3. **Ariane 5**: Um foguete que custou aproximadamente 7 bilhões de dólares para ser construído explodiu no seu primeiro voo em 1996 devido ao reuso sem verificação apropriada de partes do código do seu predecessor.

Já deixamos claro que vamos **provar** muita coisa aqui. Mas o que é uma prova? Uma resposta possível "é um argumento feito para convencer alguém"[24]. O problema deste argumento é que pessoas diferentes podem ter compreensões distintas sobre o argumento, de forma que o argumento pode ser uma prova para uma pessoa, mas não para a outra... estranho, não? Uma definição geral e abstrata para a noção de prova não é uma tarefa fácil, mas forneceremos uma definição precisa em um contexto mais restrito, a saber, o da lógica simbólica[11, 26].

Capítulo 2

Heapsort

Exercício:

Ordene os n elementos de um vetor $A[1..n]$ da seguinte forma: encontre o maior elemento de A e troque este elemento com o elemento $A[n]$. Em seguida, encontre o maior elemento de $A[1..n-1]$ e troque este elemento com $A[n-1]$, e assim por diante até que o vetor A esteja ordenado.

Escreva o pseudocódigo do seu algoritmo.

Solução força-bruta:

```
1 for i = n downto 2 do
2 |   max ← i;
3 |   for j = i − 1 downto 1 do
4 |   |   if A[j] > A[max] then
5 |   |   |   max ← j;
6 |   |   end
7 |   end
8 |   swap A[i] and A[max];
9 end
```

Algoritmo 1: selection-sort(A)

1. Análise assintótica:
$$T(n) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} 1 = \sum_{i=1}^{n-1} (n-i) = \sum_{i=1}^{n-1} i = \frac{(n-1).n}{2} = \Theta(n^2).$$

2. A correção pode ser provada a partir da seguinte invariante:

   Antes de cada iteração do laço **for** indexado por i (linhas 1-8), o subvetor $A[i+1..n]$ está ordenado e contém os $(n-i)$ maiores elementos do vetor A.

O algoritmo Heapsort

Estudaremos um novo algoritmo de ordenação baseado em comparação de chaves, mas bem diferente dos algoritmos (*insertion sort* e *mergesort*) vistos anteriormente. Este novo algoritmo, conhecido como *heapsort*, possui tempo de execução $O(n.\log n)$, como *mergesort*, e o processo de ordenação é feito *in place* (como em *insertion sort*). Portanto *heapsort* combina as vantagens de *insertion sort* e *mergesort*. A estrutura de dados utilizada por este algoritmo é conhecida como *heap*:

DEF. Um *heap* (binário) T é uma estrutura de dados que corresponde a uma árvore binária com chaves associadas aos nós, sendo uma chave por nó, que satisfaz às seguintes condições:

1. T é uma árvore binária completa em todos os níveis, exceto possivelmente o último nível;

2. Todos os caminhos para uma folha do último nível estão à esquerda de todos os caminhos para uma folha do penúltimo nível;

3. A chave de cada nó é maior ou igual do que a chave dos seus filhos.

Os itens 1 e 2 da definição acima caracterizam a chamada de **propriedade do corpo do /heap**. O item 3 corresponde a **propriedade de heap** (de máximo).

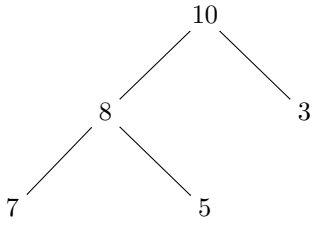
Segundo a propriedade 2, uma enumeração dos nós de um *heap* deve começar de cima para baixo, *i.e.* a partir da raiz do *heap*, e da esquerda para a direita.

Assim, em um *heap* o nó mais à direita pode ter apenas um filho à esquerda, mas não pode ter somente um filho à direita.

Todos os outros nós internos possuem dois filhos.

1. Exemplo 1

   A figura abaixo é um *heap* de máximo:



   DEF. Um *heap* (binário) T é uma estrutura de dados que corresponde a uma árvore binária com chaves associadas aos nós, sendo uma chave por nó, que satisfaz às seguintes condições:

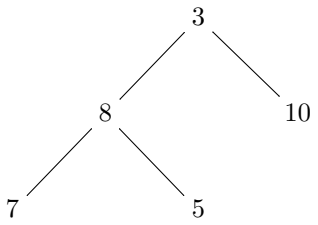
   (a) T é uma árvore binária completa em todos os níveis, exceto possivelmente o último nível;

   (b) Todos os caminhos para uma folha do último nível estão à esquerda de todos os caminhos para uma folha do penúltimo nível;

   (c) A chave de cada nó é maior ou igual do que a chave dos seus filhos.

2. Exemplo 2

   A figura abaixo não é um *heap de máximo* porque não satisfaz a propriedade 3:



DEF. Um *heap* (binário) T é uma estrutura de dados que corresponde a uma árvore binária com chaves associadas aos nós, sendo uma chave por nó, que satisfaz às seguintes condições:

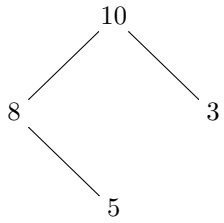
(a) T é uma árvore binária completa em todos os níveis, exceto possivelmente o último nível;

(b) Todos os caminhos para uma folha do último nível estão à esquerda de todos os caminhos para uma folha do penúltimo nível;

(c) A chave de cada nó é maior ou igual do que a chave dos seus filhos.

3. Exemplo 3

A figura abaixo não é um *heap* porque não satisfaz a propriedade 1:



DEF. Um *heap* (binário) T é uma estrutura de dados que corresponde a uma árvore binária com chaves associadas aos nós, sendo uma chave por nó, que satisfaz às seguintes condições:

(a) T é uma árvore binária completa em todos os níveis, exceto possivelmente o último nível;

(b) Todos os caminhos para uma folha do último nível estão à esquerda de todos os caminhos para uma folha do penúltimo nível;

(c) A chave de cada nó é maior ou igual do que a chave dos seus filhos.

4. Propriedades

A grande vantagem da estrutura de *heap* é que ela permite a implementação das operações de inserção de um novo elemento (ou uma nova chave), e extração do maior elemento (maior chave) em tempo logarítmico.

Note que em um vetor (ou em uma lista), a inserção pode ser feita em tempo constante, mas a extração do maior elemento vai exigir, no pior caso, uma busca em todo o vetor (ou lista), o que tem custo linear.

A estrutura de *heap* suporta simultaneamente as duas operações, e como veremos, de forma assintoticamente mais eficiente do que em listas ou vetores.

(a) Implementação em vetores

Um *heap* binário pode ser implementado como um subvetor de um vetor A, onde somente os elementos em $A[1..A.\text{heap-size}]$ $(0 \leq A.\text{heap-size} \leq A.length)$ são elementos válidos do *heap*. A raiz do *heap* é $A[1]$, e dado o índice i de um nó, o índice do filho à esquerda (resp. direita) é $2i$ (resp. $2i+1$), enquanto que o índice do nó correspondente ao pai do nó de índice i é igual a $\lfloor i/2 \rfloor$.

Alguns autores chamam a estrutura definida acima de *heap* de máximo (ou *max-heap*), isto é, um *heap* onde todo nó i diferente da raiz é tal que $A[\lfloor i/2 \rfloor] \geq A[i]$. Analogamente, podemos definir um *heap* de mínimo (ou *min-heap*) considerando que todo nó i diferente da raiz é tal que $A[\lfloor i/2 \rfloor] \leq A[i]$.

Desta forma, o maior (resp. menor) elemento de um *max-heap* (resp. *min-heap*) é armazenado na raiz, e a subárvore com raiz em um determinado nó contém apenas valores que são menores ou iguais (resp. que são maiores ou iguais) ao valor deste nó.

5. O algoritmo Build-Max-Heap

   O algoritmo Heapsort utiliza *max-heaps*, e portanto o primeiro passo do algoritmo será transformar o vetor A em um *max-heap*. Este trabalho é feito pelo algoritmo a seguir:

```
1 A.heap-size ← A.length for i = ⌊A.length/2⌋ downto 1 do
2 |  Max-Heapify(A,i);
3 end
4
```

Algoritmo 2: Build-Max-Heap(A)

   (a) Mostre que na representação vetorial de um *heap* com n elementos, as folhas são os elementos do vetor com índices $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \ldots, n$.

6. O algoritmo Max-Heapify

   O algoritmo Build-Max-Heap constrói o *max-heap* de baixo para cima a partir do primeiro vértice que não é uma folha, e o algoritmo Max-Heapify(A,i) reconstrói um *max-heap* a partir de uma árvore cuja raiz $A[i]$ seja o único elemento que precise ser reposicionado, ou seja, as subárvores com raiz $A[2i]$ e $A[2i+1]$ já são *max-heaps*:

```
1  l ← 2i;
2  r ← 2i + 1;
3  if l ≤ A.heap-size and A[l] > A[i] then
4  |  largest ← l;
5  end
6  else
7  |  largest ← i;
8  end
9  if r ≤ A.heap-size and A[r] > A[largest] then
10 |  largest ← r;
11 end
12 if largest ≠ i then
13 |  exchange A[i] with A[largest];
14 |  Max-Heapify(A,largest);
15 end
```

Algoritmo 3: Max-Heapify(A,i)

   (a) Complexidade de Max-Heapify

      i. Mostre que, em um *heap* com n elementos e raiz $A[i]$, cada uma das subárvores com raiz em $2i$ e $2i+1$ têm, no máximo, $2n/3$ elementos.

   Considerando o fato estabelecido no exercício anterior, temos que o tempo de execução de Max-Heapify é dado pela recorrência

$$T(n) \leq T(2n/3) + \Theta(1) \tag{2.1}$$

que, pelo teorema mestre, tem solução $O(\lg n)$.

Teorema Mestre:
Seja $T(n)$ uma função eventualmente não-decrescente que satisfaz a recorrência $T(n) = a.T(n/b) + f(n)$, para $n = b^k, k = 1, 2, 3, \ldots$ onde $a \geq 1, b \geq 2$ e $c \geq 0$. Se $T(1) = c$
$f(n) = \Theta(n^d)$, onde $d \geq 0$, então

$$T(n) = \begin{cases} \Theta(n^{\log_b a}), & \text{se } a > b^d \\ \Theta(n^d.\lg n), & \text{se } a = b^d \\ \Theta(n^d), & \text{se } a < b^d \end{cases}$$

(b) Complexidade de Build-Max-Heap

```
1 A.heap-size = A.length;
2 for i = ⌊A.length/2⌋ downto 1 do
3 |   Max-Heapify(A,i);
4 end
```

Algoritmo 4: Build-Max-Heap(A)

Qual o tempo de execução do procedimento Build-Max-Heap(A)? Temos a seguinte cota superior, considerando um *heap* com n elementos: $\sum_{i=1}^{n/2} O(\lg n) = O(\lg n.\sum_{i=1}^{n/2} 1) = O(\lg n.(n/2) = O(n.\lg n)$.

i. Uma cota mais precisa para Build-Max-Heap
A cota $O(n.\lg n)$, apesar de correta, não é a mais precisa que podemos encontrar. De fato, se observarmos que:

A. A altura de um *heap* contendo n elementos é igual a $\lfloor \lg n \rfloor$.

B. Um *heap* com n elementos possui, no máximo, $\lceil n/2^{h+1} \rceil$ nós com altura h.

Então, observando que Max-Heapify tem complexidade $O(h)$ quando executado em um nó de altura h, concluímos que o tempo de execução de Build-Max-Heap(A \)), assumindo que A possui n elementos, tem a seguinte cota superior: $\sum_{h=0}^{\lfloor \lg n \rfloor} \lceil n/2^{h+1} \rceil \cdot O(h) = O(n. \sum_{h=0}^{\lfloor \lg n \rfloor} \lceil h/2^h \rceil) = O(n)$, pois $\sum_{h=0}^{\lfloor \lg n \rfloor} h/2^h \leq \sum_{h=0}^{\infty} h/2^h$, que por sua vez converge.
Assim, um *heap* pode ser construído em tempo linear.

(c) A correção de Build-Max-Heap
Prove a seguinte invariante de laço, e conclua que o algoritmo Build-Max-Heap é correto:

No início de cada iteração do laço **for** (linhas 2-4),
cada nó nas posições $i + 1, i + 2 \ldots n$ é a raiz de um *max-heap*.

7. O algoritmo principal

O algoritmo *heapsort* recebe como argumento um vetor A qualquer contendo $n > 0$ elementos, e inicialmente o transforma em um *max-heap*. Neste momento, sabemos que a raiz do *heap* contém

o maior elemento do vetor A, que pode então ser movido para sua posição correta. Em seguida, decrementamos o tamanho do *heap* em uma unidade, e repetimos o processo:

```
1 Build-Max-Heap(A);
2 for i = A.length downto 2 do
3 |   exchange A[1] with A[i];
4 |   A.heap-size = A.heap-size - 1;
5 |   Max-Heapify(A,1);
6 end
```

Algoritmo 5: Heapsort(A)

(a) A complexidade de Heapsort

```
1 Build-Max-Heap(A);
2 for i = A.length downto 2 do
3 |   exchange A[1] with A[i];
4 |   A.heap-size = A.heap-size - 1;
5 |   Max-Heapify(A,1);
6 end
```

Algoritmo 6: Heapsort(A)

A complexidade de Heapsort(A) no pior caso, se A é um vetor com $n > 0$ elementos, é $O(n) + \sum_{i=2}^{n} O(\lg n) = O(n) + O(\lg n. \sum_{i=2}^{n} 1) = O(n) + O((n-1).\lg n) = O(n.\lg n)$.

(b) A correção de Heapsort

```
1 Build-Max-Heap(A);
2 for i = A.length downto 2 do
3 |   exchange A[1] with A[i];
4 |   A.heap-size = A.heap-size - 1;
5 |   Max-Heapify(A,1);
6 end
```

Algoritmo 7: Heapsort(A)

Prove a correção do algoritmo *Heapsort* utilizando a seguinte invariante:

No início de cada iteração do laço **for** (linhas 2-6), o subvetor $A[1..i]$ é um *max-heap* que contém os i menores elementos do vetor $A[1..n]$, e o subvetor $A[i+1..n]$ está ordenado e contém os $(n-i)$ maiores elementos do vetor $A[1..n]$.

(c) Leituras recomendadas

i. Capítulo 6 do livro do Cormen (Introduction to Algorithms);

ii. Capítulo 6 do livro do Levitin (Introduction to the Design and Analysis of Algorithms).

Filas de prioridade

Fila de prioridade é uma estrutura de dados que armazena elementos com base na sua prioridade, garantindo que elementos com maior prioridade sejam atendidos primeiro.

Assim, cada elemento aparece associado a um valor que é a sua prioridade.

As operações básicas de uma fila de prioridade são:

1. **remove**: remove o elemento com maior prioridade da fila;

2. **insert**: insere um novo elemento na fila.

1. Filas de prioridade em listas

   Podemos construir uma fila de prioridades utilizando listas:

   Em uma lista (qualquer) contendo n elementos,

   - a inserção (na primeira posição) tem complexidade constante: $\Theta(1)$.
   - a remoção de um elemento arbitrário tem complexidade linear já que pode ser necessário percorrer toda a lista: $\Theta(n)$.

   Em uma lista ordenada contendo n elementos,

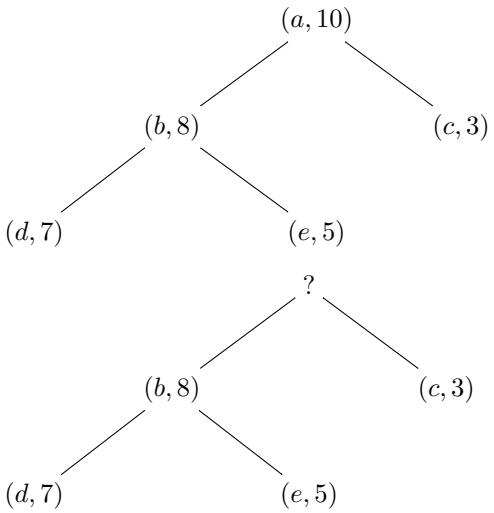
   - a inserção tem complexidade linear: $\Theta(n)$.
   - a remoção de um elemento arbitrário tem complexidade constante: $\Theta(1)$.

