Desenho e Análise de Algoritmos – Alguns Apontamentos

Ana Paula Tomás

Departamento de Ciência de Computadores

Faculdade de Ciências da Universidade do Porto

Setembro 2013 (Versão Draft - Revisto Dezembro 2013)

Conteúdo

Capítulo 1

Escrita de Algoritmos em Pseudocódigo

1.1 Descrição da linguagem

Constantes: inteiros, números reais, caracteres e sequências de caracteres constantes. Usaremos a notação `'A'`, `'B'`, `'9'`, ..., para representar o código dos caracteres `A`, `B`, `9`, ...e `"AB9"` para representar a sequência de caracteres `AB9`.

Variáveis: representadas por sequências de letras ou dígitos que começam por uma letra. Podem ser *simples* – por exemplo, $maior$, $Aux1$, y, ...; *dimensionadas* – por exemplo *arrays* unidimensionais e bidimensionais (abstrações de vetores e matrizes), ou estruturas mais complexas (a definir mais adiante). **Salvo indicação em contrário**, convencionamos que a primeira posição de um vetor x é $x[0]$, a segunda $x[1]$, ... Se x for um vetor, i uma variável simples, $x[i]$ refere a posição de índice i do vetor x. Se mat for uma matriz, i e j variáveis simples, $mat[i, j]$ refere o elemento que está na linha i e na coluna j (também indexadas a partir de 0).

Expressões Aritméticas: definidas à custa de constantes e/ou variáveis, usando operadores binários $+$, $-$, $/$ e $*$ (soma, diferença, quociente e produto) e operador unário $-$ (sinal $-$). Poderá ainda ser usado $\%$ para designar o resto da divisão inteira.

Condições: – definidas à custa de expressões, operadores relacionais $=$, $\neq$, $<$, $>$, $\geq$, e $\leq$, e operadores lógicos $\neg$ (negação), $\wedge$ (conjunção) e $\vee$ (disjunção). Convencionamos que a análise das condições que envolvem conjunções ou disjunções é feita da esquerda para a direita e as expressões só são avaliadas se ainda puderem alterar a decisão sobre a veracidade ou falsidade da condição.

Instruções: – ver Tabelas 1.1 e 1.2.

Programas (ou algoritmos) – sequências de instruções.

Tabela 1.1: A Linguagem

Sintaxe	**Semântica**
variável $\leftarrow$ expressão;	Avaliar a *expressão* e colocar o seu valor na *variável*. $x \leftarrow 3 * 2;$ — equivale $x \leftarrow 6;$ $maior \leftarrow y;$ — copia valor em y para $maior$ $y \leftarrow'$ a$' -'$ A$';$ — coloca $97 - 65 = 32$ em y
Se (condição) **então** { instruções }	Avalia a *condição*. Se for verdade, executa o bloco de *instruções*. Senão, passa à instrução seguinte. Se ($m[i] \neq 0$) então { $m[j] \leftarrow m[i];$ $j \leftarrow j + 1;$ }
Se (condição) **então** { instruções 1 } **senão** { instruções 2 }	Avalia a *condição*. Se for verdade, executa o bloco de *instruções 1*. Senão, executa *instruções 2*. Se ($m[i] \neq 0$) então { $m[j] \leftarrow m[i];$ } senão $m[j] \leftarrow 2;$

Enquanto (condição) **fazer**
{ instruções }
	Avalia a *condição*. Se for verdade executa *instruções*. Depois, volta a testar a *condição*. Se ainda for verdadeira, volta a executar as *instruções*, e procede analogamente até a *condição* ser falsa.
	Enquanto ($m[i] \neq 0$) fazer {
	$m[j] \leftarrow m[i]; \quad j \leftarrow j + 1; \quad i \leftarrow i + 1;$
	}

Repita { instruções } **até** (condição);	Executa *instruções*. Depois testa a *condição*. Se for falsa, volta a executar as *instruções*. Volta a testar,..., até a *condição* ser satisfeita. Repita { $n \leftarrow n + 1;$ $i \leftarrow i + 1;$ } até ($m[i] = 0$);

Tabela 1.2: A Linguagem (cont)

Sintaxe	**Semântica**

Para $var \leftarrow inicio$ **até** fim **fazer**
{ instruções }
 	
Para $var \leftarrow inicio$ **até**
fim **com passo** k **fazer**
{ instruções }
	Equivale a:
	$var \leftarrow inicio$;
	Enquanto ($var \leq fim$) fazer {
	instruções
	$var \leftarrow var + 1$; /* no 2ºcaso: $var \leftarrow var + k$; */
	}

ler(variável**)**;	lê (input) valor e coloca na *variável*. ler(N) | N fica com o valor dado pelo utilizador
escrever(expressão**)**; **escrever(**string**)**;	escreve (output) o valor da *expressão* escreve a sequência de caracteres. escrever($Aqui$); | escreve valor de $Aqui$ escrever($m[i]$); | escreve valor de $m[i]$ escrever("*Aqui*="); | escreve (a sequência) *Aqui*=
parar	terminar a execução.
$/\star \ldots \star/$	$/\star$ anotar comentarios $\star/$ | instrução não executável

Por vezes, na indicação de um bloco de instruções, omitiremos as chavetas mas passamos a considerar que a **indentação é relevante**. Assim, por exemplo, os dois excertos serão equivalentes.

```
Enquanto (m[i] ≠ 0) fazer {
    m[j] ← m[i];
    j ← j + 1;
    i ← i + 1;
}
```

```
Enquanto (m[i] ≠ 0) fazer
    m[j] ← m[i];
    j ← j + 1;
    i ← i + 1;
```

Representação de funções. Para representar **funções**, usamos a notação *nome(Arg1,Arg2,...,ArgN)* e consideramos que as variáveis simples são passadas por valor e as restantes por referência. Usaremos "**retorna** *expressão;*", sem aspas, para as instruções de retorno.

Capítulo 2

Provas de Correção de Algoritmos

Definição 1 *Um algoritmo* **resolve corretamente um dado problema** *se* resolve corretamente qualquer instância do problema. *Uma* instância *é um exemplo concreto que satisfaz as condições definidas no enunciado do problema.*

Não estamos interessados em algoritmos que determinam a resposta correta para um exemplo concreto do problema mas sim para todos os possíveis exemplos (i.e., para todas as instâncias do problema).

Vamos analisar alguns problemas e mostrar a correção dos algoritmos apresentados para sua resolução.

2.1 Determinar a posição da primeira ocorrência do máximo

Problema 1

Escrever uma função POSMAX(v, k, n) para determinar o índice da primeira posição que contém o elemento máximo de um vetor v de n inteiros, restringindo a análise aos elementos $v[k], v[k+1], \ldots, v[n]$. Os elementos são indexados de 1 a n. Se k for maior do que n ou menor do que 1, a função retorna -1.

Algoritmo:

POSMAX(v, k, n)
- Se $(k < 1 \vee k > n)$ então
  - retorna -1;
- $pmax \leftarrow k$;
- Para $i \leftarrow k + 1$ até n fazer
  - Se $v[i] > v[pmax]$ então
    - $pmax \leftarrow i$;
- retorna $pmax$;

Prova da correção do algoritmo: No caso em que k é maior do que n ou menor do que 1, a condição da primeira instrução é satisfeita, e a função executa "retorna -1", dando como resultado o valor pretendido.

Se $1 \leq k \leq n$, a execução prossegue na segunda instrução (i.e., em $pmax \leftarrow k$). Vamos provar que, nesse caso, o valor que a função determina é também o que se pretende. Para isso, vamos caracterizar o estado da variável $pmax$ quando a condição de ciclo é testada para cada valor de i, com $i \geq k+1$.

> **Invariante de ciclo:** Quando a condição de paragem do ciclo é testada para um dado valor de i, o valor de $pmax$ é o índice da primeira ocorrência do máximo de $v[k], v[k+1], \ldots, v[i-1]$, sendo isto verdade para todo valor de i tal que $k+1 \leq i \leq n+1$.

Prova do invariante: No ínicio do ciclo, $pmax = k$ e $i = k+1$, sendo verdade que $pmax$ guarda o índice da primeira ocorrência do máximo da sequência $v[k], v[k+1], \ldots, v[i-1]$, já que esta sequência se reduz a $v[k]$. Vamos mostrar agora que o invariante é preservado em cada iteração do ciclo, ou seja, que se se verificar para um certo valor i_0 da variável i se verifica para o valor seguinte de i, isto é, para $i = i_0 + 1$. Em cada iteração, é executada a sequência de instruções seguinte.

Se $v[i] > v[pmax]$ então $pmax \leftarrow i$;
$i \leftarrow i + 1$;

Pela hipótese sobre o estado das variáveis no momento em que a condição de paragem é testada para $i = i_0$, deduzimos que, quando se voltar a testar essa condição: $pmax$ terá o valor i_0 se $v[i_0] > \max(v[k], v[k+1], \ldots, v[i_0-1])$ ou o valor que tinha se $v[i_0] \leq \max(v[k], v[k+1], \ldots, v[i_0-1])$, ou seja, é verdade que $pmax$ terá o índice da primeira ocorrência do máximo de $v[k], v[k+1], \ldots, v[i_0-1], v[i_0]$; e i terá o valor i_0+1. Assim, a condição enunciada sobre o estado das variáveis verifica-se para $i = i_0 + 1$ se se verificar para $i = i_0$.

Tendo provado que a propriedade se verificava para $i = k+1$ e que era hereditária, concluimos, por indução matemática, que a propriedade se verifica para todo o valor de $i \geq k+1$. Assim, como o ciclo termina quando se testa a condição de paragem para $i = n+1$, podemos afirmar que, à saída do ciclo, $pmax$ tem o índice da primeira ocorrência do máximo de $v[k], v[k+1], \ldots, v[(n+1)-1]$, ou seja, o índice da primeira ocorrência do máximo de $v[k], v[k+1], \ldots, v[n]$. A seguir, a função retorna o valor de $pmax$. Logo, retorna o valor pretendido. □

2.2 Ordenar um vetor (*selection sort*)

Problema 2

Ordenar as primeiras n posições do vetor v por ordem decrescente, supondo que estas são indexadas a partir de 1.

Algoritmo: (usando a função POSMAX definida acima)

SELECTIONSORT(v, n)
  Para cada $k \leftarrow 1$ até $n-1$ fazer
    $j \leftarrow$ POSMAX(v, k, n);
    Se $j \neq k$ então
      $aux \leftarrow v[k]$;
      $v[k] \leftarrow v[j]$;
      $v[j] \leftarrow aux$;

Prova da correção do algoritmo: Nesta prova vamos assumir que POSMAX(v, k, n) determina corretamente o índice da primeira ocorrência do máximo de $v[k], v[k+1], \ldots, v[n]$, uma vez que já o demonstrámos anteriormente.

> **Invariante de ciclo:** Para todo $i \geq 1$, imediatamente após a i-ésima iteração do ciclo "Para" (e incremento de k), o valor da variável k é $i+1$, o vetor v contém exatamente os mesmos elementos que tinha antes da entrada no ciclo, embora possam estar em posições distintas, $v[1] \geq v[2] \geq \ldots \geq v[i]$ e, se $i < n$ então $v[i] \geq \max(v[i+1], \ldots, v[n])$.

Prova do invariante (por indução matemática): Vamos assumir que $n \geq 2$ porque, caso contrário, não realizaria qualquer iteração do ciclo e v estaria ordenado.

No início da primeira iteração, $k = 1$ e, depois de executar $j \leftarrow$ POSMAX$(v, 1, n)$, a variável j fica com o índice da primeira ocorrência do máximo de $v[1], \ldots, v[n]$. Na instrução seguinte, se $j \neq 1$, troca entre $v[1]$ com $v[j]$. Portanto, a propriedade enunciada verifica-se na primeira iteração: os elementos do vetor são preservados mas, se necessário, foi efetuada uma troca entre dois elementos para garantir que $v[1] \geq \max(v[2], \ldots, v[n])$.

Suponhamos agora, como hipótese de indução, que a propriedade se verifica na i-ésima iteração, para $i \geq 1$ fixo, e vamos mostrar que então se verificará na $(i+1)$-ésima. No início da $(i+1)$-ésima iteração, o valor de k é $i+1$ e, consequentemente, j fica com POSMAX$(v, i+1, n)$, ou seja, o índice da primeira ocorrência do elemento máximo de $v[i+1], \ldots, v[n]$. A seguir, os elementos $v[i+1]$ e $v[j]$ trocam entre si se $j \neq i+1$, preservando os elementos de v ainda que em posições distintas. Portanto, o elemento que fica na posição $v[i+1]$ é maior ou igual ao $\max(v[i+2], \ldots, v[n])$ e, pela hipótese, podemos concluir que $v[i] \geq v[i+1]$ para o novo valor de $v[i+1]$. Logo, $v[1] \geq v[2] \geq \ldots \geq v[i] \geq v[i+1]$ e $v[i+1] \geq \max(v[i+2], \ldots, v[n])$ (se $i+2 \leq n$). □

O ciclo termina depois de efetuar a $(n-1)$-ésima iteração, porque $k = n$. Da propriedade provada conclui-se que, nesse instante, $v[1] \geq v[2] \geq \ldots \geq v[n-1]$ e $v[n-1] \geq v[n]$, e v contém os elementos que tinha no ínicio, ainda que possam estar em posições diferentes. Portanto, o vetor v ficou ordenado por ordem decrescente.

O algoritmo de ordenação apresentado é conhecido por **método de ordenação por seleção** (*selection sort*), sendo, neste caso, por seleção do máximo

2.3 Contar o número de ocorrências de um valor num vetor

Problema 3

Definir uma função CONTAOCORRENCIAS(v, n, x) para calcular o número de ocorrências do valor x nos n primeiros elementos de um vetor v (de inteiros), supondo que o primeiro elemento é $v[0]$.

Algoritmo:

```
CONTAOCORRENCIAS(v, n, x)
    t ← 0;
    i ← 0;
    Enquanto (i < n) fazer
        Se (v[i] = x) então
            t ← t + 1;
        i ← i + 1;
    retorna t;
```

Prova da correção do algoritmo: Para mostrar a correção do algoritmo, é útil mostrar o seguinte *invariante de ciclo* (i.e., propriedade preservada pelo ciclo):

> Quando a condição de ciclo é testada pela $(i+1)$-ésima vez, o valor de t é o número de ocorrências de x na sequência $v[0], \ldots, v[i-1]$, qualquer que seja $i \geq 0$.

Note que, por convenção de notação, $v[0], \ldots, v[i-1]$ designa uma sequência vazia se $i - 1 < 0$, o que acontece quando $i = 0$. Nesse caso, o número de ocorrências de x é zero e é esse o valor de t quando $i = 0$ (ou seja, no ínicio da primeira iteração do ciclo).

Os detalhes da prova do invariante ficam ao cuidado do leitor. O ciclo termina quando $i = n$, ou seja, quando a condição de ciclo é testada pela $(n+1)$-ésima vez. De acordo com o invariante de ciclo enunciado, nessa altura, o valor de t é o número de ocorrências de x em $v[0], \ldots, v[n-1]$, pelo que, ao executar "retorna t", a função retorna o valor pretendido.

2.4 Contar o número de ocorrências de cada valor de um intervalo

Problema 4

Escrever uma função CONTAOCORRENCIASINTERVALO(v, n, a, b, r) para obter o número de ocorrências de cada um dos inteiros do intervalo $[a, b]$ nos n primeiros elementos de um vetor v (de inteiros), supondo que o primeiro elemento é $v[0]$, a e b são inteiros e $a \leq b$, ficando $r[0], r[1], \ldots, r[b-a]$ com o número de ocorrências de $a, a+1, \ldots, b$, respetivamente.

Algoritmo (trivial): Usando a função CONTAOCORRENCIAS definida acima, bastaria percorrer o vetor várias vezes: de cada vez, contaria o número de ocorrências de um dos inteiros do intervalo (se j for esse inteiro, guardaria o resultado em $r[j-a]$).

```
CONTAOCORRENCIASINTERVALO(v, n, a, b, r)
    Para j ← a até b fazer
        r[j − a] ← CONTAOCORRENCIAS(v, n, j);
```

Para concluir que a atribuição é feita corretamente, importa notar que, quaisquer que sejam a e b inteiros com $a \leq b$, se $j \in [a, b]$ então $j - a \in [0, b-a]$, com a seguinte correspondência entre j e $j-a$.

j	a	$a+1$	$a+2$	$\ldots$	b
$j-a$	0	1	2	$\ldots$	$b-a$

Este algoritmo é simples mas não é eficiente, porque analisa $b-a+1$ vezes o vetor e bastaria analisá-lo uma vez para dar a resposta. Recorde que $b-a+1$ é o número de elementos do intervalo $[a, b]$.

Algoritmo (mais eficiente):

```
CONTAOCORRENCIASINTERVALO(v, n, a, b, r)
    Para j ← 0 até b − a fazer
        r[j] ← 0;
    Para j ← 0 até n − 1 fazer
        Se (v[j] ≥ a ∧ v[j] ≤ b) então
            r[v[j] − a] ← r[v[j] − a] + 1;
```

O primeiro ciclo inicializa a zero todos os contadores relevantes, isto é, $r[j]$ para $j \in [0, b-a]$. O segundo ciclo, percorre o vetor v, considerando as n primeiras posições: para $j \in [0, n-1]$, analisa $v[j]$ e se $v[j] \in [a, b]$ então incrementa o contador de ocorrências de $v[j]$, o qual está na posição de índice $v[j] - a$ no vetor r.

Mais adiante veremos que, sendo m o número de valores do intervalo $[a, b]$, isto é, $m = b - a + 1$, o primeiro algoritmo tem complexidade temporal $\Theta(mn)$, e o segundo tem complexidade $\Theta(m + n)$.

Em particular, por exemplo, se soubessemos que m era sempre $\Theta(n)$ em qualquer instância que pretendessemos resolver, então o primeiro algoritmo teria complexidade $\Theta(n^2)$ e o segundo teria complexidade $\Theta(n)$. Contudo, se em qualquer instância que pretendessemos resolver m fosse $\Theta(1)$, i.e, limitado por uma constante fixa, então ambos teriam exatamente a mesma complexidade assintótica, seriam $\Theta(n)$.

2.5 Determinar o máximo de uma sequência lida da entrada padrão

Problema 5

Imprimir o máximo de 1000 inteiros a indicar pelo utilizador.

Algoritmo:

ler($maximo$);
$contagem \leftarrow 1$;
Repita {
    ler($valor$);
    Se ($valor > maximo$) então $maximo \leftarrow valor$;
    $contagem \leftarrow contagem + 1$;
} até ($contagem = 1000$);
escrever($maximo$);

Prova de correção do algoritmo:

Seja $a_1, a_2, \ldots, a_{1000}$ a sequência de valores que o utilizador indicará, sendo estes dados à medida que forem sendo pedidos.

Vamos mostrar que, qualquer que seja $k \geq 1$, quando testa a condição de paragem do ciclo pela k-ésima vez, o estado das variáveis é: $contagem = k + 1$, $maximo = \max(a_1, \ldots, a_{k+1})$ e $valor = a_{k+1}$, e a_{k+1} foi o último valor lido.

Antes de testar a condição de paragem do ciclo pela primeira vez, executou a sequência de instruções que se encontra na tabela representada abaixo, à esquerda. Da análise efetuada à direita conclui-se que o estado final das variáveis satisfaz a condição enunciada para $k = 1$.

ler($maximo$);	$maximo \leftarrow a_1$
$contagem \leftarrow 1$;	$contagem \leftarrow 1$
ler($valor$);	$valor \leftarrow a_2$ (último lido)
Se ($valor > maximo$) então $maximo \leftarrow valor$;	$maximo \leftarrow \max(a_1, a_2)$
$contagem \leftarrow contagem + 1$;	$contagem \leftarrow 2$

Vamos agora mostrar que, se o estado das variáveis for o indicado quando se testa a condição de paragem pela k-ésima vez então, quando se testar essa condição pela $(k+1)$-ésima vez seria $contagem = k + 2$, $maximo = \max(a_1, \ldots, a_{k+1}, a_{k+2})$ e $valor = a_{k+2}$, em que a_{k+2} designa o último valor lido. Ou seja, vamos mostrar que a propriedade é preservada em cada iteração. O diagrama seguinte mostra que assim seria.

Hipótese sobre o estado no k-ésimo teste
$contagem = k + 1$
$maximo = \max(a_1, \ldots, a_{k+1})$
$valor = a_{k+1}$
a_{k+1} último lido

→

Instruções que executa se $contagem = k + 1 \neq 1000$
ler($valor$);
Se ($valor > maximo$) então $maximo \leftarrow valor$;
$contagem \leftarrow contagem + 1$;

→

O que se concluiria sobre o estado no $(k+1)$-ésimo teste
$valor \leftarrow a_{k+2}$
$maximo \leftarrow \max(maximo, a_{k+2}) = \max(a_1, \ldots, a_{k+1}, a_{k+2})$
$contagem \leftarrow (k + 1) + 1$;
a_{k+2} último lido

Logo, por indução matemática, podemos concluir que a propriedade se mantém para qualquer $k \geq 1$, enquanto o ciclo não terminar.

O ciclo pára quando $contagem = 1000$, o que acontece quando se testar a condição de ciclo pela 999-ésima vez. De acordo com a propriedade provada, nesse momento, a variável $maximo$ contém o valor máximo de $a_1, \ldots, a_{1000}$, e, consequentemente, quando executa "escrever($maximo$)", o algoritmo faz sair o resultado pretendido.

Observação: O algoritmo começa com a instrução "ler($maximo$)" porque quando se pretende determinar um máximo (ou mínimo) de uma sequência não vazia, o candidato natural para a inicialização do máximo (ou mínimo) é sempre **o primeiro valor da sequência**. Se a sequência puder ser vazia, isto é, não ter elementos, o enunciado do problema tem que definir o que seria o resultado pretendido nesse caso (e, possivelmente, o algoritmo irá tratá-lo como um caso à parte).

2.6 Determinar a soma dos dois últimos valores lidos antes do terminador

Problema 6:

Imprimir a soma dos dois últimos valores de uma sequência de valores dada pelo utilizador, sendo -1 o valor que indica que a sequência terminou. São sempre dados pelo menos dois valores antes do terminador -1, o qual não é relevante para o resultado.

Algoritmo:

ler($penultimo$);
ler($ultimo$);
ler($novo$);
Enquanto ($novo \neq -1$) fazer
    $penultimo \leftarrow ultimo$;
    $ultimo \leftarrow novo$;
    ler($novo$);
escrever($penultimo + ultimo$);

Invariante de ciclo: De cada vez que testa a condição de ciclo, as variáveis $penultimo$ e $ultimo$ têm os dois últimos valores lidos e já analisados (os nomes refletem a ordem em que foram dados) e a variável $novo$ tem o último valor dado mas ainda não analisado.

Essa propriedade verifica-se à entrada do ciclo, e é preservada em cada iteração de ciclo, pois, se $novo \neq -1$, então $novo$ substituirá o $ultimo$ mas antes de se efetuar essa substituição $penultimo$ fica com o valor de $ultimo$. Depois, é lido o valor seguinte para $novo$ e esse valor só será analisado quando se voltar a testar a condição de paragem.

O ciclo pára quando $novo = -1$. A seguir, o programa escreve a soma do último e penúltimo elementos lidos antes de -1, como se pretendia.

2.7 Compactar um vetor retirando elementos consecutivos iguais

Problema 7:

Compactar uma sequência de inteiros dada num vetor x, retirando as repetições de elementos que ocorram em posições adjacentes. Deve considerar os elementos que ocupam as n primeiras posições (sendo $n \geq 1$ já conhecido e $x[0]$ o primeiro elemento). No fim, n terá o número de elementos da versão compactada.

Observação: A compactação corresponde a um possível deslocamento de valores para a esquerda, que substituirão os valores "retirados". O algoritmo seguinte percorre o vetor uma única vez e vai construindo o resultado final por cópia dos valores relevantes para a posição final.

Algoritmo:

$pos \leftarrow 0$;
$atual \leftarrow 1$;
Enquanto $(atual < n)$ fazer
    Se $x[pos] \neq x[atual]$ então
        $pos \leftarrow pos + 1$;
        $x[pos] \leftarrow x[atual]$;
    $atual \leftarrow atual + 1$;
$n \leftarrow pos + 1$;

Invariante de ciclo: Quando **testa a condição** de paragem do ciclo pela k-ésima vez, $atual = k$ e $x[0], \ldots, x[pos]$ tem o resultado da compactação da sequência **inicial** $x[0], \ldots, x[atual - 1]$, sendo $pos < atual$, e $x[atual], \ldots, x[n-1]$ a parte de x que falta analisar.

Para facilitar a escrita da prova, designemos por $y[0], y[1], \ldots, y[n-1]$ a sequência inicial. O invariante é: "quando testa a condição de paragem do ciclo pela k-ésima vez, $atual = k$ e $x[0], \ldots, x[pos]$ tem o resultado da compactação da sequência $y[0], y[1], \ldots, y[atual-1]$, sendo $pos < atual$, e a parte de y que falta analisar é $y[atual], \ldots, y[n-1]$, e coincide com $x[atual], \ldots, x[n-1]$."

Para a prova é crucial observar que, se $y[atual] = x[pos]$ então também $y[atual-1] = x[pos]$, pois, caso contrário, o resultado da compactação de $y[0], y[1], \ldots, y[atual-1]$ não poderia ser $x[0], \ldots, x[pos]$ pois teria necessariamente $y[atual-1]$ no fim. Assim, quando no teste da condição da instrução "Se" se tem $x[atual] = x[pos]$, (i.e., se $y[atual] = x[pos]$) é correto passar $y[atual]$ à frente, só se copiando $y[atual]$ para o resultado se $y[atual] \neq x[pos]$.

Os restantes detalhes da prova do invariante ficam ao cuidado do leitor.

O ciclo pára quando testa a condição de paragem do ciclo pela n-ésima vez. De acordo com o invariante, nesse momento, $x[0], \ldots, x[pos]$ é o resultado da compactação da sequência inicial $x[0], \ldots, x[n-1]$. Logo, o número de elementos da sequência compactada é $pos + 1$, e esse é o valor que se atribui a n a seguir, como se pretendia.

2.8 Encontrar um valor num vetor estritamente ordenado

Problema 8: Escrever uma função para verificar se o valor dado em x se encontra num vetor v entre as posições de índices a e b, sabendo que o vetor está ordenado estritamente por ordem decrescente e que $0 \leq a \leq b < n$, sendo n o número de elementos do vetor e $v[0]$ o primeiro elemento. A função retorna o índice da posição em que x se encontrar ou -1 se x não estiver na secção do vetor indicada.

Algoritmo 1:

PESQUISALINEAR(v, a, b, x)

Enquanto $(a \leq b)$ fazer
    Se $(v[a] = x)$ então retorna a;
    $a \leftarrow a + 1$;
retorna -1;

Algoritmo 2:

PESQUISALINEARMELHORADA(v, a, b, x)

Enquanto $(a \leq b \wedge v[a] > x)$ fazer
    $a \leftarrow a + 1$;
Se $(a \leq b \wedge v[a] = x)$ então retorna a;
retorna -1;

Algoritmo 3:

PESQUISABINARIA(v, a, b, x)

Enquanto $(a \leq b)$ fazer
    $m \leftarrow \lfloor (a + b)/2 \rfloor$;
    Se $(v[m] = x)$ então retorna m;
    Se $(v[m] > x)$ então $a \leftarrow m + 1$;
    senão $b \leftarrow m - 1$;
retorna -1;

Algoritmo 4:

PESQUISABINARIAREC(v, a, b, x)

Se $(a > b)$ então retorna -1;
$m \leftarrow \lfloor (a + b)/2 \rfloor$;
Se $(v[m] = x)$ então retorna m;
Se $(v[m] > x)$ então
    retorna PESQUISABINARIAREC$(v, m + 1, b, x)$;
retorna PESQUISABINARIAREC$(v, a, m - 1, x)$;

Observação: Convencionamos que a análise das condições que envolvem conjunções (ou disjunções) é feita da esquerda para a direita e as expressões só são avaliadas se ainda puderem alterar a decisão sobre a validade da condição. Assim, na condição $a \leq b \wedge v[a] > x$, a execução de $v[a] > x$ só será feita se $a \leq b$. Consequentemente, se $a > b$, não se acederá a posições que não pertenciam à secção de v a analisar.

Notação: $\lfloor y \rfloor$ representa o maior inteiro que não excede y e $\lceil y \rceil$ representa o menor inteiro que não é inferior a y.

Observação: Mais adiante veremos que, se $b - a + 1 = n$, a complexidade temporal dos Algoritmos 1 e 2 é $O(n)$ e a dos Algoritmos 3 e 4 é $O(\log_2 n)$, no pior caso. O Algoritmo 1 percorre o vetor todo se não encontrar o valor enquanto que os restantes usam o facto de o vetor estar ordenado para descartar secções em que seria impossível encontrar o elemento tendo em conta outros que já encontraram. O Algoritmo 1 é o único que se pode aplicar **se o vetor não estiver ordenado** (ou se não se souber se está ou não ordenado).

Provas de correção dos Algoritmos 1, 2 e 3

De acordo com o enunciado do problema, o vetor está ordenado por ordem estritamente decrescente, pelo que, se $0 \leq a < b < n$ então $v[a] > v[a + 1] > v[a + 2] > \ldots > v[b - 1] > v[b]$.

Para a prova de correção dos Algoritmos 1, 2 e 3, designemos por a_k e b_k os valores das variáveis a e b quando a condição de ciclo é testada pela k-ésima vez, e por $I_k = [a_k, b_k]$ o intervalo de inteiros definido por a e b, e $|I_k|$

o seu número de elementos. Se $I_k = \emptyset$ (conjunto vazio) então $|I_k| = 0$, o que acontece sse $b_k < a_k$. Assim, $|I_k| = \max(0, b_k - a_k + 1)$ quaisquer que sejam a_k e b_k.

Invariante de ciclo para o Algoritmo 1: Para $k \geq 1$, quando se testa a condição de ciclo pela k-ésima vez, $a_k = a + (k-1)$, $b_k = b$, sabe-se que o índice da posição de x não é inferior a a_k e que ainda não se analisou $v[a + (k-1)], v[a + k], \ldots, v[b]$.

Conclusão: Se sair da função durante a execução do bloco de instruções do ciclo numa certa iteração k, então x foi encontrado numa posição a_k válida pois o invariante garante que $a \leq a_k \leq b$, e o valor retornado é o índice dessa posição. Se o ciclo terminar por $a_k > b_k = b$, a função retorna corretamente -1, já que o invariante garante que o índice da posição de x não pode ser inferior a a_k, e portanto terá que ser superior a b (não estando no intervalo $[a, b]$).

Invariante de ciclo para o Algoritmo 2: Para $k \geq 1$, quando se testa a condição de ciclo pela k-ésima, vez, $a_k = a + (k-1)$, $b_k = b$, sabe-se que o índice da posição de x não é inferior a a_k, que ainda não se analisou $v[a + (k-1)], v[a + k], \ldots, v[b]$ e que $v[a + (k-2)] > x$ (se $k \neq 1$).

Conclusão: O ciclo termina quando testa a condição de ciclo pela k-ésima vez se $a_k > b$ ou se $a_k \leq b \wedge v[a_k] \leq x$, sendo $a_k \geq a$. A seguir, a função retorna corretamente a_k se $a \leq a_k \leq b \wedge v[a_k] = x$. Caso contrário, retorna -1, o que é correto porque se $a \leq a_k \leq b$ então $v[a_k] < x$ e como v está ordenado então x teria de ocorrer antes da posição a_k. Mas, o invariante garante que o índice da posição de x não pode estar em $[a, a_k - 1]$ e, portanto, x não está na secção de v definida por $[a, b]$. E, se $a_k > b$, do invariante resulta também que x não está nessa secção de v.

Invariante de ciclo para o Algoritmo 3: Para $k \geq 1$, quando se testa a condição de ciclo pela k-ésima vez, sabe-se que se x ocorrer na secção $[a, b] = I_1$ que se pretendia analisar, então o índice da posição de x tem que estar em $I_k = [a_k, b_k]$, e se $k > 1$ então $|I_k| < |I_{k-1}|$.

Prova: Para $k = 1$, segue trivialmente do enunciado do problema que a propriedade se verifica, pois I_k é $[a, b]$. Suponhamos agora que se verifica para um certo k fixo e que $I_k = [a_k, b_k] \neq \emptyset$. Então, ao executar novamente o bloco de instruções do ciclo, considera uma posição $m \in I_k$ (mais concretamente, o ponto médio de I_k) e sai da função se $v[m] = x$ (retornando m) ou define I_{k+1} como $I_{k+1} = [m + 1, b_k]$ se $v[m] > x$ ou como $I_{k+1} = [a_k, m - 1]$ se $v[m] < x$, atualizando o valor das variáveis a e b consistentemente. Se não saiu, então $a_k \leq m \leq b_k$, podemos concluir que $I_{k+1} \subset I_k$ (podendo mesmo I_{k+1} ser o conjunto vazio). Logo, $|I_{k+1}| < |I_k|$. Por outro lado, como o vetor está ordenado, a atualização garante que se x não estiver na secção I_{k+1} de v então x não poderia estar na secção I_k, e, consequentemente, pela hipótese sobre I_k, não estaria na secção que se pretendida. Portanto, a propriedade é preservada em cada iteração. □

Conclusão: O ciclo terá de terminar sempre porque, em cada iteração, o número de elementos do intervalo diminui. Se sair da função na execução do bloco do ciclo, então $v[m] = x$ e a função retorna corretamente m. Se o ciclo terminar com $I_k = \emptyset$, então o invariante provado garante que x não estava em v na secção que se pretendia analisar. Nesse caso, retorna -1 corretamente também.

Prova de correção do Algoritmo 4

Algoritmo 4:

PESQUISABINARIAREC(v, a, b, x)

1. Se $(a > b)$ então retorna -1;
2. $m \leftarrow \lfloor (a + b)/2 \rfloor$;
3. Se $(v[m] = x)$ então retorna m;
4. Se $(v[m] > x)$ então retorna PESQUISABINARIAREC$(v, m + 1, b, x)$;
5. retorna PESQUISABINARIAREC$(v, a, m - 1, x)$;

Vamos mostrar que a chamada PESQUISABINARIAREC(v, a, b, x) termina e retorna o valor pretendido, qualquer que seja o número de valores do intervalo $[a, b]$, sendo $0 \leq a \leq b < n$ ou $[a, b] = \emptyset$. Faremos a prova por indução sobre o número de elementos do intervalo $[a, b]$, i.e., por indução sobre t definido por $t = \max(b - a + 1, 0)$.

Prova: Se $t = 0$, isto é, se $a > b$, a chamada PESQUISABINARIAREC(v, a, b, x) termina e retorna -1 corretamente pois, quando $a > b$, a função executa a primeira instrução e é verdade que x não está na secção vazia.

Suponhamos agora que a chamada PESQUISABINARIAREC(v, a', b', x) termina dando o valor pretendido sempre que $[a', b']$ tem **no máximo** t **elementos, para** $t \geq 0$ **fixo**, sendo a' e b' quaisquer, com $0 \leq a' \leq b' < n$ ou $[a', b'] = \emptyset$.

Para concluir a prova, vamos mostrar que, se esta hipótese se verificar, então PESQUISABINARIAREC(v, a, b, x) termina e dá o resultado correto quaisquer que sejam a e b tais que $b - a + 1 = t + 1$ e $0 \leq a \leq b < n$.

De facto, como $b \geq a$, o programa executa a instrução 2., e como $0 \leq a \leq \lfloor (a + b)/2 \rfloor \leq b < n$, o valor que atribui a m é o índice de uma posição de v e que está na secção $[a, b]$. Portanto, é correto aceder a $v[m]$. Ao executar a instrução 3., retorna m se $v[m] = x$, dando o resultado esperado. Se $v[m] \neq x$ então, como o vetor está ordenado por ordem decrescente: (i) se $v[m] > x$ então x de tem de estar depois da posição m (i.e., na secção definida por $[m+1, b]$); e (ii) se $v[m] < x$ então x tem de estar antes da posição m (i.e., na secção definida por $[a, m-1]$). Assim, é correto, no caso (i), o programa retornar o valor que a chamada PESQUISABINARIAREC$(v, m + 1, b, x)$ determinar. E, é correto, no caso (ii), retornar o valor que a chamada PESQUISABINARIAREC$(v, a, m - 1, x)$ determinar. Notar que, quer $[m + 1, b]$ quer $[a, m - 1]$ têm no máximo t elementos (de facto, têm cerca de $(t + 1)/2$ elementos) sendo, no caso (i), $0 \leq a < m + 1 \leq b < n \vee [m + 1, b] = \emptyset$ e, no caso (ii), $0 \leq a \leq m - 1 < b < n \vee [a, m - 1] = \emptyset$. Assim, podemos invocar a hipótese de indução para concluir que qualquer uma destas chamadas obteria corretamente o valor esperado (mesmo quando a secção passada na chamada é vazia). □

Observação (indução forte vs. indução fraca): Nesta prova usámos *indução forte* pois, como hipótese de indução supusemos que a propriedade se verificava quando os intervalos tinham até t elementos. A hipótese *mais fraca* de que a propriedade se verificava para intervalos que tinham exatamente t elementos não seria útil, em geral, porque o número de elementos dos intervalos $[a, m-1]$ e $[m+1, b]$ é cerca de $(t+1)/2$ e, portanto, na maioria dos casos, muito menor do que t.

Ordem não estrita: De modo análogo, podemos mostrar que se v estiver ordenado por ordem crescente não estrita, então as funções PESQUISABINARIA e PESQUISABINARIAREC encontram alguma ocorrência de x, se x ocorrer no segmento $[a, b]$ de v. Note-se também que a partir da ocorrência localizada, podemos, se necessário, aceder a todas as outras, pois são contíguas (no vetor ordenado).

Capítulo 3

Complexidade Assintótica de Algoritmos

Um critério importante na avaliação de algoritmos é o tempo que demoram e o modo como esse tempo varia quando o tamanho das instâncias cresce. Para que a comparação de eficiência faça sentido, procura-se uma estimativa do tempo de execução que seja função do comprimento dos dados mas independente da máquina, do sistema operativo, da linguagem em que o algoritmo é implementado, da versão do compilador, etc. Habitualmente, calcula-se uma estimativa do *tempo de execução no pior caso*, que é um majorante do tempo de execução para qualquer instância do problema. Se se conhecer a distribuição das instâncias, pode-se tentar caracterizar o *tempo de execução no caso médio*, que é o valor esperado para o tempo de execução, no sentido da Estatística. Pode ainda ser útil uma estimativa do *tempo de execução no melhor caso*, a qual dá um minorante do tempo de execução para qualquer instância. Assumindo um modelo para o custo (i.e., duração) de cada operação básica, o custo total (i.e., duração total) é dado por uma soma em que cada parcela é o produto do custo de uma operação básica pelo número de vezes que foi realizada.

Exemplo 1 Supondo que os valores dados são inteiros, o algoritmo abaixo lê um inteiro z e uma sequência de inteiros terminada por 0, e escreve o valor da expressão mz, sendo m o número de inteiros negativos na sequência.

```
y ← 0;
ler(z);
ler(x);
Enquanto (x ≠ 0) fazer
    Se (x < 0) então
        y ← y + z;
    ler(x);
escrever(y);
```

Ideia para justificar a correção: Seja $z, a_1, a_2, \ldots, a_n$ a sequência de valores lidos, com $a_n = 0$. Por indução sobre k, podemos mostrar que, quando a condição de ciclo $x \neq 0$ é testada pela k-ésima vez (para $k \geq 1$) a variável z guarda o primeiro valor lido, x guarda o valor a_k (último valor dado), e y guarda o valor $m_k z$, sendo m_k o número de inteiros negativos na sequência $a_1, \ldots, a_{k-1}$. O ciclo pára quando a condição é testada pela n-ésima vez, sendo $y = m_n z$ e m_n o número de inteiros negativos na sequência $a_1, \ldots, a_{n-1}$, o qual é idêntico para $a_1, \ldots, a_{n-1}, a_n$, por a_n ser 0. Logo, o valor escrito é o pretendido.

Análise do tempo de execução do algoritmo. Vamos determinar a expressão que define o tempo de execução para uma instância genérica $z, a_1, \ldots, a_n$, com $a_n = 0$, supondo que a duração da leitura de um valor é c_2, a da atribuição de uma constante a uma variável é c_1, a do teste de uma condição simples e transferência de controlo é c_3, etc.

