ACH2024

Aula 13 – Grafos:

Caminhos de peso mínimo (shortest paths na literatura)

Algoritmos de Dijkstra

Profa. Ariane Machado Lima

Aula passada

Caminhos de peso mínimo

Um caminho **mais curto** é aquele com **menor número de arestas**

Muitas vezes não estamos interessados no número de arestas, e sim no custo do caminho (soma dos pesos das arestas do caminho), ou seja, no caminho de peso mínimo

A aplicação direta da busca em largura, como feita para caminhos mais curtos, não é mais suficiente

Infelizmente, esse problema é também chamado “caminho mais curto” (ex: Cormen e Ziviani)

Usarei o termo “caminho mais curto” como sinônimo de “caminho de peso mínimo” nesta aula por usar os slides do Ziviani

Assume-se um grafo **direcionado** e **ponderado**

Caminhos mais curtos de s a v: quando ele não existe?

- Se v não é alcançável por s:

$$\delta(s, v) = \text{infinito}$$

- Se v fizer parte de um ciclo negativo (ie, com peso total negativo), ou se tal ciclo estiver em um caminho de s a v:

$$\delta(s, v) = -\text{infinito}$$

Caminhos mais curtos – quando eles não existem

Rótulo do Vértice

Custo do caminho mais curto do vértice s até ele

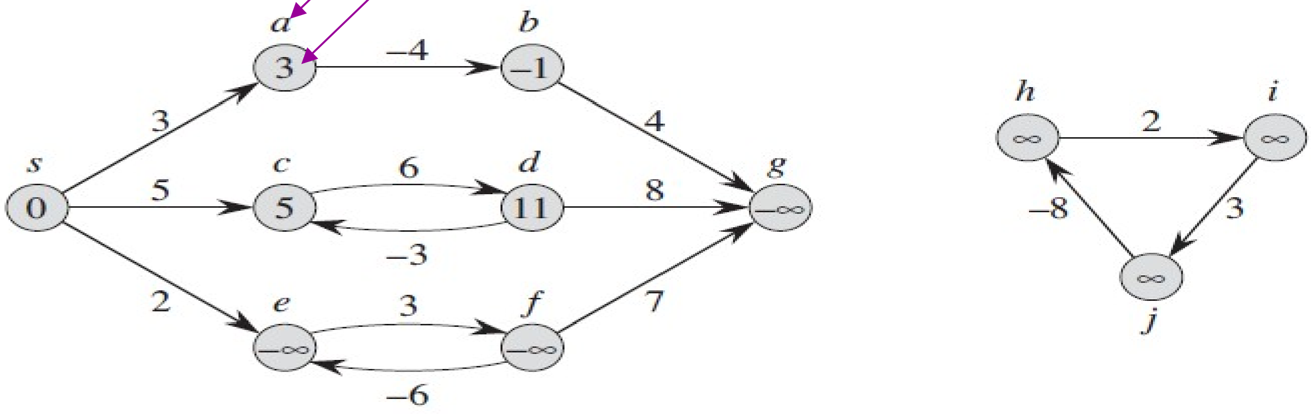


Figura: Livro do Cormem cap 24

Algoritmos (os dois que veremos)

- Recebem o vértice origem s como parâmetro
- Calculam o caminho mínimo de um vértice s a CADA UM dos outros vértices
- Para cada vértice calculam:
  - d[v]: distância do vértice origem s até v (soma dos pesos)
  - π[v]: antecessor do vértice v no caminho mínimo de s a v
    - ImprimeCaminho(s, v, π) da aula 8 (slide 71) usa esse π (antecessor) para imprimir o caminho mínimo de s a v

Algoritmos para construção de árvores de caminhos mais curtos (origem única)

- Caso quase geral (arestas podem possuir pesos negativos mas não ciclos negativos)
  - **Bellman-Ford**
- Caso em que todas as arestas possuem valores de peso não negativos (**mais eficiente**)
  - **Dijkstra**

