Projeto e Análise de Algoritmos

A. G. Silva

Baseado nos materiais de
Fernandes, Feofiloff, Pina, Roman – USP
Souza, Silva, Lee, Rezende, Miyazawa – Unicamp
Monteforte – UFABC

25 de maio de 2018

Conteúdo programático

- Introdução (4 horas/aula)
- Notação Assintótica e Crescimento de Funções (4 horas/aula)
- Recorrências (4 horas/aula)
- Divisão e Conquista (12 horas/aula)
- Grafos (4 horas/aula)
- Buscas (4 horas/aula)
- Algoritmos Gulosos (8 horas aula)
- Programação Dinâmica (8 horas/aula)
- NP-Completude e Reduções (6 horas/aula)
- Algoritmos Aproximados e Busca Heurística (6 horas/aula)

Cronograma

- **02mar** – Apresentação da disciplina. Introdução.
- **09mar** – *Prova de proficiência/dispensa.*
- **16mar** – Notação assintótica. Recorrências.
- **23mar** – *Dia não letivo.* Exercícios.
- **30mar** – *Dia não letivo.* Exercícios.
- **06abr** – Recorrências. Divisão e conquista.
- **13abr** – Divisão e conquista. Ordenação.
- **20abr** – Ordenação. Estatística de ordem.
- **27abr** – **Primeira avaliação.**
- **04mai** – Estatística de ordem. Grafos. Buscas.
- **11mai** – Buscas. Algoritmos gulosos.
- **18mai** – Algoritmos gulosos.
- **25mai** – Algoritmos gulosos. Programação dinâmica .
- **01jun** – *Dia não letivo.* Exercícios.
- **08jun** – *Semana Acadêmica.* Exercícios.
- **15jun** – Programação dinâmica. NP-Completude e reduções.
- **22jun** – Exercícios *(copa)*.
- **29jun** – **Segunda avaliação.**
- **06jul** – **Avaliação substitutiva** *(opcional)*.

Algoritmos gulosos: conceitos básicos

- São aqueles que, a cada decisão:
  - Sempre escolhem a alternativa que parece mais promissora naquele instante: critério guloso – decisão localmente ótima!
  - Nunca reconsideram essa decisão
    - Uma escolha que foi feita nunca é revista
    - Não há *backtracking*

Algoritmos gulosos – caminho mínimo

Problema do(s) Caminho(s) Mínimo(s)

Seja G um grafo orientado e suponha que para cada aresta (u, v) associamos um peso (custo, distância) $w(u, v)$.
Usaremos a notação (G, w).

- Problema do Caminho Mínimo entre Dois Vértices:
  Dados dois vértices s e t em (G, w), encontrar um caminho (de peso) mínimo de s a t.
- Aparentemente, este problema não é mais fácil do que o

  Problema dos Caminhos Mínimos com Mesma Origem:
  Dados (G, w) e $s \in V[G]$, encontrar para cada vértice v de G, um caminho mínimo de s a v.

Subsestrutura ótima de caminhos mínimos

Teorema. Seja (G, w) um grafo orientado e seja

$$P = (v_1, v_2, \ldots, v_k)$$

um caminho mínimo de v_1 a v_k.

Então para quaisquer i, j com $1 \leq i \leq j \leq k$

$$P_{ij} = (v_i, v_{i+1}, \ldots, v_j)$$

é um caminho mínimo de v_i a v_j.

Representação de caminhos mínimos

- Usamos uma idéia similar à usada em Busca em Largura nos algoritmos de caminhos mínimos que veremos.
- Para cada vértice $v \in V[G]$ associamos um predecessor $\pi[v]$.
- Ao final do algoritmo obtemos uma Árvore de Caminhos Mínimos com raiz s.
- Um caminho de s a v nesta árvore é um caminho mínimo de s a v em (G, w).

Estimativa de distâncias

- Para cada $v \in V[G]$ queremos determinar $dist(s, v)$, o peso de um caminho mínimo de s a v em (G, w) (distância de s a v.)
- Os algoritmos de caminhos mínimos associam a cada $v \in V[G]$ um valor $d[v]$ que é uma estimativa da distância $dist(s, v)$.

Inicialização

INITIALIZE-SINGLE-SOURCE(G, s)

1 **para cada** vértice $v \in V[G]$ **faça**
2 $\quad d[v] \leftarrow \infty$
3 $\quad \pi[v] \leftarrow$ NIL
4 $d[s] \leftarrow 0$

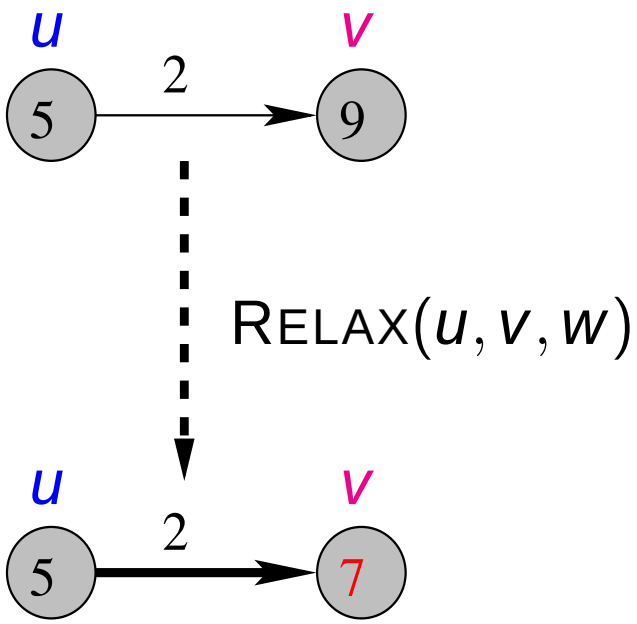
O valor $d[v]$ é uma estimativa superior para o peso de um caminho mínimo de s a v.

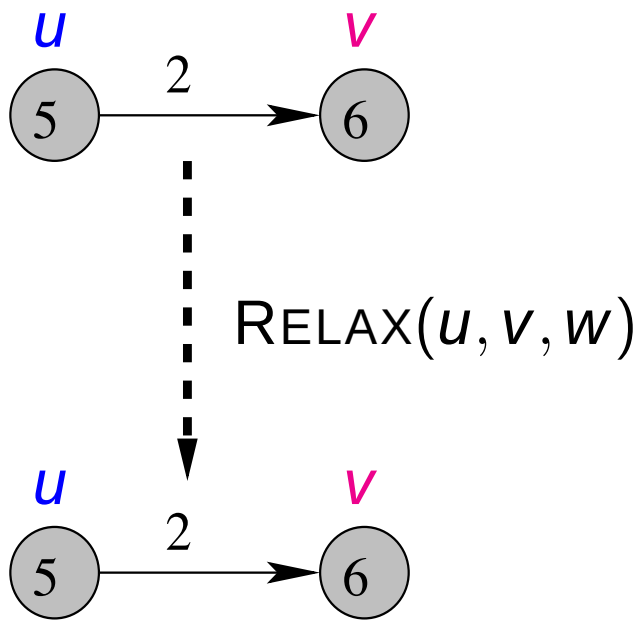
Ele indica que o algoritmo encontrou até aquele momento um caminho de s a v com peso $d[v]$.

O caminho pode ser recuperado por meio dos predecessores $\pi[\,]$.

Relaxação

Tenta melhorar a estimativa $d[v]$ examinando (u, v).



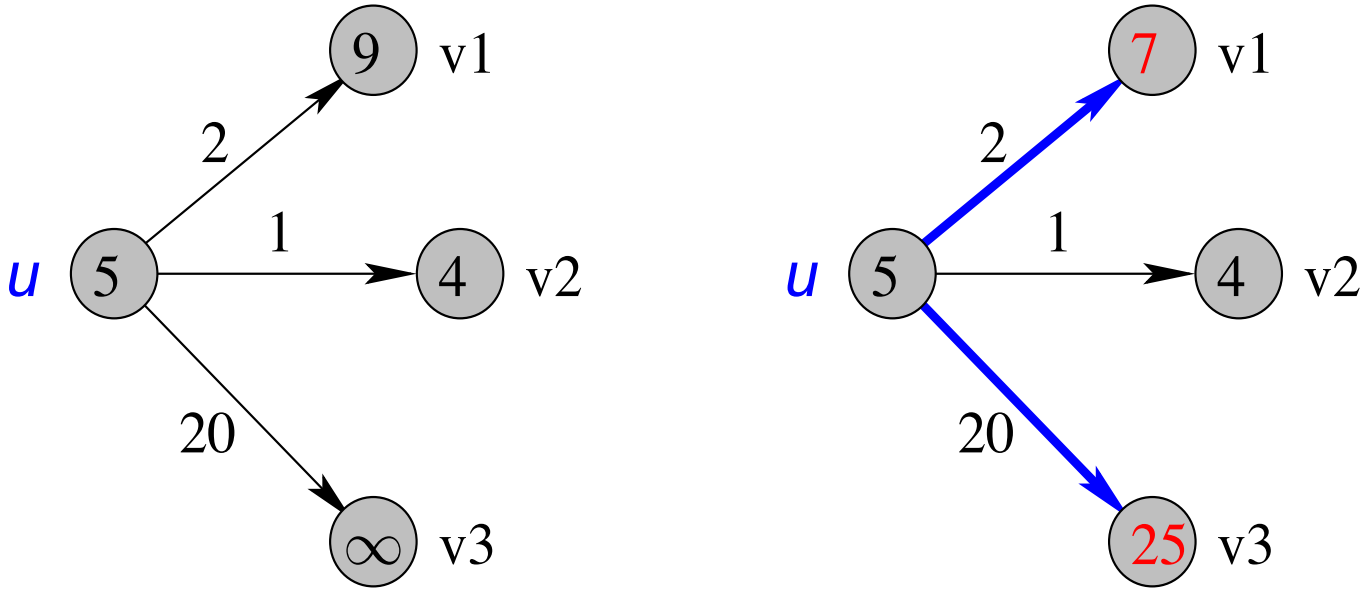


RELAX(u, v, w)

1 **se** $d[v] > d[u] + w(u, v)$
2 **então** $d[v] \leftarrow d[u] + w(u, v)$
3 $\pi[v] \leftarrow u$

Relaxação dos vizinhos

Em cada iteração o algoritmo seleciona um vértice u e para cada vizinho v de u aplica RELAX(u, v, w).



RELAX(u, v, w)

1 **se** $d[v] > d[u] + w(u, v)$ **faça**
2 $\quad d[v] \leftarrow d[u] + w(u, v)$
3 $\quad \pi[v] \leftarrow u$

Caminhos Mínimos

Veremos três algoritmos baseados em relaxação para tipos de instâncias diferentes de Problemas de Caminhos Mínimos.

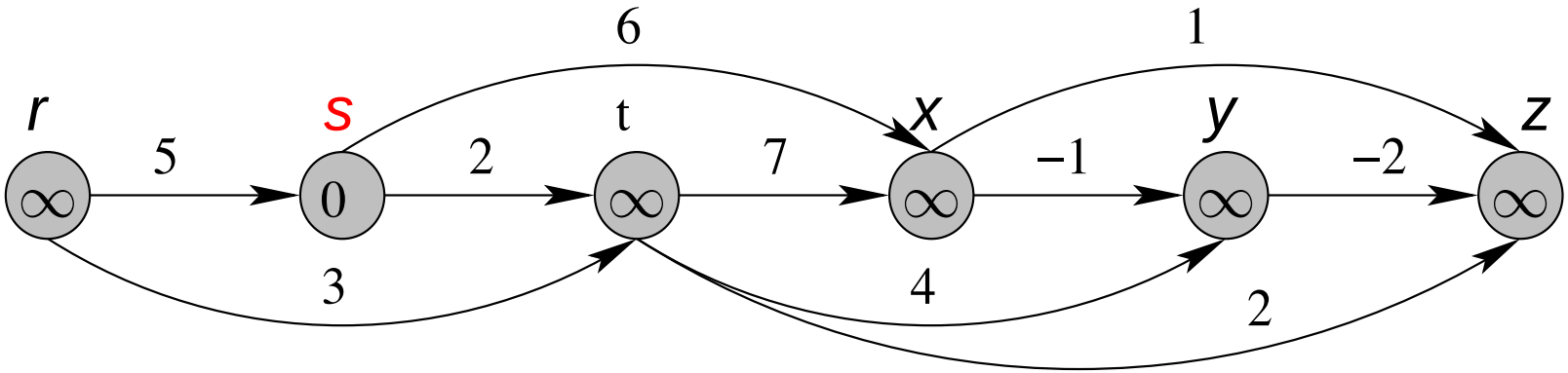
- G é acíclico: aplicação de ordenação topológica
- (G, w) não tem arestas de peso negativo: algoritmo de Dijkstra
- (G, w) tem arestas de peso negativo, mas não contém ciclos negativos: algoritmo de Bellman-Ford.

Caminhos mínimos em grafos acíclicos

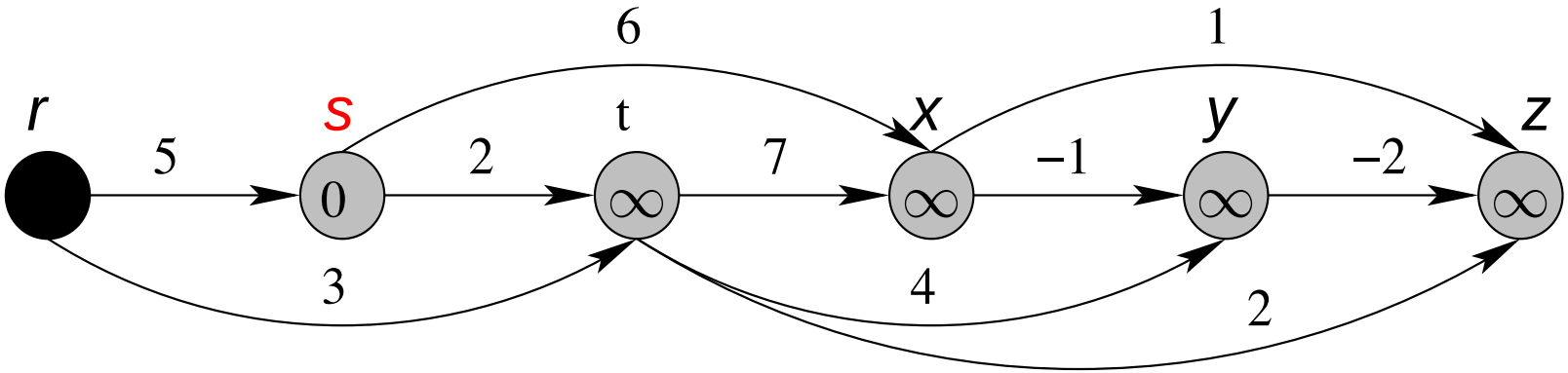
Entrada: grafo orientado acíclico $G = (V, E)$ com função peso w nas arestas e uma origem s.
Saída: vetor $d[v] = dist(s, v)$ para $v \in V$
e uma Árvore de Caminhos Mínimos definida por $\pi[\,]$.