$y \leftarrow 0$;	c_1
ler(z);	c_2
ler(x);	c_2
Enquanto ($x \neq 0$) fazer	$c_3 \times n$
Se ($x < 0$) então	$c_3 \times (n-1)$
$y \leftarrow y + z$;	$c_4 \times m$
ler(x);	$c_2 \times (n-1)$
escrever(y);	c_5

No **pior caso**, todos os valores $a_1, \ldots, a_{n-1}$ são negativos e, portanto, $m = n - 1$. Consequentemente, no pior caso,

$$\begin{aligned} T(n) &= c_1 + 2c_2 + c_3(n + n - 1) + c_4(n-1) + c_2(n-1) + c_5 \\ &= (c_1 + c_2 - c_3 - c_4 + c_5) + (2c_3 + c_4 + c_2)n \end{aligned}$$

Por outro lado, **em qualquer caso**, $T(n) > (c_2 + c_3)n$, qualquer que seja $n \geq 1$, uma vez que é necessário pelo menos ler a sequência e testar se chegou ao fim. Assim, qualquer que seja a instância de tamanho $n \geq 1$ que possa ser considerada, temos

$$(c_2 + c_3)n < T(n) \leq (c_1 + c_2 - c_3 - c_4 + c_5) + (2c_3 + c_4 + c_2)n.$$

Exprimimos a complexidade temporal dos algoritmos (e também a complexidade espacial) em termos de **ordens de grandeza**, ignorando constantes multiplicativas e o comportamento para valores de n pequenos. Interessa-nos estimar o desempenho do algoritmo para instâncias de tamanho n suficientemente grande, sendo a noção de "suficientemente grande" traduzida por uma ordem n_0, a partir da qual $T(n)$ verifica a condição enunciada sobre o seu comportamento.

3.1 Ordens de grandeza

No que se segue f e g designam funções de $\mathbb{N}$ em $\mathbb{R}$, ou seja, sucessões. As ordens de grandeza usadas para exprimir a complexidade assintótica, designadas por O ("Big-O"), Θ ("Big-Theta") e Ω ("Big-Omega"), são assim definidas.

$$\begin{aligned} O(g(n)) &= \{f(n) \mid \text{existem } c > 0 \text{ e } n_0 > 0 \text{ tais que } f(n) \leq cg(n) \text{ para todo } n \geq n_0\} \\ \Omega(g(n)) &= \{f(n) \mid \text{existem } c > 0 \text{ e } n_0 > 0 \text{ tais que } f(n) \geq cg(n) \text{ para todo } n \geq n_0\} \\ \Theta(g(n)) &= \{f(n) \mid \text{existem } c_1 > 0, c_2 > 0 \text{ e } n_0 > 0 \text{ tais que } c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ para todo } n \geq n_0\} \end{aligned}$$

Habitualmente, em Matemática, $f(n)$ designa a *imagem* de n pela função f, mas, na definição acima, $f(n)$ designa também a função $f : \mathbb{N} \to \mathbb{R}$ que a cada n associa $f(n)$. Do mesmo modo, $g(n)$ designa a função $g : \mathbb{N} \to \mathbb{R}$ que a cada n associa $g(n)$. De facto, as notações $f(n)$ e $g(n)$ estão a ser usadas com as duas interpretações. Em $f(n) \in O(g(n))$ referimos as funções f e g e em $f(n) \leq cg(n)$ estamos a referir as imagens de n por f e por g.

Em suma, dizemos que:

- $f(n)$ é de ordem $O(g(n))$ e escrevemos $f(n) \in O(g(n))$ sse $f(n)$ é majorada por $cg(n)$ para alguma constante $c > 0$, a partir de uma certa ordem (definida por n_0), ou seja,

$$f(n) \in O(g(n)) \quad \text{sse} \quad \exists c \in \mathbb{R}^+ \; \exists n_0 \in \mathbb{Z}^+ \; \forall n \geq n_0 \;\; f(n) \leq cg(n);$$

- $f(n)$ é de ordem $\Omega(g(n))$ e escrevemos $f(n) \in \Omega(g(n))$ sse $f(n)$ é minorada por $cg(n)$ para alguma constante $c > 0$, a partir de uma certa ordem (definida por n_0), ou seja,

$$f(n) \in \Omega(g(n)) \quad \text{sse} \quad \exists c \in \mathbb{R}^+ \; \exists n_0 \in \mathbb{Z}^+ \; \forall n \geq n_0 \;\; f(n) \geq cg(n);$$

- $f(n)$ é de ordem $\Theta(g(n))$ e escrevemos $f(n) \in \Theta(g(n))$ sse $f(n)$ é majorada por $c_2 g(n)$ e minorada por $c_1 g(n)$ para algum $c_1 > 0$ e algum $c_2 > 0$, a partir de uma certa ordem (definida por n_0), ou seja, ou seja,

$$f(n) \in \Theta(g(n)) \quad \text{sse} \quad \exists c_1 \in \mathbb{R}^+ \; \exists c_2 \in \mathbb{R}^+ \; \exists n_0 \in \mathbb{Z}^+ \; \forall n \geq n_0 \;\; c_1 g(n) \leq f(n) \leq c_2 g(n).$$

Recorde que:

- Sendo B um número real e $f : \mathbb{R} \to \mathbb{R}$ uma função, dizemos que f é minorada por B se e só se $B \leq f(x)$ para todo $x \in \mathbb{R}$ (ou no domínio de f). Nesse caso, B diz-se minorante de f. O mínimo global de f, se existir, é o maior dos minorantes (mínimo requer que $f(x) = B$ para algum x). Por exemplo, a função $h(x) = 1/|x - 2|$, definida em $\mathbb{R} \setminus \{2\}$, tem contradomínio $\mathbb{R}^+$, qualquer $B \in \mathbb{R}_0^-$ é minorante de h e h não tem mínimo.

- Do mesmo modo, dizemos que f é majorada por B se e só se $f(x) \leq B$ para todo $x \in \mathbb{R}$ (ou no domínio de f). Nesse caso, B diz-se majorante de f. O máximo global de f, se existir, é o menor dos majorantes (máximo requer que $f(x) = B$ para algum x). Por exemplo, qualquer $B \in \mathbb{R}_0^+$ é majorante de $t(x) = -|x - 2|$, para $x \in \mathbb{R}$, e 0 é o valor máximo de $t(x)$, sendo atingido para $x = 2$.

Exemplo 2 Para o Exemplo 1, já vimos que $(c_2 + c_3)n < T(n) \leq (c_1 + c_2 - c_3 - c_4 + c_5) + (2c_3 + c_4 + c_2)n$, qualquer que seja a instância de tamanho $n \geq 1$. Fazendo $c_2 + c_3 = C'$, temos $T(n) \geq C'n$ para todo $n \geq 1$. Portanto, $T(n) \in \Omega(n)$. Vamos agora ver que $T(n) \in O(n)$. Como $T(n) \leq (c_1 + c_2 - c_3 - c_4 + c_5) + (2c_3 + c_4 + c_2)n$, obtemos $T(n) \leq \beta + \alpha n$, para $\alpha = 2c_3 + c_4 + c_2$ e $\beta = c_1 + c_2 - c_3 - c_4 + c_5$ constantes, com $\alpha > 0$. Considerando o

comportamento da função afim, podemos concluir que se tomarmos $C > \alpha$ então existe n_0 tal que $\beta + \alpha n \leq Cn$ para todo $n \geq n_0$. Por exemplo, se fixarmos $C = \alpha + 1$, concluimos que $\beta + \alpha n \leq (\alpha + 1)n$ se e só se $n \geq \beta$. Logo, para tal $C = \alpha + 1$, bastaria tomar $n_0 = \max(1, \lceil \beta \rceil)$, para ser verdade $\forall n \geq n_0\ T(n) \leq Cn$. Portanto, $T(n) \in O(n)$.

Em alternativa, para encontrar a constante C necessária à prova de que $T(n) \in O(n)$, podiamos tomar uma duração γ maior ou igual do que a máxima elementar, ou seja, por exemplo, $\gamma = \max(c_1, c_2, c_3, c_4, c_5)$, e majorar a expressão que define $T(n)$ no pior caso assim:

$$\begin{aligned} T(n) &= c_1 + 2c_2 + c_3(n + n - 1) + c_4(n - 1) + c_2(n - 1) + c_5 \\ &\leq \gamma + 2\gamma + \gamma(n + n - 1) + \gamma(n - 1) + \gamma(n - 1) + \gamma \\ &= \gamma + 4\gamma n \leq \gamma n + 4\gamma n = 5\gamma n. \end{aligned}$$

Concluiamos que $T(n) \leq 5\gamma n$, para todo $n \geq 1$, pelo que, tomando $C = 5\gamma$, teriamos $\forall n \geq 1\ \ T(n) \leq Cn$.

Provámos que existem C, C' e n_0 tais que $C'n \leq T(n) \leq Cn$ para $n \geq n_0$, ou seja, que $T(n) \in \Theta(n)$. Qualquer algoritmo que resolva o problema enunciado no Exemplo 1 terá de ler a sequência dada e, portanto, o tempo que demorará será de ordem $\Omega(n)$. Concluimos que o algoritmo de ordem $\Theta(n)$ apresentado acima é **assintoticamente ótimo**, pois qualquer algoritmo que resolva o problema terá a mesma complexidade assintótica ou pior.

Sendo n o parâmetro que caracteriza o tamanho da instância, um algoritmo tal que $T(n) \in \Theta(n)$ para qualquer instância (para n suficientemente grande) diz-se que tem **complexidade temporal (exatamente) linear em** n.

3.1.1 Funções mais comuns

São relativamente poucas as funções mais usadas para descrever a complexidade de muitos algoritmos — 1, $\log n$, n, $n \log n$, n^2, n^3 e 2^n — constante, logarítmica, linear, linearítmica (ou $n \log n$), quadrática, cúbica e exponencial.

Para $c \in \mathbb{R}^+$ constante, tem-se $c < c \log_2 n < cn < cn \log_2 n < cn^2 < cn^3 < c2^n$, para todo $n \geq 10$. Esta relação permite justificar, por exemplo, que:

$$O(1) \subset O(\log_2 n) \subset O(n) \subset O(n \log_2 n) \subset O(n^2) \subset O(n^3) \subset O(2^n).$$

Por exemplo, $O(n^2) \subset O(n^3)$ porque se $f(n) \leq cn^2$ a partir de uma certa ordem n_0, então também $f(n) \leq cn^3$ a partir da mesma ordem. Por outro lado, $n^3 \in O(n^3)$ mas $n^3 \notin O(n^2)$, o que justifica que a inclusão é estrita. Para provar que $n^3 \notin O(n^2)$, é necessário mostrar que quaisquer que sejam $c \in \mathbb{R}^+$ e $n_0 \in \mathbb{R}^+$ se tem $n^3 > cn^2$ para algum $n \geq n_0$. Como a relação $x^3 > cx^2$ se verifica para todo $x > c$, basta definir esse n por $n = \max(n_0, \lfloor c \rfloor + 1)$.

Comparação de ordens de grandeza logaritmicas

O *logaritmo de x na base a* é o número a que se deve elevar a para obter x, i.e., o número que verifica $x = a^{\log_a(x)}$, para $a \in \mathbb{R}^+ \setminus \{1\}$ e $x \in \mathbb{R}^+$. Recordar que $a^{\log_a(x)} = x = 2^{\log_2(x)} = (a^{\log_a(2)})^{\log_2(x)} = a^{\log_a(2)\log_2(x)}$, e as funções exponenciais são injetivas. Assim, de $a^{\log_a(x)} = a^{\log_a(2)\log_2(x)}$ deduzimos que $\log_a(x) = \log_a(2)\log_2(x)$, qualquer que seja $a \in \mathbb{R}^+ \setminus \{1\}$ e $x \in \mathbb{R}^+$. Usando esta relação podemos concluir que

$$O(\log_2 n) = O(\log_a n).$$

De facto, se $f(n) \in O(\log_2 n)$ então existem constantes $c \in \mathbb{R}^+$ e $n_0 \in \mathbb{Z}^+$ tais que $f(n) \leq c\log_2 n$. Logo, $f(n) \leq c\log_2 n = (c/log_a(2))\log_a(n)$, e, consequentemente, para $n \geq n_0$, $f(n) \leq c'\log_a(n)$, para $c' = c/\log_a(2)$ e $c' \in \mathbb{R}^+$. Portanto, concluimos que se $f(n) \in O(\log_2 n)$ então $f(n) \in O(\log_a n)$. Do mesmo modo se justifica que se $f(n) \in O(\log_a n)$ então $f(n) \in O(\log_2 n)$.

Comparação de ordens de grandeza exponenciais

A este propósito, é importante salientar que a igualdade não se verifica para as ordens exponenciais: para $B > 2$, tem-se $O(B^n) \neq O(2^n)$ ainda que $O(2^n) \subseteq O(B^n)$. Observe-se que $B^n \notin O(2^n)$ se $B > 2$. Em particular, podemos usar a função $f(n) = 3^n$ para mostrar que existe $f(n) \in O(3^n)$ tal que $f(n) \notin O(2^n)$, e assim concluir que $O(3^n) \not\subset O(2^n)$ (ou justificar que a inclusão de $O(2^n)$ em $O(3^n)$ seria estrita).

Para ver que se $B > 2$ então $B^n \notin O(2^n)$, comecemos por observar que se existissem $c \in \mathbb{R}^+$ e $n_0 \in \mathbb{Z}^+$ tais que $B^n \leq c2^n$ para todo $n \geq n_0$, então c seria maior do que 1 (pois sabemos que $B^n > 2^n$, para todo $n \in \mathbb{Z}^+$).

Como $c\,2^n = 2^{\log_2(c)}\,2^n = 2^{n+\log_2(c)}$ e $B^n = 2^{n\log_2(B)}$, vamos ver que $2^{n\log_2(B)} > 2^{n+\log_2(c)}$, para algum n suficientemente grande, qualquer que seja $c > 1$. Sendo a função exponencial $x \to 2^x$ estritamente crescente, $2^{n\log_2(B)} > 2^{n+\log_2(c)}$ se $n\log_2(B) > n + \log_2(c)$. Como $B > 2$, temos $\log_2(B) > 1$ (pois, $\log_2(2) = 1$). Então, se $n > \log_2(c)/(\log_2(B) - 1)$, obtemos $n(\log_2(B) - 1) > \log_2(c)$ e, assim,

$$n\log_2(B) = n(1 + \log_2(B) - 1) = n + n(\log_2(B) - 1) > n + \log_2(c).$$

Portanto, dado $c \in \mathbb{R}^+$ e $n_0 \in \mathbb{Z}^+$ podemos definir n por $n = \max(n_0, 1 + \lfloor \log_2(c)/(\log_2(B) - 1) \rfloor)$, para mostrar que

$$\forall c \in \mathbb{R}^+ \, \forall n_0 \in \mathbb{Z}^+ \, \exists n \geq n_0 \;\; B^n > c2^n,$$

ou seja, que $B^n \notin O(2^n)$, se $B > 2$.

Exemplo 3 (execício do 2ºTeste de 2012/2013)

Utilizando os símbolos $\subseteq$ ou $\subset$, relacionar as classes $\Theta(n^2 \log n)$, $O(n^4)$, $\Omega(n^2)$ e $O(1)$, se comparáveis (log designa $\log_{10}$). Justificar formalmente a resposta, recorrendo às definições das notações O, Θ e Ω.

- $\Theta(n^2 \log n) \subset \Omega(n^2)$ porque se $f(n) \in \Theta(n^2 \log n)$ então existem constantes $c_1 > 0$ e $n_0 > 0$ tais que $f(n) \geq c_1 n^2 \log n$ para todo $n \geq n_0$. Como $n^2 \log n \geq n^2$ para $n \geq 10$, concluímos que, se $n_0' = \max(n_0, 10)$ então $f(n) \geq c_1 n^2 \log n \geq c_1 n^2$ para $n > n_0'$. Logo, se $f(n) \in \Theta(n^2 \log n)$ então $f(n) \in \Omega(n^2)$, qualquer que seja a função $f(n)$. Portanto, $\Theta(n^2 \log n) \subseteq \Omega(n^2)$, mas a inclusão é estrita porque, por exemplo, a função $h(n) = n^2$ pertence a $\Omega(n^2)$ mas $h(n) \notin \Theta(n^2 \log n)$. Sabemos que, qualquer que seja $c > 0$, se $n > 10^{1/c}$ então $n^2(1 - c \log n) < 0$ (ou seja, $n^2 < cn^2 \log n$). Assim, $h(n) \notin \Omega(n^2 \log n)$ e, consequentemente, $h(n) \notin \Theta(n^2 \log n)$.

- $\Theta(n^2 \log n) \subset O(n^4)$ porque se $f(n) \in \Theta(n^2 \log n)$ então existem $c_2 > 0$ e $n_0 > 0$ tais que $f(n) \leq c_2 n^2 \log n$ para todo $n \geq n_0$. Como $c_2 n^2 \log n \leq c_2 n^4$ para todo $n \geq 1$, concluimos que se $f(n) \in \Theta(n^2 \log n)$ então $f(n) \in O(n^4)$. A inclusão é estrita porque, por exemplo, $h(n) = n^4 \in O(n^4)$ mas $h(n) \notin \Theta(n^2 \log n)$, dado que $n^4 \notin O(n^2 \log n)$, pois $n^4 > cn^2 \log n$ para todo $n > c$, qualquer que seja $c > 0$.

- $O(1) \subset O(n^4)$ porque se $f(n) \in O(1)$ então existem constantes $c > 0$ e $n_0 > 0$ tais que $f(n) \leq c$. Logo, $f(n) \leq cn^4$ para todo $n \geq n_0$, e portanto $f(n) \in O(n^4)$. A inclusão é estrita porque, por exemplo, a função $h(n) = n^4$ pertence a $O(n^4)$ mas não a $O(1)$.

Nos restantes casos, as classes não são comparáveis. Por exemplo, $\Theta(n^2 \log n) \not\subseteq O(1)$ pois $h(n) = n^2 \log n \notin O(1)$. Também $O(1) \not\subseteq \Theta(n^2 \log n)$ pois se $f(n) \in O(1)$ então $f(n) \notin \Omega(n^2 \log n)$ e portanto $f(n) \notin \Theta(n^2 \log n)$...

3.2 Estudo da complexidade de alguns algoritmos

Antes de avançar, é importante recordar algumas fórmulas matemáticas e definições que poderão ser úteis na contagem do número de operações realizadas pelos algoritmos que iremos analisar. Sendo $(u_i)_{i \in \mathbb{Z}^+}$ uma sucessão:

- $\sum_{i=k}^{n} u_i = u_k + u_{k+1} + \cdots + u_{n-1} + u_n$, se $k \leq n$. Convenciona-se que $\sum_{i=k}^{n} u_i = 0$, se $k > n$, porque 0 é o elemento neutro da adição. Assim, $\sum_{i=k}^{n} u_i = (\sum_{i=k}^{n-1} u_i) + u_n$, se $1 \leq k \leq n$.

- Se $u_i = c \in \mathbb{R}$, para todo $i \geq 1$, então $\sum_{i=k}^{n} u_i = c \sum_{i=k}^{n} 1 = c(n - k + 1)$, se $k \leq n$.

- $(u_i)_{i \geq 1}$ é uma progressão aritmética de razão r se $u_{i+1} = u_i + r = u_1 + r\,i$, para todo $i \geq 1$. Nesse caso, $\sum_{i=k}^{n} u_i = \frac{(u_k + u_n)(n-k+1)}{2}$, para $n \geq k$. Para concluir que assim é, basta ver que as as $n - k + 1$ parcelas da expressão de $2 \sum_{i=k}^{n} u_i$ apresentada abaixo são iguais a $(u_k + u_n)$.

$$\begin{aligned} 2 \sum_{i=k}^{n} u_i &= (u_k + u_{k+1} + u_{k+2} \cdots + u_n) + (u_n + u_{n-1} + u_{n-2} \cdots + u_k) \\ &= (u_k + u_n) + (u_{k+1} + u_{n-1}) + (u_{k+2} + u_{n-2}) + \cdots + (u_n + u_k) \end{aligned}$$

  Em particular, por exemplo, $\sum_{i=1}^{n} i = \frac{(n+1)n}{2}$.

- $(u_i)_{i\geq 1}$ é uma progressão geométrica de razão $r \neq 1$ se $u_{i+1} = u_i r = u_1 r^i$, para todo $i \geq 1$. Nesse caso, $\sum_{i=k}^{n} u_i = \frac{u_k - u_{n+1}}{1-r} = \frac{u_1(1-r^n)}{1-r}$, para $n \geq k$. Para deduzir esta fórmula, basta ver que

$$(1-r)\sum_{i=k}^{n} u_i = \sum_{i=k}^{n} u_i - r\sum_{i=k}^{n} u_i = u_k - u_{n+1}$$

  porque $-r\sum_{i=k}^{n} u_i = -ru_k - ru_{k+1} - \cdots - ru_{n-1} - ru_n = -\sum_{i=k+1}^{n+1} u_i$.

3.2.1 Ordenar um vetor por seleção do máximo

A função SELECTIONSORT, introduzida na secção 2.2, ordena um vetor por ordem decrescente, por aplicação do *método de seleção* (seleção do máximo). Para isso, utiliza a função POSMAX, introduzida na secção 2.1, a qual, quando aplicada a uma instância (v, k, n), retorna o índice da primeira ocorrência do máximo de $v[k], v[k+1], \ldots, v[n]$ no segmento $[k, n]$ de v.

```
SELECTIONSORT(v, n)
    Para cada k ← 1 até n − 1 fazer
        j ← POSMAX(v, k, n);
        Se j ≠ k então
            aux ← v[k];
            v[k] ← v[j];
            v[j] ← aux;
```

```
POSMAX(v, k, n)
    Se (k < 1 ∨ k > n) então
        retorna −1;
    pmax ← k;
    Para i ← k + 1 até n fazer
        Se v[i] > v[pmax] então
            pmax ← i;
    retorna pmax;
```

Análise da complexidade temporal de POSMAX

- Não é difícil concluir que, para instâncias (v, k, n) com $k < 1 \vee k > n$, o tempo de execução é de ordem $O(1)$.

- Vamos ver agora que, para instâncias (v, k, n) com $1 \leq k \leq n$, o tempo de execução de POSMAX é de ordem $\Theta(n-k+1)$, ou seja, é exatamente linear no número de elementos de v a analisar (recordemos que a expressão $n-k+1$ define o número de inteiros no intervalo $[k, n]$).

  Para (k, n) fixo, a diferença entre o tempo de execução de duas instâncias resulta da diferença entre o número de vezes que a instrução $pmax \leftarrow i$ é executada. No pior caso, $v[k] < v[k+1] < \ldots < v[n]$ e tal instrução é executada em todas as iterações do ciclo. No melhor caso, $v[k] \geq \max(v[k+1], \ldots, v[n])$ e essa instrução não é executada. Para determinar a expressão que define o tempo de execução, consideremos como custos das operações básicas as constantes $c_1, \ldots, c_7$ indicadas abaixo à direita.

POSMAX(v, k, n)	
Se $(k < 1 \vee k > n)$ então	c_1 (teste da condição e transferência de controlo)
retorna -1;	
$pmax \leftarrow k$;	c_2 (atribuição de valor de variável simples a outra)
$i \leftarrow k + 1$;	c_3 (atribuição de valor de expressão a variável simples)
Enquanto $(i \leq n)$ fazer	c_4 (teste de condição e transferência de controlo)
Se $v[i] > v[pmax]$ então	c_5 (acesso indirecto, teste de condição e transferência de controlo)
$pmax \leftarrow i$;	c_2
$i \leftarrow i + 1$;	c_6 (incremento de variável e transferência de controlo)
retorna $pmax$;	c_7 (retornar valor e transferência de controlo)

No pior caso, $T(k, n) = c_1 + c_2 + c_3 + c_4(n - k + 1) + c_5(n - k) + c_2(n - k) + c_6(n - k) + c_7$.

No melhor caso, $T(k, n) = c_1 + c_2 + c_3 + c_4(n - k + 1) + c_5(n - k) + c_6(n - k) + c_7$.

Assim, para $\alpha = \min\{c_i \mid 1 \leq i \leq 7\}$ e $\beta = \max\{c_i \mid 1 \leq i \leq 7\}$, temos

$$4\alpha + 2\alpha(n - k) + \alpha(n - k + 1) \leq T(k, n) \leq 4\beta + 3\beta(n - k) + \beta(n - k + 1)$$

para todo v, e como $4\beta + 3\beta(n - k) + \beta(n - k + 1) \leq 4\beta(n - k + 1) + 3\beta(n - k + 1) + \beta(n - k + 1)$, para $k \leq n$, e $\alpha(n - k + 1) \leq 4\alpha + 2\alpha(n - k) + \alpha(n - k + 1)$, deduzimos que

$$\alpha(n - k + 1) \leq T(k, n) \leq 8\beta(n - k + 1)$$

qualquer que seja a instância (v, k, n) com $1 \leq k \leq n$. Concluimos que existem constantes C_1 e C_2 positivas tais que $C_1(n - k + 1) \leq T(k, n) \leq C_2(n - k + 1)$ qualquer que seja $(n - k + 1) \geq 1$. Por exemplo, podemos tomar $C_1 = \alpha$ e $C_2 = 8\beta$. Logo, $T(k, n) \in \Theta(n - k + 1)$. □

Nesta análise mantivemos T como função de (k, n), em vez de fazer $n - k + 1 = N$ e definir T como função de N. A forma escolhida será mais útil na determinação da complexidade da função SELECTIONSORT porque mostra como k intervém na complexidade da chamada de POSMAX quando n está fixo.

Na análise de complexidade supusemos que as constantes que definiam os tempos elementares podiam ser distintas. Alguns autores assumem que todas as operações elementares têm **custo unitário**. Tal pode ser feito sem perda de generalidade se o objetivo for simplesmente a caraterização da complexidade assintótica, mas vamos manter a abordagem inicial.

Análise da complexidade temporal de SELECTIONSORT

SELECTIONSORT(v, n)

Para cada $k \leftarrow 1$ até $n-1$ fazer	A instrução de troca (de $v[j]$ com $v[k]$) pode ser efetuada no máximo
$j \leftarrow$ POSMAX(v, k, n);	$n-1$ vezes. Quando v está ordenado por ordem estritamente decres-
Se $j \neq k$ então	cente, a troca entre $v[j]$ e $v[k]$ nunca é efetuada e a instrução $pmax \leftarrow i$
$aux \leftarrow v[k]$;	de POSMAX não é executada em nenhuma das chamadas de POSMAX.
$v[k] \leftarrow v[j]$;	Tal instância carateriza o custo no melhor caso.
$v[j] \leftarrow aux$;	

Designemos por $T_{SSort}(n)$ o tempo de execução de SELECTIONSORT numa instância qualquer com n elementos. Para o melhor caso, se contabilizarmos apenas o tempo gasto na inicialização da variável k de controlo do ciclo (i.e., pela instrução $k \leftarrow 1$, com custo c_8) e o tempo gasto na execução de POSMAX nas várias chamadas, temos

$$T_{SSort}(n) \geq \alpha' + \sum_{k=1}^{n-1} \alpha'(n-k+1),$$

com $\alpha' = \min\{c_i \mid 1 \leq i \leq 8\}$. Sendo

$$\alpha' + \sum_{k=1}^{n-1} \alpha'(n-k+1) = \sum_{k=1}^{n} \alpha'(n-k+1) = \alpha' \sum_{k=1}^{n}(n-k+1) = \alpha' \sum_{k=1}^{n} k = \alpha' \frac{(n+1)n}{2}$$

concluimos que $T_{SSort}(n) \in \Omega(n^2)$ pois, para $C = \alpha'/2$, obtemos $T_{SSort}(n) \geq Cn^2$, qualquer que seja $n \geq 1$, em virtude de

$$T_{SSort}(n) \geq \alpha' \frac{(n+1)n}{2} > \alpha' \frac{(n)n}{2} = \frac{\alpha'}{2} n^2.$$

Por outro lado, no pior caso, o tempo de execução $T_{SSort}(n)$ não excederá

$$c_8 + c_9(n) + c_{10}(n-1) + \sum_{k=1}^{n-1} T^\star(k, n) + c_{11}(n-1) + c_6(n-1) + c_{12}$$

em que $T^\star(k, n)$ designa o tempo de execução máximo de POSMAX em qualquer instância (v, k, n) e os tempos c_i, para $8 \leq i \leq 12$, correspondem informalmente a:

- c_8 – inicialização de variável simples com constante ($k \leftarrow 1$)
- c_9 – teste da condição $k \leq n-1$ e transferência de controlo no ciclo
- c_{10} – chamada de POSMAX, passagem de valores (criação de variáveis locais) e atribuição de resultado a j
- c_{11} – instrução "Se" e troca entre $v[k]$ e $v[j]$
- c_{12} – retorno da função.

De acordo com a análise de POSMAX, $T^\star(k,n) \leq 8\beta(n-k+1)$, e sendo $\beta \leq \beta'$ para $\beta' = \max\{c_i \mid 1 \leq i \leq 12\}$, podemos concluir que

$$T_{SSort}(n) \leq 2\beta' + \beta'(n) + 3\beta'(n-1) + \sum_{k=1}^{n-1} 8\beta'(n-k+1).$$

Como, para todo $n \geq 1$, se tem

$$\begin{aligned} T_{SSort}(n) &\leq 2\beta' + \beta'(n) + 3\beta'(n-1) + \sum_{k=1}^{n-1} 8\beta'(n-k+1) \\ &< 2\beta' + \beta'(n) + 3\beta'(n) + \sum_{k=1}^{n} 8\beta'(n-k+1) \\ &= 2\beta' + 4\beta' n + 8\beta' \frac{n(n+1)}{2} \\ &= 2\beta' + 8\beta' n + 4\beta' n^2 \\ &\leq 2\beta' n^2 + 8\beta' n^2 + 4\beta' n^2 \\ &= 14\beta' n^2 \end{aligned}$$

então $T_{SSort}(n) \in O(n^2)$, para qualquer instância de tamanho n. Tinhamos mostrado acima que $T_{SSort}(n) \in \Omega(n^2)$ e, portanto, $T_{SSort}(n) \in \Theta(n^2)$, ou seja, o algoritmo de ordenação por seleção apresentado é de ordem $\Theta(n^2)$, i.e., sempre quadrático em n.

3.2.2 Ordenar um vetor pelo método da bolha

A função BUBBLESORT definida abaixo ordena os n primeiros elementos do vetor x por ordem crescente, por aplicação do Método da Bolha (*bubble sort*).

```
BUBBLESORT(x, n)
    i ← 1;
    trocas ← 1;
    Enquanto (trocas ≠ 0 ∧ i < n) fazer
        trocas ← 0;
        Para j ← 1 até n − i fazer
            Se (x[j − 1] > x[j]) então
                trocas ← 1;
                aux ← x[j];
                x[j] ← x[j − 1];
                x[j − 1] ← aux;
        i ← i + 1;
```

Vamos provar a sua correção e ver que, no melhor caso, a sua complexidade temporal é de ordem $\Theta(n)$, mas, em geral, é $O(n^2)$. Começamos por recordar as ideias principais do método.

- No ciclo "Para", a função analisa $x[0], x[1], \ldots, x[n-i]$, comparando cada elemento com o anterior (se existir). A variável $trocas$ mantém o valor 0 se e só se essa sequência estiver ordenada por ordem crescente, ou seja, sse $x[0] \leq x[1] \leq \ldots \leq x[n-i]$. A descida de $x[0], x[1], \ldots, x[n-i]$ é aproveitada para aproximar alguns elementos das suas posições finais no vetor ordenado. Para isso, sempre que detecta uma inversão de ordem entre dois elementos consecutivos (traduzida por $x[j-1] > x[j]$), permuta esse par de elementos.

- Na iteração i do ciclo "Enquanto", a função analisa $x[0], x[1], \ldots, x[n-i]$ e, possivelmente, altera a ordem de alguns elementos, garantindo que, no fim dessa iteração, $x[n-i]$ contém o máximo dessa sequência. Como mostraremos a seguir, a condição $\max(x[0], \ldots, x[n-i]) \leq x[n-i+1] \leq \ldots \leq x[n-1]$ é preservada ao longo do ciclo, o que implica que o vetor está já ordenado se não houver trocas na execução do ciclo "Para".

Prova de correção do BUBBLESORT

Para mostrar a correção de um algoritmo, importa sempre tentar caraterizar com algum rigor **o estado de cada variável** ao longo do programa. Neste caso, a interpretação é a seguinte:

- i: variável usada para contar o número de vezes que se analisa o vetor; quando a condição do ciclo "Enquanto" é testada pela i-ésima vez, a parte do vetor que já está ordenada tem $i-1$ elementos (e, se $i > 1$, a parte já ordenada é $x[n-i+1], x[n-i+2], \ldots, x[n-1]$);

- $trocas$: variável de estado (*flag*) e indica se já se detetou que o vetor está ordenado;

- j: variável auxiliar usada para indexar elementos do vetor numa descida;

- aux: variável auxiliar usada para efetuar a troca de dois elementos (i.e., de $x[j-1]$ com $x[j]$), se necessário.

Importa também ver qual é **o estado** do vetor quando testa a condição do ciclo "Enquanto" na iteração $i+1$ e **que alteração sofreu** relativamente à iteração i. Comecemos por observar que, se $n \leq 1$, o vetor está ordenado, e o algoritmo não efetua qualquer iteração do ciclo "Enquanto", mantendo o vetor sem alterações. Assim, iremos considerar agora o caso $n > 1$. Nesse caso, podemos afirmar que o algoritmo efetua pelo menos uma iteração do ciclo já que, inicialmente, se tem $trocas = 1 \wedge i < n$. Em cada iteração do ciclo "Enquanto", o algoritmo efetua o ciclo "Para". Podemos mostrar o lema seguinte, já acima referido (os detalhes da prova ficam ao cuidado do leitor).

Lema: No ciclo "Para", o algoritmo analisa a sequência $x[0], x[1], \ldots, x[n-i]$, compara cada elemento com o anterior e troca-os entre si se não estiverem na ordem correta. No fim do ciclo, $x[n-i]$ contém o máximo da

sequência $x[0], x[1], \ldots, x[n-i]$ e, se não tiverem ocorrido trocas, $trocas$ será 0 e essa sequência estava ordenada (caso contrário, $trocas$ terá o valor 1 e nada mais se pode dizer sobre a sequência).

Usando o Lema, vamos mostrar que, quando se está a testar a condição do ciclo "enquanto" pela (i_0+1)-ésima vez, se tem o seguinte:

1. os últimos i_0 elementos de x estão ordenados, i.e., $x[n-i_0] \leq x[n-i_0+1] \leq x[n-i_0+2] \leq \cdots \leq x[n-1]$,
2. $x[n-i_0]$ é maior ou igual a qualquer elemento de índice inferior,
3. se $trocas = 0$ então, na iteração i_0, verificou-se que $x[0] \leq x[1] \leq \cdots \leq x[n-i_0-1]$; caso contrário, $trocas = 1$ e nada se pode afirmar sobre $x[0], x[1], \ldots, x[n-i_0-1]$.
4. o valor da variável i é (i_0+1).

Prova (por indução sobre o número de iterações): A condição (4) é fácil de mostrar, pois a variável i tem o valor 1 quando testa a condição pela primeira vez e é incrementada de uma unidade em cada iteração do ciclo "Enquanto" imediatamente antes de voltar a testar a condição de ciclo. Vamos agora mostrar que as restantes condições se verificam.

(caso de base) Para $i_0 = 1$, as condições (1)–(3) verificam-se pois, pelo Lema, após a primeira iteração, o elemento que está na posição $n-1$ é maior ou igual a qualquer um dos elementos $x[0], \ldots, x[n-2]$, e se $trocas$ for 0 então $x[0] \leq \ldots \leq x[n-2] \leq x[n-1]$ e, caso contrário, $trocas = 1$ e nada se pode garantir sobre esta ordem.

(hereditariedade) Se o ciclo não parar antes de testar a condição de ciclo pela i_0+1 vez, sendo $i_0 \geq 1$, então quando testou essa condição pela i_0-ésima vez, o valor de $trocas$ era 1 e $i_0 < n$ (pois, caso contrário, o ciclo "Enquanto" terminava e não realizaria o (i_0+1)-ésimo teste). Suponhamos então, como **hipótese de indução**, que a propriedade se verifica para esse i_0 fixo quando o algoritmo vai iniciar a i_0-ésima iteração do ciclo "Enquanto", ou seja, que:

- se $i_0 > 1$ então $x[n-i_0+1] \leq x[n-i_0+2] \leq \cdots \leq x[n-1]$,
- se $i_0 > 1$ então $x[n-i_0+1]$ é maior ou igual a qualquer elemento de índice inferior,
- $trocas = 1$ e nada se pode afirmar sobre $x[0], x[1], \ldots, x[n-i_0]$.

Na iteração i_0, começa por atribuir o valor 0 à variável $trocas$ e a seguir executa o ciclo "Para", em que analisa $x[0], x[1], \ldots, x[n-i_0]$. Pelo Lema, podemos afirmar o seguinte sobre os valores finais de $x[0]$, $x[1]$, …, $x[n-i_0]$ e de $trocas$:

- se $trocas$ for 0 então $x[0] \leq x[1] \leq \cdots \leq x[n-i_0]$.

- se $trocas$ for 1 então $x[n - i_0]$ é maior ou igual que $x[0], x[1], \ldots, x[n - i_0 - 1]$ e nada mais se sabe sobre a ordem relativa desses elementos.

Conjuntamente com a hipótese de indução, essas condições implicam que:

- se $trocas = 0$ então $x[0] \leq x[1] \leq \ldots \leq x[n - i_0] \leq x[n - i_0 + 1] \leq x[n - i_0 + 2] \leq \cdots \leq x[n - 1]$, ou seja, o vetor está ordenado.

- se $trocas = 1$ então $x[n - i_0] \leq x[n - i_0 + 1] \leq x[n - i_0 + 2] \leq \cdots \leq x[n - 1]$, que $x[n - i_0]$ é maior ou igual que os elementos de índice inferior, e também que nada se sabe sobre $x[0], x[1], \ldots, x[n - i_0 - 1]$.

Assim, quando testar novamente a condição do ciclo "Enquanto", isto é, quando a testar pela $(i_0 + 1)$-ésima vez, as condições (1), (2), e (3) verificam-se. □

O **algoritmo termina** porque em cada iteração do ciclo "Enquanto…", o valor da variável i cresce. O ciclo pára quando $trocas = 0$ ou quando $trocas = 1 \wedge i = n$. Se $trocas = 0$ então o vetor está ordenado, como se concluiu acima. Se parar quando $trocas = 1 \wedge i = n$, então $trocas = 1$ quando a condição de ciclo foi testada pela $(n - 1)$-ésima vez, e mostrou-se que, no fim da iteração $n - 1$, se teria $x[1] \leq x[2] \leq \cdots \leq x[n - 1]$ e $x[0] \leq x[1]$, e que $i = n$. Ou seja, o vetor fica ordenado e a condição $i = n$ determina o fim do ciclo quando efetuar o teste pela n-ésima vez. Portanto, no fim do ciclo, o vetor está ordenado por ordem crescente, como se pretendia provar.

Análise da complexidade de BUBBLESORT

No melhor caso, o vetor dado está ordenado e o algoritmo realiza uma única iteração do ciclo "Enquanto". No pior caso, o vetor está ordenado mas por ordem estritamente decrescente. Nesse caso, o algoritmo efetua $n - 1$ iterações do ciclo "Enquanto" e, em cada ciclo "Para", troca sempre os elementos que comparou (ou seja, para cada i, efetua $n - i$ trocas).

Sejam c_k, com $0 \leq k \leq 9$, constantes que representam as durações das instruções mais simples (não necessariamente elementares), assim definidas.

c_0 : avaliar conjunção e transferir controlo
c_1 : atribuir um valor a variável simples
c_2 : comparar o conteúdo de uma variável simples com um inteiro,
c_3 : comparar os valores de duas variáveis simples
c_4 : testar da condição $j \leq n - i$ e transferir controlo
c_5 : incrementar uma variável simples
c_6 : comparar um elemento do vetor com o seu anterior e transferir controlo
c_7 : copiar um elemento do vetor para uma variável simples
c_8 : copiar um elemento do vetor para a posição anterior
c_9 : copiar valor de variável simples para uma posição no vetor (com cálculo do indíce)

Podemos assim construir a tabela seguinte para $n > 1$, onde $t_i = c_1 + c_4(n - i + 1) + c_5(n - i)$, para cada i. Notemos que, para deduzir uma expressão válida para $n = 1$, podiamos utilizar a função de Kronecker ($\delta_{n1} = 1$ se $n = 1$ e $\delta_{n1} = 0$ se $n \neq 1$) e escrever $c_0 + c_2 + c_3 + (c_0 + c_2)\delta_{n1}$ em vez de $2c_0 + c_2 + c_3 + c_2$, por exemplo. Contudo, o caso $n = 1$ não é relevante para o estudo da complexidade assintótica e por isso vamos assumir que $n > 1$.