Relaxamento

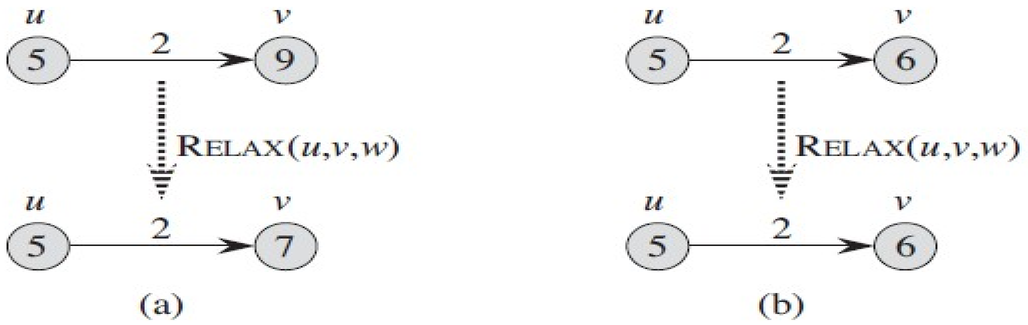
Técnica usada por algoritmos de caminhos mais curtos

Cada vértice v terá um valor d[v], que é uma estimativa de pior caso (limite superior) do custo mínimo do caminho de s (origem) a v

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

Relaxamento

Relaxar uma aresta (u,v): verificar se d[v] pode ser decrementada ao se considerar um caminho de s a v passando por u (ou seja, verificar se usar essa aresta melhora a estimativa atual):



w representa a informação dos pesos

RELAX(u, v, w)
1 **if** d[v] $>$ d[u] $+ w(u, v)$
2 d[v] $=$ d[u] $+ w(u, v)$
3 π[v] $= u$

Profa. Ariane Machado Lima

Algoritmo Bellman-Ford

Resolve o caso geral (arestas podem ter pesos negativos)

Retorna falso se o grafo tiver um ciclo negativo

```
BELLMAN-FORD(G, w, s)
INITIALIZE-SINGLE-SOURCE(G, s)
for i = 1 to |G.V| − 1
    for each edge (u, v) ∈ G.E
        RELAX(u, v, w)
for each edge (u, v) ∈ G.E
    if d[v] > d[u] + w(u, v)
        return FALSE
return TRUE
```

Como o caminho de peso mínimo não tem ciclo, tem comprimento no máximo |V|-1

Logo, em |V|-1 rodadas, todas as arestas deste caminho são corretamente relaxadas para seus valores reais

Complexidade do Algoritmo Bellman-Ford

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

L. 1: O(V)

L. 2-4: O(VA)

L. 5-7: O(A)

Total: O(VA)

Algoritmo de Dijkstra

Considerando que todas as arestas são não-negativas

```
DIJKSTRA(G, w, s)
    INITIALIZE-SINGLE-SOURCE(G, s)
    S = ∅
    Q = G.V
    while Q ≠ ∅
        u = EXTRACT-MIN(Q)
        S = S ∪ {u}
        for each vertex v ∈ G.Adj[u]
            RELAX(u, v, w)
```

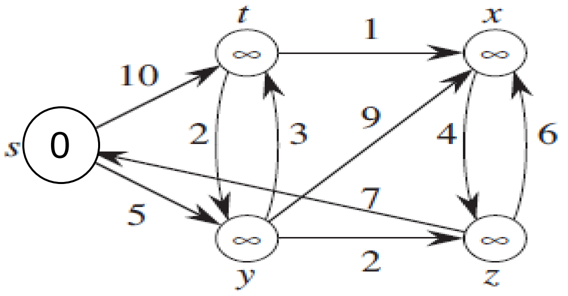
S: usado para prova de corretude no livro do Cormen
(conjunto de vértices já processados)

Q é uma fila de prioridades baseada no valor d
(quanto menor o d maior a prioridade)

Algoritmo de Dijkstra - Exemplo

```
DIJKSTRA(G, w, s)
    INITIALIZE-SINGLE-SOURCE(G, s)
    S = ∅
    Q = G.V
    while Q ≠ ∅
        u = EXTRACT-MIN(Q)
        S = S ∪ {u}
        for each vertex v ∈ G.Adj[u]
            RELAX(u, v, w)
```

```
RELAX(u, v, w)
    if d[v] > d[u] + w(u, v)
        d[v] = d[u] + w(u, v)
        π[v] = u
```



(a)

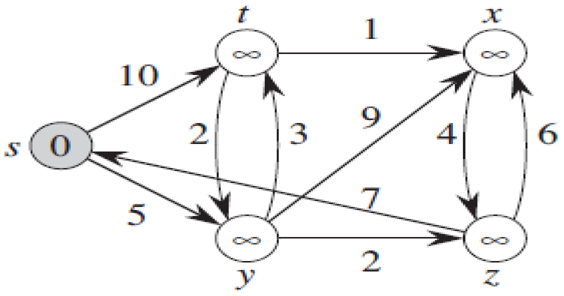
Vértices brancos estão em Q, cinza é o que acaba de ser removido, pretos já tiveram todas as arestas que saem dele ralaxadas

