CAPÍTULO

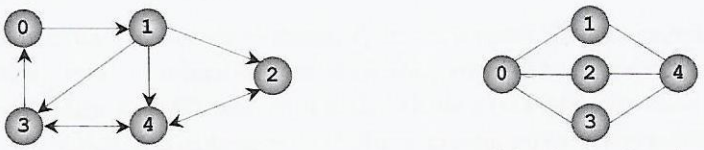
15

GRAFOS

Será introduzido agora o conceito da estrutura de dados grafo e dois exemplos interessantes de aplicação prática dessa estrutura serão usados: busca de caminhos e ordenação topológica.

15.1 Fundamentos

Um *grafo orientado* G é uma estrutura composta de um conjunto finito de *vértices* V e um conjunto de *arestas* $A \subseteq V \times V$. Por exemplo, na Figura 15.1a, os vértices do grafo orientado são os círculos e suas arestas são as setas entre os círculos.



(a) grafo orientado **(b)** grafo não orientado

Figura 15.1 | Exemplos de grafos.

Sejam v e w dois vértices em um grafo orientado G. Uma aresta que vai de v para w, denotada por $v \rightarrow w$, indica que w é *adjacente* a v ou que w é *sucessor* de v. Por exemplo, no grafo da Figura 15.1a, 0 é sucessor de 3, mas 3 não é sucessor de 0. Uma aresta $v \rightarrow v$ é um *laço*. Um vértice que não tem sucessores é um *beco*.

Um *caminho* de v para w em G é uma sequência $[v_0, v_1, ..., v_k]$ tal que:

- $v_0 = v$, ou seja, o primeiro vértice da sequência é v.
- $v_k = w$, ou seja, o último vértice da sequência é w.
- $v_0 \rightarrow v_1 \rightarrow ... \rightarrow v_k$, *isto é, entre cada dois vértices consecutivos na sequência, há uma aresta em G que parte do primeiro e chega ao segundo.*

Por exemplo, no grafo da Figura 15.1a, a sequência [0,1,3] é um caminho de 0 para 3, mas não é um caminho de 3 para 0. Por outro lado, a sequência [3,0] é um caminho de 3 para 0, mas não é um caminho de 0 para 3.

Um *ciclo* é um caminho que inicia e termina num mesmo vértice. Por exemplo, no grafo da Figura 15.1a, o caminho [0,1,3,0] é um ciclo. Um grafo que não tem ciclos é um grafo *acíclico*.

Um grafo orientado G é *simétrico* se, para toda aresta $v \rightarrow w$ em G, há uma aresta $w \rightarrow v$. Um grafo simétrico e sem laços é chamado *grafo não orientado*. Como toda aresta num grafo não orientado é bidirecional, o sentido das arestas não precisa ser indicado. Por exemplo, a Figura 15.1b apresenta um grafo não orientado. O termo *grafo* pode ser usado para designar qualquer tipo de grafo.

Há diversas aplicações de grafos em computação. Nesse capítulo, veremos duas dessas aplicações: busca de caminhos e ordenação topológica.

15.1.1 Representação de grafos

Há duas formas básicas de representar um grafo $G = (V, A)$: *matriz de adjacências* e *listas de adjacências*, como mostra a Figura 15.2.

(a) grafo orientado **(b)** matriz de adjacências **(c)** listas de adjacências

Figura 15.2 | Formas de representar de um grafo.

A representação por matriz de adjacências consiste em uma matriz quadrada *Adj*, de ordem $|V|$, tal que $Adj\ [v, w] = 1$ se existe uma aresta $v \rightarrow w$ em G; caso contrário, $Adj\ [v, w] = 0$. Essa representação é preferível quando o grafo é *denso* (isto é, quando $|A|$ é próximo de $|V|^2$).

