Projeto e Análise de Algoritmos

Flávio L. C. de Moura*

1 Algoritmos Gulosos

Nesta seção veremos outra técnica para resolver problemas de otimização. Esta técnica é utilizada para construir os chamados *algoritmos gulosos*, que a cada passo faz a escolha local ótima na esperança de obter uma solução ótima global.

Vamos iniciar com um exemplo antes de apresentarmos a técnica propriamente dita. 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 um horário de início s_i, e outro de término t_i, onde $0 \leq s_i < t_i < \infty$ $(1 \leq i \leq n)$, queremos selecionar o número máximo de atividades mutuamente compatíveis de S. Se selecionada, a atividade a_i utiliza o recurso no intervalo $[s_i, t_i)$. Duas atividades distintas a_i e a_j são ditas *compatíveis* se $s_i \geq t_j$ ou $s_j \geq t_i$. Assumiremos que as atividades estão ordenadas de forma monotonicamente crescente pelo 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, precisamos que os problemas tenham a propriedade da subestrutura ótima porque utilizaremos as soluções ótimas dos subproblemas para construir uma solução ótima do 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 o final da atividade a_i, e terminam antes do início da atividade a_j. 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 temos 2 subproblemas: encontrar o subconjunto máximo A_{ik} de S_{ik}, e o subconjunto máximo A_{kj} de S_{kj}.

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

Prova. De fato, se existisse um subconjunto A'_{kj} com atividades mutuamente compatíveis de S_{kj} com $|A'_{kj}| > |A_{kj}|$ 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}. □

Dado o fato de que o problema possui a propriedade da subestrutura ótima sugere que o mesmo possa ser resolvido utilizando programação dinâmica. Denotando $c[i, j]$ o tamanho de uma solução ótima para S_{ij} nos dá a recorrência

$$c[i,j] = \begin{cases} 0, & \text{se } S_{ij} = \emptyset \\ \max\limits_{a_k \in S_{ij}} \{c[i,k] + c[k,j] + 1\}, & \text{se } S_{ij} \neq \emptyset \end{cases} \quad (1)$$

e procedemos como usual para construirmos uma solução via programação dinâmica. Mas e se pudéssemos escolher uma atividade sem ter que resolver todos os subproblemas?

*flaviomoura@unb.br

Isto é o que faz a estratégia gulosa ao selecionar a melhor solução local. Neste caso, a melhor solução local seria selecionar a atividade que usa o recurso de forma mínima, ou seja, selecionar a atividade que termina primeiro. 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 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. A ideia então é 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?

Teorema 1.1. *Considere um subproblema não vazio S_k, e seja $a_m \in S_k$ com o menor tempo de finalização. Então a_m está incluída em algum subconjunto maximal de atividades mutuamente compatíveis de S_k.*

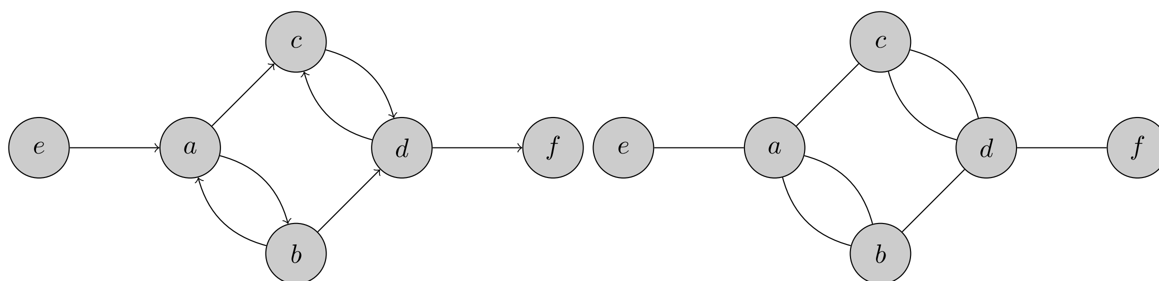
Prova. Seja A_k um subconjunto maximal de atividades mutuamente compatíveis de S_k, e seja a_j a atividade em A_k que termina primeiro, ou seja, a_j é tal que t_j é o menor valor de término em A_k. Se $a_j = a_m$ então estamos prontos. Se $a_j \neq a_m$ então seja $A'_k = A_k - \{a_j\} \cup \{a_m\}$. As atividades em A'_k são disjuntas já que as atividades em A_k, a_j é a atividade que termina primeiro em A_k e $t_m \leq t_j$. Como $|A'_k| = |A_k|$, concluímos que A'_k é um subconjunto maximal de atividades mutuamente compatíveis de S_k 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 possui a propriedade da escolha gulosa: uma solução ótima global pode ser obtida a partir de escolhas gulosas ótimas locais.
3. Enquanto programação dinâmica é naturalmente *bottom-up*, os algoritmos gulosos são naturalmente *top-down*.

