3 Busca básica em grafos

O percorrimento dos vértices e arestas respeitando a estrutura dos grafos é chamada de "busca" e constitui uma operação básica em grafos dirigidos e não dirigidos. Este capítulo apresenta os dois métodos diferentes para busca: em largura e em profundidade. São mostrados exemplos de pseudocódigo que podem ser facilmente implementados em uma linguagem de programação. Ao final do capítulo, dois exemplos de aplicação direta de busca são apresentados e discutidos.

3.1 Introdução

Estruturas de dados frequentemente precisam ser percorridas para que diversas operações sejam efetuadas, tais como: inserir ou remover novos elementos ou simplesmente fazer leitura de dados armazenados na estrutura. Estruturas mais simples geralmente também são manipuladas com maior facilidade. Por exemplo, para atribuirmos um valor a uma variável simples, basta atribuí-lo diretamente utilizando o identificador da variável, conforme apresentado no exemplo abaixo.

```
int a;
a = 10;
```

No caso de vetores, seus elementos são acessados por meio de índices numéricos. Isto é possível, visto que os elementos de um vetor são dispostos em um bloco contínuo de memória. Assim, o identificador do vetor é uma referência para o endereço de memória do início deste bloco. Os demais elementos são acessados por meio da indexação numérica do identificador. Por exemplo, se quisermos percorrer um vetor de inteiros com 10 elementos, para atribuir valor nulo a todos eles, podemos simplesmente construir um laço de 10 repetições utilizando o contador do laço como índice para os elementos do vetor.

```
int v[10]; // declaração do vetor
int i;       // contador
for (i = 0; i < 10; i++)
    v[i] = 0;
```

Para estruturas alocadas dinamicamente, tais como listas encadeadas, o processo se torna mais complicado, pois cada elemento é armazenado em um endereço de memória que não necessariamente está em uma posição contígua à do elemento anterior. Por esse motivo,

para se chegar a um determinado elemento de uma lista, deve ser feito o percorrimento da lista passando por todos os seus elementos desde o elemento inicial, até o elemento desejado.

No caso dos grafos, o problema de manipulação da estrutura de dados é ainda mais complexo, pois normalmente queremos percorrer a estrutura respeitando a sua topologia. Os algoritmos de busca constituem métodos para resolver este problema, promovendo a pesquisa em grafos. **Pesquisar** um grafo significa visitar seus vértices passando sistematicamente por suas arestas, ou seja, não é apenas listar os vértices de um grafo em determinada ordem, mas sim atingir cada vértice obedecendo a estrutura do grafo, definida pelas arestas. Existem basicamente dois métodos de busca: a busca em largura e a busca em profundidade.

O fato dos métodos de busca servirem para explorar a estrutura de um grafo nos traz algumas consequências importantes. Em primeiro lugar, os algoritmos de busca servem de base para construção de outros algoritmos mais especializados, pois a solução de vários problemas envolvendo grafos requer que seus vértices sejam visitados. Exemplos destes algoritmos são o algoritmo de Dijkstra, para cálculo de caminhos de custo mínimo, o algoritmo de Prim, para determinação da árvore geradora de custo mínimo e o algoritmo de componentes fortemente conexas, entre outros.

Como aplicação direta dos algoritmos de busca para solução de problemas, temos a busca em grafos de estados, na área de Inteligência Artificial e a ordenação topológica de grafos acíclicos dirigidos. Estes exemplos serão apresentados e discutidos ao final deste capítulo.

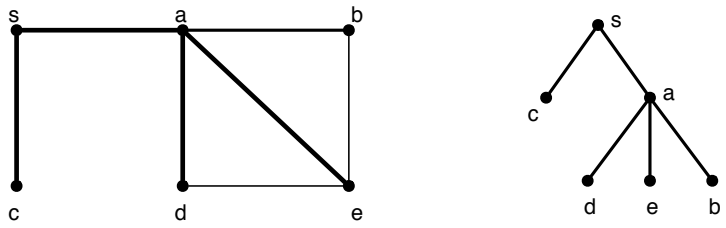
3.2 Busca em largura

O método de busca em largura explora o grafo a partir de um vértice inicial arbitrário. A partir daí, a busca vai se alastrando uniformemente pelo grafo. Em cada grande iteração do algoritmo, são visitados todos os vértices a uma mesma distância (em quantidade de arestas) do vértice inicial. Assim, se imaginássemos um grande grafo desenhado no chão e derramássemos um balde de tinta sobre o vértice inicial, a tinta se espalharia uniformemente a partir desse ponto, da mesma forma que a busca em largura o faz.