	Tempo Mínimo (melhor caso)	Tempo Máximo (pior caso)	
$i \leftarrow 1$;	c_1	c_1	$\leq \beta$
$trocas \leftarrow 1$;	c_1	c_1	$\leq \beta$
Enquanto ($trocas \neq 0 \wedge i < n$) fazer	$2c_0 + c_2 + c_3 + c_2$	$(c_0 + c_2 + c_3)n$	$\leq 3\beta n$
$trocas \leftarrow 0$;	c_1	$c_1(n-1)$	$\leq \beta(n-1)$
Para $j \leftarrow 1$ até $n - i$ fazer	t_1	$\sum_{i=1}^{n-1} t_i$	$\leq 2\beta(n-1) + \beta n(n-1)$
Se ($x[j-1] > x[j]$) então	$c_6(n-1)$	$c_6 \sum_{i=1}^{n-1}(n-i)$	$\leq \beta n(n-1)/2$
$trocas \leftarrow 1$	0	$c_1 \sum_{i=1}^{n-1}(n-i)$	$\leq \beta n(n-1)/2$
$aux \leftarrow x[j]$;	0	$c_7 \sum_{i=1}^{n-1}(n-i)$	$\leq \beta n(n-1)/2$
$x[j] \leftarrow x[j-1]$;	0	$c_8 \sum_{i=1}^{n-1}(n-i)$	$\leq \beta n(n-1)/2$
$x[j-1] \leftarrow aux$;	0	$c_9 \sum_{i=1}^{n-1}(n-i)$	$\leq \beta n(n-1)/2$
$i \leftarrow i + 1$;	c_5	$c_5(n-1)$	$\leq \beta(n-1)$

Somando as durações indicadas em cada coluna, obtemos a expressão que define o tempo $T(n)$ que o algoritmo requer quando aplicado a vetores de tamanho n, no melhor e pior caso. Concluimos que $T(n) \in \Omega(n)$, para qualquer instância de tamanho n, pois, no melhor caso, $T(n) = (2c_0 + 4c_1 + 2c_2 + c_3 - c_6) + (c_4 + c_5 + c_6)n$. Por outro lado,

recordando que $\sum_{i=1}^{n-1}(n-i) = \sum_{i=1}^{n-1} i = \frac{(n-1)(n-1+1)}{2} = \frac{n(n-1)}{2}$ e que

$$\begin{aligned}
\sum_{i=1}^{n-1} t_i &= \sum_{i=1}^{n-1}(c_1 + c_4(n-i+1) + c_5(n-i)) \\
&= \sum_{i=1}^{n-1} c_1 + c_4 \sum_{i=1}^{n-1}(n-i+1) + c_5 \sum_{i=1}^{n-1}(n-i) \\
&= c_1 \sum_{i=1}^{n-1} 1 + c_4 \sum_{i=1}^{n-1}(n-i) + c_4 \sum_{i=1}^{n-1} 1 + c_5 \sum_{i=1}^{n-1}(n-i) \\
&= (c_1 + c_4)(n-1) + (c_4 + c_5)\frac{n(n-1)}{2}
\end{aligned}$$

e substituindo cada c_k por $\beta = \max_k c_k$, como vemos na última coluna da tabela, obtemos

$$T(n) \leq -2\beta + \frac{7}{2}\beta n + \frac{7}{2}\beta n^2 < \frac{7}{2}\beta n + \frac{7}{2}\beta n^2 < \frac{7}{2}\beta n^2 + \frac{7}{2}\beta n^2 = 7\beta n^2$$

para todo $n > 1$. Portanto, $T(n) \in O(n^2)$, para qualquer instância, e $T(n) \in \Theta(n^2)$ no pior caso, ainda que $T(n) \in O(n)$ no melhor caso. Para ver que $T(n) \in \Omega(n^2)$ no pior caso, basta ver que a expressão que se obtemos se substituirmos cada c_k por $\min_k c_k$ na expressão de $T(n)$ é um polinómio de grau 2 e é um minorante de $T(n)$.

3.2.3 Encontrar um elemento num vetor ordenado (pesquisa binária)

Na secção 2.8, apresentámos uma função PESQUISABINARIA que quando aplicada a uma instância (v, a, b, x), em que v é um vetor ordenado por ordem decrescente encontra x em v se x ocorrer entre as posições de índices a e b, assumindo que $0 \leq a \leq b < n$, sendo n o número de elementos do vetor.

Vamos estudar a complexidade temporal de uma função parecida mas em que se procura x em $v[0], \ldots, v[n-1]$, estando v ordenado por ordem **decrescente**.

PROCURABINARIA(v, n, x)	*Caso $x \notin v[\cdot]$*
$a \leftarrow 0$;	c_1
$b \leftarrow n - 1$;	c_2
Enquanto ($a \leq b$) fazer	$c_3 \times (k_1 + k_2 + 1)$
$\quad m \leftarrow \lfloor (a+b)/2 \rfloor$;	$c_4 \times (k_1 + k_2)$
$\quad$ Se ($v[m] = x$) então	$c_5 \times (k_1 + k_2)$
$\quad\quad$ retorna m;	$c_6 \times 0$
$\quad$ Se ($v[m] > x$) então	$c_5 \times (k_1 + k_2)$
$\quad\quad a \leftarrow m + 1$;	$c_6 k_1$
$\quad$ senão	
$\quad\quad b \leftarrow m - 1$;	$c_6 k_2$
retorna -1;	c_7

Quando x não ocorre em v, o número de vezes que o bloco de instruções do ciclo é executado (isto é, o *número de iterações* do ciclo) é $k_1 + k_2$, sendo k_1 e k_2 o número de vezes que se altera cada um dos extremos do intervalo. A condição de ciclo é testada $k_1 + k_2 + 1$ vezes. Como $(c_1 + c_2 + c_3 + c_7) + (c_3 + c_4 + c_5 + c_6)(k_1 + k_2) \in O(k_1 + k_2)$, vamos mostrar que, **no pior caso,** $k_1 + k_2 = \lfloor \log_2(n) \rfloor + 1$, para todo $n > 1$, concluindo que a complexidade deste algoritmo de pesquisa binária é $O(\log\ n)$.

Antes de prosseguirmos, notemos que:

- para contemplar também o caso $n = 1$, podiamos escrever $O(1 + \log n)$, mas não é habitual, porque estamos a analisar a complexidade assintótica, isto é, para valores de n suficientemente grandes;
- escrevemos $O(\log\ n)$ em vez de $O(\log_2\ n)$ porque, como vimos, $\log_B n = (\log_B 2)(\log_2 n)$, para toda a base B, e, consequentemente, $O(\log_2\ n) = O(\log_B\ n)$, qualquer que seja a base que se considere.

Tempo no pior caso. O número de iterações é máximo quando x não se encontra no vetor e $v[n-1] > x$, embora esse número possa ser atingido também noutras situações. Tal resulta de o ciclo terminar somente quando o intervalo fica vazio e do facto de, em cada iteração, o intervalo $[m+1, b]$ ter tantos ou mais elementos que o intervalo $[a, m-1]$. Como o tempo gasto em cada iteração é de ordem $O(1)$, podemos restringir a análise do pior caso às instâncias em que $v[n-1] > x$, isto é, em que os n elementos a considerar de v são maiores do que x.

Comecemos por analisar o caso em que $n = 2^p$ para algum p, isto é, o **caso em que** $\log_2 n$ **é inteiro**. Sejam $\tilde{a}_k$ e $\tilde{b}_k$ os valores das variáveis a e b imediatamente após a k-ésima iteração e $\tilde{I}_k = [\tilde{a}_k, \tilde{b}_k]$ o intervalo que definem. Como $v[n-1] > x$, temos $\tilde{b}_k = n - 1$ e $\tilde{a}_k = \lfloor (n - 1 + \tilde{a}_{k-1})/2 \rfloor + 1 = \sum_{i=1}^{k} 2^{p-i}$, para $k \geq 1$. De facto, observemos que, para $n = 2^p$, se tem

$$
\begin{aligned}
&\tilde{a}_1 = \lfloor (n-1)/2 \rfloor + 1 = \lfloor 2^{p-1} - 1/2 \rfloor + 1 = (2^{p-1} - 1) + 1 = 2^{p-1} \\
&\tilde{a}_2 = \lfloor (n - 1 + 2^{p-1})/2 \rfloor + 1 = \lfloor 2^{p-1} + 2^{p-2} - 1/2 \rfloor + 1 = (2^{p-1} + 2^{p-2} - 1) + 1 = 2^{p-1} + 2^{p-2} \\
&\tilde{a}_3 = \lfloor (n - 1 + 2^{p-1} + 2^{p-2})/2 \rfloor + 1 = 2^{p-1} + 2^{p-2} + 2^{p-3} \\
&\ldots \\
&\tilde{a}_p = 2^{p-1} + 2^{p-2} + \cdots + 2^0 = 2^p - 1 = n - 1
\end{aligned}
$$

e $|\tilde{I}_k| = 2^p/2^k$, ou seja, $|\tilde{I}_k| = 2^{p-k}$. Ao fim da p-ésima iteração, $|\tilde{I}_k| = 1$, pelo que, o ciclo termina imediatamente após a iteração $p + 1$. Logo, se $n = 2^p$ e $v[n-1] > x$, o número de iterações realizadas é $1 + \log_2 n = 1 + \lfloor \log_2 n \rfloor$.

Analisemos agora o **caso em que** $\log_2 n$ **não é inteiro**, ou seja, em que $p - 1 < \log_2 n < p$, para $p = \lceil \log_2 n \rceil \geq 2$, sendo $2^{p-1} + 1 \leq n \leq 2^p - 1$. Quando $v[n-1] > x$, o número de iterações total é $p = \lfloor \log_2 n \rfloor + 1$, pois, se $0 \leq k \leq p - 1$, o número de elementos de $\tilde{I}_k$ satisfaz $2^{p-1-k} \leq |\tilde{I}_k| \leq \frac{n+1}{2^k} - 1 \leq 2^{p-k} - 1$, o que implica que $|\tilde{I}_{p-2}| \in \{2, 3\}$ e, portanto, $|\tilde{I}_{p-1}| = 1$ e $|\tilde{I}_p| = 0$. Para ver que assim é, basta analisar os dois casos extremos, em que $n = 2^{p-1} + 1$ ou $n = 2^p - 1$.

No caso em que $n = 2^{p-1} + 1$, o valor da variável a após a k-ésima iteração é $\tilde{a}_k = (\sum_{i=p-1-k}^{p-2} 2^i) + 1$, concluindo-se que $|\tilde{I}_k| = (n-1) - \tilde{a}_k + 1 = 2^{p-1} - 1 - \sum_{i=p-1-k}^{p-2} 2^i = 2^{p-1-k}$. Do mesmo modo, se $n = 2^p - 1$, temos $|\tilde{I}_k| = 2^{p-k} - 1$, para $k \leq p$, pois $\tilde{a}_k = \sum_{i=p-k}^{p-1} 2^i = 2^{p-1} - 2^{p-k}$.

Logo, se $2^{p-1} + 1 \leq n \leq 2^p - 1$, o número de elementos de $\tilde{I}_k$ está no intervalo indicado, o que implica que o número de iterações necessárias e suficientes para reduzir $[0, n-1]$ a um intervalo vazio é $p = \lceil \log_2 n \rceil = \lfloor \log_2 n \rfloor + 1$.

Em suma, no pior caso, a complexidade temporal de PROCURABINARIA é de ordem $\Theta(\log_2 n)$.

Tempo no melhor caso. No melhor caso, $v[m] = x$ para $m = \lfloor (n-1)/2 \rfloor$ e PROCURABINARIA localiza x na primeira iteração, sendo a complexidade temporal de ordem $O(1)$.

Portanto, para instâncias genéricas com n elementos, PROCURABINARIA tem complexidade $O(\log_2 n)$. □

3.2.4 Ordenar um vetor em $O(n \log n)$ por *merge sort*

O método de ordenação *merge sort* é mais eficiente, no pior caso, do que os métodos de ordenação anteriormente apresentados (selection sort e bubble sort). Este método utiliza uma abordagem de "divisão e conquista" (*divide-and-conquer*). A função MERGESORT, apresentada a seguir, ordena por ordem crescente uma secção de um vetor v, constituída pelas posições cujos índices estão num intervalo $[a, b]$, com $0 \leq a \leq b \leq n-1$, sendo n o número de elementos guardados em v. Se for chamada com $a = 0$ e $b = n - 1$, então ordena v por ordem crescente.

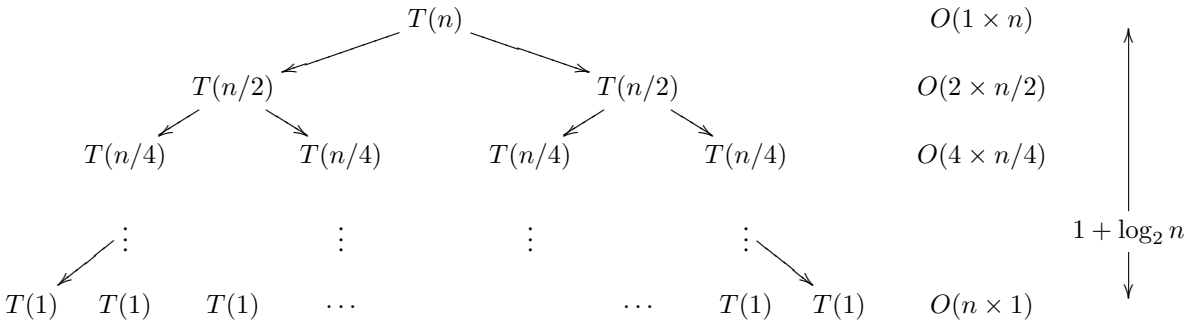
```
MERGESORT(v, a, b)
    Se (a < b) então
        m ← ⌊(a + b)/2⌋;
        MERGESORT(v, a, m);
        MERGESORT(v, m + 1, b);
        MERGE(v, a, m, b);
```

```
MERGE(v, a, m, b)
    i ← a;  j ← m + 1;  k ← 0;
    Enquanto (i ≤ m ∧ j ≤ b) fazer
        Se v[i] < v[j] então {aux[k] ← v[i];  i ← i + 1;}
        senão {aux[k] ← v[j];  j ← j + 1;}
        k ← k + 1;
    Enquanto (i ≤ m) fazer
        aux[k] ← v[i];
        i ← i + 1;  k ← k + 1;
    Para i ← 0 até k − 1 fazer
        v[i + a] ← aux[i];
```

Como vemos, a versão recursiva é muito simples: o segmento a ordenar é dividido ao meio, cada uma das metades é ordenada por um processo semelhante e, a seguir, as duas partes são combinadas ordenamente por MERGE para obter a sequência final. Na função MERGE, a variável aux é um vetor auxiliar, e assumimos que tem n posições pelo

menos. Nesta função, quando o primeiro ciclo "Enquanto" termina com $j \leq b$, os elementos $v[j], v[j+1], \ldots, v[b]$ estão já na posição correta em v, razão pela qual não são copiados para aux.

A função MERGE tem complexidade $O(b-a+1)$, se $a < b$, e $O(1)$ se $a \geq b$. Assim, a complexidade temporal de MERGESORT pode ser definida pela recorrência $T(1) = c_1$ e $T(n) \leq 2T(\lceil n/2 \rceil) + c_2 n$, para $n \geq 2$. Para facilitar, focamos o caso em que n é potência de 2. Nos restantes casos, podemos seguir uma abordagem semelhante à usada na análise do algoritmo de pesquisa binária. O número de níveis da árvore de recursão é $1 + \log_2 n$ e a complexidade de cada nível é $O(n)$. Logo, $T(n) \in O(n \log_2 n)$.



Observação. É conhecido que a complexidade temporal assintótica de MERGESORT é **ótima** na classe dos algoritmos de ordenação que se baseiam na **comparação de elementos**, pois qualquer algoritmo dessa classe requer $\Omega(n \log_2 n)$, no pior caso. Existem outros algoritmos com a mesma complexidade, como, por exemplo, *heapsort*. Note-se também que, apesar da designação, o método *quicksort* é $O(n \log n)$ no caso médio, mas $\Theta(n^2)$ no pior caso.

3.2.5 Determinar um par de pontos a distância mínima num conjunto de n pontos em $\mathbb{R}^2$

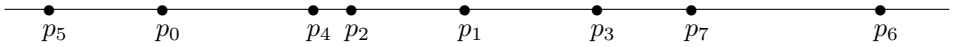
O problema da determinação de um par de pontos a distância mínima num conjunto $\mathcal{P}$ de $n > 1$ pontos em $\mathbb{R}^2$ ou em $\mathbb{R}$ pode ser resolvido trivialmente em $\Theta(n^2)$, por força bruta.

PARMAISPROXIMO_TRIVIAL(p, n, par)
- $distmin \leftarrow$ QDIST$(p, 0, 1)$; $par[0] \leftarrow 0$; $par[1] \leftarrow 1$;
- Para $i \leftarrow 0$ até $n-2$ fazer
  - Para $j \leftarrow i$ até $n-1$ fazer
    - $aux \leftarrow$ QDIST(p, i, j);
    - Se $(aux < distmin)$ então $\{distmin \leftarrow aux;\ par[0] \leftarrow i;\ par[1] \leftarrow j;\}$

Para evitar o cálculo de raízes quadradas, QDIST(p, i, j) deve retornar o **quadrado** da distância euclideana entre p_i e p_j, ou seja, $(x_i - x_j)^2 + (y_i - y_j)^2$ para pontos em $\mathbb{R}^2$, e $(x_i - x_j)^2$ para $\mathcal{P} \subset \mathbb{R}$.

Este algoritmo tem complexidade $\Theta(n^2)$ pois calcula a distância entre qualquer par de pontos para identificar um par para o qual esse valor é mínimo. Vamos ver que o problema pode ser resolvido em $O(n \log_2 n)$, em ambos os casos, isto é, para pontos colineares (isto é, numa recta) ou num plano.

Para o caso em que $\mathcal{P} \subseteq \mathbb{R}$, podemos explorar o facto de os pontos que estão a distância mínima ocorrerem em posições consecutivas na recta real, como se ilustra abaixo.



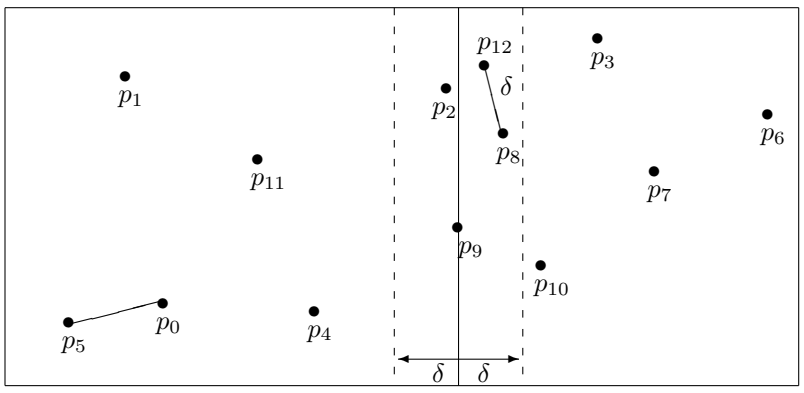
Assim, em alternativa ao método de força-bruta, bastaria aplicar um algoritmo $O(n \log_2 n)$ para ordenar os pontos e a seguir processar a sequência ordenada para identificar um par a distância mínima (em $\Theta(n)$).

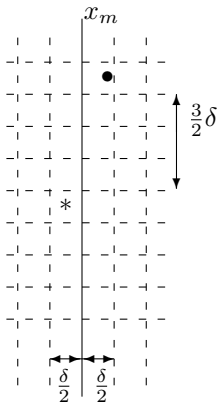
PARMAISPROXIMO_RECTA(p, n, par)

ORDENAR(p, n, pos);
$distmin \leftarrow$ QDIST$(p, pos[0], pos[1])$;
$par[0] \leftarrow pos[0]$; $par[1] \leftarrow pos[1]$;
Para $i \leftarrow 2$ até $n-1$ fazer
    $aux \leftarrow$ QDIST$(p, pos[i-1], pos[i])$;
    Se $(aux < distmin)$ então
        $distmin \leftarrow aux$;
        $par[0] \leftarrow pos[i-1]$; $par[1] \leftarrow pos[i]$;

A função ORDENAR(p, n, pos) deve ter complexidade $O(n \log_2 n)$ e não deve alterar as posições dos elementos de p mas produzir o vetor pos tal que $pos[i]$ indica o índice do elemento que estaria na posição i na sequência ordenada. Por exemplo, para $p = [5, -4, 7, 3, 9, -1, 13, 18]$, obteria $pos = [1, 5, 3, 0, 2, 4, 6, 7]$.

Este algoritmo não pode ser estendido para conjuntos de pontos no plano e, por isso, vamos descrever um método alternativo (**método de Shamos**, 1975) baseado em "divisão e conquista". Esse método vai explorar propriedades geométricas do problema para poder realizar a fase de combinação em $O(n)$.





Para facilitar a descrição, assumimos que os pontos de $\mathcal{P}$ se encontram em *posição geral* o que, neste caso, significa que não haverá dois pontos com a mesma abcissa nem dois pontos com a mesma ordenada.

Numa fase de pré-processamento, o método começa por ordenar os pontos por ordem crescente de abcissa e por ordem decrescente de ordenada, o que pode ser feito em $O(2n \log_2 n) = O(n \log_2 n)$. A ordenação é efetuada apenas uma vez (antes de iniciar o processo recursivo).

Para determinar o par mais próximo em $\mathcal{P}$, parte o conjunto em dois conjuntos $\mathcal{P}_1$ e $\mathcal{P}_2$ usando a recta $x = x_m$, em que x_m é próximo da mediana das abcissas. Cada conjunto fica com $n/2$ pontos, se n for par, e com $\lceil n/2 \rceil$ ou $\lfloor n/2 \rfloor$, se for ímpar. O valor de x_m é a abcissa do ponto mais à direita em $\mathcal{P}_1$ (conjunto à esquerda). A seguir, determina o par mais próximo em $\mathcal{P}_1$ e em $\mathcal{P}_2$, recursivamente. Sendo δ_1 e δ_2 as distâncias mínimas encontradas para $\mathcal{P}_1$ e $\mathcal{P}_2$, toma $\delta = \min(\delta_1, \delta_2)$ e filtra $\mathcal{P}$ (ordenado por coordenada y) para obter a lista de pontos que se situam na faixa $]x_m - \delta, x_m + \delta[\times\mathbb{R}$. Nesta faixa poderá ser necessário comparar pontos de $\mathcal{P}_1$ com pontos de $\mathcal{P}_2$. Fora desta faixa, a distância entre qualquer par de pontos $(p_1, p_2) \in \mathcal{P}_1 \times \mathcal{P}_2$ seria sempre pelo menos δ.

Supondo então que os pontos situados na faixa estão por ordem decrescente de ordenada (poderia também ser por ordem crescente), será suficiente comparar cada ponto com um número constante K de pontos que lhe seguem, para uma constante K segura. Pode-se mostrar que $K = 15$ servirá. Se K fosse maior, os pontos já estariam a uma distância superior a δ, como sugere a figura acima, à direita, já que cada quadrado $\frac{\delta}{2} \times \frac{\delta}{2}$ na grelha teria no máximo um ponto da faixa. De facto, não pode ter mais do que um porque qualquer par de pontos que pertencesse a um desses quadrados estaria a uma distância não superior ao comprimento da sua diagonal, ou seja, $\frac{\sqrt{2}}{2}\delta$, o que é absurdo, pois cada quadrado está contido ou em $\mathcal{P}_1$ ou em $\mathcal{P}_2$ (aqui, consideramos que se um ponto estiver sobre uma aresta vertical partilhada por dois quadrados, pertence ao quadrado da esquerda, e se estiver sobre uma aresta horizontal, pertence ao quadrado superior). Vemos que se dois pontos estiverem separados por 15 quadrados ou mais (e, portanto, 15 posições ou mais no vetor ordenado) então a distância entre eles será sempre maior do que $\frac{3}{2}\delta$. Logo, basta comparar cada ponto com os 15 seguintes. Este valor de K pode ser reduzido (consultar a bibliografia recomendada), mas, para a análise da complexidade assintótica só nos interessa perceber que a existência desta constante garante que a fase de combinação possa ser realizada em tempo $O(n)$.

Capítulo 4

Estruturas de Dados (Revisão)

4.1 Vetores, matrizes, listas ligadas, pilhas e filas

Vetores, matrizes, listas ligadas (simples, duplamente ligadas, ou circulares), filas (com disciplina FIFO "first-in first-out") e pilhas (LIFO, "last-in first-out") são exemplos de estruturas de dados usadas para suportar alguns dos algoritmos que vamos estudar. Nestes apontamentos, usamos a designação de vetor como tradução de *array* unidimensional. Para cada tipo de estrutura, é necessário conhecer a complexidade de algumas operações básicas como por exemplo: criar a estrutura; inicializá-la, se necessário; consultar ou procurar um elemento; alterar um elemento; remover e inserir um elemento; e verificar se contém elementos (não está vazia). A complexidade pode depender de uma implementação particular mas, em geral, gostariamos de dispor de uma implementação que fosse eficiente (se possível, com complexidade assintótica ótima).

4.2 Alguns exemplos de aplicação

4.2.1 Contar o número de ocorrências de cada valor de um intervalo

Na secção 2.4, apresentámos dois algoritmos para contar o número de ocorrências de cada valor de um intervalo $[a, b]$ num vetor. O algoritmo mais rápido efetua uma única passagem no vetor e tem complexidade $\Theta(m + n)$, sendo m o número de valores do intervalo e n o número de valores a analisar. Como $\Theta(m + n) = \Theta(n)$ se $m \in O(n)$, então nesse caso o algoritmo tem complexidade ótima. O algoritmo trivial (ou força bruta), em que se efetua uma descida por cada elemento a contar, é de ordem $\Theta(mn)$. Assim, teria complexidade $O(n^2)$ se $m \in O(n)$, e mesmo $\Theta(n^2)$ se $m \in \Theta(n)$. Na análise da complexidade assintótica deste algoritmo, importa perceber que o a instrução $t \leftarrow t + 1$ é executada no máximo n vezes no **total** das m chamadas de CONTAOCORRENCIAS (pois o vetor só tem n elementos)

mas a complexidade de cada chamada é dominada pela necessidade de percorrer o vetor e, por isso, é $\Theta(n)$.

4.2.2 Decompor uma permutação em ciclos ("Disse que disse")

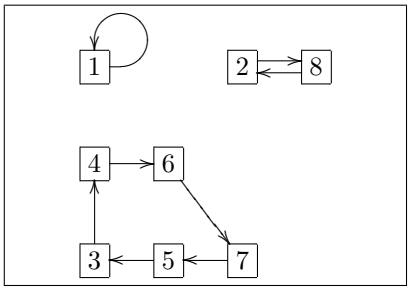
Uma permutação de $1, 2, 3, \ldots n$ corresponde a uma bijeção de π de $\{1, 2, \ldots, n\}$ em si mesmo. Por exemplo, em $(1, 8, 4, 6, 3, 7, 5, 2)$, teriamos $\pi(1) = 1$, $\pi(2) = 8$, $\pi(3) = 4$, ..., $\pi(8) = 2$. Cada permutação é definida por uma sequência de ciclos. Por exemplo, $(1, 8, 4, 6, 3, 7, 5, 2)$ é formada por três ciclos $(1)(28)(34675)$

A permutação pode ser representada por um vetor x, com $x[i] = \pi(i)$. Para cada i, podemos procurar o ciclo a que i pertence, se ainda não tiver sido encontrado. Notar que $i, x[i], x[x[i]], \ldots$ terminará num elemento k que fechará o ciclo, isto é, tal que $x[k] = i$. Essa é ideia do algoritmo seguinte, onde $visitado[i]$ indica se i já está num ciclo visto.

CICLOSPERMUT(x, n)

Para $i \leftarrow 1$ até n fazer $visitado[i] =$ `false`;
Para $i \leftarrow 1$ até n fazer
  Se ($visitado[i] =$ `false`) então
    $inicio \leftarrow i$;
    $corr \leftarrow i$;
    escrever(`"("`);
    Repita
      escrever($corr$);
      $visitado[corr] =$ `true`;
      $corr \leftarrow x[corr]$;
    até ($corr = inicio$);
    escrever(`")"`);

Exemplo: (18463752)



Cada elemento é analisado no máximo duas vezes: da primeira vez é incluído num ciclo e da segunda apenas se verifica se já está nalgum ciclo (i.e., anotado como visitado). Assim, o tempo de execução é $\Theta(n)$ e é ótimo. Qualquer algoritmo que resolva o problema tem complexidade temporal $\Omega(n)$ pois terá de analisar a permutação para determinar os ciclos. Portanto, um algoritmo que resolva o problema e tenha com complexidade $O(n)$ é ótimo.

No problema "Disse que disse", este algoritmo deve ser adaptado para implementar uma solução com complexidade espacial e temporal $\Theta(n)$.

4.2.3 Eliminar ciclos de um percurso ("Cigarras tontas")

No problema "Cigarras tontas", é necessário analisar um percurso e retirar os ciclos que se forem formando à medida que os locais forem sendo visitados. O percurso pode ser arbitrariamente grande, mas é conhecido o número máximo

de locais existentes (seja N esse número) e também se sabe que os locais se identificam por números de 1 a L, mas não necessariamente consecutivos. No enunciado, $N \leq 30$ e $L \leq 10000$. Supõe-se ainda que o percurso envolve pelo menos dois locais distintos.

Apresentamos a seguir dois algoritmos. O primeiro algoritmo utiliza um vetor $caminho[\cdot]$, de N elementos, para guardar o caminho relevante i.e., a parte do percurso que constitui a resposta se o local seguinte fosse `FimSequencia` (o identificador de que a sequência terminou). Para cada local novo, verifica se já ocorre nesse caminho. Se já ocorrer, trunca **implicitamente** o caminho (por atualização do valor de $nnos$). Se não ocorrer, coloca-o no fim caminho.

```
ELIMINACICLOS_A(caminho)
    ler(inicio);
    caminho[0] ← inicio;
    nnos ← 1;
    ler(novo);
    Enquanto (novo ≠ FimSequencia) fazer
        nnos ← INSERE(caminho, nnos, novo);
        ler(novo);
    ESCREVECAMINHO_A(caminho, nnos);
    /* ou retorna nnos */
```

```
INSERE(caminho, nnos, novo)
    Para i ← 0 até nnos − 1 fazer
        Se caminho[i] = novo então
            retorna i + 1;
    caminho[nnos] ← novo;
    retorna nnos + 1;
```

```
ESCREVECAMINHO_A(caminho, nnos)
    Para i ← 0 até nnos − 1 fazer
        escrever(caminho[i]);
```

O segundo algoritmo utiliza um vetor $caminho[\cdot]$, de $L + 1$ elementos, para guardar o identificador do local que segue cada local no caminho relevante (se i pertencer a esse caminho, então $caminho[i]$ indica o local que segue i no caminho). O início do caminho relevante é guardado na variável $inicio$ e é sempre o identificador do local de partida. Note-se que, o conteúdo e a dimensão da variável $caminho$ são muito diferentes nos dois algoritmos.

```
ELIMINACICLOS_B(caminho)
    ler(inicio);
    corr ← inicio;
    Repita
        ler(novo);
        caminho[corr] ← novo;
        corr ← novo;
    até (corr = FimSequencia);
    ESCREVECAMINHO_B(caminho, inicio);
    /* ou retorna inicio */
```

```
ESCREVECAMINHO_B(caminho, p)
    Enquanto (p ≠ FimSequencia) fazer
        escrever(p);
        p ← caminho[p];
```

Em cada iteração do ciclo "Repita", a sequência $inicio$, $caminho[inicio]$, $caminho[caminho[inicio]]$, ..., que

termina em $corr$, define o caminho relevante encontrado até ao momento (i.e., até ao início dessa iteração). Tal caminho constitui uma lista ligada implementada sobre o vetor. Com a utilização deste vetor, consegue-se descartar um ciclo em tempo $O(1)$ pois não é preciso analisar o caminho guardado. De facto, o vetor permite aceder a $caminho[corr]$ em tempo constante e, se $corr$ já estiver no caminho, a atribuição $caminho[corr] \leftarrow novo$ descarta o ciclo em $O(1)$. Quando se define $novo$ como o local a visitar após $corr$, ignora-se toda a volta dada. O último valor lido, `FimSequencia` (que é 0 em "Cigarras tontas"), é usado para terminar o caminho relevante. Deste modo, no fim, a lista que tem $inicio$ como primeiro elemento fica bem formada, terminando por `FimSequencia`. Como a atribuição $caminho[corr] \leftarrow novo$ é feita antes de testar $novo$, um ciclo que se formasse mesmo no fim (i.e. imediatamente antes de encontrar o terminador) também seria descartado, como se espera de um algoritmo correto.

Complexidade de ELIMINACICLOS_A

Em cada iteração do ciclo "Enquanto", a complexidade é dominada pela execução da função INSERE, a qual tem complexidade $O(nnos)$. No melhor caso, a complexidade de INSERE é $O(1)$ em todas as chamadas. Tal verifica-se, por exemplo, para instâncias em que a sequência é da forma "$1, 2, 1, 2, 1, 2, 1, 2, \ldots, 1, 2, 1, 2, 3$". Para instâncias como "$1, 2, 3, \ldots, N-2, N-1, N-2, N-1, N-2, \ldots, N-1, N-2, N-1, N$", cada chamada de INSERE tem complexidade $\Omega(N)$, excepto para alguns valores iniciais. A partir daqui, não é difícil ver que ELIMINACICLOS_A tem complexidade $O(NS)$, e que no pior caso é $\Theta(NS)$, onde S designa o número total de locais lidos. É de notar que, se $S \in \Theta(N)$, podemos concluir que este algoritmo tem complexidade $\Theta(N^2)$ no pior caso. Contudo, se $N \in O(1)$, então podemos concluir tem complexidade $\Theta(S)$. No problema "Cigarras Tontas", sabe-se que $N \leq 30$, em qualquer instância. Portanto, uma solução baseada neste algoritmo teria complexidade temporal assintótica $\Theta(S)$.

Complexidade de ELIMINACICLOS_B

A complexidade temporal de ELIMINACICLOS_B é de ordem $\Theta(S)$. É importante notar que não é feita qualquer inicialização do vetor $caminho$ e que se assume que é possível criá-lo (ou reservar esse espaço) em tempo constante. Se se inicializasse o vetor $caminho$, por exemplo, para uma "limpeza" que garantisse que todas as posições tinham inicialmente o valor 0 (o que não tem qualquer relevância para a correção do algoritmo e, por isso, é pura perda de tempo), então a complexidade poderia passar a ter de ser definida como $\Theta(S + L)$.

Dualidade Espaço-Tempo

Quando se compara a complexidade temporal dos dois algoritmos apresentados, $O(NS)$ e $\Theta(S)$, e ambos $\Omega(S)$, pareceria natural optar pelo segundo. No entanto, é preciso não esquecer que **se trocou tempo por espaço**. A complexidade espacial do primeiro é $\Theta(N)$ e a do segundo é $\Theta(L)$ e, por definição de N e L, sabemos que N não pode exceder L, mas, como acontece no problema "Cigarras tontas", L pode ser muito maior do que N.

Importa aqui realçar que, qualquer solução que começasse por (tentar) guardar o percurso todo para só posteriormente o analisar, nunca seria uma opção razoável. Na prática, conduziria sempre a implementações com tempo de execução pior e, muito provavelmente, a um gasto de memória excessivo (e descontrolado, pois não se conhece um majorante para S), o que poderia inviabilizar a execução do programa em algumas instâncias. Embora a análise de algoritmos foque muitas vezes mais a complexidade temporal do que a espacial, é necessário ponderar ambas.

4.2.4 Analisar o emparelhamento de parentesis em expressões ("Casamentos perfeitos")

(analisado nas aulas; utilização de uma pilha)

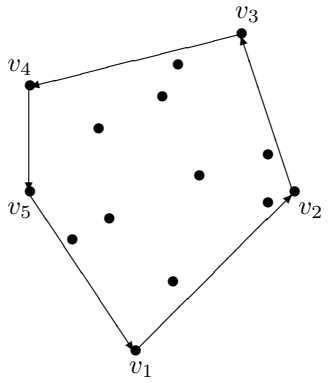
4.2.5 Gerir uma fila com disciplina FIFO ("Combate final")

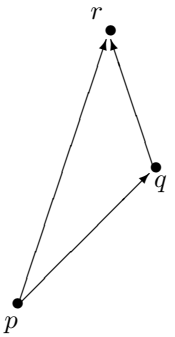
(analisado nas aulas, utilização de duas filas FIFO, uma para cada baralho; também proposto "Barbearia Chic"... simular fila de espera limitada)

4.2.6 Determinar o invólucro convexo de n pontos no plano (por *Graham scan*)

Um conjunto de pontos $S \subseteq \mathbb{R}^n$ é **convexo** se, quaisquer que sejam $a, b \in S$, o **segmento de recta** $[ab]$ está contido em S. Por definição, o segmento de recta $[ab]$ é $\{a+\lambda(b-a) \mid \lambda \in [0,1]\}$. Logo, S é convexo se $\lambda a + (1-\lambda)b \in S$, para todo $\lambda \in [0,1]$, quaisquer que sejam $a, b \in S$.

Chama-se **invólucro convexo** (*convex hull*) de $S \subseteq \mathbb{R}^n$ ao menor conjunto convexo que contém S ("menor" no sentido da inclusão de conjuntos). $\mathcal{CH}(S)$ designará o invólucro convexo de S. Obviamente, se S é convexo então S coincide com o seu invólucro convexo, isto é, $\mathcal{CH}(S) = S$. O invólucro convexo de um conjunto **finito** de pontos $\mathcal{P}$ de $\mathbb{R}^2$ é uma região poligonal e, como tal, fica bem caraterizado se definirmos a sua fronteira. No exemplo abaixo, a fronteira de $\mathcal{CH}(\mathcal{P})$ é definida pela sequência de pontos $v_1, v_2, v_3, v_4, v_5, v_1$, se for percorrida no sentido anti-horário, i.e., no sentido contrário ao dos ponteiros do relógio (sigla *CCW*, em Inglês).





r está à esquerda de $\overrightarrow{pq}$

$$\vec{u} \cdot \vec{v} = x_1 y_1 + x_2 y_2$$

$$\vec{u} \times \vec{v} = \begin{vmatrix} x_1 & y_1 & 0 \\ x_2 & y_2 & 0 \\ \vec{i} & \vec{j} & \vec{k} \end{vmatrix} = (x_1 y_2 - y_1 x_2)\vec{k}$$

O algoritmo de Graham explora as propriedades geométricas de $\mathcal{CH}(\mathcal{P})$ para obter a sequência dos seus vértices. *Um polígono é convexo se e só se quando se percorre a sua fronteira no sentido CCW, se vira à esquerda em cada vértice*. Por vezes, o termo *polígono* designa apenas a fronteira da região poligonal que delimita, mas aqui consideramos que engloba também o seu interior.

Teste de viragem. Dados três pontos (p, q, r) no plano, não colineares, o sinal da componente não nula do produto vetorial $\overrightarrow{pq} \times \overrightarrow{pr}$ permite-nos verificar se (p, q, r) constitui uma **viragem à esquerda**. Se o sinal for positivo, a viragem é à esquerda e, se for negativo, é à direita. Estamos a supôr que o referencial $(\mathcal{O}, \vec{i}, \vec{j}, \vec{k})$ tem orientação positiva e é ortonormado, como o canónico. Recordamos também que o vetor $\overrightarrow{pq}$ tem coordenadas $(x_q - x_p, y_q - y_p, z_q - z_p)$ se os pontos p e q tiverem coordenadas (x_p, y_p, z_p) e (x_q, y_q, z_q).

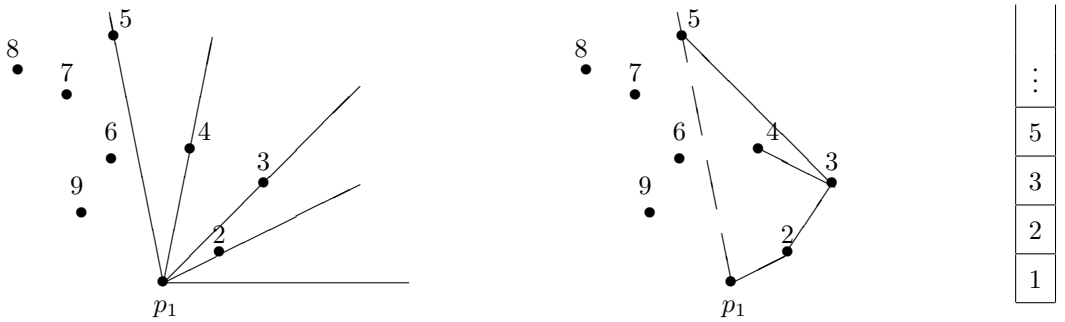
Para implementar o teste de viragem para (p, q, r), não é necessário começar por determinar os vetores $\overrightarrow{pq}$ e $\overrightarrow{pr}$. Podemos calcular o determinante seguinte cujo valor é igual ao valor da coordenada não nula de $\overrightarrow{pq} \times \overrightarrow{pr}$, quando p, q e r são pontos não colineares do plano $(\mathcal{O}, \vec{i}, \vec{j})$, como se vê no seu desenvolvimento.

$$\begin{vmatrix} x_p & y_p & 1 \\ x_q & y_q & 1 \\ x_r & y_r & 1 \end{vmatrix} = \begin{vmatrix} x_p & y_p & 1 \\ x_q - x_p & y_q - y_p & 0 \\ x_r - x_p & y_r - y_p & 0 \end{vmatrix} = (x_q - x_p)(y_r - y_p) - (y_q - y_p)(x_r - x_p)$$

Algoritmo de Graham

Na descrição do algoritmo de Graham (*Graham scan*), vamos assumir que os pontos de $\mathcal{P}$ estão em **posição geral**, o que neste caso significa que não existem três (ou mais) pontos colineares nem dois com a mesma ordenada ou abcissa. Esta é uma prática comum na descrição de algoritmos para problemas geométricos e não é díficil adaptar o algoritmo para permitir tratar os restantes casos, como veremos mais à frente.