2. Filas de prioridade em *heaps*

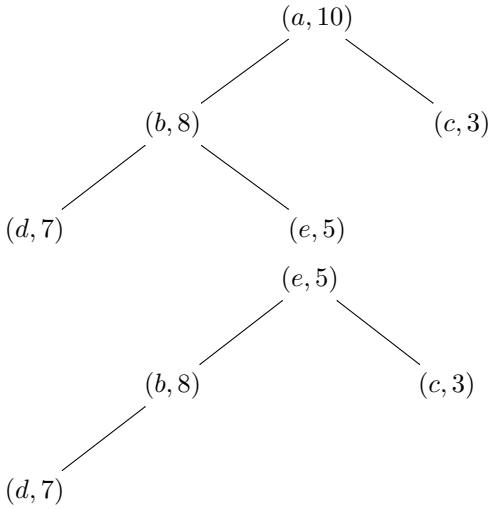
   Uma das principais aplicações da estrutura de *heap* consiste na implementação de filas de prioridade, já que tanto a inserção quanto a remoção podem ser implementados em tempo logarítmico.

   (a) Remoção

   A remoção do elemento com maior prioridade é feita em tempo constante, mas em seguida o *heap* precisa ser reconstruído via o algoritmo Max-heapify que tem complexidade $\Theta(\lg(n))$.



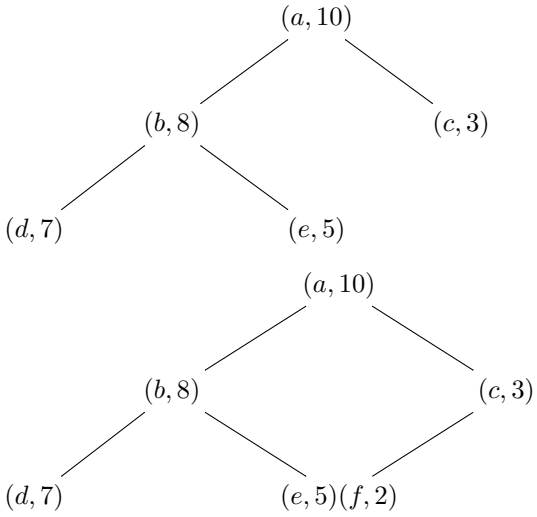
   i. remove(A)




```
1 max ← A[1];
2 exchange A[1] with A[n];
3 A.heap-size ← (n − 1);
4 Max-Heapify(A,1);
5 return max;
```

Algoritmo 8: remove($A[1..n]$)

(b) Inserção




```
1 A.size ← (n + 1);
2 k ← (n + 1);
3 A[k] ← (x, y);
4 while k > 1 and A[⌊k/2⌋] < A[k] do
5 |   exchange A[⌊k/2⌋] with A[k];
6 |   k ← ⌊k/2⌋;
7 end
```

Algoritmo 9: insert($(x, y), A[1..n]$)

i. Exercício
Prove que o algoritmo insert é correto.

```
1 A.size ← (n + 1);
2 k ← (n + 1);
3 A[k] ← (x, y);
4 while k > 1 and A[⌊k/2⌋] < A[k] do
5 |   exchange A[⌊k/2⌋] with A[k];
6 |   k ← ⌊k/2⌋;
7 end
```

Algoritmo 10: insert$((x, y), A[1..n])$

Capítulo 3

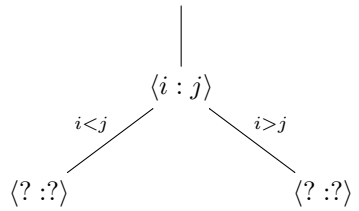
Cota inferior para algoritmos de ordenação baseados na comparação de chaves

Queremos estabelecer uma cota inferior para algoritmos baseados na comparação de chaves. Para facilitar a análise, assumiremos que os elementos a serem ordenados são **distintos**. A execução de um algoritmo qualquer de ordenação pode ser simulada por meio de uma árvore binária, chamada de **árvore de decisão**.

Considere o vetor $[x_1, x_2, \ldots, x_n]$ que queremos ordenar. Cada nó da árvore de decisão tem duas formas possíveis:

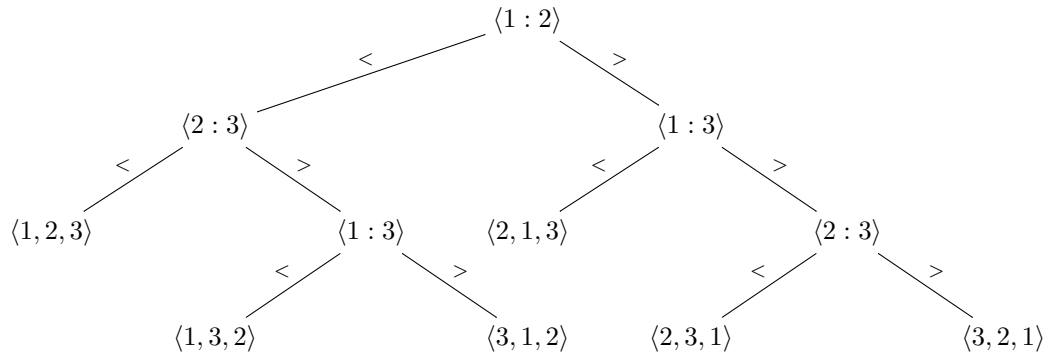
1. Uma folha da forma $\langle i_1, i_2, \ldots, i_n \rangle$ indicando que a ordenação foi finalizada, e neste caso, o vetor ordenado é $[x_{i_1}, x_{i_2}, \ldots, x_{i_n}]$ que corresponde a uma permutação do vetor original $[x_1, x_2, \ldots, x_n]$

2. Um nó interno $\langle i : j \rangle$ que corresponde à operação que compara x_i com x_j. Neste caso, este nó tem dois filhos:



3.0.1 Exemplo

Por exemplo, a árvore de decisão para o algoritmo *insertion sort* em um vetor com 3 elementos tem a seguinte forma:



3.0.2 Conclusões

O primeiro ponto a ser observado é que a árvore de decisão de um algoritmo correto deve conter pelo menos $n!$ folhas, uma para cada possível permutação da entrada. De fato, se alguma permutação não puder ser gerada, então o algoritmo pode falhar para esta permutação.

O número de nós internos em um caminho da raiz até uma folha da árvore de decisão corresponde ao número de comparações realizadas pelo algoritmo. Assim, o caminho mais longo da raiz até uma folha (altura da árvore) nos fornece o comportamento do algoritmo no pior caso.

Sabemos que uma árvore binária de altura h possui, no máximo, 2^h folhas (árvore binária completa).

Então, o número de folhas da árvore de decisão, digamos l, tem que ser pelo menos $n!$, e no máximo, 2^h.

$n! \leq l \leq 2^h \implies h \geq \lg(n!) = \Theta(n.\lg(n)))$

Exercícios:

1. Mostre que $\lg(n!) = O(n.\lg(n))$

2. Mostre que $\lg(n!) = \Omega(n.\lg(n))$

3. Como você ordenaria um vetor finito contendo apenas 0s e 1s? Qual a complexidade da sua solução?

Capítulo 4

Ordenação em tempo linear

Nesta seção veremos uma outra forma de ordenação baseada na ideia de contagem. O que precisamos fazer é contar, para cada elemento a ser ordenado, o número total de elementos que são menores do que ele e guardar este resultado em uma tabela. Os valores da tabela indicarão a posição dos elementos na lista ordenada. Por exemplo, se existem 6 elementos menores do que o elemento x, então x deve ser colocado na sétima posição do vetor ordenado. Desta forma podemos ordenar um vetor simplesmente deslocando os elementos para a posição correta. O pseudocódigo a seguir implementa esta ideia:

```
1  let C[0..n − 1] be a new array;
2  for i = 0 to n − 1 do
3  |  C[i] ← 0;
4  end
5  for i = 0 to n − 2 do
6  |  for j = i + 1 to n − 1 do
7  |  |  if A[i] < A[j] then
8  |  |  |  C[j] ← C[j] + 1;
9  |  |  end
10 |  |  else
11 |  |  |  C[i] ← C[i] + 1;
12 |  |  end
13 |  end
14 end
15 for i = 0 to n − 1 do
16 |  B[C[i]] ← A[i];
17 end
18 return B;
```

Algoritmo 11: comparison-counting-sort($A[0..n-1]$)

Qual é a complexidade desta abordagem? Que vantagens/desvantagens você pode apresentar?

4.0.1 Intervalo conhecido

No entanto, podemos utilizar a ideia da contagem de forma mais eficiente se os elementos a serem ordenados forem "conhecidos".

Por exemplo, se o vetor a ser ordenado contém apenas 0s e 1s então podemos utilizar esta informação para fazer a ordenação sem a necessidade de fazer comparações porque com uma única passagem sobre o vetor podemos obter o número k de 0s, e assim retornar o vetor contendo 0s da posição 0 até $k-1$, e 1s da posição k em diante.

De uma forma mais geral, se os elementos a serem ordenados são inteiros entre l e h então podemos computar a frequência de cada um destes elementos, e armazená-las em um vetor, digamos $C[0..h-l]$, de forma que as primeiras $C[0]$ posições do vetor ordenado serão preenchidas com l, as $C[1]$ posições seguintes com $l+1$, e assim por diante.

Observe que se os elementos do vetor original não puderem ser sobrescritos então precisaremos de um novo vetor (espaço adicional) para escrever o vetor ordenado.

4.0.2 Counting sort

O pseudocódigo a seguir, assume que cada um dos n inteiros a serem ordenados estão no intervalo entre l e h.

```
1  let C[0..h − l] be a new array;
2  for i = 0 to h − l do
3  |  C[i] ← 0;
4  end
5  for i = 0 to n − 1 do
6  |  C[A[i] − l] ← C[A[i] − l] + 1;
7  end
8  for j = 1 to h − l do
9  |  C[j] ← C[j] + C[j − 1];
10 end
11 for i = n − 1 downto 0 do
12 |  j ← A[i] − l;
13 |  B[C[j] − 1] ← A[i];
14 |  C[j] ← C[j] − 1;
15 end
16 return B;
```

Algoritmo 12: counting-sort$(A[0..n-1], l, h)$

Qual é a complexidade deste algoritmo?

Por simplicidade denotaremos $k = h - l$, ou seja, k denota o tamanho do intervalo que contém os elementos a serem ordenado. Assim, o laço **for** das linhas 2-4 é executado em tempo $\Theta(k)$, o laço das linhas 5-7 em tempo $\Theta(n)$, o laço das linhas 8-10 em tempo $\Theta(k)$, e por fim o laço das linhas 11-14 em tempo $\Theta(n)$, o que perfaz um total de $\Theta(n+k)$. Assumindo que $k = O(n)$, temos que o tempo de execução de counting-sort é $\Theta(n)$.

4.0.3 Radix Sort

O algoritmo radix sort ordena uma sequência de inteiros com d dígitos cada, em tempo linear. Ele utiliza um algoritmo auxiliar, que precisa ser estável, para ordenar a sequência de inteiros do dígito menos significativo para o mais significativo. Podemos utilizar counting-sort, por exemplo, como algoritmo auxiliar.

```
1 for i = 1 to d do
2 |  use a stable sort algorithm to sort array A on digit i;
3 end
```

Algoritmo 13: radix-sort(A, d)

Teorema
Dados n números com d dígitos, que por sua vez podem assumir até k valores, radix-sort ordena corretamente

estes números em tempo $\Theta(d.(n+k))$, se o algoritmo auxiliar estável tem complexidade de tempo $\Theta(n+k)$.

Definição
Um algoritmo de ordenação é dito *estável* se não altera a posição relativa dos elementos que têm o mesmo valor.

Exercício
Prove que o algoritmo counting-sort é estável.

Exercício
Prove que o algoritmo *mergesort* é estável.

Exercício
Prove que o algoritmo *insertion sort* é estável.

Exercício
Mostre como podemos ordenar n inteiros contidos no intervalo de 0 a $n^2 - 1$ em tempo linear, ou seja, em $O(n)$.

Capítulo 5

Algoritmos Gulosos

Nesta seção, exploraremos uma poderosa técnica para resolver problemas de otimização: os *algoritmos gulosos* (*greedy algorithms*). Esses algoritmos seguem uma estratégia bastante intuitiva: a cada passo, eles fazem a escolha que parece mais vantajosa no momento (escolha localmente ótima), com o objetivo de alcançar uma solução globalmente ótima. No entanto, nem sempre essa abordagem garante o melhor resultado final, e entender quando ela é aplicável é fundamental.

Para ilustrar melhor o conceito, começaremos com um exemplo prático antes de detalharmos a técnica em si. Dessa forma, você poderá observar como a estratégia gulosa funciona na prática e quais são suas característica essenciais.

5.0.1 Exemplo

Consideraremos o **Problema da Alocação de Atividades**: Dado um conjunto finito de atividades $S = \{a_1, a_2, \ldots, a_n\}$ tais que cada atividade a_i possui:

- horário de início s_i;
- horário de término t_i, com $0 \leq s_i < t_i < \infty$ para $1 \leq i \leq n$.

Assim, uma atividade, digamos a_i, utiliza o recurso no intervalo $[s_i, t_i)$. O objetivo é selecionar o número máximo de atividades mutuamente compatíveis de S. Duas atividades distintas a_i e a_j são ditas *compatíveis* se seus intervalos não se sobrepõem, ou seja, $s_i \geq t_j$ ou $s_j \geq t_i$.

Assumiremos que as atividades estão ordenadas em ordem crescente de horário de término: $t_1 \leq t_2 \leq \ldots \leq t_n$. Por exemplo,

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
t_i	4	5	6	7	9	9	10	11	12	14	16

Exemplos de subconjuntos de atividades mutuamente compatíveis são $\{a_3, a_9, a_{11}\}$, $\{a_1, a_4, a_8, a_{11}\}$ e $\{a_2, a_4, a_9, a_{11}\}$.

Como no caso de **programação dinâmica**, os algoritmos gulosos devem satisfazer **propriedade da subestrutura ótima**, permitindo que soluções ótimas de subproblemas sejam combinadas para resolver o problema original.

Para mostrarmos que o problema da alocação de tarefas satisfaz a propriedade da subsestrutura ótima, denotaremos por S_{ij} o subconjunto de S contendo as atividades que iniciam após a atividade a_i terminar, e terminam antes da atividade a_j começar. Queremos encontrar o conjunto máximo de atividades compatíveis de S_{ij}, que denotaremos por A_{ij}. Se A_{ij} possui a atividade a_k então:

- Devemos resolver o subproblemaS_{ik} (atividades entre a_i e a_k), ou seja, devemos encontrar o conjunto A_{ik};

- Devemos resolver o subproblemaS_{kj} (atividades entre a_k e a_j), ou seja, devemos encontrar o conjunto A_{kj}.

Afirmação: Uma solução ótima A_{ij} contém necessariamente soluções ótimas para os subproblemas S_{ik} e S_{kj}.

Prova. Suponha que exista um subconjunto A'_{kj} com mais atividades que A_{kj}, ou seja, tal que $|A'_{kj}| > |A_{kj}|$. Então poderíamos usar A'_{kj} no lugar de A_{kj} e teríamos $|A_{ik}|+|A'_{kj}|+1 > |A_{ik}|+|A_{kj}|+1 = |A_{ij}|$ o que contradiz a suposição de que A_{ij} é uma solução ótima. Um argumento análogo pode ser utilizado para A_{ik}. □

A ideia central da estratégia gulosa consiste em fazer a escolha ótima local ao invés de explorar todas as possibilidades como acontece em programação dinâmica. Isso faz com que a estratégia gulosa reduza drasticamente a complexidade do problema.

Para o problema da alocação de atividades, a melhor escolha local consiste em selecionar a atividade que termina primeiro, pois ela libera recurso o mais cedo possível (usa o recurso de forma mínima), maximizando o tempo disponível para as demais atividades. Considerando que as atividades estão ordenadas crescentemente de acordo com o horário de término, a escolha gulosa seria a_1. Note que ao fazermos uma escolha gulosa passamos a ter apenas um novo subproblema para resolver. Denotaremos por $S_k = \{a_i \in S \mid s_i \geq t_k\}$ o conjunto das atividades que iniciam depois do término da atividade a_k. Assim, fazendo a escolha gulosa a_1 teremos apenas o subproblema S_1 para ser resolvido. O subproblema S_1 é obtido eliminando todas as atividades que conflitam com a_1. A ideia é então construir uma solução ótima para S_1 e utilizá-la na construção da solução ótima do problema original. A questão que surge é: Esta ideia funciona? Ela está correta?

Afirmação: Se S_k é um subproblema não vazio e a_m é a atividade em S_k com o menor tempo de término, então a_m faz parte de alguma solução ótima para S_k.

