Capítulo 12

Grafos

Neste capítulo, estudamos duas representações de grafos e algoritmos básicos que usam essas representações.

Matematicamente, um *grafo (direcionado)* é um par $G = (V, A)$ onde V é um conjunto de *vértices* e A é um conjunto de pares ordenados de vértices chamados *arestas*. Uma aresta (i, j) é *direcionado* de i para j; i é chamado de *fonte* da aresta e j é chamado de *alvo*. Um *caminho* em G é uma sequência de vértices $v_0, \ldots, v_k$ tal que, para cada $i \in \{1, \ldots, k\}$, a aresta (v_{i-1}, v_i) está em A. Um caminho $v_0, \ldots, v_k$ é um *ciclo* se, além disso, a aresta (v_k, v_0) está em A. Um caminho (ou ciclo) é *simples* se todos os seus vértices forem únicos. Se houver um caminho de algum vértice v_i para algum vértice v_j então nós dizemos que v_j está ao *alcance* de v_i. Um exemplo de grafo é mostrado em Figura 12.1.

Devido à sua capacidade de modelar tantos fenômenos, os grafos têm um enorme número de aplicações. Existem muitos exemplos óbvios. Redes de computadores podem ser modeladas como grafos, com vértices correspondendo a computadores e arestas correspondendo a links de comunicação (direcionados) entre esses computadores. As ruas da cidade podem ser modeladas como grafos, com vértices representando interseções e bordas representando ruas que unem interseções consecutivas.

Exemplos menos óbvios ocorrem assim que percebemos que os grafos podem modelar quaisquer relações de pares dentro de um conjunto. Por exemplo, em um ambiente universitário, podemos ter um *grafo de conflito* de horário cujos vértices representam cursos oferecidos na universidade e nos quais a aresta (i, j) está presente se e somente se houver pelo menos

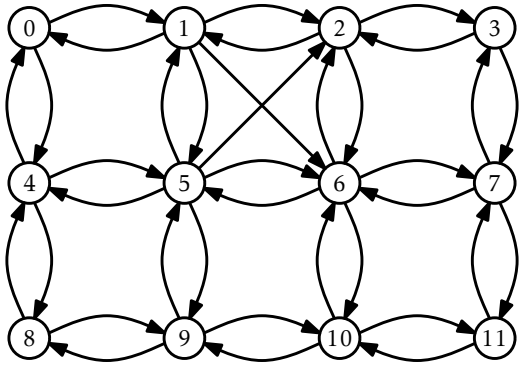


Figura 12.1: Um grafo com doze vértices. Os vértices são desenhados como círculos numerados e as arestas são desenhadas como curvas pontiagudas apontando da origem ao destino.

um aluno que está cursando as aulas i e a classe j. Assim, uma aresta indica que a prova da classe i não deve ser agendado ao mesmo tempo que a prova da classe j.

Ao longo desta seção, usaremos n para denotar o número de vértices de G e m para denotar o número de arestas de G. Ou seja, $n = |V|$ e $m = |A|$. Além disso, assumiremos que $V = \{0, \ldots, n-1\}$. Quaisquer outros dados que gostaríamos de associar aos elementos de V podem ser armazenados em um array de comprimento n.

Algumas operações típicas realizadas em grafos são:

- add_edge(i,j): Adicione a aresta (i,j) a A.
- remove_edge(i,j): Remova a aresta (i,j) de A.
- has_edge(i,j): Checa se a aresta $(i,j) \in A$
- out_edges(i): Retorna uma List de todos os inteiros j tal que $(i,j) \in A$
- in_edges(i): Retorna uma List de todos os inteiros j tal que $(j,i) \in A$

Observe que essas operações não são terrivelmente difíceis de implementar com eficiência. Por exemplo, as três primeiras operações podem ser implementadas diretamente usando um USet, para que possam ser implementadas em tempo constante esperado usando as tabelas de hash

discutidas no Capítulo 5. As duas últimas operações podem ser implementadas em tempo constante armazenando, para cada vértice, uma lista de seus vértices adjacentes.

