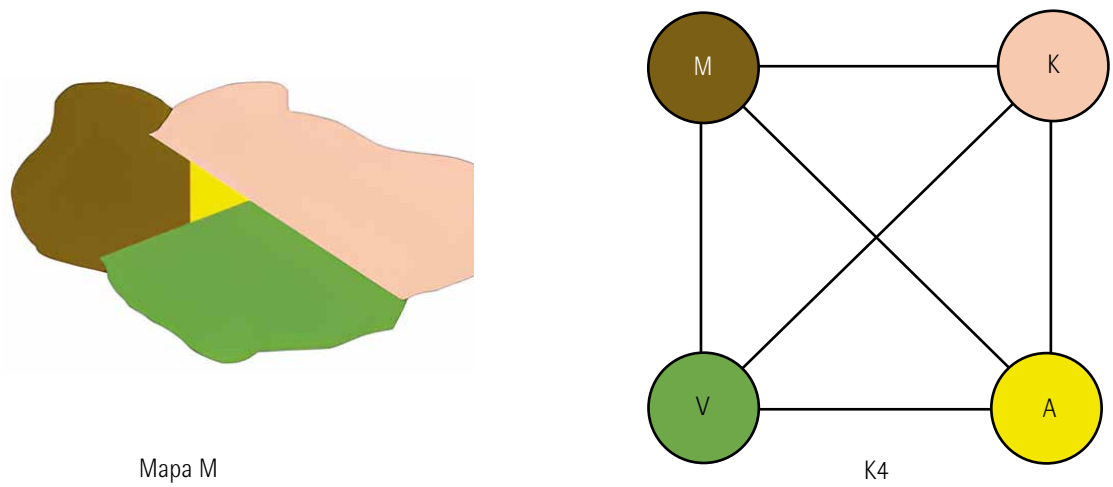



Figura 70 – Mapa dual do grafo K4

Todo mapa pode (plano ou esférico) pode ser pintado com quatro cores, de modo que duas regiões tendo uma fronteira comum não fiquem com a mesma cor?

O número cromático de um grafo é o menor número de cores necessário para se colorir um grafo.

O número cromático do grafo K4 é 4.

**Lembrete**

O problema das quatro cores pode ser resolvido usando uma variedade de métodos, desde heurísticas simples até algoritmos mais complexos. A escolha do método dependerá do tamanho e da complexidade do grafo, bem como das restrições de tempo e recursos disponíveis.

## 8 FLUXOS EM GRAFOS

Trata-se de um conceito importante em teoria dos grafos que representa a quantidade de fluxo que pode ser transportada em um grafo com arestas ponderadas. Essa teoria tem diversas aplicações em áreas como engenharia de redes, transporte e programação linear.

Formalmente, um fluxo em um grafo direcionado ponderado é uma atribuição de uma quantidade não negativa de fluxo a cada aresta que respeita a capacidade máxima de cada aresta e conservação do fluxo em cada nó. A capacidade máxima de uma aresta expressa a quantidade máxima de fluxo que pode ser transportada através dela. A conservação do fluxo em cada nó exige que a quantidade de fluxo que entra em um nó deve ser igual à quantidade de fluxo que sai dele, exceto em nós especiais, conhecidos como fontes e sumidouros, nos quais o fluxo pode entrar ou sair livremente.

O problema do fluxo máximo em um grafo é encontrar o fluxo máximo que pode ser transportado entre uma fonte e um sumidouro em um grafo com arestas ponderadas. Esse problema pode ser resolvido usando o algoritmo de Ford-Fulkerson, que utiliza o conceito de caminhos aumentantes para elevar iterativamente o fluxo até que não seja mais possível encontrar caminhos adicionais.

Outra importante aplicação dos fluxos em grafos é no problema de corte mínimo, que consiste em encontrar um conjunto mínimo de arestas cuja remoção desconecta a fonte do sumidouro no grafo. Esse problema pode ser resolvido utilizando o teorema do fluxo máximo e corte mínimo, que estabelece que o fluxo máximo é igual ao corte mínimo em um grafo.

