Estruturas de Dados e Algoritmos II

Vasco Pedro

Departamento de Informática
Universidade de Évora

2017/2018

A notação O (1)

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

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

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

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

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

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

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

$$f(n) = O(g(n)) \quad \text{significa} \quad f(n) \in O(g(n))$$

A notação O (2)

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

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

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

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

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

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

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

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

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

Pseudo-código

Exemplo

PESQUISA-LINEAR(V, k)

```
n <- |V|                                 // inicialização
i <- 1

while i <= n and V[i] != k do            // pesquisa
    i <- i + 1

if i <= n then                           // resultado
    return i                             //  - sucesso
else
    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

1. Acesso a `V`, obtenção da dimensão de um vector, afectação: operações com complexidade constante

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

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

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

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

Análise da complexidade (3)

Exemplo

Juntando tudo

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

A função PESQUISA-LINEAR tem complexidade temporal linear na dimensão do vector `V`

Se n for a dimensão do vector `V`, a função tem complexidade temporal

$$O(n)$$

Isto significa que, para um *input* de dimensão n, o tempo que a função demora a executar, no pior caso, é

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

A complexidade na prática

Pesquisa linear

De um valor num vector

PESQUISA-LINEAR(V, k)

```
n <- |V|                              // inicialização
i <- 1

while i <= n and V[i] != k do         // pesquisa
    i <- i + 1

if i <= n then                        // resultado
    return i                          //  - sucesso
else
    return -1                         //  - insucesso
```

Ainda a pesquisa linear

De um valor num vector ordenado

PESQUISA-LINEAR-ORD(V, k)

```
n <- |V|                                  // inicialização
i <- 1

while i <= n and V[i] < k do              // pesquisa
    i <- i + 1

if i <= n and V[i] = k then               // resultado
    return i                              //  - sucesso
else
    return -1                             //  - insucesso
```

Pesquisa dicotómica ou binária

De um valor num vector ordenado

PESQUISA-DICOTÓMICA(V, k)

```
n <- |V|
return PESQUISA-DICOTÓMICA-REC(V, k, 1, n)
```

PESQUISA-DICOTÓMICA-REC(V, k, i, f)

```
if i > f then
    return -1                    // intervalo vazio

m <- (i + f) / 2

if k < V[m] then
    return PESQUISA-DICOTÓMICA-REC(V, k, i, m - 1)

if k > V[m] then
    return PESQUISA-DICOTÓMICA-REC(V, k, m + 1, f)

// k = V[m]
return m
```

Solução de recorrências (1)

Master theorem (versão simplificada)

$$T(n) = \begin{cases} O(1) & n = 1 \qquad\qquad n = 0 \\ & \qquad\quad \textit{ou} \\ a\, T(\frac{n}{2}) + f(n) & n \geq 2 \qquad\qquad n \geq 1 \end{cases}$$

onde $f(n)$ corresponde ao trabalho feito no corpo da função, sem incluir o feito pelas chamadas recursivas, e $a \geq 1$ é uma constante

Soluções

$$a = 1 \qquad T(n) = \begin{cases} O(\log n) & f(n) = O(1) \\ O(n) & f(n) = O(n) \end{cases}$$

$$a = 2 \qquad T(n) = \begin{cases} O(n) & f(n) = O(1) \\ O(n \log n) & f(n) = O(n) \end{cases}$$

Solução de recorrências (2)

$$T(n) = \begin{cases} O(1) & n = 1 \qquad\qquad\qquad n = 0 \\ & \qquad\qquad\quad ou \\ a\,T(n-1) + O(1) & n \geq 2 \qquad\qquad\qquad n \geq 1 \end{cases}$$

onde $a \geq 1$ é uma constante

Soluções

$$T(n) = \begin{cases} O(n) & a = 1 \\ O(a^n) & a > 1 \end{cases}$$

Complexidade das pesquisas linear e dicotómica

Pior caso e caso esperado para a complexidade temporal das pesquisas num vector de dimensão n

Pesquisa linear

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

Pesquisa linear num vector ordenado

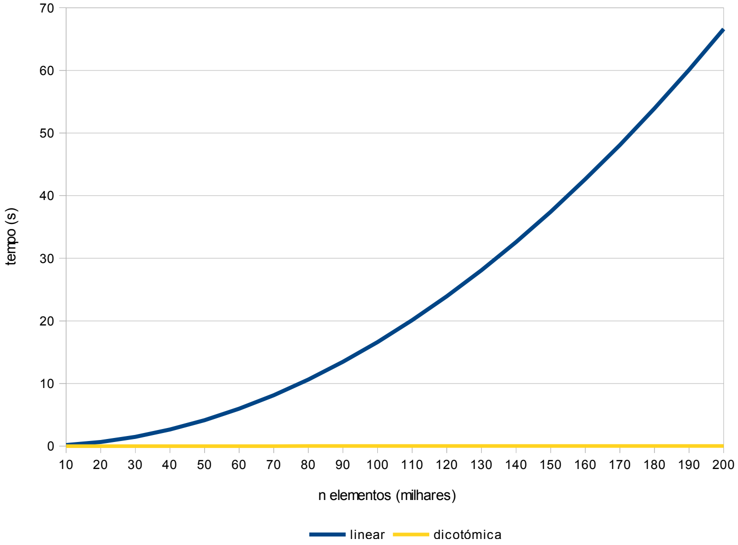
$$T(n) = O(n)$$

Pesquisa dicotómica

$$T(n) = O(\log n)$$

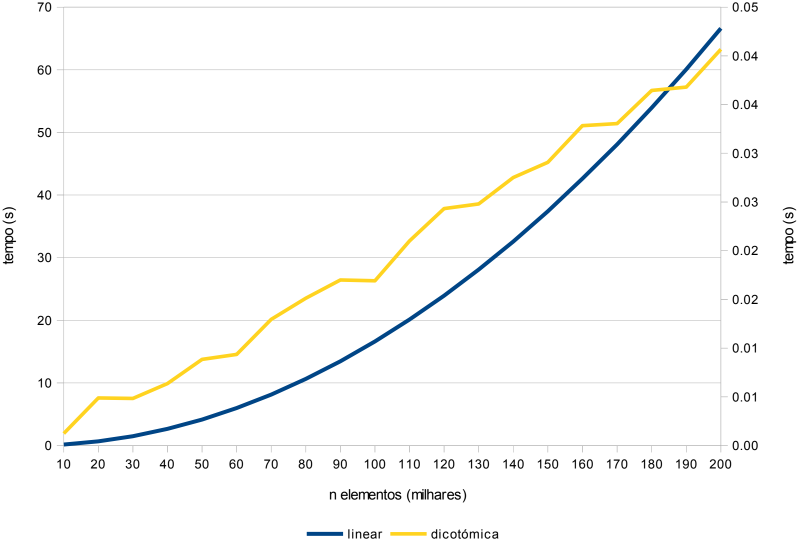
Pesquisa linear e dicotómica

Dos *n* elementos de um vector



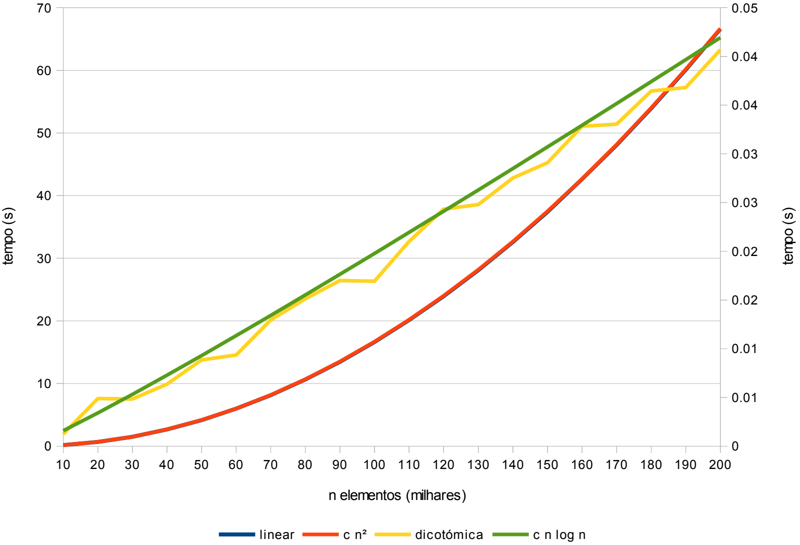
Pesquisa linear e dicotómica (com escalas diferentes)

Dos n elementos de um vector



Pesquisa linear e dicotómica (com escalas diferentes)

Dos n elementos de um vector



Números de Fibonacci

Versão recursiva

