Estruturas de Dados e Algoritmos II

Vasco Pedro

Departamento de Informática
Universidade de Évora

2019/2020

Pseudo-código

Exemplo

PESQUISA-LINEAR(V, k)

```
1 n <- |V|                                  // inicialização
2 i <- 1
3 while i <= n and V[i] != k do             // pesquisa
4     i <- i + 1
5 if i <= n then                            // resultado:
6     return i                              //  - sucesso
7 return -1                                 //  - insucesso
```

`|V|`	nº de elementos de um vector — $O(1)$
`V[1..|V|]`	elementos do vector
`and` e `or`	só é avaliado o segundo operando se necessário
`variável.campo`	acesso a um campo de um "objecto"

Análise da complexidade (1)

Exemplo

Análise da complexidade temporal, no pior caso, da função PESQUISA-LINEAR, por linha de código

1. Obtenção da dimensão de um vector, afectação: operações com complexidade (temporal) constante

$$O(1) + O(1) = O(1)$$

2. Afectação: $O(1)$
3. Acessos a `i`, `n`, `V[i]` e `k`, comparações e saltos condicionais com complexidade constante

$$4\,O(1) + 2\,O(1) + 2\,O(1) = O(1)$$

Executada, no pior caso, $|\mathtt{V}|+1$ vezes

$$(|\mathtt{V}| + 1) \times O(1) = O(|\mathtt{V}|)$$

Análise da complexidade (2)

Exemplo

4. Acesso a `i`, soma e afectação: $O(1) + O(1) + O(1) = O(1)$
   Executada, no pior caso, `|V|` vezes

$$|\texttt{V}| \times O(1) = O(|\texttt{V}|)$$

5. Acesso a `i` e `n`, comparação e salto condicional com complexidade constante

$$2\,O(1) + O(1) + O(1) = O(1)$$

6. Saída de função com complexidade constante: $O(1)$
7. Saída de função com complexidade constante: $O(1)$

Análise da complexidade (3)

Exemplo

Juntando tudo

$$O(1) + O(1) + O(|\mathtt{V}|) + O(|\mathtt{V}|) + O(1) + \max\{O(1), O(1)\} =$$
$$= 4\,O(1) + 2\,O(|\mathtt{V}|) =$$
$$= O(|\mathtt{V}|)$$

No pior caso, a função PESQUISA-LINEAR tem complexidade temporal linear na dimensão do vector `V`

Se n representar a dimensão do vector `V`, o tempo $T(n)$ que a função demora a executar tem complexidade linear em n

$$T(n) = O(n)$$

Isto significa que o tempo que a função demora a executar varia linearmente com a dimensão do *input*

A notação O (1)

$$O(g(n)) = \{f(n) : \exists_{c,n_0>0} \text{ tais que } \forall_{n \geq n_0} \; 0 \leq f(n) \leq c\, g(n)\}$$

- $O(n) = \{f(n) : \exists_{c,n_0>0} \text{ tais que } \forall_{n \geq n_0} \; 0 \leq f(n) \leq c\, n\}$

  $n = O(n)$ $\quad$ $2n + 5 = O(n)$ $\quad$ $\log n = O(n)$ $\quad$ $n^2 \neq O(n)$

- $O(n^2) = \{f(n) : \exists_{c,n_0>0} \text{ tais que } \forall_{n \geq n_0} \; 0 \leq f(n) \leq c\, n^2\}$

  $n^2 = O(n^2)$ $\quad$ $4n^2 + n = O(n^2)$ $\quad$ $n = O(n^2)$ $\quad$ $n^3 \neq O(n^2)$

- $O(\log n) = \{f(n) : \exists_{c,n_0>0} \text{ tais que } \forall_{n \geq n_0} \; 0 \leq f(n) \leq c \log n\}$

  $1 + \log n = O(\log n)$ $\quad$ $\log n^2 = O(\log n)$ $\quad$ $n \neq O(\log n)$

Escreve-se $f(n) = O(g(n))$ em vez de $f(n) \in O(g(n))$

Lê-se $f(n)$ é O de $g(n)$

A notação O (2)

$$\Omega(g(n)) = \{f(n) : \exists_{c,n_0>0} \text{ tais que } \forall_{n \geq n_0} \ 0 \leq c\, g(n) \leq f(n)\}$$

$$n = \Omega(n) \qquad n^2 = \Omega(n) \qquad \log n \neq \Omega(n^2)$$

$$\Theta(g(n)) = \{f(n) : \exists_{c_1,c_2,n_0>0} \text{ t.q. } \forall_{n \geq n_0} \ 0 \leq c_1\, g(n) \leq f(n) \leq c_2\, g(n)\}$$

$$3n^2 + n = \Theta(n^2) \qquad n \neq \Theta(n^2) \qquad n^2 \neq \Theta(n)$$

$$o(g(n)) = \{f(n) : \forall_{c>0} \ \exists_{n_0>0} \text{ tal que } \forall_{n \geq n_0} \ 0 \leq f(n) < c\, g(n)\}$$

$$n = o(n^2) \qquad n^2 \neq o(n^2)$$

$$\omega(g(n)) = \{f(n) : \forall_{c>0} \ \exists_{n_0>0} \text{ tal que } \forall_{n \geq n_0} \ 0 \leq c\, g(n) < f(n)\}$$

$$n = \omega(\log n) \qquad n^2 = \omega(\log n) \qquad \log n \neq \omega(\log n)$$

A notação O (3)

Traduzindo...

$f(n) = O(g(n))$	$f(n)$ não cresce mais depressa que $g(n)$
$f(n) = o(g(n))$	$f(n)$ cresce mais devagar que $g(n)$
$f(n) = \Omega(g(n))$	$f(n)$ não cresce mais devagar que $g(n)$
$f(n) = \omega(g(n))$	$f(n)$ cresce mais depressa que $g(n)$
$f(n) = \Theta(g(n))$	$f(n)$ e $g(n)$ crescem com o mesmo ritmo

Informação persistente (1)

Enquadramento

- Estruturas de dados em memória central desaparecem quando programa termina
- Volume dos dados pode não permitir
  - o armazenamento em memória central
  - o seu processamento sempre que é necessário aceder-lhes
- Dados persistentes, em memória secundária, requerem estruturas de dados persistentes

Condicionantes

- Acessos a memória secundária (10^{-3} s) muito mais caros que acessos à memória central (10^{-9} s)
- Transferências entre a memória central e a memória secundária processadas por páginas (4096 *bytes* é uma dimensão comum)

Informação persistente (2)

Dados em memória secundária

Estratégia

Minimizar o número de acessos a memória secundária

- Adaptando as estruturas de dados
- Usando estruturas de dados especialmente concebidas

  Em ambos os casos, procura-se tirar o maior partido possível do conteúdo das páginas acedidas

- Fazendo *cacheing* da informação

Cuidados

Garantir que a informação em memória secundária se mantém actualizada

- Operações só ficam completas quando as alterações são escritas na memória secundária

B-Trees

B-Trees

Objectivos

Grandes quantidades de informação

Armazenamento em memória secundária

Indexação eficiente

Minimização de acessos a memória secundária

B-Trees

Características (1)

São árvores

Princípios semelhantes aos das árvores binárias de pesquisa

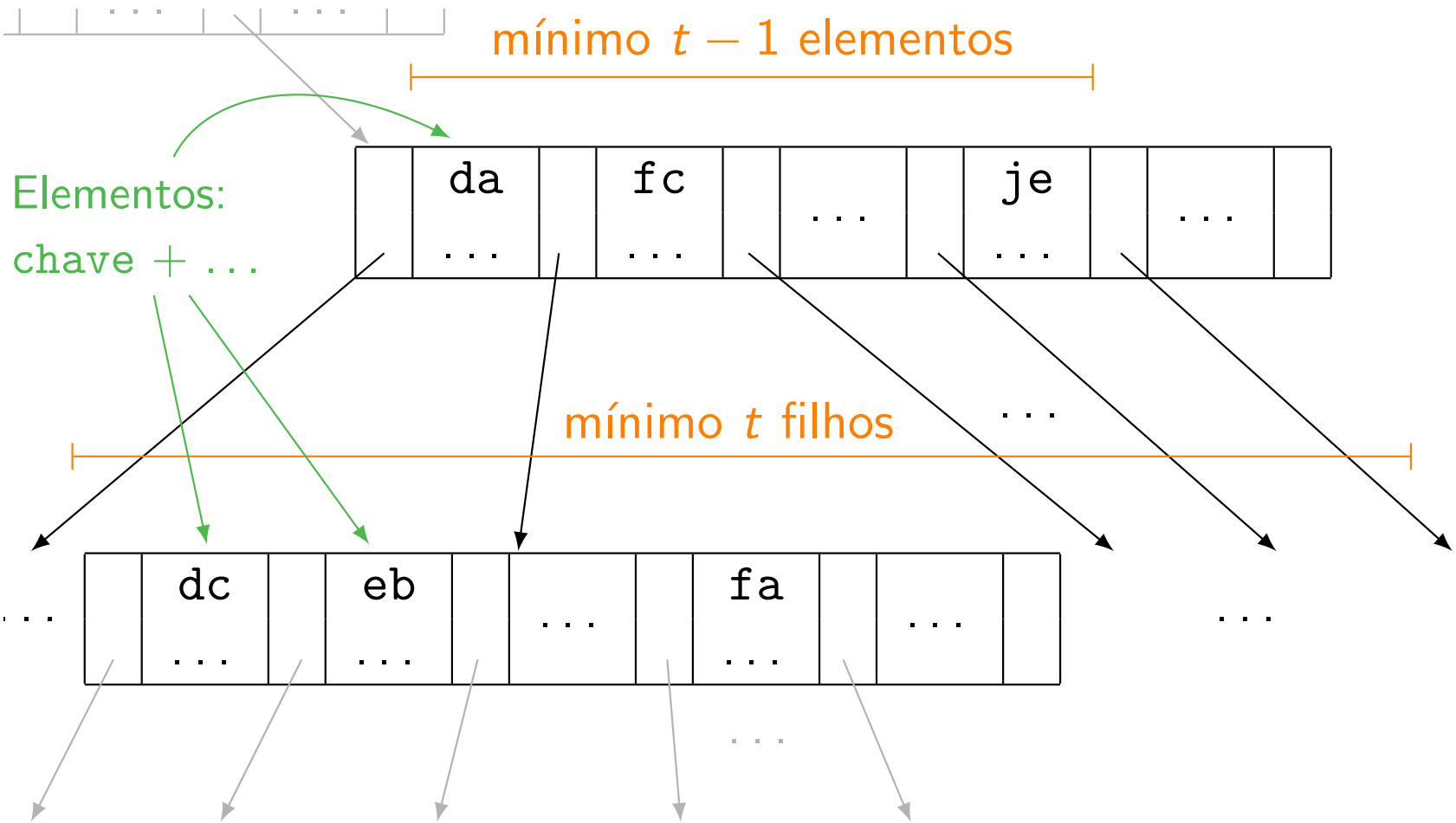
Perfeitamente equilibradas

Nós com número variável de filhos (pelo menos 2)

Nós com número variável de elementos

B-Trees

Estrutura dos nós internos (exceptuando a raiz)



B-Trees

Características (2)

Os nós internos das *B-trees* (excepto a raiz) têm, pelo menos, $t \geq 2$ filhos

t é o grau (de ramificação) mínimo de uma *B-tree*

A ordem de uma *B-tree* é $m = 2t$

Cada nó tem capacidade para $2t - 1$ elementos

Ocupação de um nó (excepto a raiz)

- entre $t - 1$ e $2t - 1$ elementos
- entre t e $2t$ filhos (excepto as folhas)

Ocupação da raiz (de uma *B-tree* não vazia)

- entre 1 e $2t - 1$ elementos
- entre 2 e $2t$ filhos (excepto se for folha)

B-Trees

Características (3)

Um nó interno com e elementos tem $e + 1$ filhos

Em todos os nós, verifica-se:

$$chave(\text{elemento}_1) \leq chave(\text{elemento}_2) \leq \ldots \leq chave(\text{elemento}_e)$$

Em todos os nós internos, verifica-se:

$$chaves(\text{filho}_1) \leq chave(\text{elemento}_1) \leq chaves(\text{filho}_2) \leq$$
$$\leq chave(\text{elemento}_2) \leq \ldots \leq chave(\text{elemento}_e) \leq chaves(\text{filho}_{e+1})$$

Todas as folhas estão à mesma profundidade

B-Trees

Implementação

Conteúdo de um nó	(campo)
ocupação	`n`
elementos $(2t-1)$	`key[1 .. 2t-1]`
filhos $(2t)$	`c[1 .. 2t]`
é-folha?	`leaf`

Um nó ocupa uma, duas páginas (do disco, do sistema de ficheiros, ...)

