Introdução a Grafos

IPRJ01-10762
Matemática Discreta II

Gustavo Barbosa Libotte

Aula 7

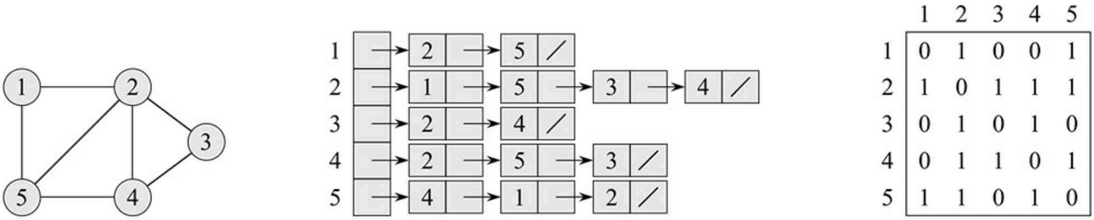
Representações de Grafos

- Podemos escolher entre dois modos padrões para representar um grafo $G = (V, E)$: como uma coleção de **listas de adjacências** ou como uma **matriz de adjacências**.
- Qualquer desses modos se aplica a grafos dirigidos e não dirigidos.
- Como a representação por lista de adjacências nos dá um modo compacto de representar grafos esparsos—aqueles para os quais $|E|$ é muito menor que $|V|^2$—, ela é, em geral, o método preferido.
- A maioria dos algoritmos de grafos que veremos supõe que um grafo de entrada é representado sob a forma de lista de adjacências.
- Contudo, uma representação por matriz de adjacências pode ser preferível quando o grafo é denso—$|E|$ está próximo de $|V|^2$—ou quando precisamos saber rapidamente se há uma aresta conectando dois vértices dados.

Representações de Grafos

- A **representação por lista de adjacências** de um grafo $G = (V, E)$ consiste em um arranjo Adj de $|V|$ listas, uma para cada vértice em V.
- Para cada $u \in V$, a lista de adjacências $Adj[u]$ contém todos o vértices v tais que existe uma aresta $(u, v) \in E$.
- Isto é, $Adj[u]$ consiste em todos os vértices adjacentes a u em G. (Alternativamente, ela pode conter ponteiros para esses vértices.)
- Visto que as listas de adjacências representam os vértices de um grafo, em pseudocódigo tratamos o arranjo Adj como um atributo do grafo, exatamente como tratamos o conjunto de vértices E.
- Portanto, em pseudocódigo, veremos notação tal como $G.Adj[u]$.

REPRESENTAÇÕES DE GRAFOS



(a) (b) (c)

Figura 1: Duas representações de um grafo não dirigido. (a) Um grafo não dirigido G com cinco vértices e sete arestas. (b) Uma representação de G por lista de adjacências. (c) A representação de G por matriz de adjacências.

Representações de Grafos

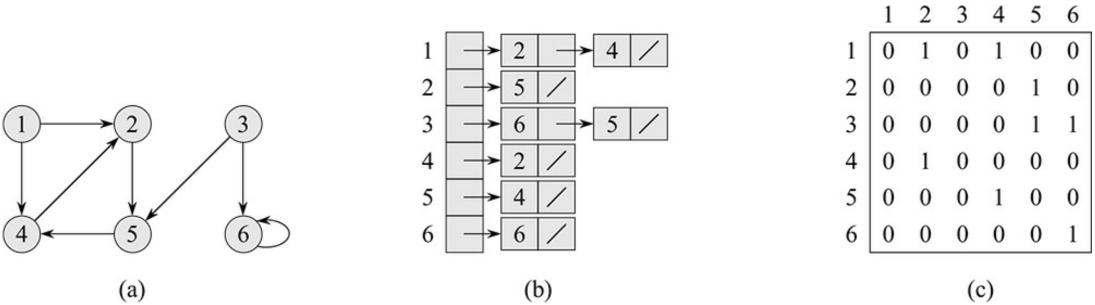


Figura 2: Duas representações de um grafo dirigido. (a) Um grafo dirigido G com seis vértices e oito arestas. (b) Uma representação de G por lista de adjacências. (c) A representação de G por matriz de adjacências.

