Capítulo 1

Algoritmos Gulosos

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

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

1.0.1 Exemplo

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Algumas considerações:

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

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

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

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

1.0.2 Exercícios

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

   (a) seleção da atividade que inicia primeiro;

   (b) seleção da atividade que tem menor duração;

   (c) seleção da atividade que termina primeiro.

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

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

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

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

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

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

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

1.0.3 Leituras complementares

1. [2] (Capítulo 16)

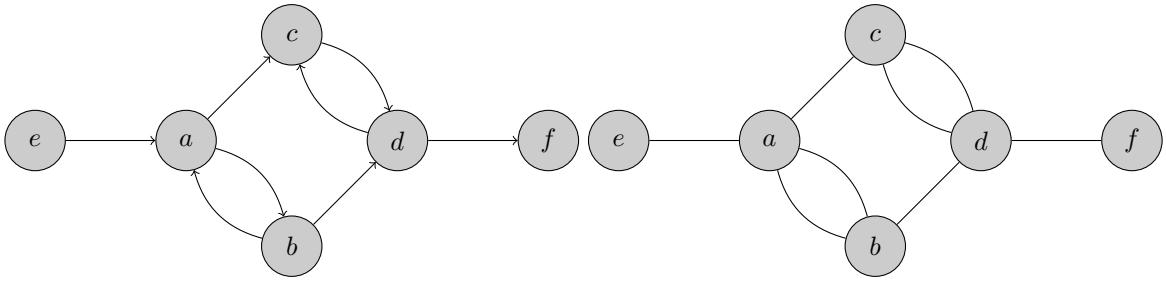
2. [3] (Capítulo 9)

3. [1] (Capítulo 6)

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

1.0.4 Algoritmos gulosos em grafos

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



Do ponto de vista formal, temos as seguintes definições:

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

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

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

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

1.0.5 Representação de grafos

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

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

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

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

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

$$a_{ij} = \begin{cases} 1, & \text{se } (i,j) \in E \\ 0, & \text{caso contrário.} \end{cases} \tag{1.1}$$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

- A conectividade em digrafos é dividida em dois casos:

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

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

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

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

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

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

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

1.0.6 Busca em Largura

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

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

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

Algoritmo 1: BFS(G, s)

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

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

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

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

$$G.A[i][j] = \begin{cases} 1, & \text{se } (i,j) \in G.E \\ 0, & \text{caso contrário.} \end{cases} \tag{1.2}$$

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

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

Algoritmo 2: BFS(G, s)

Qual é a complexidade de tempo de BFS neste caso?

1.0.7 Busca em Profundidade

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

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

Algoritmo 3: DFS(G)

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

Algoritmo 4: DFS-Visit(G, u)

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

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

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

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

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

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

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

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

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

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

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

1.0.8 Árvores Geradoras Mínimas

Introdução

Nesta seção estudaremos o problema de determinar a árvore geradora mínima (*minimum spanning tree*) de um grafo G com pesos [2]. Este problema possui diversas aplicações, como na construção de conexões em redes de computadores, conexões viárias entre diversos pontos de uma cidade ou circuitos eletrônicos onde as componentes dos circuitos são ligadas por fios. No contexto dos circuitos eletrônicos, desejamos, em geral, utilizar a menor quantidade possível de fios. Podemos modelar este problema a partir de um grafo $G = (V, E)$, onde V corresponde ao conjunto de componentes do circuito, e E o conjunto de ligações entre duas destas componentes. O peso $w(u, v)$ da aresta (u, v) especifica o custo (quantidade de fios) necessário para conectar u e v. Portanto desejamos encontrar uma árvore $T \subseteq E$ que conecta todos os

vértices de G e cujo peso total

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

seja mínimo.

Formalmente, uma árvore geradora é definida como a seguir:

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

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

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

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

Quantas árvores geradoras um grafo qualquer possui?

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

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

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

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

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

Algoritmo 5: Generic-MST(G, w)

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

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

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

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

Demonstração. Exercício.

□

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

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

O algoritmo de Prim

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

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

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

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

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

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

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

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

Algoritmo 6: Insert(S, k)

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

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

Algoritmo 7: Decrease-Key(S, i, k)

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

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

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

Algoritmo 8: Minimum(S)

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

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

Algoritmo 9: Extract-Min(S)

onde Min-Heapify é dada por:

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

Algoritmo 10: Min-Heapify(S, i)

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

bém $O(\lg n)$.

O algoritmo de Prim é dado como a seguir:

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

Algoritmo 11: MST-Prim(G, w, r)

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

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

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

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

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

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

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

Exercícios

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

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

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

1.0.9 Caminhos mínimos

Nesta seção veremos como resolver o problema do caminho mínimo em grafos com pesos[2]. Agora cada aresta do grafo está associada a um número real, de forma que o peso de um caminho p em G é a soma dos pesos das arestas que compõem o caminho p. Formalmente, temos a seguinte definição:

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

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

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

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

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

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

Subestrutura ótima do problema do caminho mínimo

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

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

Demonstração. Prova por contradição.

□

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

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

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

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

2. G' é uma árvore com raiz s, e;

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

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

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

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

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

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

Algoritmo 12: Relax(u, v, w)

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

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

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

Algoritmo 13: Initialize-Single-Source(G, s)

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

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

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

Demonstração. Indução no comprimento k do caminho p.

□

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

O algoritmo de Dijkstra

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

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

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

Algoritmo 14: Dijkstra(G, w, s)

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

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

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

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

Complete a prova deste teorema.

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

1.0.10 Leituras complementares

1. [2] (Capítulos 22 e 23)

2. [3] (Capítulo 9)

Referências Bibliográficas

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

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

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