Apontamentos de Estruturas de Dados

Grafos

Ricardo Santos| rjs@estg.ipp.pt

Escola Superior de Tecnologia e Gestão
Instituto Politécnico do Porto

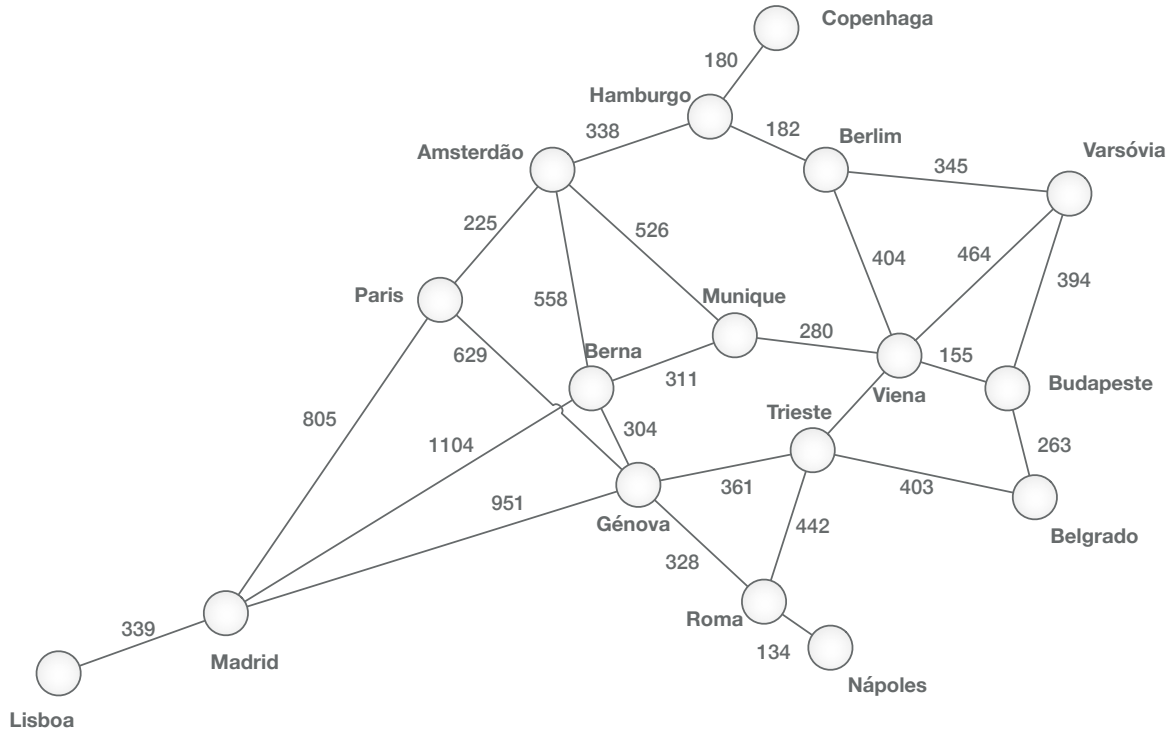
Última versão: Dezembro de 2022

Índice

15. Grafos

15.1. Introdução

Muitas vezes útil para representar informação de uma forma gráfica mais geral do que a considerada até agora, como a seguinte representação das distâncias entre cidades:



Com estruturas semelhantes (talvez deixando de fora as distâncias, ou substituindo-as por outra coisa), poderíamos representar muitas outras situações, como uma rede de túneis subterrâneos, ou uma rede de tubos (onde a etiqueta numérica pode dar os diâmetros dos tubos), ou um mapa ferroviário, ou uma indicação de quais cidades estão ligadas por voos, navios ou mesmo alianças políticas. Mesmo se assumirmos que é uma rede de caminhos ou estradas, os números não precisam necessariamente representar distâncias, podem simplesmente ser uma indicação de quanto tempo demora para percorrer a distância em questão, portanto, uma determinada distância até uma montanha íngreme demoraria mais tempo do que num terreno plano se fosse percorrida a pé por exemplo.

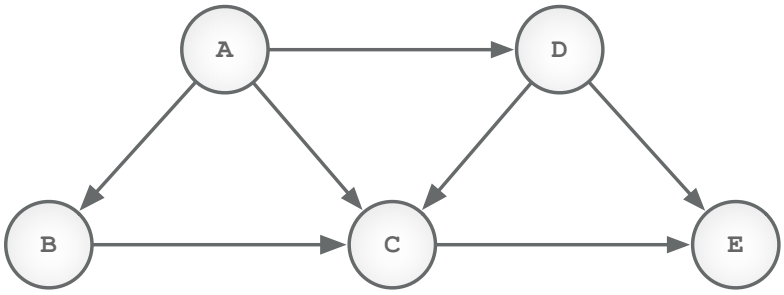
A utilização de grafos permite fazer muito mais do que apenas saber quais as localidades que estão interligadas como demonstra a figura anterior. Podemos, por exemplo, questionar se existe uma maneira de ir de A para B, ou qual é o caminho mais curto, ou qual seria o menor conjunto de tubos que interligue todos os locais. Existe também o famoso Problema do Caixeiro Viajante, que consiste em encontrar o caminho mais curto através da estrutura que visita cada cidade exatamente uma vez.

15.2. Terminologia

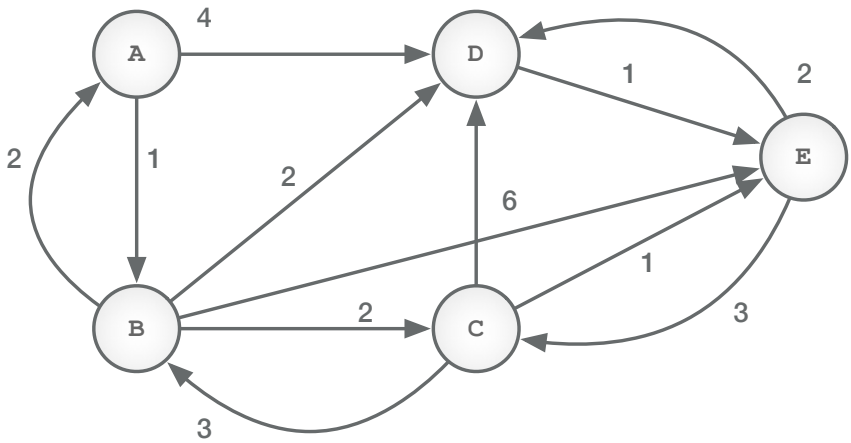
O tipo de estrutura na figura apresentada anteriormente é denominada formalmente como grafo. Um grafo consiste numa série de nós (também chamados de vértices), exibidos como nós, e arestas (também chamadas de linhas, links ou, em grafos direcionados, arcos), exibidos como conexões entre os nós. Existe bastante terminologia que nos permite especificar grafos como podemos ver de seguida:

Diz-se que um grafo é simples se não tiver laços próprios (ou seja, arestas conectadas em ambas as extremidades ao mesmo vértice) e não mais do que uma aresta conectando qualquer par de vértices. O restante deste capítulo irá assumir essa definição, o que é suficiente para a maioria das aplicações práticas.

