{algorithms}

Algoritmos: Notas e Compilações

versão: 7/24

VINÍCIUS DIAS
viniciusdias@ufla.br

Este conteúdo começou a ser reunido em 2022, mas está em contrante atualização e aprimoramento ...

Resumo

Este documento reúne algumas notas e compilações dos principais tópicos geralmente incluídos em cursos de nível superior em Ciência da Computação ou relacionados. O conteúdo inclui reflexões sobre os principais assuntos do estudo de análise, projeto e complexidade de algoritmos, além de compilações de exemplos e pontos chaves da bibliografia padrão da área. Por isso, créditos devidos aos excelentes livros: CORMEN [1], LEVITIN [4], ERICKSON [2] e TARDOS [3].

Sumário

1 Modelo e Análise de Algoritmos **4**
- 1.1 Introdução à análise de algoritmos . . . 4
- 1.2 Modelo de computação e conceitos de análise . . . 4
- 1.3 Corretude de algoritmos . . . 6
- 1.4 Notações assintóticas . . . 7
  - 1.4.1 Big-Oh ou $O(g(n))$: um limite superior . . . 7
  - 1.4.2 Big-Omega ou $\Omega(g(n))$: um limite inferior . . . 8
  - 1.4.3 Big-Theta ou $\Theta(g(n))$: um limite justo . . . 9
- 1.5 Analisando algoritmos iterativos . . . 9
- 1.6 Analisando algoritmos recursivos . . . 9

2 Divisão e Conquista: o paradigma e recorrências **10**
- 2.1 Introdução sobre algoritmos de divisão e conquista . . . 10
- 2.2 Resolvendo recorrências usando expansão de termos . . . 10
- 2.3 Resolvendo recorrências usando árvores de recursão . . . 10
- 2.4 Resolvendo recorrências usando indução matemática (método da substituição) . . . 10
- 2.5 Resolvendo recorrências usando o Teorema Mestre . . . 10
- 2.6 Resolvendo recorrências usando o Akra-Bazzi . . . 11
- 2.7 Estudo de caso: ordenação por intercalação (merge sort) . . . 11
- 2.8 Estudo de caso: ordenação usando quick sort . . . 12
- 2.9 Estudo de caso: subarranjo de soma máxima . . . 12
- 2.10 Estudo de caso: número de inversões em um arranjo . . . 13

3 Algoritmos Fundamentais para Ordenação **14**
- 3.1 Introdução sobre algoritmos de ordenação . . . 14
- 3.2 Bubble sort . . . 14
- 3.3 Insertion sort . . . 14
- 3.4 Merge sort . . . 15
- 3.5 Quick sort . . . 16
- 3.6 Heap sort . . . 17
- 3.7 Ordenação em tempo linear . . . 17
  - 3.7.1 Limite inferior para ordenação . . . 18
  - 3.7.2 Counting sort . . . 19
  - 3.7.3 Radix sort . . . 19
  - 3.7.4 Bucket sort . . . 19

4 Técnicas de Projeto de Algoritmos **20**
4.1 Projeto de algoritmos usando Força-bruta . . . 20
4.1.1 Estudo de caso: Selection sort . . . 20
4.1.2 Estudo de caso: Problema da mochila . . . 20
4.2 Projeto de algoritmos usando *Backtracking* . . . 21
4.2.1 Estudo de caso: problema das N rainhas . . . 21
4.3 Projeto de algoritmos usando estratégias gulosas . . . 22
4.3.1 Estudo de caso: problema do troco . . . 22
4.3.2 Estudo de caso: problema do escalonamento de tarefas . . . 22
4.3.3 Estudo de caso: árvore geradora mínima . . . 24
4.3.4 Estudo de caso: compressão de documentos . . . 24
4.4 Projeto de algoritmos usando Programação Dinâmica . . . 25
4.4.1 Ilustrando os princípios de programação dinâmica através de algoritmos para a sequência de Fibonacci . . . 25
4.4.2 Estudo de caso: corte de hastes . . . 27
4.4.3 Estudo de caso: problema da mochila . . . 27
4.4.4 Estudo de caso: máxima subsequência crescente . . . 27
4.5 Branch-and-Bound . . . 28
4.5.1 Problema da mochila . . . 29
4.5.2 Problema da alocação . . . 29
4.6 Programação Linear . . . 30

5 Algoritmos em Grafos **31**
5.1 Caminhamento em largura (Breadth-first search - BFS) . . . 31
5.2 Caminhamento em profundidade (Depth-first search - DFS) . . . 31
5.2.1 Classificação de arestas em DFS . . . 32
5.2.2 Ordenação topológica . . . 33
5.2.3 Encontrando componentes fortemente conectados . . . 33
5.3 Árvores geradoras de custo mínimo . . . 33
5.3.1 Algoritmo de Prim . . . 33
5.3.2 Algoritmo de Kruskal . . . 34
5.4 Propriedades de caminhos mínimos . . . 34
5.5 Caminhos mínimos a partir de uma única fonte . . . 35
5.5.1 Algoritmo de Bellman-Ford . . . 35
5.5.2 Algoritmo de Dijkstra . . . 35
5.5.3 Caminhos mínimos em DAGs . . . 36
5.6 Caminhos mínimos entre todos os pares de vértices . . . 36
5.6.1 Algoritmo de Floyd-Warshall . . . 37
5.6.2 Algoritmo de Johnson . . . 37
5.7 Fluxo máximo . . . 37
5.8 Exercícios . . . 38
5.9 Grafos Eulerianos e Hamiltonianos . . . 38
5.9.1 Definições importantes . . . 39
5.9.2 Grafos eulerianos . . . 39
5.9.3 Grafos hamiltonianos . . . 41

6 Teoria da Complexidade de Algoritmos: Classes de Problemas e NP-Completude **42**

7 Técnicas para tratar problemas NP-Completos **43**
7.1 Algoritmos aproximativos 44
7.1.1 Algoritmo aproximativo para o PCV com desigualdade triangular 44
7.2 Heurísticas construtivas 44
7.3 Metaheurísticas 44
7.4 Busca em vizinhança 45
7.5 Metaheurísticas baseadas em busca local: Variable Neighborhood Search 46
7.6 Metaheurísticas baseadas em busca local: GRASP 46
7.7 Metaheurísticas baseadas em busca local: Busca Tabu 48
7.7.1 Implementação da lista tabu 48
7.8 Metaheurísticas populacionais 48
7.9 Metaheurísticas baseadas em populações: algoritmos genéticos 49
7.10 Metaheurísticas baseadas em populações: colônia de formigas 50

Capítulo 1

Modelo e Análise de Algoritmos

1.1 Introdução à análise de algoritmos

Um algoritmo pode ser definido como uma sequência de passos usada para resolver algum problema. Naturalmente, em computação, estamos mais interessados em problemas que possam ser *abstraídos* usando matemática e algoritmos para esses problemas que sejam *precisos* na definição de cada passo e *passíveis de reprodução* por um computador (ou modelo de computação).

Sendo assim, podemos estabelecer a seguinte relação entre o que é prático e o que é abstrato no estudo da resolução de problemas:

Programa	$\approx$	Algoritmo
Linguagem de programação	$\approx$	Pseudocódigo
Computador físico	$\approx$	Computador abstrato (ou modelo de computação)

Nosso objetivo é estudar algoritmos do ponto de vista abstrato (segunda coluna acima). A primeira etapa, portanto, é formalizar bem um problema para que não se tenha dúvidas com relação ao que é dado como entrada e o que se espera como resultado (saída). Por exemplo, o problema de ordenar um conjunto de números inteiros em um arranjo pode ser formalizado da seguinte forma:

Entrada:	sequência de números inteiros organizados em um arranjo $a = [a_1, a_2, \cdots, a_n]$
Saída:	permutação Θ $[a_1^\Theta, \cdots, a_n^\Theta]$ tal que $a_1^\Theta \leq a_2^\Theta \leq \cdots \leq a_n^\Theta$

1.2 Modelo de computação e conceitos de análise

O primeiro requisito ao se trabalhar com algoritmos é ser capaz de analisá-los, isto é, precisamos ser capazes de determinar certas características do algoritmo antes mesmo de implementá-lo e executá-lo na prática. Por exemplo, é muito comum que estejamos interessados em algoritmos *eficientes*. Precisamos, portanto, estabelecer como será o computador abstrato ao qual o algoritmo estudado se refere. No estudo de algoritmos vamos adotar um modelo de computação abstrata que reflete características muito comuns de arquiteturas computacionais contemporâneas: o modelo RAM – *Random Access Machine*. As seguintes características estão presentes em algoritmos projetados para o modelo RAM:

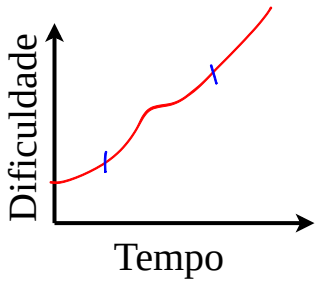
- As instruções dos algoritmos são executadas sempre sequencialmente, sem nenhum tipo de concorrência ou paralelismo;

- Vamos assumir que os dados manipulados pelos algoritmos são organizados em uma memória de acesso aleatório e organizada em palavras;
- Tipos de dados: inteiros, ponto flutuante, arranjos de palavras (sequência de inteiros ou ponto flutuante), qualquer outra estrutura de dados derivada;
- Operações aritméticas em geral: $+, -, \div, \times,$ mod , etc.;
- Controle: while, for, do .. while, return, subrotinas/funções, condicionais (if, else), operadores de comparação $(=, <, >,,$ etc.);
- Atribuições de variáveis são representadas como $\leftarrow$.

1 Exemplo de pseudocódigo no modelo RAM. Vamos ver um exemplo de algoritmo para o problema de contar quantas vezes um número inteiro n ocorre em um arranjo de inteiros v. Formalmente, entrada: inteiro n e arranjo de inteiros v; saída: inteiro c representando o número de ocorrências de n em v. O tamanho de v é representado nesse exemplo como $|v|$:

```
CONTA-OCORRENCIAS(n, v)
1  c ← 0
2  for i ← 1 to |v|
3      if v[i] = n
4          c ← c + 1
5  return c
```

2 Dimensões de análise. Diferentes instâncias podem demandar diferentes níveis de dificuldade para a execução do algoritmo. Esperamos que, seja em uma máquina abstrata ou não, cada problema é passível de encontrar instâncias fáceis (entradas fáceis) ou instâncias mais difíceis (entradas mais difíceis). Por exemplo, para o problema de contar as ocorrências de um número no arranjo não é razoável assumir que o mesmo algoritmo irá se comportar da mesma forma para um arranjo de 10 elementos em comparação a um arranjo de 10^7 elementos. Por esse motivo, vamos sempre estudar algoritmos em função do *tamanho da entrada* que lhe é apresentada. Com isso, temos nosso principal objetivo em análise de algoritmos: para um algoritmo, estimar ou representar sua dificuldade frente a instâncias cada vez maiores na entrada:



As duas noções de dificuldade para um computador abstrato que estamos interessados são:

1. Tempo: quantas *operações básicas* um algoritmo executa para uma dada instância do problema. Nesse caso, ao invés de medir tempo de relógio, nós vamos contar quantas operações um algoritmo executa.
2. Espaço: quanto de espaço (palavras do RAM) além da entrada o algoritmo utiliza em função também do tamanho da instância de entrada.

Chamamos portanto de *operação básica* aquilo que pretendemos medir em um algoritmo. Essa escolha é feita por quem analisa o algoritmo, mas por exemplo: (a) para um problema de ordenar números em um arranjo pode ser natural pensar que as operações que são mais importantes para determinar a ordem entre os elementos são as operações que *comparam* dois elementos do arranjo: $a[i] \leq a[j]$, por exemplo; (b) para um problema matemático de encontrar o máximo divisor comum entre dois números pode ser uma boa prática escolher operações aritméticas (possivelmente mod) como operação básica. Portanto, não existe regra mas existem boas práticas.

3 Contexto de análise O pior caso representa instâncias para o problema que fazem o algoritmo trabalhar ao máximo: usar o máximo de espaço ou executar o máximo de operações básicas possível. O melhor caso é o contrário, indicando cenários otimistas para o algoritmo. Finalmente, o caso médio representa os casos em que o algoritmo trabalha de acordo com esperado na média, para isso sendo necessário assumir algumas características da entrada (por exemplo, independência e uniformidade entre instâncias).

1.3 Corretude de algoritmos

Para ilustrar como identificar algoritmos corretos, vamos usar como exemplo um algoritmo famoso de ordenação: o INSERTION-SORT:

```
INSERTION-SORT(A)
1  for i ← 2 to A.length
2      elem ← A[i]
3      j ← i − 1
4      while j ≥ 1 and elem < A[j]
5          A[j + 1] ← A[j]
6          j ← j − 1
7      A[j + 1] ← elem
```

O algoritmo segue o princípio de organização de cartas de um baralho na mão de um jogador: sempre inserimos uma nova carta em sua posição correta na mão. Dessa forma, a mão sempre contém um subconjunto de cartas já ordenadas (nesse caso, usaremos inteiros para ilustrar).

Para verificar a corretude do INSERTION-SORT utilizaremos uma metodologia que lembra uma indução matemática (sendo bem sincero, É uma indução matemática adaptada ao contexto de algoritmos). Vejamos:

- Se um algoritmo não possui nenhum laço de repetição, sua corretude é de fácil verificação através da inspeção exaustiva de cada uma de suas operações, sequencialmente.
- Vamos nos concentrar, portanto, em cenários em que o algoritmo contém laços de repetição que alteram seu comportamento de acordo com o tamanho da entrada.

Para esse último caso, a técnica indutiva utilizada é denominada “invariante de loop”, e consiste da verificação de três etapas:

1. **Inicialização:** mostramos que alguma propriedade do algoritmo é garantida antes da entrada no loop.
2. **Manutenção:** mostramos que, antes de iniciar a próxima iteração do loop, a propriedade continua sendo garantida.

3. **Terminação:** mostramos que ao final do loop, a propriedade continua sendo garantida e que, desse fato, concluímos que o algoritmo é correto.

Naturalmente, essa dita propriedade deve ser algo relacionado ao que o algoritmo de propõe a resolver. No caso da ordenação, ordenar alguns números. Vamos aplicar a técnica para o INSERTION-SORT. A propriedade pertinente é: "a todo momento do algoritmo, temos que alguns números já estão ordenados ($a[1 \cdots i-1]$) e esses números são os mesmos que, originalmente, se encontravam nas mesmas posições ($1 \cdots i-1$)".

