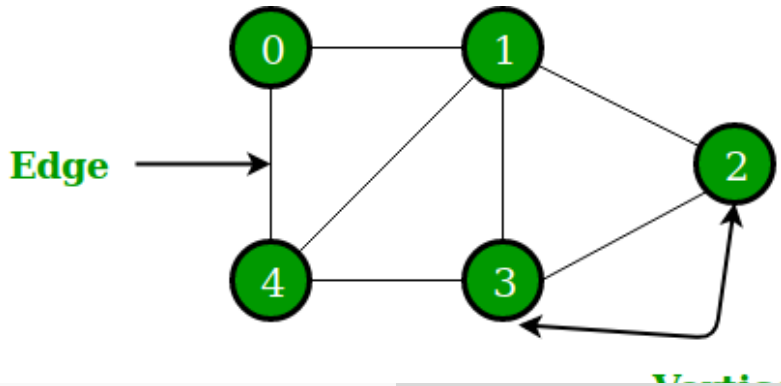
Algoritmos de grafos

Programação Avançada
Universidade Federal de Pernambuco
Adrien Durand-Petiteville
adrien.durandpetiteville@ufpe.br

Introdução aos grafos

Grafos

- Um grafo é uma estrutura de dados não-linear que consiste nos dois componentes a seguir:
  - 1. Um conjunto finito de vértices também chamados como nós.
  - 2. Um conjunto finito de par ordenado da forma (u, v) chamado como arestas.
- O par é ordenado porque (u, v) não é o mesmo que (v, u) no caso de um gráfico dirigido.
- O par da forma (u, v) indica que há uma aresta de vértice u para vértice v.
- As arestas podem conter peso/valor/custo.
- No seguinte exemplo o conjunto de vértice é $V = \{0, 1, 2, 3, 4\}$ e o conjunto de par é $E = \{01, 12, 23, 34, 04, 14, 13\}$

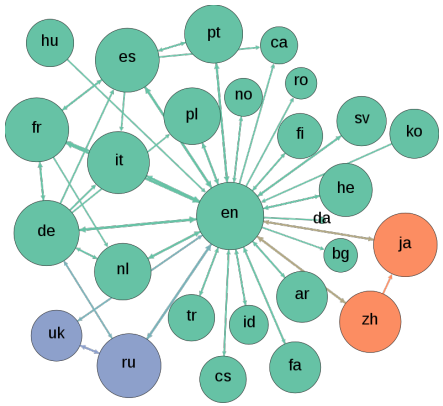


Definições

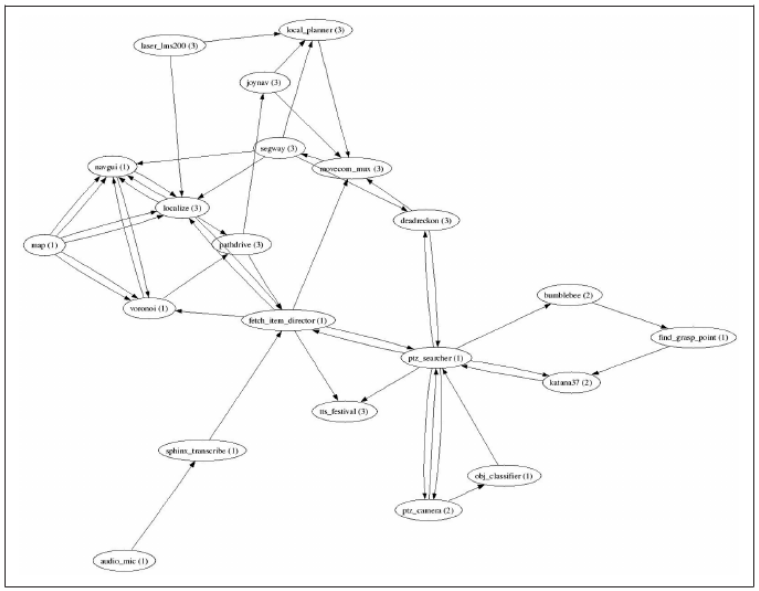
- Um grafo não direcionado é dado por
  - um conjunto V de vértices,
  - um conjunto E de arestas e
  - uma função $w : E -> P(V)$ que associa a cada aresta um subconjunto de dois ou de um elemento de V, interpretado como os pontos terminais da aresta.
- Um grafo direcionado consiste de
  - um conjunto V de vértices,
  - um conjunto E de arestas e
  - mapas $s, t : E -> V$, onde $s(e)$ é a fonte e $t(e)$ é o alvo da aresta direcionada e.
- Em um grafo com pesos, uma função adicional $E -> R$ associa um valor a cada aresta, o que pode ser considerado seu "custo"; tais grafos surgem em problemas de rota ótima tais como o problema do caixeiro viajante.

Aplicações

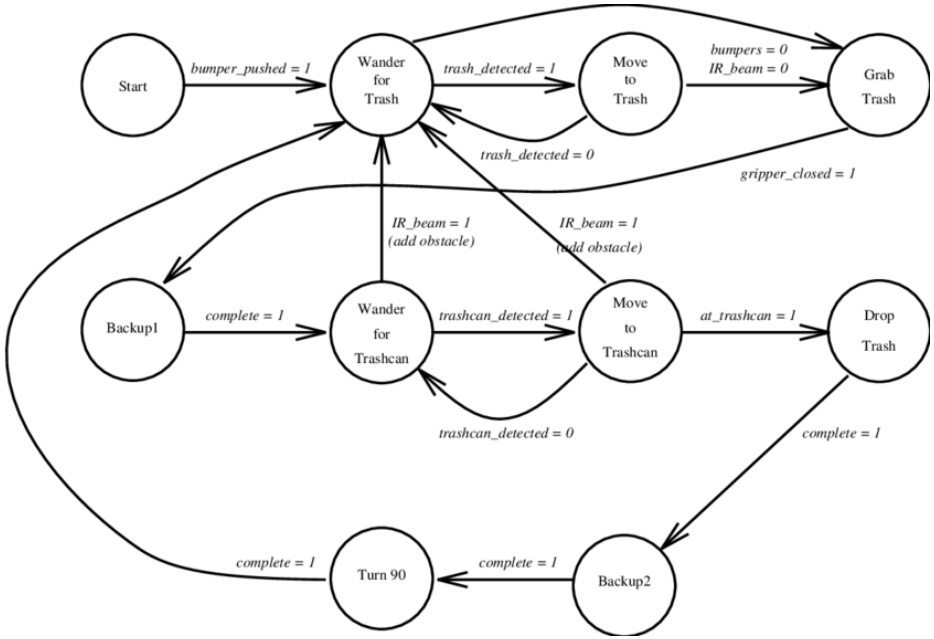
- Os grafos podem ser usados para modelar muitos tipos de relações e processos em sistemas físicos, biológicos, sociais e de informação.
- Na ciência de computação, os grafos são usados para representar redes de comunicação, organização de dados, dispositivos computacionais, o fluxo de computação, etc.
- Por exemplo, a estrutura de links de um website pode ser representada por um grafo dirigido, no qual os vértices representam páginas web e as bordas dirigidas representam links de uma página para outra.
- Uma abordagem semelhante pode ser feita para problemas nas redes sociais, viagens, biologia, desenho de chips de computador, e muitos outros campos.