Prova. Seja A_k uma solução ótima para S_k. Se $a_m \in A_k$ então a afirmação está correta. Caso contrário, seja a_j a atividade em A_k que termina primeiro, ou seja, a_j é tal que t_j é o menor valor de término em A_k. Como $t_m \leq t_j$, a substituição de a_j por a_m não causa conflitos com as outras atividades em A_k. Assim, $A'_k = A_k - \{a_j\} \cup \{a_m\}$ também é uma solução ótima e contém a_m. □

Algumas considerações:

1. A estratégia gulosa nem sempre produz uma solução ótima;

2. Para aplicar a estratégia gulosa devemos observar 2 pontos:

   (a) O subproblema deve possuir a **propriedade da subsestrutura ótima**, e;

   (b) O subproblema deve possuir a **propriedade da escolha gulosa**: escolhas locais ótimas levam a uma solução global ótima.

5.0.2 Exercícios

1. Considere o problema da alocação de atividades visto anteriormente e três algoritmos gulosos baseados nas seguintes regras:

   (a) seleção da atividade que inicia primeiro;

   (b) seleção da atividade que tem menor duração;

   (c) seleção da atividade que termina primeiro.

   Para cada um destes algoritmos, mostre que ele sempre produz uma solução ótima ou apresente um contraexemplo.

2. Considere um sistema monetário com moedas de valores $\{1, 5, 10, 25\}$.

   (a) Utilize a estratégia gulosa (seleção da maior moeda possível) para dar um troco de 36 com o menor número possível de moedas.

   (b) Mostre que a estratégia gulosa é ótima para este sistema monetário.

   (c) Agora suponha que as moedas disponíveis são $\{1, 4, 5\}$. A estratégia gulosa ainda funciona para dar um troco de 8? Justifique.

3. Considere o problema da mochila booleana (ou problema da mochila 0-1): Dado um conjunto de n objetos com peso w_i e valor v_i $1 \leq i \leq n$, e uma mochila com capacidade de carregar o peso W, onde W, w_i e v_i são inteiros para $1 \leq i \leq n$, quais objetos devem ser colocados na mochila para que o valor total seja máximo? Mostre que a estratégia gulosa não resolve este problema, e construa uma solução utilizando programação dinâmica para este problema.

4. Considere o problema da mochila fracionária: Considere n objetos com peso w_i e valor v_i $1 \leq i \leq n$, e uma mochila com capacidade de peso W, de forma que frações de cada objeto podem ser selecionadas. Que fração de cada objeto deve ser colocada na mochila de modo a maximizar o valor total? Em outras palavras, selecione frações $f_i \in [0, 1]$ dos itens tais que $\sum_{i=1}^{n} f_i.w_i \leq W$ e $\sum_{i=1}^{n} f_i.v_i$ é máximo. Mostre que este problema possui a propriedade da escolha gulosa.

5.0.3 Leituras complementares

1. [6] (Capítulo 16)

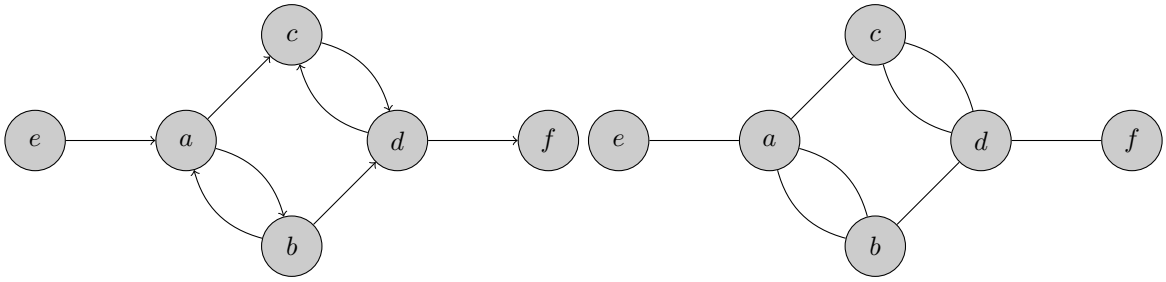
2. [14] (Capítulo 9)

3. [4] (Capítulo 6)

4. Comparativo entre a estratégia gulosa e a programação dinâmica para o problema do troco com sistemas de moedas canônicos, Lucas V. Mattioli, TCC UnB, 2018.

5.0.4 Algoritmos gulosos em grafos

um grafo G é um par (V, E), onde V é o conjunto de vértices, e E é o conjunto de arestas. Quando as arestas do grafo são dirigidas, falamos em digrafo. A seguir, apresentamos um exemplo de um digrafo à esquerda, e um grafo à direita:



Do ponto de vista formal, temos as seguintes definições:

Um *grafo* (não dirigido) G é um par (V, E) onde V é um conjunto finito não-vazio, e E é um conjunto de pares não-ordenados de elementos de V. Em grafos não-dirigidos arestas de um vértice para ele mesmo (*auto-loop*) são proibidas, e portanto toda aresta liga dois vértices distintos.

Um *digrafo* (ou um grafo dirigido) G é um par (V, E) onde V é um conjunto finito não-vazio, e E é uma relação binária sobre V. Em digrafos *auto-loops* são permitidos.

Dado um grafo $G = (V, E)$, a versão dirigida de G é o digrafo $G' = (V, E')$ onde $(u, v) \in E'$ se, e somente se $(u, v) \in E$, isto é, substituímos cada aresta $(u, v) \in E$ pelas duas arestas dirigidas (u, v) e (v, u) na versão dirigida.

Dado um digrafo $G = (V, E)$, a versão não-dirigida (ou o *grafo associado*) de G é o digrafo $G' = (V, E')$ onde $(u, v) \in E'$ se, e somente se $u \neq v$ e $(u, v) \in E$, ou seja, a versão não-dirigida de um grafo é construída removendo a direção das arestas e os auto-loops.

5.0.5 Representação de grafos

Existem duas formas bastante comuns para representar um (di)grafo $G = (V, E)$: matriz de adjacências ou listas de adjacências. As duas representações se aplicam a grafos e a digrafos.

A representação de um grafo $G = (V, E)$ por *listas de adjacências* consiste de um vetor Adj de $|V|$ listas, uma para cada vértice em V. Para cada $u \in V$, a lista de adjacências $Adj[u]$ contém todos os vértices v tais que $(u, v) \in E$, ou seja, contém todos os vértices adjacentes a u em G. Escreveremos $G.Adj[u]$ para se referir a lista $Adj[u]$ de G.

Se G é um digrafo então a soma dos comprimentos de todas as listas de adjacências é igual a $|E|$, já que uma aresta (u, v) é representada por uma ocorrência de v em $Adj[u]$. Se G for um grafo (não-dirigido) então a soma dos comprimentos de todas as listas de adjacências é igual a $2|E|$ pois uma aresta (u, v) é representada pela ocorrência de v em $Adj[u]$, e pela ocorrência de u em $Adj[v]$. Em ambos os casos, a representação por listas de adjacências utiliza espaço da ordem de $\Theta(V + E)$, ou seja, linear no tamanho da representação do grafo. Esta representação não é adequada para grafos densos, mas é adequada para grafos esparços, isto é, grafos com "poucas"arestas.

Uma desvantagem das listas de adjacências é que elas não fornecem uma forma rápida de determinar se a aresta (u, v) está ou não no (di)grafo. Para isto precisamos procurar por v em $Adj[u]$. A representação por matrizes de adjacências contorna este problema a um custo assintoticamente maior de espaço.

A representação de um grafo $G = (V, E)$ por *matrizes de adjacências* assume uma enumeração (qualquer) $1, 2, \ldots, |V|$ dos vértices de G, e consiste de uma matriz $A = (a_{ij})$ de dimensão $|V| \times |V|$ tal que

$$a_{ij} = \begin{cases} 1, & \text{se } (i,j) \in E \\ 0, & \text{caso contrário.} \end{cases} \tag{5.1}$$

A representação por matrizes de adjacências requer espaço da ordem de $\Theta(V^2)$, independentemente do número de arestas do grafo. Esta representação não é adequada para grafos esparços, mas é adequada para grafos densos ($E = \Theta(V^2)$).

Diversas definições coincidem para grafos e digrafos, mas algumas diferenças podem ocorrer dependendo do contexto. Por exemplo, se (u, v) é uma aresta de um digrafo $G = (V, E)$ então dizemos que (u, v) sai de u, e entra (ou incide) em v. Já quando (u, v) é uma aresta de um grafo, dizemos que (u, v) incide em u e v.

O *grau* de um vértice em um grafo é o número de arestas que incidem sobre ele. Um vértice de grau 0 é dito *isolado*. Em um digrafo, o *grau de saída* (resp. *grau de entrada*) de um vértice é o número de arestas que saem (resp. chegam) neste vérice. O *grau* de um vértice em um digrafo é a soma dos seus graus de saída e entrada.

Se (u, v) é uma aresta do (di)grafo $G = (V, E)$ então dizemos que v é adjacente a u. Note que a relação de adjacência é simétrica em grafos, mas não em digrafos. De fato, se um digrafo $G = (V, E)$ possui a aresta (u, v), mas não possui a aresta (v, u) então v é adjacente a u, mas u não é adjacente a v.

Um grafo (não-dirigido) é dito *completo* se qualquer par de vértices é adjacente.

Um *caminho* de comprimento k de um vértice u para um vértice v em um grafo $G = (V, E)$ é uma sequência $\langle v_0, v_1, \ldots, v_k \rangle$ de vértices tal que $v_0 = u$ e $v_k = v$, e $(v_{i-1}, v_i) \in E$ para $i = 1, 2, \ldots, k$. O comprimento de um caminho é o número de arestas deste caminho. Existe sempre um caminho de comprimento 0 de u para u, qualquer que seja o vértice u. Um *subcaminho* de um caminho $p = \langle v_0, v_1, \ldots, v_k \rangle$ é uma sequência contígua dos vértices de p, isto é, quaisquer que sejam $0 \leq i \leq j \leq k$, a subsequência de vértices $\langle v_i, v_{i+1}, \ldots, v_j \rangle$ é um subcaminho de p.

Quando existe um caminho p de u para v, dizemos que v é alcançável a partir de u, o que normalmente é denotado por $u \overset{p}{\rightsquigarrow} v$, quando o grafo é dirigido.

Um caminho é dito *simples* se todos os vértices no caminho são distintos.

A definição de ciclos em grafos requer cuidado porque difere para grafos e digrafos. Em um digrafo, um *ciclo* é um caminho não-nulo, ou seja, de comprimento estritamente maior do que 0, tal que o primeiro e o último vértices são idênticos. Em um digrafo, um caminho $\langle v_0, v_1, \ldots, v_k \rangle$ forma um *ciclo* se $v_0 = v_k$, e este caminho possui pelo menos uma aresta. Dois caminhos $\langle v_0, v_1, \ldots, v_{k-1}, v_0 \rangle$ e $\langle v'_0, v'_1, \ldots, v'_{k-1}, v'_0 \rangle$ formam o mesmo ciclo se existir j tal que $v'_i = v_{(i+j) \mod k}$ para $i = 0, 1, \ldots, k-1$. Um auto-loop é um ciclo de comprimento 1, e um digrafo sem auto-loops é dito *simples*. Em um grafo, as definições são similares, mas existe um requerimento adicional de que se qualquer aresta aparece mais de uma vez, então ela aparece com a mesma orientação: em um caminho $\langle v_0, v_1, \ldots v_{k-1}, v_k \rangle$, se $v_i = x$ e $v_{i+1} = y$ para $0 \leq i < k$, então não pode existir j tal que $v_j = y$ e $v_{j+1} = x$. Um ciclo é dito *simples* se seus vértices são distintos. Um (di)grafo sem ciclos é dito *acíclico*.

Um grafo acíclico é chamado de *floresta* (não-dirigida), e se o grafo for conexo então é chamado de *árvore* (livre ou não-dirigida). Um digrafo acíclico é normalmente abreviado por DAG. Nenhuma condição de conectividade é assumida em DAGs.

Um *caminho eulerianos* em um (di)grafo conexo G é um caminho que percorre cada aresta apenas uma vez, mas vértices podem ser visitados mais de uma vez. Um *caminho hamiltoniano* em um (di)grafo

G é um caminho simples que contém cada vértice de G. Um *ciclo hamiltoniano* em um (di)grafo G é um ciclo simples que contém cada vértice de G.

Note que em um ciclo hamiltoniano cada vértice do (di)grafo é visitado um única vez. Em um grafo (não dirigido) um caminho $\langle v_0, v_1, \ldots, v_k \rangle$ forma um ciclo se $k \geq 3$ e $v_0 = v_k$.

A definição de *conectividade* exige mais cuidado porque difere entre grafos e digrafos:

- Um grafo é dito *conexo* se para cada par de vértices v e w, existe um caminho entre v e w, ou seja, se qualquer vértice é alcançável a partir de todos os outros. As *componentes conexas* de um grafo são as classes de equivalência dos vértices sob a relação "é alcançável a partir de". Assim, um grafo é conexo se possui apenas uma componente conexa.

- A conectividade em digrafos é dividida em dois casos:

  - Um digrafo é *fortemente conexo* se o vértice u é alcançável a partir do vértice v, e vice-versa, quaisquer que sejam $u, v \in V$. As componentes fortemente conexas de um digrafo são as classes de equivalência dos vértices sob a relação "são mutuamente alcançáceis". Um digrafo é fortemente conexo se possui apenas uma componente fortemente conexa.

  - Um digrafo é *(fracamente) conexo* se o grafo associado é conexo, mas não é fortemente conexo.

Dois grafos $G = (V, E)$ e $G' = (V', E')$ são *isomorfos* se existir uma bijeção $f : V \to V'$ tal que $(u, v) \in E$ se, e somente se $(f(u), f(v)) \in E'$. Isto significa que podemos renomear os vértices de G como sendo os de G' mantendo as arestas correspondentes em G e G'. Dizemos que $G' = (V', E')$ é um *subgrafo* de $G = (V, E)$, notação $H \subseteq G$, se $V' \subseteq V$ e $E' \subseteq E$. Alguns subgrafos especiais:

- Um subgrafo H de um grafo G é dito *gerador* (*spanning*) se contém todos os vértices de G, isto é, se $H.V = G.V$ usando a notação de atributos.

- Um subgrafo H de um grafo G é *próprio*, notação $H \subset G$, se for diferente de G, isto é, se $H.V < G.V$ ou $H.E < G.E$.

- Dado $X \subseteq G.V$, o subgrafo de G *induzido* por X, notação $G[X]$, é o grafo $G' = (X, E')$ onde $E' = \{(u, v) \in E : u, v \in X\}$.

- Dado $Y \subseteq G.E$, o subgrafo de G *induzido* por Y, notação $G[Y]$, é o grafo $G' = (V', Y)$ onde se $(u, v) \in Y$ então $u, v \in V'$.

5.0.6 Busca em Largura

Busca em largura (BFS) é um dos algoritmos mais simples para busca em grafos, além de ser utilizado em outros algoritmos sobre grafos. O algoritmo funciona tanto para grafos quanto para digrafos. Dado um grafo $G = (V, E)$ e um vértice $s \in V$ que chamaremos de *origem*, o algoritmo de busca em largura sistematicamente explora as arestas de G para descobrir todos os vértices que são alcançáceis a partir de s. O nome *busca em largura* se dá porque o algoritmo separa a fronteira entre os vértices que já foram descobertos dos que ainda não o foram a partir da sua distância até a origem s. Assim, o algoritmo descobre todos os vértices que estão a uma distância k da origem antes de descobrir qualquer vértice que esteja a uma distância $k + 1$ da origem.