Inicialização: antes do loop, $i = 2$, e por isso o arranjp $a[1 \cdots 1]$ está ordenado trivialmente.
Manutenção: antes de iniciar uma próxima iteração, temos que $a[1 \cdots i-1]$ é um subarranjo ordenado. Em uma iteração i, o algoritmo posiciona $a[i]$ em sua posição correta/ordenada em $a[1 \cdots i-1]$ realizando o deslocamento de elementos maiores enquanto for necessário. Assim, ao final, da iteração i temos os mesmos elementos $a[1 \cdots i]$ mas ordenados.
Terminação: ao final da última iteração, $i = n+1$. Como $a[1 \cdots i-1]$ está ordenado, então $a[1 \cdots n+1-1] = a[1 \cdots n]$ também estará.

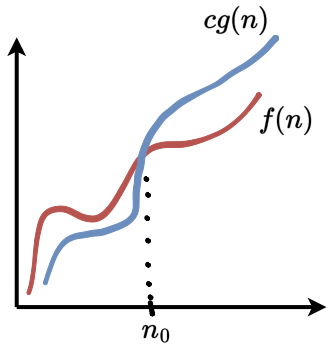
Qualquer algoritmo com laços pode ser verificado formalmente utilizando invariantes e se o algoritmo contiver vários laços, um conjunto de invariantes pode ser usado para garantir sua corretude globalmente.

1.4 Notações assintóticas

Quando analisamos um algoritmo, tipicamente queremos encontrar uma função matemática que representa a complexidade de tempo ou espaço sob algumas condições (como contexto de análise, por exemplo). Na prática, essas funções podem ser bem diversas em termos de constantes e termos. Por exemplo, $n^2/2+n/2-1$ poderia ser uma função que representa o crescimento do número de operações do algoritmo à medida que o tamanho da entrada cresce ($n \to \infty$). Esse polinômio tem grau 2, apesar de todos os coeficientes e termos da função, assim como n^2 é outro polinômio de grau 2. O interessante é que o fato de ser um polinômio de grau 2, independentemente dos coeficientes e termos, já nos remete ao formato da curva dessa função: uma parábola. Como veremos, as **notações assintóticas nos ajudam justamente a categorizar funções cujas ordens de crescimento (ou formatos de curva) são equivalentes**.

1.4.1 Big-Oh ou $O(g(n))$: um limite superior

Esta notação é utilizada para estabelecer um limite superior para o crescimento de funções (nesse caso, $g(n)$). Dessa forma, $n^2 + n \in O(n^2)$ pois nada cresce mais do que o termo quadrático quando $n \to \infty$. Da mesma forma, $n \in O(n^2)$, já que uma parábola sempre irá ultrapassar uma reta a partir de algum momento.



Formalmente, $f(n) \in O(g(n))$ sse:

Existirem constantes $c > 0$ e $n_0 \geq 0$ ($c \in \mathbb{R}_+$, $n_0 \in \mathbb{Z}_+$) tais que: $f(n) \leq cg(n)$ para todo $n \geq n_0$

Exemplo 1.1. *Mostrar que* $n^2 - n \in O(n^2)$. *Devemos garantir que* $\exists c > 0, n_0 \geq 0$ *e, para isso, desenvolvemos a inequação:*

$$n^2 - n \leq cn^2$$
$$1 - 1/n \leq c$$

Note que quando $n \to \infty$, *o lado esquerdo da inequação é limitado por uma constante, isto é, c existe. Assim, quando* $n = 1$ *então* $c \geq 0$. *Portanto, uma possível resposta possível é o par* $[c = 1, n_0 = 1]$.

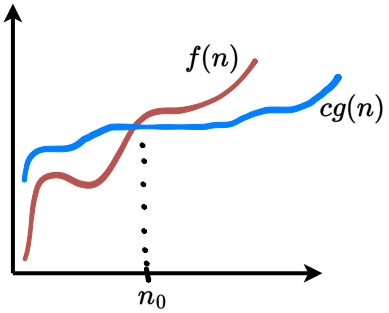
Exemplo 1.2. *Mostrar que* $n - 7 \in O(n^3)$. *Devemos garantir que* $\exists c > 0, n_0 \geq 0$ *e, para isso, desenvolvemos a inequação:*

$$n - 7 \leq cn^3$$

Note que o lado esquerdo já é menor ou igual ao lado direito para $n \geq 0$, *para qualquer que seja o* $c > 0$. *Portanto, uma possível resposta possível é o par* $[c = 1, n_0 = 0]$.

1.4.2 Big-Omega ou $\Omega(g(n))$: um limite inferior

É utilizada para representar limites inferiores de funções. Assim, quando $n \to \infty$, $n \log n \in \Omega(n)$ e $n^2 \in \Omega(n^2)$, mas $n + 7 \notin \Omega(n^2)$. Veja a seguir uma ilustraçao que indica isso:



Formalmente, $f(n) \in \Omega(g(n))$ sse:

Existirem constantes $c > 0$ e $n_0 \geq 0$ ($c \in \mathbb{R}_+$, $n_0 \in \mathbb{Z}_+$) tais que: $f(n) \geq cg(n)$ para todo $n \geq n_0$

1.4.3 Big-Theta ou $\Theta(g(n))$: um limite justo

Um limite justo é usado para representar os casos em que uma função pode ser limitada tanto inferiormente (Ω) quanto superiormente (O) por uma outra função. De maneira informal, dizer que $f(n) \in \Theta(g(n))$ equivale a afirmar que ambas as funções possuem exatamente a mesma ordem de crescimento assintótico.

Formalmente, $f(n) \in \Theta(g(n))$ sse $f(n) \in \Omega(g(n))$ e $f(n) \in O(g(n))$

1.5 Analisando algoritmos iterativos

1.6 Analisando algoritmos recursivos

Capítulo 2

Divisão e Conquista: o paradigma e recorrências

2.1 Introdução sobre algoritmos de divisão e conquista

2.2 Resolvendo recorrências usando expansão de termos

2.3 Resolvendo recorrências usando árvores de recursão

2.4 Resolvendo recorrências usando indução matemática (método da substituição)

2.5 Resolvendo recorrências usando o Teorema Mestre

Usado para recorrências do tipo

$$T(n) = \begin{cases} aT(n/b) + f(n) & \text{se não-trivial} \\ \Theta(1) & \text{se trivial} \end{cases}$$

O teorema consiste em comparar $f(n)$ com a grandeza $n^{\log_b a}$ e o que dominar assintoticamente é a solução da recorrência. A função $f(n)$ representa o custo associado às operações que são efetuadas dentro de uma única chamada recursiva, ignorando o custo adicional das outras chamadas recursivas (conquista). Por outro lado, $n^{\log_b a}$ representa o contrário, o custo associado às chamadas recursivas:

- se $f(n) \succ n^{\log_b a}$: significa que o custo dominante da recorrência se concentra nas raízes das (sub)árvores, isto é, o custo dos níveis da árvore de recursão vai progressivamente diminuindo.
- se $f(n) \prec n^{\log_b a}$: significa que o custo dominante da recorrência acontece nos filhos das (sub)árvores, isto é, o custo dos níveis da árvore de recursão vai progressivamente aumentando.
- se $f(n) \equiv n^{\log_b a}$: significa que cada nível contribui igualmente para o custo.

Caso 1. Se $f(n) \in O(n^{\log_b a+\epsilon})$ para $\epsilon > 0$, então $T(n) \in \Theta(n^{\log_b a})$

Caso 2. Se $f(n) \in \Theta(n^{\log_b a})$, então $T(n) \in \Theta(n^{\log_b a} \log n)$

Caso 3. Se $f(n) \in \Omega(n^{\log_b a - \epsilon})$ para $\epsilon > 0$ e atender uma condição de regularização $af(n/b) \leq cf(n)$ para $c > 0$ e a partir de $n \geq n_0$, então $T(n) \in \Theta(f(n))$

2.6 Resolvendo recorrências usando o Akra-Bazzi

Se trata de uma generalização do teorema mestre, possibilitando a solução de recorrências do tipo:

$$T(n) = \begin{cases} c_0 & n = x_0 \\ \sum_{i=1}^{k} a_i T(b_i n) + f(n) & n > x_0 \end{cases}$$

Onde: $c_0 \in \mathbb{R}$, $x_0 \in \mathbb{N}$, $a_i > 0$, $0 < b_i < 1$, $f(n)$ é uma função assintoticamente positiva. Além disso, seja p a única raíz real da equação:

$$\sum_{i=1}^{k} a_i b_i^p = 1$$

O teorema de Akra-Bazzi conclui um limite assintótico justo para $T(n)$:

$$T(n) \in \Theta\left(n^p + n^p \int_{n_0}^{n} \frac{f(x)}{x^{p+1}} dx\right)$$

2.7 Estudo de caso: ordenação por intercalação (merge sort)

```
MERGE-SORT(A)
1  if A.length = 1 or A.length = 0 return
2  mid ← ⌊A.length/2⌋
3  left ← A[1 ··· mid]
4  right ← A[mid + 1 ··· A.length]
5  MERGE-SORT(left)
6  MERGE-SORT(right)
7  MERGE(left, right, A)
```

```
MERGE(left, right, A)
 1  i ← 1; j ← 1; k ← 1
 2  while i ≤ left.length and j ≤ right.length
 3     if left[i] < right[j]
 4        A[k] ← left[i]
 5        i ← i + 1
 6     else
 7        A[k] ← right[j]
 8        j ← j + 1
 9     k ← k + 1
10  while i ≤ left.length
11     A[k] ← left[i]
12     i ← i + 1; k ← k + 1
13  while j ≤ right.length
14     A[k] ← right[j]
15     j ← j + 1; k ← k + 1
```

2.8 Estudo de caso: ordenação usando quick sort

```
QUICK-SORT(A, l, r)
 1  if l ≥ r return
 2  p ←PARTITION(A, l, r)
 3  QUICK-SORT(A, l, p − 1)
 4  QUICK-SORT(A, p + 1, r)
```

```
PARTITION(A, l, r)
 1  pivot ← A[r]
 2  i ← l − 1
 3  for j ← 1 to r − 1
 4     if A[j] ≤ pivot
 5        i ← i + 1
 6        swap(A, j, i)
 7  swap(A, i + 1, r)
 8  return i + 1
```

2.9 Estudo de caso: subarranjo de soma máxima

O subarranjo de soma máxima pode estar: (1) na metade esquerda; (2) na metade direita; (3) cruzando o meio. Observe que, se estamos no caso (3), a restrição nos ajuda a encontrar o melhor subarranjo: a partir do meio, basta adicionarmos elementos à esquerda rastreando em que posição a soma fica máxima e o mesmo à direita. Assim, (1) e (2) são resolvidos diretamente pela recursão durante a conquista dos subproblemas e (3) é resolvido pelo procedimento MAX-SUM-SUBARRAY-CROSSING-MID de complexidade linear $\Theta(n)$, onde n é o tamanho do arranjo.

```
MAX-SUM-SUBARRAY(a, l, r)
 1  if l = r return (l, r, a[l])
 2  mid ← ⌊(l + r)/2⌋
 3  (l1, r1, s1) ←MAX-SUM-SUBARRAY(a, l, mid)
 4  (l2, r2, s2) ←MAX-SUM-SUBARRAY(a, mid + 1, r)
 5  (l3, r3, s3) ←MAX-SUM-SUBARRAY-CROSSING-MID(a, l, r, mid)
 6  if s1 > s2 and s1 > s3 return (l1, r1, s1)
 7  else if s2 > s3 return (l2, r2, s2)
 8  else return (l3, r3, s3)
```

```
MAX-SUM-SUBARRAY-CROSSING-MID(a, l, r, mid)
 1  s1 ← −∞
 2  s ← 0
 3  for i ← mid downto l
 4     s ← s + a[i]
 5     if s > s1
 6        s1 ← s
 7        l1 ← i
 8  s2 ← −∞
 9  s ← 0
10  for i ← mid + 1 to r
11     s ← s + a[i]
12     if s > s2
13        s2 ← s
14        r1 ← i
15  return (l1, r1, s1 + s2)
```

2.10 Estudo de caso: número de inversões em um arranjo

Capítulo 3

Algoritmos Fundamentais para Ordenação

3.1 Introdução sobre algoritmos de ordenação

3.2 Bubble sort

A ideia do bubblesort é realizar sucessivas trocas de elementos adjacentes que estejam fora de ordem até que todos estejam garantidamente em sua posição correta ordenada. É importante observar que a quantidade máxima de vezes em que um elemento do arranjo se encontre fora de ordem com seu adjacente é $n-1$, sendo n o tamanho do arranjo. Em outras palavras, ele deveria ser trocado com todos os outros elementos do arranjo que não ele próprio e por isso, o laço mais externo do algoritmo é executado $n-1$ vezes. Na primeira iteração, o menor elemento é levado à primeira posição. Na segunda iteração, o segundo menor é levado à segunda posição, e assim por diante.

```
BUBBLE-SORT(A)
1  for i ← 1 to A.length − 1
2      for j ← A.length downto i + 1
3          if A[j] < A[j − 1]
4              swap(A, j, j − 1)
```

Complexidade de tempo (pior caso e melhor caso): $\Theta(n^2)$, pois a operação de comparação entre elementos acontece sem restrições dentro dos dois laços aninhados.

Complexidade de espaço (pior caso e melhor caso): $\Theta(1)$, pois usamos apenas uma quantidade constante de memória para variáveis auxiliares além da memória fornecida na entrada.

3.3 Insertion sort

A ideia do insertion sort é começar com um subarranjo ordenado e adicionar elemento a elemento em sua posição correta no nesse subarranho ordenado. Ao final, o subarranjo ordenado vai conter todos os elementos do arranjo e portanto, o problema estará resolvido. A analogia para este método é a ordenação de cartas de baralho na mão: a cada carta retirada do monte é posicionado na ordem, em sua posição correta na sequência em construção.

```
INSERTION-SORT(A)
1  for i ← 2 to A.length
2     elem ← A[i]
3     j ← i − 1
4     while j ≥ 1 and elem < A[j]
5        A[j + 1] ← A[j]
6        j ← j − 1
7     A[j + 1] ← elem
```

Complexidade de tempo (pior caso): $\Theta(n^2)$, sendo que isso acontece quando a comparação $elem < A[j]$ acontece todas as vezes possíveis a cada iteração do laço exterior, isto é, $i-1$ vezes.

Complexidade de tempo (melhor caso): $\Theta(n)$, sendo que isso acontece quando a comparação $elem < A[j]$ é feita apenas uma vez a cada iteração do laço exterior, isto é, quando o arranjo já se encontrar ordenado.

Complexidade de espaço (pior caso e melhor caso): $\Theta(1)$, pois usamos apenas uma quantidade constante de memória para variáveis auxiliares além da memória fornecida na entrada.