No entanto, diferentes aplicativos de grafos têm diferentes requisitos de desempenho para essas operações e, idealmente, podemos usar a implementação mais simples que satisfaça todos os requisitos do aplicativo. Por esse motivo, discutimos duas grandes categorias de representações gráficas.

12.1 AdjacencyMatrix: Representando um grafo por uma matriz

Uma *matriz adjacência* é uma forma de representar um grafo de n vértices $G = (V, A)$ por uma matriz $n \times n$, a, cujas entradas são valores booleanos.

```
initialize()
    a ← new_boolean_matrix(n, n)
```

A entrada da matriz $a[i][j]$ é definido como

$$a[i][j] = \begin{cases} true & \text{se } (i,j) \in A \\ false & \text{caso contrário} \end{cases}$$

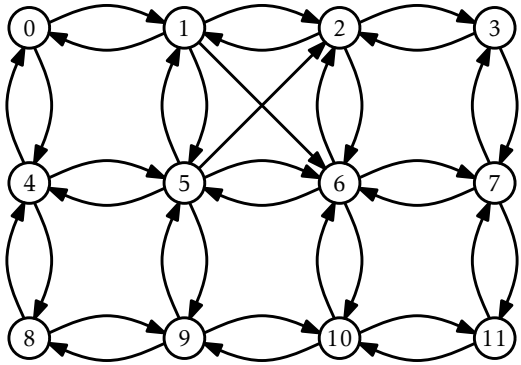
A matriz de adjacência para o grafo da Figura 12.1 é mostrada em Figura 12.2.

Nesta representação, as operações add_edge(i,j), remove_edge(i,j) e has_edge(i,j) envolvem apenas definir ou ler a entrada da matriz $a[i][j]$:

```
add_edge(i, j)
    a[i][j] ← true

remove_edge(i, j)
    a[i][j] ← false

has_edge(i, j)
```



	0	1	2	3	4	5	6	7	8	9	10	11
0	0	1	0	0	1	0	0	0	0	0	0	0
1	1	0	1	0	0	1	1	0	0	0	0	0
2	1	0	0	1	0	0	1	0	0	0	0	0
3	0	0	1	0	0	0	0	1	0	0	0	0
4	1	0	0	0	0	1	0	0	1	0	0	0
5	0	1	1	0	1	0	1	0	0	1	0	0
6	0	0	1	0	0	1	0	1	0	0	1	0
7	0	0	0	1	0	0	1	0	0	0	0	1
8	0	0	0	0	1	0	0	0	0	1	0	0
9	0	0	0	0	0	1	0	0	1	0	1	0
10	0	0	0	0	0	0	1	0	0	1	0	1
11	0	0	0	0	0	0	0	1	0	0	1	0

Figura 12.2: Um grafo e sua matriz de adjacência.

return $a[i][j]$

Essas operações claramente levam um tempo constante por operação.

O desempenho da matriz de adjacência é insatisfatório nas operações out_edges(i) e in_edges(i). Para implementá-los, devemos examinar todas as entradas n na linha ou coluna correspondente de a e reunir todos os índices, j, onde $a[i][j]$, respectivamente $a[j][i]$, é verdade. Essas operações claramente levam tempo por operação $O(n)$.

Outra desvantagem da representação da matriz de adjacência é que ela é grande. Ele armazena uma matriz booleana $n \times n$, então requer pelo

menos n^2 bits de memória. A implementação aqui usa uma matriz de valores booleanos, então ela realmente usa na ordem de n^2 bytes de memória. Uma implementação mais cuidadosa, que empacote w valores booleanos em cada palavra de memória, poderia reduzir esse uso de espaço para $O(n^2/w)$ palavras de memória.