Representações de Grafos

- Se G é um **grafo dirigido**, a soma dos comprimentos de todas as listas de adjacências é $|E|$, já que uma aresta da forma (u, v) é representada fazendo com que v apareça em $Adj[u]$.
- Se G é um **grafo não dirigido**, a soma dos comprimentos de todas as listas de adjacências é $2|E|$, já que, se (u, v) é uma aresta não dirigida, então u aparece na lista de adjacências de v e vice-versa.
- Quer os grafos sejam dirigidos ou não dirigidos, a representação por lista de adjacências tem a seguinte propriedade interessante: a quantidade de memória que ela exige é $\Theta(V + E)$.
- Podemos adaptar imediatamente as listas de adjacências para representar **grafos ponderados**, isto é, grafos nos quais cada aresta tem um peso associado, normalmente dado por uma função peso $w : E \rightarrow$.
- Por exemplo, seja $G = (V, E)$ um grafo ponderado com função peso w.
- Simplesmente armazenamos o peso $w(u, v)$ da aresta $(u, v) \in E$ com o vértice v na lista de adjacências de u.

Representações de Grafos

- A representação por lista de adjacências é bastante robusta no sentido de que podemos modificá-la para suportar muitas outras variantes de grafos.
- Uma desvantagem potencial da representação por lista de adjacências, entretanto, é que ela não proporciona nenhum modo mais rápido para determinar se uma dada aresta (u, v) está presente no grafo do que procurar v na lista de adjacências $Adj[u]$.
- Essa desvantagem pode ser contornada por uma representação por matriz de adjacências do grafo, porém ao custo de utilizar assintoticamente mais memória.
- No caso da representação por matriz de adjacências de um grafo $G = (V, E)$, supomos que os vértices são numerados $1, 2, \ldots, |V|$ de alguma maneira arbitrária.
- Então, a representação por matriz de adjacências de um grafo G consiste em uma matriz $|V| \times |V|, A = \left(a_{ij}\right)$, tal que

$$a_{ij} = \begin{cases} 1 & \text{se } (i, j) \in E \\ 0 & \text{caso contrário} \end{cases}$$

Representações de Grafos

- Observe a simetria ao longo da diagonal principal da matriz de adjacências na Figura 1(c).
- Visto que, em um **gráfico não dirigido**, (u, v) e (v, u) representam a mesma aresta, a matriz de adjacências A de um grafo não dirigido é sua própria transposta: $A = A_T$.
- Em algumas aplicações, vale a pena armazenar somente as entradas que estão na **diagonal** e **acima da diagonal** da matriz de adjacências, o que reduz quase à metade a memória necessária para armazenar o grafo.
- Assim como a representação por lista de adjacências de um grafo, uma matriz de adjacências pode representar um grafo ponderado.
- Por exemplo, se $G = (V, E)$ é um grafo ponderado com função peso de aresta w, podemos simplesmente armazenar o peso $w(u, v)$ da aresta $(u, v) \in E$ como a entrada na linha u e coluna v da matriz de adjacências.
- Se uma aresta não existe, podemos armazenar um valor NIL como sua entrada de matriz correspondente, se bem que em muitos problemas é conveniente usar um valor como 0 ou ∞.

- A maioria dos algoritmos que funcionam em grafos precisa manter **atributos** para vértices e/ou arestas.
- Indicamos esses atributos usando nossa notação usual, por exemplo, $v.d$ para um **atributo** d de um **vértice** v.
- Quando indicamos arestas como pares de vértices, usamos o mesmo estilo de notação.
- Por exemplo, se arestas têm um atributo f, denotamos esse atributo para a aresta (u, v) por $(u, v).f$.
- Para a finalidade de apresentar e entender algoritmos, nossa notação de atributo é suficiente.
- Implementar atributos de vértice e aresta em programas reais pode ser uma história inteiramente diferente. Não há nenhum modo que seja reconhecidamente melhor para armazenar e acessar atributos de vértice e de aresta.
- Dada uma situação, é provável que sua decisão dependerá da linguagem de programação que estiver usando, do algoritmo que estiver implementando e de como o resto de seu programa usará o grafo.