A busca em largura tem uma propriedade interessante: para atingir cada vértice do grafo, a busca percorre os caminhos mais curtos desde o vértice inicial, medidos em quantidade de arestas percorridas. O conjunto de caminhos percorridos forma uma estrutura de árvore, cuja raiz é o vértice inicial. Esta árvore é comumente chamada de **árvore de busca em largura**. Desta forma, é interessante que as implementações computacionais de busca em largura armazenem esta árvore enquanto exploram a estrutura do grafo. A

Figura 3.1 ilustra um exemplo simples de grafo. Se tomarmos o vértice s como vértice inicial, teríamos a árvore de busca mostrada ao lado do grafo.

Figura 3.1 – Exemplo de árvore de busca em largura



Fonte: o autor.

A árvore de busca em largura é uma árvore n-ária, pois não sabemos a priori quantos filhos cada nó da árvore terá. Nas implementações de árvores n-árias, geralmente cada nó possui uma lista encadeada de referências para os nós filhos. No caso de árvores de busca, como o objetivo principal é apenas armazenar os caminhos percorridos, podemos encarar as coisas por outro ângulo: não sabemos quantos filhos terá cada nó, mas podemos ter certeza de que cada nó tem apenas um único pai. Assim, ao invés de armazenar os n filhos de cada nó, podemos armazenar somente uma referência para seu pai, resultando em uma estrutura de dados muito mais simples, que pode ser implementada com apenas um vetor de referências para vértices.

A este vetor, damos o nome de **vetor de roteamento** ou **vetor de predecessores** e o denotamos pelo símbolo $R(G)$. Assim, a árvore de busca em largura da Figura 3.1 corresponde aos seguintes valores no vetor de roteamento: $\rho(s) = nil$, $\rho(a) = s$, $\rho(b) = a$, $\rho(c) = s$, $\rho(d) = a$ e $\rho(e) = a$.

De forma a facilitar o texto, podemos representar o vetor de roteamento $R(G)$ conforme mostrado no Quadro 2.

Quadro 2 – Exemplo de vetor de roteamento $R(G)$

v	s	a	b	c	d	e
ρ**(v)**	*nil*	s	a	s	a	a

Fonte: o autor.

3.2.1 O algoritmo de busca em largura

A seguir, veremos o pseudocódigo do algoritmo de busca em largura, também conhecido como BFS (do inglês *Breath First Search*)(CORMEN et al., 2001). Na implementação deste pseudocódigo, cada vértice possui um atributo chamado *estado*, que indica

seu estado atual na busca (não visitado, ou visitado), além disso, a busca utiliza uma fila de vértices, que conduz a ordem de visitação deles. Os elementos do pseudocódigo são os seguintes:

- r: representa o vértice inicial;
- $estado(v)$: estado do vértice v ao longo da execução do algoritmo, que pode assumir um dos seguintes valores $\{NAO\_VISITADO, VISITADO, ENCERRADO\}$. O valor $NAO\_VISITADO$ significa que a busca ainda não atingiu este vértice. O valor $VISITADO$, indica que a busca já atingiu o vértice v, mas ele ainda tem vizinhos (vértices adjacentes) ainda não visitados. O valor $ENCERRADO$ indica que a visitação deste vértice está encerrada, isto é, tanto ele como todos os seus vizinhos já foram visitados pela busca;
- $\rho(v)$: referência ao vértice predecessor do vértice v na busca, representa seu pai na árvore de busca em largura;
- F: fila de vértices, que deve implementar as operações $F.INSERE(v)$, responsável pela inserção do vértice v no final da fila, e $F.REMOVE()$, que remove e retorna o primeiro vértice da fila F;
- $d(v)$: atributo numérico que armazena a distância de s até v em quantidade de arestas percorridas.
- $adj(v)$: é uma iteração com todos os vértices adjacentes ao vértice v (veja seção 2.2).