O valor de t depende do espaço ocupado pelos elementos e da dimensão pretendida para um nó

A raiz é mantida sempre em memória

B-TREE-CREATE(T)

```
1  x <- ALLOCATE-NODE()          // cria um novo nó
2  x.leaf <- TRUE                //   sem filhos
3  x.n <- 0                      //   nem elementos
4  DISK-WRITE(x)                 // e guarda-o em disco
5  T.root <- x                   // este nó é a raiz da
                                 // nova B-tree
```

(*Introduction to Algorithms*, Cormen et al.)

B-TREE-SEARCH(x, k)

```
1  i <- 1
2  while i <= x.n and k > x.key[i] do
3      i <- i + 1
4  if i <= x.n and k = x.key[i] then
5      return (x, i)
6  if x.leaf then
7      return NIL
8  DISK-READ(x.c[i])
9  return B-TREE-SEARCH(x.c[i], k)
```

Pesquisa (recursiva) do elemento com chave k na subárvore cuja raiz é o nó x

Assume que x já está em memória quando a função é chamada

Altura máxima de uma *B-tree*

Nível	Número mínimo de nós	Número mínimo de elementos
0	1	1
1	2	$2\,(t-1)$
2	$2t$	$2t\,(t-1)$
3	$2t^2$	$2t^2(t-1)$
4	$2t^3$	$2t^3\,(t-1)$
	$\vdots$	
h	$2t^{h-1}$	$2t^{h-1}\,(t-1)$

Número de elementos de uma árvore com altura h

$$n \geq 1+\sum_{i=0}^{h-1} 2t^i(t-1) = 1+2(t-1)\frac{1-t^h}{1-t} = 1-2(1-t^h) = 2t^h-1$$

Altura de uma árvore com n elementos

$$n \geq 2t^h - 1 \quad \equiv \quad t^h \leq \frac{n+1}{2} \quad \equiv \quad h \leq \log_t \frac{n+1}{2}$$

B-Trees

Comportamento da pesquisa

Altura de uma árvore com n elementos

$$h \leq \log_t \frac{n+1}{2} = O(\log_t n)$$

Número de nós acedidos no pior caso

$$O(h) = O(\log_t n)$$

Complexidade temporal da pesquisa no pior caso

$$O(t \log_t n)$$

Alturas de árvores

Elementos	abp	*B-tree*			
		$t = 32$		$t = 64$	
	mínima	mínima	máxima	mínima	máxima
10^6	19	3	3	2	3
10^9	29	4	5	4	4
10^{12}	39	6	7	5	6

B-TREE-INSERT(T, k)

```
 1 r <- T.root
 2 if r.n = 2t - 1 then              // se a raiz está cheia...
 3     s <- ALLOCATE-NODE()
 4     T.root <- s                   // cria uma nova raiz...
 5     s.leaf <- FALSE
 6     s.n <- 0
 7     s.c[1] <- r                   // de que a antiga é filho
 8     B-TREE-SPLIT-CHILD(s, 1)      // e explode a antiga raiz
 9     B-TREE-INSERT-NONFULL(s, k)
10 else
11     B-TREE-INSERT-NONFULL(r, k)
```

A inserção é efectuada numa única passagem pela árvore

B-TREE-SPLIT-CHILD(x, i)

```
 1 y <- x.c[i]                              // nó a explodir (filho i)
 2 z <- ALLOCATE-NODE()                     // novo filho i+1
 3 z.leaf <- y.leaf
 4 z.n <- t - 1
 5 for j <- 1 to t - 1 do                   // transfere metade dos
 6     z.key[j] <- y.key[j + t]             // elementos para o novo nó
 7 if not y.leaf then
 8     for j <- 1 to t do                   // e metade dos filhos
 9         z.c[j] <- y.c[j + t]
10 y.n <- t - 1
11 for j <- x.n + 1 downto i + 1 do         // abre espaço em x para o
12     x.c[j + 1] <- x.c[j]                 // novo filho
13 x.c[i + 1] <- z
14 for j <- x.n downto i do                 // abre espaço para o
15     x.key[j + 1] <- x.key[j]             // elemento a promover
16 x.key[i] <- y.key[t]
17 x.n <- x.n + 1
18 DISK-WRITE(y)
19 DISK-WRITE(z)
20 DISK-WRITE(x)
```

B-TREE-INSERT-NONFULL(x, k)

```
 1 i <- x.n
 2 if x.leaf then  // se está numa folha, insere o elemento
 3     while i >= 1 and k < x.key[i] do
 4         x.key[i + 1] <- x.key[i]
 5         i <- i - 1
 6     x.key[i + 1] <- k
 7     x.n <- x.n + 1
 8     DISK-WRITE(x)
 9 else              // senão, desce para o filho apropriado
10     while i >= 1 and k < x.key[i] do
11         i <- i - 1
12     i <- i + 1
13     DISK-READ(x.c[i])
14     if x.c[i].n = 2t - 1 then     // o filho está cheio?
15         B-TREE-SPLIT-CHILD(x, i)
16         if k > x.key[i] then
17             i <- i + 1
18     B-TREE-INSERT-NONFULL(x.c[i], k)
```

B-Trees — Remoção do elemento com chave k (1)

Remoção do elemento efectuada numa única passagem pela árvore

Se o nó corrente contém o elemento na posição i . . .

1. . . . e é uma folha

   Remove o elemento

2. . . . e é um nó interno

   a. se o filho i tem mais do que $t - 1$ elementos

      - substitui o elemento a remover pelo seu predecessor, que é removido da subárvore com raiz c_i

   b. senão, se o filho $i + 1$ tem mais do que $t - 1$ elementos

      - substitui o elemento a remover pelo seu sucessor, que é removido da subárvore com raiz c_{i+1}

   c. senão

      - funde os filhos i e $i + 1$
      - continua a partir do novo filho i (onde agora está o elemento a remover)

B-Trees — Remoção do elemento com chave k (2)

Se o nó corrente não contém o elemento

3. Se o nó corrente não é folha, seja i o índice do filho que é raiz da subárvore onde o elemento poderá estar

   Se o filho i tem mais do que $t - 1$ elementos

   - continua a partir do filho i

   Se o filho i tem $t - 1$ elementos

   a. se algum dos irmãos esquerdo ou direito de i tem mais do que $t - 1$ elementos
      - transfere um elemento para o filho i, por empréstimo de um irmão nessas condições
      - continua a partir do filho i

   b. senão
      - funde o filho i com o irmão esquerdo ou direito
      - continua a partir do nó que resultou da fusão

Se, terminada a remoção, a raiz fica vazia e não é folha

- o seu (único) filho passa a ser a nova raiz e a altura diminui

B-TREE-DELETE(T, k)

```
1 r <- T.root
2 B-TREE-DELETE-SAFE(r, k)          // remove o elemento
3 if r.n = 0 and not r.leaf then    // raiz vazia com filho?
4     r <- r.c[1]
5     FREE-NODE(T.root)
6     T.root <- r                   // o filho é a nova raiz
```

B-TREE-DELETE-SAFE(x, k)

```
 1 i <- 1
 2 while i <= x.n and k > x.key[i] do
 3     i <- i + 1
 4 if i <= x.n and k = x.key[i] then
 5     if x.leaf then
 6         B-TREE-DELETE-FROM-LEAF(x, i)                  // Caso 1
 7     else
 8         B-TREE-DELETE-FROM-INTERNAL-NODE(x, i)         // Caso 2
 9 else if not x.leaf then
10     B-TREE-DELETE-FROM-SUBTREE(x, i)                   // Caso 3
```

B-TREE-DELETE-FROM-LEAF(x, i)

```
1 for j <- i to x.n - 1 do                                // Caso 1
2     x.key[j] <- x.key[j + 1]                            // Caso 1
3 x.n <- x.n - 1                                          // Caso 1
4 DISK-WRITE(x)                                           // Caso 1
```

B-TREE-DELETE-FROM-INTERNAL-NODE(x, i)

```
 1 y <- x.c[i]
 2 DISK-READ(y)
 3 if y.n > t - 1 then                                  // Caso 2a
 4     x.key[i] <- B-TREE-DELETE-MAX(y)                 // Caso 2a
 5     DISK-WRITE(x)                                    // Caso 2a
 6 else
 7     z <- x.c[i + 1]
 8     DISK-READ(z)
 9     if z.n > t - 1 then                              // Caso 2b
10         x.key[i] <- B-TREE-DELETE-MIN(z)             // Caso 2b
11         DISK-WRITE(x)                                // Caso 2b
12     else
13         B-TREE-MERGE-CHILDREN(x, i)                  // Caso 2c
14         B-TREE-DELETE-SAFE(x.c[i], k)                // Caso 2c
```

B-TREE-DELETE-FROM-SUBTREE(x, i)

```
 1 y <- x.c[i]
 2 DISK-READ(y)
 3 if y.n = t - 1 then
 4     borrowed <- FALSE
 5     if i > 1 then
 6         z <- x.c[i - 1]
 7         DISK-READ(z)
 8         if z.n > t - 1 then                                  // Caso 3a
 9             B-TREE-BORROW-FROM-LEFT-SIBLING(x, i)            // Caso 3a
10             borrowed <- TRUE                                 // Caso 3a
11         else
12             m <- i - 1
13     if not borrowed and i <= x.n then
14         z <- x.c[i + 1]
15         DISK-READ(z)
16         if z.n > t - 1 then                                  // Caso 3a
17             B-TREE-BORROW-FROM-RIGHT-SIBLING(x, i)           // Caso 3a
18             borrowed <- TRUE                                 // Caso 3a
19         else
20             m <- i
21     if not borrowed then                                     // Caso 3b
22         B-TREE-MERGE-CHILDREN(x, m)                          // Caso 3b
23         y <- x.c[m]                                          // Caso 3b
24 B-TREE-DELETE-SAFE(y, k)
```

B-TREE-MERGE-CHILDREN(x, i)

```
 1 y <- x.c[i]                          // fusão do filho i
 2 z <- x.c[i + 1]                      // com o i+1
 3 y.key[t] <- x.key[i]
 4 for j <- 1 to t - 1 do               // muda conteúdo de
 5     y.key[t + j] <- z.key[j]         // c[i+1] para c[i]
 6 if not y.leaf then
 7     for j <- 1 to t do               // incluindo filhos
 8         y.c[t + j] <- z.c[j]
 9 y.n <- 2t - 1                        // c[i] fica cheio
10 for j <- i + 1 to x.n do
11   x.key[j - 1] <- x.key[j]
12 for j <- i + 2 to x.n + 1 do
13   x.c[j - 1] <- x.c[j]
14 x.n <- x.n - 1
15 FREE-NODE(z)                    // apaga c[i+1] antigo
16 DISK-WRITE(y)
17 DISK-WRITE(x)
```

(NOTA: Os nós `x`, `x.c[i]` e `x.c[i + 1]` já foram lidos para memória)