DAG-SHORTEST-PATHS(G, w, s)
1 Ordene topologicamente os vértices de G
2 INITIALIZE-SINGLE-SOURCE(G, s)
3 **para cada** vértice u na ordem topológica **faça**
4 **para cada** $v \in \mathrm{Adj}[u]$ **faça**
5 RELAX(u, v, w)

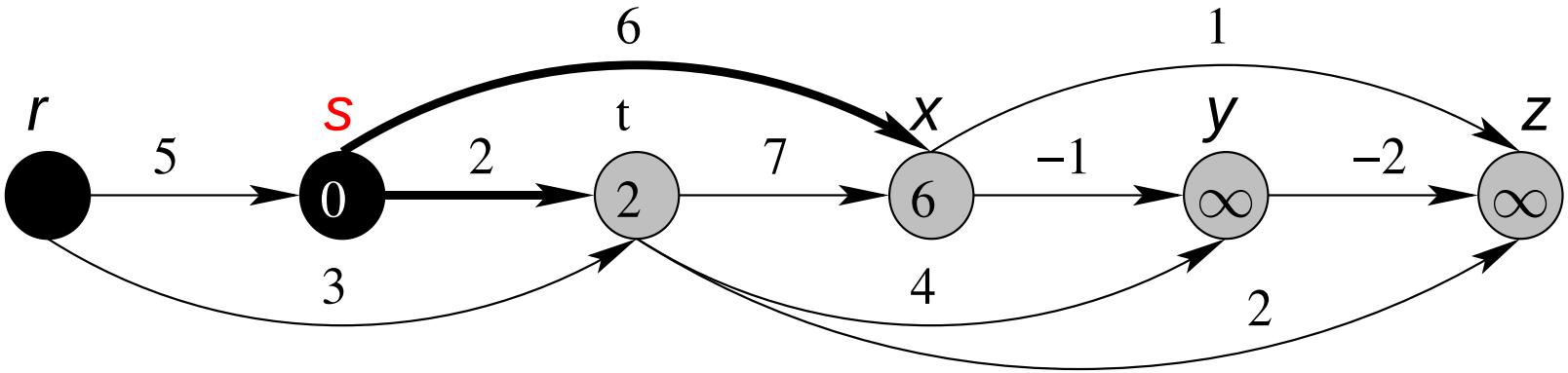
Exemplo



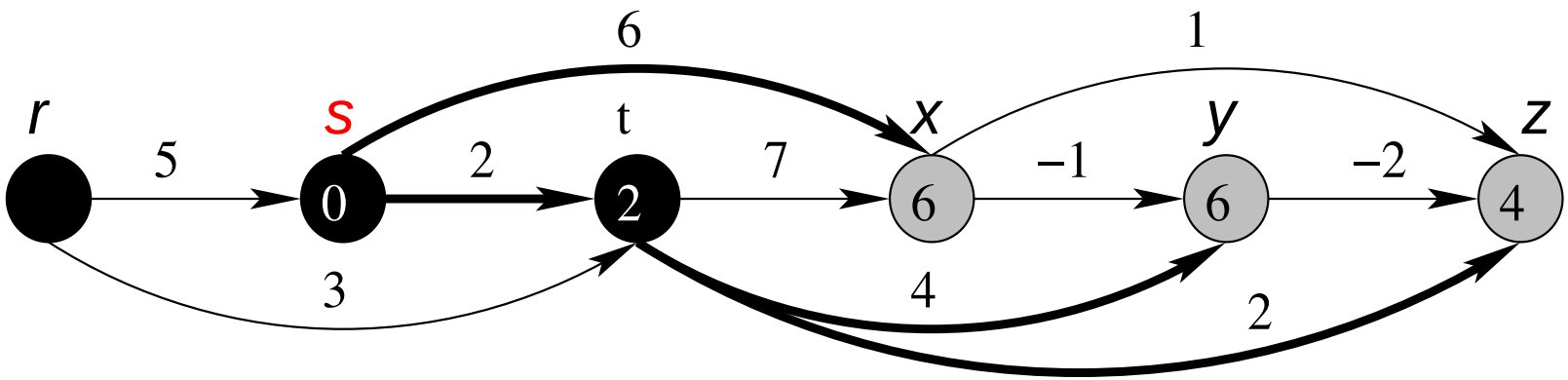
Exemplo



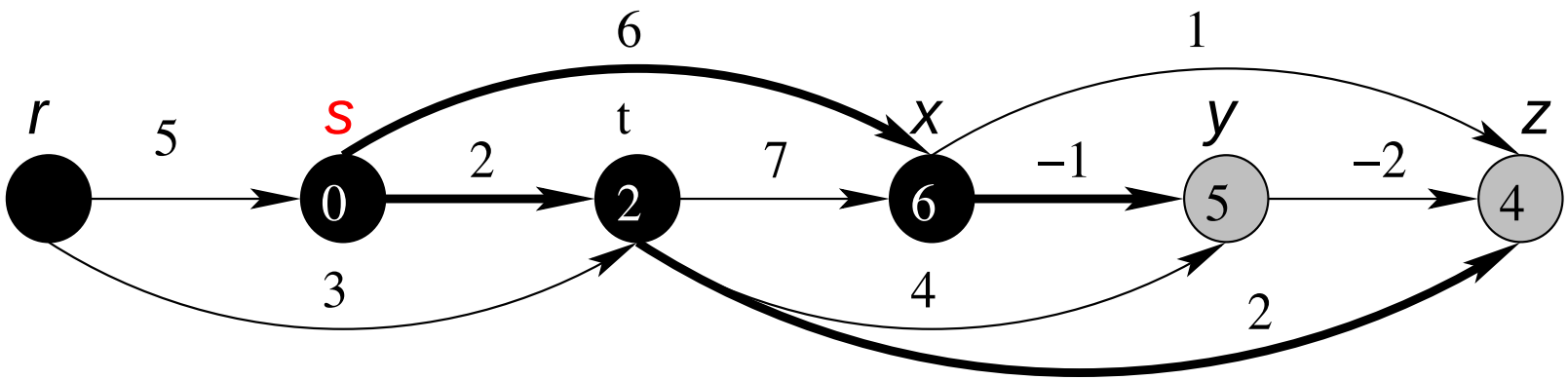
Exemplo



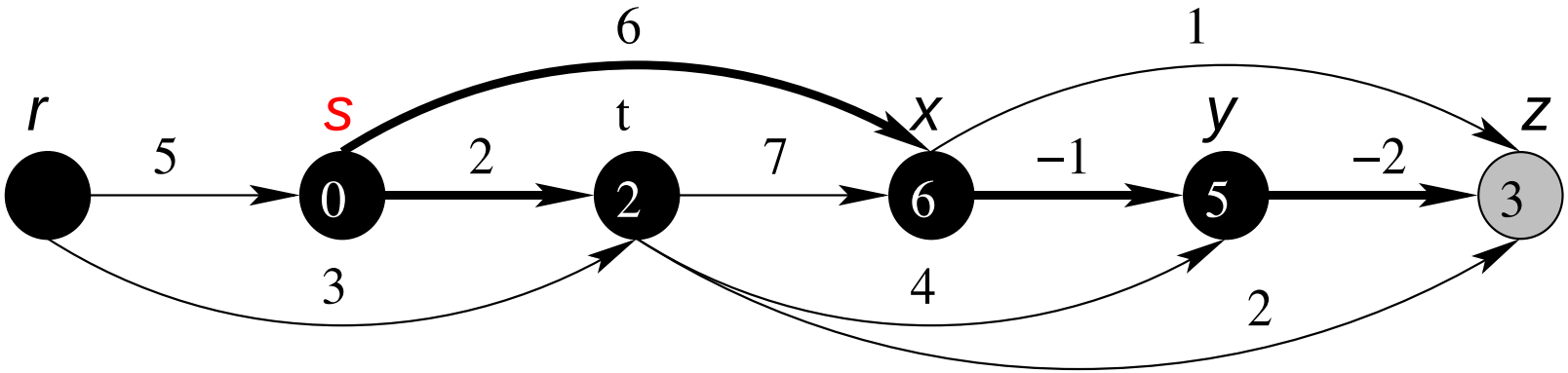
Exemplo



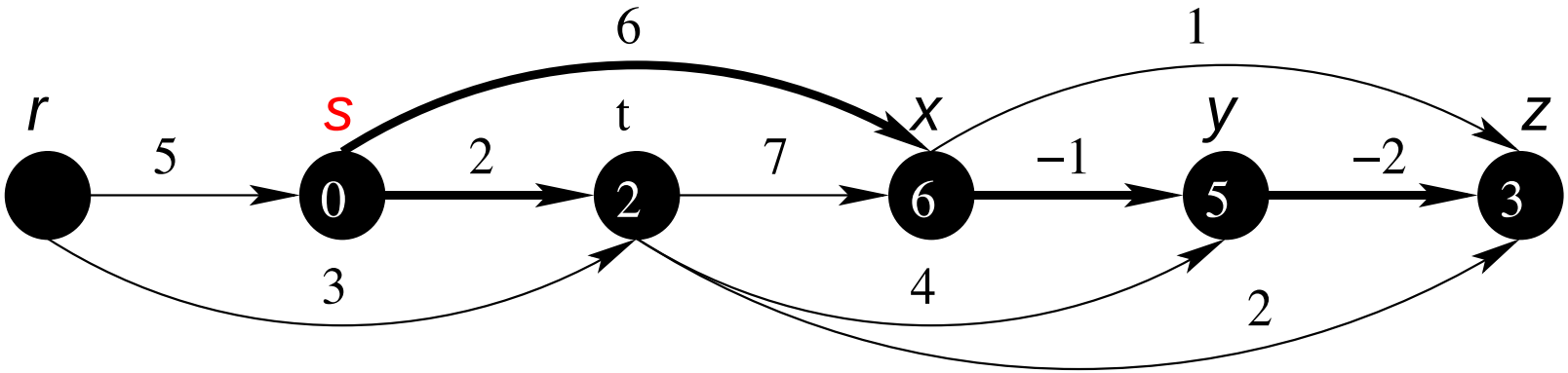
Exemplo



Exemplo



Exemplo



Complexidade

DAG-SHORTEST-PATHS(G, w, s)
1 Ordene topologicamente os vértices de G
2 INITIALIZE-SINGLE-SOURCE(G, s)
3 **para cada** vértice u na ordem topológica **faça**
4 **para cada** $v \in \mathrm{Adj}[u]$ **faça**
5 RELAX(u, v, w)

Linha(s)	Tempo total
1	$O(V + E)$
2	$O(V)$
3-5	$O(V + E)$

Complexidade de DAG-SHORTEST-PATHS: $O(V + E)$

Algoritmo de Dijkstra

O algoritmo de Dijkstra recebe um grafo orientado (G, w) (sem arestas de peso negativo) e um vértice s de G

e devolve

- para cada $v \in V[G]$, o peso de um caminho mínimo de s a v
- e uma Árvore de Caminhos Mínimos com raiz s.
  Um caminho de s a v nesta árvore é um caminho mínimo de s a v em (G, w).

Algoritmo de Dijkstra

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

O conjunto Q é implementado como uma fila de prioridade.

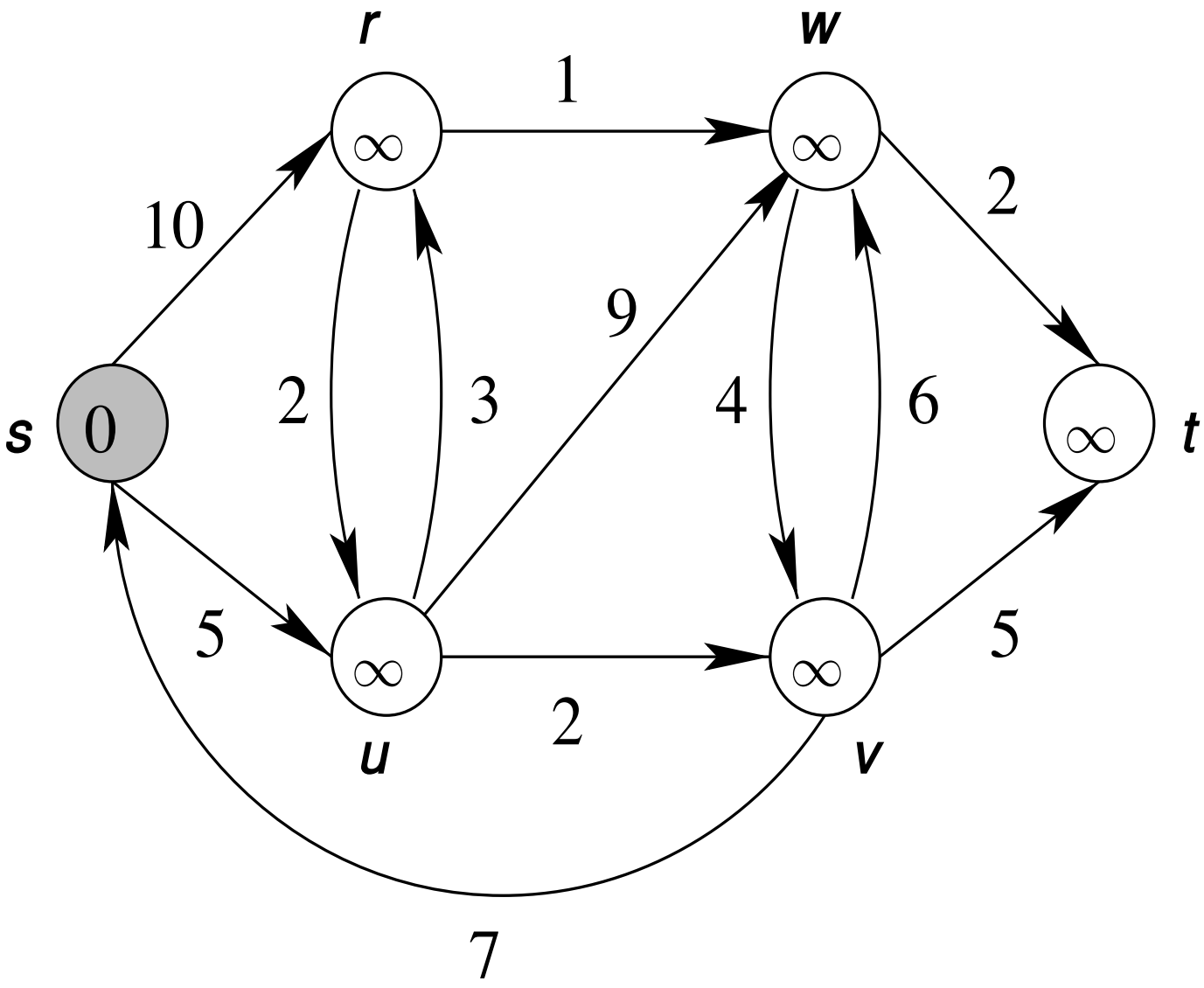
O conjunto S não é realmente necessário, mas simplifica a análise do algoritmo.

Intuição do algoritmo

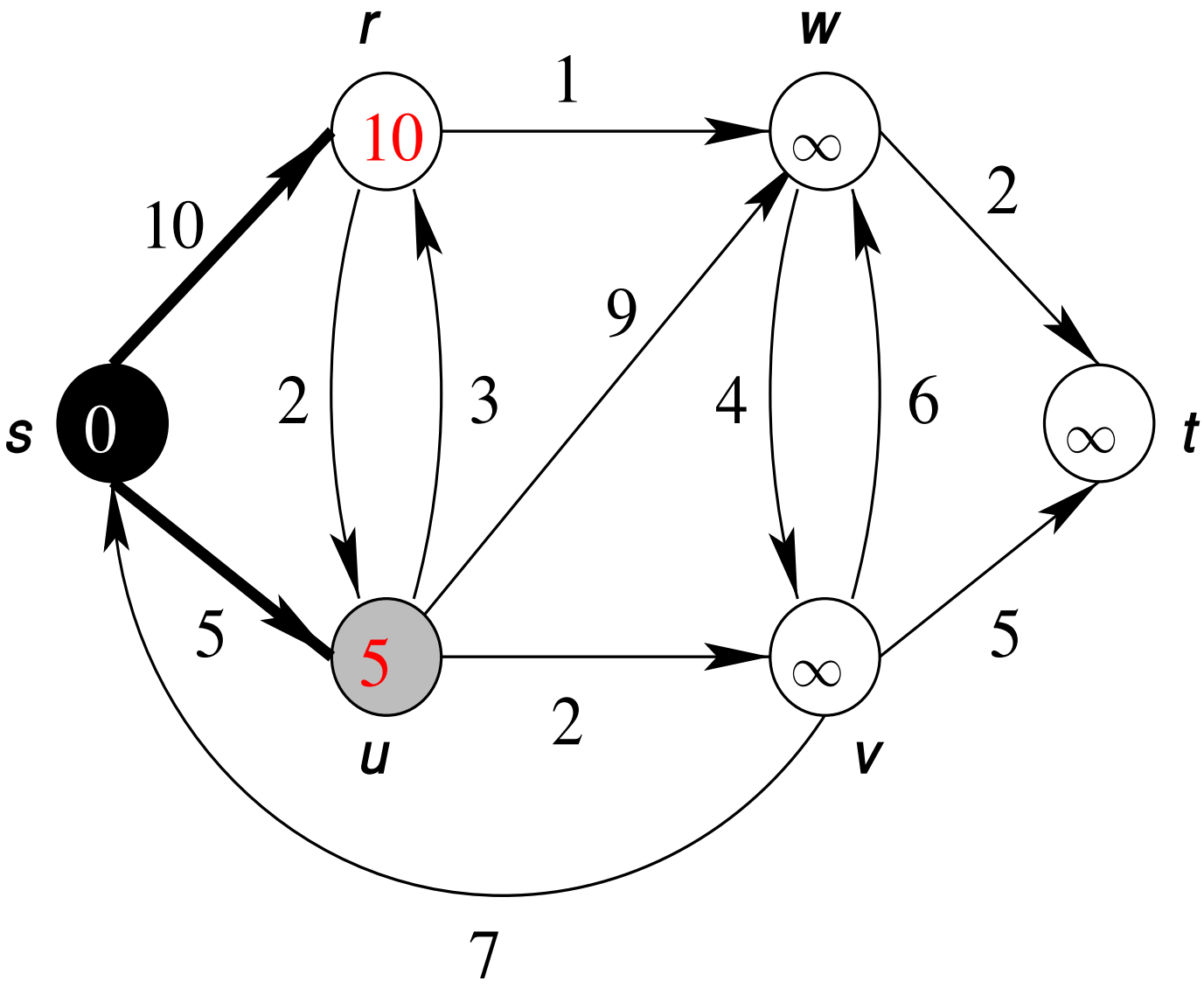
Em cada iteração, o algoritmo de DIJKSTRA

- escolhe um vértice u fora do conjunto S que esteja mais próximo a esse e acrescenta-o a S,
- atualiza as distâncias estimadas dos vizinhos de u e
- atualiza a Árvore dos Caminhos Mínimos.

