MC202 - Estruturas de Dados

Guilherme P. Telles

IC

16 de Novembro de 2019

Avisos

- Estes slides contêm erros.
- Estes slides são incompletos.
- Estes slides foram escritos usando português anterior à reforma ortográfica de 2009.

Parte I

Grafos

Grafos

- Grafos são uma forma de modelar objetos e relações entre eles.
  - pessoas e a relação “é amigo”.

Grafos

- Grafos são uma forma de modelar objetos e relações entre eles.
  - pessoas e a relação “é amigo”.
  - cidades e estradas entre elas.

Grafos

- Grafos são uma forma de modelar objetos e relações entre eles.
  - pessoas e a relação "é amigo".
  - cidades e estradas entre elas.
  - trechos de ruas e cruzamentos na cidade.

Grafos

- Grafos são uma forma de modelar objetos e relações entre eles.
  - pessoas e a relação "é amigo".
  - cidades e estradas entre elas.
  - trechos de ruas e cruzamentos na cidade.
  - escritores, livros e relações de autoria.

Grafos

- Grafos são uma forma de modelar objetos e relações entre eles.
  - pessoas e a relação "é amigo".
  - cidades e estradas entre elas.
  - trechos de ruas e cruzamentos na cidade.
  - escritores, livros e relações de autoria.
  - dutos e os pontos de entrada e saída do fluido.

Grafos

- Grafos são uma forma de modelar objetos e relações entre eles.
  - pessoas e a relação “é amigo”.
  - cidades e estradas entre elas.
  - trechos de ruas e cruzamentos na cidade.
  - escritores, livros e relações de autoria.
  - dutos e os pontos de entrada e saída do fluido.
  - moléculas e interações entre elas.

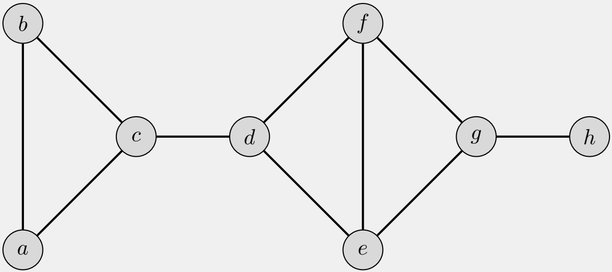
Grafos

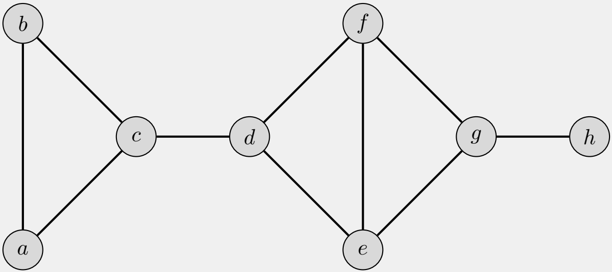
- Grafos são uma forma de modelar objetos e relações entre eles.
  - pessoas e a relação "é amigo".
  - cidades e estradas entre elas.
  - trechos de ruas e cruzamentos na cidade.
  - escritores, livros e relações de autoria.
  - dutos e os pontos de entrada e saída do fluido.
  - moléculas e interações entre elas.
  - muitas outras.

- Há uma grande teoria e um monte de algoritmos para problemas em grafos.
- Algoritmos para problemas em grafos são usados para resolver um monte de problemas reais.

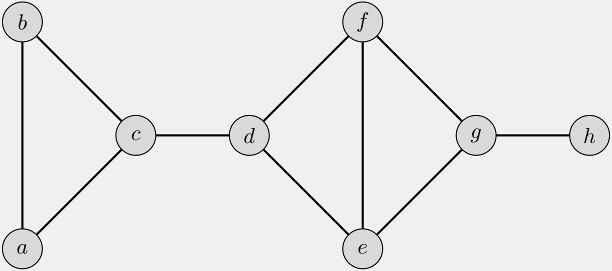
Nomenclatura

- Um *grafo* é um par $G = (V, E)$ onde
  - V é um conjunto finito de elementos chamados *vértices* e
  - E é um conjunto finito de pares não-ordenados de vértices chamados *arestas*.
- Um grafo é tipicamente representado por um diagrama onde os vértices são círculos e as arestas são segmentos de reta.