A representação por listas de adjacências consiste em um vetor *Adj* com $|V|$ listas encadeadas. Para cada vértice $v \in G$, a lista encadeada apontada por $Adj\ [v]$ contém todos os sucessores de v. Essa representação é preferível quando o grafo é *esparso* (isto é, quando $|A|$ é muito menor que $|V|^2$). Nos algoritmos apresentados a seguir, usaremos apenas listas de adjacências.

Por exemplo, usando a função `no()` definida no Capítulo 9, podemos representar o grafo da Figura 15.2 do seguinte modo:

```
Lista Adj[] = { no(1,NULL),
                no(2,no(3,no(4,NULL))),
                no(4,NULL),
                no(0,no(4,NULL)),
                no(2,no(3,NULL)) };
```

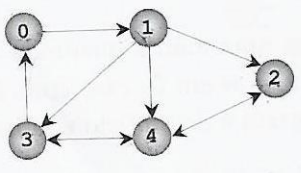
15.2 Busca de caminhos

Dados um grafo G, um vértice de *origem* v e um vértice de *destino* w, o problema de *busca de caminhos* consiste em enumerar todos os caminhos de v para w em G. Por exemplo, se G é grafo na Figura 15.3a, a origem é $v = 0$ e o destino é $w = 2$, os caminhos possíveis são [0,1,2], [0,1,3,4,2] e [0,1,4,2].

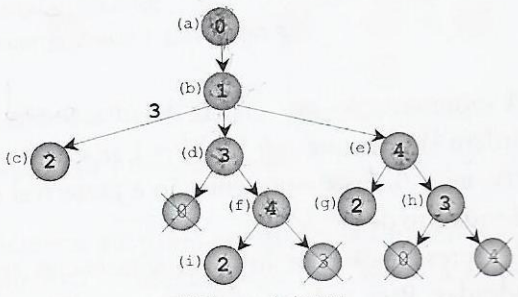
15.2.1 Árvore de busca

Dados um grafo G, uma origem v e um destino w, uma *árvore de busca* enumera todos os caminhos de v para w. Cada nó em uma árvore de busca é um vértice de G. A raiz da árvore de busca é o vértice de origem v. Além disso, para cada nó u na árvore de busca, se u é igual a w, então u é uma folha na árvore; senão, o nó u é *expandido* com todos os sucessores de u em G, que não estão no caminho que vai da raiz da árvore até o nó u (para evitar ciclos). Se todos os sucessores de u em G aparecem no caminho que vai da raiz da árvore até o nó u, então o nó u também é uma folha na árvore. Assim, toda folha na árvore de busca representa um *caminho* de v até w ou, então, um *beco*.

Por exemplo, a Figura 15.3b mostra a árvore de busca que enumera todos os caminhos de $v = 0$ para $w = 2$, no grafo da Figura 15.3a. Os nós marcados com "×" não fazem parte da árvore (pois causam ciclos). Portanto, a árvore tem apenas quatro folhas: três representam caminhos e uma representa um beco.



(a) grafo orientado



(b) árvore de busca

Figura 15.3 | Árvore de busca de caminhos.

15.2.2 Busca em largura

Busca em largura é um algoritmo que gera uma árvore de busca usando uma fila. Na fila, cada item é uma lista representando uma *folha* da árvore que está sendo gerada (Figura 15.4). A fila garante que as folhas sejam *expandidas* na mesma ordem em que são geradas (isto é, que a árvore seja gerada em *largura*). A lista permite que o caminho da raiz até uma folha seja guardado na própria folha.

Inicialmente, uma fila vazia é criada e uma lista com o vértice de origem v é inserida nela (Figura 15.4a). A partir daí, é iniciada uma repetição que só para quando a fila fica vazia. Em cada iteração, uma lista $c = [c_0, c_1, ..., c_k]$ é removida da fila e seu primeiro item c_0 é comparado ao vértice de destino w. Então, se $c_0 = w$, a lista inversa de c é exibida como um caminho de v para w em G; senão, para cada sucessor s de c_0 em G, uma lista $[s,c_0,c_1,...,c_k]$ é inserida na fila.