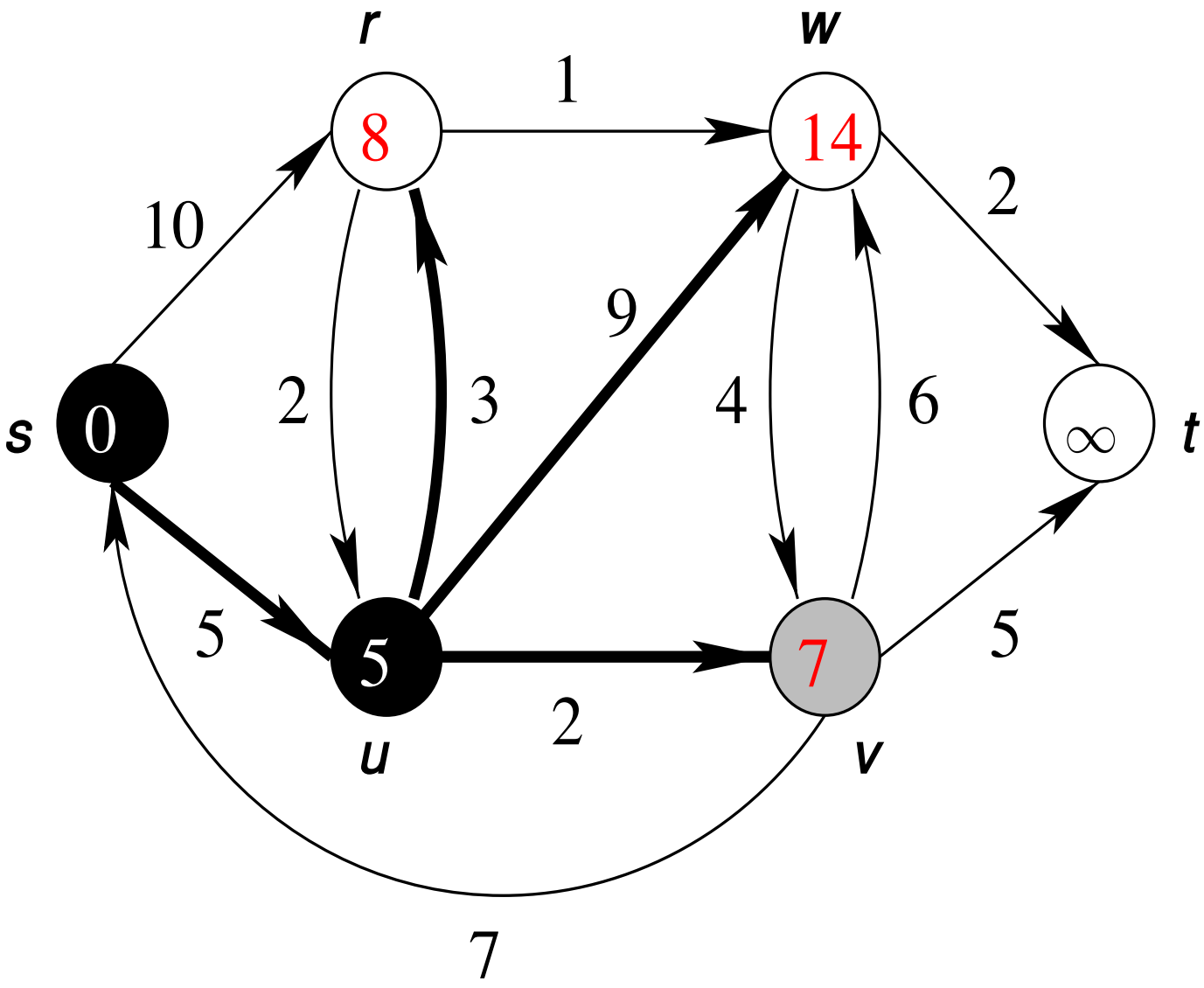
Exemplo (CLRS modificado)



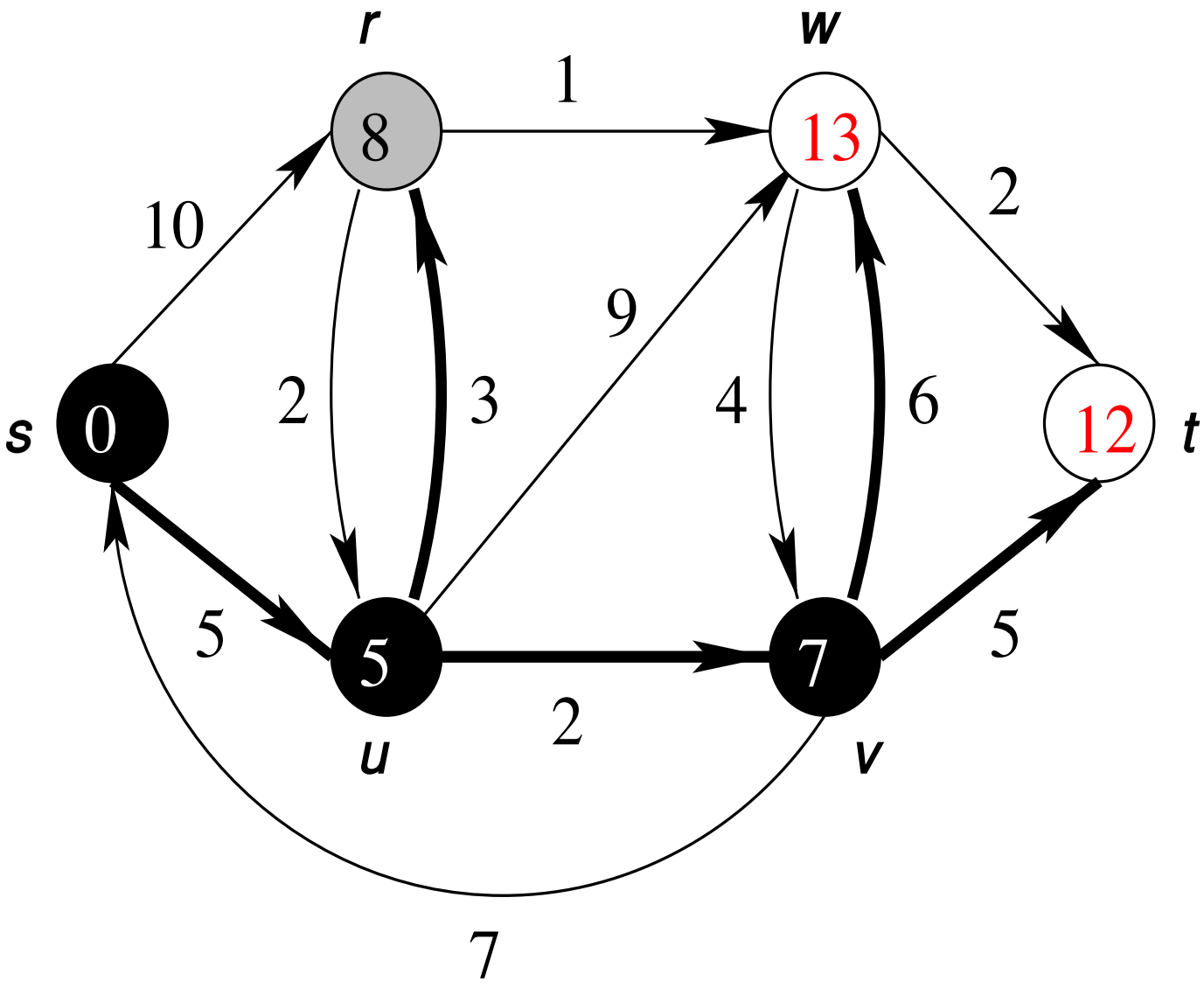
Exemplo (CLRS modificado)



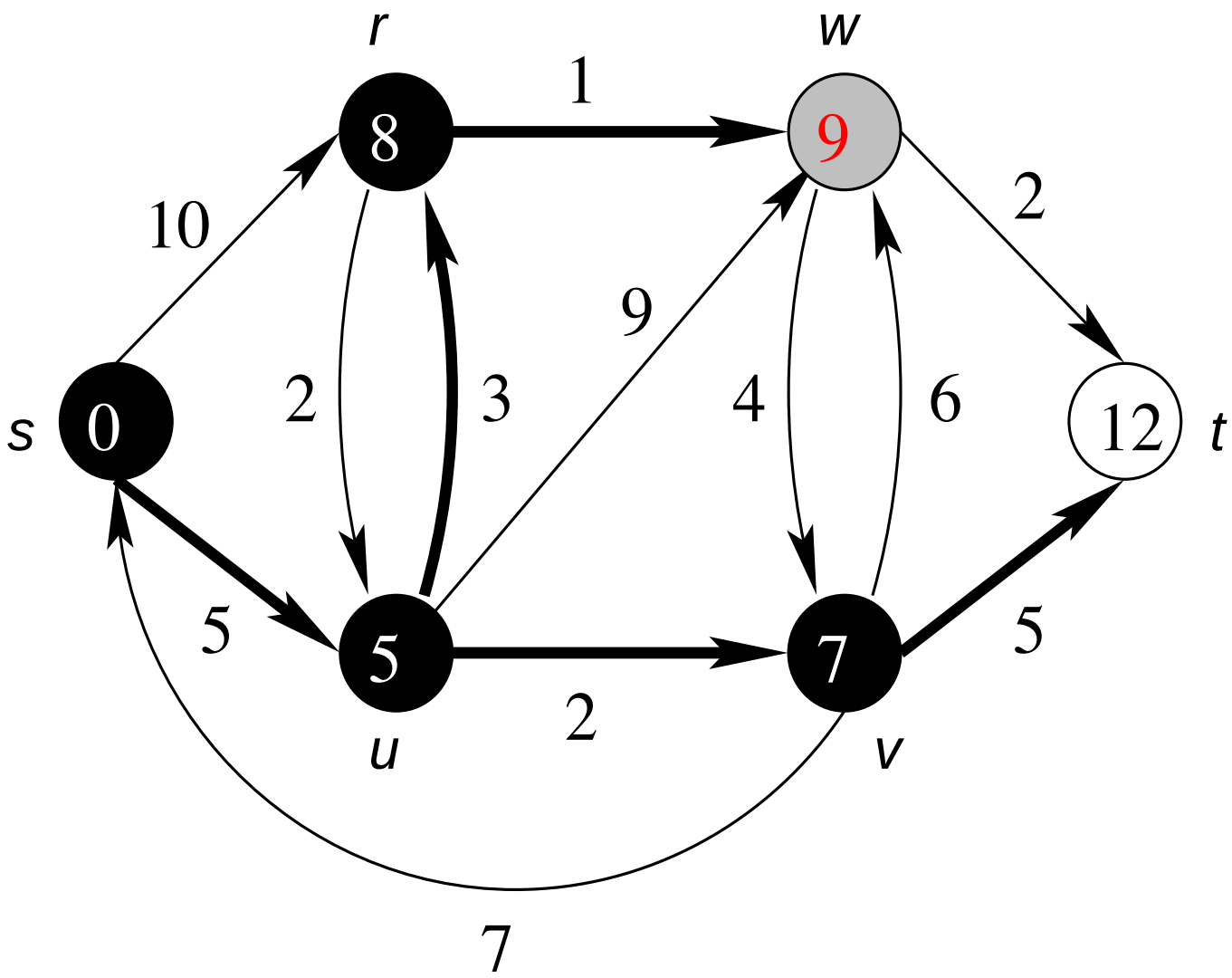
Exemplo (CLRS modificado)



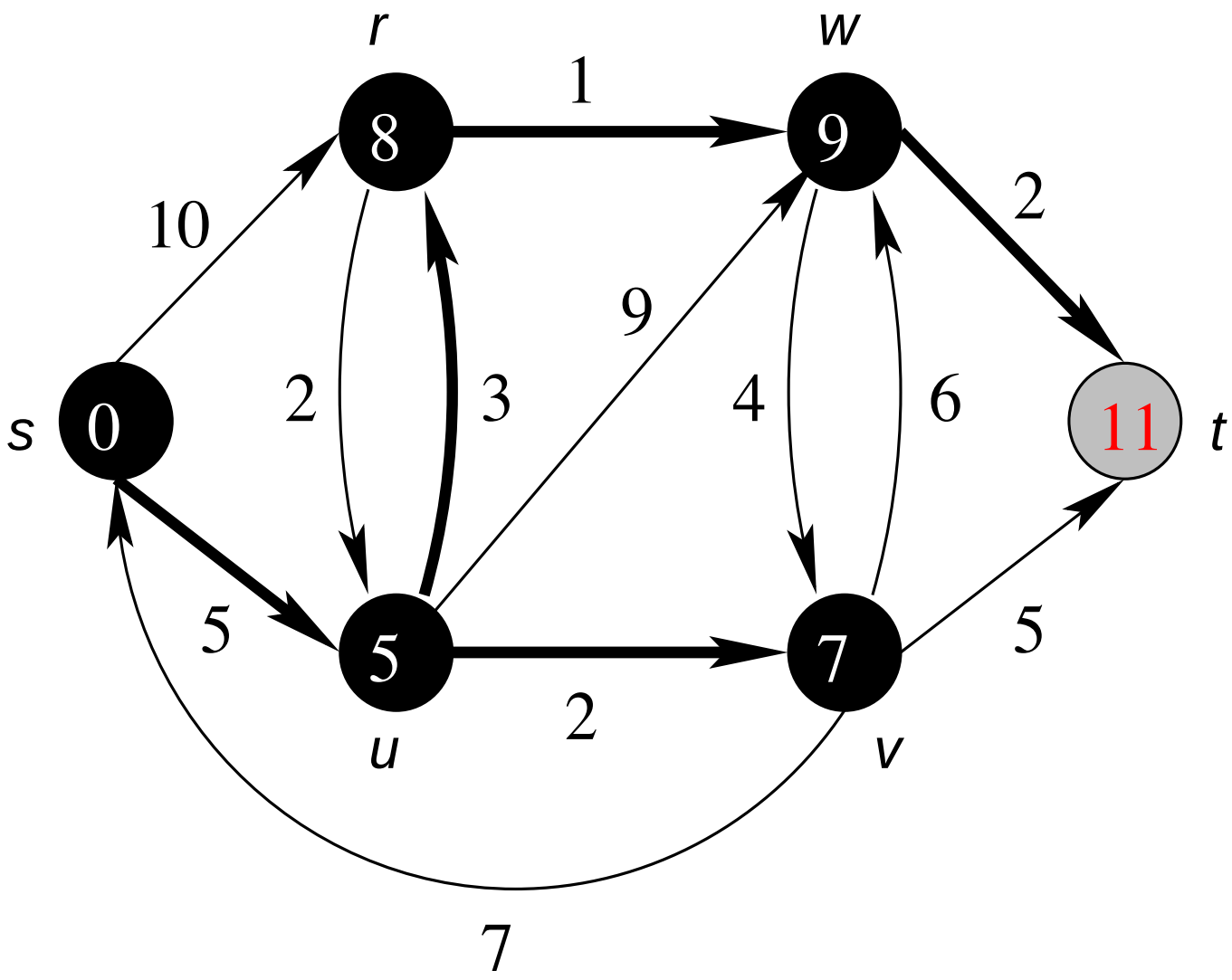
Exemplo (CLRS modificado)



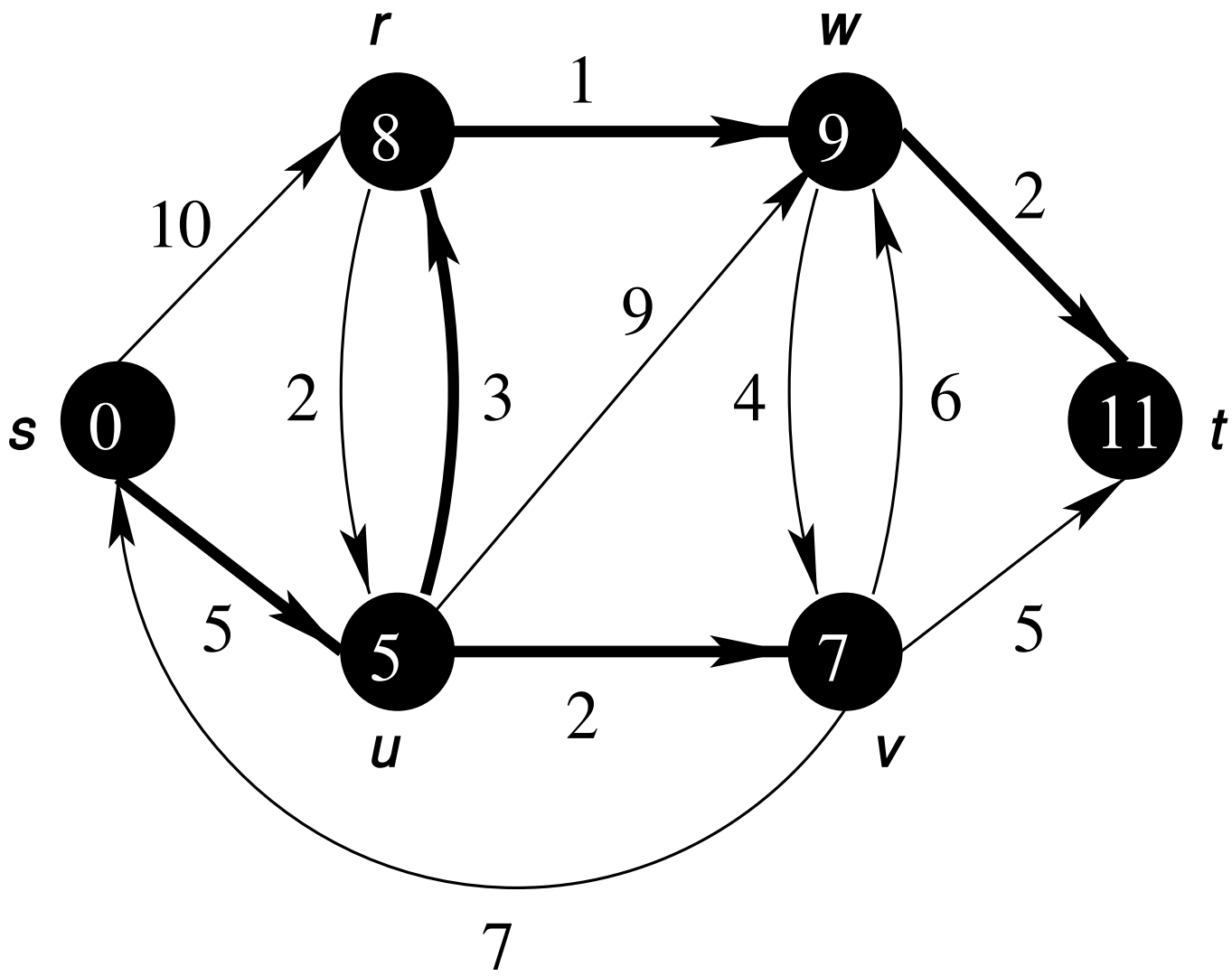
Exemplo (CLRS modificado)



Exemplo (CLRS modificado)



Exemplo (CLRS modificado)



Complexidade de tempo

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

Depende de como a fila de prioridade Q é implementada.