Algoritmo: BuscaEmLargura(G, r)

para cada $v \in V$ **faça**
    $estado(v) \leftarrow NAO\_VISITADO$;
    $\rho(v) \leftarrow nil$;
    $d(v) \leftarrow \infty$;

$d(r) \leftarrow 0$;
$estado(r) \leftarrow VISITADO$;
$F \leftarrow \emptyset$;
$F.INSERE(r)$;
enquanto $F \neq \emptyset$ **faça**
    $v_i \leftarrow F.REMOVE()$;
    para cada $v_j \in Adj(v_i)$ **faça**
        se $estado(v_j) = NAO\_VISITADO$ **então**
            $F.INSERE(v_j)$;
            $estado(v_j) \leftarrow VISITADO$;
            $\rho(v_j) \leftarrow v_i$;
            $d(v_j) \leftarrow d(v_i) + 1$;
    $cor(v_i) \leftarrow ENCERRADO$;

Algoritmo 3.1: Algoritmo de busca em largura (CORMEN et al., 2001)

3.3 Busca em profundidade

A busca em profundidade, também chamada de DFS (do inglês *Depth First Search*), pesquisa o grafo de uma forma recursiva. Ela explora o grafo procurando sempre entrar mais profundamente em sua estrutura. Ao atingir um determinado vértice, a busca visita seu vizinho, depois o vizinho do vizinho e assim sussetivamente. Quando a busca não consegue mais ir adiante, ela volta gradativamente aos passos anteriores até que consiga encontrar uma nova opção de caminho. Este procedimento de voltar atrás é comumente conhecido como *bracktracking*.

3.3.1 Busca em profundidade com um vértice origem

Assim como na busca em largura, a busca em profundidade também gera uma árvore de busca, porém ela não tem a propriedade de gerar os caminhos mínimos.

Para maior clareza, o algoritmo de busca em profundidade é desmembrado em mais de um procedimento. O primeiro deles (algoritmo 3.2) apresenta o pseudocódigo do laço principal da busca em profundidade. De forma semelhante ao algoritmo de busca em largura, podemos escolher um vértice para iniciar a busca. Este algoritmo também atinge todos os vértices possíveis de serem atingidos a partir do vértice inicial s, porém **não há garantia** de que a árvore de busca gerada contenha os menores caminhos de s até cada vértice.

Algoritmo: BuscaEmProfundidade(G, r)

para cada $v \in V$ **faça**
    $estado(v) \leftarrow NAO\_VISITADO$;
    $\rho(v) \leftarrow nil$;

$tempo \leftarrow 0$;

$VisitaVertice(r)$;

Algoritmo 3.2: Algoritmo de busca em profundidade com um vértice de origem

Os elementos do pseudocódigo são os seguintes:

- r: representa o vértice inicial;
- $estado(v)$: este atributo representa o estado de visitação do vértice v ao longo da execução do algoritmo, podendo assumir um dos seguintes valores $\{NAO\_VISITADO$, $VISITADO$, $ENCERRADO\}$. O valor $NAO\_VISITADO$ significa que a busca ainda não atingiu este vértice. O valor $VISITADO$, indica que a busca já atingiu o vértice v, mas ele ainda tem vizinhos (vértices adjacentes) ainda não visitados. O valor $ENCERRADO$ indica que a visitação deste vértice está encerrada, isto é, tanto ele como todos os seus vizinhos já foram visitados pela busca;

- $\rho(v)$: referência ao vértice predecessor do vértice v na busca, representa seu pai na árvore de busca em profundidade;
- *tempo*: Cormen et al. (2001) propõem o uso de um relógio lógico marcando a ordem de visitação dos vértices. Este dado pode ser utilizado na resolução de alguns problemas que envolvem a busca em profundidade como parte do método de resolução. O relógio lógico implementa uma técnica chamada de *time stamping*, que rotula com números inteiros uma sequência de eventos. No caso do algoritmo de busca em profundidade, estes valores variam de 0 a $2n$;
- $t_a(v)$: tempo de descoberta ou abertura do vértice v, representa o valor do tempo lógico quando o vértice é visitado pela primeira vez;
- $t_e(v)$: tempo de encerramento do vértice v, representa o valor do tempo lógico quando a visitação no vértice é encerrada.

O algoritmo 3.3 mostra o pseudocódigo da função $VisitaVertice(v)$. Trata-se de uma função recursiva que efetivamente implementa o procedimento de busca em profundidade. e que vai visitar todos os vértices que puderem ser atingidos a partir do vértice inicial r. A recursividade da função caracteriza um empilhamento dos vértices a serem visitados, com isso a ordem de visitação é feita em profundidade, sempre visitando o primeiro vértice não-visitado adjacente ao vértice atual.