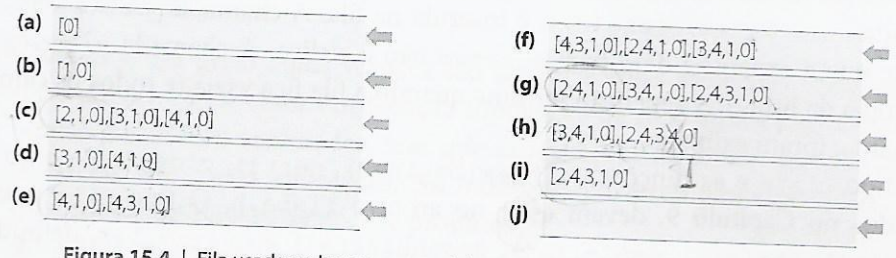


Figura 15.4 | Fila usada na busca por caminhos de 0 para 2, no grafo da Figura 15.2.

Por exemplo, quando a lista [0] é removida da fila (Figura 15.4a), a lista [1,0] é inserida (pois 1 é sucessor de 0 no grafo da Figura 15.3). Analogamente, quando a lista [1,0] é removida (Figura 15.4b), são inseridas as listas [2,1,0], [3,1,0] e [4,1,0] (pois 2, 3 e 4 são sucessores de 1). Por outro lado, quando a lista [2,1,0] é removida (Figura 15.4c), sua inversa é exibida como um caminho possível (e nada é inserido na fila). Analogamente, quando a lista [3,4,1,0] é removida da fila (Figura 15.4h), nada é inserido na fila (pois essa folha é um beco). Note que a ordem em que as listas são removidas da fila na Figura 15.4 é a mesma em que os nós da árvore estão marcados na Figura 15.3 (isto é, em largura).

A busca em largura (ou *breadth-first search*) é implementada na Figura 15.5.

```
void bfs(int v, int w, Lista Adj[]) {
   Fila F = fila(100); // tamanho da fila depende do grafo!
   enfileira(no(v,NULL),F);
   while( !vaziaf(F) ) {
      Lista c = desenfileira(F);
      if( c->item==w ) {
         exibe_inv(c);
         puts("");
      }
```

```
      else
         for(Lista s = Adj[c->item]; s; s = s->prox)
            if( !pert(s->item,c) ) enfileira(no(s->item,clone(c)),F);
      destroi(&c);
   }
   destroif(&F);
}
```

Figura 15.5 | Função para busca em largura.

A função `bfs()` cria uma fila vazia com capacidade para `100` listas e insere nela a lista com o vértice de origem `v`, criada com a chamada `no(v,NULL)`. Dentro da repetição, uma lista `c` é removida da fila. Então, se seu primeiro item (`c->item`) é igual ao vértice de destino `w`, a lista inversa de `c` é exibida no vídeo; senão, para cada sucessor `s` de `c->item`, na lista de adjacências `Adj[c->item]`, uma expansão de `c`, criada com a chamada `no(s->item,clone(c))`, é inserida na fila. A chamada `pert(s->item,c)` garante que a expansão de `c` com `s->item` não é cíclica. A chamada `clone(c)` cria uma cópia da lista `c`. A repetição termina quando a fila fica vazia (e todos os caminhos de `v` para `w` foram exibidos).

O tipo `Lista` e as funções `no()`, `exibe_inv()`, `pert()`, `clone()` e `destroi()`, definidos no Capítulo 9, devem estar no arquivo `lista.h`. O tipo `Fila` e as funções `fila()`, `vaziaf()`, `enfileira()`, `desenfileira()` e `destroif()`, definidos no Capítulo 4, devem estar no arquivo `fila.h`. Esses arquivos devem ser incluídos no programa, como indicado na Figura 15.6.

```
#include <stdio.h>
#include "../ed/lista.h" // lista de int
#include "../ed/fila.h"  // fila de Lista
// adicione a função bfs(), da Figura 15.5
int main(void) {
   Lista Adj[] = { no(1,NULL),
                   no(2,no(3,no(4,NULL))),
                   no(4,NULL),
                   no(0,no(4,NULL)),
                   no(2,no(3,NULL)) };
   bfs(0,2,Adj);
   return 0;
}
```

Figura 15.6 | Programa que exibe todos os caminhos de 0 para 2, no grafo da Figura 15.3.