3.4 Merge sort

O merge sort é um algoritmo de divisão e conquista que utiliza intercalação para combinar subarranjos ordenados dois a dois em arranjos maiores também ordenados. O processo de intercalação (merge) pode ser feito de forma eficiente em tempo linear partindo da premissa que os arranjos de entrada são fornecidos ordenados.

```
MERGE-SORT(A)
1  if A.length = 1 or A.length = 0 return
2  mid ← ⌊A.length/2⌋
3  left ← A[1 ··· mid]
4  right ← A[mid + 1 ··· A.length]
5  MERGE-SORT(left)
6  MERGE-SORT(right)
7  MERGE(left, right, A)
```

```
MERGE(left, right, A)
1  i ← 1; j ← 1; k ← 1
2  while i ≤ left.length and j ≤ right.length
3     if left[i] < right[j]
4        A[k] ← left[i]
5        i ← i + 1
6     else
7        A[k] ← right[j]
8        j ← j + 1
9     k ← k + 1
10 while i ≤ left.length
11    A[k] ← left[i]
12    i ← i + 1; k ← k + 1
13 while j ≤ right.length
14    A[k] ← right[j]
15    j ← j + 1; k ← k + 1
```

Complexidade de tempo (pior caso e melhor caso): A recorrência desse algoritmo é a seguinte:

$$T(n) = \begin{cases} 2T(n/2) + \Theta(n) & \text{se } n > 1 \\ \Theta(1) & \text{se n<=1} \end{cases}$$

Pois resolvemos dois subproblemas de metade do tamanho original n ($2T(n/2)$), representando o custo de conquistar; gastamos $\Theta(1)$ para dividir com o problema com uma operação aritmética simples; e gastamos $\Theta(n)$ para criar os dois arranjos intermediários das metades e para intercalá-los (merge).

Complexidade de espaço (pior caso e melhor caso): $\Theta(n)$, por conta dos dois arranjos das metades que precisam ser alocados para as chamadas recursivas.

3.5 Quick sort

Também se trata de um algoritmo de divisão e conquista, mas que gera dois subproblemas de tamanho variável. A ideia é escolher um pivot que servirá como elemento central que particiona os elementos do arranjo em $\leq pivot$ e $> pivot$. Cada uma das metades é então ordenada recursivamente.

```
QUICK-SORT(A, l, r)
1  if l ≥ r return
2  p ← PARTITION(A, l, r)
3  QUICK-SORT(A, l, p − 1)
4  QUICK-SORT(A, p + 1, r)
```

```
PARTITION(A, l, r)
1  pivot ← A[r]
2  i ← l − 1
3  for j ← 1 to r − 1
4     if A[j] ≤ pivot
5        i ← i + 1
6        swap(A, j, i)
7  swap(A, i + 1, r)
8  return i + 1
```

Complexidade de tempo (pior caso): $\Theta(n^2)$, representando duas partições desbalanceadas, uma com zero elementos e uma com todo o restante de elementos menos o pivot. Se isso acontecer, a cada chamada o problema diminui de apenas uma unidade (o pivot).

Complexidade de tempo (melhor caso): $\Theta(n \log n)$, representando um particionamento ideal a cada chamada recursiva, isto é, totalmente balanceado.

Complexidade de espaço: $\Theta(1)$, pois o algoritmo é *in-place*.

$$T(n) = T(n/10) + T(9n/10) + n$$

$$a_1 = 1; b_1 = 1/10; a_2 = 1; b_2 = 9/10; f(n) = n$$

$$1(1/10)^p + 1(9/10)^p = 1 \implies p = 1$$

$$T(n) \in \Theta\left(n^1 + n^1 \int_{n_0}^{n} \frac{x}{x^2} dx\right) = \Theta(n \ln n)$$

3.6 Heap sort

O algoritmo heap sort se baseia em uma estrutura de dados chamada heap, usada para garantir o acesso ao maior (ou menor) elemento com custo constante. O heap máximo pode ser visualizado como uma árvore binária de chaves que mantém a seguinte propriedade: todo nó filho da árvore é menor ou igual ao seu pai. Uma característica importante dessas estruturas heap é que se houver alguma violação de sua propriedade em um único nó, é possível reorganizar seus itens em um heap válido em tempo $\Theta(\log n)$ – isso é feito através do procedimento MAX-HEAPIFY.

```
LEFT(i)
1  return 2i
```

```
RIGHT(i)
1  return 2i + 1
```

```
MAX-HEAPIFY(A, i) // Θ(h)
1  l ← LEFT(i); r ← RIGHT(i)
2  if l ≤ A.heapsize and A[l] > A[i]
3     largest ← l
4  else largest ← i
5  if r ≤ A.heapsize and A[r] > A[largest]
6     largest ← r
7  if largest ≠ i
8     swap(A, largest, i)
9     MAX-HEAPIFY(A, largest)
```

```
BUILD-MAX-HEAP(A) // Θ(n)
1  A.heapsize ← A.length
2  for i ← ⌊A.length/2⌋ downto 1
3     MAX-HEAPIFY(A, i)
```

```
HEAP-SORT(A) // Θ(n log n)
1  BUILD-MAX-HEAP(A)
2  for i ← A.heapsize downto 2
3     swap(A, 1, i)
4     A.heapsize ← A.heapsize − 1
5     MAX-HEAPIFY(A, 1)
```

Complexidade de tempo (pior caso): $\Theta(n \log n)$. A complexidade do algoritmo MAX-HEAPIFY é da ordem da altura do nó na posição i. O procedimento BUILD-MAX-HEAP realiza chamadas ao MAX-HEAPIFY na metade dos nós (a segunda metade sempre serão folhas da árvore binária e portanto, já são heaps válidos!). Por ser uma árvore binária, temos mais nós com altura pequena do que nós com altura grande e portanto, ao somar os custos de MAX-HEAPIFY para cada nó na metade inferior do arranjo, temos um somatório que descresce muito rápido em seus termos, levando a um custo linear $\Theta(n)$. Juntando tudo, o procedimento heap sort realiza $n-1$ chamadas ao procedimento MAX-HEAPIFY e esse custo domina a chamada inicial linear da construção do heap.

Complexidade de espaço: $\Theta(1)$, pois o algoritmo é *in-place*.

3.7 Ordenação em tempo linear

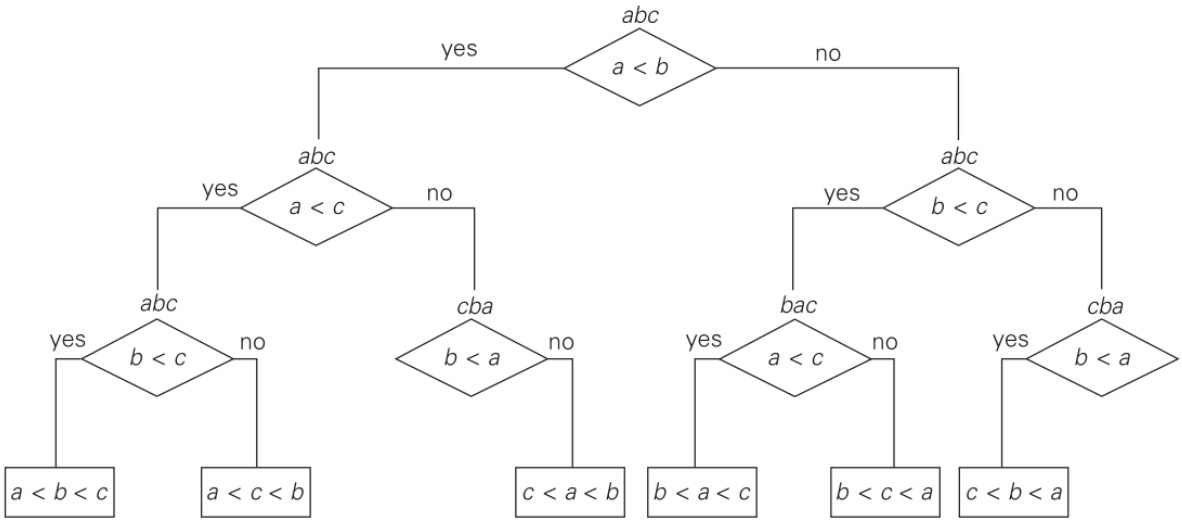
Uma abordagem tradicional em estudo de algoritmos é classificá-los em classes de complexidade assintótica, permitindo a comparação direta entre algoritmos. Uma segunda abordagem complementar é ser capaz de afirmar algo sobre a complexidade de um *problema* frente a um modelo de computação. Em outras palavras, podemos querer responder o seguinte:

> Dados um modelo de computação e um problema, qual é a complexidade mínima que um algoritmo precisa ter para ser correto em solucionar o problema?

Veja que a resposta para essa pergunta seria uma afirmação muito mais forte do que dizer algo sobre a complexidade de algoritmos isoladamente, já que ela envolve considerar todos os infinitos e possíveis algoritmos que solucionam um problema. Chamamos isso de *limites inferiores para problemas*.

3.7.1 Limite inferior para ordenação

Se considerarmos um modelo de computação baseado em operações de comparação, podemos ilustrar a operação de qualquer algoritmo de ordenação através de uma árvore de decisão. Uma árvore de decisão é uma árvore binária onde cada nó representa uma comparação feita em algum momento no algoritmo (operação) e as arestas representam as respostas para essas comparações: verdadeiro ou falso. Dessa forma, um caminho da raíz até uma das folhas dessa árvore representa uma execução do algoritmo, ou o conjunto de testes/decisões que o algoritmo toma dependendo da entrada. Veja um exemplo para um algoritmo que ordena três números:



Podemos generalizar para o problema de ordenação de um arranjo qualquer $[a_1, \cdots, a_n]$ e vamos constatar que as folhas dessa árvore binária de decisão representam possíveis saídas de um algoritmo de ordenação, isto é: $\#folhas \geq n!$, pois cada permutação do arranjo precisa ter uma folha representante, senão o algoritmo seria incorreto!

A altura dessa árvore representa exatamente o número de comparações (operações) realizadas em alguma execução, seja lá qual for o algoritmo. Dessa forma, se estimarmos a altura da árvore estaremos na verdade estimando o custo mínimo que um algoritmo de ordenação precisa ter para resolver o problema corretamente para qualquer instância de entrada. Sabemos também que a altura h de uma árvore binária com k folhas é, pelo menos, $\log_2 k$:

$$
\begin{aligned}
\text{Sabemos que: } & h \geq \log_2(\#folhas), \#folhas \geq n! \\
\text{Aplicando log e combinando: } & h \geq \log_2(\#folhas) \geq \log_2(n!) \\
\text{Desenvolvendo: } & h \geq \log_2(n!) = \log_2(n) + \log_2(n-1) + \cdots + \log_2(1) \\
& \geq \sum_{i=1}^{n} \log_2(i) \geq \sum_{i=n/2}^{n} \log_2(i) \geq \sum_{i=n/2}^{n} \log_2(n/2) \\
& = \sum_{i=n/2}^{n} \log_2(n/2) - \sum_{i=n/2}^{n} \log_2(2) \\
& = (n/2+1)\log_2(n/2) - (n/2+1) \in \Omega(n \log_2 n)
\end{aligned}
$$

Conclusão: Não pode existir nenhum algoritmo para o modelo de computação baseado em comparações que tenha um custo assintótico melhor do que $\Omega(n \log n)$.

3.7.2 Counting sort

A primeira etapa é alocar um arranjo de tamanho $k+1$ que será usado para contar o número de ocorrências de cada elemento. Com isso, vamos acumular nas posições de C de maneira que $C[i]$ contenha o número de elementos $\leq i$, dessa forma podemos identificar para cada elemento de A a posição exata que ele deve ocupar no arranjo ordenado B. Com C iremos então, para cada elemento de A em sua ordem inversa, colocá-lo na sua posição correta em B. Toda vez que adicionamos um novo elemento em sua posição correta precisamos decrementar a quantidade de elementos menores ou iguais àquele, para que repetições de um mesmo elemento possam ser posicionadas em suas posições corretas também.

```
COUNTING-SORT(A, B, k)
 1  C ←"novo arranjo com k + 1 posições: 0 ··· k"
 2  for i ← 0 to k
 3      C[i] ← 0
 4  for j ← 1 to A.length
 5      C[A[j]] ← C[A[j]] + 1
 6  for i ← 1 to k
 7      C[i] ← C[i] + C[i − 1]
 8  for j ← A.length downto 1
 9      B[C[A[j]]] ← A[j]
10      C[A[j]] ← C[A[j]] − 1
```

3.7.3 Radix sort

3.7.4 Bucket sort

Capítulo 4

Técnicas de Projeto de Algoritmos

4.1 Projeto de algoritmos usando Força-bruta

Força-bruta representa o projeto de algoritmos mais direto e que utiliza apenas as definições do próprio problem, resultando muitas vezes em algoritmos mais ingênuos e até ineficientes. Apesar disso, a importância de um algoritmo de força-bruta é expressivo já que seria a primeira estratégia de solução que um projetista de algoritmos utilizaria para estabelecer um limite inferior para o quão melhorado o projeto de algoritmos para aquele problema pode ser.

4.1.1 Estudo de caso: Selection sort

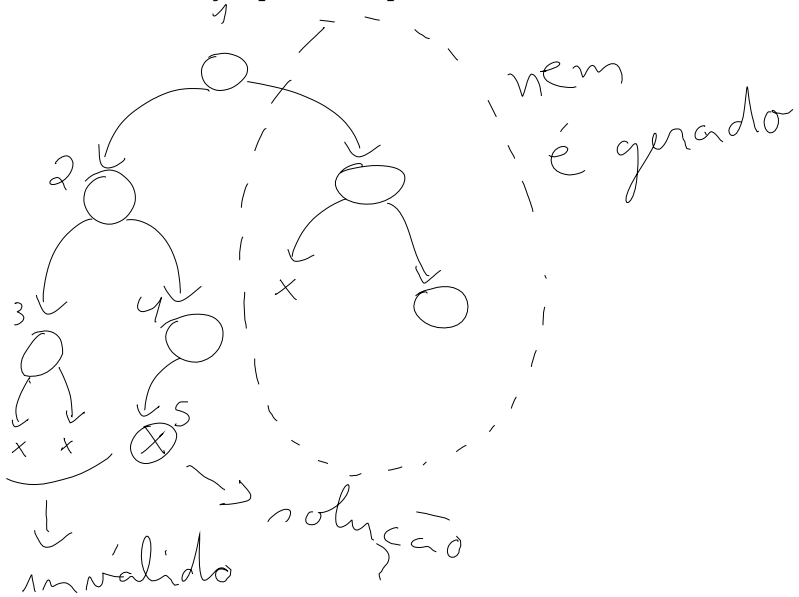
```
SELECTION-SORT(a)
 1  for i ← 1 to n − 1
 2      menor ← i
 3      for j ← i + 1 to n
 4          if a[j] < a[menor] menor ← j
 5      swap(a, i, menor)
```

4.1.2 Estudo de caso: Problema da mochila

```
KNAPSACK-BRUTE-FORCE(w, v, W)
 1  items ← [1, 2, ··· , n] // índices dos itens
 2  subsets ←COMBINATIONS(items)
 3  bestSubset ← −1
 4  bestCost ← −∞
 5  for i ← 1 to |subsets|
 6      totalWeight ←SUM-WEIGHTS(subsets[i])
 7      if totalWeight <= W
 8          totalCost ←SUM-VALUES(subsets[i])
 9          if totalCost > bestCost
10              bestCost ← totalCost
11              bestSubset ← i
12  return subsets[bestSubset]
```

4.2 Projeto de algoritmos usando *Backtracking*

Ao invés de gerar todas as soluções candidatas e testá-las uma a uma como na força-bruta, algoritmos de backtracking (tentativa e erro) constrói o espaço de soluções gradualmente. Soluções são testadas assim que geradas e soluções não-exploradas que forem inviáveis nem precisam ser geradas. Geralmente este paradigma é utilizado em problemas que não sejam de otimização, isto é, quando estamos procurando alguma solução para o problema (conhecido também como localização). A exploração do espaço de busca acontece na maioria dos casos usando uma busca em profundidade em uma árvore onde cada nó representa uma solução parcial do problema.