Em resumo, o conceito de fluxos em grafos é fundamental na teoria dos grafos, permitindo a modelagem e solução de problemas relacionados ao transporte de fluxo em redes complexas. O problema do fluxo máximo e do corte mínimo são exemplos de problemas importantes que podem ser resolvidos utilizando fluxos em grafos.

## 8.1 Problema do fluxo máximo

Em teoria dos grafos, ele é um dos problemas clássicos e fundamentais em otimização combinatória, que consiste em determinar o fluxo máximo que pode ser transportado em uma rede de fluxo com capacidades nas arestas.

Formalmente, um grafo direcionado ponderado é considerado uma rede de fluxo se houver uma fonte, um sumidouro e uma capacidade não negativa atribuída a cada aresta. A fonte é um nó especial do qual o fluxo é gerado e o sumidouro é um nó especial que consome o fluxo. O objetivo do problema do fluxo máximo é encontrar a quantidade máxima de fluxo que pode ser transportada da fonte para o sumidouro, respeitando as capacidades das arestas e a conservação do fluxo em cada nó intermediário.

O algoritmo clássico para resolver o problema do fluxo máximo é o de Ford-Fulkerson. Este utiliza o conceito de caminhos aumentantes para elevar o fluxo em cada iteração. Um caminho aumentante é um caminho de acréscimo no grafo residual que é obtido após cada iteração do algoritmo e representa um caminho no qual o fluxo pode ser expandido sem exceder as capacidades das arestas.

O algoritmo de Ford-Fulkerson é implementado em duas fases. Na primeira, é construído um fluxo inicial válido, que pode ser qualquer fluxo que respeite as capacidades das arestas e a conservação do fluxo em cada nó intermediário. Na segunda, o algoritmo encontra um caminho aumentante no grafo residual e eleva o fluxo ao longo desse caminho, repetindo esta fase até que não seja mais possível encontrar caminhos aumentantes.

O teorema do fluxo máximo e corte mínimo estabelece que o valor do fluxo máximo é igual ao valor do corte mínimo na rede de fluxo. Um corte na rede de fluxo é um conjunto de arestas que, se removido, desconecta a fonte do sumidouro. O valor de um corte é definido como a soma das capacidades das arestas que compõem o corte. Portanto, o problema do corte mínimo pode ser resolvido utilizando o teorema do fluxo máximo e corte mínimo.

**Lembrete**

Em resumo, o problema do fluxo máximo é importante em teoria dos grafos, com aplicações em diversas áreas, incluindo engenharia de redes, logística, transporte e telecomunicação. Já o algoritmo de Ford-Fulkerson é o algoritmo clássico para resolver o problema do fluxo máximo, e o teorema do fluxo máximo e corte mínimo é uma ferramenta essencial na análise de redes de fluxo.

## 8.2 Teorema de Ford-Fulkerson

É um dos resultados mais importantes em teoria dos grafos, que estabelece que o fluxo máximo em uma rede de fluxo é igual à capacidade mínima de um corte na rede de fluxo.

Formalmente, uma rede de fluxo é um grafo direcionado ponderado em que cada aresta tem uma capacidade não negativa e há uma fonte e um sumidouro. Um fluxo é uma função que atribui uma quantidade não negativa de fluxo a cada aresta, respeitando as capacidades das arestas e a conservação do fluxo em cada nó intermediário, ou seja, a soma do fluxo que entra em um nó é igual à soma do fluxo que sai do nó, exceto na fonte e no sumidouro.

Um corte na rede de fluxo é um conjunto de arestas que, se removido, desconecta a fonte do sumidouro. O valor de um corte é definido como a soma das capacidades das arestas que compõem o corte. O corte mínimo é o corte de valor mínimo na rede de fluxo.

O teorema de Ford-Fulkerson afirma que o valor do fluxo máximo na rede de fluxo é igual ao valor do corte mínimo na rede de fluxo. Em outras palavras, se f* é o fluxo máximo e c* é o corte mínimo, então f* = c*.

Ele é importante porque permite a construção de algoritmos para resolver o problema do fluxo máximo utilizando o problema do corte mínimo. O algoritmo clássico para resolver o problema do fluxo máximo é o de Ford-Fulkerson, que usa o conceito de caminhos aumentantes para elevar o fluxo em cada iteração.