B-TREE-BORROW-FROM-LEFT-SIBLING(x, i)

```
 1 y <- x.c[i]                          // irmão esquerdo
 2 z <- x.c[i - 1]                      // do nó i é o i-1
 3 for j <- t - 1 downto 1 do           // abre espaço para
 4     y.key[j + 1] <- y.key[j]         // a nova 1ª chave
 5 y.key[1] <- x.key[i - 1]
 6 x.key[i - 1] <- z.key[z.n]
 7 if not y.leaf then
 8     for j <- t downto 1 do           // abre espaço para
 9         y.c[j + 1] <- y.c[j]         // o novo 1º filho
10     y.c[1] <- z.c[z.n + 1]
11 y.n <- t
12 z.n <- z.n - 1
13 DISK-WRITE(z)
14 DISK-WRITE(y)
15 DISK-WRITE(x)
```

(NOTA: Os nós `x`, `x.c[i - 1]` e `x.c[i]` já foram lidos para memória)

B-TREE-BORROW-FROM-RIGHT-SIBLING(x, i)

Exercício

B-TREE-DELETE-MAX(x)

Exercício

(o nó x tem mais do que $t - 1$ elementos;
a função devolve o elemento removido)

B-TREE-DELETE-MIN(x)

Exercício

(o nó x tem mais do que $t - 1$ elementos;
a função devolve o elemento removido)

B-Trees

Resumo

Árvore com grau de ramificação mínimo t e com n elementos

Altura $\quad h = O(\log_t n)$

Complexidade temporal das operações

pesquisa, inserção, remoção

$$O(th) = O(t \log_t n)$$

Número de nós acedidos (nas operações acima)

$$O(h) = O(\log_t n)$$

Tries

A estrutura de dados *trie*

Uma *trie* é uma árvore cujos nós têm filhos que correspondem a símbolos do alfabeto das chaves

Uma chave está contida numa *trie* se o percurso que ela induz na *trie*, a partir da sua raiz, termina num nó que marca o fim de uma chave

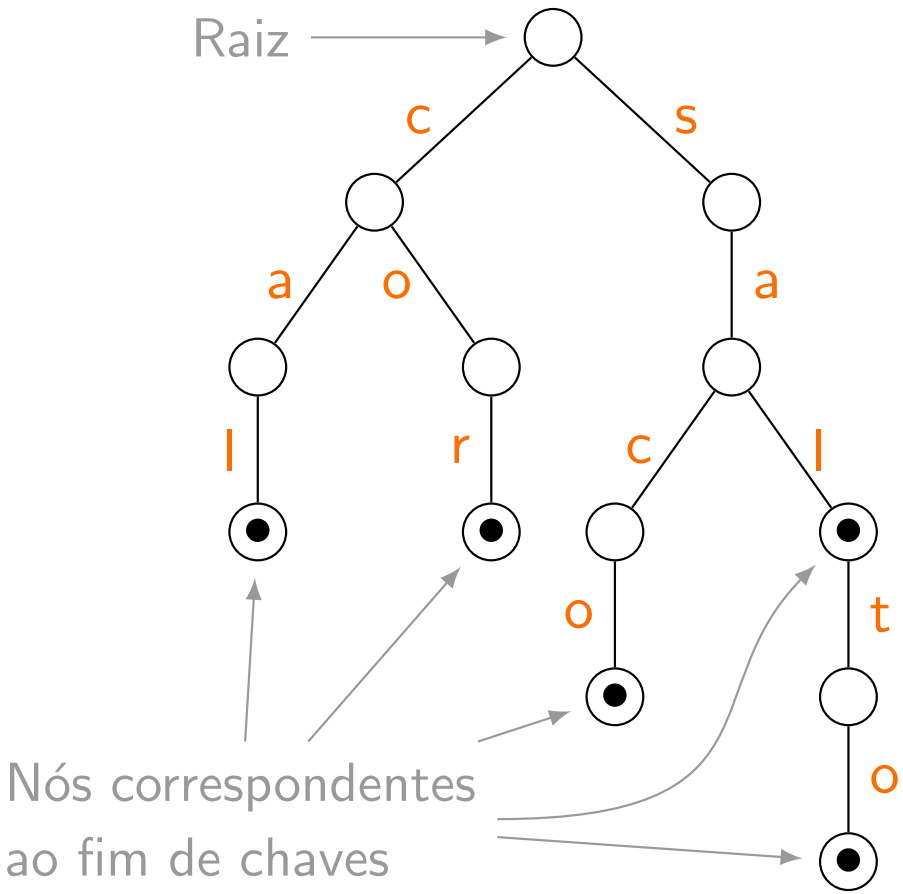
As *tries* apresentam algumas características que as distinguem de outras estruturas de dados

1. A complexidade das operações não depende do número de elementos que ela contém
2. As chaves não têm de estar explicitamente contidas na *trie*
3. As operações não se baseiam em comparações entre chaves

Uma *trie*

Exemplo

Representação de uma *trie* com as chaves (palavras) cal, cor, saco, sal e salto



Tries

d – dimensão do alfabeto (nº de símbolos distintos)

Chaves

k[1..m] – chave

|k| = m

Conteúdo dos nós (implementação com vector de filhos)

c[1..d] – filhos

p – pai (opcional)

word – TRUE sse a chave que termina no nó está na *trie*

ou

element – elemento associado à chave que termina(ria) no nó

TRIE-SEARCH(T, k)

```
1 x <- T.root
2 i <- 1
3 while x != NIL and i <= |k| do
4     x <- x.c[k[i]]
5     i <- i + 1
6 return x != NIL and x.word
```

Argumentos

T – *trie*

k – chave (palavra)

TRIE-SEARCH(T, k) — Complexidade

```
1 x <- T.root
2 i <- 1
3 while x != NIL and i <= |k| do
4     x <- x.c[k[i]]
5     i <- i + 1
6 return x != NIL and x.word
```

Análise da complexidade para o pior caso

- Linhas 1, 2, 4, 5 e 6, e testes da linha 3: custo constante

$$O(1) + O(1) + (m+1)O(1) + m\,O(1) + m\,O(1) + O(1) \;=$$
$$4\,O(1) + 3m\,O(1) \;=\; 3\,O(m) \;=$$
$$O(m)$$

TRIE-INSERT(T, k)

```
1 if T.root = NIL then
2     T.root <- ALLOCATE-NODE()
3     T.root.p <- NIL
4 x <- T.root
5 i <- 1
6 while i <= |k| and x.c[k[i]] != NIL do
7     x <- x.c[k[i]]
8     i <- i + 1
9 TRIE-INSERT-REMAINING(x, k, i)
```

ALLOCATE-NODE() devolve um novo nó da *trie* com

c[1..d] = NIL
p = NIL
word = FALSE

TRIE-INSERT-REMAINING(x, k, i)

```
1 y <- x
2 for j <- i to |k| do
3     y.c[k[j]] <- ALLOCATE-NODE()
4     y.c[k[j]].p <- y
5     y <- y.c[k[j]]
6 y.word <- TRUE
```

Função que acrescenta, a partir do nó x, os nós necessários para incluir na *trie* o sufixo da chave k que ainda não está na *trie* e que começa no i-ésimo símbolo da chave

Se $i > |k|$, só afecta a marca de fim de palavra no nó x

TRIE-DELETE(T, k) (1)

```
1 x <- T.root
2 i <- 1
3 while x != NIL and i <= |k| do
4     x <- x.c[k[i]]
5     i <- i + 1
6 if x != NIL and x.word then
7     x.word <- FALSE          // k deixa de estar na trie
8 ...
```

Falta remover os nós da *trie* que deixam de ter um papel útil, por não corresponderem ao fim de uma palavra nem terem filhos

TRIE-DELETE(T, k) (2)

```
 5 ...
 6 if x != NIL and x.word then
 7     x.word <- FALSE          // k deixa de estar na trie
 8     repeat
 9         i <- i - 1
10         childless <- TRUE              // x tem filhos?
11         j <- 1
12         while j <= d and childless do
13             if x.c[j] != NIL then
14                 childless <- FALSE
15             j <- j + 1
16         if childless then      // se não tem, é apagado
17             y <- x.p
18             if y = NIL then
19                 T.root <- NIL     // a trie ficou vazia
20             else
21                 y.c[k[i]] <- NIL
22             FREE-NODE(x)
23           x <- y
24     until x = NIL or not childless or x.word
```

Complexidade temporal das operações sobre uma *trie*

Implementação com vector de filhos — Resumo

Pesquisa da palavra k

$$O(m)$$

Inserção da palavra k

$$O(m)$$

Remoção da palavra k

$$O(m\,d)$$

Complexidade espacial

$$O(n\,w\,d)$$

Onde

- $m = |\mathrm{k}|$
- d é o número de símbolos do alfabeto
- n é o número de palavras na *trie*
- w é comprimento médio das palavras na *trie*

Programação dinâmica

Programação dinâmica

Método usado na construção de soluções iterativas para problemas cuja solução recursiva tem uma complexidade elevada (exponencial, em geral)

Aplica-se, normalmente, a problemas de optimização

- Um problema de optimização é um problema em que se procura minimizar ou maximizar algum valor associado às suas soluções

Corte de varas

Uma empresa compra varas de aço, corta-as e vende-as aos pedaços

O preço de venda de cada pedaço depende do seu comprimento

Problema

Como cortar uma vara de comprimento n de forma a maximizar o valor de venda?

Comprimento i	1	2	3	4	5	6	7	8	9	10
Preço p_i	1	5	7	11	11	17	20	20	24	27

Corte de varas

Caracterização de uma solução óptima (1)

Soluções possíveis, para uma vara de comprimento 10

- Um corte de comprimento 1, mais as soluções para uma vara de comprimento 9
- Um corte de comprimento 2, mais as soluções para uma vara de comprimento 8
- Um corte de comprimento 3, mais as soluções para uma vara de comprimento 7

...

- Um corte de comprimento 9, mais as soluções para uma vara de comprimento 1
- Um corte de comprimento 10, mais as soluções para uma uma vara de comprimento 0

Qual a melhor?

Corte de varas

Caracterização de uma solução óptima (2)

Sejam os tamanhos dos cortes possíveis

$$1, 2, \ldots, n$$

com preços

$$p_1, p_2, \ldots, p_n$$

O maior valor de venda de uma vara de comprimento n é o máximo que se obtém

- fazendo um corte inicial de comprimento $1 \leq i \leq n$, de valor p_i, somado com
- o maior valor de venda de uma vara de comprimento $n - i$

Corte de varas

Função recursiva

Corte de uma vara de comprimento n

Tamanho dos cortes: $i = 1, \ldots, n$
Preços: $P = (p_1\ p_2\ \ldots\ p_n)$

$r_P[0..n]$: função t.q. $r_P[l]$ é o maior preço que se pode obter para uma vara de comprimento l, dados os preços P

$$r_P[l] = \begin{cases} 0 & \text{se} \quad l = 0 \\ \max\limits_{1 \leq i \leq l} \{p_i + r_P[l-i]\} & \text{se} \quad l > 0 \end{cases}$$

Preço máximo (chamada inicial da função): $r_P[n]$

Corte de varas

Implementação recursiva

CUT-ROD(p, l)

```
1 if l = 0 then
2   return 0
3 q <- −∞
4 for i <- 1 to l do
5   q <- max(q, p[i] + CUT-ROD(p, l - i))
6 return q
```

Argumentos

- p Preços das varas de comprimentos $\{1, 2, \ldots, n\}$
- l Comprimento da vara a cortar

Chamada inicial da função: CUT-ROD(p, n)

Corte de varas

Alguns números

Número de cortes possíveis

$$2^{n-1}$$

Exemplo ($n = 4$)

$4 \quad 1+3 \quad 2+2 \quad 3+1$

$1+1+2 \quad 1+2+1 \quad 2+1+1 \quad 1+1+1+1$

Número de cortes distintos possíveis

$$O\left(\frac{e^{\pi\sqrt{\frac{2n}{3}}}}{4n\sqrt{3}}\right)$$

Exemplo ($n = 4$)