Aplicações em robótica

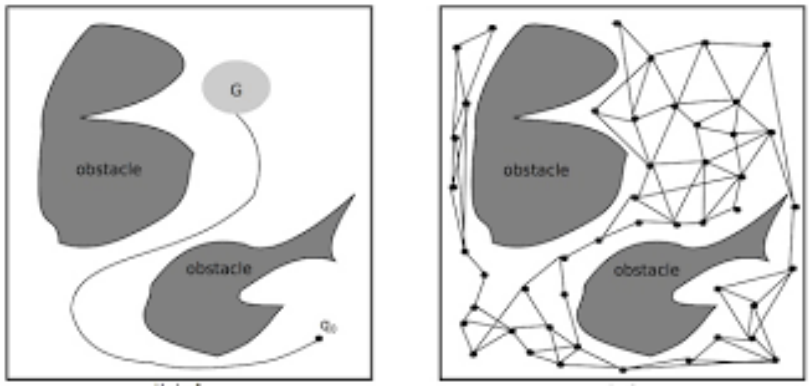


(a) Grafo ROS

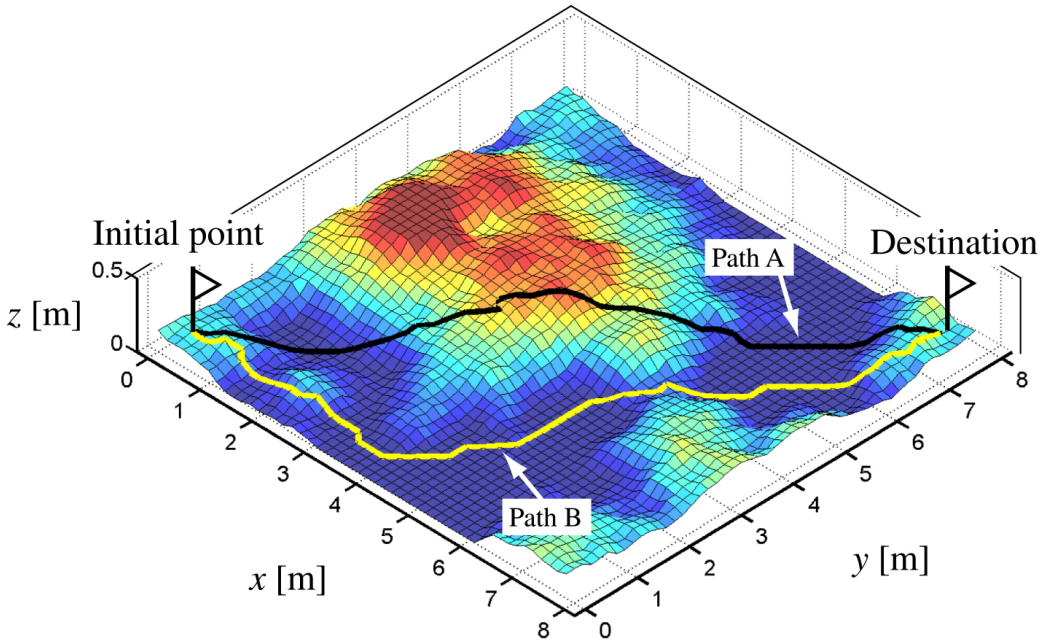


(b) Grafo de decisão

Aplicações em robótica - 2



(c) Grafo de planejamento

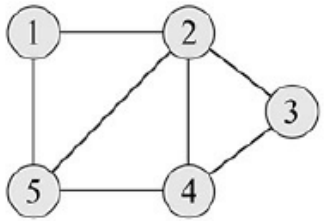


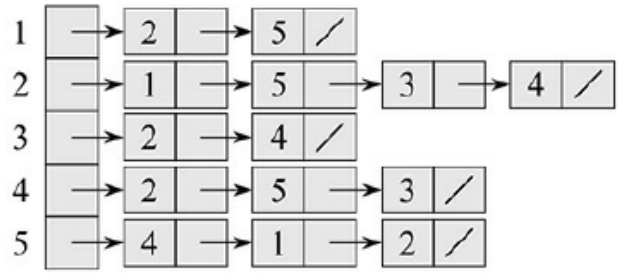
(d) Grafo de planejamento 2

Representações de grafos

Introdução

- Podemos escolher entre dois modos padrões para representar um grafo $G = (V, E)$:
  - como uma coleção de listas de adjacências;
  - 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.



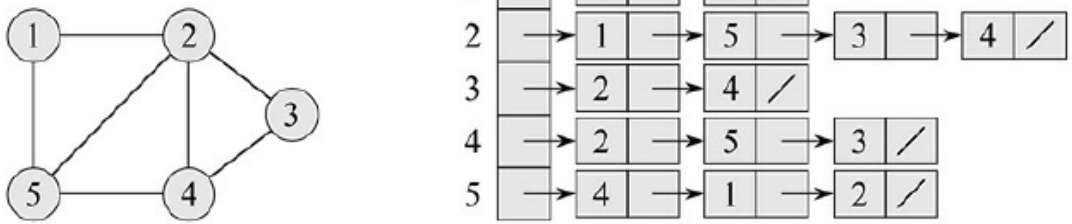


- Uma representação por matriz de adjacências pode ser preferível quando o grafo é denso - $|E|$ está próximo de $|V|^2$.

Representação por lista de adjacências

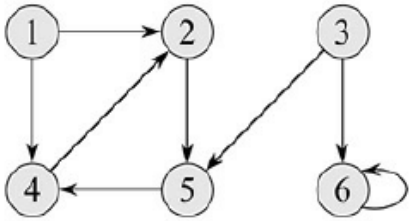
- A representação por lista de adjacências de um grafo $G = (V, E)$ consiste em um arranjo $Adj.$ de $|V|$ listas, uma para cada vértice em V.
- Para cada $u \in V$, a lista de adjacências $Adj[u]$ contém todos o vértices v tais que existe uma aresta $(u, v) \in E$.
- Isto é, $Adj[u]$ consiste em todos os vértices adjacentes a u em G.
- 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 $G.Adj[u]$.

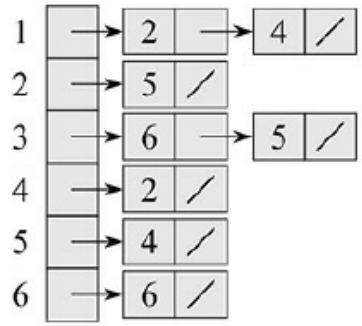
Exemplo: grafo não dirigido



- 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.

Exemplo: grafo dirigido





- 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 $Adj[u]$.

Representar grafos ponderados

- Podemos adaptar imediatamente as listas de adjacências para representar grafos ponderados (grafos nos quais cada aresta tem um peso associado, dado por uma função peso $w : E- >$).
- 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.
- A representação por lista de adjacências é bastante robusta no sentido de que podemos modificá-la para suportar muitas outras variantes de grafos.

Desvantagens

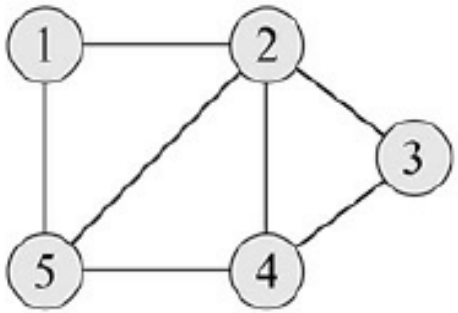
- Uma desvantagem potencial da representação por lista de adjacências é que ela não proporciona nenhum modo mais rápido para determinar se uma dada aresta (u, v) está presente no grafo do que procurar v na lista de adjacências $Adj[u]$.
- Essa desvantagem pode ser contornada por uma representação por matriz de adjacências do grafo, porém ao custo de utilizar assintoticamente mais memória.

Representação por matriz de adjacências

- No caso da representação por matriz de adjacências de um grafo $G = (V, E)$, supomos que os vértices são numerados $1, 2, ..., |V|$ de alguma maneira arbitrária.
- Então, a representação por matriz de adjacências de um grafo G consiste em uma matriz $|V|x|V|$ $A = (a_{ij})$ tal que:

$$a_{ij} = \begin{cases} 1 & \text{if } (i, j) \in E \text{ ,} \\ 0 & \text{otherwise .} \end{cases}$$

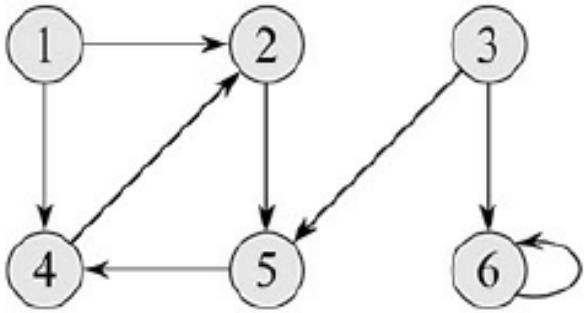
Exemplo: grafo não dirigido



	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

- Observe a simetria ao longo da diagonal principal da matriz de adjacências.
- 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.

Exemplo: grafo dirigido



	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

- 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 $Adj[u]$.

Representar grafos ponderados

- 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 ∞.

Lista de adjacências vs Matriz de adjacências

- Embora a representação por lista de adjacências seja assintoticamente, no mínimo, tão eficiente em termos de espaço quanto a representação por matriz de adjacências, matrizes de adjacências são mais simples e portanto preferíveis quando os grafos são razoavelmente pequenos.
- Além disso, matrizes de adjacências têm uma vantagem adicional para grafos não ponderados: exigem somente um bit por entrada.

Busca em largura

Introdução

- 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.
- 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.

Introdução - 2

- 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 $k + 1$.

Código de cor

- 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.
- 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.
- 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.

Árvore em largura

- A busca em largura constrói uma árvore em largura, que contém inicialmente apenas sua raiz, que é o vértice de fonte s.
- 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

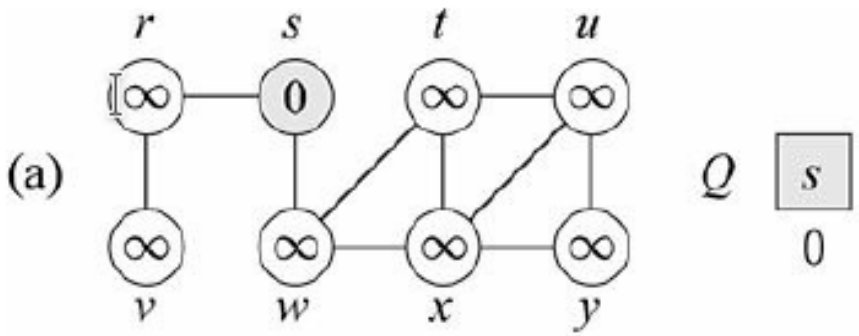
- O procedimento de busca em largura BFS mostrado a seguir supõe que o grafo de entrada $G = (V, E)$ é representado com a utilização de listas de adjacências.
- Ele anexa vários atributos adicionais a cada vértice no grafo.
- Armazenamos a cor de cada vértice $u \in V$ no atributo $u.cor$ e o predecessor de u no atributo $u.\pi$.
- Se u não tem nenhum predecessor (por exemplo, se $u = s$ ou se u não foi descoberto), então $u.\pi = NIL$.
- 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 para gerenciar o conjunto de vértices cinzentos.