Esse teorema tem implicações relevantes em teoria dos grafos e em outras áreas, como programação linear e teoria da complexidade. Além disso, o teorema é fundamental para a compreensão e o desenvolvimento de outros algoritmos em teoria dos grafos, como o algoritmo de Dinic e o de Edmonds-Karp.

Em resumo, o teorema de Ford-Fulkerson é um resultado vital em teoria dos grafos que estabelece a relação entre o fluxo máximo e o corte mínimo em uma rede de fluxo. O teorema é importante para a construção de algoritmos para resolver o problema do fluxo máximo e tem implicações em outras áreas da matemática e da ciência da computação.

O algoritmo de Ford-Fulkerson é um algoritmo clássico para encontrar o fluxo máximo em uma rede de fluxo. Seu pseudocódigo básico é o seguinte:

1. Inicialize o fluxo F em todas as arestas para 0.

2. Enquanto existir um caminho de aumento P na rede residual:

    a. Encontre a capacidade mínima residual ao longo de P (chamada de f).

    b. Atualize o fluxo F e a capacidade residual das arestas ao longo de P.

3. Retorne o fluxo F.

O algoritmo de Ford-Fulkerson usa um método guloso para encontrar caminhos de aumento na rede residual. Um caminho de aumento é um caminho da fonte ao sorvedouro no qual cada aresta tem

uma capacidade residual maior que zero. O algoritmo encontra o caminho de aumento com a menor capacidade residual ao longo dele e, em seguida, atualiza o fluxo e as capacidades residuais das arestas.

O algoritmo pode ser implementado usando estruturas de dados como lista de adjacências e fila para armazenar os caminhos de aumento. O tempo de execução do algoritmo depende da maneira como o caminho de aumento é encontrado. Se usado o algoritmo de busca em largura (BFS) para encontrar o caminho de aumento, o tempo de execução é O(E^2), sendo E o número de arestas da rede. No entanto, se usado o algoritmo de busca em profundidade (DFS), o tempo de execução pode ser reduzido para O(E * f), sendo f o fluxo máximo.

A lista de adjacências é uma estrutura de dados usada para representar grafos. Ela consiste em uma lista de vértices, sendo que cada vértice possui uma lista de seus vizinhos diretos, ou seja, os vértices adjacentes a ele. Essa representação é útil para grafos esparsos, aqueles que possuem poucas arestas em relação ao número de vértices. Ela permite uma implementação eficiente de algoritmos que percorrem o grafo a partir de um vértice, como a busca em profundidade (DFS) ou a busca em largura (BFS).

Já a fila é uma estrutura de dados que permite o armazenamento de elementos em uma ordem específica. Ela opera com o princípio FIFO (first in, first out), e o primeiro elemento adicionado é o primeiro a ser removido. As filas são úteis para implementar algoritmos que precisam processar elementos em uma ordem específica, como BFS, nos quais os vértices são visitados em ordem de distância em relação à origem. Na busca em largura, a fila é usada para armazenar os vértices a serem visitados, sendo o primeiro vértice da fila aquele que será visitado na próxima etapa do algoritmo.

**Observação**

A lista de adjacências e a fila são estruturas de dados comuns usadas na implementação de algoritmos para grafos, como o de Ford-Fulkerson. A lista de adjacências é usada para armazenar as informações do grafo, enquanto a fila é usada para armazenar os vértices a serem processados em uma ordem específica, conforme o algoritmo.

## 8.3 Grafo de folgas

Trata-se de uma técnica em teoria dos grafos que é frequentemente usada para resolver problemas de otimização linear em redes de fluxo. Ele é uma ferramenta útil para converter um problema de otimização linear em um problema de fluxo máximo, o que pode facilitar a resolução do problema.

Para construir o grafo de folgas, considere uma rede de fluxo com capacidades nas arestas e um conjunto de demandas nos nós. A ideia é adicionar arestas e pesos ao grafo original para criar um novo grafo que tenha as mesmas soluções ótimas que o problema original, mas que possa ser transformado em um problema de fluxo máximo.

