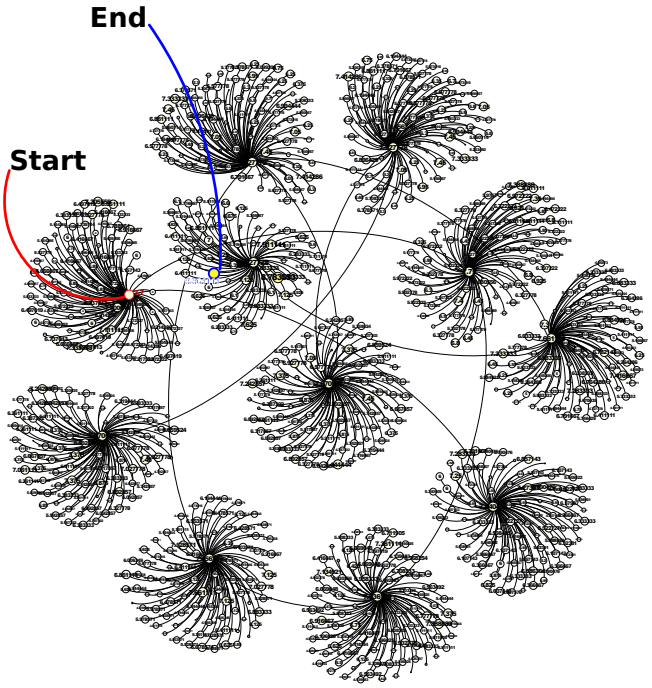
Alexandre Gonçalves Silva
Rafael de Santiago

Anotações para a disciplina de Projeto e Análise de Algoritmos

Versão de 14 de julho de 2024



Universidade Federal de Santa Catarina

Sumário

CAPÍTULO 1

Introdução ao Curso

O que veremos nessa disciplina?

- Como provar a “corretude” de um algoritmo;
- Estimar a quantidade de recursos (tempo, memória) de um algoritmo = análise de complexidade;
- Técnicas e idéias gerais de projeto de algoritmos: indução, divisão-e-conquista, programação dinâmica, algoritmos gulosos, ...
- Tema recorrente: natureza recursiva de vários problemas
- A dificuldade intrínsica de alguns problemas: desconhecimento de algoritmos eficientes.

1.1 Algoritmos

Cormen et al. (2012) define um algoritmo como “qualquer procedimento computacional bem definido que toma algum valor ou conjunto de valores como entrada e produz algum valor ou conjunto de valores como saída.”. Um algoritmo resolve um problema computacional. De acordo com Cormen et al. (2012), a especificação do problema indica quais são as entradas e quais as saídas desejadas. Um algoritmo basicamente

especifica o que deve ser feito com as entradas para que se possa gerar as saídas que atendem ao problema.

Para exemplificar, considere um problema de ordenação. Ele poderia ser definido formalmente por suas entradas e saídas (CORMEN et al., 2012):

- Entrada: uma sequência de números $\langle a_1, a_2, \ldots, a_n \rangle$.
- Saída: uma permutação (reordenação) $\langle a'_1, a'_2, \ldots, a'_n \rangle$, tal que $a'_i \leq a'_{i+1}$ para todo $i \in \{1, 2, \ldots n\}$.

. Um exemplo de entrada para esse problema seria $\langle 9, 5, 4, 10, 2 \rangle$ e um exemplo de saída seria $\langle 2, 4, 5, 9, 10 \rangle$. Um exemplo de entrada para um problema computacional é chamado de instância ou instância para o problema.

Desafio

Defina formalmente cinco problemas computacionais. Relacione um algoritmo para cada problema.

A importância dos “eficientes” algoritmos:

- projeto genoma de seres vivos;
- rede mundial de computadores;
- planejamento da produção;
- logística de distribuição;
- reconhecimento de padrões;
- entretenimento (games e filmes).

1.2 Dificuldade Computacional

O fato de conhecer algoritmos corretos não é suficiente. Precisa-se de algoritmos eficientes no contexto em que serão aplicados.

Para medir a complexidade computacional de um algoritmo, pode-se utilizar a medida empírica do tempo ou do espaço consumido. A medida empírica do tempo é questionável, pois considera situações diferentes (arquiteturas, linguagens de programaçao, sistemas operacionais e, até mesmo, condições do tempo). No contexto dessa disciplina, se deseja entender a eficiência de um algoritmo.

Para medir a dificuldade de complexidade de um algoritmo, serão utilizadas funções de complexidade. Uma função de complexidade é dada por um $f : n \in \mathbb{Z}^+ \rightarrow \mathbb{Z}^+$ e indica a quantidade de tempo requerido (contra-domínio) para uma entrada de tamanho n. De maneira geral, diz-se que um algoritmo é eficiente se o mesmo possui uma função de complexidade polinomial (polinomial × exponencial).

Para contextualizar a importância da eficiência dos algoritmos, considere dois tipos de algoritmos de ordenação: por inserção e por intercalação. Considere também que um determinado algoritmo de inserção utiliza o tempo $c_1 n^2$ e outro algoritmo de inserção utiliza $c_2 n log_2 n$. Assuma que $c_1 < c_2$ e que esses valores são constantes sem qualquer relação com n. Apesar de $c_1 < c_2$, um algoritmo de intercalação é mais eficiente para um n relativamente grande.

Considere dois computadores: um computador rápido A e um computador lento B. Considere também que o computador A executa uma ordenação por inserção e o computador B executa uma ordenação por intercalação. Cada um deve ordenar um vetor de 10 milhões de elementos. O computador A executa 10 bilhões de instruções por segundo (!!!), enquanto o computador B executa 10 milhões de instruções por segundo. Considere que $c_1 = 2$ e $c_2 = 50$. Então, o demanda de tempo o computador A é

$$\frac{2 \cdot (10^7)^2 \text{ instruções}}{10^{10} \text{ instruções/segundo}} = 20.000 \text{ segundos } (5,5 \text{ horas})$$

. O computador B demanda

$$\frac{50 \cdot (10^7) \log_2 10^7 \text{instruções}}{10^7 \text{instruções/segundo}} \approx 1.163 \text{ segundos}$$

(CORMEN et al., 2012).

Existem casos onde a eficiência é ainda mais importante como os problemas NP-Difíceis. Não se conhece algoritmo com função de complexidade polinomial para esses problemas. São exemplos:

- roteamento de veículos para entregas (*vehicle routing*);
- calcular o número mínimo de containers para transportar um conjunto de caixas com produtos (*bin-packing* 3D);
- calcular a localização e o número mínimo de antenas para garantir a cobertura de uma certa região geográfica (*facility location*).

Para contextualizar essa dificuldade, imagine usar o computador A e B discutidos acima para a ordenação acima sobre um algoritmo exato para uma versão mais genérica (mais simples) do roteamento de entregas para vários veículos conhecida como Problema do Caixeiro Viajante. Assuma o algoritmo de Held-Karp com complexidade $c_3 n^2 2^n$. Considere que $c_3 = 1$ e $n = 40$. Para o computador A (mais veloz) obtería-se

$$\frac{1 \cdot (40^2 \cdot 2^{40}) \text{ instruções}}{10^{10} \text{ instruções/segundo}} \approx 175.921,86 \text{ segundos } (48,86 \text{ horas})$$

. Para o computador B obteria-se:

$$\frac{1 \cdot (40^2 \cdot 2^{40}) \text{ instruções}}{10^{7} \text{instruções/segundo}} \approx 175.921.860,44 \text{ segundos } (5,57 \text{anos})$$

Desafio

Quais conclusões você chegaria nessa comparação?

E se fosse comprado um computador com o dobro do poder de processamento de A, qual o valor de n eu poderia executar no mesmo tempo da máquina A, considerando um algoritmo com função de complexidade 2^n?

Desafio

Responda a questão acima.

Por questões de facilidade, supõe-se que a máquina com o dobro do poder de processamente de A de computador C. Considere que y é o tamanho do problema no computador C, e x é o tamanho do problema no computador A. Se a máquina C é mais rápida, significa que ela executa o dobro das instruções que o computador A, então

$$\begin{aligned} 2^y &= 2 \cdot (2^x) \\ y &= \log_2 \left(2 \cdot (2^x)\right) \\ y &= \log_2(2) + \log_2(2^x) \\ y &= 1 + x \end{aligned} \tag{1.1}$$

Desafio

Quais conclusões você chegaria nessa comparação?

Desafio

Considerando um computador que executa 4×10^9 instruções/segundo, calcule o tamanho do problema que pode ser resolvido para cada um dos tempos abaixo:

	1 seg	**1 minuto**	**1 hora**	**1 dia**	**1 mês**	**1 ano**	**1 século**
$\mathbf{\log_2 n}$							
$\mathbf{\sqrt{n}}$							
$\mathbf{n}$							
$\mathbf{n \log_2 n}$							
$\mathbf{n^2}$							
$\mathbf{n^3}$							
$\mathbf{2^n}$							
$\mathbf{3^n}$							

Desafio

Considerando um computador que executa 4×10^9 instruções/segundo, calcule o tempo necessário para executar cada tamanho de problema destacado:

	n = 20	**n = 40**	**n = 60**	**n = 80**	**n = 100**
$\mathbf{log_2 n}$					
$\mathbf{\sqrt{n}}$					
n					
$\mathbf{n log_2 n}$					
$\mathbf{n^2}$					
$\mathbf{n^3}$					
$\mathbf{2^n}$					
$\mathbf{3^n}$					

Desafio

Se for usado um computador ?x mais rápido que o computador atual, qual o tamanho do problema que poderia ser resolvido sobre o mesmo tempo computacional?

	Tamanho do Problema no Computador Atual	**Tamanho do Problema em Computador 100 × mais rápido**	**Tamanho do Problema em 1000 × mais rápido**
$\mathbf{log_2 n}$	N_1		
$\mathbf{\sqrt{n}}$	N_2		
$\mathbf{n}$	N_3		
$\mathbf{n log_2 n}$	N_4		
$\mathbf{n^2}$	N_5		
$\mathbf{n^3}$	N_6		
$\mathbf{2^n}$	N_7		
$\mathbf{3^n}$	N_8		

1.3 Projeto e Análise de Algoritmos

Essa seção tem o objetivo de introduzir o projeto e análise de algoritmos através de alguns exemplos. Nesse contexto, é interessante analisar:

- Finitude: o algoritmo pára?
- Corretude: o algoritmo faz o que promete?
- Complexidade de tempo: quantas intruções são necessárias no pior caso[1]?

[1] Além do pior caso, podem ser considerados o melhor e o caso médio em pesquisas mais sensíveis.

1.3.1 Ordenação por Inserção

Considere o seguinte problema de ordenação:

- Entrada: uma sequência de números $\langle a_1, a_2, \ldots, a_n \rangle$.
- Saída: uma permutação (reordenação) $\langle a'_1, a'_2, \ldots, a'_n \rangle$, tal que $a'_i \leq a'_{i+1}$ para todo $i \in \{1, 2, \ldots n\}$.

Um algoritmo para resolvê-lo pode ser visualizado no Algoritmo 1.

Algoritmo 1: Insertion-Sort

Input : um vetor $A = \langle a_1, a_2, \ldots, a_n \rangle$

```
1 for j ← 2 to |A| do
2     chave ← A_j
      // Insere A_j no subvetor ⟨a_1, a_2, ..., a_{j-1}⟩.
3     i ← j − 1
4     while i ≥ 1 and A_i > chave do
5         A_{i+1} ← A_i
6         i ← i − 1
7     A_{i+1} ← chave
```

1.3.1.1 Finitude

Primeiro, verifica-se se o algoritmo pára, ou seja, não fica executando indefinidamente. No laço da linha 4, o valor de i decrementa a cada iteração. O valor inicial de i é $j-1$, ou seja, $i \geq 1$. Sua execução deve parar pois a condição de repetição desse laço determina que $i \geq 1$. O laço da linha 1 também pára, pois repete um número finito de vezes, dado pela quantidade de elementos em $|A|$, que não se altera durante o processo. Portanto, o algoritmo pára.

1.3.1.2 Corretude

Para verificarmos a corretude do Insertion-sort, é necessário primeiro conhecer um conceito chamado de "Invariante de Laço". Um invariante de laço é uma propriedade

que relaciona as variáveis do algoritmo a cada execução completa de um laço de repetição. Desse modo, ao término do laço têm-se uma propriedade útil para apoiar a corretude do algoritmo.

A estratégia utilizada para demonstrar a corretude de um algoritmo iterativo através de invariantes de laço segue os seguintes passos, que são muito semelhantes aos passos adotados em uma prova por indução (CORMEN et al., 2012):

- **Inicialização**: demonstre que o invariante é verdadeiro antes da primeira iteração;
- **Manutenção**: se o invariante for verdadeiro antes de uma iteração do laço, demonstre que ele permanece verdadeiro antes da próxima iteração;
- **Término**: quando o laço termina, o invariante fornece uma propriedade útil que ajuda a demonstrar que o algoritmo é correto.

Analisando o Algoritmo 1, podemos definir o seguinte invariante de laço:

No começo de cada iteração do laço que compreende as linhas entre 1 e 7, o subvetor $\langle a_1, \ldots, a_{j-1} \rangle$ *está ordenado.*

Considerando esse invariante de laço, verifica-se:

- **Inicialização**: mostra-se que o invariante de laço é válido quando j=2. O subvetor $\langle a_1, \ldots, a_{j-1} \rangle$ está obviamente ordenado, pois tem apenas um elemento.
- **Manutenção**: deve-se checar se o invariante continua válido a cada iteração. A cada iteração do laço entre as linhas 1 e 7, desloca-se os valores presentes nas posições $j-1$, $j-2$, ... até que encontrar a posição adequada para a *chave* (valor da posição j no início do iteração). Então, o subvetor $\langle a_1, \ldots, a_j \rangle$ está ordenado. Ao incrementar j, mantem-se o invariante de laço. Uma maneira mais formal, deve-se considerar um invariante de laço para o laço entre as linhas 4 e 6.
- **Término**: examina-se o que ocorre quando o laço termina. A condição que provoca o término do laço é $j > |A|$. Cada iteração do laço incrementa o j em mais

uma unidade; desse modo, com o término do laço, $j = |A| + 1$. Então, na última iteração, j era igual a $|A|$ e têm-se a ordenação do vetor $\langle a_1, a_2, \ldots, a_{|A|} \rangle$.

Desafio

Demonstre que encontra-se a posição adequada para a chave (descrita na manutenção) considerando o trecho de código entre as linhas 4 e 6.

1.3.1.3 Complexidade de Tempo

Agora, o objetivo é identificar a função de complexidade que corresponde ao tempo computacional demandado pelo algoritmo Insertion-sort. A complexidade de tempo deve ser realizada contanto o número de instruções básicas (operações elementares ou primitivas) em relação a entrada de tamanho n.

Primeiramente, identifica-se o custo computacional de cada instrução. Na tabela abaixo, considera-se que cada instrução possui um custo c_k, no qual k é a linha da instrução.

Linha	Instrução	Custo	# Execuções
1	**for** $j \leftarrow 2$ **to** $\|A\|$ **do**	c_1	?
2	chave $\leftarrow A_j$	c_2	?
3	$i \leftarrow j - 1$	c_3	?
4	**while** $i \geq 1$ **and** $A_i >$ chave **do**	c_4	?
5	$A_{i+1} \leftarrow A_i$	c_5	?
6	$i \leftarrow i - 1$	c_6	?
7	$A_{i+1} \leftarrow$ chave	c_7	?

Considere que t_j é o número de vezes que o teste do laço *while* na linha 4 é executado para aquele valor de j. Considere também que o vetor possui n posições. Atualiza-se então a tabela com essas considerações.

Linha	**Instrução**	**Custo**	**# Execuções**
1	**for** $j \leftarrow 2$ **to** $\lvert A \rvert$ **do**	c_1	n
2	chave $\leftarrow A_j$	c_2	$n-1$
3	$i \leftarrow j-1$	c_3	$n-1$
4	**while** $i \geq 1$ **and** $A_i >$ chave **do**	c_4	$\sum_{j=2}^{n} t_j$
5	$A_{i+1} \leftarrow A_i$	c_5	$\sum_{j=2}^{n}(t_j-1)$
6	$i \leftarrow i-1$	c_6	$\sum_{j=2}^{n}(t_j-1)$
7	$A_{i+1} \leftarrow$ chave	c_7	$n-1$

O tempo de execução total é dado por

$$\begin{aligned} T(n) = & \\ & c_1 n + c_2(n-1) + c_3(n-1) + \\ & c_4 \sum_{j=2}^{n} t_j + c_5 \sum_{j=2}^{n} (t_j - 1) + c_6 \sum_{j=2}^{n} (t_j - 1) \\ & + c_7(n-1). \end{aligned} \tag{1.2}$$

Precisa-se identificar agora o valor de t_j. No melhor caso, $t_j = 1$. Ele ocorre quando o vetor já está ordenado. Desse modo, têm-se

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_3(n-1) + c_4(n-1) + c_7(n-1) \\ &= (c_1 + c_2 + c_3 + c_4 + c_7)n - (c_2 + c_3 + c_4 + c_7) \end{aligned} \tag{1.3}$$

. Essa é uma função linear de n.

No pior caso, o vetor está na ordem inversa. Nesse caso, $t_j = j$. Sabendo que

$$\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1 \tag{1.4}$$

e

$$\sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2} \tag{1.5}$$

, têm-se

$$
\begin{aligned}
T(n) = & \\
& c_1 n + c_2(n-1) + c_3(n-1) + c_7(n-1) \\
& + c_4\left(\frac{n(n+1)}{2} - 1\right) + c_5\left(\frac{n(n-1)}{2}\right) + c_6\left(\frac{n(n-1)}{2}\right) \\
= & \left(\frac{c_4}{2} + \frac{c_5}{2} + \frac{c_6}{2}\right)n^2 + \left(c_1 + c_2 + c_3 + \frac{c_4}{2} - \frac{c_5}{2} - \frac{c_6}{2} + c_7\right)n - (c_2 + c_3 + c_4 + c_7).
\end{aligned}
\tag{1.6}
$$

Obtêm-se então uma função quadrática em relação ao tamanho da entrada no pior caso. Diz-se que a complexidade assintótica do pior caso é $\Theta(n^2)$

1.3.1.4 Complexidade Assintótica

Para essa disciplina, nos interessa a complexidade assintótica dos algoritmos, ou seja, o comportamento mais significativo da função para instâncias de tamanho grande. Portanto, podemos "esquecer" o valor dessas constantes e nos concentrar no termo mais significativo da função. Veja o exemplo abaixo

n	$\mathbf{3n^2 + 10n + 50}$	$\mathbf{3n^2}$	**Diferença percentual**
64	12978	12288	5,32%
128	50482	49152	2,63%
512	791602	786432	0,65%
1024	3156018	3145728	0,33%
2048	12603442	12582912	0,16%
4096	50372658	50331648	0,08%
8192	201408562	201326592	0,04%
16384	805470248	805306368	0,02%
32768	3221553202	322122547	0,01%

1.3.2 Ordenação por Intercalação

Muitos algoritmos são recursivos em sua estrutura. Em geral, esses algoritmos utilizam de uma abordagem conhecida como de divisão e conquista, que será alvo de maior

investigação em aulas futuras. Esse tipo de algoritmo divide o problema em subproblemas semelhantes ao original e as resolvem recursivamente, combinando as soluções para criar uma solução para o problema original (CORMEN et al., 2012).

O paradigma de divisão e conquista envolve três passos e, cada nível de recursão (CORMEN et al., 2012):

- **Divisão**: divide o problema em subproblemas menores;
- **Conquista**: resolve os subproblemas recursivamente. Se os subproblemas forem de tamanho pequeno o suficiente, os resolve de maneira direta;
- **Combinação**: combina as soluções dos subproblemas para definir uma solução para o problema original.

A ordenação por intercalação aqui visitada utiliza os Algoritmos 2 e 3, que "implementam" o algoritmo de Merge-Sort. Eles obdedecem o paradigma de divisão e conquista:

- **Divisão**: divide a sequência de n elementos em duas subsequências com $\frac{n}{2}$ elementos aproximadamente;
- **Conquista**: ordena as duas subsequências recursivamente, utilizando ordenação por intercalação;
- **Combinação**: intercala as duas subsequências para obter a resposta ordenada.

Desafio

Qual a complexidade de tempo do procedimento Merge (Algoritmo 2)?

Algoritmo 2: Merge

Input : um vetor $A = \langle a_1, a_2, \ldots, a_n \rangle$, as posições p, q e r

```
1  n1 ← q − p + 1
2  n2 ← r − q
3  Seja L um vetor indexado de 1 até n1 + 1
4  Seja R um vetor indexado de 1 até n2 + 1
5  for i ← 1 to n1 do
6      L_i ← A_{p+i−1}
7  for j ← 1 to n2 do
8      R_j ← A_{q+j}
9  L_{n1+1} ← ∞
10 R_{n2+1} ← ∞
11 i ← 1
12 j ← 1
13 for k ← p to r do
14     if L_i ≤ R_j then
15         A_k ← L_i
16         i ← i + 1
17     else
18         A_k ← R_j
19         j ← j + 1
```

Algoritmo 3: Merge-Sort

Input : um vetor $A = \langle a_1, a_2, \ldots, a_n \rangle$, as posições p e r

```
1  if p < r then
2      q ← ⌊(p+r)/2⌋
3      Merge-Sort(A, p, q)
4      Merge-Sort(A, q + 1, r)
5      Merge(A, p, q, r)
```

O procedimento Merge (Algoritmo 2) utiliza tempo $\Theta(n)$ no qual $n = r - p + 1$.

O seguinte invariante de laço é destacado para o procedimento Merge:

No início de cada iteração do laço for entre as linhas 13 e 19, o subvetor $\langle a_p, \ldots, a_{k-1} \rangle$ *contém os* $k-p$ *menores elementos de L e R ordenados.* Para demonstrar que o invariante de laço é válido, o mesmo será analisado:

- **Inicialização**: na primeira iteração do laço $k = p$, portanto o subvetor $\langle a_p, \ldots, a_{k-1} \rangle$ de A é vazio. As primeiras posições de L e R possuem os menores valores dos subvetores aos quais receberam os valores;

- **Manutenção**: para a manutenção, suponha primeiro que $L_i \leq R_j$. Então, L_i é o menor elemento ainda não copiado para A. Esse vetor, no trecho $\langle a_p, \ldots, a_{k-1} \rangle$, conterá os $k-p+1$ menores elementos. O incremento de k e de i restabelece o invariante de laço para a próxima iteração. Se $L_i > R_j$, então o mesmo ocorre, mas considerando R_j como o menor elemento que ainda não foi recolocado em A;

- **Término**: Ao terminar, $k = r+1$. Pelo invariante de laço, o subvetor $\langle a_p, \ldots, a_{k-1} \rangle$, que nesse momento é $\langle a_p, \ldots, a_r \rangle$, contém os $r-p+1$ menores elementos de L e R ordenados. Todos os elementos, exceto os valores auxiliares ∞, foram copiados em ordem crescente para A.

1.3.2.1 Complexidade de Tempo

Quando se analisa um algoritmo recursivo, a função de complexidade pode ser descrita como uma recorrência (função que usa a si mesmo para definir seu contradomínio).

Para a análise do algoritmo de Merge-Sort é mais simples de ser compreendida se o tamanho do vetor for igual a uma potência de 2, mas esse algoritmo funciona quando a quantidade de elementos não segue essa ordem (CORMEN et al., 2012).

Analisando as etapas de um algoritmo de divisão e conquista no contexto do Merge-Sort, têm-se:

- **Divisão**: a etapa de divisão encontra a posição correspondente ao meio do subvetor. Tendo a complexidade $\Theta(1)$;

- **Conquista**: Na conquista, resolve-se recursivamente dois problemas, cada qual com tamanho $\frac{n}{2}$. Portanto, a complexidade da conquista é $2T\left(\frac{n}{2}\right)$.

- **Combinação**: A combinação das partes fica por conta do procedimento Merge, cuja complexidade é $\Theta(n)$.

A função de complexidade expressa através de uma recorrência pode ser visualizada a seguir:

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 1, \\ 2T\left(\frac{n}{2}\right) + \Theta(n) & \text{se } n > 1. \end{cases} \tag{1.7}$$

A Figura 1 exibe a árvore de reccursão da recorrência $T(n) = 2T\left(\frac{n}{2}\right) + cn$. O custo do primeiro nível é cn. O custo do segundo nível é $c\left(\frac{n}{2}\right) + c\left(\frac{n}{2}\right) = cn$. No terceiro nível, têm-se $c\left(\frac{n}{4}\right) + c\left(\frac{n}{4}\right) + c\left(\frac{n}{4}\right) + c\left(\frac{n}{4}\right) = cn$. Em geral, o nível i tem 2^i nodos, cada qual contribui com $c\left(\frac{n}{2^i}\right)$, sendo que cada nível demanda $2^i c\left(\frac{n}{2^i}\right) = cn$. O número total de níveis é igual a $log_2 n + 1$. Se os custos forem somados têm-se $cn\log_2 n + cn$.

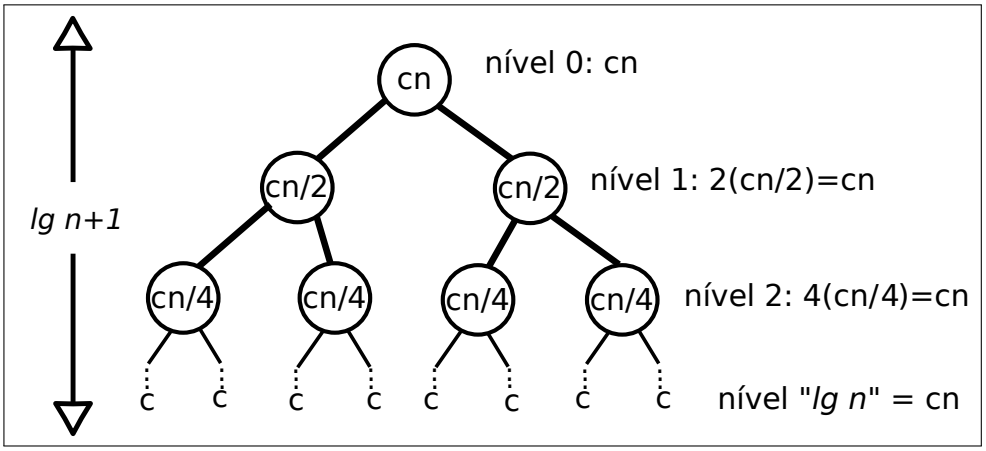


Figura 1 – Árvore de recursão para apoiar na resolução da recorrência $T(n) = 2T\left(\frac{n}{2}\right) + cn$.

CAPÍTULO 2

Notação Assintótica e Crescimento de Funções

Embora seja possível determinar o tempo computacional exato de um algoritmo, a precisão obtida não vale o esforço para instâncias suficientemente grandes. As constantes multiplicativas e os termos de ordem mais baixa ao determinar um tempo exato são dominados pelo termo de maior relevância (CORMEN et al., 2012).

"Sobre a entrada n, o algoritmo executa no máximo $1.62n^2+3.5n+8$ passos." (KLEINBERG; TARDOS, 2005). Este tipo de análise pode ser útil em alguns contextos, mas de modo geral não:

- Conseguir uma medida tão precisa pode ser exaustivo;
- O objetivo na complexidade de algoritmos é identificar classes de algoritmos que possuem comportamento similar;
- Precisa-se de uma medida menos detalhista.

2.1 Notação Assintótica

No capítulo anterior, a notação assintótica Θ já foi usada. O objetivo dessa seção é formalizar essa e outras notações assintóticas. Nesse contexto, serão ignoradas as constantes multiplicativas e os termos de menor ordem. São discutidas aqui as ordens assintóticas O, Ω, Θ, o e ω .

2.1.1 Notação Θ

Quando uma função $f(n)$ é encaixada entre os valores $c_2 g(n)$ e $c_1 g(n)$ para duas constantes c_1 e c_2 , diz-se que pertence ao grupo $\Theta\big(g(n)\big)$. Diz-se que $g(n)$ é um limite assintoticamente restrito para $f(n)$

$$\begin{aligned}\Theta\big(g(n)\big) = \big\{f(n) |\ &\text{existem constantes positivas } c_1 \text{ , } c_2 \text{ e } n_0 \\ &\text{tais que } 0 \leq c_2 g(n) \leq f(n) \leq c_1 g(n) \text{ para todo o } n \geq n_0\big\}\end{aligned} \tag{2.1}$$

A Figura 2 exibe a notação sendo utilizada para o caso $f(n) \in \Theta\big(g(n)\big)$.

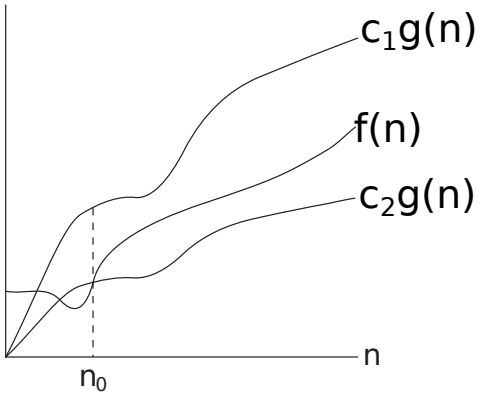


Figura 2 – Exemplo da aplicação da notação assintótica Θ. Nela, $f(n) \in \Theta\big(g(n)\big)$ (CORMEN et al., 2012).

Por força de representação, é comum considerar que $f(n) = \Theta\big(g(n)\big)$, embora o correto seja $f(n) \in \Theta\big(g(n)\big)$.

Exemplos nos quais $f(n) \in \Theta\big(g(n)\big)$:

- $f(n) = \frac{1}{2}n^2 - 3n$ e $g(n) = n^2$;

- $f(n) = 100n^3 - n^2 + 3.5n - 17$ e $g(n) = n^3$.

2.1.2 Notação O

Quando uma função $f(n)$ possui como limite assintótico superior $g(n)$ diz-se que $f(n)$ pertence ao grupo $O\big(g(n)\big)$. Diz-se que $g(n)$ é o limite assintótico superior para $f(n)$.

$$O\big(g(n)\big) = \big\{f(n) | \text{ existem constantes positivas } c \text{ e } n_0 \text{ tais que } 0 \leq f(n) \leq cg(n) \text{ para todo o } n \geq n_0\big\} \tag{2.2}$$

A Figura 3 exibe a notação sendo utilizada para o caso $f(n) \in O\big(g(n)\big)$.

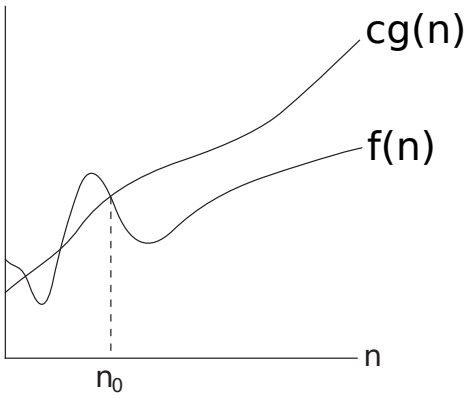


Figura 3 – Exemplo da aplicação da notação assintótica O. Nela, $f(n) \in O\big(g(n)\big)$ (CORMEN et al., 2012).

É importante ter em mente que se $f(n) \in \Theta\big(g(n)\big)$, então $f(n) \in O\big(g(n)\big)$. Isso se deve ao fato de que a notação Θ é uma noção mais forte de O.

A notação O é empregada para informar o pior caso da complexidade de um algoritmo. Desse modo, quando se diz que "a complexidade é $O(n^2)$" que o algoritmo demandará tempo de pior caso cn^2 para uma entrada de tamanho n.

Exemplos nos quais $f(n) \in O\big(g(n)\big)$:

- $f(n) = \frac{1}{4}n^2 - n$ e $g(n) = n^2 - 6n$;
- $f(n) = 100n^3 - n^2 + 3.5n - 17$ e $g(n) = 2^n + 3n^2$.

2.1.3 Notação Ω

Quando uma função $f(n)$ possui como limite assintótico inferior $g(n)$ diz-se que $f(n)$ pertence ao grupo $\Omega\big(g(n)\big)$.

$$\Omega\big(g(n)\big) = \big\{f(n) | \text{ existem constantes positivas } c \text{ e } n_0 \text{ tais que } 0 \leq cg(n) \leq f(n) \text{ para todo o } n \geq n_0\big\} \tag{2.3}$$

A Figura 4 exibe a notação sendo utilizada para o caso $f(n) \in \Omega\big(g(n)\big)$.

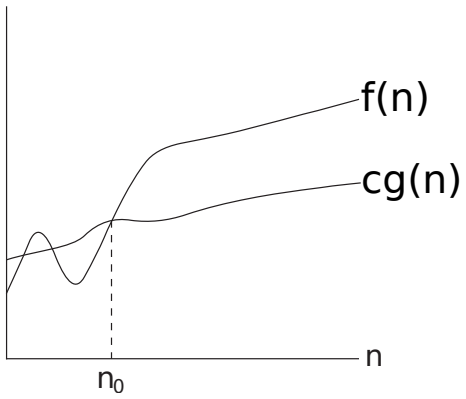


Figura 4 – Exemplo da aplicação da notação assintótica Ω. Nela, $f(n) \in \Omega\big(g(n)\big)$ (CORMEN et al., 2012).

Teorema 2.1.1. *Para quaisquer duas funções $f(n)$ e $g(n)$, $f(n) \in \Theta\big(g(n)\big)$ sse $f(n) \in O\big(g(n)\big)$ e $f(n) \in \Omega\big(g(n)\big)$.*

Exemplos nos quais $f(n) \in \Omega\big(g(n)\big)$:

- $f(n) = \frac{1}{2}n^2 - 3n$ e $g(n) = \frac{1}{2}n^2 - 2n$;
- $f(n) = 2^n + 3n^2$ e $g(n) = 100n^3 - n^2 + 3.5n - 17$.

2.1.4 Notação *o*

A notação *o* é utilizada quando o limite superior assintótico não é "justo" (CORMEN et al., 2012). Considera-se $2n^2 \in O(n^2)$ justo e $2n \in O(n^2)$ não justo. Um limite superior

assintótico não justo utiliza a notação $o\big(g(n)\big)$.

$$o\big(g(n)\big) = \big\{f(n) | \text{ para qualquer constante positiva } c, \text{ existe uma constante } n_0 > 0 \text{ tal que } 0 \leq f(n) < cg(n) \text{ para todo o } n \geq n_0\big\}. \tag{2.4}$$

Desse modo, $2n^2 \notin o\big(n^2\big)$, mas $2n \in o\big(n^2\big)$.

Exemplos nos quais $f(n) \in o\big(g(n)\big)$:

- $f(n) = 100n^3 - n^2 + 3.5n - 17$ e $g(n) = 2^n + 3n^2$;
- $f(n) = 100n^2$ e $g(n) = n^3$.

2.1.5 Notação ω

De acordo com Cormen et al. (2012), utiliza-se a notação ω para denotar um limite inferior assintótico que não é preciso. $f(n) \in \omega\big(g(n)\big)$ sse $g(n) \in o\big(f(n)\big)$.

$$\omega\big(g(n)\big) = \big\{f(n) | \text{ para qualquer constante positiva } c, \text{ existe uma constante } n_0 > 0 \text{ tal que } 0 \leq cg(n) < f(n) \text{ para todo o } n \geq n_0\big\}. \tag{2.5}$$

Exemplos nos quais $f(n) \in \omega\big(g(n)\big)$:

- $f(n) = 2^n + 3n^2$ e $g(n) = 100n^3 - n^2 + 3.5n - 17$;
- $f(n) = \frac{1}{1000}n^2$ e $g(n) = n$.

2.1.6 Propriedades das Notações

Transitividade (CORMEN et al., 2012):

- $f(n) \in \Theta\big(g(n)\big)$ e $g(n) \in \Theta\big(h(n)\big)$ implicam $f(n) \in \Theta\big(h(n)\big)$;

- $f(n) \in O\big(g(n)\big)$ e $g(n) \in O\big(h(n)\big)$ implicam $f(n) \in O\big(h(n)\big)$;
- $f(n) \in \Omega\big(g(n)\big)$ e $g(n) \in \Omega\big(h(n)\big)$ implicam $f(n) \in \Omega\big(h(n)\big)$;
- $f(n) \in o\big(g(n)\big)$ e $g(n) \in o\big(h(n)\big)$ implicam $f(n) \in o\big(h(n)\big)$;
- $f(n) \in \omega\big(g(n)\big)$ e $g(n) \in \omega\big(h(n)\big)$ implicam $f(n) \in \omega\big(h(n)\big)$.

Reflexidade:

- $f(n) \in \Theta\big(f(n)\big)$;
- $f(n) \in O\big(f(n)\big)$;
- $f(n) \in \Omega\big(f(n)\big)$.

Simetria: $f(n) \in \Theta\big(g(n)\big)$ sse $g(n) \in \Theta\big(f(n)\big)$.
Simetria de Transposição:

- $f(n) \in O\big(g(n)\big)$ sse $g(n) \in \Omega\big(f(n)\big)$;
- $f(n) \in o\big(g(n)\big)$ sse $g(n) \in \omega\big(f(n)\big)$.

Pode-se utilizar uma analogia dessa relação entre funções com as comparações entre os números reais a e b:

- $f(n) \in O\big(g(n)\big)$ é como $a \leq b$;
- $f(n) \in \Omega\big(g(n)\big)$ é como $a \geq b$;
- $f(n) \in \Theta\big(g(n)\big)$ é como $a = b$;
- $f(n) \in o\big(g(n)\big)$ é como $a < b$;
- $f(n) \in \omega\big(g(n)\big)$ é como $a > b$.

Multiplicação por uma constante:

- $\Theta(cf(n)) = \Theta(f(n))$;

- $99n^2 \in \Theta(n^2)$.

Mais alto expoente de um polinômio:

- $3n^3 - 5n^2 + 100 \in \Theta(n^3)$;
- $6n^4 - 20n^2 \in \Theta(n^4)$;
- $0.8n + 224 \in \Theta(n)$.

Termo dominante:

- $2^n + 6n^3 \in \Theta(2^n)$;
- $n! - 3n^2 \in \Theta(n!)$;
- $nlogn + 3n^2 \in \Theta(n^2)$.

CAPÍTULO 3

Recorrências

Relações de recorrências expressam a complexidade de algoritmos recursivos. É preciso resolver a recorrência para determinar a complexidade através da chamada fórmula fechada (que não depende da mesma ou de outra função).

Aqui, destacam-se quatro métodos para resolver recorrências:

- Método da substituição;
- Método da iteração;
- Método da árvore de recursão ou recorrência;
- Método mestre ou teorema mestre.

Para apoiar a explicação de alguns desses métodos, será utilizado o algoritmo de

Merge-sort, já visitado anteriormente, mas que é revisitado aqui nos Algoritmos 4 e 5.

Algoritmo 4: Merge

Input : um vetor $A = \langle a_1, a_2, \ldots, a_n \rangle$, as posições p, q e r

```
1  n_1 ← q − p + 1
2  n_2 ← r − q
3  Seja L um vetor indexado de 1 até n_1 + 1
4  Seja R um vetor indexado de 1 até n_2 + 1
5  for i ← 1 to n_1 do
6      L_i ← A_{p+i−1}
7  for j ← 1 to n_2 do
8      R_j ← A_{q+j}
9  L_{n_1+1} ← ∞
10 R_{n_2+1} ← ∞
11 i ← 1
12 j ← 1
13 for k ← p to r do
14     if L_i ≤ R_j then
15         A_k ← L_i
16         i ← i + 1
17     else
18         A_k ← R_j
19         j ← j + 1
```

Algoritmo 5: Merge-Sort

Input : um vetor $A = \langle a_1, a_2, \ldots, a_n \rangle$, as posições p e r

```
1  if p < r then
2      q ← ⌊(p+r)/2⌋
3      Merge-Sort(A, p, q)
4      Merge-Sort(A, q + 1, r)
5      Merge(A, p, q, r)
```

Para o contexto do Merge-Sort, sabe-se que a recorrência que representa sua complexidade de tempo é:

$$T(n) = \begin{cases} 1 & \text{se } n = 1, \\ 2T\left(\frac{n}{2}\right) + n & \text{se } n > 1. \end{cases} \tag{3.1}$$

Antes de começar a resolver as recorrências, é importante visitar o conceito de indução matemática.

3.1 Indução Matemática

Na demonstração por indução, procura-se testar a validade de uma propriedade P com um parâmetro n. Diz-se que realiza-se a demonstração de $P(n)$ por indução.

Geralmente, há um número infinito de casos a serem considerados, um para cada valor de parâmetro. Para isso, demonstra-se os casos infinitos de uma só vez, considerando:

- **Base da Indução**: demonstração de $P(1)$;
- **Hipótese de Indução**: supõe-se que $P(n)$ é verdadeiro;
- **Passo de Indução**: prova-se que $P(n+1)$ é verdadeiro, a partir da hipótese de indução.

Desafio

Prove que a soma dos n primeiros números ímpares é n^2 por indução.