Teorema 12.1. *A estrutura de dados AdjacencyMatrix implementa a interface Graph. Uma AdjacencyMatrix suporta as operações*

- add_edge(i,j), remove_edge(i,j), *e* has_edge(i,j) *em tempo constante por operação; e*
- in_edges(i), *e* out_edges(i) *em um tempo $O(n)$ por operação.*

O espaço usado por uma AdjacencyMatrix é $O(n^2)$.

Apesar de seus altos requisitos de memória e baixo desempenho das operações in_edges(i) e out_edges(i), uma AdjacencyMatrix ainda pode ser útil para alguns aplicativos. Em particular, quando o grafo G é *denso*, ou seja, tem quase n^2 arestas, então um uso de memória de n^2 pode ser aceitável.

A estrutura de dados AdjacencyMatrix também é comumente usada porque as operações algébricas na matriz a podem ser usadas para calcular com eficiência as propriedades do grafo G. Este é um tópico para um curso de algoritmos, mas apontamos uma dessas propriedades aqui: se tratarmos as entradas de a como inteiros (1 para *verdadeiro* e 0 para *falso*) e multiplicarmos a por si mesmo usando a matriz multiplicação então obtemos a matriz a^2. Lembre-se, a partir da definição de multiplicação de matrizes, que

$$a \oplus 2[i][j] = \sum_{k=0}^{n-1} a[i][k] \cdot a[k][j] \ .$$

Interpretando essa soma em termos do grafo G, essa fórmula conta o número de vértices, k, de forma que G contenha ambas as arestas (i,k) e (k,j). Ou seja, ele conta o número de caminhos de i a j (por meio dos vértices intermediários, k) cujo comprimento é exatamente dois. Essa observação é a base de um algoritmo que calcula os caminhos mais curtos entre todos os pares de vértices em G usando apenas $O(\log n)$ multiplicações de matriz.

12.2 AdjacencyLists: Um grafo como uma coleção de listas

As representações com *listas de adjacências* de grafos têm uma abordagem mais centrada no vértice. Existem muitas implementações possíveis de listas de adjacência. Nesta seção, apresentamos uma simples. No final da seção, discutimos diferentes possibilidades. Em uma representação de lista de adjacência, o grafo $G = (V, A)$ é representado como um array, *adj*, de listas. A lista $adj[i]$ contém uma lista de todos os vértices adjacentes ao vértice i. Ou seja, ele contém todos os índices j tais que $(i, j) \in A$.

```
initialize()
    adj ← new_array(n)
    for i in 0, 1, 2, ..., n − 1 do
        adj[i] ← ArrayStack()
```

(Um exemplo é mostrado em Figura 12.3.) Nesta implementação particular, representamos cada lista em *adj* como ArrayStack, porque gostaríamos de acesso de tempo constante por posição. Outras opções também são possíveis. Especificamente, poderíamos ter implementado *adj* como uma DLList.

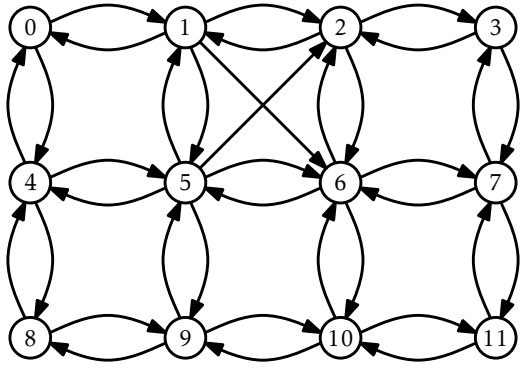
A operação add_edge(i, j) apenas acrescenta o valor j à lista $adj[i]$:

```
add_edge(i,j)
    adj[i].append(j)
```

Isso leva um tempo constante.