$4 \quad 1+3 \quad 2+2 \quad 1+1+2 \quad 1+1+1+1$

Corte de varas

Implementação recursiva com *memoização*

MEMOIZED-CUT-ROD(p, n)

```
1 let r[0..n] be a new array
2 for l <- 0 to n do
3       r[l] <- −∞
4 return MEMOIZED-CUT-ROD-2(p, n, r)
```

MEMOIZED-CUT-ROD-2(p, l, r)

```
1 if r[l] = −∞ then
2   if l = 0 then
3     q <- 0
4   else
5     q <- −∞
6     for i <- 1 to l do
7       q <- max(q, p[i] + MEMOIZED-CUT-ROD-2(p, l - i, r))
8   r[l] <- q
9 return r[l]
```

NB: isto não é programação dinâmica

Corte de varas

Cálculo iterativo de $r[n]$ (1)

p_i	1	5	7	11	11	17	20	20	24	27

Preenchimento do vector r

	0	1	2	3	4	5	6	7	8	9	10
$r[i]$	0	1	5	7	11	12	17	20	22	25	28

1. Caso base: $r[0] \leftarrow 0$
2. $r[1] \leftarrow \max\{\, p_1 + r[0] \,\} = \max\{\, 1 + 0 \,\}$
3. $r[2] \leftarrow \max\{\, p_1 + r[1], p_2 + r[0] \,\} = \max\{\, 1 + 1, 5 + 0 \,\}$
4. $r[3] \leftarrow \max\{\, p_1 + r[2], p_2 + r[1], p_3 + r[0] \,\} = \max\{\, 1 + 5, 5 + 1, 7 + 0 \,\}$

   ...

11. $r[10] \leftarrow \max\{\, p_1 + r[9], p_2 + r[8], \ldots, p_4 + r[6], \ldots, p_{10} + r[0] \,\}$

Corte de varas

Cálculo iterativo de $r[n]$ (2)

BOTTOM-UP-CUT-ROD(p, n)

```
1 let r[0..n] be a new array
2 r[0] <- 0
3 for l <- 1 to n do
4   q <- −∞
5   for i <- 1 to l do
6     q <- max(q, p[i] + r[l - i])
7   r[l] <- q
8 return r[n]
```

Corte de varas

Complexidade

Complexidade de BOTTOM-UP-CUT-ROD($p_1\ p_2\ \ldots\ p_n$)

Ciclo 3–7 é executado n vezes

Ciclo 5–6 é executado l vezes, $l = 1, \ldots, n$

$$1 + 2 + \ldots + n = \sum_{l=1}^{n} l = \frac{n(n+1)}{2} = \Theta(n^2)$$

Todas as operações têm custo constante

Complexidade temporal $\Theta(n^2)$

Complexidade espacial $\Theta(n)$

Corte de varas

Construção da solução (1)

O valor máximo por que é possível vender uma vara é calculado pela função BOTTOM-UP-CUT-ROD

Quais os cortes a fazer para obter esse valor?

Para o preenchimento da posição `l` do vector `r[]`, é escolhido o valor máximo de `p[i] + r[l - i]`

O facto de o valor máximo incluir a parcela `p[i]` significa a inclusão de um pedaço de vara de comprimento `i`

Logo, o valor máximo por que é possível vender uma vara de comprimento `l` (vector `s[]`) será obtido

- com um pedaço de comprimento `i` e
- o valor máximo por que é possível vender uma vara de comprimento `l - i`

Corte de varas

Construção da solução (2)

p_i	1	5	7	11	11	17	20	20	24	27

	0	1	2	3	4	5	6	7	8	9	10
$r[i]$	0	1	5	7	11	12	17	20	22	25	28
$s[i]$		1	2	3	4	1	6	7	2	2	4

1. Caso base: $r[0] \leftarrow 0$
2. $r[1] \leftarrow \max\{\, p_1 + r[0] \,\} = \max\{\, 1 + 0 \,\}$, $s[1] \leftarrow 1$
3. $r[2] \leftarrow \max\{\, p_1 + r[1], p_2 + r[0] \,\} = \max\{\, 1 + 1, 5 + 0 \,\}$, $s[2] \leftarrow 2$
4. $r[3] \leftarrow \max\{\, p_1 + r[2], p_2 + r[1], p_3 + r[0] \,\} =$
   $= \max\{\, 1 + 5, 5 + 1, 7 + 0 \,\}$, $s[3] \leftarrow 3$

   ...

11. $r[10] \leftarrow \max\{\, p_1 + r[9], p_2 + r[8], \ldots, p_4 + r[6], \ldots, p_{10} + r[0] \,\}$,
    $s[10] \leftarrow 4$

Corte de varas

Construção da solução (3)

$s[1..n]$: $s[l]$ é o primeiro corte a fazer numa vara de comprimento l

EXTENDED-BOTTOM-UP-CUT-ROD(p, n)

```
 1 let r[0..n] and s[1..n] be new arrays
 2 r[0] <- 0
 3 for l <- 1 to n do
 4   q <- -∞
 5   for i <- 1 to l do
 6     if q < p[i] + r[l - i] then
 7       q <- p[i] + r[l - i]
 8       s[l] <- i           // corte feito na posição i
 9   r[l] <- q
10 return r and s
```

Corte de varas

Resolução completa

PRINT-CUT-ROD-SOLUTION(p, n)

```
1 (r, s) <- EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
2 print "The best price is ", r[n]
3 while n > 0 do
4   print s[n]
5   n <- n - s[n]
```

Programação dinâmica

Condições de aplicabilidade

A programação dinâmica aplica-se a problemas que apresentam as características seguintes:

Subestrutura óptima (*Optimal substructure*)

- Um problema tem subestrutura óptima se uma sua solução óptima é construída com recurso a soluções óptimas de subproblemas

Subproblemas repetidos (*Overlapping subproblems*)

- Existem subproblemas repetidos quando os subproblemas de um problema têm subproblemas em comum

Programação dinâmica

Aplicação

1. Caracterização de uma solução óptima
2. Formulação recursiva do cálculo do valor de uma solução óptima
3. Cálculo iterativo do valor de uma solução óptima, por tabelamento
4. Construção de uma solução óptima

Produto de matrizes

Cálculo do produto de uma sequência de matrizes (*Matrix-chain multiplication*)

Problema

Dada uma sequência de matrizes a multiplicar

$$A_1 \, A_2 \, \ldots \, A_n, \quad n > 0$$

com dimensões

$$p_0 \times p_1 \quad p_1 \times p_2 \quad \ldots \quad p_{n-1} \times p_n$$

por que ordem efectuar os produtos de modo a minimizar o número de multiplicações entre elementos das matrizes?

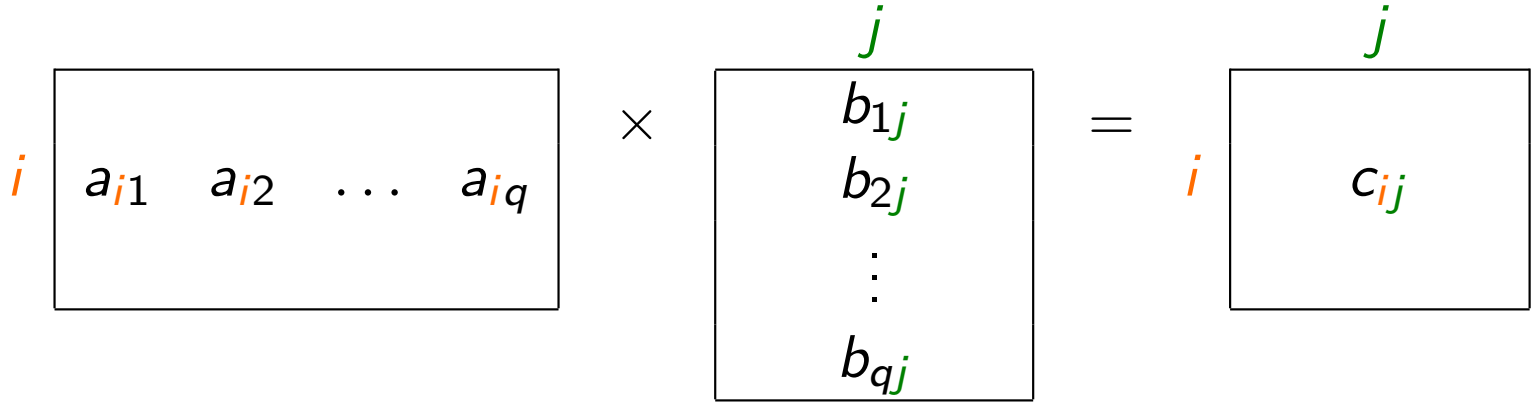
(NOTA: A matriz A_i tem dimensão $p_{i-1} \times p_i$)

(NOTA: O produto de matrizes é uma operação associativa.)

Produto de matrizes

Cálculo do produto de duas matrizes (1)

$$A\ (p \times q) \quad \times \quad B\ (q \times r) \quad = \quad C\ (p \times r)$$



$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \ldots + a_{iq}b_{qj} = \sum_{k=1}^{q} a_{ik}b_{kj}$$

No cálculo de cada elemento de C, são efectuadas q multiplicações (escalares)

Produto de matrizes

Cálculo do produto de duas matrizes (2)

MATRIX-MULTIPLY(A[1..p, 1..q], B[1..q, 1..r])

```
1 let C[1..p,1..r] be a new matrix
2 for i <- 1 to p do
3   for j <- 1 to r do
4     C[i,j] <- 0
5     for k <- 1 to q do
6       C[i,j] <- C[i,j] + A[i,k] * B[k,j]
7 return C
```

Número de multiplicações

Se A e B são matrizes com dimensões $p \times q$ e $q \times r$, respectivamente, no cálculo de $C = AB$, o número de multiplicações efectuadas entre elementos das matrizes é

$$p \times q \times r$$

(C tem $p \times r$ elementos e são efectuadas q multiplicações para o cálculo de cada um)

Produto de uma sequência de matrizes

Exemplo

Sejam A_1, A_2 e A_3 matrizes com dimensões

$$10 \times 100,\ 100 \times 5 \text{ e } 5 \times 50$$

Ordens de avaliação possíveis para o produto $A_1 A_2 A_3$

$$(A_1 A_2) A_3$$
$$A_1 (A_2 A_3)$$

Número de multiplicações

$(A_1 A_2) A_3$

$$10 \times 100 \times 5 + 10 \times 5 \times 50 = 5000 + 2500 = 7500$$

$A_1 (A_2 A_3)$

$$100 \times 5 \times 50 + 10 \times 100 \times 50 = 25000 + 50000 = 75000$$

Produto de uma sequência de matrizes

Colocação de parêntesis

Formulação alternativa

Como colocar parêntesis no produto $A_1 A_2 \ldots A_n$ de modo a realizar o menor número de multiplicações possível?