Algoritmo: VisitaVertice(v_i)

$estado(v_i) \leftarrow VISITADO$;
$tempo \leftarrow tempo + 1$;
$t_a(v_i) \leftarrow tempo$;
para cada $v_j \in Adj(v_i)$ **faça**
  se $estado(v_j) = NAO\_VISITADO$ **então**
    $\rho(v_j) \leftarrow v_i$;
    $VisitaVertice(v_j)$;
$estado(v_i) \leftarrow ENCERRADO$;
$tempo \leftarrow tempo + 1$;
$t_e(v_i) \leftarrow tempo$;

Algoritmo 3.3: Função VisitaVertice(v_i)

3.3.2 Busca em profundidade com várias origens

Em alguns casos, a busca a partir de um vértice inicial r não consegue atingir todos os vértices. Isto pode acontecer em grafos dirigidos e em grafos não-conexos (capítulo 4). Nestes casos, se for necessário visitar todos os vértices, pode-se iniciar sucessivas buscas em profundidade. Sempre que uma busca encerrar, inicia-se uma nova busca a partir de um vértice ainda não visitado. O algoritmo 3.4 demonstra este procedimento.

```
Algoritmo: BuscaProfTodos(G)

para cada v ∈ V faça
    estado(v) ← NAO_VISITADO;
    ρ(v) ← nil;

tempo ← 0;

para cada vértice v ∈ V faça
    se estado(v) = NAO_VISITADO então
        VisitaVertice(v);
```

Algoritmo 3.4: Algoritmo BuscaProfTodos(G)

O laço principal da função BuscaProfTodos(G) percorre todos os vértices do grafo G, e chama VisitaVertice(v) sempre que o vértice corrente v for um vértice ainda não visitado. Ao final do laço principal, todos os vértices são visitados, e temos no vetor de roteamento uma árvore para cada uma das buscas que eventualmente foram feitas. No vetor de roteamento, os vértices que originaram as buscas são raízes das árvores, e seus predecessores têm valor *nil*.

3.4 Percorrendo a árvore de busca

Os caminhos gerados pelos algoritmos de busca, armazenados em vetores de roteamento, podem ser reconstituídos por meio do algoritmo 3.5. Note que o algoritmo é recursivo e só funciona para caminhos começando no vértice inicial utilizado no algoritmo de busca que gerou o correspondente vetor de roteamento. Tal vértice sempre é a raiz r de uma árvore de busca

```
Algoritmo: ImprimeCaminho(G, r, v)

se v=r então
    imprime(r);
senão
    se ρ(v) = nil então
        imprime("não existe caminho de r para v");
    senão
        ImprimeCaminho(G, r, ρ(v));
        imprime(v);
```

Algoritmo 3.5: Procedimento para imprimir caminhos

3.5 Exemplos de aplicação de busca

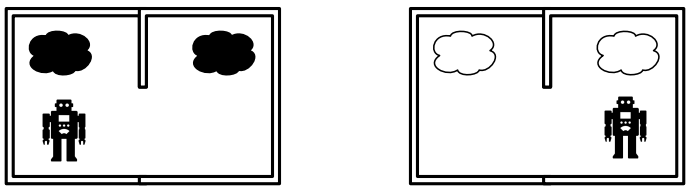
Conforme mencionado anteriormente, os algoritmos de busca em grafos podem ser utilizados diretamente na resolução de determinados problemas, mas também podem servir de ponto de partida para algoritmos mais especializados que precisem caminhar pelo grafo passando por arestas. A seguir, apresentamos dois estudos de caso que aplicam busca em grafos.

3.5.1 Estudo de caso: busca em grafos de estados

Um grafo de estados $G = (V, E)$ representa nas quais as transições entre possíveis estados de um problema são importantes. Um exemplo clássico na Ciência da Computação são as máquinas de estados finitos, que podem ser modeladas por tipos especiais de grafos chamados Autômatos Finitos Determinísticos (HOPCROFT; MOTWANI; ULLMAN, 2006). Um outro exemplo bastante popular é a programação de tarefas de um robô. Por exemplo, de forma bem simplificada, suponha que um robô aspirador de pó precisa limpar uma área dividida em dois cômodos: o quarto do lado direito (q_d) e o quarto esquerdo (q_e). O robô executa apenas três comandos: ir para o quarto esquerdo, ir para o quarto direito e limpar o quarto atual. Considerando que o robô se encontra inicialmente no quarto da esquerda e ambos os quartos estão sujos, nossa tarefa é montar um plano para que o robô limpe os dois quartos e, ao final, se posicione no quarto da direita.