BFS Pseudo-código

```
BFS(G, s)
 1    for cada vértice u ∈ V[G] − {s}
 2        u.cor = BRANCO
 3        u.d = ∞
 4        u.π = NIL
 5    s.cor = CINZENTO
 6    s.d = 0
 7    s.π = NIL
 8    Q = ∅
 9    ENQUEUE(Q, s)
10    while Q ≠ ∅
11        u = DEQUEUE(Q)
12        for cada v = Adj[u]
13            if v.cor == BRANCO
14                v.cor == CINZENTO
15                v.d = u.d + 1
16                v.π = u
17                ENQUEUE(Q, v)
18        u.cor = PRETO
```

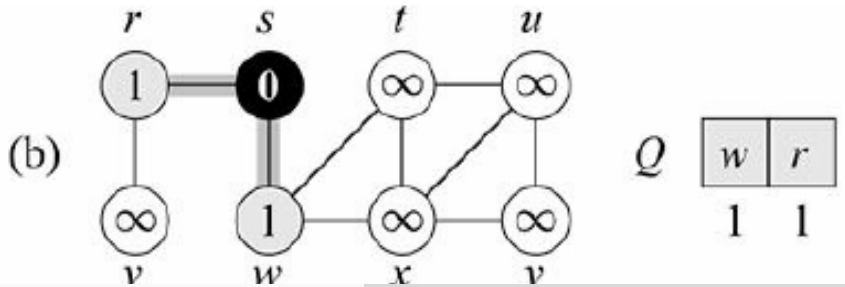
BFS Pseudo-código - 1

- 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 0, 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.



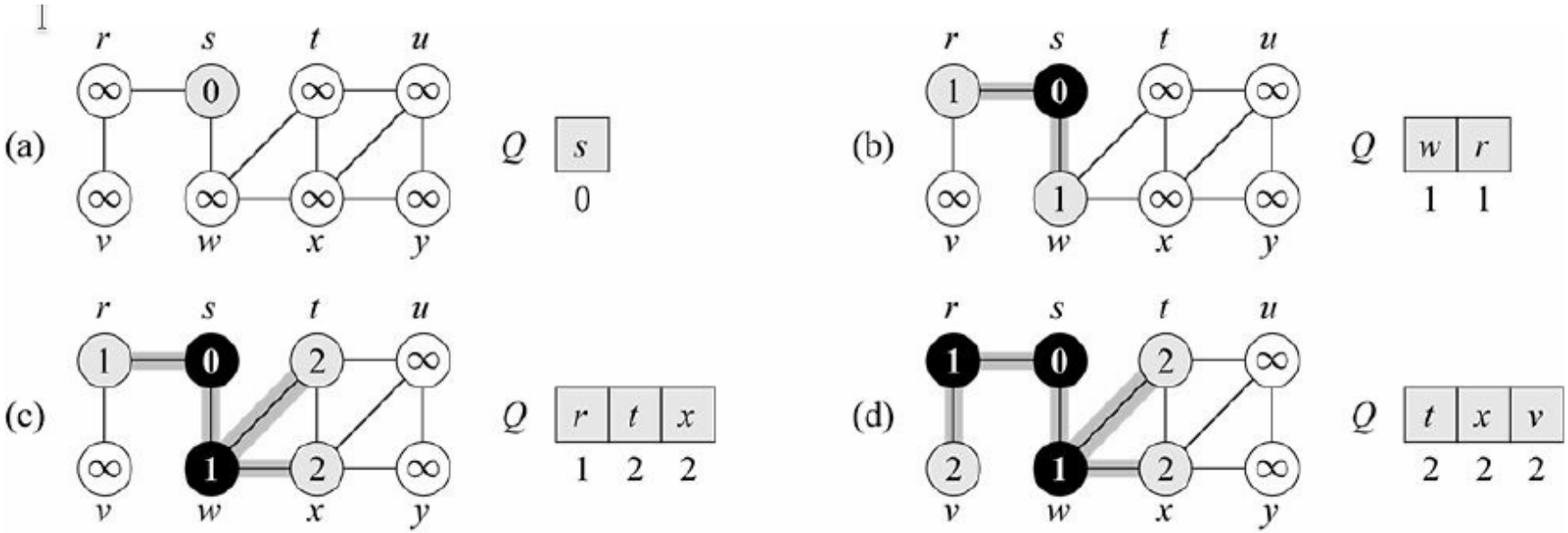
BFS Pseudo-código - 2

- 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 linha 11 determina o vértice cinzento u no início da fila Q e o remove de Q.
- 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 v de cinzento, define sua distância $v.d$ como $u.d + 1$, registra u como seu pai $v.\pi$ 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.



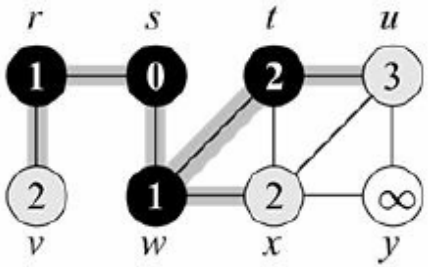
Exemplo

- 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.



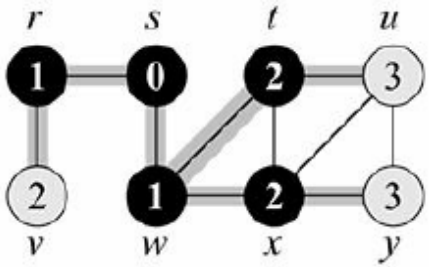
Exemplo - 2

(e)



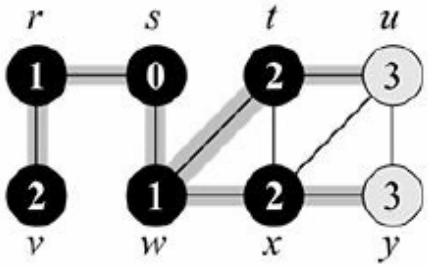
Q | x | v | u
2 | 2 | 3

(f)



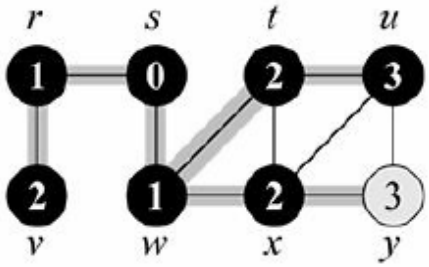
Q | v | u | y
2 | 3 | 3

(g)



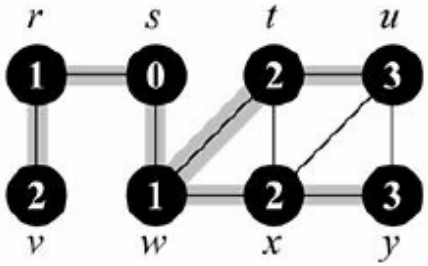
Q | u | y
3 | 3

(h)



Q | y
3

(i)



Q $\varnothing$

Caminhos mínimos

- A busca em largura encontra a distância até cada vértice que pode ser alcançado em um grafo $G = (V, E)$ partindo de um determinado vértice de fonte $s \in V$.
- Defina a distância do caminho mínimo $d(s, v)$ de s a v como o número mínimo de arestas em qualquer caminho do vértice s ao vértice v
- Se não há nenhum caminho de s a v, então $d(s, v) = \infty$.
- Denominamos um caminho de comprimento $d(s, v)$ de s a v caminho mínimo de s a v.

Teorema: Correção da busca em largura

- Seja $G = (V, E)$ um grafo dirigido ou não dirigido, e suponha que BFS seja executado em G partindo de um dado vértice de fonte $s \in V$.
- Então, durante sua execução, BFS descobre todo vértice $v \in V$ que pode ser alcançado da fonte s e, no término, $v.d = d(s, v)$ para todo $v \in V$.
- Além disso, para qualquer vértice $v \neq s$ que pode ser alcançado de s, um dos caminhos mínimos de s a v é um caminho mínimo de s a $v.\pi$ seguido pela aresta $(v.\pi, v)$.

Busca em profundidade

Introdução

- 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

Introdução - 2

- 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 $G_\pi = (V, E_\pi)$, 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_π são arestas de árvore.

Código de cor

- 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.

Carimbo de tempo

- 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);
  - 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)
- 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.
- 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.

DFS Pseudo-código

```
DFS(G)
1    for cada vértice u ∈ V[G]
2        u.cor = BRANCO
3        u.π = NIL
4    tempo = 0
5    for cada vértice u ∈ V[G]
6        if u.cor == BRANCO
7            DFS-VISIT(G, u)

DFS-VISIT(G, u)
1    tempo = tempo + 1
2    u.d = tempo
3    u.cor = CINZENTO
4    for cada v ∈ G.Adj[u]
5        if v.cor == BRANCO
6            v.π = u
7            DFS-VISIT(G, v)
8    u.cor = PRETO
9    tempo = tempo + 1
10   u.f = tempo
```

DFS Pseudo-código - 2

- 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.
- As linhas 1-3 pintam todos os vértices de branco e inicializam seus atributos π 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-VISIT`(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 $f[u]$.