Número de colocações de parêntesis distintas

$$\Omega\left(\frac{4^n}{n^{\frac{3}{2}}}\right)$$

Produto de uma sequência de matrizes

Caracterização de uma solução óptima (1)

O produto $A_1 A_2 \ldots A_n$ será calculado de uma das formas

$$
\begin{array}{l}
A_1 \, (A_2 \ldots A_n) \\
(A_1 A_2) \, (A_3 \ldots A_n) \\
(A_1 \ldots A_3) \, (A_4 \ldots A_n) \\
\qquad \vdots \\
(A_1 \ldots A_{n-2}) \, (A_{n-1} A_n) \\
(A_1 \ldots A_{n-1}) \, A_n
\end{array}
$$

O número n-mult de multiplicações a efectuar para o cálculo de

$$(A_1 \ldots A_k) \, (A_{k+1} \ldots A_n)$$

para qualquer $1 \leq k < n$, será

$$\text{n-mult}(A_1 \ldots A_k) + \text{n-mult}(A_{k+1} \ldots A_n) + p_0 \, p_k \, p_n$$

Produto de uma sequência de matrizes

Caracterização de uma solução óptima (2)

Procura-se o valor mínimo de

$$\text{n-mult}(A_1 \ldots A_n)$$

que depende do valor mínimo de

$$\text{n-mult}(A_1 \ldots A_k) \quad \text{e de} \quad \text{n-mult}(A_{k+1} \ldots A_n)$$

para algum valor de k

O número mínimo m de multiplicações a efectuar será obtido para o valor de k que minimiza

$$m(A_1 \ldots A_k) + m(A_{k+1} \ldots A_n) + p_0\, p_k\, p_n$$

Produto de uma sequência de matrizes

Função recursiva

Sequência de matrizes a multiplicar

$$A_1\, A_2\, \ldots\, A_n, \quad n > 0$$

Dimensões das matrizes: $P = (p_0\, p_1 \ldots p_n)$

$m_P[1..n, 1..n]$: $m_P[i, j]$ é o menor número de multiplicações a fazer para o cálculo do produto $A_i \ldots A_j$

$$m_P[i,j] = \begin{cases} 0 & \text{se} \quad i = j \\ \min\limits_{i \le k < j} \{m_P[i,k] + m_P[k+1,j] + p_{i-1} p_k p_j\} & \text{se} \quad i < j \end{cases}$$

Número mínimo de multiplicações (chamada inicial): $m_P[1, n]$

Produto de uma sequência de matrizes

Cálculo de $m[i, j]$

m

	1	2	3	4	5
1	0	m_{12}	m_{13}	m_{14}	m_{15}
2		0	m_{23}	m_{24}	m_{25}
3			0	m_{34}	m_{35}
4				0	m_{45}
5					0

Ordem de cálculo

1. Sequências de comprimento 1: m_{11}, m_{22}, m_{33}, m_{44}, m_{55} (Caso base)
2. Sequências de comprimento 2: m_{12}, m_{23}, m_{34}, m_{45}
3. Sequências de comprimento 3: m_{13}, m_{24}, m_{35}
4. Sequências de comprimento 4: m_{14}, m_{25}
5. Sequências de comprimento 5: m_{15}

Produto de uma sequência de matrizes

Cálculo iterativo de $m[1, n]$

MATRIX-CHAIN-ORDER(p)

```
 1 n <- |p| - 1                    // p[0..n]
 2 let m[1..n,1..n] be a new table
 3 for i <- 1 to n do
 4      m[i, i] <- 0
 5 for l <- 2 to n do              // l is the chain length
 6     for i <- 1 to n - l + 1 do
 7         j <- i + l - 1
 8         m[i, j] <- +∞
 9         for k <- i to j - 1 do
10             q <- m[i, k] + m[k + 1, j] +
                      p[i - 1] * p[k] * p[j]
11             if q < m[i, j] then
12                 m[i, j] <- q
13 return m[1, n]
```

Produto de uma sequência de matrizes

Complexidade de MATRIX-CHAIN-ORDER($p_0\ p_1\ \ldots\ p_n$)

Ciclo 3–4 é executado n vezes

Ciclo 5–12 é executado $n-1$ vezes (variável l)

Ciclo 6–12 é executado $n-l+1$ vezes (variável i)

Ciclo 9–12 é executado $l-1$ vezes (variável k)

$$\sum_{l=2}^{n}\sum_{i=1}^{n-l+1}\sum_{k=i}^{i+l-2} 1 = \sum_{l=2}^{n}\sum_{i=1}^{n-l+1} l-1 = \sum_{l=2}^{n}(n-(l-1))(l-1) = \sum_{l=1}^{n-1}(n-l)l =$$

$$n\sum_{l=1}^{n-1} l - \sum_{l=1}^{n-1} l^2 = n\frac{(n-1)n}{2} - \frac{(n-1)n(2n-1)}{6} = \frac{n^3-n}{6} = \Theta(n^3)$$

Complexidade temporal $\Theta(n^3)$

Complexidade espacial $\Theta(n^2)$

Produto de uma sequência de matrizes

Construção da solução

MATRIX-CHAIN-ORDER(p)

```
 1 n <- |p| - 1                    // p[0..n]
 2 let m[1..n,1..n] and s[1..n-1,2..n] be new tables
 3 for i <- 1 to n do
 4      m[i, i] <- 0
 5 for l <- 2 to n do              // l is the chain length
 6     for i <- 1 to n - l + 1 do
 7         j <- i + l - 1
 8         m[i, j] <- +∞
 9         for k <- i to j - 1 do
10             q <- m[i, k] + m[k + 1, j] +
                    p[i - 1] * p[k] * p[j]
11             if q < m[i, j] then
12                 m[i, j] <- q
13                 s[i, j] <- k  // break at matrix k
14 return m and s
```

Produto de uma sequência de matrizes

Solução calculada

$$p = 10 \quad 100 \quad 5 \quad 50 \quad 3$$

Matriz `m` (multiplicações)

	1	**2**	**3**	**4**
1	0	5000	7500	5250
2		0	25000	2250
3			0	750
4				0

Matriz `s` (separação)

	2	**3**	**4**
1	1	2	1
2		2	2
3			3

Número mínimo de multiplicações para calcular ...

$$\begin{aligned}
A_1A_2 &= 5000 \\
A_2A_3 &= 25000 \\
A_1A_2A_3 &= 7500 \\
A_2A_3A_4 &= 2250 \\
A_1A_2A_3A_4 &= 5250
\end{aligned}$$

Separação dos produtos

$$\begin{aligned}
A_1 \ldots A_2 &= (A_1)(A_2) \\
A_1 \ldots A_3 &= (A_1A_2)(A_3) \\
A_2 \ldots A_4 &= (A_2)(A_3A_4) \\
A_1 \ldots A_4 &= (A_1)(A_2 \ldots A_4) \\
&= (A_1)(A_2(A_3A_4))
\end{aligned}$$

Produto de uma sequência de matrizes

Melhor colocação de parêntesis

$s[1..n-1, 2..n]$: $s[i, j]$ é a posição onde a sequência $A_i \ldots A_j$ é dividida: $(A_i \ldots A_{s[i,j]})(A_{s[i,j]+1} \ldots A_j)$

PRINT-OPTIMAL-PARENS(s, i, j)

```
1 if i = j then
2   print "A"i
3 else
4   print "("
5   PRINT-OPTIMAL-PARENS(s, i, s[i, j])
6   PRINT-OPTIMAL-PARENS(s, s[i, j] + 1, j)
7   print ")"
```

Sequências e subsequências

Seja x a sequência

$$x_1\ x_2\ \ldots\ x_m, \quad m \geq 0$$

A sequência $z = z_1\ z_2 \ldots z_k$ é uma subsequência de x se

$$z_j = x_{i_j}, \quad j = 1, \ldots, k \quad \text{e} \quad i_j < i_{j+1}$$

Exemplo

$$x = \text{A}\ \text{B}\ \text{C}\ \text{B}\ \text{D}\ \text{A}\ \text{B}$$

São subsequências de x:

A A B C B B B C B A B C B A
A B C B D A B *a sequência vazia* ...

Não são subsequências de x:

A A A D C E

Subsequências comuns

Sejam x e y as sequências

$$x_1\ x_2\ \ldots\ x_m \quad \text{e} \quad y_1\ y_2\ \ldots\ y_n, \quad m, n \geq 0$$

A sequência z é uma subsequência comum a x e y se

- z é uma subsequência de x e
- z é uma subsequência de y

Exemplo

$$x = \text{A B C B D A B}$$
$$y = \text{B D C A B A}$$

Subsequências comuns a x e a y

A A B C B A ...

Maiores subsequências comuns a x e a y

B C A B B C B A B D A B

Maior subsequência comum

Longest common subsequence

Problema

Dadas duas sequências x e y

$$x_1\ x_2\ \ldots\ x_m \quad \text{e} \quad y_1\ y_2\ \ldots\ y_n, \quad m, n \geq 0$$

determinar uma maior subsequência comum a x e a y

Número de subsequências de uma sequência de comprimento m

$$2^m$$

Maior subsequência comum

Caracterização de uma solução óptima

$$x = x_1\ x_2\ \ldots\ x_m \quad \text{e} \quad y = y_1\ y_2\ \ldots\ y_n, \quad m, n > 0$$

- $x_m = y_n$

Uma maior subsequência comum a x e y será uma maior subsequência comum a

$$x_1\ x_2\ \ldots\ x_{m-1} \quad \text{e} \quad y_1\ y_2\ \ldots\ y_{n-1}$$

acrescida de x_m

- $x_m \neq y_n$

Uma maior subsequência comum a x e y será uma maior de entre as maiores subsequências comuns a

$$x_1\ x_2\ \ldots\ x_m \quad \text{e} \quad y_1\ y_2\ \ldots\ y_{n-1}$$

e as maiores subsequências comuns a

$$x_1\ x_2\ \ldots\ x_{m-1} \quad \text{e} \quad y_1\ y_2\ \ldots\ y_n$$

Maior subsequência comum

Função recursiva

Comprimento de uma maior subsequência comum às sequências

$$x = x_1\, x_2\, \ldots\, x_m \quad \text{e} \quad y = y_1\, y_2\, \ldots\, y_n, \quad m, n \geq 0$$

$c_{xy}[0..m, 0..n]$: $c_{xy}[i, j]$ é o comprimento das maiores subsequências comuns a $x_1 \ldots x_i$ e $y_1 \ldots y_j$

$$c_{xy}[i,j] = \begin{cases} 0 & \text{se } i = 0 \vee j = 0 \\ 1 + c_{xy}[i-1, j-1] & \text{se } i, j > 0 \wedge x_i = y_j \\ \max\{c_{xy}[i-1, j], c_{xy}[i, j-1]\} & \text{se } i, j > 0 \wedge x_i \neq y_j \end{cases}$$

Comprimento de uma maior subsequência comum a x e y: $c_{xy}[m, n]$

Maior subsequência comum

Tabela para $c_{xy}[i, j]$

A função c_{xy} tem dois argumentos, logo, os valores da função serão guardados numa matriz

Valores possíveis para i

- O valor inicial é $m \geq 0$
- Nas chamadas recursivas, o valor do primeiro argumento mantém-se ou diminui em 1 unidade
- O caso base é atingido quando i é 0

Valores possíveis para j

- O valor inicial é $n \geq 0$
- Nas chamadas recursivas, o valor do segundo argumento mantém-se ou diminui em 1 unidade
- O caso base é atingido quando j é 0

A tabela terá índices $(0..m) \times (0..n)$

Maior subsequência comum

Tabulação de $c_{xy}[i, j]$

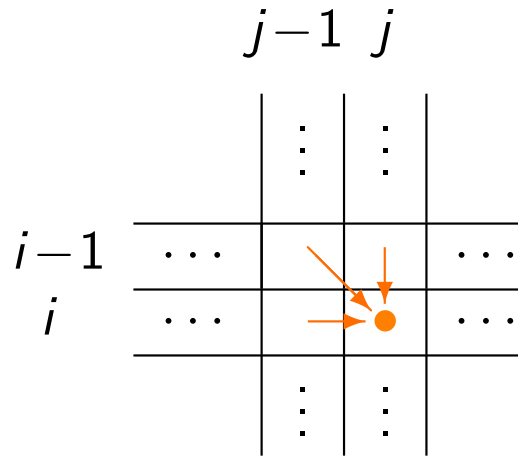
Caso base

Se $i = 0$ ou $j = 0$, então $c[i, j] = 0$

Casos recursivos

O valor de $c[i, j]$ depende:

- Do valor de $c[i - 1, j - 1]$ ou
- Dos valores de $c[i - 1, j]$ e de $c[i, j - 1]$