Algoritmo de Dijkstra - Exemplo

```
DIJKSTRA(G, w, s)
INITIALIZE-SINGLE-SOURCE(G, s)
S = ∅
Q = G.V
while Q ≠ ∅
    u = EXTRACT-MIN(Q)
    S = S ∪ {u}
    for each vertex v ∈ G.Adj[u]
        RELAX(u, v, w)
```

```
RELAX(u, v, w)
if d[v] > d[u] + w(u, v)
    d[v] = d[u] + w(u, v)
    п[v] = u
```

Vértices brancos estão em Q, cinza é o que acaba de ser removido, pretos já tiveram todas as arestas que saem dele relaxadas



(a)

Algoritmo de Dijkstra - Exemplo

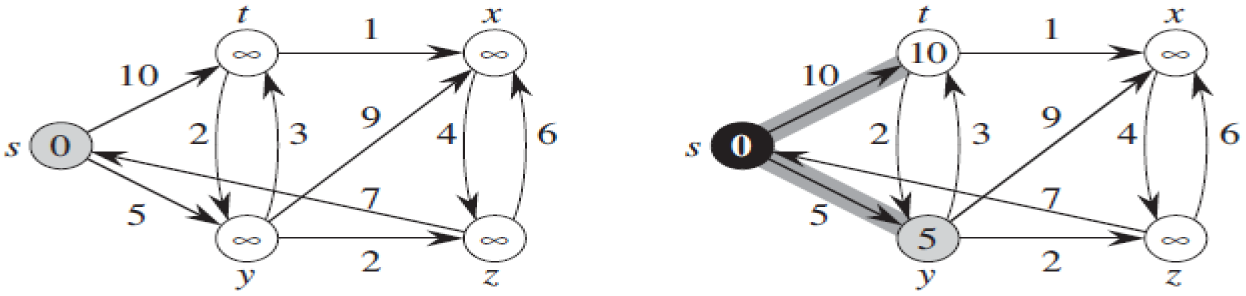
RELAX(u, v, w)

```
if d[v] > d[u] + w(u, v)
    d[v] = d[u] + w(u, v)
    π[v] = u
```

Vértices brancos estão em Q, cinza é o que acaba de ser removido, pretos já tiveram todas as arestas que saem dele relaxadas

DIJKSTRA(G, w, s)

```
INITIALIZE-SINGLE-SOURCE(G, s)
S = Ø
Q = G.V
while Q ≠ Ø
    u = EXTRACT-MIN(Q)
    S = S ∪ {u}
    for each vertex v ∈ G.Adj[u]
        RELAX(u, v, w)
```



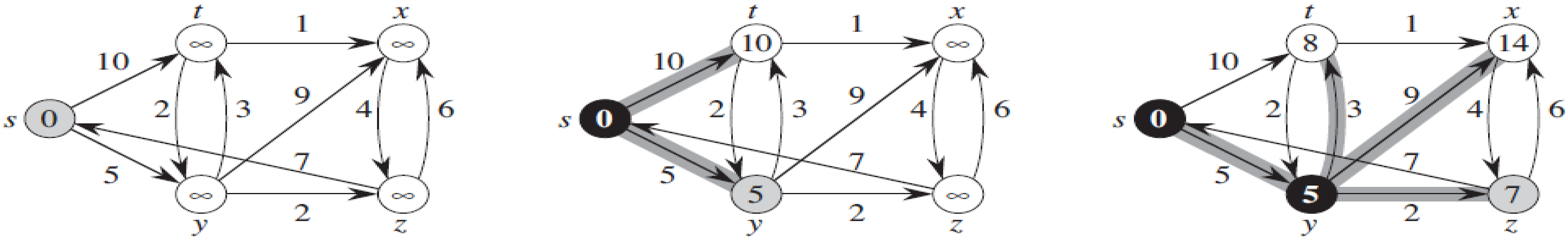
(a) (b)

Algoritmo de Dijkstra - Exemplo

```
RELAX(u, v, w)
if d[v] > d[u] + w(u, v)
    d[v] = d[u] + w(u, v)
    π[v] = u
```

Vértices brancos estão em Q, cinza é o que acaba de ser removido, pretos já tiveram todas as arestas que saem dele relaxadas

```
DIJKSTRA(G, w, s)
INITIALIZE-SINGLE-SOURCE(G, s)
S = ∅
Q = G.V
while Q ≠ ∅
    u = EXTRACT-MIN(Q)
    S = S ∪ {u}
    for each vertex v ∈ G.Adj[u]
        RELAX(u, v, w)
```