Uma propriedade importante da busca em largura é que ela garante que os caminhos são exibidos em ordem crescente de tamanho (isto é, o primeiro caminho exibido tem um número mínimo de arestas). Então, se apenas um caminho de *comprimento mínimo* for desejado, o algoritmo pode ser facilmente modificado para terminar a busca assim que o primeiro caminho for encontrado.

15.3 Ordenação topológica

Uma *ordenação topológica* de um grafo orientado *acíclico* G é uma permutação π dos vértices de G tal que, se existe uma aresta $v \rightarrow w$ em G, então o vértice v aparece antes de w em π. Por exemplo, há três ordenações topológicas possíveis para o grafo da Figura 15.7: [0,3,1,2,4], [0,1,3,2,4] e [0,1,2,3,4].

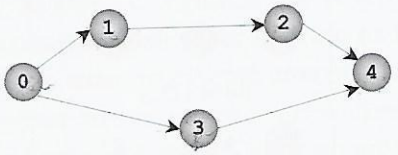


Figura 15.7 | Grafo orientado acíclico.

15.3.1 Redes de tarefas

Rede de tarefas é um grafo cujos vértices são *tarefas* e cujas arestas são *restrições de ordem* entre tarefas. Nesse caso, a ordenação topológica da rede indica uma ordem em que as tarefas podem ser executadas, sem que as restrições de ordem sejam violadas. Redes de tarefas têm várias aplicações. Na indústria, por exemplo, elas servem para definir a ordem de montagem de peças e produtos.

A Figura 15.8 mostra uma rede de tarefas bem intuitiva que especifica, por exemplo, que as meias devem ser calçadas *antes* dos sapatos (essa restrição de ordem é imposta pela aresta *meias* → *sapatos*). É fácil ver que uma rede de tarefas é um grafo orientado *acíclico* (pois a execução de uma tarefa não pode depender de si mesma). Há 362.880 permutações possíveis dos 9 vértices dessa rede, mas apenas 1.728 delas obedecem às restrições impostas por suas arestas.

(a) rede de tarefas **(b)** grafo orientado acíclico

Figura 15.8 | Rede de tarefas e grafo orientado acíclico correspondente.

15.3.2 Geração de uma ordenação topológica

Para gerar *uma* ordenação topológica de um grafo orientado acíclico G, basta remover um vértice v de G que não tenha *predecessores* (as arestas da forma $v \rightarrow^*$ também devem ser removidas de G). Por exemplo, para o grafo na Figura 15.8b, o primeiro vértice escolhido pode ser 2, 3, 6 ou 7 (note que, caso o vértice 3 seja escolhido, as opções de escolha na próxima etapa serão os vértices 0, 2, 6 e 7). O vértice v removido pode, então, ser exibido como

primeiro vértice da permutação desejada. A partir daí, o processo pode ser repetido de forma análoga para o grafo resultante, até que todo vértice de *G* tenha sido exibido. O processo completo é ilustrado na Figura 15.9.