4.2.1 Estudo de caso: problema das N rainhas

Vamos assumir um tabuleiro de xadrez $n \times n$ e queremos posicionar n rainhas sem que elas se ataquem. Como apenas uma rainha pode ser posicionada em cada linha, nosso objetivo é gerar como saída um arranjo $m[1..n]$ onde $m[i]$ indica a coluna na qual a rainha da linha i foi posicionada. Em um algoritmo de backtracking, começamos com o tabuleiro vazio e a toda chamada recursiva posicionamos uma rainha em uma nova linha. Se o algoritmo conseguir chegar e posicionar com sucesso a rainha da última linha, isso indicará que encontramos uma solução.

```
N-QUEENS-BACKTRACKING(m, r)  r é a nova linha que precisa de rainha
 1  if r = n + 1 return m
 2  for j ← 1 to n
 3     valid ← true
 4     for i ← 1 to r − 1
 5        if m[i] = j or m[i] = j + (r − i) or m[i] = j − (r − i)
 6           valid ← false
 7     if valid
 8        m[r] ← j
 9        N-QUEENS-BACKTRACKING(m, r + 1)
```

4.3 Projeto de algoritmos usando estratégias gulosas

Estratégias gulosas para projeto de algoritmos são úteis em diversas áreas de estudo, desde algoritmos exatos até algoritmos aproximativos e heurísticas. Um algoritmo guloso é caracterizado por uma sequência de *escolhas gulosas* feitas com o objetivo de construir iterativamente uma solução final para um problema. Tal solução final geralmente é representada como um conjunto de itens definidos de acordo com o problema em questão. Uma escolha para ser gulosa deve possuir as seguintes características:

1. **Localmente ótima:** dentre as opções de escolha em um dado momento, precisa decidir incluir na solução aquele item que demonstrar o melhor benefício imediato. Naturalmente, diversos critérios podem ser usados para essa avaliação, o que nos dá margem para produzir mais de uma escolha gulosa para um problema;
2. **Factível:** não deve violar nenhuma restrição do problema;
3. **Irrevocável:** uma vez tomada, a escolha gulosa nunca mais será desfeita.

Como escolhas do algoritmo não são desfeitas, algoritmos gulosos tendem a ser eficientes em tempo mas nem sempre levam à solução ótima global. Quando isso acontece, temos um procedimento que pode ser um algoritmo aproximativo ou apenas uma heurística gulosa. Por hora vamos focar nos problemas para os quais existem algoritmos gulosos ótimos que os solucionam.

Para um dado problema e algoritmo guloso, duas propriedades devem ser garantidas para que a otimalidade da estratégia seja garantida:

- **Propriedade da escolha gulosa:** existe sempre alguma solução ótima para qualquer instância do problema que contém a escolha gulosa do algoritmo.
- **Propriedade da subestrutura ótima:** um problema atende essa propriedade se o mesmo puder ser decomposto em subproblemas e a solução ótima para o problema original puder ser construída a partir de soluções ótimas desses subproblemas.

Essas duas propriedades se verificadas em conjunto, simplificam a tarefa de demonstrar que um algoritmo guloso é ótimo. De fato, a outra alternativa seria demonstrar de forma indutiva que alguma invariante do algoritmo se mantém e garante a resposta correta – abordagem mais genérica, porém geralmente mais desafiadora.

4.3.1 Estudo de caso: problema do troco

4.3.2 Estudo de caso: problema do escalonamento de tarefas

Cada tarefa i é representada por um início s_i e um término f_i. O objetivo é escalonar um conjunto de tarefas que não se sobrepõem (são compatíveis).

```
ESCALONA-COMPATIVEIS(s, f)
 1  seja a = [1, 2, ··· , n] um arranjo de índices de tarefas
 2  ordene os índices i em a usando como chave f_i
 3  T ← ∅
 4  i ← 1
 5  while i ≤ n
 6     m ← a[i]
 7     T ← T ∪ {m}
 8     i ← i + 1
 9     while i ≤ n and s_{a[i]} < f_m ⟨⟨incompatíveis⟩⟩
10        i ← i + 1
11  return T
```

O algoritmo acima é ótimo e, para garantir isso, precisamos verificar as duas propriedades pertinentes: (1) escolha gulosa; e (2) subestrutura ótima. Via de regra, com essas duas propriedades o algoritmo será automaticamente válido por indução: (1) garante que é sempre seguro fazer a escolha gulosa e ainda assim encontrar alguma solução ótima, isto é, passo base; e (2) garante que é seguro fazer a escolha gulosa a todo passo, para todos os subproblemas que forem sobrando, isto é, passo indutivo. Vamos verificar cada uma das propriedades a seguir:

Propriedade da escolha gulosa

Como o conjunto de itens que compõem uma solução são exatamente as tarefas escolhidas, vamos representar o problema de escalonamento como um conjunto de todas essas tarefas candidatas. Portanto, seja $P = \{1, \cdots, n\}$ o problema de escalonar essas n tarefas. Assim, quando o algoritmo realiza a escolha gulosa (a tarefa m de menor término), o subproblema restante a se resolver é: $P' = \{i \in P \mid s_i \geq f_m\}$, isto é, o restante de tarefas compatíveis com m. Com isso, acabamos de decompor o problema original em subproblemas a partir da escolha gulosa do algoritmo. Da mesma forma, uma solução para P que faz a escolha gulosa pode ser indicada como $S_P = \{m\} \cup S_{P'}$.

Essa decomposição inicial é fundamental pois, de acordo com a propriedade da escolha gulosa, deveríamos ser capazes de transformar *qualquer* solução genérica ótima S'_P em uma solução de igual qualidade S_P que necessariamente faça a escolha gulosa (**não confundir $S_{P'}$ com S'_P**). Quase sempre, o argumento para garantir essa transformação é incluir a escolha gulosa na solução genérica (uma troca) de maneira a garantir a manter a otimalidade da solução.

Para o caso do escalonamento de tarefas, queremos transformar S'_P em S_P. Para isso, seja m a escolha gulosa e seja j a tarefa de menor término em S'_P. Pense sobre isso, m termina primeiro em P e j termina primeiro em uma solução genérica de P, denotada como S'_P:

1. Se $m = j$, a solução genérica já faz a escolha gulosa. Portanto, $S_P = S'_P$.
2. Se $m \neq j$, só pode ser que $f_j \geq f_m$. Dessa forma, qualquer tarefa que seja compatível com j também será compatível com m. Assim, trocar j por m em S'_P não altera a qualidade nem a viabilidade da solução. Portanto, $S_P = (S'_P \setminus \{j\}) \cup \{m\}$.

Veja que em ambos os casos, a qualidade da solução não mudou porque a cardinalidade dos conjuntos permaneceu a mesma. Além disso, em ambos os casos, garantimos que m pertence à solução ótima. □

Propriedade da subestrutura ótima

Subestrutura ótima tem tudo a ver com como problemas são decompostos em subproblemas. No caso de algoritmos gulosos, *sempre teremos apenas um subproblema restante*. Já fizemos essa decomposição na seção anterior, considerando a escolha gulosa que agora é segura: $S_P = \{m\} \cup S_{P'}$. Isso nos diz, que solução para P inclui a escolha gulosa mais alguma solução para o subproblema restante P'.

A subestrutura ótima lida justamente com a questão: será que $S_{P'}$ precisa ser ótima para que S_P o seja? Se a resposta for sim, o problema atende a propriedade da subestrutura ótima. Caso contrário, não atende. Veja que no caso de atender, estamos implicitamente afirmando que soluções ótimas para problemas são compostas de soluções também ótimas de seus respectivos subproblemas.

Para o problema do escalonamento, isso é facilmente verificado por contradição. Suponha que S_P seja uma solução ótima para P, mas que $S_{P'}$ não seja ótima para P'. Dessa forma, deve existir uma solução ótima para P', denotada como $S'_{P'}$. Segue dessa definição que $|S'_{P'}| > |S_{P'}|$. Com isso, poderíamos construír uma solução para P de melhor qualidade:

$$S'_P = \{m\} \cup S'_{P'}, \text{ de cardinalidade } |S'_P| = 1 + |S'_{P'}| > |S_P| = 1 + |S_{P'}|.$$

Isso contradiz a premissa de que S_P é ótima. Conclusão, $S_{P'}$ precisa ser ótima também. Juntando este resultado com o resultado da propriedade da escolha gulosa, concluímos que o algoritmo que faz a escolha gulosa da tarefa de menor término primeiro no problema do escalonamento é ótimo. □

4.3.3 Estudo de caso: árvore geradora mínima

4.3.4 Estudo de caso: compressão de documentos

Seja um documento formado por caracteres, nosso objetivo será encontrar uma forma de comprimí-lo de forma ótima. Por exemplo, seja o seguinte documento:

```
escolha gulosa e subestrutura otima
```

Se ignorarmos os espaços apenas para facilitar o exemplo, percebemos que é muito comum que a distribuição das frequências de caracteres seja bem desbalanceada. Para o exemplo acima:

a	b	c	e	g	h	i	l	m	o	r	s	t	u
4	1	1	3	1	1	1	2	1	3	2	4	3	4

Olhando para essas frequências, fica evidente que se representarmos cada caractere usando a mesma quantidade de bits, espaço seria desperdiçado com os caracteres que acontecem raramente. Assim, definimos este problema de compressão como: dados um conjunto de caracteres que ocorrem em um documento e suas respectivas frequências, queremos encontrar uma codificação para cada caratere que minimize a quantidade de bits necessária para representar o documento inteiro. Por conveniência, vamos representar essas codificações através de uma árvore binária de prefixos. Considerando um outro exemplo:

a	b	c	d	e	f
45	13	12	16	9	5

Em uma codificação de tamanho fixo, como temos apenas 6 caracteres no documento, 3 bits seriam suficientes, já que $2^3 = 8 \geq 6$:

Incluir figura ...

Veja que com essa representação, temos a oportunidade de encontrar outras codificações melhores: possivelmente as que utilizem menos bits para caracteres mais frequentes e mais bits para caracteres menos frequentes. O objetivo do problema é portanto encontrar uma árvore codificadora T como esta que minimize:

$B(T) = \sum_{c \in C} freq(c) * d_T(c)$, onde *freq* representa a frequência dos caracteres e d_T representa a profundidade do caractere na árvore, isto é, quantos bits são utilizados para representá-lo.

O algoritmo a seguir implementa a codificação de Huffman, que é uma estratégia gulosa ótima para o problema:

```
HUFFMAN(c, f)
 1  construa uma fila de prioridade Q com os itens c(i) e prioridades f(i)
 2  for i ← 1 to n − 1
 3     l ←EXTRACT-MIN(Q)
 4     r ←EXTRACT-MIN(Q)
 5     seja z um novo nó de uma árvore binária
 6     LEFT-CHILD(z) ← l
 7     RIGHT-CHILD(z) ← r
 8     f(z) ← f(l) + f(r)
 9     INSERT(Q, z, f(z))
10  return EXTRACT-MIN(Q)
```

4.4 Projeto de algoritmos usando Programação Dinâmica

Este é um paradigma de projeto de algoritmos também aplicado a problemas de otimização. Partimos da observação de que muitas vezes, o processo de decomposição de um problema em subproblemas de forma recursiva pode gerar repetidas vezes subproblemas idênticos. Esse comportamento que aparece em diversos problemas de otimização que exibem *subestrutura ótima* é conhecido como *sobreposição de subproblemas*. Portanto, essas são as características que um problema precisa exibir para que seja passível de aplicar estratégias de programação dinâmica: subestrutura ótima e sobreposição de subproblemas.