Em alguns casos, é necessário demonstrar que uma proposição $P(n)$ vale para $n \geq n_0$ para algum n_0. Nesse caso, utiliza-se:

- **Base da Indução**: demonstração de $P(n_0)$;
- **Hipótese de Indução**: supõe-se que $P(n-1)$ é verdadeiro;

- **Passo de Indução**: prova-se que $P(n)$ é verdadeiro, a partir da hipótese de indução.

Uma indução pode ser fraca (simples) ou forte. Para a indução forte, deve-se supor que a propriedade vale para todos os casos anteriores e não somente para o anterior. Na indução forte, têm-se:

- **Base da Indução**: demonstração de $P(1)$;
- **Hipótese de Indução**: supõe-se que $P(k)$ é verdadeiro para todo $1 \leq k < n$;
- **Passo de Indução**: prova-se que $P(n)$ é verdadeiro, a partir da hipótese de indução.

Exemplo 1

Demonstrando que a seguinte inequação vale para todo natural n e real x tal que $(1+x) > 0$:

$$(1+x)^n \geq 1 + nx. \tag{3.2}$$

A base da indução é $n = 1$. Nesse caso, ambos os lados da inequação são iguais, mostrando sua validade. Isso encerra a prova para o caso base.

A hipótese de indução é a de que a inequação $(1+x)^n \geq 1 + nx$ valha para n em qualquer valor real de x, desde que $(1+x) > 0$.

O passo da indução deve mostrar que a inequação vale supondo a hipótese da indução usando o valor $n+1$, isto é, $(1+x)^{n+1} \geq 1 + (n+1)x$ para todo x, desde que $(1+x) > 0$. A dedução é simples:

$$\begin{aligned}
(1+x)^{n+1} &= (1+x)^n(1+x) \\
&\geq (1+xn)(1+x) \text{ pela hipótese da indução e } (1+x) > 0 \\
&= 1 + (n+1)x + nx^2 \\
&\geq 1 + (n+1)x \text{ já que } nx^2 \geq 0.
\end{aligned} \tag{3.3}$$

A última linha demonstra que a inequação vale para $n+1$, completando a prova.

Exemplo 2

Demonstra-se que o número de regiões criadas por n retas *em posição geral* no plano é igual a

$$T_n = \frac{n(n+1)}{2} + 1. \tag{3.4}$$

Um conjunto de retas está *em posição geral* se todas elas são concorrentes (não há retas paralelas) e não há três retas interceptando o mesmo ponto. A Figura 5 apóia essa explicação.

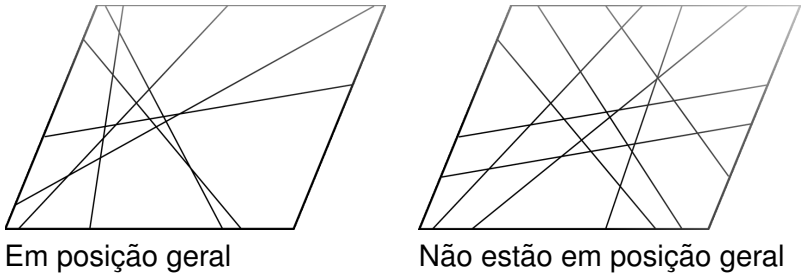


Figura 5 – Exemplos de retas que estão em posição geral e não estão.

Para a base da indução, têm-se $n = 1$. Uma reta sozinha divide o plano em duas regiões. Utilizando a equação acima, têm-se

$$T_1 = \frac{1(1+1)}{2} + 1 = 2. \tag{3.5}$$

Isso conclui a prova para o passo base.

A hipótese da indução supõe que $T_n = \frac{n(n+1)}{2} + 1$ para n retas *em posição geral*.

Para o passo de indução, demonstra-se que para $n+1$ retas *em posição geral* vale,

$$T_{n+1} = \frac{(n+1)(n+2)}{2} + 1. \tag{3.6}$$

supondo a hipótese de indução.

Considere o conjunto L com $n+1$ retas *em posição geral* no plano e seja r uma dessas retas. Então, as retas do conjunto $L' = L \backslash \{r\}$ obedecem a hipótese de indução. Além

disso, r intersecta as outras n retas retas em n pontos distintos. O que significa que, saindo de uma ponta de r no infinito e após cruzar n retas de L', a reta r terá cruzado $n+1$ regiões, dividindo cada uma destas em outras duas. Assim, pode-se escrever que:

$$\begin{aligned} T_{n+1} &= T_n + n + 1 \\ &= \frac{n(n+1)}{2} + 1 + n + 1 \text{ pela hipótese de indução} \\ &= \frac{(n+1)(n+1)}{2} + 1. \end{aligned} \tag{3.7}$$

Isso conclui a demonstração.

Exemplo 3

Em um conjunto de n retas no plano, define-se de regiões convexas cujas bordas são segmentos de n retas. Duas dessas regiões são adjacentes se suas bordas intersectam em segmento de reta não trivial, isto é contendo mais que um ponto.

Uma k-coloração dessas regiões é uma atribuição de k cores a cada uma das regiões de forma que regiões adjacentes tenham cores distintas.

Suponha que deseja-se demonstrar que existe uma 2-coloração das regiões formadas por n retas no plano para todo $n \geq 1$. Para demonstração via indução matemática, considere:

- **Base da indução**: Para $n = 1$, considera-se apenas uma reta que divide o plano em duas regiões. Atribuindo-se cores diferentes em cada região, obtém-se a propriedade desejada (2-coloração e adjacências com cores distintas). Isto conclui a prova para $n = 1$.

- **Hipótese de indução**: existe uma 2-coloração das regiões convexas formadas por n retas em um plano.

- **Passo da indução**: supondo a hipótese de indução, exibe-se uma 2-coloração para as regiões formadas por $n+1$ retas no plano. A demonstração do passo consiste em observar que a adição de uma nova reta r divide cada região atravessada por

r em duas, e definir a nova 2-coloração da seguinte forma: as regiões em um lado de r mantém a cor herdada da hipótese de indução; as regiões no outro lado de r têm suas cores trocadas. **(continuar ...)**

Desafio

Você é capaz de demonstrar que a 2-coloração obtida nesse processo obedece à definição?

Desafio

Prove que a soma dos n termos de uma Progressão Aritmética é igual a $\sum_{i=0}^{n-1}(a_0 + ir)$.

Desafio

Prove que a soma dos n termos de uma Progressão Geométrica é igual a $\sum_{i=0}^{n-1}(a_0 q^i)$.

3.2 Método da Substituição

Para o método da substituição, arrisca-se um palpite sobre a solução e tenta provar-se que ele funciona utilizando o método de indução. Requer prática e experiência.

Sabendo da recorrência do Merge-sort, deduz-se que $T(n) \in O(n \lg n)$; mais especificamente, que $T(n) \leq 3n \lg n$.

Substituindo, têm-se:

$$\begin{aligned} T(n) &= T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + n \\ &\leq 3\frac{n}{2}\lg\frac{n}{2} + 3\frac{n}{2}\lg\frac{n}{2} + n \\ &= 3n\lg\frac{n}{2} + n \\ &= 3n\big(\lg n - \lg 2\big) + n \\ &= 3n\lg n - 3n + n \\ &\leq 3n\lg n \end{aligned} \tag{3.8}$$

A substituição ocorreu perfeitamente. No entanto, $T(1) = 1$ e $3(1)\lg(1) = 0$ e a base não funciona. Contudo, precisamos lembrar das definições do $O(\cdot)$. Portanto, só precisa-se provar que $T(n) \leq 3n\lg n$ para um $n \geq n_0$. Nesse caso, $n_0 = 2$, pois

$$T(2) = T(1) + T(1) + 2 = 4 \leq 3(2\lg 2). \tag{3.9}$$

Portanto, conclui-se a substituição.

Definindo a constante

De onde vem a constante 3? E se não a tivermos, como encontrá-la? Uma forma simples é substituindo a constante por c:

$$\begin{aligned} T(n) &= T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + n \\ &\leq c\frac{n}{2}\lg\frac{n}{2} + c\frac{n}{2}\lg\frac{n}{2} + n \\ &= cn\lg\frac{n}{2} + n \\ &= cn\big(\lg n - \lg 2\big) + n \\ &= cn\lg n - cn + n \\ &\leq cn\lg n. \end{aligned} \tag{3.10}$$

Para que $cn\lg n - cn + n \leq cn\lg n$, considerando $c \geq 1$ já seria suficiente.

Desse modo, então chega-se a conclusão que $T(n) \leq cn\lg n$ e portanto $T(n) \in O(n\lg n)$.

Desafio

Seria melhor mostrar que $T(n) \in \Theta(n\lg n)$. Para isso, mostre que $T(n) \in \Omega(n\lg n)$.

Um outro exemplo

Considere a recorrência

$$T(n) = \begin{cases} 1 & \text{se } n = 1, \\ T\left(\lceil \frac{n}{2} \rceil\right) + T\left(\lfloor \frac{n}{2} \rfloor\right) + 1 & \text{se } n > 1. \end{cases} \tag{3.11}$$

Assumindo que $T(n) \in O(n)$, tenta-se demonstrar que $T(n) \leq cn$ para alguma constante c:

$$\begin{aligned} T(n) &= T\left(\left\lceil \frac{n}{2} \right\rceil\right) + T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + 1 \\ &\leq 2c\frac{n}{2} + 1 \\ &= cn + 1. \end{aligned} \tag{3.12}$$

Veja que não está correto considerar $cn + 1 \leq cn$. Para facilitar esse tipo de ajuste detalhado, uma ideia interessante é usar uma constante $b > 0$. Assim a substituição poderia acontecer considerando $T(n) \leq cn - b$. Desse modo, teria-se:

$$\begin{aligned} T(n) &= T\left(\left\lceil \frac{n}{2} \right\rceil\right) + T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + 1 \\ &\leq 2c\frac{n}{2} - 2b + 1 \\ &= cn - 2b + 1 \\ &\leq cn + 1. \end{aligned} \tag{3.13}$$

Para qualquer $b \geq 1$, a desigualdade vale.

3.3 Método da Iteração

O método iterativo (ou expansão telescópica) é interessante, pois não há a necessidade de "advinhar" o resultado. No entanto, é necessário utilizar-se de mais fundamentos matemáticos. A ideia principal é expandir a recorrência e escrevê-la como uma somatória de termos que dependem de n e das condições iniciais. Para isso, deve-se conhecer limitantes para várias somatórias.

Deve-se expandir até que seja identificado seu comportamento no caso geral. Para isso, as seguintes etapas são úteis:

1. Copie a fórmula original;

2. Descubra o passo;

3. Isole as equações para os próximos níveis;
4. Substitua os valores isolados na fórmula original;
5. Identifique a fórmula do i-ésimo nível;
6. Descubre o valor de i para igualar o parâmetro de $T(x)$ (valor de n) ao parâmetro no caso base;
7. Substitua o valor de i na fórmula do i-ésimo nível;
8. Identifique a complexidade dessa fórmula;
9. Prove por indução que a equação foi corretamente encontrada.

Exemplo 1

Seguindo as etapas para

$$T(n) = \begin{cases} 1 & \text{se } n = 1, \\ 2T\left(\frac{n}{2}\right) + 2 & \text{se } n > 1. \end{cases} \tag{3.14}$$

1. Copie a fórmula original: $2T\left(\frac{n}{2}\right) + 2$;
2. Descubra o passo: $T(n)$ está escrito em função de $T\left(\frac{n}{2}\right)$;
3. Isole as equações para os próximos níveis:
   $T\left(\frac{n}{2}\right) = 2T\left(\frac{n}{4}\right) + 2$ e
   $T\left(\frac{n}{4}\right) = 2T\left(\frac{n}{8}\right) + 2$;
4. Substitua os valores isolados na fórmula original: substituindo o valor isolado de $T\left(\frac{n}{2}\right)$:
   $T(n) = 2\left(2T\left(\frac{n}{4}\right) + 2\right) + 2$
   $T(n) = 2^2 T\left(\frac{n}{2^2}\right) + 2^2 + 2$, e
   substituindo o valor isolado de $T\left(\frac{n}{4}\right)$:

$T(n) = 2^2 T\left(\frac{n}{4}\right) + 2^2 + 2$

$T(n) = 2^2\left(2T\left(\frac{n}{8}\right) + 2\right) + 2^2 + 2$

$T(n) = 2^3 T\left(\frac{n}{2^3}\right) + 2^3 + 2^2 + 2$

$T(n) = 2^3 T\left(\frac{n}{2^3}\right) + 2^{i+1} - 2$ (veja notas [1]);

5. Identifique a fórmula do i-ésimo nível:

   $T(n) = 2^i T\left(\frac{n}{2^i}\right) + 2^{i+1} - 2$;

6. Descubre o valor de i para igualar o parâmetro de $T(x)$ (valor de n) ao parâmetro no caso base:

   $T\left(\frac{n}{2^i}\right) \equiv T(1)$

   $\frac{n}{2^i} = 1$

   $n = 2^i$

   $i = \lg n$;

7. Substitua o valor de i na fórmula do i-ésimo nível: $T(n) = 2^{\lg n} T(1) + 2^{\lg n+1} - 2$

   $T(n) = n + 2n - 2$

   $T(n) = 3n - 2$;

8. Identifique a complexidade dessa fórmula:

   $T(n) \in \Theta(n)$;

9. Prove por indução que a equação foi corretamente encontrada:

   - **Passo base:** para $n = 1$, o resultado esperado é 1, o que se confirma: $T(n) = 3n - 2$

     $T(1) = 3 - 2 = 1$;

   - **Passo indutivo:** por hipótese de indução, assume-se que está correta para $\frac{n}{2}$, ou seja $T\left(\frac{n}{2}\right) = 3\frac{n}{2} - 2$. Então, verifica-se se $T(n) = 3n - 2$, sabendo que $T(n) = 2T\left(\frac{n}{2}\right) + 2$ e partindo da hipótese de indução que $T(\frac{n}{2}) = 3\frac{n}{2} - 2$, o

[1] $\sum_{j=1}^{i} 2^j = \sum_{j=0}^{i-1} 2^j - 1 + 2^i = \frac{1-2^i}{1-2} - 1 + 2^i = 2^{i+1} - 2$

também se confirma:

$T(n) = 2T\left(\frac{n}{2}\right) + 2$

$T(n) = 2\left(3\frac{n}{2} - 2\right) + 2$

$T(n) = 6\frac{n}{2} - 4 + 2$

$T(n) = 3n - 2;$

- Demonstra-se então que $2T\left(\frac{n}{2}\right) + 2 = 3n - 2$ para $n \geq 1$.

Exemplo 2

Seguindo as etapas para

$$T(n) = \begin{cases} 1 & \text{se } n = 1, \\ 2T(n-1) + 1 & \text{se } n > 1. \end{cases} \tag{3.15}$$

1. Copie a fórmula original: $2T(n-1) + 1$;

2. Descubra o passo: $T(n)$ está escrito em função de $T(n-1)$;

3. Isole as equações para os próximos níveis:

   $T(n-1) = 2T(n-2) + 1$ e

   $T(n-2) = 2T(n-3) + 1$;

4. Substitua os valores isolados na fórmula original: substituindo o valor isolado de $T(n-1)$:

   $T(n) = 2\big(2T(n-2) + 1\big) + 1$

   substituindo o valor isolado de $T(n-2)$:

   $T(n) = 2^2 T(n-2) + 2 + 1$

   $T(n) = 2^2\big(2T(n-3) + 1\big) + 2 + 1$

   $T(n) = 2^3 T(n-3) + 2^2 + 2 + 1$

   $T(n) = 2^3 T(n-3) + 2^3 - 1$

5. Identifique a fórmula do i-ésimo nível:

   $T(n) = 2^i T(n-i) + 2^i - 1$;

6. Descubre o valor de i para igualar o parâmetro de $T(x)$ (valor de n) ao parâmetro no caso base:

   $T(n) = 2^i T(n-i) + 2^i - 1$; $n - i = 1$

   $i = n - 1$;

7. Substitua o valor de i na fórmula do i-ésimo nível: $T(n) = 2^{n-1} T(1) + 2^{n-1} - 1$

   $T(n) = 2^{n-1} + 2^{n-1} - 1$

   $T(n) = 2 \cdot 2^{n-1} - 1$

   $T(n) = 2^n - 1$;

8. Identifique a complexidade dessa fórmula:

   $T(n) \in \Theta(2^n)$;

9. Prove por indução que a equação foi corretamente encontrada:

   - **Passo base:** para $n = 1$, o resultado esperado é 1, o que se confirma: $T(n) = 2^n - 1$;

     $T(1) = 2^1 - 1 = 1$;

   - **Passo indutivo:** por hipótese de indução, assume-se que está correta para $n-1$, ou seja $T(n-1) = 2^{n-1} - 1$. Então, verifica-se se $T(n) = 2^n - 1$, sabendo que $T(n) = 2^n - 1$ e partindo da hipótese de indução que $T(n-1) = 2^{n-1} - 1$, o também se confirma:

     $T(n) = 2T(n-1) + 1$

     $T(n) = 2\left(2^{n-1} - 1\right) + 1$

     $T(n) = 2^n - 2 + 1$

     $T(n) = 2^n - 1$;

   - Demonstra-se então que $2T(n-1) + 1 = 2^n - 1$ para $n \geq 1$.

3.4 Método da Árvore de Recursão

O método da árvore de recursão converte a recorrência em uma árvore nos quais os nodos representam os custos de um único subproblema. Analisa-se então os custos em cada nível da árvore gerada. Usa-se algumas técnicas para limitar o tamanho da árvore.

De acordo com Cormen et al. (2012), uma árvore de recursão é bem mais usada para gerar um bom palpite, que é então verificado pelo método da substituição. Se a árvore for desenhada detalhadamente e criteriosamente, a mesma pode ser utilizada como prova direta de uma solução de recorrência.

Como exemplo da aplicação de uma árvore de recursão, considera-se a recorrência $T(n) = 3T\left(\frac{n}{4}\right) + cn^2$. A Figura 6 demonstra a execução do passo-a-passo para expandir uma árvore de recursão para a recorrência exemplo. Nota-se que o tamanho do problema para cada nodo da profundidade i é de $\frac{n}{4^i}$. Considerando o subproblema mínimo igual a 1, então o número de níveis pode ser calculado imaginando o maior nível para n:

$$\begin{aligned} \frac{n}{4^i} &= 1 \\ n &= 4^i \\ i &= \log_4 n. \end{aligned} \tag{3.16}$$

Considerando as profundidades $0, 1, \ldots, \log_4 n$, têm-se $\log_4 n + 1$ níveis. Observando a

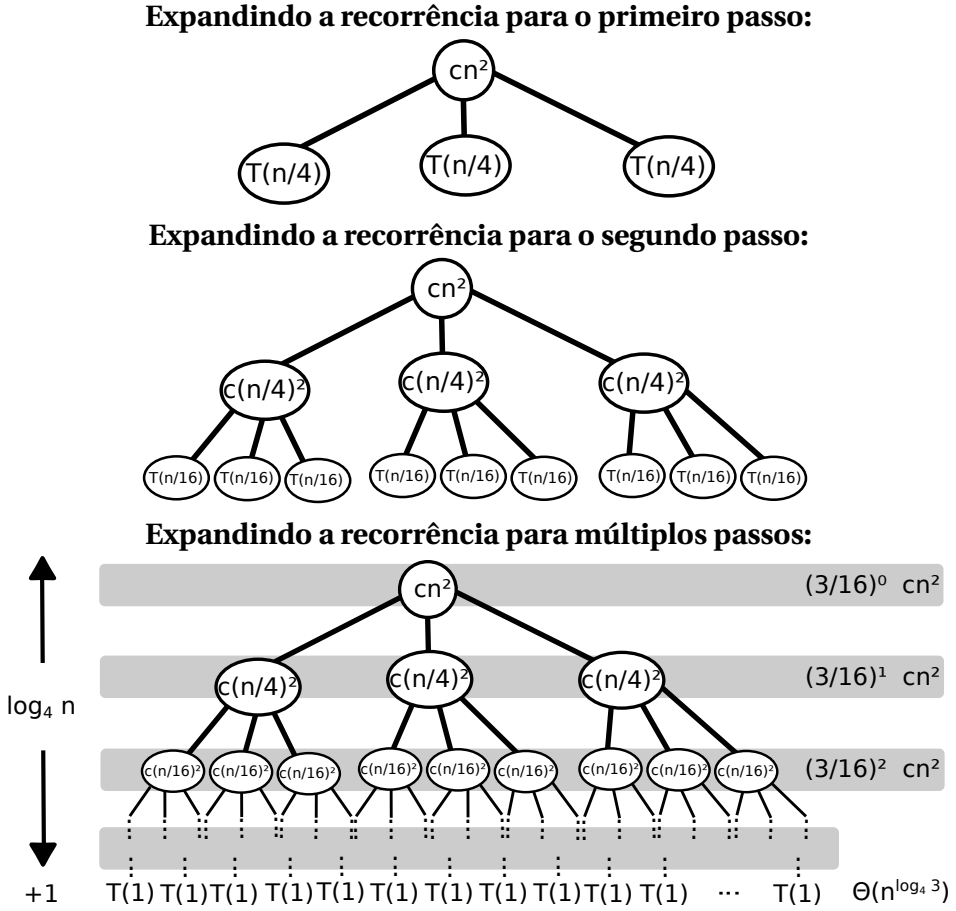


Figura 6 – Árvore de recursão sobre a recorrência $T(n) = 3T\left(\frac{n}{4}\right) + cn^2$. Adaptado de Cormen et al. (2012).

complexidade requerida em cada nível, têm-se uma nova definição para $T(n)$:

$$
\begin{aligned}
T(n) &= \left(\frac{3}{16}\right)^0 cn^2 + \left(\frac{3}{16}\right)^1 cn^2 + \left(\frac{3}{16}\right)^2 cn^2 + \ldots + \left(\frac{3}{16}\right)^{\log_4 n-1} cn^2 + \Theta\left(n^{log_4 3}\right) \\
&= \sum_{i=0}^{\log_4 n-1} \left(\frac{3}{16}\right)^i cn^2 + \Theta\left(n^{log_4 3}\right) \\
&< \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i cn^2 + \Theta\left(n^{log_4 3}\right) \\
&= \frac{1}{1-\left(\frac{3}{16}\right)} cn^2 + \Theta\left(n^{log_4 3}\right) \\
&= \frac{16}{13} cn^2 + \Theta\left(n^{log_4 3}\right) \\
&= O(n^2)
\end{aligned}
\tag{3.17}
$$

Usando o método de substituição para verificar o palpite de que $T(n) \in O(n^2)$,

deseja-se demonstrar que o mesmo é um limitante superior para $T(n) = 3T\left(\frac{n}{4}\right) + \Theta(n^2)$. Em outras palavras, demonstrar que $T(n) \leq dn^2$ para uma constante $d > 0$:

$$\begin{aligned} T(n) &\leq 3T\left(\frac{n}{4}\right) + cn^2 \\ &\leq 3d\left(\frac{n}{4}\right)^2 + cn^2 \\ &= \frac{3}{16}dn^2 + cn^2 \\ &\leq dn^2. \end{aligned} \tag{3.18}$$

A última parte é válida desde que $d \geq \frac{16}{13}c$.

3.5 Método da Mestre

O método mestre é utilizado para resolver recorrências na forma $T(n) = aT\left(\frac{n}{b}\right) + f(n)$, onde $a, b > 1$ e $f(n)$ é uma função assintoticamente positiva. Considera-se que $\left\lfloor \frac{n}{b} \right\rfloor = \left\lceil \frac{n}{b} \right\rceil = \frac{n}{b}$. Então, $T(n)$ tem os seguintes limites assintóticos:

1. Se $f(n) \in O(n^{log_b a - \epsilon})$, para uma constante $\epsilon > 0$, então $T(n) \in \Theta(n^{log_b a})$;
2. Se $f(n) \in \Theta(n^{log_b a})$, então $T(n) \in \Theta(n^{log_b a} \lg n)$;
3. Se $f(n) \in \Omega(n^{log_b a + \epsilon})$, para uma constante $\epsilon > 0$ e $af\left(\frac{n}{b}\right) \leq cf(n)$ para uma constante $c < 1$, então $T(n) \in \Theta\big(f(n)\big)$.

Em cada um dos três casos, compara-se a função $f(n)$ com a função $n^{\log_b a}$. A maior entre as duas funções estabelece a solução para a recorrência. Há ainda detalhes técnicos importantes a considerar (CORMEN et al., 2012):

- Para o primeiro caso, $f(n)$ não só tem que ser menor que $n^{\log_b a}$, mas tem que ser polinomialmente menor. Ou seja, $f(n)$ deve ser assintoticamente menor que $n^{\log_b a}$ por um fator n^{ϵ} para uma constante $\epsilon > 0$;
- Para o terceiro caso, $f(n)$ não só tem que ser maior que $n^{\log_b a}$, mas tem que ser polinomialmente maior e satisfazer a condição "$af\left(\frac{n}{b}\right) \leq cf(n)$". Ou seja, $f(n)$

deve ser assintoticamente maior que $n^{\log_b a}$ por um fator n^{ϵ} para uma constante $\epsilon > 0$.

Exemplo 1

Considere a seguinte recorrência $T(n) = 9T\left(\frac{n}{3}\right) + n$. Em seu formato, considera-se:

- $a = 9$;
- $b = 3$;
- $f(n) = n$.

Portanto, $n^{\log_b a} = n^{\log_3 9} \in \Theta(n^2)$. Desse modo, $f(n) = n^{\log_b a - \epsilon}$ com $\epsilon = 1$, então aplica-se o primeiro caso, obtendo $T(n) \in \Theta(n^2)$.

Exemplo 2

Considere a seguinte recorrência $T(n) = T\left(\frac{2n}{3}\right) + 1$. Em seu formato, considera-se:

- $a = 1$;
- $b = \frac{3}{2}$;
- $f(n) = 1$.

Portanto, $n^{\log_b a} = n^{\log_{\frac{3}{2}} 1} = n^0 = 1$. Desse modo, $f(n) = n^{\log_b a}$, então aplica-se o segundo caso, obtendo $T(n) \in \Theta(\lg n)$.

Exemplo 3

Considere a seguinte recorrência $T(n) = 3T\left(\frac{n}{4}\right) + n\lg n$. Em seu formato, considera-se:

- $a = 3$;
- $b = 4$;

- $f(n) = n \lg n$.

Portanto, $n^{\log_b a} = n^{\log_4 3} = n^{0,793}$. Desse modo, $f(n) \in \Omega(n^{\log_4 3+\epsilon})$ com $\epsilon \approx 0,2$, então precisa-se verificar uma última condição para aplicar o terceiro caso: considerando $af\left(\frac{n}{b}\right) = 3\frac{n}{4}\lg\frac{n}{4}$ e $cf(n) = \frac{3}{4}n\lg n$, então para $af\left(\frac{n}{b}\right) \leq cf(n)$, $3\frac{n}{4}\lg\frac{n}{4} \leq \frac{3}{4}n\lg n$ para um $c = \frac{3}{4}$. Desse modo, aplica-se o caso 3, obtendo-se $T(n) \in \Theta(n \lg n)$.

Exemplo 4

Considere a seguinte recorrência $T(n) = 2T\left(\frac{n}{2}\right) + n\lg n$. Em seu formato, considera-se:

- $a = 2$;
- $b = 2$;
- $f(n) = n \lg n$.

Portanto, $n^{\log_b a} = n^{\log_2 2} = n$. Nesse caso, poderia-se enganar dizendo que $n \lg n$ é assintoticamente maior que $n^{\log_b a} = n$. O problema é que ela não é polinomialmente maior. A razão $\frac{f(n)}{n^{\log_b a}} = \frac{n\lg n}{n} = \lg n$ é assintoticamente menor que n^{ϵ}, o que faz a recorrência ficar entre os casos 2 e 3.

Exemplo 5

Considere a seguinte recorrência $T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$. Em seu formato, considera-se:

- $a = 2$;
- $b = 2$;
- $f(n) = n$.

Portanto, $n^{\log_b a} = n^{\log_2 2} = n$. Desse modo, $f(n) = n^{\log_b a}$, então aplica-se o segundo caso, obtendo $T(n) \in \Theta(n \lg n)$.

Exemplo 6

Considere a seguinte recorrência $T(n) = 8T\left(\frac{n}{2}\right) + \Theta(n^2)$. Em seu formato, considera-se:

- $a = 8$;
- $b = 2$;
- $f(n) = \Theta(n^2)$.

Portanto, $n^{\log_b a} = n^{\log_2 8} = n^3$. Desse modo, $f(n) \in O(n^{\log_b a - \epsilon})$ com $\epsilon = 1$, então aplica-se o primeiro caso, obtendo $T(n) \in \Theta(n^3)$.

Exemplo 7

Considere a seguinte recorrência $T(n) = 7T\left(\frac{n}{2}\right) + \Theta(n^2)$. Em seu formato, considera-se:

- $a = 7$;
- $b = 2$;
- $f(n) = n^2$.

Portanto, $n^{\log_b a} = n^{\lg 7} < n^{2,81}$. Desse modo, $f(n) \in O(n^{\log_b a - \epsilon})$ com $\epsilon \approx 0,8$, então aplica-se o primeiro caso, obtendo $T(n) \in \Theta(n^{\lg 7})$.

Exemplos onde não se aplica o método mestre

Estes são alguns exemplos onde não se aplica o método mestre:

- $T(n) = T(n-1) + n$;
- $T(n) = T(n-a) + T(a) + n$, com $a \in \mathbb{Z}^+$;
- $T(n) = T(\alpha n) + T\big((1-\alpha)n\big) + n$, com $0 < \alpha < 1$;
- $T(n) = T(n-1) + \log n$;
- $T(n) = 2T\left(\frac{n}{2}\right) + n \log n$.

CAPÍTULO 4

Divisão e Conquista

A Divisão e Conquista é um paradigma em que se resolve um problema em três etapas a cada nível de recursão (CORMEN et al., 2012):

- **Divisão**: divide-se o problema em um número de subproblemas que são instâncias menores do problema original;
- **Conquista**: resolve-se os subproblemas recursivamente. Entretanto, se o tamanho dos subproblemas for suficientemente pequeno, passa-se a resolvê-los diretamente;
- **Combinação**: as soluções para os subproblemas são combinadas para resolver o problema que as originou.

4.1 Multiplicação de Inteiros

Esta seção discute a multiplicação de dois números inteiros no qual o algoritmo de tempo quadrático tradicional é melhorado (ou acelerado) ao utilizar o paradigma de divisão e conquista (KLEINBERG; TARDOS, 2005).

No algoritmo tradicional, multiplica-se dois números x e y. Para cada dígito de y, multiplica-se os dígitos de x. Os resultados para cada dígito de y são somados,

considerando um deslocamento equivalente a posição do dígito multiplicador de y. Ao contar o número de multiplicações para cada dígito de y, têm-se $O(n)$ operações, no qual n é o número de dígitos do número com maior quantidade de dígitos. Para somar cada um dos resultados obtidos, demanda-se mais $O(n)$ operações.

O algoritmo melhorado se baseia em uma maneira mais eficiente de dividir o produto em somas parciais. Para exemplificar, irá se considerar um número na base 2, mas isso não importa, pois o algoritmo é o mesmo para qualquer base, deve-se apenas se fazer os ajustes necessários. Assuma que $x = x_1 \cdot 2^{\frac{n}{2}} + x_0$, no qual x_1 corresponde aos $\frac{n}{2}$ bits de maior ordem (significância) e x_0 os $\frac{n}{2}$ bits de menor ordem. De maneira similar se considera o número $y = y_1 \cdot 2^{\frac{n}{2}} + y_0$. Desse modo, pode-se considerar a multiplicação de x e y da seguinte forma (KLEINBERG; TARDOS, 2005):

$$\begin{aligned} xy &= (x_1 \cdot 2^{\frac{n}{2}} + x_0)(y_1 \cdot 2^{\frac{n}{2}} + y_0) \\ &= x_1 y_1 \cdot 2^n + (x_1 y_0 + x_0 y_1) \cdot 2^{\frac{n}{2}} + x_0 y_0. \end{aligned} \tag{4.1}$$

O tempo computacional para computar uma multiplicação como essa é de $T(n) \leq 4T\left(\frac{n}{2}\right) + n$.

No entanto, é possível melhorar considerando o seguinte (truque de Karatsuba):

$$\begin{aligned} xy &= x_1 y_1 \cdot 2^n + (x_1 y_0 + x_0 y_1) \cdot 2^{\frac{n}{2}} + x_0 y_0 \\ &= x_1 y_1 \cdot 2^n + \left((x_1 + x_0) \cdot (y_1 + y_0) - x_0 y_0 - x_1 y_1\right) \cdot 2^{\frac{n}{2}} + x_0 y_0. \end{aligned} \tag{4.2}$$

A parte $(x_1 + x_0) \cdot (y_1 + y_0) - x_0 y_0 - x_1 y_1$ é equivalente a $x_1 y_0 + x_0 y_1$:

$$\begin{aligned} (x_1 + x_0)(y1 + y_0) - x_0 y_0 - x_1 y_1 &= x_1 y_1 + x_1 y_0 + x_0 y_1 + x_0 y_0 - x_0 y_0 - x_1 y_1 \\ &= x_1 y_0 + x_0 y_1. \end{aligned} \tag{4.3}$$

Com essa forma de calcular, se reaproveita de cálculo que $x_0 y_0$ e $x_1 y_1$. O Algoritmo 6,

exibe o algoritmo que utliza essa forma de multiplicação.

Algoritmo 6: Multiplicação-Recursiva

Input :valores inteiros na base-2 x e y, número de bits n

if $n = 1$ **then**
  return $x \cdot y$
else
  $m \leftarrow \frac{n}{2}$
  $x_1 \leftarrow \frac{x}{2^m}$
  $x_0 \leftarrow x \mod 2^m$
  $y_1 \leftarrow \frac{y}{2^m}$
  $y_0 \leftarrow y \mod 2^m$
  $p \leftarrow$ Multiplicação-Recursiva$(x_1 + x_0 \, , \, y_1 + y_0)$
  $x_1 y_1 \leftarrow$ Multiplicação-Recursiva$(x_1 \, , \, y_1)$
  $x_0 y_0 \leftarrow$ Multiplicação-Recursiva$(x_0 \, , \, y_0)$
  return $x_1 y_1 \cdot 2^n + (p - x_1 y_1 - x_0 y_0) \cdot 2^{\frac{n}{2}} + x_0 y_0$

4.1.1 Complexidade

O tempo computacional requerido é $T(n) \leq 3T\left(\frac{n}{2}\right) + cn$.

Desafio

Dê a fórmula fechada para a recorrência $T(n) \leq 3T\left(\frac{n}{2}\right) + cn$, identificando a respectiva complexidade.

4.2 Multiplicação de Matrizes

Para multiplicação de matrizes, há um algoritmo bem conhecido que demanda complexidade de tempo $\Theta(n^3)$. Uma versão do mesmo para multiplicação de duas matrizes

quadradas é apresentado no Algoritmo 7.

Algoritmo 7: Multiplicação-Matriz-Quadrada

Input : Duas matrizes quadradas $A = \mathbb{R}^{n \times n}$ e $B = \mathbb{R}^{n \times n}$.

```
1 n ← rows(A)
2 C ← ℝ^{n×n}
3 for i ← 1 to n do
4     for j ← 1 to n do
5         c_ij ← 0
6         for k ← 1 to n do
7             c_ij ← c_ij + a_ik · b_kj
8 return C
```

Um algoritmo simples de divisão e conquista para multiplicação de matrizes pode ser visualizado no Algoritmo 8. Na linha 6, o algoritmo divide as matrizes em quatro

partes de mesmo formato para as matrizes.

Algoritmo 8: Multiplicação-Matriz-Quadrada-DC

Input : Duas matrizes quadradas $A = \mathbb{R}^{n \times n}$ e $B = \mathbb{R}^{n \times n}$.

```
1  n ← rows(A)
2  C ← ℝ^{n×n}
3  if n = 1 then
4      c_11 ← a_11 · b_11
5  else
6      particionar A, B e C em quatro matrizes cada
7      C_11 ← Multiplicação-Matriz-Quadrada-DC(A_11, B_11) +
         Multiplicação-Matriz-Quadrada-DC(A_12, B_21)
8      C_12 ← Multiplicação-Matriz-Quadrada-DC(A_11, B_12) +
         Multiplicação-Matriz-Quadrada-DC(A_12, B_22)
9      C_21 ← Multiplicação-Matriz-Quadrada-DC(A_21, B_11) +
         Multiplicação-Matriz-Quadrada-DC(A_22, B_21)
10     C_22 ← Multiplicação-Matriz-Quadrada-DC(A_21, B_12) +
         Multiplicação-Matriz-Quadrada-DC(A_22, B_22)
11 return C
```

Assumindo que as entradas possuem um tamanho na potência de 2 exatamente, torna a análise da complexidade mais simples. A complexidade de tempo é igual $T(n) = 8T\left(\frac{n}{2}\right) + \Theta(n^2)$, sendo que $T(1) = \Theta(1)$. A parte $\Theta(n^2)$ é devido a soma das matrizes obtidas pelas chamadas recursivas.

Desafio

Dê a fórmula fechada para a recorrência $T(n) \leq 8T\left(\frac{n}{2}\right) + \Theta(n^2)$, identificando a respectiva complexidade.

Ainda há como melhorar a complexidade de tempo, utilizando como base a versão de divisão e conquista. Essa versão mais eficiente é conhecida como algoritmo de Strassen (Algoritmo 9). A complexidade do algoritmo de Strassen é dada pela recorrência $T(n) = 7T\left(\frac{n}{2}\right) + \Theta(n^2)$, sendo $T(1) = \Theta(1)$.

Desafio

Dê a fórmula fechada para a recorrência $T(n) = 7T\left(\frac{n}{2}\right) + \Theta(n^2)$, identificando a respectiva complexidade.

Algoritmo 9: Multiplicação-Matriz-Quadrada-DC

Input : Duas matrizes quadradas $A = \mathbb{R}^{n \times n}$ e $B = \mathbb{R}^{n \times n}$.

1 $n \leftarrow \text{rows}(A)$
2 $C \leftarrow \mathbb{R}^{n \times n}$
3 **if** $n = 1$ **then**
4 $\quad c_{11} \leftarrow a_{11} \cdot b_{11}$
5 **else**
6 $\quad$ particionar A, B e C em quatro matrizes cada
7 $\quad S_1 \leftarrow B_{12} - B_{22}$
8 $\quad S_2 \leftarrow A_{11} + A_{12}$
9 $\quad S_3 \leftarrow A_{21} + A_{22}$
10 $\quad S_4 \leftarrow B_{21} - B_{11}$
11 $\quad S_5 \leftarrow A_{11} + A_{22}$
12 $\quad S_6 \leftarrow B_{11} + B_{22}$
13 $\quad S_7 \leftarrow A_{12} - A_{22}$
14 $\quad S_8 \leftarrow B_{21} + B_{22}$
15 $\quad S_9 \leftarrow A_{11} - A_{21}$
16 $\quad S_{10} \leftarrow B_{11} + B_{12}$
17 $\quad P_1 \leftarrow$ Multiplicação-Matriz-Quadrada-DC(A_{11}, S_1)
18 $\quad P_2 \leftarrow$ Multiplicação-Matriz-Quadrada-DC(S_2, B_{22})
19 $\quad P_3 \leftarrow$ Multiplicação-Matriz-Quadrada-DC(S_3, B_{11})
20 $\quad P_4 \leftarrow$ Multiplicação-Matriz-Quadrada-DC(A_{22}, S_4)
21 $\quad P_5 \leftarrow$ Multiplicação-Matriz-Quadrada-DC(S_5, S_6)
22 $\quad P_6 \leftarrow$ Multiplicação-Matriz-Quadrada-DC(S_7, S_8)
23 $\quad P_7 \leftarrow$ Multiplicação-Matriz-Quadrada-DC(S_9, S_{10})
24 $\quad C_{11} \leftarrow P_5 + P_4 - P_2 + P_6$
25 $\quad C_{12} \leftarrow P_1 + P_2$
26 $\quad C_{21} \leftarrow P_3 + P_4$
27 $\quad C_{22} \leftarrow P_5 + P_1 - P_3 - P_7$
28 **return** C

4.3 Medianas e Estatística de Ordem

A i-ésima estatística de ordem é o i-ésimo menor elemento de uma coleção. O mínimo é a primeira estatística de ordem. O máximo elemento é a n-ésima estatística de ordem para um conjunto com n elementos (CORMEN et al., 2012). Mediana é o 50º percentil, ou seja, metade dos números são maiores que ele e a outra metade é menor. Se o conjunto possui uma quantidade de elementos par, há duas medianas (inferior e superior) (DASGUPTA; PAPADIMITRIOU; VAZIRANI, 2008; CORMEN et al., 2012). Por questões de simplicidade, considera-se será menor dois elementos como a mediana.

Como pode-se fazer para encontrar o mínimo e o máximo? Qual o algoritmo para tal tarefa? Qual a complexidade desse algoritmo? Para o mínimo e máximo é mais simples encontrar um algoritmo linear. Para outras estatísticas de ordem, a solução não parece tão simples quanto a de encontrar o mínimo e o maximo de um conjunto.