Vamos montar este plano por meio de um grafo de estados. Cada vértice representa um estado do problema e cada aresta modela uma possível transição entre dois estados. Um estado do problema é caracterizado pela condição atual de limpeza de cada quarto (limpo ou sujo) e pela posição atual do robô (quarto esquerdo ou quarto direito). Podemos representar os vértices por uma tupla de 3 valores (Q_e, Q_d, P_r), sendo Q_e e Q_d os conjuntos de estados de limpeza dos dois quartos, ou seja, com $Q_e = \{l, s\}$ e $Q_e = \{l, s\}$. P_r é o conjunto de possíveis localizações do robô, $P_r = \{e, d\}$. O estado inicial do problema, portanto, é representado pela tupla (s, s, e) e o estado final pela tupla (l, l, d), conforme ilustrado na Figura 3.2.

Figura 3.2 – Estados inicial e final do problema

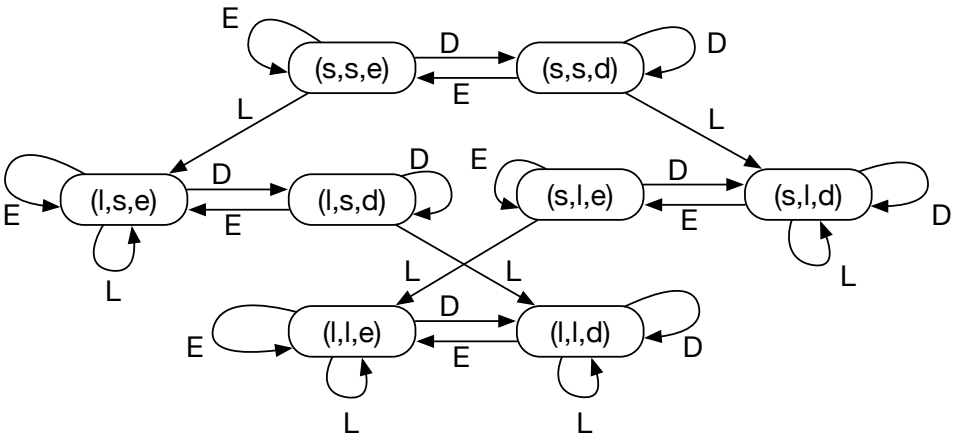


(a) Estado inicial (s, s, e) (b) Estado final (l, l, d)

Fonte: o autor.

Aqui surge uma questão interessante sobre a modelagem de problemas com grafos: nosso problema será modelado como um grafo dirigido ou não-dirigido? Na verdade, isso é decorrente da natureza das relações entre os elementos (vértices) do problema. Neste caso os únicos fatores que alteram o estado do problema e, consequentemente, serão as arestas do garfo, são os comandos aceitos pelo robô: ir para o quarto esquerdo, ir para o quarto direito e limpar. Como não existe o comando para devolver a sujeira, uma vez que o quarto foi limpo ele não volta mais ao estado sujo. Isso nos indica que se trata de um grafo dirigido, que pode ser visto na Figura 3.3. Os rótulos E, D e L das arestas correspondem aos comandos "ir para a esquerda", "ir para a direita" e limpar, respectivamente.

Figura 3.3 – Grafo de estados do robô



Fonte: o autor.

Finalmente, para encontrar uma sequência de comandos para montar o plano de atuação do robô, basta efetuar uma busca no grafo, iniciando pelo vértice (s, s, e). A resposta pode ser encontrada percorrendo-se a árvore de busca para determinar a sequência de comandos que gera as transições de estado até o estado final. Por exemplo: $(s, s, e) \rightarrow$ limpa $\rightarrow (l, s, e) \rightarrow$ direita $\rightarrow (l, s, d) \rightarrow$ limpa $\rightarrow (l, l, d) \rightarrow$ esquerda $\rightarrow (l, l, e)$.

Note que em problemas deste tipo, tanto a busca em largura quanto a busca em profundidade podem ser utilizadas, entretanto, por garantir encontrar os caminhos com menor número de arestas, a busca em largura garante que a solução use a menor quantidade possível de transições de estado.

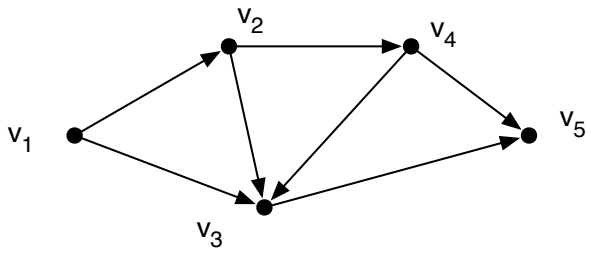
3.5.2 Estudo de caso: ordenação topológica

A ordenação topológica em um grafo acíclico dirigido é uma ordenação linear de vértices tal que para cada aresta dirigida (v_i, v_j), o vértice v_i vem antes de v_j na ordenação. Este tipo de ordenação só é possível em grafos acíclicos dirigidos.