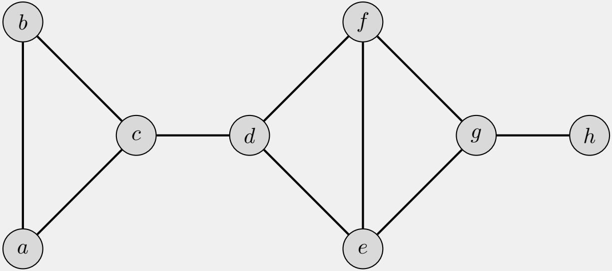

b
f
c
d
g
h
a
e



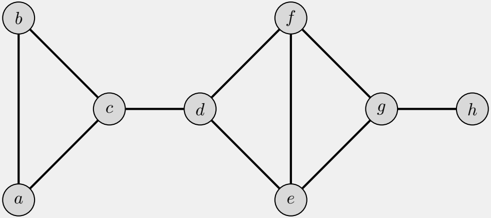
- Para uma aresta $e = (u, v)$ dizemos que os vértices u e v são os *extremos* de e.
- Uma aresta é *incidente* em seus vértices.
- Um vértice é *incidente* nas arestas de que é extremo.

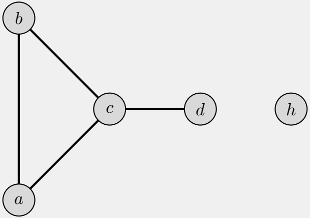


- Dois vértices são *adjacentes* se são extremos da mesma aresta.
- Duas arestas são *adjacentes* se têm um vértice em comum.
- A *vizinhança* de um vértice u, $N(u)$, é o conjunto de vértices adjacentes a u.

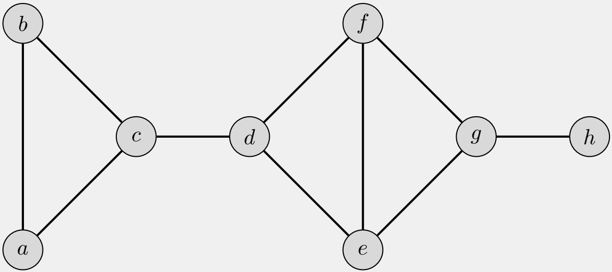


- O *grau* de um vértice v, $\deg(v)$, é o número de arestas incidentes a ele.

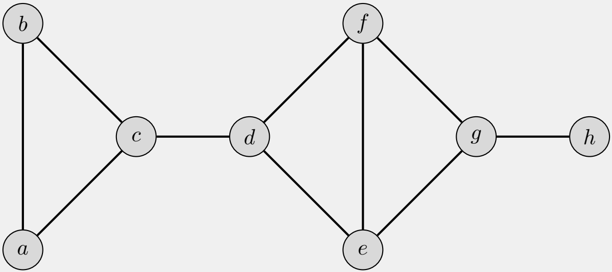




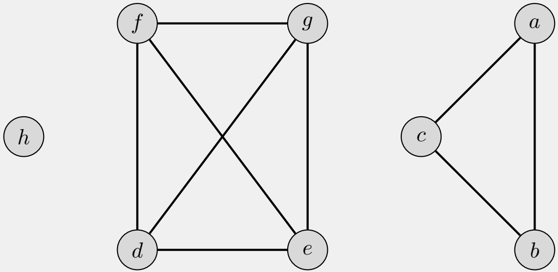
- Um *subgrafo* $H = (V', E')$ de um grafo $G = (V, E)$ é um grafo em que $V' \subseteq V$ e $E' \subseteq E$.



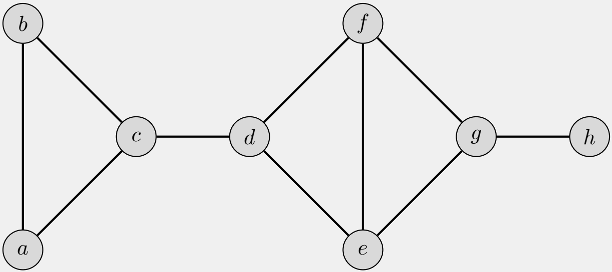
- Um *caminho* P de v_0 a v_k em um grafo é uma seqüência não-vazia $v_0, e_1, v_1, e_2, v_2, \ldots, e_k, v_k$ tal que $e_i = (v_{i-1}, v_i) \in E$ para todo $1 \leq i \leq k$.
- Um caminho é *simples* se todos os vértices dele são distintos.