Complexidade de tempo

A fila de prioridade Q é mantida com as seguintes operações:

- INSERT (implícito na linha 3)
- EXTRACT-MIN (linha 5) e
- DECREASE-KEY (implícito em RELAX na linha 8).

São executados (no máximo) $|V|$ operações EXTRACT-MIN.

Cada vértice $u \in V[G]$ é inserido em S (no máximo) uma vez e cada aresta (u, v) com $v \in \mathrm{Adj}[u]$ é examinada (no máximo) uma vez nas linhas 7-8 durante todo o algoritmo. Assim, são executados no máximo $|E|$ operações DECREASE-KEY.

Complexidade de tempo

No total temos $|V|$ chamadas a EXTRACT-MIN e $|E|$ chamadas a DECREASE-KEY.

- Implementando Q como um vetor (coloque $d[v]$ na posição v do vetor), INSERT e DECREASE-KEY gastam tempo $\Theta(1)$ e EXTRACT-MIN gasta tempo $O(V)$, resultando em um total de $O(V^2 + E) = O(V^2)$.
- Implementando a fila de prioridade Q como um min-heap, INSERT,EXTRACT-MIN e DECREASE-KEY gastam tempo $O(\lg V)$, resultando em um total de $O(V + E) \lg V$.
- Usando heaps de Fibonacci (EXTRACT-MIN é $O(\lg V)$ e DECREASE-KEY é $O(1)$) a complexidade reduz para $O(V \lg V + E)$.

Arestas/ciclos de peso negativo

- O algoritmo de Dijkstra resolve o Problema dos Caminhos Mínimos quando (G, w) não possui arestas de peso negativo.
- Quando (G, w) possui arestas negativas, o algoritmo de Dijkstra não funciona (Exercício).
- Uma das dificuldades com arestas negativas é a possível existência de ciclos de peso negativo ou simplesmente ciclos negativos.

Ciclos negativos — uma dificuldade

- Se um ciclo negativo C é atingível a partir da fonte s, em princípio o problema não tem solução pois o “caminho” pode passar ao longo do ciclo infinitas vezes obtendo caminhos cada vez menores.
- Naturalmente, podemos impor a restrição de que os caminhos tem que ser simples, sem repetição de vértices. Entretanto, esta versão do problema é NP-difícil.
- Assim, vamos nos restringir ao Problema de Caminhos Mínimos sem ciclos negativos.

O algoritmo de Bellman-Ford

O algoritmo de Bellman-Ford recebe um grafo orientado (G, w) (possivelmente com arestas de peso negativo) e um vértice origem s de G

Ele devolve um valor booleano

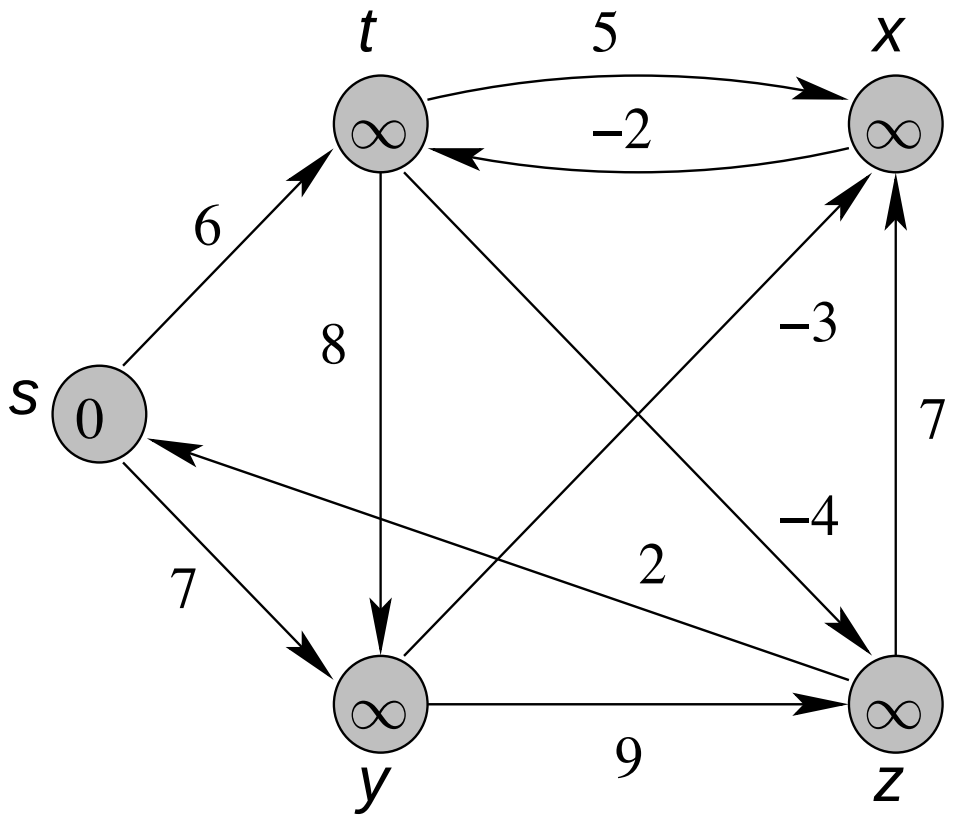
- FALSE se existe um ciclo negativo atingível a partir de s, ou
- TRUE e neste caso devolve também uma Árvore de Caminhos Mínimos com raiz s.

O algoritmo de Bellman-Ford

```
BELLMAN-FORD(G, w, s)
1   INITIALIZE-SINGLE-SOURCE(G, s)
2   para i ← 1 até |V[G]| − 1 faça
3       para cada aresta (u, v) ∈ E[G] faça
4           RELAX(u, v, w)
5   para cada aresta (u, v) ∈ E[G] faça
6       se d[v] > d[u] + w(u, v)
7           então devolva FALSE
8   devolva TRUE
```

Complexidade de tempo: $O(VE)$

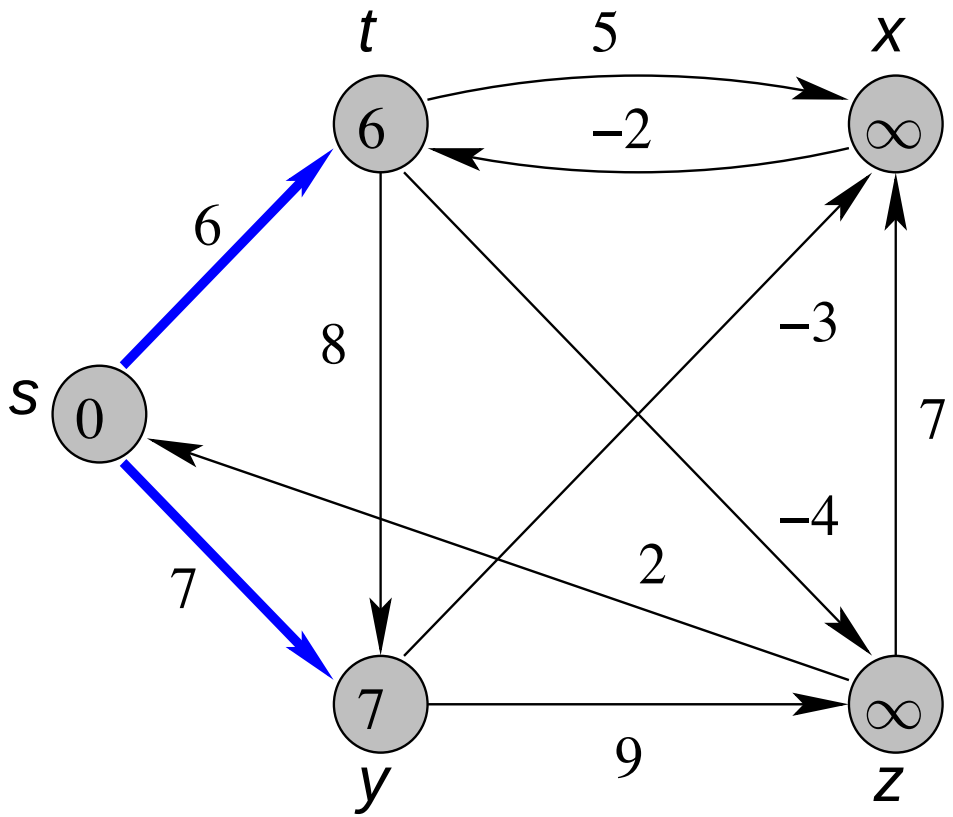
Exemplo (CLRS)



Ordem:

$(t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)$.

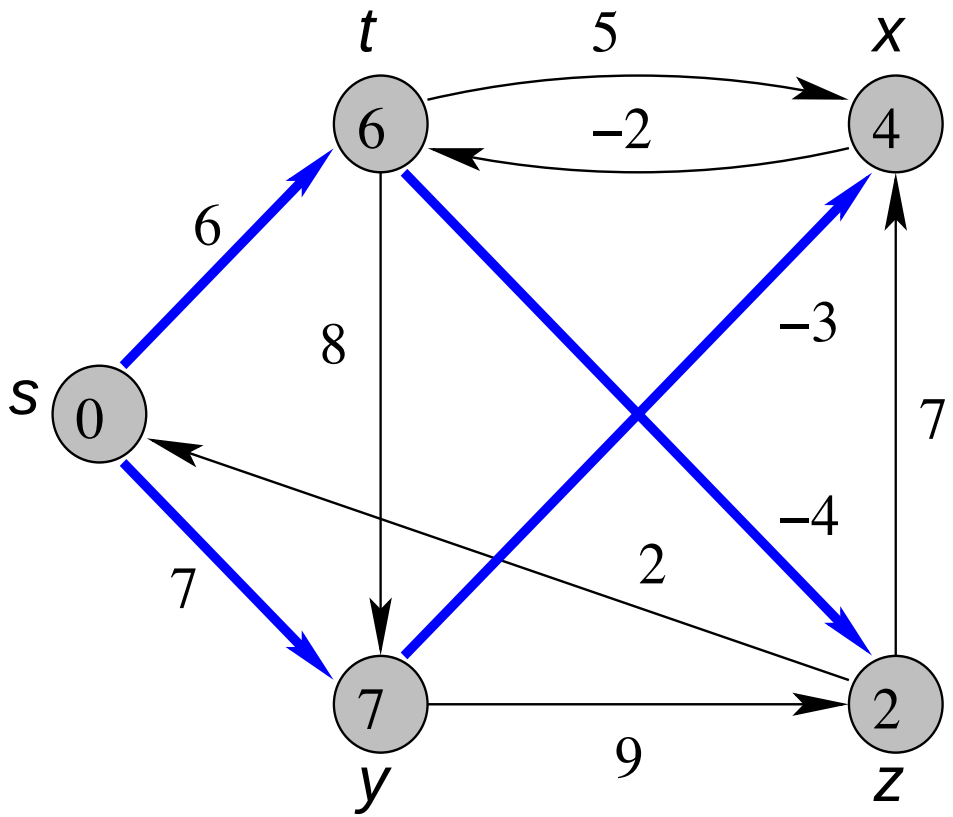
Exemplo (CLRS)



Ordem:

$(t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)$.

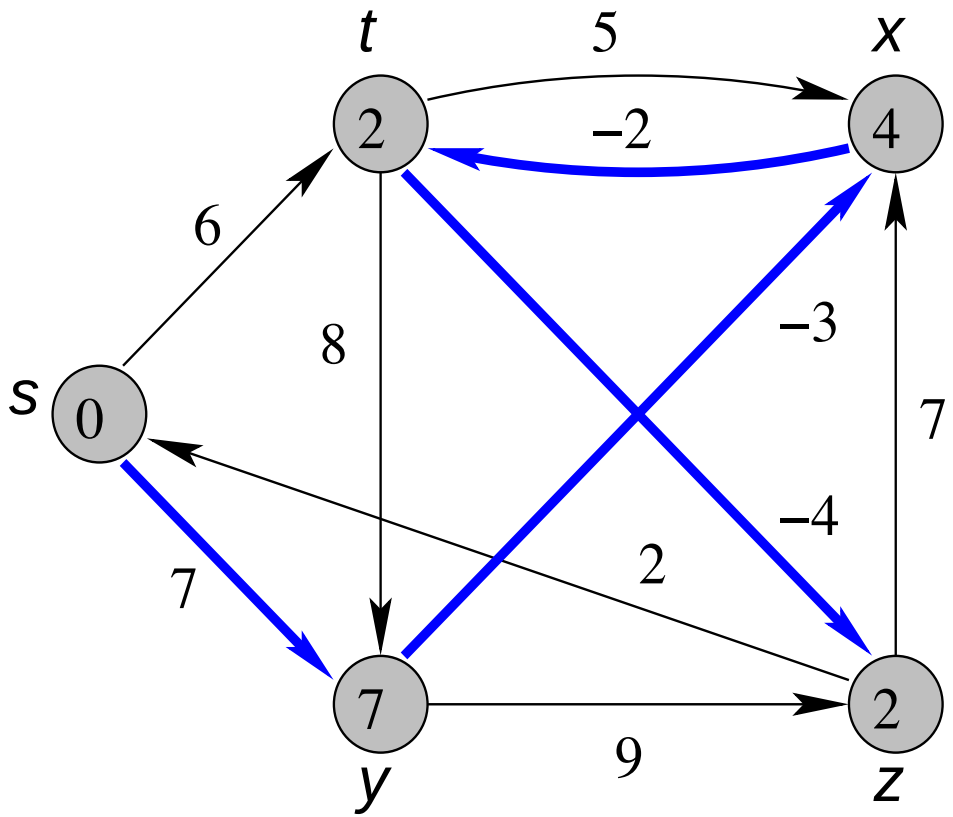
Exemplo (CLRS)



Ordem:

$(t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)$.

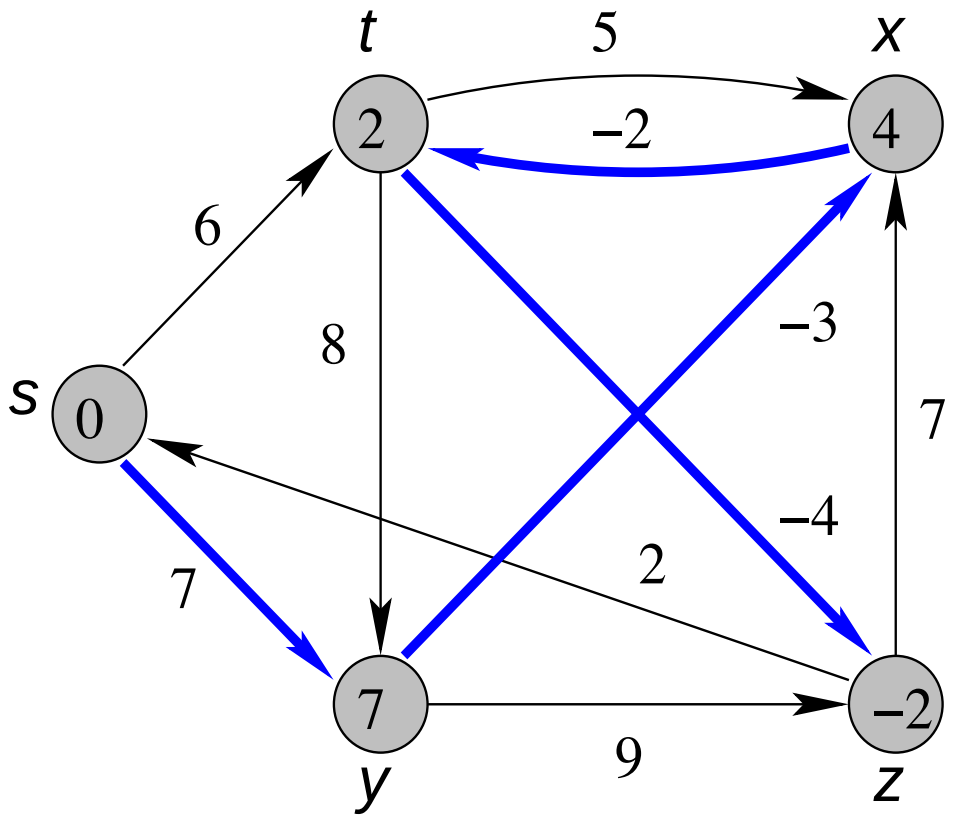
Exemplo (CLRS)



Ordem:

$(t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)$.

Exemplo (CLRS)



Ordem:

$(t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)$.

Caminhos mínimos entre todos os pares

O problema agora é dado um grafo (G, w) encontrar para todo para u, v de vértices um caminho mínimo de u a v.

Obviamente podemos executar $|V|$ vezes um algoritmo de Caminhos Mínimos com Mesma Origem.