A operação remove_edge(i, j) pesquisa a lista $adj[i]$ até encontrar j e depois a remove:

```
remove_edge(i,j)
    for k in 0, 1, 2, ..., length(adj[i]) − 1 do
        if adj[i].get(k) = j then
            adj[i].remove(k)
```



0	1	2	3	4	5	6	7	8	9	10	11
1	0	1	2	0	1	5	6	4	8	9	10
4	2	3	7	5	2	2	3	9	5	6	7
	6	6		8	6	7	11		10	11	
	5				9	10					
					4						

Figura 12.3: Um grafo e suas listas adjacentes

```
        return
```

Isso leva um tempo $O(\deg(i))$, onde $\deg(i)$ (o *grau* de i) conta o número de arestas em A que têm i como fonte.

A operação has_edge(i,j) é semelhante; ela pesquisa na lista $adj[i]$ até encontrar j (e retorna verdadeiro), ou chega ao final da lista (e retorna falso):

```
has_edge(i,j)
    for k in adj[i] do
        if k = j then
            return true
    return false
```

Isso também leva tempo $O(\deg(i))$.

A operação out_edges(i) é muito simples; retorna a lista $adj[i]$:

```
out_edges(i)
    return adj[i]
```

A operação in_edges(i) é muito mais trabalhosa. Ele examina cada vértice j verificando se a aresta (i,j) existe e, em caso afirmativo, adicionando j à lista de saída:

```
in_edges(i)
    out ← ArrayStack()
    for j in 0,1,2,...,n-1 do
        if has_edge(j,i) then out.append(j)
    return out
```

Esta operação é muito lenta. Ele faz a varredura da lista de adjacências de cada vértice, portanto, leva tempo $O(n+m)$.

O teorema a seguir resume o desempenho da estrutura de dados acima:

Teorema 12.2. *A estrutura de dados AdjacencyLists implementa a interface Graph. Uma* AdjacencyLists *suporta as operações*

- add_edge(i, j) *em tempo constante por operação;*
- remove_edge(i, j) *e* has_edge(i, j) *em tempo* $O(\deg(i))$ *por operação;*
- in_edges(i) *em tempo* $O(n + m)$ *por operação.*

O espaço usado por uma AdjacencyLists é $O(n + m)$.

Conforme mencionado anteriormente, existem muitas opções diferentes a serem feitas ao implementar um grafo como uma lista de adjacências. Algumas perguntas que surgem incluem:

- Que tipo de coleção deve ser usado para armazenar cada elemento de *adj*? Pode-se usar uma lista baseada em array, uma lista encadeada ou até mesmo uma tabela de hash.
- Deve haver uma segunda lista de adjacência, *inadj*, que armazena, para cada i, a lista de vértices, j, tal que $(j, i) \in A$? Isso pode reduzir bastante o tempo de execução da operação in_edges(i), mas requer um pouco mais de trabalho ao adicionar ou remover arestas.
- A entrada para a aresta (i, j) em $adj[i]$ deve ser encadeada por uma referência à entrada correspondente em $inadj[j]$?
- As arestas devem ser objetos de primeira classe com seus próprios dados associados? Desta forma, *adj* conteria listas de arestas em vez de listas de vértices (inteiros).

A maioria dessas questões se resume a uma troca entre complexidade (e espaço) de implementação e recursos de desempenho da implementação.

12.3 Percurso em Grafos

Nesta seção, apresentamos dois algoritmos para explorar um grafo, começando em um de seus vértices, i, e encontrando todos os vértices que são acessíveis a partir de i. Ambos os algoritmos são mais adequados para grafos representados usando uma representação de lista de adjacência.

Portanto, ao analisar esses algoritmos, assumiremos que a representação subjacente é uma AdjacencyLists.

12.3.1 Busca em Largura

O algoritmo *busca em largura* começa em um vértice i e visita, primeiro os vizinhos de i, depois os vizinhos dos vizinhos de i, então os vizinhos dos vizinhos dos vizinhos de i e assim por diante.