Para garantir que os valores necessários já estão calculados:

- As linhas são calculadas da linha 1 à linha m
- Dentro de cada linha, os valores são calculados da coluna 1 à coluna n

Argumentos da função iterativa

Os parâmetros do problema, que são as sequências x e y

Maior subsequência comum

Cálculo iterativo de $c[m, n]$

LONGEST-COMMON-SUBSEQUENCE-LENGTH(x, y)

```
 1 m <- |x|
 2 n <- |y|
 3 let c[0..m, 0..n] be a new table
 4 for i <- 1 to m do                    // caso base (j = 0)
 5   c[i, 0] <- 0
 6 for j <- 0 to n do                    // caso base (i = 0)
 7   c[0, j] <- 0
 8 for i <- 1 to m do                    // linhas 1 a m
 9   for j <- 1 to n do                  // colunas 1 a n
10     if x[i] = y[j] then
11       c[i, j] = 1 + c[i - 1, j - 1]
12     else if c[i - 1, j] >= c[i, j - 1] then
13       c[i, j] = c[i - 1, j]
14     else
15       c[i, j] = c[i, j - 1]
16 return c[m, n]
```

Maior subsequência comum

Análise da complexidade

LONGEST-COMMON-SUBSEQUENCE-LENGTH($x_1 \ldots x_m$, $y_1 \ldots y_n$)

Todas as operações executadas têm custo constante

Ciclo 4–5 é executado m vezes

Ciclo 6–7 é executado $n + 1$ vezes

Ciclo 8–15 é executado m vezes

Ciclo 9–15 é executado n vezes em cada iteração do ciclo 8–15

Complexidade temporal $\Theta(mn)$

Complexidade espacial $\Theta(mn)$

Maior subsequência comum

Construção da sequência

Para identificar a maior subsequência comum, basta saber se a posição a partir da qual o valor de $c[i, j]$ foi calculado foi

$$\underset{\text{(NW)}}{c[i-1, j-1]} \quad \text{ou} \quad \underset{\text{(N)}}{c[i-1, j]} \quad \text{ou} \quad \underset{\text{(W)}}{c[i, j-1]}$$

Se foi a partir de $c[i-1, j-1]$, o símbolo $x_i = y_j$ é um elemento da subsequência

É possível reconstruir a maior subsequência comum calculada, seguindo as direcções NW, N e W, partindo da posição $[m, n]$ e até chegar à linha ou à coluna 0

Maior subsequência comum

Construção da solução

LONGEST-COMMON-SUBSEQUENCE(x, y)

```
 1 m <- |x|
 2 n <- |y|
 3 let c[0..m, 0..n] and b[1..m, 1..n] be new tables
 4 for i <- 1 to m do
 5   c[i, 0] <- 0
 6 for j <- 0 to n do
 7   c[0, j] <- 0
 8 for i <- 1 to m do
 9   for j <- 1 to n do
10     if x[i] = y[j] then
11       c[i, j] = 1 + c[i - 1, j - 1]
12       b[i, j] = NW
13     else if c[i - 1, j] >= c[i, j - 1] then
14       c[i, j] = c[i - 1, j]
15       b[i, j] = N
16     else
17       c[i, j] = c[i, j - 1]
18       b[i, j] = W
19 return c and b
```

Maior subsequência comum

Resultado da aplicação a x = ABCBDA e y = BDCA

			B	D	C	A	y
		0	1	2	3	4	j
	0	0	0	0	0	0	
A	1	**0**	0 N	0 N	0 N	1 NW	
B	2	0	**1** **NW**	**1** **W**	1 W	1 N	
C	3	0	1 N	1 N	**2** **NW**	**2** **W**	
B	4	0	1 NW	1 N	**2** **N**	2 N	
D	5	0	1 N	2 NW	**2** **N**	2 N	
A	6	0	1 N	2 N	2 N	**3** **NW**	
x	i						

Maior subsequência comum a x e y calculada: B C A

Maior subsequência comum

Reconstrução da subsequência

$b[1..m, 1..n]$: $b[i, j]$ é

- NW se $x_i = y_j$
- N se a subsequência calculada é comum a $x_1 \ldots x_{i-1}$ e $y_1 \ldots y_j$
- W se a subsequência calculada é comum a $x_1 \ldots x_i$ e $y_1 \ldots y_{j-1}$

PRINT-LCS(b, x, i, j)

```
1 if i = 0 or j = 0 then
2   return
3 if b[i, j] = NW then
4   PRINT-LCS(b, x, i - 1, j - 1)
5   print x[i]
6 else if b[i, j] = N then
7   PRINT-LCS(b, x, i - 1, j)
8 else
9   PRINT-LCS(b, x, i, j - 1)
```

Grafos

Grafos

Orientados ou **não orientados**

Pesados (ou **etiquetados**) ou **não pesados** (**não etiquetados**)

Grafo $G = (V, E)$

V – conjunto dos **nós** (ou **vértices**)

$E \subseteq V^2$ – conjunto dos **arcos** (ou **arestas**)

$w : E \to \mathbb{R}$ – **peso** (ou **etiqueta**) de um arco

Vértices e arcos

Se $G = (V, E)$ e $(u, v) \in E$

- O nó v diz-se adjacente ao nó u
- Os nós u e v são vizinhos
- Se G é orientado:
  - O nó u é a origem do arco (u, v)
  - O nó v é o destino do arco (u, v)
  - O nó u é um predecessor (ou antecessor) do nó v
  - O nó v é um sucessor do nó u
- Se G é não orientado:
  - Os nós u e v são as extremidades do arco (u, v)
  - Os arcos (u, v) e (v, u) são o mesmo arco
  - Logo, o nó u também é adjacente ao nó v

O grau do nó u é o número de arcos $(u, v) \in E$

Caminhos

Um caminho num grafo $G = (V, E)$ qualquer é uma sequência não vazia de vértices $v_i \in V$

$$v_0\, v_1\, \ldots\, v_k \quad (k \geq 0)$$

tal que $(v_i, v_{i+1}) \in E$, para $i < k$

O comprimento do caminho $v_0 v_1 \ldots v_k$ é k, o número de arestas que contém

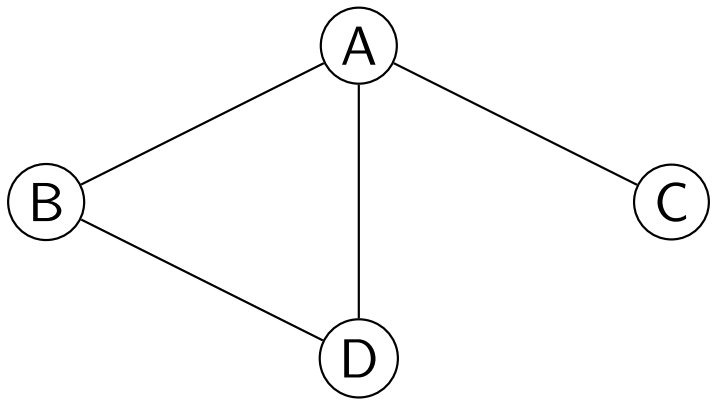
O caminho v_0 é o caminho de comprimento 0, de v_0 para v_0

Um caminho é simples se $v_i \neq v_j$ quando $i \neq j$

Exemplos de grafos

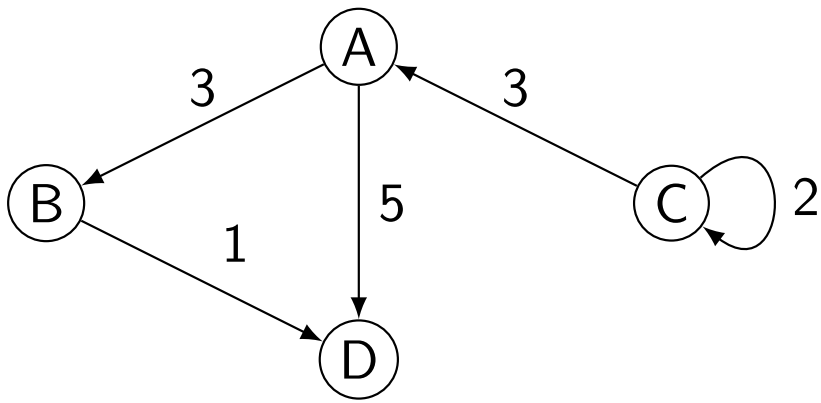
Grafo não orientado e não pesado

$$G = (\{A, B, C, D\}, \{(A, B), (B, D), (A, D), (C, A)\})$$



Grafo orientado pesado

$$G = (\{A, B, C, D\}, \{(A, B, 3), (B, D, 1), (A, D, 5), (C, A, 3), (C, C, 2)\})$$



Ciclos

Um ciclo, num grafo orientado, é um caminho em que

$$v_0 = v_k \quad \text{e} \quad k > 0$$

Num grafo não orientado, um caminho forma um ciclo se

$$v_0 = v_k \quad \text{e} \quad k \geq 3$$

Um ciclo é simples se $v_1, v_2, \ldots, v_k$ são distintos

Um grafo é acíclico se não contém qualquer ciclo simples

Representação / Implementação

Listas de adjacências

- Grafos esparsos $(|E| \ll |V|^2)$
- Permite descobrir rapidamente os vértices adjacentes a um vértice
- Complexidade espacial $O(V + E)$

Matriz de adjacências

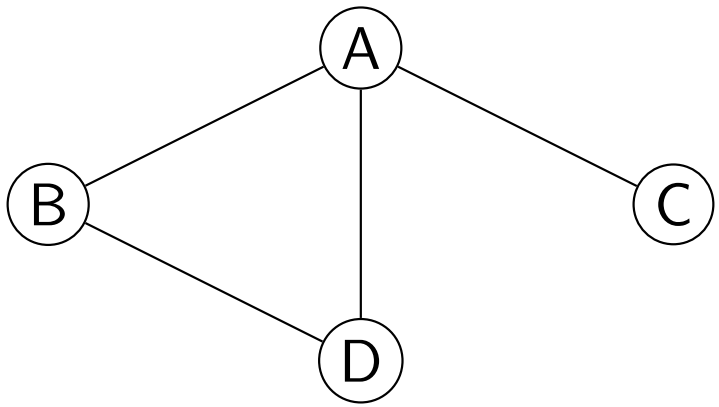
- Grafos densos $(|E| = O(V^2))$
- Permite verificar rapidamente se $(u, v) \in E$
- Complexidade espacial $O(V^2)$