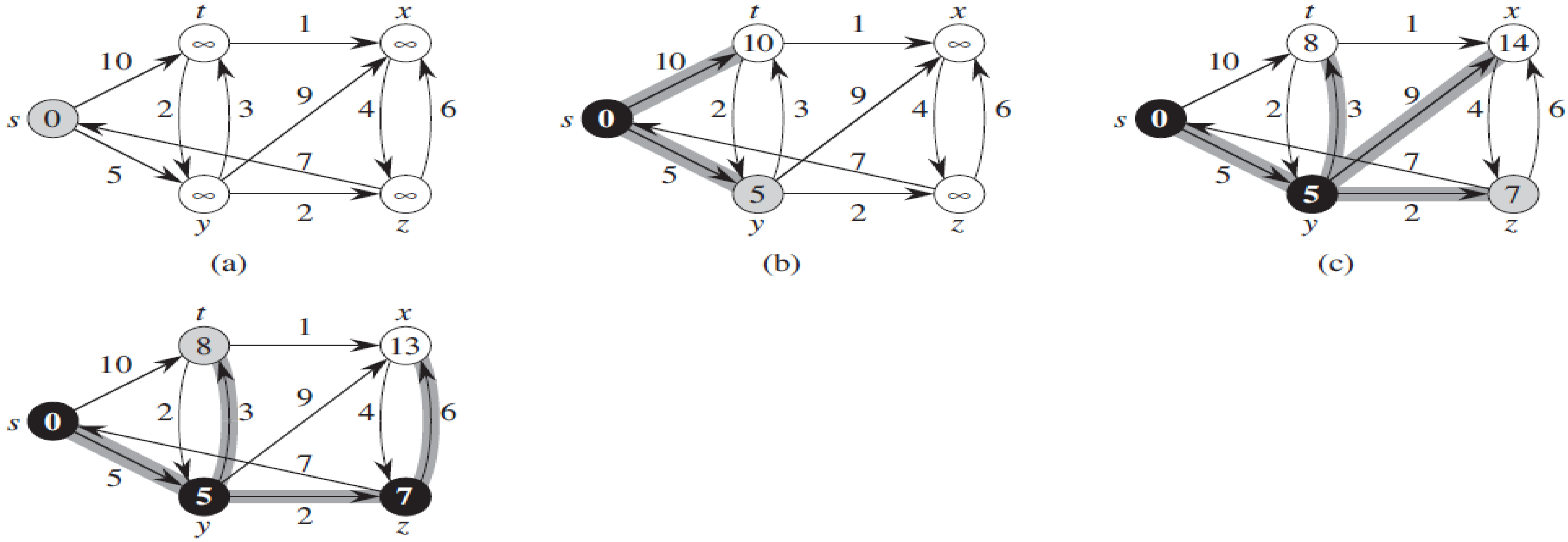
(a) (b) (c)

Algoritmo de Dijkstra - Exemplo

```
DIJKSTRA(G, w, s)
INITIALIZE-SINGLE-SOURCE(G, s)
S = Ø
Q = G.V
while Q ≠ Ø
    u = EXTRACT-MIN(Q)
    S = S ∪ {u}
    for each vertex v ∈ G.Adj[u]
        RELAX(u, v, w)
```

```
RELAX(u, v, w)
if d[v] > d[u] + w(u, v)
    d[v] = d[u] + w(u, v)
    π[v] = u
```

Vértices brancos estão em Q, cinza é o que acaba de ser removido, pretos já tiveram todas as arestas que saem dele relaxadas



(a) (b) (c) (d)