A seguir apresentamos o pseudocódigo do algoritmo BFS, que recebe como argumento o grafo G, e o vértice s a partir do qual a busca é iniciada.

```
1  for each vertex u ∈ G.V − {s} do
2  |  u.color ← WHITE;
3  end
4  s.color ← GRAY;
5  Q ← ∅;
6  enqueue(Q, s);
7  while Q ≠ ∅ do
8  |  u ← dequeue(Q);
9  |  for each v ∈ G.Adj[u] do
10 |  |  if v.color == WHITE then
11 |  |  |  v.color ← GRAY;
12 |  |  |  enqueue(Q, v);
13 |  |  end
14 |  end
15 end
```

Algoritmo 14: BFS(G, s)

A inicialização (linhas 1-3) percorre todos os vértices, exceto s, e portanto tem tempo de execução limitado pelo número de vértices do grafo, i.e. $\Theta(V)$. As operações das linhas 4, 5 e 6 são executadas em tempo constante. O *loop* das linhas 7-15 precisa de uma análise mais cuidadosa. Os vértices marcados com WHITE durante a inicialização correspondem aos vértices que ainda não foram visitados e, uma vez que um vértice é visitado, ele é imediatamente marcado com GRAY (linha 11) e inserido na fila Q (linha 12). Durante a primeira execução do *loop*, s é retirado da fila, e cada um dos vértices da lista de adjacências de s é marcado como visitado (GRAY), e então colocado na fila Q. Assim, o percorrimento do grafo inicia pelo vértice s, e em seguida são percorridos todos os vértices adjacentes a s perfazendo um total de $1 + |Adj[s]|$ vértices visitados nesta etapa. Na segunda execução do laço, um vértice adjacente a s, digamos u, é retirado da fila Q e todos os vértices adjacentes a u, que ainda não tenham sido visitados, serão marcados como visitados, e inseridos na fila, de forma que, no máximo, $|Adj[u]|$ vértices serão visitados porque alguns dos elementos em $Adj[u]$ já podem ter sido visitados: de fato, se um vértice v for simultaneamente adjacente aos vértices s e u, que por sua vez é também adjacente a s, então v será inserido na fila na primeira execução do laço, i.e. durante o percorrimento de $Adj[s]$, e será ignorado durante o percorrimento de $Adj[u]$. Portanto $1 + |Adj[s]| + |Adj[u]|$ é uma cota superior para o número de vértices visitados até este momento. Note que um vértice só é inserido na fila uma única vez. Mais ainda, para que um vértice seja inserido na fila Q é necessário que ele seja alcançável a partir de s, e portanto, o processo de enfileiramento e desenfileiramento tem custo limitado por $O(V)$. Cada um dos vértices enfileirados terá sua lista de adjacências percorrida, mas apenas alguns de seus elementos serão visitados. Assim, se $\{s, v_1, v_2, \ldots, v_k\}$ é o conjunto de todos os vértices alcançáveis a partir de s no grafo, então todos eles serão inseridos na fila Q logo após serem visitados, o que tem custo limitado por $O(V)$. Em seguida, para cada vértice $u \in \{s, v_1, v_2, \ldots, v_k\}$, os vértices de $Adj[u]$ que ainda não foram visitados são marcados com GRAY, o que tem custo $O(Adj[u])$. Somando este custo para cada um dos vértices do conjunto $\{s, v_1, v_2, \ldots, v_k\}$, temos custo limitado por $\sum_{u \in \{s, v_1, v_2, \ldots, v_k\}} |Adj[u]| \leq \sum_{u \in V} |Adj[u]| = \Theta(E)$. Assim, o custo total para percorrer o grafo é limitado por $O(V + E)$.

Mais sucintamente: depois da inicialização de BFS (linhas 1-5) nenhum vértice volta a ser marcado com WHITE. Então o teste da linha 13 garante que cada vértice é enfileirado apenas uma vez, e portanto desenfileirado apenas uma vez também. As operações de enfileiramento e desenfileiramento tomam tempo constante $\Theta(1)$, e portanto o tempo requerido pela operação de enfileiramento é da ordem de $O(V)$. Como BFS percorre a lista de adjacências somente quando o vértice é desenfileirado, concluímos que cada lista de adjacências é percorrida apenas uma vez. Como a soma dos comprimentos das listas de adjacências é $\Theta(E)$, o tempo total utilizado no percorrimento das listas de adjacências é limitado por $O(E)$. Portanto BFS possui tempo de execução limitado por $O(V + E)$, isto é, é linear no tamanho da representação de G. Observe que se $|E| \geq |V|$ então $|V| + |E| \leq |E| + |E| = 2.|E|$, e portanto neste

caso, $O(V + E)$ significa $O(E)$. Analogamente, se $|E| < |V|$ então $O(V + E)$ significa $O(V)$. Em geral, $O(x + y)$ significa $O(max(x, y))$.

Exercício 1. *Considere uma enumeração qualquer* $1, 2, \ldots, |V|$ *dos vértices de* G*. A matriz de adjacências* $G.A$ *de dimensão* $|V| \times |V|$ *é dada por:*

$$G.A[i][j] = \begin{cases} 1, & \text{se } (i,j) \in G.E \\ 0, & \text{caso contrário.} \end{cases} \tag{5.2}$$

O pseudocódigo a seguir apresenta o algoritmo BFS onde o grafo G *é representado por sua matriz de adjacências:*

```
1  for i = 1 to |V| do
2  |  i.color ← WHITE;
3  end
4  s.color ← GRAY;
5  Q ← ∅;
6  enqueue(Q, s);
7  while Q ≠ ∅ do
8  |  u ← dequeue(Q);
9  |  for i = 1 to |V| do
10 |  |  if G.A[u][i] = 1 and i.color == WHITE then
11 |  |  |  i.color ← GRAY;
12 |  |  |  enqueue(Q, i);
13 |  |  end
14 |  end
15 end
```

Algoritmo 15: BFS(G, s)

Qual é a complexidade de tempo de BFS neste caso?

5.0.7 Busca em Profundidade

Nesta seção estudaremos outro algoritmo de busca em grafos, o algoritmo de busca em profundidade (*Depth-first search* - DFS). Neste caso, diferentemente do algoritmo BFS visto anteriormente, a busca vai o mais profundo possível no grafo visitando um vértice adjacente ao vértice que acaba de ser visitado, e em seguida visita outro vértice adjacente ao vértice adjacente visitado até que não seja mais possível, quando a busca retorna (*backtrack*) para o vértice a partir do qual o vértice que não tem mais adjacentes a serem visitados foi descoberto. Este processo de busca continua até que todos os vértices alcançáveis a partir do vértice inicial (fonte) forem visitados. Caso ainda existam vértices não visitados, DFS seleciona algum destes vértices como fonte, e repete esta estratégia de busca. O algoritmo para depois que todos os vértices tenha sido visitados. Vejamos o pseudocódigo de DFS:

```
1  for each vertex u ∈ G.V do
2  |  u.color ← WHITE u.π ← NIL end
3  |  time ← 0 for each vertex u ∈ G.V do
4  |  |  if u.color == WHITE then
5  |  |  |  DFS-Visit(G, u)
6  |  |  end
7  |  end
8  |
```

Algoritmo 16: DFS(G)

```
1 time ← time + 1 u.d ← time u.color ← GRAY for each v ∈ G.Adj[u] do
2 |   if v.color == WHITE then
3 |   |   v.π ← u;
4 |   |   DFS-Visit(G, v)
5 |   end
6 end
7 u.color ← BLACK time ← time + 1 u.f ← time
```

Algoritmo 17: DFS-Visit(G, u)

Qual o tempo de execução de DFS? O laço das linhas 1-4 é executado em tempo $\Theta(V)$. Já o laço das linhas 6-10 exige um pouco mais de atenção porque este laço faz uma chamada ao algoritmo DFS-visit. Então vamos determinar o custo de DFS-Visit primeiro. Em cada execução de DFS-Visit(G, u), o *loop* das linhas 4-9 é executado $|Adj[u]|$ vezes. Como

$$\sum_{v \in V} |Adj[v]| = \Theta(E)$$

o custo total de DFS-Visit é $\Theta(E)$, e portanto, o custo total de DFS é $\Theta(V + E)$.

Definição 1. *O* subgrafo predecessor *da busca em profundidade é definido por:*

- $G_\pi = (V, E_\pi)$*, onde* $E_\pi = \{(v.\pi, v) : v \in V \text{ and } v.\pi \neq NIL\}$

O subgrafo predecessor de uma busca em profundidade forma uma floresta.

Teorema 2. *Para dois vértices u e v quaisquer de um (di)grafo G, apenas uma das seguintes propriedades ocorre em uma busca em profundidade (DFS) em G, considerando que $u.d < v.d$:*

1. *$u.d < v.d < v.f < u.f$, e v é um descendente de u no subgrafo predecessor de G;*

2. *$u.d < u.f < v.d < v.f$, e u não é um descendente de v no subgrafo predecessor de G, ou vice-versa.*

Teorema 3. *Seja $G = (V, E)$ um grafo, e considere a floresta F obtida após a execução de DFS(G). As componentes de F são precisamente as componentes conexas de G.*

Corolário 4. *As componentes conexas de um grafo $G = (V, E)$ podem ser encontradas em tempo $\Theta(V + E)$.*

5.0.8 Árvores Geradoras Mínimas

Introdução

Nesta seção estudaremos o problema de determinar a árvore geradora mínima (*minimum spanning tree*) de um grafo G com pesos [6]. Este problema possui diversas aplicações, como na construção de conexões em redes de computadores, conexões viárias entre diversos pontos de uma cidade ou circuitos eletrônicos onde as componentes dos circuitos são ligadas por fios. No contexto dos circuitos eletrônicos, desejamos, em geral, utilizar a menor quantidade possível de fios. Podemos modelar este problema a partir de um grafo $G = (V, E)$, onde V corresponde ao conjunto de componentes do circuito, e E o conjunto de ligações entre duas destas componentes. O peso $w(u, v)$ da aresta (u, v) especifica o custo (quantidade de fios) necessário para conectar u e v. Portanto desejamos encontrar uma árvore $T \subseteq E$ que conecta todos os

vértices de G e cujo peso total

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

seja mínimo.

Formalmente, uma árvore geradora é definida como a seguir:

Sejam $G = (V, E)$ um grafo e G' um subgrafo de G que seja uma árvore. Dizemos que G' é uma *árvore geradora (spanning tree)* de G se contém todos os vértices de G.

Sabemos que árvores são conexas, e portanto se o grafo G possui árvore geradora então G é necessariamente conexo. Reciprocamente, se G é um grafo conexo então G possui pelo menos uma árvore geradora.

Seja $G = (V, E)$ um grafo com função peso w. Uma *árvore geradora mínima* (*Minimum Spanning Tree* (MST)) de G é uma árvore geradora de custo mínimo, isto é, tal que a soma dos pesos de suas arestas é mínimo.

Assim, G' é uma árvore geradora mínima do grafo G se contém todos os vértices de G e nenhuma outra árvore geradora de G possui custo estritamente menor do que G'.

Quantas árvores geradoras um grafo qualquer possui?

(Teorema de Cayley, 1889) Existem V^{V-2} árvores geradoras em um grafo completo com V vértices.

Portanto, utilizar um algoritmo força bruta para obter uma árvore geradora mínima não é uma boa ideia. Inicialmente, definiremos um método genérico que manipula o conjunto $A \subseteq E$ de arestas baseado na seguinte invariante:

Antes de cada iteração, o conjunto A é um subconjunto de alguma árvore geradora mínima de G.

A ideia é adicionar novas arestas ao conjunto A de forma a não violar a invariante acima. Chamaremos de *seguras* (*safe*) as arestas que podem ser adicionadas ao conjunto A sem violar a invariante. Desta forma, o procedimento genérico para construir árvores geradoras mínimas em grafos conexos é dado como a seguir:

```
1 A = ∅;
2 while A does not form a spanning tree do
3 |   find a safe edge (u, v) for A;
4 |   A = A ∪ {(u, v)};
5 end
6 return A;
```

Algoritmo 18: Generic-MST(G, w)

Como identificar uma aresta segura? A seguir veremos uma regra para reconhecer arestas seguras.

Um *corte* $(S, V - S)$ de um grafo $G(V, E)$ é uma partição de V. Dizemos que a aresta (u, v) *cruza o corte*, se $u \in S$ e $v \in S - V$. Dizemos que o corte *respeita* o conjunto A de arestas se nenhuma aresta de A cruza o corte. Uma aresta que cruza um corte é dita *leve* se tem peso mínimo dentre todas as arestas que cruzam o corte.

Teorema 5. *Sejam $G = (V, E)$ um grafo conexo com função peso w, e $A \subseteq E$ um conjunto contido em alguma árvore geradora mínima de G. Se $(S, V - S)$ é um corte de G que respeita A, e (u, v) é uma*

aresta leve que cruza o corte $(S, V - S)$, então (u, v) é segura para A.

Demonstração. Exercício.

□

Corolário 6. *Sejam $G = (V, E)$ um grafo conexo com função peso w, $A \subseteq E$ um conjunto contido em alguma árvore geradora mínima de G, e $C = (V_C, E_C)$ uma componente conexa na floresta $G_A = (V, A)$. Se (u, v) é uma aresta leve que conecta C a outra componente em G_A então (u, v) é segura para A.*

Exercício 2. *Seja (u, v) uma aresta de peso mínimo em um grafo conexo G. Mostre que (u, v) pertence a alguma árvore geradora mínima de G.*

O algoritmo de Prim

O algoritmo de Prim (Robert C. Prim, 1957) consiste em uma especialização do algoritmo genérico dado acima. Para construir uma árvore geradora mínima de um grafo G conexo com função peso w, o algoritmo vai partir de um vértice r dado como entrada. A árvore então será desenvolvida a partir de r, que chamaremos de raiz da árvore geradora mínima. A ideia é que em cada passo, o algoritmo vai adicionar uma nova aresta leve à árvore construída a partir de r. Pelo Corolário 6, temos que cada aresta adicionada à árvore é segura. O algoritmo de Prim é classificado como algoritmo guloso porque em cada passo a aresta que é adicionada à árvore é a que tem menor peso dentre as que podem ser selecionadas. Ou seja, a estratégia gulosa, a cada passo, faz a escolha local ótima na esperança de obter uma solução ótima global. Algumas considerações:

1. A estratégia gulosa nem sempre produz uma solução ótima;

2. Para aplicar a estratégia gulosa devemos observar 2 pontos:

   (a) O subproblema deve possuir a propriedade da subsestrutura ótima, e;

   (b) O subproblema deve possui a propriedade da escolha gulosa: uma solução ótima global pode ser obtida a partir de escolhas gulosas ótimas locais.

Uma implementação eficiente do algoritmo de Prim necessita de uma forma rápida para selecionar uma aresta segura a ser inserida na árvore construída até aquele momento. Para isto, todos os vértices que ainda não fazem parte da árvore em construção são armazenados em uma fila de prioridade, onde o atributo *key* de cada vértice corresponde à sua prioridade. Uma maneira eficiente de implementar filas de prioridade é utilizando *heaps*. No caso de um *heap* de máximo (resp. mínimo) teremos uma fila de prioridade de máximo (resp. mínimo). No caso do algoritmo de Prim utilizaremos filas de prioridade de mínimo que possuem as seguintes operações:

- Insert(S, k): ins a chave k no conjunto S:

```
1 S.heap_size ← S.heap_size + 1;
2 S[S.heap_size] ← ∞;
3 Decrease-Key(S, S.heap_size, k);
```

Algoritmo 19: Insert(S, k)

- Decrease-Key(S, i, k): decrementa o valor da chave $S[i]$ para o novo valor k, que deve ser menor ou igual a $S[i]$:

```
1 if k > S[i] then
2 |   error "new key is larger than current key";
3 end
4 S[i] ← k;
5 while i > 1 and S[Parent(i)] > S[i] do
6 |   exchange S[i] with S[Parent(i)];
7 |   i ← Parent(i);
8 end
```

Algoritmo 20: Decrease-Key(S, i, k)

Este algoritmo tem complexidade $O(\lg n)$ que corresponde ao comprimento máximo da distância entre o elemento que teve sua prioridade alterada (linha 4), e a raiz do *heap*.

- Minimum(S): retorna o elemento de S com a maior prioridade, ou seja, o elemento de S que possui a menor chave:

```
1 if S.heap_size < 1 then
2 |   error "heap underflow";
3 end
4 min ← S[1];
```

Algoritmo 21: Minimum(S)

- Extract-Min(S): remove e retorna o elemento de S que possui a menor chave:

```
1 min ← Minimum(S);
2 S[1] ← S[S.heap_size];
3 S.heap_size ← S.heap_size − 1;
4 Min-Heapify(S, 1);
5 return min;
```

Algoritmo 22: Extract-Min(S)

onde Min-Heapify é dada por:

```
1  l ← 2i;
2  r ← 2i + 1;
3  if l ≤ S.heap-size and S[l] < S[i] then
4  |   smallest ← l;
5  end
6  else
7  |   smallest ← i;
8  end
9  if r ≤ S.heap-size and S[r] < S[smallest] then
10 |   smallest ← r;
11 end
12 if smallest ≠ i then
13 |   exchange S[i] with S[smallest];
14 |   Min-Heapify(S,smallest);
15 end
```

Algoritmo 23: Min-Heapify(S, i)

A complexidade de Min-Heapify é obtida a partir da recorrência $T(n) \leq T(2n/3) + O(1)$ que tem solução $O(\lg n)$ (Veja a aula sobre o algoritmo *heapsort*). Assim, a complexidade de Extract-Min é tam-

bém $O(\lg n)$.

O algoritmo de Prim é dado como a seguir:

```
1  for each vertex v ∈ G.V do
2  |  u.key ← ∞;
3  |  u.π ← NIL;
4  end
5  r.key ← 0;
6  Q ← ∅ ;                                          // Inicializa uma fila vazia
7  for each vertex u ∈ G.V do
8  |  Insert(Q, u);
9  end
10 while Q ≠ ∅ do
11 |  u ← Extract-Min(Q);
12 |  for each v ∈ G.Adj[u] do
13 |  |  if v ∈ Q and w(u, v) < v.key then
14 |  |  |  v.π ← u;
15 |  |  |  v.key ← w(u, v);
16 |  |  |  Decrease-Key(Q, v, w(u, v));
17 |  |  end
18 |  end
19 end
```

Algoritmo 24: MST-Prim(G, w, r)

Agora conclua que a complexidade do algoritmo de Prim é $O((V + E) \lg V) = O(E \lg V)$, pois $E \geq V - 1$ em um grafo conexo.