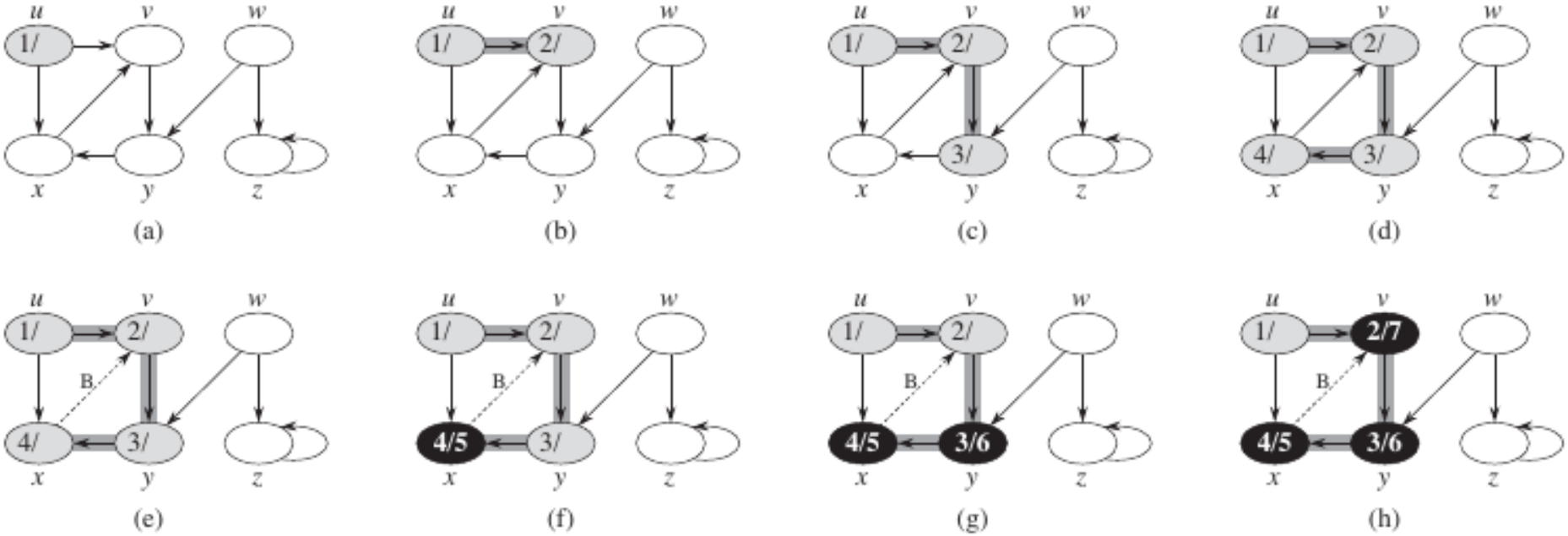
DFS Pseudo-código - 3

- Em cada chamada `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 $f[u]$.

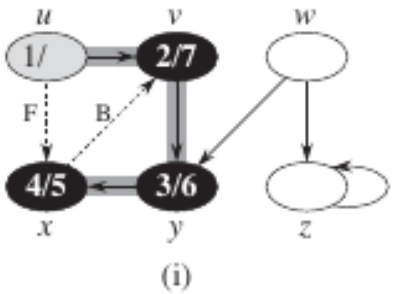
DFS Pseudo-código - 4

- Observe que os resultados da busca em profundidade podem depender da ordem em que a linha 5 de `DFS` examina os vértices e da ordem em que a linha 4 de `DFS-VISIT` visita os vizinhos de um vértice.
- Essas diferentes ordens de visitação tendem a não causar problemas na prática, já que em geral podemos usar eficientemente qualquer resultado da busca em profundidade e obter, em essência, resultados equivalentes.

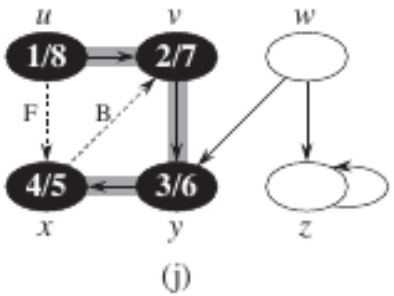
Exemplo



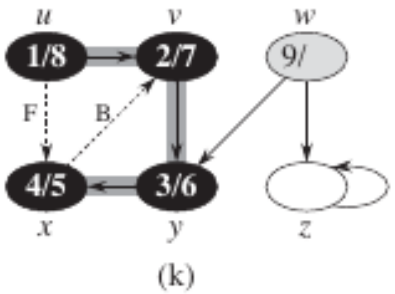
Exemplo - 2



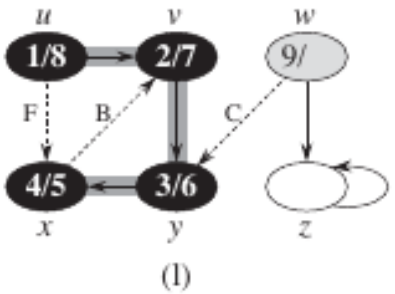
(i)



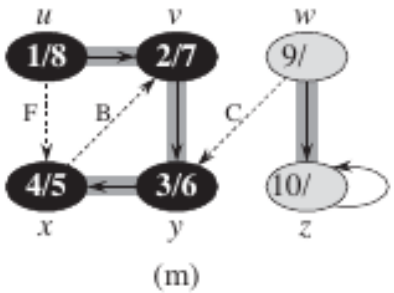
(j)



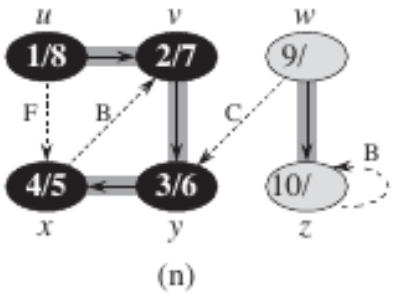
(k)



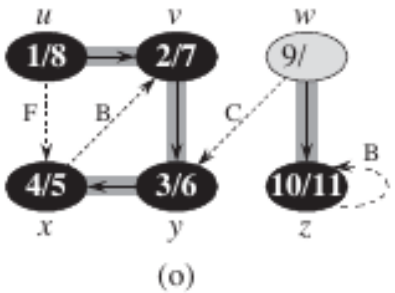
(l)



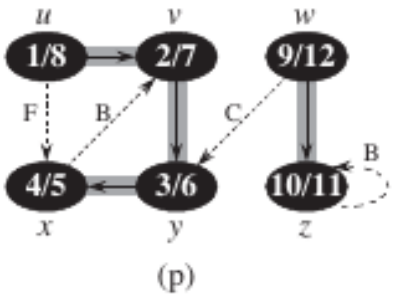
(m)



(n)



(o)



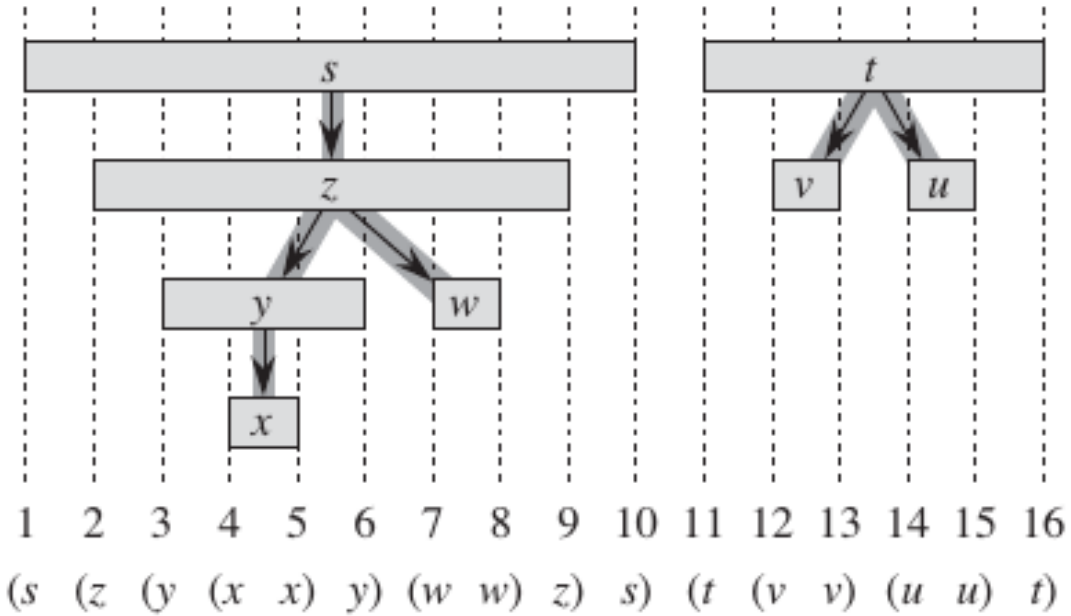
(p)

Propriedades da busca em profundidade

- A busca em profundidade produz informações valiosas sobre a estrutura de um grafo.
- Talvez a propriedade mais básica da busca em profundidade seja que o subgrafo predecessor G_π realmente forma uma floresta de árvores, já que a estrutura das árvores em profundidade reflete exatamente a estrutura de chamadas recursivas de `DFS-VISIT`.
- Isto é, $u = v.\pi$ se e somente se `DFS-VISIT`(G, v) foi chamado durante uma busca da lista de adjacências de u.
- Além disso, o vértice v é um descendente do vértice u na floresta em profundidade se e somente se v é descoberto durante o tempo em que u é cinzento.