4.4.1 Ilustrando os princípios de programação dinâmica através de algoritmos para a sequência de Fibonacci

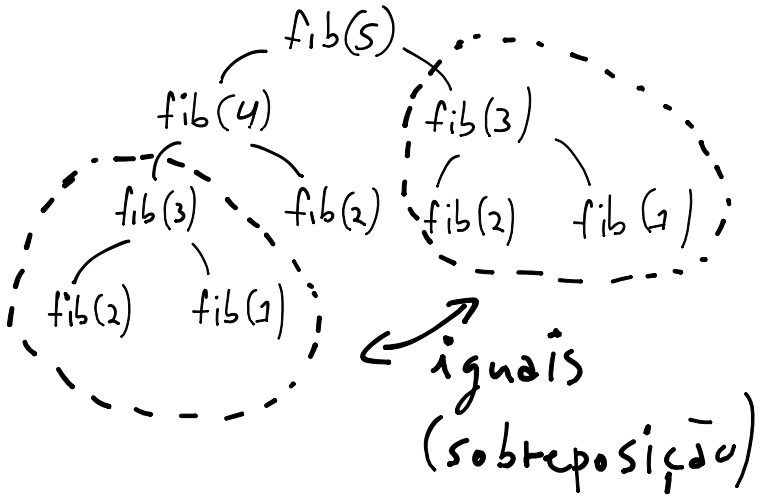
A seguinte função recursiva define o valor do n-ésimo elemento da sequência de Fibonacci:

$$fib(n) = \begin{cases} 1 & \text{se } n = 1 \text{ ou } n = 2 \\ fib(n-1) + fib(n-2) & \text{se n>2} \end{cases}$$

Observe que uma definição como essa nos permite, imediatamente, uma algoritmo recursivo para o problema:

```
FIB(n)
1  if n = 1 or n = 2
2     f ← 1
3  else
4     f ← FIB(n − 1) + FIB(n − 2)
5  return f
```

Ao utilizar o algoritmo acima para calcular o quinto elemento da sequência, a seguinte árvore de recursão representaria a execução do algoritmo, indicando a existência de sobreposição entre subproblemas:



Podemos analisar o algoritmo recursivo por exemplo, em relação ao número de somas. A seguinte recorrência representa esse custo em tempo:

$$T(n) = \begin{cases} T(n-1) + T(n-2) + 1 & \text{se } n > 2 \\ 0 & \text{se n=1 ou n=2} \end{cases}$$

A resolução da recorrência revela que o custo do algoritmo é exponencial. Na verdade, como a árvore de recursão para o custo dessa recorrência é desbalanceada podemos encontrar um limite inferior para o custo. Sabemos que sempre um dos ramos da árvore alcançará as folhas mais rapidamente ($T(n-2)$) e outro demorará mais para alcançar as folhas ($T(n-1)$). Na verdade, a diferença entre essas duas profundidades é exatamente a metade da altura total da árvore, já que um decresce de um-em-um e outro decresce de dois-em-dois (duas vezes mais rápido). Sendo assim, das profundidades 0 até $\lfloor n/2 \rfloor$ com certeza a árvore permanece completa. Portanto, metade dessa árvore de custos nos dá um limite inferior para a complexidade:

$$T(n) \geq \sum_{i=0}^{n/2} 2^i = 2^{n/2+1} - 1 \in O(2^n)$$

A sobreposição de subproblemas nos permite "memorizar" soluções já encontradas. Veja o seguinte algoritmo modificado:

```
FIB-MEMO(n)
 1  a[1, ··· , n] ← [0, ··· , 0]
 2  return FIB-MEMO-REC(n, a)
FIB-MEMO-REC(n, a)
 3  if a[n] ≠ 0
 4     return a[n]
 5  if n = 1 or n = 2
 6     f ← 1
 7  else
 8     f ← FIB(n − 1) + FIB(n − 2)
 9  a[n] ← f
10  return f
```

Esse é a primeira técnica utilizada em todo algoritmo de programação dinâmica: memorização. Entretanto, nem toda memorização é um algoritmo de programação dinâmica. Para configurar um algoritmo de programação dinâmica, precisamos ter um algoritmo que memoriza soluções de subproblemas mas que ao mesmo tempo resolve os subproblemas de forma deliberada, isto é, utilizando uma ordem específica. Essa ordem sempre parte de subproblemas menores para subproblemas maiores e, portanto, dizemos que programação dinâmica é memorização feita de forma *bottom-up*. Para o problema estudado, temos a seguinte algoritmo com todas essas características:

```
FIB-PD(n)
1  a[1,⋯,n] ←"novo arranjo com posições 1⋯n"
2  a[1] ← 1
3  a[2] ← 1
4  for i ← 3 to n
5     a[i] ← a[i−1] + a[i−2]
6  return a[n]
```

4.4.2 Estudo de caso: corte de hastes

Entrada: Inteiros $p_1, p_2, \cdots, p_n$ representando preços de corte de uma haste de tamanho n; Saida: preço de venda do corte ótimo.

Seja r_n o preço de venda ótimo, vamos representar a equação de custos de uma solução ótima:

$$r_n = \max(\text{todas as possibilidades de se cortar haste de tamanho } n)$$

$$r_n = \max(\text{preço de não se cortar a haste}, \text{preço total de cortar a haste em cada uma das } i \text{ posições})$$

$$r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \cdots, r_{n-1} + r_1)$$

$$r_n = \max_{1 \le i \le n}(p_i + r_{n-i}); r_0 = 0$$

4.4.3 Estudo de caso: problema da mochila

Seja $c(i, j)$ o custo ótimo de uma solução para o problema da mochila que considerou até o item i ($0, \cdots, i$, com 0 representando que nenhum item foi considerado até então) para uma mochila de capacidade j:

$$c(i,j) = \begin{cases} 0 & \text{se } i = 0 \text{ ou } j = 0 \\ c(i-1,j) & \text{se } w_i > j \text{ // não cabe mais nada nessa submochila} \\ \max(c(i-1,j), v_i + c(i-1, j-w_i)) & \text{se } w_i \le j \text{ // o item cabe na mochila e portanto temos essa opção} \end{cases}$$

$$i : 0 \cdots n, j : 0 \cdots W$$

4.4.4 Estudo de caso: máxima subsequência crescente

Vamos assumir que o arranjo começa de 0 e que nessa posição contém $-\infty$ para indicar o último elemento de uma subsequência vazia.

Seja $c(i, j)$ o custo ótima de uma solução ótima para a subsequência $a[j..n]$ com todos os elementos sendo maiores do que $a[i]$. Isto é, se $a[i]$ for o primeiro elemento da subsequência, podemos contatenar

com esse elemento $a[i]$ uma subsequência crescente de $a[j..n]$ desde que todos os seus elementos sejam maiores do que $a[i]$, para garantir a propriedade de estritamente crescente:

$$c(i,j) = \begin{cases} 0 & \text{se } i = 0 \text{ ou } j > n \\ c(i,j+1) & \text{se } a[j] \leq a[i] \\ \max(c(i,j+1), 1 + c(j,j+1)) & \text{se } a[j] > a[i] \end{cases}$$

Resultado estará em $c(0,1)$: maior subsequência de $a[1..n]$ com todos os elementos maiores do que $a[0] = -\infty$.

4.5 Branch-and-Bound

O objetivo desta técnica de projeto de algoritmos é ser capaz de melhorar (não assintoticamente) um algoritmo de backtracking em relação a tempo e para problemas de otimização. Por ser de otimização, as soluções para uma instância podem ser classificadas em: factíveis sse ela não viola nenhuma das restrições do problema; e ótimas sse são factíveis e também de melhor qualidade em relação à função objetivo. Em linhas gerais, a ideia será a de explorar a árvore do espaço de busca de maneira a podar o máximo de nós. Para isso, um algoritmo de branch-and-bound precisa manter a seguinte informação:

- Melhor solução: o custo da melhor solução observada até então;
- Potencial de um nó: para cada nó, precisamos de uma forma de determinar o quão boa a solução obtida a partir dali pode se tornar;

Observe que o potencial de um nó não representa necessariamente que existirá uma solução a partir daquele nó que tem o custo do potencial – apenas significa que não tem como obter melhor solução do que aquilo a partir dali. Outra observação é que a noção do que é melhor ou pior depende de ser um problema de maximização ou minimização: para minimização o potencial é um *limite inferior*, para maximização o potencial é um *limite superior*.

Com isso, um algoritmo de branch-and-bound pode podar um nó de um espaço de busca sempre que: (1) o nó representar uma solução não factível; (2) o nó representa um ponto onde nenhuma escolha pode ser feita, isto é, a solução candidata está completa; **(3) o potencial de um nó não é melhor do que a melhor solução observada até então**. Para que essa estratégia tenha vantagens em relação ao bactracking, precisamos então sempre manter os nós ativos da árvore de busca a todo momento para então sempre escolher expandir o nó de melhor potencial (best first). Nós ativos são aqueles nós folha que ainda não foram podados.

```
BRANCH-AND-BOUND(b)
 1  seja S* ← NIL uma solução ótima inicialmente nula
 2  seja N ← {∅} o conjunto de nós ativos (soluções parciais)
 3  enquanto N ≠ ∅
 4     seja S ∈ N a solução com o melhor potencial b(S)
 5     para cada item viável i a partir de S
 6        S' ← S ∪ {i} ⟨⟨novo nó⟩⟩
 7        se S' não puder ser expandida (solução completa)
 8           se S* = NIL ou b(S') for melhor do que b(S*)
 9              S* ← S'
10              remova de N todo nó S'' tal que b(S'') é pior do que b(S*)
11        senão
12           N ← N ∪ {S'}
13  retorne S*
```

4.5.1 Problema da mochila

Limite superior para o custo de soluções parciais: o potencial máximo de ganho a partir da escolha de alguns itens. Considerando v' e w' como sendo o que se tem de custo e benefício em um dado momento: $ub = v' + (W - w')(v_{i+1}/w_{i+1})$, isto é, o benefício que já temos mais o máximo de custo benefício para cada unidade de peso restante da mochila.

```
MOCHILA-BRANCH-AND-BOUND(S, v', w', i) ⟨⟨assumir w, v, W como entradas do problema⟩⟩
 1  se solução atual é inviável, retorne a solução indicando pior potencial possível
 2  se todos os itens foram considerados, retorne a solução
 3  gere duas soluções parciais: (1) adiciona item i à solução; (2) não adiciona i à solução
 4  determine os custos e benefícios parciais (v'/w') e os limites superiores (ub) de cada solução gerada
 5  seja (A) o nó (solução gerada) com maior potencial
 6  seja (B) o nó (solução gerada) com menor potencial
 7  chame recursivo MOCHILA-BRANCH-AND-BOUND com o nó (A)
 8  se o limite superior de (B) for ≤ à solução de (A)
 9     retorne solução de (A) como melhor
10  senão
11     chame recursivo MOCHILA-BRANCH-AND-BOUND com o nó (B)
12     retorne melhor entre as soluções de (A) e (B)
```

4.5.2 Problema da alocação

O problema da alocação recebe como entrada um conjunto de trabalhadores $w_1, w_2, \cdots, w_n$, um conjunto de tarefas $t_1, t_2, \cdots, t_n$ e um custo c_{ij} para cada alocação $w_i \rightarrow t_j$. Uma tarefa só pode ser atribuída a um único trabalhador e vice versa. O objetivo é encontrar o conjunto de n alocações que minimize a soma dos custos. Vamos representar uma instância do problema como uma matriz $n \times n$ onde linhas representam os trabalhadores e colunas representam as tarefas. Por exemplo:

$$\begin{bmatrix} 9 & 2 & 7 & 8 \\ 6 & 4 & 3 & 7 \\ 5 & 8 & 1 & 8 \\ 7 & 6 & 9 & 4 \end{bmatrix}$$

Veja que agora o objetivo é encontrar, para cada linha, uma coluna não escolhida por nenhuma outra linha.

4.6 Programação Linear

```
SIMPLEX(A, b, c) ⟨⟨na forma padrão⟩⟩
 1  faça a conversão do programa para a forma relaxada: (N, B, A, b, c, v)
 2  seja b = (x̄_1, x̄_2, ···) a solução básica obtida ao atribuir 0 a cada variável não-básica
 3  enquanto existir coeficiente positivo (c_i) na função objetivo
 4     selecione na função objetivo uma variável não-básica x_e ∈ N cujo coeficiente (c) seja positivo
 5     selecione a restrição da variável básica x_l ∈ B cujo coeficiente de x_e seja negativo e o menor possível
 6     isole x_e na restrição da variável básica x_l
 7     substitua x_e nas outras equações e função objetivo
 8     neste ponto x_e se tornou básica e x_l, não-básica
 9     atualize (N, B, A, b, c, v) e a solução básica b = (x̄_1, x̄_2, ···)
10  retorne solução b = (x̄_1, x̄_2, ···) de custo v
```

Capítulo 5

Algoritmos em Grafos

5.1 Caminhamento em largura (Breadth-first search - BFS)

Caminhamento em largura geralmente é usado para encontrar distâncias mínimas de um vértice a todos os outros em relação ao *nuúmero de arestas*.

```
BFS(G(V,E),s)
 1  for u in V \ {s}
 2     u.π ← NIL
 3     u.d ← ∞
 4     u.color ← WHITE
 5  s.color ← GRAY
 6  s.d ← 0
 7  s.π ← NIL
 8  Q ←CREATE-QUEUE({s})
 9  while Q ≠ ∅
10     u ←DEQUEUE(Q)
11     for v in G.AdjList[u]
12        if v.color = WHITE
13           v.color ← GRAY
14           v.π ← u
15           v.d ← u.d + 1
16           ENQUEUE(v)
17     u.color ← BLACK
```

5.2 Caminhamento em profundidade (Depth-first search - DFS)

Caminhamento em profundidade tem uma utilidade adicional que é ser usado como subrotina para outros algoritmos. De fato, podemos descobrir propriedades interessantes do grafo a partir de um caminhamento em profundidade que registre para cada vértice a ordem em que o mesmo foi descoberto ($u.d$) e finalizado ($u.f$). No intervalo de tempo em que um vértice está descoberto mas não finalizado o mesmo possui a cor *GRAY*.

```
 1  time ← 0 // variável global que marca o timestamp
DFS(G(V,E))
 2  for u in V
 3     u.color ← WHITE
 4     u.π ← NIL
 5  for u in V
 6     if u.color = WHITE
 7        DFS-VISIT(G(V,E),u)
DFS-VISIT(G(V,E),u)
 8  time ← time + 1
 9  u.d ← time
10  u.color ← GRAY
11  for v in G.AdjList[u]
12     if v.color = WHITE
13        v.π ← u
14        DFS-VISIT(G,v)
15  u.color ← BLACK
16  time ← time + 1
17  u.f ← time
```

Parentização de intervalos em DFS. Uma propriedade importante do caminhamento em profundidade diz respeito a como os intervalos de dois vértices u e v se comparam:

- O intervalo $[u.d, u.f]$ está totalmente contido dentro do intervalo $[v.d, v.f]$ sse u é descendente de v em alguma arvore na floresta de predecessores do DFS (ou vice versa).
- O intervalo $[u.d, u.f]$ é totalmente disjunto de $[v.d, v.f]$ sse os dois vértices pertencem a árvores diferentes na floresta de predecessores do DFS.
- **Não é possível** que intervalos tenham sobreposição mas que um não esteja totalmente contido dentro do outro: se isso pudesse acontecer, teríamos que o vértice ancestral seria terminado antes do seu descendente, o que contraria a definição de caminhamento em profundidade já que vértices ancestrais só terminam quando todos os seus descendentes tenham terminado.