```
int fibonacci(int n)
{
  if (n == 0)
    return 0;

  if (n == 1)
    return 1;

  return fibonacci(n - 1) + fibonacci(n - 2);
}
```

Números de Fibonacci

Versão iterativa

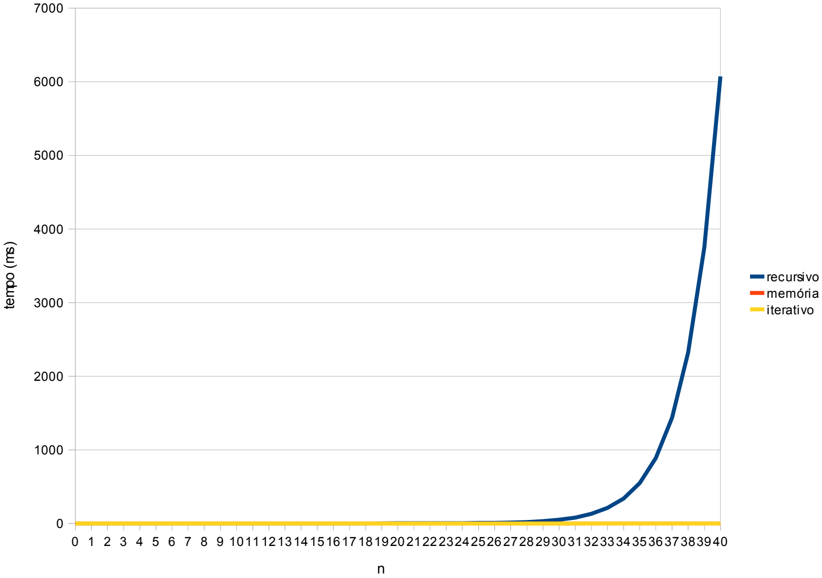
```
int fibonacci(int n)
{
  int i = 0;

  int corrente = 0;     // fibonacci(i)
  int anterior = 1;     // fibonacci(i - 1)

  while (i < n)
    {
      // fibonacci(i + 1)
      int proximo = corrente + anterior;

      anterior = corrente;
      corrente = proximo;

      i++;
    }

  return corrente;
}
```

Números de Fibonacci

Versão recursiva com memória

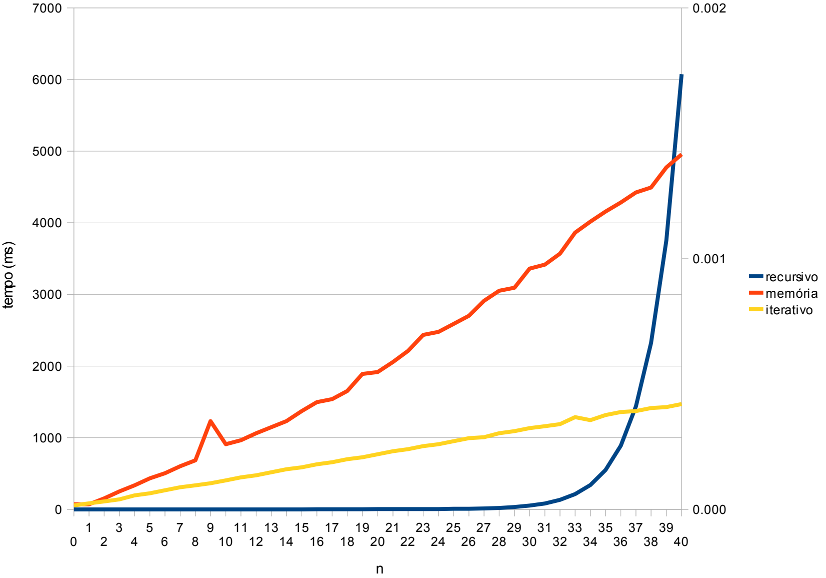
```
#define ELEMENTOS ...