Definição 3.1 (Grafo acíclico dirigido)**.** *Um grafo acíclico dirigido é um grafo dirigido sem ciclos dirigidos fechados.*

A Figura 3.4 mostra um exemplo de grafo acíclico dirigido. Note que se caminharmos pelo grafo saindo de um vértice v arbitrário, nunca conseguimos retornar para v.

Figura 3.4 – Exemplo de grafo acíclico dirigido



Fonte: o autor.

Uma ordenação topológica pode ser vista como uma organização linear dos vértices de um grafo de tal forma que vértices subsequentes dependem dos vértices precedentes, com estas relações de dependência modeladas pelas arestas. Isto é particularmente útil quando o grafo modela conjuntos interdependentes de atividades, como nos diagramas PERT, usados para planejamento e controle de projetos (KERZNER, 2005).

O algoritmo para ordenação topológica utiliza a busca em profundidade, e pode ser descrito pelos seguintes passos:

Entrada: um grafo acíclico dirigido G.

Passos:

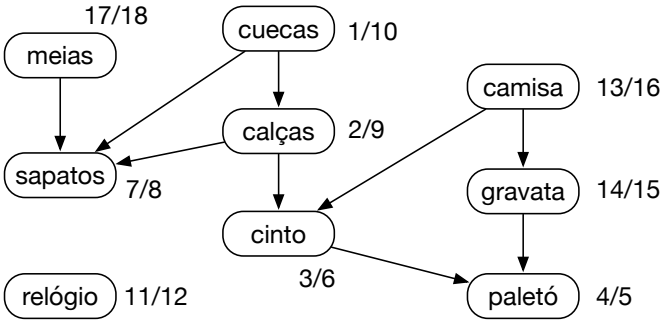
1. Execute a busca em profundidade no grafo — $BuscaProfTodos(G)$;
2. À medida que cada vértice for encerrado (isto é, quando for computado seu tempo de encerramento $t_e(v)$), insira o vértice no início de uma lista encadeada;
3. Retorne a lista encadeada de vértices.

Saída: uma lista encadeada ordenada de vértices.

Para ilustrar um exemplo de ordenação topológica, vamos organizar a sequência de passos para um homem se vestir com terno e gravata, incluindo sapatos e relógio. No grafo acíclico dirigido da Figura 3.5 os vértices são as peças de roupa e as arestas mostram as relações de dependência entre as peças. Por exemplo, só se pode calçar os sapatos depois de calçar as meias. Por isso, existe uma aresta dirigida ligando o vértice "meias" até o vértice "sapatos" indicando que os "sapatos" dependem das "meias". Os números próximos

a cada vértice representam os seus tempos de abertura e fechamento em uma execução da busca em profundidade.

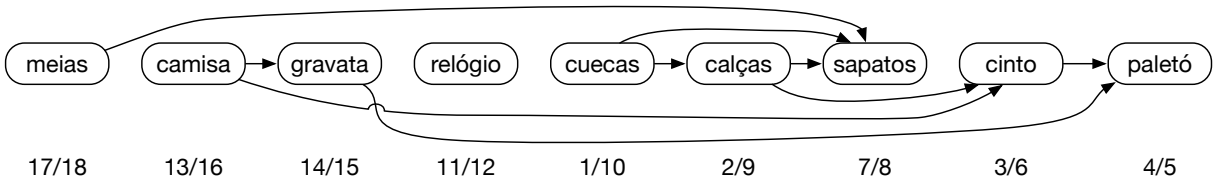
Figura 3.5 – Grafo acíclico dirigido



Fonte: adaptado de Cormen et al. (2001).

A Figura 3.6 mostra a ordenação topológica obtida com a busca em profundidade realizada. A sequência de vértices encontrada é decorrente da ordem de visitação dos vértices, que em última instância, é também função de como os vértices estão dispostos nas estruturas de dados que modelam o grafo. Note que, por mais incomum que possa parecer esta sequência de passos para uma pessoa se vestir, ela é semanticamente correta, conforme evidenciado pelas arestas dirigidas na Figura 3.6.