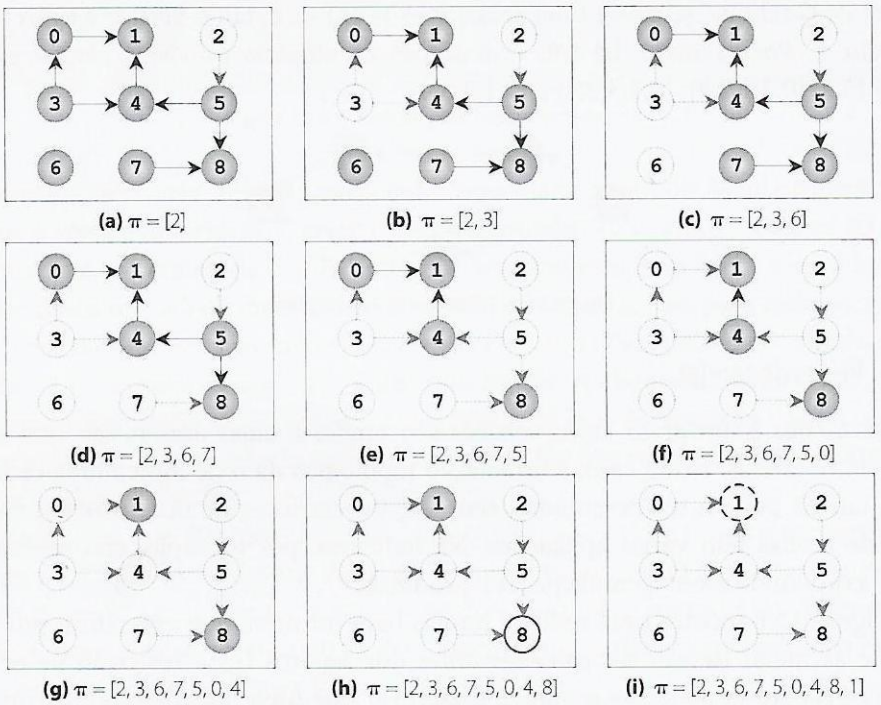


Figura 15.9 | Processo para geração de uma ordenação topológica.

A ordenação topológica (ou *topological sort*) é implementada na Figura 15.10.

```
void ts(Lista Adj[], int n) {
   int pred[n];
   for(int i=0; i<n; vi++) {
      pred[i] = 0;
      for(int j=0; j<n; j++)
         if( pert(i,Adj[j]) )
            pred[i]++;
   }
   Fila F = fila(n);
   for(int v=0; v<n; v++)
      if( pred[v]==0 )
         enfileira(v,F);

   while( !vaziaf(F) ) {
      int v = desenfileira(F);
      printf("%d ",v);
      for(Lista s = Adj[v]; s; s = s->prox) {
         pred[s->item]--;
         if( pred[s->item]==0 )
            enfileira(s->item,F);
      }
   }
}
```

Figura 15.10 | Função para ordenação topológica.

A função `ts()` inicia criando o vetor `pred`, que indica o total de predecessores de cada vértice do grafo representado por `Adj`. Em seguida, ela cria uma fila contendo todo vértice `v` que não tem predecessor no grafo (`pred[v]==0`). A partir daí, é iniciada uma repetição que só para quando a fila fica vazia. Em cada iteração, o vértice `v` removido da fila é exibido como próximo elemento da permutação sendo gerada e, para cada sucessor `s->item` de `v` na lista `Adj[v]`, o contador `pred[s->item]` é decrementado. Então, se `pred[s->item]` se torna `0`, significa que o vértice `s->item` não tem mais predecessores no grafo e, portanto, ele deve ser inserido na fila. Quando a repetição termina, a permutação exibida no vídeo representa uma ordenação topológica do grafo.

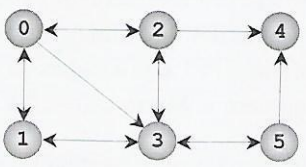
O tipo `Fila` e as funções `fila()`, `vaziaf()`, `enfileira()`, `desenfileira()` e `destroif()`, definidos no Capítulo 4, devem estar no arquivo `fila.h`. O tipo `Lista` e a função `no()`, definidos no Capítulo 9, devem estar no arquivo `lista.h`. Esses arquivos devem ser incluídos no programa, como na Figura 15.11.

```
#include <stdio.h>
#include "../ed/lista.h" // lista de int
#include "../ed/fila.h"  // fila de int
// adicione a função ts(), da Figura 15.10
int main(void) {
   Lista Adj[] = { no(1,NULL),
                   NULL,
                   no(5,NULL),
                   no(0,no(4,NULL)),
                   no(1,NULL),
                   no(4,no(8,NULL)),
                   NULL,
                   no(8,NULL),
                   NULL };
   ts(Adj,9);
   return 0;
}
```

Figura 15.11 | Programa que exibe uma ordenação topológica para o grafo da Figura 15.8.