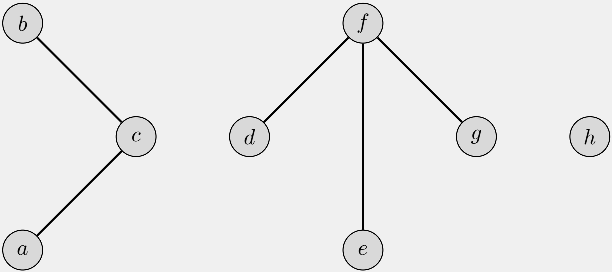
- O *tamanho* de um caminho P é o número de arestas em P.
- A *distância* entre dois vértices u e v é o tamanho do menor caminho entre eles.



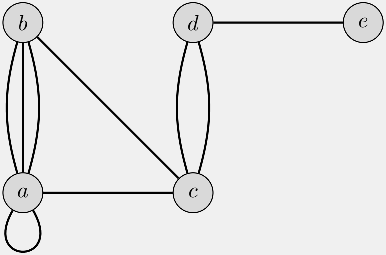
- Um grafo é *conexo* se, para todo par de vértices $u, v \in G$ existe um caminho de u a v em G.
- Quando o grafo não é conexo podemos particioná-lo em subgrafos conexos maximais chamados *componentes conexos*.



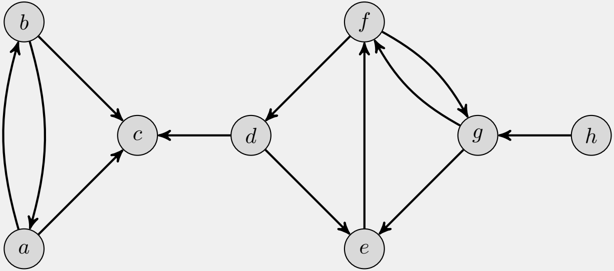
- Um *ciclo* é um caminho em que $v_0 = v_k$ e os vértices $v_0, v_1, \ldots, v_{k-1}$ são distintos.



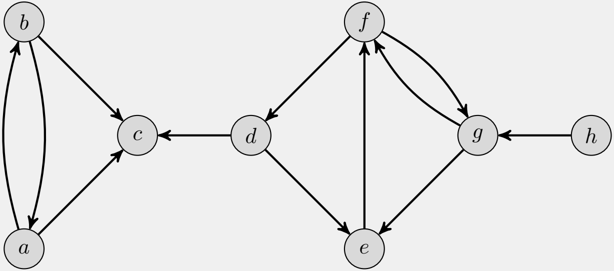
- Um grafo sem ciclos é *acíclico*.
- Um grafo acíclico e conexo é uma *árvore*. (Não confundir essa árvore com árvore enraizada, como as árvores de busca.)
- Um grafo acíclico é uma *floresta*. Cada componente conexo de uma floresta é uma árvore.



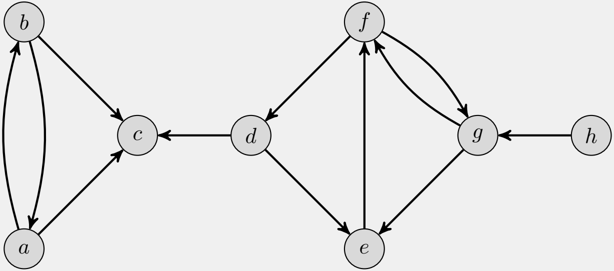
- Um *laço* (loop) é uma aresta com extremos idênticos.
- *Arestas múltiplas* ou paralelas são duas ou mais arestas com os mesmos extremos.
- Um grafo com loops ou arestas múltiplas é chamado de *multi-grafo*.