Os pontos de $\mathcal{P}$ que têm abcissa mínima e os que têm abcissa máxima, bem como os que têm ordenada mínima e máxima, pertencem sempre à fronteira de $\mathcal{CH}(\mathcal{P})$. Na versão do algoritmo de Graham que vamos apresentar, o ponto que tem **ordenada mínima** vai desempenhar um papel central. Seja p_1 esse ponto. Quando se efetua um varrimento rotacional (em CCW) num polígono convexo a partir de um vértice, os vértices do polígono vão sendo visitados pela ordem em que seriam se se percorresse a fronteira no sentido anti-horário a partir desse vértive. O mesmo acontece se esse vértice for p_1 e os pontos de $\mathcal{P}$ forem visitados por ordem crescente de ângulo polar relativamente a um referencial com origem em p_1 (como se ilustra na figura abaixo). A fronteira de $\mathcal{CH}(\mathcal{P})$ é definida por vértices que são consecutivos ou passam a ser consecutivos se as cadeias que formam zonas não convexas forem retiradas. Estas cadeias são detectadas porque estão associadas a viragens à direita (por exemplo, $(3, 4, 5)$ no caso seguinte).



Para ordenar os pontos por ordem de ângulo polar crescente relativamente a p_1, não é preciso calcular os ângulos explicitamente para depois os comparar. Basta notar que p tem ângulo polar menor do que q se (p_1, p, q) constituir uma viragem à esquerda. Assim, evitamos os erros numéricos inerentes ao cálculo de ângulos.

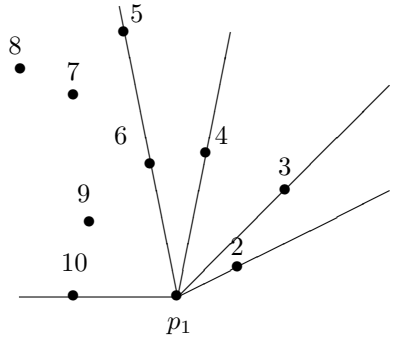
GRAHAM-SCAN($\mathcal{P}$)

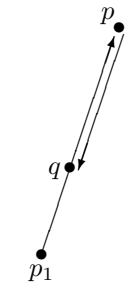
1. Seja p_1 o ponto de $\mathcal{P}$ com menor ordenada
2. Seja $\{p_2, \ldots, p_n\}$ o conjunto $\mathcal{P} \setminus \{p_1\}$ ordenado em CCW em torno de p_1
   (i.e., por ordem crescente de ângulo polar)
3. Criar uma pilha S e inserir p_1, p_2, p_3 em S (p_3 fica no topo)
4. Para $i \leftarrow 4$ até n fazer
   /* sendo w o topo atual da pilha e w^- o elemento abaixo dele */
5. Enquanto (w^-, w, p_i) não for viragem à esquerda fazer
6. Retirar w de S
7. Colocar p_i em S
8. retorna S

No varrimento rotacional (passos 4.–7.), cada elemento que seja retirado da pilha no passo 6. não voltará a ser colocado novamente na pilha. Em cada iteração do ciclo "Para", entra um novo elemento para pilha (podendo sair um, vários ou nenhum). O número total de testes de viragem à esquerda no ciclo "Para" é maior ou igual que $n - 3$ e inferior a $2n$. Portanto, a complexidade dos passos 1. e 3.–8. é $\Theta(n)$ e a complexidade do algoritmo é dominada pela complexidade da ordenação (no passo 2.). Assim, GRAHAM-SCAN($\mathcal{P}$) é de ordem $O(n \log n)$ se a ordenação for efetuada em $O(n \log n)$, por exemplo, por aplicação de *merge sort* (adaptado para que a comparação de ângulos polares determine a relação de ordem).

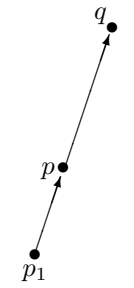
Pontos em posição não geral. Se houver mais do que um ponto com a mesma ordenada, p_1 será o que tem a abcissa maior entre esses pontos. Se houver três pontos colineares sobre um mesmo raio com origem em p_1, então $\overrightarrow{p_1p} \times \overrightarrow{p_1q} = \vec{0}$. Nesse caso, na ordenação, aparecerá primeiro o ponto que estiver mais afastado de p_1. Para o determinar, basta analisar o sinal do *produto interno*, i.e., produto escalar, $\overrightarrow{p_1p} \cdot \overrightarrow{pq}$, pois, o produto escalar permite-nos

calcular a projeção de um vetor na direção do outro. Se for negativo, p é o mais afastado. Se for positivo, q é o mais afastado. A noção de viragem à esquerda é também adaptada: se $\overrightarrow{p_1p} \times \overrightarrow{p_1q} = \vec{0}$ assume-se que a viragem é à esquerda se $\overrightarrow{p_1p} \cdot \overrightarrow{pq} < 0$.





$\overrightarrow{p_1p} \cdot \overrightarrow{pq} < 0$



$\overrightarrow{p_1p} \cdot \overrightarrow{pq} > 0$

Capítulo 5

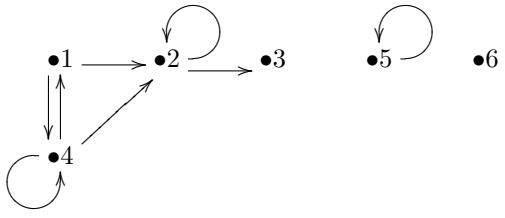
Algoritmos em Grafos

Os grafos representam um modelo fundamental em computação, surgindo em problemas de caminhos (por exemplo, caminho mínimo ou máximo numa rede de transportes), representação de redes (por exemplo, de computadores, de tráfego rodoviário), descrição de sequência de programas, desenho de circuitos integrados, representação de relações (por exemplo, ordenação, emparelhamento), análise sintática de linguagens (árvores sintáticas), etc.

5.1 Revisão de nomenclatura e de alguns resultados

5.1.1 Grafos dirigidos (digrafos)

Um **grafo dirigido** (digrafo) é definido por um par $G = (V, E)$ em que V é um conjunto de vértices (nós) e E um conjunto de ramos (arestas ou arcos) orientados que ligam pares de vértices de V, não existindo mais do que um ramo com a mesma orientação a ligar dois vértices. Quando o conjunto de vértices é **finito** (o que implica que o conjunto de ramos também o seja), o grafo é **finito**, podendo-se desenhar. Por exemplo,



é um grafo em que o conjunto de ramos é $E = \{(1,4),(1,2),(2,2),(2,3),(4,1),(4,4),(5,5)\}$ e o conjunto de vértices é $V = \{1,2,3,4,5,6\}$.

Se $(x, y) \in E$, então o vértice x é a **origem** e o vértice y o **fim** do ramo (x, y), dizendo-se que x e y são os **extremos** do ramo (x, y) e que o ramo é **incidente** em y. Os ramos da forma (x, x), isto é com origem e fim no

mesmo vértice, chamam-se **lacetes**. No exemplo, são três os ramos **incidentes no vértice** 2, isto é, que terminam no vértice 2. Este vértice tem grau de entrada três e grau de saída dois. O **grau de entrada** de um vértice é o número de ramos que têm fim nesse vértice. O **grau de saída** é o número de ramos que têm origem nesse vértice. No exemplo, os vértices 5 e 6 são **vértices isolados**, pois não são origem nem fim de nenhum ramo com algum extremo noutro vértice (têm grau de entrada e de saída zero).

Um **percurso (finito)** é uma sequência finita formada por um ou mais ramos do grafo tal que o extremo final de qualquer ramo coincide com o extremo inicial do ramo seguinte na sequência. Um percurso num grafo dirigido fica bem identificado se se indicar a sequência de vértices por onde passa. Para o exemplo, o percurso $(1,4),(4,1),(1,2),(2,2),(2,2),(2,3)$ pode ser dado pela sequência de vértices $1,4,1,2,2,2,3$.

A **origem do percurso** é a origem do primeiro ramo nesse percurso e o **fim do percurso** é o fim do último ramo no percurso. Um percurso com origem em v e fim em w é um **percurso de** v **para** w. Chama-se **circuito** a um percurso em que a origem e o fim coincidem. O **comprimento** (ou **ordem**) de um percurso (finito) é o número de ramos que o constituem. Assim, por exemplo, $1,4,1,2,2,2,3$ tem origem em 1, fim em 3 e comprimento seis, não sendo um circuito. Note que, qualquer percurso de comprimento k envolve $k+1$ vértices, não necessariamente distintos. No percurso $1,4,1,2,2,2,3$, alguns vértices **são visitados** mais do que uma vez, concretamente os vértices 1 e 2, e o ramo $(2,2)$ é usado mais do que uma vez.

Sendo u e v vértices de um grafo dirigido G, diz-se que v **é acessível de** u em G se e só se $u=v$ ou existe em G algum percurso de u para v. Chama-se **componente fortemente conexa** a um subgrafo de G em que qualquer vértice é acessível de qualquer outro vértice e que é máximo (não se pode acrescentar outros vértices sem quebrar essa propriedade). O grafo é **fortemente conexo** se e só se tem uma única componente fortemente conexa (i.e., qualquer vértice de G é acessível de qualquer outro vértice de G).

Um **subgrafo** de um grafo $G=(V,E)$ é um grafo $G'=(V',E')$ em que $V'\subseteq V$ e $E'\subseteq E$. Também se pode definir subgrafo de um multigrafo (dirigido ou não). Alguns autores requerem que E' tenha todos os ramos de E que unem vértices em V', chamando *subgrafo parcial* se algum ramo que liga vértices em V' não pertencer a E'.

Na terminologia moderna da teoria de grafos, um **caminho** é um percurso sem repetição de vértices e um **ciclo** é um percurso fechado que não tem repetição de vértices com excepção do inicial e final. Qualquer percurso com $|V|$ ramos é um ciclo ou contém um ciclo. Um grafo dirigido é **acíclico** se e só se não contém ciclos (nem lacetes).

A terminologia de grafos é pouco consistente, pelo que é sempre necessário contextualizar cada termo usado na bibliografia da área. Na terminologia clássica de grafos, *caminho* era sinónimo de percurso. Chamava-se *caminho simples* a um percurso em que nenhum ramo ocorre duas ou mais vezes e *caminho elementar* a um percurso em que nenhum vértice ocorre duas ou mais vezes. De acordo com a mesma terminologia, *ciclo* era sinónimo de circuito, chamando-se "ciclo simples" se fosse também caminho simples.

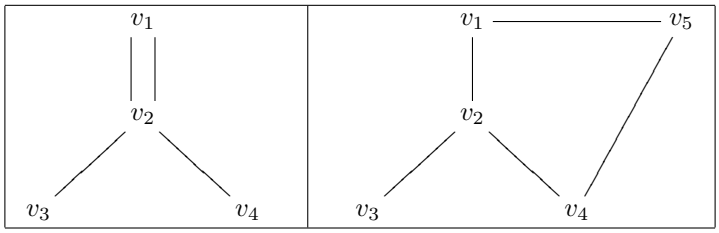
O conceito de grafo pode ser generalizado para permitir a existência de vários ramos com a mesma origem e fim.

Um **multigrafo dirigido** é um terno (V, A, Φ) em que V é um conjunto de **vértices** (ou **nós**), A um conjunto de **ramos** (ou **arestas** ou **arcos**) e Φ é uma **função** de A em $V \times V$ que a cada ramo associa um par de vértices. Um percurso será definido por uma sequência de ramos, não podendo ser definido apenas pela sequência de vértices por onde passa.

5.1.2 Grafos não dirigidos

Um **multigrafo não dirigido** é um terno (V, E, Φ) em que V é um conjunto de vértices (nós), E um conjunto de ramos (arestas) e Φ é uma função de E no conjunto de pares não ordenados de elementos de V. Escreve-se $\Phi(\alpha) = \langle u, v\rangle$. Usaremos $\langle u, v\rangle$ ou $\{u, v\}$ para representar um par não ordenado. Os nós u e v dizem-se **extremos** do ramo α. O **grau de um vértice** num multigrafo *não dirigido* é o número de ramos que têm esse vértice como extremo.

Um **grafo não dirigido** é um multigrafo não dirigido (V, E, Φ) em que Φ é uma função injetiva. Cada par de vértices não ordenado fica associado por Φ quando muito a um ramo. Habitualmente, representamos um grafo não dirigido por um par (V, E).

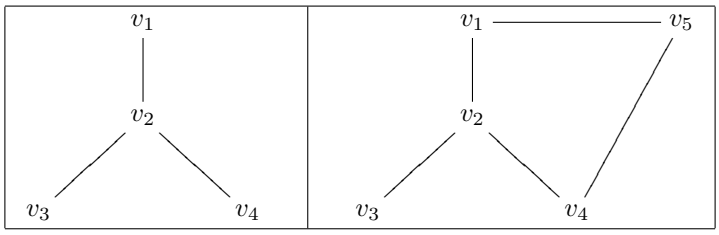


À esquerda está representado um multigrafo não dirigido que não é grafo dirigido, já que há mais do que um ramo a ligar v_1 e v_2. À direita temos um grafo não dirigido.

A um multigrafo não dirigido $G = (V, E, \Phi)$ podemos associar um multigrafo *dirigido* $G_A = (V, A, \Psi)$ tal que a cada ramo de E unindo vértices diferentes correspondem dois ramos em A, e vice-versa. Se $\Phi(\alpha) = \langle u, v\rangle$ e $u \neq v$ então existem dois ramos $\vec{\alpha_1}$ e $\vec{\alpha_2}$ em A tais que $\Psi(\vec{\alpha_1}) = (u, v)$ e $\Psi(\vec{\alpha_2}) = (v, u)$. O multigrafo G_A diz multigrafo adjunto de G. Se G for um grafo não dirigido então G_A é um grafo dirigido simétrico.

Um **percurso** (finito) num multigrafo não dirigido é uma sequência finita formada por um ou mais ramos do multigrafo que corresponde a um percurso no multigrafo adjunto. Num grafo não dirigido $(\langle v_0, v_1\rangle, \langle v_1, v_2\rangle, \ldots \langle v_{k-1}, v_k\rangle)$ representa um percurso **entre** v_0 e v_k. Se se indicar a sequência $(v_0, v_1, v_2, \ldots, v_{k-1}, v_k)$ de vértices por onde passa, pode ser entendido como um percurso de v_0 **para** v_k. O **comprimento** de um percurso (finito) é o seu número de ramos. Um **circuito** num grafo não dirigido é um percurso fechado $(\langle v_0, v_1\rangle, \langle v_1, v_2\rangle, \ldots \langle v_{k-1}, v_k\rangle)$ que v_0 e v_k coincidem. Um **ciclo** (por vezes designado *ciclo simples*) é um circuito que não tem repetição de vértices com excepção do inicial e final. Um grafo não dirigido é **acíclico** se e só se não contém ciclos (nem lacetes). O grafo representado à esquerda tem circuitos mas é acíclico, pois não tem ciclos. O grafo representado à direita não é acíclico porque contém

o ciclo (v_1, v_2, v_4, v_5).



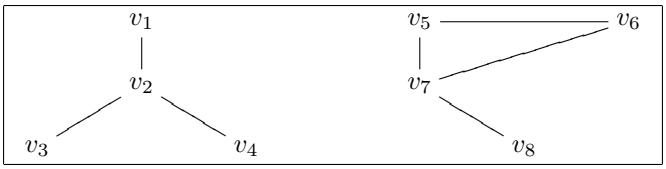
Grafos não dirigidos conexos e componentes conexas

Como anteriormente, diz-se que v **é acessível de** u em $G = (V, E)$ se e só se $u = v$ ou existe em G algum percurso entre u e v. Facilmente se reconhece que esta relação de "acessibilidade" é uma **relação de equivalência** em V (reflexiva, simétrica e transitiva). Reflexiva porque qualquer vértice é acessível de si mesmo. Simétrica porque se u é acessível de v então v é acessível de u (pois, se $u \neq v$, o percurso de v para u pode ser percorrido em sentido inverso para chegar de u a v). A transividade resulta da observação de que se existe um percurso de u para v e um percurso de v para w, então a junção dos dois percursos constitui um percurso de u para w. Note-se que a relação de acessibilidade em grafos dirigidos pode não ser de equivalência (ainda que a simetria do grafo não seja essencial para o poder ser).

Um grafo não dirigido G chama-se **conexo** se e só se qualquer vértice é acessível de qualquer outro vértice nesse grafo. Observe-se que neste caso a relação de acessibilidade tem uma única classe de equivalência. Recordemos que se ρ é uma relação de equivalência definida num conjunto A, então x e y são equivalentes segundo ρ se $(x, y) \in \rho$. As classes de equivalência de ρ são conjuntos máximos de elementos equivalentes.

Em geral, cada classe de equivalência da relação de acessibilidade corresponde a um subgrafo de G que é conexo e que é máximo (no sentido de deixar de ser conexo se se tentar juntar mais algum vértice de G). Tais subgrafos dizem-se **componentes conexas** de G. Por abuso de linguagem, chamaremos ainda *componentes conexas* às classes de equivalência da relação de acessibilidade. O conjunto de vértices na componente identifica-a perfeitamente.

O grafo (não dirigido) seguinte tem duas componentes conexas.



A relação de acessibilidade não depende da existência de vários ramos entre dois vértices. A cada multigrafo não dirigido podemos associar um grafo não dirigido substituindo cada conjunto de ramos entre um mesmo par de vértices por apenas um ramo. Podemos ainda retirar todos os lacetes (se existissem). O grafo não dirigido assim definido determina a mesma relação de acessibilidade que o multigrafo dado.

Seja $G = (V, E)$ um grafo não dirigido finito. Apresentamos a seguir alguns resultados conhecidos envolvendo a noção de conectividade.

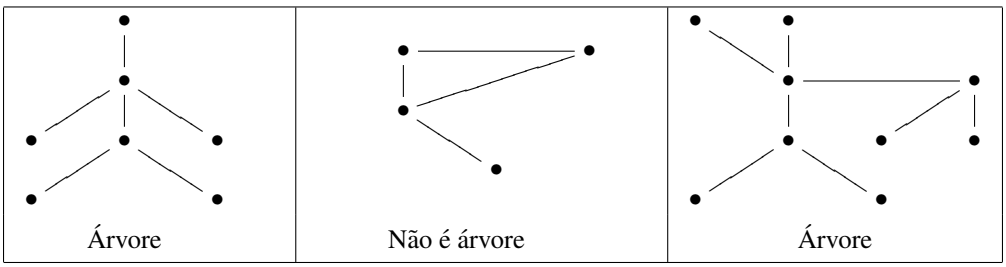
- Se G for conexo e $|E| \geq 1$ então se se retirar um ramo de G, o grafo resultante tem quando muito duas componentes conexas, e, se $v \in V$ tiver grau m, então se retirar v e todos os ramos com extremo em v, o grafo resultante tem quando muito m componentes conexas.

- É condição necessária mas não suficiente para que G seja conexo que $|E| \geq |V| - 1$. Os grafos conexos tais que $|E| = |V| - 1$ designam-se por árvores.

- Se G for conexo e não for acíclico então é possível retirar algum ramo a G de modo que o grafo resultante seja também conexo.

5.1.3 Árvores e árvores com raíz

Uma **árvore** é um grafo não dirigido *conexo* com n vértices e $n - 1$ ramos. Se tiver 1 vértice (e 0 ramos) diz-se árvore **degenerada**. Numa árvore não degenerada, os vértices com grau 1 chamam-se **folhas** e os de grau superior chamam-se *nós internos* ou simplesmente **nós**.

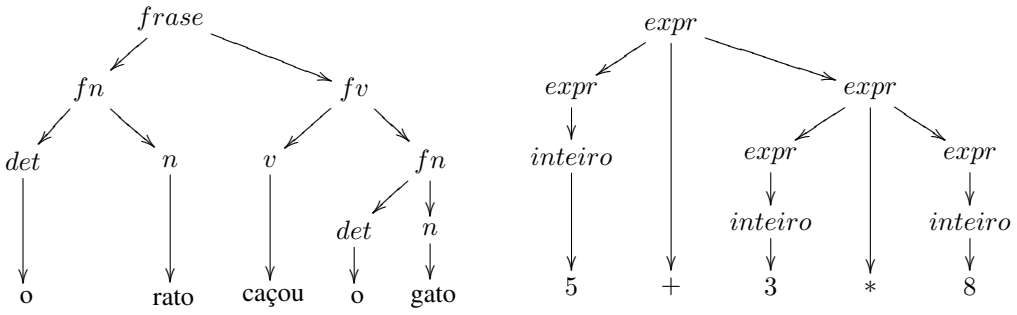


Pode-se mostrar que: qualquer árvore é um grafo acíclico; numa árvore não degenerada, existe um só caminho entre cada par de vértices distintos; se se juntar a uma árvore mais algum ramo sem alterar o conjunto de vértices, obtém-se um grafo com um e um só ciclo; uma árvore não degenerada tem pelo menos duas folhas.

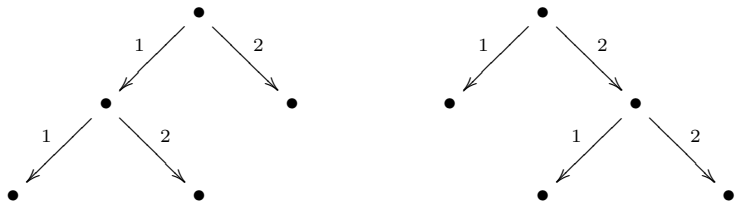
Uma **árvore com raíz** é um grafo dirigido com n vértices e $n - 1$ ramos e tal que existe um e um só vértice do qual todos são acessíveis. Tal vértice chama-se a **raíz** da árvore. As folhas numa árvore com raíz são os vértices com grau de saída zero. A raíz da árvore é o único vértice com grau de entrada zero. Numa árvore com raíz os **filhos** ou **descendentes *diretos* de um nó** v são os extremos finais de ramos com origem em v. Os **descendentes** de um nó são os filhos desse nó e os descendentes dos filhos desse nó. Um nó é **pai** dos seus filhos.

Uma **árvore ordenada** (ou **orientada**) é uma árvore com raíz e tal que o conjunto de ramos que saem dum vértice

está *totalmente ordenado*. As *árvores sintáticas* são exemplo de árvores ordenadas.



As árvores seguintes poderiam ser iguais se se considerasse que eram árvores com raíz, mas são diferentes se se considerar que são árvores ordenadas (e $1 < 2$).

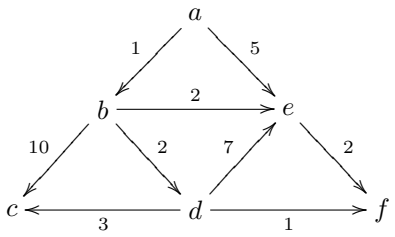


Os valores atribuídos aos ramos especificam a ordem subentendida (não se confundam com pesos). Em geral, não se colocam esses valores pois convenciona-se que a representação já obedece a essa ordem: *ou o ramo mais à esquerda é o maior ou é o menor.* Uma árvore ordenada diz-se **árvore n-ária** se o grau de saída de cada vértice não excede n. As árvores das figura anterior são árvores **binárias**. Quando se tem uma árvore binária é usual falar da **sub-árvore direita** e na **sub-árvore esquerda** dum nó.

5.1.4 Grafos com valores associados aos ramos

Um **multigrafo com valores associadas aos ramos** é um quarteto $(V, E, \Phi, \mathcal{L})$ em que (V, E, Φ) é um multigrafo, e $\mathcal{L} : E \to R$ é uma função de E num conjunto R, sendo R um conjunto de valores. É habitual R ser o conjunto dos números reais positivos, i.e., $R = \mathbb{R}^+$, mas, dependendo da aplicação, pode ser útil considerar que os valores nos ramos podem ser de qualquer tipo (símbolos, sequências de carateres, pares de números, números positivos e negativos, etc). Para um grafo (dirigido ou não dirigido) podemos reduzir o quarteto a um terno $(V, E, \mathcal{L})$. No grafo seguinte, o valor associado a cada ramo é um inteiro. Em certas aplicações, esse valor pode representar uma distância e, consequentemente, a soma dos valores nos ramos de um percurso seria a distância total percorrida. Noutras aplicações,

pode ser, por exemplo, uma capacidade, uma temperatura, etc.



5.2 Estruturas de dados para representação de grafos

No que se segue, $G = (V, E)$ é um grafo finito e os seus vértices são identificados por $v_1, \ldots, v_n$, com $n \geq 1$.

5.2.1 Representação por matriz de adjacências

Um grafo dirigido pode ser representado por uma matriz M, com n linhas e n colunas e elementos booleanos, designada por **matriz de adjacências**, tal que $M_{ij} = 1$ se $(v_i, v_j) \in E$ e $M_{ij} = 0$ se $(v_i, v_j) \notin E$. Um grafo não dirigido pode ser representado por uma matriz de adjacências simétrica (i.e., pela matriz de adjacências do seu grafo adjunto).

O espaço ocupado por esta representação é de ordem $\Theta(n^2)$, o que pode ser uma desvantagem se o grafo tiver poucas arestas (i.e., for esparso).

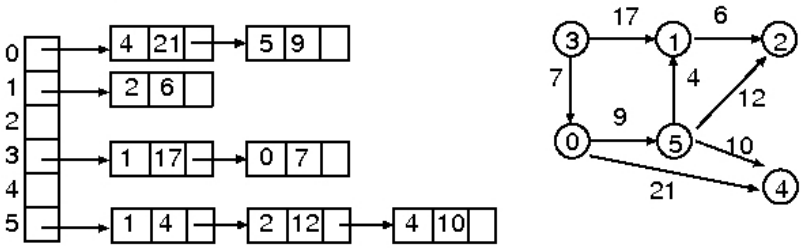
Um grafo dirigido com n vértices pode ter no máximo n^2 arestas. Se for não dirigido (e sem lacetes), pode ter no máximo $\frac{n(n-1)}{2}$. Os grafos que têm número máximo de arestas chamam-se **completos**. Um grafo cujo número de arestas é de ordem $\Omega(n^2)$ diz-se **denso**. Se o número de arestas for de ordem $O(n)$, diz-se **esparso**.

A representação por matriz de adjacências é mais adequada para grafos densos do que para grafos esparsos. Tem a vantagem de o acesso a um dado ramo ser feito em tempo constante, isto é, em $O(1)$, bem como a remoção de um ramo ou a inserção de um novo. Já a visita a todos os adjacentes de um nó requer $\Theta(n)$ e a inicialização da matriz requer $\Theta(n^2)$, o que pode constituir uma desvantagem em algumas aplicações.

5.2.2 Representação por listas de adjacências

Na representação por listas de adjacências, associa-se a cada vértice v a lista de nós **adjacentes** a v, i.e., a lista de nós w tais que $(v, w) \in E$.

No exemplo seguinte, os vértices são identificados por inteiros de 0 a $n - 1$, e a lista de adjacências de cada nó é uma lista ligada simples. Cada elemento da lista guarda a informação sobre o extremo final e, como se trata de um grafo com valores associados aos ramos, guarda também o valor associado ao arco. Na representação esquematizada, o identificador (i.e., apontador ou endereço) do nó inicial da lista de adjacentes de v_i fica guardado na posição i de um vetor de n posições (que poderá guardar também outra informação sobre os vértices).



O espaço ocupado por esta representação é de ordem $\Theta(n+m)$, em que m designa o número de arestas do grafo. A estrutura pode ser construída em tempo $\Theta(n+m)$. O acesso à lista de adjacentes de um nó é feito em tempo constante $O(1)$, mas a procura de um ramo com origem em v pode requerer $O(n_v)$, sendo n_v o número de adjacentes de v, o qual pode ser $O(n)$. A inserção de um ramo novo (não existente) pode ser feita em $O(1)$, por exemplo, se for inserido à cabeça da lista (ou se se conhecer o fim da lista e se inserir o elemento no fim). A remoção de um arco pode requerer $O(n)$ (se for necessário procurar o arco ou o arco que o precede na lista de adjacentes).

Vamos estudar alguns algoritmos elementares em grafos que têm complexidade $O(n+m)$, ou mesmo $\Theta(n+m)$, se o grafo for representado por listas de adjacências mas que teriam complexidade $O(n^2)$ e, respetivamente, $\Theta(n^2)$, se o grafo fosse representado por uma matriz de adjacências. Nos casos em que o grafo é denso, esses valores são iguais, do ponto de vista da complexidade assintótica. Mas, se o grafo for esparso, $\Theta(n+m)$ é $\Theta(n)$, e $\Theta(n)$ é melhor do que $\Theta(n^2)$.

O esquema apresenta uma implementação possível. Podem ser consideradas outras, com a mesma ideia principal (por exemplo, se a estrutura puder ser estática, a lista de adjacentes de v pode ser um vetor com n_v elementos).

5.3 Pesquisa em largura

A pesquisa em largura (*breadth-first search*) é uma estratégia genérica de análise exaustiva de um conjunto em que cada elemento tem um conjunto de elementos vizinhos (que pode ser vazio). Quando aplicada a partir de um elemento origem s, começa por visitar os vizinhos de s, a seguir os vizinhos dos vizinhos de s que ainda não tenham sido visitados, e sucessivamente. Assim, esta pesquisa corresponde à visita de um grafo em que os nós representam os elementos do conjunto e os ramos definem a relação de vizinhança. A visita a partir de um nó s é feita por ordem crescente de distância: na iteração k do processo, são visitados os nós v que estão a distância k da origem s, sendo a distância dada pelo número de ramos do caminho mais curto de s até v. Apresentamos a seguir os algoritmos em pseudocódigo. À esquerda, a função BFS_VISIT(s, G) para efetuar a busca a partir de s. À direita, uma função análoga, BFS_VISIT(s, G, Q), mas mais adequada para ser integrada com a função BFS(G) para visitar todo o grafo.

```
BFS_VISIT(s, G)
    Para cada v ∈ G.V fazer
        visitado[v] ← false;
        pai[v] ← NULL;
    visitado[s] ← true;
    Q ← MKEMPTYQUEUE();
    ENQUEUE(s, Q);
    Repita
        v ← DEQUEUE(Q);
        Para cada w ∈ G.Adjs[v] fazer
            Se visitado[w] = false então
                ENQUEUE(w, Q);
                visitado[w] ← true;
                pai[w] ← v;
    até (QUEUEISEMPTY(Q) = true );
```

```
ESCREVECAMINHO(v, pai)
    Se pai[v] ≠ NULL então
        ESCREVECAMINHO(pai[v], pai);
        escrever(v);
```

```
BFS(G)
    Para cada v ∈ G.V fazer
        visitado[v] ← false;
        pai[v] ← NULL;
    Q ← MKEMPTYQUEUE();
    Para cada v ∈ G.V fazer
        Se visitado[v] = false então
            BFS_VISIT(v, G, Q);
```

```
BFS_VISIT(s, G, Q)
    visitado[s] ← true;
    ENQUEUE(s, Q);
    Repita
        v ← DEQUEUE(Q);
        Para cada w ∈ G.Adjs[v] fazer
            Se visitado[w] = false então
                ENQUEUE(w, Q);
                visitado[w] ← true;
                pai[w] ← v;
    até (QUEUEISEMPTY(Q) = true );
```

Estamos a assumir que as variáveis $pai[\cdot]$ e $visitado[\cdot]$ são globais, embora as pudessemos ter considerado também como parâmetros das funções (o que pode fazer sentido em algumas aplicações).

Extensões ao pseudocódigo. Nos algoritmos em grafos, usaremos algumas extensões da linguagem de pseudocódigo introduzida no Capítulo 1. Por exemplo, se $G = (V, E)$, podemos usar $G.V$ e $G.E$ para referir o conjunto de vértices e de ramos de G, e $G.Adjs[v]$ para designar a lista de adjacentes de v (e não incluir a referência a G se for claro). Ciclos como "Para cada $v \in G.V$ fazer", "Para cada $(v, w) \in G.E$ fazer" e "Para cada $w \in G.Adjs[v]$ fazer" devem ser codificados de acordo com as estruturas de dados utilizadas. A menos que seja dito algo em contrário, assumimos que a estrutura de suporte dos grafos é baseada numa representação por listas de adjacências.

5.3.1 Determinar caminho com menor número de ramos de um vértice para outro

Na pesquisa a partir de s, por BFS(s,G), o valor de $pai[v]$ identifica o vértice que precede v no caminho que se encontrou de s para v, caso $s \neq v$ e v seja acessível de s. Implicitamente, $pai[\cdot]$ define uma árvore com raíz em s, que é **a árvore de pesquisa em largura a partir de** s. Os ramos da árvore são definidos pelos pares $(pai[v], v)$ tais que $pai[v] \neq$ `NULL`. O caminho de s até v nessa árvore é exemplo de um caminho mínimo de s até v no grafo (aqui, *mínimo* quer dizer que tem o menor número de ramos possível). A função ESCREVECAMINHO(v, pai) mostra como se pode utilizar $pai[\cdot]$ para obter o caminho de s para v que se encontrou (se $s \neq v$ e v for acessível de s). Seguindo uma abordagem recursiva, a função escreve primeiro o caminho de s até $pai[v]$, e a seguir acrescenta v.

Apresentamos a seguir uma adaptação de BFS_VISIT(s, G) para obter a distância mínima de s a cada nó v.

```
BFS_VISIT_ADAPTADO(s, G)
    Para cada v ∈ G.V fazer
        visitado[v] ← false;
        pai[v] ← NULL;
        dist[v] ← ∞;
    visitado[s] ← true;
    dist[s] ← 0;
    Q ← MKEMPTYQUEUE();
    ENQUEUE(s, Q);
    Repita
        v ← DEQUEUE(Q);
        Para cada w ∈ G.Adjs[v] fazer
            Se visitado[w] = false então
                dist[w] ← dist[v] + 1;
                ENQUEUE(w, Q);
                visitado[w] ← true;
                pai[w] ← v;
    até (QUEUEISEMPTY(Q) = true );
```

Observação. Em linguagem C ou Java, o ciclo *"Repita... até (condição)"* traduz-se por um ciclo `do...while`, mas a condição tem de ser negada. Como os caminhos (mínimos) têm no máximo $|V| - 1$ ramos, podemos definir ∞ como $|V|$, numa implementação deste algoritmo. Pode-se provar que o valor final de $dist[v]$ é o número de ramos do caminho mais curto de s para v, se v for acessível de s (c.f. Proposição 1, abaixo), ou $dist[v] = |V|$ e v não é acessível de s. Em vez de `NULL`, deve ser usado um valor adequado e que não possa ser confundido com o identificador de um

vértice (por exemplo, -1 se os vértices forem numerados a partir de 1 ou de 0).

Proposição 1 *Se v é acessível de s em G então $dist[v] = \delta(s, v)$, onde $\delta(s, v)$ é o número de ramos dos caminhos mais curtos s para v, qualquer que seja $v \neq s$.*

Prova: (por indução sobre a distância d) Se $\delta(s, v) = 1$ então $(s, v) \in E$ e $dist[v] = 1$, pois s visita todos os seus adjacentes. Suponhamos agora, como hipótese de indução, que para todo o u tal que $dist[u] < d$ se tem $dist[u] = \delta(s, u)$. Seja v tal que $dist[v] = d$. De todos os caminhos mínimos de s para v, tomemos aquele em que o vértice v' que precede imediatamente v no caminho foi o primeiro a ser visitado no algoritmo. Por definição de caminho mínimo, $\delta(s, v) = \delta(s, v') + 1$. Por outro lado, como $(v', v) \in E$, e v se encontra por visitar quando v' sai da fila, v' visita v, pelo que $dist[v] = dist[v'] + 1$ e portanto $dist[v'] < d$. Logo, $dist[v'] = \delta(s, v')$, pela hipótese, e $dist[v] = \delta(s, v') + 1 = \delta(s, v)$.

5.3.2 Determinar componentes conexas de grafos não dirigidos

Em BFS(G), o valor de $pai[v]$ identifica o primeiro nó que descobriu v durante a procura. No fim, o vetor $pai[.]$ define uma floresta de árvores pesquisa em largura. Por análise para trás a partir de v, podemos localizar a raíz da árvore de pesquisa em largura a que v pertence (podendo esta ser v se $pai[v] =$ `NULL`). Se G for um grafo dirigido, os vértices da árvore que contém v após a aplicação de BFS(G), poderão depender da ordem pela qual os vértices de G são explorados no segundo ciclo de BFS(G). Para grafos não dirigidos, tal não acontece.

Proposição 2 *Se G for um grafo não dirigido, cada árvore da floresta obtida por* BFS*(G) identifica uma componente conexa do grafo.*

Ideia da prova: Quando G é não dirigido, os vértices que constituem a árvore a que w pertence não dependem do nó raíz (embora a estrutura da árvore possa ser diferente os vértices são os mesmos). Na chamada de BFS(v, G, Q) no segundo ciclo de BFS(G), serão visitados todos os vértices acessíveis de v em G. Como o adjunto de G é simétrico, se algum w acessível de v tivesse sido visitado numa chamada anterior, então também v teria de ter sido marcado como visitado por algum dos descendentes de w.

Se G for conexo, a floresta reduz-se a uma árvore, à qual pertencem todos os vértices de G.

Complexidade

Em BFS_VISIT(s, G), cada nó entra para a fila quando muito uma vez, pois só pode entrar na fila se ainda não estiver marcado como visitado e é marcado como visitado quando é colocado na fila, sendo essa marcação persistente. A fila pode ser implementada de modo que MKEMPTYQUEUE(), QUEUEISEMPTY(Q) DEQUEUE(Q) e ENQUEUE(s,Q) tenham complexidade constante, i.e., $O(1)$. Logo, o primeiro ciclo tem complexidade $O(n)$ e o segundo $O(n + m)$,

se a estrutura de representação para o grafo for baseada em listas de adjacências, sendo $n = |V|$ e $m = |E|$. Portanto, BFS_VISIT(s, G) tem complexidade $O(n+m)$. Se G for não dirigido, cada ramo $\{u, v\}$ aparece na lista de adjacentes de u e de v, pelo que poderia ser usado duas vezes (uma vez quando cada um dos extremos sair da fila). Mas, $O(n+2m) = O(n+m)$. Observe-se que $\sum_{v \in V} |Adjs[v]| = 2m$, se o grafo for não dirigido e é $\sum_{v \in V} |Adjs[v]| = m$, se o grafo for dirigido.

Pela mesma razão, podemos concluir que BFS(G) tem complexidade temporal $\Theta(n+m)$. Neste caso, o tempo de execução é de ordem $\Omega(n + m)$ porque todos os ramos serão considerados, enquanto que, na pesquisa a partir de um nó, BFS_VISIT(s, G), pode haver uma parte do grafo que não é visitada. Basta por exemplo pensar no que aconteceria em BFS_VISIT(s, G) se o grafo fosse não dirigido e não conexo, ou mesmo, no que aconteceria se s fosse um vértice isolado, nas mesmas condições.

Por aplicação de BFS(G) podemos obter as **componentes conexas** de G (não dirigido) em tempo $\Theta(n + m)$. Basta recordar quais os vértices que pertencem a cada árvore, ou seja, quais os vértices que são visitados em cada chamada de BFS_VISIT(s, G, Q), no segundo ciclo de BFS(G).

Por aplicação de BFS_VISIT(G, s) podemos resolver o problema da determinação de caminhos mínimos (em termos de número de ramos) de s para qualquer outro nó, em tempo $O(n + m)$. A informação recolhida em $pai[\cdot]$ nesta fase de pré-processamento pode ser depois usada para obter em $\Theta(1 + \delta(s, v)))$ um caminho mínimo de s para v, sendo $\delta(s, v)$ a distância mínima de s a v, i.e., em tempo linear no comprimento do caminho. Observe-se que, $\Theta(1 + f(n)) = \Theta(f(n))$, qualquer que seja a função $f(n) > 0$. Por isso, podiamos ter escrito, $\Theta(\delta(s, v))$. Note-se também que $\delta(s, v) \in O(n)$, por definição de caminho.

5.4 Pesquisa em profundidade

À semelhança da pesquisa em largura, a pesquisa em profundidade (*depth-first search*) é uma estratégia genérica de análise exaustiva de um conjunto em que cada elemento tem um conjunto de elementos adjacentes (que pode ser vazio). Quando aplicada a partir de um nó origem s, começa por visitar um dos adjacentes de s, a seguir um adjacente desse adjacente de s que ainda não tenha sido visitado, e sucessivamente. Quando um nó já não tem mais adjacentes por visitar, a pesquisa prossegue com a análise de outro adjacente do *pai* desse nó (que ainda não tenha sido visitado).