...

int fibonacci(int n)
{
  static int mem[ELEMENTOS] = { 0, 1 };

  if (n > 1 && mem[n] == 0)
    mem[n] = fibonacci(n - 1) + fibonacci(n - 2);

  return mem[n];
}
```

Números de Fibonacci



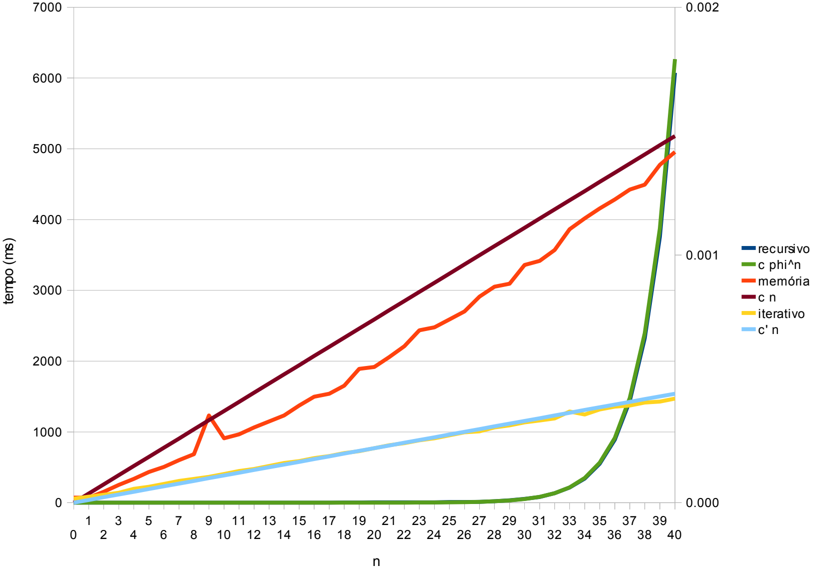
Números de Fibonacci

Escalas diferentes



Números de Fibonacci

Escalas diferentes



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

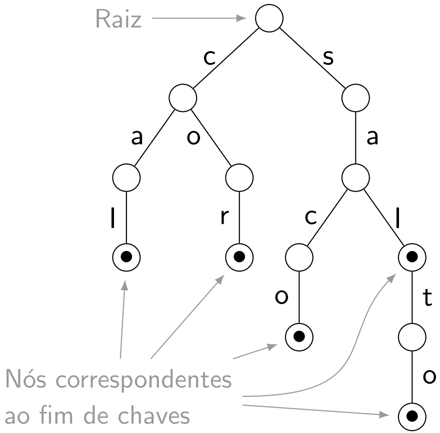
As *tries* apresentam algumas características que as distinguem de outras estruturas de dados

1. A complexidade das operações sobre uma *trie* não depende do número de elementos que ela contém
2. As chaves não têm de estar explicitamente presentes numa *trie*
3. As operações sobre uma *trie* 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 vectores)

$c[1..d]$ – 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)

```
x <- T.root
i <- 1
while x != NIL and i <= |k| do
    x <- x.c[k[i]]
    i <- i + 1
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 teste 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)

```
if T.root = NIL then
    T.root <- ALLOCATE-NODE()
    T.root.p <- NIL
x <- T.root
i <- 1
while i <= |k| and x.c[k[i]] != NIL do
    x <- x.c[k[i]]
    i <- i + 1
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)

```
y <- x
for j <- i to |k| do
    y.c[k[j]] <- ALLOCATE-NODE()
    y.c[k[j]].p <- y
    y <- y.c[k[j]]
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* (começando no i-ésimo símbolo)

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

TRIE-DELETE(T, k) (1)

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

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)

```
...
if x != NIL and x.word then
    x.word <- FALSE          // k deixa de estar na trie
    repeat
        i <- i - 1
        childless <- TRUE               // x tem filhos?
        for j <- 1 to d do
            if x.c[j] != NIL then
                childless <- FALSE
        if childless then      // se não tem, é apagado
            y <- x.p
            if y = NIL then
                T.root <- NIL     // a trie ficou vazia
            else
                y.c[k[i]] <- NIL
            FREE-NODE(x)
            x <- y
    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*

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

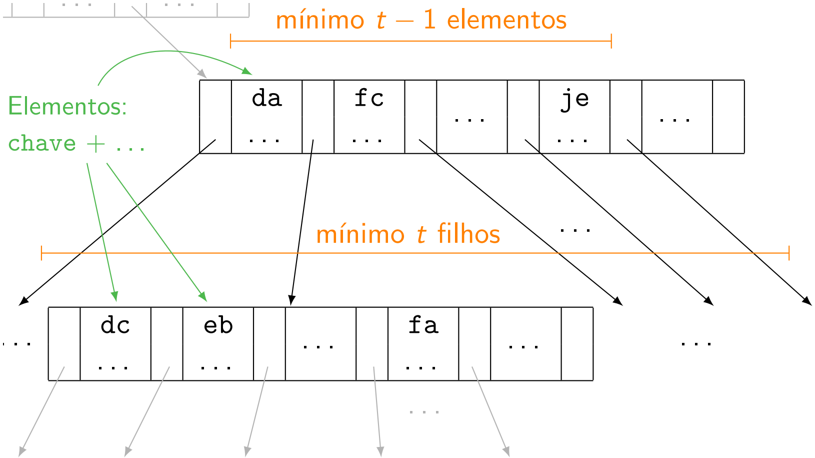
Armazenamento em memória secundária

Indexação eficiente

Minimização de acessos

B-Trees

Estrutura dos nós internos (exceptuando a raiz)



B-Trees

Características (1)

B-trees são árvores cujos nós internos (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

- 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 (2)

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

Em todos os nós, verifica-se:

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

Em todos os nós internos, verifica-se:

$$\mathit{chaves}(\text{filho}_1) \leq \mathit{chave}(\text{elemento}_1) \leq \mathit{chaves}(\text{filho}_2) \leq$$

$$\leq \mathit{chave}(\text{elemento}_2) \leq \ldots \leq \mathit{chave}(\text{elemento}_k) \leq \mathit{chaves}(\text{filho}_{k+1})$$

Todas as folhas estão à mesma profundidade

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, ...)

A raiz é mantida sempre em memória

B-TREE-CREATE(T)

```
x <- ALLOCATE-NODE()        // create a new node
x.leaf <- TRUE              //   with no children
x.n <- 0                    //   and no elements
DISK-WRITE(x)               // write it to disk
T.root <- x                 // it is the root of
                            //   the new B-tree
```

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

B-TREE-SEARCH(x, k)

```
i <- 1
while i <= x.n and k > x.key[i] do
    i <- i + 1
if i <= x.n and k = x.key[i] then
    return (x, i)
if x.leaf then      // not found in the current node
    return NIL
DISK-READ(x.c[i])
return B-TREE-SEARCH(x.c[i], k)
```

Pesquisa (recursiva) do elemento com chave k no nó x, que já está em memória

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

$$O(t \log_t n)$$

Altura de uma árvore

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)