Na notação O, V e E significam, respectivamente, $|V|$ e $|E|$

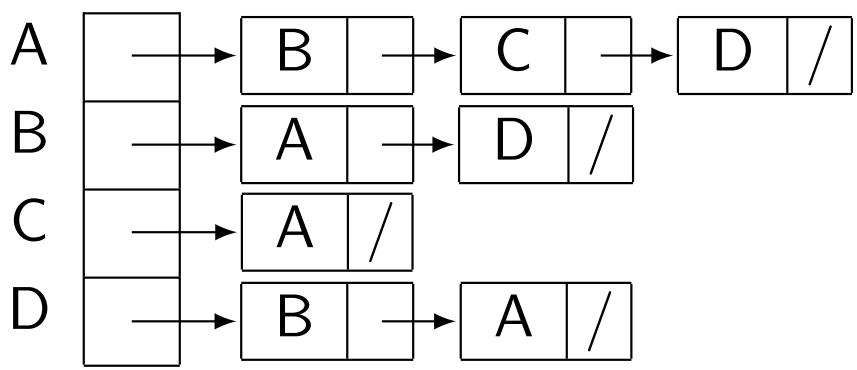
Representação / Implementação

Grafo não orientado e não pesado

Grafo $G = (\{A, B, C, D\}, \{(A, B), (B, D), (A, D), (C, A)\})$



Listas de adjacências



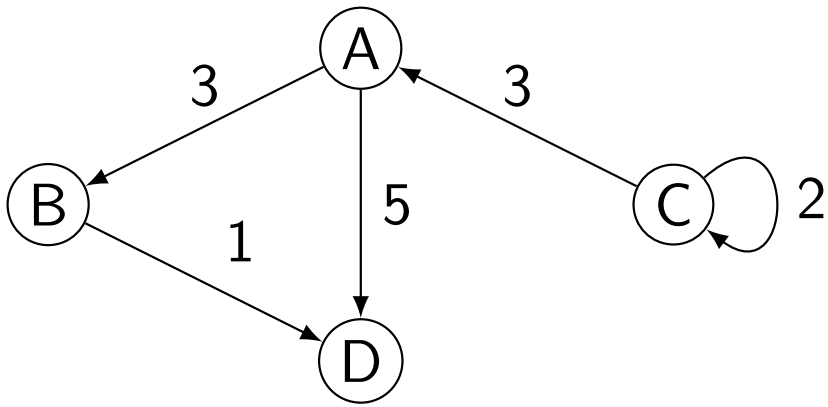
Matriz de adjacências

	A	B	C	D
A	0	1	1	1
B	1	0	0	1
C	1	0	0	0
D	1	1	0	0

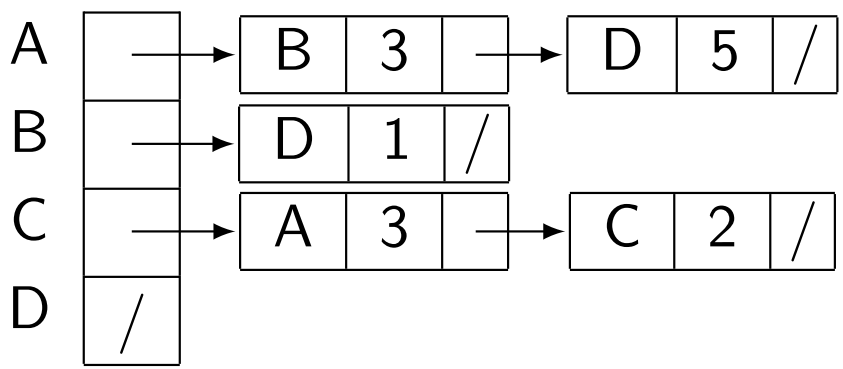
Representação / Implementação

Grafo orientado pesado

Grafo $G = (\{A, B, C, D\}, \{(A, B, 3), (B, D, 1), (A, D, 5), (C, A, 3), (C, C, 2)\})$



Listas de adjacências



Matriz de adjacências

	A	B	C	D
A	0	3	0	5
B	0	0	0	1
C	3	0	2	0
D	0	0	0	0

Percursos básicos em grafos

Percurso em largura

Nós são tratados por ordem crescente de distância ao nó em que o percurso se inicia

Percurso em profundidade

Nós são tratados pela ordem por que são encontrados

Percurso em largura (a partir do vértice s)

BFS(G, s)

```
 1  for each vertex u in G.V - {s} do
 2      u.color <- WHITE
 3      u.d <- INFINITY
 4      u.p <- NIL
 5  s.color <- GREY
 6  s.d <- 0
 7  s.p <- NIL
 8  Q <- EMPTY                          // queue
 9  ENQUEUE(Q, s)
10  while Q != EMPTY do
11      u <- DEQUEUE(Q)                 // explore next vertex
12      for each vertex v in G.adj[u] do
13          if v.color = WHITE then
14              v.color <- GREY
15              v.d <- u.d + 1
16              v.p <- u
17              ENQUEUE(Q, v)
18      u.color <- BLACK                // u has been explored
```

Percurso em largura

Breadth-first search

Descobre um caminho mais curto de um vértice s a qualquer outro vértice

Calcula o seu comprimento (linhas 3, 6 e 15)

Constrói a árvore da pesquisa em largura (linhas 4, 7 e 16), que permite reconstruir o caminho identificado

Atributos dos vértices

`color`	`WHITE`	não descoberto
	`GREY`	descoberto, mas não processado
	`BLACK`	processado
`d`		distância a s
`p`		antecessor do nó no caminho a partir de s

Análise da complexidade temporal de BFS (1)

Grafo implementado através de listas de adjacências

BFS(G, s)

```
1  for each vertex u in G.V - {s} do
2      u.color <- WHITE
3      u.d <- INFINITY
4      u.p <- NIL
```

- Ciclo das linhas 1–4 é executado $|V| - 1$ vezes

```
5  s.color <- GREY
6  s.d <- 0
7  s.p <- NIL
8  Q <- EMPTY                          // queue
9  ENQUEUE(Q, s)
```

- Linhas 5–9 com custo constante

Análise da complexidade temporal de BFS (2)

- Ciclo das linhas 10–18 é executado $|V|$ vezes, no pior caso

```
10  while Q != EMPTY do
11      u <- DEQUEUE(Q)
12      for each vertex v in G.adj[u] do
13          if v.color = WHITE then
14              v.color <- GREY
15              v.d <- u.d + 1
16              v.p <- u
17              ENQUEUE(Q, v)
18      u.color <- BLACK
```

- Mas o ciclo das linhas 12–17 é executado, no pior caso

$$\sum_{v \in V} |\,\texttt{G.adj}[v]\,| = |E| \text{ (orientado) ou } 2|E| \text{ (não orientado) vezes}$$

porque cada vértice só entra na fila uma vez

Análise da complexidade temporal de BFS (3)

Considerando que todas as operações, incluindo ENQUEUE e DEQUEUE, têm custo $O(1)$

- O ciclo das linhas 1–4 tem custo $O(V)$
- Conjuntamente, os ciclos das linhas 10–18 e 12–17 têm custo $O(E)$

Logo, a complexidade temporal de BFS é $O(V + E)$

Análise da complexidade temporal de BFS (4)

Grafo implementado através da matriz de adjacências

Na linha 12, é necessário percorrer uma linha da matriz, com $|V|$ elementos

Como o ciclo das linhas 10–18 é executado $|V|$ vezes, no pior caso, o custo combinado dos dois ciclos é $O(V^2)$

- Corresponde a aceder a todas as posições de uma matriz $|V| \times |V|$

Neste caso, a complexidade temporal de BFS será $O(V^2)$

Percurso em profundidade

DFS(G)

```
 1 for each vertex u in G.V do
 2     u.color <- WHITE
 3     u.p <- NIL
 4 time <- 0                        // global variable
 5 for each vertex u in G.V do
 6     if u.color = WHITE then
 7         DFS-VISIT(G, u)
```

DFS-VISIT(G, u)

```
 1 time <- time + 1                 // white vertex u has just
 2 u.d <- time                      // been discovered
 3 u.color <- GREY
 4 for each vertex v in G.adj[u] do // explore edge (u, v)
 5     if v.color = WHITE then
 6         v.p <- u
 7         DFS-VISIT(G, v)
 8 u.color <- BLACK                 // blacken u; it is finished
 9 time <- time + 1
10 u.f <- time                      // record u's finishing time
```

Percurso em profundidade

Depth-first search

Constrói a floresta da pesquisa em profundidade (linhas 3 [DFS] e 6 [DFS-VISIT])

Atributos dos vértices

`color`	`WHITE`	não descoberto
	`GREY`	descoberto e em processamento
	`BLACK`	processado
`d`		instante em que foi descoberto
`f`		instante em que terminou de ser processado
`p`		antecessor do nó num caminho que o contém

Análise da complexidade temporal de DFS

O ciclo das linhas 1–3 [DFS] é executado $|V|$ vezes

DFS-VISIT é chamada para cada um dos $|V|$ vértices

Para cada vértice u (e considerando a implementação através de listas de adjacências), o ciclo das linhas 4–7 [DFS-VISIT] é executado

$$|\,\texttt{G.adj}[u]\,| \quad \text{vezes}$$

Tendo todas as operações custo constante, considerando todas as chamadas a DFS-VISIT, DFS corre em tempo

$$O(V + \sum_{u \in V} |\,\texttt{G.adj}[u]\,|) = O(V + E)$$

Ordenação topológica

Seja $G = (V, E)$ um grafo orientado acíclico (DAG, de *directed acyclic graph*)

Ordem topológica

Se existe um arco de u para v, u está antes de v na ordem dos vértices

$$(u, v) \in E \Rightarrow u < v$$

TOPOLOGICAL-SORT(G)

1. Aplicar DFS(G)
2. Inserir cada vértice à cabeça de uma lista, quando termina o seu processamento
3. Devolver a lista, que contém os vértices por (alguma) ordem topológica

Ordenação topológica

Adaptação de DFS

$G = (V, E)$ – grafo orientado acíclico (DAG)

TOPOLOGICAL-SORT(G)

```
1 for each vertex u in G.V do
2     u.color <- WHITE
3 L <- EMPTY                          // lista, global
4 for each vertex u in G.V do
5     if u.color = WHITE then
6         DFS-VISIT'(G, u)
7 return L
```

DFS-VISIT'(G, u)

```
1 u.color <- GREY
2 for each vertex v in G.adj[u] do
3     if v.color = WHITE then
4         DFS-VISIT'(G, v)
5 u.color <- BLACK
6 LIST-INSERT-HEAD(L, u)
```

Ordenação topológica

Outro algoritmo

TOPOLOGICAL-SORT'(G)

```
 1 for each vertex u in G.V do
 2     u.i <- 0
 3 for each edge (u,v) in G.E do
 4     v.i <- v.i + 1                  // arcos com destino v
 5 L <- EMPTY                          // lista
 6 S <- EMPTY                          // conjunto
 7 for each vertex u in G.V do
 8     if u.i = 0 then
 9         SET-INSERT(S, u)
10 while S != EMPTY do
11     u <- SET-DELETE(S)              // retira um nó de S
12     for each vertex v in G.adj[u] do
13         v.i <- v.i - 1
14         if v.i = 0 then
15             SET-INSERT(S, v)
16     LIST-INSERT-TAIL(L, u)
17 return L
```

Conectividade (1)

Seja $G = (V, E)$ um grafo não orientado

G é conexo se existe algum caminho entre quaisquer dois nós

$V' \subseteq V$ é uma componente conexa de G se

- existe algum caminho entre quaisquer dois nós de V' e
- não existe qualquer caminho entre algum nó de V' e algum nó de $V \setminus V'$

Conectividade (2)

Seja $G = (V, E)$ um grafo orientado

G é fortemente conexo se existe algum caminho de qualquer nó para qualquer outro nó

$V' \subseteq V$ é uma componente fortemente conexa de G se

- existe algum caminho de qualquer nó de V' para qualquer outro nó de V' e
- se, qualquer que seja o nó $u \in V \setminus V'$
  - não existe qualquer caminho de algum nó de V' para u ou
  - não existe qualquer caminho de u para algum nó de V'

Grafo transposto

O grafo transposto do grafo orientado $G = (V, E)$ é o grafo

$$G^T = (V, E^T)$$

tal que

$$E^T = \{(v, u) \mid (u, v) \in E\}$$

Componentes fortemente conexas

Strongly connected components

G – grafo orientado

SCC(G)

1. Aplicar DFS(G) para calcular o instante `u.f` em que termina o processamento de cada vértice `u`
2. Calcular G^T
3. Aplicar DFS(G^T), processando os vértices por ordem decrescente de `u.f` (calculado em 1), no ciclo principal de DFS (linha 5)
4. Devolver os vértices de cada árvore da floresta da pesquisa em profundidade (construída em 3) como uma componente fortemente conexa distinta

Árvore de cobertura mínima (1)

Minimum(-weight) spanning tree

Seja $G = (V, E)$ um grafo pesado não orientado conexo

Uma árvore é um grafo não orientado conexo acíclico

(Retirando qualquer arco de uma árvore, obtém-se um grafo não conexo)