Estrutura parentizada

- Uma outra propriedade importante da busca em profundidade é que os tempos de descoberta e término têm estrutura parentizada.
- Se representarmos a descoberta do vértice u com um parêntese à esquerda "$(u$" e representarmos seu término por um parêntese à direita "$u)$", então a história de descobertas e términos gera uma expressão bem formada, no sentido de que os parênteses estão adequadamente aninhados.



Teorema dos parênteses

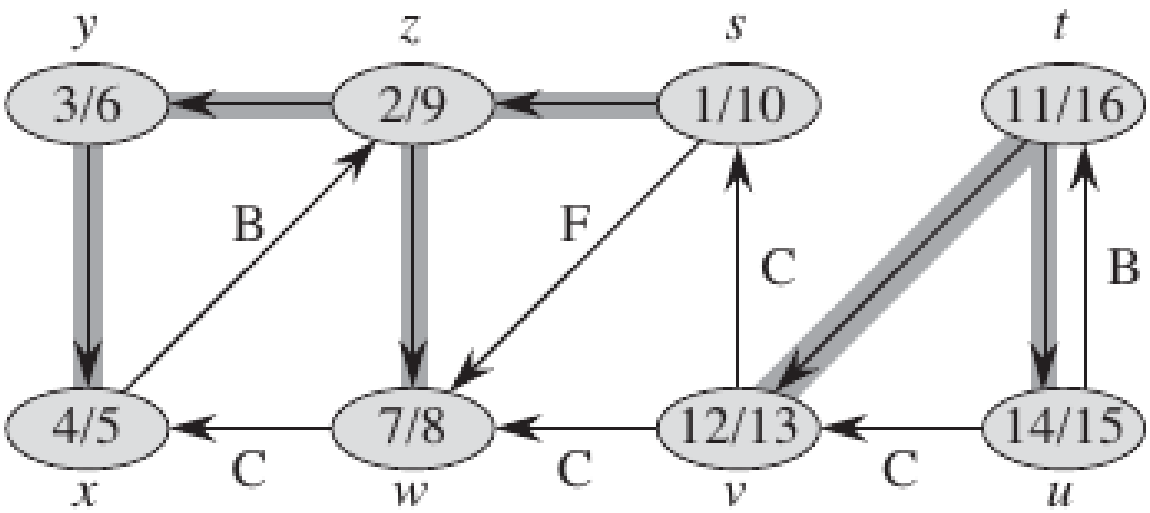
Em qualquer busca em profundidade de um grafo (dirigido ou não dirigido) $G = (V, E)$, para quaisquer dois vértices u e v, exatamente uma das três condições seguintes é válida:

- Os intervalos $[u.d, u.f]$ e $[v.d, v.f]$ são completamente disjuntos, e nem u nem v é um descendente do outro na floresta em profundidade.
- O intervalo $[u.d, u.f]$ está contido inteiramente dentro do intervalo $[v.d, v.f]$, e u é um descendente de v em uma árvore em profundidade.
- O intervalo $[v.d, v.f]$ está contido inteiramente dentro do intervalo $[u.d, u.f]$, e v é um descendente de u em uma árvore em profundidade.

Classificação de arestas

- Uma outra propriedade interessante da busca em profundidade é que a busca pode ser usada para classificar as arestas do grafo de entrada $G = (V, E)$.
- O tipo de aresta nos da informações importantes sobre um grafo.
- Podemos definir quatro tipos de arestas em termos da floresta em profundidade G_π produzida por uma busca em profundidade em G:
  - **Arestas de árvore** são arestas na floresta em profundidade G_π.
    A aresta (u, v) é uma aresta de árvore se v foi descoberto primeiro pela exploração da aresta (u, v).
  - **Arestas de retorno** são as arestas (u, v) que conectam um vértice u a um ancestral v em uma árvore em profundidade.
    Consideramos laços, que podem ocorrer em grafos dirigidos, como arestas de retorno.
  - **Arestas diretas** são as arestas (u, v) não de árvore que conectam um vértice u a um descendente v em uma árvore em profundidade.
  - **Arestas cruzadas** são todas as outras arestas.
    Elas podem estar entre vértices na mesma árvore, desde que um vértice não seja um ancestral do outro, ou podem estar entre vértices em diferentes árvores de busca.

Exemplo



Caminhos mínimos de fonte única

Introdução

- Um motorista deseja encontrar a rota mais curta possível do Rio de Janeiro a São Paulo.
- Dado um mapa rodoviário do Brasil no qual a distância entre cada par de interseções adjacentes esteja marcada, como ele pode determinar essa rota mais curta?
- Um modo possível seria enumerar todas as rotas do Rio de Janeiro a São Paulo, somar as distâncias em cada rota e selecionar a mais curta.
- Porém, é fácil ver que até mesmo se deixarmos de lado as rotas que contêm ciclos, o motorista teria de examinar um número enorme de possibilidades, a maioria das quais simplesmente não valeria a pena considerar.
- Por exemplo, uma rota do Rio de Janeiro a São Paulo passando por Brasília é sem dúvida uma escolha ruim porque Brasília está várias centenas de quilômetros fora do caminho.

Problema

- Em um problema de caminhos mínimos, temos um **grafo dirigido ponderado** $G = (V, E)$, com função peso $w : E- > \mathbb{R}$ que mapeia arestas para pesos de valores reais.
- O peso do caminho $p =< v_0, v_1, ..., v_k >$ é a soma dos pesos de suas arestas constituintes:

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

- Definimos o peso do caminho mínimo de u a v por:

$$\delta(u, v) = \begin{cases} min\{w(p) : u \xrightarrow{p} v\} & \text{Se há um caminhode } u \text{ para } v, \\ \infty & \text{case contrário} \end{cases}$$

- Então, um caminho mínimo do vértice u ao vértice v é definido como qualquer caminho p com peso $w(p) = \delta(u, v)$.

Problema - 2

- No exemplo da rota entre o Rio de Janeiro e São Paulo, podemos modelar o mapa rodoviário como um grafo: vértices representam interseções, arestas representam segmentos de estradas entre interseções e pesos de arestas representam distâncias rodoviárias.
- Nossa meta é encontrar um caminho mínimo de um dado entroncamento de rodovias no Rio de Janeiro a um dado entroncamento de rodovias em São Paulo.
- Pesos de arestas podem representar outras medidas que não sejam distâncias, como tempo, custo, multas, prejuízos ou qualquer outra quantidade que se acumule linearmente ao longo de um caminho e que seria interessante minimizar.
- O algoritmo de busca em largura é um algoritmo de caminhos mínimos que funciona em grafos não ponderados, isto é, grafos nos quais cada aresta tem peso unitário.

Subestrutura ótima de um caminho mínimo

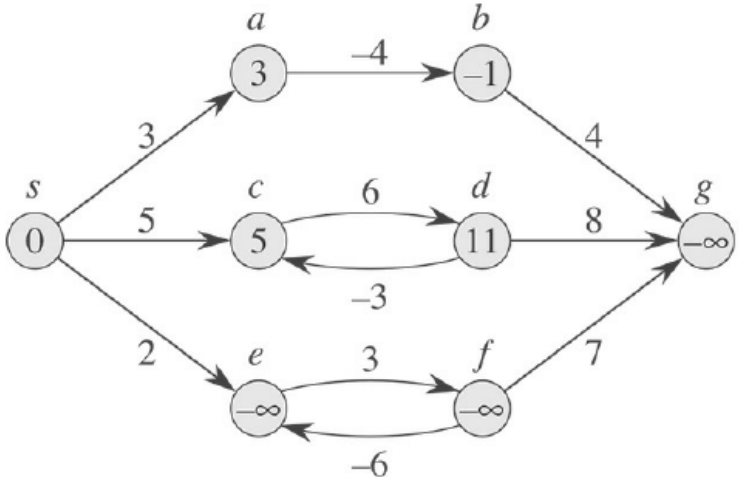
- Em geral, algoritmos de caminhos mínimos se baseiam na seguinte propriedade:
  Um caminho mínimo entre dois vértices contém outros caminhos mínimos.
- **Lema**: Subcaminhos de caminhos mínimos são caminhos mínimos

Arestas de peso negativo

- Algumas instâncias do problema de caminhos mínimos de fonte única podem incluir arestas cujos pesos são negativos.
- Se o grafo $G = (V, E)$ não contém nenhum ciclo de peso negativo que possa ser alcançado da fonte s, então para todo $v \in V$, o peso do caminho mínimo $\delta(s, v)$ permanece bem definido, mesmo que tenha um valor negativo.
- Contudo, se o grafo contém um ciclo de peso negativo que possa ser alcançado a partir de s, os pesos de caminhos mínimos não são bem definidos.
- Nenhum caminho de s a um vértice no ciclo pode ser um caminho mínimo - sempre podemos encontrar um caminho de peso menor seguindo o caminho "mínimo" proposto e depois percorrendo o ciclo de peso negativo.
- Se houver um ciclo de peso negativo em algum caminho de s a v, definiremos $\delta(s, v) = -\infty$.