Por fim, a correção do algoritmo de Prim pode ser estabelecida pelo seguinte teorema:

Teorema. Seja $G = (V, E)$ um grafo conexo com função peso w, e $r \in V$. Após a execução de MST-Prim(G, w, r), o subgrafo $G' = (V', E')$ com $V' = \{v \in V : v.\pi \neq NIL\} \cup \{r\}$ e $E' = \{(v.\pi, v) : v \in V' - \{r\}\}$ é uma árvore geradora mínima de G.

Prova. O algoritmo mantém a seguinte invariante: Antes de cada iteração do laço **while** (linhas 10-19), temos:

1. $A = \{(v, v.\pi) : v \in V - \{r\} - Q\}$;

2. Os vértices já colocados na árvore geradora mínima são os que estão em $V - Q$.

3. Para todos os vértices $v \in Q$, se $v.\pi \neq NIL$ então $v.key < \infty$ e $v.key$ é igual ao peso da aresta leve $(v, v.\pi)$ que conecta v a algum vértice que já faz parte da árvore geradora mínima.

Exercícios

1. Seja (u, v) uma aresta de peso mínimo em um grafo conexo G. Mostre que (u, v) pertence a alguma árvore geradora mínima de G.

2. Mostre que um grafo possui uma única árvore geradora mínima se, para todo corte do grafo, existe uma única aresta leve que cruza o corte. Mostre que a outra direção desta afirmação é falsa.

3. Construa uma implementação do algoritmo de Prim utilizando a representação de matriz de adjacências para o grafo G. Em seguida, faça a análise assintótica do algoritmo.

5.0.9 Caminhos mínimos

Nesta seção veremos como resolver o problema do caminho mínimo em grafos com pesos[6]. Agora cada aresta do grafo está associada a um número real, de forma que o peso de um caminho p em G é a soma dos pesos das arestas que compõem o caminho p. Formalmente, temos a seguinte definição:

Definição 7. *Considere um grafo $G = (V, E)$ com função peso $w : E \to \mathbb{R}$. O peso $w(p)$ de um caminho $p = \langle v_0, v_1, \ldots, v_k \rangle$ é a soma dos pesos das arestas que formam p:*

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

Os grafos sem peso podem ser vistos como um caso particular da definição acima. De fato, um grafo sem pesos $G = (V, E)$ pode ser visto como um grafo com pesos onde $w(e) = 1, \forall e \in E$, e apesar de termos utilizado o algoritmo BFS para computar os caminhos de comprimento mínimo (menor número de arestas) em grafos sem peso, ele não computa corretamente os caminhos mínimos em grafos com pesos (por que?). Assim, o contexto adequado para estudarmos o problema dos caminhos mínimos é com digrafos com pesos. Apesar disto, grafos com pesos também poderão ser considerados, bastando para isto trocarmos cada aresta não dirigidas (u, v) com peso $w(u, v)$ pelas arestas dirigidas (u, v) e (v, u), ambas com peso $w(u, v)$. Definimos o peso do caminho mínimo de u para v no digrafo G, denotado por $\delta(u, v)$, como a seguir:

Definição 8. *Sejam $G = (V, E)$ um digrafo com função peso $w : E \to \mathbb{R}$, e $u, v \in V$. O* caminho mínimo *do vértice u para o vértice v é definido como sendo qualquer caminho p com peso $w(p) = \delta(u, v)$.*

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

Estudaremos os caminhos mínimos a partir de um dado vértice (uma fonte), isto é, dados um digrafo $G = (V, E)$ e um vértice $s \in V$, queremos encontrar o caminho mínimo a partir de s para cada vértice $v \in V$ de G.

Subestrutura ótima do problema do caminho mínimo

Algoritmos para caminhos mínimos normalmente se baseiam na propriedade de que um caminho mínimo entre dois vértices são formados a partir de outros caminhos mínimos:

Lema 9. *Dado um digrafo $G = (V, E)$ com função peso $w : E \to \mathbb{R}$, seja $p = \langle v_0, v_1, \ldots, v_k \rangle$ um caminho mínimo do vértice v_0 para o vértice v_k, e para todo i e j tais que $0 \leq i \leq j \leq k$, seja $p_{ij} = \langle v_i, v_{i+1}, \ldots, v_j \rangle$ o subcaminho de p do vértice v_i para o vértice v_j. Então p_{ij} é um caminho mínimo de v_i para v_j.*

Demonstração. Prova por contradição.

□

Se um digrafo possui um ciclo com peso negativo então o caminho mínimo entre dois vértice que contenham este ciclo não está bem definido. De fato, sempre podemos diminuir o peso do caminho dando um nova volta pelo ciclo de peso negativo. Assim, se existe um ciclo de peso negativo entre os

vértice s e v, definimos $\delta(s, v) = -\infty$.

Os caminhos mínimos a partir da fonte s para qualquer vértice alcançável a partir de s no digrafo G serão representados pela árvore de caminho mínimo $G' = (V', E')$, onde $V' \subseteq V$ e $E' \subseteq E$ tal que:

1. V' é o conjunto de vértices alcançáveis a partir de s em G;

2. G' é uma árvore com raiz s, e;

3. Para todo $v \in V'$, o único caminho simples de s para v em G' é um caminho mínimo de s para v em G.

Utilizaremos o subgrafo predecessor $G_\pi = (V_\pi, E_\pi)$, onde

- $V_\pi = \{v \in V : v.\pi \neq NIL\} \cup \{s\}$ e;

- $E_\pi = \{(v.\pi, v) \in E : v \in V_\pi - \{s\}\}$, e mostraremos que G_π coincide com a árvore de caminho mínimo de G.

É importante observar que caminhos mínimos não são necessariamente únicos, assim como as árvores de caminho mínimo. Os algoritmos que veremos a seguir utilizam a técnica conhecida como *relaxamento de arestas*. Para cada vértice $v \in V$, manteremos um atributo $v.d$, que é uma cota superior para o peso do caminho mínimo da fonte s para v. O processo de relaxar uma aresta (u, v) consiste em testar se é possível diminuir o peso do caminho para v encontrado até o momento via o vértice u; se for possível, atualizamos $v.d$ e $v.\pi$. O código a seguir faz o relaxamento da aresta (u, v) em tempo constante:

```
1 if v.d > u.d + w(u, v) then
2 |   v.d = u.d + w(u, v);
3 |   v.π = u;
4 end
```

Algoritmo 25: Relax(u, v, w)

O lema a seguir é conhecido como *propriedade da convergência*:

Lema 10. *Sejam $G = (V, E)$ um digrafo com função peso w, $s \in V$, e $s \rightsquigarrow u \to v$ um caminho mínimo para os vértices u e v. Suponha que G tenha sido inicializado com Initialize-Single-Source:*

```
1 for each vertex v ∈ G.V do
2 |   v.d = ∞;
3 |   v.π = NIL;
4 end
5 s.d = 0;
```

Algoritmo 26: Initialize-Single-Source(G, s)

e então considere uma sequência de passos de relaxamento que incluem a chamada Relax(u, v, w). Se $u.d = \delta(s, u)$ em qualquer momento antes desta chamada então $v.d = \delta(s, v)$ em qualquer momento após esta chamada.

O lema a seguir é conhecido como *propriedade do relaxamento do caminho*:

Lema 11. *Se $p = \langle v_0, v_1, \ldots, v_k \rangle$ é um caminho mínimo de $s = v_0$ para v_k, e se relaxamos as arestas de p na ordem $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$, então $v_k.d = \delta(s, v_k)$.*

Demonstração. Indução no comprimento k do caminho p.

□

Exercício 3. *Prove que todo caminho mínimo é simples.*

O algoritmo de Dijkstra

Edsger W. Dijkstra nasceu em 1930 em Roterdã (Holanda). Filho de cientistas, seu pai era químico, e sua mãe, matemática. Estudou Física Teórica na Universidade de Leiden, e em 1952 começou a trabalhar no Centro de Matemática de Amsterdã, onde progressivamente foi se envolvendo com Computação. Em 1956 foi convidado para demostrar o poder do computador ARMAC, que ocupava uma sala inteira do Centro de Matemática, e começou a pensar no problema de determinar o menor caminho entre duas cidades em um mapa. Conta-se que na manhã de um domingo ensolarado enquanto tomava um café encontrou a solução do problema: a ideia básica é ir construindo um conjunto de cidades, digamos X, a partir da origem s até atingirmos o destino u. Em qualquer momento da execução do algoritmo é possível saber a distância mínima de s para qualquer cidade em X, que inicialmente contém apenas s. Em cada passo subsequente é possível encontrar uma cidade fora do conjunto X, digamos v, com a propriedade que a distância de s para v é menor do que a distância de s para qualquer outra cidade fora do conjunto X. Como a distância entre duas cidades quaisquer é sempre não-negativa, temos que v deve estar ligada diretamente com alguma cidade em X, digamos w. Assim, a distância mínima de s para v é dada pela distância mínima de s para w adicionada da distância de w para v. Neste momento, adicionamos v ao conjunto X, e repetimos o processo que vai parar quando u for adicionado ao conjunto X.

Seja $G = (V, E)$ um digrafo com função peso w, e tal que o peso de cada aresta seja não-negativo. Ou seja, assumimos que $w(u, v) \geq 0$, para toda aresta $(u, v) \in E$.

```
1  Initialize-Single-Source(G, s);
2  S = ∅;
3  Q = G.V;
4  while Q ≠ ∅ do
5  |   u = Extract-Min(Q);
6  |   S = S ∪ {u};
7  |   for each vertex v ∈ G.Adj[u] do
8  |   |   Relax(u, v, w)
9  |   end
10 end
```

Algoritmo 27: Dijkstra(G, w, s)

Observe que Dijkstra é um algoritmo guloso. Sabemos que a estratégia gulosa nem sempre resulta em uma solução ótima, mas o teorema a seguir mostra que Dijkstra(G, w, s) computa corretamente os caminhos mínimos a partir do vértice s:

Teorema 12. *O algoritmo de Dijkstra, ao ser executado em um digrafo $G = (V, E)$ com função peso não-negativa w e fonte s, termina com $u.d = \delta(s, u)$ para todo $u \in V$.*

Exercício 4. *A prova do teorema acima é baseada na seguinte invariante:*

> *Antes de cada iteração do laço **while** (linhas 4-10), $v.d = \delta(s, v)$ para todo $v \in S$.*

Complete a prova deste teorema.

Exercício 5. *Faça a análise assintótica do algoritmo de Dijkstra.*

5.0.10 Leituras complementares

1. [6] (Capítulos 22 e 23)

2. [14] (Capítulo 9)

Capítulo 6

Programação Dinâmica

A metodologia conhecida como programação dinâmica foi inventada pelo matemático americano Richard Bellman por volta de 1950 como um método genérico para otimizar processos de decisão. Assim, a palavra programação está mais relacionada com a ideia de planejamento, e não com programação de computadores. Depois de se estabelecer como uma importante técnica em Matemática Aplicada, a programação dinâmica passou a ser utilizada como uma estratégia de 'dividir e conquistar' juntamente com uma tabela [14], pois ao invés de resolver os subproblemas recursivamente, os mesmos são resolvidos sequencialmente e as soluções são armazenadas em uma tabela. Desta forma, esta metodologia é utilizada para resolver problemas subdividindo-os em subproblemas como na estratégia de dividir e conquistar, mas com uma diferença fundamental: os subproblemas se sobrepõem, e para evitar que o mesmo subproblema seja calculado mais de uma vez, os resultados são armazenados em uma tabela.

6.0.1 Números de Fibonacci

Considere o problema de computar o n-ésimo número de Fibonacci:

$$F_n = \begin{cases} 0, & \text{se } n = 0 \\ 1, & \text{se } n = 1 \\ F_{n-1} + F_{n-2}, & \text{se } n > 1 \end{cases} \tag{6.1}$$

Complexidade exponencial

Uma implementação direta da definição acima nos permite analisar a complexidade $T(n)$ necessária para computar o n-ésimo número de Fibonacci pela seguinte equação de recorrência:

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

que tem solução exponencial, i.e. $T(n) = O(2^n)$. De fato, $\forall n > 1, T(n) = T(n-1) + T(n-2) \Rightarrow 2\,T(n-2) \leq T(n) \leq 2\,T(n-1)$. De acordo com a paridade de n, temos os seguintes casos:

1. n é par: $\sqrt{2}^n/2 = 2^{n/2-1} = 2^{n/2-1}T(2) \leq T(n) \leq 2^{n-1}T(1) = 2^{n-1}$

2. n é ímpar: $\sqrt{2}^{n-1} = 2^{(n-1)/2}T(1) \leq T(n) \leq 2^{n-1}T(1) = 2^{n-1}$

Consequentemente, $T(n) = O(2^n)$ e $T(n) = \Omega(\sqrt{2}^n)$, ou seja, $T(n)$ é limitada tanto inferiormente quanto superiormente por funções de classe de complexidade exponencial.

Mais precisamente, temos que $T(n) = \Theta(\phi^n)$, onde $\phi = \frac{1+\sqrt{5}}{2}$. Este resultado é coerente com os limites anteriores pois $\sqrt{2} < \phi = \frac{1+\sqrt{5}}{2} < 2$.

Complexidade linear

Note que o problema neste caso é que cada subproblema foi calculado diversas vezes. A ideia para resolver este problema de forma mais eficiente é evitar refazer computações já feitas anteriormente preenchendo a entrada f[i] do vetor f com o i-ésimo número de Fibonacci:

```
1 f[i] = i, if i < 2;
2 for i = 2 to n do
3 |  f[i] = f[i − 1] + f[i − 2]
4 end
5 return f[n]
```

Algoritmo 28: fib(n)

No pseudocódigo acima, a linha 1 é executada em tempo constante, o **for** das linhas 2-4 é executado $n - 1$ vezes, de forma que o tempo de execução deste algoritmo é linear!

Observações

Os problemas que podem ser resolvidos usando programação dinâmica normalmente estão relacionados com *otimização*. No exemplo anterior, encontramos uma forma de *minimizar* o número de somas necessárias para calcular fib(n). Tipicamente, a tabela vai conter os valores correspondentes a todas as chamadas recursivas de forma que o algoritmo principal não precisará fazer nenhuma chamada recursiva já que as mesmas já foram resolvidas e armazenadas na tabela. Por exemplo, no pseudocódigo acima, no momento de calcular f[k], os valores f[$k-1$] e f[$k-2$] já foram computados.

Para que esta metodologia possa ser aplicada é importante que o problema a ser resolvido satisfaça o *princípio da subestrutura ótima*: as soluções ótimas do problema contêm as soluções ótimas dos subproblemas. Programação Dinâmica pode ser vista como uma troca entre espaço e tempo, onde o tempo de execução é reduzido ao custo de espaço extra.

6.0.2 O problema do corte das hastes

Consideremos um outro problema de otimização: o problema do corte das hastes [6]. Suponha que uma empresa deseja cortar hastes de forma a maximizar o valor total obtido pela venda dos pedaços cortados. Assumiremos os seguintes fatos:

1. O corte não tem custo;

2. As hastes são cortadas em pedaços cujos comprimentos são números inteiros.

3. O preço de uma haste de comprimento $i \geq 1$ é igual a p_i.

O problema do corte das hastes pode, então, ser apresentado da seguinte forma: Dados uma haste de comprimento n e uma tabela de preços $P = \langle p_1, p_2, \ldots, p_n \rangle$ para a haste de comprimento $1 \leq i \leq n$, determine a melhor forma de cortar a haste de comprimento n de forma a obter o valor máximo da venda dos pedaços resultantes do corte.

De quantas formas distintas podemos cortar uma haste de comprimento n?

Recorrência

Note que temos $n-1$ possíveis pontos de corte em uma haste de comprimento n.