Apresentamos a seguir o algoritmo DFS(G) para visita integral do grafo, na versão recursiva.

DFS(G)

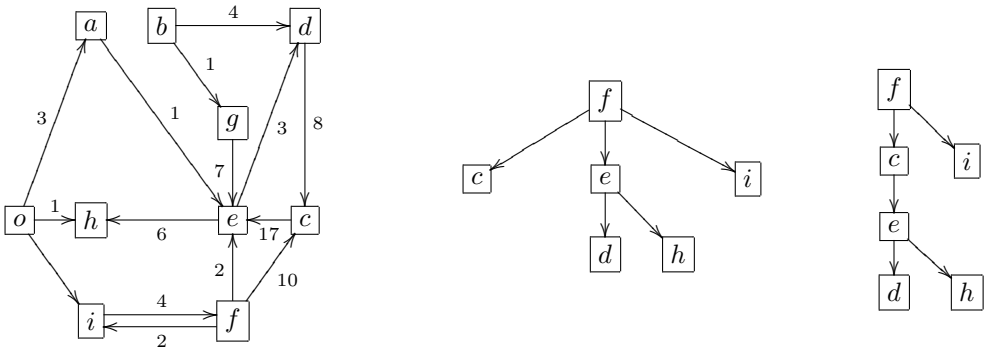
Para cada $v \in G.V$ fazer
  $visitado[v] \leftarrow$ `false`;
  $pai[v] \leftarrow$ `NULL`;
Para cada $v \in G.V$ fazer
  Se $visitado[v] =$ `false` então
    DFS_VISIT(v, G);

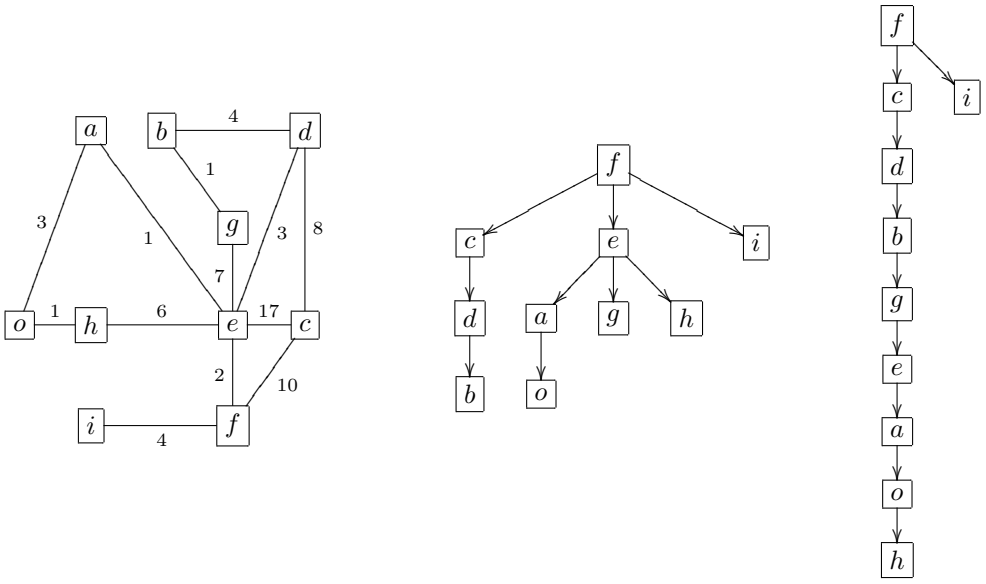
DFS_VISIT(v, G)

$visitado[v] \leftarrow$ `true`;
Para cada $w \in G.Adjs[v]$ fazer
  Se $visitado[w] =$ `false` então
    $pai[w] \leftarrow v$;
    DFS_VISIT(w, G);

Procura a partir de um nó origem. Para determinar apenas os nós acessíveis de um nó origem s, podemos adaptar DFS_VISIT para incluir a inicialização das variáveis $visitado[\cdot]$ e $pai[\cdot]$, como fizemos para BFS_VISIT.

Ilustramos a seguir a diferença entre as duas estratégias. Nos dois exemplos, supomos que a pesquisa é aplicada a partir do nó f e que os adjacentes de um nó estão ordenados por ordem alfabética nas listas de adjacências. Ao centro, vemos a árvore de pesquisa em largura e à direita a de pesquisa em profundidade, em cada caso.





Complexidade. A complexidade temporal de DFS(G) é $\Theta(n+m)$ e a de DFS_VISIT(s, G) é $O(n+m)$, ou seja, idêntica à da pesquisa em largura. A pesquisa em profundidade, DFS(G), pode ser usada para determinar as componentes conexas de um grafo não dirigido, em tempo $\Theta(n+m)$.

Quando as árvores de pesquisa em largura são bastante mais largas do que profundas, a pesquisa em profundidade pode constituir uma alternativa importante como técnica de procura, evitando gastos de memória consideráveis (para a manutenção da fila de BFS). É importante observar que, a implementação da versão recursiva de DFS requer a utilização da pilha do sistema, portanto, também tem um custo de memória (que é linear na altura da árvore de pesquisa). No segundo exemplo, a árvore de pesquisa em DFS a partir do nó f tem altura muito maior do que em BFS.

Instantes de descoberta e de finalização e deteção de ciclos

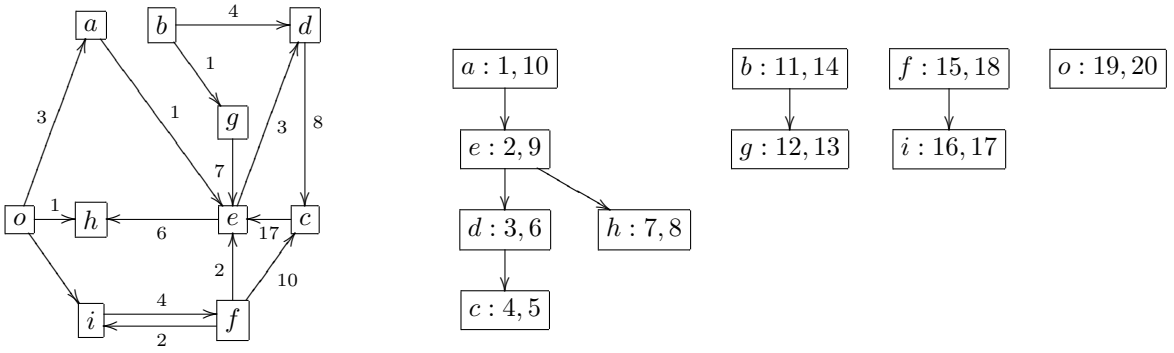
As árvores de pesquisa em profundidade têm propriedades interessantes que podem ser exploradas no desenvolvimento de algoritmos eficientes para resolução de alguns problemas. Para as perceber melhor, vamos apresentar uma nova versão dos algoritmos DFS(G) e DFS_VISIT(v, G), em que a variável $visitado$ é substituída por uma variável cor, que pode assumir três valores: (`branco`, se o nó ainda não tiver sido descoberto, `cinzento` se já estiver a decorrer a pesquisa a partir desse nó mas ainda não estiver concluída, `preto` se o nó já foi visitado bem como todos os seus descendentes). Numa implementação, estes valores não devem ser sequências de caracteres mas três inteiros (ou três `char`). Além disso, é introduzido um relógio, através da variável $instante$, que vai servir para marcar o instante de entrada ($t\_inicial$) e de saída (t_final) de cada nó. Abaixo, admitimos que essas variáveis são globais, para poderem ser modificadas nas várias chamadas das funções.

DFS(G)
- $instante \leftarrow 0$;
- Para cada $v \in G.V$ fazer
  - $cor[v] \leftarrow$ `branco`;
  - $pai[v] \leftarrow$ `NULL`;
- Para cada $v \in G.V$ fazer
  - Se $cor[v] =$ `branco` então
    - DFS_VISIT(v, G);

DFS_VISIT(v, G)
- $instante \leftarrow instante + 1$;
- $t\_inicial[v] \leftarrow instante$;
- $cor[v] \leftarrow$ `cinzento`;
- Para cada $w \in G.Adjs[v]$ fazer
  - Se $cor[w] =$ `branco` então
    - $pai[w] \leftarrow v$;
    - DFS_VISIT(w, G);
- $cor[v] \leftarrow$ `preto`;
- $instante \leftarrow instante + 1$;
- $t\_final[v] \leftarrow instante$;

Para cada nó v, o intervalo $[t\_inicial[v], t\_final[v]]$ contém os intervalos de todos os seus *descendentes* (i.e., dos nós que estão na *sub-árvore* que tem esse nó por raíz).

O exemplo seguinte mostra a floresta gerada pela chamada DFS(G) para o grafo indicado, bem como os valores finais dos tempos de entrada e de saída de cada nó. Supõe-se que os vértices são visitados por ordem alfabética.

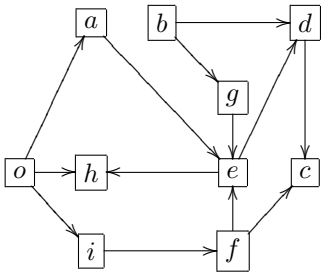


Quando o nó c é descoberto no instante 4, os nós a, e e d estão marcados a `cinzento`, o que quer dizer que c está nas sub-árvores que têm raíz nesses nós. Em particular, como e está marcado com `cinzento`, o no c é descendente do nó e na pesquisa. Isso quer dizer que existe um caminho de e para c no grafo, e esse caminho forma um ciclo com o ramo (c, e). A existência destes ciclos pode ser detetada se necessário (quando a cor de um adjacente não é `branco`, passa-se a verificar também se é `cinzento`).

5.5 Ordenação topológica de grafos dirigidos acíclicos (DAGs)

Seja $G = (V, E)$ um grafo dirigido acíclico (DAG) finito e $n = |V|$ o seu número de vértices. Chama-se **ordenação topológica** dos nós de G a uma função bijectiva σ de V em $\{1 \ldots, n\}$ tal que $\sigma(u) < \sigma(v)$ se $(u, v) \in E$, quaisquer que sejam $u, v \in V$. Esta função define o número de ordem de cada vértice nessa ordenação. O primeiro número

ordinal é 1, mas podiamos numerar os vértices a partir de 0, pois uma ordem topológica corresponde a uma permutação dos vértices que satisfaz a condição indicada se compararmos as posições que ocupam nessa permutação. Como a relação $<$ é transitiva, se $(u, v) \in E \wedge (v, w) \in E$ então $\sigma(u) < \sigma(v) < \sigma(w)$. Qualquer DAG admite alguma ordenação topológica e alguns DAGs admitem mesmo várias, como se vê no exemplo seguinte.



$o:1 \quad a:2 \quad b:3 \quad i:4 \quad g:5 \quad f:6 \quad e:7 \quad d:8 \quad c:9 \quad h:10$

$b:1 \quad o:2 \quad g:3 \quad a:4 \quad i:5 \quad f:6 \quad e:7 \quad d:8 \quad h:9 \quad c:10$

$b:1 \quad g:2 \quad o:3 \quad i:4 \quad a:5 \quad f:6 \quad e:7 \quad h:8 \quad d:9 \quad c:10$

Para obter uma tal ordenação, podemos começar por numerar um dos vértices que tem grau de entrada zero. A seguir, removemos os ramos que têm origem nesse vértice e aplicamos o mesmo processo ao DAG resultante (notar que, se retirarmos algum ramo a um DAG obtemos um DAG). Retirar um ramo (v, w) quando v é numerado corresponde a assinalar que a restrição $\sigma(v) < \sigma(w)$ se encontra satisfeita, pois w só será numerado depois de v. A existência de vértices com grau de entrada zero é garantida pelo resultado seguinte.

Proposição 3 *Qualquer DAG finito tem algum vértice com grau de entrada zero.*

Prova: Seja $G = (V, E)$ um DAG, com $|V| < \infty$. Vamos mostrar que se todos os vértices de V tivessem grau de entrada maior ou igual a 1, o grafo G teria algum ciclo, o que é um absurdo. É trivial que G não tem lacetes (pois lacetes são ciclos). Por isso, $(v, v) \notin V$ para todo $v \in V$. Tomemos agora um vértice $v_1 \in V$ qualquer. Como o grau de entrada de v_1 é maior ou igual a 1, então existe $v_2 \in V \setminus \{v_1\}$ tal que $(v_2, v_1) \in E$. Do mesmo modo, como v_2 tem grau de entrada maior ou igual a 1, existe $v_3 \in V \setminus \{v_2\}$ tal que $(v_3, v_1) \in E$. Se $v_3 = v_1$ então G teria um ciclo: (v_1, v_2, v_1). Se $v_3 \neq v_1$, então, por raciocínio análogo, podemos afirmar que existe $v_4 \in V \setminus \{v_3\}$ tal que $(v_4, v_3) \in E$ e $v_4 \neq v_1$, $v_4 \neq v_2$, e $v_4 \neq v_3$, e sucessivamente, $\ldots \boxed{v_{??}} \rightarrow \boxed{v_4} \rightarrow \boxed{v_3} \rightarrow \boxed{v_2} \rightarrow \boxed{v_1}$. Como V é finito, algum dos v_i's vai acabar por se repetir. A primeira repetição, mostra a existência do ciclo procurado.

Apresentamos abaixo o algoritmo em pseudocódigo. A remoção de ramos não é feita explicitamente, pois, normalmente, não pretendemos nem podemos destruir o grafo.

ORDENACAOTOPOLOGICA(G)

Para cada $v \in G.V$ fazer $GrauE[v] \leftarrow 0$;
Para cada $(v, w) \in G.E$ fazer $GrauE[w] \leftarrow GrauE[w] + 1$;
$S \leftarrow \{v \in G.V \mid GrauE[v] = 0\}$;
$i \leftarrow 0$;
Enquanto ($S \neq \emptyset$) fazer
    $v \leftarrow$ RETIRAUMELEMENTO(S);
    $i \leftarrow i + 1$;
    $sigma[v] \leftarrow i$;
    Para cada $w \in G.Adjs[v]$ fazer
        $GrauE[w] \leftarrow GrauE[w] - 1$;
        Se $GrauE[w] = 0$ então $S \leftarrow S \cup \{w\}$;

O vetor $GrauE$ é usado para determinar o grau de entrada inicial de cada vértice e, posteriormente, vai sendo decrementado à medida que os ramos vão sendo "retirados". Para suportar a variável S, deve ser usada uma pilha ou uma fila, embora tal não seja determinante para a correção do algoritmo, uma vez qualquer vértice em S poderia ser retirado de S na chamada de RETIRAUMELEMENTO(S). Neste algoritmo, a condição $S \neq \emptyset$ que controla a paragem do ciclo "Enquanto" deverá ser concretizada quando se fixar a estrutura de suporte de S, assim como a atribuição $S \leftarrow S \cup \{w\}$ e a inicial $S \leftarrow \{v \in G.V \mid GrauE[v] = 0\}$.

Complexidade. Assumindo que S é suportado por uma pilha (ou fila), e que as operações de criação, inserção e remoção de elementos são efetuadas em tempo $O(1)$, bem como o teste de que não está vazia, a complexidade temporal do algoritmo apresentado é $\Theta(n + m)$. Tal resulta de complexidade da determinação dos graus de entrada iniciais ser $\Theta(n + m)$ e a do o ciclo "Enquanto" ser $O(n + m)$, pois cada nó só entra e sai de S uma vez, quando sai percorre os seus adjacentes. Se o algoritmo for (inadvertidamente) aplicado a um grafo dirigido não acíclico, então S fica vazia antes de todos os nós serem numerados, ou seja, no fim do ciclo, $i < n$ (pela Proposição 3). Contudo, se o grafo for um DAG, o ciclo "Enquanto" tem complexidade $\Theta(n + m)$ e, no final do ciclo, $i = n$.

Podemos também desenvolver um algoritmo de ordenação topológica, com a mesma complexidade assintótica, mas que se baseia numa adaptação *do algoritmo de pesquisa em profundidade*. Apresentamos uma versão a seguir.

TOPSORT_DFS(G)

$S \leftarrow$ MKEMPTYSTACK($|G.V|$);
Para cada $v \in G.V$ fazer $cor[v] \leftarrow$ `branco`;
Para cada $v \in G.V$ fazer
    Se $cor[v] =$ `branco` então
        TOPSORT_DFSVISIT(v, G, S);
$i \leftarrow 1$;
Enquanto (STACKISEMPTY(S) $=$ `false`) fazer
    $sigma$[POP(S)] $\leftarrow i$;
    $i \leftarrow i + 1$;

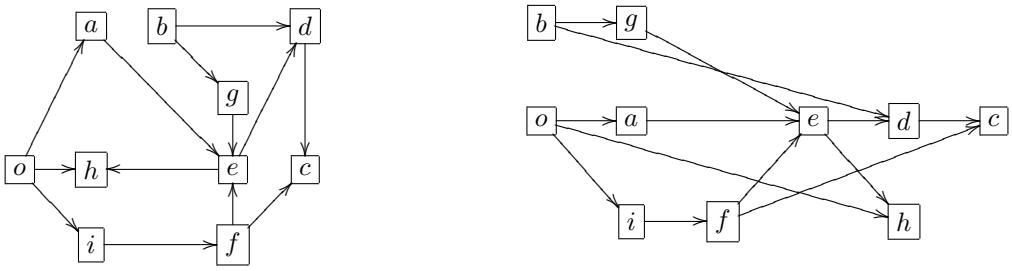
TOPSORT_DFSVISIT(v, G, S)

$cor[v] \leftarrow$ `cinzento`;
Para cada $w \in G.Adjs[v]$ fazer
    Se $cor[w] =$ `branco` então
        TOPSORT_DFSVISIT(w, G, S);
    senão se $cor[w] =$ `cinzento` então
        escrever (`"tem ciclos"`);
        parar com erro;
$cor[v] \leftarrow$ `preto`;
PUSH(v, S);

Observação. Se se souber que o grafo G não tem ciclos (i.e., que é mesmo um DAG), a marcação `cinzento` não é necessária e a instrução "senão se $cor[w] =$ `cinzento` então..." deve ser retirada. A instrução $cor[v] \leftarrow$ `cinzento` pode substituída por $cor[v] \leftarrow$ `preto`. Quando o grafo é um DAG, o algoritmo permanece correto se a alteração do estado de v (para cor `preto`, i.e., "visitado") for feita depois de ter visitado todos os seus adjacentes. É de salientar que, em muitas aplicações, **o vetor $sigma[\cdot]$ não é necessário**, e pode ser mais útil retornar a pilha S. A ordem pela qual os elementos sairão da pilha é uma ordenação topológica.

5.5.1 Caminho máximo num DAG

Em algumas aplicações práticas, um DAG serve de modelo à definição de uma relação de precedências (por exemplo, entre tarefas de um projeto). A figura mostra um DAG re-desenhado de modo a evidenciar as precedências que impõe.



Cenário 1. Suponhamos que $V = \{a, b, c, \ldots, g, h, i, o\}$ é um conjunto de unidades curriculares semestrais e que o grafo representado traduz precedências entre unidades curriculares: um ramo (x, y) no grafo indica que x terá de ser concluída antes de y, qualquer que seja esse ramo. Da análise do grafo podemos perceber que, no mínimo, se consegue realizar V em cinco semestres (algumas unidades terão de ser realizadas no mesmo semestre). Esse número é igual ao número de unidades curriculares no maior caminho existente no DAG.

Cenário 2. Suponhamos que os ramos do grafo um conjunto de tarefas $T = \{oa, oh, oi, ae, bg, bd, dc, ed, eh, \ldots\}$ que constituem um projeto e que a relação de precedência resulta de se impor que as tarefas que têm fim num nó têm de estar todas concluídas antes de iniciar tarefas com início nesse nó. Se cada uma das tarefas tiver duração 1, vemos que é possível concluir T em quatro unidades de tempo (algumas tarefas decorrerão em simultâneo). Esse número é igual ao número de ramos do maior caminho existente no DAG.

Apresentamos a seguir um algoritmo para determinar um caminho de comprimento máximo num grafo dirigido acíclico $G = (V, E)$, sendo o comprimento dado pelo número de ramos do caminho.

CAMINHOMAXIMODAG(G)
Para cada $v \in G.V$ fazer $\{ES[v] \leftarrow 0; pai[v] \leftarrow$ NULL; $GrauE[v] \leftarrow 0;\}$
Para cada $(v, w) \in G.E$ fazer $GrauE[w] \leftarrow GrauE[w] + 1$;
$S \leftarrow \{v \in G.V \mid GrauE[v] = 0\}$; /* S deve ser suportado por uma fila ou uma pilha. */
$max \leftarrow -1$; $v_f \leftarrow$ NULL;
Enquanto ($S \neq \emptyset$) fazer
    $v \leftarrow$ RETIRAUMELEMENTO(S);
    Se $max < ES[v]$ então $\{max \leftarrow ES[v]; v_f \leftarrow v;\}$
    Para cada $w \in G.Adjs[v]$ fazer
        Se $ES[w] < ES[v] + 1$ então
            $\{ES[w] \leftarrow ES[v] + 1; pai[w] \leftarrow v;\}$
        $GrauE[w] \leftarrow GrauE[w] - 1$;
        Se $GrauE[w] = 0$ então $S \leftarrow S \cup \{w\}$;
ESCREVECAMINHO(v_f, pai); escrever(max);

O algoritmo tem complexidade $\Theta(n + m)$, com $n = |V|$ e $m = |E|$. O valor final de $ES[w]$ representa o comprimento do caminho máximo encontrado até w e $pai[w]$ o nó que antecede w nesse caminho. Usamos a sigla ES (iniciais de *earliest start*) em vez de $dist$. Inicialmente $ES[w]$ é zero porque se não for encontrado nenhum caminho então a diatância máxima será zero. No cenário 2, isso queria dizer que as tarefas com início em w poderiam começar no instante 0. À medida que as precedências vão sendo tratadas, o valor da estimativa $ES[w]$ poderá crescer, a menos que grau de entrada de w fique zero. Nesse caso, a estimativa da distância é a distância máxima pois nenhum outro vértice poderá empurrar $ES[w]$ para a frente. O valor de max é o comprimento do caminho máximo de G (ou de um caminho máximo se houver vários) e v_f o vértice final nesse caminho.

Distâncias não unitárias. No Cenário 2, assumimos que as tarefas tinham durações idênticas e iguais a 1, o que não é verdade em muitas aplicações. Contudo, se $d(v, w) \in \mathbb{R}^+$ fosse a duração da tarefa (v, w), bastava substituir

$ES[v] + 1$ por $ES[v] + d(v, w)$, no ciclo "Para", para determinar o comprimento dos caminhos máximos no novo cenário. A expressão $ES[v] + d(v, w)$ representa o comprimento máximo dos caminhos que chegam a w passando por v e que depois de v só têm mais um ramo (que será necessariamente (v, w)). O algoritmo que se obtém está na base do *método do caminho crítico* (CPM, em Inglês, *critical path method*), o qual tem aplicações importantes na resolução de problemas de calendarização de tarefas, horários, etc (designados por *scheduling problems*). Os caminhos máximos, i.e., com comprimento igual a max, designam-se por *caminhos critícos*. Se atrasarmos uma tarefa que pertence a um caminho critíco, atrasamos o projeto (deixando de o poder concluir até max). Por vezes, estes problemas são bem mais complexos pois é necessário atender também a novas restrições decorrentes da partilha de recursos como, máquinas, salas, e trabalhadores. Por exemplo, num horário de um curso, não pode haver sobreposição dos horários de certas unidades curriculares (mas, à partida, uma permutação do seus horários seria possível).

Caminho máximo em grafos dirigidos não acíclicos

Como vimos, a determinação do comprimento do caminho máximo num DAG pode ser efetuada em tempo $O(n+m)$, ou seja, linear no tamanho da instância. Embora não seja relevante em DAGs, é importante realçar que pretendemos um caminho máximo (acíclico, por definição de caminho) e não um percurso máximo (o que não teria significado se o grafo tivesse algum ciclo). Para grafos com ciclos, **não são conhecidos algoritmos polinomiais para resolver o problema e é possível que não existam**. O *problema é NP-hard*. Se alguém conseguir encontrar um algoritmo polinomial que o resolva então terá uma demonstração para um dos problemas mais famosos ainda em aberto em Ciência de Computadores, o de saber se *as classes de complexidade P e NP* são iguais. Esta matéria será tratada mais à frente (altura em que serão introduzidos os conceitos aqui em itálico).

5.5.2 Determinar componentes fortemente conexas em grafos dirigidos

Recordemos que uma **componente fortemente conexa** de um grafo dirigido $G = (V, E)$ é um subgrafo de G em que qualquer vértice é acessível de qualquer outro vértice e que é máximo, no sentido de não se poder acrescentar outros vértices de V sem quebrar essa propriedade. Apresentamos abaixo, em linhas gerais, o algoritmo de Kosaraju-Sharir (1978) para determinar as componentes fortemente conexas de G. Existem outros algoritmos mas descrevemos este por ser bastante simples e ter complexidade assintótica ótima (no pior caso). Na descrição, G^T designa o **grafo transposto** de $G = (V, E)$, o qual resulta de G por inversão da orientação dos ramos, ou seja, $G^T = (V, E^T)$, com $E^T = \{(v, u) \mid (u, v) \in E\}$. Na representação matricial, a matriz de G^T é a transposta da matriz de G.

COMPONENTESFORTEMENTECONEXAS(G)

1. Aplicar DFS, colocando os vértices por ordem decrescente de $t\_final[\cdot]$ numa pilha S;
2. Para $v \in V$ fazer $cor[v] \leftarrow$ `branco`;
3. Enquanto ($S \neq \{\ \}$) fazer
4. $\quad v \leftarrow$ POP(S);
5. $\quad$ Se $cor[v] =$ `branco` então DFS_VISIT(v, G^T) e indicar os vértices visitados nesta descida;

Para que o algoritmo de Kosaraju-Sharir tenha complexidade temporal $\Theta(n+m)$, o passo 1 tem de ser realizado em $O(m+n)$. Por isso, não pode começar por determinar $t\_final[\cdot]$ e só depois ordenar os vértices. O método a aplicar será semelhante ao utilizado em TOPSORT_DFS(G): não se constrói o vetor de tempos de finalização; cada vértice é colocado em S quando a pesquisa a partir desse vértice termina. Os vértices ficam naturalmente colocados em S por ordem decrescente de tempos finais, como se pretende.

Para a prova de correção do algoritmo são necessárias algumas propriedades do conjunto das componentes fortemente conexas G e da estrutura do **grafo das componentes fortemente conexas**. Este grafo tem um conjunto de vértices que corresponde às componentes conexas de G e os ramos são os pares (C, C') tais que no grafo G existe algum ramo de algum vértice de C para algum vértice de C', com $C \neq C'$ componentes quaisquer.

As propriedades enunciadas nas três proposições seguintes são exploradas no algoritmo de Kosaraju-Sharir.

Proposição 4 *O grafo das componentes fortemente conexas de um grafo dirigido é um grafo dirigido acíclico.*

Ideia da prova: (por redução ao absurdo) Se não fosse acíclico, então conteria algum ciclo. Usando esse ciclo e, possivelmente, percursos internos nas componentes por onde passa, conseguiamos mostrar que todos os vértices das componentes visitadas são acessíveis uns dos outros. Isto é um absurdo pois, se assim fosse, esses vértices teriam de pertencer à mesma componente e não a componentes distintas. Portanto, o grafo é acíclico.

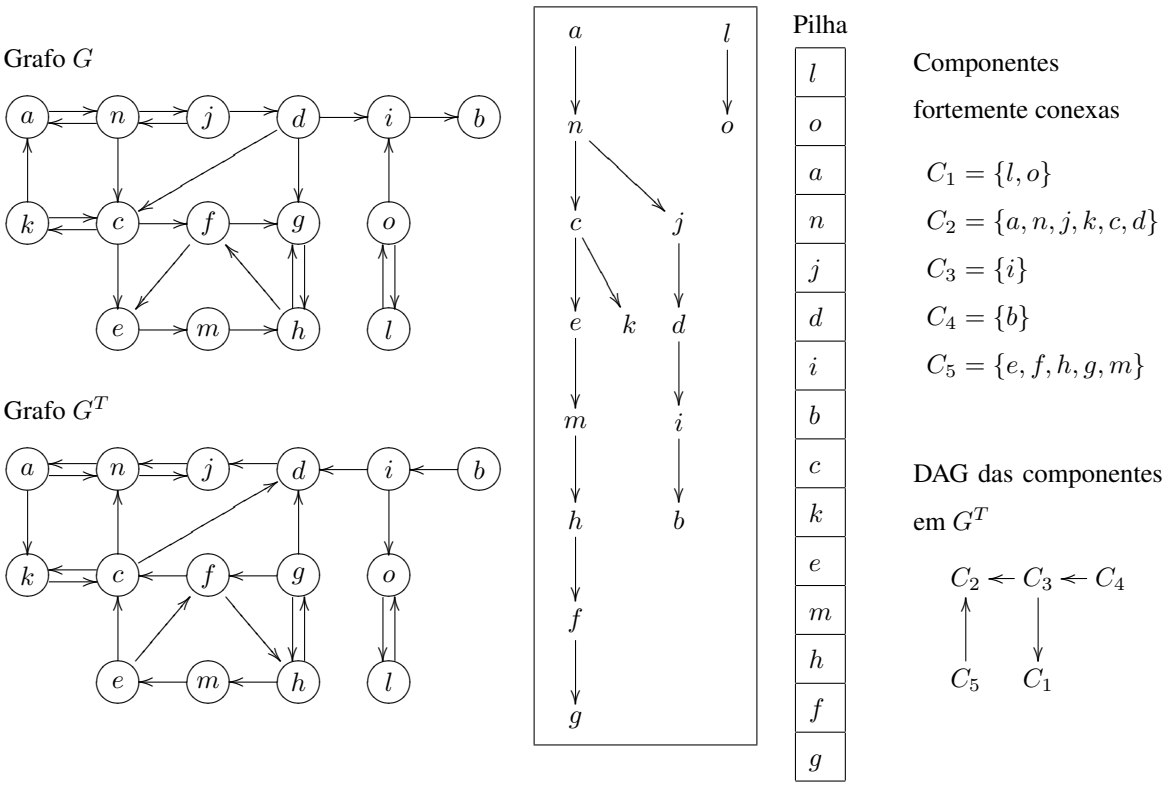
Proposição 5 *As componentes fortemente conexas de G e G^T são caraterizadas pelos mesmos vértices.*

Ideia da prova: Se x e y pertencerem à mesma componente fortemente conexa de G, então x é acessível de y e y é acessível de x em G. Podemos concluir que x e y seriam também mutuamente acessíveis em G^T pois qualquer percurso $\gamma = (u, w_1, w_2, \ldots, w_k, v)$, com $k \geq 0$, de u para v em G corresponde a um perurso $\gamma^T = (v, w_k, \ldots, w_2, w_1, u)$ de v para u em G^T, e vice-versa.

Proposição 6 *Uma ordenação topológica das componentes de G corresponde a uma ordenação topológica por ordem inversa (da cronológica) para as componentes de G^T.*

Ideia da prova: Um ramo (C, C') no DAG definido pelas componentes fortemente conexas de G corresponde a um ramo (C', C) no DAG definido pelas componentes fortemente conexas de G^T, e vice-versa.

Estes resultados fundamentam a correção do algoritmo de Kosaraju-Sharir. Como o grafo das componentes fortemente conexas é acíclico, pode ser ordenado topologicamente. A ordenação dos vértices por ordem decrescente de tempos de finalização (i.e, a ordem em que sairão da pilha S) determina uma ordenação topológica do DAG das componentes fortemente conexas de G, a que corresponde uma ordenação do DAG das componentes de G^T por ordem topológica (cronológica) **inversa**. Quando na pesquisa em profundidade em G^T retira v com cor `branco` de S, os únicos vértices **ainda não visitados** a que pode aceder a partir de v são todos e apenas os da componente de v em G^T (que são os mesmos que definem a sua componente em G).



5.6 Árvores geradoras de peso ótimo para um grafo conexo

Exemplo 4 *Uma companhia de distribuição de gás natural pretende construir uma rede que assegure a distribuição a um certo número de locais a partir de um dado local. Dados os custos da ligação entre cada par de locais, há que determinar as ligações a efetuar de modo a reduzir os custos globais.*

As árvores são os grafos dirigidos conexos com menos ramos. O problema enunciado corresponde à determinação de uma árvore geradora mínima num grafo $G = (V, E, d)$ não dirigido, finito e **conexo** com valores associados aos ramos, em que $d : E \to \mathbb{R}_0^+$ indica o valor associado a cada ramo. Uma **árvore geradora de** G (ou de suporte de G) é um qualquer subgrafo de G que é uma árvore e tem V como conjunto de vértices. Qualquer árvore geradora de G cuja

soma dos valores associados aos ramos seja mínima, quando consideradas todas as possíveis árvores geradoras de G, designa-se por **árvore geradora de G com peso mínimo** ou árvore geradora mínima (em Inglês, *minimum (weight) spanning tree*). De modo análogo, podiamos definir **árvore geradora de G com peso máximo**.

Para determinar uma árvore geradora mínima em tempo polinomial, podemos aplicar, por exemplo, os algoritmos de Prim (1957) e de Kruskal (1956), que apresentamos a seguir.

Os algoritmos de Prim e Kruskal seguem uma **estratégia gulosa** ("greedy"), também designada por ávida ou gananciosa. Em cada iteração é escolhida a alternativa que **localmente** se apresenta como a melhor e não haverá qualquer retrocesso para analisar outras possibilidades. Neste problema, essa estratégia permite determinar uma solução que é também (globalmente) ótima. Noutros problemas como, por exemplo, *no problema da mochila* (booleano ou inteiro), não há garantia de que a solução obtida por uma estratégia *gulosa* seja ótima. Mais à frente voltaremos a este assunto.

5.6.1 Algoritmo de Prim

No algoritmo de Prim, a árvore é construída a partir de um vértice qualquer e, em cada iteração, escolhe um dos vértices que está "mais próximo" dos vértices que na sub-árvore já construída incluídos e liga esse vértice à árvore. Aqui, a "proximidade" corresponde a uma interpretação dos valores (ou pesos) nos ramos como distâncias. O vértice a ligar será um dos que possa ser ligado por um ramo de peso menor. O processo termina quando a árvore tiver os $|V|$ vértices. Em **linhas gerais**, o algoritmo de Prim, quando aplicado a partir de um nó s, pode ser traduzido assim:

Para cada $v \in V$ fazer { $pai[v] \leftarrow$ `NULL`; $dist[u] \leftarrow \infty$;}
$dist[s] \leftarrow 0$;
$Q \leftarrow V$;
$T \leftarrow \{\}$;
Enquanto $Q \neq \emptyset$ fazer
    Escolher $v \in Q$ tal que $dist[v] = min_{u \in Q}\ dist[u]$;
    $Q \leftarrow Q \setminus \{v\}$;
    $T \leftarrow T \cup \{(pai[v], v)\}$;
    Para cada (v, w) tal que $w \in Adjs[v]$ fazer
        Se $d(v, w) < dist[w]$ então
            $dist[w] \leftarrow d(v, w)$;
            $pai[w] \leftarrow v$;

A complexidade do algoritmo de Prim depende do tipo de estrutura de dados que se utiliza para suportar Q, que deverá ser uma *fila de prioridade*. Se essa fila corresponder a uma *heap de mínimo* então, como veremos mais adiante, tanto a operação de remoção do elemento que tem *chave* mínima como a operação de redução da *chave* de um elemento

(e, consequente, promoção desse elemento em Q por $dist$ ter diminuído) podem ser implementadas em $O(\log_2 n)$, sendo n o número de elementos da fila. A fila inicial com os $|V|$ vértices pode ser construída em $\Theta(|V|)$.

ALGORITMOPRIM(G, s)

1.	Para cada $v \in V$ fazer $\{$ $pai[v] \leftarrow$ `NULL`; $dist[u] \leftarrow \infty$;$\}$	$\Theta(\|V\|)$
2.	$dist[s] \leftarrow 0$;	$O(1)$
3.	$Q \leftarrow$ MK_PQ_HEAPMIN$(dist, V)$;	$\Theta(\|V\|)$
4.	$T \leftarrow \{\}$;	$O(1)$
5.	Enquanto (PQ_NOT_EMPTY(Q)) fazer	
6.	$\quad v \leftarrow$ EXTRACTMIN(Q);	$O(\log_2 \|V\|)$
7.	$\quad T \leftarrow T \cup \{(pai[v], v)\}$;	$O(1)$
8.	$\quad$ Para cada $w \in Adjs[v]$ fazer	
9.	$\quad\quad$ Se $d(v, w) < dist[w]$ então	$O(1)$
10.	$\quad\quad\quad dist[w] \leftarrow d(v, w)$;	$O(1)$
11.	$\quad\quad\quad pai[w] \leftarrow v$;	$O(1)$
12.	$\quad\quad\quad$ DECREASEKEY$(Q, w, dist[w])$;	$O(\log_2 \|V\|)$

O algoritmo de Prim, suportado por uma **heap de mínimo**, tem complexidade $O(|E| \log |V|)$. Como $|E| \geq |V|-1$ por G ser conexo, a complexidade do algoritmo é dominada pela complexidade do ciclo "Enquanto", que é dada por:

$$\begin{aligned} O(\sum_{v \in V}(1 + \log_2 |V| + |Adjs[v]| \log_2 |V|)) &= O(|V| \log_2 |V| + |E| \log_2 |V|) \\ &= O((|V| + |E|) \log_2 |V|) = O(|E| \log_2 |V|). \end{aligned}$$

5.6.2 Algoritmo de Kruskal

O algoritmo de Kruskal parte de uma floresta em que cada vértice de V está isolado numa árvore sem ramos. Em cada iteração, seleciona um ramo para ligar duas árvores, e, consequentemente, quando tiver inserido $|V| - 1$ ramos, a floresta estará reduzida a uma árvore única. Os ramos são considerados por ordem crescente de valor de d, e só não farão parte da árvore de suporte se o grafo resultante da sua junção à floresta construída até esse passo for cíclico. A junção do ramo não formará um ciclo se os seus extremos pertencerem a árvores distintas na floresta. Em **linhas gerais**, o algoritmo de Kruskal consiste no seguinte.

Ordenar E por ordem crescente de valores nos ramos.
$T \leftarrow \emptyset$; $\mathcal{C} \leftarrow \{\{v\} \mid v \in V\}$;
Enquanto ($|T| \neq |V| - 1$) fazer
    Seja $\langle u, v\rangle \in E$ o primeiro ramo não escolhido (na ordem considerada).
    Sejam C_u e C_v os elementos de $\mathcal{C}$ tais que $u \in C_u$ e $v \in C_v$.
    Se $C_u \neq C_v$ então
        $T \leftarrow T \cup \{\langle u, v\rangle\}$;
        $\mathcal{C} \leftarrow (\mathcal{C} \setminus \{C_u, C_v\}) \cup \{C_u \cup C_v\}$;

O conjunto de ramos de G pode ser ordenado em tempo $O(|E| \log_2 |E|) = O(|E| \log_2 |V|)$, porque $|E| \leq |V|(|V| - 1)/2 < |V|^2$, o que implica que $\log_2 |E| < 2 \log_2 |V|$, pois $\log_B x^p = p \log_B x$, para todo $x \in \mathbb{R}^+$, qualquer que seja a base B, e a função $\log_2(\cdot)$ é crescente.

A complexidade do algoritmo de Kruskal depende da estrutura de dados que será usada para representar a floresta, definida como uma partição de V. Por definição, uma **partição $\mathcal{C}$ de um conjunto** V é uma família de subconjuntos de V que são não vazios, disjuntos dois a dois e que, conjuntamente, cobrem V. O tipo abstrato de dados usado para representar a partição deverá suportar, de forma eficiente, a localização do conjunto a que pertence um elemento dado e união de dois conjuntos (disjuntos). Seria trivial implementar cada uma dessas operações em $O(n)$, sendo $n = |V|$, mas é possível implementar também em $O(\log_2 n)$, no pior caso, e é isso que iremos assumir. Nesse caso, a complexidade do algoritmo de Kruskal é $O(|E| \log_2 |V|)$ se o método usado para efetuar a ordenação dos ramos (no passo 1.) tiver essa complexidade.

AlgoritmoKruskal(G)
1. $Q \leftarrow$ Fila que representa E por ordem crescente de valores nos ramos;
2. $T \leftarrow \emptyset$;
3. $\mathcal{C} \leftarrow$ Init_Singletons(V);
4. Enquanto ($|T| \neq |V| - 1 \wedge$ QueueIsEmpty(Q) = `false`) fazer
5. $\langle u, v\rangle \leftarrow$ Dequeue(Q);
6. Se FindSet($u, \mathcal{C}$) $\neq$ FindSet($v, \mathcal{C}$) então
7. $T \leftarrow T \cup \{\langle u, v\rangle\}$;
8. Union($u, v, \mathcal{C}$);

A condição QueueIsEmpty(Q) = `false` é redundante se o grafo for conexo. Contudo, permite que o algoritmo possa ser aplicado a um grafo não conexo para obter a floresta de árvores geradoras mínimas das suas componentes conexas.