```
r <- T.root
if r.n = 2t - 1 then      // see if the root is full
    s <- ALLOCATE-NODE()
    T.root <- s
    s.leaf <- FALSE
    s.n <- 0
    s.c[1] <- r
    B-TREE-SPLIT-CHILD(s, 1)
    B-TREE-INSERT-NONFULL(s, k)
else
    B-TREE-INSERT-NONFULL(r, k)
```

Inserção efectuada numa única passagem pela árvore

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

```
y <- x.c[i]                          // node to split (i-th child)
z <- ALLOCATE-NODE()                 // new (i+1)-th child
z.leaf <- y.leaf
z.n <- t - 1
for j <- 1 to t - 1 do               // move elements to new node
    z.key[j] <- y.key[j + t]
if not y.leaf then
    for j <- 1 to t do               // move children as well
         z.c[j] <- y.c[j + t]
y.n <- t - 1
for j <- x.n + 1 downto i + 1 do // make room for x's new
    x.c[j + 1] <- x.c[j]             // child
x.c[i + 1] <- z
for j <- x.n downto i do             // make room for the element
    x.key[j + 1] <- x.key[j]         // that will be promoted
x.key[i] <- y.key[t]
x.n <- x.n + 1
DISK-WRITE(y)
DISK-WRITE(z)
DISK-WRITE(x)
```

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

```
i <- x.n
if x.leaf then   // if in a leaf, insert the new element
    while i >= 1 and k < x.key[i] do
        x.key[i + 1] <- x.key[i]
        i <- i - 1
    x.key[i + 1] <- k
    x.n <- x.n + 1
    DISK-WRITE(x)
else      // otherwise, descend to the appropriate child
    while i >= 1 and k < x.key[i] do
        i <- i - 1
    i <- i + 1
    DISK-READ(x.c[i])
    if x.c[i].n = 2t - 1 then      // is the child full?
        B-TREE-SPLIT-CHILD(x, i)
        if k > x.key[i] then
            i <- i + 1
    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 ...

1. ... e é uma folha
   Remove o elemento
2. ... na posição i 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 o elemento a remover foi parar)

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 a raiz ficou vazia (sem elementos) e não é folha
- o seu único filho passa a ser a nova raiz

B-TREE-DELETE(T, k)

```
r <- T.root
B-TREE-DELETE-SAFE(r, k)
if r.n = 0 and not r.leaf then
    r <- r.c[1]
    FREE-NODE(T.root)
    T.root <- r
```

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

```
i <- 1
while i <= x.n and k > x.key[i] do
    i <- i + 1
if i <= x.n and k = x.key[i] then
    if x.leaf then                                        // Caso 1
        for j <- i to x.n - 1 do                          // Caso 1
            x.key[j] <- x.key[j + 1]                      // Caso 1
        x.n <- x.n - 1                                    // Caso 1
        DISK-WRITE(x)                                     // Caso 1
    else
        y <- x.c[i]
        DISK-READ(y)
        if y.n > t - 1 then                               // Caso 2a
            x.key[i] <- B-TREE-DELETE-MAX(y)              // Caso 2a
            DISK-WRITE(x)                                 // Caso 2a
        else
            z <- x.c[i + 1]
            DISK-READ(z)
            if z.n > t - 1 then                           // Caso 2b
                x.key[i] <- B-TREE-DELETE-MIN(z)          // Caso 2b
                DISK-WRITE(x)                             // Caso 2b
            else
                B-TREE-MERGE-CHILDREN(x, i)               // Caso 2c
                B-TREE-DELETE-SAFE(x.c[i], k)             // Caso 2c
else if not x.leaf then
    ...
```

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

```
else if not x.leaf then
    y <- x.c[i]
    DISK-READ(y)
    if y.n = t - 1 then
        borrowed <- FALSE
        if i > 1 then
            z <- x.c[i - 1]
            DISK-READ(z)
            if z.n > t - 1 then                           // Caso 3a
                B-TREE-BORROW-FROM-LEFT-SIBLING(x, i)     // Caso 3a
                borrowed <- TRUE                          // Caso 3a
            else
                m <- i - 1
        if not borrowed and i <= x.n then
            z <- x.c[i + 1]
            DISK-READ(z)
            if z.n > t - 1 then                           // Caso 3a
                B-TREE-BORROW-FROM-RIGHT-SIBLING(x, i)    // Caso 3a
                borrowed <- TRUE                          // Caso 3a
            else
                m <- i
        if not borrowed then                              // Caso 3b
            B-TREE-MERGE-CHILDREN(x, m)                   // Caso 3b
            y <- x.c[m]                                   // Caso 3b
    B-TREE-DELETE-SAFE(y, k)
```

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

```
y <- x.c[i]                            // merge children i
z <- x.c[i + 1]                        // and i+1
y.key[t] <- x.key[i]
for j <- 1 to t - 1 do                 // transfer c[i+1]
    y.key[t + j] <- z.key[j]           // contents to c[i]
if not y.leaf then
    for j <- 1 to t do
        y.c[t + j] <- z.c[j]
y.n <- 2t - 1                          // c[i] is now full
for j <- i + 1 to x.n do
  x.key[j - 1] <- x.key[j]
for j <- i + 2 to x.n + 1 do
  x.c[j - 1] <- x.c[j]
x.n <- x.n - 1
FREE-NODE(z)                          // delete old c[i+1]
DISK-WRITE(y)
DISK-WRITE(x)
```

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

```
y <- x.c[i]                      // node i's left
z <- x.c[i - 1]                  // sibling is i-1
for j <- t - 1 downto 1 do       // make room for
    y.key[j + 1] <- y.key[j]     // new 1st key
y.key[1] <- x.key[i - 1]
x.key[i - 1] <- z.key[z.n]
if not y.leaf then
    for j <- t downto 1 do       // make room for
        y.c[j + 1] <- y.c[j]     // new 1st child
    y.c[1] <- z.c[z.n + 1]
y.n <- t
z.n <- z.n - 1
DISK-WRITE(z)
DISK-WRITE(y)
DISK-WRITE(x)
```

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)

Árvore com grau de ramificação mínimo t e com n elementos

Complexidade temporal das operações

pesquisa, inserção, remoção

$$O(t \log_t n)$$

Número de nós acedidos (nas operações acima)

$$O(\log_t n)$$

Programação dinâmica

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

Alguns números

Número de cortes possíveis

$$2^{n-1}$$

Exemplo ($n = 4$)