Este algoritmo é uma generalização do algoritmo de percurso em largura para árvores binárias (Seção 6.1.2) e é muito semelhante; ele usa uma fila, q, que inicialmente contém apenas i. Em seguida, extrai repetidamente um elemento de q e adiciona seus vizinhos a q, desde que esses vizinhos nunca tenham estado em q antes. A única grande diferença entre o algoritmo de busca por largura para grafos e o algoritmo para árvores é que o algoritmo para grafos deve garantir que não adiciona o mesmo vértice a q mais de uma vez. Ele faz isso usando um array booleano auxiliar, *seen*, que rastreia quais vértices já foram descobertos.

```
bfs(g, r)
    seen ← new_boolean_array(n)
    q ← SLList()
    q.add(r)
    seen[r] ← true
    while q.size() > 0 do
        i ← q.remove()
        for j in g.out_edges(i) do
            if seen[j] = false then
                q.add(j)
                seen[j] ← true
```

Um exemplo de execução de bfs$(g, 0)$ no grafo de Figura 12.1 é mostrado em Figura 12.4. Diferentes execuções são possíveis, dependendo da ordem das listas de adjacência; Figura 12.4 usa as listas de adjacência em Figura 12.3.

Analisar o tempo de execução da rotina bfs(g, i) é bastante simples. O

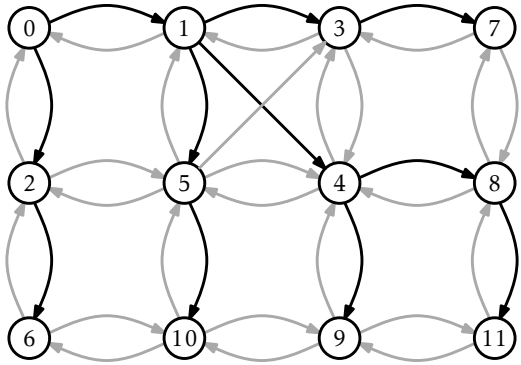


Figura 12.4: Um exemplo de pesquisa em amplitude começando no nó 0. Os nós são rotulados com a ordem em que são adicionados a q. As arestas que resultam na adição de nós a q são desenhadas em preto, outras arestas são desenhadas em cinza.

uso do array *seen* garante que nenhum vértice seja adicionado a q mais de uma vez. Adicionar (e posteriormente remover) cada vértice de q leva um tempo constante por vértice para um total de $O(n)$ tempo. Uma vez que cada vértice é processado pelo laço interno no máximo uma vez, cada lista de adjacência é processada no máximo uma vez, então cada aresta de G é processada no máximo uma vez. Esse processamento, que é feito no loop interno, leva um tempo constante por iteração, para um tempo total de $O(m)$. Portanto, todo o algoritmo é executado em tempo $O(n+m)$.

O teorema a seguir resume o desempenho do algoritmo bfs(g,r).

Teorema 12.3. *Quando dado como entrada um Grafo, g, que é implementado usando a estrutura de dados AdjacencyLists, o algoritmo* bfs(g,r) *é executado em tempo $O(n+m)$.*

Uma travessia em largura tem algumas propriedades muito especiais. Chamar bfs(g,r) eventualmente enfileirará (e eventualmente removerá da fila) cada vértice j de forma que haja um caminho direcionado de r para j. Além disso, os vértices na distância 0 de r (o próprio r) entrarão q antes dos vértices na distância 1, que entrarão q antes dos vértices na distância 2, e assim por diante. Assim, o método bfs(g,r) visita vértices em ordem crescente de distância de r e vértices que não podem ser alcançados a

partir de r nunca são visitados.