5.6.3 Correção dos algoritmos de Prim e de Kruskal

A correção dos algoritmos descritos resulta da propriedade seguinte.

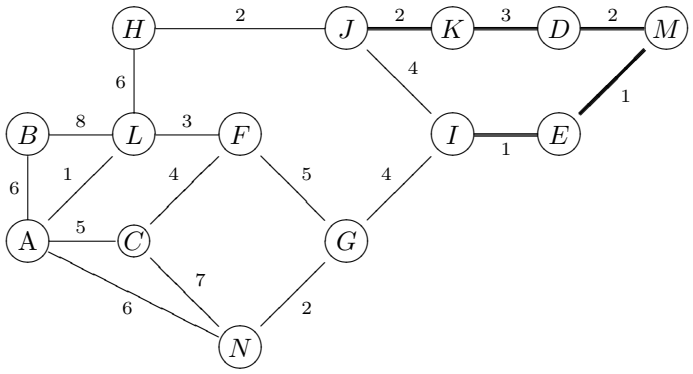
Proposição 7 *Seja T uma árvore geradora mínima de um grafo $G = (V, E, d)$ não dirigido e conexo. Qualquer que seja a partição $\{V_1, V_2\}$ do conjunto de vértices V, a árvore T tem algum ramo $\langle v_1, v_2\rangle$ com $v_1 \in V_1$ e $v_2 \in V_2$ e tal que $d(v_1, v_2) = \min\{d(x, y) \mid x \in V_1, y \in V_2, \langle x, y\rangle \in E\}$.*

Prova: (por redução ao absurdo) Seja T uma árvore geradora mínima de G e suponhamos que $\{V_1, V_2\}$ é uma partição de V tal que T não contém nenhum dos ramos $\langle v_1, v_2\rangle$ com $v_1 \in V_1$, $v_2 \in V_2$ e $d(v_1, v_2) = \min\{d(x, y) \mid x \in V_1, y \in V_2, \langle x, y\rangle \in E\}$. Seja $\langle v_1, v_2\rangle$ um desses ramos. Como T é uma árvore geradora de G, existe um e um só caminho entre v_1 e v_2 em T. Esse caminho tem que ter algum ramo $\langle x, y\rangle$ com $x \in V_1$ e $y \in V_2$, pois, caso contrário, os vértices em V_1 (respectivamente, em V_2) só estariam ligados em T a vértices em V_1 (respectivamente, em V_2), e a árvore T não seria conexa (o que é absurdo). É possível que ou $x = v_1$ ou $y = v_2$. Pela hipótese inicial, $d(x, y) > d(v_1, v_2)$. Por outro lado, se substituirmos $\langle x, y\rangle$ em T por $\langle v_1, v_2\rangle$, o grafo resultante ainda é uma árvore geradora de G e tem "peso" menor do que a árvore T, o que contradiz o facto de T ser mínima. Portanto, a árvore T tem de ter algum dos ramos de menor peso nesse corte (por definição, o **corte** determinado pela partição $\{V_1, V_2\}$ de V é o conjunto de ramos que ligam vértices de V_1 a vértices de V_2).

No algoritmo de Prim, esta propriedade das árvores geradoras mínimas garante que é *seguro* ligar o vértice v à sub-árvore já construída. Em cada iteração do ciclo "Enquanto", V_1 seria o conjunto dos vértices que já estão na sub-árvore e V_2 seria o conjunto dos restantes (onde v se apresenta como melhor candidato, ou um dos melhores). Para cada $v \in V_2$, o valor de $dist[v]$ é o custo dos ramos mais leves com extremidade em v e que estão no corte definido por $\{V_1, V_2\}$. Este invariante é preservado pelo ciclo.

No algoritmo de Kruskal, quando $\langle u, v\rangle$ pode ser usado para ligar duas componentes, então se tomarmos V_1 como os vértices da árvore a que pertence u e V_2 os restantes vértices, podemos concluir que $\langle u, v\rangle$ é seguro. Do resultado que demonstrámos, deduzimos que alguma árvore geradora mínima contém $\langle u, v\rangle$, pois este ramo tem peso mínimo no corte definido por $\{V_1, V_2\}$.

Para o grafo seguinte, o resultado enunciado na Proposição 7 permite-nos concluir que o ramo $\langle I, J\rangle$ não pertence a **nenhuma** árvore geradora mínima $\mathcal{T}$ de G.



Aplicando a Proposição 7, deduzimos que os ramos $\langle K, J\rangle$, $\langle D, K\rangle$, $\langle M, D\rangle$, $\langle E, I\rangle$ e $\langle M, E\rangle$ pertencem a todas as árvores geradoras mínimas $\mathcal{T}$ do grafo. Por isso, se $\langle J, I\rangle \in \mathcal{T}$ então $\mathcal{T}$ não era uma árvore já que teria um ciclo.

$V_1 = \{K, D, M, I, E\}$ e $V_2 = V \setminus V_1$	$\langle K, J\rangle \in \mathcal{T}$
$V_1 = \{D, M, I, E\}$ e $V_2 = V \setminus V_1$	$\langle D, K\rangle \in \mathcal{T}$
$V_1 = \{M, E, I, G, N\}$ e $V_2 = V \setminus V_1$	$\langle M, D\rangle \in \mathcal{T}$

$V_1 = \{D, M, E\}$ e $V_2 = V \setminus V_1$	$\langle E, I\rangle \in \mathcal{T}$
$V_1 = \{M\}$ e $V_2 = V \setminus V_1$	$\langle M, E\rangle \in \mathcal{T}$

Subestrutura ótima. As árvores geradoras mínimas têm **subestrutura ótima**. Se retirarmos um ramo a uma árvore $\mathcal{T}$, as duas árvores $\mathcal{T}_1$ e $\mathcal{T}_2$ em que se decompõe são árvores geradoras mínimas dos subgrafos G_1 e G_2 de G definidos pelos conjuntos de vértices de $\mathcal{T}_1$ e $\mathcal{T}_2$. Se, por exemplo, $\mathcal{T}_1'$ fosse uma árvore geradora mínima de G_1 com peso inferior a $\mathcal{T}_1$, então podiamos podar $\mathcal{T}$ para retirar $\mathcal{T}_1$ e depois enxertar $\mathcal{T}_1'$, e a árvore resultante seria uma árvore geradora G de peso menor do que o peso de $\mathcal{T}$ (o que seria absurdo se $\mathcal{T}$ fosse mínima). O que se disse para peso mínimo é válido para peso máximo.

Árvores geradoras de peso máximo. Os algoritmos de Prim e de Kruskal podem ser adaptados para determinar uma árvore geradora com peso máximo. No algoritmo de Prim, os valores de $dist[v]$ seriam inicializados com $-\infty$, para todo $v \neq s$ e $dist[s]$ com 0, em que s designa o nó origem (ou com $dist[s] = \infty$ e $dist[v] = 0$, para todo $v \neq s$). A fila de prioridades será suportada por uma *heap de máximo*, com operações EXTRACTMAX e INCREASEKEY. De modo análogo, no algoritmo de Kruskal, os ramos seriam ordenados por ordem decrescente de peso (i.e., valor associado). Um resultado análogo ao enunciado na Proposição 7 é válido para $\max$ também.

Proposição 8 *Seja T uma árvore geradora de peso máximo de um grafo $G = (V, E, d)$ não dirigido e conexo. Qualquer que seja a partição $\{V_1, V_2\}$ do conjunto de vértices V, a árvore T tem algum ramo $\langle v_1, v_2\rangle$ com $v_1 \in V_1$ e $v_2 \in V_2$ e tal que $d(v_1, v_2) = \max\{d(x, y) \mid x \in V_1, y \in V_2, \langle x, y\rangle \in E\}$.*

Observação. Lemos $-\infty$ e ∞ como "indeterminado" ou "indefinido". Definimos $\min(x, \infty) = \min(\infty, x) = x$, para todo $x \in \mathbb{R}_0^+ \cup \{\infty\}$. E, $\max(x, -\infty) = \max(-\infty, x) = x$, para todo $x \in \mathbb{R}_0^+ \cup \{-\infty\}$.

5.7 Caminhos mínimos em grafos com pesos

Exemplo 5 Pretendemos determinar o tempo mínimo necessário para chegar de comboio de um certo local a um outro. Estão disponíveis os horários dos comboios portugueses. Admitimos que os comboios circulam sem atraso e que é possível mudar de um comboio para outro em 10 minutos.

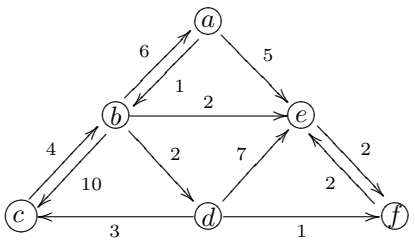
Com algum cuidado, este problema traduz-se por um problema de determinação de um **caminho mínimo** num grafo (ou num multigrafo) dirigido $G = (V, E, d)$, finito e com pesos (*distâncias* ou *valores*) positivos associados aos ramos, $d(u, v) > 0$, para todo $(u, v) \in E$. Neste contexto, a **distância associada a um percurso de** u **para** v é a soma das distâncias associadas aos ramos que constituem o percurso. Supomos também que a distância mínima de qualquer nó do grafo a si mesmo é zero. Dependendo da aplicação, podemos querer encontrar:

- um caminho mínimo de s para t, para um par $(s, t) \in V \times V$, com $s \neq t$;
- um caminho mínimo de s para cada um dos outros vértices do grafo, dado $s \in V$;
- um caminho mínimo de s para t, para todos os pares $(s, t) \in V \times V$, com $s \neq t$.

Em alguns casos, pode ser útil representar o grafo por uma **matriz de distâncias** D do modo seguinte:

$$\begin{aligned} D_{ij} &= d(v_i, v_j) \quad \text{sse } (v_i, v_j) \in E \\ D_{ij} &= \infty \qquad\quad \text{sse } (v_i, v_j) \notin E \end{aligned}$$

para $i, j \in \{1, \ldots, |V|\}$. Os elementos da matriz D são elementos de $\mathbb{R}_0^+ \cup \{\infty\}$. Quando o ramo (v_i, v_j) não existe, definimos D_{ij} por ∞, lendo ∞ como indeterminado ou indefinido. No exemplo, $v_1 = a, v_2 = b, \ldots, v_6 = f$, e seguimos a convenção de que a distância de um vértice a si próprio é zero, definindo $D_{ii} = 0$, para todo i.



$$\begin{bmatrix} 0 & 1 & \infty & \infty & 5 & \infty \\ 6 & 0 & 10 & 2 & 2 & \infty \\ \infty & 4 & 0 & \infty & \infty & \infty \\ \infty & \infty & 3 & 0 & 7 & 1 \\ \infty & \infty & \infty & \infty & 0 & 2 \\ \infty & \infty & \infty & \infty & 2 & 0 \end{bmatrix}$$

Para efetuar operações com ∞, definimos uma **operação binária** $\oplus$ em $\mathbb{R}_0^+ \cup \{\infty\}$, prolongamento da operação usual de adição em $\mathbb{R}_0^+$, do modo seguinte.

$$\begin{aligned} x \oplus y &= x + y \quad \text{se } x \in \mathbb{R}_0^+ \text{ e } y \in \mathbb{R}_0^+ \\ x \oplus y &= \infty \qquad \text{se } x = \infty \text{ ou } y = \infty \end{aligned}$$

Se $u \neq v$ e não existir percurso de u para v, a distância de u para v é ∞. Do mesmo modo, consideramos que $\min(x, \infty) = \min(\infty, x) = x$ e $\max(x, \infty) = \max(\infty, x) = \infty$, para todo $x \in \mathbb{R}_0^+ \cup \{\infty\}$.

Observação. Uma operação binária num conjunto A é uma função de $A \times A$ em A. Não a confundamos com uma relação binária num conjunto A, que corresponde a um subconjunto de $A \times A$. A definição de $\oplus$ está matematicamente correta e deve ser implementada de modo análogo num programa que use ∞ com a mesma semântica. A tradução de $dist[a] \oplus d(a, b)$, por exemplo, não pode ser $dist[a] + d(a, b)$. Numa implementação, **não devemos efetuar adições com** ∞ se definirmos ∞ como o maior inteiro que uma variável do tipo utilizado, por exemplo, do tipo `int`, pode guardar. Tal resultaria num erro de *overflow* que conduziria a respostas erradas. Na descrição dos algoritmos **vamos utilizar apenas o símbolo** $+$, devendo ficar claro, no contexto, se se trata de $\oplus$.

5.7.1 Algoritmo de Dijkstra

O algoritmo de Dijkstra (1959) pode ser usado para obter um caminho mínimo de um vértice s a cada um dos restantes vértices de $G = (V, E, d)$, quando $d(e) > 0$ para todo $e \in E$.

ALGORITMODIJKSTRA(G, s)

1. Para cada $v \in V$ fazer { $pai[v] \leftarrow$ `NULL`; $dist[u] \leftarrow \infty$;}
2. $dist[s] \leftarrow 0$;
3. $Q \leftarrow$ MK_PQ_HEAPMIN$(dist, V)$;
4. Enquanto (PQ_NOT_EMPTY(Q)) fazer
5. $\quad v \leftarrow$ EXTRACTMIN(Q);
6. $\quad$ Para cada $w \in Adjs[v]$ fazer
7. $\quad\quad$ Se $dist[v] + d(v, w) < dist[w]$ então
8. $\quad\quad\quad dist[w] \leftarrow dist[v] + d(v, w)$;
9. $\quad\quad\quad pai[w] \leftarrow v$;
10. $\quad\quad\quad$ DECREASEKEY$(Q, w, dist[w])$;

Terminar o ciclo antes da fila estar vazia

Como veremos na prova de correção do algoritmo de Dijkstra, quando v é extraído da fila (no Passo 5.), tem-se $dist[v] = \delta(s, v)$, sendo $\delta(s, v)$ **a distância mínima de** s **a** v. Por isso, só fará sentido prosseguir o ciclo "Enquanto"

se o nó v que sair da fila tiver $dist[v] < \infty$, porque se $dist[v] = \infty$, o nó v não é acessível de s e nenhum dos restantes ainda na fila o é também. Por outro lado, quando se pretende determinar um caminho mínimo de s para t, para um certo par (s, t), o ciclo "Enquanto" deve ser interrompido logo que o destino t for extraído da fila Q, pois tudo o que o algoritmo fizer a seguir nesse ciclo é desnecessário para a resposta pretendida.

Para as alterações referidas, é útil estender a linguagem de pseudocódigo introduzida para ter uma instrução semelhante ao `break`, da linguagem Java ou C, e se poder escrever "Se $(v = t)$ então **terminar o ciclo**;". A utilização desta instrução deve ser feita de forma criteriosa para que não conduza a uma estruturação deficiente dos algoritmos e, consequentemente, dos programas que os codificam numa linguagem de programação.

Complexidade temporal do algoritmo de Dijkstra

Por analogia com o algoritmo de Prim, concluimos que a complexidade temporal do algoritmo é $O((|V|+|E|)\log_2|V|)$, se a representação do grafo for baseada em listas de adjacências e a fila de prioridade for suportada por uma *heap de mínimo*. A complexidade é dominada pela complexidade do ciclo "Enquanto", que é dada por:

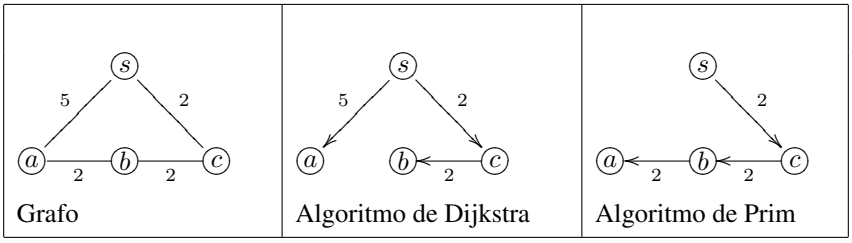
$$O(\sum_{v\in V}(1+\log_2|V|+|Adjs[v]|\log_2|V|)) = O(|V|\log_2|V|+|E|\log_2|V|) = O((|V|+|E|)\log_2|V|).$$

Neste caso, devemos manter a expressão nesta forma pois não sabemos qual é a ordem de grandeza de $|E|$.

Árvore de caminhos mínimos com origem em s

Como nos algoritmos anteriores, o valor $pai[v]$ permite indicar o caminho mínimo de s até v que foi encontrado pelo método. O vetor pai permite também criar uma árvore com raíz em s e que contém esses caminhos mínimos. Essa árvore de caminhos mínimos é formada por s e pelos vértices v tais que $dist[v] \neq \infty$, tendo por ramos $(pai[v], v)$, para esses vértices.

O algoritmo de Dijkstra pode ser aplicado também a grafos $G = (V, E, d)$ não dirigidos, bastando considerar o grafo dirigido simétrico associado, i.e., o grafo adjunto de G, com pesos idênticos nos dois ramos que substituem $\langle u, v\rangle \in E$. Quando G é conexo, **a árvore dos caminhos mínimos com origem em** s contém todos os vértices mas nem sempre corresponde a uma árvore geradora mínima de G. Consequentemente, **não pode ser obtida por aplicação do algoritmo de Prim** com origem em s. O exemplo seguinte mostra isso mesmo.



Prova de correção do algoritmo de Dijkstra

A prova será por indução sobre o número de iterações já realizadas no ciclo "Enquanto". No final da iteração k, seja $\mathcal{Q}_k$ o conjunto de vértices que se encontram na fila Q e $\mathcal{M}_k = V \setminus \mathcal{Q}_k$ o conjunto de vértices que já sairam de Q. Vamos mostrar que o algoritmo de Dijkstra mantém o invariante seguinte, para $k \geq 1$:

1. $dist[v] = \delta(s, v)$, para todo $v \in \mathcal{M}_k$;
2. $dist[v]$ seria a distância mínima de s a v se os percursos só pudessem passar por vértices de $\mathcal{M}_k \cup \{v\}$, para todo $v \in \mathcal{Q}_k$.

(Caso de base) Quando $k = 1$, o vértice que sai da fila Q é a origem s e, nos passos 6.-10., os valores de $dist$ para os adjacentes de s são alterados ficando $dist[v] = d(s, v)$, para todo $v \in Adjs[s]$. Portanto, as condições 1. e 2. verificam-se, já que, $\mathcal{M}_1 = \{s\}$, $dist[s] = 0 = \delta(s, s)$, por definição, para $v \in \mathcal{Q}_1 = V \setminus \{s\}$, os caminhos mínimos de s para v que não têm vértices intermédios são constituídos apenas pelo ramo (s, v) e, portanto, têm comprimento $d(s, v)$, ou não existem e $dist[v] = \infty$.

(Hereditariedade) Suponhamos, como hipótese de indução, que o invariante se verifica no final da iteração k, para k fixo, sendo $k \geq 1$, e que $\mathcal{M}_k \neq V$, ou seja, Q não está vazia. Vamos mostrar que então se verifica também no final da iteração $k + 1$. Designemos por $\hat{d}(\gamma)$ o comprimento de um percurso γ, ou seja, $\hat{d}(\gamma) = \sum_{(x,y)\in\gamma} d(x, y)$.

Se existirem vértices em $\mathcal{Q}_k$ acessíveis de s, seja w um vértice de $\mathcal{Q}_k$ que se encontra a distância mínima de s, isto é, tal que $\delta(s, w) = \min\{\delta(s, v) \mid v \in \mathcal{Q}_k\}$ e seja $\gamma_{s,w}$ um caminho mínimo de s para w em G, isto é, tal que $\hat{d}(\gamma_{s,w}) = \delta(s, w)$.

Se $\gamma_{s,w}$ for constituído por um só ramo então w é adjacente de s e, do que provámos para $k = 1$, podemos concluir que $dist[w] = \delta(s, w) = \hat{d}(\gamma_{s,w})$.

Se $\gamma_{s,w}$ for constituído por dois ou mais ramos, seja u o vértice que precede w no caminho $\gamma_{s,w}$ e $\gamma_{s,u}$ o subcaminho até u, i.e., $\gamma_{s,w} = (\gamma_{s,u}, (u, w))$. Como $d(u, w) > 0$ e $\delta(s, w) = \hat{d}(\gamma_{s,w}) = \hat{d}(\gamma_{s,u}) + d(u, w)$, concluimos que $\delta(s, u) \leq \hat{d}(\gamma_{s,u}) < \delta(s, w)$. Assim, $u \notin \mathcal{Q}_k$, porque se u estivesse em $\mathcal{Q}_k$ então w não seria o vértice de $\mathcal{Q}_k$ que estava mais próximo de s. Como $u \notin Q_k$, então $u \in \mathcal{M}_k$. Do mesmo modo, concluimos que $\gamma_{s,u}$ só pode ter vértices que estão em $\mathcal{M}_k$ (porque a distância de s a qualquer vértice nesse caminho anterior a u seria também inferior a $\delta(s, u)$). Assim, pela hipótese de indução, $dist[u] = \delta(s, u)$ e $dist[w] \leq \hat{d}(\gamma_{s,w})$. Mas, por definição de $\delta(s, w)$ e por $\delta(s, w) = \hat{d}(\gamma_{s,w})$, essa condição só se pode verificar se $dist[w] = \delta(s, w)$.

Em resumo, mostrámos que qualquer vértice w que esteja em $\mathcal{Q}_k$ a distância mínima de s terá $dist[w] = \delta(s, w)$. Logo, para o **vértice v que se retira de Q na iteração** $k + 1$ tem-se $dist[v] = \delta(s, v)$. Como $\mathcal{M}_{k+1} = \mathcal{M}_k \cup \{v\}$, verifica-se assim também para o novo vértice v a condição 1. do invariante, no final da iteração $k + 1$. Observemos que se $r \in Adjs[v] \cap \mathcal{M}_k$, o valor de $dist[r]$ não pode ser reduzido na iteração $k + 1$ pois $dist[r] = \delta(s, r)$.

Por outro lado, para $r \in \mathcal{Q}_{k+1} = V \setminus \mathcal{M}_{k+1}$, os caminhos mínimos de s para r que não passam por vértices de $Q_{k+1} \setminus \{r\}$ são:

- ou caminhos mínimos de s para r que não passam por vértices de $Q_k \setminus \{r\}$
- ou caminhos mínimos que passam em v mas não em vértices de $Q_k \setminus \{r, v\}$ (que é $Q_{k+1} \setminus \{r\}$).

Se se verificar o primeiro caso então, de acordo com a hipótese de indução, no final da iteração k, o valor de $dist[r]$ seria a já a distância mínima de s a r com a restrição de que o caminho não passa em $Q_{k+1} \setminus \{r\}$. Se se verificar o segundo caso, mas não o primeiro, então esse caminho mínimo passa por v e, por ser mínimo terá de ser da forma $\Gamma_{s,v}, (v, r)$, para qualquer $\Gamma_{s,v}$ mínimo. Mas, r é adjacente a v e $dist[v] + d(v, r) = \delta(s, v) + d(v, r) = \hat{d}(\Gamma_{s,v}, (v, r)) < dist[r]$ (de acordo com a hipótese sobre os valores no final da iteração k). Logo, na iteração $k + 1$, o valor de $dist[r]$ seria atualizado no ciclo "Para" (passo 6.). Em resumo, mostrámos que para todo $r \in \mathcal{Q}_{k+1}$, no final da iteração $k + 1$, o valor de $dist[r]$ é a distância mínima de s a r se os percursos só pudessem passar por vértices de $\mathcal{M}_{k+1} \cup \{r\}$. Ou seja, a condição 2. do invariante é preservada no ciclo na iteração $k + 1$.

Resta analisar o caso em que **não existem vértices em $\mathcal{Q}_k$ acessíveis de s**. Nesse caso, todo $v \in \mathcal{Q}_k$ está, por convenção, a distância mínima ∞ de s e, de acordo com o invariante, no final da iteração k, tem-se $dist[v] = \infty$ para todo $v \in \mathcal{Q}_k$ (como se definiu inicialmente). O vértice v que sai da fila na iteração $k + 1$ tem $dist[v] = \infty$ e, assim, como $dist[v] + d(v, w) = \infty + d(v, w) = \infty$, não altera o valor de $dist[w]$, para nenhum $w \in Adjs[v]$. Logo, todos os vértices em $r \in \mathcal{Q}_{k+1}$ se manterão com $dist[r] = \infty$, e a condição 2. do invariante mantém-se. □

Adaptação do algoritmo de Dijkstra para obter caminhos de capacidade máxima

Sendo $G = (V, E, c)$ um grafo dirigido em que $c(u, v) \in \mathbb{R}^+$ designa a capacidade máxima do ramo (u, v), pode ser útil saber qual é a capacidade máxima dos caminhos com origem em s e destino t, para $t \in V$. A **capacidade de um caminho** γ é igual ao mínimo das capacidades dos ramos que o constituem.

Podemos resolver o problema em $O((|V| + |E|) \log_2 |V|)$ aplicando uma estratégia semelhante à do algoritmo de Dijkstra. Em vez de $dist[v]$, utilizamos $cap[v]$ para guardar a capacidade máxima do caminhos de s a v encontrados e, como anteriormente, $pai[v]$ para guardar o identificador do vértice que precede v num desses caminhos ótimos.

CAMINHOSCAPACIDADEMAXIMA(G, s)

1. Para cada $v \in V$ fazer { $pai[v] \leftarrow$ NULL; $cap[v] \leftarrow 0$;}
2. $cap[s] \leftarrow \infty$;
3. $Q \leftarrow$ MK_PQ_HEAPMAX(cap, V);
4. Enquanto (PQ_NOT_EMPTY(Q)) fazer
5. $\quad v \leftarrow$ EXTRACTMAX(Q);
6. $\quad$ Para cada $w \in Adjs[v]$ fazer
7. $\quad\quad$ Se $\min(cap[v], c(v, w)) > cap[w]$ então
8. $\quad\quad\quad cap[w] \leftarrow \min(cap[v], c(v, w))$;
9. $\quad\quad\quad pai[w] \leftarrow v$;
10. $\quad\quad\quad$ INCREASEKEY$(Q, w, cap[w])$;

É importante notar a inicialização de $cap[v]$ com zero, para todo $v \neq s$, cuja interpretação é *se não existir caminho, a capacidade do melhor caminho é 0*. Do mesmo modo, $cap[s] = \infty$ porque serão os ramos do grafo que irão restringir a capacidade máxima dos caminhos (à partida de s, a capacidade não estaria limitada).

No problema de **determinação de caminhos mínimos**, se um percurso mínimo γ_{st} de s para t passar por v, então γ_{st} tem de ser um caminho e os percursos γ_{sv} e γ_{vt} têm de ser subcaminhos mínimos. O caminho mínimo tem **subestrutura ótima**. No problema da **determinação de percursos (ou caminhos) de capacidade máxima**, os percursos de capacidade máxima não têm de ser construídos à custa de sub-caminhos de capacidade máxima. Contudo é verdade que se um percurso de capacidade máxima γ_{st} passar por v, sendo $\gamma_{st} = \gamma_{sv}\gamma_{vt}$, então podemos substituir cada um dos percursos γ_{sv} e γ_{vt} por caminhos $\gamma^{\star}_{sv}$ e $\gamma^{\star}_{vt}$ de capacidade máxima. Esta propriedade estrutural é importante na prova de correção do algoritmo apresentado, e permite demonstrar o invariante seguinte, onde $\mathcal{Q}_k$ e $\mathcal{M}_k = V \setminus \mathcal{Q}_k$ são definidos como na prova de correção do algoritmo de Dijkstra, para $k \geq 1$, e $\tau(s, v)$ é a capacidade de um percurso de capacidade máxima de s para t, no grafo G:

1. $cap[v] = \tau(s, v)$, para todo $v \in \mathcal{M}_k$;
2. $cap[v]$ seria a capacidade máxima de s a v se os percursos só pudessem passar por vértices de $\mathcal{M}_k \cup \{v\}$, para todo $v \in \mathcal{Q}_k$.

Caminhos de capacidade máxima com origem em s em grafos não dirigidos

No caso de $G = (V, E, c)$ ser um grafo não dirigido e conexo, a árvore geradora de **peso máximo criada a partir da raíz** s por adaptação do algoritmo de Prim contém um caminho de capacidade máxima de s para v, para cada

$v \in V \setminus \{s\}$. Por isso, em instâncias deste tipo, o algoritmo de Prim (adaptado para obter árvores de peso máximo) seria uma alternativa ao que apresentámos acima.

Outras variantes

Nesta secção apresentamos mais um problema (do exame de 2012/13) que pode ser resolvido por uma estratégia semelhante. Seja $\mathcal{G} = (\mathcal{V}, \mathcal{A}, p)$ um grafo dirigido, e $p : \mathcal{A} \rightarrow \mathbb{R}^+$ define os valores nos arcos. Sejam s e t dois nós, s origem e t destino, $s \neq t$. Para cada percurso γ_{uv} em $\mathcal{G}$, com origem u e fim v, designe-se por $\mathcal{P}(\gamma_{uv})$ o valor máximo nos arcos que o constituem, i.e., $\mathcal{P}(\gamma_{uv}) = \max\{p(x,y) \mid (x,y) \text{ é arco de } \gamma_{uv}\}$. O objetivo é encontrar percursos de s para t que sejam $\mathcal{P}$-*mínimos*, dizendo que γ_{uv} é $\mathcal{P}$-mínimo sse $\mathcal{P}(\gamma_{uv})$ for mínimo quando considerados todos os percursos alternativos de u para v. Neste contexto, **ótimo** significa $\mathcal{P}$-*mínimo*.

Começamos por provar algumas propriedades estruturais de qualquer percurso ótimo $\gamma^\star_{st}$ de s para t. Como no problema da determinação de percursos (ou caminhos) de capacidade máxima, não é necessário que os percursos *ótimos* que agora procuramos sejam *caminhos* ótimos constituídos por subcaminhos ótimos, mas é importante que qualquer percurso ótimo possa ser relacionado com um caminho ótimo (com exatamente o mesmo valor) e que é constituído por subcaminhos ótimos.

1. Se $\gamma^\star_{st}$ contiver ciclos, existe um percurso ϕ_{st} sem ciclos tal que $\mathcal{P}(\gamma^\star_{st}) = \mathcal{P}(\phi_{st})$, ou seja, se existe um percurso ótimo de s para t então existe um caminho ótimo de s para t.

   Prova: Se $\gamma^\star_{st}$ contiver ciclos, então existe v tal que $\gamma^\star_{vv}$ é um ciclo contido em $\gamma^\star_{st}$. Assim, $\gamma^\star_{st}$ pode-se decompor como $\gamma^\star_{sv}\gamma^\star_{vv}\gamma^\star_{vt}$ e, por definição de $\mathcal{P}$, tem-se $\mathcal{P}(\gamma^\star_{st}) = \max\{\mathcal{P}(\gamma^\star_{sv}), \mathcal{P}(\gamma^\star_{vv}), \mathcal{P}(\gamma^\star_{vt})\}$ (aqui, $\gamma^\star_{vt} = \epsilon$ se $v = t$, e $\gamma^\star_{sv} = \epsilon$ se $s = v$, e $\mathcal{P}(\epsilon) = 0$). Se retirarmos o ciclo $\gamma^\star_{vv}$, obtemos o percurso $\gamma'_{st} = \gamma^\star_{sv}\gamma^\star_{vt}$ de s para t, com $\mathcal{P}(\gamma'_{st}) \leq \mathcal{P}(\gamma^\star_{st})$. Por outro lado, como $\gamma^\star_{st}$ é ótimo, $\mathcal{P}(\gamma'_{st}) \geq \mathcal{P}(\gamma^\star_{st})$. Logo, $\mathcal{P}(\gamma'_{st}) = \mathcal{P}(\gamma^\star_{st})$, o percurso γ'_{st} é ótimo e tem menos arcos do que $\gamma^\star_{st}$. Se γ'_{st} ainda contiver ciclos, podemos aplicar o mesmo procedimento a γ'_{st} para continuar a reduzir o número de arcos do percurso ótimo. Mas, como um percurso tem um número de arcos finito e $s \neq t$, este procedimento terá de terminar, resultando um percurso com pelo menos um arco e sem ciclos, ou seja um caminho.

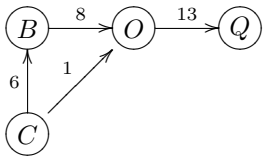
2. Se $\gamma^\star_{st}$ for um caminho com dois ou mais arcos e que passa num vértice v (fixo), então existem *caminhos* ótimos γ_{sv} e γ_{vt} tais que o percurso $\gamma_{sv}\gamma_{vt}$ de s para t é ótimo (i.e., $\mathcal{P}(\gamma^\star_{st}) = \mathcal{P}(\gamma_{sv}\gamma_{vt})$).

   Prova: Se o caminho $\gamma^\star_{st}$ passa em v, podemos decompor $\gamma^\star_{st}$ como $\gamma^\star_{sv}\gamma^\star_{vt}$, sendo $\gamma^\star_{sv}$ e $\gamma^\star_{vt}$ caminhos. Substituimos $\gamma^\star_{sv}$ e $\gamma^\star_{vt}$ por caminhos ótimos γ_{sv} e γ_{vt}, se não o forem, e $\gamma_{sv}\gamma_{vt}$ será um percuso tal que $\mathcal{P}(\gamma^\star_{st}) \geq \mathcal{P}(\gamma_{sv}\gamma_{vt})$, porque $\mathcal{P}(\gamma^\star_{st}) = \max\{\mathcal{P}(\gamma^\star_{sv}), \mathcal{P}(\gamma^\star_{vt})\} \geq \max\{\mathcal{P}(\gamma_{sv}), \mathcal{P}(\gamma_{vt})\} = \mathcal{P}(\gamma_{sv}\gamma_{vt})$. Mas, como $\gamma^\star_{st}$ é ótimo, então $\mathcal{P}(\gamma^\star_{st}) \leq \mathcal{P}(\gamma_{sv}\gamma_{vt})$, e consequentemente, $P(\gamma^\star_{st}) = \mathcal{P}(\gamma_{sv}\gamma_{vt})$.

3. Se $\gamma^\star_{st}$ for um caminho com dois ou mais arcos que passa num vértice v (fixo), pelo menos um dos dois subcaminhos $\gamma^\star_{sv}$ e $\gamma^\star_{vt}$ que constituem $\gamma^\star_{st}$ é ótimo, mas o outro pode ser ótimo ou não.

   Prova: Se o caminho $\gamma^\star_{st}$ passa em v, podemos decompor $\gamma^\star_{st}$ como $\gamma^\star_{sv}\gamma^\star_{vt}$, sendo $\gamma^\star_{sv}$ e $\gamma^\star_{vt}$ caminhos. Se estes subcaminhos fossem ambos não ótimos então, se os substituissemos por caminhos ótimos γ_{sv} e γ_{vt}, obteriamos um percurso $\gamma_{sv}\gamma_{vt}$ de s para t, tal que $\mathcal{P}(\gamma^\star_{st}) > \mathcal{P}(\gamma_{sv}\gamma_{vt})$, porque $\mathcal{P}(\gamma^\star_{st}) = \max\{\mathcal{P}(\gamma^\star_{sv}), \mathcal{P}(\gamma^\star_{vt})\} > \max\{\mathcal{P}(\gamma_{sv}), \mathcal{P}(\gamma_{vt})\} = \mathcal{P}(\gamma_{sv}\gamma_{vt})$. Assim, $\gamma^\star_{st}$ não seria ótimo, contrariando o pressuposto de que era. Portanto, pelo menos um dos subcaminhos $\gamma^\star_{sv}$ e $\gamma^\star_{vt}$ tem de ser ótimo se $\gamma^\star_{sv}$ for ótimo.

   É verdade que um dos dois subcaminhos pode não ser ótimo. Por exemplo, no grafo seguinte, $CBOQ$ é um caminho ótimo de C para Q (porque $\mathcal{P}(CBOQ) = \mathcal{P}(COQ) = 13$), contém CBO como subcaminho, e CBO não é um caminho ótimo de C para Q (porque $\mathcal{P}(CBO) = 8 > \mathcal{P}(CO) = 1$).



DIJKSTRA_ADAPTADO_MINMAX(s, t, G)

```
Para cada v ∈ V fazer { P[v] ← ∞; pai[v] ← Null;}
P[s] ← 0;
Q ← MK_PQ_HEAPMIN(P, V);
Enquanto(PQ_NOT_EMPTY(Q) ) fazer
    v ← EXTRACTMIN(Q);
    Se (v = t ou P[v] = ∞) então sai do ciclo;
    Para cada w ∈ Adjs[v] fazer
        Se P[w] > max(P[v], p(v, w)) então
            P[w] ← max(P[v], p(v, w));
            DECREASEKEY(Q, w, P[w]);
            pai[w] ← v;
Se P[t] < ∞ então ESCREVECAMINHO(t, pai);
senão escrever("Não existe caminho");
```

Se existir um percurso ótimo, terá de existir um caminho ótimo formado por subcaminhos ótimos. Este algoritmo explora essa propriedade. Se $\pi(s, v)$ for o valor ótimo de $\mathcal{P}$ para os caminhos ótimos de s para v, então, em cada passo, $P[v]$ é um majorante de $\pi(s, v)$ e corresponde ao valor ótimo se o caminho só puder ter como vértices intermédios os que já sairam da fila. Quando v é extraído da fila, $P[v] = \pi(s, v)$ e o nó que antecede v no caminho ótimo encontrado é $pai[v]$, e, para os restantes vértices y ainda na fila, $Pai[y] \geq \pi(s, y) \geq \pi(s, v)$.

5.7.2 Algoritmo de Floyd-Warshall

Dado um grafo finito $G = (V, E, d)$, em que $d : E \to \mathbb{R}^+$ define o peso de cada aresta, pode ser útil conhecer percursos γ_{st} de peso total $\sum_{e\in\gamma_{st}} p(e)$ mínimo, para **todos** os pares $(s, t) \in V \times V$, com $s \neq t$. Por exemplo, em aplicações em que $d(e)$ representa uma distância, pode ser conhecer a distância mínima entre cada par de vértices e um ou mais percursos nessas condições.

Sem perda de generalidade, vamos supor que os vértices de V são identificados por $v_1, v_2, \ldots, v_n$, ou numerados de 1 a n. Seja $D^\star_{ij}$ é distância mínima de v_i a v_j. Pretendemos obter $D^\star_{ij}$, para todo par $(v_i, v_j) \in V \times V$, isto é, queremos determinar a *matriz das distâncias mínimas* $D^\star = (D^\star_{ij})$.

O problema pode ser resolvido por aplicação do algoritmo de Dijkstra, n vezes. Tomando cada vértice v_i como origem, aplica-se o algoritmo de Dijkstra para determinar $D^\star_{ij}$, para todo j. Para uma implementação suportada por uma heap binária de mínimo, com complexidade $O((|E| + |V|) \log_2 |V|)$, a construção de $D^\star$ seria realizada em $O(|V|(|E| + |V|) \log_2 |V|)$. Assim, para grafos densos, i.e., com $|E| \in \Theta(|V|^2)$, esta estratégia resultaria num algoritmo $O(n^3 \log_2 n)$. O **algoritmo de Floyd-Warshall** (1962) que apresentamos a seguir tem complexidade $\Theta(n^3)$.

```
ALGORITMOFLOYD-WARSHALL(D, n)
    Para k ← 1 até n fazer
        Para i ← 1 até n fazer
            Para j ← 1 até n fazer
                Se D[i,j] > D[i,k] + D[k,j] então D[i,j] ← D[i,k] + D[k,j];
```

Nesta descrição, a matriz $D^\star$ é representada apenas pela variável D. Supomos que, no início, D é a matriz de distâncias associada ao grafo: $D[i, i] = 0$, para todo $v_i \in V$; e se $i \neq j$ então $D[i, j] = d(v_i, v_j)$, se $(v_i, v_j) \in E$ e $D[i, j] = \infty$, se $(v_i, v_j) \notin E$.

Idea e prova de correção. O algoritmo de Floyd-Warshall explora a subestrutura ótima dos caminhos mínimos e usa **programação dinâmica** para obter a matriz das distâncias mínimas. A sua correção baseia-se na recorrência

$$\begin{aligned} D^{(0)}_{ij} &= D[i, j] \\ D^{(k)}_{ij} &= \min(D^{(k-1)}_{ij}, D^{(k-1)}_{ik} + D^{(k-1)}_{kj}), \text{ para } k \geq 1 \end{aligned}$$

que define a distância mínima $D^{(k)}_{ij}$ de v_i para v_j quando se restringe os percursos requerendo que *só possam ter como nós intermédios $v_1, v_2 \ldots, v_k$, isto é, os numerados até k, ou nenhum*, para todo (i, j), para cada $k \geq 0$. Quando $k = n$, qualquer nó é admitido como possível nó intermédio no percurso. Logo, $D^{(n)}_{ij} = D^\star_{ij}$, para todo (v_i, v_j). Neste sentido, $D^{(k)}_{ij}$ corresponde a uma estimativa da distância mínima $D^\star_{ij}$, que se vai melhorando à medida que k aumenta. A recorrência resulta de todo o percurso ótimo de v_i para v_j que só possa ter $v_1, \ldots, v_k$ como vértices intermédios ser:

- um percurso ótimo que só pode ter $v_1, \ldots, v_{k-1}$ como vértices intermédios (ou nenhum), ou
- a concatenação (junção) de um percurso ótimo de v_i a v_k com um percurso ótimo de v_k a v_j, em que ambos só podem ter $v_1, \ldots, v_{k-1}$ como vértices intermédios (ou nenhum).