1.1 Algoritmos 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:

Definição 1.2. *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.*

Definição 1.3. *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.

1.1.1 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.

Definição 1.4. *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 $\Theta(V + E)$. Esta representação não é adequada para grafos densos, mas é adequada para grafos esparcos, 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.

Definição 1.5. *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*

$$a_{ij} = \begin{cases} 1, & \text{se } (i, j) \in E \\ 0, & \text{caso contrário.} \end{cases} \tag{2}$$

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.

Definição 1.6. *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.

Definição 1.7. *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 $u \overset{p}{\leadsto} v$, quando o grafo é dirigido.

Definição 1.8. *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 $v'_i = v_{(i+j) \bmod k}$ 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.

Definição 1.9. *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[1].

- 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] Sempre que for conveniente, utilizaremos a notação de atributos $G.V$ (resp. $G.E$) para denotar o conjunto dos vértices (resp. das arestas) do grafo G.

Ao considerarmos o tempo de execução de um algoritmo sobre um (di)grafo $G = (V, E)$, normalmente consideramos tanto o número de vértices $|V|$, como o número de arestas $|E|$ como parâmetros para considerar o tamanho da entrada. É usual o abuso de notação $O(VE)$ quando se quer dizer $O(|V|.|E|)$, uma vez que a leitura se torna mais fácil e não há risco de ambiguidade. Adicionalmente, a operação básica pode ser tanto uma visita a um vértice quanto o percorrimento de uma aresta.

Definição 1.10. *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.

Definição 1.11. *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 qualquer grafo possui?

Teorema 1.12 (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:

Algorithm 1: Generic-MST(G, w)

```
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;
```

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

Definição 1.13. *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 1.14. *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.*

Prova. Construiremos uma árvore geradora mínima de G que contém a aresta (u, v). □

Corolário 1.15. *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.*

1.1.2 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 1.15, 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:

Algorithm 2: Insert(S, k)

```
1 S.heap_size ← S.heap_size + 1;
2 S[S.heap_size] ← ∞;
3 Decrease-Key(S, S.heap_size, 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 $S[i]$:

Algorithm 3: Decrease-Key(S, i, k)

```
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
```

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:

Algorithm 4: Minimum(S)

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

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

Algorithm 5: Extract-Min(S)

```
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;
```

onde Min-Heapify é dada por:

Algorithm 6: Min-Heapify(S, i)

```
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
```

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 também $O(\lg n)$.

O algoritmo de Prim é dado como a seguir:

Algorithm 7: MST-Prim(G, w, r)

```
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
```

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 1.16. *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.

□

2 Exercícios

Exercício 2.1. *Considere o problema da mochila booleana (ou problema da mochima 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.*

Exercício 2.2. *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.*

Exercício 2.3. *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.*

Exercício 2.4. *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.*

3 Leitura complementar:

- [2] (Capítulo 15)
- [1] (Capítulo 16)
- [3] (Capítulo 9)

Referências

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

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

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