Uma aplicação particularmente útil do algoritmo de busca por largura é, portanto, na computação de caminhos mais curtos. Para calcular o caminho mais curto de r para todos os outros vértices, usamos uma variante de bfs(g,r) que usa uma matriz auxiliar, p, de comprimento n. Quando um novo vértice j é adicionado a q, definimos $p[j] \leftarrow i$. Desta forma, $p[j]$ se torna o penúltimo nó em um caminho mais curto de r a j. Repetindo isso, tomando $p[p[j]$, $p[p[p[j]]]$, e assim por diante, podemos reconstruir o (reverso de) um caminho mais curto de r para j.

12.3.2 Pesquisa em profundidade

O algoritmo *Pesquisa em profundidade* é semelhante ao algoritmo padrão para percorrer árvores binárias; ele primeiro explora completamente uma subárvore antes de retornar ao nó atual e, em seguida, explorar a outra subárvore. Outra maneira de pensar na pesquisa em profundidade é dizer que ela é semelhante à pesquisa em largura, exceto que usa uma pilha em vez de uma fila.

Durante a execução do algoritmo de pesquisa em profundidade, cada vértice, i, recebe uma cor, $c[i]$: *branco* se nunca vimos o vértice antes, *cinza* se estivermos atualmente visitando aquele vértice, e *preto* se terminarmos de visitar aquele vértice. A maneira mais fácil de pensar na pesquisa em profundidade é como um algoritmo recursivo. Ele começa visitando r. Ao visitar um vértice i, primeiro marcamos i como *cinza*. Em seguida, varremos a lista de adjacências de i e visitamos recursivamente qualquer vértice branco que encontrarmos nesta lista. Finalmente, terminamos o processamento i, então colorimos i de preto e retornamos.

```
dfs(g,r)
    c ← new_array(g.n)
    dfs(g,r,c)

dfs(g,i,c)
    c[i] ← grey
    for j in g.out_edges(i) do
        if c[j] = white then
```

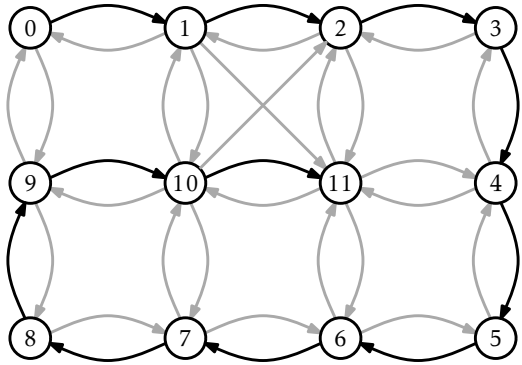


Figura 12.5: Um exemplo de pesquisa em profundidade começando no nó 0. Os nós são rotulados com a ordem em que são processados. As arestas que resultam em uma chamada recursiva são desenhadas em preto, outras arestas são desenhadas em *cinza*.

```
            c[j] ← grey
            dfs(g,j,c)
    c[i] ← black
```

Um exemplo da execução deste algoritmo é mostrado em Figura 12.5.

Embora a pesquisa em profundidade possa ser melhor considerada um algoritmo recursivo, a recursão não é a melhor maneira de implementá-la. Na verdade, o código fornecido acima falhará para muitos grafos grandes, causando um estouro de pilha. Uma implementação alternativa é substituir a pilha de recursão por uma pilha explícita, s. A implementação a seguir faz exatamente isso:

```
dfs2(g,r)
    c ← new_array(g.n)
    s ← SLList()
    s.push(r)
    while s.size() > 0 do
        i ← s.pop()
        if c[i] = white then
```

```
        c[i] ← grey
        for j in g.out_edges(i) do
            s.push(j)
```

No código anterior, quando o próximo vértice, i, é processado, i é colorido *cinza* e então substituído, na pilha, por seus vértices adjacentes. Durante a próxima iteração, um desses vértices será visitado.