Se existirem etiquetas nas arestas (geralmente números inteiros não negativos), dizemos que o grafo é pesado. Distinguimos entre grafos direcionados e não direcionados. Nos grafos direcionados (também chamados de dígrafos), cada aresta vem com uma ou duas direções, que geralmente são indicadas por setas. Podemos pensar nas arestas como ligações que podem por exemplo representar estradas, onde algumas estradas podem ser apenas de sentido único. Ou pensar nos números associados a uma viagem: como subir uma montanha que demora mais tempo do que descer. Um exemplo de um dígrafo não pesado é:



e um exemplo de dígrafo pesado, com etiquetas nas arestas:



Em grafos não direcionados, assumimos que cada aresta pode ser vista como passível de ser percorrida nos dois sentidos, ou seja, uma aresta entre A e B vai de A a B assim como de B a A. O primeiro grafo dado no início deste capítulo é pesado e não direcionado.

Um caminho é uma sequência de nós ou vértices v1, v2, . . . , vn tal que vi e vi+1 são conectados por uma aresta para todo $1 \leq i \leq n-1$. Observe que num grafo direcionado, a aresta de vi a vi+1 é aquela que tem a direção correspondente. Um laço é um caminho não vazio cujo primeiro vértice é o mesmo que seu último vértice. Um caminho é simples se nenhum vértice aparecer duas vezes (com exceção de um laço, onde o primeiro e o último vértice podem ser o mesmo).

Um grafo não direcionado é conexo se cada par de vértices tem um caminho que os liga. Para grafos direcionados, a noção de conexidade tem duas versões distintas: Dizemos que um dígrafo é conexo se para cada dois vértices A e B existe um caminho de A para B ou um caminho de B para A.

Um grafo tem claramente muitas propriedades semelhantes a uma árvore. De facto, qualquer árvore pode ser vista como um grafo simples de um tipo particular, ou seja, aquele que é conexo e não contém laços. Como um grafo ao contrário de uma árvore, não vem com um "ponto de partida" natural a partir do qual existe um caminho único para cada vértice, não faz sentido falar de pais e filhos num grafo. Em vez disso, se dois vértices A e B são ligados por uma aresta dizemos que são vizinhos. Duas arestas que têm um vértice em comum (por exemplo, uma que liga A e B e uma que liga B e C) são adjacentes.

15.3. Implementar grafos

Todas as estruturas de dados que consideramos até agora foram projetadas para armazenar determinada informação ao mesmo tempo que queríamos realizar determinadas operações tais como inserir novos elementos, excluir elementos específicos, pesquisar elementos específicos assim como ordenar a coleção. Em nenhum momento existiu qualquer ligação entre todos os elementos armazenados além da ordem dos elementos. Além disso, essa ligação entre os elementos nunca foi algo inerente à estrutura e que, portanto, tenhamos tentado representar de alguma forma – era apenas uma propriedade que foi usada para armazenar os elementos de forma que fosse mais fácil e rápido ordenar e pesquisar. Agora, por outro lado, são as ligações a informação crucial que precisamos definir na estrutura de dados.

Implementação baseada em *array*. A primeira ideia subjacente para a implementação baseada em *array* é que podemos renomear convenientemente os vértices do grafo para que sejam rotulados por índices inteiros não negativos, digamos de 0 a n − 1, para o caso de ainda não terem esses rótulos. No entanto, isso só funciona se o grafo for dado explicitamente, ou seja, se soubermos antecipadamente quantos vértices irão existir e quais pares terão arestas entre eles. Então, precisamos apenas acompanhar que vértices têm uma aresta para outro vértice e, para os grafos pesados quais são os pesos definidos para as arestas. Para grafos não pesados podemos representar isso facilmente num *array* bidimensional binário n × n, também chamada de matriz de adjacência. No caso de grafos pesados, temos uma matriz de pesos n × n. As representações de *array*/matriz para os dois grafos de exemplo apresentados anteriormente são:

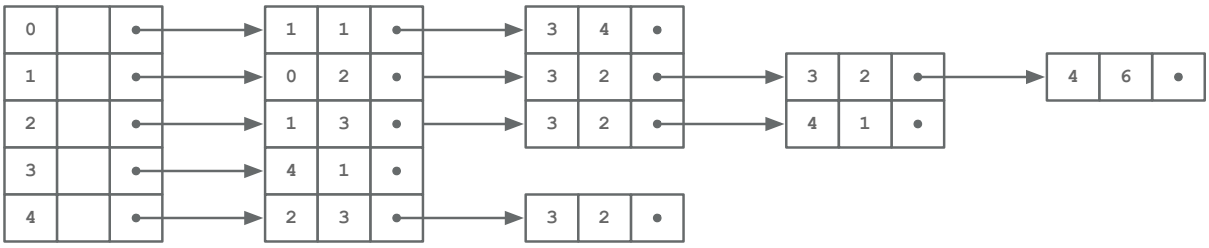
		A	B	C	D	E
		0	1	2	3	4
A	0	0	1	0	1	0
B	1	0	0	1	0	0
C	2	1	0	0	0	1
D	3	0	0	1	0	1
E	4	0	0	0	0	0

		A	B	C	D	E
		0	1	2	3	4
A	0	0	1	∞	4	∞
B	1	2	0	2	2	6
C	2	∞	3	0	2	1
D	3	∞	∞	∞	0	1
E	4	∞	∞	3	2	0

No primeiro caso, para o grafo não pesado, um ‘0’ na posição adj[i][j] é lido como falso, ou seja, não há aresta do vértice i ao vértice j. Um '1', por outro lado é lido como verdadeiro indicando que existe uma aresta. Muitas vezes é útil usar valores booleanos aqui em vez dos números 0 e 1 porque nos permite realizar operações nos booleanos. No segundo caso, temos um grafo pesado e em vez dos valores inteiros 0 e 1 ou os booleanos temos os valor dos pesos reais das arestas e usamos o símbolo de infinito ∞ para indicar quando não existe aresta.
Para um grafo não direcionado, se existir um 1 na coluna i e na linha j, sabemos que existe uma aresta do vértice i ao vértice com o número j o que significa que também existe uma aresta do vértice j ao vértice i. Isso significa que adj[i][j] == adj[j][i] para todo i e j de 0 a n − 1, então nestes casos existe alguma informação redundante. Dizemos que esse tipo de matriz é simétrica.