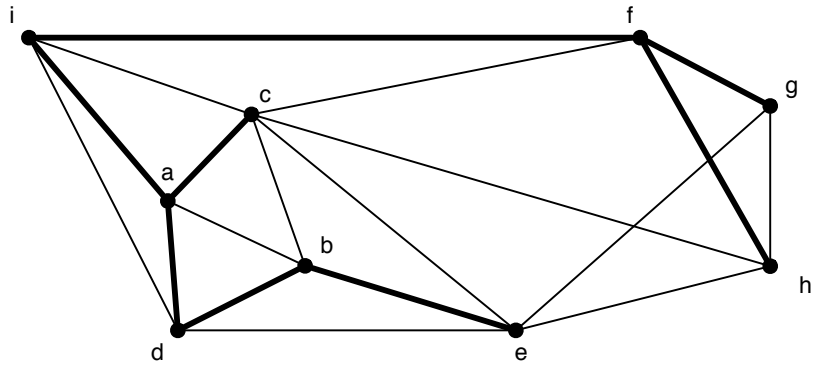
Figura 3.6 – Resultado da ordenação topológica



Fonte: o autor.

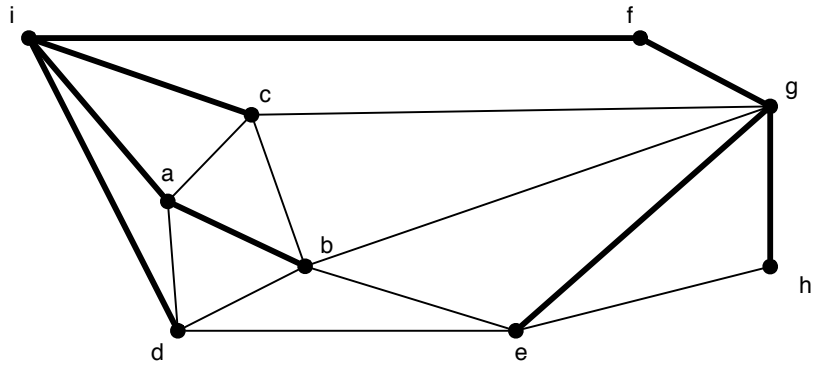
3.6 Exercícios

1. A figura abaixo mostra uma árvore de busca contendo caminhos do vértice a para todos os demais vértices. Preencha a tabela abaixo com os respectivos valores do vetor de roteamento que representa a árvore.



v	a	b	c	d	e	f	g	h	i
$\rho(\mathbf{v})$									

2. A figura abaixo mostra uma árvore de busca contendo caminhos do vértice f para todos os demais vértices. Preencha a tabela abaixo com os respectivos valores do vetor de roteamento que representa a árvore.



v	a	b	c	d	e	f	g	h	i
$\rho(\mathbf{v})$									

3. Dado o vetor de roteamento abaixo, escreva a sequência de vértices de cada caminho (se existir caminho):

v	a	b	c	d	e	f	g	h	i	j
ρ**(v)**	b	c	d	e	h	d	nil	g	j	h

   a) Caminho $g \rightsquigarrow c$: _____

   b) Caminho $b \rightsquigarrow f$: _____

   c) Caminho $g \rightsquigarrow i$: _____

4. Dado o vetor de roteamento abaixo, escreva a sequência de vértices de cada caminho (se existir caminho):

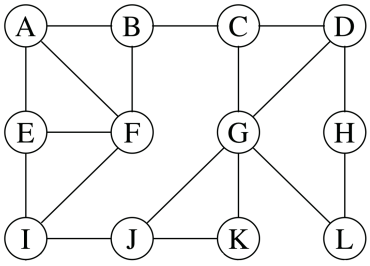
v	a	b	c	d	e	f	g	h	i	j	f	l	m	n
ρ**(v)**	j	j	f	i	h	nil	m	i	nil	f	j	k	h	g

   a) Caminho $i \rightsquigarrow n$: _____

   b) Caminho $f \rightsquigarrow k$: _____

   c) Caminho $g \rightsquigarrow h$: _____

5. A figura abaixo mostra um grafo não dirigido, e um vetor de roteamento gerado neste grafo ao se executar um algoritmo de busca.