O algoritmo apresentado a seguir utiliza o paradigma de divisão e conquista que particiona o vetor em duas partes e segue em apenas uma delas com a recursão (Algoritmo

11).

Algoritmo 10: Partição-Aleatoria

Input : Um vetor A, dois números inteiros p e r.

```
1  i ← Random(p, r)
2  swap(A_r, A_i)
3  x ← A_r
4  i ← p − 1
5  for j ← p to r − 1 do
6      if A_j ≤ x then
7          i ← i + 1
8          swap(A_i, A_j)
9  swap(A_{i+1}, A_r)
10 return i + 1
```

Algoritmo 11: Seleção-Aleatória-Estatística-de-Ordem

Input : Um vetor A, dois números inteiros p e r, identificando as posições de início e fim da busca, um valor inteiro i, indicando qual i-ésimo número deve ser encontrado.

```
1  if p = r then
2      return A_p
3  q ← Partição-Aleatoria(A, p, r)
4  k ← q − p + 1
5  if i = k then
6      return A_q
7  else
8      if i < k then
9          return Seleção-Aleatória-Estatística-de-Ordem(A, p, q − 1, i)
10     else
11         return Seleção-Aleatória-Estatística-de-Ordem(A, q + 1, r, i − k)
```

Complexidade de Tempo

A complexidade de tempo do Algoritmo 11 para o pior caso é $\Theta(n^2)$, pois a escolha de um pivô ruim pode eliminar apenas um candidato ao i-ésimo menor para a próxima chamada recursiva. Desse modo, teria-se a demanda de tempo $T(n) \leq T(n-1) + \Theta(n)$. No entanto, esse algoritmo possui uma complexidade esperada linear (CORMEN et al., 2012).

O procedimento Partição-Aleatoria (Algoritmo 10) tem igual probabilidade de retornar qualquer elemento como pivô. Então, cada elemento $k \in \{1, 2, \ldots, n\}$ tem probabilidade $\frac{1}{n}$ de ser selecionado. Considere as variáveis aleatórias X_k, nas quais $X_k = I\{\text{o subvetor tem exatamente } k \text{ elementos}\}$. Elas que correspondem ao subvetor de A com k elementos. Então, $E[X_k] = \frac{1}{n}$.

Quando se chama a função "Seleção-Aleatória-Estatística-de-Ordem" e se escolhe o elemento pivô A_q, não se sabe, a priori, se terminará com a resposta correta. Então, faz-se chamadas recursivas para tratar das posições entre $A_{p\ldots q-1}$ ou entre $A_{q+1\ldots r}$. Para obter um limite superior de $T(n)$, assume-se que está no subvetor com mais elementos. A variável $X_k = 1$ para um valor de k elementos e 0 caso contrário. Quando $X_k = 1$, os dois subvetores tem $k-1$ e $n-k$ elementos. Têm-se então a seguinte recorrência (CORMEN et al., 2012):

$$\begin{aligned} T(n) &\leq \sum_{k=1}^{n} X_k \cdot \Big(T\big(\max(k-1, n-k)\big) + O(n) \Big) \\ &= \sum_{k=1}^{n} X_k \cdot T\big(\max(k-1, n-k)\big) + O(n). \end{aligned} \tag{4.4}$$

Considerando em termos de valores esperados, obtém-se (CORMEN et al., 2012):

$$\begin{aligned} E\big[T(n)\big] &\leq E\left[\sum_{k=1}^{n} X_k \cdot T\big(\max(k-1, n-k)\big) + O(n) \right] \\ &= \sum_{k=1}^{n} E\Big[X_k \cdot T\big(\max(k-1, n-k)\big) \Big] + O(n) \\ &= \sum_{k=1}^{n} E\Big[X_k \Big] \cdot E\Big[T\big(\max(k-1, n-k)\big) \Big] + O(n) \\ &= \sum_{k=1}^{n} \frac{1}{n} \cdot E\Big[T\big(\max(k-1, n-k)\big) \Big] + O(n) \end{aligned} \tag{4.5}$$

Considera-se $T\big(\max(k-1,n-k)\big)$ e X_k como valores aleatórios independentes.

Considere que a expressão $\max(k-1,n-k)$, têm-se (CORMEN et al., 2012):

$$\max(k-1,n-k) = \begin{cases} k-1 & \text{se } k > \left\lceil \frac{n}{2} \right\rceil, \\ n-k & \text{se } k \leq \left\lceil \frac{n}{2} \right\rceil. \end{cases} \tag{4.6}$$

Se n é par, cada termo de $T\left(\left\lceil \frac{n}{2} \right\rceil\right)$ até $T(n-1)$ aparece duas vezes na soma. Se for ímpar, os termos aparecem duas vezes e o termo $T\left(\left\lfloor \frac{n}{2} \right\rfloor\right)$ uma vez. Então têm-se:

$$E\big[T(n)\big] \leq \frac{2}{n} \sum_{k=n/2}^{n-1} E\big[T(k)\big] + O(n). \tag{4.7}$$

Assumindo por substituição que o valor esperado para $T(n)$ possui como limite superior cn, considerando a base $T(n) = O(1)$ para um n menor do que uma constante, faz-se a prova da complexidade linear esperada (CORMEN et al., 2012):

$$\begin{aligned} E\big[T(n)\big] &\leq \frac{2}{n} \sum_{k=n/2}^{n-1} ck + an \\ &= \frac{2c}{n}\left(\sum_{k=1}^{n-1} k - \sum_{k=1}^{\lfloor n/2 \rfloor - 1} k\right) + an \\ &= \frac{2c}{n}\left(\frac{(n-1)n}{2} - \frac{(\lfloor n/2 \rfloor - 1)\lfloor n/2 \rfloor}{2}\right) + an \\ &\leq \frac{2c}{n}\left(\frac{(n-1)n}{2} - \frac{(n/2-2)(n/2-1)}{2}\right) + an \\ &= \frac{2c}{n}\left(\frac{n^2-n}{2} - \frac{n^2/4 - 3n/2 + 2}{2}\right) + an \\ &= \frac{c}{n}\left(\frac{3n^2}{4} + \frac{n}{2} - 2\right) + an \\ &= c\left(\frac{3n}{4} + \frac{1}{2} - \frac{2}{n}\right) + an \\ &\leq \frac{3cn}{4} + \frac{c}{2} + an \\ &= cn - \left(\frac{cn}{4} - \frac{c}{2} - an\right). \end{aligned} \tag{4.8}$$

Para completar a prova da substituição, demonstra-se que para um n suficientemente grande, essa expressão é no máximo cn, então:

$$\begin{aligned} \frac{cn}{4} - \frac{c}{2} - an &\geq 0 \\ n\left(\frac{c}{4} - a\right) - \frac{c}{2} &\geq 0 \\ n\left(\frac{c}{4} - a\right) &\geq \frac{c}{2} \\ n &\geq \frac{\frac{c}{2}}{\left(\frac{c}{4} - a\right)} \\ n &\geq \frac{2c}{c-4a}. \end{aligned} \tag{4.9}$$

Então, se for assumido $T(n) = O(1)$ para $n < \frac{2c}{c-4a}$, têm-se $E\big[T(n)\big] = O(n)$. Conclui-se então que pode-se encontrar qualquer estatística de ordem, e em particular a mediana, em tempo esperado $O(n)$, assumindo que os elementos são distintos (CORMEN et al., 2012).

4.3.1 Método de Seleção Determinístico Linear

Há um método para encontrar o i-ésimo menor elemento de um conjunto de valores, cujo o tempo computacional é linear (BLUM et al., 1973). Esse método é detalhado no Algoritmo 12. Nele, o vetor A é dividido em partes de no máximo 5 elementos cada (linha 1). Depois, ordena-se cada uma dessas partes e encontra-se a mediana de cada uma delas (linha 2). Cria-se então um vetor com a mediana de cada uma das partes e procura-se a mediana do vetor de medianas (linha 3). Essa mediana servirá de pivô para dividir o vetor em duas partes: uma com os elementos maiores que o pivô e outra parte com os menores elementos que o pivô (linha 5). Se a posição procurada é a mesma da mediana pivô, o algoritmo pára, retornando-a. Caso contrário, continua procurando na

partição menor ou maior que a mediana.

Algoritmo 12: Seleção-Determinística

Input : Um vetor A, dois números inteiros p e r, identificando as posições de início e fim da busca, um valor inteiro i, indicando qual i-ésimo número deve ser encontrado.

```
1  if p = r then
2      return (A_p, p)
   // Escolha do pivô
3  quebrar o vetor A em m partes de modo que cada parte tenha 5 elementos
4  ordenar cada uma das m partes
5  criar um vetor C contendo as medianas de cada uma das m partes
   // Calcula a mediana do vetor de medianas (C)
6  (v, l) ← Seleção-Determinística(C, 1, |C|, |C|/2)
7  k ← posição do valor v em A
   // Particionamento e seleção
8  Particionar o vetor A na posição k
9  if i = k then
10     return (A_k, k)
11 else
12     if i < k then
13         return Seleção-Determinística(A, p, k − 1, i)
14     else
15         return Seleção-Determinística(A, k + 1, r, i)
```

Complexidade

Para determinar a complexidade de tempo, considere que x é a mediana obtida na linha 4 do Algoritmo 12. Como a mediana x é o valor pivô para a partição do algoritmo, considerar limites relacionados a ela ajudam a entender a complexidade.

Pelo menos metade das medianas encontradas na linha 3 são maiores ou iguais a x.

Mais da metade dos elementos com medianas superiores a x possuem pelo menos 3 elementos maiores que x. Desconsiderando um potencial grupo que tenha menos que 5 elementos e o próprio grupo de x, obtém-se o seguinte limite inferior para elementos maiores que x:

$$3\left(\left\lceil\frac{1}{2}\left\lceil\frac{n}{5}\right\rceil\right\rceil-2\right)\geq\frac{3n}{10}-6. \tag{4.10}$$

No mínimo, há $\frac{3n}{10}-6$ são menores que x. Então, no pior caso, as linhas entre 9 e 12 executam uma chamada do Algoritmo 12 para um subvetor com no máximo $n-\frac{3n}{10}+6=\frac{7n}{10}+6$ elementos.

Agora, pode-se definir a recorrência correspondente a complexidade de tempo do algoritmo determinístico de seleção. As linhas 1 (divisão), 2 (ordenação) e 4 (partição) demandam $O(n)$ de tempo computacional. Especificamente quanto a linha 2, são necessários $O(n)$ operações de ordenação em conjuntos com cinco ou menos elementos, ou seja, $O(1)$ elementos. Considerando que a linha 3 toma o tempo $T\left(\left\lceil\frac{n}{5}\right\rceil\right)$ e uma das recorrências entre as linhas 9 e 12 toma tempo $T\left(\frac{7n}{10}+6\right)$, têm-se como função de complexidade:

$$T(n)=\begin{cases}O(1) & \text{se } n<n_0\\ T\left(\left\lceil\frac{n}{5}\right\rceil\right)+T\left(\frac{7n}{10}+6\right)+O(n) & \text{se } n\geq n_0.\end{cases} \tag{4.11}$$

Ainda há que se definir qual o valor de n_0. Considerando sob a hipótese que a função é linear, $T(n)\leq cn$ para uma constante $c>0$ suficientemente grande, então:

$$\begin{aligned}T(n)&=T\left(\left\lceil\frac{n}{5}\right\rceil\right)+T\left(\frac{7n}{10}+6\right)+O(n)\\&\leq c\left\lceil\frac{n}{5}\right\rceil+c\left(\frac{7n}{10}+6\right)+an\\&\leq c\frac{n}{5}+c+c\left(\frac{7n}{10}+6\right)+an\\&=9c\frac{n}{10}+7c+an\\&=cn+\left(-c\frac{n}{10}+7c+an\right).\end{aligned} \tag{4.12}$$

Desse modo, $T(n)$ é no máximo cn se

$$-c\frac{n}{10}+7c+an\leq 0. \tag{4.13}$$

Então, $c \geq 10a\left(\frac{n}{n-70}\right)$ quando $n > 70$. Concluindo, para $n_0 > 70$, a complexidade do algoritmo é linear.

4.4 Transformada Rápida de Fourier

O produto de dois polinômios de grau d é um polinômio de grau $2d$ Veja o seguinte exemplo (DASGUPTA; PAPADIMITRIOU; VAZIRANI, 2008):

$$(1+2x+3x^2)\cdot(2+x+4x^2) = 2+5x+12x^2+11x^3+12x^4. \tag{4.14}$$

De forma genérica, considerando os seguintes polinômios $A(x) = a_0 + a_1x + \ldots + a_dx^d$ e $B(x) = b_0 + b_1x + \ldots + b_dx^d$, o produto seria $C(x) = A(x)\cdot B(x) = c_0 + c_1x + \ldots + c_{2d}x^{2d}$. Cada coeficiente k de $C(x)$ seria:

$$c_k = a_0b_k + a_1b_{k-1} + \ldots + a_kb_0 = \sum_{i=0}^{k} a_ib_{k-i}. \tag{4.15}$$

Desse modo, computar cada coeficiente c_k demandaria $O(k)$ instruções. Intuitivamente, pode-se imaginar que o limite inferior de instruções para calcular todos os $2d+1$ coeficientes demandaria algo na ordem de $\Theta(n^2)$ (DASGUPTA; PAPADIMITRIOU; VAZIRANI, 2008). Será visto a seguir um algoritmo no qual é possível calcular os coeficientes em $O(n\lg n)$. Essa solução é chamada de Transformada Rápida de Fourier e revolucionou o processamento de sinais.

A representação de polinômios $A(x) = a_0 + a_1x + a_2x^2 + \ldots + a_dx^d$ pode ser realizada de dois modos:

- **Por coeficiente**: $a_0, a_1, a_2, \ldots, a_d$;
- **Por valor**: $A(x_0), A(x_1), A(x_2), \ldots, A(x_d)$.

A segunda opção é a mais atrativa para a multiplicação de polinômios. Desde que o produto de $C(x) = A(x)\cdot B(x)$ tenha grau $2d$, basta multiplicar em ordem $\Theta(2d)$, ou seja, em tempo linear. No entanto, para atingir a representação por valor, deve-se realizar

uma avaliação que custa $\Theta(d \lg d)$ para cada polinômio, utilizando-se da transformada rápida de Fourier.

O algoritmo para realizar a multiplicação rápida de polinômios é destacado no Algoritmo 13.

Algoritmo 13: Multiplicação de Polinômios

Input :Dois vetores $A = (a_0, a_1, \ldots, a_{n-1})$ e $B = (b_0, b_1, \ldots, b_{n-1})$ de grau d.

`// Seleção`

1 Escolher alguns pontos $x_0, x_1, \ldots, x_{n-1}$ para $n \geq 2d+1$

`// Avaliação`

2 Computar $A(x_0), A(x_1), A(x_2), \ldots, B(x_d)$ e $B(x_0), B(x_1), B(x_2), \ldots, B(x_d)$ através do algoritmo de Transformada Rápida de Fourier

`// Multiplicação`

3 Computar $C(x_k) = A(x_k) \cdot B(x_k)$ para todo $k = 0, 1, 2, n-1$

`// Interpolação`

4 Resgatar a representação de coeficientes de $C(x) = c_0 + c_1 x + c_2 x^2 + \ldots + c_{2d} x^{2d}$

5 **return** $(c_0 + c_1 x + c_2 x^2 + \ldots + c_{2d} x^{2d})$

Avaliação por Divisão e Conquista

A avaliação segue uma ideia de escolher n pontos nos quais avalia-se um polinômio de grau menor ou igual a $n-1$. Se esses pontos forem pares positivos e negativos, é possível usar-se da sobreposição, pois as potências de 2 se coincidem, para acelerar a computação do algoritmo:

$$\pm x_0, \pm x_1, \ldots, \pm x_{n/2-1}. \tag{4.16}$$

Para entender isso, precisa-se dividir $A(x)$ em potências pares e ímpares, como no exemplo:

$$3 + 4x + 6x^2 + 2x^3 + x^4 + 10x^5 = (3 + 6x^2 + x^4) + x(4 + 2x^2 + 10x^4). \tag{4.17}$$

Note que os polinõmios estão em x^2. De forma mais geral, têm-se:

$$A(x) = A_e(x^2) + xA_o(x^2). \tag{4.18}$$

Considerando os $n/2$ pares $\pm x_i$, os cálculos necessários são:

$$\begin{aligned} A(x_i) &= A_e(x_i^2) + x_i A_o(x_i^2) \\ A(-x_i) &= A_e(x_i^2) - x_i A_o(x_i^2). \end{aligned} \tag{4.19}$$

Considerando essas convenções, avaliar $A(x)$ em n pontos pareados $\pm x_0, \pm x_1, \ldots, x_{n/2-1}$ reduz avaliar $A_e(x)$ e $A_o(x)$ em apenas $n/2$ pontos $x_0^2, x_1^2, \ldots, x_{n/2-1}^2$.

Apesar dessas características, ainda há um problema: o "truque menos-mais" funciona apenas para a recursão do primeiro nível. Para o próximo nível, precisa-se que os $x/2$ pontos avaliados $x_0^2, x_1^2, \ldots, x_{n/2-1}^2$ sejam menos-mais pares. Como seria possível fazer isso para o quadrado de um número? Deve-se usar números complexos para isso.

Quanto aos números complexos selecionados para a divisão e conquista, utiliza-se as n-ésimas raizes de unidade: os números complexos $1, \omega, \omega^2, \ldots, \omega^{n-1}$, na qual $\omega = e^{2\pi i/n}$ (DASGUPTA; PAPADIMITRIOU; VAZIRANI, 2008). Considerando que n é par:

1. As n-ésimas raízes de unidade são "mais-menos" pareadas, $\omega^{n/2+j} = -\omega^j$;

2. O quadrado delas produzem as $(n/2)$-enésimas raízes de unidade.

Desse modo, se o procedimento for iniciado com esses números para algum n potência de 2, então nos sucessivos passos de recursão se terá as n-ésimas raízes de unidade para $k = 1, 2, 3, \ldots$. Todos esses conjuntos de números serão "mais-menos" pareados, o

que permitirá a abordagem de divisão por conquista apresentada no Algoritmo 14.

Algoritmo 14: Transformada-Rápida-de-Fourier

Input : Um vetor $a = (a_0, a_1, \ldots, a_{n-1})$, uma primitiva da enésima raiz de unidade ω.

```
1 if ω = 1 then
2     return a
3 else
4     (s_0, s_1, ..., s_{n/2-1}) ← Transformada-Rápida-de-Fourier((a_0, a_2, ..., a_{n-2}), ω^2)
5     (s'_0, s'_1, ..., s'_{n/2-1}) ← Transformada-Rápida-de-Fourier((a_1, a_3, ..., a_{n-1}), ω^2)
6     for j ← 0 to n/2 − 1 do
7         r_j ← s_j + ω^j s'_j
8         r_{j+n/2} ← s_j − ω^j s'_j
9     return (r_0, r_1, ..., r_{n-1})
```

Interpolação

A última etapa da multiplicação de polinômios presente no Algoritmo 13 é o passo que troca a representação de valores por coeficientes. Para esse passo, utiliza-se a operação inversa à transformada rápida de Fourier:

$$\langle \text{coeficientes} \rangle = \frac{1}{n} \text{Transformada-Rápida-de-Fourier}\big(\langle \text{valores} \rangle, \omega^{-}1\big). \tag{4.20}$$

4.4.1 Complexidade

Para calcular a complexidade do Algoritmo 13 (multiplicação de polinômios), precisa-se primeiro identificar a complexidade do algoritmo de Transformada Rápida de Fourier (Algoritmo 14). O tempo computacional é dado por:

$$T(n) = \begin{cases} O(1) & \text{se } n = 1, \\ 2T\left(\frac{n}{2}\right) + \Theta(n) & \text{se } n > 1. \end{cases} \tag{4.21}$$

A recorrência indica a complexidade de $\Theta(n \lg n)$.

Voltando a analisar o Algoritmo 13, enumera-se a complexidade de cada um de seus principais passos, considerando $n = 2d$:

- **Seleção**: a escolha dos pontos $\Theta(n)$ passos;

- **Avaliação**: para a avaliação, têm-se duas chamadas da Transformada Rapida de Fourier, demandando $\Theta(n \lg n)$;

- **Multiplicação**: Multiplicar os n valores dos polinômios A e B para gerar o polinômio C demanda $\Theta(n)$ operações;

- **Interpolação**: para encontrar os coeficientes de C, utiliza-se a transformada inversa, que invoca a Transformada Rápida de Fourier e aplica uma adaptação constante ao resultado, demandando $\Theta(n \lg n)$ operações.

A complexidade da multiplicação de polinômios é $\Theta(n \lg n)$.

4.5 Quicksort

O algoritmo de Quicksort é um método de ordenação que utiliza a paradigma de divisão e conquista. O algoritmo é dividido então em três etapas (CORMEN et al., 2012):

- **Divisão**: Particionar o vetor em dois subvetores. Um dos subvetores é composto por elementos maiores que o elemento na posição q (elemento pivô) e o outro subvetor por elementos menores.

- **Conquista**: Ordenar os dois subvetores por chamadas recursivas.

- **Combinação**: Como os subvetores já estão ordenados, nada é feito.

O Algoritmo 15 traz uma descrição em pseudo-código do Quicksort. A versão do

Quicksort depende exclusivamente de seu método de partição.

Algoritmo 15: Quicksort

Input : Um vetor A, um índice p e r identificando o intervalo fechado de posições que serão ordenadas.

```
1 if p < r then
2     q ← Partição(A, p, r)
3     Quicksort(A, p, q − 1)
4     Quicksort(A, q + 1, r)
```

O algoritmo de partição (Algoritmo 16) seleciona um pivô na posição mais à direita do vetor. Depois, realiza uma movimentação dos menores elementos para à esquerda do pivô e dos maiores para a direita.

Algoritmo 16: Partição

Input : Um vetor A, um índice p e r identificando o intervalo fechado de posições particionadas.

```
1 x ← A_r
2 i ← p − 1
3 for j ← p to r − 1 do
4     if A_j ≤ x then
5         i ← i + 1
6         swap(A_i, A_j)
7 swap(A_{i+1}, A_r)
8 return i + 1
```

4.5.1 Corretude

As seguintes propriedades se mantém no início de cada iteração do laço das linhas entre 3 e 6 (CORMEN et al., 2012) para qualquer índice k do vetor:

1. Se $p \leq k \leq i$, então $A_k \leq x$;
2. Se $i+1 \leq k \leq j-1$, então $A_k > x$;
3. Se $k = r$, então $A_k = x$.

Essa invariante de laço é verdadeira e é útil para demonstrar a corretude de Quicksort:

- **Inicialização**: Antes da primeira iteração, $i = p - 1$ e $j = p$, portanto as duas primeiras propriedades são satisfeitas. A atribuição da linha 1 satisfaz a teceira propriedade.

- **Manutenção**: Dependendo do teste da linha 4, considera-se dois casos:

  – Quando $A_j > x$, a única ação do laço é incrementar j. Depois que j é incrementado, a propriedade 2 é válida para A_{j-1} e todas as outras entradas permanecem inalteradas.

  – Quando $A_j \leq x$, o laço incrementa i e a propriedade 1 é satisfeita. Desse modo, têm-se $A_{j-1} > x$, visto que o elemento que foi permutado para dentro de A_{j-1} é maior que x.

- **Término**: No término, $j = r$. Portanto o divide-se o vetor em três partes: os elementos maiores que x estão no subvetor à direita, os elementos menores que x estão à esquerda e x está na i-ésima estatística de ordem.

Como o algoritmo de Quicksort subdivide o vetor até uma unidade e a cada unidade submete-se ao algoritmo de partição, ao final do Quicksort, todos os valores estarão em suas respectivas estatísticas de ordem, ou seja, o vetor estará ordenado.

4.5.2 Complexidade Pessimista

O tempo de execução do Quicksort no pior caso pode ser dado pelo seguinte limitante superior

$$T(n) \leq \max_{0 \leq q \leq n-1} \big(T(q) + T(n-q-1)\big) + \Theta(n). \tag{4.22}$$

Produz-se problemas com tamanho total $n-1$. O pior caso acontece quando dado n elementos, uma partição tem 0 elementos e a outra possui $n-1$ elementos. Esse é o pior

caso, pois na linha 4 do Algoritmo 16, as posições r (denominada de valor x) é comparada com todos os demais elementos. Considerando essa divisão, há mais comparações entre os $n-1$ elementos do que se obtiver duas partições de $\frac{n-1}{2}$ elementos. Para esse último caso, um número maior de elementos da esquerda não será comparado com a partição da direita na linha 4. Portanto, têm-se a seguinte recorrência representando o pior caso:

$$T(n) \leq T(n-1) + \Theta(n). \tag{4.23}$$

Desafio

Encontre a fómula fechada para a recorrência que define a complexidade pessimista do Quicksort.

4.5.3 Complexidade Esperada

Considere X o número de comparações executadas na linha 4 da partição (Algoritmo 16. Considerando um vetor Z, sabe-se que no máximo Z_i será comparado uma vez com Z_j. Nesse sentido, a análise a seguir utiliza a variável aleatória X_{ij} que identifica se Z_i é comparado com Z_j. O número total de comparações para o algoritmo de Quicksort é

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}. \tag{4.24}$$

Tomando os valores esperados, têm-se:

$$\begin{aligned} E[X] &= E\left[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}\right] \\ &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E[X_{ij}] \\ &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} Pr\{Z_i \text{ é comparado com } Z_j\}. \end{aligned} \tag{4.25}$$

$Pr\{Z_i$ é comparado com $Z_j\}$ é a probabilidade de Z_i ser comparado com Z_j. Para

calculá-la, têm-se:

$$\begin{aligned} Pr\{Z_i \text{ é comparado com } Z_j\} &= Pr\{Z_i \text{ ou } Z_j \text{ é o primeiro pivô escolhido de } Z_{ij}\} \\ &= Pr\{Z_i \text{ é o primeiro pivô escolhido de } Z_{ij}\} \\ &+ Pr\{Z_j \text{ é o primeiro pivô escolhido de } Z_{ij}\} \\ &= \frac{1}{j-i+1} + \frac{1}{j-i+1} \\ &= \frac{2}{j-i+1} \end{aligned} \tag{4.26}$$

Considere que $Z_{ij} = \{Z_i, Z_{i+1}, \ldots, Z_j\}$ é o subvetor com $j-i+1$ elementos (elementos entre i e j incluindo-os). Desse modo, obtêm-se o número esperado de comparações:

$$\begin{aligned} E[X] &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \\ &= \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} \\ &< \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k} \\ &< \sum_{i=1}^{n-1} 2\Big(\ln n + o(1)\Big) \\ &= \sum_{i=1}^{n-1} O(\lg n) \\ &= O(n \lg n) \end{aligned} \tag{4.27}$$

Então, o tempo esperado para execução do Quicksort é de $O(n \lg n)$.

CAPÍTULO 5

Grafos e Buscas

5.1 Introdução

5.1.1 Histórico

Uma breve história do passado da Teoria de Grafos (NETTO, 2006):

- 1847: Kirchhoff utilizou modelos de grafos no estudo de circuitos elétricos, criando a teoria de árvores;
- 1857: Cayley usou grafos em química orgânica para enumeração de isômeros dos hidrocarbonetos alifáticos saturados;
- 1859: Hamilton inventou um jogo de buscar um percurso fechado envolvendo todos os vértices de um dodecaedro regular, de tal modo que cada vértice fosse visitado apenas uma vez;
- 1869: Jordan estudou matematicamente as árvores (grafos acíclicos);
- 1878: Sylvester foi o primeiro a utilizar o termo *graph*;
- 1879: Kempe não conseguiu demonstrar a conjectura das 4 cores;
- 1880: Tait falhou ao demonstrar uma prova falsa da conjectura das 4 cores;

- 1890: Haewood provou que a prova de Kempe estava errada e demonstrou uma prova consistente para 5 cores. A de 4 cores só saiu em 1976;
- 1912: Birkhoff definiu os polinômios cromáticos;
- 1926: Menger demonstrou um importante teorema sobre o problema de desconexão de itinerários em grafos;
- 1930: Kuratowski encontrou uma condição necessária e suficiente para a planaridade de um grafo;
- 1931: Whitney criou a noção de grafo dual;
- 1936: Primeiro livro sobre grafos foi lançado por König;
- 1941: Brooks enunciou um teorema fornecendo um limite para o número cromático de um grafo;
- 1941: Turán foi o primeiro da teoria extremal dos grafos;
- 1947: Tutte resolveu o problema da existência de uma cobertura minimal em um grafo;
- 1956+: Com as publicações de Ford e Fulkerson, Berge (1957) e Ore (1962), a teoria de grafos passa a receber mais interesse;

5.1.2 Definições Iniciais

Antes de visitar a representação de grafos, é importante que saibamos o que são vértices e arestas. Vértices geralmente são representados como unidades, elementos ou entidades, enquanto as arestas representam as ligações/conexões entre pares de vértices. Geralmente, chamaremos o conjunto de vértices de V e o conjunto de arestas de E. Define-se que $E \subseteq V \times V$. Também usaremos n e m para denotarem o número de vértices e arestas respectivamente, então $n = |V|$ e $m = |E|$. O número de arestas possível em um grafo é $\frac{n^2-n}{2}$.

Um grafo pode ser representado de duas formas (CORMEN et al., 2012). A primeira forma é chamada de lista de adjacências e tem mais popularidade em artigos científicos. Nela, o grafo é representado como uma dupla para especificar vértices e arestas. Por exemplo, para um grafo G, pode-se dizer que o mesmo é uma dupla $G = (V, E)$, especificando assim que o grafo G possui um conjunto V de vértices e E de arestas. A segunda forma seria uma através de uma matriz binária, chamada de matriz de adjacência. Normalmente representada pela letra $A(G)$, a matriz é definida por $A(G) = \{0,1\}^{|V|\times|V|}$, a qual seus elementos $a_{u,v} = 1$ se existir uma aresta entre os vértices u e v. Um exemplo das das formas para um mesmo grafo pode ser visualizado no Exemplo 5.1.1.

Example 5.1.1. A Figura 7 exibe um grafo de 4 vértices e 4 arestas. Na representação por listas de adjacências, o grafo pode ser representado da seguinte forma

$$G = (\{1,2,3,4\}, \{\{1,2\},\{1,4\},\{2,4\},\{3,4\}\}). \tag{5.1}$$

A representação por uma matriz de adjacência ficaria assim

$$A(G) = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 1 & 0 & 1 \\ 2 & 1 & 0 & 0 & 1 \\ 3 & 0 & 0 & 0 & 1 \\ 4 & 1 & 1 & 1 & 0 \end{array}.$$

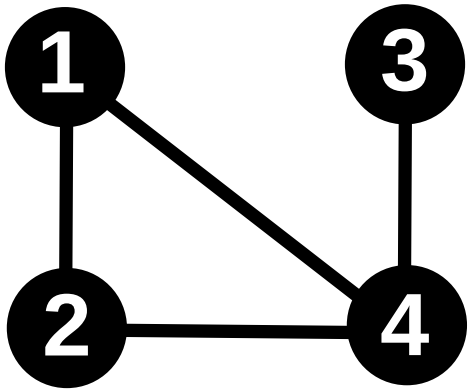


Figura 7 – Exemplo de grafo com 4 vértices e 4 arestas.

5.1.2.1 Grafos Valorados ou Ponderados

Um grafo é valorado quando um peso ou valor é associado a suas arestas. Na literatura, a definição do grafo passa a ser uma tripla $G = (V, E, w)$, na qual V é o conjunto de vértices, E é o conjunto de arestas e $w : e \in E \rightarrow \mathbb{R}$ é a função que especifica o valor.

Quando não se possui valornas arestas, parte-se de uma relação binária entre existir ou não uma aresta entre dois vértices. Neste caso, se u e v possui uma aresta, geralmente se simboliza essa ligação com o valor 1, e se não existir 0.

Em uma matriz de adjacências para grafos valorados, o valor das arestas aparecem nas células da matriz. Em um par de vértices que não possui valor estabelecido (não há aresta), representa-se com uma lacuna ou com um valor simbólico para o problema que o grafo representa. Por exemplo, se os valores representam as distâncias, geralmente se associa o valor infinito aos pares de vértices que não possuem arestas.

Um exemplo de grafo valorado e suas representações pode ser visualizado no Exemplo 5.1.2.

Example 5.1.2. A Figura 8 exibe um grafo valorado de 4 vértices e 4 arestas. Na representação por listas de adjacências, o grafo pode ser representado da seguinte forma

$$G = (\{1,2,3,4\}, \{\{1,2\},\{1,4\},\{2,4\},\{3,4\}\}, w). \tag{5.2}$$

A função w teria os seguintes valores: $w(\{1,2\}) = 8$, $w(\{1,4\}) = 9$, $w(\{2,4\}) = 5$ e $w(\{3,4\}) = 7$.

A representação por uma matriz de adjacência ficaria assim

$A(G) =$

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

ou desta outra forma para o caso de uma aplicação a problemas que envolvam distâncias

$$A(G) = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & \infty & 8 & \infty & 9 \\ 2 & 8 & \infty & \infty & 5 \\ 3 & \infty & \infty & \infty & 7 \\ 4 & 9 & 5 & 7 & \infty \end{array}.$$

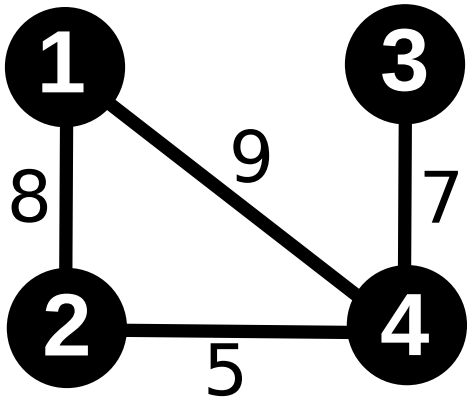


Figura 8 – Exemplo de grafo valorado com 4 vértices e 4 arestas.

Grafos com Sinais

Para representar alguns problemas, utiliza-se valores negativos associados às arestas. Um exemplo disso, seriam grafos que representem relações de amizade e de inimizade. Para amizade, utiliza-se o valor 1 e para inimizade o valor -1. Nesse caso, não dizemos que o grafo é valorado ou ponderado, mas sim um grafo com sinais. Quando os valores negativos e positivos podem ser diferentes de 1 e -1, diz-se que os grafos são valorados e com sinais.

5.1.2.2 Grafos Orientados

Um grafo orientado é aquele no qual suas arestas possuem direção. Nesse caso, não chamamos mais de arestas e sim de arcos. Um grafo orientado é definido como uma

dupla $G = (V, A)$, a qual V é o conjunto de vértices e A é o conjunto de arcos. O conjunto de arcos é composto por pares ordenados (u, v), os quais $u, v \in V$ e representam um arco saindo de u e incidindo em v. Duas funções importantes devem ser consideradas nesse contexto: a função de arcos saintes $\delta^+(v) = \{(v,u) : (v,u) \in A\}$ e arcos entrantes $\delta^-(v) = \{(u,v) : (u,v) \in A\}$.

O Exemplo 5.1.3 exibe a representação de um grafo orientado.

Example 5.1.3. A Figura 9 exibe um grafo orientado de 4 vértices e 4 arestas. Na representação por listas de adjacências, o grafo pode ser representado da seguinte forma

$$G = (\{1,2,3,4\}, \{(1,4),(2,1),(4,2),(4,3)\}). \tag{5.3}$$

A representação por uma matriz de adjacência ficaria assim

$$A(G) = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 0 & 0 & 1 \\ 2 & 1 & 0 & 0 & 0 \\ 3 & 0 & 0 & 0 & 0 \\ 4 & 0 & 1 & 1 & 0 \end{array}.$$

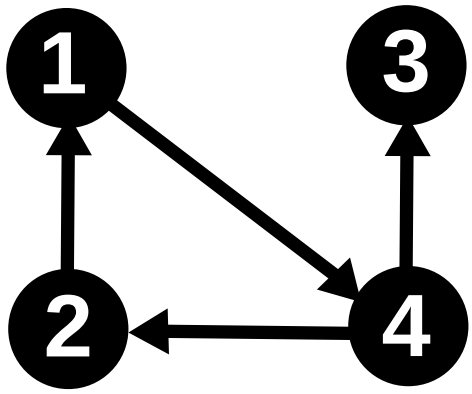


Figura 9 – Exemplo de grafo orientado com 4 vértices e 4 arcos.

5.1.2.3 Hipergrafo

Um hipergrafo $H = (V, E)$ é um grafo no qual as arestas podem conectar qualquer número de vértices. Cada aresta é chamada de hiperaresta $E \subseteq 2^V \setminus \{\}$.

5.1.2.4 Multigrafo

Um multigrafo $G = (V, E)$ é um grafo que permite múltiplas arestas para o mesmo par de vértices. Logo, não se tem mais um conjunto de arestas, mas sim uma tupla de arestas. Para o exemplo da Figura 10, têm-se $E = (\{1,2\}, \{1,2\}, \{1,4\}, \{2,4\}, \{3,4\}, \{3,4\})$.

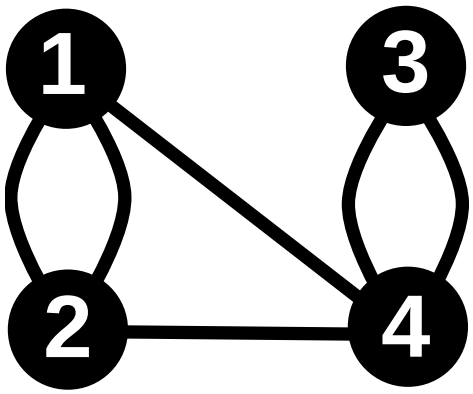


Figura 10 – Exemplo de um multigrafo com 4 vértices e 6 arestas.

5.1.2.5 Grau de um Vértice

O grau de um vértice é a quantidade de arestas que se conectam a determinado vértice. É denotada por uma função d_v, onde $v \in V$. Em um grafo orientado, o número de arcos saintes para um vértice v é denotado por d_v^+, e o número de arcos entrantes é denotado por d_v^-.

5.1.2.6 Igualdade e Isomorfismo

Diz-se que dois grafos $G_1 = (V_1, E_1)$ e $G_2 = (V_2, E_2)$ são iguais se $V_1 = V_2$ e $E_1 = E_2$. Os dois grafos são considerados isomorfos se existir uma função bijetora (uma-por-uma) para todo $v \in V_1$ e para todo $u \in V_2$ preserve as relações de adjacência (NETTO, 2006).

5.1.2.7 Partição de Grafos

Uma partição de um grafo é uma divisão disjunta de seu conjunto de vértices. Um grafo $G = (V, E)$ é dito k-partido se existir uma partição $P = \{p_i | i = 1, \ldots, k \wedge \forall j \in \{1, \ldots, k\}, j \neq i (p_i \cap p_j \neq \{\})\}$. Quando $k = 2$, dize que o grafo é bipartido (NETTO, 2006).

5.1.2.8 Matriz de Incidência

Sobre um grafo orientado $G = (V, E)$, uma matriz de incidência $B(G) = \{+1, -1, \}^{|V| \times |A|}$ mapeia a origem e o destino de cada arco no grafo G. Dado um arco (u, v), $b_{u,(u,v)} = +1$ e $b_{v,(u,v)} = -1$ (NETTO, 2006).

5.1.2.9 Operações com Grafos

As seguintes operações binárias são descritas em Netto (2006):

- União: Dados os grafos $G_1 = (V_1, E_1)$ e $G_2 = (V_2, E_2)$, $G_1 \cup G_2 = (V_1 \cup V_2, E_1 \cup E_2)$;
- Soma (ou *join*): Dados os grafos $G_1 = (V_1, E_1)$ e $G_2 = (V_2, E_2)$, $G_1 + G_2 = (V_1 \cup V_2, E_1 \cup E_2 \cup \{\{u, v\} : u \in V_1 \wedge v \in V_2\})$;
- Produto cartesiano: Dados os grafos $G_1 = (V_1, E_1)$ e $G_2 = (V_2, E_2)$, $G_1 \times G_2 = (V_1 \times V_2, E)$, onde $E = \{\{(v, w), (x, y)\} : (v = x \wedge \{w, y\} \in E_2) \vee (w = y \wedge \{x, y\} \in E_1)\}$. $G_1 \times G_2$ e $G_2 \times G_1$ são isomorfos;
- Composição ou produto lexicográfico: Dados os grafos $G_1 = (V_1, E_1)$ e $G_2 = (V_2, E_2)$, $G_1 \circ G_2 = (V_1 \times V_2, E)$, onde $E = \{\{(v, w), (x, y)\} : (\{v, x\} \in E_1 \vee v = x) \wedge \{w, y\} \in E_2\}$;
- Soma de arestas: Dados os grafos $G_1 = (V_1, E_1)$ e $G_2 = (V_2, E_2)$, os quais $V_1 = V_2$, $G_1 \oplus G_2 = (V_1, E_1 \cup E_2)$.

A seguinte operação unária é descrita em Netto (2006):

- Contração de dois vértices: Dado um grafo $G = (V, E)$ e dois vértices $u, v \in V$, a operação de contração desses dois vértices em G, gera um grafo $G' = (V', E')$ o qual $V' = V \setminus \{u, v\} \cup \{uv\}$ e $E' = \{\{x, y\} \in E : x \neq u \wedge x \neq v\} \cup \{\{x, uv\} : \{x, u\}, \{x, v\} \in E\}$.