O grafo de folgas é construído adicionando um nó fonte s' e um nó sumidouro t' ao grafo original, com arestas direcionadas que conectam s' a todos os nós do grafo original que têm demanda positiva e todos os nós do grafo original que têm oferta positiva a t'. A capacidade de cada aresta é igual à demanda ou oferta do nó correspondente.

Para cada aresta (i, j) no grafo original, adicionamos uma nova aresta (i, j') ao grafo de folgas com capacidade igual à capacidade da aresta original. Além disso, adicionamos uma nova aresta (i', j) com capacidade infinita. O peso de cada aresta é igual à diferença entre o custo original e o custo da aresta (i, j').

Uma vez que o grafo de folgas foi construído, podemos encontrar o fluxo máximo nesse grafo usando o algoritmo de Ford-Fulkerson. A solução do problema original pode ser recuperada a partir do fluxo máximo no grafo de folgas.

O grafo de folgas é uma técnica poderosa que é amplamente utilizada em otimização linear e programação inteira. Ele permite que problemas complexos sejam transformados em problemas de fluxo máximo, o que pode facilitar sua resolução. Além disso, o grafo de folgas é uma ferramenta útil para visualizar e entender as soluções de problemas de otimização linear em redes de fluxo.

No entanto, é importante notar que a construção do grafo de folgas pode levar a um aumento significativo no número de nós e arestas no grafo. Isso pode tornar o problema mais difícil de resolver, especialmente para problemas maiores. A construção do grafo de folgas pode levar a uma perda de informação, já que a solução do problema original é recuperada a partir do fluxo máximo no grafo de folgas, que pode não ser exatamente o mesmo que o fluxo máximo no grafo original.

Apesar dessas limitações, o grafo de folgas é uma ferramenta valiosa que é amplamente utilizada em pesquisa operacional, engenharia de sistemas e áreas correlatas. Ele permite que os problemas de otimização linear sejam resolvidos de forma eficiente e precisa, o que é essencial para a tomada de decisões informadas em uma ampla gama de situações.

## 8.4 Fluxos com custos

Em teoria dos grafos, eles são uma extensão dos problemas de fluxo máximo, em que cada aresta tem um custo associado a ela. O objetivo é maximizar o fluxo total ao mesmo tempo que se minimiza o custo total das arestas usadas.

Mais formalmente, dado um grafo direcionado G com capacidades cij e custos hij para cada aresta (i, j), um fluxo fij é uma função que associa a cada aresta (i, j) uma quantidade de fluxo fij, com a restrição de que o fluxo total que entra e sai de cada nó é conservado. O objetivo é encontrar um fluxo máximo f* que minimize o custo total das arestas utilizadas.

Existem várias maneiras de resolver o problema de fluxo com custo. Uma abordagem comum é usar o algoritmo de ciclo mais curto, que encontra um caminho de custo mínimo no grafo residual e aumenta

o fluxo ao longo desse caminho até atingir um fluxo máximo. Esse processo é repetido até que nenhum caminho de custo negativo possa ser encontrado no grafo residual.

Outra abordagem é usar o algoritmo de fluxo de custo mínimo, que é uma variação do algoritmo de Ford-Fulkerson que usa o custo da aresta como uma medida de distância para guiar a busca pelo caminho aumentador.

Os problemas de fluxo com custos são amplamente utilizados em problemas de otimização de transporte, como a determinação de rotas de transporte de mercadorias entre diferentes locais, bem como em problemas de alocação de recursos e planejamento de produção. Eles também são usados em problemas de roteamento de rede, como a determinação de rotas em redes de comunicação e transporte.

Uma aplicação importante dos fluxos com custos é o problema de fluxo de custo mínimo em um grafo com demandas. Nele, cada nó do grafo é rotulado com uma demanda, que pode ser positiva ou negativa. O objetivo é encontrar um fluxo de custo mínimo que satisfaça todas as demandas.

Existem várias técnicas para resolver o problema de fluxo de custo mínimo em um grafo com demandas. Uma delas é a redução de custo de ciclo, que envolve a adição de arestas artificiais para transformar o grafo em um grafo acíclico. Outra técnica é a decomposição de capacidades, que engloba a redução do problema a um conjunto de problemas de fluxo de custo mínimo em grafos menores.