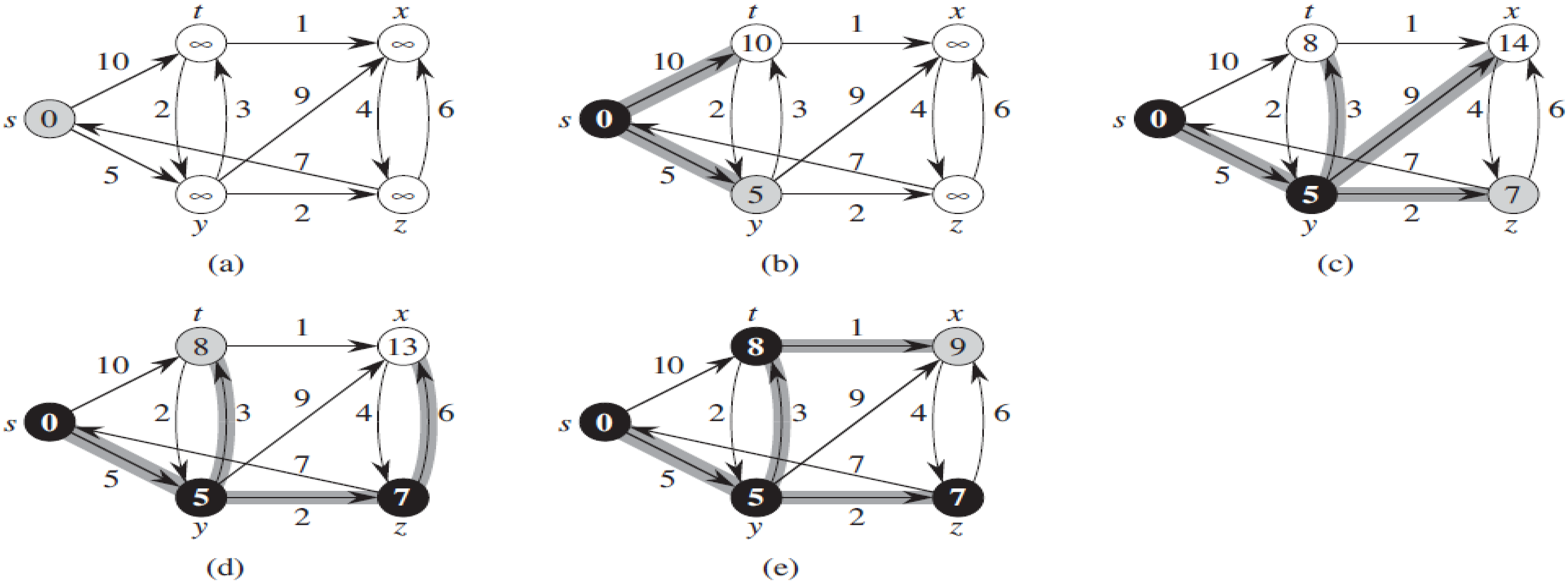
Profa. Ariane Machado Lima

Algoritmo de Dijkstra - Exemplo

```
DIJKSTRA(G, w, s)
INITIALIZE-SINGLE-SOURCE(G, s)
S = Ø
Q = G.V
while Q ≠ Ø
    u = EXTRACT-MIN(Q)
    S = S ∪ {u}
    for each vertex v ∈ G.Adj[u]
        RELAX(u, v, w)
```

```
RELAX(u, v, w)
if d[v] > d[u] + w(u, v)
    d[v] = d[u] + w(u, v)
    π[v] = u
```

Vértices brancos estão em Q, cinza é o que acaba de ser removido, pretos já tiveram todas as arestas que saem dele relaxadas