Outras operações sobre grafos são descritas na literatura. Esse texto irá omití-las por enquanto para que sejam utilizados no momento mais oportuno. Sõ elas: inserção e remoção de vértices e arestas, desdobramento de um vértice. Essa última depende do contexto de aplicação.

5.1.2.10 Vizinhança

A vizinhança de vértices é diferente para grafos não-orientados e orientados. Para um grafo grafo não-orientado $G = (V, E)$, uma função de vizinhança é definida por $N : v \in V \rightarrow \{u \in V : \{v, u\} \in E\}$ e indica o conjunto de todos os vizinhos de um vértice específico. Para o grafo do Exemplo 5.1.1, $N(1) = \{2, 4\}$.

Para um grafo orientado $G = (V, A)$, diz-se que um vértice $u \in V$ é sucessor de $v \in V$ quando $(v, u) \in A$; e $u \in V$ é antecessor de $v \in V$ quando $(u, v) \in A$. As funções de vizinhança para um grafo orientado G são $N^+ : v \in V \rightarrow \{u \in V : (v, u) \in A\}$, $N^- : v \in V \rightarrow \{u \in V : (u, v) \in A\}$, e $N(v) = N^+(v) \cup N^-(v)$.

Diz-se que a vizinhança de v é fechada quando esse mesmo vértice se inclui no conjunto de vizinhos. A função que representa vizinhança fechada v é simbolizada neste texto como $N_*(v) = N(v) \cup \{v\}$.

As funções de vizinhança também podem ser utilizadas para identificar um conjunto de vértices vizinhos de um grupo de vértices em um grafo $G = (V, E)$ (orientado ou não). Nesse contexto, $N(S) = \bigcup_{v \in S} N(v)$, $N^+(S) = \bigcup_{v \in S} N^+(v)$, e $N^-(S) = \bigcup_{v \in S} N^-(v)$.

As noções de sucessor e antecessor podem ser aplicadas iterativamente. As Equações (5.4), (5.5), (5.6) e (5.7) exibem exemplos de fechos transitivos diretos.

$$N^0(v) = \{v\} \tag{5.4}$$

$$N^{+1}(v) = N^+(v) \tag{5.5}$$

$$N^{+2}(v) = N^+\left(N^{+1}(v)\right) \tag{5.6}$$

$$N^{+n}(v) = N^{+}\left(N^{+(n-1)}(v)\right) \tag{5.7}$$

Chama-se de fecho transitivo direto aqueles que correspondem aos vizinhos sucessivos e os inversos os que correspondem aos vizinhos antecessores. Um fecho transitivo direto de um vértice v de um grafo $G = (V, E)$ são todos os vértices atingíveis a partir v no grafo G; ele é representado pela função $R^{+}(v) = \bigcup_{k=0}^{|V|} N^{+k}(v)$. Um fecho transitivo inverso de v é o conjunto de vértices que atingem v; ele é representado pela função $R^{-}(v) = \bigcup_{k=0}^{|V|} N^{-k}(v)$. Diz-se que w é descendente de v se $w \in R^{+}(v)$. Diz-se que w é ascendente de v se $w \in R^{-}(v)$.

5.1.2.11 Grafo Regular

Um grafo não-orientado $G = (V, E)$ que tenha $d(v) = k \forall\, v \in V$ é chamado de grafo k-regular ou de grau k. Um grafo orientado $G_o = (V, A)$ que possui a propriedade $d^{+}(v) = k \forall\, v \in V$ é chamado de grafo exteriormente regular de semigrau k. Se G_o tiver $d^{-}(v) = k \forall\, v \in V$ é chamado de grafo interiormente regular de semigrau k.

5.1.2.12 Grafo Simétrico

Um grafo orientado $G = (V, A)$ é simétrico se $(u, v) \in A \iff (v, u) \in A \forall\, u, v \in V$.

5.1.2.13 Grafo Anti-simétrico

Um grafo orientado $G = (V, A)$ é anti-simétrico se $(u, v) \in A \iff (v, u) \notin A \forall\, u, v \in V$.

5.1.2.14 Grafo Completo

Um grafo completo $G = (V, E)$ é completo se $E = V \times V$.

Grafos bipartidos completos $G_B = ((X, Y), E)$ possuem $E = X \times Y$.

5.1.2.15 Grafo Complementar

Para um grafo $G = (V, E)$, um grafo complementar é definido por $G^c = \overline{G} = (V, (V \times V) \setminus E)$.

5.1.2.16 Percursos em Grafos

"Um percurso, itinerário ou cadeia é uma família de ligações sucessivamente adjacentes, cada uma tendo uma extremidade adjacente a anterior e a outra à subsequente (à execeção da primeira e da última)" (NETTO, 2006). Diz-se que um percurso é aberto quando a última ligação é adjacente a primeira. Têm-se desse modo um ciclo.

Um percurso é considerado simples se não repetir ligações (NETTO, 2006).

Caminhos são cadeias em grafos orientados.

Circuitos são ciclos em grafos orientados.

5.1.2.17 Cintura e Circunferência

Cintura de um grafo G é comprimendo do menor ciclo existente no grafo. É representada pela função $g(G)$. A circunferência é comprimento do maior ciclo. A circunferência do grafo G é representada pela função $c(G)$.

5.2 Representações Computacionais

Duas formas de representação computacional de grafos são amplamente utilizadas. São elas "listas de adjacências" e "por matriz de adjacências" (CORMEN et al., 2012). Elas possuem vantagens e desvantagens principamente relacionadas à complexidade computacional (consumo de recursos em tempo e espaço). Detalhes sobre vantagens e desvantagens não aparecerão nesse documento. Um de nossos objetivos do momento será implementar e avaliar as duas formas de representação.

5.2.1 Lista de Adjacências

A representação de um grafo $G = (V, E)$ por listas de adjacências consiste em um arranjo, chamado aqui de *Adj*. Esse arranjo é composto por $|V|$ listas, e cada posição do arranjo representa as adjacências de um vértice específico (CORMEN et al., 2012). Para cada

$\{u, v\} \in E$, têm-se $Adj[u] = (\ldots, v, \ldots)$ e $Adj[v] = (\ldots, u, \ldots)$ quando G for não-dirigido. Quando G for dirigido, para cada $(u, v) \in E$, têm-se $Adj[u] = (\ldots, v, \ldots)$.

Para grafos ponderados, Cormen et al. (2012) sugere o uso da própria estrutura de adjacências para armazenar o peso. Dado um grafo ponderado não-dirigido $G = (V, E, w)$, para cada $\{u, v\} \in E$, têm-se $Adj[u] = \big(\ldots, \big(v, w(\{u, v\})\big), \ldots\big)$ e $Adj[v] = \big(\ldots, \big(u, w(\{u, v\})\big), \ldots\big)$. Quando o grafo for dirigido, para cada $(u, v) \in E$, têm-se $Adj[u] = \big(\ldots, \big(v, w((u, v))\big), \ldots\big)$.

O Algoritmo 17 representa a carga de um grafo dirigido e ponderado $G = (V, A, w)$ em uma lista de adjacências Adj.

Algoritmo 17: Criação de uma lista de adjacências para um grafo dirigido e ponderado.

Input : um grafo dirigido e ponderado $G = (V, A, w)$

```
1 criar arranjo Adj[|V|]
2 foreach v ∈ V do
3     Adj[v] ← listaVazia()
4 foreach (u, v) ∈ A do
5     Adj[u] ← Adj[u] ∪ (v, w((u, v)))
6 return Adj
```

5.2.2 Matriz de Adjacências

Uma matriz de adjacência é uma representação de um grafo através de uma matriz A. Para um grafo não-dirigido $G = (V, E)$, $A = \mathbb{B}^{|V|\times|V|}$, na qual cada elemento $a_{u,v} = 1$ e $a_{v,u} = 1$ se $\{u, v\} \in E$; $a_{u,v} = 0$ e $a_{v,u} = 0$ caso $\{u, v\} \notin E$. Para todo grafo não-dirigido G, $a_{u,v} = a_{v,u}$.

Para um grafo dirigido $G = (V, X)$, $A = \mathbb{B}^{|V|\times|V|}$, na qual cada elemento $a_{u,v} = 1$ se $(u, v) \in A$; $a_{u,v} = 0$ e $a_{u,v} = 0$ caso $(u, v) \notin X$.

Para um grafo não-dirigido e ponderado $G = (V, E, w)$, a matriz será formada por células que comportem o tipo de dado representado pelos pesos. Assumindo que os pesos serão números reais, então a matriz de adjacências será $A = \mathbb{R}^{|V|\times|V|}$. Cada elemento $a_{u,v} = w(\{u, v\})$ e $a_{v,u} = w(\{u, v\})$ se $\{u, v\} \in E$; $a_{u,v} = \epsilon$ e $a_{v,u} = \epsilon$ caso $\{u, v\} \notin$

E. ϵ é um valor que representa a não conexão, geralmente 0, $+\infty$ ou $-\infty$ dependendo do contexto de aplicação.

O Algoritmo 18 representa a carga de um grafo dirigido e ponderado $G = (V, A, w)$ em uma matriz de adjacências Adj.

Algoritmo 18: Criação de uma matriz de adjacências para um grafo dirigido e ponderado.

Input : um grafo dirigido e ponderado $G = (V, A, w : A \to \mathbb{R})$, um símbolo ϵ que representa a não adjacência

```
1 Adj ← ℝ^(|V|×|V|)
2 foreach v ∈ V do
3     foreach u ∈ V do
4         Adj_{u,v} ← ϵ
5 foreach (u, v) ∈ A do
6     Adj_{u,v} ← w((u, v))
7 return Adj
```

5.2.3 Exercícios

Implemente as duas bibliotecas para grafos. Preencha a seguinte tabela a partir da análise computacional, de acordo com as operações abaixo determinadas.

	Lista de Adjacências	**Matriz de Adjacências**
Inserção de vértice		
Inserção de arestas		
Remoção de vértice		
Remoção de arestas		
Teste se $\{u, v\} \in E$		
Percorrer vizinhos		
Grau de um vértice		

5.3 Buscas em Grafos

5.3.1 Busca em Largura

Dado um grafo $G = (V, E)$ e uma origem s, a **busca em largura** (Breadth-First Search - BFS) explora as arestas/arcos de G a partir de s para cada vértice que pode ser atingido a partir de s. É uma exploração por nível. O procedimento descobre as distâncias (número de arestas/arcos) entre s e os demais vértices atingíveis de G. Pode ser aplicado para grafos orientados e não-orientados (CORMEN et al., 2012).

O algoritmo pode produzir uma árvore de busca em largur com raiz s. Nesta árvore, o caminho de s até qualquer outro vértice é um caminho mínimo em número de arestas/arcos (CORMEN et al., 2012).

O Algoritmo 19 descreve as operações realizadas em uma busca em largura. Nele, criam-se três estruturas de dados que serão utilizadas para armazenar os resultados da busca. O arranjo C_v é utilizado para determinar se um vértice $v \in V$ foi visitado ou não; D_v determina a distância percorrida até encontrar o vértice $v \in V$; e A_v determina o vértice antecessor ao $v \in V$ em uma busca em largura a partir de s (CORMEN et al., 2012).

5.3.1.1 Complexidade da Busca em Largura

O número de operações de enfileiramento e desenfileiramento é limitado a $|V|$ vezes, pois visita-se no máximo $|V|$ vértices. Como as operações de enfileirar e desenfileirar podem ser realizadas em tempo $\Theta(1)$, então para realizar estas operações demanda-se tempo de $O(|V|)$. Deve-se considerar ainda, que muitas arestas/arcos incidem em vértices já visitados, então inclui-se na complexidade de uma BFS a varredura de todas as adjacências, que demandaria $\Theta(|E|)$. Diz-se então, que a complexidade computacional da BFS é $O(|V| + |E|)$.

Algoritmo 19: Busca em largura.

Input : um grafo $G = (V, E)$, vértice de origem $s \in V$

```
// configurando todos os vértices
```
$C_v \leftarrow$ **false** $\forall v \in V$
$D_v \leftarrow \infty \; \forall v \in V$
$A_v \leftarrow$ **null** $\forall v \in V$
```
// configurando o vértice de origem
```
$C_s \leftarrow$ **true**
$D_s \leftarrow 0$
```
// preparando fila de visitas
```
$Q \leftarrow$ **Fila()**
Q.enqueue(s)
```
// propagação das visitas
```
while $Q.empty() =$ ***false*** **do**
    $u \leftarrow Q$.dequeue()
    foreach $v \in N(u)$ **do**
        if $C_v =$ ***false*** **then**
            $C_v \leftarrow$ **true**
            $D_v \leftarrow D_u + 1$
            $A_v \leftarrow u$
            Q.enqueue(v)
return (D, A)

5.3.1.2 Propriedades e Provas

Caminhos Mínimos

A busca em largura garante a descoberta dos caminhos mínimos em um grafo não-ponderados $G = (V, E)$ de um vértice de origem $s \in V$ para todos os demais atingíveis. Para demonstrar isso, Cormen et al. (2012) examina algumas propriedades importantes a seguir. Considere a distância de um caminho mínimo $\delta(s, v)$ de s a v como o número mínimo de arestas/arcos necessários para percorrer esse caminho.

Lema 5.3.1. *Seja $G = (V, E)$ um grafo orientado ou não-orientado e seja $s \in V$ um vértice arbitrário, então $\delta(s, v) \leq \delta(s, u) + 1$ para qualquer aresta/arco $(u, v) \in E$.*

Prova: Se u pode ser atingido a partir de s, então o mesmo ocorre com v. Desse modo, o caminho mínimo de s para v não pode ser mais longo do que o caminho de s para u seguido pela aresta/arco (u, v) e a desigualdade vale. Se u não pode ser alcançado por s, então $\delta(s, u) = \infty$, e a desigualdade é válida. ■

Lema 5.3.2. *Seja $G = (V, E)$ um grafo orientado ou não-orientado e suponha que G tenha sido submetido ao algoritmo BFS (Algoritmo 19) partindo de um dado vértice de origem $s \in V$. Ao parar, o algoritmo BFS satisfará $D_v \geq \delta(s, v) \forall v \in V$.*

Prova: Utiliza-se a indução em relação ao número de operações de enfileiramento (*enqueue*). A hipótese indutiva é $D_v \geq \delta(s, v) \forall v \in V$.

A base da indução é a situação imediatamente após s ser enfileirado na linha 7 do Algoritmo 19. A hipótese indutiva se mantém válida nesse momento porque $D_s = 0 = \delta(s, s)$ e $D_v = \infty \geq \delta(s, v) \forall v \in V \setminus \{s\}$.

Para o passo da indução, considere um vértice v não-visitado (C_v =**false**) que é descoberto depois do último desempinhamento. Consideramos que o vértice desempunhado é $u \in V$. A hipótese da indução implica que $D_u \geq \delta(s, u)$. Pela atribuição da linha 13 e pelo Lema 5.3.1, obtem-se

$$D_v = D_u + 1 \geq \delta(s, u) + 1 \geq \delta(s, v). \tag{5.8}$$

Então, o vértice v é enfileirado e nunca será enfileirado novamente porque ele também é marcado como visitado e as operações entre as linhas 12 e 15 são apenas executadas para vértices não-visitados. Desse modo, o valor de D_v nunca muda novamente e a hipótese de indução é mantida. ■

Lema 5.3.3. *Suponha que durante a execução do algoritmo de busca em largura (Algoritmo 19) em um grafo $G = (V, E)$, a fila Q contenha os vértices $(v_1, v_2, \ldots, v_r)$, onde v_1 é o início da fila e v_r é o final da fila. Então, $D_{v_r} \leq D_{v_1} + 1$ e $D_{v_i} \leq D_{v_{i+1}}$ para todo $i \in \{1, 2, \ldots, r-1\}$.*

Prova: A prova é realizada por indução relacionada ao número de operações de fila.

Para a base da indução, imediatamente antes do laço de repetição (antes da linha 8), têm-se apenas o vértice s na fila. O lema se mantém nessa condição.

Para o passo da indução, deve-se provar que o lema se mantém para depois do desenfileiramento quanto do enfileiramento de um vértice. Se o início v_1 é desenfileirado,

v_2 torna-se o início. Pela hipótese de indução, $D_{v_1} \leq D_{v_2}$. Então $D_{v_r} \leq D_{v_1} + 1 \leq D_{v_2} + 1$. Assim, o lema prossegue com v_2 no início.

Quando enfileira-se um vértice v (linha 15), ele se torna v_{r+1}. Nesse momento, já se removeu da fila o vértice u cujo as adjacências estão sendo analisadas, e pela hipótese de indução, o novo início v_1 deve ter $D_{v_1} \geq D_u$. Assim, $D_{v_{i+1}} = D_v = D_u + 1 \leq D_{v_1} + 1$. Pela hipótese indutiva, têm-se $D_{v_r} \leq D_u + 1$, portanto $D_{v_r} \leq D_u + 1 = D_v = D_{v_{i+1}}$ e o lema se mantém quando um vértice é enfileirado. ■

Corolário 5.3.4. *Suponha que os vértices v_i e v_j sejam enfileirados durante a execução do algoritmo de busca em largura (Algoritmo 19) e que v_i seja enfileirado antes de v_j. Então, $D_{v_i} \leq D_{v_j}$ no momento que v_j é enfileirado.*

Prova: Imediata pelo Lema 5.3.3 e pela propriedade de que cada vértice recebe um valor D finito no máximo uma vez durante a execução do algoritmo. ■

Teorema 5.3.5. *Seja $G = (V, E)$ um grafo orientado ou não-orientado, e suponha que o algoritmo de busca em largura (Algoritmo 19) seja executado em G partindo de um dado vértice $s \in V$. Então, durante sua execução, o algoritmo descobre todo o vértice $v \in V$ atingível por s. Ao findar sua execução, o algoritmo retornará a distância mínima entre s e $v \in V$, então $D_v = \delta(s, v) \forall\, v \in V$.*

Prova: Por contradição, suponha que algum vértice receba um valor d não igual à distância de seu caminho mínimo. Seja v um vértice com $\delta(s, v)$ mínimo que recebe tal valor d incorreto. O vértice v não poderia ser s, pois o algoritmo define $D_s = 0$, o que estaria correto. Então deve-se encontrar um outro $v \neq s$. Pelo Lema 5.3.2, $D_v \geq \delta(s, v)$ e portanto, temos $D_v > \delta(s, v)$. O vértice v deve poder ser visitado a partir de s, se não puder, $\delta(s, v) = \infty \geq D_v$. Seja u o vértice imediatamente anterior a v em um caminho mínimo de s a v, de modo que $\delta(s, v) = \delta(s, u) + 1$. Como $\delta(s, u) < \delta(s, v)$, e em razão de selecionar-se v, têm-se $D_u = \delta(s, u)$. Reunindo essas propriedades, têm-se

$$D_v > \delta(s, v) = \delta(s, u) + 1 = D_u + 1. \tag{5.9}$$

Considere o momento que o algoritmo opta por desenfileirar o vértice u de Q. Nesse momento, o vértice v pode ter sido não-visitado, visitado e está na fila, ou visitado e já foi removido da fila. O restante da prova trabalha em cada um desses casos:

- Se v é não-visitado (C_v =**false**), então a operação na linha 13 define $D_v = D_u + 1$, contradizendo o que é dito na Equação 5.9.
- Se v já foi visitado (C_v =**true**) e foi removido da fila, pelo Corolário 5.3.4, têm-se $D_v \leq D_u$ que também contradiz o que é dito na Equação 5.9.
- Se v já foi visitado e permanece na fila, quando v fora enfileirado w era o vértice antecessor imediato no caminho até v, logo $D_v = D_w + 1$. Considere também que w já foi desenfileirado. Porém, pelo Corolário 5.3.4, $D_w \leq D_u$, então, temos $D_v = D_w + 1 \leq D_u + 1$, contradizendo a Equação 5.9.

■

Árvores em Largura

O algoritmo de busca em largura (Algoritmo 19) criar uma árvore de busca em largura à medida que efetua busca no grafo $G = (V, E)$. Também chamada de "subgrafo dos predecessores", uma árvore de busca em lagura pode ser definida como $G_\pi = (V_\pi, E_\pi)$, na qual $V_\pi = \{v \in V : A_v \neq \textbf{null}\} \cup \{s\}$ e $E_\pi = \{(A_v, \pi, v) : v \in V_\pi \backslash \{s\}\}$.

5.3.2 Busca em Profundidade

A busca em profundidade (Depth-First Search - DFS) realiza a visita a vértices cada vez mais profundos/distantes de um vértice de origem s até que todos os vértices sejam visitados. Parte-se a busca do vértice mais recentemente descoberto do qual ainda saem arestas inexploradas. Depois que todas as arestas foram visitadas no mesmo caminho, a busca retorna pelo mesmo caminho para passar por arestas inexploradas. Quando não houver mais arestas inexploradas a busca em profundidade pára (CORMEN et al., 2012).

O Algoritmo 20 apresenta um pseudo-código para a busca em profundidade. Note que no lugar de usar uma fila, como na busca em largura (vide Algoritmo 19, utiliza-se uma pilha. Os arranjos C_v, T_v, e A_v $\forall v \in V$ são respectivamente o arranjo de marcação de visitados, do tempo de visita e do vértice antecessor à visita.

Algoritmo 20: Busca em profundidade.

```
   Input  :um grafo G = (V, E), vértice de origem s ∈ V
   // configurando todos os vértices
 1 C_v ← false ∀v ∈ V
 2 T_v ← ∞ ∀v ∈ V
 3 A_v ← null ∀v ∈ V
   // configurando o vértice de origem
 4 C_s ← true
 5 tempo ← 0
   // preparando fila de visitas
 6 S ← Pilha()
 7 S.push(s)
   // propagação das visitas
 8 while S.empty() = false do
 9     tempo ← tempo +1
10     u ← S.pop()
11     T_u ← tempo
12     foreach v ∈ N(u) do
13         if C_v = false then
14             C_v ← true
15             A_v ← u
16             S.push(v)
17 return (C, T, A)
```

Cormen et al. (2012) afirma que é mais comum realizar a busca em profundidade de várias fontes. Desse modo, seu livro reporta um algoritmo que sempre que um subgrafo conexo é completamente buscado, parte-se de um outro vértice de origem não-visitado ainda (um vértice não atingível por s).

5.3.2.1 Complexidade da Busca em Profundidade

Da mesma maneira que a complexidade da busca em largura, a busca em profundidade possui complexidade $O(|V| + |E|)$. As operações da pilha resultariam tempo $O(|V|)$.

Muitos arestas/arcos incidem em vértices já visitados, então inclui-se na complexidade de uma DFS a varredura de todas as adjacências, que demandaria $\Theta(|E|)$.

5.3.3 Aplicações de Buscas em Profundidade

5.3.3.1 Componentes Fortemente Conexas

Um grafo conexo é aquele no qual há um caminho entre todos os pares de vértices. É dita uma componente fortemente conexa de um grafo dirigido não-ponderado $G = (V, A)$ é um conjunto máximo de vértices $C \subseteq V$, tal que para todo o par de vértices u, v em C têm-se $u \rightsquigarrow v$ e $v \rightsquigarrow u$.

Um algoritmo para identificar as Componentes Fortemente Conexas é relatado por Cormen et al. (2012) e apresentado aqui no Algoritmo 21. Nele, utiliza-se duas vezes a busca em profundidade sugerida também por Cormen et al. (2012) (Algoritmos 22 (DFS) e 23 (DFS-Visit)). Primeiramente, faz-se a busca em largura para descobrir os caminhos de todos os vértices para todos os outros. Depois, percorre-se os mesmos caminhos em um grafo transposto (G^T). As árvores representadas como os antecessores

em A^T trarão a resposta: cada árvore é uma componente fortemente conexa.

Algoritmo 21: Algoritmo de Componentes-Fortemente-Conexas

Input : um grafo dirigido não ponderado $G = (V, A)$

```
/* Chamar a DFS (do Algoritmo 22) para computar os tempos de término para
   cada vértice                                                          */
```

1 $(C, T, A', F) \leftarrow \text{DFS}(G)$

```
/* Criar grafo transposto de G, chamado de G^T.                          */
```

2 $A^T \leftarrow \{\}$

3 **foreach** $(u, v) \in A$ **do**

4 $\quad A^T \leftarrow A^T \cup \{(v, u)\}$ `/* Inverte-se todos os arcos para` G^T`.` `*/`

5 $G^T \leftarrow (V, A^T)$

```
/* Chamar a DFS (do Algoritmo 22) alterado para que ele execute o laço da
   linha 6, selecionando vértices em ordem decrescente de F              */
```

6 $(C^T, T^T, A'^T, F^T) \leftarrow \text{DFS-adaptado}(G^T)$

```
/* dar saída de cada árvore na floresta em profundidade em A^T como uma
   componente fortemente conexa.                                         */
```

7 **return** A^T

Algoritmo 22: DFS de Cormen et al. (2012).

Input : um grafo dirigido não ponderado $G = (V, E)$

```
// Configurando todos os vértices
```

1 $C_v \leftarrow$ **false** $\forall v \in V$

2 $T_v \leftarrow \infty \; \forall v \in V$

3 $F_v \leftarrow \infty \; \forall v \in V$

4 $A_v \leftarrow$ **null** $\forall v \in V$

```
// configurando o tempo de início
```

5 tempo $\leftarrow 0$

6 **foreach** $u \in V$ **do**

7 $\quad$ **if** $C_u =$ ***false*** **then**

```
        // DFS-Visit é especificado no Algoritmo 23
```

8 $\quad\quad$ DFS-Visit(G, u, C, T, A, F, tempo)

9 **return** (C, T, A, F)

Algoritmo 23: DFS-Visit de Cormen et al. (2012).

Input : um grafo $G = (V, E)$, vértice de origem $v \in V$, e os vetores C, T, A e F, e uma variável *tempo* $\in \mathbb{Z}_*^+$

```
1 C_v ← true
2 tempo ← tempo +1
3 T_v ← tempo
4 foreach u ∈ N⁺(v) do
5     if C_u = false then
6         A_u ← v
7         DFS-Visit(G, u, C, T, A, F, tempo)
8 tempo ← tempo +1
9 F_v ← tempo
```

Complexidade do Algoritmo de Componentes Fortemente Conexas

A complexidade de tempo computacional do Algoritmo 21 é dependente da complexidade de tempo do algoritmo DFS de Cormen et al. (2012) (Algoritmos 22 (DFS) e 23 (DFS-Visit)). Para o algoritmo DFS, as instruções das linhas 1 a 4 demandam uma quantidade de instruções igual a $\Theta(|V|)$. O laço entre as linhas 6 a 8 é executado por um número de iterações dependente do número de vértices ($\Theta(|V|)$). O procedimento DFS-Visit é invocado apenas uma vez para cada vértice, graças ao vetor de visitados C. Como nesse procedimento as adjacências de cada vértice são visitadas, há também uma dependência do número de arcos saintes em cada vértice. Logo, a complexidade computacional do Algoritmo 22, e consequentemente, a do algoritmo de Componentes Fortemente Conexas, é $\Theta(|V| + |E|)$.

Corretude do Algoritmo de Componentes Fortemente Conexas

Propriedades de Buscas em Profundidade

Teorema 5.3.6. *Em qualquer busca em profundidade em um grafo dirigido ou não $G = (V, E)$, para quaisquer dois vértices u e v, exatamente uma das três condições é válida:*

- *Os intervalos $[T_u, F_u]$ e $[T_v, F_v]$ são completamente disjuntos, e nem u nem v é*

descendente do outro na floresta de profundidade.

- *O intervalo* $[T_u, F_u]$ *está contido inteiramente dentro do intervalo* $[T_v, F_v]$ *e* u *é um descendente de* v *em uma árvore de profundidade.*

- *O intervalo* $[T_v, F_v]$ *está contido inteiramente dentro do intervalo* $[T_u, F_u]$ *e* v *é um descendente de* u *em uma árvore de profundidade.*

Prova: A prova começará com o caso de $T_u < T_v$. Para isso, consideramos dois subcasos: $T_v < F_u$ ou não. O primeiro subcaso ocorre quando $T_v < F_u$, portanto v foi descoberto quando $C_u =$**true**, ou seja, v foi descoberto mais recentemente que u, o que implica que v é descendente de u. Ao findar a busca de u (linha 9 do Algoritmo 23), todas as arestas de u foram exploradas, e consequentemente $[T_v, F_v]$ está completamente contido no intervalo $[T_u, F_u]$.

Ainda considerando $T_u < T_v$, mas no subcaso de $F_u < T_v$, devido a $T_w < F_w$ para todo $w \in V$, $T_u < F_u < T_v < F_v$, assim os intervalos $[T_u, F_u]$ e $[T_v, F_v]$ são disjuntos. Como os intervalos são disjuntos, nenhum vértice foi descoberto enquanto o outro ainda não havia findado (linha 9 do Algoritmo 23), então nenhum é descendente do outro.

O caso de $T_v < T_u$ é semelhante, com papéis de u e v invertidos no argumento anterior. ■

Corolário 5.3.7. *O vértice* v *é um descendente adequado do vértice* u *na floresta em profundidade para um grafo dirigido ou não* $G = (V, E)$ *sse* $T_u < T_v < F_v < F_u$.

Prova: Imediata pelo Teorema 5.3.6. ■

Teorema 5.3.8. *Em uma floresta em profundidade de um grafo dirigido ou não* $G = (V, E)$*, o vértice* v *é um descendente do vértice* u *sse no momento* T_u*, em que uma busca descobre* u*, há um caminho de* u *a* v *inteiramente de vértices* w *marcados com* $C_w =$***false***.

Prova: Se $v = u$, então o caminho de u a v não possui vértices além dele mesmo.

Agora, supõe-se que v seja um descendente próprio de u, que ainda tem C_v =**false** quando se define o valor de T_u. Pelo Corolário 5.3.7, $T_u < T_v$ e portanto C_v =**false** no tempo T_u. Visto que v pode ser descendente de u, todos os vértices w num caminho simples de u até v tem C_w =**false**.

Agora, supõe-se que haja um caminho de vértices w marcados com C_w =**false** no caminho de u a v no tempo T_u, mas v não se torna descendente de u na árvore de profundidade. Sem prejuízo, considera-se que todo o vértice exceto v ao longo do caminho se torne um descendente de u. Seja w o predecessor de v no camino, de modo que w seja um descendente de u. Pelo Corolário 5.3.7, $F_w < F_u$. Como v tem que ser descoberto depois de u ser descoberto, mas antes de w ser terminado, têm-se $T_u < T_v < F_w < F_u$. Então o Teorema 5.3.6 implica que o intervalo $[T_v, F_v]$ está contido no intervalo $[T_u, F_u]$. Pelo Corolário 5.3.7, v deve ser descendente de u. ■

Propriedades de Componentes Fortemente Conexas

Para as propriedades abaixo, considere que:

- $d(S) = \min_{v \in S}\{T_v\}$;
- $f(S) = \max_{v \in S}\{F_v\}$;

Lema 5.3.9. *Considerando C e C' componentes fortemente conexas distintas em um grafo dirigido $G = (V, A)$, seja $u, v \in C$ e $u', v' \in C'$ e suponha que G tenha um caminho $u \rightsquigarrow u'$. Então, G não pode conter um caminho $v' \rightsquigarrow v$.*

Prova: Se G possui um caminho $v' \rightsquigarrow v$, então contém os caminhos $u \rightsquigarrow u' \rightsquigarrow v'$ e $v' \rightsquigarrow v \rightsquigarrow u$ em G. Assim, u e v' podem ser visitados um a partir do outro, o que contradiz a hipótese de que C e C' são componentes fortemente conexas distintas. ■

Lema 5.3.10. *Sejam C e C' componentes fortemente conexas distintas no grafo dirigido $G = (V, A)$. Suponha que haja um arco $(u, v) \in A$, na qual $u \in C$ e $v \in C'$. Então, $f(C) > f(C')$.*

Prova: Considera-se dois casos dependendo qual das componentes fortemente conexas (C ou C') têm o primeiro vértice descoberto na busca em profundidade.

Se $d(C) < d(C')$, seja x o primeiro vértice descoberto em C. No tempo T_x, todos os vértices em C e C' são não visitados (C_v =**false** para todo $v \in C \cup C'$). Pode-se afirmar então que há um caminho em x a w para qualquer $w \in C'$ por causa do arco $(u,v) \in A$, então $x \rightsquigarrow u \rightarrow v \rightsquigarrow w$. Pelo Teorema 5.3.8, todos os vértices em C e C' se tornam descendentes de x ma árvore de profundidade. Pelo Corolário 5.3.7, x tem o tempo de término mais recente que qualquer um de seus descendentes, portanto $F_x = f(C) > f(C')$;

Se $d(C) > d(C')$, seja y o primeiro vértice descoberto em C'. No tempo T_y, todos os vértices w em C' têm C_w =**false** e G contém um caminho de y a cada vértice em C' formado apenas por vértices z com C_z =**false**. Pelo Teorema 5.3.8, todos os vértices em C' se tornam descendentes de y na árvore de profundidade. Pelo Teorema 5.3.8, $F_y = F(C')$. No tempo T_y, todos os vértices w em C têm C_w =**false**. Como existe um arco (u,v) de C a C', o Lema 5.3.9 implica que não pode haver um caminho de C' a C. Consequentemente, nenhum vértice em C pode ser visitado por y. Portanto, no tempo F_y, todos os vértices w em C ainda tem C_w =**false**. Assim, para qualquer vértice em $w \in C$, têm-se $F_w > F_y$, o que implica que $f(C) > f(C')$. ■

Corolário 5.3.11. *Considerando C e C' componentes fortemente conexas distintas no grafo dirigido $G = (V,A)$, suponha que hava um arco $(u,v) \in A^T$, na qual $u \in C$ e $v \in C'$. Então, $f(C) < f(C')$.*

Prova: Como $(u,v) \in A^T$, têm-se $(v,u) \in A$. Visto que as componentes fortemente conexas em G^T são as mesmas, o Lema 5.3.10 implica que $f(C) < f(C')$. ■

Teorema 5.3.12. *O Algoritmo 21 (de Componentes-Fortemente-Conexas) encontra corretamente as componentes fortemente conexas de um grafo dirigido $G = (V,A)$ dado como sua entrada.*

Prova: A prova é realizada por indução em relação ao número de árvores de busca encontradas na busca em profundidade de G^T na linha 6. Cada árvore forma uma componente fortemente conexa.

A hipótese da indução é que as primeiras k árvores produzidas na linha 6 são componentes fortemente conexas.

A base da indução, quando $k = 0$, é trivial.

No passo da indução, supõe-se que cada uma das k primeiras árvores de profundidade produzidas na linha 6 é uma componente fortemente conexa, e consideramos a $(k+1)$-ésima árvore produzida. Seja u a raiz dessa árvore, e supondo que u esteja na componente fortemente conexa C. Como resultado do modo que se escolhe a raiz da árvore na linha 6, $F_u - f(C) > f(C')$ para qualquer componente fortemente conexa C' exceto C que ainda tenha de ser visitada. Pela hipótese de indução, no momento da busca de u na árvore de profundidade, todos os vértices w em C tem C_w =**false**. Então, pelo Teorema 5.3.8, todos os outros vértices de C são descendentes de u nessa árvore de profundidade. Além disso, pela hipótese de indução e pelo Corolário 5.3.11, qualquer arco em G^T que saem de C devem ir até componentes fortemente conexas que já foram visitadas. Assim, nenhum vértice em uma componente fortemente conexa, exceto C, será um descendente de u durante a busca em profundidade de G^T. Portanto, os vértices da árvore de busca em profundidade em G^T enraizada em u formam exatamente uma componente fortemente conexa, o que conclui o passo de indução e a prova. ■

5.3.3.2 Ordenação Topológica

A ordenação topológica no contexto de grafos, recebe um grafo acíclico dirigido $G = (V, A)$ e ordena linearmente todos os vértices tal que se existe um arco $(u, v) \in A$ então u aparece antes de v na ordenação. O algoritmo de Ordenação Topológica tem como base uma busca em largura com a adição de uma lista O para inserir os vértices, como pode ser visto no Algoritmo 25. Os vértices são inseridos sempre no início da lista O

logo que o algoritmo termina de visitá-lo (linha 10 do Algoritmo 25).

Algoritmo 24: DFS para Ordenação Topológica

Input : um grafo dirigido não ponderado $G = (V, A)$

```
// Configurando todos os vértices
```

1 $C_v \leftarrow$ **false** $\forall v \in V$

2 $T_v \leftarrow \infty \; \forall v \in V$

3 $F_v \leftarrow \infty \; \forall v \in V$

```
// configurando o tempo de início
```

4 tempo $\leftarrow 0$

```
// Criando lista com os vértices ordenados topologicamente
```

5 $O \leftarrow ()$

6 **foreach** $u \in V$ **do**

7 **if** $C_u =$ ***false*** **then**

```
// DFS-Visit-OT é especificado no Algoritmo 25
```

8 DFS-Visit-OT(G, u, C, T, F, tempo, O)

9 **return** O

Algoritmo 25: DFS-Visit-OT.

Input : um grafo $G = (V, E)$, vértice de origem $v \in V$, e os vetores C, T e F, e uma variável *tempo* $\in \mathbb{Z}_*^+$, uma lista O

1 $C_v \leftarrow$ **true**

2 tempo $\leftarrow$ tempo $+1$

3 $T_v \leftarrow$ tempo

4 **foreach** $u \in N^+(v)$ **do**

5 **if** $C_u =$ ***false*** **then**

6 DFS-Visit-OT(G, u, C, T, F, tempo, O)

7 tempo $\leftarrow$ tempo $+1$

8 $F_v \leftarrow$ tempo

```
// Adiciona o vértice v no início da lista O
```

9 $O \leftarrow (v) \cup O$

Complexidade da Ordenação Topológica

Como a inserção no início de uma lista demanda tempo $O(1)$, a complexidade de tempo da Ordenação Topológica de um grafo acíclico é dependente da Busca em Profundidade. Logo, a complexidade da Ordenação Topológica é $O(|V| + |A|)$. 0

Corretude da Ordenação Topológica

Lema 5.3.13. *Um grafo dirigido $G = (V, A)$ é acíclico sse uma busca em profundidade de G não produz nenhum arco de retorno.*

Prova: Supondo que uma busca em profundidade produza um arco de retorno (u, v). Então, o vértice v é um ascendente (ancestral) do vértice u na floresta em profundidade. Assim, G contém um caminho de v a u, e o arco (u, v) completa o ciclo.

Supondo que G contenha um ciclo c, mostrou-se que uma busca em profundidade de G produz um arco de retorno. Seja v o primeiro vértice a ser descoberto em c e seja (u, v) o arco precedente em c. No tempo T_v, os vértices em c formam um caminho de vértices w com C_w =**false** de v a u. Pelo Teorema 5.3.8, o vértice u se torna um descendente de v na floresta em profundidade. Então (u, v) é um arco de retorno. ■

Teorema 5.3.14. *O Algoritmo 24 produz uma ordenação topológica de um grafo dirigido acíclico $G = (V, A)$ dado como entrada.*

Prova: Supondo que o algoritmo seja executado sobre um determinado grafo dirigido acíclico $G = (V, A)$ para determinar os tempos de término para seus vértices. É suficiente mostrar que, para qualquer par de vértices distintos $u, v \in V$, se G contém um arco de u a v, então $F_v < F_u$. Considere qualquer arco (u, v) explorado no algoritmo. Quando esse arco é explorado, v ainda não foi visitado (por DFS-Visit-OT), já que v é ascendente (ancestral) de u e (u, v) é um arco de retorno, o que contradiz o Lema 5.3.13. Portanto, v já deve ter C_v =**false** ou já foi visitado. Se v tem C_v =**false**, ele se torna um descendente de u e $F_v < F_u$. Se v já foi visitado F_v já foi definido, então $F_v < F_u$. Assim, para qualquer arco $(u, v) \in A$, têm-se $F_v < F_u$. ■

5.4 Caminhos Mínimos

Em um problema de Caminho Mínimo, há um grafo ponderado orientado ou não $G = (V, E, w)$, onde V é o conjunto de vértices, E é o conjunto de arcos ou arestas, e $w : E \to \mathbb{R}$ é a função que representa o peso entre dois vértices (distância ou custo das arestas). Para um caminho $p = \langle v_1, v_2, \ldots, v_k \rangle$ seu peso é dado por $w(p) = \sum_{i=2}^{k} w\big((v_{i-1}, v_i)\big)$ (CORMEN et al., 2012).