Os problemas de fluxo com custo também são usados em problemas de roteamento de veículos, como a determinação de rotas para um conjunto de veículos para atender um conjunto de clientes. Nesse caso, cada veículo tem uma capacidade limitada e cada cliente tem uma demanda a ser atendida. O objetivo é encontrar um conjunto de rotas para os veículos que atenda todas as demandas com o menor custo possível.

Além disso, os problemas de fluxo com custos são amplamente estudados em teoria dos jogos e economia, em que as arestas podem representar fluxos de dinheiro ou informações entre os jogadores ou agentes econômicos. O problema de fluxo máximo com custos pode ser usado para modelar a distribuição de recursos em uma economia ou o fluxo de informações em uma rede de comunicação.

## Resumo

Vimos nesta unidade a definição de árvore mínima (ou árvore geradora mínima), uma árvore que conecta todos os vértices de um grafo ponderado com o menor custo total possível. Ela é útil em muitas aplicações, como em redes de computadores, em que é desejável conectar todos os computadores com o menor custo possível de cabos de rede. Existem vários algoritmos para encontrar a árvore mínima de um grafo, sendo os mais comuns o de Kruskal e o de Prim. Ambos são algoritmos "gulosos" (ou vorazes) que tomam a decisão ótima local em cada passo na esperança de encontrar a solução ótima global.

Em seguida, destacamos que o algoritmo de Kruskal começa com um conjunto vazio de arestas e adiciona as arestas de menor peso em ordem crescente até que todos os vértices estejam conectados. Ele usa uma estrutura de dados para detectar ciclos no grafo e evitar a adição de arestas que formam ciclos. Por sua vez, o algoritmo de Prim começa com um vértice arbitrário e adiciona as arestas de menor peso que conectam o vértice atual a um vértice ainda não visitado, criando um subconjunto de vértices conectados. Ele usa uma estrutura de dados chamada heap (ou fila de prioridade) para escolher a próxima aresta de menor peso. Ambos os algoritmos têm tempo de execução O(E log E), sendo E o número de arestas do grafo. Geralmente, o algoritmo de Prim é mais rápido em grafos densos (com muitas arestas), e o algoritmo de Kruskal é mais rápido em grafos esparsos (com poucas arestas).

Acentuamos que caminhos mínimos são uma classe de problemas em grafos cujo objetivo é encontrar o caminho com o menor custo entre dois vértices em um grafo ponderado. Existem vários algoritmos para resolver esse problema, sendo os mais comuns o de Dijkstra e o de Bellman-Ford. Dijkstra é usado para encontrar o caminho mínimo de um vértice de origem a todos os outros vértices em um grafo ponderado não negativo. Ele usa uma estrutura de dados chamada heap (ou fila de prioridade) para escolher o vértice de menor distância não visitado em cada passo. Já o de Bellman-Ford é usado para encontrar o caminho mínimo de um vértice de origem a todos os outros vértices em um grafo ponderado que pode ter arestas com pesos negativos. Ele atualiza as distâncias dos vértices várias vezes, relaxando as arestas até que o caminho mínimo seja encontrado ou um ciclo negativo seja detectado.

Estudamos também o problema das quatro cores, que é um problema clássico da teoria dos grafos que pergunta se é possível colorir os vértices

de qualquer mapa plano com apenas quatro cores, de modo que vértices adjacentes tenham cores diferentes. O problema foi proposto pela primeira vez em 1852 pelo matemático britânico Francis Guthrie, que notou que era possível colorir um mapa de condados com apenas quatro cores. Embora a questão seja fácil de entender, provar que é possível colorir todos os mapas planares com apenas quatro cores é um problema matemático difícil e levou mais de um século para ser resolvido. O primeiro resultado importante sobre o problema foi a prova de que é possível colorir todos os mapas planares com até 25 regiões com apenas quatro cores, o que foi demonstrado por Kempe em 1879. No entanto, a questão completa só foi resolvida em 1976 por Appel e Haken, que desenvolveram um algoritmo computacional para provar que qualquer mapa plano pode ser colorido com quatro cores. A prova foi controversa, pois dependia de muitos cálculos complexos e detalhados e levou anos para ser verificada e aceita pela comunidade matemática. Hoje em dia, o problema das quatro cores é considerado como um dos problemas clássicos da matemática e é estudado em teoria dos grafos e computação gráfica. Além disso, as técnicas desenvolvidas para resolver o problema têm aplicações em outras áreas da matemática e da ciência da computação.