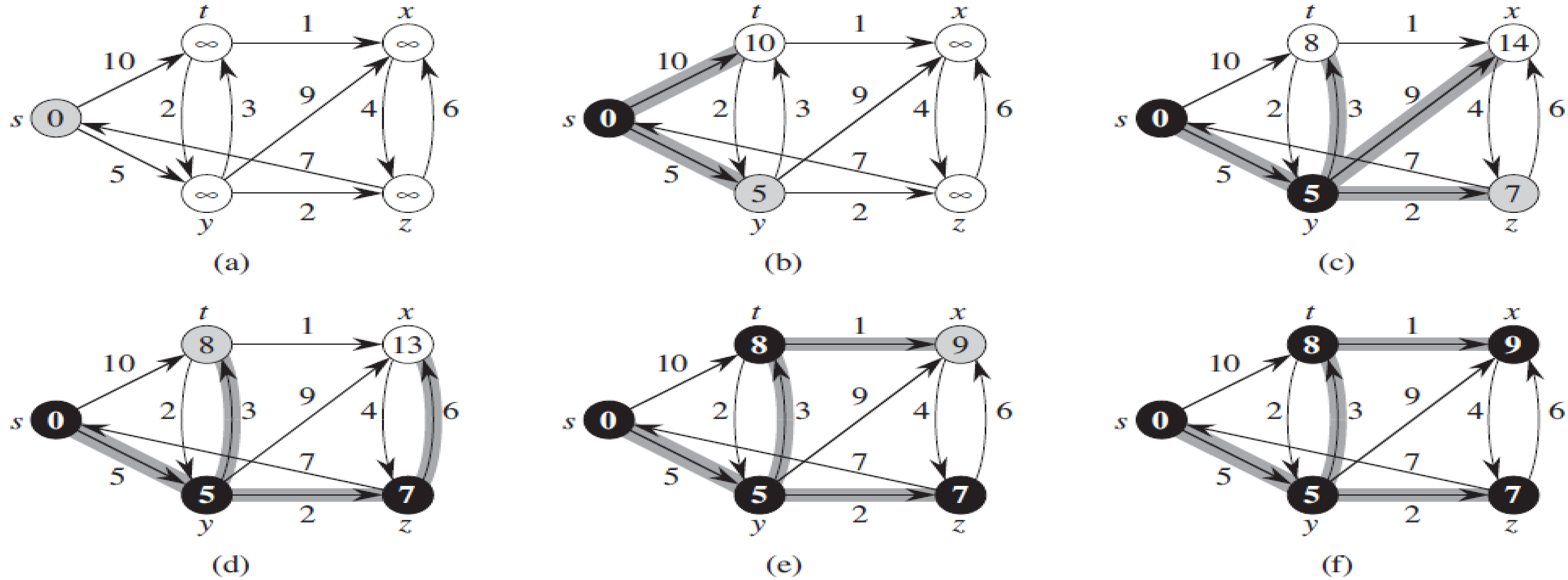


Algoritmo de Dijkstra - Exemplo

```
RELAX(u, v, w)
if d[v] > d[u] + w(u, v)
    d[v] = d[u] + w(u, v)
    π[v] = u
```

Vértices brancos estão em Q, cinza é o que acaba de ser removido, pretos já tiveram todas as arestas que saem dele relaxadas

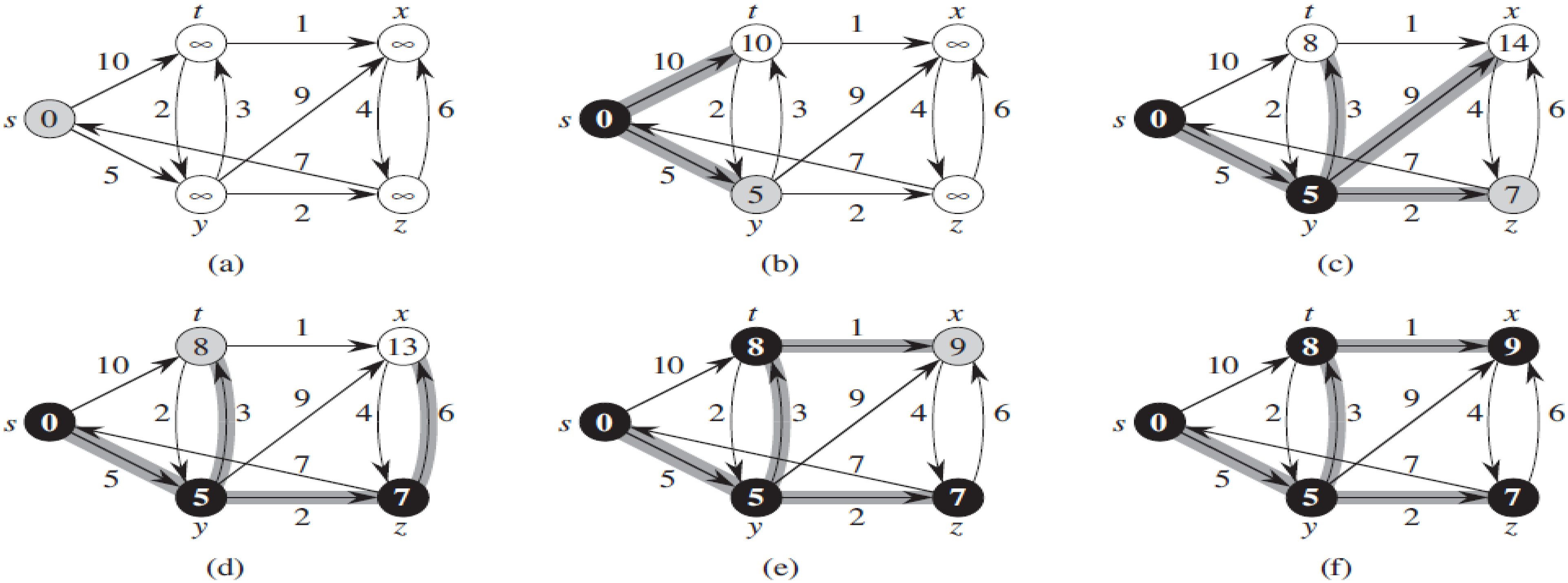
```
DIJKSTRA(G, w, s)
INITIALIZE-SINGLE-SOURCE(G, s)
S = Ø
Q = G.V
while Q ≠ Ø
    u = EXTRACT-MIN(Q)
    S = S ∪ {u}
    for each vertex v ∈ G.Adj[u]
        RELAX(u, v, w)
```