O peso de um caminho mínimo de u a v é dado por

$$\delta(u, v) = \begin{cases} \textbf{min}\{w(p) : u \overset{p}{\rightsquigarrow} v\}, & \text{se há um caminho de } u \text{ para } v\,, \\ \infty, & \text{caso contrário.} \end{cases} \tag{5.10}$$

Há algumas variantes para os problemas de caminho mínimo (CORMEN et al., 2012):

- Problema de caminhos mínimos de fonte única: dado um grafo ponderado $G = (V, E, w)$ e um vértice de origem $s \in V$, encontrar o caminho de custo $\delta(s, v)$ para todo o $v \in V$;

- Problema de caminhos mínimos para um destino:dado um grafo ponderado $G = (V, E, w)$, um vértice de destino $t \in V$, determinar o caminho de custo $\delta(v, t)$ para todo o $v \in V$;

- Problema de caminhos mínimos para um par: dado um grafo ponderado $G = (V, E, w)$, um vértice de origem $s \in V$ e um vértice de destino t, determinar o caminho de custo $\delta(s, t)$;

- Problema de caminhos mínimos para todos os pares: dado um grafo ponderado $G = (V, E, w)$, encontrar o caminho de curso $\delta(u, v)$ para todo o par $u, v \in V$.

Pesos Negativos

Os problemas de caminho mínimo geralmente operam sem erros em grafos com pesos negativos. Uma exceção a isso é quando há um ciclo com peso negativo. Nesse caso, nunca haverá um peso definido, pois é sempre possível diminuir o peso total do caminho percorrendo o ciclo mais uma vez.

Inicialização e Relaxamento

Os Algoritmos 26 e 27 são utilizados em diversos algoritmos de resolução de caminhos mínimos. A estrutura de dados D é referente a estimativa de caminho que será obtida ao longo da execução de um caminho mínimo para cada vértice $v \in V$. A estrutura de dados A é utilizada para identificar o vértice anterior em cada caminho mínimo para um vértice $v \in V$.

Algoritmo 26: Inicialização de G.

Input : um grafo $G = (V, E, w)$, um vértice de origem $s \in V$

```
// inicialização
```

1 $D_v \leftarrow \infty \; \forall v \in V$

2 $A_v \leftarrow$ **null** $\forall v \in V$

3 $D_s \leftarrow 0$

4 **return** (D, A)

Algoritmo 27: Relaxamento de v.

Input : um grafo $G = (V, E, w)$[a], $(u, v) \in E$, A, D

1 **if** $D_v > D_u + w\big((u, v)\big)$ **then**

2 $\quad D_v \leftarrow D_u + w\big((u, v)\big)$

3 $\quad A_v \leftarrow u$

[a] Para o caso de G ser não-dirigido, deve-se realizar o relaxamento em (u, v) e (v, u) para uma aresta $\{u, v\} \in E$, ou seja, deve-se realizar o relaxamento nos dois sentidos.

5.4.1 Propriedades de Caminhos Mínimos

5.4.1.1 Propriedade de subcaminhos de caminhos mínimos o são

Lema 5.4.1. *Dado um grafo ponderado $G = (V, E, w)$, um caminho mínimo entre v_1 e v_k $p = \langle v_1, v_2, \ldots, v_k \rangle$, suponha que para quaisquer i e j, $1 \leq i \leq j \leq k$. Todo o subcaminho de p chamado de $p_{ij} = \langle v_i, v_{i+1}, \ldots, v_j \rangle$ é um caminho mínimo de v_i a v_j.*

Prova: Se o caminho p for decomposto em $v_1 \overset{p_{1i}}{\rightsquigarrow} v_i \overset{p_{ij}}{\rightsquigarrow} v_j \overset{p_{jk}}{\rightsquigarrow} v_k$, têm-se $w(p) = w(p_{0j}) + w(p_{ij}) + w(p_{jk})$. Suponha que exista um caminho p'_{ij} de v_i a v_j com peso $w(p'_{ij}) < w(p_{ij})$. Então, $v_1 \overset{p_{1i}}{\rightsquigarrow} v_i \overset{p'_{ij}}{\rightsquigarrow} v_j \overset{p_{jk}}{\rightsquigarrow} v_k$ é um caminho de v_1 a v_k cujo o peso $w(p) = w(p_{0j}) + w(p'_{ij}) + w(p_{jk})$ é menor do que $w(p)$, o que contradiz a hipótese de que p seja um caminho mínimo de v_1 a v_k. ■

5.4.1.2 Propriedade de desigualdade triangular

Lema 5.4.2. *Seja $G = (V, E, w)$ um grafo ponderado e $s \in V$ um vértice de origem, então para todas as arcos/arestas $(u, v) \in E$ têm-se*

$$\delta(s, v) \leq \delta(s, u) + w\big((u, v)\big). \tag{5.11}$$

Prova: Suponha que p seja um caminho entre s e v. Então, p não tem peso maior do que qualquer outro caminho de s a v. Especificamente, p não possui peso maior que o caminho de s até o vértice u que utiliza a aresta/arco (u, v) para atingir o destino v. ■

5.4.1.3 Propriedade de limite superior

Lema 5.4.3. *Seja $G = (V, E, w)$ um grafo ponderado dirigido ou não com a função de peso $w : E \to \mathbb{R}$. Seja $s \in V$ o vértice de origem, considera-se também o grafo G inicializado (Algoritmo 26). Então, $D_v \geq \delta(s, v)$ para todo $v \in V$, e esse invariante é mantido para qualquer sequência de etapas de relaxamentos em G (Algoritmo 27). Além disso, tão logo D_v alcance seu limite inferior $\delta(s, v)$, nunca mais se altera.*

Prova: Prova-se que o invariante $D_v \geq \delta(s,v)$ para todo o vértice $v \in V$ por indução em relação ao número de etapas de relaxamento.

Para a base da indução, $D_v \geq \delta(s,v)$ é verdadeiro após a inicialização (Algoritmo 26), pois esse procedimento define que $D_v = \infty$ para todo $v \in V \setminus \{s\}$, ou seja, $D_v \geq \delta(s,v)$ mesmo que v seja inatingível em um caminho mínimo a partir de s. Nesse momento $D_s = 0 \geq \delta(s,s)$, observando que $\delta(s,s) = \infty$ caso s participa de um ciclo negativo.

Para o passo da indução, considere o relaxamento de uma aresta/arco (u,v). Pela hipótese de indução, $D_x \geq \delta(s,x)$ para todo o $x \in V$ antes do relaxamento. O único valor de D que pode mudar é D_v. Se ele mudar, têm-se

$$\begin{aligned} D_v &= D_u + w\big((u,v)\big) \\ &\geq \delta(s,u) + w\big((u,v)\big) \text{ (pela hipótese da indução)} \\ &\geq \delta(s,v) \text{ (pela desigualdade triangular,} \\ &\qquad \text{Lema 5.4.2)} \end{aligned} \tag{5.12}$$

e, portanto o invariante é mantido.

Para demonstrar que D_v não se altera depois que $D_v = \delta(s,v)$, por ter alcaçado seu limite inferior, D_v não pode diminuir porque $D_v \geq \delta(s,v)$ e não pode aumentar porque o relaxamento não aumenta valores de D. ■

5.4.1.4 Propriedade de inexistência de caminho

Corolário 5.4.4. *Supõe-se que, em um grafo $G = (V,E,w)$ ponderado dirigido ou não, nenhum caminho conecte o vértice de origem $s \in V$ a um vértice $v \in V$. Então, depois que o grafo G é inicializado (Algoritmo 26), temos $D_v = \delta(s,v) = \infty$ e essa desigualdade é mantida como um invariante para qualquer sequência de etapas de relaxamento (Algoritmo 27) nas arestas de G;*

Prova: Pela propriedade de limite superior (Lema 5.4.3), têm-se sempre $\infty = \delta(s,v) \leq D_v$, portanto $D_v = \infty = \delta(s,v)$. ■

5.4.1.5 Propriedade de convergência

Lema 5.4.5. *Seja $G = (V, E, w)$ um grafo ponderado dirigido ou não, $s \in V$ um vértice de origem e $s \rightsquigarrow u \rightarrow v$ um caminho mínimo de s a v em G. Suponha que G seja inicializado (Algoritmo 26) e depois uma sequência de etapas de relaxamento (Algoritmo 27) é executado para todas as arestas/arcos de G. Se $D_u = \delta(s, u)$ em qualquer tempo anterior da chamada, então $D_u = \delta(s, u)$ igual em toda a chamada.*

Prova: Pela propriedade do limite superior (Lema 5.4.3), se $D_u = \delta(s, u)$ em algum momento antes do relacamento da aresta/arco (u, v), então essa igualdade se mantém válida a partir de sua definição. Em particular, após o relaxamento (Algoritmo 27) da aresta/arco (u, v), têm-se:

$$\begin{aligned} D_v &\leq D_u + w\big((u, v)\big) \text{ pelo Corolário 5.4.4)} \\ &= \delta(s, u) + w\big((u, v)\big) \\ &= \delta(s, v) \text{ pelo Lema 5.4.1.)} \end{aligned} \tag{5.13}$$

Pela propriedade do limite superior (Lema 5.4.3) $D_v \geq \delta(s, v)$, da qual concluímos que $D_v = \delta(s, v)$, e essa igualdade é mantida daí em diante. ■

5.4.1.6 Propriedade de relaxamento de caminho

Lema 5.4.6. *Seja $G = (V, E, w)$ um grafo ponderado dirigido ou não e $s \in V$ um vértice de origem. Considere qualquer caminho mínimo $p = \langle v_1, v_2, \ldots v_k \rangle$ de $s = v_1$ a v_k. Se G é inicializado (Algoritmo 26) e depois ocorre uma sequência de etapas de relaxamento (Algoritmo 27) que inclui, pela ordem, relaxar as arestas/arcos $(v_1, v_2), (v_2, v_3), \ldots (v_{k-1}, v_k)$, então $D_k = \delta(s, v_k)$ depois desses relaxamentos e todas as vezes daí em diante. Essa propriedade se mantém válida, não importa quais outros relaxamentos forem realizados.*

Prova: Esta prova é realizada por indução, na qual têm-se $D_{v_i} = \delta(s, v_i)$ depois que o i-ésimo aresta/arco do caminho p é relaxado.

Para a base, $i = 1$ antes que quaisquer arestas/arcos em p sejam relaxados. Têm-se $D_{v_1} = D_s = 0 = \delta(s,s)$ pela inicialização. Pela propriedade do limite superior (Lema 5.4.3) o valor D_s nunca se altera depois da inicialização.

Pelo passo da indução, supõe-se que $v_{i-1} = \delta(s, v_{i-1})$ e examina-se o que acontece quando se relaxa a aresta (v_{i-1}, v_i). Pela propriedade de convergência (Lema 5.4.5), após o relaxamento dessa aresta, têm-se $D_v = \delta(s, v_i)$ e essa igualdade é mantida todas as vezes depois disso. ■

5.4.1.7 Propriedade de relaxamento e árvores de caminho mínimo

Lema 5.4.7. *Seja $G = (V, E, w)$ um grafo ponderado dirigido ou não e $s \in V$ um vértice de origem, suponha que G não possua um ciclo de peso negativo que possa ser atingido por s. Então, depois que o grafo G é inicializado (Algoritmo 26), o subgrafo dos predecessores $G_\pi = (V_\pi, E_\pi)$ forma uma árvore enraizada em s, e qualquer sequência de etapas de relaxamento em arestas em G (Algoritmo 27) mantém essa propriedade invariante.*

Prova: Inicialmentem o único vértice em G_π é o vértice s, e o lema é trivialmente verdade. Considere um subgrafo dos predecessores G_π que surja depois de uma sequência de etapas de relavamento.

Primeiro, prova-se que o subgrafo é acíclico. Suponha por contradição que alguma etapa de relaxamento cria um ciclo no grafo G_π. Seja $c = \langle v_1, v_2, \ldots, v_k \rangle$ o ciclo onde $v_1 = v_k$. Então, $A_{v_i} = v_{i-1}$ para $i = 1, 2, \ldots, k$ e, sem prejuízo de generalidade, pode-se supor que o relaxamento de arestas (v_{k-1}, v_k) criou o ciclo em G_π. Afirma-se que todos os vértices do ciclo c podem ser atingidos por s, pois cada um tem um predecessor não nulo (**null**). Portanto, uma estimativa de caminho mínimo fora atribuída a cada vértice em c quando um valor atribuídoo à A_v não foi igual a **null**. Pela propriedade do limite superior (Lema 5.4.3), cada vértice no ciclo c tem um peso de caminho mínimo infinito, o que implica que ele pode ser atingido por s.

Examina-se as estimativas de caminhos mínimos em c imediatamente antes de chamar o procedimento de relaxamento (Algoritmo 27) passando os parâmetros $G, (v_{k-1}, v_k), A, D$

e mostra-se que c é um ciclo de peso negativo, contradizendo a hipótese que G não possui um ciclo negativo que possa ser atingido por s. Imediatamente antes da chamada, têm0se $A_{v_i} = v_{i-1}$ para $i = 2,3,\ldots,k-1$. Assim, para $i = 2,3,\ldots,k-1$, a última atualização para D_{v_i} foi realizada pela atribuição $D_{v_i} \leftarrow D_{v_{i-1}} + w\big((v_{i-1}, v_i)\big)$. Se $D_{v_{i-1}}$ mudou desde então, ela diminuiu. Por essa razão, imediatamente antes da chamada de relaxamento, têm-se

$$D_{v_i} \geq D_{v_{i-1}} + w\big((v_{i-1}, v_i)\big) \forall\, i \in \{2,3,\ldots,k-1\} \tag{5.14}$$

Como A_k é alterado pela chamada, imediatamente antes têm-se também a desigualdade estrita

$$D_{v_k} > D_{v_{k-1}} + w\big((v_{k-1}, v_k)\big). \tag{5.15}$$

Somando essa desigualdade estrita com as $k-1$ desigualdades (Equação (5.14)), obtêm-se a soma das estimativas dos caminhos mínimos em torno do ciclo c:

$$\sum_{i=2}^{k} D_{v_i} > \sum_{i=2}^{k} \Big(D_{v_{i-1}} + w\big((v_{i-1}, v_i)\big)\Big) = \sum_{i=2}^{k} D_{v_{i-1}} + \sum_{i=2}^{k} w\big((v_{i-1}, v_i)\big). \tag{5.16}$$

Mas,

$$\sum_{i=2}^{k} D_{v_i} = \sum_{i=2}^{k} D_{v_{i-1}}, \tag{5.17}$$

já que cada vértice no ciclo c aparece exatamente uma vez em cada somatório. Esa desigualdade implica

$$0 > \sum_{i=2}^{k} w\big((v_{i-1}, v_i)\big). \tag{5.18}$$

Assim a soma dos pesos no ciclo c é negativa, o que dá acontradição desejada.

Agora, provamos que G_π é acíclico. Para mostrar que ele forma uma árvore enraizada em s, basta provar que há um único caminho simples de s a v em G_π para cada $v \in V_\pi$.

Primeiro, deve-se mostrar que existe um caminho de s a cada vértice em $v \in V_\pi$. Os vértices em V_π são os que têm os valores A não **null**, e o vértice s. Aqui a ideia é provar a indução que existe em um caminho de s para todos os vértices em V_π.

Para concluir a prova do lema, deve-se mostrar agora que, para qualquer vértice $v \in V_\pi$, o grafo G_π contém no máximo um caminho simples de s a v. Suponha o contrário:

que existam dois caminhos simples de s a algum outro vértice $p_1\langle s \rightsquigarrow u \rightsquigarrow x \rightarrow z \rightsquigarrow v$, e $p_2\langle s \rightsquigarrow u \rightsquigarrow y \rightarrow z \rightsquigarrow v$ onde $x \neq y$. Mas então $A_z = x$ e $A_z = y$ o que implica uma contradição, pois $x = y$. Conclui-se que G_π contém um caminho simples único de s a v e G_π forma uma árvore enraizada em s. ■

5.4.1.8 Propriedade de subgrafo dos predecessores

Lema 5.4.8. *Seja $G = (V, E, w)$ um grafo ponderado orientado ou não e um vértice de origem $s \in V$. Suponha que G não possua um ciclo de peso negativo que possa ser atingido por s. Chama-se de inicialização o procedimento do inicializado Algoritmo 26 e depois executar qualquer sequência de etapas de relaxamento de arestas de G (Algoritmo 27) que produza $D_v = \delta(s, v)$ para todo $v \in V$. Então, o subgrafo predecessor $G_\pi(V_\pi, E_\pi)$ é uma árvore de caminhos mínimos com uma raiz em s.*

Prova: Para ilustrar a primeira propriedade, deve-se mostrar V_π é o conjunto de vértices atingidos por s. Por definição, um peso de caminho mínimo $\delta(s, v)$ é finito sse v pode ser alcançado por s. Isso implica que os vértices atingidos por s possuem peso de caminho finito. Porém, um vértice $v \in V \setminus \{s\}$ recebeu um valor finito para D_v sse $A_v \neq$ **null**. Assim, os vértices em V_π são exatamente aqueles que podem ser alcançados por s.

O Lema 5.4.7 define que após a inicialização, G_π possui raiz em s e assim permanece mesmo depois de sucessivas etapas de relaxamento.

Agora, prova-se que para todo vértice em $v \in V_\pi$, o único caminho simples em G_π de s a v é o caminho mínimo de s a v em G. Seja $p = \langle v_1, v_2, \ldots v_k \rangle$, onde $v_1 = s$ e $v_k = v$. Para $i = 2, 3, \ldots, k$, temos $D_v = \delta(s, v_i)$ e também $D_v \geq D_{v_{i-1}} + w\big((v_{i-1}, v_i)\big)$, do que concluímos $w\big((v_{i-1}, v_i)\big) \leq \delta(s, v_i) - \delta(s, v_{i-1})$. A soma dos pesos ao longo de p

produz

$$\begin{aligned} w(p) &= \sum_{i=2}^{k} w\big((v_{i-1}, v_i)\big) \\ &\leq \sum_{i=2}^{k} \delta(s, v_i) - \delta(s, v_{i-1}) \\ &= \delta(s, v_k) - \delta(s, v_0) \\ &= \delta(s, v_k) \end{aligned} \tag{5.19}$$

Assim $w(p) \leq \delta(s, v_k)$. Visto que $\delta(s, v_k)$ é um limite inferior para o peso de qualquer caminho de s a v_k, conclui-se que $w(p) = \delta(s, v_k)$. Deste modo, p é um caminho mínimo de s a $v = v_k$.

■

5.4.2 Bellman-Ford

O algoritmo de Bellman-Ford resolve o problema de caminhos minimos de uma única fonte. Um pseudo-código está representado no Algoritmo 28. Como entrada para o algoritmo deve-se determinar um grafo ponderado orientado ou não $G = (V, E, w)$, onde V é o conjunto de vértices, E o conjunto de arestas/arcos e $w : E \to \mathbb{R}$, e um vértice de origem $s \in V$. O algoritmo devolve um valor booleano **false** quando não foi encontrado um ciclo de peso negativo em G. Caso contrário, retorna **true**, o antecessor de cada vértice v no caminho mínimo em A_v e a peso $\delta(s, v)$ em D_v.

O algoritmo vai progressivamente diminuindo a estimativa de peso do caminho de

s a $v \in V$ até que se obtenha o caminho mínimo e $D_v = \delta(s, v)$ para todo $v \in V$.

Algoritmo 28: Algoritmo de Bellman-Ford.

Input : um grafo $G = (V, E, w)$, um vértice de origem $s \in V$

```
// inicialização
```

1 $D_v \leftarrow \infty \; \forall v \in V$

2 $A_v \leftarrow$ **null** $\forall v \in V$

3 $D_s \leftarrow 0$

4 **for** $i \leftarrow 1$ ***to*** $|V| - 1$ **do**

5 **foreach** $(u, v) \in E$ **do**

```
// relaxamento
```

6 **if** $D_v > D_u + w\big((u, v)\big)$ **then**

7 $D_v \leftarrow D_u + w\big((u, v)\big)$

8 $A_v \leftarrow u$

9 **foreach** $(u, v) \in E$ **do**

10 **if** $D_v > D_u + w\big((u, v)\big)$ **then**

11 **return** (**false**,**null**,**null**)

12 **return** (**true**, D, A)

5.4.2.1 Complexidade de Bellman-Ford

Quanto a complexidade computacional em tempo computacional de Bellman-Ford, observando as primeiras instruções, têm-se a inicialização que demanda $\Theta(|V|)$ pois as estruturas são inicializadas para cada vértice. A partir do primeiro conjunto de laços de repetição, há o laço mais externo que repete $|V| - 1$ vezes. Para cada repetição desse laço, passa-se por cada aresta em E, logo esse primeiro conjunto de laços dita $(|V| - 1)|E|$ execuções da comparação na linha 6. O último laço de repetição, faz ao máximo $|E|$ verificações da comparação na linha 10. Então, o algoritmo de Bellman-Ford é executado no tempo computacional de $O(|V||E|)$.

5.4.2.2 Corretude de Bellman-Ford

Lema 5.4.9. *Seja $G = (V, E, w)$ um grafo ponderado e um vértice de origem $s \in V$, suponha que G não possua nenhum ciclo de peso negativo que possa ser alcançado por s. Então, depois de executar as $|V| - 1$ iterações nas linhas 4 a 8 com o algoritmo de Bellman-Ford (Algoritmo 28), têm-se $D_v = \delta(s, v)$ para todo $v \in V$.*

Prova: Prova-se o esse lema através da propriedade de relaxamento de caminho (Lema 5.4.6). Considera-se que qualquer vértice v possa ser atingido por s e seja $p = \langle v_1, v_2, \ldots, v_k \rangle$ um caminho mínimo de s a v, no qual $v_1 = s$ e $v_k = v$. Como caminhos mínimos são simples, p tem no máximo $|V| - 1$ arestas/arcos, sendo $k \leq |V| - 1$. Cada uma das $|V| - 1$ iterações do laço da linha 4 relaxa todas as $|E|$ arestas/arcos. Entre as arestas relaxadas na i-ésima iteração, para $i = 1, 2, \ldots k$, está (v_{i-1}, v_i). Então, pela propriedade de relaxamento de caminho $D_v = D_{v_k} = \delta(s, v_k) = \delta(s, v)$. ■

Corolário 5.4.10. *Seja $G = (V, E, w)$ um grafo ponderado dirigido ou não e $s \in V$ o vértice de origem, supõe-se que G não tenha nenhum ciclo negativo que possa ser atingido por s. Então, para cada vértice $v \in V$, existe um caminho de s a v sse o algoritmo de Bellman-Ford termina com $D_v < \infty$ quando é executado para G e s.*

Prova: Se $v \in V$ pode ser atingido por s, então existe uma aresta/arco (u, v). Então, $\delta(s, v) < \infty$ através da propriedade de convergência (Lema 5.4.5). ■

Teorema 5.4.11. *Considera-se o algoritmo de Bellman-Ford (Algoritmo 28) executado para um grafo $G = (V, E, w)$ e o vértice de origem $s \in V$. Se G não contém nenhum ciclo de custo negativo, que pode ser alcançado de s, então o algoritmo retorna **true**, $D_v = \delta(s, v)$ para todo $v \in V$ e o subgrafo predecessor G_π é uma árvore de caminhos mínimos com raiz em s. Se G contém um ciclo de peso negativo que possa ser atingido por s, então o algoritmo retorna **false**.*

Prova: Suponha que o grafo G não tenha um ciclo de peso negativo atingível por s. Primeiro, prova-se que $D_v = \delta(s, v)$ para todo $v \in V$. Se o vértice v pode ser atingido

por s, então o Lema 5.4.9 prova essa afirmação. Se v não pode ser atingido por s, a prova decorre da propriedade da inexistência de caminho (Corolário 5.4.4). Portanto, a afirmação está provada. A propriedade de subgrafo dos predecessores (Lema 5.4.8) juntamente com essa última afirmação implica que G_π é uma árvore de caminhos mínimos. Agora, usa-se a afirmação para mostrar que Bellman-Ford retorna **true**. No término, têm-se para todas as arestas/arcos $(u,v) in E$,

$$\begin{aligned} D_v &= \delta(s,v) \\ &\leq \delta(s,u) + w\big((u,v)\big) \text{ (pela desigualdade triangular – Lema 5.4.2)} \\ &= D_u + w\big((u,v)\big), \end{aligned} \tag{5.20}$$

e, assim, nenhum dos testes na linha 10 serão verdadeiros e Bellamn-Ford retorna **false**. Então, ele retorna **true**.

Agora, suponha que o grafo G contenha o ciclo de peso negativo que possa ser atingido por s. Seja esse ciclo $c = \langle v_1, v_2, \ldots, v_k \rangle$, onde $v_1 = v_k$. Então,

$$\sum_{i=2}^{k} w\big((v_{i-1}, v_i)\big) < 0 \tag{5.21}$$

Considere, por contradição, que o algoritmo de Bellman-Ford retorna **true**. Assim, $D_{v_i} \leq D_{v_{i-1}} + w\big((v_{i-1}, v_i)\big)$ para $i = 2,3,\ldots,k$. Somando as desigualdades em torno do ciclo c têm-se

$$\sum_{i=2}^{k} D_{v_i} \leq \sum_{i=2}^{k} D_{v_{i-1}} + w\big((v_{i-1}, v_i)\big) = \sum_{i=2}^{k} D_{v_{i-1}} + \sum_{i=2}^{k} w\big((v_{i-1}, v_i)\big). \tag{5.22}$$

Como $v_0 = v_k$, cada vértice em c aparece exatamente apenas uma vez em cada um dos somatórios, portanto

$$\sum_{i=2}^{k} D_{v_i} = \sum_{i=2}^{k} D_{v_{i-1}} \tag{5.23}$$

Além disso, pelo Corolário 5.4.10, D_{v_i} é finito para $i = 2,3,\ldots,k$. Assim,

$$0 \leq \sum_{i=2}^{k} w\big((v_{i-1}, v_i)\big), \tag{5.24}$$

o que contradiz a desigualdade da Equação (5.21). Conclui-se que o algoritmo de Bellman-Ford retorna **true** se o grafo G não contém nenhum ciclo negativo que possa ser alcançado a partir da fonte e **false** caso contrário.

■

5.4.3 Floyd-Warshall

O algoritmo de Floyd-Warshall (Algoritmo 29) encontra o caminho mínimo para um grafo $G(V,E,w)$ ponderado dirigido ou não para todos os pares de vértices. O algoritmo suporta arestas/arcos de pesos negativos, mas não opera em grafos com ciclos de peso negativo.

A função W (Equation (5.25)) define uma matriz de adjacências[1] para o algoritmo de Floyd-Warshall.

$$W\Big(G(V,E,w)\Big)_{uv} = \begin{cases} 0, & \text{se } u = v, \\ w\big((u,v)\big), & \text{se } u \neq v \wedge (u,v) \in E, \\ \infty, & \text{se } u \neq v \wedge (u,v) \notin E. \end{cases} \tag{5.25}$$

O algoritmo define um número de matrizes igual a $|V|$. Inicialmente, a matriz $D^{(0)}$ é definida como a matriz de adjacência de G (linha 1). Depois repete-se o procedimento de criar uma nova matriz e atualizar as distâncias de cada celula da nova matriz por $|V|$

[1] Para grafos não-dirigidos, deve-se preencher a matriz resultante de $W(G)$ nas coordenadas (u,v) e (v,u)

vezes (linhas 2 a 6).

Algoritmo 29: Algoritmo de Floyd-Warshall.

Input : um grafo $G = (V, E, w)$

1 $D^{(0)} \leftarrow W\Big(G\Big)$

2 **foreach** $k \in V$ **do**

3 seja $D^{(k)} = \left(d_{uv}^{(k)}\right)$ uma nova matriz $|V| \times |V|$

4 **foreach** $u \in V$ **do**

5 **foreach** $v \in V$ **do**

6 $d_{uv}^{(k)} \leftarrow \textbf{min}\left\{d_{uv}^{(k-1)}, d_{uk}^{(k-1)} + d_{kv}^{(k-1)}\right\}$

7 **return** $D^{(|V|)}$

5.4.3.1 Complexidade de Floyd-Warshall

O algoritmo Floyd-Warshall (Algoritmo 29) demanda $|V|^2$ operações para executar a linha 1. Como há o aninhamento de três laços limitados a $|V|$ iterações na sequência, a operação na linha 6 será executada $|V|^3$ vezes. Logo, a complexidade de tempo computacional de Floyd-Warshall é de $\Theta(|V|^3)$.

Para este algoritmo, é interessante notar que foram utilizadas $|V|$ matrizes de $|V|$ linhas e $|V|$ colunas. Logo a complexidade de espaço para uma implementação que utilize o pseudo-código do Algoritmo 29 utilizaria espaço computacional de $\Theta(|V|^3)$. No entanto, é possível reduzir essa complexidade de espaço computacional em $\Theta(|V|^2)$.

Desafio

Crie um pseudo-código para o Algoritmo 29 que demande complexidade de espaço computacional $\Theta(|V|^2)$.

5.4.3.2 Corretude de Floyd-Warshall

Desafio

Prove que quando o Floyd-Warshall pára sobre uma entrada $G = (V, E, w)$ ele retornará as distâncias de caminhos mínimos para todo o par de vértice em V.

5.4.3.3 Construção de Caminhos Mínimos para Floyd-Warshall

O Algoritmo 30 além do peso dos caminhos mínimos em D, retorna a matriz dos predecessores Π, ou seja, uma matriz que indica o vértice anterior no caminho de cada coordenada u, v.

Algoritmo 30: Algoritmo de Floyd-Warshall com Matriz dos Predecessores.

Input : um grafo $G = (V, E, w)$

$D^{(0)} \leftarrow W\Big(G\Big)$

`// Matriz dos predecessores`

$\Pi^{(0)}_{uv} \leftarrow \left(\pi^{(0)}_{uv}\right)$ uma nova matriz $|V| \times |V|$

foreach $u \in V$ **do**

  foreach $v \in V$ **do**

    if $(u, v) \in E$ **then**

      $\pi^{(0)}_{uv} \leftarrow u$

    else

      $\pi^{(0)}_{uv} \leftarrow$ **null**

foreach $k \in V$ **do**

  seja $D^{(k)} = \left(d^{(k)}_{uv}\right)$ uma nova matriz $|V| \times |V|$

  seja $\Pi^{(k)} = \left(\pi^{(k)}_{uv}\right)$ uma nova matriz $|V| \times |V|$

  foreach $u \in V$ **do**

    foreach $v \in V$ **do**

      `// atualizando matriz dos predecessores`

      if $d^{(k-1)}_{uv} > d^{(k-1)}_{uk} + d^{(k-1)}_{kv}$ **then**

        $\pi^{(k)}_{uv} \leftarrow \pi^{(k-1)}_{kv}$

      $d^{(k)}_{uv} \leftarrow \mathbf{min}\left\{d^{(k-1)}_{uv}, d^{(k-1)}_{uk} + d^{(k-1)}_{kv}\right\}$

return $(D^{(|V|)}, \Pi^{(|V|)})$

A impressão de cada caminho pode ser realizada através do Algoritmo 31, que recebe como entrada a matriz de predecessores gerada pelo Algoritmo 30, um vértice de origem u e um de destino v. Ao terminar, o algoritmo terá impresso na tela o caminho

mínimo de u a v.

Algoritmo 31: Print-Shortest-Path.

Input : a matriz dos predecessores Π, um vértice de origem u, um vértice de destino v

```
1 if u = v then
2     print(u)
3 else
4     if π_uv = null then
5         print("Caminho inexistente de u para v")
6     else
7         Print-Shortest-Path(Π, u, π_uv)
8         print(v)
```

CAPÍTULO 6

Algoritmos Gulosos

Um algoritmo é dito guloso ou cobiçoso se constrói uma solução em pequenos passos, tomando uma decisão "míope" a cada passo para otimizar algum critério relacionado ao problema (geralmente não óbvio). Quando um algoritmo guloso resolve algum problema não trivial, geralmente implica na descoberta de algum padrão útil na estrutura do problema (KLEINBERG; TARDOS, 2005).

6.1 Exemplo de Algoritmos Gulosos

6.1.1 Agendamento de Intervalos

Considere um conjunto de intervalos $R = \{r_1, r_2, \ldots, r_n\}$ no qual cada intervalo r_i inicia em $s(r_i)$ e termina em $f(r_i)$. Diz-se que dois intervalos são compatíveis se eles não possuem sobreposição de tempo. Deseja-se encontrar um subconjunto de R com o maior número de intervalos compatíveis entre si. Pense sobre quais regras naturais poderiam ser empregadas nessa busca (como critério simples) e imagine como elas trabalham. Alguns exemplos simples (KLEINBERG; TARDOS, 2005):

- Inserir pela ordem de $s(r_i)$;

- Inserir pela ordem de menor intervalo, pois quer se selecionar o maior número possível deles;

- Contagem de recursos não compatíveis e utilizar aquele com o menor número de incompatibilidades.

Nenhum desses critérios levariam a solução ótima para todas as instâncias do problema. A Figura 11 ajuda a enteder o porquê. Utilizando a ordem $s(r_i)$, o exemplo presente na Figura 11(a) a solução seria apenas um intervalo. Quanto a ordem pelo menor intervalo, a Figura 11(b) demonstra que apenas um intervalo seria marcado para uma instância cuja solução ótima conta com dois intervalos. Ao utilizar o critério a quantidade mínima de intervalos não compatíveis, a Figura 11(c) demonstra uma instância cuja solução ótima seria selecionar os 4 intervalos da primeira linha. No entanto, com esse último critério, iniciaria-se selecionando o segundo intervalo da segunda linha e depois qualquer outro intevalo não conflitante na esquerda e na direita, ou seja, no máximo três intervalos.

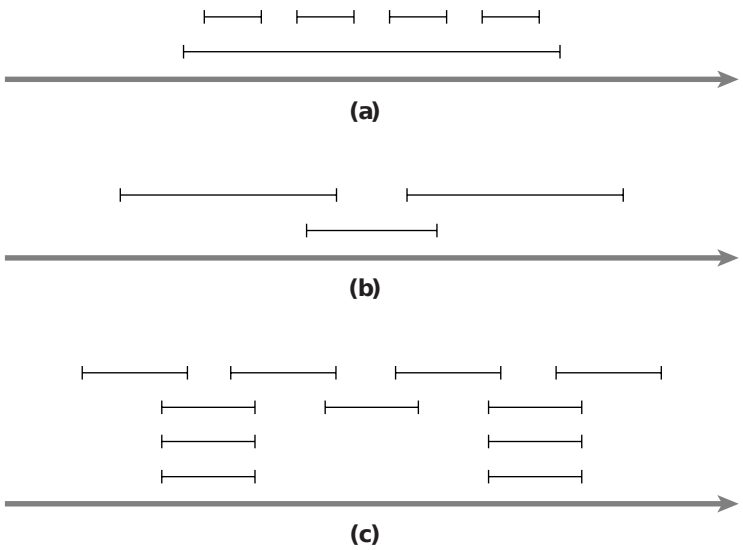


Figura 11 – Exemplos de Kleinberg e Tardos (2005) para refutar as três regras naturais levantadas anteriormente para o problema de agendamento do maior número de intervalos.

Para resolver o problema com uma estratégia gulosa, Kleinberg e Tardos (2005)

utilizam o menor valor de $f(i)$. Então, inicialmente se escolhe o intervalo de menor tempo que termina mais cedo na instância. Depois, seleciona-se o próximo intervalo compatível de menor tempo de fim, até que não haja mais intervalos compatíveis.

O Algoritmo 32 exibe o algoritmo guloso sugerido por (KLEINBERG; TARDOS, 2005).

Algoritmo 32: Maior número de Intervalos.

Input :O conjunto de intervalos R, os valores $s(r_i)$ e $f(r_i)$ para cada intervalo $r_i \in R$.

```
// Criar lista L com os intervalos na ordem crescente de f(·)
```

1 $L \leftarrow sort(R, f(\cdot))$

```
// Agora, considere que L = (l_1, l_2, ..., l_n)
```

2 $A \leftarrow \{\}$

3 $i \leftarrow 1$

4 **while** $i \leq n$ **do**

5 $\quad A \leftarrow A \cup \{l_i\}$

6 $\quad j \leftarrow i + 1$

$\quad$ `// Busca próximo intervalo compatível`

7 $\quad$ **while** $j \leq n \wedge s(l_j) \leq f(l_i)$ **do**

8 $\quad\quad j \leftarrow j + 1$

9 $\quad i \leftarrow j$

10 **return** A

6.1.1.1 Complexidade do Problema de Agendamento de Intervalos

Analisando a complexidade de tempo computacional para o Algoritmo 32, destacam-se duas partes mais significativas do pseudo-código. A primeira é relativa ao algoritmo de ordenação utilizado na linha 1. Sabe-se que o algoritmo mais eficiente para ordenação demanda $\Theta(n \log_2 n)$. Além dessa parte do algoritmo, há o laço de repetição nas linhas 4 à 9. Pode-se observar que as instruções internas ao laço de repetição da linha 4 serão executadas um número de vezes igual a $|R|$. Então, o algoritmo demanda tempo computacional de $\Theta(|R| \log_2 |R|)$.

6.1.1.2 Corretude do Problema de Agendamento de Intervalos

Lema 6.1.1. *Considerando que a solução ótima para o problema de agendamento de intervalos é* $O = \{j_1, j_2, \ldots, j_m\}$ *e a solução obtida pelo Algoritmo 32 é* $A = \{i_1, i_2, \ldots, i_k\}$, *têm-se* $f(i_r) \leq f(j_r)$ *para todo* $r \leq k$.

Prova: Essa prova será realizada por indução. Para $r = 1$, a afirmação é claramente verdadeira, pois o algoritmo inicia selecionando.

Agora se analisa os casos nos quais $r > 1$. Assume-se a hipótese de que a afirmação é verdadeira para $r-1$, e tenta-se provar isso para r. Desse modo, $f(i_{r-1}) \leq f(j_{r-1})$. Sabe-se que $f(j_{r-1}) \leq s(j_r)$, combinando isso com a hipótese da indução, têm-se $f(i_{r-1}) \leq s(j_r)$. Então, o intervalo j_r está no conjunto de intervalos disponíveis quando o algoritmo seleciona i_r. Como o algoritmo sempre seleciona o intervalo de menor tempo de término, têm-se $f(i_r) \leq f(j_r)$. Isso completa o passo da indução. ■

Teorema 6.1.2. *Algoritmo 32 retorna o conjunto ótimo A.*

Prova: Prova-se por contradição. Se A não é ótimo, então um conjunto O precisa ter mais intervalos, ou seja, $m > k$. Aplicando o Lema 6.1.1 com $r = k$, têm-se $f(i_k) \leq f(j_k)$. Como $m > k$, então há um intervalo em j_{k+1} em O. Esse intervalo inicia depois que j_k termina, e depois que i_k termina. Então, depois de passar todos os intervalos incompatíveis (linhas 7 e 8) com os intervalos $i_1, i_2, \ldots, i_k$, haveria ainda um intervalo j_{k+1}. Mas o algoritmo guloso parou com o intervalo i_k, e ele só pára quando não houver mais intervalos em L a analisar, ou seja, uma contradição. ■

6.1.2 Dijkstra

O algoritmo de Dijkstra resolve o problema de encontrar um caminho mínimos de fonte única em um grafo $G = (V, E, w)$ ponderados dirigidos ou não. Para esse algoritmo, as arestas/arcos não devem ter pesos negativos. Então, a função de pesos é redefinida como $w : E \to \mathbb{R}^+_*$. A vantagem está em o algoritmo de Dijkstra ser mais eficiente que o

do Bellman-Ford caso uma estrutura de dados adequada seja utilizada (CORMEN et al., 2012).

O algoritmo repetidamente seleciona o vértice de menor custo estimado até então. Quando esse vértice é selecionado, ele não é mais atualizado e sua distância é propagada para suas adjacências. A estrutura de dados C é utilizada no pseudo-código abaixo para definir se um vértice foi visitado (contém **true**) ou não (contém **false**).

Algoritmo 33: Algoritmo de Dijkstra.

Input : um grafo $G = (V, E, w : E \to \mathbb{R}_*^+)$, um vértice de origem $s \in V$

1 $D_v \leftarrow \infty \; \forall v \in V$
2 $A_v \leftarrow$ **null** $\forall v \in V$
3 $C_v \leftarrow$ **false** $\forall v \in V$
4 $D_s \leftarrow 0$
5 **while** $\exists v \in V (C_v =$ ***false***$)$ **do**
6 $\quad u \leftarrow \textbf{arg min}_{v \in V}\{D_v | C_v =$**false**$\}$
7 $\quad C_u \leftarrow$ **true**
8 $\quad$ **foreach** $v \in N(u) : C_v =$***false*** **do**
9 $\quad\quad$ **if** $D_v > D_u + w\big((u,v)\big)$ **then**
10 $\quad\quad\quad D_v \leftarrow D_u + w\big((u,v)\big)$
11 $\quad\quad\quad A_v \leftarrow u$
12 **return** (D, A)

6.1.2.1 Complexidade de Dijkstra

Se o algoritmo de Dijkstra manter uma fila de prioridades mínimas para mapear a distância estimada no lugar de D, o algoritmo torna-se mais eficiente que o Bellman-Ford. Seria utilizada uma operação do tipo "EXTRACT-MIN" no lugar da que está na linha 6, para encontrar o vértice com a menor distância. Ao extraí-lo da estrutura de prioridade, não mais seria necessário. Poderia-se gravar sua distância mínima em uma estrutura auxiliar e não mais utilizar a estrutura de visitas C. Ao atualizar as distâncias, poderia-se utilizar a operação de "DECREASE-KEY" da fila de prioridades no lugar da

operação da linha 10.