É fácil ver que terá de ser assim, pois os percursos mínimos não têm ciclos. No primeiro caso, a distância percorrida seria $D_{ij}^{(k-1)}$ e, no segundo caso, seria $D_{ik}^{(k-1)} + D_{kj}^{(k-1)}$. Como $D_{ij}^{(k)}$ é a distância no melhor dos dois casos, concluimos que $D_{ij}^{(k)} = \min(D_{ij}^{(k-1)}, D_{ik}^{(k-1)} + D_{kj}^{(k-1)})$. Por outro lado, se os percursos não pudessem ter vértices intermédios, então $D[i,j]$ seria a distância mínima de v_i para v_j, pelo que $D_{ij}^{(0)} = D[i,j]$.

Um aspeto interessante da definição de $D^{(k)}$ é o facto de $D^{(k)}$ depender *apenas* de $D^{(k-1)}$. Contudo, é importante recordar que $D^{(k-1)}$ guarda já toda a informação que é relevante (para a resposta final) e que possa ter sido acumulada na análise de $D^{(0)}, D^{(1)}, \ldots, D^{(k-1)}$.

Para concluir a prova da correção do algoritmo de Floyd-Warshall basta ver que, se $D_{ij}^{(k-1)} > D_{ik}^{(k-1)} + D_{kj}^{(k-1)}$, então $k \neq i$ e $k \neq j$ e $D[i,j]$ é alterado na iteração k ficando com $D_{ik}^{(k-1)} + D_{kj}^{(k-1)}$. É simples ver que $k \neq i$ e $k \neq j$, pois, caso contrário, isto é, se $k = i$ ou $k = j$ então $D_{ij}^{(k-1)} = D_{ik}^{(k-1)} + D_{kj}^{(k-1)}$, por a distância de qualquer nó a si mesmo ter sido definida como zero. Admitindo que o algoritmo estava correto até à iteração $k-1$, então $D[i,k] = D_{ik}^{(k-1)}$, $D[k,j] = D_{kj}^{(k-1)}$ e $D[i,j] = D_{ij}^{(k-1)}$, no início da iteração k.

O **percurso que marca a distância $D_{ij}^{(k)} = D_{ik}^{(k-1)} + D_{kj}^{(k-1)}$ é um caminho**. Claramente, por ser mínimo, só passa uma vez em v_k, por construção, e em v_i e em v_j, por ter origem em v_i e fim em v_j. Se não fosse um caminho e se, por exemplo, v_t se repetisse, o sub-percurso de v_t a v_t incluiria v_k, e $D_{ij}^{(k)} = D_{it}^{(k-1)} + D_{tk}^{(k-1)} + D_{kt}^{(k-1)} + D_{tj}^{(k-1)}$. Se retirassemos esse sub-percurso, obteriamos um percurso de v_i para v_j que não passava em v_k (só teria vértices $v_1, \ldots, v_{k-1}$ como possíveis intermédios) e teria menor distância. Então, $D_{it}^{(k-1)} + D_{tj}^{(k-1)} < D_{ij}^{(k)}$ e, como $1 \leq t < k$, $D_{ij}^{(k-1)} \leq D_{it}^{(k-1)} + D_{tj}^{(k-1)}$, contradizendo o pressuposto de que $D_{ij}^{(k)} < D_{ij}^{(k-1)}$. Portanto, o percurso que marca a distância $D_{ik}^{(k-1)} + D_{kj}^{(k-1)}$ só passa uma vez em k (e é um caminho). Logo, se algum dos valores $D[i,k]$ ou $D[k,j]$ fosse alterado na iteração k, então $D[i,j]$ não o seria. Portanto, $D[i,k]$ e $D[k,j]$ têm os valores corretos, i.e., $D[i,k] = D_{ik}^{(k-1)}$ e $D[k,j] = D_{kj}^{(k-1)}$, quando são usados na alteração de $D[i,j]$. Qualquer que seja $k \geq 1$, no final da iteração k, a matriz D é $D^{(k)}$ se no início da iteração k for $D^{(k-1)}$. □

Programação dinâmica. A programação dinâmica é um método de resolução de problemas que explora a estrutura das soluções do problema para as tentar construir à custa de soluções de subproblemas (ou de problemas relacionados). A construção da solução é muitas vezes realizada por *fases*, como no algoritmo de Floyd-Warshall. A ideia fundamental é efetuar uma tabelação das soluções dos subproblemas, de modo que cada um deles seja resolvido apenas uma vez. Assim, a programação dinâmica torna-se particularmente eficiente quando a partilha de subproblemas entre os subproblemas é significativa. No caso dos problemas de otimização, esta abordagem aplica-se quando uma estratégia ótima é (ou pode ser) constituída por subestratégias ótimas. Os problemas de determinação da distância

mínima são exemplo disso, bem como, os restantes analisados na secção 5.7.1. Por isso, a concepção de um algoritmo que implemente uma estratégia de programa dinâmica requer uma análise do problema para caraterizar a estrutura das soluções ótimas em termos das soluções ótimas dos subproblemas. Dessa caraterização pode resultar uma recorrência que define o **valor ótimo** e uma **estratégia ótima**.

Como no algoritmo de Floyd-Warshall, em que se vai sucessivamente obtendo $D^{(0)}, D^{(1)}, D^{(2)}, \ldots, D^{(n)}$, é usual implementar uma abordagem *bottom-up* (de baixo para cima) para construir a solução (que é $D^{(n)}$, nesse caso).

Também é possível seguir uma abordagem *top-down* (de cima para baixo), baseada em recursão com **memoização**. As soluções dos subproblemas visitados são tabeladas e a tabela será consultada sempre que for necessário resolver um subproblema. Este só será resolvido se a sua solução ainda não constar da tabela. Caso contrário, a solução que constar da tabela é a que é indicada. Esta abordagem tem de ser ponderada e analisada com algum cuidado porque pode requerer demasiada memória.

Caminho mínimo

Como no algoritmo de Dijkstra, em que se memorizou o nó $pai[v]$ que precede v no caminho mínimo de s para v encontrado, para cada v, também neste caso interessa manter informação que permita indicar um caminho mínimo para cada par (v_i, v_j). Ou seja, além do **valor ótimo** (distância mínima $D^{\star}_{ij}$), interessa conhecer uma **estratégia ótima** (caminho mínimo de v_i para v_j).

Para permitir também determinar o caminho mínimo, uma possibilidade é construir uma matriz $P^{(k)}$ em que $P^{(k)}_{ij}$ identifica algum vértice relevante no caminho ótimo de v_i para v_j encontrado até à iteração k. Essa matriz pode ser definida de várias formas. Para a construir e para indicar o caminho é importante saber qual é a interpretação de $P^{(k)}_{ij}$, em cada caso. Vamos analisar três possibilidades distintas: $P^{(k)}_{ij}$ identifica ou o **nó intermédio** no caminho (mínimo) de v_i para v_j que determinou $D^{(k)}_{ij}$ ou o **segundo vértice no caminho ótimo** de v_i para v_j encontrado até à iteração k ou o **penúltimo vértice nesse caminho ótimo**. Vamos analisar cada um dos casos.

- **$P^{(k)}_{ij}$ designa o nó intermédio no caminho (mínimo) de v_i para v_j que determinou $D^{(k)}_{ij}$.**

  Inicialmente, define-se $P^{(0)}_{ij} = 0$, para todo o par (i, j), o que significa que não é conhecido nenhum vértice intermédio no caminho de v_i para v_j (não existe caminho ou não existe vértice intermédio).

$$P^{(k)}_{ij} = \begin{cases} P^{(k-1)}_{ij} & \text{se } D^{(k)}_{ij} = D^{(k-1)}_{ij} \\ k & \text{se } D^{(k)}_{ij} = D^{(k-1)}_{ik} + D^{(k-1)}_{kj} < D^{(k-1)}_{ij}. \end{cases}$$

- **$P^{(k)}_{ij}$ identifica o segundo vértice no caminho ótimo de v_i para v_j encontrado até à iteração k.**

  Inicialmente, pode-se definir,

$$P^{(0)}_{ij} = \begin{cases} j & \text{se } i \neq j \text{ e } (v_i, v_j) \in E \\ 0 & \text{se } i = j \text{ ou } (v_i, v_j) \notin E. \end{cases}$$

Neste caso, $P_{ij}^{(k)} = 0$ significa que não foi encontrado percurso de v_i para v_j. Para $k > 1$, define-se

$$P_{ij}^{(k)} = \begin{cases} P_{ij}^{(k-1)} & \text{se } D_{ij}^{(k)} = D_{ij}^{(k-1)} \\ P_{ik}^{(k-1)} & \text{se } D_{ij}^{(k)} = D_{ik}^{(k-1)} + D_{kj}^{(k-1)} < D_{ij}^{(k-1)} \end{cases}$$

pois, se houver atualização, então o segundo vértice no caminho de de v_i para v_j encontrado é o segundo vértice no caminho de v_i para v_k (já conhecido). Note-se que esse vértice pode ser v_k mas não é necessário que o seja.

- **$P_{ij}^{(k)}$ identifica o penúltimo vértice no caminho ótimo de v_i para v_j encontrado até à iteração k.**

  Inicialmente, pode-se definir,

$$P_{ij}^{(0)} = \begin{cases} i & \text{se } i \neq j \text{ e } (v_i, v_j) \in E \\ 0 & \text{se } i = j \text{ ou } (v_i, v_j) \notin E. \end{cases}$$

  Para $k > 1$, define-se

$$P_{ij}^{(k)} = \begin{cases} P_{ij}^{(k-1)} & \text{se } D_{ij}^{(k)} = D_{ij}^{(k-1)} \\ P_{kj}^{(k-1)} & \text{se } D_{ij}^{(k)} = D_{ik}^{(k-1)} + D_{kj}^{(k-1)} < D_{ij}^{(k-1)} \end{cases}$$

  pois, se houver atualização, então o penúltimo vértice no caminho de de v_i para v_j encontrado é o penúltimo vértice no caminho de v_k para v_j (já conhecido). Como acima, esse vértice pode ser v_k mas não é necessário que o seja.

Apresentamos abaixo o algoritmo de Floyd-Warshall para o **primeiro caso**, supondo que P foi inicializada como $P[i,j] = P_{ij}^{(0)} = 0$, para todo (i,j). À direita, apresentamos a função para escrita do caminho ótimo encontrado para o par (v_i, v_j), para $i \neq j$. Se P for $P^{(k)}$, escreverá o caminho encontrado até à iteração k.

```
ALGORITMOFLOYD-WARSHALL(D, P, n)
    Para k ← 1 até n fazer
        Para i ← 1 até n fazer
            Para j ← 1 até n fazer
                Se D[i,j] > D[i,k] + D[k,j] então
                    D[i,j] ← D[i,k] + D[k,j];
                    P[i,j] ← k;
```

```
CAMINHO(D, P, i, j)
    Se i ≠ j ∧ D[i,j] < ∞ então
        escrever(i);
        INTERMEDIOS(P, i, j);
        escrever(j);

INTERMEDIOS(P, i, j)
    Se (P[i,j] ≠ 0) então
        INTERMEDIOS(P, i, P[i,j]);
        escrever(P[i,j]);
        INTERMEDIOS(P, P[i,j], j);
```

Quando existe caminho de v_i para v_j, a chamada CAMINHO(D, P, i, j) imprime i, os vértices *intermédios* no percurso de i para j (se existir algum), e finalmente j.

Grafos, percursos e relações binárias

Uma relação binária R definida num conjunto A é qualquer subconjunto de $A \times A$. Quando A é finito, a relação R pode ser representada por um grafo dirigido $G_R = (A, R)$, identificando os vértices de G com os elementos do conjunto A e o conjunto de ramos de G com os pares que constituem a relação R. Sendo $R \subseteq A \times A$ e $S \subseteq A \times A$, definimos a **relação composta** RS por

$$RS = \{(a, c) \mid \text{existe } b \in A \text{ tal que } (a, b) \in R \wedge (b, c) \in S\}.$$

Quando $R = S$, dizer que $(a, b) \in R \wedge (b, c) \in R$, para algum b, equivale a dizer que (a, b, c) é um percurso de a para c com dois ramos em G_R, para algum b. Ou seja, $(a, c) \in RR$ se e só se existe algum percurso de comprimento 2 de a para c em G_R.

A composição de relações binárias é associativa. Assim, podemos definir:

$$\begin{aligned} R^1 &= R \\ R^2 &= RR \\ R^3 &= R(RR) = (RR)R = R^2R = RR^2 \\ &\vdots \\ R^p &= RR^{p-1} = R^{p-1}R, \quad \text{para todo } p \in \mathbb{Z}^+, p \geq 2. \end{aligned}$$

A equivalência das duas definições de R^p resulta da associatividade da composição. Se usarmos como definição $R^p = RR^{p-1}$, para $p > 1$, e $R^1 = R$, concluimos que, se $p > 1$ então $(a, c) \in R^p$ sse existe um $b \in A$ tal que $(a, b) \in R \wedge (b, c) \in R^{p-1}$. Usando essa definição podemos mostrar o lema seguinte por indução matemática.

Lema 1 *Seja $G = (A, R)$ um grafo dirigido e sejam u e v vértices em A, quaisquer. Dado $p \in \mathbb{Z}^+$ existe percurso de u para v de comprimento p (i.e., com p ramos) se e só se $(u, v) \in R^p$.*

Uma relação R definida em A é transitiva sse quaisquer que sejam $a, b, c \in A$, se $(a, b) \in R \wedge (b, c) \in R$ então $(a, c) \in R$. Isto quer dizer que $R^2 \subseteq R$.

O **fecho transitivo** da relação R é a menor relação transitiva que contém R. Representa-se por R^+ e podemos provar que

$$R^+ = \bigcup_{p=1}^{\infty} R^p$$

o que corresponde a dizer que $(a, b) \in R^+$ se e só se existir algum **percurso** de a para b em G_R.

Contudo, sabemos que qualquer percurso com p ramos envolve $p+1$ vértices (não necessariamente distintos), pelo que, quando A é finito, qualquer percurso com **mais do que** $n = |A|$ ramos não acrescenta nenhum par novo a R^+.

Assim, nesse caso

$$R^+ = \bigcup_{p=1}^{n} R^p.$$

Para cálculo do fecho transitivo, pode ser importante considerar R^n. Por exemplo, se $R = \{(1,2),(2,1)\}$, então $R^+ = R \cup R^2 = \{(1,2),(2,1),(2,2),(1,1)\}$.

A relação de acessibilidade em G_R corresponde à relação $R^+ \cup \{(a,a) \mid a \in A\}$, que se designa por fecho **reflexivo e transitivo** de R e representa por $R^\star$. Quando A é finito, $R^\star = \bigcup_{p=0}^{n-1} R^p$, sendo $R^0 = \{(a,a) \mid a \in A\}$, a relação identidade. Neste caso, os percursos os percursos com n ramos já não acrescentariam pares a $R^\star$.

Para determinar a matriz do **fecho transitivo** da relação R, podemos usar o algoritmo de Warshall, apresentado a seguir.

```
FECHOTRANSITIVO(M, P, n)
    Para k ← 1 até n fazer
        Para i ← 1 até n fazer
            Para j ← 1 até n fazer
                Se (M[i,j] = 0 ∧ M[i,k] = 1 ∧ M[k,j] = 1) então
                    M[i,j] ← 1;
                    P[i,j] ← P[i,k];
```

```
PERCURSO(P, i, j)
    Se (P[i,j] ≠ 0) então
        Repita
            escrever(i);
            i ← P[i,j];
        até (i = j);
        escrever(j);
```

O algoritmo de Floyd-Warshall segue uma ideia semelhante a este. Para cada k, são considerados os caminhos que, além de $v_1, \ldots, v_{k-1}$, podem ter v_k como vértice intermédio. No início M é a matriz da relação R, isto é, M_R, e é assim definida

$$M_R[i,j] = \begin{cases} 1 & \text{se } (v_i, v_j) \in R \\ 0 & \text{se } (v_i, v_j) \notin R \end{cases}$$

sendo 0 e 1 são valores booleanos (falso e verdade). Definimos $P[i,j] = j$, para todo $(v_i, v_j) \in R$ e $P[i,j] = 0$, para todo $(v_i, v_j) \notin R$. Neste caso, $P[i,j]$ identifica o **segundo vértice no percurso** de v_i para v_j encontrado. Quando $(v_i, v_i) \in R$, ou seja, quando o grafo G_R tem um lacete em v_i, o segundo vértice no percurso (v_i, v_i) é v_i.

5.7.3 Algoritmo de Bellman-Ford

O algoritmo de Bellman-Ford que é habitualmente descrito na bibliografia calcula a distância mínima $\delta(s,v)$ para todo $v \in V$, para uma origem s fixa, como o algoritmo de Dijkstra. Contudo pode ser aplicado quando existem pesos negativos, contrariamente ao algoritmo de Dijkstra. Este algoritmo (um para todos) segue uma ideia semelhante ao algoritmo que vamos apresentar, realizando uma última iteração no fim para verificar se o grafo tem ciclos com peso total negativo (caso em que a distância mínima não estaria definida).

A versão que vamos apresentar determina a **matriz das distâncias mínimas para todos os pares** $(s,t) \in V \times V$, como na secção anterior. Neste caso, vamos definir $\tilde{D}^{(r)}$ como a **matriz das distâncias mínimas por percursos de ordem não superior a** r, isto é, com r ramos no máximo, para $r \geq 1$. Se tivermos calculado $\tilde{D}^{(r)}$, podemos obter as distâncias mínimas por percursos de ordem não superior a $r+1$, analisando a distância associada a percursos que têm mais um ramo do que os percursos mínimos obtidos anteriormente. Se esses percursos com mais um ramo não forem melhores então deve-se dar preferência aos anteriores. Assim, $\tilde{D}^{(r+1)}$ pode ser definida pela recorrência:

$$\begin{aligned} \tilde{D}^{(1)} &= D^{(0)} \\ \tilde{D}^{(r+1)}_{ij} &= \min(\ \tilde{D}^{(r)}_{ij},\ \min\{\ \tilde{D}^{(1)}_{ik} + \tilde{D}^{(r)}_{kj} \mid 1 \leq k \leq n\}\ ), \quad \text{para todo } r \geq 1 \end{aligned}$$

para todo $(v_i, v_j) \in V \times V$. Como anteriormente, podemos definir $P^{(r)}_{ij}$ como o segundo vértice no caminho ótimo de i para j. Assim,

$$\begin{aligned} P^{(1)}_{ij} &= \begin{cases} j & \text{se } i \neq j \text{ e } (v_i, v_j) \in E \\ 0 & \text{se } i = j \text{ ou } (v_i, v_j) \notin E. \end{cases} \\ P^{(r+1)}_{ij} &= \begin{cases} P^{(r)}_{ij} & \text{se } \tilde{D}^{(r+1)}_{ij} = \tilde{D}^{(r)}_{ij} \\ P^{(r)}_{ik} & \text{se } k \text{ é um qualquer nó tal que } \tilde{D}^{(r+1)}_{ij} = D^{(1)}_{ik} + D^{(r)}_{kj} < D^{(r)}_{ij}. \end{cases} \end{aligned}$$

A **matriz das distâncias mínimas** $D^\star$ é uma matriz $\tilde{D}^{(r)}$ tal que $\tilde{D}^{(r+1)} = \tilde{D}^{(r)}$, isto é, que se mantém invariante quando se efetua a transformação. Como convencionámos que $D^\star_{ii} = 0$, para todo i, sabemos que $D^\star = \tilde{D}^{(n-1)}$, para $n \geq 2$, pois só será relevante considerar percursos sem ciclos. Pode acontecer que $D^\star = \tilde{D}^{(r)}$ para $r < n-1$, mas tal já depende da estrutura concreta do grafo. Será sempre verdade que r não excede o número máximo de ramos que possa ter um caminho mínimo.

Por outro lado, como $D^\star_{ii} = 0$, para todo i, a fórmula apresentada pode ser simplificada, pois equivale a:

$$\tilde{D}^{(r+1)}_{ij} = \min\{\tilde{D}^{(1)}_{ik} + \tilde{D}^{(r)}_{kj} \mid 1 \leq k \leq n\}.$$

Recordemos que para X e Y duas matrizes quadradas $n \times n$, a expressão $Z_{ij} = \sum_{k=1}^{n} X_{ik} Y_{kj}$ define o elemento Z_{ij} da matriz produto $Z = XY$, sendo Z_{ij} dado pelo produto escalar da linha i de X pela coluna j de Y. Por analogia,

podemos ver $\tilde{D}^{(r+1)}$ **como o produto de** $\tilde{D}^{(1)} \otimes \tilde{D}^{(r)}$, se substituirmos a operação "soma" (habitual) por "minímo" e a operação "produto" (habitual) por $+$.

Esta analogia permite-nos ver o cálculo de $\tilde{D}^{(r)}$ como o cálculo de uma **potência**. Recordemos o método binário para cálculo de x^n (potência de expoente natural), sendo $x > 0$ e $n \in \mathbb{N}$.

```
POTENCIA(x, n)
    p ← 1;
    Enquanto (n > 1) fazer
        Se (n%2 = 1) então
            p ← p * x;
        n ← n/2;
        x ← x * x;
    retorna p;
```

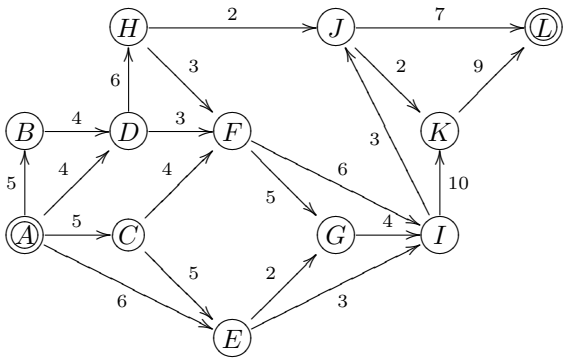
Para provar a correção deste método é importante perceber a relação de x^n com $x^{2^t b_t + 2^{t-1} b_{t-1} + \cdots + 2b_1 + b_0}$, em que $b_t b_{t-1} \cdots b_1 b_0$ é a representação de n na base 2. Por exemplo, $x^6 = x^4 x^2$ porque $6 = 110_{(2)}$, $x^{35} = x^{32} x^2 x$ porque $35 = 100011_{(2)}$. O número de iterações realizadas no ciclo para cálculo de x^n é $\lceil \log_2 n \rceil$, para $n \geq 1$. Supondo que cada operação produto tem **custo unitário**, a complexidade de POTENCIA(x, n) é $\Theta(\log_2 n)$.

Tendo em atenção o facto de $\tilde{D}^{(2)}$ agregar a informação sobre os caminhos ótimos que têm até dois ramos, $\tilde{D}^{(4)}$ pode ser obtida como $\tilde{D}^{(2)} \otimes \tilde{D}^{(2)}$, e agregará a informação sobre os caminhos ótimos que têm até quatro ramos. Assim, $\tilde{D}^{(8)} = \tilde{D}^{(4)} \otimes \tilde{D}^{(4)}$, e sucessivamente.

Tal implica que, para obter $D^\star$, basta realizar no máximo $O(\lceil \log_2 n \rceil)$ iterações, se em cada iteração calcularmos a matriz $\tilde{D}^{(2r)}$ como $\tilde{D}^{(r)} \otimes \tilde{D}^{(r)}$, para $r \geq 1$. Notar que se $t \geq \lceil \log_2 n \rceil$, então $2^t \geq n$ e $\tilde{D}^{(2^t)} = \tilde{D}^{(n)}$ (pela invariância referida acima). Como cada iteração tem um custo $\Theta(n^3)$, o algoritmo terá complexidade $O(n^3 \log_2 n)$, em vez de $O(n^4)$, que teria a versão trivial.

5.8 Fluxo máximo numa rede

Exemplo 6 A figura representa parte da rede de uma empresa de telecomunicações.



Os nós A e L representam dois centros de distribuição. Quando em A é recebida uma chamada, a mesma é encaminhada para L através de um conjunto de terminais de reencaminhamento. A capacidade de cada ligação está indicada em centenas de chamadas. Enquanto estiver a decorrer, cada chamada ocupa uma unidade de cada uma das linhas usadas para a estabelecer. Admite-se que não há corte das chamadas. Pretende-se determinar o número máximo de chamadas que podem estar a ser efetuadas em simultâneo.

Exemplo 7 Uma dada fábrica P dispõe de cinco máquinas capazes de produzir um dado produto, três à razão máxima de 5 m^3/h e as restantes à razão máxima de 10 m^3/h. Esse produto é posteriormente encaminhado através de uma rede de distribuição até um centro de distribuição D_f que o envia (sem restrições de transporte) para duas fábricas F_1 e F_2 as quais têm capacidade para consumir até 7 e 20 m^3/h respetivamente, cada uma delas necessitando de pelo menos 2 m^3/h. Não é possível enviar o produto de P para D_f diretamente, sendo necessário recorrer a centros de distribuição intermédios os quais são identificados como $D_1, \ldots, D_7$. Não há qualquer possibilidade de armazenar produto nas fábricas nem nos centros de distribuição. As capacidades máximas (em m^3/h) para escoar o produto de um dado centro para outro são as que constam da tabela seguinte.

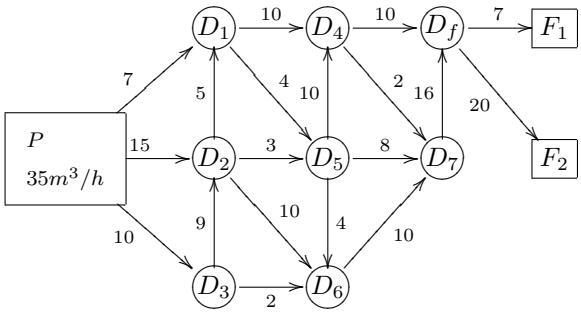
$Origem$	D_1	D_1	D_2	D_2	D_2	D_3	D_3	D_4	D_4	D_5	D_5	D_5	D_6	D_7
$Capacidade$	10	4	5	3	10	9	2	2	10	10	4	8	10	16
$Destino$	D_4	D_5	D_1	D_5	D_6	D_2	D_6	D_7	D_f	D_4	D_6	D_7	D_7	D_f

À saída da fábrica P o produto pode ser encaminhado até aos três centros D_1, D_2 e D_3 à razão máxima 7, 15 e 10 m^3/h, respetivamente.

Pretende-se determinar um plano de produção e de distribuição do produto de forma a maximizar o lucro da fábrica P. Cada uma das máquinas mais potentes tem um custo de operação (por m^3 de produto) cerca do triplo do

custo para cada uma das menos potentes, mas segundo os analistas o preço de venda do produto é suficientemente elevado para que qualquer um dos tipos seja rentável. Mostrar que é possível fazer chegar 20 m^3/h de produto a D_f e determinar uma solução óptima para o problema. Em que medida a rede de distribuição descrita pode estar a prejudicar as fábricas?

Neste exemplo, a rede de distribuição pode ser descrita esquematicamente pelo grafo seguinte, em que o valor em cada ramo representa a capacidade da ligação.



Como a produção é rentável, há que determinar o valor máximo do *fluxo máximo* que a rede de distribuição permite encaminhar. Assim, será possível perceber se a rede limita a produção. Não é necessário que toda a produção siga o mesmo trajeto na rede. Ambos os exemplos apresentados requerem a determinação do **fluxo máximo** numa rede.

No problema "Encomenda", em que se pretendia aferir a capacidade de uma rede para transporte de um animal numa jaula, não seria possível fraccionar a jaula. Por isso, o problema era o da determinação de um caminho de capacidade máxima de uma origem s para um destino t, e foi analisado na secção 5.7.1. Nesta secção vamos considerar o caso em que o produto não tem que seguir todo o mesmo trajeto.

Uma **rede** é um grafo $G = (V, A, c, \{s, t\})$ com valores nos ramos e em que se distinguem dois vértices $s, t \in V$, sendo s a **origem** ("source") e t o **destino** ("target"). A função c indica a capacidade de cada ramo: $c(u, v) \in \mathbb{R}_0^+$ é a **capacidade de** $(u, v) \in A$. Para simplificar o modelo matemático, estendemos c a $V \times V$, definindo $c(u, v) = 0$ se $(u, v) \notin A$, e assumimos que $c(u, v) > 0$ se $(u, v) \in A$. Um **fluxo** na rede G é uma função $f : V \times V \longrightarrow \mathbb{R}$ com as propriedades seguintes:

- $f(u, v) = -f(v, u)$, para todo $(u, v) \in V \times V$;
- $f(u, v) \leq c(u, v)$, para todo $(u, v) \in V \times V$;
- $\sum_{v \in V} f(u, v) = 0$, para todo $u \in V \setminus \{s, t\}$.

A primeira propriedade implica a não existência de fluxo positivo simultaneamente de u para v e de v para u. Assim, nesta definição de fluxo $f(u, v)$ corresponde ao **fluxo neto** em (u, v). Se passassem 5 unidades em (u, v) e 3 em

(v,u) então definiriamos $f(u,v)=2$ e, pela definição, $f(v,u)=-2$. Pela segunda propriedade, o fluxo não excede a capacidade de nenhuma ligação. Diz-se que que o **ramo** (u,v) **está saturado** quando $f(u,v)=c(u,v)$. Se $f(u,v)>0$ diz-se que **existe fluxo de** u **para** v. O **valor do fluxo na rede** representa-se por $|f|$ e é o fluxo total que sai da origem, sendo dado por:

$$|f|=\sum_{v\in V} f(s,v)=\sum_{v\in V} f(v,t).$$

É igual ao fluxo total que entra no nó destino pois a terceira propriedade implica a conservação de fluxo. Podemos mostrar que quantidade de fluxo que chega a um nó intermédio u é igual à quantidade que u reenvia para outros nós, para todo $u\in V\setminus\{s,t\}$, isto é

$$\sum_{v\in V, f(v,u)>0} f(v,u)=\sum_{v\in V, f(u,v)>0} f(u,v).$$

De facto,

$$0=\sum_{v\in V} f(u,v)=\sum_{v\in V, f(u,v)>0} f(u,v)+\sum_{v\in V, f(u,v)<0} f(u,v)$$

e, da primeira condição, resulta que $\sum_{v\in V, f(u,v)<0} f(u,v)=\sum_{v\in V, f(v,u)>0}(-f(v,u))$. Consequentemente,

$$\sum_{v\in V, f(u,v)>0} f(u,v)=-\sum_{v\in V, f(u,v)<0} f(u,v)=-\sum_{v\in V, f(v,u)>0}(-f(v,u))=\sum_{v\in V, f(v,u)>0} f(v,u).$$

Assim, o débito em t é igual ao valor do fluxo que sai de s e não há retenção nem criação de fluxo nos nós intermédios.

5.8.1 Método de Ford-Fulkerson

O teorema de Ford-Fulkerson (1956) carateriza o fluxo máximo e a teoria subjacente à sua prova fundamenta vários métodos de resolução do problema.

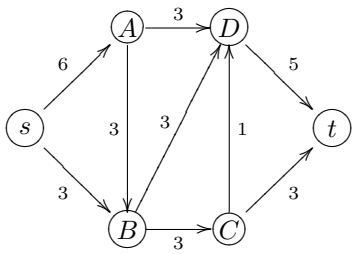
Teorema de Ford-Fulkerson: *O fluxo máximo numa rede é igual à capacidade do corte mínimo.*

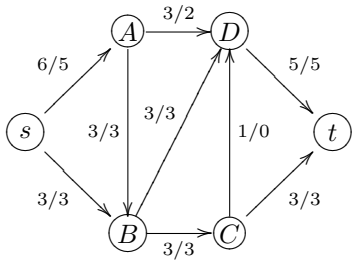
Um **corte** $\{S,T\}$ é uma qualquer partição $\{S,T\}$ de V tal que $s\in S$ e $t\in T$. Por definição de partição, tem-se $S\cup T=V$, $S\neq\emptyset$, $T\neq\emptyset$ e $S\cap T=\emptyset$. A **capacidade de um corte** $\{S,T\}$ é a soma das capacidades dos ramos de A que ligam nós de S a nós em T, ou seja,

$$c(S,T)=\sum_{u\in S, v\in T} c(u,v)$$

sendo o **corte mínimo** aquele (ou um daqueles) que tiver capacidade mínima. **O número de cortes é exponencial**. Por isso, só para redes pequenas, como a ilustrada abaixo, é possível enumerar todos os cortes e determinar desse

modo o valor do fluxo máximo. À direita está representado o fluxo ótimo ($|f| = 8$). Cada par c/f indica a capacidade e o fluxo no ramo correspondente (só representamos $f(u,v) > 0$).





Neste caso, os cortes $\{S,T\}$ e as capacidades correspondentes são:

Corte (S,T)	$c(S,T)$
$(\{s\},\{A,B,C,D,t\})$	9
$(\{s,C\},\{A,B,D,t\})$	13
$(\{s,A,C\},\{B,D,t\})$	13
$(\{s,B,D\},\{A,C,t\})$	14
$(\{s,A,B,D\},\{C,t\})$	**8**
$(\{s,A,B,C,D\},\{t\})$	**8**

Corte (S,T)	$c(S,T)$
$(\{s,A\},\{B,C,D,t\})$	9
$(\{s,D\},\{A,B,C,t\})$	14
$(\{s,A,D\},\{B,C,t\})$	11
$(\{s,C,D\},\{A,B,t\})$	17
$(\{s,A,C,D\},\{B,t\})$	14

Corte (S,T)	$c(S,T)$
$(\{s,B\},\{A,C,D,t\})$	12
$(\{s,A,B\},\{C,D,t\})$	9
$(\{s,B,C\},\{A,D,t\})$	13
$(\{s,A,B,C\},\{D,t\})$	10
$(\{s,B,C,D\},\{A,t\})$	14

O **fluxo através do corte** $\{S,T\}$ é definido por $f(S,T) = \sum_{u\in S, v\in T} f(u,v)$. Se cortarmos todas as ligações (u,v), com $u \in S$ e $v \in T$, para um corte $\{S,T\}$ qualquer, deixa de ser possível enviar fluxo de s para t. Tomando $\{S,T\}$ como um corte mínimo, conclui-se que não é possível fazer chegar a t um fluxo maior do que $c(S,T)$. Esta é uma das ideias básicas do teorema de Ford-Fulkerson. Analisando a rede (no exemplo, à direita), vemos que o valor do fluxo através de cada corte $\{S,T\}$ é igual para todos os cortes ($f(S,T) = 8 = |f|$). Tal não é uma coincidência, como se deduz do Lema 2.

Lema 2 *Qualquer que seja a rede G, tem-se: (i) o fluxo através de qualquer corte $\{S,T\}$ é igual ao fluxo na rede; (ii) $|f| \leq c(S,T)$, para todo o fluxo f e corte $\{S,T\}$.*

Prova: Por definição, $f(S,T) = \sum_{u\in S, v\in T} f(u,v) = \sum_{v\in T} f(s,v) + \sum_{u\in S\setminus\{s\}} \sum_{v\in T} f(u,v)$. Para concluir que $f(S,T) = |f|$ basta ver que $\sum_{u\in S\setminus\{s\}} \sum_{v\in T} f(u,v) = 0$. Como pela $u \in S \setminus \{s\}$ implica que $u \neq s$ e $u \neq t$ (pois $t \in T$), tem-se $\sum_{v\in T} f(u,v) = 0$, pela definição de fluxo. Daí conclui-se que $\sum_{u\in S\setminus\{s\}} \sum_{v\in T} f(u,v) = 0$.

Por outro lado, de $|f| = f(S,T) = \sum_{u\in S, v\in T} f(u,v) \leq \sum_{u\in S, v\in T} c(u,v) = c(S,T)$, segue $|f| \leq c(S,T)$, qualquer que seja o corte $\{S,T\}$. □

Do Lema 2 resulta que $|f|$ não excede a capacidade do corte mínimo, pois $|f| \leq c(S,T)$ para todos os cortes $\{S,T\}$. Portanto, a capacidade do corte de capacidade mínima é um majorante do fluxo máximo.

A determinação do fluxo máximo por **enumeração de todos os cortes (um método "força-bruta")** teria complexidade exponencial no número de vértices de G. Basta ver que o número de subconjuntos de $V \setminus \{s,t\}$ é $2^{|V|-2}$ e que cada subconjunto determina S numa partição $\{S,T\}$, pois tem-se sempre $T = V \setminus S$.

Contudo, da prova do teorema Ford-Fulkerson vão resultar as ferramentas necessárias para o desenvolvimento de métodos polinomiais para resolver o problema da determinação do fluxo máximo. Esta prova baseia-se na análise da existência de caminhos de s para t numa rede naturalmente associada a um fluxo f. Essa rede chama-se **rede residual associada à rede G e ao fluxo f** e é definida por $G_f = (V, A_f, c_f, \{s,t\})$, sendo a função de capacidade c_f, designada por **capacidade residual**, dada por

$$c_f(u,v) = c(u,v) - f(u,v)$$

e o conjunto de ramos $A_f = \{(u,v) \mid (u,v) \in V \times V,\ c_f(u,v) > 0\}$. À direita, podemos ver a rede residual associado ao fluxo f analisado acima (representado à esquerda). Notar que, por exemplo, $c_f(s,A) = 1$ e $c_f(A,s) = 5$ porque $c_f(A,s) = c(A,s) - f(A,s) = 0 - (-f(s,A)) = 5$ e $c_f(s,A) = c(s,A) - f(s,A) = 6 - 5 = 1$.

Iremos ver que, do teorema de Ford-Fulkerson resulta que o fluxo f é máximo se e só se não existe caminho de s para t na rede residual G_f. Logo, neste caso, concluimos que o fluxo $|f| = 8$ é o máximo. Analisando G_f podemos identificar um corte mínimo: $S = \{$ vértices acessíveis de s em G_f $\} = \{s, A, B, D\}$ e $T = V \setminus S = \{C, t\}$.

Chama-se **caminho para aumento do fluxo** f a qualquer caminho de s para t no grafo residual G_f e **capacidade residual desse caminho** à capacidade residual mínima dos arcos no caminho. Tal designação é justificada pelo Lema 3.

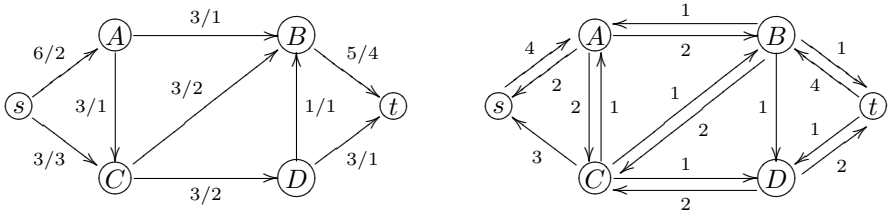
Lema 3 *Se f' é um fluxo em G_f, então $f + f'$ é um fluxo em G, tal que $|f + f'| = |f| + |f'|$.*

Prova: Sendo f um fluxo em G e f' um fluxo em G_f, podemos mostrar que $f + f'$ satisfaz as três condições da definição de fluxo em G:

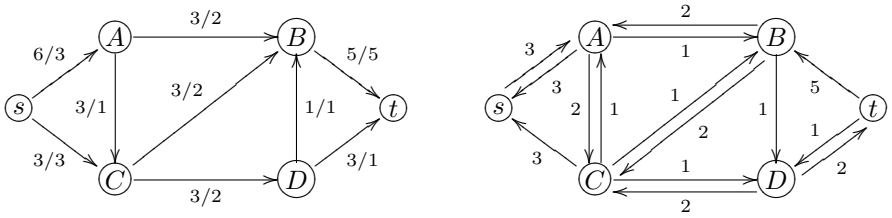
- $(f + f')(u,v) = f(u,v) + f'(u,v) = -f(v,u) - f'(v,u) = -(f + f')(v,u)$
- $(f + f')(u,v) = f(u,v) + f'(u,v) \leq f(u,v) + c_f(u,v) = f(u,v) + c(u,v) - f(u,v) = c(u,v)$
- $\sum_{v \in V}(f + f')(u,v) = \sum_{v \in V} f(u,v) + \sum_{v \in V} f'(u,v) = 0 + 0 = 0$, para todo $u \in V \setminus \{s,t\}$.

Resta agora verificar que $|f + f'| = |f| + |f'|$. Para isso basta notar que $|f + f'| = \sum_{v \in V}(f + f')(s, v) = \sum_{v \in V} f(s, v) + \sum_{v \in V} f'(s, v) = |f| + |f'|$. □

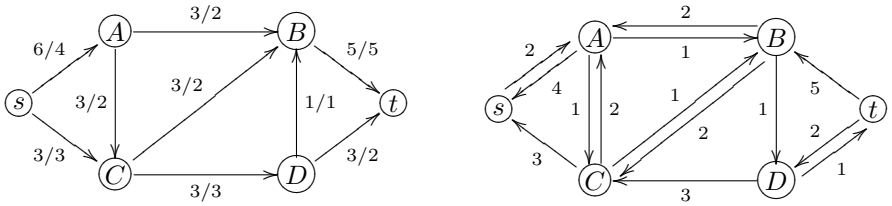
Apresentamos novamente o exemplo partindo de um fluxo de 5 unidades.