Algoritmo de Dijkstra - Exemplo

```
DIJKSTRA(G, w, s)
    INITIALIZE-SINGLE-SOURCE(G, s)
    S = Ø
    Q = G.V
    while Q ≠ Ø
        u = EXTRACT-MIN(Q)
        S = S ∪ {u}
        for each vertex v ∈ G.Adj[u]
            RELAX(u, v, w)
```

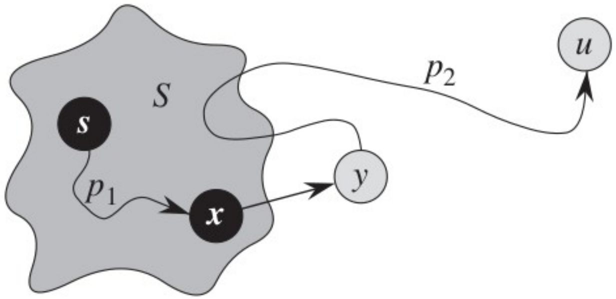
Note que, como as arestas possuem peso >= 0, vértices que ainda estão em Q não terão d menor do que as dos vértices que já foram processados, garantindo que só uma passada sobre todos os vértices é o suficiente para o cálculo correto das distâncias.



Vértices brancos estão em Q, cinza é o que acaba de ser removido, pretos já tiveram todas as arestas que saem dele relaxadas

Algoritmo de Dijkstra – Prova de corretude

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



Queremos provar que no final do algoritmo d[v] = δ(s, v) para todo vértice

Invariante no início de cada loop l. 4:
Para cada v ∈ S, d[v] = δ(s, v)

Para provar a corretude do algoritmo basta mostrar que d[u] = δ(s, u) no momento em que entra em S (l. 6):

- Quando S = ∅, isso é verdade (u = s)
- em cada loop (u ≠ s):

Vamos assumir por contradição que u seja o primeiro vértice a entrar em S que d[v] ≠ δ(s, v). Há pelo menos um caminho de s até u (senão d[u] = δ(s, u) = ∞), sendo um deles (p) o de custo mínimo. Esse caminho caminho p conecta s a u, ou seja, um vértice em S a um vértice em V-S. Seja y o primeiro vértice em V-S desse caminho, e x seu predecessor (em S). Ou seja, p = (s, <p1>, x, y, <p2>, u), sendo p1 e p2 os caminhos entre s e x, e y e u, respectivamente.

Como u foi o o primeiro vértice a entrar em S que d[v] ≠ δ(s, v), então d[x] = δ(s, x) quando ele entrou em S, e neste momento suas arestas foram relaxadas, logo d[y] = δ(s, y) também.

Como y aparece antes de u no caminho mínimo de s a u, d[y] = δ(s, y) ≤ δ(s, u) ≤ d[u]. Mas como u e y estavam em V-S, e u foi escolhido antes de y, então d[u] ≤ d[y]. Considerando as duas inequações em vermelho, então d[y] = δ(s, y) = δ(s, u) = d[u], o que contradiz nossa escolha de u.

- Quando o loop acaba, Q = ∅, S = V => **d[v] = δ(s, v) para todo vértice**

Algoritmo de Dijkstra - complexidade

```
DIJKSTRA(G, w, s)
    INITIALIZE-SINGLE-SOURCE(G, s)
    S = ∅
    Q = G.V
    while Q ≠ ∅
        u = EXTRACT-MIN(Q)
        S = S ∪ {u}
        for each vertex v ∈ G.Adj[u]
            RELAX(u, v, w)
```

```
RELAX(u, v, w)
    if d[v] > d[u] + w(u, v)
        d[v] = d[u] + w(u, v)
        π[v] = u
```

Algoritmo de Dijkstra - complexidade

```
DIJKSTRA(G, w, s)
INITIALIZE-SINGLE-SOURCE(G, s)
S = ∅
Q = G.V
while Q ≠ ∅
    u = EXTRACT-MIN(Q)
    S = S ∪ {u}
    for each vertex v ∈ G.Adj[u]
        RELAX(u, v, w)
```

```
RELAX(u, v, w)
if d[v] > d[u] + w(u, v)
    d[v] = d[u] + w(u, v)
    π[v] = u
```

Depende de como Q é implementada!!!