5.2.1 Classificação de arestas em DFS

Se considerarmos o caso mais geral com grafos direcionados, podemos classificar as arestas de um grafo em quatro categorias:

1. Arestas da árvore (tree edges) são aquelas que aparecem na floresta de predecessores – aquelas que são selecionadas para visitar novos vértices;
2. Arestas de volta (back edges) são aquelas que vão de um descendente para algum de seus ancestrais na árvore DFS;
3. Arestas avançadas (forward edges) são aquelas que vão de um vértice até algum de seus descendentes na árvore DFS;

4. Arestas de cruzamento (cross edges) são as outras arestas: pode acontecer entre vértices de diferentes árvores da floresta ou mesmo entre vértices da mesma árvore que não compartilham relação ancestral/descendente (irmãos, por exemplo);

Toda vez que visitamos um novo vértice v no DFS, digamos através da aresta (u, v), temos alguma informação sobre qual tipo de aresta ela será. Se a cor de v for $WHITE$, então a aresta (u, v) é uma aresta da árvore (tree edge). Se a cor do vértice v for $GRAY$, então a aresta (u, v) é de volta (back). Se a cor de v for $BLACK$, então a aresta pode ser avançada (forward) ou de cruzamento (cross). Podemos tirar essa última dúvida se a aresta é avançada ou de cruzamento ao observar os tempos de cada aresta: se além de ser $BLACK$, o intervalo de u for disjunto do intervalo de v então temos uma aresta de cruzamento sem relação ancestral/descendente; caso contrário, se os intervalos estiverem contido um dentro do outro, temos uma aresta avançada, já que a sobreposição indica ancestralidade (pela parentização).

5.2.2 Ordenação topológica

Ideia é que vértices que terminam primeiro devem ser posicionados depois de todos os seus ancestrais na ordenação topológica.

TOPOLOGICAL-SORT$(G(V, E))$ // $\Theta(V + E)$
1 chame DFS(G), anotando os tempos dos vértices
2 a cada término de vértice, adicione-o ao início de uma lista ligada
3 retorne a lista ligada de vértices

5.2.3 Encontrando componentes fortemente conectados

A ideia aqui é que ao executar o DFS em G^T, sempre visitamos primeiro aqueles vértices de maior tempo de término em G, ou seja, aqueles vértices que não possuem aresta para outros vértices posicionados antes na ordem topológica. Dessa forma, toda vez que um vértice branco é encontrado, podemos ter certeza de que ele pertence à componente atual e toda vez que encontramos vértices já terminados, sabemos se tratar de vértices cuja componente já determinamos em uma visita anterior.

STRONGLY-CONNECTED-COMPONENTS(G) $\langle\langle$*complexidade:* $O(V + E)\rangle\rangle$
1 chame DFS(G) para determinar os tempos de término de cada vértice do grafo
2 construa G^T, que representa G com suas arestas invertidas
3 chame DFS(G^T), visitando primeiro os vértices com maior tempo de término (linha 1)
4 cada árvore da floresta visitada pelo último caminhamento representa um componente fortemente connectado do grafo original

5.3 Árvores geradoras de custo mínimo

5.3.1 Algoritmo de Prim

Fazemos $|V|$ operações de EXTRACT-MIN, uma para cada vértice, totalizando $O(V \log V)$. Além disso, no pior caso, as chaves dos vértices são atualizadas $O(E)$ vezes, já que fazemos isso para as listas de adjacências do grafo. Como cada operação DECREASE-KEY custa $O(\log V)$, temos o total de $O(V \log V + E \log V) = O(E \log V)$. Usando Fibonacci heaps, podemos melhorar esse custo assintótico para $O(E + V \log V)$: isso acontece porque nessa estrutura, o custo amortizado de V operações EXTRACT-MIN continua sendo $O(\log V)$, mas o custo amortizado de $O(E)$ operações DECREASE-KEY fica reduzido para $O(1)$ (ao invés de $O(\log V)$ no heap tradicional).

```
MST-PRIM(G(V,E), w, r)
 1  for u in V
 2     u.π ← NIL
 3     u.key ← ∞
 4  r.key ← 0
 5  Q ←BUILD-HEAP(V) ⟨⟨chave u.key⟩⟩
 6  while Q ≠ ∅
 7     u ←EXTRACT-MIN(Q)
 8     for v in G.AdjList[u]
 9        if v ∈ Q and w(u,v) < v.key
10           v.key ← w(u,v) ⟨⟨redução de prioridade: decrease-key⟩⟩
11           v.π ← u
```

5.3.2 Algoritmo de Kruskal

O algoritmo faz V operações MAKE-SET e E operações FIND-SET/UNION. O custo dessas operações usando uma estrutura de dados para union/find é uma função que cresce bem lentamente e é limitada superiormente por $O(\log V)$. Portanto, o custo total do algoritmo é $O((V+E)\log V)$. Mas como em um grafo conectado temos $E \geq V-1$, totalizando $O(E \log V)$.

```
MST-KRUSKAL(G(V,E), w)
 1  A ← ∅
 2  for u in V
 3     MAKE-SET(u)
 4  E' ←SORT-BY-WEIGHT-NONDECREASING(E)
 5  for (u,v) in E' ⟨⟨em ordem⟩⟩
 6     if FIND-SET(u) ≠ FIND-SET(v)
 7        A ← A ∪ {(u,v)}
 8        UNION(u,v)
```

5.4 Propriedades de caminhos mínimos

Desigualdade triangular $\delta(s,v) \leq \delta(s,u) + w(u,v)$: o menor caminho entre s e v não pode ser maior do que o menor caminho entre s e u mais o custo da aresta (u,v), senão teríamos um caminho de custo menor para v (contradição).

Propriedade do limite superior é sempre verdade que a estimativa de menor caminho para v, $v.d$, é sempre maior do que o menor caminho real até v: $\delta(s,v) \leq v.d$.

Propriedade do não-caminho se não existir caminho entre dois vértices (grafo orientado), temos que $\delta(s,v) = \infty$.

Propriedade da convergência seja um caminho $s \rightsquigarrow u \rightarrow v$, se relaxamos a aresta (u,v) depois de que já encontramos o caminho mínimo para u (isto é, $u.d = \delta(s,u)$), então vamos encontrar o caminho mínimo para u: $v.d = \delta(s,v)$ e mais importante, essa estimativa ótima nunca vai se alterar mais.

Propriedade do relaxamento de um caminho seja um caminho mínimo $s \rightsquigarrow v$, se relaxamos as arestas na ordem desse caminho, então a estimativa do menor caminho para v se tornará o menor caminho: $v.d = \delta(s, v)$.

Propriedade do predecessor ao computar todas as estimativas para o menor caminho, $v.d = \delta(s, v)$, teremos uma árvore de caminhos mínimos cuja raíz é s.

5.5 Caminhos mínimos a partir de uma única fonte

5.5.1 Algoritmo de Bellman-Ford

```
RELAX(u, v, w)
1  if v.d > u.d + w(u, v)
2     v.d ← u.d + w(u, v)
3     v.π ← u
```

```
BELLMAN-FORD(G(V, E), w, s)
 1  inicialize todos os predecessores dos vértices para NIL
 2  inicialize todos as estimativas dos vértices para ∞
 3  s.d ← 0
 4  for i ← 1 to |V| − 1
 5     for (u, v) in E
 6        RELAX(u, v, w)
 7  for (u, v) in E
 8     if v.d > u.d + w(u, v)
 9        return false ⟨⟨existe ciclo negativo⟩⟩
10  return true
```

5.5.2 Algoritmo de Dijkstra

Este algoritmo utiliza uma escolha gulosa de sempre escolher para alcançar em seguida o vértice não alcançado que tenha a menor estimativa de distância até o momento. A inteligência do algoritmo vem do fato de que ao se alcançar um novo vértice, temos a oportunidade de melhorar a estimativa de distância de todos aqueles vértices não alcançados e que sejam adjacentes ao novo vértice.

O algoritmo faz $|V|$ operações EXTRACT-MIN e faz $O(|E|)$ operações DECREASE-KEY (que acontece quando relaxamos uma aresta e diminuímos a prioridade de um vértice). Essas duas operações podem ser implementadas com complexidade $O(\log|V|)$ em um heap binário mínimo, totalizando $O((V + E)\log V)$, ou $O(E \log V)$ se todos os vértices forem alcançáveis (grafo conectado). É possível obter um custo assintótico melhor se usarmos heaps Fibonacci, que oferecem um custo amortizado de $O(\log V)$ por EXTRACT-MIN (o mesmo de antes), mas melhora o custo amortizado de DECREASE-KEY para $O(1)$, totalizando $O(V \log V + E)$.

```
DIJKSTRA(G(V,E),w,s)
1  inicialize todos os predecessores dos vértices para NIL
2  inicialize todos as estimativas dos vértices para ∞
3  s.d ← 0
4  Q ←BUILD-HEAP(V, u → u.d) ⟨⟨chave é a estimativa do vértice⟩⟩
5  while Q ≠ ∅
6     u ← EXTRACT-MIN(Q)
7     for v in G.AdjList[u]
8        RELAX(u,v,w) ⟨⟨pode mudar a prioridade do elemento no heap⟩⟩
```

5.5.3 Caminhos mínimos em DAGs

Em grafos acíclicos direcionados (DAG), não temos ciclos por definição. Isso facilita nossa tarefa de obter caminhos mínimos pois podemos estabelecer qual serão as ordens dos caminhos mínimos. Sabendo disso, podemos relaxar as arestas em uma ordem específica e pela propriedade do relaxamento de um caminho, estamos garantidos que teremos a estimativa ótima para cada vértice. A ordem assim mencionada é diretamente obtida a partir de uma ordenação topológica do grafo de entrada. Observe que assim conseguimos obter os caminhos mínimos a partir de uma fonte em tempo linear $\Theta(V+E)$.

```
DAG-SHORTEST-PATHS(G(V,E),w,s)
1  inicialize todos os predecessores dos vértices para NIL
2  inicialize todos as estimativas dos vértices para ∞
3  s.d ← 0
4  V' ← "ordenação topológica de G"
5  for u in V' ⟨⟨em ordem⟩⟩
6     for v in G.AdjList[u]
7        RELAX(u,v,w)
```

5.6 Caminhos mínimos entre todos os pares de vértices

Vamos assumir que os pesos das arestas estão organizados em uma matriz quadrada w_{ij} e que uma matriz de predecessores Π_{ij} armazena os caminhos mínimos entre cada par de vértices. Veremos que essa formulação nos ajudará a construir algoritmos para o caso em que os grafos são densos. A subestrutura de um caminho mínimo nos permite caracterizar uma equação para custos ótimos de subproblemas e também um algoritmo de programação dinâmica. Seja l_{ij} o custo de um caminho mínimo entre o vértice i e j. Sabemos que esse caminho pode ter, no máximo, $n-1$ arestas onde $n = |V|$. Por isso, podemos caracterizar qualquer caminho mínimo em subcaminhos que usam uma quantidade de arestas. Seja $l_{ij}^{(m)}$ o custo do caminho mínimo entre i e j que usa, no máximo, m arestas:

$$l_{ij}^{(m)} = \begin{cases} 0 & \text{se } m = 0 \text{ e } i = j \\ \infty & \text{se } m = 0 \text{ e } i \neq j \\ \min_{1 \leq k \leq n}(l_{ik}^{(m-1)} + w_{kj}) & \text{caso contrário} \end{cases}$$

Com essa equação, conseguimos dois algoritmos para o problema. O primeiro constrói $l_{ij}^{(0)} \rightsquigarrow l_{ij}^{(1)} \rightsquigarrow \cdots \rightsquigarrow l_{ij}^{(m-1)}$ para todo par de vértices i, j. Computar cada entrada da matriz requer tempo linear porque é o mínimo entre n alternativas. Temos $O(n^2)$ pares e isso quer dizer que cada passo custa $O(n^3)$. Temos ao todo $n-1$ passos até conseguir obter os menores caminhos com até $n-1$ arestas, totalizando

$O(V^4)$. Podemos melhorar esse custo ao observar que esse processo iterativo é similar à multiplicação de matrizes e, portanto, podemos "dobrar"sucessivamente a quantidade de arestas consideradas e encontrar $l_{ij}^{(n-1)}$ em menos passos, totalizando $O(V^3 \log n)$.

5.6.1 Algoritmo de Floyd-Warshall

Este algoritmo melhora a definição recursiva de custo de um caminho mínimo ao considerar quais vértices intermediários podem aparecer em um caminho mínimo. Se fixamos um vértice intermediário como k, teremos $p : i \rightsquigarrow^{p_1} k \rightsquigarrow^{p_2} j$. Veja que os vértices intermediários que aparecem no subcaminho p_1 não podem ser k (mesma coisa para p_2). Dessa forma, podemos melhorar um custo de caminho que considere $\{1, ..., k\}$ como vértice intermediário se já tivermos calculado os custos ótimos de caminhos que considerem $\{1, ..., k-1\}$ como intermediário:

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

Veja que agora continuamos com um algoritmo iterativo sobre matrizes, mas cada passo de refinamento requer custo constante e não linear. Portanto, temos $O(n)$ iterações e cada uma custa $O(n^2)$, totalizando $O(n^3)$.

```
FLOYD-WARSHALL(w)
1  d ←"matriz (0..n) × (1..n) × (1..n)"
2  d[0] ← w ⟨⟨caso base: nenhum vértice intermediário⟩⟩
3  for k ← 1 to n
4     for i ← 1 to n
5        for j ← 1 to n
6           d[k][i][j] ← min(d[k−1][i][j], d[k−1][i][k] + d[k−1][k][j])
7  return d[n]
```

Veja que podemos melhorar esse custo de espaço do algoritmo de $O(n^3)$ para $O(n^2)$. Podemos reconstruir também os caminhos mínimos além dos custos ótimos ao armazenar em outra matriz Π quais decisões a cada passo levaram ao melhor custo. Uma alternativa é guardar o índice do predecessor do vértice que levou ao melhor custo: $\Pi^{(0)}$ considera como predecessores as arestas isoladas.

5.6.2 Algoritmo de Johnson

5.7 Fluxo máximo

Alguns conceitos importantes em fluxo máximo: entender as premissas do problema como conservação e skew, entender o conceito de cortes nessas redes, entender que o fluxo máximo é limitado superiormente por qualquer corte nessa rede, entender o conceito de redes residuais, entender que um caminho em uma rede residual que vai da origem até o destino indica que a rede admite mais fluxo e que portanto, a função ainda não está maximizada.

Propriedade da conservação do fluxo. Para cada vértice na rede que não for nem a fonte nem o destino, a quantidade de fluxo chegando no vértice deve ser igual à quantidade de fluxo saindo do vértice

$$\sum_{v \in V} f(v, u) = \sum_{v \in V} f(u, v) \text{ para todo } u \in V \setminus \{s, t\}$$

Propriedade da restrição de capaciidade. Para cara par de vértices da rede, a quantidade de fluxo passando entre eles não pode ultrapassar a capacidade:

$$\text{para } u, v \in V : 0 \leq f(u,v) \leq c(u,v)$$

```
FORD-FULKERSON(G = (V, E))
 1  construa rede residual G_f a partir de G
 2  enquanto existir caminho de aumento em G_f
 3     seja p algum caminho de aumento
 4     Δ ← min{c_f(u, v) : (u, v) ∈ p}
 5     para cada aresta (u, v) ∈ p
 6        se (u, v) ∈ G.E
 7           (u, v).f ← (u, v).f + Δ
 8        senão
 9           (v, u).f ← (v, u).f − Δ
10     atualize rede residual G_f ao longo do caminho p
11  cada aresta contém o seu fluxo máximo em (u, v).f
```

5.8 Exercícios

1 (crls 3ed. – 24.1.5) Dado um grafo direcionado com pesos nas arestas $G(V,E)$, $w : E \to \mathbb{R}$. Forneça um algoritmo $O(VE)$ que encontre, para cada vértice v, o valor $\delta^*(v) = \min_{u \in V}\{\delta(u,v)\}$.