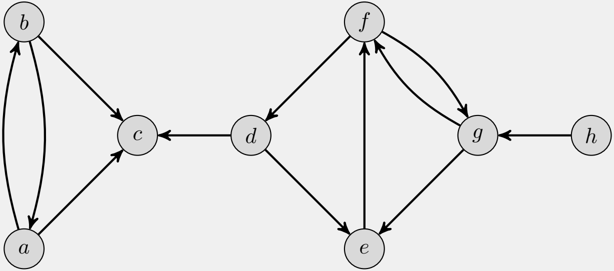
- Um *grafo orientado* é um grafo em que as arestas são pares ordenados de vértices.



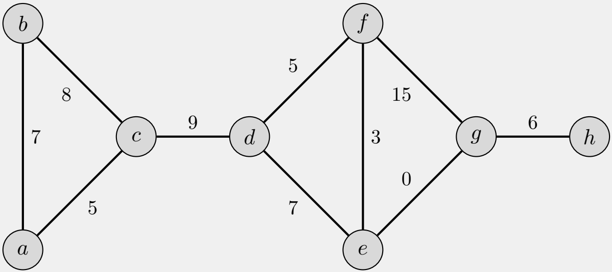
- Se $e = (u, v)$ é uma aresta de um grafo orientado G, então dizemos que e *sai* de u e *entra* em v.
- Dizemos que u é o vértice *inicial* e que v é o vértice *terminal* de e.
- Um vértice u é adjacente a um vértice v se existe uma aresta que sai de u e entra em v.



- O *grau de saída* de um vértice u, $\deg^+(u)$, é o número de arestas que saem de u.
- A *vizinhança de saída* de um vértice u é o conjunto de vértices terminais das arestas que saem de u.
- O *grau de entrada* e a *vizinhança de entrada* são análogos.



- Caminhos e ciclos em grafos orientados devem ser formados por arestas com a mesma orientação.



- Um grafo (orientado ou não-orientado) é um *grafo com pesos* se a cada aresta e do grafo está associado um valor real $w(e)$, chamado de *peso* ou *custo* da aresta.

- Um grafo é *simples* quando não possui laços ou arestas múltiplas.
- Vamos supor desse ponto em diante que os grafos são sempre simples.
- Grafos que não são simples podem ser representados e tratados pelos algoritmos com modificações diretas.

Representação de grafos

Representação de grafos em memória

- Há algumas operações bastante freqüentes em grafos:
  - Verificar se (u, v) é uma aresta do grafo (e recuperar atributos de (u, v)).
  - Percorrer a vizinhança (vizinhança de saída) de um vértice.
- A forma de representação do grafo tem impacto no uso de tempo e memória pelos algoritmos.
- Há duas representações muito usadas: matriz de adjacências e listas de adjacências.

Rótulos

- Vamos supor que se o grafo tem n vértices então os rótulos dos vértices são $\{1, 2, \ldots, n\}$.
- Se os rótulos não formam um intervalo compacto dos inteiros ou são cadeias podemos usar uma tabela de hashing (ou outra estrutura de dados) para remapeá-los.

Matriz de adjacências

- A *matriz de adjacências* de para um gafo simples $G = (V, E)$ é uma matriz quadrada de ordem $|V|$ com linhas e colunas indexadas pelos vértices em V tal que

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

- Se G não é orientado, então a matriz de adjacências é simétrica.

Memória

- A matriz tem $|V|^2$ elementos e então ocupa memória para $|V|^2$ bits.

Operações

- Verificar se (u, v) é uma aresta de um grafo: basta um acesso ao elemento (u, v) na matriz.
- Percorrer a vizinhança (vizinhança de saída) do vértice u: é necessário percorrer a linha u da matriz completamente, fazendo $|V|$ acessos a elementos da matriz.

Modificações

- A matriz de adjacências pode ser modificada facilmente para representar atributos das arestas.
- Para representar atributos dos vértices podemos usar um vetor adicional.

Listas de adjacências

- Para um grafo G a estrutura de dados *listas de adjacências* é um vetor de apontadores, um para cada vértice.
- Cada apontados i aponta para o primeiro elemento de uma lista não-ordenada de arestas, com um nó para cada vértices adjacente ao vértice i.