Uma árvore de cobertura de G é um subgrafo $G' = (V', E')$ de G tal que

- $V' = V$
- $E' \subseteq E$
- G' é uma árvore

Árvore de cobertura mínima (2)

Minimum(-weight) spanning tree

Seja o peso de um grafo $w(G)$ a soma dos pesos dos arcos de G

$$w(G) = \sum_{e \in E} w(e)$$

Uma árvore de cobertura mínima de G é uma árvore de cobertura G' de peso mínimo:

Para qualquer árvore de cobertura G'' de G tem-se

$$w(G') \leq w(G'')$$

Árvore de cobertura mínima

Algoritmo de Prim

$G = (V, E)$ – grafo pesado não orientado conexo

MST-PRIM(G, w, s)

```
 1 for each vertex u in G.V do
 2     u.key <- INFINITY                  // cost of adding u
 3     u.p <- NIL
 4 s.key <- 0
 5 Q <- G.V                  // priority queue, with key u.key
 6 while Q != EMPTY do
 7     u <- EXTRACT-MIN(Q)
 8     for each vertex v in G.adj[u] do
 9         if v in Q and w(u,v) < v.key then
10             v.p <- u
11             v.key <- w(u,v)          // decrease key in Q
```

Análise da complexidade do algoritmo de Prim

Grafo representado através de listas de adjacências

Linhas

1–3 Ciclo executado $|V|$ vezes

5 Construção da fila com prioridade (*heap*): $O(V)$

6–11 Ciclo executado $|V|$ vezes

7 Remoção do menor elemento da fila: $O(\log V)$

8–11 Ciclo executado $2|E|$ vezes **no total**

11 Alteração da prioridade de um elemento na fila: $O(\log V)$

Operação executada, no pior caso, $|E|$ vezes

Complexidade temporal do algoritmo

$$O(V + V + V \log V + 2E + E \log V) = O(E \log V)$$

Restantes operações com complexidade temporal constante

Árvore de cobertura mínima

Algoritmo de Kruskal

$G = (V, E)$ – grafo pesado não orientado conexo

MST-KRUSKAL(G, w)

```
 1 n <- |G.V|
 2 A <- EMPTY                          // set with the MST edges
 3 P <- MAKE-SETS(G.V)                 // partition of G.V
 4 Q <- G.E          // priority queue, key is weight w(u,v)
 5 e <- 0
 6 while e < n - 1 do
 7     (u,v) <- EXTRACT-MIN(Q)
 8     if FIND-SET(P, u) != FIND-SET(P, v) then
 9         A <- A + {(u,v)}
10         UNION(P, u, v)
11         e <- e + 1
12 return A
```

Análise da complexidade do algoritmo de Kruskal (1)

Linhas

3 Construção da partição

MAKE-SETS

4 Construção da fila com prioridade (*heap*)

$O(E)$

6–11 Ciclo executado entre $|V| - 1$ e $|E|$ vezes

7 Remoção do menor elemento da fila (*heap*)

$$O(\log E) = O(\log V)$$

$(|E| < |V|^2$ e $\log |E| < log|V|^2 = 2 \log |V| = O(\log V))$

8 $2 \times$ FIND-SET

10 Executada $|V| - 1$ vezes

UNION

Restantes operações com complexidade temporal constante

Análise da complexidade do algoritmo de Kruskal (2)

Juntando tudo, obtém-se

$$\text{MAKE-SETS} + O(E) + |E| \times O(\log V) + |E| \times 2 \times \text{FIND-SET} + (|V| - 1) \times \text{UNION}$$

ou

$$O(E) + |E| \times O(\log V) + f(V, E)$$

com

$$f(V, E) = \text{MAKE-SETS} + 2 \times |E| \times \text{FIND-SET} + (|V| - 1) \times \text{UNION}$$

Partição

Conjuntos disjuntos (*Disjoint sets*)

Abstracção da implementação de conjuntos disjuntos com os elementos do conjunto $\{1, 2, \ldots, n\}$

Operações suportadas

MAKE-SETS(n)
: Cria conjuntos singulares com os elementos $\{1, 2, \ldots, n\}$

FIND-SET(i)
: Devolve o representante do conjunto que contém o elemento i

UNION(i, j)
: Reúne os conjuntos a que pertencem os elementos i e j

Também é conhecido como *Union-Find*

Partição

Implementação em vector

MAKE-SETS(n)

```
1 let P[1..n] be a new array
2 for i <- 1 to n do
3   P[i] <- -1   // i is the representative for set {i}
4 return P
```

FIND-SET(P, i)

```
1 while P[i] > 0 do
2       i <- P[i]
3 return i
```

UNION(P, i, j)

```
1 P[FIND-SET(P,j)] <- FIND-SET(P, i)
```

Partição

Implementação em vector

Reunião por tamanho

Se P[i] $= -k$, o conjunto de que i é o representante contém k elementos

UNION-BY-SIZE(P, i, j)

```
1 ri <- FIND-SET(P, i)     // get i's set representative
2 rj <- FIND-SET(P, j)     // get j's set representative
3 if (P[rj] < P[ri])       // j's set is larger than i's
4     P[rj] <- P[rj] + P[ri]
5     P[ri] <- rj
6 else                     // i's is larger or both have the same size
7     P[ri] <- P[ri] + P[rj]
8     P[rj] <- ri
```

Partição

Implementação em vector

Reunião por altura

Se P[i] $= -h$, a árvore do conjunto de que i é o representante tem altura h

UNION-BY-RANK(P, i, j)

```
1 ri <- FIND-SET(P, i)
2 rj <- FIND-SET(P, j)
3 if (P[rj] < P[ri])     // j's set tree taller than i's
4     P[ri] <- rj
5 else
6     if (P[rj] == P[ri])     // both heights are equal
7         P[ri] <- P[ri] - 1
8     P[rj] <- ri
```

Partição

Implementação em vector

Compressão de caminho

FIND-SET-WITH-PATH-COMPRESSION(P, i)

```
1 if P[i] < 0 then
2     return i
3 P[i] <- FIND-SET-WITH-PATH-COMPRESSION(P, P[i])
4 return P[i]
```

Análise da complexidade do algoritmo de Kruskal (3)

$$O(E) + |E| \times O(\log V) + f(V, E)$$

$$f(V, E) = \text{MAKE-SETS} + 2 \times |E| \times \text{FIND-SET} + (|V| - 1) \times \text{UNION}$$

Implementação da Partição	Básica	União por tam./altura	+ Compressão de caminho
MAKE-SETS	$O(V)$	$O(V)$	$O((V + E)\,\alpha(V))$ [Tarjan 1975]
$2 \times \|E\| \times$ FIND-SET	$O(EV)$	$O(E \log V)$	
$(\|V\| - 1) \times$ UNION	$O(V^2)$	$O(V \log V)$	
$f(V, E)$	$O(EV)$	$O(E \log V)$	$O(E\alpha(V))$
Algoritmo de Kruskal	$O(EV)$	$O(E \log V)$	$O(E \log V)$

$\alpha(n) \leq 4$ para $n < 10^{80}$

Análise da complexidade do algoritmo de Kruskal (4)

$\alpha(n) = \min\{k \mid A_k(1) \geq n\}$

onde

$$A_k(j) = \begin{cases} j+1 & \text{se } k = 0 \\ A_{k-1}^{(j+1)}(j) & \text{se } k \geq 1 \end{cases}$$

$$A_0(1) = 2$$
$$A_1(1) = A_0(A_0(1)) = 3$$
$$A_2(1) = A_1(A_1(1)) = 7$$
$$A_3(1) = 2047$$
$$A_4(1) \gg 2^{2048} \gg 10^{80}$$

Iteração de uma função

$A_{k-1}^{(0)}(j) = j$ e $A_{k-1}^{(i)}(j) = A_{k-1}(A_{k-1}^{(i-1)}(j))$, para $i \geq 1$

Caminho mais curto

Num grafo pesado, com pesos w, o peso do caminho

$$p = v_0\, v_1\, \ldots\, v_k$$

é a soma dos pesos dos arcos que o integram

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

O caminho p é mais curto que o caminho p' se o peso de p é menor que o peso de p'

Cálculo dos caminhos mais curtos

Algoritmos

Cálculo dos caminhos mais curtos num grafo orientado acíclico (DAG), com pesos possivelmente negativos

Algoritmo de Dijkstra, para grafos sem pesos negativos

Algoritmo de Bellman-Ford, para quaisquer grafos pesados

Estes algoritmos calculam os caminhos mais curtos de um nó s para os restantes nós do grafo (*single-source shortest paths*)

Caminhos mais curtos

Subrotinas comuns aos diversos algoritmos

O peso do caminho mais curto de s a qualquer outro nó é inicializado com ∞

INITIALIZE-SINGLE-SOURCE(G, s)

```
1 for each vertex v in G.V do
2     v.d <- INFINITY   // peso do caminho mais curto de s a v
3     v.p <- NIL        // predecessor de v nesse caminho
4 s.d <- 0
```

Se o caminho de s a v, que passa por u e pelo arco (u, v), tem menor peso do que o mais curto anteriormente encontrado, encontrámos um caminho mais curto

RELAX(u, v, w)

```
1 if u.d + w(u,v) < v.d then
2     v.d <- u.d + w(u,v)
3     v.p <- u
```

Caminhos mais curtos a partir de um vértice

Algoritmo para DAGs

$G = (V, E)$ – DAG pesado (pode ter pesos negativos)

DAG-SHORTEST-PATHS(G, w, s)

```
1 topologically sort the vertices of G
2 INITIALIZE-SINGLE-SOURCE(G, s)
3 for each vertex u, taken in topologically sorted
                                                order do
4     for each vertex v in G.adj[u] do
5         RELAX(u, v, w)
```

Caminhos mais curtos a partir de um vértice

Algoritmo de Dijkstra

$G = (V, E)$ – grafo pesado orientado (sem pesos negativos)

DIJKSTRA(G, w, s)

```
1 INITIALIZE-SINGLE-SOURCE(G, s)
2 S <- EMPTY
3 Q <- G.V                          // priority queue (key: u.d)
4 while Q != EMPTY do
5     u <- EXTRACT-MIN(Q)
6     S <- S + {u}
7     for each vertex v in G.adj[u] do
8         RELAX(u, v, w)            // may alter v.d key in Q
```

Quando é encontrado um novo caminho mais curto para um vértice (na função RELAX), é necessário reorganizar a fila Q (DECREASE-KEY)

Caminhos mais curtos a partir de um vértice

Algoritmo de Bellman-Ford

$G = (V, E)$ – grafo pesado orientado (pode ter pesos negativos)

BELLMAN-FORD(G, w, s)

```
1 INITIALIZE-SINGLE-SOURCE(G, s)
2 for i <- 1 to |G.V| - 1 do
3     for each edge (u,v) in G.E do
4         RELAX(u, v, w)
5 for each edge (u,v) in G.E do
6     if u.d + w(u,v) < v.d then
7         return FALSE
8 return TRUE
```

Complexidade dos algoritmos

$G = (V, E)$

	Compl. Temporal
Percurso em largura	$O(V + E)$
Percurso em profundidade	$O(V + E)$
Grafo transposto	$O(V + E)$
Cálculo das componentes fortemente conexas	$O(V + E)$
Ordenação topológica (ambos os algoritmos)	$O(V + E)$
Algoritmos de Prim e de Kruskal	$O(E \log V)$
Caminhos mais curtos num DAG	$O(V + E)$
Algoritmo de Dijkstra	$O(E \log V)$
Algoritmo de Bellman-Ford	$O(VE)$

Pressupostos

Grafo representado através de listas de adjacências (excepto algoritmo de Bellman-Ford)

Algoritmos de Prim e de Dijkstra recorrem a uma fila tipo *heap* binário (EXTRACT-MIN e DECREASE-KEY com complexidade temporal logarítmica no número de elementos da fila)

Algoritmo de Kruskal usa Partição com compressão de caminho