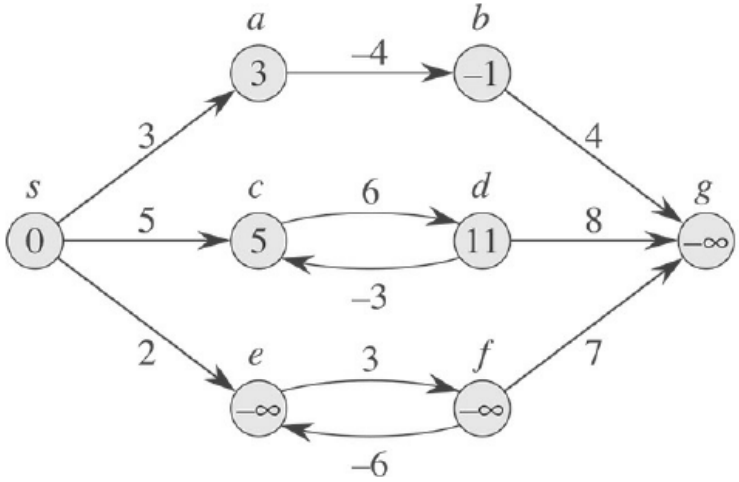
Exemplo

- Como há somente um caminho de s a a (o caminho $< s, a >$), temos $\delta(s, a) = w(s, a) = 3$.
- De modo semelhante, há somente um caminho de s a b, e assim $\delta(s, b) = w(s, a) + w(a, b) = 3 + (-4) = -1$.



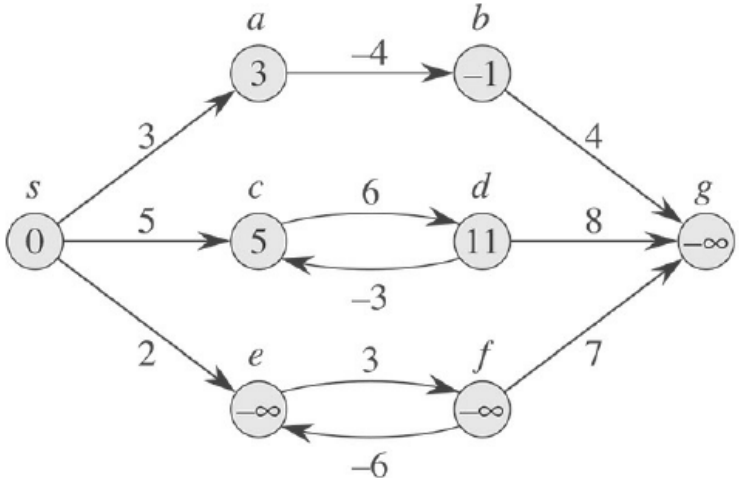
Exemplo - 2

- Há um número infinito de caminhos de s a c: $< s, c >, < s, c, d, c >, < s, c, d, c, d, c >$, e assim por diante.
- Como o ciclo $< c, d, c >$, tem peso $6 + (-3) = 3 > 0$, o caminho mínimo de s a c é $< s, c >$, com peso $\delta(s, c) = w(s, c) = 5$.
- Do mesmo modo, o caminho mínimo de s a d é $< s, c, d >$, com peso $\delta(s, d) = w(s, c) + w(c, d) = 11$.



Exemplo - 3

- Analogamente, há um número infinito de caminhos de s a e: $< s, e >, < s, e, f, e >, < s, e, f, e, f, e >$, e assim por diante.
- Porém, visto que o ciclo $< e, f, e >$, tem peso $3 + (-6) = -3 < 0$, não há nenhum caminho mínimo de s a e.



- Percorrendo o ciclo de peso negativo $< e, f, e >$ um número arbitrário de vezes, podemos encontrar caminhos de s a e com pesos negativos arbitrariamente grandes e, assim, $\delta(s, e) = -\infty$.
- De modo semelhante, $\delta(s, f) = -\infty$.
- Como g pode ser alcançado de f, também podemos encontrar caminhos com pesos negativos arbitrariamente grandes de s a g e, portanto, $\delta(s, g) = -\infty$.

Arestas de peso negativo - 2

- Alguns algoritmos de caminhos mínimos, como o algoritmo de Dijkstra, consideram que todos os pesos de arestas no grafo de entrada são não negativos, como no exemplo do mapa rodoviário.
- Outros, como o algoritmo de Bellman-Ford, permitem arestas de peso negativo no grafo de entrada e produzem uma resposta correta desde que nenhum ciclo de peso negativo possa ser alcançado da fonte.
- Normalmente, se houver tal ciclo de peso negativo, o algoritmo poderá detectar e relatar sua existência.

Ciclos

- Um caminho mínimo não pode conter um ciclo de peso negativo.
- Nem pode conter um ciclo de peso positivo, já que remover o ciclo do caminho produz um caminho com os mesmos vértices de fonte e destino, e um peso de caminho mais baixo.
- Isto é, se $p' =< v_0, v_1, ..., v_k >$ é um caminho e $c =< v_i, v_{i+1}, ..., v_j >$ é um ciclo de peso positivo nesse caminho (de modo que $v_i = v_j$ e $w(c) > 0$), então o caminho $p' =< v_0, v_1, ..., v_i, v_{j+1}, v_{j+2}, ..., v_k >$ tem peso $w(p') = w(p) - w(c) < w(p)$ e, portanto, p não pode ser um caminho mínimo de v_0 a v_k.

Representação de caminhos mínimos

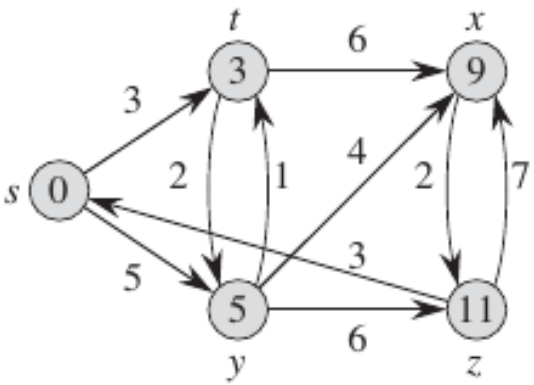
- Muitas vezes, desejamos calcular não apenas pesos de caminhos mínimos, mas também os vértices nos caminhos mínimos.
- A representação que usamos para caminhos mínimos é semelhante à que utilizamos para árvores em largura.
- Dado um grafo $G = (V, E)$, mantemos para cada vértice $v \in V$ um predecessor $v.\pi$ que é um outro vértice ou NIL.
- Os algoritmos de caminhos mínimos definem os atributos π de modo que a cadeia de predecessores que se origina em um vértice v percorra um caminho mínimo de s a v em sentido contrário.

Árvore de caminhos mínimos

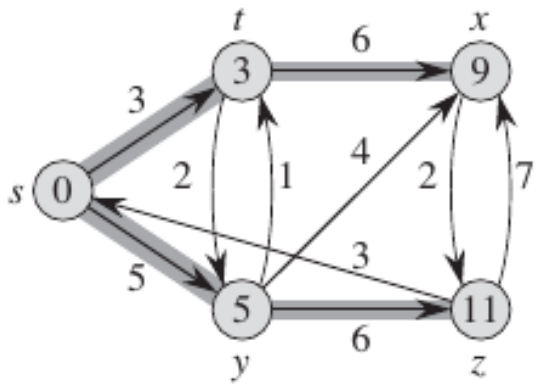
- Seja $G = (V, E)$ um grafo dirigido ponderado com função peso $w : E -> \mathbb{R}$, e considere que G não suponha nenhum ciclo de peso negativo que possa ser alcançado do vértice de fonte $s \in V$, de modo que esses caminhos mínimos são bem definidos.
- Uma árvore de caminhos mínimos com raiz em s é um subgrafo dirigido $G' = (V', E')$, onde $V' \subseteq V$ e $E' \subseteq E$, tal que
  - 1. V' é o conjunto de vértices que podem ser alcançados de s em G,
  - 2. G' forma uma árvore enraizada com raiz s e
  - 3. para todo $v \in V'$, o único caminho simples de s a v em G' é um caminho mínimo de s a v em G.

Exemplos

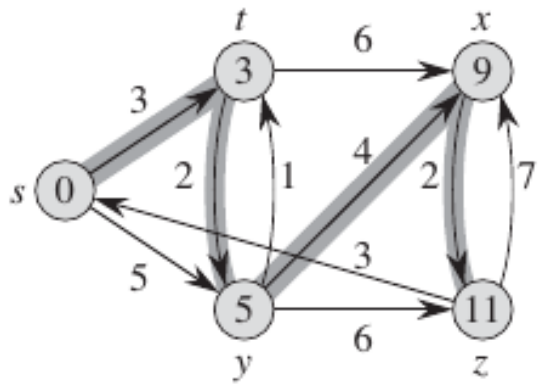
- Caminhos mínimos não são necessariamente únicos nem são necessariamente únicas as árvores de caminhos mínimos.