- Se (G, w) não possui arestas negativas podemos usar o algoritmo de Dijkstra implementando a fila de prioridade como

  um vetor: $|V|.O(V^2 + E) = O(V^3 + VE)$ ou
  min-heap binário: $|V|.O(E \lg V) = O(VE \lg V)$ ou
  heap de Fibonacci: $|V|.O(V \lg V + E) = O(V^2 \lg V + VE)$.
- Se (G, w) possui arestas negativas podemos usar o algoritmo de Bellman-Ford: $|V|.O(VE) = O(V^2E)$.

O algoritmo de Floyd-Warshall

Veremos agora um método direto para resolver o problema que é assintoticamente melhor se G é denso.

O algoritmo de Floyd-Warshall baseia-se em programação dinâmica e resolve o problema em tempo $O(V^3)$.

O grafo (G, w) pode ter arestas negativas, mas suporemos que não contém ciclos negativos.

Vamos adotar a convenção de que (i, j) não é uma aresta de G então $w(i, j) = \infty$.

Estrutura de um caminho mínimo

Seja $P = (v_1, v_2, \ldots, v_k)$ um caminho (simples).

Um vértice intermediário de P é qualquer vértice de P distinto de v_1 e v_k, ou seja, em $\{v_2, \ldots, v_{k-1}\}$.

Para simplificar, suponha que $V = \{1, 2, \ldots, n\}$.

Estrutura de um caminho mínimo

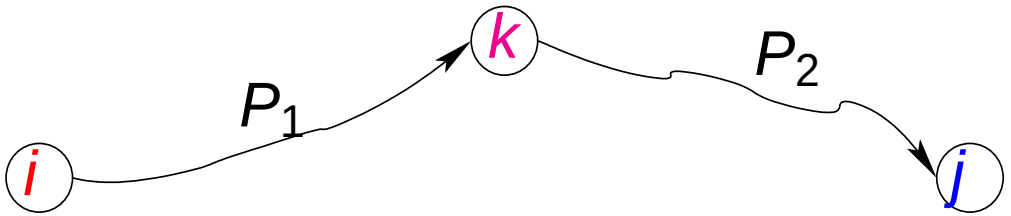
Sejam i e j dois vértices de G. Considere todos os caminhos cujos vértices intermediários pertencem a $\{1, \ldots, k\}$. Seja P um caminho mínimo entre todos eles.

O algoritmo de Floyd-Warshall explora a relação entre P e um caminho mínimo de i a j com vértices intermediários em $\{1, \ldots, k-1\}$.

Se k não é um vértice intermediário de P então P é um caminho mínimo de i a j com vértices intermediários em $\{1, \ldots, k-1\}$.

Estrutura de um caminho mínimo

- Se k é um vértice intermediário de P então P pode ser dividido em dois caminhos P_1 (com início em i e fim em k) e P_2 (com início em k e fim em j).



- P_1 é um caminho mínimo de i a k com vértices intermediários em $\{1, \ldots, k-1\}$
- P_2 é um caminho mínimo de k a j com vértices intermediários em $\{1, \ldots, k-1\}$.

Recorrência para caminhos mínimos

Seja $d_{ij}^{(k)}$ o peso de um caminho mínimo de i a j com vértices intermediários em $\{1, 2, \ldots, k\}$.

Quando $k = 0$ então $d_{ij}^{(0)} = w(i, j)$.

Temos a seguinte recorrência:

$$d_{ij}^{(k)} = \begin{cases} w(i,j) & \text{se } k = 0, \\ \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}) & \text{se } k \geq 1. \end{cases}$$

Assim, queremos calcular a matrix $D^{(n)} = (d_{ij}^{(n)})$ com $d_{ij}^{(n)} = dist(i, j)$.

Algoritmo de Floyd-Warshall

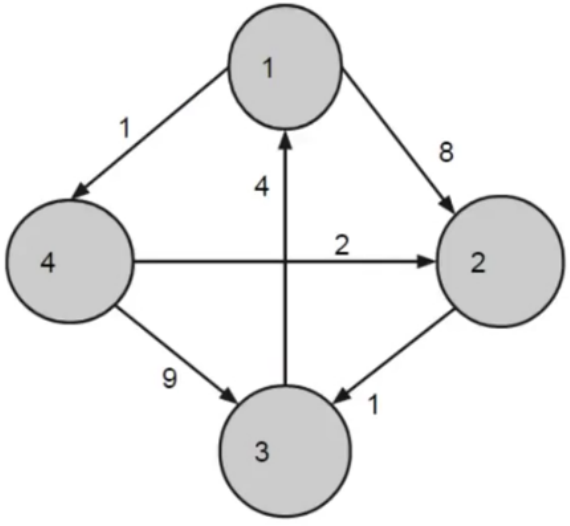
A entrada do algoritmo é a matriz $W = (w(i,j))$ com $n = |V|$ linhas e colunas.
A saída é a matriz $D^{(n)}$.

FLOYD-WARSHALL(W)

1 $D^{(0)} \leftarrow W$
2 **para** $k \leftarrow 1$ **até** n **faça**
3 **para** $i \leftarrow 1$ **até** n **faça**
4 **para** $j \leftarrow 1$ **até** n **faça**
5 $d_{ij}^{(k)} \leftarrow \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$
6 **devolva** $D^{(n)}$

Complexidade: $O(V^3)$

Algoritmo de Floyd-Warshall – exemplo

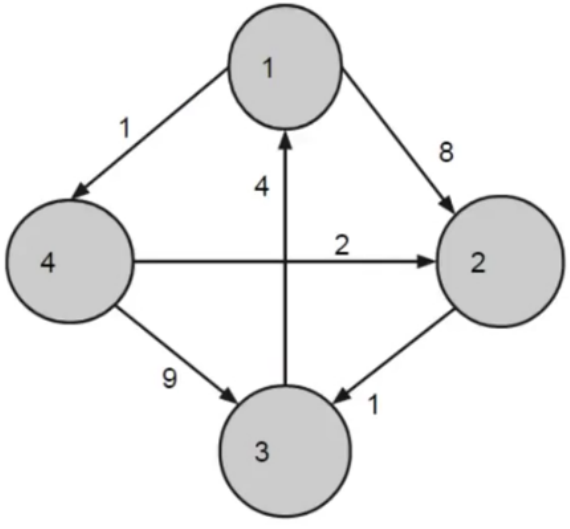


K = 0

	1	2	3	4
1	0	8	inf	1
2	inf	0	1	inf
3	4	inf	0	inf
4	inf	2	9	0

K = 1

	1	2	3	4
1	0	8	inf	1
2	inf	0	1	inf
3	4	inf	0	inf
4	inf	2	9	0



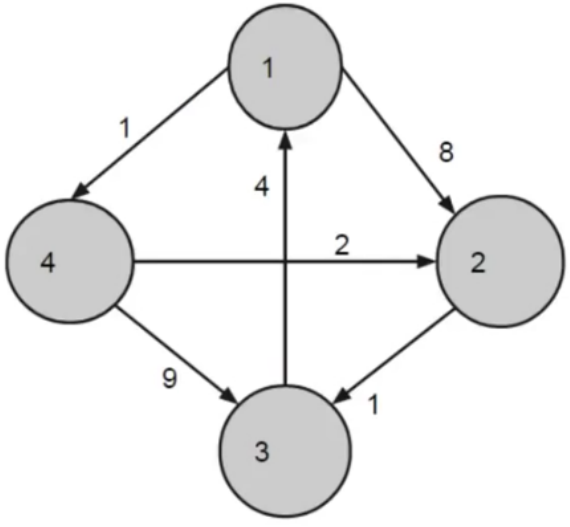
K = 2

	1	2	3	4
1	0	8	inf	1
2	inf	0	1	inf
3	4	12	0	5
4	inf	2	9	0

K = 3

	1	2	3	4
1	0	8	9	1
2	inf	0	1	inf
3	4	12	0	5
4	inf	2	3	0

Algoritmo de Floyd-Warshall – exemplo



K = 4

	1	2	3	4
1	0	8	9	1
2	5	0	1	6
3	4	12	0	5
4	7	2	3	0

	1	2	3	4
1	0	3	4	1
2	5	0	1	6
3	4	7	0	5
4	7	2	3	0

Como encontrar os caminhos?

O algoritmo precisa devolver também uma matriz $\Pi = (\pi_{ij})$ tal que $\pi_{ij} = \text{NIL}$ se $i = j$ ou se não existe caminho de i a j, e caso contrário, π_{ij} é o predecessor de de j em algum caminho mínimo a partir de i.

Podemos computar os predecessores ao mesmo tempo que o algoritmo calcula as matrizes $D^{(k)}$. Determinamos uma seqüência de matrizes $\Pi^{(0)}, \Pi^{(1)}, \ldots, \Pi^{(n)}$ e $\pi_{ij}^{(k)}$ é o predecessor de j em um caminho mínimo a partir de i com vértices intermediários em $\{1, 2, \ldots, k\}$.

Quando $k = 0$ temos

$$\pi_{ij}^{(0)} = \begin{cases} \text{NIL} & \text{se } i = j \text{ ou } w(i,j) = \infty, \\ i & \text{se } i \neq j \text{ e } w(i,j) < \infty. \end{cases}$$

Como encontrar os caminhos?

Para $k \geq 1$ procedemos da seguinte forma. Considere um caminho mínimo P de i a j.

Se k não aparece em P então tomamos como predecessor de j o predecessor de j em um caminho mínimo de i a j com vértices intermediários em $\{1, 2, \ldots, k-1\}$.

Caso contrário, tomamos como predecessor de j o predecessor de j em um caminho mínimo de k a j com vértices intermediários em $\{1, 2, \ldots, k-1\}$.
Formalmente,

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & \text{se } d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)}, \\ \pi_{kj}^{(k-1)} & \text{se } d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)}. \end{cases}$$

Exercício. Incorpore esta parte no algoritmo!

Passos do projeto de algoritmos gulosos: resumo

1. Formule o problema como um **problema de otimização** no qual uma escolha é feita, restando-nos então resolver um único subproblema a resolver.
2. Provar que existe sempre uma solução ótima do problema que atende à **escolha gulosa**, ou seja, a escolha feita pelo algoritmo guloso é segura.
3. Demonstrar que, uma vez feita a escolha gulosa, o que resta a resolver é um subproblema tal que se combinarmos a resposta ótima deste subproblema com o(s) elemento(s) da escolha gulosa, chega-se à solução ótima do problema original.
   Esta é a parte que requer mais engenhosidade!
   Normalmente a prova começa com uma solução ótima *genérica* e a modificamos até que ela inclua o(s) elemento(s) identificados pela escolha gulosa.

Programação dinâmica

Programação Dinâmica: Conceitos Básicos

- Tipicamente o paradigma de programação dinâmica aplica-se a problemas de **otimização**.
- Podemos utilizar programação dinâmica em problemas onde há:
  - **Subestrutura Ótima:** As soluções ótimas do problema incluem soluções ótimas de subproblemas.
  - **Sobreposição de Subproblemas:** O cálculo da solução através de recursão implica no recálculo de subproblemas.

Programação Dinâmica: Conceitos Básicos (Cont.)

- A técnica de programação dinâmica visa evitar o recálculo desnecessário das soluções dos subproblemas.
- Para isso, soluções de subproblemas são armazenadas em tabelas.
- Logo, para que o algoritmo de programação dinâmica seja eficiente, é preciso que o número total de subproblemas que devem ser resolvidos seja pequeno (polinomial no tamanho da entrada).

Programação dinâmica – números de Fibonacci

Números de Fibonacci

$$F_0 = 0 \qquad F_1 = 1 \qquad F_n = F_{n-1} + F_{n-2}$$

n	0	1	2	3	4	5	6	7	8	9
F_n	0	1	1	2	3	5	8	13	21	34

Algoritmo recursivo para F_n:

```
FIBO-REC (n)
1  se n ≤ 1
2      então devolva n
3      senão a ← FIBO-REC (n − 1)
4            b ← FIBO-REC (n − 2)
5            devolva a + b
```

Consumo de tempo

```
FIBO-REC (n)
1  se n ≤ 1
2      então devolva n
3      senão a ← FIBO-REC (n − 1)
4            b ← FIBO-REC (n − 2)
5            devolva a + b
```

Tempo em segundos:

n	16	32	40	41	42	43	44	45	47
tempo	0.002	0.06	2.91	4.71	7.62	12.37	19.94	32.37	84.50

$F_{47} = 2971215073$

Consumo de tempo

```
FIBO-REC (n)
1  se n ≤ 1
2      então devolva n
3      senão a ← FIBO-REC (n − 1)
4            b ← FIBO-REC (n − 2)
5            devolva a+b
```

$T(n) :=$ número de somas feitas por FIBO-REC (n)

linha	número de somas
1-2	$= 0$
3	$= T(n-1)$
4	$= T(n-2)$
5	$= 1$
$T(n)$	$= T(n-1) + T(n-2) + 1$

Recorrência

$$T(0) = 0$$

$$T(1) = 0$$

$$T(n) = T(n-1) + T(n-2) + 1 \text{ para } n = 2, 3, \ldots$$

A que classe Ω pertence $T(n)$?
A que classe O pertence $T(n)$?

Solução: $T(n) > (3/2)^n$ para $n \geq 6$.

n	0	1	2	3	4	5	6	7	8	9
T_n	0	0	1	2	4	7	12	20	33	54
$(3/2)^n$	1	1.5	2.25	3.38	5.06	7.59	11.39	17.09	25.63	38.4

Recorrência

Prova: $T(6) = 12 > 11.40 > (3/2)^6$ e $T(7) = 20 > 18 > (3/2)^7$.
Se $n \geq 8$, então

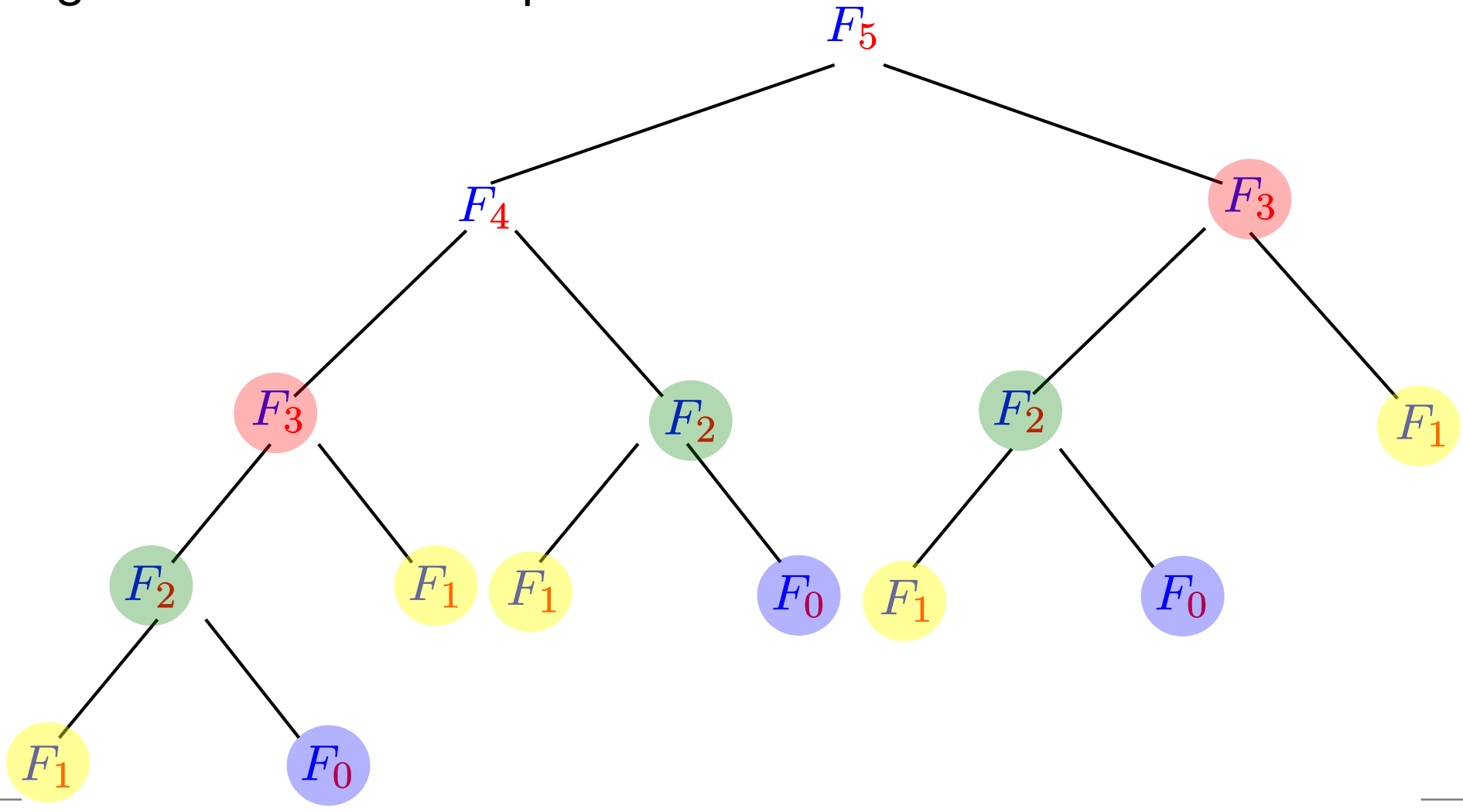
$$\begin{aligned} T(n) &= T(n-1) + T(n-2) + 1 \\ &\overset{\text{hi}}{>} (3/2)^{n-1} + (3/2)^{n-2} + 1 \\ &= (3/2+1)\,(3/2)^{n-2} + 1 \\ &> (5/2)\,(3/2)^{n-2} \\ &> (9/4)\,(3/2)^{n-2} \\ &= (3/2)^2 (3/2)^{n-2} \\ &= (3/2)^n\,. \end{aligned}$$

Logo, $T(n)$ é $\Omega((3/2)^n)$. Verifique que $T(n)$ é $\mathrm{O}(2^n)$.