Algoritmo de Dijkstra - complexidade

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

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

Considerando Q como um heap binário:

L. 1: O(V)
L. 3: O(V)
Loop L.4 executado |V| vezes
L. 5: no total O(V lg V)
L. 7-8: no total A chamadas a RELAX, cada uma com um DECREASE-KEY implícito: O(A lgV)
Total: O((V+A) lg V) ou
O(A lgV) se todos os vértices forem alcançáveis a partir da origem

Algoritmo de Dijkstra - complexidade

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

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

Considerando Q como um heap binário:

L. 1: O(V)
L. 3: O(V)
Loop L.4 executado |V| vezes
L. 5: no total O(V lg V)
L. 7-8: no total A chamadas a RELAX,
cada uma com um DECREASE-KEY
implícito: O(A lgV)
Total: O((V+A) lg V) ou
O(A lgV) se todos os vértices forem
alcançáveis a partir da origem

Usando heaps Fibonacci: O(V lg V + A)

Algoritmos para construção de árvores de caminhos mais curtos (origem única)

- Caso quase geral (arestas podem possuir pesos negativos mas não ciclos negativos)
  - **Bellman-Ford:** $O(VA)$
- Caso em que todas as arestas possuem valores de peso não negativos (**mais eficiente**)
  - **Dijkstra:** $O(A\ lgV)$ usando heap binário ou $O(V\ lgV + A)$ usando heap Fibonacci

Referências

Ziviani: seção 7.9 (cap 7) – apenas a definição de caminhos mais curtos e o algoritmo de Dijkstra (este livro não apresenta o algoritmo de Bellman-Ford)

Cormen: cap 24

E COM ISSO FECHAMOS O CONTEÚDO DE GRAFOS !!!

Lembrando que nossa prova é 24/04!!!

Cai até a aula de hoje!

Pode ter questões de:

- Implementação:
  - você escolhe pseudocódigo ou C (ou “pseudo-C”)
  - Usando a interface de grafos ou não
  - Se precisar usar algum algoritmo estudado precisa implementá-lo, não basta chamar a função sem mostrá-la
- “Desenho” (simulações passo a passo dos algoritmos)
- Conceituais (ex: vantagens e desvantagens entre estruturas de dados e algoritmos distintos)
- Complexidades dos algoritmos

Sobre o EP1

Por favor, releiam todos os emails sobre o EP 1 que eu mandei. Em particular, se atentem que SOMENTE PODE EXISTIR UMA implementação de:

- busca/vista profundidade, busca/vista largura, componentes conexos, vértices de articulação

que devem estar no ep1.c !!! Utilizando as funções da interface!

Essas funções (buscas em prof/larg, comp. conexos, vertices de artic.) NÃO podem assumir a implementação por matriz ou lista.

Ou seja, em grafo_listaAdj.c e grafo_matriz.c SOMENTE poder conter as implentações para:

```
bool inicializaGrafo(Grafo* grafo, int nv);
int obtemNrVertices(Grafo* grafo);
int obtemNrArestas(Grafo* grafo);
bool verificaValidadeVertice(int v, Grafo *grafo);
void insereAresta(int v1, int v2, Peso peso, Grafo *grafo);
bool existeAresta(int v1, int v2, Grafo *grafo);
Peso obtemPesoAresta(int v1, int v2, Grafo *grafo);
bool removeArestaObtendoPeso(int v1, int v2, Peso* peso, Grafo *grafo);
bool removeAresta(int v1, int v2, Grafo *grafo);
bool listaAdjVazia(int v, Grafo* grafo);
Apontador primeiroListaAdj(int v, Grafo* grafo);
Apontador proxListaAdj(int v, Grafo* grafo, Apontador atual);
int obtemVerticeDestino(Apontador p, Grafo* grafo);
void imprimeGrafo(Grafo* grafo);
void liberaGrafo(Grafo* grafo);
```

Todo o resto deve estar em ep1.c utilizando-as.

Pensem que a struct Grafo (seja por matriz ou lista) é uma classe, e todos os seu campos são atributos private, que só podem ser acessados pelos métodos public da classe (ou seja, aqueles cujo protótipos colocamos no .h).