(a)



(b)



(c)

Relaxamento

- Os algoritmos nesta parte usam a técnica de relaxamento.
- Para cada vértice $v \in V$, mantemos um atributo $v.d$, que é um limite superior para o peso de um caminho mínimo da fonte s a v.
- Denominamos $v.d$ uma estimativa de caminho mínimo.
- Inicializamos as estimativas de caminhos mínimos e predecessores pelo seguinte procedimento de tempo $Q(V)|$:

```
INITIALIZE-SINGLE-SOURCE(G, s)
1  for cada vértice v ∈ V[G]
2      v.d = ∞
3      v.π = NIL
4  s.d = 0
```

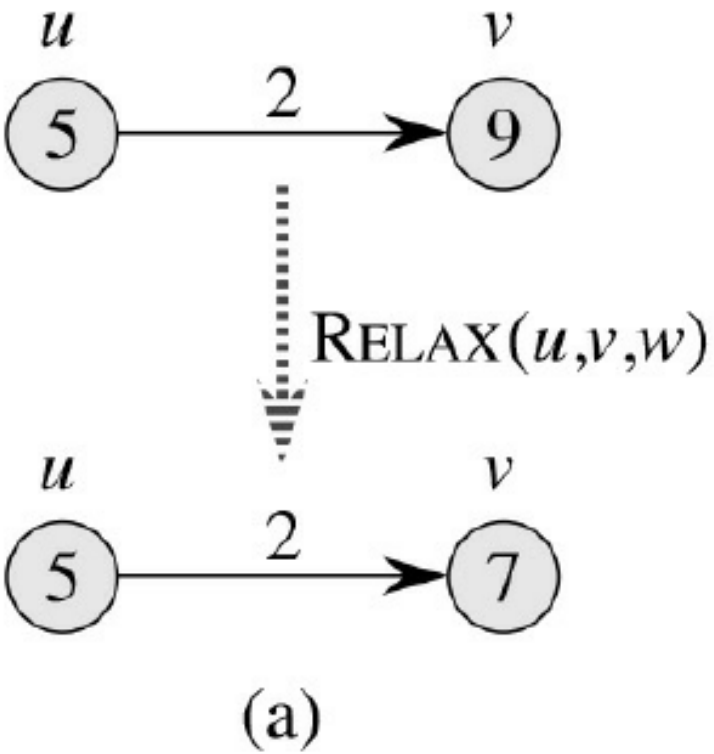
- Após a inicialização, temos $v.\pi = \text{NIL}$ para todo $v \in V$, $s.d = 0$ e $v.d = \infty$ para $v \in V - s$.

Relaxamento - 2

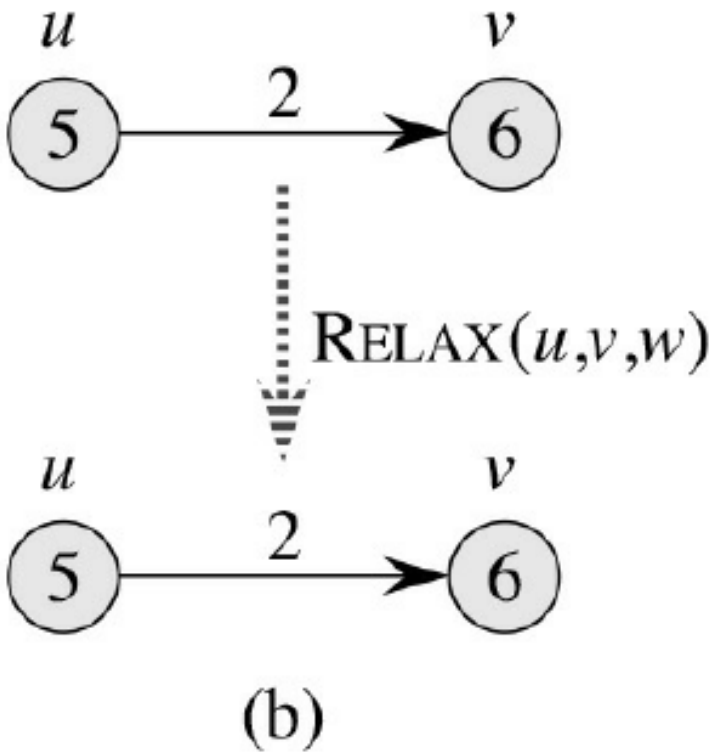
- O processo de relaxar uma aresta (u, v) consiste em testar se podemos melhorar o caminho mínimo até v que encontramos até agora passando por u e, em caso positivo, atualizar $v.d$ e $v.\pi$.
- Uma etapa de relaxamento pode diminuir o valor da estimativa do caminho mínimo $v.d$ e atualizar o atributo predecessor de v, $v.\pi$.
- O seguinte código executa uma etapa de relaxamento para a aresta (u, v) no tempo $\mathbf{O}(1)$.

```
RELAX(u, v, w)
1   if v.d > u.d + w(u, v)
2       v.d = u.d + w(u, v)
3       v.π = u
```

Exemplo



(a)



(b)

Relaxamento - 3

- Cada algoritmo neste capítulo chama `INITIALIZE-SINGLE-SOURCE` e depois relaxa arestas repetidamente.
- Além disso, o relaxamento é o único meio de mudar estimativas de caminhos mínimos e predecessores.
- As diferenças entre os algoritmos apresentados neste capítulo são a quantidade de vezes que relaxam cada aresta e a ordem em que relaxam arestas.
- O algoritmo de Dijkstra e o algoritmo de caminhos mínimos para grafos acíclicos dirigidos relaxam cada aresta exatamente uma vez.
- O algoritmo de BellmanFord relaxa cada aresta $|V| - 1$ vezes.

O algoritmo de Bellman-Ford

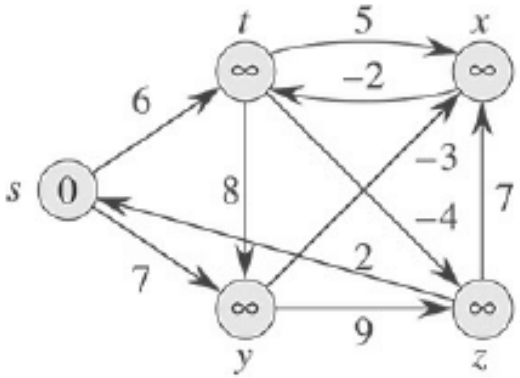
- O algoritmo de Bellman-Ford resolve o problema de caminhos mínimos de fonte única no caso geral no qual os pesos das arestas podem ser negativos.
- Dado um grafo dirigido ponderado $G = (V, E)$ com fonte s e função peso $w : E -> \mathbb{R}$, o algoritmo de Bellman-Ford devolve um valor booleano que indica se existe ou não um ciclo de peso negativo que pode ser alcançado da fonte.
- Se tal ciclo existe, o algoritmo indica que não há nenhuma solução.
- Se tal ciclo não existe, o algoritmo produz os caminhos mínimos e seus pesos.

O algoritmo de Bellman-Ford - 2

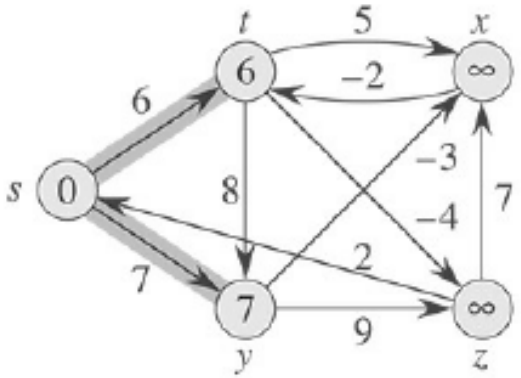
- O algoritmo relaxa arestas diminuindo progressivamente uma estimativa $v.d$ do peso de um caminho mínimo da fonte s a cada vértice $v \in V$ até chegar ao peso propriamente dito do caminho mínimo $\delta(s, v)$.
- O algoritmo retorna TRUE se e somente se o grafo não contém nenhum ciclo de peso negativo que possa ser alcançado da fonte.

```
BELLMAN-FORD(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i = 1 to |V[G]| − 1
3     for cada aresta (u, v) ∈ E[G]
4        RELAX(u, v, w)
5  for cada aresta (u, v) ∈ E[G]
6     if v.d > u.d + w(u, v)
7        return FALSE
8  return TRUE
```

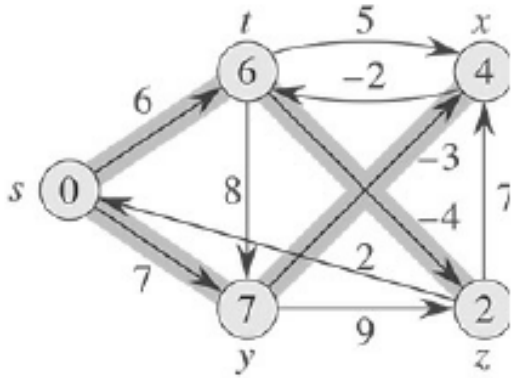
Exemplo


t
5
x
∞
−2
∞
6
−3
s
0
8
−4
7
2
7
∞
9
∞
y
z

(a)

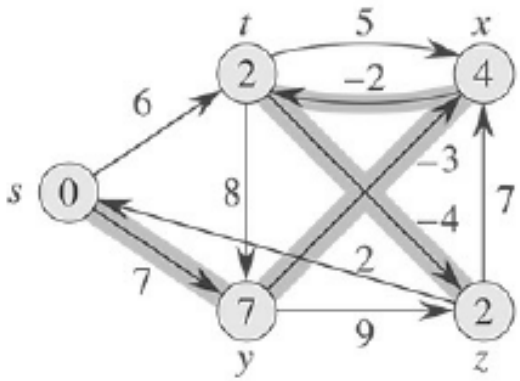

t
5
x
6
−2
∞
6
−3
s
0
8
−4
7
2
7
7
9
∞
y
z

(b)

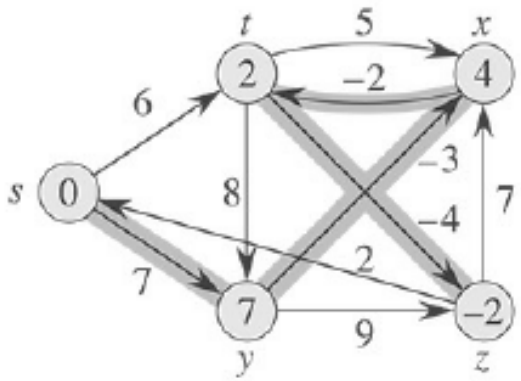

t
5
x
6
−2
4
6
−3
s
0
8
−4
2
7
7
9
2
y
z

(c)

Exemplo - 2



(d)



(e)

Exemplo - 3

Execução do algoritmo de Bellman-Ford para um grafo com cinco vértices.