Memória

- Há um vetor de tamanho $|V|$, e $|V|$ listas que têm nós que guardam um inteiro e um apontador cada.
  - Para grafos não-orientados, as listas têm $2|E|$ nós e então a estrutura tem $|V| + 2|E|$ apontadores e $2|E|$ inteiros.
  - Para grafos orientados, as listas têm $|E|$ nós e então a estrutura tem $|V| + |E|$ apontadores e $|E|$ inteiros.

Operações

- Verificar se (u, v) é uma aresta de um grafo: é necessário percorrer a lista de adjacências do vértice u até encontrar v ou o fim da lista, fazendo até $\deg(u)$ $(\deg^+(u))$ acessos a nós de lista.
- Percorrer a vizinhança (vizinhança de saída) do vértice u: basta percorrer a lista de adjacências do vértice u até o fim, fazendo $\deg(u)$ $(\deg^+(u))$ acessos a nós de lista.

Modificações

- As listas de adjacências podem ser modificadas facilmente para representar atirbutos dos vértices e das arestas e para representar grafos que não são simples.

Vetor de adjacências

- Para grafos que mudam pouco ou não mudam podemos usar dois vetores de inteiros, o vetor V para os vértices e o vetor E para as arestas.
    - E contém os vizinhos dos vértices $0, 1, \ldots, |V|$, consecutivamente.
    - Para cada vértice u, $V[u]$ registra a posição em E do primeiro vizinho de u. $V[|V|] = |E|$
    - A vizinhança de u estará no intervalo $E[V[u], V[u+1] - 1]$.
- São necessários $|V| + 1 + 2|E|$ inteiros para grafos não-orientados e $|V| + 1 + |E|$ inteiros para grafos orientados.

Buscas em grafos

Buscas em grafos

- Uma busca em um grafo é uma forma sistemática de visitar os vértices e as arestas dele.
- Duas formas de busca têm propriedades interessantes: a busca em largura e a busca em profundidade.
- Essas buscas funcionam para grafos orientados ou não-orientados.
- Nos algoritmos vamos associar atributos a cada vértice. Esses atributos podem ser implementados como campos em registros, vetores ou outras estruturas.

Busca em largura

- A busca em largura a partir de um vértice inicial s visita os vértices do grafo em ordem crescente da distância de s.
- Em inglês é BFS (breath-first search).
- O algoritmo mantém o estado de cada vértice durante a busca:
  - não-marcado: não descoberto pela busca.
  - marcado: descoberto pela busca.

BFS

```
BFS(G, s)
for each vertex u ∈ G.V
    u.marked = FALSE
queue Q = ∅
s.marked = TRUE
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ G.N(u)
        if v.marked == FALSE
            ENQUEUE(Q, v)
            v.marked = TRUE
```

Número de operações

- A busca inicializa todos os vértices, cada vértice é adicionado à fila zero ou uma única vez e no máximo todas as arestas são visitadas. Então o número total de operações é proporcional ao tamanho do grafo, $|V| + |E|$.
- A correção do cálculo das distâncias é garantida pela ordem com que os vértices são adicionados à fila.

Distâncias

- O algoritmo pode ser modificado para registrar a distância do vértice inicial s até cada vértice v.
- Para todo vértice v, vamos supor que a distância de s a v vai ser armazenada em um campo $v.d$.

Árvore de busca em largura

- O algoritmo pode ser modificado para construir uma árvore de busca em largura. Essa árvore permite determinar os caminhos para alcançar os vértices a partir de s.
- Para cada vértice v, vamos supor que o vértice a partir do qual v foi alcançado durante a busca será armazenado no campo $v.\pi$.
- Para um grafo $G = (V, E)$ a árvore de busca em largura é o grafo orientado $G_\pi = (V_\pi, E_\pi)$ tal que

$$
\begin{aligned}
V_\pi &= \{v \in V : v.\pi \neq \text{NULL}\} \cup \{s\} \\
E_\pi &= \{(v.\pi, v) : v \in V_\pi - \{s\}\}
\end{aligned}
$$

BFS

```
BFS(G, s)
for each vertex u ∈ G.V
    u.d = ∞
    u.π = NULL
    u.marked = FALSE
queue Q = ∅
s.d = 0
s.marked = TRUE
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ G.N(u)
        if v.marked == FALSE
            v.d = u.d + 1
            v.π = u
            ENQUEUE(Q, v)
            v.marked = TRUE
```

Observações

- Outras modificações da busca em largura são possíveis.
- Ordens distintas na visitação dos vértices adjacentes podem produzir árvores de busca em largura diferentes, mas as distâncias serão sempre as distâncias da origem.