Busca em Largura

- A **busca em largura** é um dos algoritmos mais simples para executar busca em um grafo e é o arquétipo de muitos algoritmos de grafos importantes.
- O algoritmo de árvore geradora mínima de Prim e o algoritmo de caminhos mínimos de fonte única de Dijkstra usam ideias semelhantes às que aparecem na busca em largura.
- Dado um grafo $G = (V, E)$ e um vértice fonte s, a busca em largura explora sistematicamente as arestas de G para "descobrir" cada vértice que pode ser alcançado a partir de s.
- O algoritmo calcula a distância (menor número de arestas) de s até cada vértice que pode ser alcançado.
- Produz também uma "árvore de busca em largura" com raiz s que contém todos os vértices que podem ser alcançados.

Busca em Largura

- Para qualquer vértice v que pode ser alcançado de s, o caminho simples na árvore de busca em largura de s até v corresponde a um "caminho mínimo" de s a v em G, isto é, um caminho que contém o menor número de arestas.
- O algoritmo funciona em grafos dirigidos, bem como em grafos não dirigidos.
- A busca em largura tem esse nome porque expande a fronteira entre vértices descobertos e não descobertos uniformemente ao longo da extensão da fronteira.
- Isto é, o algoritmo descobre todos os vértices à distância k de s, antes de descobrir quaisquer vértices à distância $k + 1$.
- Para controlar o progresso, a busca em largura pinta cada vértice de **branco**, **cinzento** ou **preto**.
- No início, todos os vértices são brancos, e mais tarde eles podem se tornar cinzentos e depois pretos.

Busca em Largura

- Um vértice é **descoberto** na primeira vez em que é encontrado durante a busca, e nesse momento ele se torna **não branco**.
- Portanto, vértices cinzentos e pretos são vértices descobertos, mas a busca em largura distingue entre eles para assegurar que a busca prossiga sendo em largura.[1]
- Se $(u, v) \in E$ e o vértice u é preto, então o vértice v é cinzento ou preto; isto é, todos os vértices adjacentes a vértices pretos foram descobertos.
- Vértices cinzentos podem ter alguns vértices adjacentes brancos; eles representam a **fronteira** entre vértices descobertos e não descobertos.
- A busca em largura constrói uma árvore em largura, que contém inicialmente apenas sua raiz, que é o vértice de fonte s.

[1] Distinguimos entre vértices cinzento e preto para nos ajudar a entender como a busca em largura opera. De fato, podemos mostrar que obteríamos o mesmo resultado mesmo se não distinguíssemos os vértices cinzento e preto.

Busca em Largura

- Sempre que a busca descobre um vértice branco v no curso da varredura da lista de adjacências de um vértice u já descoberto, o vértice v e a aresta (u, v) são acrescentados à árvore.
- Dizemos que u é o **predecessor** ou **pai** de v na árvore de busca em largura.
- Visto que um vértice é descoberto no máximo uma vez, ele tem no máximo um pai.
- Relações de ancestral e descendente na árvore de busca em largura são definidas em relação à raiz s da maneira usual: se u está em um caminho simples na árvore que vai da raiz s até o vértice v, então u é um ancestral de v,e v é um descendente de u.
- O procedimento de busca em largura BFS mostrado a seguir supõe que o grafo de entrada $G = (V, E)$ é representado com a utilização de **listas de adjacências**.
- Ele anexa vários atributos adicionais a cada vértice no grafo.
- Armazenamos a cor de cada vértice $u \in V$ no atributo $u.cor$ e o predecessor de u no atributo $u.p$.

Busca em Largura

- Se u não tem nenhum predecessor (por exemplo, se $u = s$ ou não foi descoberto), então $u.p = \text{NIL}$.
- O atributo $u.d$ mantém a distância da fonte s ao vértice u calculada pelo algoritmo.
- O algoritmo também utiliza uma fila Q do tipo primeiro a entrar, primeiro a sair (FIFO) para gerenciar o conjunto de vértices cinzentos.