Consumo de tempo

Consumo de tempo é exponencial.

Algoritmo resolve subproblemas muitas vezes.



Resolve subproblemas muitas vezes

```
FIBO-REC(5)
  FIBO-REC(4)
    FIBO-REC(3)
      FIBO-REC(2)
        FIBO-REC(1)
        FIBO-REC(0)
      FIBO-REC(1)
    FIBO-REC(2)
      FIBO-REC(1)
      FIBO-REC(0)
  FIBO-REC(3)
    FIBO-REC(2)
      FIBO-REC(1)
      FIBO-REC(0)
    FIBO-REC(1)
```

FIBO-REC(5) = 5

Resolve subproblemas muitas vezes

```
FIBO-REC(8)
  FIBO-REC(7)
    FIBO-REC(6)
      FIBO-REC(5)
        FIBO-REC(4)
          FIBO-REC(3)
            FIBO-REC(2)
              FIBO-REC(1)
              FIBO-REC(0)
            FIBO-REC(1)
          FIBO-REC(2)
            FIBO-REC(1)
            FIBO-REC(0)
        FIBO-REC(3)
          FIBO-REC(2)
            FIBO-REC(1)
            FIBO-REC(0)
          FIBO-REC(1)
      FIBO-REC(4)
        FIBO-REC(3)
          FIBO-REC(2)
            FIBO-REC(1)
            FIBO-REC(0)
          FIBO-REC(1)
        FIBO-REC(2)
          FIBO-REC(1)
          FIBO-REC(0)
```

```
    FIBO-REC(1)
  FIBO-REC(2)
    FIBO-REC(1)
    FIBO-REC(0)
FIBO-REC(5)
  FIBO-REC(4)
    FIBO-REC(3)
      FIBO-REC(2)
        FIBO-REC(1)
        FIBO-REC(0)
      FIBO-REC(1)
    FIBO-REC(2)
      FIBO-REC(1)
      FIBO-REC(0)
  FIBO-REC(3)
    FIBO-REC(2)
      FIBO-REC(1)
      FIBO-REC(0)
    FIBO-REC(1)
FIBO-REC(6)
  FIBO-REC(5)
    FIBO-REC(4)
      FIBO-REC(3)
```

```
      FIBO-REC(2)
        FIBO-REC(1)
        FIBO-REC(0)
      FIBO-REC(1)
    FIBO-REC(2)
      FIBO-REC(1)
      FIBO-REC(0)
  FIBO-REC(3)
    FIBO-REC(2)
      FIBO-REC(1)
      FIBO-REC(0)
    FIBO-REC(1)
FIBO-REC(4)
  FIBO-REC(3)
    FIBO-REC(2)
      FIBO-REC(1)
      FIBO-REC(0)
    FIBO-REC(1)
  FIBO-REC(2)
    FIBO-REC(1)
    FIBO-REC(0)
```

Programação dinâmica

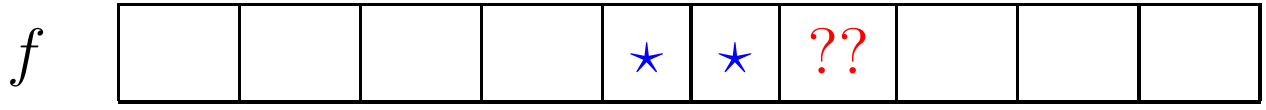
"Dynamic programming is a fancy name for divide-and-conquer with a table. Instead of solving subproblems recursively, solve them sequentially and store their solutions in a table. The trick is to solve them in the right order so that whenever the solution to a subproblem is needed, it is already available in the table. Dynamic programming is particularly useful on problems for which divide-and-conquer appears to yield an exponential number of subproblems, but there are really only a small number of subproblems repeated exponentially often. In this case, it makes sense to compute each solution the first time and store it away in a table for later use, instead of recomputing it recursively every time it is needed."

I. Parberry, *Problems on Algorithms*, Prentice Hall, 1995.

Algoritmo de programação dinâmica

```
FIBO (n)
1  f[0] ← 0
2  f[1] ← 1
3  para i ← 2 até n faça
4       f[i] ← f[i − 1] + f[i − 2]
5  devolva f[n]
```

Note a tabela $f[0\,..\,n-1]$.



Consumo de tempo (e de espaço) é $\Theta(n)$.

Algoritmo de programação dinâmica

Versão com economia de espaço.

```
FIBO (n)
0  se n = 0 então devolva 0
1  f_ant ← 0
2  f_atual ← 1
3  para i ← 2 até n faça
4      f_prox ← f_atual + f_ant
5      f_ant ← f_atual
6      f_atual ← f_prox
7  devolva f_atual
```

Consumo de tempo é $\Theta(n)$.

Consumo de espaço é $\Theta(1)$.

Versão recursiva eficiente

```
MEMOIZED-FIBO (f, n)
1  para i ← 0 até n faça
2       f[i] ← −1
3  devolva LOOKUP-FIBO (f, n)

LOOKUP-FIBO (f, n)
1  se f[n] ≥ 0
2       então devolva f[n]
3  se n ≤ 1
4       então f[n] ← n
5       senão f[n] ← LOOKUP-FIBO(f, n − 1)
                   + LOOKUP-FIBO(f, n − 2)
6  devolva f[n]
```

Não recalcula valores de f.

Programação dinâmica – multiplicação de cadeias de matrizes

Multiplicação de Cadeias de Matrizes

Problema: Multiplicação de Matrizes

Calcular o número mínimo de operações de multiplicação (escalar) necessários para computar a matriz M dada por:

$$M = M_1 \times M_2 \times \ldots M_i \ldots \times M_n$$

onde M_i é uma matriz de b_{i-1} linhas e b_i colunas, para todo $i \in \{1, \ldots, n\}$.

- Matrizes são multiplicadas aos pares sempre. Então, é preciso encontrar uma parentização (agrupamento) ótimo para a cadeia de matrizes.
- Consideraremos que para calcular a matriz M' dada por $M_i \times M_{i+1}$ são necessárias $b_{i-1} * b_i * b_{i+1}$ multiplicações entre os elementos de M_i e M_{i+1}.

Multiplicação de Cadeias de Matrizes (Cont.)

- **Exemplo:** Qual é o mínimo de multiplicações escalares necessárias para computar $M = M_1 \times M_2 \times M_3 \times M_4$ com $b = \{200, 2, 30, 20, 5\}$?
- As possibilidades de parentização são:

$$
\begin{array}{lcl}
M = (M_1 \times (M_2 \times (M_3 \times M_4))) & \rightarrow & 5.300 \text{ multiplicações} \\
M = (M_1 \times ((M_2 \times M_3) \times M_4)) & \rightarrow & 3.400 \text{ multiplicações} \\
M = ((M_1 \times M_2) \times (M_3 \times M_4)) & \rightarrow & 4.500 \text{ multiplicações} \\
M = ((M_1 \times (M_2 \times M_3)) \times M_4) & \rightarrow & 29.200 \text{ multiplicações} \\
M = (((M_1 \times M_2) \times M_3) \times M_4) & \rightarrow & 152.000 \text{ multiplicações}
\end{array}
$$

- A ordem das multiplicações faz **muita** diferença !

Multiplicação de Cadeias de Matrizes (Cont.)

- Poderíamos calcular o número de multiplicações para todas as possíveis parentizações.
- O número de possíveis parentizações é dado pela recorrência:

$$P(n) = \begin{cases} 1, & n = 1 \\ \sum_{k=1}^{n-1} P(k) \cdot P(n-k) & n > 1, \end{cases}$$

- Mas $P(n) \in \Omega(4^n / n^{\frac{3}{2}})$, a estratégia de força bruta é impraticável !

Multiplicação de Cadeias de Matrizes (Cont.)

- Inicialmente, para todo (i, j) tal que $1 \leq i \leq j \leq n$, vamos definir as seguintes matrizes:

$$M_{i,j} = M_i \times M_{i+1} \times \ldots \times M_j.$$

- Agora, dada uma parentização ótima, existem dois pares de parênteses que identificam o último par de matrizes que serão multiplicadas.
  Ou seja, existe k tal que $M = M_{i,k} \times M_{k+1,j}$.
- Como a parentização de M é ótima, as parentizações no cálculo de $M_{i,k}$ e $M_{k+1,j}$ devem ser ótimas também, caso contrário, seria possível obter uma parentização de M ainda melhor !
- Eis a subestrututra ótima do problema: a parentização ótima de M inclui a parentização ótima de $M_{i,k}$ e $M_{k+1,j}$.

Multiplicação de Cadeias de Matrizes (Cont.)

- De forma geral, se $m[i,j]$ é número mínimo de multiplicações que deve ser efetuado para computar $M_i \times M_{i+1} \times \ldots \times M_j$, então $m[i,j]$ é dado por:

$$m[i,j] := \min_{i \leq k < j} \{m[i,k] + m[k+1,j] + b_{i-1} * b_k * b_j\}.$$

- Podemos então projetar um algoritmo recursivo (indutivo) para resolver o problema.

Multiplicação de Matrizes - Algoritmo Recursivo

MinimoMultiplicacoesRecursivo(b, i, j)

▷ **Entrada:** Vetor b com as dimensões das matrizes e os índices i e j que delimitam o início e término da subcadeia.

▷ **Saída:** O número mínimo de multiplicações escalares necessário para computar a multiplicação da subcadeia. Esse valor é registrado em uma tabela ($m[i,j]$), bem como o índice da divisão em subcadeias ótimas ($s[i,j]$).

1. **se** $i = j$ **então devolva** 0
2. $m[i,j] := \infty$
3. **para** $k := i$ **até** $j - 1$ **faça**
4. $\quad q := \mathit{MinimoMultiplicacoesRecursivo}(b, i, k) +$
   $\quad\quad \mathit{MinimoMultiplicacoesRecursivo}(b, k+1, j) +$
   $\quad\quad b[i-1] * b[k] * b[j]$
5. $\quad$ **se** $m[i,j] > q$ **então**
6. $\quad\quad m[i,j] := q \; ; \; s[i,j] := k$
7. **devolva** $m[i,j]$.

Efetuando a Multiplicação Ótima

- É muito fácil efetuar a multiplicação da cadeia de matrizes com o número mínimo de multiplicações escalares usando a tabela s, que registra os índices ótimos de divisão em subcadeias.

MultiplicaMatrizes(M, s, i, j)

▷ **Entrada:** Cadeia de matrizes M, a tabela s e os índices i e j que delimitam a subcadeia a ser multiplicada.
▷ **Saída:** A matriz resultante da multiplicação da subcadeia entre i e j, efetuando o mínimo de multiplicações escalares.

1. **se** $i < j$ **então**
2. $\quad X := \mathit{MultiplicaMatrizes}(M, s, i, s[i, j])$
3. $\quad Y := \mathit{MultiplicaMatrizes}(M, s, s[i, j] + 1, j)$
4. $\quad$ **devolva** $\mathit{Multiplica}(X, Y, b[i-1], b[s[i, j]], b[j])$
5. **senão devolva** M_i;

Algoritmo Recursivo - Complexidade

- O número mínimo de operações feita pelo algoritmo recursivo é dada pela recorrência:

$$T(n) \geq \begin{cases} 1, & n = 1 \\ 1 + \sum_{k=1}^{n-1} [T(k) + T(n-k) + 1] & n > 1, \end{cases}$$

- Portanto, $T(n) \geq 2 \sum_{k=1}^{n-1} T(i) + n$, para $n > 1$.
- É possível provar (por substituição) que $T(n) \geq 2^{n-1}$, ou seja, o algoritmo recursivo tem complexidade $\Omega(2^n)$, ainda impraticável !

Algoritmo Recursivo - Complexidade

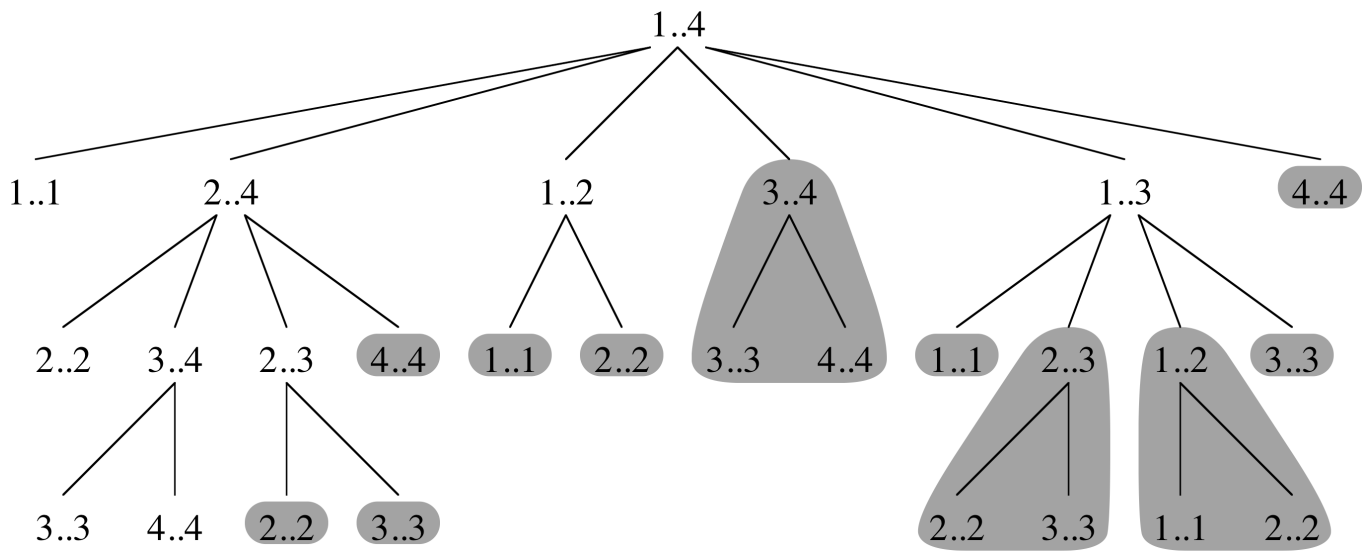
- A ineficiência do algoritmo recursivo deve-se à sobreposição de subproblemas: o cálculo do mesmo $m[i,j]$ pode ser requerido em vários subproblemas.
- Por exemplo, para $n = 4$, $m[1,2]$, $m[2,3]$ e $m[3,4]$ são computados duas vezes.
- O número de total de $m[i,j]$'s calculados é $O(n^2)$ apenas !
- Portanto, podemos obter um algoritmo mais eficiente se evitarmos recálculos de subproblemas.

Memorização x Programação Dinâmica

- Existem duas técnicas para evitar o recálculo de subproblemas:
  - **Memorização:** Consiste em manter a estrutura recursiva do algoritmo, registrando em uma tabela o valor ótimo para subproblemas já computados e verificando, antes de cada chamada recursiva, se o subproblema a ser resolvido já foi computado.
  - **Programação Dinâmica:** Consiste em preencher uma tabela que registra o valor ótimo para cada subproblema de forma apropriada, isto é, a computação do valor ótimo de cada subproblema depende somente de subproblemas já previamente computados. Elimina completamente a recursão.

Algoritmo de Memorização

- Árvore de recursão do algoritmo de memorização para multiplicação de cadeia de matrizes **Memorizacao**$(b, 1, 4)$
  - As chamadas às subárvores em cinza são trocadas por consultas a valores já calculados e *memorizados* (ou, se preferir, o neologismo "*memoizados*") na tabela



Algoritmo de Memorização

MinimoMultiplicacoesMemorizado(b, n)

1. **para** $i := 1$ **até** n **faça**
2. **para** $j := 1$ **até** n **faça**
3. $m[i,j] := \infty$
4. **devolva** $Memorizacao(b, 1, n)$

Memorizacao(b, i, j)

1. **se** $m[i,j] < \infty$ **então devolva** $m[i,j]$
2. **se** $i = j$ **então** $m[i,j] := 0$
3. **senão**
4. **para** $k := i$ **até** $j - 1$ **faça**
5. $q := Memorizacao(b, i, k) +$
$Memorizacao(b, k+1, j) + b[i-1] * b[k] * b[j];$
6. **se** $m[i,j] > q$ **então** $m[i,j] := q$; $s[i,j] := k$
7. **devolva** $m[i,j]$

Algoritmo de Programação Dinâmica

- O uso de programação dinâmica é preferível pois elimina completamente o uso de recursão.
- O algoritmo de programação dinâmica para o problema da multiplicação de matrizes torna-se trivial se computarmos, para valores crescentes de u, o valor ótimo de todas as subcadeias de tamanho u.