Para essa fila de prioridades, poderia se utilizar um Heap, como o Heap Binário, no qual a implementação das operações supracitadas tem complexidade de tempo computacional $O(\log_2 n)$ para o "DECREASE-KEY" e $O(\log_2 n)$ para o "EXTRACT-MIN". Para essa aplicação, $n = |V|$. Utilizando essa estrutura de dados, sabe-se que no máximo executa-se $|E|$ operações de "DECREASE-KEY" e $|V|$ operações de "EXTRACT-MIN". Então a complexidade computacional seria $O\big((|V|+|E|)\log_2|V|\big)$. Com Heap Fibonacci, é possível obter resultados ainda melhores em tempo computacional.

6.1.2.2 Corretude do Algoritmo de Dijkstra

Teorema 6.1.3. *Dado um grafo $G = (V, E, w : E \to \mathbb{R}^+_*)$ e um vértice de origem $s \in V$, o algoritmo de Dijkstra termina com $D_v = \delta(s, v)$ para todo $v \in V$.*

Prova: Usa-se a seguinte invariante de laço: no início de cada iteração do laço das linhas 5 – 11, $D_v = \delta(s, v)$ para todo v o qual $C_v =$**true**. Para demonstrar isso, diz-se que $D_v = \delta(s, v)$ no momento em que v é marcado como visitado, ou seja, na linha 7. Uma vez demonstrado isso, recorre-se à propriedade do limite superior (Lema 5.4.3) para demonstrar que a igualdade é válida em todos os momentos a partir desse.

Inicialização: Inicialmente, $C_u =$**false** para todos $u \in V$. Então, a invariante é trivialmente verdadeiro.

Manutenção:

Por contradição, seja u o primeiro vértice para o qual $D_u \neq \delta(s, u)$ quando C_u torna-se **true**. O vértice u não pode ser s, pois $D_s = \delta(s, v) = 0$ nesse momento. Como $u \neq s$, deve existir algum caminho de s a u, senão $D_u = \delta(s, u) = \infty$ pela propriedade de inexistência de caminho (Lema 5.4.4), o que contradiz $D_u \neq \delta(s, u)$.

Assume-se então que haja um mínimo caminho p de s a u. Antes de C_u se tornar **true**, o caminho p conecta um vértice v o qual $C_v =$**true** a u. Decompõe-se o caminho p em $s \overset{p_1}{\rightsquigarrow} x \to y \overset{p_2}{\rightsquigarrow} u$, no qual $C_x =$**true** e $C_y =$**false**. Afirma-se que $D_y = \delta(s, y)$ no momento que C_u se torna **true**. Para provar essa afirmação, observa-se que $C_x =$**true**.

Então, u foi escolhido como primeiro vértice para o qual $D_u \neq \delta(s,u)$ quando C_u se torna **true**, tínha-se $D_x = \delta(s,x)$ quando C_x se tornou **true**. A aresta/arco (x,y) foi relaxada naquele momento, e a afirmação decorre da propriedade de convergência (Lema 5.4.5). Agora, pode-se obter uma contradição para provar que $D_u = \delta(s,u)$. Como y aparece antes de u em um caminho mínimo de s a u e todos os pesos das arestas/arcos são não-negativos, temos $\delta(s,y) \leq \delta(s,u)$ e assim,

$$\begin{aligned} D_y &= \delta(s,y) \\ &\leq \delta(s,u) \\ &\leq D_u (\text{ pela propriedade do limite superior – Lema 5.4.3}). \end{aligned} \tag{6.1}$$

Porém, como C_u =**false** e C_y =**false** quando u foi escolhido na linha 6, tem-se $D_u \leq D_y$. Assim, as duas desigualdades da Equação (6.1) são de fato igualdades, o que dá

$$D_y = \delta(s,y) = \delta(s,u) = D_u. \tag{6.2}$$

Consequentemente, $D_u = \delta(s,u)$, o que contradiz a escolha de u. Conclui-se que $D_u = \delta(s,u)$ quando C_u se torna **true** e que essa igualdade é mantida até o término do algoritmo.

Término:

No término, quanto para C_v =**true** para todo $v \in V$, $D_v = \delta(s,v)$ para todo $v \in V$. ■

Corolário 6.1.4. *Ao executar algoritmo de Dijkstra (Algoritmo 33) sobre o grafo $G = (V,E,w)$ ponderado orientado ou não, sem ciclos de peso negativo, para o vértice de origem $s \in V$, o subgrafo dos predecessores G_π será uma árvore de caminhos mínimos em s.*

Prova: Imediata pelo Teorema 6.1.3 e a propriedade do subgrafo dos predecessores (Lema 5.4.8). ■

6.1.3 Árvores Geradoras Mínimas

Uma árvore geradora é um subconjunto acíclico (uma árvore) de todos os vértices de um grafo. Dado um grafo ponderado $G = (V, E, w)$, o problema de encontrar uma arvore geradora mínima é aquele no qual busca-se encontrar uma árvore que conecte todos os vértices do grafo G, tal que a soma dos pesos das arestas seja o menor possível. Seja T uma árvore geradora, diz-se que o custo da árvore geradora de T é dado por $w(T) = \sum_{\{u,v\} \in T} w\big(\{u, v\}\big)$.

Este capítulo apresenta dois métodos para encontrar árvores geradoras mínimas: Kruskal e Prim. Esses dois algoritmos gulosos se baseiam em um método genérico apresentado por Cormen et al. (2012). Dado um grafo não-dirigido e ponderado $G = (V, E, w)$, esse método genérico encontra uma árvore geradora mínima.

O Algoritmo 34 apresenta o método genérico para uma árvore geradora mínima para G. O método se baseia na seguinte invariante de laço: "Antes de cada iteração, A é um subconjunto de alguma árvore geradora mínima.". A cada iteração, determina-se uma aresta que pode ser adicionada a A sem violar a invariante de laço. A aresta segura é uma aresta que se adicionada a A mantém a invariante.

Algoritmo 34: Método Genérico de Cormen et al. (2012).

Input : um grafo $G = (V, E, w)$

```
1 A ← {}
2 while A não formar uma árvore geradora do
3     encontrar uma aresta {u, v} que seja segura para A
4     A ← A ∪ {{u, v}}
5 return A
```

6.1.3.1 Propriedades do Método Genérico

Teorema 6.1.5. *Considere um grafo conexo não-dirigido ponderado $G = (V, E, w)$. Seja $A \subseteq E$ uma árvore geradora mínima de G. Seja $(S, V \setminus S)$ qualquer conjunto de corte de G que respeita A e seja $\{u, v\}$ uma aresta leve[1] que cruza $(S, V \setminus S)$, então $\{u, v\}$ é uma aresta*

[1] de baixo custo

segura para A.

Prova:

Seja T uma árvore geradora mínima que inclui A e suponha que T não contenha a aresta leve $\{u, v\}$, pois se tiver, o processo termina. Constrói-se então outra árvore geradora mínima T' que inclui $A \cup \big\{\{u, v\}\big\}$ usando uma técnica de recortar e colar, mostrando assim que $\{u, v\}$ é uma aresta segura para A.

Desde que u esteja em S e v em $V \backslash S$, a aresta $\{u, v\}$ forma um ciclo com as arestas de caminho simples p de u a v em T. No mínimo uma aresta em T está no caminho simples p e cruza o corte. Considere que $\{x, y\}$ seja essa aresta de corte. A aresta $\{x, y\}$ não está em A, pois o corte respeita A. Desde que $\{x, y\}$ está no único caminho de u a v em T, removê-lo divide T em duas componentes. Adicionar $\{u, v\}$ o reconecta a forma de uma nova árvore geradora $T' = T \cup \big\{\{u, v\}\big\} \backslash \big\{\{x, y\}\big\}$.

Depois, mostra-se que T' é uma árvore geradora mínima. Desde que $\{u, v\}$ é uma aresta leve atravessando $(S, S \backslash V)$ e $\{x, y\}$ também atravessa esse corte, $w\big(\{u, v\}\big) \leq w\big(\{x, y\}\big)$. Por essa razão:

$$\begin{aligned} w(T') = w(T) + w\big(\{u, v\}\big) - w\big(\{x, y\}\big) \\ \leq w(T'). \end{aligned} \tag{6.3}$$

Mas, T é uma árvore geradora mínima, então $w(T) \leq w(T')$. Desse modo, T' precisa ser uma árvore geradora mínima também.

Falta mostrar que $\{u, v\}$ é uma aresta segura para A. Têm-se $A \subseteq T'$, desde que $A \subseteq T$ e $\{x, y\} \notin A$; então $A \cup \big\{\{u, v\}\big\} \subseteq T'$. Consequentemente, desde que T' é uma árvore geradora mínima, $\{u, v\}$ é uma aresta segura para A. ■

Corolário 6.1.6. *Considere um grafo conectado não-dirigido e ponderado $G = (V, E, w)$. Seja $A \subseteq E$ incluído em alguma árvore geradora mínima de G e seja $C = (V_C, E_C)$ um componente conectado (uma árvore) na floresta $G_A = (V, A)$. Se $\{u, v\}$ é uma aresta leve conectando C a algum outro componente em G_A, então $\{u, v\}$ é uma aresta segura.*

Prova: O corte $(V_C, V \setminus V_C)$ respeita A e $\{u, v\}$ é uma aresta leve para esse corte. Por essa razão, $\{u, v\}$ é uma aresta segura para A (com base no Teorema 6.1.5). ■

6.1.3.2 Algoritmo de Kruskal

O algoritmo de Kruskal inicia com $|V|$ árvores (conjuntos de um vértice cada). Ele encontra uma aresta segura e a adiciona a uma floresta, que está sendo montada, conectando duas árvores na floresta. Essa aresta $\{u, v\}$ é de peso mínimo. Suponha que C_1 e C_2 sejam duas árvores conectadas por uma aresta $\{u, v\}$. Visto que essa deve ser uma aresta leve, o Corolário 6.1.6 implica que $\{u, v\}$ é uma aresta segura para C_1. Kruskal é um algoritmo guloso, pois ele adiciona à floresta uma aresta de menor peso possível a cada iteração.

Um pseudo-código de Kruskal pode ser visualizado no Algoritmo 35. O algoritmo se inicia definindo as $|V|$ árvores desconectadas; cada uma contendo um vértice (linhas 2 a 4). Então, cria-se uma lista das arestas E' ordenadas por peso (linha 5). Depois, iterativamente, se tenta inserir uma aresta leve em duas árvores que contém nodos não foram conectadas ainda (linhas 7 a 9). Quando a inserção ocorre, a estrutura de dados S_v que mapeia a árvore de cada vértice v é é atualizada. O procedimento se repete até

que todas as arestas tenham sido avaliadas no laço.

Algoritmo 35: Algoritmo de Kruskal.

Input : um grafo $G = (V, E, w)$

```
1  A ← {}
2  S ← {{}}^|V|
3  foreach v ∈ V do
4      S_v ← {v}
5  E' ← lista de arestas ordenadas por ordem crescente de peso
6  foreach {u, v} ∈ E' do
7      if S_u ≠ S_v then
8          A ← A ∪ {{u, v}}
9          x ← S_u ∪ S_v
10         foreach w ∈ x do
11             S_w ← x
12 return A
```

Complexidade do Algoritmo de Kruskal

A complexidade de tempo do algoritmo de Kruskal depende da estrutura de dados utilizada para implementar o mapeamento das árvores de cada vértice, ou seja, da estrutura S do Algoritmo 35. Para a implementação mais eficiente, verifique as operações de conjuntos disjuntos no Capítulo 21 de Cormen et al. (2012). Sabe-se que para ordenar o conjunto de arestas na linha 5, deve-se executar um algoritmo de ordenação. O mais eficiente conhecido demanda tempo $\Theta(|E| \log_2 |E|)$.

Desafio

Qual estrutura de dado implementa a versão mais eficiente do Algoritmo de Kruskal?

6.1.3.3 Algoritmo de Prim

O algoritmo de Prim é (também) um caso especial do método genérico apresentado no Algoritmo 34. O algoritmo de Prim é semelhante ao algoritmo de Dijkstra, como pode ser observado no pseudo-código do Algoritmo 36. As arestas no vetor A formam uma árvore única. Essa árvore tem raízes em um vértice arbitrário r. Essa arbitrariedade não gera problemas de corretudo no algoritmo, pois uma árvore geradora mínima deve conter todos os vértices do grafo. A cada iteração, seleciona-se o vértice que é atingido por uma aresta de custo mínimo (linha 7), sendo que o vértice selecionado adiciona suas adjacências e pesos na estrutura de prioridade Q, que, futuramente, selecionará a chave de menor custo, ou seja, o vértice conectado a árvore que possui o menor custo. Pelo Corolário 6.1.6, esse comportamento adiciona-se apenas arestas seguras em A.

Antes de cada iteração do laço da linha 6, a árvore obtida é dada por $\{\{v, A_v\} : v \in V \backslash \{r\} \backslash Q\}$. Os vértices que já participam da solução final são $V \backslash Q$. Além disso, para todo $v \in Q$, se $A_v \neq$**null**, então $K_v < \infty$ e K_v é o peso de uma resta leve $\{v, A_v\}$ que conecta o vértice v a algum vértice que já está na árvore que está sendo construída. As linhas 7 e 8 indentificam um vértice $u \in Q$ incidente em alguma aresta leve que cruza o corte $(V \backslash Q, Q)$, exceto na primeira iteração. Ao remover u de Q, o mesmo é acrescido ao conjunto $V \backslash Q$ na árvore, adicionando a aresta (u, A_u) na solução.

Se G é conexo, para montar a árvore geradora mínima a partir do vetor resultante A,

deve-se criar o conjunto $\big\{\{v, A_v\} : v \in V \backslash \{r\}\big\}$.

Algoritmo 36: Algoritmo de Prim.

Input : um grafo $G = (V, E, w)$

$r \leftarrow$ selecionar um vértice arbitrário em V

`// Definindo o vetor dos antecessores` A `e uma chave para cada vértice` K

$A_v \leftarrow$ **null** $\forall v \in V$

$K_v \leftarrow \infty \; \forall v \in V$

$K_r \leftarrow 0$

`// Definindo a estrutura de prioridade de chave mínima` Q

$Q \leftarrow (V, K)$

while $Q \neq \{\}$ **do**

  $u \leftarrow$ **arg min**$_{v \in Q}\{K_v\}$

  $Q \leftarrow Q \backslash \{u\}$

  foreach $v \in N(u)$ **do**

    if $v \in Q \wedge w\big(\{u, v\}\big) < K_v$ **then**

      $A_v \leftarrow u$

      $K_v \leftarrow w\big(\{u, v\}\big)$

return A

Complexidade do Algoritmo de Prim

Para implementar o algoritmo de Prim de maneira mais eficiente possível, deve-se encontrar uma estrutura de prioridade que realize as operações de extração do valor mínimo e da alteração das chave de maneira rápida.

Desafio

Qual a complexidade em tempo computacional da versão mais eficiente do Algoritmo de Prim?

6.1.4 Códigos de Huffman

Códigos de Huffman comprimem dados efetivamente de 20% a 90%. A estratégia utilizada é empregar a frequência em que os caracteres aparecem para reduzir a representação dos mesmos. Ao usar-se uma tabela de tamanho fixo, caracteres frequentes e os demais utilizam a mesma quantidade de bits. A ideia de códigos de Huffman é aplicar a frequência na redução de codificação de caracteres mais frequentemente utilizados (CORMEN et al., 2012).

Para o Algoritmo 37, considere que C é um conjunto de nodos de uma árvore binária (ou árvore de Huffman), no qual c.frequencia$\in \mathbb{Z}_*^+$ é a frequência dos caracteres representados em cada $c \in C$ (presentes no nodo e seus descendentes). $c.esquerda$ =**null** e $c.direita$ =**null** são os ponteiros para os descendentes na árvore de Huffman.

Algoritmo 37: Huffman.

Input : um conjunto de caracteres C

```
// Criando fila de prioridade pela frequência mapeada por f()
```

1 $Q \leftarrow (C, f)$
2 **for** $i \leftarrow 1$ ***to*** $|C| - 1$ **do**
3 alocar um novo nodo de uma árvore binária r
4 $e \leftarrow EXTRACT - MIN(Q)$
5 $d \leftarrow EXTRACT - MIN(Q)$
6 $r.esquerda \leftarrow e$
7 $r.direita \leftarrow d$
8 $r.frequencia \leftarrow e.frequencia + d.frequencia$
9 $INSERT(Q, r)$
10 **return** $EXTRACT - MIN(Q)$

O Algoritmo 37 retorna a Árvore de Huffman que é utilizada para estabelecer a nova codificação do conjunto de caracteres representados por C. Geralmente, considera-se que cada ramificação para esquerda produz um número 0 e cada para direita produz um número 1. Então, as arestas da raiz até um caractere (nodo folha) estabelece a nova codificação do mesmo.

6.1.4.1 Complexidade

Desafio

Qual a complexidade de tempo do Algoritmo 37?

CAPÍTULO 7

Programação Dinâmica

Em momentos anteriores nessas notas, problemas foram tratados por estratégias gulosas que operavam eficientemente. Infelizmente, para a maioria dos problemas, a dificuldade não está em determinar qual critério guloso utilizar, pois não há estratégia gulosa que conduza à solução ótima. Para problemas como esses, é importante usar outras estratégias. A Divisão e Conquista pode servir como uma alternativa, mas as versões conhecidas podem não ser suficientemente fortes para evitar uma busca de força-bruta (KLEINBERG; TARDOS, 2005).

Neste capítulo lida-se com a Programação Dinâmica. É fácil dizer o que é programação dinâmica depois de trabalhar com alguns exemplos. A ideia básica vem de uma intuição a partir da divisão e conquista e é essencialmente o oposto de uma estratégia gulosa. A Programação Dinâmica implica explorar o espaço de todas as soluções possíveis, por cuidadosamente, decompor o problema em subproblemas, e então construir soluções corretas para problemas maiores e maiores. Pode-se considerar que a Programação Dinâmica opera perigosamente próxima a algoritmos de força-bruta. Apesar disso, essa estratégia opera sobre um conjunto exponencialmente grande de subproblemas sem a necessidade de examinar todos eles explicitamente (KLEINBERG; TARDOS, 2005). Na estratégia de Programação Dinâmica, soluções ótimas de tamanho menor auxiliam na busca por uma solução ótima maior.

Problemas de Otimização

Neste capítulo, lida-se com problemas os quais buscam por uma solução chamada de "solução ótima". Esses problemas são chamados de problemas de otimização. Uma solução ótima é uma resposta para o problema que obtém a melhor (mínima ou máxima) avaliação possível dada por uma função. Essa função é denominada de função objetivo. Um algoritmo para um problema de otimização sempre encontra a solução ótima.

Soluções que não são ótimas, podem ou não satisfazer restrições relacionadas ao problema. As soluções que satisfazem as restrições do problema, mas não possuem a melhor avaliação possível pela função objetivo, são chamadas de solução admissíveis não-ótimas. Toda solução ótima é uma solução admissível.

7.1 Organização de Intervalos com Pesos

Anteriormente, fora conhecido um problema de identificar o conjunto maximal de intervalos quando visitou-se o conceito de algoritmos gulosos, agora trabalha-se com uma variação desse problema, chamado de problema de Intervalos com Pesos. Considere um conjunto de intervalos $I = \{I_1, I_2, \ldots, I_n\}$ onde cada intervalo I_i inicia no tempo $s(i) \in \mathbb{Z}^+$, termina no tempo $f(i) \in \mathbb{Z}^+$, e possui o valor $v(i) \in \mathbb{Z}^+$, precisa-se encontrar o conjunto de intervalos que maximiza o valor de $\sum_{i \in S} v(i)$, onde S é um subconjunto de intervalos sem sobreposição de tempo (KLEINBERG; TARDOS, 2005).

Supõe-se que o os intervalos estejam organizados em uma ordem crescente de tempos de fim: $F = (f(I_1) \leq f(I_2) \leq \ldots \leq f(I_n))$. Agora supõe-se a existência de uma função $p : I \rightarrow I$. $p(I_i)$ determina o intervalo I_j que é o primeiro intervalo compatível antes de I_i na lista F. Considere a solução ótima O. Se $I_n \in O$, não há intervalos compatíveis entre $p(I_n)$ e I_n. Ainda considerando que I_n pertence a solução ótima, considera-se como potenciais candidatos a solução ótima $\{I_1, I_2, \ldots, p(I_n)\}$. Caso $I_n \notin O$, deve-se considerar como potenciais candidatos a solução ótima $\{I_1, I_2, \ldots, I_{n-1}\}$.

Pensando na recorrência para a programação dinâmica, considera-se $OPT(I_i)$ como

o valor da solução ótima para os intervalos $\{I_1, I_2, \ldots, I_i\}$. Por questões de convenção, assume-se que $OPT(I_0) = 0$. A recorrência representando a solução ótima considerando o problema é dado por

$$OPT(I_i) = \begin{cases} 0 & \text{para } i = 0, \\ \textbf{max}\Big\{v(I_i) + OPT\big(p(I_i)\big), OPT(I_{i-1})\Big\} & \text{para } i > 0. \end{cases} \tag{7.1}$$

Então, se está assumindo que I_i pertence a solução ótima sse $v(I_i) + OPT\big(p(I_i)\big) > OPT(I_{i-1})$.

Um algoritmo recursivo para resolver a recorrência pode ser visualizado abaixo.

Algoritmo 38: ARPIP: Algoritmo Recursivo para o Problema de Intervalos com Peso.

Input : um conjunto de itens $I = \{1, 2, \ldots, n\}$, $s(i) \in \mathbb{Z}^+$, $f(i) \in \mathbb{Z}^+$, $v(i) \in \mathbb{Z}^+$, o intervalo I_j

```
1 if j = 0 then
2     return 0
3 else
4     return max{ v(I_j) + ARPIP(I, s, f, v, p(I_j)), ARPIP(I, s, f, v, I_{j-1}) }
```

Teorema 7.1.1. *O Algoritmo 38 calcula o valor e $OPT(I_j)$.*

Prova: Por definição $OPT(I_0) = 0$ (linha 1 e 2). Para algum $j > 0$, supõe-se, por indução, que o algoritmo calcula corretamente o valor $OPT(I_i)$ para um $i < j$. Pela hipótese da indução, sabe-se que o algoritmo encontra a solução para $OPT(I_{p(I_j)})$ e para $OPT(I_{j-1})$; e por essa razão

$$OPT(I_j) = \textbf{max}\Big\{v(I_j) + ARPIP\big(I, s, f, v, p(I_j)\big), ARPIP(I, s, f, v, I_{j-1})\Big\} = ARPIP(I, s, f, v, I_j). \tag{7.2}$$

■

Infelizmente, ao executar o Algoritmo 38, percebe-se que a árvore de subproblemas cresce muito rapidamente, o que impacta no tempo de execução (não-polinomial). A árvore cresce rapidamente porque há recálculo de subproblemas. A próxima tentativa

irá utilizar uma memória para evitar tais recálculos (KLEINBERG; TARDOS, 2005). Tal método é conhecido como Memoização (CORMEN et al., 2012).

No novo algoritmo, a memoização é implementada através de um vetor M indexado de 0 a n. Assuma que $M_i = -1$ para todo $i \in \{1,2,\ldots,n\}$. O Algoritmo 39 exibe o pseudo-código.

Algoritmo 39: AMIP: Algoritmo "Memoizado" para Intervalos com Peso.

Input : um conjunto de itens $I = \{1,2,\ldots,n\}$, $s(i) \in \mathbb{Z}^+$, $f(i) \in \mathbb{Z}^+$, $v(i) \in \mathbb{Z}^+$, o intervalo I_j

```
1 if j = 0 then
2     return 0
3 else
4     if M_j = −1 then
5         M_j ← max{ v(I_j) + AMIP(I, s, f, v, p(I_j)), AMIP(I, s, f, v, I_{j−1}) }
6     return M_j
```

Para encontrar a solução a partir do vetor M, pode-se utilizar o Algoritmo 40 .

Algoritmo 40: AESPA: Algoritmo para encontrar solução a partir de AMIP.

Input : um conjunto de itens $I = \{1,2,\ldots,n\}$, $s(i) \in \mathbb{Z}^+$, $f(i) \in \mathbb{Z}^+$, $v(i) \in \mathbb{Z}^+$, o intervalo I_j, vetor M

```
1 if j = 0 then
2     não gera saída
3 else
4     if v_j + M_{p(I_j) ≥ M_{I_{j−1}}} then
5         dar saída I_j concatenada com o resultado de AESPA(I, s, f, v, p(I_j), M)
6     else
7         dar saída o resultado de AESPA(I, s, f, v, I_{j−1}), M)
8     return M_j
```

7.1.1 Complexidade

A complexidade de tempo dos Algoritmo 39 e 40 é $O(n)$.

7.2 O Problema da Mochila

Antes de começar a analisar o problema da mochila, apresenta-se um problema mais simples, que chamaremos aqui de "Subconjunto de Somas".

O Subconjunto de Somas é um problema no qual, dado um conjunto dos items $I = \{1, 2, \ldots, n\}$, uma função de peso $w : I \to \mathbb{Z}^+$ e um limite de peso $W \in \mathbb{Z}^+$, deseja-se encontrar um subconjunto de $S \subseteq I$ com a maior soma $\sum_{s \in S} w(s)$ tal que $\sum_{s \in S} w(s) \leq W$ (KLEINBERG; TARDOS, 2005).

Pode-se tentar resolver por programação dinâmica utilizando o $OPT(i)$ para denotar o valor da solção ótima para a subsolução $\{1, 2, \ldots, n\}$. Desse modo, considerando O a solução ótima, se $i \notin O$, então $OPT(i) = OPT(i-1)$, senão $OPT(i) = w(i) + OPT(i-1)$. O problema nessa tentativa está em que ao aceitar i não implica a rejeição de qualquer outro item. Isso indica a necessidade de trabalhar com mais subproblemas.

A recorrência que funciona para o problema de Subconjunto de Somas utiliza a ideia de considerar o limite de peso W, no qual, se um item é adicionado a subsolução, ele diminui seu peso ao valor de W. Logo, utiliza-se a seguinte recorrência

$$OPT(i,p) = \begin{cases} 0 & \text{para } i = 0 \text{ ou } p = 0, \\ OPT(i-1,p) & \text{para } p < w(i), \\ \textbf{max}\{OPT(i-1,p), w(i) + OPT(i-1,p-w(i))\} & \text{para } p \geq w(i). \end{cases} \tag{7.3}$$

, na qual w é o valor corrente de peso restante a ser considerado para os subproblemas.

O Problema da Mochila (ou *Knapsack Problem*) é muito semelhante ao de Subconjunto de Somas. Nele, além do conjunto de itens I, da função de peso w para cada item e do limite de peso W, considera-se que cada item tem um valor associado, dado pela função $v : I \to \mathbb{R}$. Deve-se buscar o subconjunto $S \subseteq I$ no qual $\sum_{s \in S} v(s)$ tal que $\sum_{s \in S} w(s) \leq W$. Podemos então utilizar uma função de recorrência muito semelhante à

proposta para o problema do Subconjunto de Somas. Segue a recorrência adaptada

$$OPT(i,p) = \begin{cases} 0 & \text{para } i = 0 \text{ ou } p = 0, \\ OPT(i-1,p), & \text{para } p < w(i) \\ \textbf{max}\{OPT(i-1,p), v(i) + OPT(i-1,p-w(i))\}, & \text{para } p \geq w(i). \end{cases} \tag{7.4}$$

O Algoritmo 41 proposto pode ser visualizado abaixo, já com a memorização.

Algoritmo 41: Algoritmo para o Problema da Mochila.

Input : um conjunto de itens $I = \{1,2,\ldots,n\}$, uma função de pesos $w : I \to \mathbb{Z}^+$, uma função de valor $v : I \to \mathbb{R}$ e um peso limite $W \in \mathbb{Z}^+$

```
// Matriz de memorização
```

1 Criar matriz $M \in \mathbb{R}^{(|I|+1)\times(W+1)}$

2 $M[0,w] = 0 \; \forall w \in \{0,1,2,\ldots,W\}$

3 **foreach** $i \in (1,2,\ldots,n)$ **do**

4 **for** $w \leftarrow 0$ ***to*** W **do**

5 **if** $w < w(i)$ **then**

6 $M[i,w] \leftarrow M[i-1,w]$

7 **else**

8 $M[i,w] \leftarrow \textbf{max}\{M[i-1,w], v(i) + M[i-1,w-w(i)]\}$

9 **return** $M[n,W]$

7.2.1 Complexidade

A complexidade do algoritmo de programação dinâmica proposto é de $\Theta(|I|W)$ ou seja, não pode ser considerado resolvível em tempo polinomial.

Desafio

Pense sobre o projeto de um algoritmo que encontre os itens presentes na solução ótima para o problema da mochila.

7.3 Subsequência Comum Mais Longa

Um filamento de DNA consiste em uma cadeia de moléculas denominadas bases. Dentre essas bases têm-se Adenina (A), Guanina (G), Citosina (C) e Timina (T). Dois indivíduos podem ser considerados semelhantes se suas cadeias de DNA também o são. Nesse contexto, emerge a necessidade de comparar duas cadeias que compartilhem de uma terceira cadeia formada em uma mesma ordem, mas não necessariamente na mesma sequência. Chama-se esse problema de Subsequência Comum Mais Longa (SCML, ou *Longest Common Subsequence*) (CORMEN et al., 2012).

Considere que as cadeias $X = \langle x_1, x_2, \ldots, x_m \rangle$ e $Y = \langle y_1, y_2, \ldots, y_n \rangle$, procura-se encontrar a subsequência mais longa $Z = \langle z_1, z_2, \ldots, z_k \rangle$. Para compreender as propriedades o teorema a seguir, assuma que dada uma cadeia W, $W_i = \langle w_1, w_2, \ldots, w_i \rangle$ é o prefixo de W até o i-ésimo símbolo.

A seguir, há um importante teorema quanto a esse problema que define a subestrutura ótima

Teorema 7.3.1. *Sejam $X = \langle x_1, x_2, \ldots, x_m \rangle$ e $Y = \langle y_1, y_2, \ldots, y_n \rangle$ as sequências, e $Z = \langle z_1, z_2, \ldots, z_k \rangle$ uma subsequência mais longa de X e Y, têm-se:*

1. *Se $x_m = y_n$, então $z_k = x_m = y_n$ e Z_{k-1} uma subsequência mais longa de X_{m-1} e Y_{n-1};*
2. *Se $x_m \neq y_n$, então $z_k = x_m$ implica que Z é uma subcadeia mais longa de X_{m-1} e Y;*
3. *Se $x_m \neq y_n$, então $z_k = y_n$ implica que Z é uma subcadeia mais longa de X e Y_{n-1}.*

Prova: Para o item (1), se $z_k \neq x_m$, então podemos anexar a $x_m = y_n$ a Z para obter a subsequência comum mais longa de comprimento $k+1$, contradizendo de que Z é uma subsequência comum mais longa. Desse modo, deve-se ter $z_k = x_m = y_n$. O prefixo Z_{k-1} é uma subsequência comum de comprimento $k-1$ de X_{m-1} e Y_{n-1}. Deseja-se mostrar que ela é uma subsequência comum mais longa. Supõe-se por contradição, que há uma sequência comum W de X_{m-1} e Y_{n-1} com comprimento maior que $k-1$. Então, anexar

$x_m = y_n$ a W produz uma subsequência máxima de X e Y cujo o comprimento é maior que k, o que é uma contradição.

Para o item (2), se $z_k \neq x_m$, então Z é a subsequência comum de X_{m-1} e Y. Se existisse uma subsequência comum W de X_{m-1} e Y com comprimento maior que k, então W seria também uma subsequência comum de X_m e Y, contradizendo a suposição de que Z é uma subsequência comum mais longa.

Para o item (3), a prova é simétrica à explicação ao item (2). ■

Segue a recorrência da subsequência mais longa, considerando as cadeias $X = \langle x_1, x_2, \ldots \rangle$ e $Y = \langle y_1, y_2, \ldots \rangle$:

$$OPT(m,n) = \begin{cases} 0 & \text{para } m = 0 \text{ ou } n = 0, \\ OPT(m-1, n-1) + 1, & \text{quando } x_m = y_n, \\ \textbf{max}\{OPT(m-1,n), OPT(m,n-1)\}, & \text{quando } x_m \neq y_n. \end{cases} \tag{7.5}$$

Algoritmo 42: Algoritmo para encontrar o comprimento da Subsequência Comum Mais Longa.

Input : duas palavras $X = \langle x_1, x_2, \ldots, x_m \rangle$ e $Y = \langle y_1, y_2, \ldots, y_n \rangle$.

```
1  Criar matriz M ∈ ℤ^{(m+1)×(n+1)}
2  Criar matriz B ∈ {←, ↖, ↑}^{m×n}
3  for i ← 0 to m do
4      M[i, 0] ← 0
5  for j ← 0 to n do
6      M[0, j] ← 0
7  for i ← 1 to m do
8      for j ← 1 to n do
9          if x_i = y_j then
10             M[i, j] ← M[i − 1, j − 1] + 1
11             B[i, j] ← "↖"
12         else
13             if M[i − 1, j] ≥ M[i, j − 1] then
14                 M[i, j] ← M[i − 1, j]
15                 B[i, j] ← "↑"
16             else
17                 M[i, j] ← M[i, j − 1]
18                 B[i, j] ← "←"
19 return (M, B)
```

7.3.1 Complexidade

A complexidade de tempo computacional para se determinar a subsequência comum mais longa é de $\Theta(|X| \cdot |Y|)$ devido a dominância assintótica da complexidade do Algoritmo 43.

Algoritmo 43: SCMLalg – Algoritmo para encontrar a Subsequência Comum Mais Longa.

Input : duas palavras $X = \langle x_1, x_2, \ldots, x_m \rangle$ e $Y = \langle y_1, y_2, \ldots, y_n \rangle$, as matrizes M e B geradas no Algoritmo 42, os inteiros i e j correspondentes aos índices de X e Y analisados na chamada atual, uma *String result*.

```
1  if i = 0 ∨ j = 0 then
2      return
3  if B[i, j] = "↖" then
4      SCMLalg(X, Y, i − 1, j − 1, result)
5      result ← result · x_i
6  else
7      if B[i, j] = "↑" then
8          SCMLalg(X, Y, i − 1, j, result)
9      else
10         SCMLalg(X, Y, i, j − 1, result)
```

7.4 Multiplicação de Cadeias de Matrizes

O problema de Multiplicação de Cadeias de Matrizes busca determinar quais matrizes multiplicar primeiro para reduzir o número de operações em uma multiplicação. A multiplicação de matrizes é uma operação binária que demanda $\Theta(nml)$ para multiplicar as matrizes $A_{n\times m}$ e $B_{m\times l}$.

Dada um conjunto de matrizes $A_1 A_2 \ldots A_j$ arranjadas respeitando os requisitos de linha e coluna necessários para que ocorra a multiplicação, deseja-se saber qual a ordem de multiplicações tal que o número de operações seja o mínimo possível.

Considere $A_{i..j}$ como a matriz resultante da multiplicação entre as matrizes da sequência $A_i, A_{i+1}, \ldots, A_j$. Considere também $p_0, p_1, \ldots, p_j$ os valores de linha e coluna tal que A_i tenha p_{i-1} linhas e p_i colunas. A recorrência para encontrar o valor ótimo de número de operações pode ser visualizada abaixo:

$$OPT(i,j) = \begin{cases} 0, \text{ se i=j} \\ \\ \mathbf{min}_{i \le k < j} \{ OPT(i,k) + OPT(k+1,j) + p_{i-1} p_k pj \} \end{cases} \tag{7.6}$$

. onde $OPT(i,j)$ representa o valor mínimo de operações para realizar a multiplicação das matrizes $A_i, A_{i+1}, \ldots, A_j$.

O Algoritmo 44 descreve a resolução do problema. A matriz M realiza a memorização dos valores e S possui dados necessários para construir a solução.

O Algoritmo 45 exibe como construir a solução.

Algoritmo 44: Encontra ordem de multiplicação das matriz.

Input :o conjunto p

```
1  Criar tabela M_{n×n}
2  Criar tabela S_{n×n}
3  for i ← 1 to n do
4      M[i,i] ← 0
5  for l ← 2 to n do
6      for i ← 1 to n − l − 1 do
7          j ← i + l − 1
8          M[i,j] ← ∞
9          for k ← i to j − 1 do
10             q ← M[i,k] + M[k+1,j] + p_{i−1} p_k, p_j
11             if q < M[i,j] then
12                 M[i,j] ← q
13                 S[i,j] ← k
```

Algoritmo 45: PRINT-OPTIMAL-PARENS.

Input :a matriz s, os inteiros i e j

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

7.4.1 Complexidade

A complexidade de tempo cadeia computacional para o Algoritmo 44 é de $O(n^3)$.

CAPÍTULO 8

NP-Completude e Reduções

Muitos problemas que foram visitados na disciplina são conhecidos como **problemas de otimização**. Para eles, há um valor que quantifica uma solução. Os algoritmos que resolvem esses problemas devem encontrar a solução de valor máximo ou mínimo, também chamado de valor ótimo. A função que mapeia a solução para seu valor é chamada de função objetivo [1].

No entanto, as definições e teorias da NP-Completude baseiam-se em problemas diferentes, chamados de **problemas de decisão**. Para esses problemas, espera-se uma resposta **sim** ou **não**. O uso desses problemas vem de resultados obtidos através de demonstrações utilizando a máquina de Turing, que aceita (decisão **sim**) ou rejeita (decisão **não**) uma palavra de entrada.

Embora a NP-Completude trata de problemas de decisão, pode-se tirar proveito de uma relação conveniente com problemas de otimização. Geralmente, pode-se representar um problema de otimização em um problema de decisão, relacionando-o a um determinado valor limite em relação ao objetivo do problema de otimização. Um exemplo citado por Cormen et al. (2012) é um problema de otimização cujo objetivo é encontrar o caminho com o menor número de arcos entre os vértices u e v em um grafo.

[1] Alguns problemas de otimização necessitam de mais de uma função objetivo. Por exemplo, pode-se desejar fazer o máximo de entregas possíveis em um percurso de menor curso. Uma função objetivo poderia representar a quantidade de entregas realizadas; outra função objetivo representaria o custo do percuso.

Sua versão como problema de decisão pode impor um limite k: é possível estabelecer um caminho entre u e v com no máximo k arestas?

Este capítulo tem o objetivo de revisar rapidamente os conceitos e requisitos relacionadas a NP-Completude.

8.1 Máquinas de Turing

Uma máquina de Turing é formalmente definida como $M = (\Sigma, Q, \Gamma, \delta, q_0, q_{aceita}, q_{rejeita})$ (SIPSER, 2007):

- Σ é o conjunto finito de símbolos da palavra de entrada (símbolos do alfabeto). $\sqcup \notin \Sigma$ não pertence ao alfabeto e marca uma célula que não possui símbolo (célula vazia);
- Q é o conjunto finito de estados;
- Γ é o conjunto finito de símbolos que podem ser lidos ou escritos na fita, no qual $\Sigma \subset \Gamma$, $\{\sqcup, <\} \subseteq \Gamma$;
- $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{E, D\}$ é a função de transição, onde E e D representam a próxima célula vizinha a ser visitada pela cabeça de leitura e gravação;
- $q_0 \in Q$ é o estado inicial;
- $q_{aceita} \in Q - \{q_{rejeita}\}$ é o estado que denota a aceitação da palavra de entrada;
- $q_{rejeita} \in Q - \{q_{aceita}\}$ é o estado que denota a rejeição da palavra de entrada.

A **fita de entrada** é o dispositivo de "memória" que armazena a palavra de entrada. A fita é limitada à esquerda e infinita à direita. Na extremidade esquerda da fita de entrada, há o símbolo < seguido da palavra de entrada. As demais células possuem o símbolo $\sqcup$.

A **cabeça de leitura e gravação** está sempre sobre uma célula. A função de transição δ determina o qual o próximo estado dado o símbolo da célula que está sob a cabeça de

leitura e gravação. Esta função também determina qual o símbolo deverá ser substituir o símbolo antigo da célula e qual a próxima posição da cabeça. Se a função especificar E, a cabeça deverá se movimentar para a célula adjacente da esquerda; se especificar D, a cabeça se movimenta para a célula adjacente da direita.

Há três conclusões possíveis quando uma máquina de Turing recebe uma palavra de entrada. Caso o estado atual seja q_{aceita}, a máquina pára e a palavra de entrada é aceita (pertence a linguagem que a máquina reconhece). Caso o estado atual seja $q_{rejeita}$, a máquina pára e a palavra de entrada é rejeitada (não pertence a linguagem que a máquina reconhece). Uma última conclusão possível seria a máquina não parar, executando infinitamente sobre a palavra de entrada. Esta última conclusão tem relação direta com problemas (linguagens) que os computadores não conseguem resolver.

8.1.1 Algoritmos e a Máquina de Turing

Em 1900, David Hilbert proferiu uma palestra no Congresso Internacional de Matemática em Paris. Nesta apresentação, ele enumerou 23 problemas matemáticos e os colocou como um desafio para o século XX. O décimo problema era sobre algoritmos (SIPSER, 2007).