Implementação mista. Existe um problema potencial com a representação através da matriz de adjacências: se o grafo tiver muitos vértices a matriz associada será extremamente grande (por exemplo, são necessárias 10.000 entradas se o grafo tiver apenas 100 vértices). Então, se o grafo é esparso (ou seja, tem relativamente poucas arestas), a matriz de adjacências irá conter muitos 0s e apenas alguns 1s que é um desperdício de espaço reservar tanta memória para tão pouca informação.

Uma solução para esse problema é numerar todos os vértices como anteriormente mas em vez de usar um *array* bidimensional usar um *array* unidimensional em que cada posição é uma referência para uma lista ligada de vizinhos para cada vértice. Por exemplo, o grafo pesado acima pode ser representado da seguinte forma:



No caso de existirem muito poucas arestas teremos listas muito curtas em cada entrada do *array* economizando espaço na representação versus a representação através de uma matriz de adjacências. Esta implementação usa uma abordagem denominada lista de adjacências. Observe que se estivermos a considerar grafos não direcionados ainda existe uma certa redundância na representação pois cada aresta é representada duas vezes, uma em cada lista correspondente aos dois vértices que ela conecta. Em Java, isso pode ser feito com algo como:

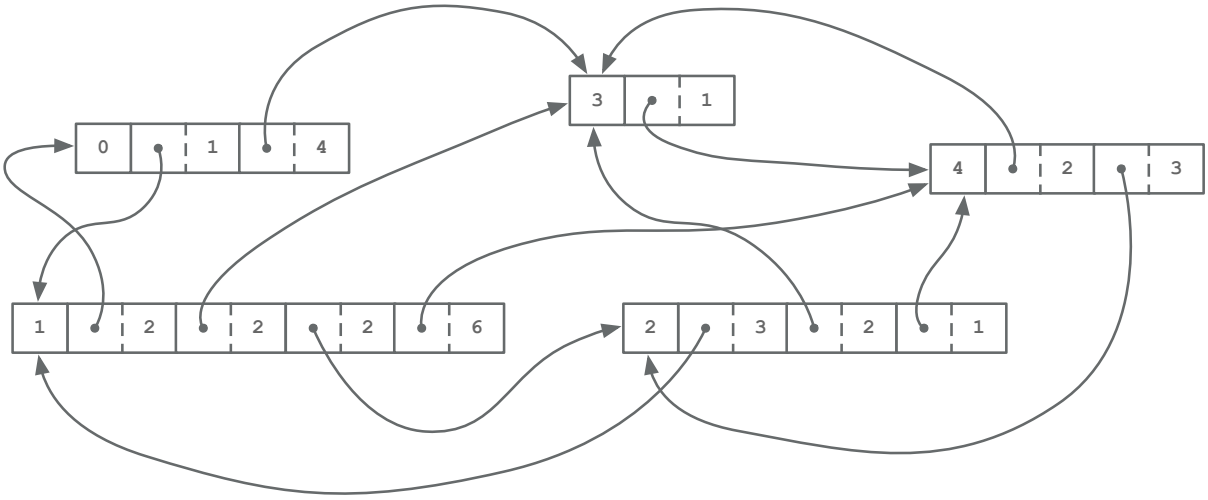
```
1. class Graph {
2.     Vertex[] heads;
3.     private class Vertex {
4.        int name;
5.        double weight;
6.        Vertex next;
7.        ...//methods for vertices
8.     }
9.     ...//methods for graphs
10.}
```

Implementação em lista ligada. A implementação padrão baseada em referências de árvores binárias, que é essencialmente uma generalização de listas ligadas pode ser generalizada para grafos. Numa linguagem como Java, uma classe `Graph` pode ter o seguinte como uma classe interna:

```
1. class Vertex {
2.     String name;
3.     Vertex[] neighbours;
4.     double[] weights;
5. }
```

Quando cada vértice é criado e alocado um *array* de vizinhos grande o suficiente para acomodar (referências para) todos os seus vizinhos, com (para grafos pesado) um *array* de

pesos de tamanho igual para acomodar os pesos associados. Em seguida, colocamos os vizinhos de cada vértice nesses *arrays* em alguma ordem arbitrária. Quaisquer entradas no *array* de vizinhos que não sejam necessárias irão ter uma referência a `null` como de costume. Por exemplo, o grafo pesado apresentado anteriormente seria representado da seguinte forma: cada peso é colocado a seguir à respetiva referência ao qual está associado (etiqueta origem; referência destino, peso, etc):