Descrição de algoritmo: Vamos substituir o procedimento RELAX do algoritmo de Bellman-Ford pelo procedimento RELAX-MIN abaixo. Outra modificação, é inicializar os predecessores de cada vértices como eles próprios: $u.\pi \leftarrow u$ para todo $u \in V$. Veja que o último passo é necessário e coerente, já que $\delta(u,u) = 0$ e portanto, representa um limite superior para a função em cada vértice.

```
RELAX-MIN(u, v, w)
1  if v.d > MIN(w(u, v), u.d + w(u, v))
2     v.d ← MIN(w(u, v), u.d + w(u, v))
3     v.π ← u.π ⟨⟨O predecessor de v é o predecessor de u: u ou u′ ≠ u⟩⟩
```

2 (crls 3ed. – 24.1.6) Suponha que um grafo direcionado $G(V,E)$ e com pesos nas arestas contenha um ciclo de peso negativo. Forneça um algoritmo eficiente para listar os vértices desse ciclo.

Ideia de algoritmo: faça um caminhamento em profundidade (DFS) no grafo mantendo a soma dos pesos acumulados no caminho. Se o caminhamento encontrar uma aresta de volta (back edge), então um ciclo foi encontrado e tem-se também a soma dos pesos do ciclo. Ao detectar esse ciclo negativo, interrompemos o caminhamento imprimindo os vértices do caminho de volta à raíz, isto é, nas voltas da recursão até o destino da aresta de volta encontrada. Isso pode ser implementado (1) executando o Bellman-Ford; (2) marcando $u.d \leftarrow -\infty$ para vértices alcançáveis através de vértices do ciclo negativo (DFS); (3) fazendo um novo DFS para detectar ciclo a partir de vértices cuja estimativa seja $u.d = -\infty$ – se encontrar aresta de volta (back-edge, cinza) significa que um ciclo foi detectado.

5.9 Grafos Eulerianos e Hamiltonianos

Conceitos importantes: caminhos e ciclos eulerianos, teorema que argumenta que em um grafo euleriano todo vértice tem grau par, algoritmo de Fleury que constrói um ciclo euleriano.

5.9.1 Definições importantes

Percurso: coleção de vértices que são sequencialmente adjacentes

Caminho: percurso em um grafo direcionado com todas as arestas no sentido início-fim do percurso.

Percurso simples: não repete ligações.

Percurso elementar: não repete vértices.

Ciclo: percurso elementar fechado.

Circuito: caminho elementar fechado – ciclo orientado.

5.9.2 Grafos eulerianos

Teorema 5.1. *Um grafo $G = (V, E)$ não direcionado e conexo possui um ciclo euleriano se, e somente se, todos os seus vértices tiverem grau par.*

Prova: *Para $\implies$ devemos observar que em um ciclo precisamos ter uma forma de entrar e sair de cada vértice, ou seja, pelo menos uma quantidade par de arestas incidentes a cada um deles. Para $\impliedby$ devemos fazer por indução no número de arestas. Todo grafo conexo cujos graus dos vértices forem todos ≥ 2 possui um percurso fechado: basta escolher iterativamente para cada vértice duas arestas, uma para entrada e uma para saída. Vamos considerar o percurso fechado de maior tamanho. Ao retirar as arestas desse percurso fechado do grafo, temos dois casos: ou todas as arestas foram retiradas e o percurso em questão é por si só um ciclo euleriano (percurso fechado que não repete arestas), ou sobraram algumas arestas formando uma componente também com todos os vértices de grau par – cada vértice tem seu grau diminuído de duas unidades, já que estamos retirando um percurso fechado. Se a componente restante tem vértices com grau par apenas e ela tem arestas, então novamente podemos dizer que existe um percurso fechado. Isso é um absurdo, já que poderíamos combinar esse novo percurso fechado com o percurso fechado inicial e máximo que teríamos um percurso fechado maior ainda, contradizendo a premissa.* ***Em suma, podemos visualizar um ciclo euleriano (percurso fechado com todas as arestas) como uma decomposição disjunta das arestas em vários ciclos.*** □

```
CICLO-EULERIANO-FLEURY(G(V, E)) // assume que G é euleriano
 1  seja s ∈ V um vértice inicial qualquer
 2  u ← s ⟨⟨vértice corrente no percurso⟩⟩
 3  repita enquanto existirem arestas no grafo
 4     se d(u) = 1 ⟨⟨grau de u⟩⟩
 5        escolha a única aresta possível (u, v) ∈ E
 6     senão
 7        escolha (u, v) ∈ E de tal maneira que mesmo que (u, v) seja removida,
          s continua na mesma componente do restante das arestas
 8     adicione (u, v) ao percurso
 9     remova a aresta (u, v) do grafo
10     u ← v
11  retorne a sequência de arestas
```

```
CARTEIRO-CHINES(G = (V, E))
 1  se G for não-euleriano
 2     seja I ⊆ V o conjunto de vértices de grau ímpar
 3     para cada u ∈ I
 4        descubra os caminhos mínimos d_uv a partir de u (usando Dijkstra, por exemplo)
 5        d_uu ← ∞
 6     P ← "pares de vértices u, v ∈ I cuja soma das distâncias d_uv seja mínima" (Alg. húngaro)
 7     para cada (u, v) ∈ P
 8        adicionar aresta artificial (u, v) em G com custo d_uv
 9  C ← CICLO-EULERIANO-FLEURY(G = (V, E))
10  para cada aresta artifical (u, v) ∈ C
11     C ← substitua em C a aresta (u, v) pelo caminho mínimo u ⇝ v
12  retorne C
```

```
CARTEIRO-CHINES-DIRECIONADO(G = (V, E))
 1  se G for não-euleriano
 2     seja S ⊆ V o conjunto de vértices u tal que d⁺(u) − d⁻(u) > 0
 3     seja T ⊆ V o conjunto de vértices u tal que d⁺(u) − d⁻(u) < 0
 4     para cada u ∈ S
 5        descubra os caminhos mínimos d_uv a partir de u para todo v ∈ T (Alg. Dijkstra)
 6        d_uu ← ∞
 7     S' ←REPLICA-VERTICES(S) ⟨⟨w^(u) denota uma cópia de u ∈ S⟩⟩
 8     T' ←REPLICA-VERTICES(T) ⟨⟨w^(u) denota uma cópia de u ∈ T⟩⟩
 9     PREENCHE-DISTANCIAS-REPLICAS(S', T', S, T)
10     P ← "pares u ∈ (S ∪ S'), v ∈ (T ∪ T') cuja soma das distâncias d_uv seja mínima" (Alg. húngaro)
11     para cada (w, z) ∈ P
12        se w é réplica w^(r) então u ← r senão u ← w
13        se z é réplica z^(r) então v ← r senão v ← z
14        adicionar aresta artificial (u, v) em G com custo d_uv
15  C ← CICLO-EULERIANO-FLEURY(G = (V, E))
16  para cada aresta artifical (u, v) ∈ C
17     C ← substitua em C a aresta (u, v) pelo caminho mínimo u ⇝ v
18  retorne C
```

```
REPLICA-VERTICES(V)
1  V' ← ∅
2  para cada u ∈ V
3     for i ← 2 to |d⁺(u) − d⁻(u)|
4        seja w^(u) uma cópia de u
5        V' ← V' ∪ {w^(u)}
6  retorne V'
```

```
PREENCHE-DISTANCIAS-REPLICAS(S', T', S, T)
1 para cada w^(u) ∈ S'
2    para cada z^(v) ∈ T'
3       d_{w^(u) z^(v)} ← d_uv
4 para cada w^(u) ∈ S'
5    para cada v ∈ T
6       d_{w^(u) v} ← d_uv
7 para cada u ∈ S
8    para cada w^(v) ∈ T'
9       d_{u w^(v)} ← d_uv
```

Algumas características importantes do algoritmo que resolve o problema do carteiro chinês:

- Se trata de um algoritmo de complexidade polinomial: algoritmo húngaro e chamadas do algoritmo de Djkstra continuam sendo polinomiais.
- Quando organizamos os vértices de grau ímpar em um grafo não euleriano em pares, isso sempre é seguro já que para qualquer grafo devemos ter uma quantidade par de vértices com grau ímpar: senão a soma dos graus não seria par e sabemos que isso é verdade.
- Se um grafo é euleriano já sabemos exatamente qual é o custo mínimo da instância: exatamente a soma dos pesos de cada aresta.
- Se um grafo não é euleriano, será inevitável passar por algumas arestas mais de uma vez e isso é capturado pelas arestas artificiais adicionadas: passar por uma aresta artificial (u, v) no percurso é equivalente a um percurso de mesmo custo que usa apenas arestas do grafo original: $u...v$.

5.9.3 Grafos hamiltonianos

Capítulo 6

Teoria da Complexidade de Algoritmos: Classes de Problemas e NP-Completude

Capítulo 7

Técnicas para tratar problemas NP-Completos

- Começa definindo os termos
- Heurísticas construtivas: funções gulosas e exemplos mochila, PCV, bin packing. Neste ponto, existe o conceito de escolher o item que mais melhora o custo a todo momento e para o problema do bin packing, temos o conceito de maximizar a utilização dos recursos.
- Os slides e pseudocodigos do Thiago Noronha parecem mais simples de descrever do que os slides da referência que eu achei.
- Bin packing / first fit: árvore de torneios para fazer em $O(n \log n)$
- Outro princípio: inserir elementos mais difíceis primeiro usando first fit decreasing
- Heurísticas são mais específicas e seus princípios diferem de problema para problema, apesar de existirem similaridades. Metaheurísticas, por outro lado, se referem a estratégias mais genéricas que podem ser aplicadas diretamente a qualquer problema.
- Algumas diferenciações de metaheurísticas: (1) memória: as soluções parciais podem ser armazenadas e classificadas de acordo com a qualidade delas, isso vai permitir um mecanismo de aprendizagem; (2) intensificação: soluções parecidas tendem a ter custos parecidos e podemos sempre intensificar o que de melhor achamos até o momento – *exploitation*; (3) diversificação: é importante que a execução seja capaz de explorar novos caminhos para que o conjunto de soluções não fiquem enviesadas apenas a um subconjunto de soluções, é preciso haver um equilíbrio – *exploration*.
- Características de metaheurísticas: simplicidade, generalidade, eficácia (soluções quase ótimas), eficiência (quão rápido as soluções são obtidas), robustez (variância pequena entre execuções repetidas – no caso de operações não determinísticas).

7.1 Algoritmos aproximativos

7.1.1 Algoritmo aproximativo para o PCV com desigualdade triangular

```
PCV-APROXIMATIVO-AGM(G = (V, E))
1 T* ← "alguma AGM de G" ⟨⟨Alg. Prim ou Kruskal⟩⟩
2 C ← "DFS em T*, anotando os vértices na ida e na volta do percurso"
3 S ← "elimine repetições de vértices em C, exceto o primeiro e o último"
4 retorne S
```

Sabemos que o custo de uma solução ótima S^* para o PCV é maior ou igual ao custo de uma árvore geradora qualquer T, que por sua vez é maior do que o custo de uma AGM T^*:

$$c(S^*) \geq c(T) \geq c(T^*) \implies c(S^*) \geq c(T^*)$$

O custo do caminhamento DFS em T^* no algoritmo é $2c(T^*)$, já que passamos duas vezes por cada aresta: na ida e na volta. Assim:

$$c(S^*) \geq c(T^*) \implies 2c(S^*) \geq 2c(T^*)$$

Mas também sabemos que o custo da solução retornada pelo algoritmo, $c(S)$, não é maior do que o caminhamento DFS de custo $2c(T^*)$, já que o primeiro foi obtido através da remoção de vértices (e arestas) do segundo. Então, por esse motivo e considerando a desigualdade triangular:

$$2c(S^*) \geq 2c(T^*) \geq c(S) \implies c(S) \leq 2c(S^*)$$

Dessa forma, temos nosso fator de aproximação para este problema de minimização: $\rho = 2$. Em outras palavras, os ciclos hamiltonianos retornados pelo algoritmo aproximativo são, no máximo, 2× piores do que a solução ótima, para qualquer que seja a instância de entrada!

7.2 Heurísticas construtivas

Apesar de existirem diversas divergências de nomenclatura, podemos considerar que *algoritmo* é um procedimento que resolve um problema de forma *exata*. Por outro lado, uma *heurística* é um procedimento que não resolve um problema sempre de forma exata mas que é projetada com o objetivo de chegar o mais próximo do exato.

Uma heurística construtiva são aqueles procedimentos que constroem passo-a-passo uma solução viável para um problema de otimização. Muitas vezes essas heurísticas construtivas são de complexidade polinomial (senão compensava mais resolver o problema de forma exata) e utilizam o paradigma de algoritmos gulosos em seu projeto. Uma heurística construtiva é, portanto, muito dependente do problema em questão e das características que temos nos elementos que compõem a solução.

7.3 Metaheurísticas

Uma metaheurística pode ser definida como uma metologia que guia o projeto de heurísticas para um determinado problema. Por esse motivo, metaheurísticas tem uma característica principal que é a de ser mais generalizável e aplicável a diferentes problemas de otimização. Já que o alvo desse tipo de estratégia são aqueles problemas para os quais não se conhece algoritmo polinomial determinístico para obter a solução exata, metaheurísticas trabalham no intuito de amostrar o espaço de busca de soluções

de maneira a selecionar o que de melhor foi encontrado dada restrições de recursos computacionais. Três mecanismos são fundamentais:

1. **Intensificação (exploitation)** se refere à capacidade da metaheurística em melhorar (intensificar) soluções encontradas: soluções parecidas tendem a ter custos parecidos.
2. **Diversificação (exploration)** se refere à capacidade da metaheurística em diversificar o conjunto de soluções amostradas do espaço de busca, de maneira a aumentar a confiança de que o melhor encontrado de fato está razoavelmente próximo da solução ótima.
3. **Memória** se refere à capacidade da metaheurística de armazenar e classificar soluções de acordo com suas qualidades com o objetivo de *aprender* durante a execução quais decisões possuem maior potencial de produzir melhores resultados.