Suponha que uma solução ótima divide a haste em $1 \leq k \leq n$ pedaços. Assim, $n = i_1 + i_2 + \ldots + i_k$, onde i_j denota o comprimento do j-ésimo pedaço da haste. O valor a ser obtido a partir da venda destes k pedaços é $v(n) = p_{i_1} + p_{i_2} + \ldots + p_{i_k}$. Nosso objetivo é determinar k de forma que $v(n)$ seja máximo. A recursão que corresponde ao valor máximo a ser obtido é dada por:

$$v(n) = \max\{p_n, v(1) + v(n-1), v(2) + v(n-2), \ldots, v(n-1) + v(1)\}$$

Podemos simplificar esta recursão observando que a divisão da haste consiste em fazer o primeiro corte obtendo um pedaço de comprimento i que não será mais dividido, e um segundo pedaço de comprimento $n-i$ que ainda será dividido de forma a maximizar o valor a ser obtido:

$$v(n) = \max_{1 \leq i \leq n} \{p_i + v(n-i)\} \tag{6.2}$$

Pseudocódigo força bruta

O pseudocódigo a seguir implementa a computação correspondente à recursão.

```
1 if n = 0 then
2 |  return 0;
3 end
4 q ← −∞;
5 for i = 1 to n do
6 |  q ← max{q, p[i]+ corte-haste(p, n − i)};
7 end
8 return q;
```

Algoritmo 29: corte-haste(p, n)

O tempo $T(n)$ de execução deste algoritmo é dado por

$$T(n) = \begin{cases} 1, & n = 0 \\ 1 + \sum_{j=0}^{n-1} T(j), & n > 0 \end{cases}$$

que tem solução exponencial.

Isto não é surpreendente porque o algoritmo corte-haste considera todas as 2^{n-1} possíveis formas de cortar uma haste de comprimento n.

Pseudocódigo (programação dinâmica)

Utilizando programação dinâmica, cada subproblema será resolvido apenas uma vez:

```
1  let r[0..n] be a new array;
2  r[0] = 0;
3  for j = 1 to n do
4  |  q = −∞;
5  |  for i = 1 to j do
6  |  |  q = max{q, p[i] + r[j − i]};
7  |  end
8  |  r[j] = q;
9  end
10 return r[n];
```

Algoritmo 30: corte-hasteDP(p, n)

O tempo $T(n)$ de execução deste algoritmo é determinado pelo número de vezes que o **for** das linhas 5-7 é executado:

$$T(n) = \sum_{j=1}^{n} \sum_{i=1}^{j} 1 = \sum_{j=1}^{n} j = \frac{n.(n+1)}{2}$$

e portanto o algoritmo é quadrático no tamanho da entrada.

Observe que o pseudocódigo acima retorna o valor máximo que pode ser obtido com a venda dos pedaços da haste, mas não nos diz como decompor a haste. O pseudocódigo a seguir, além de retornar o valor máximo $r[n]$, também retorna o vetor $s[0..n]$ contendo

6.0.3 Multiplicação de uma cadeia de matrizes.

Queremos computar o produto $A_1 \cdot A_2 \cdot \ldots \cdot A_n$ de forma a executar o menor número possível de multiplicações.

Multiplicação de matrizes

Utilizaremos o algoritmo padrão para multiplicação de duas matrizes:

```
1  if A.columns ≠ B.rows then
2  |  error "incompatible dimensions"
3  end
4  else
5  |  let C be a new A.rows × B.columns matrix;
6  |  for i = 1 to A.rows do
7  |  |  for j = 1 to B.columns do
8  |  |  |  c_ij = 0;
9  |  |  |  for k = 1 to A.columns do
10 |  |  |  |  c_ij = c_ij + a_ik.b_kj
11 |  |  |  end
12 |  |  end
13 |  end
14 |  return C
15 end
```

Algoritmo 31: mult-matrix(A, B)

O número exato de multiplicações $T(n)$ realizadas pelo algoritmo acima corresponde ao número de vezes que a linha 10 é executada: Suponha que as dimensões das matrizes A e B sejam, respectivamente iguais a $p \times q$ e $q \times r$, ou seja, a matriz A possui p linhas e q colunas, enquanto que a matriz B possui q

linhas e r colunas. Então o total de multiplicações é dado por:

$$\sum_{i=1}^{p}\sum_{j=1}^{r}\sum_{k=1}^{q} 1 = \sum_{i=1}^{p}\sum_{j=1}^{r} q = \sum_{i=1}^{p}(r.q) = p.q.r$$

Em particular, se $n = p = q = r$ então $T(n) = n^3$.

Exemplo

Assim, para multiplicarmos duas matrizes de dimensões respectivamente iguais a 10×4 e 4×20, realizaremos $10.4.20 = 800$ multiplicações. Suponha que estejamos interessados em multiplicar as matrizes A, B e C (nesta ordem) de dimensões respectivamente iguais a 10×4, 4×20 e 20×32. Sabendo que o produto de matrizes é associativo, o produto $A.B.C$ pode ser realizado de duas formas distintas, a saber: $(A.B).C$ ou $A.(B.C)$. O número de multiplicações necessárias para computar o produto $(A.B).C$ utilizando o algoritmo acima é igual ao número de multiplicações para computar o produto $A.B$, que já sabemos ser igual a 800, mais o número de multiplicações necessárias para computar o produto de $A.B$ com C, que é igual a $10.20.32 = 6400$; ou seja, no total precisamos de $800 + 6400 = 7200$ multiplicações para computar o produto $(A.B).C$. Para computar o produto $A.(B.C)$ precisamos de $4.20.32 = 2560$ multiplicações para computar $B.C$, mais $10.4.32 = 1280$ multiplicações para computar o produto de A com $B.C$ perfazendo um total de $2560 + 1280 = 3840$ multiplicações. Portanto, é mais "econômico" multiplicar $A.(B.C)$ do que $(A.B).C$. Este exemplo, nos leva a um problema mais geral que tentaremos resolver:

Complexidade do método de força bruta

Suponha que queiramos multiplicar n matrizes $A_1, A_2, \ldots, A_n$ $(n \geq 1)$. Isto é, queremos computar o produto $A_1.A_2.\ldots.A_n$. Como fazer isto de forma a realizar o menor número de multiplicações possível? Em outras palavras, como devemos associar o produto $A_1.A_2.\ldots.A_n$ de forma a minimizar o número de multiplicações a serem realizadas?

Para motivarmos a utilização de programação dinâmica para resolver este problema, vejamos que a abordagem ingênua (força-bruta) não é eficiente. A abordagem ingênua consiste em escolher a melhor dentre todas as associações possíveis para o produto $A_1, A_2, \ldots, A_n$. Seja $P(n)$ o número total de distintas formas de associar o produto em consideração. Quando $n = 1$, temos apenas uma matriz, e portanto $P(1) = 1$. Quando $n > 1$ então para cada $1 \leq k \leq n-1$ temos $P(k)$ formas de associar o produto $A_1.A_2.\ldots.A_k$, e $P(n-k)$ formas de associar o produto $A_{k+1}.A_{k+2}.\ldots.A_n$. Desta forma, o número total de distintas formas de associar o produto $A_1, A_2, \ldots, A_n$ é dado pela recorrência:

$$P(n) = \begin{cases} 1, & \text{se } n = 1 \\ \sum_{k=1}^{n-1} P(k).P(n-k), & \text{se } n > 1 \end{cases} \tag{6.3}$$

É possível mostrar que $P(n) = \Omega(2^n)$, e portanto a solução ingênua (força bruta) é exponencial.

Solução via Programação Dinâmica

1. Subestrutura ótima

   Antes de construirmos uma solução usando programação dinâmica, vejamos que este problema satisfaz o princípio da subestrutura ótima. Denote o produto $A_i.A_{i+1}\ldots A_j$, onde $i \leq j$ por $A_{i..j}$. Agora suponha que uma associação ótima para $A_{i..j}$ separa o produto entre A_k e A_{k+1} para algum

$i \leq k \leq j$, *i.e.* a associação ótima será dada pela associação ótima de $A_{i..k}$ e $A_{k+1..j}$. Em seguida, uma solução será recursivamente construída para o produto $A_{i..k}$ (e para $A_{k+1..j}$).

Questão: Será que a parentização construída para $A_{i..k}$ na parentização de $A_{i..j}$ é uma solução ótima para $A_{i..k}$? Sim, pois uma possível solução mais eficiente para $A_{i..k}$ nos permitiria substituí-la na solução de $A_{i..j}$ produzindo uma solução melhor do que a ótima, o que é uma contradição.

2. Recorrência

   Se denotarmos por $m[i, j]$ o número mínimo de multiplicações necessárias para computar o produto $A_{i..j}$ $(i \leq j)$, então podemos caracterizar o problema de determinar o número mínimo de multiplicações para computar o produto $A_{i..j}$ através da recursão

   $m[i, j] = m[i, k] + m[k + 1, j] + p_{i-1}.p_k.p_j$ para algum $i \leq k \leq j$.

   mas qual é o valor de k que devemos tomar? Precisamos considerar todos os valores possíveis para descobrirmos qual minimiza o número de multiplicações:

$$m[i,j] = \begin{cases} 0, & \text{se } i = j \\ \min\limits_{i \leq k \leq j} \{m[i,k] + m[k+1,j] + p_{i-1}.p_k.p_j\}, & \text{se } i < j \end{cases} \tag{6.4}$$

   Observe que para calcular $m[i, j]$ precisamos conhecer os valores $m[i, k]$ e $m[k + 1, j]$ para todo $i \leq k \leq j - 1$, ou seja, precisamos conhecer todos os pares de valores $m[i, i]$ e $m[i + 1, j]$, $m[i, i + 1]$ e $m[i + 2, j]$, ..., $m[i, j - 1]$ e $m[j, j]$.

   Assim, precisamos considerar todos os valores possíveis de $1 \leq k \leq n$, mas em que ordem devemos preencher a matriz m? A partir da diagonal principal da matriz m, uma vez que $m[i, i] = 0$ para todo $1 \leq i \leq n$.

$$m[1,n] = \min_{1 \leq k \leq n} \{m[1,k] + m[k+1,n] + p_0.p_k.p_n\} \tag{6.5}$$

   onde a dimensão da matriz A_i $(1 \leq i \leq n)$ é igual a $p_{i-1} \times p_i$.

3. Pseudocódigo

   O pseudocódigo a seguir recebe como argumento o vetor $p = \langle p_0, p_1, \ldots, p_n \rangle$ contendo as dimensões das matrizes

   $(A_1)_{p_0 \times p_1}, (A_2)_{p_1 \times p_2}, \ldots, (A_n)_{p_{n-1} \times p_n}$,

   e retorna as matrizes $m[1..n, 1..n]$ e $s[1..n - 1, 2..n]$, onde $m[i, j]$ corresponde ao número mínimo de multiplicações necessárias para computar o produto $A_{i..j}$, e $s[i, j]$ contém o índice k que indica como o produto $A_{i..j}$ deve ser separado:

```
1  n ← p.length − 1;
2  let m[1..n, 1..n] and s[1..n − 1, 2..n] be new tables;
3  for i = 1 to n do
4  |  m[i, i] ← 0;
5  end
6  for l = 2 to n do
7  |  for i = 1 to n − l + 1 do
8  |  |  j ← i + l − 1;
9  |  |  m[i, j] ← ∞;
10 |  |  for k = i to j − 1 do
11 |  |  |  q ← m[i, k] + m[k + 1, j] + p_{i−1}.p_k.p_j;
12 |  |  |  if q < m[i, j] then
13 |  |  |  |  m[i, j] ← q;
14 |  |  |  |  s[i, j] ← k;
15 |  |  |  end
16 |  |  end
17 |  end
18 end
```

Algoritmo 32: matrix-chain-order(p)

(a) Parentização ótima

```
1  if i == j then
2  |  print A_i;
3  end
4  else
5  |  print "(";
6  |  print-optimal-parens(s, i, s[i, j]);
7  |  print-optimal-parens(s, s[i, j] + 1, j);
8  |  print ")";
9  end
```

Algoritmo 33: print-optimal-parens(s, i, j)

A complexidade de tempo de matrix-chain-order é $\Theta(n^3)$ e complexidade de espaço $\Theta(n^2)$.

6.0.4 Exercícios

1. Use o método da substituição para mostrar que a recorrência

$$P(n) = \begin{cases} 1, & \text{se } n = 1 \\ \sum_{k=1}^{n-1} P(k).P(n-k), & \text{se } n > 1 \end{cases} \tag{6.6}$$

é $\Omega(2^n)$.

2. Mostre que a recorrência

$$T(n) = \begin{cases} 1, & n = 0 \\ 1 + \sum_{j=0}^{n-1} T(j), & n > 0 \end{cases}$$

tem solução $T(n) = 2^n$.

3. Considere o seguinte problema: Há uma fila de n moedas cujos valores são alguns inteiros positivos $c_1, c_2, ..., c_n$, não necessariamente distintos. O objetivo é pegar a quantidade máxima de dinheiro sujeita à restrição de que não se pode pegar duas moedas adjacentes na fila inicial.

   (a) Construa uma recorrência para calcular o montante máximo $F(n)$ que pode ser obtido de uma fila com n moedas.

   Solução.

   $$F(n) = \begin{cases} 0, & \text{se } n = 0 \\ c_1, & \text{se } n = 1 \\ \max(c_n + F(n-2), F(n-1)), & \text{se } n > 1 \end{cases}$$

   (b) Qual a complexidade da abordagem de força bruta para este problema?

   Solução.
   A solução é exponencial. Para isto basta notar a semelhança da recorrência acima com a função de Fibonacci.

   (c) Construa uma solução utilizando programação dinâmica e faça a análise assintótica da sua solução.

   Solução.
   A ideia é ir armazenando as soluções dos problemas menores no vetor F (abordagem *bottom up*) de forma que o mesmo subproblema não será computado mais de uma vez. A entrada do algoritmo é o vetor C contendo os valores $c_1, c_2, \ldots, c_n$ das moedas.

   ```
   1 F[0] ← 0;
   2 F[1] ← C[1];
   3 for i = 2 to n do
   4 |  F[i] ← max(C[i] + F[i − 2], F[i − 1]);
   5 end
   6 return F[n];
   ```

   O algoritmo é linear, isto é, $\Theta(n)$.

4. Construa um algoritmo eficiente para calcular o coeficiente binomial $C(n, k)$, também denotado por $\binom{n}{k}$, sem utilizar multiplicações. Em seguida, faça a análise assintótica do seu algoritmo.

   Solução.

   ```
   1  for i = 0 to n do
   2  |  for j = 0 to min(i, k) do
   3  |  |  if j = 0 or j = i then
   4  |  |  |  C[i, j] ← 1;
   5  |  |  end
   6  |  |  else
   7  |  |  |  C[i, j] ← C[i − 1, j − 1] + C[i − 1, j];
   8  |  |  end
   9  |  end
   10 end
   11 return C[n, k];
   ```

   Algoritmo 34: Binomial(n, k)

$$T(n) = \sum_{i=0}^{n}(\sum_{j=0}^{\max(i,k)} 1) \leq \sum_{i=0}^{n}(\sum_{j=0}^{k} 1) = \sum_{i=0}^{n}(k+1) = (n+1)(k+1) = O(n.k).$$

6.0.5 Leitura complementar:

- [7] (Capítulo 14)
- [6] (Capítulo 15)
- [14] (Capítulo 14)

Referências Bibliográficas

[1] Emilio Jesús Gallego Arias, Benoît Pin, and Pierre Jouvelot. jsCoq: Towards Hybrid Theorem Proving Interfaces. *Electronic Proceedings in Theoretical Computer Science*, 239:15–27, January 2017.

[2] Jeremy Avigad, Kevin Donnelly, David Gray, and Paul Raff. A formally verified proof of the prime number theorem. *ACM Transactions on Computational Logic*, 9(1):2–es, December 2007.

[3] Jeremy Avigad and John Harrison. Formally verified mathematics. *Communications of the ACM*, 57(4):66–75, April 2014.

[4] Gilles Brassard and Paul Bratley. *Fundamentals of Algorithmics*. Prentice-Hall, Inc., USA, 1996.

[5] A. Chlipala. *Certified Programming with Dependent Types*. MIT Press, 2017.

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[7] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, Cambridge, MA, USA, 4 edition, April 2022.

[8] Leonardo de Moura and Sebastian Ullrich. The Lean 4 Theorem Prover and Programming Language. In André Platzer and Geoff Sutcliffe, editors, *Automated Deduction – CADE 28*, Lecture Notes in Computer Science, pages 625–635, Cham, 2021. Springer International Publishing.

[9] G. Gonthier. A computer-checked proof of the Four Colour Theorem. Technical report, Microsoft Research Cambridge, 2008.

[10] Thomas Hales, Mark Adams, Gertrud Bauer, Dat Tat Dang, John Harrison, Truong Le Hoang, Cezary Kaliszyk, Victor Magron, Sean McLaughlin, Thang Tat Nguyen, Truong Quang Nguyen, Tobias Nipkow, Steven Obua, Joseph Pleso, Jason Rute, Alexey Solovyev, An Hoai Thi Ta, Trung Nam Tran, Diep Thi Trieu, Josef Urban, Ky Khac Vu, and Roland Zumkeller. A formal proof of the Kepler conjecture, January 2015.