Outro assunto importante abordado nesta unidade foi o fluxo em grafos, que é uma ideia central em teoria dos grafos e é usado em diversas aplicações, como transporte de fluidos, redes de computadores e distribuição de energia. Ele consiste em modelar o fluxo de uma quantidade através de um grafo, no qual os vértices representam pontos de origem e destino, e as arestas expressam canais de transporte. Em um grafo de fluxo, cada aresta é associada com uma capacidade, que representa a quantidade máxima de fluxo que pode passar por essa aresta. Além disso, cada aresta tem um fluxo, que representa a quantidade de fluxo que está passando por essa aresta no momento. O objetivo é encontrar um fluxo máximo, ou seja, o fluxo total máximo que pode ser enviado de um vértice de origem a um vértice de destino, respeitando as capacidades das arestas e a condição de conservação de fluxo em cada vértice intermediário. Existem vários algoritmos para encontrar o fluxo máximo em um grafo, sendo o mais conhecido o algoritmo de Ford-Fulkerson, que usa o método de caminho de aumento para encontrar o fluxo máximo.

## Exercícios

**Questão 1**. (UFMG 2019, adaptada) O famoso algoritmo de Dijkstra soluciona um problema de grafos direcionados e não direcionados com certa complexidade. A respeito desse algoritmo, avalie as afirmativas a seguir.

I – Trata-se de um algoritmo que resolve o problema de encontrar o caminho com menor número de arestas entre um vértice e outro.

II – Só pode ser utilizado em grafos ponderados cujas arestas não são negativas.

III – Tem complexidade O(V!), em que V é o número de vértices no grafo.

É correto o que se afirma em

A) I, apenas.

B) II, apenas.

C) III, apenas.

D) II e III, apenas.

E) I, II e III.

Resposta correta: alternativa B.

**Análise das afirmativas**

I – Afirmativa incorreta.

Justificativa: o algoritmo de Dijkstra é um algoritmo de caminho mínimo de fonte única e que resolve o problema de encontrar o caminho mais curto em um grafo ponderado com arestas não negativas. O objetivo do algoritmo é determinar o caminho mais curto a partir de um nó de origem para todos os outros vértices do grafo considerando os pesos das arestas. Seu objetivo não é diminuir o número de arestas ao longo do caminho, mas encontrar o caminho cujo somatório dos pesos é menor.

II – Afirmativa correta.

Justificativa: o algoritmo de Dijkstra trabalha com grafos ponderados, direcionados ou não, que não têm arestas negativas.

III – Afirmativa incorreta.

Justificativa: o algoritmo de Dijkstra tem um tempo de execução de O(E + V log V), em que E é o número de arestas e V é o número de nós no grafo.

**Questão 2**. (Enade 2021) O algoritmo de Dijkstra para o problema do caminho mínimo em dígrafos com pesos utiliza uma fila de prioridades de vértices em que as prioridades são uma estimativa do custo final. A cada iteração, um vértice é retirado da fila, e os arcos que começam nesse vértice são analisados. Considere o seguinte grafo, no qual se deseja conhecer o custo de um caminho mínimo para cada vértice a partir do vértice D. Considere que -1 representa um custo "infinito", ou seja, nenhum caminho até o vértice foi até o momento foi descoberto.