BFS(G, s)

```
for cada vértice u ∈ V[G] − {s} do
    u.cor ← BRANCO
    u.d ← ∞
    u.π ← NIL
s.cor ← CINZENTO
s.d ← 0
s.π ← NIL
Q ← ∅
ENQUEUE(Q, s)
while Q ≠ ∅ do
    u ← DEQUEUE(Q)
    for cada vértice v ∈ Adj[u] do
        if v.cor == BRANCO then
            v.cor ← CINZENTO
            v.d ← u.d + 1
            v.π ← u
            ENQUEUE(Q, v)
    u.cor ← PRETO
```

Busca em Largura

- O procedimento BFS funciona da maneira descrita a seguir. Com a exceção do vértice de fonte s, as linhas 1–4 pintam todos os vértices de branco, definem $u.d$ como infinito para todo vértice u e definem o pai de todo vértice como NIL.
- A linha 5 pinta s de cinzento, já que consideramos que ele é descoberto quando o procedimento começa.
- A linha 6 inicializa $s.d$ como zero, e a linha 7 define o predecessor do fonte como NIL.
- As linhas 8–9 inicializam Q como a fila que contém apenas o vértice s.
- O laço *while* das linhas 10–18 itera enquanto houver vértices cinzentos, que são vértices descobertos cujas listas de adjacências ainda não foram totalmente examinadas.
- A fila Q consiste no conjunto de vértices cinzentos. Antes da primeira iteração, o único vértice cinzento, e o único vértice em Q, é o vértice de fonte s.
- A linha 11 determina o vértice cinzento u no início da fila Q e o remove de Q.

Busca em Largura

- O laço for das linhas 12–17 considera cada vértice v na lista de adjacências de u.
- Se v é branco, então ainda não foi descoberto, e o procedimento o descobre executando as linhas 14–17.
- O procedimento pinta o vértice v de cinzento, define sua distância $v.d$ como $u.d + 1$, registra u como seu pai $v.p$ e o coloca no final da fila Q.
- Uma vez examinados todos os vértices na lista de adjacências de u, o procedimento pinta u de preto na linha 18.
- Os resultados da busca em largura podem depender da ordem na qual os vizinhos de um determinado vértice são visitados na linha 12; a árvore de busca em largura pode variar, mas as distâncias d calculadas pelo algoritmo não variam.

BUSCA EM PROFUNDIDADE

- A estratégia seguida pela busca em profundidade é, como seu nome implica, buscar "**mais fundo**" no grafo, sempre que possível.
- A busca em profundidade explora arestas partindo do vértice v mais recentemente descoberto do qual ainda saem arestas inexploradas.
- Depois que todas as arestas de v foram exploradas, a busca "**regressa pelo mesmo caminho**" para explorar as arestas que partem do vértice do qual v foi descoberto.
- Esse processo **continua até** descobrirmos todos os vértices que podem ser visitados a partir do vértice fonte inicial.
- Se restarem quaisquer vértices não descobertos, a busca em profundidade **seleciona um deles como fonte** e repete a busca partindo dessa fonte.
- O algoritmo repete esse processo inteiro até descobrir todos os vértices.

Busca em Profundidade

- Como ocorre na busca em largura, sempre que a busca em profundidade descobre um vértice v durante uma varredura da lista de adjacências de um vértice já descoberto u, registra esse evento definindo o atributo predecessor de $v, v.\pi$ como u.
- Diferentemente da busca em largura, cujo subgrafo dos predecessores forma uma árvore, o subgrafo dos predecessores produzido por uma busca em profundidade pode ser composto por **várias árvores** porque a busca pode ser repetida **partindo de várias fontes**.
- Portanto, definimos o subgrafo dos predecessores de uma busca em profundidade de um modo ligeiramente diferente do da busca em largura: fazemos $G_p = \left(V, E_p\right)$, onde

$$E_\pi = \{(v.\pi, v) : v \in V \text{ e } v.\pi \neq \text{NIL}\}$$

- O subgrafo dos predecessores de uma busca em profundidade forma uma **floresta de busca em profundidade** que abrange várias árvores de busca em profundidade. As arestas em E_p são arestas de árvore.