[11] M. Huth and M. Ryan. *Logic in Computer Science: Modelling and Reasoning About Systems*. Cambridge University Press, New York, NY, USA, 2004.

[12] Cezary Kaliszyk, Stephan Schulz, Josef Urban, and Jiří Vyskočil. System Description: E.T. 0.1. In Amy P. Felty and Aart Middeldorp, editors, *Automated Deduction - CADE-25*, volume 9195, pages 389–398. Springer International Publishing, Cham, 2015.

[13] Xavier Leroy. Formal Verification of a Realistic Compiler. *Communications of the ACM*, 52(7):107, 2009.

[14] A. V. Levitin. *Introduction to the Design and Analysis of Algorithms, Third Edition*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2012.

[15] W. McCune. Prover9 and mace4. http://www.cs.unm.edu/~mccune/prover9/, 2005–2010.

[16] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lncs*. Springer, 2002.

[17] R. B. Nogueira, A. C. A. Nascimento, F. L. C. de Moura, and M. Ayala-Rincón. Formalization of Security Proofs Using PVS in the Dolev-Yao Model. In *Booklet Proc. Computability in Europe - CiE*, 2010.

[18] S. Owre, J. M. Rushby, and N. Shankar. PVS: A Prototype Verification System. In D. Kapur, editor, *CADE*, volume 607 of *Lnai*, pages 748–752. sv, 1992.

[19] C. Paulin-Mohring. Introduction to the Calculus of Inductive Constructions. 2014.

[20] Lawrence C. Paulson. A Mechanised Proof of Gödel's Incompleteness Theorems Using Nominal Isabelle. *J Autom Reasoning*, 55(1):1–37, 2015.

[21] Benjamin C. Pierce, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Catvalin Hriatcu, Vilhelm Sjoberg, and Brent Yorgey. *Software Foundations*. Electronic textbook, 2014.

[22] Alexandre Riazanov and Andrei Voronkov. The design and implementation of VAMPIRE. *AI Commun.*, 15(2-3):91–110, 2002.

[23] Michael Sipser. *Introduction to the Theory of Computation*. International Thomson Publishing, 1st edition, 1996.

[24] Raymond Smullyan. *Logical Labyrinths*. AK Peters, 2009.

[25] The Coq Development Team. The Coq Proof Assistant. Zenodo, October 2021.

[26] D. van Dalen. *Logic and Structure*. Universitext. Springer London, 2013.

Capítulo 7

NP-completude

Praticamente todos os algoritmos estudados até aqui têm complexidade polinomial, i.e. o tempo de execução destes algoritmos no pior caso está em $O(p(n))$, onde $p(n)$ é um polinômio no tamanho da entrada n [7, 14]. Estes algoritmos formam a classe dos algoritmos que são considerados "eficientes". Os problemas que podem ser resolvidos por algoritmos polinomiais são chamados de "tratáveis". No entanto, vimos algoritmos cujos tempos de execução são exponenciais no tamanho da entrada, como por exemplo, a abordagem força bruta para calcular o n-ésimo número de Fibonacci, para descobrir a melhor forma de associar o produto de n matrizes, entre outros.

Neste capítulo estudaremos uma classe de problemas para os quais as técnicas estudadas até aqui não se aplicam. Estes problemas não têm soluções polinomiais conhecidas, mas também não existe uma prova de que tais soluções não sejam possíveis: esta é a famosa questão "P versus NP"que constitui um dos mais interessantes problemas em aberto na Computação. Adicionalmente, essa classe de problemas possui uma característica muito interessante: um algoritmo eficiente que resolva qualquer um dos problemas desta classe resulta de forma imediata em algoritmos eficientes para todos os problema da classe.

Definimos um problema como a seguir:

Definição 13. *Um problema (abstrato) Q é uma relação binária que associa um conjunto I de instâncias a um conjunto S de soluções.*

Por exemplo:

- O problema PATH é uma relação que associa cada instância de um digrafo e dois vértices com um caminho que contém os dois vértices.

- O problema SHORTEST-PATH é uma relação que associa cada instância de um digrafo e dois vértices com um caminho mínimo que contém os dois vértices. Como caminhos mínimos não são necessariamente únicos, uma instância de um problema pode ter mais de uma solução.

Neste capítulo trabalharemos essencialmente com *problemas de decisão*, que são problemas cujas respostas são sempre *sim* ou *não*:

Definição 14. *Um problema de decisão é uma relação binária sobre um conjunto I de instâncias e um conjunto binário (sim ou não) de soluções.*

Assim, podemos ver um problema de decisão como sendo uma função que associa instâncias em I ao conjunto de soluções $\{0, 1\}$.

- O problema PATH pode ser visto como um problema de decisão: Dados um digrafo G, e vértices u e v de G, determinar se existe um caminho de u para v em G.

Daqui em diante, trataremos apenas de problemas de decisão, que chamaremos apenas de problemas.

7.1 A classe P

A classe P consiste dos problemas que podem ser resolvidos em tempo polinomial por algoritmos determinísticos. A seguir listamos alguns exemplos, mas poderíamos incluir diversos outros problemas estudados anteriormente.

Exemplo 15. *Problema do caminho em grafos: PATH está em P. De fato, considere o seguinte algoritmo que recebe como entrada um digrafo G e vértices u e v.*

1. *Marque o vértice u*
2. *Enquanto existir aresta $(a, b) \in G$ com a marcado, e b não-marcado, marque b.*
3. *Se v está marcado retorne 1, caso contrário retorne 0.*

***Análise do algoritmo**: O laço da linha 2 pode ser executado segundo o algoritmo de busca em largura, por exemplo, que é linear no tamanho da representação do grafo G. As outras linhas do algoritmo são executadas apenas uma vez, portanto o algoritmo é polinomial.*

Exemplo 16. *Problema da multiplicação de matrizes: Dadas matrizes A, B e C, verificar se $A \times B = C$. O algoritmo de multiplicação padrão ou o algoritmo de Strassen resolvem este problema em tempo polinomial.*

Exemplo 17. *Problema da multiplicação de cadeia de matrizes: Dada uma sequência de matrizes $A_1, A_2, \ldots, A_n$ e um inteiro k, verificar se existe uma parentização que realiza, no máximo, k multiplicações escalares. Este problema pode ser resolvido em tempo polinomial utilizando programação dinâmica.*

Exemplo 18. *Problema da ordenação: Dada uma lista l, verificar se existe uma permutação de l que seja ordenada. Este problema pode ser resolvido em tempo polinomial utilizando algum algoritmo de ordenação, como* insertion sort*,* merge sort*,* heapsort*,* quicksort*, etc.*

7.1.1 Redução polinomial

A *redutibilidade* entre dois problemas é uma técnica de projeto de algoritmos muito utilizada para dar informações sobre a dificuldade de problemas. Por exemplo, considere um problema A que queremos

resolver, e suponha que sabemos como resolver um outro problema B. Se for possível transformar instâncias do problema A em instâncias do problema B com as seguintes características:

- A transformação é feita em tempo polinomial;
- As respostas são as mesmas para ambas as instâncias.

Então chamamos este procedimento de *algoritmo de redução*, e o mesmo nos fornece uma forma de resolver o problema A a partir do problema B:

- Dada uma instância α do problema A, usamos o algoritmo de redução para transformá-la em uma instância β do problema B;
- Executamos o algoritmo polinomial para a instância β de B;
- Usamos a resposta de β como resposta de α.

Ou seja, podemos assim construir um algoritmo para o problema A. O que podemos concluir a partir da existência de um tal algoritmo de redução? A existência de um algoritmo eficiente para B nos permite construir um algoritmo eficiente para A, mas também que a não existência de um algoritmo eficiente para A implica na não existência de um algoritmo eficiente para B. Em outras palavras, o problema B é tão difícil quanto o problema A, ou ainda, A não é mais difícil do que B.

Definição 19. *Dizemos que um problema L_1 é redutível polinomialmente ao problema L_2, notação $L_1 \leq_p L_2$, se existe uma função computável em tempo polinomial f que transforma instâncias do problema L_1 em instâncias de L_2 de forma que $x \in L_1$ se, e somente se, $f(x) \in L_2$. A função f é chamada de* função de redução*, e o algoritmo polinomial F que computa a função f é chamado de* algoritmo de redução*.*

Observe que ao reduzirmos uma linguagem (ou problema) L_1 para outra linguagem (ou problema) L_2, queremos que cada instância de L_1 seja transformada em uma instância de L_2, isto é, se $x \in L_1$ então $f(x) \in L_2$.

Adicionalmente, precisamos que elementos que não sejam instância de L_1 não sejam levados em L_2: $x \notin L_1$ então $f(x) \notin L_2$, o que é equivalente por contraposição a se $f(x) \in L_2$ então $x \in L_1$. Juntando estas duas informações temos a equivalência "$x \in L_1$ se, e somente se, $f(x) \in L_2$"da definição acima.

Reduções polinomiais são uma ferramenta poderosa que nos permitem provar que outras linguagens estão em P:

Lema 20. *Sejam L_1 e L_2 problemas tais que $L_1 \leq_p L_2$, então $L_2 \in P$ implica que $L_1 \in P$.*

Demonstração. Seja A_2 um algoritmo polinomial que decide a linguagem L_2, e F um algoritmo de redução polinomial que computa a função de redução f. Construiremos um algoritmo A_1 que decide L_1 da seguinte forma: dado $x \in \{0,1\}^*$, o algoritmo A_1 inicialmente usa F para transformar x em $f(x)$, e então usa o algoritmo A_2 para responder. Note que A_1 é polinomial já que tanto A_2 quanto F são polinomiais.

□

Considere o problema 2-SAT, cujas instâncias são expressões lógicas formadas por conjunções de disjunções de dois literais, onde um literal é uma variável booleana ou a negação de uma variável booleana. Por exemplo, a expressão a seguir é uma instância de 2-SAT:

$$(x_1 \vee \neg x_2) \wedge (\neg x_1 \vee \neg x_3) \wedge (x_1 \vee x_2)$$

Uma solução da instância acima é uma designação de valores booleanos (0 ou 1) para as variáveis que satisfaçam a expressão, ou seja, que retornem 1. Por exemplo, a designação $x_1 = 1$, $x_2 = 1$ e $x_3 = 0$ satisfaz a expressão acima.

Exercício 6. *Mostre que 2-SAT $\in$ P.*

7.2 A classe NP

A classe NP consiste dos problemas que podem ser resolvidos em tempo polinomial por um algoritmo não-determinístico.

A ideia é que inicialmente, o algoritmo advinhe uma solução (fase não-determinística), e em seguida esta solução deve ser verificada em tempo polinomial deterministicamente.

Assim, a forma mais usual de apresentar a classe NP, consiste em considerar os problemas que podem ser verificados em tempo polinomial por um algoritmo determinístico [6, 23].

Como vimos, em alguns casos é possível evitar uma abordagem força bruta e encontrar soluções polinomiais para o problema em questão. Mas é fácil imaginar que isto nem sempre será possível.

De fato, veremos que existem diversos problemas interessantes/importantes para os quais soluções polinomiais não foram encontradas até hoje, mas que ainda é possível verificar em tempo polinomial, dado um certificado.

Exemplo 21. *O problema de encontrar ciclos Hamiltonianos em (di)grafos tem sido estudado por muito tempo (mais de 100 anos!). Formalmente, um ciclo Hamiltoniano de um grafo $G = (V, E)$ é um ciclo simples que contém cada vértice de V, i.e. cada vértice de G é visitado uma única vez. Um (di)grafo que contém um ciclo Hamiltoniano é dito Hamiltoniano.*

Denotaremos por HAM-CYCLE o problema de encontrar ciclos Hamiltonianos em (di)grafos.

HAM-CYCLE $= \{\langle G \rangle : G$ *é um (di)grafo Hamiltoniano* $\}$

Podemos verificar uma possível solução em tempo polinomial: Suponha que um colega te diz que um (di)grafo G é Hamiltoniano, e como justificativa, fornece uma sequência de vértices na ordem que ele diz formar um caminho Hamiltoniano.

1. *Verifique que os vértices dados constituem o conjunto V dos vértices de G;*

2. *Verifique que cada par de vértices consecutivos da sequência dada corresponde a uma aresta de G.*

Como a verificação acima pode ser feita em tempo polinomial, temos que HAM-CYCLE $\in NP$.

Exemplo 22. *Um clique em um grafo (não dirigido) é um subgrafo onde dois vértices quaisquer estão ligados por uma aresta. Um k-clique é um clique que contém k vértices. O problema CLIQUE consiste em determinar se um grafo contém um clique de um tamanho especificado:*

CLIQUE $= \{(G, k) : G$ *é um grafo com um k-clique*$\}$

***Afirmação**: CLIQUE* $\in NP$

O clique é o certificado. Para a entrada $((G, k), c)$

1. *Verifique se c é um subconjunto de $G.V$ de tamanho k;*

2. *Verifique se G contém todas as arestas que conectam vértices em c;*

3. *Se ambas as verificações podem ser feitas então retorne 1, caso contrário, retorne 0.*

Exemplo 23. ***Afirmação***. *3-SAT* $\leq_P$ *CLIQUE*

Prova. *Nos grafos a serem construídos, cliques de um tamanho específico correspondem a designações satisfatíveis da fórmula.*

Seja φ *uma fórmula com* k *cláusulas*

$$\varphi = (a_1 \vee b_1 \vee c_1) \wedge (a_2 \vee b_2 \vee c_2) \wedge \ldots (a_k \vee b_k \vee c_k)$$

A redução f *constrói a codificação* $\langle G, k \rangle$ *onde* G *é um grafo não dirigido dado por:*

- *Os vértices de* G *são organizados em* k *grupos de 3 vértices cada* $t_1, t_2, \ldots, t_k$. *Cada tripla* t_i *corresponde a uma das cláusulas de* φ, *e cada vértice na tripla corresponde a um literal da cláusula associada. Marque cada vértice de* G *com o literal correspondente em* φ. *As arestas de* G *conectam todos os vértices exceto:*
  - *vértices contraditórios, como* x *e* $\neg x$;
  - *vértices da mesma tripla.*

  Afirmação: φ *é satisfatível se, e somente se,* G *possui um* k*-clique.*

  Suponha que φ *é satisfatível, e portanto cada cláusula possui pelo menos um literal verdadeiro. Em cada tripla em* G, *selecionamos um vértice correspondente ao literal verdadeiro. Se mais de um literal for verdadeiro na mesma cláusula, escolhemos um deles aleatoriamente. Os vértices selecionados formam um* k*-clique: o número de vértices selecionados é* k, *cada par de vértices selecionado está ligado por uma aresta.*

  Suponha que G *possui um* k*-clique. Nenhum par de vértices do clique ocorre na mesma tripla porque vértices da mesma tripla não são ligados por arestas. Portanto, cada tripla contém exatamente um dos vértices do* k*-clique. Designamos valores para as variáveis de* φ *de forma que cada literal que marca um vértice assume valor 1 (verdadeiro). Isto é possível porque vértices contraditórios não são ligados. Esta designação de variáveis satisfaz a fórmula* φ *porque cada tripla corresponde a um vértice do clique, e portanto cada cláusula de* φ *tem valor 1.*

7.3 A classe NPC

Seja $\mathcal{C}$ uma classe de problemas caracterizados por uma propriedade, como por exemplo, possuir solução em tempo polinomial. Estamos interessados em identificar os problemas mais difíceis em $\mathcal{C}$, de tal forma que uma solução eficiente para algum destes problemas difíceis resulte em soluções eficientes para todos os outros problemas de $\mathcal{C}$.

Informalmente, dizemos que um problema é *NP-completo*, i.e. que está na classe NPC, se está na classe NP e é tão difícil quanto qualquer problema em NP. Assim, ao mostrarmos que um problema é NP-completo, não estamos tentando provar a existência de um algoritmo eficiente, mas concluir que a existência de um tal algoritmo é improvável. De fato, estamos concluindo sobre o quão difícil ele é, e

não sobre quão fácil como fizemos até então. Portanto, um algoritmo eficiente provavelmente não existe para este problema.

Uma importante questão em aberto é quando P é ou não um subconjunto próprio de NP, o que corresponde ao problema "P vs NP"citado anteriormente. Problemas NP-completos normalmente são considerados intratáveis dada a grande quantidade de problemas NP-completos já estudados, e sem solução polinomial encontrada.

Seria uma surpresa encontrar uma solução polinomial para algum (e portanto para todos) destes problemas. Neste sentido, se um problema é NP-completo então isto pode ser visto como uma evidência da sua intratabilidade. Neste caso, algoritmos aproximados devem ser considerados, ao invés de soluções rápidas e exatas.

Por que até hoje ninguém conseguiu encontrar soluções polinomiais para estes problemas? Não sabemos, talvez porque elas simplesmente não existam, ou porque elas estejam baseadas em princípios ainda desconhecidos.