- Depois de inicializar os valores de d e p de todos os vértices na linha 1, o algoritmo faz $|V| - 1$ passagens pelas arestas do grafo.
- Cada passagem é uma iteração do laço `for` das linhas 2-4 e consiste em relaxar cada aresta do grafo uma vez.
- As figuras (b)-(e) mostram o estado do algoritmo após cada uma das quatro passagens pelas arestas.
- Depois de executar $|V| - 1$ passagens, as linhas 5-8 verificam se há um ciclo de peso negativo e devolvem o valor booleano adequado.

Algoritmo de Dijkstra

- O algoritmo de Dijkstra resolve o problema de caminhos mínimos de fonte única em um grafo dirigido ponderado $G = (V, E)$ para o caso no qual todos os pesos de arestas são não negativos.
- Então, nesta parte suporemos que $w(u, v) \geq 0$ para cada aresta $(u, v) \in E$.
- Como veremos, com uma boa implementação, o tempo de execução do algoritmo de Dijkstra é inferior ao do algoritmo de Bellman-Ford

Algoritmo de Dijkstra - 2

- O algoritmo de Dijkstra mantém um conjunto S de vértices cujos pesos finais de caminhos mínimos que partem da fonte s já foram determinados.
- O algoritmo seleciona repetidamente o vértice $u \in V - S$ que tem a mínima estimativa do caminho mínimo, adiciona u a S e relaxa todas as arestas que saem de u.
- Na implementação a seguir, usamos uma fila de prioridades mínimas Q de vértices cujas chaves são os valores de d.

Pseudo código

```
DIJKSTRA(G, w, s)
1   INITIALIZE-SINGLE-SOURCE(G, s)
2   S = ∅
3   Q = V[G]
4   while Q ≠ ∅
5       u = EXTRACT-MIN(Q)
6       S = S ∪ {u}
7       for cada vértice v ∈ G.Adj[u]
8           RELAX(u, v, w)
```

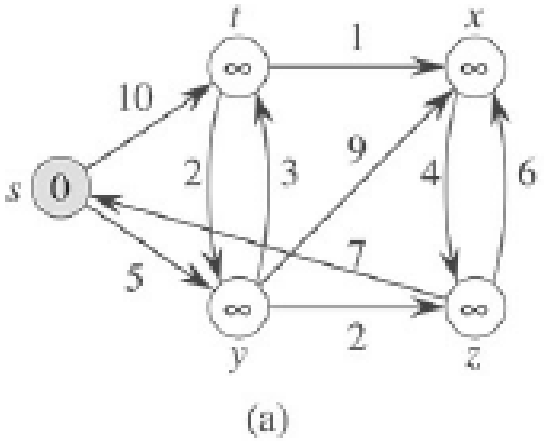
Pseudo código - 2

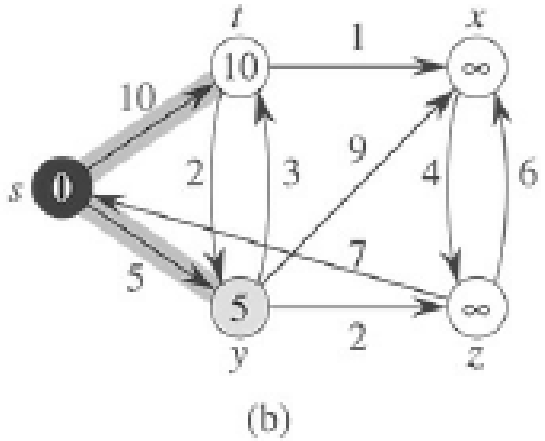
- A linha 1 inicializa os valores de d e p do modo usual.
- A linha 2 inicializa o conjunto S como o conjunto vazio.
- O algoritmo mantém o invariante $Q = V - S$ no início de cada iteração do laço `while` das linhas 4-8.
- A linha 3 inicializa a fila de prioridades mínimas Q para conter todos os vértices em V.
  Visto que $S = /0$ nesse momento, o invariante é verdadeiro após a linha 3.

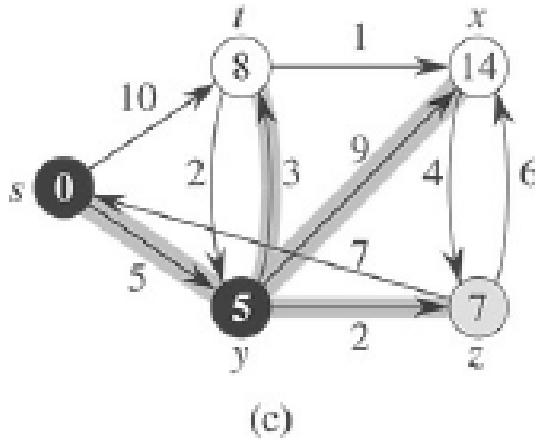
Pseudo código - 3

- Em cada passagem pelo laço `while` das linhas 4-8, a linha 5 extrai um vértice u de $Q = V - S$ e a linha 6 o adiciona ao conjunto S, mantendo assim o invariante.
- Portanto, o vértice u tem a menor estimativa de caminhos mínimos em comparação com qualquer vértice em $V - S$.
- Então, as linhas 7-8 relaxam cada aresta(u, v) que sai de u, atualizando assim a estimativa $v.d$ e o predecessor $v.\pi$ se, passando por u, pudermos melhorar o caminho mínimo até v que encontramos até aqui.
- Observe que o algoritmo nunca insere vértices em Q após a linha 3 e que cada vértice é extraído de Q e adicionado a S exatamente uma vez, de modo que o laço `while` das linhas 4-8 itera exatamente $|V|$ vezes.

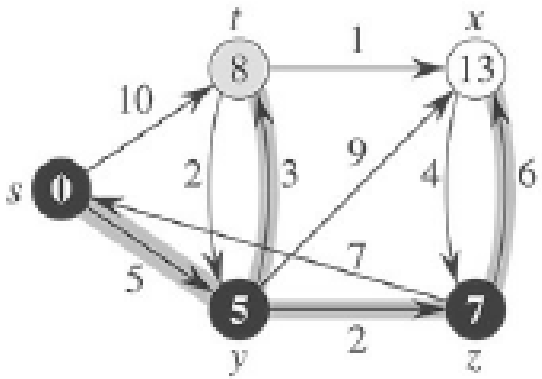
Exemplo


t
x
s
y
z
0
∞
∞
∞
∞
10
1
2
3
9
4
6
5
7
2
(a)

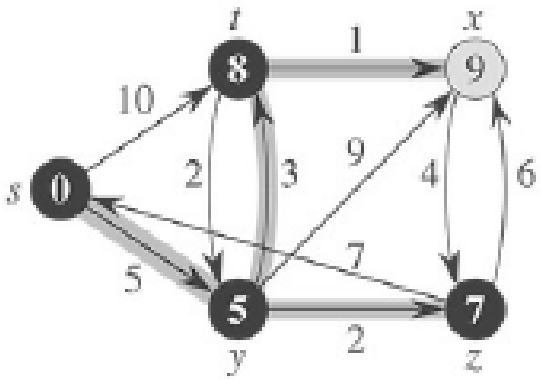

t
x
s
y
z
0
10
∞
5
∞
10
1
2
3
9
4
6
5
7
2
(b)


t
x
s
y
z
0
8
14
5
7
10
1
2
3
9
4
6
5
7
2
(c)

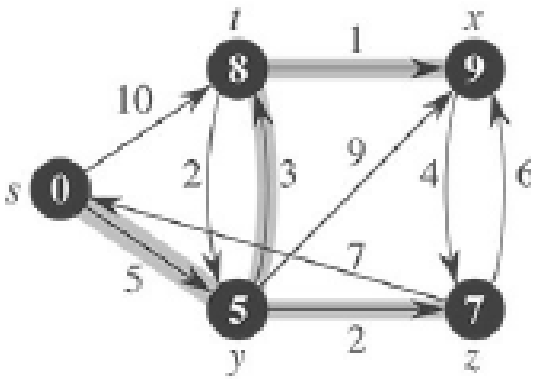
Exemplo - 2



(d)



(e)



(f)