Impressão de caminhos em uma árvore de busca em largura

```
PRINT-PATH(G, s, v)
if v == s
    print s
elseif v.π == NULL
    print "unreachable"
else
    PRINT-PATH(G, s, v.π)
    print v
```

Busca em profundidade

- A busca em profundidade (DFS, depth-first search) tende a afastar-se da origem ao invés de explorar vizinhanças de vértices.
- Percorre o grafo inteiro.
- O algoritmo mantém o estado de cada vértice durante a busca:
  - não-marcado: não descoberto pela busca.
  - marcado: descoberto pela busca.

DFS

```
DFS(G)
for each vertex u ∈ G.V
    u.marked = FALSE
for each vertex u ∈ G.V
    if u.marked == FALSE
        DFS-VISIT(G, u)
```

```
DFS-VISIT(G, u)
for each v ∈ G.N(u)
    if v.marked == FALSE
        v.marked = TRUE
        DFS-VISIT(G, v)
```

Número de operações

- A função DFS inicializa todos os vértices e depois percorre todos eles.
- DFS-VISIT é chamada uma vez para cada vértice: apenas quando o vértice é não-marcado. Todas as arestas são visitadas.
- O número total de operações é proporcional ao tamanho do grafo, $|V| + |E|$.

Floresta de busca em profundidade

- O algoritmo pode ser modificado para construir uma floresta de busca em profundidade F.
- Para cada vértice v, o vértice a partir do qual v foi alcançado será armazenado no campo $v.\pi$.
- A floresta de busca em profundidade é um grafo orientado $F = (V, E')$ tal que

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

Timestamps e cores

- A DFS pode ser modificada para registrar um timestamp $v.d$ quando um vértice é descoberto e um timestamp $v.f$ quando um vértice é finalizado (isto é, toda a vizinhança dele foi explorada).
- O algoritmo pode usar cores para manter o estado de cada vértice durante a busca:
  - branco: não descoberto pela busca.
  - cinza: descoberto pela busca.
  - preto: finalizado.
- Os timestamps e as cores podem ser usados para classificar as arestas e em seguida encontrar ciclos, articulações, pontes e outras estruturas no grafo.

DFS

DFS(G)

```
for each vertex u ∈ G.V
        u.color = WHITE
        u.π = NULL
time = 0
for each vertex u ∈ G.V
        if u.color == WHITE
                DFS-VISIT(G, u)
```

DFS-VISIT(G, u)

```
time = time + 1
u.d = time
u.color = GRAY
for each v ∈ G.N(u)
        if v.color == WHITE
                v.π = u
                DFS-VISIT(G, v)
u.color = BLACK
time = time + 1
u.f = time
```

Observações

- Outras extensões da busca em profundidade são possíveis.
- Ordens distintas na visitação dos vértices podem produzir florestas de busca em profundidade diferentes.
- O caminho de um vértice até a raiz de uma árvore pode ser impresso usando PRINT-PATH.

Classificação de arestas em grafos orientados

- A DFS pode ser usada para classificar as arestas do grafo.
  - A aresta (u, v) é de árvore se v foi descoberto através dela.
  - A aresta (u, v) é de retorno se conecta u a um ancestral v em F.
  - A aresta (u, v) é de avanço se não é de árvore e conecta u a um descendente v em F.
  - A aresta (u, v) é de cruzamento se não é nenhuma das anteriores, ela conecta vértices sem relação de ancestralidade em F.

Classificação de arestas em grafos não-orientados

- Em grafos não-orientados uma aresta (u, v) pode ser percorrida em dois sentidos.
- Nesse caso, cada aresta vai ser classificada da primeira forma possível.
- Assim, em um grafo não-orientado, uma aresta é de árvore ou de retorno.