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

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

Número de cortes distintos possíveis

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

Exemplo ($n = 4$)

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

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

$$1, 2, \dots, n$$

com preços

$$p_1, p_2, \dots, p_n$$

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

- fazendo um corte inicial de comprimento i, de valor p_i, somado com
- o maior valor de venda de uma vara de comprimento $n - i$
- onde i pode ter qualquer valor entre 1 e n

Corte de varas

Função recursiva

Corte de uma vara de comprimento n

Tamanho dos cortes: $i = 1, \ldots, n$
Preços: p_i, $i = 1, \ldots, n$

$r[0..n]$: $r[l]$ é o maior preço que se pode obter para uma vara de comprimento l

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

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

Corte de varas

Implementação recursiva

CUT-ROD(p, l)

```
if l = 0 then
  return 0
q <- -INFINITY
for i <- 1 to l do
  q <- max(q, p[i] + CUT-ROD(p, l - i))
return q
```

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

Corte de varas

Implementação recursiva com *memoização*

MEMOIZED-CUT-ROD(p, n)

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

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

```
if r[l] = -INFINITY then
  if l = 0 then
    q <- 0
  else
    q <- -INFINITY
    for i <- 1 to l do
      q <- max(q, p[i] + MEMOIZED-CUT-ROD-2(p, l - i, r))
  r[l] <- q
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)

```
let r[0..n] be a new array
r[0] <- 0
for l <- 1 to n do
  q <- -INFINITY
  for i <- 1 to l do
    q <- max(q, p[i] + r[l - i])
  r[l] <- q
return r[n]
```

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, $l = 1, \ldots, n$

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

Restantes linhas executam em tempo constante

Complexidade temporal $\Theta(n^2)$

Complexidade espacial $\Theta(n)$

Corte de varas

Construção da solução

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

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

```
let r[0..n] and s[1..n] be new arrays
r[0] <- 0
for l <- 1 to n do
  q <- -INFINITY
  for i <- 1 to l do
    if q < p[i] + r[l - i] then
      q <- p[i] + r[l - i]
      s[l] <- i
  r[l] <- q
return r and s
```

Corte de varas

Resolução completa

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

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

Programação dinâmica

Técnica de programação usada 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

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 duas matrizes

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

```
let C[1..p,1..r] be a new matrix
for i <- 1 to p do
  for j <- 1 to r do
    C[i,j] <- 0
    for k <- 1 to q do
      C[i,j] <- C[i,j] + A[i,k] * B[k,j]
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 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 $p_{i-1} \times p_i$)

Produto 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

$(A_1 A_2) A_3$

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

$A_1 (A_2 A_3)$

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

Formulação alternativa

Como colocar parêntesis no produto $A_1 A_2 \dots A_n$ 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 matrizes

Caracterização de uma solução óptima (1)

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

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

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

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

para qualquer $1 \leq k < n$, será

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

Produto 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 matrizes

Função recursiva

Sequência de matrizes a multiplicar

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

Dimensões das matrizes: $p_0\, p_1 \ldots p_n$

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

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

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

Produto de matrizes

Cálculo de $m[i, j]$

$$\begin{array}{c|c|c|c|c|c|}
\multicolumn{6}{c}{m} \\
 & 1 & 2 & 3 & 4 & 5 \\
\hline
1 & 0 & m_{12} & m_{13} & m_{14} & \color{red}{m_{15}} \\
\hline
2 & & 0 & m_{23} & m_{24} & m_{25} \\
\hline
3 & & & 0 & m_{34} & m_{35} \\
\hline
4 & & & & 0 & m_{45} \\
\hline
5 & & & & & 0 \\
\hline
\end{array}$$

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 matrizes

Cálculo iterativo de $m[1, n]$

MATRIX-CHAIN-ORDER(p)

```
n <- |p| - 1                 // p[0..n]
let m[1..n,1..n] be a new table
for i <- 1 to n do
     m[i, i] <- 0
for l <- 2 to n do           // l is the chain length
    for i <- 1 to n - l + 1 do
        j <- i + l - 1
        m[i, j] <- INFINITY
        for k <- i to j - 1 do
            q <- m[i, k] + m[k + 1, j] +
                 p[i - 1] * p[k] * p[j]
            if q < m[i, j] then
                m[i, j] <- q
return m[1, n]
```

Produto 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)

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

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

Complexidade temporal $O(n^3)$

Complexidade espacial $O(n^2)$

Produto de matrizes

Construção da solução

MATRIX-CHAIN-ORDER(p)

```
n <- |p| - 1                  // p[0..n]
let m[1..n,1..n] and s[1..n-1,2..n] be new tables
for i <- 1 to n do
     m[i, i] <- 0
for l <- 2 to n do            // l is the chain length
    for i <- 1 to n - l + 1 do
        j <- i + l - 1
        m[i, j] <- INFINITY
        for k <- i to j - 1 do
            q <- m[i, k] + m[k + 1, j] +
                 p[i - 1] * p[k] * p[j]
            if q < m[i, j] then
                m[i, j] <- q
                s[i, j] <- k
return m and s
```

Produto 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 matrizes

Melhor colocação de parêntesis

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

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

```
if i = j then
  print "A"_i
else
  print "("
  PRINT-OPTIMAL-PARENS(s, i, s[i, j])
  PRINT-OPTIMAL-PARENS(s, s[i, j] + 1, j)
  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

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

São subsequências:

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

Não são subsequências:

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

$$\begin{aligned} x = &\ \text{A B C B D A B} \\ y = &\ \text{B D C A B A} \end{aligned}$$

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

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

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

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

Comprimento de uma maior subsequência comum a x e y: $c[m, n]$

Maior subsequência comum

Cálculo iterativo de $c[m, n]$

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

```
m <- |x|
n <- |y|
let c[0..m, 0..n] be a new table
for i <- 1 to m do
  c[i, 0] <- 0
for j <- 0 to n do
  c[0, j] <- 0
for i <- 1 to m do
  for j <- 1 to n do
    if x[i] = y[j] then
      c[i, j] = 1 + c[i - 1, j - 1]
    else if c[i - 1, j] >= c[i, j - 1] then
      c[i, j] = c[i - 1, j]
    else
      c[i, j] = c[i, j - 1]
return c[m, n]
```

Maior subsequência comum

Análise da complexidade

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

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 9–15

Complexidade temporal $O(mn)$

Complexidade espacial $O(mn)$

Maior subsequência comum

Construção da solução

LONGEST-COMMON-SUBSEQUENCE(x, y)