Para seguirmos a mesma abordagem anterior temos que definir um conjunto de interfaces que irão representar a nossa estrutura de grafo quer seja pesada ou não e definir as nossas operações.

```
1. /**
2.  * GraphADT defines the interface to a graph data structure.
3.  *
4.  */
5. public interface GraphADT<T> {
6.    /**
7.     * Adds a vertex to this graph, associating object with vertex.
8.     *
9.     * @param vertex  the vertex to be added to this graph
10.    */
11.   void addVertex (T vertex);
```

```
12.    /**
13.     * Removes a single vertex with the given value from this graph.
14.     *
15.     * @param vertex  the vertex to be removed from this graph
16.     */
17.    void removeVertex (T vertex);
18.
19.    /**
20.     * Inserts an edge between two vertices of this graph.
21.     *
22.     * @param vertex1  the first vertex
23.     * @param vertex2  the second vertex
24.     */
25.    void addEdge (T vertex1, T vertex2);
26.
27.    /**
28.     * Removes an edge between two vertices of this graph.
29.     *
30.     * @param vertex1  the first vertex
31.     * @param vertex2  the second vertex
32.     */
33.    void removeEdge (T vertex1, T vertex2);
34.    /**
35.     * Returns a breadth first iterator starting with the given vertex.
36.     *
37.     * @param startVertex  the starting vertex
38.     * @return             a breadth first iterator beginning at
39.     *                     the given vertex
40.     */
41.    Iterator iteratorBFS(T startVertex);
42.
43.    /**
44.     * Returns a depth first iterator starting with the given vertex.
45.     *
46.     * @param startVertex  the starting vertex
47.     * @return             a depth first iterator starting at the
48.     *                     given vertex
49.     */
```

```
50.    Iterator iteratorDFS(T startVertex);
51.    /**
52.     * Returns an iterator that contains the shortest path between
53.     * the two vertices.
54.     *
55.     * @param startVertex   the starting vertex
56.     * @param targetVertex  the ending vertex
57.     * @return              an iterator that contains the shortest
58.     *                      path between the two vertices
59.     */
60.    Iterator iteratorShortestPath(T startVertex, T targetVertex);
61.
62.    /**
63.     * Returns true if this graph is empty, false otherwise.
64.     *
65.     * @return  true if this graph is empty
66.     */
67.    boolean isEmpty();
68.
69.    /**
70.     * Returns true if this graph is connected, false otherwise.
71.     *
72.     * @return  true if this graph is connected
73.     */
74.    boolean isConnected();
75.    /**
76.     * Returns the number of vertices in this graph.
77.     *
78.     * @return  the integer number of vertices in this graph
79.     */
80.    int size();
81.
82.    /**
83.     * Returns a string representation of the adjacency matrix.
84.     *
85.     * @return  a string representation of the adjacency matrix
86.     */
87.    String toString();
```

```
88.}
```

De seguida é apresentada a interface que define um grafo pesado.

```
1. /**
2.  * NetworkADT defines the interface to a network.
3.  *
4.  */
5. public interface NetworkADT<T> extends GraphADT<T> {
6.    /**
7.     * Inserts an edge between two vertices of this graph.
8.     *
9.     * @param vertex1  the first vertex
10.    * @param vertex2  the second vertex
11.    * @param weight   the weight
12.    */
13.   void addEdge (T vertex1, T vertex2, double weight);
14.
15.   /**
16.    * Returns the weight of the shortest path in this network.
17.    *
18.    * @param vertex1  the first vertex
19.    * @param vertex2  the second vertex
20.    * @return         the weight of the shortest path in this network
21.    */
22.   double shortestPathWeight(T vertex1, T vertex2);
23.}
```

Atenção que a interface que define um grafo pesado dificilmente é reutilizada para outros problemas já que normalmente os pesos são específicos de cada problema.

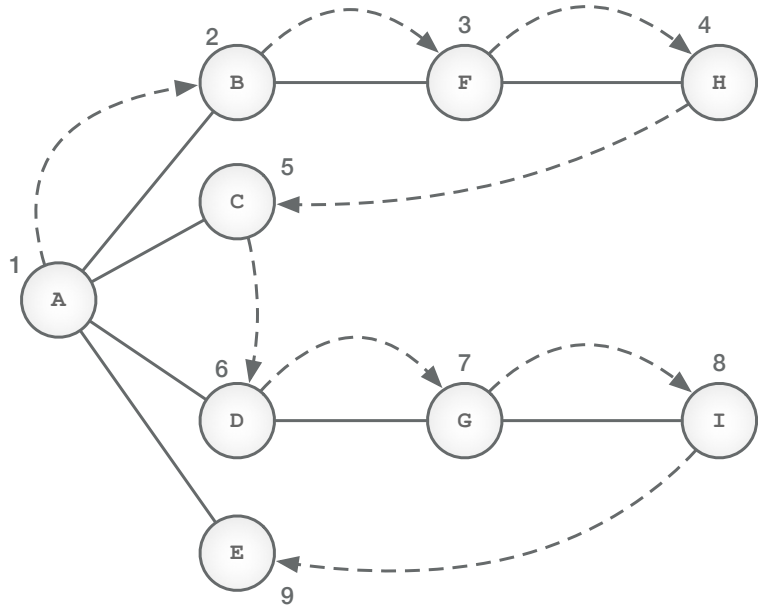
15.4. Travessias

Para percorrer um grafo, ou seja, visitar sistematicamente todos os seus vértices, precisamos claramente de uma estratégia de exploração de grafos que garanta que não iremos perder nenhuma aresta ou vértice. Como ao contrário das árvores, os grafos não têm um vértice raiz,

não existe um lugar natural para iniciar uma travessia e, portanto, assumimos que recebemos, ou escolhemos aleatoriamente, um vértice inicial i. Existem duas estratégias para realizar a travessia de grafos. Ambas as estratégias eventualmente irão alcançar todos os vértices conectados. A travessia em profundidade é implementada com uma pilha, enquanto a travessia em largura é implementada com recurso a uma fila. Esses mecanismos resultam, como veremos, no grafo sendo percorrido de diferentes maneiras.

15.4.1. Travessia em profundidade

A travessia em profundidade usa uma pilha para armazenar o caminho para onde deve ir quando chegar a um ponto sem saída. De seguida é apresentada uma figura que ilustra esta travessia. Os números na figura mostram a ordem em que os vértices são visitados.



Para realizar a travessia em profundidade é necessário primeiro escolher um ponto de partida - neste caso, o vértice A. Depois disso temos de fazer três coisas: visitar esse vértice, colocá-lo numa pilha para que possamos voltar a ele mais tarde e marcá-lo para não ser visitado novamente.

De seguida, vamos para qualquer vértice adjacente a A que ainda não foi visitado. Vamos supor que os vértices são selecionados por ordem alfabética, que nesse caso seria o B. Visitamos B e marcamo-lo como visitado e colocamo-lo na pilha.

E agora? Como estamos em B e fazemos a mesma coisa: vamos para um vértice adjacente que não foi visitado, neste caso seria o F. A este processo podemos dar o nome de Regra 1.

Regra 1

Se possível, visitar um vértice adjacente não visitado, marcá-lo como visitado e colocá-lo na pilha.

Ao aplicar a Regra 1 novamente temos o H. Neste ponto, entretanto, precisamos fazer outra coisa porque não existem vértices não visitados adjacentes a H. É aqui que entra a Regra 2.

Regra 2

Se não podemos seguir a Regra 1, então, se possível, retirar um vértice da pilha.

Seguindo esta regra retiramos H da pilha, o que o traz de volta o F. O F não tem vértices adjacentes não visitados, então é também retirado. Acontece o mesmo com o B. Agora só nos resta A.