O décimo problema de Hilbert era conceber um algoritmo que testasse se um polinômio possui uma raiz inteira. Até então não se tinha o conceito de algoritmo, então a ideia dele era identificar *um processo que determine a raiz com um número finito de operações*.

Hoje, já se sabe que não existe algoritmo para tal tarefa. Na época, isto era impossível de se conceber, pois não existia uma concepção do que era um algoritmo e quais suas limitações.

A definição chegou nos artigos de Alonzo Church e Alan Turing em 1936. Church usou um sistema denominado $\lambda-$cálculo e Turing usou suas máquinas (máquina de Turing). Estas duas definições se demonstraram equivalentes. A noção precisa de algoritmo veio a ser chamada de **Tese de Church-Turing**. Deste modo, a noção intuitiva

de algoritmos é igual a algoritmos de máquinas de Turing.

Em 1970, a tese de Church-Turing proveu a definição necessária para que Yuri Matijasevic (inspirado no trabalho de Martin Davis, Hilary Putnam, e Julia Robinson) demonstrasse que o décimo problema de Hilbert não existe algoritmo para se testar se um polinômio tem raizes inteiras.

8.1.2 Redutibilidade

Uma redução é uma meneira de converter um problema em outro de forma que uma solução para o segundo problema possa ser utilizada para resolver o primeiro. Sejam A e B dois problemas, se A reduz B, pode-se usar uma solução de B para resolver A.

Geralmente, são utilizadas para demonstrar que problemas são computacionalmente insolúveis.

8.2 Verificação de Problemas

Um algoritmo que verifica um problema recebe como entrada a instância para o problema e um certificado. Esse certificado é uma construção que auxília na identificação se aquele problema decide sim. Normalmente, o certificado é a própria solução do problema.

Para exemplificar a verificação, imagine o problema

$$HAM-CYCLE = \{\langle G\rangle : G \text{ é um grafo hamiltoniano }\}.$$

O problema de decisão $HAM-CYCLE$ determina se um grafo não-dirigido e não-ponderado $G = (V, E)$ possui um ciclo hamiltoniano, ou seja, um ciclo que passa por todos os vértices de G sem repetição de vértices. Um verificador para o problema $HAM-CYCLE$ deve identificar se dada um grafo não-dirigido e não-ponderado G e um certificado y deve determinar se y é um ciclo hamiltoniano para G. O certificado y

pode ser a representação do ciclo $\langle y_1, y_2, \ldots, y_n \rangle$, ou seja, uma sequência de vértices. O verificador terá que identificar se $\{y_i, y_{i+1}\} \in E$ para todo $i \in \{1, 2, \ldots, n-1\}$.

Note algo interessante: para $HAM-CYCLE$, não se conhece um algoritmo decisor cuja a função de complexidade de tempo seja polinomial; em contraste, um algoritmo verificador demanda tempo polinomial.

Uma definição mais formal é apresentada em Sipser (2007):

Definição 1. *Um verificador para uma linguagem A é um algoritmo V, onde $A = \{w | V$ aceita $< w, c >$ para alguma cadeia $c\}$.*

8.3 Complexidade de Tempo

Definição 2. *Seja M uma máquina de Turing que pára sobre todas as entradas, o tempo de execução ou complexidade de tempo de M é a função $f : \mathbb{N} \to \mathbb{N}$, onde $f(n)$ é o número máximo de passos que M usa sobre entradas de comprimento n. Se $f(n)$ for o tempo de execução de M, diz-se que M roda em tempo $f(n)$ e que M é uma máquina de Turing de tempo $f(n)$.*

8.3.1 Classes de Complexidade

As classes de complexidade são maneiras de agrupar problemas computacionais que possuam semelhanças quanto a dificuldade computacional para resolvê-los.

Definição 3. *Seja $t : \mathbb{N} \to \mathbb{R}^+$ uma função, define-se a classe de complexidade de tempo $TIME(t(n))$, como a coleção de todas as linguagens que são decidíveis por uma máquina de Turing determinística em tempo $O(t(n))$. $TIME = \{L | L$ é uma linguagem decidida por uma máquina de Turing determinística de tempo $O(t(n))\}$.*

Definição 4. *Seja $t : \mathbb{N} \to \mathbb{R}^+$ uma função, define-se a classe de complexidade de tempo $NTIME(t(n))$, como a coleção de todas as linguagens que são decidíveis por uma má-*

quina de Turing não-determinística em tempo $O(t(n))$. $NTIME = \{L|L$ é uma linguagem decidida por uma máquina de Turing não-determinística de tempo $O(t(n))\}$.

Definição 5. *Seja N uma máquina de Turing não-determinística decisora, seu tempo de execução é a função $f : \mathbb{N} \rightarrow \mathbb{N}$, onde $f(n)$ representa a profundidade da árvore de computações determinísticas e não a quantidade de nodos nesta árvore.*

8.3.1.1 A Classe P

P ou $PTIME$ é a classe constituída de problemas resolvidos em tempo determinístico polinomial.

Definição 6. *P ou $PTIME$ é a classe constituída de todas as linguagens decidíveis por uma máquina de Turing determinística em tempo polinomial. Em outras palavras*

$$P = \bigcup_k TIME(n^k), \tag{8.1}$$

onde k são valores constantes.

8.3.1.2 A Classe NP

NP ou $NPTIME$ é a classe de problemas resolvidos em tempo não-determinístico polinomial. Também pode ser definida como a classe de todos os problemas verificáveis em tempo polinomial determinístico[2].

Quanto a definição mais formal de (SIPSER, 2007), considera-se um verificador para uma linguagem A é um algoritmo V, onde $A = \{w|V$ aceita $< w, c >$ para alguma cadeia $c\}$. Mede-se o tempo de um verificador em termos apenas de comprimento de w, portanto um verificador de tempo polinomial executa a verificação em tempo polinomial no comprimento de w. Uma linguagem A é polinomialmente verificável se ela tem um verificador de tempo polinomial.

Definição 7. *NP e $NPTIME$ é a classe de linguagens que tem verificadores de tempo polinomial.*

[2] De acordo com Arora e Barak (2009), essa definição é mais fácil de compreender.

Definição 8. $NP = \bigcup_k NTIME(n^k)$, *onde k são valores constantes.*

8.3.1.3 P versus NP

A questão "$P = NP$?" é uma grande questão aberta. Ela basicamente poderia ser lida como "Todos os problemas verificados em tempo polinomial determinístico podem ser resolvidos em tempo polinomial determinístico?". Sabe-se que

$$NP \subseteq EXPTIME = \bigcup_k TIME(n^k), \tag{8.2}$$

k é uma constante.

8.3.1.4 NP-Completude

Um avanço importante na questão "P versus NP" surgiu nos anos iniciais da década de 70 com os trabalhos de Stephen Cook, Leonid Levin e Richard Karp. Stephen Cook e Leonid Levin descobriram que certos problemas em NP cuja a complexidade individual está relacionada a toda a classe. Se existir um algoritmo polinomial para um destes problemas, então todas os demais problemas da classe NP seriam decididos em tempo polinomial. Estes problemas são definidos como NP-Completos.

Um dos primeiros problemas NP-Completo (descoberto por Stephen Cook) foi o *problema da satisfazibilidade* $SAT = \{< \phi > | \phi$ é uma fórmula booleana satisfazível $\}$. ϕ é uma fórmula booleana composta por variáveis booleanas usando as operações do **e** ($\wedge$), **ou** ($\vee$), e **não** ($\neg$) lógicos. O Teorema de Cook-Levin relaciona a complexidade de SAT às complexidades de todos os problemas em NP.

Teorema 8.3.1. $SAT \in P$ *se e somente se* $P = NP$.

Redutibilidade em Tempo Polinomial

Redutibilidade em tempo polinomial é o método central do teorema de Cook-Levin. Quando um problema A é eficientemente redutível ao problema B, uma solução efici-

ente para B pode ser utilizada para resolver A eficientemente.

Definição 9. *Uma função $f : \Sigma^* \to \Sigma^*$ é uma função computável em tempo polinomial se existe alguma máquina de Turing de tempo polinomial M que pára com exatamente $f(w)$ na sua fita, quando iniciada com a entrada de w.*

Definição 10. *A linguagem A é redutível por mapeamento em tempo polinomial, ou simplesmente redutível em tempo polinomial, ou ainda redutível em tempo polinomial de muitos-para-um, à linguagem B em símbolos $A \leq_P B$, se existe uma função computável em tempo polinomial $f : \Sigma^* \to \Sigma^*$, onde para toda w, $w \in A \iff f(w) \in B$. A função f é denominada redução de tempo polinomial de A para B.*

Definição 11. *Uma linguagem B é NP-Completa se satisfaz duas condições:*

1. *$B \in NP$;*
2. *toda $A \in NP$ é redutível em tempo polinomial a B.*

Teorema 8.3.2. *Se B for NP-Completa e $B \leq_P C$ para $C \in NP$, então C é NP-Completa.*

Prova: Prove ...

Teorema 8.3.3. *SAT é NP-Completo.*

Para provar que SAT é NP-Completo, deve-se primeiro demonstrar que ele está em NP. Em seguida, deve-se mostrar que todas as linguagens em NP são redutíveis polinomialmente a SAT. Para este último, deve-se construir uma redução polinomial para cada linguagem A em NP para SAT. A redução para A recebe uma cadeia w e produz uma fórmula booleana ϕ que simula a máquina NP para A sobre a entrada w. Se a máquina aceita, ϕ é satisfazível.

Prova: Primeiro, deve-se mostrar que SAT está em NP. Uma máquina de tempo polinomial não-determinístico pode encontrar se existe uma atribuição às variáveis binárias é satisfazível. Uma árvore de tentativas não-determinísticas possui profundidade igual ao número de variáveis binárias. Portanto, $SAT \in NP$.

A outra parte da prova deve demonstrar que toda linguagem $A \in NP$ é redutível em tempo polinomial a SAT. Seja N uma máquina de Turing não-determinística que decide A em tempo n^k para alguma constante k.

Um *tableau* para N sobre w é uma tabela $\{0,1\}^{n^k \times n^k}$ cujas linhas são as configurações de um ramo da computação de N sobre a entrada w. Na primeira e na última célula de cada linha, assume-se que há o símbolo #. Na primeira linha, há a configuração inicial (estado inicial + w). Um *tableau* é de aceitação se qualquer linha dele for uma configuração de aceitação. O problema de determinar se N aceita w é equivalente ao problema de se determinar se existe um *tableau* de aceitação para N sobre w.

Com isto, pode-se introduzir a descrição da redução polinomial f de A para SAT. Sobre a entrada w, a redução produz uma fórmula ϕ. As variáveis de ϕ representam cada uma das $(n^k)^2$ células do *tableau*. Define-se $x_{i,j,s}$ uma variável binária de ϕ, onde i e j representam a coordenada da célula no *tableau* e $s \in C = Q \cup \Gamma \cup \{\#\}$ o símbolo ou o estado. Se $x_{i,j,s}$ é 0, então o símbolo $s \in C$ está presente na célula de coordenada (i,j) no *tableau*.

Deve-se projetar um ϕ de modo que se houver uma atribuição às variáveis que satisfaça ϕ, então N aceita w. A formula se divide em quatro partes $\phi = \phi_{célula} \wedge \phi_{início} \wedge \phi_{movimento} \wedge \phi_{aceita}$.

A parte $\phi_{célula}$ garante que cada célula tenha um símbolo e apenas uma célula tenha um símbolo. Logo, tem-se a definição de

$$\phi_{célula} = \bigwedge_{1 \leq i,j \leq n^k} \left[\left(\bigvee_{s \in C} x_{i,j,s} \right) \wedge \left(\bigwedge_{t,u \in C, u \neq t} (\neg x_{i,j,u} \vee \neg x_{i,j,t}) \right) \right]. \tag{8.3}$$

A parte $\phi_{início}$ garante que a primeira linha seja uma configuração inicial de N sobre

w. Deste modo,

$$\phi_{início} = x_{1,1,\#} \wedge x_{1,2,q_0} \wedge x_{1,3,w_1} \wedge x_{1,4,w_2} \wedge \ldots \wedge x_{1,n^k-1,\sqcup} \wedge x_{1,n^k,\#}. \tag{8.4}$$

Para representar a aceitação de N, define-se a parte de aceitação

$$\phi_{aceita} = \bigvee_{1 \leq i,j \leq n^k} x_{i,j,q_{aceita}}. \tag{8.5}$$

$\phi_{movimento}$ garante cada linha correspondente a uma configuração siga legalmente uma configuração da linha precedente conforme definido nas transições de N. Para isto, define-se uma sequência de trechos para ϕ configurando as possíveis janelas legais que representam as transições possíveis de N. Uma janela legal é uma matriz 2×3 dentro do *tableau*. Por exemplo, a Janela Legal 2 define demonstram uma janela legal para a transição $\delta(q_1, a) = \{(q_2, b, D)\}$.

a	q_1	a
a	b	q_2

Tabela 1 – Um exemplo de janela legal para a transição $\delta(q_1, a) = \{(q_2, b, D)\}$.

Logo, $\phi_{movimento} = \bigwedge_{1 \leq i \leq n^k, 1 \leq j \leq n^k} \Big($ a janela (i, j) é legal$\Big)$. Substitui-se "a janela legal" por

$$\bigvee_{w \in W(i,j)} \Big(x_{i,j-1,a_1} \wedge x_{i,j,a_2} \wedge x_{i,j+1,a_3} \wedge x_{i+1,j-1,a_4} \wedge x_{i+1,j,a_5} \wedge x_{i+1,j+1,a_6}\Big), \tag{8.6}$$

onde $W(i, j)$ é o conjunto de todas as janelas legais para a coordenada (i,j), a_1, a_2, a_3, a_4, a_5, e a_6 são os símbolos de uma janela legal.

A seguir demonstra-se que a redução demanda tempo polinomial determinística por relacionar o tamanho de ϕ. O número de variáveis de ϕ é $n^{2k}|C|$, ou seja, o tamanho do *tableau* vezes o número de símbolos. Como $|C|$ depende apenas de N e não de $n = |w|$, então diz-se que o total de variáveis é $O(n^{2k})$.

Em $\phi_{célula}$, há um fragmento de tamanho fixo para cada variável, portanto demanda tempo $O(n^{2k})$. Para formular a parte $\phi_{início}$ é necessário uma construção de tamanho

igual a quantidade de células da primeira linha do *tableau*, portanto demanda tempo $O(n^k)$. As fórmulas ϕ_{aceita} e $\phi_{movimento}$ possuem um fragmento fixo para cada célula do *tableau*, então o tempo computacional é $O(n^{2k})$. A formulação completa de ϕ é um polinômio sobre a variável n (tamanho do problema) então a redução é polinomial.

Deste modo, conclui-se que $SAT \in NP$-Completo. ■

Definição 12. *Uma linguagem B é NP-Difícil se toda $A \in NP$ é redutível em tempo polinomial a B.*

Com a definição acima, podemos assumir que NP-Completo $\subseteq$ NP-Difícil. Mesmo que $P = NP$, há problemas em NP-Difícil que não seriam resolvidos em tempo polinomial determinístico.

Mantenha em mente que:

- todo problema que "resolve" toda uma classe e pertence a mesma é dito COMPLETO;
- todo problema que "resolve" toda uma classe é dito DIFÍCIL.

Logo, todo problema COMPLETO é DIFÍCIL, mas não o contrário.

Teorema de Cook-Levin com CIRCUIT-SAT

Em 1971, Cook e Levin provaram que SAT (por Cook) e 3-SAT (por Levin) provaram que esses problemas resolviam toda a classe NP. Ou seja, encontraram os primeiros NP-Completos. Kleinberg e Tardos (2005) coloca que talvez a forma mais simples de entender esse teorema seja utilizar uma variação desses dois problemas, chamado aqui de $CIRCUIT-SAT$. Para esse problema, há um circuito K que pode ser representado por um grafo acíclico, no qual os vértices podem representar:

- Os vértices de entradas podem ser rotulados com valores 0 ou 1, ou podem estar associados a uma variável v_i distinta (sem valor ainda definido para 0 ou 1);

- Cada outro vértice que não está na entrada está associado a um dos seguintes operadores booleanos $\wedge$ (2 entradas), $\vee$ (2 entradas) ou $\neg$ (1 entrada);
- um dos vértices é o resultante e define a aceitação (1) ou a rejeição (0) da entrada.

O problema $CIRCUIT-SAT$ deve determinar se há um conjunto de valores binários associados às variáveis booleanas de entrada cuja saída resulte em 1.

Utiliza-se agora o problema para se ter uma ideia da prova do Teorema de Cook-Levin através de $CIRCUIT-SAT$. Considere um algoritmo que toma uma entrada de tamanho n e produza uma resposta SIM ou NÃO. Esse algoritmo pode ser representado por um circuito do tipo $CIRCUIT-SAT$. O circuito deve responder 1 quando a resposta do algoritmo é SIM e 0 quando a resposta for NÃO. A transformação de um algoritmo para um circuito demanda um número de passos polinomial em n, então circuito terá tamanho polinomial.

Agora imagine um problema arbitrário X em NP. Deseja-se decidir se uma entrada s pertence ou não a X usando uma "black-box" que pode resolver instâncias do $CIRCUIT-SAT$. Será tentado demonstrar que $X \leq_P CIRCUIT-SAT$.

Sabe-se que X possui um certificador eficiente $B(s,t)$ que determina se $s \in X$. Agora a pergunta é, existe um certificado t com tamanho polinomial em relação ao comprimento de s? Considerando que s possui n bits e t possui $p(n)$ bits, somando uma entrada de tamanho $n+p(n)$, converte-se $B(s,t)$ para um circuito K de tamanho polinomial com $n+p(n)$ vértices de entrada. Os primeiros n vértices serão receberam literalmente os bits em s e as demais entradas receberão $p(n)$ variáveis que representarão o certificado t.

Agora simplesmente observa-se que $s \in X$ sse há um conjunto de bits de entrada em K que produzem a saída 1, ou seja, se K for satisfazível.

Problemas de Otimização e NP

De acordo com (ARORA; BARAK, 2009), para que um problema esteja em NP, ele deve ser um problema de decisão. Problemas de otimização são problemas claramente mais

difíceis que suas versões correspondentes de decisão. Logo, problemas de otimização podem ser NP-Difíceis, mas não poderiam ser NP. No entanto, se P=NP, então a versão de decisão é resolvida em tempo polinomial, então a versão de otimização também pode ser resolvida em tempo polinomial.

CAPÍTULO 9

Algoritmos Aproximados e Buscas Heurísticas

9.1 Algoritmos Aproximados

Algoritmos aproximados não garantem a solução ótima para o problema, mas sim uma solução aproximada por uma razão. Essa razão é chamada de razão de aproximação e é denotada por $\rho(n)$ para toda entrada de tamanho n (CORMEN et al., 2012). Considerando C^* o valor objetivo da solução ótima, e C o valor objetivo obtido por um algoritmo aproximado, uma razão de aproximação é dada por

$$\mathbf{max}\left\{\frac{C}{C^*}, \frac{C^*}{C}\right\} \leq \rho(n). \tag{9.1}$$

Se a razão de aproximação de um algoritmo é $\rho(n)$ diz-se que ele é um algoritmo de $\rho(n)-$aproximação. A razão nunca é menor que 1 (CORMEN et al., 2012).

Existem problemas para os quais se conhece um algoritmo de tempo polinomial com razões de aproximação constante, mas existem problemas que a razão de aproximação é uma função relacionada ao tamanho do problema (CORMEN et al., 2012).

Existem também problemas nos quais pode-se permutar o tempo de computação por uma aproximação melhor. Algoritmos aproximados com essa característica são

chamados de esquemas de aproximação. Um esquema de aproximação é um algoritmo de aproximação que adota como entrada não somente uma instância do problema, mas também um valor $\epsilon > 0$, tal que para todo ϵ fixo, o esquema do algoritmo é de $(1+\epsilon)-$aproximação. O tempo de um esquema de aproximação pode aumentar muito rapidamente a medida que ϵ diminui.

9.1.1 O Problema do Caixeiro Viajante

O problema do caixeiro viajante é dado por encontrar um ciclo hamiltoniano de menor custo em um grafo não-dirigido e ponderado $G = (V, E, w : E \rightarrow \mathbb{R}^+_*)$. Cormen et al. (2012) adiciona uma notação $c(A)$ que seria o custo total no subconjunto de arestas $A \subseteq E$

$$c(A) = \sum_{\{u,v\} \in A} w(\{u, v\}).$$

Cormen et al. (2012) define que $c(A)$ deve respeitar a desigualdade triangular, então

$$w(\{u, w\}) \leq w(\{u, v\}) + w(\{v, w\})$$

para todo $u, v, w \in V$. Ela basicamente traduz a ideia "o modo menos caro de ir de um lugar u para um lugar w é ur diretamente, sem nenhuma etapa intermediária".

O algoritmo 2-aproximado para o problema do caixeiro viajante que respeita a desigualdade triangular pode ser visualizado no Algoritmo 46. A complexidade do algoritmo acompanha a do que demanda o algoritmo de PRIM, portanto, o algoritmo 2-aproximado demanda tempo polinomial determinístico.

Algoritmo 46: Algoritmo 2-aproximado para o Problema do Caixeiro Viajante

Input :Grafo não-dirigido e ponderado $G = (V, E, w)$

1 $r \leftarrow$ selecionar um vértice arbitrariamente um vértice
2 obter a árvore geradora mínima T para G partindo de r usando o algoritmo PRIM
3 seja H a lista de vértices, ordenados de acordo com a pré-ordem em T
4 **return** H sem repetiçao de vértices

Teorema 9.1.1. *O Algoritmo 46 é um algoritmo de tempo polinomial de 2-aproximação do problema do Caixeiro Viajante com a desigualdade de triângulos.*

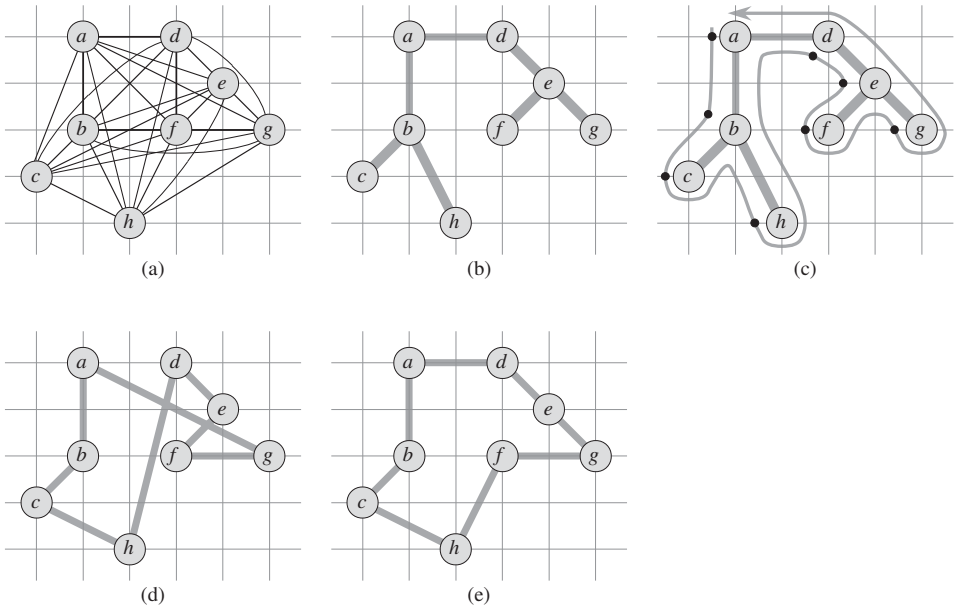


Figura 12 – Exemplo da execução do algoritmo de 2-aproximação para o PCV: (a) é o grafo de entrada; (b) é a árvore geradora mínima T; (c) é o caminho realizado na pré-ordem em T; (d) é a solução obtida pelo algoritmo; (e) é a solução ótima (CORMEN et al., 2012).

Prova: Já foi visto que o algoritmo executa em tempo polinomial.

Seja H^* a solução ótima para o problema. Obtém-se a árvore geradora mínima T na linha 2, o que gera um limite inferior

$$c(T) \leq c(H^*), \tag{9.2}$$

pois a árvore geradora mapeia os vértices de menor custo para conectar todos os vértices.

Quando se faz a pré ordem da linha 3, considere a formação de uma lista W na qual adiciona-se o vértice no final da lista a cada visita no vértices e no retorno da visita a uma adjacência. Para o exemplo da Figura 12, $W = \langle a, b, c, b, h, b, a, d, e, f, e, g, e, d, a\rangle$. Então

$$c(W) = 2c(T), \tag{9.3}$$

pois nesse processo, visita-se cada aresta duas vezes. Considerando as Equações (9.2) e (9.3) têm-se o seguinte limite inferior

$$c(W) \leq 2c(H^*). \tag{9.4}$$

Ao remover os vértices repetidos em W antes de retornar (linha 4), não há um aumento no custo, devido a propriedade de deseigualdade triangular (Equação (9.2)). Então dada a subsequência x, y, z em W, ao remover y não aumentará o custo de W, pois $w(\{x,z\}) \leq w(\{x,y\}) + w(\{y,z\})$. Considerando H sendo W sem vértices repetidos têm-se

$$c(H) \leq c(W). \tag{9.5}$$

Considerando o que foi relatado nas Equações (9.2), (9.3) e (9.5) têm-se:

$$c(H) \leq c(W) \leq 2c(H^*). \tag{9.6}$$

■

9.2 Heurísticas

Heurísticas são métodos computacionais que tratam problemas de otimização e tentam explorar um conhecimento específico do problema para o qua não há garantia de solução ótima. Frequentemente, heurísticas procuram por soluções que possuem características presentes em soluções ótimas (ROTHLAUF, 2011).

Dois tipos de heurísticas clássicas são (ROTHLAUF, 2011):

- Heurísticas de construção: constróem uma solução incrementalmente a partir de uma solução vazia. Executam um processo iterativo que usa informações sobre o problema para compor uma solução. Elas não tentam melhorar uma solução já criada. Terminam de executar quando a solução estiver completa
- Heurísticas de melhoria: iniciam com uma solução completa e tentam iterativamente melhorar a solução. Terminam quando um ótimo local[1] é encontrado.

Há muitas heurísticas de propósito geral modernas. Elas geralmente assumem os nomes de métodos de otimização heurísticos ou metaheurísticas. Elas possuem

[1] Ótimo local é uma solução no qual apenas mudanças significativas poderiam melhorar seu valor objetivo.

os objetivos de intensificar soluções para melhorar soluções já constituídas, ou de diversificar, que busca explorar novas áreas do espaço de busca.

Em Wolpert e Macready (1997), Wolpert and Macready apresentaram o teorema "não há almoço livre" para a otimização. Ele basicamente demonstra que não é possível definir uma heurística de propósito geral que funciona bem para todos os problemas de otimização.

Agradecimentos

Agradeço o Prof. Alexandre Gonçalves Silva pelo apoio e suporte quando a disciplina de Projeto e Análise de Algoritmos fora cedida aos meus cuidados.

Referências

ARORA, S.; BARAK, B. *Computational Complexity: A Modern Approach*. 1st. ed. New York, NY, USA: Cambridge University Press, 2009. ISBN 0521424267, 9780521424264. Citado 2 vezes nas páginas 150 e 156.

BLUM, M. et al. Time bounds for selection. *Journal of Computer and System Sciences*, v. 7, n. 4, p. 448 – 461, 1973. ISSN 0022-0000. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0022000073800339>. Citado na página 61.

CORMEN, T. H. et al. *Algoritmos: teoria e prática*. Rio de Janeiro: Elsevier, 2012. Citado 43 vezes nas páginas 5, 6, 7, 13, 17, 19, 21, 22, 23, 24, 25, 42, 43, 44, 49, 57, 59, 60, 61, 68, 69, 75, 83, 84, 86, 87, 90, 91, 92, 93, 94, 101, 121, 124, 127, 130, 136, 139, 145, 159, 160, 161 e 169.

DASGUPTA, S.; PAPADIMITRIOU, C. H.; VAZIRANI, U. *Algorithms*. 1. ed. New York, NY, USA: McGraw-Hill, Inc., 2008. ISBN 0073523402, 9780073523408. Citado 3 vezes nas páginas 57, 64 e 66.

GROSS, J. T.; YELLEN, J. *Graph Theory and Its Applications*. FL: CRC Press, 2006. Citado na página 171.

KLEINBERG, J.; TARDOS, E. *Algorithm Design*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2005. ISBN 0321295358. Citado 11 vezes nas páginas 21, 49, 50, 117, 118, 119, 133, 134, 136, 137 e 155.

NETTO, P. O. B. *Grafos: teoria, modelos, algoritmos*. São Paulo: Edgard Blucher, 2006. Citado 4 vezes nas páginas 73, 79, 80 e 83.

ROTHLAUF, F. *Design of Modern Heuristics: Principles and Application*. 1st. ed. [S.l.]: Springer Publishing Company, Incorporated, 2011. ISBN 3540729615, 9783540729617. Citado na página 162.

SIPSER, M. *Introdução à Teoria da Computação*. São Paulo, Brazil: Cengage Learning, 2007. Citado 4 vezes nas páginas 146, 147, 149 e 150.

WOLPERT, D. H.; MACREADY, W. G. No free lunch theorems for optimization. *Trans. Evol. Comp*, IEEE Press, Piscataway, NJ, USA, v. 1, n. 1, p. 67–82, abr. 1997. ISSN 1089-778X. Disponível em: <https://doi.org/10.1109/4235.585893>. Citado na página 163.

APÊNDICE A

Ordenação em Tempo Linear

A.1 Ordenação por Contagem

A ordenação por contagem é utilizada para ordenar inteiros na faixa de 1 a k para qualquer inteiro $k > 0$. Quando $k \in \Theta(n)$, então a ordenação por contagem demanda tempo $\Theta(n)$ (CORMEN et al., 2012).

Algoritmo 47: Ordenação Por Contagem

Input : Um vetor de A de tamanho n contendo os valores a serem ordenados, e um vetor B de tamanho n que receberá os valores de A em ordem crescente.

criar vetor C indexável de 0 até k
for $i \leftarrow 0$ ***to*** k **do**
    $C_i \leftarrow 0$
// Faz C_i ter a contagem do número de elementos iguais ao inteiro i em A.
for $j \leftarrow 1$ ***to*** $|A|$ **do**
    $C_{A_j} \leftarrow C_{A_j} + 1$
// Faz C_i conter o número de elementos menores ou iguais a i em A.
for $j \leftarrow 1$ ***to*** k **do**
    $C_i \leftarrow C_i + C_{i-1}$
for $j \leftarrow |A|$ ***downto*** 1 **do**
    $B_{C_{A_j}} \leftarrow A_j$
    $C_{A_j} \leftarrow C_{A_j} - 1$

A.2 Ordenação Digital

A.3 Ordenação por Balde

APÊNDICE B

Caminhos e Ciclos

Este capítulo tem o objetivo de introduzir o conceito de caminhos e ciclos, e seus principais problemas. Dois problemas clássicos serão definidos e algoritmos para os mesmos, apresentados.

Antes de iniciar a abordar os conteúdos deste capítulo, é importante entender o que é um caminho e um ciclo, para estabelecer suas diferenças no contexto de grafos. Um caminho[1] é uma sequência de vértices $\langle v_1, v2, \ldots v_n \rangle$ conectados por uma aresta ou arco. Gross e Yellen (2006) definem um caminho como um grafo com dois vértices com grau 1 e os demais vértices com grau 2, formando uma estrutura linear. Um ciclo (ou circuito)[2] é uma cadeia fechada de vértices $\langle v_1, v_2, \ldots, v_n, v_1 \rangle$ onde cada par consecutivo é conectado por uma aresta ou arco. É como um caminho com o fim e o início conectados.

B.1 Caminhos e Ciclos Eulerianos

Dado um grafo orientado ou não orientado $G = (V, E)$, um caminho Euleriano é uma "trilha" ou seja, uma sequência de arestas/arcos onde cada aresta/arco é visitada(o) uma única vez. O ciclo Euleriano é semelhante ao caminho, com exceção de que começa e termina na mesma aresta/arco. Um grafo é dito Euleriano se possui um ciclo Euleriano.

Os problemas de caminho e ciclo Euleriano surgiram com o conhecido problema das Sete Pontes de Königsberg por Euler em 1736. O problema consistia em atravessar as todas as sete pontes da cidade de Königsberg da Prussia (hoje Kaliningrado na Rússia) sem repetí-las.

---

1 Em inglês, chamado de *path*.

2 Em inglês, chamado de *cycle* ou *circuit*.

Desafio

Observando um mapa antigo das sete pontes, você consegue determinar o caminho Euleriano?

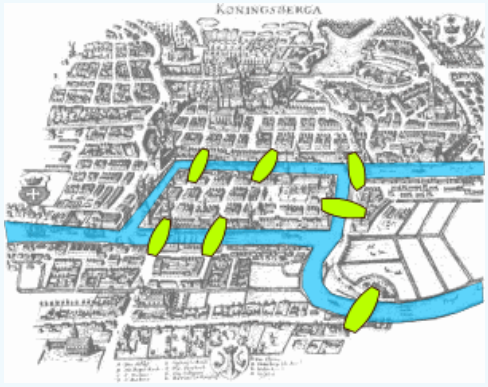


Figura 13 – Mapa das sete pontes de Königsberg na época de in Euler.

B.1.1 Algoritmo de Hierholzer

O algoritmo de Hierholzer (Algoritmo 48) foi desenvolvido em 1873. Ele identifica o ciclo Euleriano em tempo $O(|E|)$.

Algoritmo 48: Algoritmo de Hierholzer.

```
   Input  : um grafo G = (V, E)
1  foreach e ∈ E do
2      C_e ← false
3  v ← selecionar um v ∈ V arbitrariamente, que esteja conectado a uma aresta
   // "buscarSubcicloEuleriano" invoca o Algoritmo 49
4  (r, Ciclo) ← buscarSubcicloEuleriano(G, v, C)
5  if r = false then
6      return (false,null)
7  else
8      if ∃e ∈ E : C_e = false then
9          return (false,null)
10     else
11         return (true, Ciclo)
```

Algoritmo 49: Algoritmo de Auxiliar "$buscarSubcicloEuleriano$".

Input : um grafo $G = (V, E)$, um vértice $v \in V$, o vetor de arestas visitadas C

1 $Ciclo \leftarrow (v)$
2 $t \leftarrow v$
3 **repeat**
  `// Só prossegue se existir uma aresta não-visitada conectada a` $Ciclo$.
4 **if** $\nexists u \in N(v) : C_{u,v} =$ ***false*** **then**
5 **return** (**false**,**null**)
6 **else**
7 $\{v, u\} \leftarrow$ selecionar uma aresta $e \in E$ tal que $C_e =$ **false**
8 $C_{\{v,u\}} \leftarrow$ **true**
9 $v \leftarrow u$
  `// Adiciona o vértice` v `ao final do ciclo.`
10 $Ciclo \leftarrow Ciclo \cdot (v)$
11 **until** $v = t$
`/* Para todo vértice` x `no` $Ciclo$ `que tenha uma aresta adjacente não visitada.` `*/`
12 **foreach** $x \in \{u \in Ciclo : \exists \{u, w\} \in \{e \in E : C_e =$ ***false***$\}\}$ **do**
13 $(r, Ciclo') \leftarrow buscarSubcicloEuleriano(G, x, C)$
14 **if** $r =$ ***false*** **then**
15 **return** (**false**,**null**)
16 Assumindo que $Ciclo = \langle v_1, v_2, \ldots, x, \ldots, v_1 \rangle$ e $Ciclo' = \langle x, u_1, u_2, \ldots, u_k, x \rangle$, alterar $Ciclo$ para $Ciclo = \langle v_1, v_2, \ldots, \underline{x}, u_1, u_2, \ldots, u_k, x, \ldots, v_1 \rangle$, ou seja, inserir o $Ciclo'$ no lugar da posição de x em $Ciclo$.
17 **return** (**true**, $Ciclo$)

Desafio

Explique porque a complexidade de Algoritmo de Hierholzer é de $O(|E|)$.

Teorema B.1.1. *Um grafo não-orientado $G = (V, E)$ é (ou possui um ciclo) Euleriano se e somente se G é conectado e cada vértice tem um grau par.*

Prova: Para o grafo conectado G, para todo $m \geq 0$, considere $S(m)$ ser a hipótese de que se G têm m arestas e todos os graus dos vértices forem pares, então G é Euleriano.

Vamos a prova por indução.

A base da indução é o $S(0)$. Nessa hipótese, G não tem arestas, então para todo $v \in V$, $d_v = 0$. Como zero é par G é trivialmente Euleriano.

O passo da indução implica que as hipoteses $S(0) \wedge S(1) \wedge \ldots \wedge S(k-1) \implies S(k)$. Suponha um $k \geq 1$ e assuma que $S(1) \wedge \ldots \wedge S(k+1)$ é verdade. Precisa-se provar que $S(k)$ é verdade. Suponha que G tenha k arestas, é conectado e possui somente vértices com valor de grau par.

- Desde que G é um grafo conectado e possui vértices com grau par, o menor grau é 2. Então esse grafo G precisa ter um ciclo C.

- Suponha um novo grafo H gerado a partir de G sem as arestas que estão no ciclo C. Note que H pode estar desconectado. Pode-se dizer que H é a união dos componentes conectados $H_1, H_2, \ldots H_t$. O grau dos vértices cada H_i precisa ser par.

- Aplicando a hipótese de indução a cada H_i, que é $S(|E(H_1)|), \ldots, S(|E(H_t)|)$, cada H_i terá um ciclo Euleriano C_i.

- Pode-se criar um circuito Euleriano para G por dividir o ciclo C em ciclos C_i. Primeiro, comece em qualquer vértice em C_i e percorra até atingir outro H_i. Então, percorra C_i e volte ao C até atingir o próximo H_i.

Finalmente, G precisa ser Euleriano. Isso completa o passo da indução como $S(0) \wedge S(1) \wedge \ldots \wedge S(k-1) \implies S(k)$. Por esse princípio, para $m \geq 0$, $S(m)$ é verdadeiro. ■

B.2 Caminhos e Ciclos Hamiltonianos

Ciclos ou caminhos Hamiltonianos são aqueles que percorrem todos os vértices de um grafo apenas uma vez. Mais especificamente para um ciclo Hamiltoniano, o início e o fim terminam no mesmo vértice. O nome Hamiltoniano vem de William Rowan Hamilton, o inventor de um jogo que desafia a buscar um ciclo pelas arestas de dodecaedro (figura tridimensional de 12 faces).

Um grafo é dito Hamiltoniano se possui um ciclo Hamiltoniano.

Há $|V|!$ diferentes sequências de vértices que podem ser caminhos Hamiltonianos, então, um algoritmo de força-bruta demanda muito tempo computacional. O problema de decisão para encontrar um caminho ou ciclo Hamiltoniano é considerado NP-Completo.

B.2.1 Caixeiro Viajante

Dado um grafo completo[3] $G = (V, E, w)$ no qual V é o conjunto de vértices, E é o conjunto de arestas e $w : E \to \mathbb{R}^+$ é a função dos pesos (ou custo ou distâncias), busca-se pelo ciclo Hamiltoniano de menor soma total de peso (menor custo ou distância).

Um dos algoritmos mais eficientes para resolvê-lo é o de programação dinâmica Held-Karp (ou Bellman–Held–Karp, no Algoritmo 50). No entanto, o mesmo demanda tempo computacional de $O(2^{|V|}|V|^2)$.

Algoritmo 50: Algoritmo de Bellman-Held-Karp.

```
Input : um grafo G = (V, E = V × V, w)
for k ← 2 to |V| do
    C({k}, k) ← w({1, k})
for s ← 2 to |V| − 1 do
    foreach S ∈ {x ⊆ {2, 3, ..., |V|} : |x| = s} do
        foreach v ∈ S do
            C(S, v) ← min_{u≠v, u∈S} {C(S\{v}, u) + w({u, v})}
return min_{v∈V\{1}} {C({2, 3, ..., |V|}, v) + w({v, 1})}
```

[3] Em um grafo completo, o conjunto de arestas é definido por $E = V \times V$.

Desafio

Execute o Algoritmo 50 sobre o grafo $G = (V = \{1,2,3,4\}, E = V \times V, w)$, no qual $w\big(\{1,2\}\big) \rightarrow 10$, $w\big(\{1,3\}\big) \rightarrow 15$, $w\big(\{1,4\}\big) \rightarrow 20$, $w\big(\{2,3\}\big) \rightarrow 35$, $w\big(\{2,4\}\big) \rightarrow 25$ e $w\big(\{3,4\}\big) \rightarrow 30$.

A resposta deve ser 80.