v	a	b	c	d	e	f	g	h	i	j	f	l
ρ**(v)**	nil	a	b	c	i	e	d	l	j	g	l	g

   a) A árvore foi gerada por busca em largura ou profundidade? _____

   b) Qual o caminho $a \rightsquigarrow e$? _____

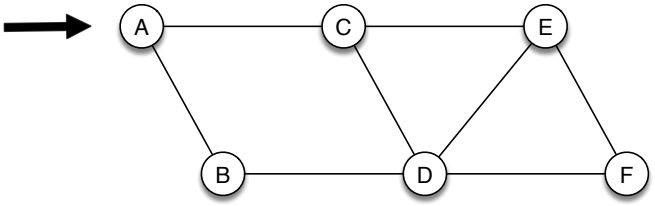
   c) Qual o comprimento do caminho $a \rightsquigarrow e$? _____

6. Elabore um algoritmo para verificar se um grafo possui ciclos.

7. Implemente o algoritmo de busca em largura (algoritmo 3.1).

8. Implemente o algoritmo de busca em profundidade (algoritmo 3.2 e algoritmo 3.3).

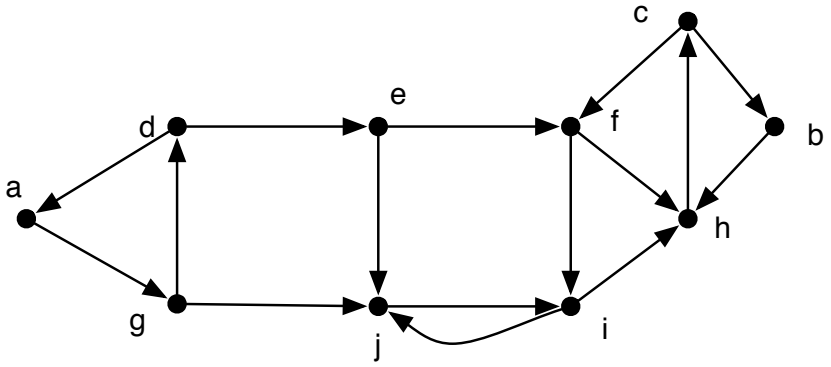
9. (PosComp – 2009) Considere o algoritmo de busca em largura em grafos. Dado o grafo a seguir e o vértice A como ponto de partida, a ordem em que os vértices são descobertos é dada por:



a) A B C D E F
b) A B D C E F
c) A C D B F E
d) A B C E D F
e) A B D F E C

10. Dado o grafo G dirigido abaixo, mostre passo a passo a execução de $BuscaProfTodos(G)$, considerando que as listas de adjacência estão em ordem alfabética:

a) Mostre a sequencia de chamadas a $VisiteVertice$;
b) Mostre os tempos de abertura e encerramento de todos os vértices no grafo G;
c) Mostre o vetor de roteamento final.



11. (PosComp – 2011) Seja G um grafo conexo com n vértices. Considere duas rotulações dos vértices de G obtidas por duas buscas em G, uma em largura, $l()$, e outra em profundidade, $p()$, ambas iniciadas no vértice v. Em cada rotulação, os vértices receberam um número de 1 a n, o qual representa a ordem em que foram alcançados na busca em questão. Assim, $l(v) = p(v) = 1$; enquanto $l(x) > 1$ e $p(x) > 1$ para todo vértice x diferente de v. Considere dois vértices u e w de G e denote por $d(u, w)$ a distância em G de u até w. Com base nesses dados, assinale a alternativa correta.

a) Se $l(u) < l(w)$ e $p(u) < p(w)$, então $d(v, u) < d(v, w)$.
b) Se $l(u) < l(w)$ e $p(u) > p(w)$, então $d(v, u) = d(v, w)$.
c) Se $l(u) > l(w)$ e $p(u) < p(w)$, então $d(v, u) \leq d(v, w)$.
d) Se $l(u) > l(w)$ e $p(u) > p(w)$, então $d(v, u) < d(v, w)$.
e) Se $l(u) < l(w)$ e $p(u) > p(w)$, então $d(v, u) \leq d(v, w)$.

12. (PosComp – 2015) Centenas de problemas computacionais são expressos em termos de grafos, e os algoritmos para resolvê-los são fundamentais para a computação. O algoritmo de busca em:

    a) largura utiliza pilha, enquanto o de busca em profundidade utiliza fila.
    b) largura á o responsável pela definição do vértice inicial.
    c) profundidade é utilizado para obter uma ordenação topológica em um dígrafo acíclico.
    d) largura explora as arestas a partir do vértice mais recentemente visitado.
    e) profundidade expande a fronteira entre vértices conhecidos e desconhecidos uniformemente.

13. Faça a ordenação topológica do dígrafo abaixo. Utilize o algoritmo de busca em profundidade e mostre a lista de vértices ordenados.