Classificação de arestas em grafos orientados

- Quando uma aresta (u, v) é explorada:
  - se v é branco então (u, v) é de árvore.
  - se v é cinza então (u, v) é de retorno (u é cinza e como v é o vértice cinza mais profundo na recursão, u é ancestral de v).
  - se v é preto e $u.d < v.d$ então (u, v) é de avanço (u é ancestral de v e há mais de um caminho entre u e v)
  - se v é preto e $u.d > v.d$ então (u, v) é de cruzamento (u não é ancestral de v nem v é ancestral de u).

Caminhos de custo mínimo um-para-todos

Caminhos de custo mínimo um-para-todos

- Seja $G = (V, E)$ um grafo (orientado ou não-orientado) com pesos nas arestas dado por uma função $w : E \mapsto \mathbb{R}$.
- O custo de um caminho é a soma dos pesos das arestas dele.
- O problema dos caminhos mínimos um-para-todos é encontrar os caminhos com o menor custo entre um vértice origem s e todos os outros vértices do grafo.

Algoritmo de Dijkstra

- Edsger W. Dijkstra, 1959.
- Funciona para grafos sem arestas com peso negativo.

Idéia

- Inicialmente o custo do vértice origem s até ele mesmo é 0 e para todos os outros vértices é ∞.
- A cada passo o vértice u alcançável a partir de s com o menor custo é garantidamente alcançavel por um caminho de custo mínimo. Então para cada vizinho v de u a aresta (u, v) é visitada e se o custo em v for menor ele é atualizado. u não é mais considerado pelo algoritmo.
- Uma fila de prioridades Q é usada para selecionar um vértice alcançável de s por um caminho de custo mínimo.

Algoritmo

$\textsc{Dijkstra}(G, w, s)$

for each vertex $v \in G.V$
    $v.cost = \infty$
    $v.\pi = \text{NIL}$
$s.cost = 0$
priority queue $Q = G.V$
while $Q \neq \emptyset$
    $u = \textsc{Extract-Min}(Q)$
    for each $v \in G.Adj[u]$
        if $u.cost + w(u, v) < v.cost$
            $v.cost = u.cost + w(u, v)$
            $v.\pi = u$
            $\textsc{Decrease-Cost}(Q, v, v.cost)$

Fila de prioridades por vetor

- Sejam $|V| = n$ e $|E| = m$.
- Vamos supor que usemos um vetor A de tamanho n para a fila de prioridades.
- O custo em cada vértice i fica na posição $A[i]$.
- Encontrar o custo mínimo exige percorrer A. Para remover o vértice com custo mínimo i da fila de prioridades armazenamos -1 em $A[i]$. Para reduzir o custo em um vértice i atualizamos $A[i]$.

- A inicialização (linhas 1–5) faz $O(n)$ operações, incluindo a inicialização de Q com todos os vértices.
- O while (linha 6) é executado n vezes. Cada $\textsc{Extract-Min}$ faz $O(n)$ operações e as demais linhas fazem $O(1)$ operações, sem contar o for. O número de operações feito pelo while é $O(n^2)$.
- O for (linha 8) é executado m vezes no total. No pior caso o if sempre é tomado. Todas as linhas fazem $O(1)$ operações, inclusive $\textsc{Decrease-Cost}$. O número de operações feitas pelo for é $O(m)$.
- O número total de operações é $O(n^2 + m)$.

Fila de prioridades por heap

- Vamos supor que usemos um heap de mínimo indexado por um vetor para a fila de prioridades. Cada nó do heap guarda o índice de um nó e o custo nele.
- Encontrar o mínimo no heap indexado leva tempo constante. Remover o mínimo leva tempo $O(\log n)$. Reduzir o custo de um nó no heap indexado leva tempo $O(\log n)$.

- A inicialização (linhas 1–5) faz $O(n)$ operações.
- O while (linha 6) é executado n vezes. Cada EXTRACT-MIN faz $O(\log n)$ operações e as demais linhas fazem $O(1)$ operações, sem contar o for. O número de operações do while é $O(n \log n)$.
- O for (linha 8) é executado m vezes no total. No pior caso o if sempre é tomado. DECREASE-COST faz $O(\log n)$ operações e as demais linhas fazem $O(1)$ operações. O número de operações feitas pelo for é $O(m \log n)$.
- O número total de operações é $O(m \log n + n \log n)$.