APÊNDICE C

Revisão de Matemática Discreta

Conjuntos é uma coleção de elementos sem repetição em que a sequência não importa. No Brasil, utilizamos a seguinte notação para enumerar todos os elementos de um conjunto. Na Equação (C.1), é possível visualizar a representação de um conjunto denominado A, formado pelos elementos e_1, e_2, ..., e_n. Devido ao uso da vírgula como separador de decimais, usa-se formalmente o ponto-e-vírgula. Para essa disciplina, podemos utilizar a vírgula como o separador de elementos em um conjunto, desde que utilizados o ponto como separador de decimais[1]. Para dar nome a um conjunto, geralmente utiliza-se uma letra maiúscula ou uma palavra com a inicial em maiúscula.

$$A = \{e_1; e_2; \ldots; e_n\} \tag{C.1}$$

Há duas formas de definir conjuntos. A forma por enumeração por elementos, utiliza notação semelhante a da Equação (C.1). São exemplos de definição de conjuntos por enumeração:

- $N = \{\diamondsuit, \spadesuit, \heartsuit, \clubsuit\}$;
- $V = \{a, e, i, o, u\}$;
- $G = \{\alpha, \beta, \gamma, \delta, \epsilon, \zeta, \eta, \theta, \iota, \kappa, \lambda, \mu, \nu, \xi, \pi, \rho, \sigma, \tau, \upsilon, \phi, \chi, \psi, \omega\}$;
- $R = \{-100.9, 12.432, 15.0\}$;
- $D = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$.

A forma por descrição de propriedades utiliza-se de uma notação que evidencia a natureza de cada elemento pela descrição de um em um formato genérico. Por exemplo o conjunto D, descrito na Equação (C.2), denota um conjunto com os mesmos elementos em $\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$.

$$D = \{x \in \mathbb{Z} | x > 1 \wedge x \leq 10\} \tag{C.2}$$

. Então para que complicar utilizando uma notação não enumerativa? Por dois motivos: por questões de simplicidade, dado a quantidade de conjuntos; ou para representar conjuntos infinitos, como no exemplo dos inteiros pares $Pares = \{x \in \mathbb{Z} | x \equiv 0 (\mod 2)\}$.

[1] Nas anotações presentes nesse documento, utiliza-se a "notação americana". Para a Equação (C.1) , teria-se $A = \{e_1, e_2, \ldots, e_n\}$.

Para o conjunto dos pares, ainda podemos utilizar uma descrição mais informal, mas que é dependente da conhecimento sobre a linguagem Portuguesa: *Pares* = $\{x \in \mathbb{Z} |$ *x é inteiro e par* $\}$.

Para denotar a cardinalidade (quantidade de elementos) de um conjunto, utilizamos o símbolo "|". Para os conjuntos apresentados acima, é correto afirmar que:

- $|N| = 4$;
- $|V| = 5$;
- $|R| = 3$;
- $|D| = 10$;
- $|Pares| = \infty$.

A cardinalidade pode ser utilizada para identificar quantos símbolos são necessários para representar um elemento. Por exemplo, $|12,66| = 5$

Para denotar conjuntos vazios, adota-se duas formas de representação: $\{\}$ ou $\emptyset$. Utilizando o operador de cardinalidade, têm-se $|\{\}| = |\emptyset| = 0$.

Como principais operações entre conjuntos, pode-se destacar:

- União ($\cup$): união de dois conjuntos. Exemplo: $\{1,2,3,4,5\} \cup \{2,4,6,8\} = \{1,2,3,4,5,6,8\}$;
- Intersecção ($\cap$): intersecção de dois conjuntos. Exemplo: $\{1,2,3,4,5\} \cup \{2,4,6,8\} = \{2,4\}$;
- Diferença ($-$ ou $\setminus$): diferença de dois conjuntos. Exemplo $\{1,2,3,4,5\} \setminus \{2,4,6,8\} = \{1,3,5\}$;
- Produto cartesiano ($\times$): Exemplo $\{1,2,3\} \times \{A,B\} = \{(1,A),(2,A),(3,A),(1,B),(2,B),(3,B)\}$;
- Conjunto de partes (ou *power set*): o conjunto de todos os subconjuntos dos elementos de um conjunto. Para o conjunto $A = \{1,2,3\}$ o conjunto das partes seria $2^A = P(A) = \{\{\},\{1\},\{2\},\{3\},\{1,2\},\{1,3\},\{2,3\},\{1,2,3\}\}$.

Funções são representadas de forma diferente na matemática discreta. Busca-se estabelecer a relação entre um conjunto de domínio (entrada da função) e um contra-domínio (resposta da função). A Equação (C.3) exibe a forma como é utilizada para formalizar uma função. Nesse formato, passa-se a natureza da entrada e da saída de um problema. Por exemplo, a função que gera a correspondência entre o domínio dos inteiros positivos em base decimal para base binária seria $f : x \in Z^+ \rightarrow \{0,1\}^{\log_2(|x|+1)}$.

$$\text{nome da funcao} : \text{dominio} \rightarrow \text{contradominio} \tag{C.3}$$

Para representar uma coleção de itens onde a sequência importa e a repetição pode ocorrer, utiliza-se as tuplas. Uma tupla é representada da forma demonstrada na Equação (C.4).

$$A = (e_1, e_2, \ldots, e_n) \tag{C.4}$$

. Um exemplo de uma tupla, pode ser lista de chamada de uma turma ordenada lexicograficamente.

APÊNDICE D

Teoria da Complexidade

D.1 Máquinas de Turing

Uma máquina de Turing é definida formalmente como $(\Sigma, Q, \Gamma, \delta, q_0, q_{aceita}, q_{rejeita})$, onde:

- Σ é o conjunto finito de símbolos da palavra de entrada (símbolos do alfabeto). O símbolo $\sqcup \notin \Sigma$ não pertence ao alfabeto e marca uma célula que não possui símbolo (célula vazia);
- Q é o conjunto finito de estados;
- Γ é o conjunto finito de símbolos que podem ser lidos ou escritos na fita, onde $\Sigma \subset \Gamma$, $\{\sqcup, <\} \subseteq \Gamma$;
- $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{E, D\}$ é a função de transição, onde E e D representam a próxima célula vizinha a ser visitada pela cabeça de leitura e gravação;
- $q_0 \in Q$ é o estado inicial;
- $q_{aceita} \in Q - \{q_{rejeita}\}$ é o estado que denota a aceitação da palavra de entrada;
- $q_{rejeita} \in Q - \{q_{aceita}\}$ é o estado que denota a rejeição da palavra de entrada.

A **fita de entrada** é o dispositivo de "memória" que armazena a palavra de entrada. A fita é limitada à esquerda e infinita à direita. Na extremidade esquerda da fita de entrada, há o símbolo < seguido da palavra de entrada. As demais células possuem o símbolo $\sqcup$.

A **cabeça de leitura e gravação** está sempre sobre uma célula. A função de transição δ determina o qual o próximo estado dado o símbolo da célula que está sob a cabeça de leitura e gravação. Esta função também determina qual o símbolo deverá ser substituir o símbolo antigo da célula e qual a próxima posição da cabeça. Se a função especificar E, a cabeça deverá se movimentar para a célula adjacente da esquerda; se especificar D, a cabeça se movimenta para a célula adjacente da direita.

Há três conclusões possíveis quando uma máquina de Turing recebe uma palavra de entrada. Caso o estado atual seja q_{aceita}, a máquina pára e a palavra de entrada é aceita (pertence a linguagem que a máquina reconhece). Caso o estado atual seja $q_{rejeita}$, a máquina pára e a palavra de entrada é rejeitada (não pertence a linguagem

que a máquina reconhece). Uma última conclusão possível seria a máquina não parar, executando infinitamente sobre a palavra de entrada. Esta última conclusão tem relação direta com problemas (linguagens) que os computadores não conseguem resolver.

D.2 Turing-reconhecível

Uma linguagem L é Turing-reconhecível se e somente se existir uma máquina de Turing M que:

- aceita todas as palavras que pertençam a L; ou
- para palavras que não pertençam à L, M pára rejeitando ou executa indefinidamente.

L é chamada de Linguagem Recursivamente Enumerável.

D.3 Turing-decidível

Uma linguagem L é dita Turing-decidível se alguma máquina de Turing aceita qualquer palavra $w \in L$ e rejeta qualquer palavra $w \notin L$. L é chamada de Linguagem Recursiva.

D.4 Variantes da Máquina de Turing

...

D.5 Máquinas de Turing Determinísticas e Não-determinísticas

Toda a máquina de Turing não-determinística pode ser convertida em uma máquina de Turing determinística.
Prova: Suponha uma máquina de Turing não-determinística N. Suponha uma máquina de Turing simuladora D com três fitas. A fita 1 somente permite leitura e sempre contém a palavra de entrada. A fita 2 mantém uma cópia da fita de N em algum ramo de sua computação não-determinística. A fita 3 mantém o registro da posição de D na árvore de computação não-determinística de N.

Todos os nodos na árvore podem ter no máximo b filhos, onde b é o maior conjunto de possíveis escolhas dado pela função de transição da máquina N. A cada nodo na árvore, associa-se um endereço sobre o alfabeto $\Sigma_b = \{1, 2, \ldots, b\}$. Associa-se o endereço 231 ao nodo ao qual chegamos iniciando na raiz, para seu segundo filho, indo para o terceiro filho deste nodo, e finalmente para o primeiro filho desse nodo. Cada símbolo define que escolha fazer a seguir quando similamos um passo em um ramo da computação não-determinística de N. A fita 3 contém uma cadeia sobre o alfabeto Σ_b. Ela representa o ramo da computação de N da raiz para o nó endereçado por essa cadeia. Uma cadeia vazia represeta o endereço da raiz.

Deste modo, pode-se descrever os estágios de D:

1. Inicialmente, a fita 1 contém a palavra de entrada w e as fitas 2 e 3 estão vazias;
2. Copie a fita 1 para a fita 2;

3. Use a fita 2 para simular N com a entrada w sobre o ramo de sua computação não-determinística. Antes de cada passo de N, consulte o próximo símbolo na fita 3 para determinar qual escolha fazer entre aquelas permitidas pela função de transição de N. Se não restam mais símbolos na fita 3, ou se essa escolha não-determinística não for válida, aborte esse ramo indo para o estágio 4. Também vá para o estágio 4 se uma configuração de rejeição for encontrada. Se uma configuração de aceitação for encontrada, aceite a palavra;

4. Substitua a cadeia na fita 3 pela próxima cadeia na ordem lexicográfica. Simule o próximo ramo da computação de N indo para o estágio 2.

Para concluir, qualquer máquina de Turing determinística é automaticamente uma máquina de Turing não-determinística.
■

D.6 Enumeradores

Informalmente, um enumerador é uma máquina de Turing com uma impressora em anexo para imprimir as cadeias. Toda a vez que a máquina de Turing quiser adicionar uma cadeia a lista, ela a envia para a impressora.

Logo, uma linguagem é Turing-reconhecível se e somente se algum enumerador a enumera.

D.7 Algoritmos e a Máquina de Turing

Em 1900, David Hilbert proferiu uma palestra no Congresso Internacional de Matemática em Paris. Nesta apresentação, ele enumerou 23 problemas matemáticos e os colocou como um desafio para o século XX. O décimo problema era sobre algoritmos.

O décimo problema de Hilbert era conceber um algoritmo que testasse se um polinômio possui uma raiz inteira. Até então não se tinha o conceito de algoritmo, então a ideia dele era identificar *um processo que determine a raiz com um número finito de operações*.

Hoje, já se sabe que não existe algoritmo para tal tarefa. Na época, isto era impossível de se conceber, pois não existia uma concepção do que era algoritmo e quais suas limitações.

A definição chegou nos artigos de Alonzo Church e Alan Turing em 1936. Church usou um sistema denominado $\lambda-$cálculo e Turing usou suas máquinas (máquina de Turing). Estas duas definições se demonstraram equivalentes. A noção precisa de algoritmo veio a ser chamada de **Tese de Church-Turing**. Deste modo, a noção intuitiva de algoritmos é igual a algoritmos de máquinas de Turing.

Em 1970, a tese de Church-Turing proveu a definição necessária para que Yuri Matijasevic (inspirado no trabalho de Martin Davis, Hilary Putnam, e Julia Robinson) demonstrasse que o décimo problema de Hilbert não existe algoritmo para se testar se um polinômio tem raizes inteiras.

D.7.1 O Décimo Problema e a Máquina de Turing

Formalizando o décimo problema de Hilbert na terminologia que utilizamos, tranformando-o na linguagem $D = \{p | p \text{ é um polinômio com uma raiz inteira}\}$.

Basicamente, o décimo problema de Hilbert pergunta se a linguagem D é decidível. A reposta é negativa.

Iremos demonstrar que D é Turing-reconhecível. Para isto, vamos utilizar um problema mais simples, mas análogo a D. Nesta linguagem $D_1 = \{p | p$ é um polinômio sobre x com uma raiz inteira $\}$.

A máquina de Turing que reconhece D_1 é M_1: "A entra p é um polinômio sobre a variável x. Calcular o valor de p com x sendo substituída sucessivamente pelos valores 0, 1, -1, 2, -2, ... Se em algum ponto o valor de p resultar em 0, aceite".

Se p tem uma raiz inteira, M_1 vai parar e aceitar em algum momento. Se não, vai rodar para sempre. Para o caso de múltiplas variáveis do problema original D, pode-se utilizar uma máquina de Turing M semelhante a M_1. M apenas teria que passar por cada uma das variáveis que se deseja encontrar uma raiz inteira.

Diz-se que M_1 e M são reconhecedores, mas não decisores. O teorema de Matijasevic mostra que não há um decisor para D e D_1.

D.8 Decidibilidade

Linguagens decidíveis são Turing-decidíveis.

A linguagem $L_{AFD} = \{< B, w > | B$ é um AFD que aceita $w\}$ é decidível? Uma forma de provar isso é demonstrar que uma máquina de Turing pára sobre todas as palavras de L_{AFD}.

Teorema: A linguagem L_{AFD} é decidível.

Prova: Considere os seguintes estágios para uma máquina de Turing M. Sobre a entrada $< B, w >$, onde B é um AFD e w é uma cadeia:

1. Simule B sobre a cadeia w;

2. Se a simulação terminar em um estado de aceitação, aceite. Se ela termina em um estado de rejeição, rejeite.

Uma representação de B deve conter cinco componentes: Q, Σ, δ, q_0, e F. A máquina M mantém o registro do estado atual de B. Quando M recebe sua entrada $< B, w >$, ela primeiro determina se a AFD B está bem formada e se w é uma cadeia. Caso contrário, rejeita.

Então M realiza a simulação diretamente. Ela mantém o registro do estado atual de B e da posição atual na entrada w escrevendo essa informação na fita. Inicialmente, o estado de B é q_0 e o primeiro símbolo a ser lido é o primeiro símbolo de w. Os estados e a posição da fita são atualizados de acordo com a função de transição de B (função δ). Quando terminar de ler w, M aceita caso B esteja num estado de aceitação. Qualquer outra computação gera uma rejeição por parte de M.

Ao simular um AFD, realiza-se finitos passos à direita. Como w possui um número finito de símbolos, são necessárias finitas computações para chegar a uma aceitação ou uma rejeição, mesmo que seja necessário ler todos os símbolos de w para isto. ■

D.8.1 Uma Linguagem Indecidível

Teorema D.8.1. *A linguagem $L_{MT} = \{< M, w > | M$ é uma máquina de Turing e M aceita $w\}$ é indecidível.*

Prova: Supõe-se que L_{MT} é decidível e obtermos uma contradição. Suponha que H seja um decisor para a linguagem L_{MT}. A entrada de H é $< M, w >$, onde M é uma máquina de Turing e w é uma palavra. H pára e aceita se M pára e aceita w, e H rejeita se M rejeita w.

Considere uma nova máquina de Turing D que receba a entrada M, onde M é uma máquina de Turing. D aceita se M rejeita $< M >$ como palavra de entrada, e D rejeita se M aceita $< M >$ como palavra de entrada. Claramente, H poderia ser utilizado como subrotina de D. Executando $H(< M, < M >>)$, rejeitar se H aceite, e aceitar caso H rejeite.

É importante lembrar que $< D, < D >>$ pode ser uma entrada para a linguagem L_{MT}, então H deve decidir sobre $< D, < D >>$.

O que acontece quando D executa sobre a entrada de D? D é forçada a fazer o oposto de si própria o que é uma contradição. ■

D.8.2 Algumas Linguagens são Turing-irreconhecíveis

Teorema D.8.2. *Algumas linguagens não são Turing-reconhecíveis.*

Prova: Para mostrar que o conjunto de todas as máquinas de Turing é contável, primeiro observamos que o conjunto de todas as cadeias Σ^* é contável para qualquer alfabeto Σ. Com apenas uma quanidade finita de cadeias de cada comprimento, podemos formar uma lista de Σ^* listando todas as cadeias de comprimento 0, 1, 2, e assim por diante.

O conjunto de todas as máquinas de Turing é contável porque cada máquina de Turing M pode ser serializada em uma cadeia $< M >$. Se simplesmente omitirmos as cadeias que não representam uma legítima máquina de Turing, podemos obter uma lista de todas as máquinas de Turing.

Para mostrar que o conjunto de todas as linguagens é incontável, primeiro observamos que o conjunto de todas as sequências binárias infinitas é incontável. Uma sequência binária infinita é formada por uma sequência interminável 0s e 1s. Seja $\mathbb{B}$ o conjunto de todas as sequências binárias infinitas é incontável, por diagonalização [1].

Seja $\mathbb{L}$ o conjunto de todas as linguagens sobre o alfabeto Σ. Mostramos que $\mathbb{L}$ é incontável dando uma correspondência com $\mathbb{B}$, considerando que ambos os conjuntos são de mesmo tamanho. Seja $\Sigma^* = \{s_1, s_2, \ldots\}$. Cada linguagem $A \in \mathbb{L}$ tem uma sequência única em $\mathbb{B}$. O $i-$ésimo bit da sequência é 1 se $s_i \in A$.

A função $f : \mathbb{L} \to \mathbb{B}$, é um-para-um, sobrejetora[2], deste modo uma correspondência. Consequentemente $\mathbb{L}$ é incontável, assim como $\mathbb{B}$.

Deste modo, o conjunto de todas as linguagens não pode ter correspondência com o conjunto de todas as máquinas de Turing. Conclui-se que algumas linguagens não são reconhecidas por uma máquina de Turing. ■

D.8.3 Uma Linguagem Turing-irreconhecível

De acordo com o teorema a seguir, se uma linguagem L e seu complemento[3] $\overline{L}$ são Turing-reconhecíveis, então L é decidível. Logo, para qualquer linguagem indecidível, ou ela ou seu complemento não é Turing-reconhecível. Diz-se que uma linguagem é **co-Turing-reconhecível** se ela for o complemento de uma linguagem Turing-reconhecível.

Teorema D.8.3. *Uma linguagem é decidível se e somente se ela é Turing-reconhecível e co-Turing-reconhecível (Uma linguagem é decidível exatamente quando ela e seu complemento são ambas Turing-reconhecíveis).*

[1] Método da Diagonalização foi descoberto por Georg Cantor em 1873 quando o mesmo se preocupava em comparar conjuntos infinitos. Veja página 183 do livro de Michael Sipser.

[2] Uma função $g : X \to Y$ é sobrejetora se $\forall y \in Y, \exists x \in X(y = g(x))$

[3] O complemento de uma linguagem L é uma linguagem $\overline{L}$ constituída de todas as cadeias que não pertencem a L

Prova: Deve-se provar em dias direções. Primeiro, se a linguagem A for decidível, pode-se facilmente ver que tanto A quanto seu complemento $\overline{A}$ são Turing-reconhecíveis. Qualquer linguagem decidível é Turing-reconhecível, e o complemento de uma linguagem decidível também é decidível.

Para a outra direção, se tanto A e $\overline{A}$ são Turing-reconhecíveis, fazemos M_1 ser o reconhecedor para A e M_2 o reconhecedor para $\overline{A}$. M_2 pode ser uma máquina de Turing que executa M_1 sobre w. Se M_1 rejeita, M_2 aceita, se M_1 aceita, M_2 rejeita.

Toda cadeia w ou está em A ou em $\overline{A}$. Consequentemente, ou M_1 ou M_2 aceita w. Uma vez que as duas páram sobre w, então M_1 e M_2 são decisores para A e $\overline{A}$ respectivamente. Neste caso, A é decidível. ■

$\overline{L_{MT}}$ não é Turing-reconhecível.

Teorema D.8.4. *$\overline{L_{MT}}$ não é Turing-reconhecível.*

Prova: Sabe-se que L_{MT} é Turing-reconhecível. Se $\overline{L_{MT}}$ fosse Turing-reconhecível, então L_{MT} seria decidível. ■

D.9 Redutibilidade

Uma redução é uma meneira de converter um problema em outro de forma que uma solução para o segundo problema possa ser utilizada para resolver o primeiro. Sejam A e B dois problemas, se A reduz B, pode-se usar uma solução de B para resolver A.

Geralmente, são utilizadas para demonstrar que problemas não computacionalmente insolúveis.

D.9.1 Um Problema Indecidível e a Redutibilidade

Teorema D.9.1. *A linguagem $PARA_{MT} = \{< M, w > | M$ é uma máquina de Turing e M pára sobre $w\}$ é Turing-indecidível.*

Prova: Por contradição, se supõe que a máquina de Turing R decida $PARA_{MT}$. Então, uma máquina de Turing S é construída para decidir L_{MT}, com S operando da seguinte forma sobre a entrada $< M, w >$:

1. Rode a máquina de Turing R sobre a entrada $< M, w >$;
2. Se R rejeita, então rejeite;
3. Se R aceita, então simule M sobre a entrada w até que ela páre;
4. Se M aceitar w, aceite; se rejeitar, rejeite.

Se R decide $PARA_{MT}$, então S decide L_{MT}. Como L_{MT} é indecidível, $PARA_{MT}$ também deve ser. ■

D.9.2 Reduções via Histórias de Computação

O método da história de computação é uma técnica para provar que L_{MT} é redutível a certas linguagens. Ela é o registro completo da computação de uma máquina de Turing.

Seja M uma máquina de Turing e w uma cadeia de entrada. Uma *história de computação de aceitação* para M sobre w é uma sequência de configurações C_1, C_2, ..., C_l, onde C_1 é a configuração inicial da máquina de Turing M sobre w, C_l pe uma configuração de aceitação de M e cada C_i é uma configuração legítima na sequência de C_{i-1}. Uma *história de computação de rejeição* para M sobre w é semelhante, mas C_l representa uma configuração de rejeição.

D.9.2.1 Um Problema Decidível

Lema D.9.2. *Seja M uma máquina de Turing com fita limitada com q estados e g símbolos no alfabeto da fita (Γ), existem exatamente qng^n configurações distintas de M para uma fita de comprimento n.*

Prova: Uma configuração de M é constituída do estado do controle, da posição da cabeça na fita e do conteúdo da fita. A cabeça só pode estar em uma das n posições da fita. Há g^n cadeias possíveis de símbolos na fita. Considerando que o controle pode estar em apenas q estados, o número de configurações diferentes é qng^n. ■

Teorema D.9.3. *A linguagem $L_{MTFL} = \{< M, w > | M$ é uma máquina de Turing com fita limitada que aceita a cadeia $w\}$ é decidível.*

Para determinar se M aceita sobre w, simula-se M sobre w. Durante a simulação, se M pára e aceita ou rejeita, aceita-se ou rejeita-se de acordo com M. O que é difícil de determinar é se a máquina M entra numa computação infinita durante esta simulação. Então, precisa-se detectar se a máquina entra nesta computação indefinida para que possa-se rejeitá-la sem a necessidade de simulá-la.

Para detectar se M entrará em uma computação infinita, durante a simulação, ela vai de configuração a configuração. Se em algum momento M repetir alguma configuração, M estará em uma computação infinita. Devido a fita ser limitada, o número de configurações é finito (verificar **Lema** acima), então consegue-se determinar esta repetição de configuração com uma quantidade finita de tempo.
Prova: A máquina de Turing que decide L_{MTFL} recebe a entrada $< M, w >$, onde M é uma máquina de Turing com fita limitada e w uma cadeia e realiza as seguintes operações:

1. Simular M sobre w por qng^n passos ou até que ela páre;

2. Se M parou, aceite se ela aceitou ou rejeite se ela rejeitou. Se ela não parou, rejeite.

Se M sobre w não parar em qng^n passos, significa que M está em uma computação infinita, logo, rejeita-se a entrada $< M, w >$. Deste modo, L_{MTFL} é decidível mesmo que M sobre w gere uma computação infinita. ■

D.9.2.2 Um Problema Indecidível

Teorema D.9.4. *A linguagem $V_{MTFL} = \{< M > | M$ é uma máquina de Turing com fita limitada e $L(M) = \emptyset\}$ é Turing-indecidível.*

Para provar por redução a L_{MT}, mostramos que se V_{MTFL} fosse decidível, A_{MT} também seria. Suponha que V_{MTFL} seja decidível. Para uma máquina de Turing M e uma entrada w, pode-se determinar se M aceita w construindo uma máquina de Turing com fita limitada B e então testa-se se $L(B) = \emptyset$. A linguagem que B reconhece compreende todas as *histórias de computação de aceitação* para M sobre w. Se M aceita w, a linguagem de B possui uma cadeia e portanto não é vazia; caso contrário M não aceita w (a linguagem de B é vazia).

Constrói-se a máquina de Turing com fita limitada B para aceitar a entrada x se x representar uma *história de computação de aceitação* para M sobre w. As configurações $C_1, C_2, \ldots, C_l$ são colocadas numa fita separadas pelo caracter #.

A máquina B recebe a entrada x e quebra esta entrada conforme a posição dos caracteres #. Então B determina se cada configuração C_i respeita três condições de uma *história de computação de aceitação*:

1. C_1 é a configuração inicial de M sobre w;
2. C_{i+1} segue a legitimamente a configuração C_i;
3. C_l é uma configuração de aceitação para M.

Prova: Suponha que exista uma máquina de Turing R que decida V_{MTFL}. Então, constrói-se uma máquina de Turing S que decide L_{MT} sobre a entrada $< M, w >$, onde M é uma máquina de Turing e w uma cadeia:

1. Construir a máquina de Turing com fita limitada B a partir de M e w, conforme descrito anteriormente;
2. Rode R sobre a entrada $< B >$;
3. Se R rejeitar, então S aceita; se R aceitar, então S rejeita.

Se R aceita $< B >$, então $L(< B >) = \emptyset$. Consequentemente, não existiria uma *história de computação de aceitação* de M sobre w, então S rejeitaria. Caso contrário, S aceitaria. Deste modo, podemos decidir L_{MT} utilizando um decisor de V_{MTFL}. Como L_{MT} é indecidível, então V_{MTFL} também é. ■

D.10 Complexidade de Tempo

Seja M uma máquina de Turing que pára sobre todas as entradas, o tempo de execução ou complexidade de tempo de M é a função $f : \mathbb{N} \to \mathbb{N}$, onde $f(n)$ é o número máximo de passos que M usa sobre entradas de comprimento n. Se $f(n)$ for o tempo de execução de M, diz-se que M roda em tempo $f(n)$ e que M é uma máquina de Turing de tempo $f(n)$.

Em uma *análise do pior caso*, considera-se o tempo mais longo de execução para todas as entradas de um tamanho específico. Em uma *análise de melhor caso*, considera-se o tempo mais curto de executção para todas as entradas de um tamanho específico.

D.10.1 Classes de Complexidade

Definição 13. *Seja $t : \mathbb{N} \to \mathbb{R}^+$ uma função, define-se a classe de complexidade de tempo $TIME(t(n))$, como a coleção de todas as linguagens que são decidíveis por uma máquina de Turing determinística em tempo $O(t(n))$. $TIME = \{L | L$ é uma linguagem decidida por uma máquina de Turing determinística de tempo $O(t(n))\}$.*

Definição 14. *Seja $t : \mathbb{N} \rightarrow \mathbb{R}^+$ uma função, define-se a classe de complexidade de tempo $NTIME(t(n))$, como a coleção de todas as linguagens que são decidíveis por uma máquina de Turing não-determinística em tempo $O(t(n))$. $NTIME = \{L|L$ é uma linguagem decidida por uma máquina de Turing não-determinística de tempo $O(t(n))\}$.*

Definição 15. *Seja N uma máquina de Turing não-determinística decisora, seu tempo de execução é a função $f : \mathbb{N} \rightarrow \mathbb{N}$, onde $f(n)$ representa a profundidade da árvore de computações determinísticas e não a quantidade de nodos nesta árvore.*

D.10.1.1 A Classe P

P ou $PTIME$ é a classe constituída de problemas resolvidos em tempo determinístico polinomial.

Definição 16. *P ou $PTIME$ é a classe constituída de todas as ilnguagens decidíveis por uma máquina de Turing determinística em tempo polinomial. Em outras palavras*

$$P = \bigcup_k TIME(n^k), \tag{D.1}$$

onde k são valores constantes.

D.10.1.2 A Classe NP

NP ou $NPTIME$ é a classe de problemas resolvidos em tempo não-determinístico polinomial. Também pode ser definida como a classe de todos os problemas verificáveis em tempo polinomial determinístico.

Definição 17. *Um verificador para uma linguagem A é um algoritmo V, onde $A = \{w|V$ aceita $< w, c >$ para alguma cadeia $c\}$. Mede-se o tempo de um verificador em termos apenas de comprimento de w, portanto um verificador de tempo polinomial executa a verificação em tempo polinomial no comprimento de w. Uma linguagem A é polinomialmente verificável se ela tem um verificador de tempo polinomial.*

Definição 18. *NP e $NPTIME$ é a classe de linguagens que tem verificadores de tempo polinomial.*

Definição 19. *$NP = \bigcup_k NTIME(n^k)$, onde k são valores constantes.*

D.10.1.3 P versus NP

A questão "$P = NP$?" é uma grande questão aberta. Ela basicamente poderia ser lida como "Todos os problemas verificados em tempo polinomial determinístico podem ser resolvidos em tempo polinomial determinístico?". Sabe-se que

$$NP \subseteq EXPTIME = \bigcup_k TIME(n^k), \tag{D.2}$$

k é uma constante.

D.10.1.4 NP-Completude

Um avanço importante na questão "P versus NP" surgiu nos anos iniciais da década de 70 com os trabalhos de Stephen Cook, Leonid Levin e Richard Karp. Stephen Cook e Leonid Levin descobriram que certos problemas em NP cuja a complexidade individual está relacionada a toda a classe. Se existir um algoritmo polinomial para um destes problemas, então todas os demais problemas da classe NP seriam decididos em tempo polinomial. Estes problemas são definidos como NP-Completos.

Um dos primeiros problemas NP-Completo (descoberto por Stephen Cook) foi o *problema da satisfazibilidade* $SAT = \{< \phi > | \phi$ é uma fórmula booleana satisfazível $\}$. ϕ é uma fórmula booleana composta por variáveis booleanas usando as operações do **e** ($\wedge$), **ou** ($\vee$), e **não** ($\neg$) lógicos. O Teorema de Cook-Levin relaciona a complexidade de SAT às complexidades de todos os problemas em NP.

Teorema D.10.1. *$SAT \in P$ se e somente se $P = NP$.*

Redutibilidade em Tempo Polinomial

Redutibilidade em tempo polinomial é o método central do teorema de Cook-Levin. Quando um problema A é eficientemente redutível ao problema B, uma solução eficiente para B pode ser utilizada para resolver A eficientemente.

Definição 20. *Uma função $f : \Sigma^* \to \Sigma^*$ é uma função computável em tempo polinomial se existe alguma máquina de Turing de tempo polinomial M que pára com exatamente $f(w)$ na sua fita, quando iniciada com a entrada de w.*

Definição 21. *A linguagem A é redutível por mapeamento em tempo polinomial, ou simplesmente redutível em tempo polinomial, ou ainda redutível em tempo polinomial de muitos-para-um, à linguagem B em símbolos $A \leq_P B$, se existe uma função computável em tempo polinomial $f : \Sigma^* \to \Sigma^*$, onde para toda w, $w \in A \iff f(w) \in B$. A função f é denominada redução de tempo polinomial de A para B.*

Definição 22. *Uma linguagem B é NP-Completa se satisfaz duas condições:*

1. *$B \in NP$;*
2. *toda $A \in NP$ é redutível em tempo polinomial a B.*

Teorema: Se B for NP-Completa e $B \leq_P C$ para $C \in NP$, então C é NP-Completa.
Prova: Prove ...

Teorema: SAT é NP-Completo.

Para provar que SAT é NP-Completo, deve-se primeiro demonstrar que ele está em NP. Em seguida, deve-se mostrar que todas as linguagens em NP são redutíveis polinomialmente a SAT. Para este último, deve-se construir uma redução polinomial para cada linguagem A em NP para SAT. A redução para A recebe uma cadeia w e produz uma fórmula booleana ϕ que simula a máquina NP para A sobre a entrada w. Se a máquina aceita, ϕ é satisfazível.

Prova: Primeiro, deve-se mostrar que SAT está em NP. Uma máquina de tempo polinomial não-determinístico pode encontrar se existe uma atribuição às variáveis binárias é

satisfazível. Uma árvore de tentativas não-determinísticas possui profundidade igual ao número de variáveis binárias. Portanto, $SAT \in NP$.

A outra parte da prova deve demonstrar que toda linguagem $A \in NP$ é redutível em tempo polinomial a SAT. Seja N uma máquina de Turing não-determinística que decide A em tempo n^k para alguma constante k.

Um *tableau* para N sobre w é uma tabela $\{0,1\}^{n^k \times n^k}$ cujas linhas são as configurações de um ramo da computação de N sobre a entrada w. Na primeira e na última célula de cada linha, assume-se que há o símbolo #. Na primeira linha, há a configuração inicial (estado inicial + w). Um *tableau* é de aceitação se qualquer linha dele for uma configuração de aceitação. O problema de determinar se N aceita w é equivalente ao problema de se determinar se existe um *tableau* de aceitação para N sobre w.

Com isto, pode-se introduzir a descrição da redução polinomial f de A para SAT. Sobre a entrada w, a redução produz uma fórmula ϕ. As variáveis de ϕ representam cada uma das $(n^k)^2$ células do *tableau*. Define-se $x_{i,j,s}$ uma variável binária de ϕ, onde i e j representam a coordenada da célula no *tableau* e $s \in C = Q \cup \Gamma \cup \{\#\}$ o símbolo ou o estado. Se $x_{i,j,s}$ é 0, então o símbolo $s \in C$ está presente na célula de coordenada (i,j) no *tableau*.

Deve-se projetar um ϕ de modo que se houver uma atribuição às variáveis que satisfaça ϕ, então N aceita w. A formula se divide em quatro partes $\phi = \phi_{célula} \wedge \phi_{início} \wedge \phi_{movimento} \wedge \phi_{aceita}$.

A parte $\phi_{célula}$ garante que cada célula tenha um símbolo e apenas uma célula tenha um símbolo. Logo, tem-se a definição de

$$\phi_{célula} = \bigwedge_{1 \le i,j \le n^k} \left[\left(\bigvee_{s \in C} x_{i,j,s} \right) \bigwedge \left(\bigwedge_{t,u \in C, u \neq t} (\neg x_{i,j,u} \vee \neg x_{i,j,t}) \right) \right]. \tag{D.3}$$

A parte $\phi_{início}$ garante que a primeira linha seja uma configuração inicial de N sobre w. Deste modo,

$$\phi_{início} = x_{1,1,\#} \wedge x_{1,2,q_0} \wedge x_{1,3,w_1} \wedge x_{1,4,w_2} \wedge \ldots \wedge x_{1,n^k-1,\sqcup} \wedge x_{1,n^k,\#}. \tag{D.4}$$

Para representar a aceitação de N, define-se a parte de aceitação

$$\phi_{aceita} = \bigvee_{1 \le i,j \le n^k} x_{i,j,q_{aceita}}. \tag{D.5}$$

$\phi_{movimento}$ garante cada linha correspondente a uma configuração siga legalmente uma configuração da linha precedente conforme definido nas transições de N. Para isto, define-se uma sequência de trechos para ϕ configurando as possíveis janelas legais que representam as transições possíveis de N. Uma janela legal é uma matriz 2×3 dentro do *tableau*. Por exemplo, a Janela Legal 2 define demonstram uma janela legal para a transição $\delta(q_1, a) = \{(q_2, b, D)\}$.

a	q_1	a
a	b	q_2

Tabela 2 – Um exemplo de janela legal para a transição $\delta(q_1, a) = \{(q_2, b, D)\}$.

Logo, $\phi_{movimento} = \bigwedge_{1 \le i \le n^k, 1 \le j \le n^k} \Big($ a janela (i,j) é legal$\Big)$. Substitui-se "a janela legal" por

$$\bigvee_{w \in W(i,j)} \Big(x_{i,j-1,a_1} \wedge x_{i,j,a_2} \wedge x_{i,j+1,a_3} \wedge x_{i+1,j-1,a_4} \wedge x_{i+1,j,a_5} \wedge x_{i+1,j+1,a_6} \Big), \tag{D.6}$$

onde $W(i,j)$ é o conjunto de todas as janelas legais para a coordenada (i,j), a_1, a_2, a_3, a_4, a_5, e a_6 são os símbolos de uma janela legal.

A seguir demonstra-se que a redução demanda tempo polinomial determinística por relacionar o tamanho de ϕ. O número de variáveis de ϕ é $n^{2k}|C|$, ou seja, o tamanho do *tableau* vezes o número de símbolos. Como $|C|$ depende apenas de N e não de $n = |w|$, então diz-se que o total de variáveis é $O(n^{2k})$.

Em $\phi_{célula}$, há um fragmento de tamanho fixo para cada variável, portanto demanda tempo $O(n^{2k})$. Para formular a parte $\phi_{início}$ é necessário uma construção de tamanho igual a quantidade de células da primeira linha do *tableau*, portanto demanda tempo $O(n^k)$. As fórmulas ϕ_{aceita} e $\phi_{movimento}$ possuem um fragmento fixo para cada célula do *tableau*, então o tempo computacional é $O(n^{2k})$. A formulação completa de ϕ é um polinômio sobre a variável n (tamanho do problema) então a redução é polinomial.

Deste modo, conclui-se que $SAT \in NP$-Completo. ■

Definição 23. *Uma linguagem B é NP-Difícil se toda $A \in NP$ é redutível em tempo polinomial a B.*

Com a definição acima, podemos assumir que NP-Completo $\subseteq$ NP-Difícil. Mesmo que $P = NP$, há problemas em NP-Difícil que não seriam resolvidos em tempo polinomial determinístico.

Mantenha em mente que:

- todo problema que "resolve" toda uma classe e pertence a mesma é dito COMPLETO;
- todo problema que "resolve" toda uma classe é dito DIFÍCIL.

Logo, todo problema COMPLETO é DIFÍCIL, mas não o contrário.

D.11 Complexidade de Espaço

Definição 24. *Seja M uma máquina de Turing determinística que pára sobre todas as entradas. A complexidade de espaço de M é a função $f : \mathbb{N} \to \mathbb{N}$, onde $f(n)$ é o número máximo de cédulas de fita que M visita sobre qualquer entrada de comprimento n. Se a complexidade de espaço de M é $f(n)$, também diz-se que M roda em espaço $f(n)$.*

Definição 25. *Seja M uma máquina de Turing não-determinística na qual todos os ramos páram sobre todas as entradas, definimos sua complexidade de espaço $f(n)$ (onde $f : \mathbb{N} \to \mathbb{N}$) como o número máximo de cédulas de fita que M visita sobre qualquer ramo de sua computação para qualquer entrada de comprimento n.*

D.11.1 Classes de Complexidade

Definição 26. *Seja $f : \mathbb{N} \to \mathbb{R}^+$ uma função. As classes de complexidade de espaço, $SPACE(f(n))$ e $NSPACE(f(n))$ são definidas da seguinte forma:*

- $SPACE(f(n)) = \{L | L$ *é uma linguagem decidida por uma máquina de Turing determinística de espaço* $O(f(n))\}$;
- $NSPACE(f(n)) = \{L | L$ *é uma linguagem decidida por uma máquina de Turing não-determinística de espaço* $O(f(n))\}$.

Definição 27. *$PSPACE$ é a classe de linguagens decidíveis em espaço polinomial em uma máquina de Turing determinística, ou seja,*

$$PSPACE = \bigcup_k SPACE(n^k), \tag{D.7}$$

onde k é uma constante.

Definição 28. *$NPSPACE$ é a classe de linguagens decidíveis em espaço polinomial em uma máquina de Turing não-determinística, ou seja,*

$$NPSPACE = \bigcup_k NSPACE(n^k), \tag{D.8}$$

onde k é uma constante.

Definição 29. *Uma linguagem B é $PSPACE - Completa$ se ela satisfaz duas condições:*

1. *B está em $PSPACE$;*
2. *toda $A \in PSPACE$ é redutível em tempo polinomial a B.*

Definição 30. *Uma linguagem B é $PSPACE - Difícil$ se $A \in PSPACE$ é redutível em tempo polinomial a B.*

Definição 31. *L é a classe das linguagens que são decidíveis por uma máquina de Turing determinística em espaço logarítmico, ou seja,*

$$L = SPACE(\log_n). \tag{D.9}$$

Definição 32. *NL é a classe das linguagens que são decidíveis por uma máquina de Turing não-determinística em espaço logarítmico, ou seja,*

$$NL = NSPACE(\log_n). \tag{D.10}$$

D.11.1.1 Teorema de Savitch

Teorema: Para qualquer função $f : \mathbb{N} \to \mathbb{R}^+$, onde $f(n) \geq n$, $NSPACE(f(n)) \subseteq SPACE(f(n)^2)$.
Prova: ■