Algoritmo de Programação Dinâmica

MinimoMultiplicacoes(b)

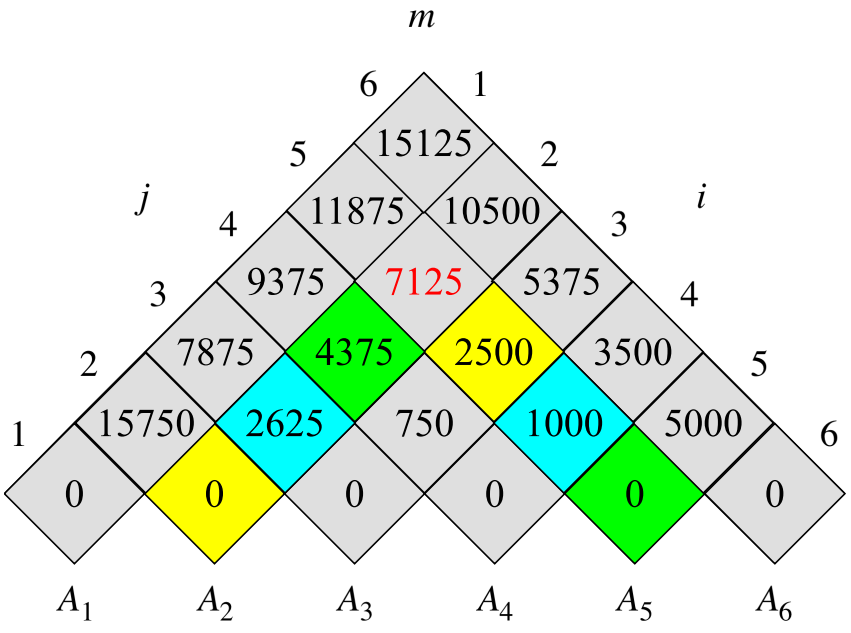
▷ **Entrada:** Vetor b com dimensões das matrizes.
▷ **Saída:** As tabelas m e s preenchidas.

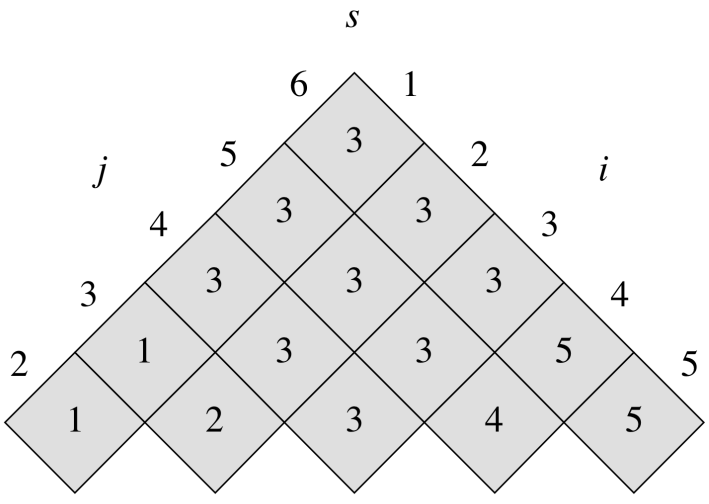
```
 1: n ← b.tamanho − 1
 2: para i ← 1 até n faça
 3:     m[i, i] ← 0
 4: para u ← 2 até n faça
 5:             ▷ calcula o mínimo de todas sub-cadeias de tamanho u
 6:     para i ← 1 até n − u + 1 faça
 7:         j ← i + u − 1
 8:         m[i, j] ← ∞
 9:         para k ← i até j − 1 faça
10:             q ← m[i, k] + m[k + 1, j] + b[i − 1] ∗ b[k] ∗ b[j]
11:             se q < m[i, j] então
12:                 m[i, j] ← q    ;    s[i, j] ← k
13: devolve (m, s)
```

Algoritmo de Programação Dinâmica – exemplo

- Cadeia de $n = 6$ matrizes com as seguintes dimensões:

matriz		A_1	A_2	A_3	A_4	A_5	A_6
dimensão		30×35	35×15	15×5	5×10	10×20	20×25
vetor b	30	35	15	5	10	20	25





$$m[2,5] = \min \begin{cases} \text{m[2,2]} + \text{m[3,5]} + b[1] * b[2] * b[5] = 0 + 2500 + 35 \cdot 15 \cdot 20 = 13000 \\ \text{m[2,3]} + \text{m[4,5]} + b[1] * b[3] * b[5] = 2625 + 1000 + 35 \cdot 5 \cdot 20 = 7125 \\ \text{m[2,4]} + \text{m[5,5]} + b[1] * b[4] * b[5] = 4375 + 0 + 35 \cdot 10 \cdot 20 = 11375 \end{cases}$$

$$= 7125$$

Cadeia de Matrizes – Paretinzação Ótima

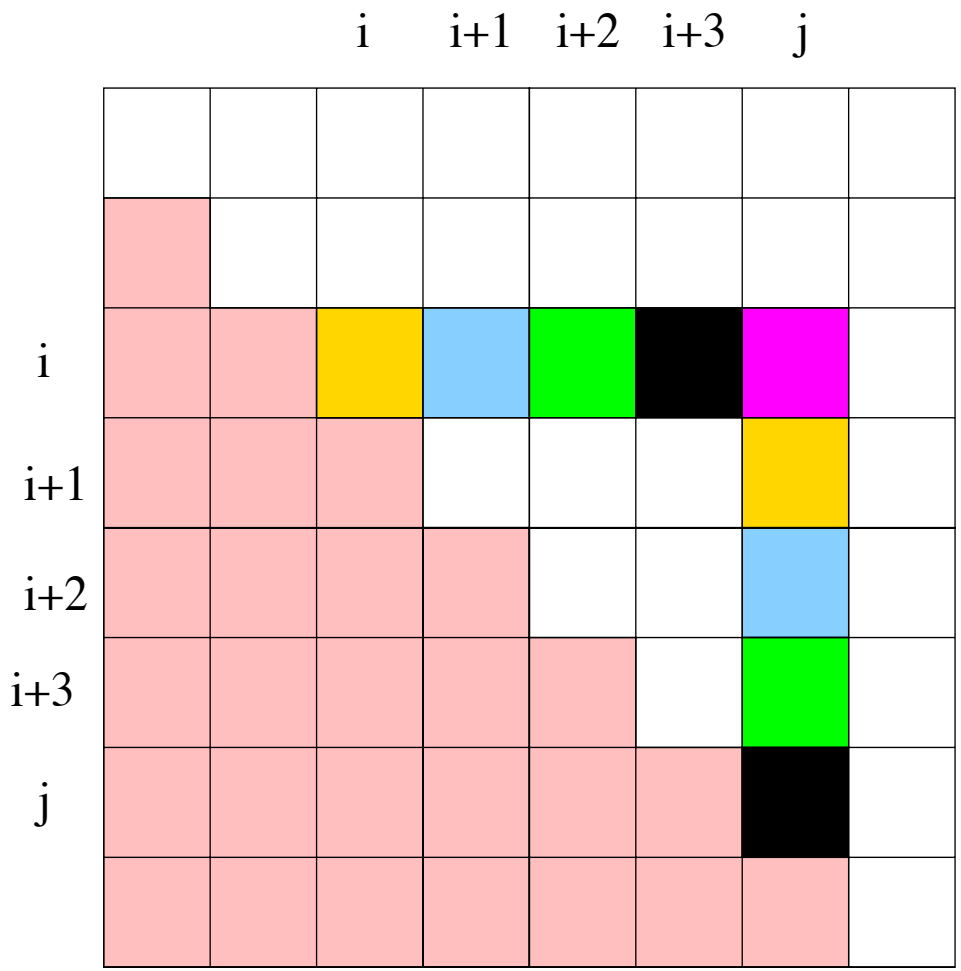
IMPRIMA-PARENTIZACAO-OTIMA(s, i, j)

```
1: se i = j então
2:     imprima "A_i"
3: senão
4:     imprima "("
5:     IMPRIMA-PARENTIZACAO-OTIMA(s, i, s[i, j])
6:     IMPRIMA-PARENTIZACAO-OTIMA(s, s[i, j] + 1, j)
7:     imprima ")"
```

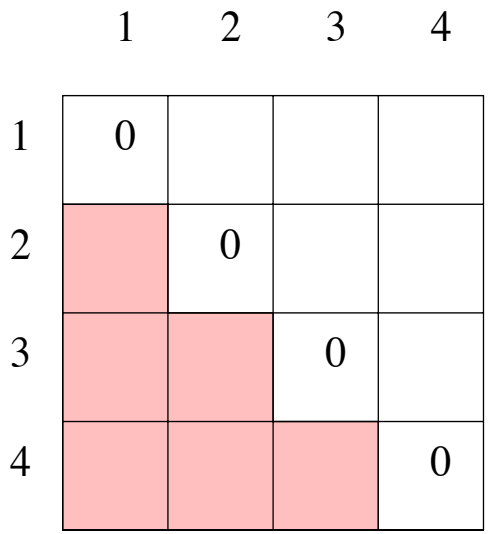
No exemplo anterior, se for feita a chamada
IMPRIMA-PARENTIZACAO-OTIMA$(s, 1, 6)$
será impresso:

$$((A_1(A_2A_3))((A_4A_5)A_6))$$

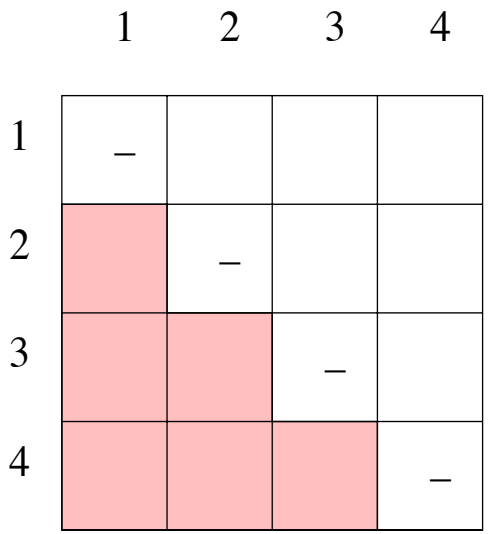
Algoritmo de Programação Dinâmica - Exemplo



Algoritmo de Programação Dinâmica - Exemplo



m



s

Algoritmo de Programação Dinâmica - Exemplo

	1	2	3	4
1	0	12000		
2		0	1200	
3			0	3000
4				0

m

	1	2	3	4
1	_	1		
2		_	2	
3			_	3
4				_

s

$\{ 200, 2, 30, 20, 5 \}$

Algoritmo de Programação Dinâmica - Exemplo

	1	2	3	4
1	0	12000	9200	
2		0	1200	
3			0	3000
4				0

m

	1	2	3	4
1	_	1	1	
2		_	2	
3			_	3
4				_

s

{ 200, 2, 30, 20, 5 }

b0*b1*b3=200*2*20=8000

b0*b2*b3=200*30*20=120000

Algoritmo de Programação Dinâmica - Exemplo

	1	2	3	4
1	0	12000	9200	
2		0	1200	1400
3			0	3000
4				0

m

	1	2	3	4
1	_	1	1	
2		_	2	3
3			_	3
4				_

s

{ 200, 2, 30, 20, 5 }

b1*b2*b4=2*30*5=300

b1*b3*b4=2*20*5=200

Algoritmo de Programação Dinâmica - Exemplo

	1	2	3	4
1	0	12000	9200	3400
2		0	1200	1400
3			0	3400
4				0

m

	1	2	3	4
1	_	1	1	1
2		_	2	3
3			_	3
4				_

s

b0*b1*b4=200*2*5=2000

b0*b2*b4=200*30*5=30000

b0*b3*b4=200*20*5=20000

{ 200, 2, 30, 20, 5 }

Cadeia de Matrizes – Paretinzação Ótima

IMPRIMA-PARENTIZACAO-OTIMA(s, i, j)

1: **se** $i = j$ **então**
2: imprima "A_i"
3: **senão**
4: imprima "("
5: IMPRIMA-PARENTIZACAO-OTIMA$(s, i, s[i, j])$
6: IMPRIMA-PARENTIZACAO-OTIMA$(s, s[i, j] + 1, j)$
7: imprima ")"

Algoritmo de Programação Dinâmica - Exemplo

	1	2	3	4
1	0	12000	9200	3400
2		0	1200	1400
3			0	3000
4				0

m

	1	2	3	4
1	_	1	1	1
2		_	2	3
3			_	3
4				_

s

M1 ((M2 . M3) . M4)

Algoritmo de Programação Dinâmica - Complexidade

- A complexidade de tempo do algoritmo é dada por:

$$
\begin{aligned}
T(n) &= \sum_{u=1}^{n-1} \sum_{i=1}^{n-u} \sum_{k=i}^{i+u-1} \Theta(1) \\
&= \sum_{u=1}^{n-1} \sum_{i=1}^{n-u} u\ \Theta(1) \\
&= \sum_{u=1}^{n-1} u(n-u)\ \Theta(1) \\
&= \sum_{u=1}^{n-1} (nu - u^2)\ \Theta(1).
\end{aligned}
$$

Algoritmo de Programação Dinâmica - Complexidade

- Como

$$\sum_{u=1}^{n-1} nu = n^3/2 - n^2/2$$

e

$$\sum_{u=1}^{n-1} u^2 = n^3/3 - n^2/2 + n/6.$$

Então,

$$T(n) = (n^3/6 - n/6)\ \Theta(1).$$

- A complexidade de tempo do algoritmo é $\Theta(n^3)$.
- A complexidade de espaço é $\Theta(n^2)$, já que é necessário armazenar a matriz com os valores ótimos dos subproblemas.

Programação dinâmica – o problema binário da mochila

O Problema Binário da Mochila

O Problema da Mochila

Dada uma mochila de capacidade W (inteiro) e um conjunto de n itens com tamanho w_i (inteiro) e valor c_i associado a cada item i, queremos determinar quais itens devem ser colocados na mochila de modo a **maximizar** o valor total transportado, respeitando sua capacidade.

- Podemos fazer as seguintes suposições:
  - $\sum_{i=1}^{n} w_i > W$;
  - $0 < w_i \leq W$, para todo $i = 1, \ldots, n$.

O Problema Binário da Mochila

- Podemos formular o problema da mochila com Programação Linear Inteira:
  - Criamos uma variável x_i para cada item: $x_i = 1$ se o item i estiver na solução ótima e $x_i = 0$ caso contrário.
  - A modelagem do problema é simples:

$$\max \sum_{i=1}^{n} c_i x_i \tag{1}$$

$$\sum_{i=1}^{n} w_i x_i \leq W \tag{2}$$

$$x_i \in \{0, 1\} \tag{3}$$

- (1) é a função objetivo e (2-3) o conjunto de restrições.

O Problema Binário da Mochila

- Como podemos projetar um algoritmo para resolver o problema?
- Existem 2^n possíveis subconjuntos de itens: um algoritmo de força bruta é impraticável!
- É um problema de otimização. Será que tem subestrutura ótima?
- Se o item n estiver na solução ótima, o valor desta solução será c_n mais o valor da melhor solução do problema da mochila com capacidade $W - w_n$ considerando-se só os $n - 1$ primeiros itens.
- Se o item n não estiver na solução ótima, o valor ótimo será dado pelo valor da melhor solução do problema da mochila com capacidade W considerando-se só os $n - 1$ primeiros itens.

O Problema Binário da Mochila

- Seja $z[k, d]$ o valor ótimo do problema da mochila considerando-se uma capacidade d para a mochila que contém um subconjunto dos k primeiros itens da instância original.
- A fórmula de recorrência para computar $z[k, d]$ para todo valor de d e k é:

$$z[0, d] = 0$$

$$z[k, 0] = 0$$

$$z[k, d] = \begin{cases} z[k-1, d], & \text{se } w_k > d \\ \max\{z[k-1, d], z[k-1, d-w_k] + c_k\}, & \text{se } w_k \leq d \end{cases}$$

O Problema Binário da Mochila - Complexidade Recursão

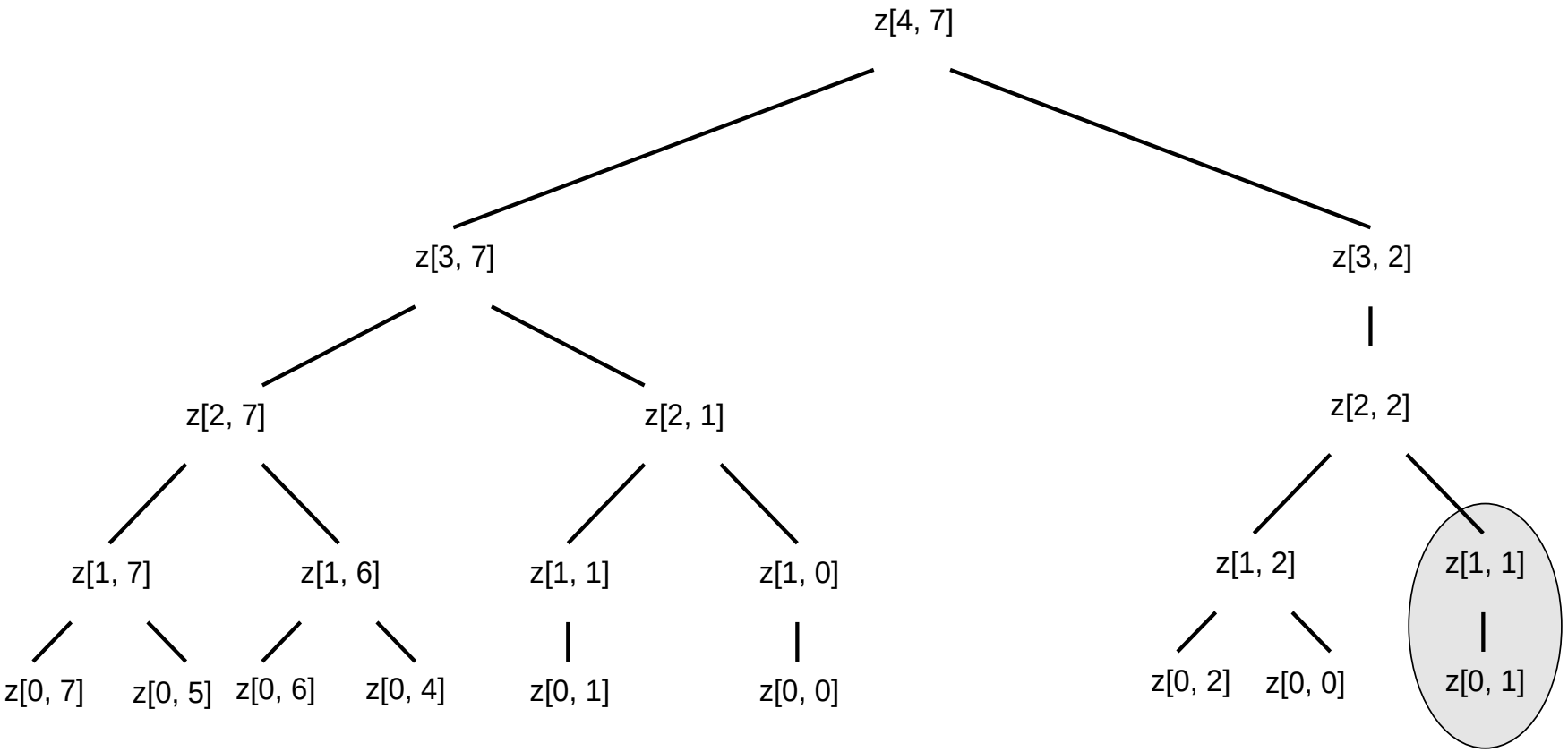
- A complexidade do algoritmo recursivo para este problema no pior caso é dada pela recorrência:

$$T(k,d) = \begin{cases} 1, & k = 0 \text{ ou } d = 0 \\ T(k-1,d) + T(k-1,d-w_k) + 1 & k > 0 \text{ e } d > 0. \end{cases}$$

- Portanto, no pior caso, o algoritmo recursivo tem complexidade $\Omega(2^n)$. É impraticável!
- Mas há sobreposição de subproblemas: o recálculo de subproblemas pode ser evitado!