A, no entanto, tem vértices adjacentes não visitados, então visitamos o próximo, C. Mas C é o fim da linha novamente, então é retirado e voltamos para A. De seguida são visitados D, G, I e retirados da pilha levando a um ponto sem saída em I. Agora estamos de volta a A. Visitamos E e voltamos novamente para A.

Desta vez, no entanto, A não tem vizinhos não visitados, então retiramo-lo da pilha fazendo com que não exista outro vértice no grafo que é a condição que define a Regra 3.

Regra 3

Se não podemos usar a Regra 1 ou a Regra 2, a travessia está concluída.

A tabela seguinte mostra os vários estados da pilha durante a travessia em profundidade.

Evento	Stack

Visitar A/Push A	A
Visitar B/Push B	AB
Visitar F/Push F	ABF
Visitar H/Push H	ABFH
Pop H	ABF
Pop F	AB
Pop B	A
Visitar C/Push C	AC
Pop C	A
Visitar D/Push D	AD
Visitar G/Push G	ADG
Visitar I/Push I	ADGI
Pop I	ADG
Pop G	AD
Pop D	A
Visitar E/Push E	AE
Pop E	A
Pop A	
Concluído	

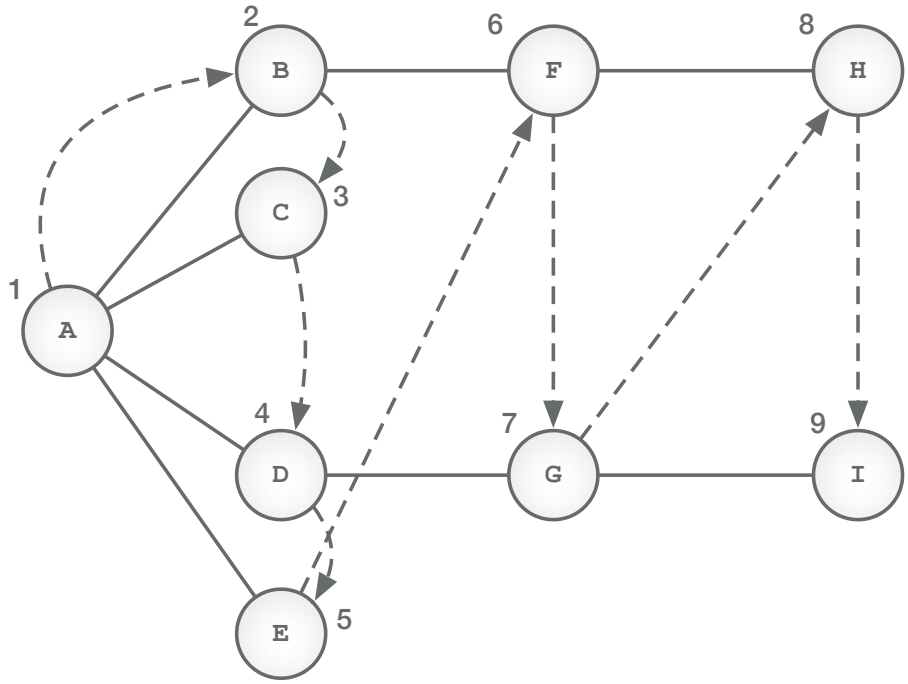
O conteúdo da pilha é a rota percorrida a partir do vértice inicial. À medida que nos afastamos do vértice inicial os vértice são colocados na pilha à medida que avançamos. À medida que nos aproximamos de volta ao vértice inicial vamos retirando os vértices da pilha. A ordem com que são visitados os vértices é a seguinte: ABFHCDGIE.

Podemos dizer que o algoritmo de travessia em profundidade gosta de se afastar o mais rápido possível do ponto de partida e retornar apenas quando chega a um ponto sem saída. Se usarmos o termo profundidade para definir a distância ao ponto de partida compreendemos de onde vem o nome travessia em profundidade.

15.4.2. Travessia em largura

Como vimos na travessia em profundidade, o algoritmo age como se se quisesse afastar o mais rápido possível do ponto de partida. Na travessia em largura, por outro lado, o algoritmo

gosta de ficar o mais próximo possível do ponto de partida. Começa por visitar todos os vértices adjacentes ao vértice inicial e só depois vai mais longe. Esse tipo de travessia é implementado através da utilização de uma fila em vez de uma pilha. A seguinte apresenta o mesmo grafo da figura anterior, mas aqui é usada a travessia em largura. Novamente, os números indicam a ordem em que os vértices são visitados.



A é o vértice inicial então iremos visitá-lo e o torná-lo o vértice atual. De seguida temos de seguir as seguintes regras:

Regra 1

Visite o próximo vértice não visitado (se existir) adjacente ao vértice atual, marque-o como visitado, e insira-o na fila.

Regra 2

Se não conseguirmos aplicar a Regra 1 porque não existem mais vértices não visitados, remova um vértice da fila (se possível) e torná-lo o vértice atual.

Regra 3

Se não conseguirmos cumprir a Regra 2 porque a fila está vazia, então terminou.

Assim, primeiro são visitados todos os vértices adjacentes a A, inserindo cada um na fila à medida que são visitados. Agora foram visitados A, B, C, D e E. Neste ponto, a fila (da frente para trás) contém BCDE.

Não existem mais vértices não visitados adjacentes a A, então é removido B da fila e procuramos por vértices adjacentes a B. Encontramos F que é inserido na fila. Não existem mais vértices não visitados adjacentes a B, então removemos C da fila. Não tem vértices não visitados adjacentes, então removemos D e visitamos G. D não tem mais vértices não visitados adjacentes, então removemos E. Agora a fila é FG. Removemos F e visitamos H, de seguida removemos G e visitamos I.

Agora a fila é HI, mas quando removemos cada um deles não encontramos nenhum vértice não visitado adjacente, a fila está então vazia o que indica que está concluída a travessia. A tabela seguinte mostra essa sequência.

Evento	Queue
Visitar A	
Visitar B/Enqueue B	B
Visitar C/Enqueue C	BC
Visitar D/Enqueue D	BCD
Visitar E/Enqueue E	BCDE
Dequeue B	CDE
Visitar F/Enqueue F	CDEF
Dequeue C	DEF
Dequeue D	EF
Visitar G/Enqueue G	EFG
Dequeue E	FG
Dequeue F	G
Visitar H/Enqueue H	GH
Dequeue G	H
Visitar I/Enqueue I	HI
Dequeue H	I
Dequeue I	
Concluído	

A cada momento, a fila contém os vértices que foram visitados, mas cujos vizinhos ainda não foram totalmente explorados. (Compare esta travessia em largura com a travessia em profundidade, onde o conteúdo da pilha é a rota que foi percorrida desde o ponto inicial até ao vértice atual.) Os vértices são visitados na seguinte ordem ABCDEFGHI.