```
m <- |x|
n <- |y|
let c[0..m, 0..n] and b[1..m, 1..n] be new tables
for i <- 1 to m do
  c[i, 0] <- 0
for j <- 0 to n do
  c[0, j] <- 0
for i <- 1 to m do
  for j <- 1 to n do
    if x[i] = y[j] then
      c[i, j] = 1 + c[i - 1, j - 1]
      b[i, j] = NW
    else if c[i - 1, j] >= c[i, j - 1] then
      c[i, j] = c[i - 1, j]
      b[i, j] = N
    else
      c[i, j] = c[i, j - 1]
      b[i, j] = W
return c and b
```

Maior subsequência comum

Resultado da aplicação a x = ABCBDAB e y = BDCABA

			B	D	C	A	B	A	y
		0	1	2	3	4	5	6	j
	0	0	0	0	0	0	0	0	
A	1	0	0 N	0 N	0 N	1 NW	1 W	1 NW	
B	2	0	1 NW	1 W	1 W	1 N	2 NW	2 W	
C	3	0	1 N	1 N	2 NW	2 W	2 N	2 N	
B	4	0	1 NW	1 N	2 N	2 N	3 NW	3 W	
D	5	0	1 N	2 NW	2 N	2 N	3 N	3 N	
A	6	0	1 N	2 N	2 N	3 NW	3 N	4 NW	
B	7	0	1 NW	2 N	2 N	3 N	4 NW	4 N	
x	i								

Maior subsequência comum a x e y calculada: B C B 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 $y_1 \ldots y_j$; e
- 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)

```
if i = 0 or j = 0 then
  return
if b[i, j] = NW then
  PRINT-LCS(b, x, i - 1, j - 1)
  print x[i]
else if b[i, j] = N then
  PRINT-LCS(b, x, i - 1, j)
else
  PRINT-LCS(b, x, i, j - 1)
```

Grafos

Grafos

Orientados ou **não orientados**

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

$G = (V, E)$

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 $(u, v) \in E$

- O nó v diz-se adjacente ao nó u

Num grafo 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

Num grafo 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

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 $v_i \in V$

$$v_0 \, v_1 \, \ldots \, v_k$$

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 $i \neq j$

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

Conectividade (1)

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

G é conexo se existir um caminho entre quaisquer dois nós

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

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

Conectividade (2)

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

G é fortemente conexo se existir um caminho de qualquer nó para qualquer outro nó

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

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

Representação / Implementação

Listas de adjacências

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

Matriz de adjacências

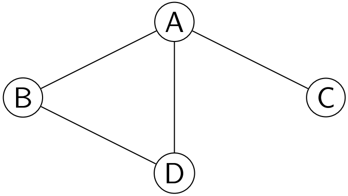
- Grafos densos $\quad (\,|E| = O(V^2)\,)$
- Permite verificar rapidamente se $(u, v) \in E$
- Complexidade espacial $\;O(V^2)$

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

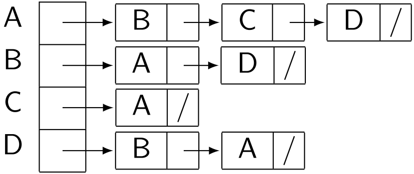
Representação / Implementação

Exemplos

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



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

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

BFS(G, s)

```
for each vertex u in G.V - {s} do
    u.color <- WHITE
    u.d <- INFINITY
    u.p <- NIL
s.color <- GREY
s.d <- 0
s.p <- NIL
Q <- EMPTY                          // queue
ENQUEUE(Q, s)
while Q != EMPTY do
    u <- DEQUEUE(Q)                 // explore next vertex
    for each vertex v in G.adj[u] do
        if v.color = WHITE then
            v.color <- GREY
            v.d <- u.d + 1
            v.p <- u
            ENQUEUE(Q, v)
    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 $O(V + E)$

Percurso em profundidade

DFS(G)

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

DFS-VISIT(G, u)

```
time <- time + 1                // white vertex u has just
u.d <- time                     // been discovered
u.color <- GREY
for each vertex v in G.adj[u] do // explore edge (u, v)
    if v.color = WHITE then
        v.p <- u
        DFS-VISIT(G, v)
u.color <- BLACK                // blacken u; it is finished
time <- time + 1
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, DFS corre em tempo

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

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

Ordenação topológica

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

Ordem

Se existe um arco de u para v, u está antes de v na ordenação 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

Ordenação topológica

Adaptação de DFS

G – grafo orientado acíclico (DAG)

TOPOLOGICAL-SORT(G)

```
for each vertex u in G.V do
    u.color <- WHITE
L <- EMPTY                              // list
for each vertex u in G.V do
    if u.color = WHITE then
        DFS-VISIT'(G, u)
return L
```

DFS-VISIT'(G, u)

```
u.color <- GREY
for each vertex v in G.adj[u] do
    if v.color = WHITE then
        DFS-VISIT'(G, v)
u.color <- BLACK
LIST-INSERT-FIRST(L, u)
```

Ordenação topológica

Outro algoritmo

TOPOLOGICAL-SORT'(G)

```
for each vertex u in G.V do
    u.i <- 0                        // incident edges
for each edge (u,v) in G.E do
    v.i <- v.i + 1
L <- EMPTY                          // list
S <- EMPTY                          // set
for each vertex u in G.V do
    if u.i = 0 then
        SET-INSERT(S, u)
while S != EMPTY do
    u <- SET-DELETE(S)              // take any vertex from S
    for each vertex v in G.adj[u] do
        v.i <- v.i - 1
        if v.i = 0 then
            SET-INSERT(S, v)
    LIST-INSERT-LAST(L, u)
return L
```

Árvore de cobertura mínima

Minimum(-weight) spanning tree

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

Uma árvore de cobertura de G é um subgrafo $G' = (V, E')$ de G, com $E' \subseteq E$

- conexo e
- acíclico (é uma árvore)

(Retirando qualquer arco de G', obtém-se um grafo não conexo)

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

Se $w(G')$ for a soma dos pesos dos arcos de G', para qualquer árvore de cobertura G'' de G tem-se

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

Árvore de cobertura mínima

Algoritmo de Prim

G – grafo pesado não orientado conexo

MST-PRIM(G, w, r)