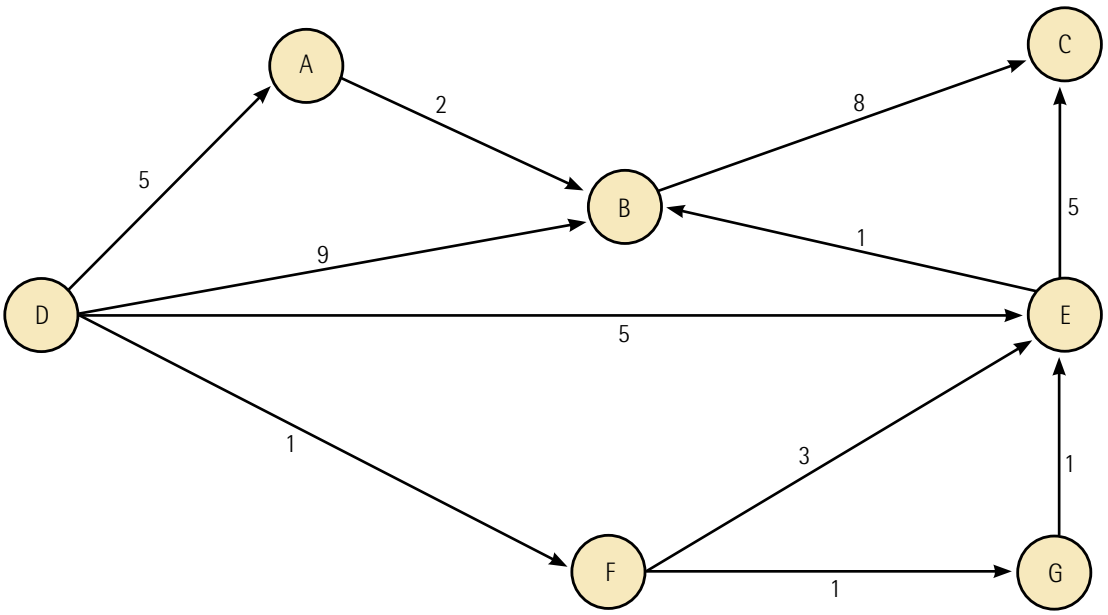



Figura 71

Com base nas informações e no grafo ilustrado, assinale a alternativa que representa a estimativa de custo após duas iterações do algoritmo.

A) A: 5 B: 6 C: 10 D: 0 E: 4 F: 1 G: -1

B) A: 5 B: 9 C: -1 D: 0 E: 5 F: 1 G: -1

C) A: 5 B: 9 C: -1 D: 0 E: 4 F: 1 G: 2

D) A: 5 B: 7 C: 8 D: 0 E: 4 F: 1 G: 2

E) A: 5 B: 6 C: 8 D: 0 E: 3 F: 1 G: 2

Resposta correta: alternativa C.

**Análise da questão**

O algoritmo de Dijkstra começa selecionando um nó inicial e definindo sua distância como zero. Em seguida, o algoritmo avalia todos os nós adjacentes a este nó inicial calculando suas distâncias até o nó inicial. Se a distância do nó adjacente é menor do que a distância atual armazenada, a distância é atualizada. Esse processo é repetido para todos os nós adjacentes até que todos os nós sejam visitados.

Nesta questão, precisamos calcular um caminho mínimo, em termos de distância (ou custo) de arestas, do nó D para os outros nós. De acordo com o enunciado, faremos apenas duas iterações do algoritmo.

O algoritmo de Dijkstra prevê um conjunto de nós IN em que, inicialmente, anotamos o nó de origem (s, de source) igual a D. De D para D, temos distância (d) igual a 0. Observando os pesos do grafo, temos distância 5 de D para A, distância 9 de D para B, e assim sucessivamente. Os nós não adjacentes a D (que são os nós C e G) recebem, inicialmente, o valor ∞, que representa custo infinito. Esses valores, relativos à primeira iteração, são indicados a seguir.

**Tabela 1 – Primeira iteração: IN = {D}**

| | A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|---|
| **d** | 5 | 9 | ∞ | 0 | 5 | 1 | ∞ |
| **s** | D | D | D | D | D | D | D |

Em seguida, elegemos um próximo nó para o caminho, que será aquele que apresenta menor distância não negativa (no caso, o nó F). Vamos, então, para a segunda iteração. O nó F passa a integrar o conjunto IN. Testaremos, em sequência, todos os nós não pertencentes a IN. Se o somatório entre d e a distância do nó a F é menor do que a distância atual armazenada, a distância é atualizada, assim como a origem, que passará a ser F.