15.5. Caminho mais curto – algoritmo *Dijkstra*

Um problema comum baseado em grafos é que temos alguma situação representada como um dígrafo pesado com arestas rotuladas por números não negativos e precisamos responder à seguinte pergunta: Para dois vértices particulares, qual é o caminho mais curto de um para o outro?

Aqui, por "caminho mais curto" queremos dizer um caminho que, quando somamos os pesos ao longo das suas arestas dá o menor peso total para o caminho. Esse número é chamado de comprimento do caminho. Assim, um caminho mais curto é aquele com comprimento mínimo. Observe que não precisa haver um caminho mais curto exclusivo, pois vários caminhos podem ter o mesmo comprimento. Num grafo não conexo não haverá um caminho entre os vértices.

Observe que os pesos não precisam necessariamente corresponder às distâncias; podem ser, por exemplo, tempo (neste caso poderíamos falar de "caminhos mais rápidos") ou dinheiro (neste caso poderíamos falar de "caminhos mais baratos"), entre outras possibilidades. Ao considerar grafos "abstratos" nos quais os pesos numéricos são deixados sem interpretação podemos resolver todas essas situações e outras. No entanto observe que precisamos restringir os pesos das arestas a números não negativos, porque se existirem números e ciclos negativos podemos ter caminhos cada vez mais longos com custos cada vez menores e sem qualquer caminho com custo mínimo.

As aplicações de algoritmos de caminho mais curto incluem roteamento de pacotes de internet (porque, se pretendermos enviar uma mensagem de e-mail no nosso computador para outra pessoa, esse e-mail precisa passar por vários servidores de e-mail, até chegar ao seu destino final), sistemas de reserva de bilhetes de comboio (com a necessidade de

descobrir as melhores estações para as ligações) e pesquisa de trajetos de condução (que precisam encontrar uma rota ideal em algum sentido).

Algoritmo de *Dijkstra*. Acontece que, se quisermos calcular o caminho mais curto de um dado nó inicial s para um determinado nó final z, na verdade é mais conveniente calcular os caminhos mais curtos de s para todos os outros nós, não apenas o nó z dado que estamos interessados. Dado o nó inicial, o algoritmo de *Dijkstra* calcula os caminhos mais curtos a partir de s e terminando em cada nó possível. O algoritmo mantém todas as informações necessárias em *arrays* simples, que são atualizados iterativamente até que a solução seja alcançada. Como o algoritmo, embora elegante e curto, é bastante complicado, vamos considerá-lo um componente de cada vez.

Superestimação dos caminhos mais curtos. Mantemos um *array* D de distâncias indexado pelos vértices. A ideia é que D[z] mantenha a distância do caminho mais curto de s a z quando o algoritmo terminar. No entanto, antes que o algoritmo termine, D[z] é a melhor superestimativa que temos atualmente da distância de s a z. Inicialmente temos D[s] = 0, e definimos D[z] = ∞ para todos os vértices z exceto o nó inicial s. De seguida, o algoritmo diminui repetidamente as superestimativas até que não seja mais possível diminuí-las ainda mais. Quando isso acontece, o algoritmo termina, com cada estimativa totalmente restrita e considerada curta (ou pequena).

Melhorar as estimativas. A ideia geral é procurar sistematicamente por atalhos. Vamos supor que para dois vértices dados u e z: D[u] + peso[u][z] < D[z]. Então existe uma forma de ir de s para u e depois para z cujo comprimento total é menor do que a atual superestimativa D[z] da distância de s para z, e portanto podemos substituir D[z] por esta estimativa melhor. Isso corresponde ao fragmento de código do algoritmo completo fornecido abaixo.

```
1.    if ( D[u] + weight[u][z] < D[z] )
2.       D[z] = D[u] + weight[u][z]
```

O problema é, portanto, reduzido a desenvolver um algoritmo que aplicará sistematicamente essa melhoria de modo que (1) eventualmente iremos obter estimativas mais rigorosas e (2) que isso seja feito da maneira mais eficiente possível.

Algoritmo de *Dijkstra*, Versão 1. A primeira versão do algoritmo não é tão eficiente quanto poderia ser, mas é relativamente simples de implementar. (É sempre uma boa ideia começar com um algoritmo simples mesmo que ineficiente, para que os resultados possam ser usados para verificar a operação de um algoritmo eficiente mais complexo). A ideia geral é que, em cada fase da operação do algoritmo, se uma entrada D[u] do *array* D tem o valor mínimo entre todos os valores registados em D, então a superestimativa D[u] deve ser realmente curta (ou pequena), porque o algoritmo de melhoria discutido acima não pode encontrar um atalho. O algoritmo a seguir implementa essa ideia:

```
1.    // Input:  A directed graph with weight matrix 'weight' and
2.    //         a start vertex 's'.
3.    // Output: An array 'D' of distances as explained above.
4.    // We begin by buiding the distance overestimates.
5.    D[s] = 0   // The shortest path from s to itself has length zero.
6.    for ( each vertex z of the graph ) {
7.       if ( z is not the start vertex s )
8.          D[z] = infinity    // This is certainly an overestimate.
9.    }
10.   // We use an auxiliary array 'tight' indexed by the vertices,
11.   // that records for which nodes the shortest path estimates
12.   // are ''known'' to be tight by the algorithm.
13.   for ( each vertex z of the graph ) {
14.      tight[z] = false
15.   }
16.// We now repeatedly update the arrays 'D' and 'tight' until
17.   // all entries in the array 'tight' hold the value true.
18.   repeat as many times as there are vertices in the graph {
19.      find a vertex u with tight[u] false and minimal estimate D[u]
20.      tight[u] = true
21.      for ( each vertex z adjacent to u )
22.         if ( D[u] + weight[u][z] < D[z] )
```

```
23.            D[z] = D[u] + weight[u][z]    // Lower overestimate exists.
24.    }
25.    // At this point, all entries of array ‘D’ hold tight estimates.
```