7.4 Busca em vizinhança

Podemos modelar o espaço de busca das soluções de um problema como um grafo (direcionado ou não): vértices representam soluções e arestas representam soluções vizinhas, isto é, soluções que puderam ser transformadas em outras a partir de algum critério (função) de transformação. Funções de vizinhança são específicas de cada problema, alguns exemplos são:

2-opt pode ser usado em problemas cuja solução pode ser representada como uma permutação de itens (vértices no PCV, por exemplo) e consiste em trocar pares de elementos da solução. Todas as maneiras diferentes de se trocar dois elementos da solução corrente, portanto, induzem às soluções vizinhas.

reinsert outra vizinhança comum para permutações consistem em reinserir cada item em todas as posições diferentes da permutação original, gerando uma vizinhança mais densa.

swap se a solução for representada por vários subconjuntos de itens, podemos considerar escolher um item de um dos subconjuntos e consider trocá-lo com outros elementos de subconjuntos diferentes. Por exemplo, problema do empacotamento.

vizinhança n^1 utilizado quando a solução pode ser representada como uma sequência de bits e consiste em considerar como vizinhas aquelas soluções que diferem de, exatamente, um bit.

Considerando o grafo implícito de soluções e a função de vizinhança, podemos delimitar duas estratégias padrão:

- Em uma **busca aleatória** na vizinhança, a heurística salta de solução em solução aleatoriamente, sempre mantendo o que de melhor for encontrado. Conhecida como heurística do passeio aleatório (*random walk*).
- Em uma **busca local** na vizinhança, a heurística salta para soluções sempre aprimorantes, até que um ótimo local seja encontrado e nenhuma solução vizinha é melhor do que a corrente.

Temos duas abordagens para realizar a busca local: (1) saltar sempre para a melhor solução aprimorante; (2) saltar sempre para a primeira solução aprimorante. A primeira abordagem tende a alcançar o ótimo local mais rapidamente, mas ao custo potencialmente elevado de gerar todos os vizinhos de uma solução a cada salto. Por outro lado, a segunda abordagem custa menos para cada salto mas tende a alcançar o ótimo local com mais passos.

Alguns problemas de se realizar busca local isoladamente:

- Ótimo local pode ser muito distante do ótimo global;
- A solução inicial pode influenciar muito na qualidade do ótimo local encontrado;
- A vizinhança pode impactar na busca local, tanto do ponto de vista de complexidade como de qualidade;
- Pode ser exponencial no pior caso: por exemplo, com apenas um ótimo local sendo o ótimo global juntamente com uma função de vizinhança que aprimora pouco a solução.

Por todos esses motivos, existem metaheurísticas baseadas em busca local que tentam fugir desses ótimos locais para encontrar soluções mais diversificadas.

7.5 Metaheurísticas baseadas em busca local: Variable Neighborhood Search

A ideia principal é que, para escapar de ótimos locais, podemos utilizar várias vizinhanças de forma sistemática. Uma solução pode ser um ótimo local para uma vizinhança mas não o ser para outra e portanto, temos maior chance de continuar aprimorando as soluções, mesmo que elas representem um ótimo local para alguma(s) vizinhança.

Utiliza um método conhecido como Variable Neighborhood Decent para refinar soluções utilizando *várias funções de vizinhança*. Chamamos de VND uma versão de VNS na qual a escolha das vizinhanças é feita de forma determinística (e não estocástica). Esse VNS baseado em VND utiliza o procedimento a seguir para aprimorar uma solução s até que uma condição de parada seja alcançada: por exemplo, tempo de relógio em um processador, número de iterações, etc.

```
VND(f, N_1..N_k, s)
1  i ← 1
2  while i ≤ k
3     s' ← melhor vizinho em N_i(s)
4     if f(s') é melhor do que f(s)
5        s ← s'
6        i ← 1
7     else
8        i ← i + 1
9  return s
```

Outras versões de VNS que não são determinísticas existem. Nesses casos, a uma solução é aleatoriamente selecionada da vizinhança corrente (ao contrário de ser um procedimento determinístico *descent*) e o procedimento continua normalmente.

7.6 Metaheurísticas baseadas em busca local: GRASP

Se trata de uma heurística de multi-partida onde depois de encontrar um ótimo local, reinicia a busca a partir de outra solução inicial para tentar obter soluções potencialmente diferentes. O termo é uma sigla para: *Greedy Randomized Adaptive Search Procedure*.

O GRASP é composto de várias iterações e cada uma contendo duas fases: (1) fase construtiva, onde alguma solução inicial é gerada de forma aleatória e gulosa; (2) fase de busca local onde soluções vizinhas aprimorantes são visitadas até que o ótimo local seja alcançado. Naturalmente, o diferencial do GRASP está na fase (1).

Para conseguir fazer (1), o GRASP considera que alguma solução viável é composta de um conjunto de itens – isso é sempre verdade para algoritmos gulosos. A cada passo da construção desse conjunto solução, consideramos escolher o melhor item candidato dentre um *subconjunto de todos os possíveis*. Portanto, se escolhemos o melhor item dentre todos os possíveis, temos um algoritmo guloso puro; se escolhemos o melhor item dentre um subconjunto de itens, introduzimos uma certa aleatoriedade na escolha da solução inicial. Veja que esse espectro de decisão representa o compromisso do GRASP entre intensificação e diversidade, sendo controlado por um parâmetro α. Considerando um problema de minimização, a lista de itens considerados no sorteio aleatório da construção da solução inicial é o seguinte, selecionamos todo item e cujo custo de sua inclusão seja $c(e)$:

$$c(e) \leq c_{min} + \alpha(c_{max} - c_{min})$$

c_{min} representa o custo do item que tem o menor custo

c_{max} representa o custo do item que tem o maior custo

```
SOLUCAO-INICIAL-GRASP(α)
 1  s ← ∅
 2  enquanto s não for uma solução completa
 3     G ← "lista de itens candidatos"
 4     avalie custos incrementais da solução para cada g ∈ G
 5     c_min ← min(G)
 6     c_max ← max(G)
 7     G' ← {g ∈ G | c(e) ≤ c_min + α(c_max − c_min)} ⟨⟨lista restrita de candidatos⟩⟩
 8     e ← "item aleatório em G'"
 9     se s ∪ {e} for inviável
10        se s não puder se tornar viável, interrompa laço
11  retorne s
```

```
GRASP(α)
1  seja s a melhor solução encontrada
2  enquanto condição de parada não for atingida
3     s_0 ←SOLUCAO-INICIAL-GRASP(α)
4     s' ←BUSCA-LOCAL(s_0)
5     se s' for melhor do que s
6        s ← s'
7  retorne s
```

Como podemos ver, a característica de ser guloso é dada pela seleção de itens de baixo custo na lista restrita. A característica aleatorizada é dada pela escolha aleatória de um item da lista restrita. A característica adaptativa é dada pelo cálculo da cardinalidade da lista e controlada pelo parâmetro α, deixando a estratégia *independente de valores absolutos de custo*.

Existem estratégias que configuram o α de maneira automática e de forma reativa. Nesse caso, podemos definir uma lista inicial de possíveis valores α_i e a cada iteração escolhemos um desses valores de forma aleatória para guiar o GRASP. Inicialmente, consideramos todos os α_i com a mesma probabilidade de escolha, mas toda vez que terminamos uma iteração atualizamos a chance de cada α_i ser escolhido ao favorecer valores que geraram soluções melhores. A ideia é, portanto, que α convirja para α_i que na média produz as melhores soluções para o problema.

7.7 Metaheurísticas baseadas em busca local: Busca Tabu

O princípio da busca tabu é: quando um ótimo local for alcançado numa busca local, continue realizando busca local. De certa forma, se trata de um mecanismo de tentar escapar de ótimos locais e não sofrer de convergência precoce. Naturalmente, ao se fazer isso estaremos sujeitos a revisitar soluções já visitadas no passado e pior: o algoritmo de busca pode inclusive entrar em ciclos intermináveis. Por isso, a busca tabu precisa contar com uma chamada *lista tabu* que armazena uma certa quantidade constante de soluções visitadas, já que não é praticável na maioria das vezes armazenar todas as soluções. Com essa informação, a busca local pode evitar sempre pular para soluções que já tenham sido visitadas.

1 Lista tabu. Se trata de um mecanismo de memória de curto prazo. Se armazenamos na lista tabu k soluções, então somos capazes de evitar ciclos de tamanho $\leq k$.

2 Intensificação. Utiliza a memória de médio prazo para indentificar soluções elite e enviesar a busca na direção dessas melhores soluções.

3 Diversificação. Utiliza a memória de longo prazo para enviesar a busca no sentido oposto das soluções já amostradas – faz o inverso da intensificação ao modificar os pesos dos itens considerados para uma solução vizinha.

```
BUSCA-TABU()
1 s ←"solução inicial"
2 inicialize a lista tabu e as memórias de médio e longo prazos
3 enquanto condição de parada não for atingida
4    s' ← "solução vizinha de s que não é tabu e que atenda aos mecanismos de aspiração"
5    s ← s'
6    atualize a lista tabu e outros mecanismos de memória
7    aplique intensificação e/ou diversificação se necessário
8 retorne s
```

7.7.1 Implementação da lista tabu

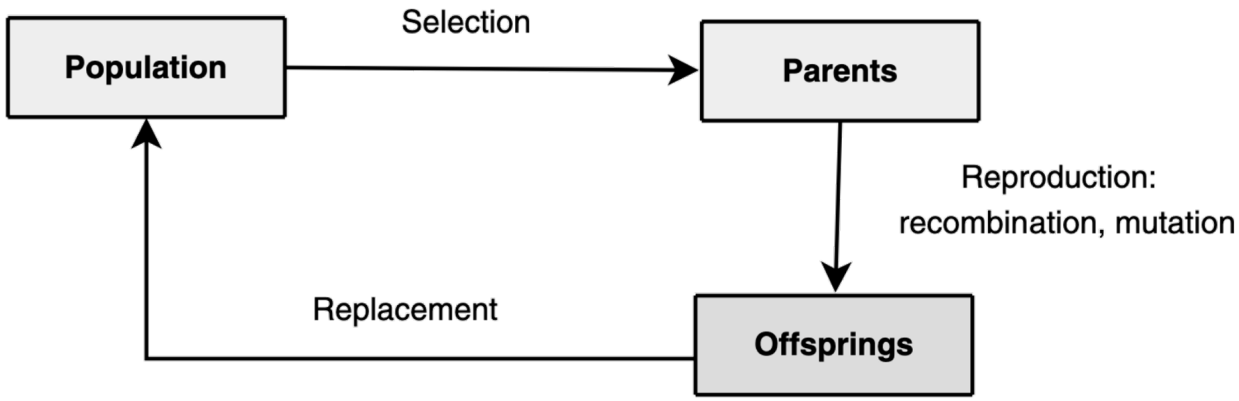
Existem várias possíveis implementações para uma lista tabu. As menos usadas são aquelas que consideram armazenar todas as soluções encontradas, quantidade que comumente é exponencial. Outra forma utilizada é através de árvores de prefixo, que são capazes de comprimir o conjunto de soluções encontradas. Entretanto, a estratégia mais utilizada é utilizar algum mecanismo de hashing para rapidamente identificar soluções que foram encontradas.

7.8 Metaheurísticas populacionais

Diferentemente de metaheurísticas de busca local, estratégias populacionais trabalham iterativamente com *conjuntos de soluções* (populações). Dessa forma, um conjunto de soluções é substituído por outro a toda iteração, potencialmente com soluções (indivíduos) mais diversificados e/ou intensificados. Dois tipos são principais: aqueles baseados em evolução, como nos algoritmos genéticos, e aqueles baseados em memória compartilhada, como na otimização baseada em colônia de formigas.

7.9 Metaheurísticas baseadas em populações: algoritmos genéticos

São estratégias inspiradas na evolução por seleção natural.



1 Seleção. Nesta etapa alguns indivíduos mais aptos serão escolhidos para se reproduzirem, isto é, soluções do conjunto serão escolhidas. Existem diversas formas para se implementar essa etapa, duas mais comuns são: sorteio por roleta e torneio. No sorteio por roleta, vamos atribuir certa probabilidade a cada solução de ser escolhida e aqui podemos utilizar diversos mecanismos de intensificação e/ou diversificação. No torneio, algumas soluções são escolhidas de forma aleatória para participarem de um torneio, por exemplo, ganhando aquela que for a melhor. Independente da forma usada para selecionar soluções do conjunto corrente, teremos um subconjunto pronto para a etapa de reprodução, na qual novas soluções derivadas das seleções serão geradas.

2 Reprodução por recombinação (intensificação). Duas ou mais soluções são combinadas em novas soluções. As soluções resultantes devem conter apenas caracteríticas presentes nos pais e por isso, qualidade não é garantida apenas a viabilidade.

3 Reprodução por mutação (diversifiação). Aqui pequenas modificações são realizadas nas soluções de forma aleatória. Uma maneira comum é considerar a vizinhança da solução e sortear um vizinho de forma aleatória.

4 Substituição. Com os novos indivíduos obtidos durante a reprodução, um novo conjunto de soluções. Podemos substituir todo o conjunto, substituir um dos pais que seja pior (estável), ou mesmo selecionar sempre as melhores soluções dentre pais e filhos (elitista).

```
ALGORITMO-GENETICO()
1 gere um conjunto inicial S
2 enquanto condição de parada não atingida
3    P ← algumas soluções para reprodução em S (seleção)
4    S' ← soluções recombinadas em P (reprodução)
5    S' ← soluções em S' com mutações (reprodução)
6    S ← soluções escolhidas em S ∪ S' (substituição)
7 retorne melhor solução em S
```

7.10 Metaheurísticas baseadas em populações: colônia de formigas

```
COLONIA-FORMIGAS(α, β, ρ, δ, n)
 1  seja S um conjunto de soluções (população)
 2  inicialize a memória adaptativa τ_e para cada item e
 3  p ← CALCULA-PROBABILIDADES-ITENS(τ, n, α, β)
 4  enquanto condição de parada não atingida
 5     S ← CONSTROI-SOLUCOES(p)
 6     τ ← ATUALIZA-MEMORIA(τ, ρ, δ)
 7     p ← CALCULA-PROBABILIDADES-ITENS(τ, n, α, β)
 8  retorne melhor solução em S
```

Referências Bibliográficas

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

[2] Erickson, J. *Algorithms*. Independently published, USA, 2019.

[3] Kleinberg, J., and Tardos, E. *Algorithm Design*. Addison-Wesley Longman Publishing Co., Inc., USA, 2005.

[4] Levitin, A. V. *Introduction to the Design and Analysis of Algorithms*. Addison-Wesley Longman Publishing Co., Inc., USA, 2002.