Busca em Profundidade

- Como na busca em largura, a busca em profundidade **pinta os vértices** durante a busca para indicar o estado de cada um.
- Cada vértice é inicialmente **branco**, pintado de **cinzento** quando descoberto na busca e pintado de **preto** quando terminado, isto é, quando sua lista de adjacências já foi totalmente examinada.
- Essa técnica garante que cada vértice acabe em exatamente uma árvore, de forma que essas árvores são **disjuntas**.
- Além de criar uma floresta, a busca em profundidade também identifica cada vértice com um **carimbo de tempo**.
- Cada vértice v tem dois carimbos de tempo: o primeiro carimbo de tempo $v.d$ registra quando v é **descoberto pela primeira vez** (e pintado de cinzento), e o segundo carimbo de tempo $v.f$ registra quando a busca **termina de examinar a lista de adjacências** de v (e pinta v de preto).

Busca em Profundidade

- Esses carimbos de tempo dão informações importantes sobre a estrutura do grafo e em geral são úteis para deduzir o comportamento da busca em profundidade.
- O procedimento DFS a seguir registra no atributo $u.d$ o momento em que descobre o vértice u e registra no atributo $u.f$ o momento em que liquida o vértice u.
- Esses carimbos de tempo são inteiros entre 1 e $2|V|$, já que existe um evento de descoberta e um evento de término para cada um dos $|V|$ vértices.
- Para todo vértice u,
$$u.d < u.f$$
- O vértice u é BRANCO antes do tempo $u.d$, CINZENTO entre o tempo $u.d$ e o tempo $u.f$ e PRETO daí em diante.

BUSCA EM PROFUNDIDADE

- O pseudocódigo a seguir é o algoritmo básico de busca em profundidade. O grafo de entrada G pode ser **dirigido** ou **não dirigido**.
- A variável tempo é uma variável global que utilizamos para definir carimbos de tempo.

DFS(G)

```
for cada vértice u ∈ V[G] do
    u.cor ← BRANCO
    u.π ← NIL
tempo ← 0
for cada vértice u ∈ V[G] do
    if u.cor == BRANCO then
        DFS-Visit(G, u)
```

DFS-Visit(G, u)

```
tempo ← tempo + 1
u.d ← tempo
u.cor ← CINZENTO
for v ∈ G.Adj[u] do
    if v.cor == BRANCO then
        v.π ← u
        DFS-Visit(G, v)
u.cor ← PRETO
tempo ← tempo + 1
u.f ← tempo
```

- O procedimento DFS funciona da maneira descrita a seguir. As linhas 1–3 pintam todos os vértices de branco e inicializam seus atributos p como NIL.
- A linha 4 reajusta o contador de tempo global. As linhas 5–7 verificam cada vértice de V por vez e, quando um vértice branco é encontrado, elas o visitam usando DFS-Visit.
- Toda vez que DFS-Vitis(G, u) é chamado na linha 7 , o vértice u se torna a **raiz de uma nova árvore** na floresta em profundidade.
- Quando DFS retorna, a todo vértice u foi atribuído um tempo de descoberta $d[u]$ e um tempo de término $f[u]$.
- Em cada chamada de DFS-Visit(G, u), o vértice u é inicialmente branco. A linha 1 incrementa a variável global tempo, a linha 2 registra o novo valor de tempo como o tempo de descoberta $d[u]$ e a linha 3 pinta u de cinzento.

- As linhas 4–7 examinam cada vértice v adjacente a u e visitam recursivamente v se ele é branco.
- À medida que cada vértice $v \in Adj[u]$ é considerado na linha 4, dizemos que a aresta (u, v) é explorada pela busca em profundidade.
- Finalmente, depois que toda aresta que sai de u foi explorada, as linhas 8–10 pintam u de preto, incrementam tempo e registram o tempo de término em $f[u]$.