Fica claro que quando este algoritmo termina, as entradas de D não podem conter subestimativas dos comprimentos dos caminhos mais curtos. O que talvez não seja tão claro é porquê que as estimativas mantidas são curtas (ou apertadas), ou seja, são os comprimentos mínimos do caminho. Para entender o porquê, observe primeiro que um subcaminho inicial de um caminho mais curto é ele mesmo um caminho mais curto. Suponha que pretende navegar de um vértice s para um vértice z e que o caminho mais curto de s para z passa por um certo vértice u. Dessa forma o caminho de s a z pode ser dividido em dois caminhos, um de s para u (um subcaminho inicial) e o outro de u para z (um subcaminho final). Dado que todo o caminho não dividido é um caminho mais curto de s a z, o subcaminho inicial tem que ser um caminho mais curto de s a u, caso contrário o caminho mais curto de s a z poderia ser reduzido se fosse substituindo o subcaminho inicial de s a u por um outro mais curto. Sendo assim para qualquer vértice inicial existe uma árvore de caminhos mais curtos a partir desse vértice para todos os outros vértices. A razão é que os caminhos mais curtos não podem ter ciclos. Implicitamente, o algoritmo de *Dijkstra* constrói essa árvore a partir da raiz, ou seja, a partir do vértice inicial.

Se também pretendermos calcular a rota do caminho mais curto em vez de ser calculado apenas o seu comprimento precisamos introduzir um terceiro *array* `pred` para acompanhar o 'predecessor' ou o 'vértice anterior' de cada vértice para que o caminho possa ser seguido de volta do ponto final ao ponto inicial. O algoritmo também pode ser adaptado para trabalhar com grafos não pesado ao atribuirmos uma matriz de peso adequada, por exemplo 1s para vértices conectados e 0s para vértices não conectados.

A complexidade de tempo deste algoritmo é claramente $O(n^2)$ onde n é o número de vértices pois existem operações de O(n) aninhadas dentro do ciclo de O(n).

Algoritmo de *Dijkstra*, Versão 2. A complexidade de tempo do algoritmo de *Dijkstra* pode ser melhorada fazendo uso de uma fila de prioridade (por exemplo, alguma forma de *heap*) para

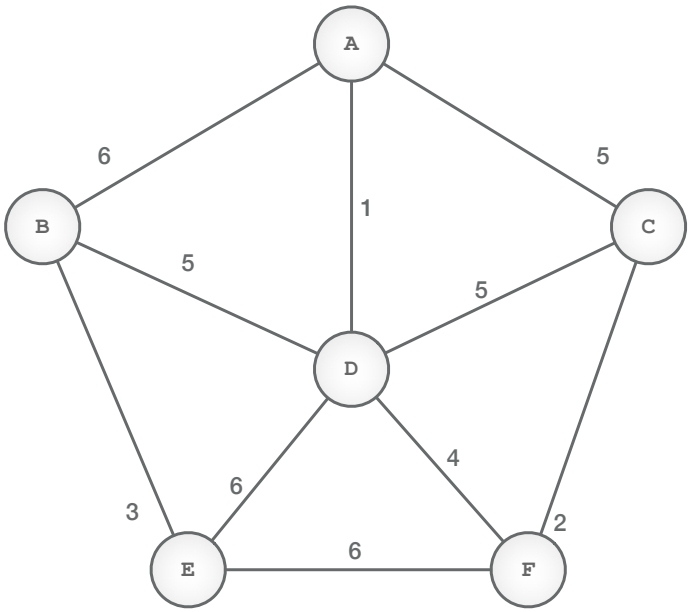
acompanhar qual a estimativa de distância do nó que se irá tornar curta (ou apertada). Aqui é conveniente usar a convenção de que números mais baixos têm prioridade mais alta. O algoritmo anterior torna-se então:

```
1.    // Input:  A directed graph with weight matrix 'weight' and
2.    //         a start vertex 's'.
3.    // Output: An array 'D' of distances as explained above.
4.    // We begin by buiding the distance overestimates.
5.    D[s] = 0   // The shortest path from s to itself has length zero.
6.    for ( each vertex z of the graph ) {
7.       if ( z is not the start vertex s )
8.          D[z] = infinity    // This is certainly an overestimate.
9.    }
10.   // Then we set up a priority queue based on the overestimates.
11.   Create a priority queue containing all the vertices of the graph,
12.   with the entries of D as the priorities
13.   // Then we implicitly build the path tree discussed above.
14.   while ( priority queue is not empty ) {
15.      // The next vertex of the path tree is called u.
16.      u = remove vertex with smallest priority from queue
17.      for ( each vertex z in the queue which is adjacent to u ) {
18.         if ( D[u] + weight[u][z] < D[z] ) {
19.            D[z] = D[u] + weight[u][z]    // Lower overestimate exists.
20.            Change the priority of vertex z in queue to D[z]
21.      } }
22.   }
23.   // At this point, all entries of array 'D' hold tight estimates.
```

Se a fila de prioridade é implementada como uma *heap* binária tanto a inicialização de D como a criação da fila de prioridade têm complexidade O(n), onde n é o número de vértices do grafo sendo isso insignificante em comparação com o resto do algoritmo. De seguida, remover vértices e alterar as prioridades dos elementos na fila de prioridade requer alguma reordenação da árvore que leva a termos um eficiência de $O(\log_2 n)$ porque essa é a altura máxima da árvore. As remoções acontecem O(n) vezes assim como as mudanças de prioridade.

15.6. Árvore geradora de custo mínimo

Passamos agora para outro problema comum baseado em grafos. Suponha que você tenha recebido um gráfico não direcionado ponderado, como o seguinte:

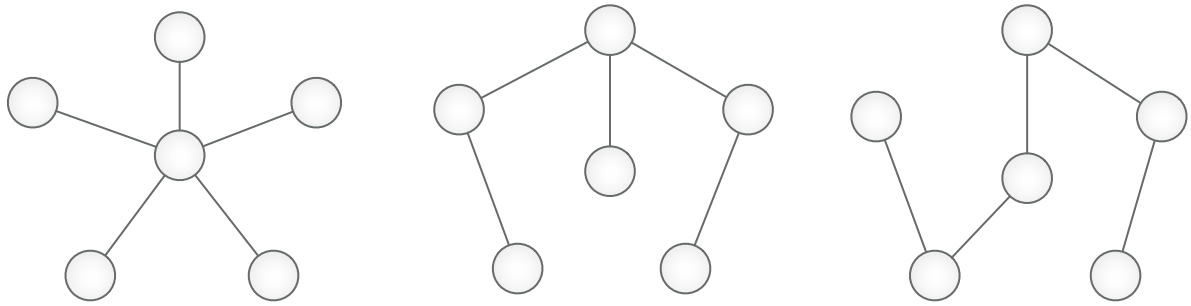


Poderíamos pensar nos vértices como representando casas e os pesos como as distâncias entre elas. Agora imagine que estamos encarregados de abastecer todas essas casas com alguns serviços como água, gás ou eletricidade. Por razões óbvias queremos manter a quantidade de fundações ou escavações para a colocação de tubos ou cabos no seu mínimo.