Qualquer problema em P está também em NP porque pode ser verificada em tempo polinomial pelo mesmo algoritmo que a decide em P sem a necessidade de utilizar certificados.

A seguir, apresentamos a definição formal de problemas NP-completos:

Definição 24. *Um problema $L \subseteq \{0,1\}^*$ é dito **NP-completo** se:*

1. *$L \in NP$;*

2. *$L' \leq_P L, \forall L' \in NP$.*

Denotamos por NPC a classe dos problemas NP-completos.

Se um problema L satisfaz a propriedade 2 acima, mas não necessariamente a propriedade 1, dizemos que L é **NP-difícil**.

Propriedade interessante: Se algum problema NP-completo puder ser resolvido em tempo polinomial então qualquer problema em NP terá solução polinomial.

Como NP-completude consiste em mostrar quão difícil é um problema, utilizamos a redução polinomial na outra direção para mostrar que um problema é NP-completo:

Para mostrarmos que um problema B não possui solução polinomial, consideremos um problema A, para o qual sabemos não existir solução polinomial. Suponha também que temos um algoritmo de redução que transforma instâncias de A em instâncias de B em tempo polinomial. Agora, se existisse uma solução polinomial para B então poderíamos construir uma solução polinomial para A como acima, contradizendo a suposição de que A não possui solução polinomial.

Lema 25. *Se L é um problema tal que $L' \leq_P L$ para algum $L' \in NPC$, então L é NP-difícil. Se adicionalmente, $L \in NP$ então $L \in NPC$.*

Demonstração. Como L' é NP-completo, então $\forall L'' \in$ NP, temos que $L'' \leq_P L'$, e por hipótese temos que $L' \leq_P L$. Logo. $L'' \leq_P L$, o que mostra que L é um problema NP-difícil. Agora, se $L \in$ NP então, por definição temos que $L \in$ NPC.

□

O lema acima nos dá um método para mostrar que um problema L é NP-completo:

1. Prove que $L \in$ NP;

2. Escolha um problema L' que seja NP-completo;

3. Descreva um algoritmo que computa uma função f que mapeia cada instância x de L' em uma instância $f(x) \in L$;

4. Prove que $x \in L'$, se e somente se, $f(x) \in L$;

5. Prove que o algoritmo que computa f é polinomial

Os passos de 2-5 mostram que L é NP-difícil.

Podemos determinar em tempo exponencial se uma dada fórmula booleana φ contendo n variáveis é satisfatível: basta checar cada uma das 2^n possíveis valorações para as variáveis de φ. Nenhum algoritmo polinomial é conhecido para SAT.

SAT $= \{\langle\varphi\rangle : \varphi$ é uma fórmula booleana satisfatível$\}$.

O teorema a seguir, conhecido como o Teorema de Cook-Levin, mostra que é muito improvável que tal algoritmo exista.

Teorema 26. *SAT* $\in$ *NPC.*

Exemplo 27. *Mostre que 3-SAT $\in$ NPC.*

1. *3-SAT $\in$ NP: Dada uma designação de valores para as variáveis de uma 3-FNC fórmula (certificado), o algoritmo de verificação substitui cada variável pelo valor dado e avalia a expressão. Se a expressão resulta em 1 então o certificado é válido e a fórmula é satisfatível.*

2. *SAT $\leq_P$ 3-SAT.*

   Considere a função que toma uma instância φ de SAT e a transforma em uma instância φ' de 3-SAT de acordo com os seguintes casos:

   (a) *C tem apenas um literal, digamos l: Sejam x_1 e x_2 duas variáveis novas. Troque C pelas cláusulas $(l \vee x_1 \vee x_2)$, $(l \vee \overline{x_1} \vee x_2)$, $(l \vee x_1 \vee \overline{x_2})$, $(l \vee \overline{x_1} \vee \overline{x_2})$;*

   (b) *C possui dois literais, ou seja $C = l_1 \vee l_2$: Seja x uma variável nova. Troque C pelas cláusulas $(l_1 \vee l_2 \vee x)$,$(l_1 \vee l_2 \vee \overline{x})$;*

   (c) *C possui três literais: Mantenha C inalterada;*

   (d) *C possui mais de três literais, digamos que $C = l_1 \vee l_2 \vee \ldots \vee l_k$: Sejam $x_1, x_2, \ldots, x_{k-3}$ variáveis novas. Troque C pela fórmula $(l_1 \vee l_2 \vee x_1) \wedge (\overline{x_1} \vee l_3 \vee x_2) \wedge (\overline{x_2} \vee l_4 \vee x_3) \wedge \ldots \wedge (\overline{x_{k-3}} \vee l_{k-1} \vee l_k)$.*

   Agora precisamos mostrar que esta transformação preserva a satisfatibilidade das fórmulas, isto é, φ é satisfatível se, e somente se, φ' é satisfatível. A preservação da satisfatibilidade é trivial para os casos (a), (b) e (c) acima. ***Complete o exercício mostrando que a satisfatibilidade é preservada pela transformação do caso (d)****, isto é, $l_1 \vee l_2 \vee \ldots \vee l_k$ é satisfatível se, e somente se, $(l_1 \vee l_2 \vee x_1) \wedge (\overline{x_1} \vee l_3 \vee x_2) \wedge (\overline{x_2} \vee l_4 \vee x_3) \wedge \ldots \wedge (\overline{x_{k-3}} \vee l_{k-1} \vee l_k)$ é satisfatível.*

Exercício 7. *Mostre que CLIQUE $\in$ NPC.*

Exercício 8. *Uma cobertura de vértices de um grafo $G = (V, E)$ (não-dirigido) é um subconjunto $V' \subseteq V$ tal que $(u, v) \in E$ então $u \in V'$ ou $v \in V'$ (ou ambos!). O tamanho de uma cobertura de vértices é o seu número de vértices, ou seja, $|V'|$. O problema da cobertura de vértices consiste em encontrar uma cobertura de vértices de tamanho mínimo em um grafo. Reescrevendo este problema como um problema de decisão, queremos determinar se um grafo possui uma cobertura de vértices de tamanho dado k:*

VERTEX-COVER $= \{\langle G, k\rangle : G$ possui uma cobertura de vértices de tamanho $k\}$

Mostre que VERTEX-COVER $\in$ NPC.

Exercício 9. *Um* caminho Hamiltoniano *em um digrafo G é um caminho de um vértice u para um vértice v que visita todos os vértices de G exatamente uma vez. O problema de decisão HAMPATH consiste em, dado um digrafo G, responder se G possui um caminho Hamiltoniano ou não.*

$HAMPATH = \{\langle G, u, v\rangle : G$ *é um digrafo com um caminho Hamiltoniano de* u *para* $v\}$

Mostre que $HAMPATH \in NPC$.

Exercício 10. *Um* ciclo Hamiltoniano *em um digrafo G é um ciclo simples contendo cada vértice de G. O problema de decisão HAM-CYCLE consiste em, dado um digrafo G, responder se G possui um ciclo Hamiltoniano ou não.*

$HAM\text{-}CYCLE = \{\langle G\rangle : G$ *é um digrafo com um ciclo Hamiltoniano.*$\}$

Mostre que $HAM\text{-}CYCLE \in NPC$.

Exercício 11. *Considere o seguinte jogo em um grafo (não-dirigido) $G = (V, E)$, que inicialmente contém 0 ou mais bolas de gude em seus vértices: um movimento deste jogo consiste em remover duas bolas de gude de um vértice $v \in V$, e adicionar uma bola a algum vértice adjacente de v. Agora, considere o seguinte problema (de decisão): Dado um grafo G, e uma função $p(v)$ que retorna o número de bolas de gude no vértice v, existe uma sequência de movimentos que remove todas as bolas de G, exceto uma? Mostre que este problema está em NPC.*

Solução. *Se $\langle G, u, v\rangle$ é uma instância de HAMPATH, construa uma instância $\langle G', p\rangle$ de T: $G' = G$, $p(u) = 2$, $p(v) = 0$ e $p(x) = 1, \forall x \in G.V \backslash \{u, v\}$.*

Agora mostre que $\langle G, u, v\rangle$ tem um caminho Hamiltoniano, se e somente se, $\langle G', p\rangle$ tem uma solução:

1. *Se $\langle G, u, v\rangle$ tem um caminho Hamiltoniano então $\langle G', p\rangle$ tem uma solução.*

   Suponha que $\langle G, u, v\rangle$ tenha um caminho Hamiltoniano de u para v, digamos $\langle u = x_0, x_1, x_2, \ldots, x_n = v\rangle$. Vamos construir uma sequência de movimentos da seguinte forma: Como o vértice u de G' é o único vértice contendo duas bolas $\langle G', p\rangle$, removemos as duas bolas de u e adicionamos uma bola no vértice adjacente x_1. Em seguida, removemos as duas bolas do vértice x_1, e adicionamos uma bola no vértice x_2, e assim prosseguimos ao longo do caminho Hamiltoniano $\langle u = x_0, x_1, x_2, \ldots, x_n = v\rangle$. Note que no último passo removeremos as duas bolas do vértice x_{n-1} e adicionaremos uma bola no vértice v. Desta forma, finalizamos com G' contendo apenas uma bola no vértice v, e as bolas de todos os outros vértices foram removidas.

2. *Se $\langle G', p\rangle$ tem uma solução então $\langle G, u, v\rangle$ tem um caminho Hamiltoniano.*

   Suponha que exista uma sequência de movimentos que remove todas as bolas de G', exceto uma. Pela forma como G' foi construído, apenas um de seus vértices contém duas bolas, digamos x_0, e portanto a sequência de movimentos tem que iniciar em x_0. Adicione cada vértice visitado por esta sequência de movimentos ao caminho, na ordem em que a visita é feita. Nenhum vértice pode ser visitado mais de uma vez porque, com exceção de x_0 e de x_n, todos os vértices possuem apenas uma bola. Todos os vértices têm que ser visitados porque esta é a única forma de remover as bolas. Assim, a sequência de movimento constrói um caminho Hamiltoniano em G', e portanto $G (= G')$ possui um caminho Hamiltoniano.

Exercício 12. *Considere um jogo de tabuleiro que contém n linhas e n colunas. Cada uma das n^2 posições, pode ter uma bola azul, uma bola vermelha ou pode estar vazia (configuração inicial). O jogo consiste em remover as bolas do tabuleiro de forma que cada coluna tenha bolas de uma única cor, e cada linha contenha pelo menos uma bola (configuração vencedora). Mostre que o problema de determinar se uma configuração inicial resulta em uma configuração vencedora está em NPC.*

Solução. *Denote por T a configuração inicial do tabuleiro, representaremos o problema correspondente a este jogo por $S = \{T$ corresponde a uma configuração vencedora$\}$. Considere a seguinte transformação: Dada uma instância φ de 3-SAT com m variáveis $v_1, v_2, \ldots, v_m$ e k cláusulas $\varphi = (l_1^1 \vee l_2^1 \vee l_3^1) \wedge (l_1^2 \vee l_2^2 \vee l_3^2) \wedge \ldots \wedge (l_1^k \vee l_2^k \vee l_3^k)$, construa um tabuleiro $k \times m$, assumindo que nenhuma cláusula de φ contém simultaneamente as variáveis v_i e $\overline{v_i}$ (tais cláusulas podem ser removidas sem afetar a satisfatibilidade de φ), da seguinte forma:*

1. *Se x_i ocorre na cláusula c_j coloque uma pedra azul na linha c_j e coluna x_i.*

2. *Se $\overline{x_i}$ ocorre na cláusula c_j coloque uma pedra vermelha na linha c_j e coluna x_i.*

O tabuleiro pode ser completado, para que tenha o mesmo número de linhas e colunas, repetindo uma linha ou adicionando uma coluna em branco sem afetar a solvabilidade.

Agora precisamos mostrar que φ é satisfatível se, e somente se, T possui uma solução.

Se φ é satisfatível, então considere uma designação d que faz com que cada cláusula de φ seja verdadeira. Se x_i é verdadeira (resp. falsa) segundo a designação d então remova as pedras vermelhas (resp. azuis) da coluna x_i. Portanto, pedras que correspondem a literais verdadeiros permanecem, e como toda cláusula possui um literal verdadeiro, toda linha possui uma pedra.

Reciprocamente, suponha que T possui uma solução. Se pedras vermelhas (resp. azuis) foram removidas de uma coluna então associe o valor verdadeiro (resp. falso) à variável correspondente. Como cada linha possui uma pedra, toda cláusula possui um literal positivo, e portanto φ é satisfatível.

Referências Bibliográficas

[1] Emilio Jesús Gallego Arias, Benoît Pin, and Pierre Jouvelot. jsCoq: Towards Hybrid Theorem Proving Interfaces. *Electronic Proceedings in Theoretical Computer Science*, 239:15–27, January 2017.

[2] Jeremy Avigad, Kevin Donnelly, David Gray, and Paul Raff. A formally verified proof of the prime number theorem. *ACM Transactions on Computational Logic*, 9(1):2–es, December 2007.

[3] Jeremy Avigad and John Harrison. Formally verified mathematics. *Communications of the ACM*, 57(4):66–75, April 2014.

[4] Gilles Brassard and Paul Bratley. *Fundamentals of Algorithmics*. Prentice-Hall, Inc., USA, 1996.

[5] A. Chlipala. *Certified Programming with Dependent Types*. MIT Press, 2017.

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[7] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, Cambridge, MA, USA, 4 edition, April 2022.

[8] Leonardo de Moura and Sebastian Ullrich. The Lean 4 Theorem Prover and Programming Language. In André Platzer and Geoff Sutcliffe, editors, *Automated Deduction – CADE 28*, Lecture Notes in Computer Science, pages 625–635, Cham, 2021. Springer International Publishing.

[9] G. Gonthier. A computer-checked proof of the Four Colour Theorem. Technical report, Microsoft Research Cambridge, 2008.

[10] Thomas Hales, Mark Adams, Gertrud Bauer, Dat Tat Dang, John Harrison, Truong Le Hoang, Cezary Kaliszyk, Victor Magron, Sean McLaughlin, Thang Tat Nguyen, Truong Quang Nguyen, Tobias Nipkow, Steven Obua, Joseph Pleso, Jason Rute, Alexey Solovyev, An Hoai Thi Ta, Trung Nam Tran, Diep Thi Trieu, Josef Urban, Ky Khac Vu, and Roland Zumkeller. A formal proof of the Kepler conjecture, January 2015.

[11] M. Huth and M. Ryan. *Logic in Computer Science: Modelling and Reasoning About Systems*. Cambridge University Press, New York, NY, USA, 2004.

[12] Cezary Kaliszyk, Stephan Schulz, Josef Urban, and Jiří Vyskočil. System Description: E.T. 0.1. In Amy P. Felty and Aart Middeldorp, editors, *Automated Deduction - CADE-25*, volume 9195, pages 389–398. Springer International Publishing, Cham, 2015.

[13] Xavier Leroy. Formal Verification of a Realistic Compiler. *Communications of the ACM*, 52(7):107, 2009.

[14] A. V. Levitin. *Introduction to the Design and Analysis of Algorithms, Third Edition*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2012.

[15] W. McCune. Prover9 and mace4. http://www.cs.unm.edu/~mccune/prover9/, 2005–2010.

[16] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lncs*. Springer, 2002.

[17] R. B. Nogueira, A. C. A. Nascimento, F. L. C. de Moura, and M. Ayala-Rincón. Formalization of Security Proofs Using PVS in the Dolev-Yao Model. In *Booklet Proc. Computability in Europe - CiE*, 2010.

[18] S. Owre, J. M. Rushby, and N. Shankar. PVS: A Prototype Verification System. In D. Kapur, editor, *CADE*, volume 607 of *Lnai*, pages 748–752. sv, 1992.

[19] C. Paulin-Mohring. Introduction to the Calculus of Inductive Constructions. 2014.

[20] Lawrence C. Paulson. A Mechanised Proof of Gödel's Incompleteness Theorems Using Nominal Isabelle. *J Autom Reasoning*, 55(1):1–37, 2015.

[21] Benjamin C. Pierce, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Catvalin Hriatcu, Vilhelm Sjoberg, and Brent Yorgey. *Software Foundations*. Electronic textbook, 2014.

[22] Alexandre Riazanov and Andrei Voronkov. The design and implementation of VAMPIRE. *AI Commun.*, 15(2-3):91–110, 2002.

[23] Michael Sipser. *Introduction to the Theory of Computation*. International Thomson Publishing, 1st edition, 1996.

[24] Raymond Smullyan. *Logical Labyrinths*. AK Peters, 2009.

[25] The Coq Development Team. The Coq Proof Assistant. Zenodo, October 2021.

[26] D. van Dalen. *Logic and Structure*. Universitext. Springer London, 2013.