Mochila - Sobreposição de Subproblemas

- Considere vetor de tamanhos $w = \{2, 1, 6, 5\}$ e capacidade da mochila $W = 7$. A árvore de recursão seria:

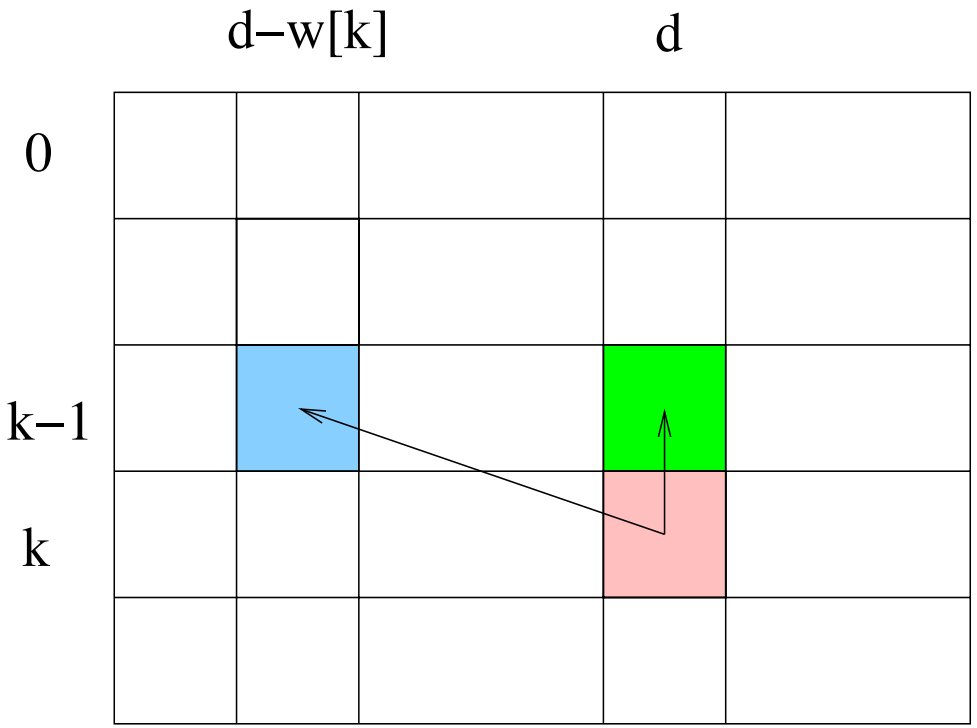


- O subproblema $z[1, 1]$ é computado duas vezes.

Mochila - Programação Dinâmica

- O número total máximo de subproblemas a serem computados é nW.
- Isso porque tanto o tamanho dos itens quanto a capacidade da mochila são inteiros!
- Podemos então usar programação dinâmica para evitar o recálculo de subproblemas.
- Como o cálculo de $z[k, d]$ depende de $z[k-1, d]$ e $z[k-1, d-w_k]$, preenchemos a tabela linha a linha.

Mochila



$$z[k,d] = \max \{ z[k-1,d] , z[k-1,d-w[k]] + c[k] \}$$

O Problema Binário da Mochila - Algoritmo

Mochila(c, w, W, n)

▷ **Entrada:** Vetores c e w com valor e tamanho de cada item, capacidade W da mochila e número de itens n.

▷ **Saída:** O valor máximo do total de itens colocados na mochila.

1. **para** $d := 0$ **até** W **faça** $z[0, d] := 0$
2. **para** $k := 1$ **até** n **faça** $z[k, 0] := 0$
3. **para** $k := 1$ **até** n **faça**
4. **para** $d := 1$ **até** W **faça**
5. $z[k, d] := z[k-1, d]$
6. **se** $w_k \leq d$ **e** $c_k + z[k-1, d-w_k] > z[k, d]$ **então**
7. $z[k, d] := c_k + z[k-1, d-w_k]$
8. **devolva** $(z[n, W])$

Mochila - Exemplo

- Exemplo: $c = \{10, 7, 25, 24\}$, $w = \{2, 1, 6, 5\}$ e $W = 7$.

k \ d	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0							
2	0							
3	0							
4	0							

Mochila - Exemplo

- Exemplo: $c = \{10, 7, 25, 24\}$, $w = \{2, 1, 6, 5\}$ e $W = 7$.

k \ d	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	10	10	10	10	10	10
2	0							
3	0							
4	0							

Mochila - Exemplo

- Exemplo: $c = \{10, 7, 25, 24\}$, $w = \{2, 1, 6, 5\}$ e $W = 7$.

k \ d	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	10	10	10	10	10	10
2	0	7	10	17	17	17	17	17
3	0							
4	0							

Mochila - Exemplo

- Exemplo: $c = \{10, 7, 25, 24\}$, $w = \{2, 1, 6, 5\}$ e $W = 7$.

k \ d	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	10	10	10	10	10	10
2	0	7	10	17	17	17	17	17
3	0	7	10	17	17	17	25	32
4	0							

Mochila - Exemplo

- Exemplo: $c = \{10, 7, 25, 24\}$, $w = \{2, 1, 6, 5\}$ e $W = 7$.

k \ d	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	10	10	10	10	10	10
2	0	7	10	17	17	17	17	17
3	0	7	10	17	17	17	25	32
4	0	7	10	17	17	24	31	34

Mochila - Exemplo

- Exemplo: $c = \{10, 7, 25, 24\}$, $w = \{2, 1, 6, 5\}$ e $W = 7$.

k \ d	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	10	10	10	10	10	10
2	0	7	10	17	17	17	17	17
3	0	7	10	17	17	17	25	32
4	0	7	10	17	17	24	31	34

Mochila - Complexidade

- Claramente, a complexidade do algoritmo de programação dinâmica para o problema da mochila é $O(nW)$.
- É um algoritmo pseudo-polinomial: sua complexidade depende do valor de W, parte da entrada do problema.
- O algoritmo não devolve o subconjunto de valor total máximo, apenas o valor máximo.
- É fácil recuperar o subconjunto a partir da tabela z preenchida.

Mochila - Recuperação da Solução

MochilaSolucao(z, n, W)

▷ **Entrada:** Tabela z preenchida, capacidade W da mochila e número de itens n.

▷ **Saída:** O vetor x que indica os itens colocados na mochila.

para $i := 1$ **até** n **faça** $x[i] := 0$

MochilaSolucaoAux(x, z, n, W)

devolva (x)

MochilaSolucaoAux(x, z, k, d)

se $k \neq 0$ **então**

  se $z[k, d] = z[k-1, d]$ **então**

    $x[k] := 0$; *MochilaSolucaoAux*$(x, z, k-1, d)$

  senão

    $x[k] := 1$; *MochilaSolucaoAux*$(x, z, k-1, d-w_k)$

Mochila - Exemplo

- Exemplo: $c = \{10, 7, 25, 24\}$, $w = \{2, 1, 6, 5\}$ e $W = 7$.

k \ d	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	10	10	10	10	10	10
2	0	7	10	17	17	17	17	17
3	0	7	10	17	17	17	25	32
4	0	7	10	17	17	24	31	34

Mochila - Exemplo

- Exemplo: $c = \{10, 7, 25, 24\}$, $w = \{2, 1, 6, 5\}$ e $W = 7$.

k \ d	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	10	10	10	10	10	10
2	0	7	10	17	17	17	17	17
3	0	7	10	17	17	17	25	32
4	0	7	10	17	17	24	31	34

Mochila - Exemplo

- Exemplo: $c = \{10, 7, 25, 24\}$, $w = \{2, 1, 6, 5\}$ e $W = 7$.

k \ d	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	10	10	10	10	10	10
2	0	7	10	17	17	17	17	17
3	0	7	10	17	17	17	25	32
4	0	7	10	17	17	24	31	34

Mochila - Exemplo

- Exemplo: $c = \{10, 7, 25, 24\}$, $w = \{2, 1, 6, 5\}$ e $W = 7$.

k \ d	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	10	10	10	10	10	10
2	0	7	10	17	17	17	17	17
3	0	7	10	17	17	17	25	32
4	0	7	10	17	17	24	31	34

Mochila - Exemplo

- Exemplo: $c = \{10, 7, 25, 24\}$, $w = \{2, 1, 6, 5\}$ e $W = 7$.

k \ d	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	10	10	10	10	10	10
2	0	7	10	17	17	17	17	17
3	0	7	10	17	17	17	25	32
4	0	7	10	17	17	24	31	34

$x[1] = x[4] = 1\ ,\ x[2] = x[3] = 0$

Mochila - Complexidade

- O algoritmo de recuperação da solução tem complexidade $O(n)$.
- Portanto, a complexidade de tempo e de espaço do algoritmo de programação dinâmica para o problema da mochila é $O(nW)$.
- É possível economizar memória, registrando duas linhas: a que está sendo preenchida e a anterior. Mas isso inviabiliza a recuperação da solução.

Programação dinâmica – subcadeia comum máxima

Subcadeia comum máxima

Definição: Subcadeia

Dada uma cadeia $S = a_1 \ldots a_n$, dizemos que $S' = b_1 \ldots b_p$ é uma *subcadeia* de S se existem p índices $i(j)$ satisfazendo:

(a) $i(j) \in \{1, \ldots, n\}$ para todo $j \in \{1, \ldots, p\}$;

(b) $i(j) < i(j+1)$ para todo $j \in \{1, \ldots, p-1\}$;

(c) $b_j = a_{i(j)}$ para todo $j \in \{1, \ldots, p\}$.

- **Exemplo:** $S = ABCDEFG$ e $S' = ADFG$.

Problema da Subcadeia Comum Máxima

Dadas duas cadeias de caracteres X e Y de um alfabeto Σ, determinar a maior subcadeia comum de X e Y

Subcadeia comum máxima (cont.)

- É um problema de otimização. Será que tem subestrutura ótima?
- **Notação:** Seja S uma cadeia de tamanho n. Para todo $i = 1, \ldots, n$, o prefixo de tamanho i de S será denotado por S_i.
- **Exemplo:** Para $S = ABCDEFG$, $S_2 = AB$ e $S_4 = ABCD$.
- **Definição:** $c[i, j]$ é o tamanho da subcadeia comum máxima dos prefixos X_i e Y_j. Logo, se $|X| = m$ e $|Y| = n$, $c[m, n]$ é o valor ótimo.

Subcadeia comum máxima (cont.)

- **Teorema (subestrutura ótima):** Seja $Z = z_1 \ldots z_k$ a subcadeia comum máxima de $X = x_1 \ldots x_m$ e $Y = y_1 \ldots y_n$, denotado por $Z = \mathtt{SCM}(X, Y)$.
  1. Se $x_m = y_n$ então $z_k = x_m = y_n$ e $Z_{k-1} = \mathtt{SCM}(X_{m-1}, Y_{n-1})$.
  2. Se $x_m \neq y_n$ então $z_k \neq x_m$ implica que $Z = \mathtt{SCM}(X_{m-1}, Y)$.
  3. Se $x_m \neq y_n$ então $z_k \neq y_n$ implica que $Z = \mathtt{SCM}(X, Y_{n-1})$.

- **Fórmula de Recorrência:**

$$c[i,j] = \begin{cases} 0 & \text{se } i = 0 \text{ ou } j = 0 \\ c[i-1, j-1] + 1 & \text{se } i, j > 0 \text{ e } x_i = y_j \\ \max\{c[i-1, j], c[i, j-1]\} & \text{se } i, j > 0 \text{ e } x_i \neq y_j \end{cases}$$

Subcadeia comum máxima (cont.)

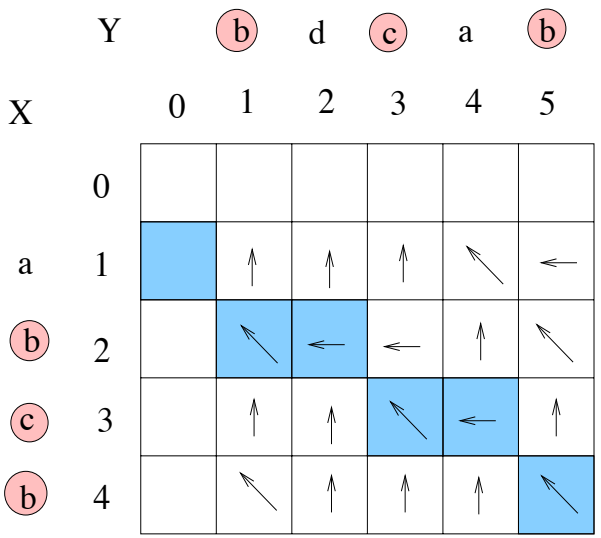
SCM(X, m, Y, n, c, b)

```
01.  para i = 0 até m faça c[i,0] := 0
02.  para j = 1 até n faça c[0,j] := 0
03.  para i = 1 até m faça
04.      para j = 1 até n faça
05.          se x_i = y_j então
06.              c[i,j] := c[i-1,j-1] + 1 ; b[i,j] := "↖"
07.          senão
08.              se c[i,j-1] > c[i-1,j] então
09.                  c[i,j] := c[i,j-1] ; b[i,j] := "←"
10.              senão
11.                  c[i,j] := c[i-1,j]; b[i,j] := "↑";
12.  devolva (c[m,n], b).
```

Subcadeia comum máxima - Exemplo

- **Exemplo:** $X = abcb$ e $Y = bdcab$, $m = 4$ e $n = 5$.

		Y	b	d	c	a	b
		0	1	2	3	4	5
X	0	0	0	0	0	0	0
a	1	0	0	0	0	1	1
b	2	0	1	1	1	1	2
c	3	0	1	1	2	2	2
b	4	0	1	1	2	2	3



Subcadeia comum máxima - Complexidade

- Claramente, a complexidade do algoritmo é $O(mn)$.
- O algoritmo não encontra a subcadeia comum de tamanho máximo, apenas seu tamanho.
- Com a tabela b preenchida, é fácil encontrar a subcadeia comum máxima.

Subcadeia comum máxima (cont.)

- Para recuperar a solução, basta chamar $Recupera\_MSC(b, X, m, n)$.

Recupera_SCM(b, X, i, j)

1. **se** $i = 0$ e $j = 0$ **então devolva**
2. **se** $b[i,j] =$ "$\nwarrow$" **então**
3. $Recupera\_MSC(b, X, i-1, j-1)$; **imprima** x_i
4. **senão**
5. **se** $b[i,j] =$ "$\uparrow$" **então**
6. $Recupera\_MSC(b, X, i-1, j)$
7. **senão**
8. $Recupera\_MSC(b, X, i, j-1)$

Subcadeia comum máxima - Complexidade

- A determinação da subcadeia comum máxima é feita em tempo $O(m+n)$ no pior caso.
- Portanto, a complexidade de tempo e de espaço do algoritmo de programação dinâmica para o problema da subcadeia comum máxima é $O(mn)$.
- Note que a tabela b é dispensável, podemos economizar memória recuperando a solução a partir da tabela c. Ainda assim, o gasto de memória seria $O(mn)$.
- Caso não haja interesse em determinar a subcadeia comum máxima, mas apenas seu tamanho, é possível reduzir o gasto de memória para $O(\min\{n,m\})$: basta registrar apenas a linha da tabela sendo preenchida e a anterior.