Exercícios

15.1 Codifique a representação do grafo a seguir usando listas de adjacências:



15.2 Explique por que é possível garantir que o primeiro caminho encontrado pela busca em largura tem comprimento mínimo.

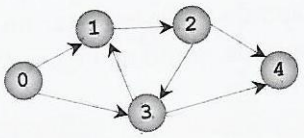
15.3 Usando a função `bfs()`, crie um programa para exibir todos os caminhos que vão do vértice `0` ao vértice `5` no grafo do Exercício 15.1.

15.4 Na busca em largura, as folhas da árvore de busca são guardadas numa fila. Quando uma pilha é usada, em vez de uma fila, passamos a ter um novo algoritmo conhecido como *busca em profundidade* (ou *depth-first search*).

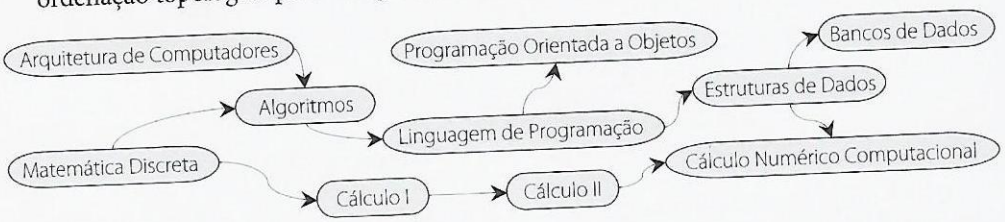
(a) Crie uma nova versão da função `bfs()`, chamada `dfs()`, para fazer busca em profundidade e faça um programa para testá-la.

(b) Há alguma garantia de que o primeiro caminho encontrado pela busca em profundidade tenha comprimento mínimo?

15.5 Explique por que não há uma ordenação topológica para o grafo a seguir:



15.6 O grafo a seguir representa os pré-requisitos que um aluno deve ter para cursar algumas disciplinas do curso de Ciência da Computação. Simule o funcionamento do algoritmo de ordenação topológica para este grafo, desenhando cada etapa do processo.



15.7 O algoritmo de ordenação topológica pode ser alterado para informar tarefas que podem ser executadas em paralelo. Usando essa nova versão do algoritmo, poderíamos, por exemplo, descobrir que disciplinas no grafo do Exercício 15.6 podem ser feitas em um mesmo semestre. Altere a função `ts()`, de modo que ela possa produzir uma saída como indicado a seguir:

```
1: Arquitetura de Computadores, Matemática Discreta,
2: Algoritmos, Cálculo I,
3: Linguagem de Programação, Cálculo II,
4: Programação Orientada a Objetos, Estruturas de Dados,
5: Cálculo Numérico Computacional, Bancos de Dados,
```

Dica: Use um grafo cujos vértices são representados por números naturais consecutivos, a partir de `0`. Para exibir os nomes das disciplinas, em vez dos números dos vértices, crie um vetor `d[n]` de cadeias, tal que `d[v]` seja a cadeia de caracteres que descreve disciplina representada pelo vértice de número `v`. Na hora de exibir o vértice `v`, exiba a cadeia `d[v]`.

15.8 Modifique a função `ts()` para que ela informe quando o grafo cuja ordenação topológica é desejada (representado pelo vetor `Adj`) não for *acíclico*. *Dica*: quando o grafo não é acíclico, a fila fica vazia antes que todos os seus vértices tenham sido exibidos.