Não surpreendentemente, os tempos de execução de dfs(g,r) e dfs2(g,r) são iguais aos de bfs(g,r):

Teorema 12.4. *Quando dado como entrada um Grafo, g, que é implementado usando a estrutura de dados AdjacencyLists, os algoritmos* dfs(g,r) *e* dfs2(g,r) *são executados em tempo $O(n+m)$.*

Tal como acontece com o algoritmo de pesquisa em largura, há uma árvore subjacente associada a cada execução da pesquisa em profundidade. Quando um nó $i \neq r$ vai de *branco* para *cinza*, isso ocorre porque dfs(g,i,c) foi chamado recursivamente durante o processamento de algum nó i'. (No caso do dfs2(g,r) algoritmo, i é um dos nós que substituiu i' na pilha.) Se pensarmos em i' como o pai de i, então obtemos uma árvore enraizada em r. Em Figura 12.5, esta árvore é um caminho do vértice 0 ao vértice 11.

Uma propriedade importante do algoritmo de pesquisa em profundidade é a seguinte: Suponha que quando o nó i é colorido *cinza*, existe um caminho de i para algum outro nó j que usa apenas vértices brancos. Então j será colorido primeiro *cinza* depois *preto* antes de i ser colorido *preto*. (Isso pode ser provado por contradição, considerando qualquer caminho P de i a j.)

Uma das aplicações desta propriedade é a detecção de ciclos. Consulte a Figura 12.6. Considere algum ciclo, C, que pode ser alcançado a partir de r. Seja i o primeiro nó de C colorido *cinza* e seja j o nó que precede i no ciclo C. Então, pela propriedade acima, j será colorido *cinza* e a aresta (j,i) será considerada pelo algoritmo enquanto i ainda é *cinza*. Assim, o algoritmo pode concluir que existe um caminho, P, de i a j na árvore de pesquisa em profundidade e a aresta (j,i) existe. Portanto, P também é um ciclo.

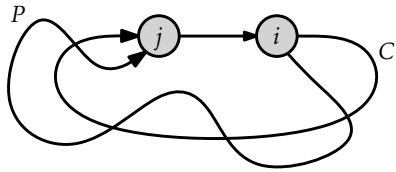


Figura 12.6: O algoritmo de pesquisa em profundidade pode ser usado para detectar ciclos em G. O nó j é colorido *cinza* enquanto i ainda é *cinza*. Isso implica que há um caminho, P, de i para j na árvore de pesquisa em profundidade, e a aresta (j,i) implica que P também é um ciclo.

12.4 Discussão e Exercícios

Os tempos de execução dos algoritmos de pesquisa em profundidade e em largura primeiro são um tanto exagerados pelos Teoremas 12.3 e 12.4. Defina n_r como o número de vértices, i, de G, para os quais existe um caminho de r para i. Defina m_r como o número de arestas que têm esses vértices como suas fontes. Então, o teorema a seguir é uma declaração mais precisa dos tempos de execução dos algoritmos de pesquisa em largura e em profundidade. (Esta declaração mais refinada do tempo de execução é útil em algumas das aplicações desses algoritmos descritos nos exercícios.)

Teorema 12.5. *Quando fornecido como entrada um Grafo, g, que é implementado usando a estrutura de dados AdjacencyLists, e algoritmos* bfs(g,r), dfs(g,r) *e* dfs2(g,r), *cada um executado em tempo* $O(n_r + m_r)$.

A pesquisa em largura parece ter sido descoberta independentemente por Moore [52] e Lee [49] nos contextos de exploração de labirinto e roteamento de circuito, respectivamente.

As representações de listas de adjacências de grafos foram apresentadas por Hopcroft e Tarjan [40] como uma alternativa à (então mais comum) representação de matriz de adjacências. Essa representação, bem como a pesquisa em profundidade, desempenhou um papel importante no famoso algoritmo de teste de planaridade Hopcroft-Tarjan que pode determinar, em um tempo $O(n)$, se um grafo pode ser desenhado, no plano, e de forma que nenhum par de arestas se cruze [41].