```
for each vertex u in G.V do
    u.key <- INFINITY          // cost of adding u
    u.p <- NIL
r.key <- 0
Q <- G.V                       // priority queue
while Q != EMPTY do
    u <- EXTRACT-MIN(Q)
    for each vertex v in G.adj[u] do
        if v in Q and w(u,v) < v.key then
            v.p <- u
            v.key <- w(u,v)    // decrease key in Q
```

Árvore de cobertura mínima

Algoritmo de Kruskal

G – grafo pesado não orientado conexo

MST-KRUSKAL(G, w)

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

Análise da complexidade do algoritmo de Kruskal (1)

Linha

3 Construção da partição

$$\text{MAKE-SETS}$$

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*)

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

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

8 $2 \times \text{FIND-SET}$

10 Executada $|V| - 1$ vezes

$$\text{UNION}$$

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)

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

FIND-SET(P, i)

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

UNION(P, i, j)

```
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)

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

```
ri <- FIND-SET(P, i)
rj <- FIND-SET(P, j)
if (P[rj] < P[ri])    // j's set tree taller than i's
    P[ri] <- rj
else
    if (P[rj] == P[ri])     // both heights are equal
        P[ri] <- P[ri] + 1
    P[rj] <- ri
```

Compressão de caminho

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

```
if P[i] < 0 then
    return i
P[i] <- FIND-SET-WITH-PATH-COMPRESSION(P, P[i])
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]
$2 \times \lvert E \rvert \times$ FIND-SET	$O(EV)$	$O(E \log V)$	
$(\lvert V \rvert - 1) \times$ 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

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

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

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

Caminhos mais curtos

Subrotinas comuns aos diversos algoritmos

INITIALIZE-SINGLE-SOURCE(G, s)

```
for each vertex v in G.V do
    v.d <- INFINITY
    v.p <- NIL
s.d <- 0
```

RELAX(u, v, w)

```
if v.d > u.d + w(u,v) then
    v.d <- u.d + w(u,v)
    v.p <- u
```

Caminhos mais curtos a partir de um vértice

DAGs

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

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

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

```
INITIALIZE-SINGLE-SOURCE(G, s)
S <- EMPTY
Q <- G.V                          // priority queue (key: u.d)
while Q != EMPTY do
    u <- EXTRACT-MIN(Q)
    S <- S + {u}
    for each vertex v in G.adj[u] do
        RELAX(u, v, w)            // may affect v.d (and Q)
```

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)

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

Caminhos mais curtos entre cada dois vértices de um grafo

All-pairs shortest paths

Problema

Como calcular os caminhos mais curtos entre cada dois vértices de um grafo pesado (orientado ou não)

Soluções

- Aplicar um dos algoritmos anteriores a partir de cada um dos vértices
- ...?

Caminhos mais curtos entre cada dois vértices de um grafo

Algoritmo de Floyd-Warshall

Os vértices intermédios de um caminho simples $v_1\ v_2\ \ldots\ v_l$ são os vértices $\{v_2, \ldots, v_{l-1}\}$

Seja $G = (V, E)$ um grafo pesado, com $V = \{1, 2, \ldots, n\}$

Seja p um caminho mais curto do vértice i para o vértice j, cujos vértices intermédios estão contidos em $\{1, 2, \ldots, k\}$

- Se k não é um nó intermédio de p, os nós intermédios de p estão contidos em $\{1, 2, \ldots, k-1\}$
- Se k é um nó intermédio de p, então p pode decompor-se num caminho p_1 de i para k e num caminho p_2 de k para j
  - Os nós intermédios de p_1 e de p_2 estão contidos em $\{1, 2, \ldots, k-1\}$ (porque p é um caminho simples)
  - p_1 e p_2 são caminhos mais curtos de i para k e de k para j, respectivamente

Caminhos mais curtos entre cada dois vértices de um grafo

Função recursiva

w_{ij}: matriz de adjacências do grafo

$$w_{ij} = \begin{cases} 0 & \text{se} \quad i = j \\ w(i,j) & \text{se} \quad i \neq j \wedge (i,j) \in E \\ \infty & \text{se} \quad i \neq j \wedge (i,j) \notin E \end{cases}$$

$d_{ij}^{(k)}$: peso de um caminho mais curto de i para j com nós intermédios contidos em $\{1, 2, \ldots, k\}$

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{se} \quad k = 0 \\ \min\left\{d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right\} & \text{se} \quad k \geq 1 \end{cases}$$

Caminhos mais curtos entre cada dois vértices de um grafo

Cálculo iterativo de $d_{ij}^{(k)}$

FLOYD-WARSHALL-1(w)

```
n <- w.rows
d^(0) <- w
for k <- 1 to n do
  let d^(k)[1..n,1..n] be a new matrix
  for i <- 1 to n do
    for j <- 1 to n do
      d^(k)[i,j] <-
        min(d^(k-1)[i,j], d^(k-1)[i,k] + d^(k-1)[k,j])
return d^(n)
```

Complexidade temporal $O(V^3)$

Complexidade espacial $O(V^3)$

Caminhos mais curtos entre cada dois vértices de um grafo

Cálculo iterativo melhorado

FLOYD-WARSHALL(w)

```
n <- w.rows
d <- w
for k <- 1 to n do
  for i <- 1 to n do
    for j <- 1 to n do
      if d[i,k] + d[k,j] < d[i,j] then
        d[i,j] <- d[i,k] + d[k,j]
return d
```

Complexidade temporal $O(V^3)$

Complexidade espacial $O(V^2)$

Caminhos mais curtos entre cada dois vértices de um grafo

Predecessores

O predecessor de v_j no caminho $q = v_i \dots v_j$

- Não existe, se $q = v_j$
- É v_i, se $q = v_i\ v_j$
- É o predecessor de v_j no caminho $v_k \dots v_j$, se

$$q = v_i \dots v_k \dots v_j$$

π_{ij} é o predecessor de v_j num caminho mais curto de v_i para v_j

$$\pi_{ij} = \begin{cases} \text{NIL} & \text{se} \;\; i = j \\ \pi_{kj} & \text{se} \;\; \text{um caminho mais curto de } i \text{ para } j \text{ é } v_i \dots v_k \dots v_j \\ \text{NIL} & \text{se} \;\; d_{ij} = \infty \end{cases}$$

Caminhos mais curtos entre cada dois vértices de um grafo

Inclusão do cálculo dos predecessores

FLOYD-WARSHALL(w)