Existe caminho de s para t em G_f: (s, A, B, t), com capacidade 1, permite aumentar o fluxo de 1 unidade.



Continua a existir caminho de s para t na nova rede residual: o caminho (s, A, C, D, t), que tem capacidade 1, permite aumentar o fluxo de 1 unidade. Após essa iteração, o fluxo não é ainda máximo pois (s, A, C, B, D, t) é um caminho para aumento (seria necessário aumentar novamente o fluxo para finalmente obter o fluxo máximo $|f^\star| = 8$).



Prova: (do Teorema de Ford & Fulkerson) Para provar que o valor do fluxo máximo é igual à capacidade do (ou de um) corte mínimo, vamos provar que é equivalente dizer que f é máximo, dizer que não existe caminho para aumento de f em G_f e dizer que $|f| = c(S, T)$ para algum corte $\{S, T\}$. Para mostrar essas três equivalências basta mostrar as três implicações (5.1)–(5.3).

$$f \text{ é fluxo máximo} \quad \Rightarrow \quad \text{Não existe caminho de } s \text{ para } t \text{ em } G_f \tag{5.1}$$

$$\text{Não existe caminho de } s \text{ para } t \text{ em } G_f \quad \Rightarrow \quad |f| = c(S, T) \text{ para algum corte } \{S, T\} \tag{5.2}$$

$$|f| = c(S, T) \text{ para algum corte } \{S, T\} \quad \Rightarrow \quad f \text{ é fluxo máximo} \tag{5.3}$$

Para justificar (5.1), observamos que se existisse caminho γ de s para t e c_γ fosse a capacidade (residual) mínima dos arcos nesse caminho, então era possível enviar um fluxo c_γ em G_f de s para t. Pelo Lema 3, era possível aumentar f

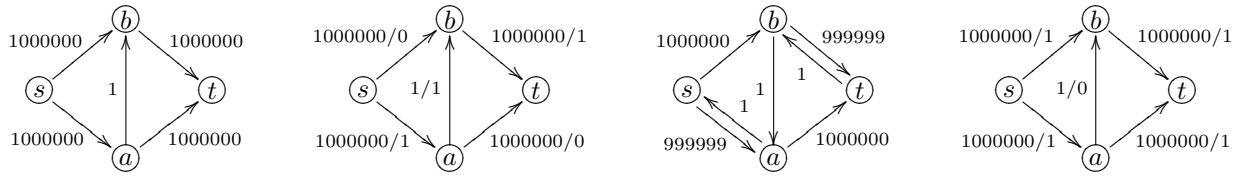
de c_γ, e portanto f não seria máximo. Para (5.2), sabemos que se não existir caminho em G_f de s para t, então $S = \{u \mid u \text{ é acessível de } s \text{ em } G_f\}$ e $T = V \setminus S$ definem um corte na rede tal que $f(u,v) = c(u,v)$, para todo $(u,v) \in (S \times T)$, pois, se não existe o ramo (u,v) em G_f então (u,v) está saturado em f. Logo, $f(S,T) = c(S,T)$, e como, pelo Lema 2, o fluxo na rede é igual ao fluxo através de qualquer corte, deduzimos $|f| = f(S,T) = c(S,T)$. Também do Lema 2 se conclui (5.3), uma vez que sendo $|f|$ menor ou igual à capacidade de qualquer corte, se $|f| = c(S,T)$, então $c(S,T)$ será a capacidade do corte mínimo, e consequentemente $|f|$ será máximo. □

O método de Ford-Fulkerson, apresentado abaixo resulta da prova deste teorema.

METODO_FORD-FULKERSON(G)

Para cada $(u,v) \in G.A$ fazer $f(u,v) \leftarrow 0$; $f(v,u) \leftarrow 0$;
Determinar a rede residual G_f; /* neste caso $G_f = G$ porque f é nulo */
Enquanto existir um caminho γ de s para t em G_f fazer:
    $c_\gamma \leftarrow \min\{c_f(u,v) \mid (u,v) \in \gamma\}$;
    Para cada $(u,v) \in \gamma$ fazer
        $f(u,v) \leftarrow f(u,v) + c_\gamma$;
        $f(v,u) \leftarrow -f(u,v)$;
    Actualizar G_f; /* alteração afeta apenas os ramos de γ e simétricos */

O método apresentado **não define nenhum critério para a escolha do caminho para aumento**. O exemplo seguinte mostra que o número de iterações pode ser exponencial no tamanho da instância: se se for escolhendo alternadamente os caminhos (s,a,b,t) e (s,b,a,t) são necessárias 2000000 iterações (mas, bastariam duas se se tomasse os caminhos (s,a,t) e (s,b,t)).

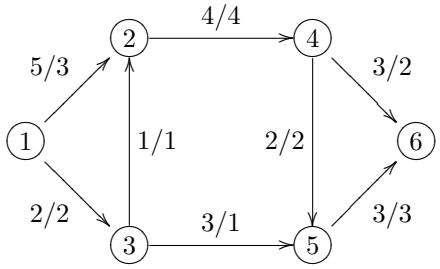


Para **capacidades inteiras**, o tamanho da instância é definido por $(n, m, \lceil \log_2 C \rceil)$, em que $n = |V|$, $m = |A|$, e C é o máximo das capacidades dos ramos (supõe-se que C é representado em binário, como é usual). Como a determinação do caminho para aumento pode ser feita em $O(m)$, e o exemplo dado mostra que o método pode requer $O(Cn)$ iterações, concluimos que o método de Ford-Fulkerson tem complexidade $O(mnC)$. Como $C = 2^{\log_2 C}$, concluimos que é $O(mn2^p)$, sendo $p = \lceil \log_2 C \rceil$. Por outras palavras, o método pode ter **complexidade exponencial** no número de bits necessários para representar a instância. Este aspeto é importante mesmo quando admitimos que as capacidades se podem representar por inteiros de 32 bits (ou de 64 bits). Nesse caso, estamos a supor só serão consideradas instâncias em que $C \leq \mathcal{K}$, para uma certa constante $\mathcal{K}$, e, consequentemente, $O(mnC) = O(mn)$,

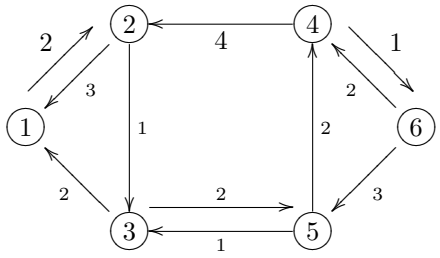
pela definição de complexidade assintótica $O(\cdot)$. No entanto, há que recordar que, nesta análise se "esquece" uma constante que pode ser muito grande, $\mathcal{K} \approx 2^p$, que para $p = 64$ é $2^{64} = 18446744073709551616 > 10^{19}$).

Um aspeto interessante do método de Ford-Fulkerson é permitir construir o fluxo máximo a partir de qualquer fluxo f já conhecido. Basta determinar a rede residual associada f e iniciar a procura de caminhos para aumento do fluxo. Assim, a **inicialização do fluxo com valor zero** faz sentido se se estiver a resolver o problema usando um computador (e não se conhecer um fluxo inicial), mas não é prudente se se estiver a resolver o problema "à mão". Nesse caso devemos começar por determinar um fluxo já *suficientemente grande* para diminuir o número de iterações do processo.

Por exemplo, consideremos uma rede de distribuição de água com a seguinte configuração, $s = 1$ e $t = 6$.

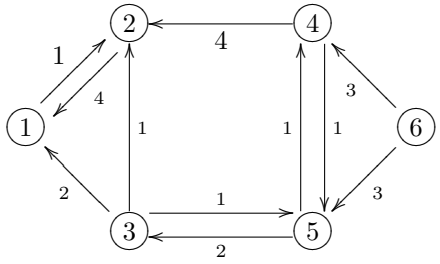


Em cada par c/d associado aos ramos, c representa a capacidade máxima do tubo correspondente e d o débito atual de água nesse tubo. A água circula no sentido indicado pelos arcos. Concluimos que o débito de água em 6 é $f(4,6) + f(5,6) = 2 + 3 = 5$. O grafo residual associado ao fluxo dado é



e como existe caminho de 1 para 6 neste grafo – o caminho (1,2,3,5,4,6) – é possível aumentar o fluxo de 1 unidade (a capacidade mínima dos arcos no caminho). O valor do fluxo passa a ser 6. Não é agora possível melhorar esse fluxo, pois a capacidade dos tubos que ligam ao nó 6 é $c(4,6) + c(5,6) = 6$. Ou seja, o valor do fluxo é igual à capacidade do corte $(\{1,2,3,4,5\},\{6\})$ e pela prova do teorema de Ford-Fulkerson, tal corte será um **corte mínimo**.

Chegar-se-ia à mesma conclusão se se voltasse a desenhar o grafo residual.



Da prova do teorema de Ford-Fulkerson conclui-se também que se as capacidades forem inteiras então existe algum fluxo máximo com valores inteiros.

Corolário 8.1 *Existe um fluxo máximo f tal que $f(u,v) \in \mathbb{Z}$, para todo (u,v), se $c(u,v) \in \mathbb{Z}_0^+$, para todo $(u,v) \in A$.*

Vimos que a escolha dos caminhos para aumento pode ter uma influência significativa no número de iterações necessárias para atingir o fluxo máximo. Mas, mais problemático ainda é o facto de a não especificação do critério de seleção do caminho para aumento poder conduzir à **não terminação do método de Ford-Fulkerson** se as capacidades puderem ser números reais (não necessariamente racionais). Por definição de algoritmo, um método que não termina em alguns casos **não é um algoritmo**.

5.8.2 Algoritmo de Edmonds-Karp

O algoritmo de Edmonds-Karp é uma variante do método de Ford-Fulkerson em que os caminhos para aumento são obtidos por pesquisa em largura. Desta forma, o método termina sempre e tem complexidade polinomial $O(m^2n)$. A prova baseia-se no seguinte. Se um certo ramo (u,v) de E pertencer ao caminho γ obtido numa certa iteração para aumento de fluxo e $c(u,v) = c_\gamma$, então (u,v) não pertencerá à rede residual na iteração seguinte (estará apenas (v,u) pois (u,v) fica saturado). Diz-se que o ramo (u,v) é critíco em γ. Se o ramo (u,v) voltar a aparecer novamente como ramo crítico numa iteração posterior do algoritmo, então pode-se demonstrar que o comprimento do caminho de s até v nessa nova rede residual excede em pelo menos duas unidades o comprimento que tinha da vez anterior. Assim, (u,v) não poderá ser crítico mais do que $n/2$ vezes pois, caso contrário, o comprimento do caminho de s até v teria pelo menos n ramos, e não seria um caminho (pois teria ciclos). Portanto, como o número de ramos é m, o número de iterações não pode exceder $mn/2$. Consequentemente, se o custo de cada iteração é $O(m)$, então a complexidade do algoritmo é $O(m^2n)$.

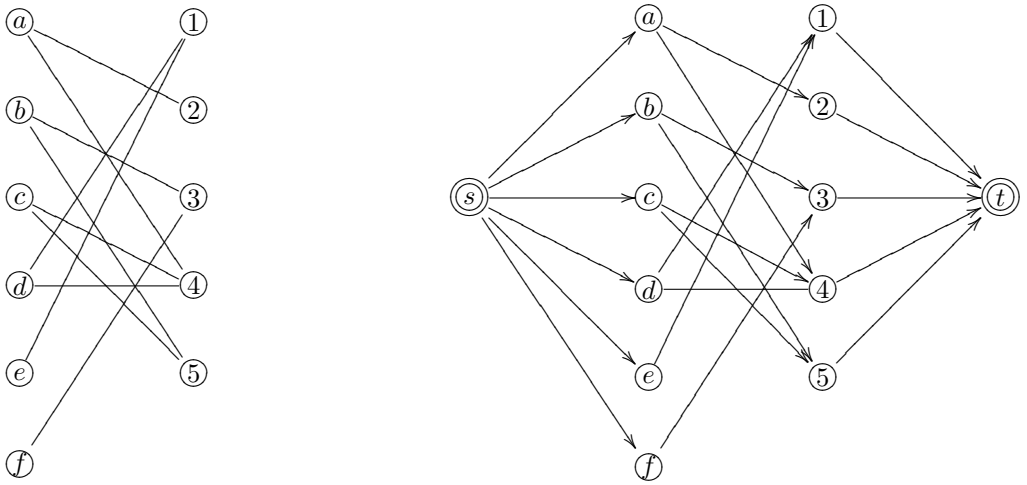
5.9 Emparelhamentos de cardinal máximo em grafos bipartidos

Um **grafo bipartido** $G = (V, E)$ é um grafo em que o conjunto de vértices admite uma partição $\{V_1, V_2\}$ tal que qualquer ramo do grafo tem um extremo em V_1 e um extremo em V_2.

Chama-se **emparelhamento** a qualquer subconjunto de ramos em E que não partilham extremos. Um emparelhamento máximo (ou de cardinal máximo) é um subconjunto máximo de ramos em E que não partilham extremos.

O grafo pode servir de modelo a um problema de afetação de tarefas a máquinas (de pessoas a postos de trabalho, etc). Nem todos os pares tarefa/máquina são admissíveis. Os ramos do grafo definem os pares admissíveis. Se cada máquina só tiver capacidade para realizar uma tarefa e cada tarefa só puder ser atribuída a uma máquina, então um emparelhamento corresponde a uma solução possível, que pode não ser ótima. Aqui, *ótima* significa que assegura a realização do número máximo de tarefas possível. A determinação de uma solução ótima corresponde à determinação de um emparelhamento de cardinal máximo no grafo.

O problema da determinação de um emparelhamento máximo num grafo bipartido pode ser reduzido a um problema de determinação de fluxo máximo. Para isso, consideramos uma rede $G' = (V \cup \{s, t\}, E', c, \{s, t\})$ em que s, t são dois novos vértices e E' é o conjunto de ramos de E orientados de V_1 para V_2, a que acrescentamos ramos (s, v_1) para cada um dos vértices de $v_1 \in V_1$ e (v_2, t) para cada $v_2 \in V_2$. Definimos as **capacidades de todos os ramos como** 1. Por exemplo, para o grafo representado à esquerda, o modelo para a rede de fluxo seria o representado à direita (com capacidades unitárias).



Se representassemos a rede residual correspondente ao emparelhamento $M = \{\langle a, 4\rangle, \langle b, 3\rangle, \langle c, 5\rangle, \langle d, 1\rangle\}$, concluiamos que não é óptimo. Nessa rede residual, o caminho $(s, e, 1, d, 4, a, 2, t)$ seria um caminho para aumento. Esse caminho levaria ao emparelhamento $M' = \{\langle a, 2\rangle, \langle b, 3\rangle, \langle c, 5\rangle, \langle d, 4\rangle, \langle e, 1\rangle\}$ que tem mais um elemento. Na rede residual correspondente a M', não existiria caminho de s para t. Note-se que neste caso não era sequer necessário construir essa rede uma vez que o número máximo de ramos em qualquer emparelhamento não pode exceder

$\min(|V_1|, |V_2|)$, que neste caso é 5, pois $V_1 = \{a, b, c, d, e, f\}$ e $V_2 = \{1, 2, 3, 4, 5\}$.

A estrutura dos problemas de fluxo máximo que servem de modelo a este tipo de problemas tem algumas propriedades que podem ser exploradas para desenvolver algoritmos específicos mais eficientes. Em particular, podemos concluir que, neste caso, qualquer caminho para aumento de fluxo é um caminho de comprimento ímpar e que se excluirmos o ramo inicial e final (que terá de ter sempre) usará *alternadamente* ramos fora do emparelhamento e ramos no emparelhamento. Assim, no exemplo, podemos ver que o caminho para aumento $(s, e, 1, d, 4, a, 2, t)$ corresponde ao **caminho alternado para aumento** $\gamma = (e, 1, d, 4, a, 2)$, isto é, a $\langle e, 1\rangle$, $\underline{\langle \mathbf{1}, \mathbf{d}\rangle}$, $\langle d, 4\rangle$, $\underline{\langle \mathbf{4}, \mathbf{a}\rangle}$, $\langle a, 2\rangle$, que é um caminho que parte de um vértice exposto de V_1 e termina num vértice exposto de V_2. Os **vértices expostos no emparelhamento** M (ou **livres)** são os que ficam sem par. Mais formalmente, são os vértices que não são extremo de nenhum ramo de M. A estrutura da rede residual permite-nos concluir que tal caminho terá sempre de começar por um ramo que não está em M e usar alternadamente ramos fora de M (isto é, em $E \setminus M$), e ramos em M. Se retirarmos do emparelhamento M os ramos de γ que lá estavam e os substituirmos pelos ramos de γ que lá não estavam, obtemos um emparelhamento M' com mais um ramo do que M. Como o número de ramos no caminho para aumento é ímpar, a troca terá sempre esse resultado (e.g., $\underline{\langle \mathbf{e}, \mathbf{1}\rangle}$, $\langle 1, d\rangle$, $\underline{\langle \mathbf{d}, \mathbf{4}\rangle}$, $\langle 4, a\rangle$, $\underline{\langle \mathbf{a}, \mathbf{2}\rangle}$).

O **algoritmo de Hopcroft-Karp** determina um emparelhamento de cardinal máximo em $O(m\sqrt{n})$, explorando as propriedades estruturais dos caminhos alternados para aumento. Este algoritmo adapta uma ideia semelhante ao algoritmo de Edmonds-Karp, mas **explora em simultâneo vários caminhos para aumento** que se obtêm numa visita da rede residual associada ao emparelhamento M (que determina o fluxo na rede). Esta visita é feita a partir dos vértices expostos de V_1 e organizada por níveis (pesquisa em largura para construir um DAG em que cada vértice só pode estar ligado a vértices do nível seguinte). Esse grafo é usado para identificar um conjunto de caminhos para aumento disjuntos, que serão usados nessa iteração para aumentar o emparelhamento. Cada iteração pode ser realizada em $O(m)$ e os autores provaram que o número de iterações é $O(\sqrt{n})$.

5.10 Casamentos estáveis e variantes

Alguns problemas práticos requerem a determinação de emparelhamentos em que os agentes que intervêm indicaram preferências que terão de ser atendidas. As preferências podem ser estritas ou não estritas, mútuas ou unilaterais, e são vários os critérios para definição do emparelhamento a escolher. Pode ser, *estabilidade*, como veremos a seguir, mas também, popularidade (veja-se por exemplo o problema **"Sob pressão"**, proposto nas aulas práticas), etc.

O problema dos Casamentos Estáveis (*Stable Marriage Problem*) é um problema de emparelhamento em grafos bipartidos, com preferências mútuas. Este problema e diversas variantes têm sido objeto de estudo dado o seu interesse prático (e.g., veja-se o **prémio Nobel da Economia 2012**). A versão clássica do problema traduz-se da forma seguinte.

STABLEMARRIAGE: Seja $\mathcal{H} = \{h_1, \ldots, h_n\}$ um conjunto de n homens e seja $\mathcal{M} = \{m_1, \ldots, m_n\}$ um

conjunto de n mulheres. Cada elemento de cada um dos conjuntos ordenou todos os de sexo oposto por ordem de preferência estrita. Pretende-se determinar um emparelhamento estável.

Informalmemte, um emparelhamento é, neste caso, um conjunto de n casais em $\mathcal{H} \times \mathcal{M}$, monogâmicos. Um emparelhamento E diz-se **instável** se e só se existir um par $(h, m) \notin E$ tal que h prefere m à sua parceira em E e m também prefere h ao seu parceiro em E. Caso contrário, diz-se **estável**. A estabilidade implica a não existência de divórcio.

Gale e Shapley (1962) mostraram que qualquer instância de STABLEMARRIAGE admite pelo menos um emparelhamento estável. Por exemplo, consideremos a instância seguinte, em que $n = 4$ e as listas de preferências se consideram ordenadas por ordem estritamente decrescente da esquerda para a direita.

$$
\begin{array}{llll}
h_1: & m_4, & m_2, & m_3, \quad m_1 \\
h_2: & m_2, & m_3, & m_4, \quad m_1 \\
h_3: & m_2, & m_3, & m_1, \quad m_4 \\
h_4: & m_1, & m_3, & m_2, \quad m_4
\end{array}
\qquad\qquad
\begin{array}{llll}
m_1: & h_4, & h_2, & h_1, \quad h_3 \\
m_2: & h_3, & h_1, & h_4, \quad h_2 \\
m_3: & h_2, & h_3, & h_1, \quad h_4 \\
m_4: & h_3, & h_4, & h_2, \quad h_1
\end{array}
$$

Através de uma análise de casos, podemos ver que $\{(h_1, m_4), (h_2, m_3), (h_3, m_2), (h_4, m_1)\}$ é um emparelhamento estável. Este emparelhamento pode ser obtido por aplicação do algoritmo de Gale-Shapley, apresentado abaixo em linhas gerais.

ALGORITMO DE GALE-SHAPLEY

Considerar inicialmente que todas as pessoas estão livres.
Enquanto houver algum homem h livre fazer:
    seja m a primeira mulher na lista de h a quem este ainda não se propôs;
    se m estiver livre então
        emparelhar h e m (ficam noivos)
    senão
        se m preferir h ao seu noivo atual h' então
            emparelhar h e m (ficam noivos), voltando h' a estar livre
        senão
            m rejeita h e assim h continua livre.

A prova de correção do algoritmo é simples. Por redução ao absurdo, vamos supor que existia um par $(h, m) \notin E$ que tornava o emparelhamento instável, Então, h preferia m à noiva m' com que ficou e m preferia h ao noivo com que ficou. Se h prefere m a m' então h propôs-se a m antes de se propor a m'. Mas, m só rejeitaria h, na altura ou posteriormente, para manter ou ficar com um noivo que preferisse a m. De acordo com o algoritmo, nenhuma mulher

rejeita um noivo que prefere para se tornar noiva de outro de que gosta menos. Portanto, não pode existir (h, m) nas condições referidas e, consequentemente, E é estável.

Gale e Shapley mostraram ainda que o algoritmo pode ser implementado em $O(n^2)$, o que, mais uma vez, resulta de, no pior dos casos, nenhum homem efetuar mais de n propostas, uma a cada mulher. Para tal, para que o tempo gasto por proposta seja $O(1)$, é importante que em vez de guardar a lista de preferências de cada mulher (como no exemplo), se guarde o seu nível de preferência por cada homem. Ou seja, ter $mpref[k, i]$ definida como o número de ordem que h_i ocupa na lista de preferências m_k.

Mostra-se também que o emparelhamento obtido pelo algoritmo de Gale-Shapley é *ótimo* para os homens e *péssimo* para as mulheres: qualquer homem fica com a melhor parceira que pode ter em qualquer emparelhamento estável e cada mulher fica com o pior parceiro. Obviamente, a situação reverte-se se passarem a ser as mulheres que se propõem.

Sendo E e E' emparelhamentos estáveis, diz-se que E *domina* E' *segundo os homens* (isto é, os homens preferem E a E'), e escreve-se $E \preceq E'$, se e só se qualquer homem que não tenha a mesma parceira em ambos, tem em E uma que prefere à que tem em E'. Esta relação de dominância confere ao conjunto $\mathcal{S}$ dos emparelhamentos estáveis, a estrutura de reticulado distributivo $(\mathcal{S}, \preceq)$. Dados dois quaisquer emparelhamentos estáveis E e E', o emparelhamento $E \wedge E'$, em que cada homem fica com a mulher que prefere dentre as suas parceiras em E e E', é estável. Também é estável o emparelhamento $E \vee E'$, em que cada homem fica com a mulher de que gosta menos dentre as suas parceiras em E e E'. Estes emparelhamentos $E \wedge E'$ e $E \vee E'$ são o ínfimo e o supremo entre E e E', respetivamente. Mostra-se também que E domina E' segundo os homens se e só se E' domina E segundo as mulheres.

O emparelhamento ótimo para os homens, o qual é determinado pelo algoritmo de Gale-Shapley, é o mínimo do reticulado. O máximo é o emparelhamento ótimo para as mulheres, o qual, como já se referiu pode ser determinado pelo mesmo algoritmo, se se trocarem entre si os papeis que nele desempenham os homens e as mulheres. Se estes emparelhamentos coincidirem então a instância admite um único emparelhamento estável.

A versão seguinte do algoritmo, ainda da autoria dos mesmos autores, permitiu reconhecer estas e outras propriedades estruturais das soluções do problema.

EXTENSÃO DO ALGORITMO DE GALE-SHAPLEY (1962)

Considerar inicialmente que todas as pessoas estão livres.
Enquanto houver algum homem h livre fazer:
    seja m a primeira mulher na lista atual de h;
    se algum homem p estiver noivo de m então
        p passará a estar livre;
    h e m ficam noivos;
    para cada sucessor h' de h na lista de m
        retirar o par (h', m) das listas de h' e m

Observação complementar: Estas e outras propriedades dos emparelhamentos estáveis de STABLEMARRIAGE são exploradas, por exemplo, por Gusfield (1987) para desenvolver algoritmos eficientes para alguns problemas. Em particular, desenvolveu métodos para a determinação de todos os pares estáveis com complexidade $O(n^2)$, para a enumeração de todos os emparelhamentos estáveis, com uma complexidade temporal $O(n^2 + n|\mathcal{S}|)$ e espacial $O(n^2)$ e para a construção em $O(n^2)$ de um emparelhamento estável que minimiza o grau de descontentamento máximo. Um par (h, m) é estável se ocorrer nalgum emparelhamento estável. Para uma melhor apreciação da relevância destes resultados, é de notar que o número $|\mathcal{S}|$ de emparelhamentos estáveis de uma instância de STABLEMARRIAGE pode ser exponencial em n.

5.10.1 Colocações de candidatos universidades ou postos de trabalho

O trabalho de Gale e Shapley teve como principal motivação a resolução do problema de colocação de alunos em cursos universitários nos EUA. Cada aluno candidata-se a algumas universidades e formula uma lista de preferências ordenadas estritamente. Cada universidade tem um certo número de vagas e ordena também estritamente os seus candidatos, podendo liminarmente não aceitar alguns. Note-se que nem todas as universidades têm a mesma lista de preferências. Pretende-se colocar os candidatos de acordo com as preferências mútuas.

Este problema tem por modelo uma variante de STABLEMARRIAGE que é designada na bibliografia por STABLEMARRIAGEWITHINCOMPLETELISTS. As vagas correspondem às mulheres, os alunos aos homens e as listas de preferências são incompletas (existem parceiros inaceitáveis) mas estritamente ordenadas.

Um emparelhamento (maximal) será um conjunto E de pares (h, m) com $h \in \mathcal{H}$ e $m \in \mathcal{M}$, tal que h e m se consideram mutuamente aceitáveis, não existem pares em E que partilhem elementos e não existem pares de $\mathcal{H} \times \mathcal{M}$ que possam ser acrescentados a E.

Na extensão do algoritmo de Gale-Shapley que resolve este problema, são os alunos que se propõem às universidades, resultando um emparelhamento ótimo do ponto de vista dos alunos e péssimo para as universidades. Por razões

que abaixo se apresentam, esta versão refere internos e hospitais em vez de candidatos e universidades.

ALGORITMO DE GALE-SHAPLEY (ORIENTADO POR INTERNOS)

Considerar inicialmente que todos os internos estão livres.
Considerar também que todas as vagas nos hospitais estão livres.
Enquanto existir algum interno r livre cuja lista de preferências é não vazia
    seja h o primeiro hospital na lista de r;
    se h não tiver vagas
        seja r' o pior interno colocado provisoriamente em h;
        r' fica livre (passa a não estar colocado);
    colocar provisoriamente r em h;
    se h ficar sem vagas então
        seja s o pior dos colocados provisoriamente em h;
        para cada sucessor s' de s na lista de h
            remover s' e h das respetivas listas

O critério de estabilidade das soluções é reformulado do modo seguinte. Um emparelhamento é **instável** se e só se existir um candidato r e um hospital h tais que h é aceitável para r e r é aceitável para h, o candidato r não ficou colocado ou prefere h ao seu atual hospital e h ficou com vagas por preencher ou h prefere r a pelo menos um dos candidatos com que ficou. Caso contrário, diz-se **estável**. Nesta situação, o conceito de emparelhamento é o mesmo que tinhamos anteriormente, se se considerar que a atribuição é de candidatos a vagas.

Uma das propriedades interessantes das suas soluções é a de que, para qualquer instância deste problema, os conjuntos de homens e de mulheres podem ser subdivididos em dois grupos: o daqueles que ficam com parceiro em todos os emparelhamentos estáveis e o daqueles que não têm parceiro em nenhum emparelhamento estável. Tendo em conta esta propriedade pode-se concluir que, independentemente do algoritmo que é usado, serão sempre colocados os mesmos alunos e ocupadas as mesmas vagas (não necessariamente pelos mesmos alunos).

Alguns anos depois da publicação deste algoritmo descobriu-se que um algoritmo essencialmente análogo estava já a ser usado desde 1952 nos EUA, pelo *National Intern Matching Program* (depois designado *National Resident Matching Program*, NRMP), para colocação de estudantes de medicina nos hospitais para realizarem o internato. Também aqui, cada hospital tem uma lista de preferências própria. Neste algoritmo, são, no entanto, os hospitais que se propõem aos candidatos, resultando num emparelhamento ótimo do ponto de vista dos hospitais.

Demonstra-se que é estável o emparelhamento definido pelas colocações provisórias depois da execução do algoritmo de Gale-Shapley (orientado por internos). Nesse emparelhamento, cada interno que ficar colocado fica no melhor hospital em que poderia ser colocado de forma a garantir a estabilidade do emparelhamento. Cada interno que não ficar colocado por este algoritmo, não poderá ficar colocado em nenhum outro emparelhamento estável determi-

nado por outro método. Dada uma qualquer instância do problema NRMP (internos-hospitais, alunos-universidades) se um hospital h não preencher as suas vagas, então qualquer interno que ficar colocado em h no emparelhamento estável ótimo do ponto de vista dos internos, fica também colocado em h em todos os emparelhamentos estáveis. Este último resultado é conhecido como o *Teorema dos Hospitais Rurais*, por serem aqueles que habitualmente ficam com vagas por preencher.

Em resumo, pode-se mostrar que o número de internos colocados em cada hospital é o mesmo em todos os emparelhamentos estáveis. Ficam por colocar exatamente os mesmos candidatos em todos os emparelhamentos estáveis. Qualquer hospital que não preencha as suas vagas num dado emparelhamento estável, fica com exatamente os mesmos internos em todos os emparelhamentos estáveis.

Colocações com lista de ordenação única. O problema **"Sentar ou não sentar"** (proposto nas aulas práticas) corresponde à situação em que os candidatos estão estritamente ordenados. A lista de preferências é a mesma para todas as instituições e coincide com a seriação dos candidatos. Para determinar a colocação dos candidatos basta seguir a seriação. Na colocação de um dado candidato, analisa-se as suas preferências por ordem decrescente de preferência, e coloca-se o candidato na primeira instituição que ainda tiver vagas (na lista que indicou). Não há qualquer tipo de retrocesso neste processo.

5.10.2 Listas de preferências não ordenadas estritamente

Os modelos analisados até aqui pressupõem uma ordenação estrita das listas de preferências, de ambos os grupos. Uma ordenação estrita é o que tecnicamente se designa por *uma ordem total*. Face a duas quaisquer possibilidades distintas x e y, o interveniente (homem/mulher, aluno/universidade, interno/hospital) terá que especificar se prefere x ou se prefere y, não podendo ser indiferente, isto é manifestar igual preferência por ambas.

Existem estudos sobre variantes de STABLEMARRIAGE que contemplam a possibilidade de os conjuntos de preferências não estarem totalmente ordenados, mas apenas parcialmente ordenados. Um caso particular ocorre quando a ordem é quase uma ordem total, mas existem indiferenças que se traduzem por empates. Duas dessas variantes são STABLEMARRIAGEWITHTIES e STABLEMARRIAGEWITHTIESANDINCOMPLETELISTS. Na primeira supõe-se que as listas de preferências estão completas. Isto é, que qualquer um dos n homens considera aceitável qualquer uma das n mulheres, e vice-versa, embora possa preferir umas a outras. Nestas situações há a possibilidade de reformular o conceito de estabilidade, tendo Irving (1994) introduzido três noções distintas:

- Estabilidade *fraca*: não existe um par $(h, m) \notin E$ tal que h prefere estritamente m à sua parceira em E e m prefere estritamente h ao seu parceiro em E.
- Estabilidade *forte*: não existe um par $(h, m) \notin E$ tal que pelo menos um dos dois prefere estritamente o outro ao seu par em E, e o outro também o prefere estritamente ou de igual modo.

- Estabilidade *super*: não existe um par $(h, m) \notin E$ tal que algum prefere o outro pelo menos tanto quanto prefere o seu par em E.

Uma instância de STABLEMARRIAGEWITHTIES, com preferências não estritas (empates) mas listas completas, pode não admitir qualquer emparelhamento fortemente estável ou super estável. O exemplo trivial é o caso em que cada mulher manifesta igual preferência por todos os homens, e cada homem manifesta igual preferência por todas as mulheres. Irving (1994) apresentou um algoritmo de complexidade $O(n^2)$ que permite decidir se uma dada instância de STABLEMARRIAGEWITHTIES admite algum emparelhamento fortemente estável ou super estável e, em caso afirmativo, determiná-lo. É também conhecido que qualquer instância desse problema admite sempre um emparelhamento com estabilidade fraca, bastando resolver os empates por especificação duma ordem arbitrária que os quebre, e aplicar o algoritmo de Gale-Shapley. Observe-se que, qualquer que seja a instância de STABLEMARRIAGEWITHTIES considerada, todos os emparelhamentos estáveis têm exatamente n pares, desde que exista algum emparelhamento estável (segundo o critério pretendido).

Listas de preferências incompletas e ordem não estrita

A situação é bastante diferente para STABLEMARRIAGEWITHTIESANDINCOMPLETELISTS. Por um lado, é fácil construir instâncias para as quais existem emparelhamentos fracamente estáveis que não têm o mesmo número de pares ou que diferem nos grupos de elementos que ficam emparelhados e sem par. Recorde-se que isto não se verificava no caso de STABLEMARRIAGEWITHINCOMPLETELISTS. Por outro lado, os trabalhos de Iwana et al. (1999) e de Manlove et al. (2002) mostram que, devido a esse facto, alguns problemas são agora NP-completos (*NP-complete*) e NP-difíceis (*NP-hard*), o que, à partida, não augura nada de bom quanto à possibilidade de resolver computacionalmente instâncias genéricas desses problemas. Voltaremos a este assunto num capítulo mais à frente.

Entre esses problemas incluem-se o da determinação dum emparelhamento estável que envolva um número máximo (ou mínimo) de pares e o da verificação da estabilidade dum dado par (i.e., verificar se pode ocorrer em algum emparelhamento estável). Tal acontece mesmo num caso mais restritivo, em que a indiferença se traduz por empates num dos lados apenas, esses empates ocorrem no fim das listas de preferências, existindo quando muito um empate por lista, o qual tem comprimento dois (ou seja, envolve apenas um par de escolhas, pelas quais o interveniente manifestou igual preferência). A noção de estabilidade é a de *estabilidade fraca*.

5.10.3 Colocações em postos de trabalho com prioridades e indiferença

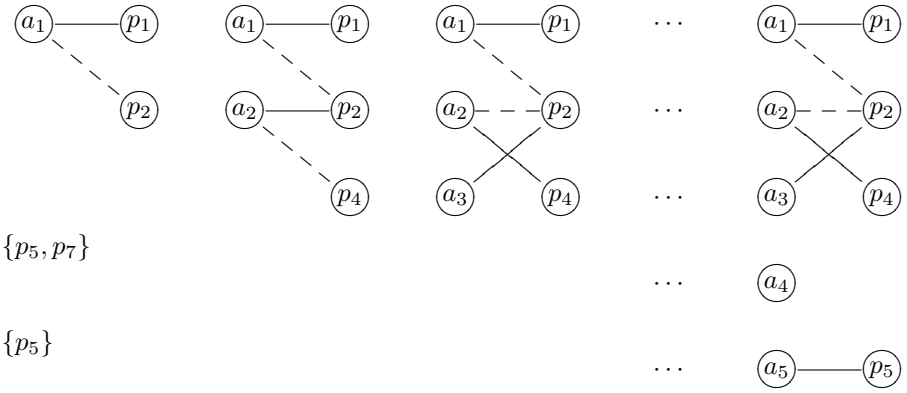
Nesta secção, vamos apresentar um problema que utiliza uma técnica de resolução semelhante à que tem sido aplicada em diversas variantes.

Seja $\mathcal{P} = \{p_1, \ldots, p_m\}$ um conjunto de postos de trabalho. Seja $\mathcal{A} = \{a_1, \ldots, a_n\}$ um conjunto ordenado de candidatos a esses postos, seriados (sem empates) de acordo com certos critérios, sendo a_1 o primeiro na seriação e

a_n o último. Os candidatos indicaram as suas listas de preferências, podendo estas conter empates. Seja $Pref(a_i) = \Gamma_{i,1} \cup \Gamma_{i,2} \cup \ldots \cup \Gamma_{i,f_i}$, a lista de preferências do candidato a_i, sendo $\Gamma_{i,1}$ o conjunto de postos que a_i indicou como sua primeira preferência, $\Gamma_{i,2}$ os que indicou como segunda preferência, etc. Pretendemos colocar os candidatos de modo que nenhum candidato seja ultrapassado nas suas preferências por um abaixo de si na seriação.

Este problema pode ser resolvido determinando emparelhamentos **máximos** numa sucessão de subgrafos que respeita a ordem dos candidatos e a ordem das suas preferências. No primeiro subgrafo, está a_1 apenas e os ramos que o ligam aos nós de Γ_{11}; no segundo, acrescenta-se a_2 e $\Gamma_{2,1}$, ou, se não for possível, $\Gamma_{2,2}$ (as preferências seguintes de a_2); depois considera-se a_3, e sucessivamente (e, em alternativa) $\Gamma_{3,1}$, $\Gamma_{3,2}$, $\Gamma_{3,3}$ (até ser possível encontrar caminho para aumento do emparelhamento), etc. Pode acontecer que alguns candidatos fiquem sem colocação. Por exemplo a_4 fica sem colocação na situação seguinte Na figura, os pares que não estão no emparelhamento estão a tracejado.

a_1 : $\{p_1, p_2\}, \{p_3\}$
a_2 : $\{p_2, p_4\}, \{p_5\}$
a_3 : $\{p_2\}, \{p_1, p_4\}, \{p_5, p_7\}$
a_4 : $\{p_1, p_4\}, \{p_2\}$
a_5 : $\{p_1, p_4\}, \{p_2\}, \{p_5\}$



Capítulo 6

Estruturas de Dados – Complementos

6.1 Filas de prioridade

*Sobre filas de prioridade suportadas por heaps binárias (*heap de mínimo e heap de máximo*)* ...

(matéria já lecionada nas aulas)

6.2 Árvores de Pesquisa e Árvores "Red-Black"

(matéria ainda a lecionar)

6.3 Conjuntos de conjuntos disjuntos

sobre estrutura de dados útil, por exemplo, para o algoritmo de Kruskal...

(matéria ainda a lecionar)

Capítulo 7

Programação Dinâmica

Outros exemplos...

7.1 Problemas de trocos

(matéria já lecionada nas aulas)

7.2 Contagem de caminhos em DAGs

(matéria já lecionada nas aulas teóricas)

7.3 Caixotes de morangos

(matéria já lecionada nas aulas)

Capítulo 8

Problemas Computacionalmente Díficeis

Sobre . . . Breve introdução das classes de complexidade de problemas de decisão P, NP e NPC e referência ao problema P=NP (matéria a aprofundar na unidade curricular de "Computabilidade", e possivelmente, de "Complexidade"). Introdução do problema de satisfação de cláusulas (SAT). Exemplos de redução (polinomial) de um problema a outro.

Existência de métodos exatos, heurísticos e metaheurísticos para resolução de problemas computacionalmente díficeis (que conseguem resolver algumas instâncias). Introdução da noção de algoritmo de aproximação, aleatorizado e probabilistico. Inaproximabilidade. (matéria a aprofundar noutras unidades curriculares, e.g., "Sistemas Inteligentes", "Métodos de Apoio à Decisão", "Métodos de Pesquisa Avançados").

Algoritmos Ávidos

(matéria já lecionada, possivelmente a complementar)

Sobre . . . Exemplos de problemas de otimização para os quais são conhecidos algoritmos greedy *para sua resolução.*

Exemplos de problemas de otimização difíceis e de estratégias greedy *que, embora não garantindo a otimalidade da solução que produzem, obtêm uma solução com alguma garantia de qualidade (ou talvez não).*