Segunda iteração: IN={D,F}

d=1

d[A]=min(5,1+D[F,A])=min(5,1+∞)=min(5,∞)=5

d[B]=min(9,1+D[F,B])=min(9,1+∞)=min(9,∞)=9

d[C]=min(∞,1+D[F,C])=min(∞,1+∞)=min(∞,∞)=∞

d[E]=min(5,1+D[F,E])=min(5,1+3)=min(5,4)=4

d[G]=min(∞,1+D[F,G])=min(∞,1+1)=min(∞,2)=2

Apenas os nós E e G tiveram seus valores atualizados, com a redução da distância ao passar pelo nó F. A tabela atualizada é demonstrada a seguir, relativa à segunda iteração do algoritmo.

**Tabela 2**

| | A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|---|
| **d** | 5 | 9 | ∞ | 0 | 4 | 1 | 2 |
| **s** | D | D | D | D | F | D | F |

O enunciado da questão pede que consideremos o valor -1 como custo infinito. Vamos, então, trocar o símbolo ∞ pelo valor -1 e, assim, teremos a estimativa de custo após duas iterações do algoritmo, conforme acentuado a seguir:

A:5 B:9 C:-1 D:0 E:4 F:1 G:2.

## REFERÊNCIAS

### Textuais


BOAVENTURA NETTO, P. O.; JURKIEWICZ, S. *Grafos*: introdução e prática São Paulo: Blucher, 2009.

CALAÇA, O. Uma introdução às redes neurais para grafos (GNN). *Medium*, 2020. Disponível em: https://bit.ly/3oxL8Od. Acesso: 24 abr. 2023.

CORMEN, T. H. *et al. Algoritmos*: teoria e prática. Rio de Janeiro: Elsevier, 2012.

GERSTING, J. L. *Fundamentos matemáticos para a ciência da computação*. Matemática discreta e suas aplicações. 7. ed. Rio de Janeiro: LTC, 2014.

GOLDBARG, M.; GOLDBARG, E. *Grafos*: conceitos, algoritmos e aplicações. Rio de Janeiro: Elsevier, 2012.

GÖTTLICH, S.; TOTZECK, C. Parameter calibration with stochastic gradient descent for interacting particle systems driven by neural networks. *Mathematics of Control, Signals, and Systems*, 2021. Disponível em: https://bit.ly/40y154f. Acesso em: 24 abr. 2023.

HAMILTON, W. L.; YING, R.; LESKOVEC, J. Inductive representation learning on large graphs. *NIPS*, Califórnia, 2017.

HAYKIN, S. *Redes neurais*: princípios e prática. Porto Alegre: Bookman, 2007.

LESKOVEC, J. *CS224W*: Machine learning with graphs. Califórnia: Universidade de Stanford, 2022.

LIPSCHUTZ, S.; LIPSON, M. *Matemática discreta*: Coleção Schaum. 3. ed. Porto Alegre: Bookman, 2013.

MUCHALSKI, F.; MAZIERO, C. A. Alocação de máquinas virtuais em ambientes de computação em nuvem considerando o compartilhamento de memória. *In:* WORKSHOP DE COMPUTAÇÃO EM CLOUDS E APLICAÇÕES – WCGA/SBRC, 12., 2014, Florianópolis, SC. *Anais* [...]. Florianópolis, 2014.

NICOLETTI, M.; HRUSCHKA JÚNIOR, E. R. *Fundamentos da teoria dos grafos para computação*. 3. ed. Rio de Janeiro: LTC, 2018.

NORVIG, P. *Inteligência artificial*. 3. ed. Rio de Janeiro: LTC, 2013.

PRESSMAN, R. S.; MAXIM, B. R. *Engenharia de software*: uma abordagem profissional. 8. ed. Porto Alegre: AMGH, 2016.

SIMÕES-PEREIRA, J. *Grafos e redes: teoria e algoritmos básicos*. Rio de Janeiro: Interciência, 2014.

SZWARCFITER, J. L.; MARKENZON, L. *Estruturas de dados e seus algoritmos*. 3. ed. Rio de Janeiro: LTC, 2010.

SZWARCFITER, J. L.; PINTO, P. E. D. *Teoria computacional de grafos*: os algoritmos com programas Python. Rio de Janeiro: Elsevier, 2018.


Informações:
www.sepi.unip.br ou 0800 010 9000