```
n <- w.rows
d <- w
let p[1..n,1..n] be a new matrix
for i <- 1 to n do
  for j <- 1 to n do
    if i = j or w[i,j] = ∞ then
      p[i,j] <- NIL
    else
      p[i,j] <- i
for k <- 1 to n do
  for i <- 1 to n do
    for j <- 1 to n do
      if d[i,k] + d[k,j] < d[i,j] then
        d[i,j] <- d[i,k] + d[k,j]
        p[i,j] <- p[k,j]
return d and p
```

Caminho mais curto entre dois vértices

Reconstrução do caminho

PRINT-ALL-PAIRS-SHORTEST-PATH(p, i, j)

Exercício

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	$O(VE)$
Algoritmo de Floyd-Warshall	$O(V^3)$

Pressupostos

Grafo representado através de listas de adjacências (excepto algoritmos de Bellman-Ford e de Floyd-Warshall)

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

Teoria da complexidade

Computabilidade

O problema do caixeiro viajante

The travelling salesman problem (TSP)

Enunciado

Dados um conjunto de n cidades, as distâncias entre elas e uma cidade de partida, qual o circuito mais curto que percorre todas as cidades e volta à cidade de partida?

Facto

Não é conhecido nenhum algoritmo que resolva o problema que corra em tempo polinomial em n

Problema de decisão

Um problema de decisão é um problema cujas instâncias têm resposta sim ou não

Problema do caixeiro viajante formulado como problema de decisão

Existe um circuito com comprimento não superior a um valor k dado que percorre todas as cidades e volta à cidade de partida?

É tão difícil quanto o problema original

Se houver um algoritmo polinomial que o resolva, o problema original poderá ser resolvido usando para k os valores 1, 2, 4, 8, ..., até obter uma resposta positiva, e, depois, recorrendo a uma pesquisa binária para encontrar o menor valor de k para o qual a resposta ao problema de decisão é sim

Classes de complexidade

P (*polynomial time*)

Classe dos problemas (de decisão) para os quais existe um algoritmo que corre em tempo polinomial

NP (*nondeterministic polynomial time*)

Classe dos problemas (de decisão) para os quais existe um algoritmo não determinista que corre em tempo polinomial

Equivalentemente, classe dos problemas para os quais é possível verificar uma solução das respectivas instâncias em tempo polinomial

O TSP pertence a **NP**

$$\mathbf{P} \subseteq \mathbf{NP}$$

Problemas tratáveis e intratáveis

Os problemas em **P** são considerados problemas tratáveis

Os algoritmos que resolvem as respectivas instâncias correm em tempo *razoável*

Os problemas para os quais não existe um algoritmo polinomial são considerados intratáveis

Um problema NP-completo é um problema em **NP** "tão difícil" quanto qualquer outro problema em **NP**

Problemas NP-completos

Exemplos

Encontrar o caminho simples mais longo (com maior peso) entre dois vértices de um grafo

Determinar se num grafo há um caminho simples com pelo menos k arcos entre dois vértices

Circuito de Hamilton Determinar se existe um ciclo simples num grafo não orientado que contém todos os nós do grafo

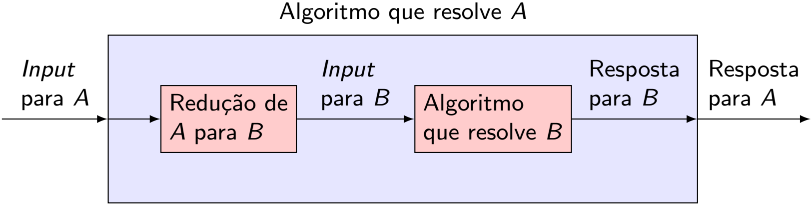
> Circuito de Euler Determinar se existe um ciclo num grafo não orientado que passa por todos os arcos está em **P**

Satisfazibilidade 3-CNF Determinar se uma fórmula lógica da forma $C_1 \wedge C_2 \wedge \ldots \wedge C_k$, onde cada C_i tem a forma $l_{i_1} \vee l_{i_2} \vee l_{i_3}$, com $l_{i_j} = v_m$ ou $l_{i_j} = \neg v_m$, para algum m, é satisfazível

> Satisfazibilidade 2-CNF Se cada C_i for da forma $l_{i_1} \vee l_{i_2}$, o problema está em **P**

Redução de problemas

O problema A pode ser reduzido ao problema B se qualquer instância de A puder ser expressa como uma instância de B cuja resposta é a resposta à instância de A



Se A pode ser reduzido em tempo polinomial a B e se existe um algoritmo polinomial que resolve B, então existe um algoritmo polinomial que resolve A

Se A pode ser reduzido em tempo polinomial a B e se A é NP-completo, então B é NP-*hard*; se $B \in \textbf{NP}$ então B é NP-completo

Problema da Terminação (*Halting Problem*) (1)

Enunciado O programa p termina quando corre com dados d?

Seja *termina* a função

$$termina(p, d) = \begin{cases} \text{true} & \text{se } p \text{ termina quando} \\ & \text{corre com dados } d \\ \text{false} & \text{se } p \text{ não termina quando} \\ & \text{corre com dados } d \end{cases}$$

e seja `t` o programa que implementa a função *termina*: quando corrido com dados `(p,d)`, o resultado de `t` é

- `true` se o programa `p` termina quando corre com dados `d`
- `false` no caso contrário

Problema da Terminação (*Halting Problem*) (2)

Recorrendo ao programa `t`, constrói-se o programa `t'` seguinte:

```
read P;             (P conterá o programa p)
T := t (P,P);       (chama t(p,p))
while T do
  skip;
R := true;
write R
```

`t'(p)` tem o seguinte comportamento

- se o resultado de `t(p,p)` é `true`, `t'(p)` não termina
- se o resultado de `t(p,p)` é `false`, o resultado de `t'(p)` é `true`

Problema da Terminação (*Halting Problem*) (3)

Qual o resultado de `t'(t')`?

- Se `t'(t')` termina, então o resultado de `t(t',t')` é `true` e `t'(t')` não termina
- Se `t'(t')` não termina, então o resultado de `t(t',t')` é `false` e o resultado de `t'(t')` é `true`

Há uma contradição em ambos os casos!

O programa `t` não existe

O problema da terminação é indecidível

A função *termina* é não computável

Redução de problemas e indecidibilidade

Se A pode ser reduzido a B e se A é um problema indecidível, então B também é indecidível

Outros problemas indecidíveis

A variável v é inicializada durante a execução de um programa?

A função f é chamada durante a execução de um programa?

O programa escreve o valor k?