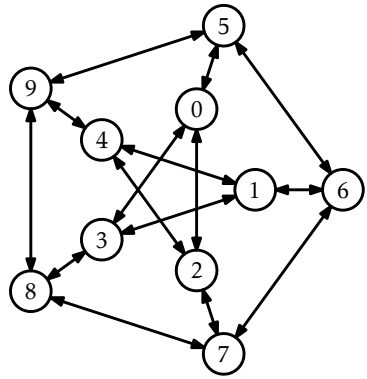


Figura 12.7: Um exemplo de grafo.

Nos exercícios a seguir, um grafo não direcionado é aquele em que, para cada i e j, a aresta (i,j) está presente se e somente se a aresta (j,i) está presente.

Exercício 12.1. Desenhe uma representação de lista de adjacência e uma representação de matriz de adjacência do grafo em Figura 12.7.

Exercício 12.2. A representação *matriz de incidência* de um grafo, G, é uma $n \times m$, A, onde

$$A_{i,j} = \begin{cases} -1 & \text{se o vértice } i \text{ é a origem da aresta } j \\ +1 & \text{se o vértice } i \text{ é o alvo da aresta } j \\ 0 & \text{caso contrário.} \end{cases}$$

1. Desenhe a representação da matriz de incidentes do grafo na Figura 12.7.

2. Projetar, analisar e implementar uma representação de matriz de incidência de um grafo. Certifique-se de analisar o espaço, o custo de add_edge(i,j), remove_edge(i,j), has_edge(i,j), in_edges(i), e out_edges(i).

Exercício 12.3. Ilustre uma execução de bfs$(G,0)$ e dfs$(G,0)$ no grafo, G, in Figura 12.7.

Exercício 12.4. Seja G um grafo não direcionado. Dizemos que G é *conectado* se, para cada par de vértices i e j em G, há um caminho de i para j

(uma vez que G é não direcionado, há também um caminho de j para i). Mostre como testar se G está conectado em um tempo $O(n+m)$.

Exercício 12.5. Seja G um grafo não direcionado. Uma *etiquetagem de componente conectado* de G divide os vértices de G em conjuntos máximos, cada um dos quais forma um subgrafo conectado. Mostre como calcular a etiquetagem de um componente conectado de G em um tempo $O(n+m)$.

Exercício 12.6. Seja G um grafo não direcionado. Uma *floresta extensa* de G é uma coleção de árvores, uma por componente, cujas arestas são arestas de G e cujos vértices contêm todos os vértices de G. Mostre como calcular uma floresta extensa de G em tempo $O(n+m)$.

Exercício 12.7. Dizemos que um grafo G é *fortemente conectado* se, para cada par de vértices i e j em G, houver um caminho de i para j. Mostre como testar se G está fortemente conectado em um tempo $O(n+m)$.

Exercício 12.8. Dado um grafo $G = (V, A)$ e algum vértice especial $r \in V$, mostre como calcular o comprimento do caminho mais curto de r a i para cada vértice $i \in V$.

Exercício 12.9. Dê um exemplo (simples) em que o código dfs(g, r) visita os nós de um grafo em uma ordem diferente daquela do código dfs2(g, r). Escreva uma versão de dfs2(g, r) que sempre visita os nós exatamente na mesma ordem que dfs(g, r). (Dica: basta começar a rastrear a execução de cada algoritmo em algum grafo onde r é a origem de mais de 1 aresta.)

Exercício 12.10. Um *sumidouro universal* em um grafo G é um vértice que é o alvo de $n-1$ arestas e a fonte de nenhuma aresta.[1] Projete e implemente um algoritmo que testa se um grafo G, representado como AdjacencyMatrix, tem um sumidouro universal. Seu algoritmo deve ser executado em tempo $O(n)$.

[1] Um sumidouro universal, v, também é às vezes chamado de *celebridade*: todos na sala reconhecem v, mas v não reconhece ninguém na sala.