Então, qual é o melhor layout de tubos ou cabos que podemos passar, ou seja, qual layout tem o menor comprimento total?

Obviamente, teremos que escolher algumas arestas para escavar, mas não todas. Por exemplo, se já escolhemos a aresta entre A e D, e aquela entre B e D, então não existe qualquer razão para escavarmos também a aresta entre A e B. De forma mais geral, é claro que queremos evitar círculos. Além disso, supondo que tenhamos apenas um ponto de alimentação (não importa qual dos vértices seja), precisamos que todo o layout esteja conectado. Já vimos anteriormente que um grafo conexo sem círculos é uma árvore.

Portanto, o que estamos à procura é de uma árvore geradora mínima do grafo. Uma árvore geradora de um grafo é um subgrafo que é uma árvore que conecta todos os vértices juntos de modo que "abrange" o grafo original mas através da utilização de menos arestas. Aqui, mínimo refere-se à soma de todos os pesos das arestas contidas nessa árvore, de modo que uma árvore geradora mínima tem peso total menor ou igual ao peso total de todas as outras árvores geradoras. Como veremos, não irá existir necessariamente uma única árvore geradora mínima para um determinado grafo. Vamos supor que todos os pesos no grafo anterior são iguais para nos dar uma ideia de que tipo de forma uma árvore geradora mínima pode ter nessas circunstâncias. aqui estão alguns exemplos:



Podemos notar imediatamente que a sua forma geral é tal que, se adicionarmos qualquer uma das arestas restantes, criaremos um círculo. Então podemos ver que ir de uma árvore geradora para outra pode ser alcançado removendo uma aresta e substituindo-a por outra (para o vértice que de outra forma seria desconectado) de modo que nenhum círculo seja criado.

15.7. Exercícios Propostos

Exercício 1

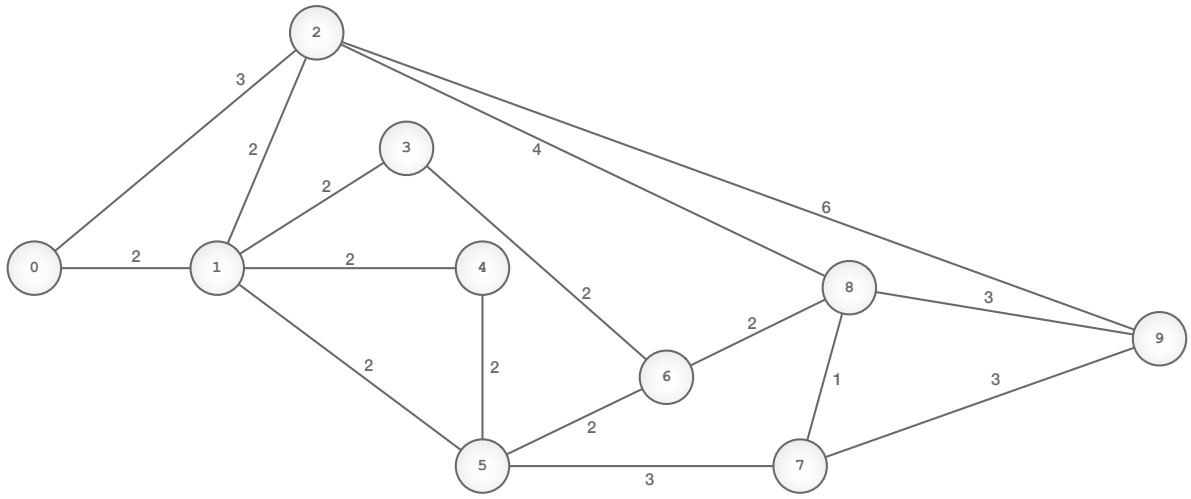
O que entende por grafo conexo?

Exercício 2

Comente a seguinte afirmação: "*A implementação de um grafo através de uma matriz não é muito eficiente*".

Exercício 3

Apresente o resultado das travessias BFS e DFS para o seguinte grafo:

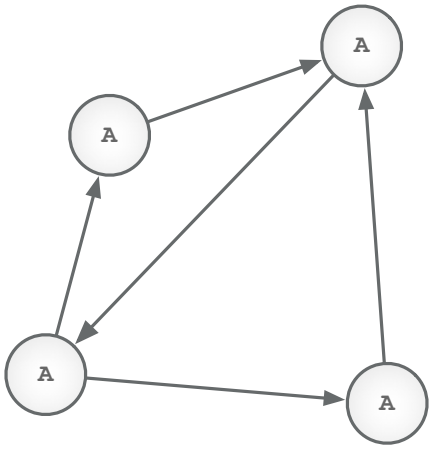


Exercício 4

Qual a relação entre um grafo e uma árvore?

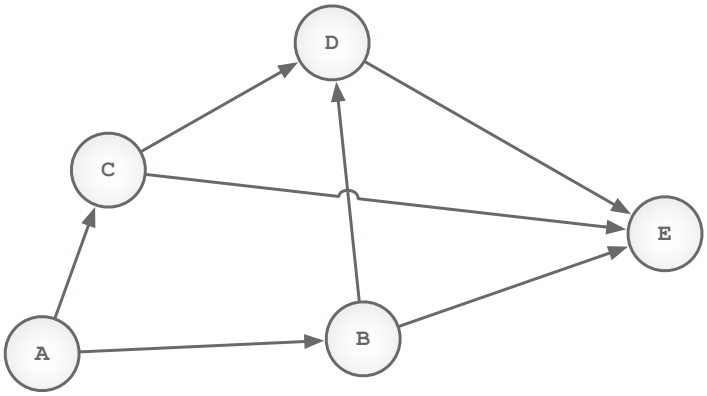
Exercício 5

Considere o grafo seguinte e apresente a respetiva matriz de adjacência e lista de adjacências.



Exercício 6

Apresentar o caminho mais curto do grafo apresentado de seguida de A a E.



Exercício 7

O que entende por árvore geradora de custo mínimo?