Algoritmo de Prim

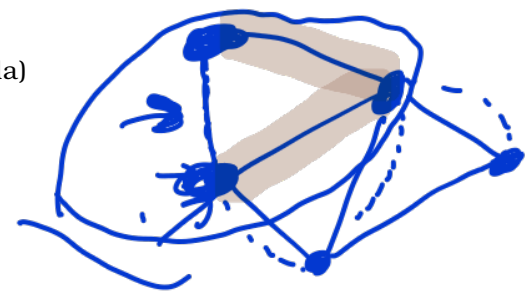
Este capítulo trata do célebre algoritmo de Prim para o problema da MST. (Veja os conceitos básicos sobre esse problema no capítulo *Árvores geradoras de custo mínimo*.) O algoritmo foi publicado por Robert C. Prim em 1957 e por E. W. Dijkstra pouco depois.

> PROBLEMA: Encontrar uma MST (árvore geradora de custo mínimo) de um grafo não-dirigido com custos nas arestas.

Os custos das arestas são números inteiros arbitrários (positivos e negativos). O problema tem solução se e somente se o grafo é conexo. Assim, trataremos apenas de grafos conexos. (Mas veja exercício abaixo.)

O algoritmo de Prim é simples, mas sua implementação eficiente apresenta dificuldades inesperadas. A solução dessas dificuldade ensina interessantes lições de programação.

Sumário:



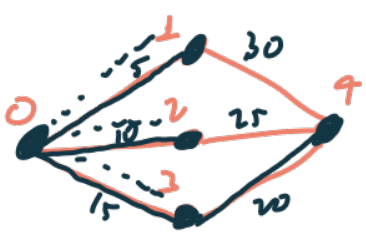
O algoritmo

Dado um grafo não-dirigido conexo G com custos nas arestas, o algoritmo de Prim cultiva uma subárvore de G até que ela se torne geradora. No fim do processo, a árvore é uma MST.

Para discutir os detalhes, precisamos de um pouco de terminologia. Suponha que T é uma subárvore (não necessariamente geradora) de G. A *franja* (= *fringe*) de T é que o corte cuja margem é o conjunto de vértices de T. Em outras palavras, a franja de T é o conjunto de todas as arestas de G que têm uma ponta em T e outra fora de T.

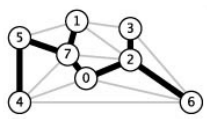
Podemos agora descrever o algoritmo de maneira precisa. Cada iteração começa com uma subárvore T. No início da *primeira* iteração, T consiste em um único vértice. O processo iterativo consiste no seguinte: ⚠ enquanto a franja de T não estiver vazia,

1. escolha uma aresta da franja que tenha custo mínimo,
2. seja x-y a aresta escolhida, com x em T,
3. acrescente a aresta x-y e o vértice y a T.



Como se vê, o algoritmo tem caráter *guloso*: em cada iteração, abocanha a aresta mais barata da franja sem se preocupar com o efeito global, a longo prazo, dessa escolha. A prova de que essa estratégia está correta decorre do critério de minimalidade baseado em cortes.

Exemplo A. Considere o grafo não-dirigido conexo com custos nas arestas definido a seguir. (O custo de cada aresta é proporcional ao comprimento geométrico do segmento de reta que representa a aresta na figura.)



```
5-4 7-4 7-5 0-7 1-5 0-4 2-3 7-1 0-2 1-2 1-3 7-2 2-6 3-6 0-6 6-4
 35  37  28  16  32  38  17  19  26  36  29  34  40  52  58  93
```

A primeira iteração do algoritmo de Prim pode começar com qualquer vértice. Neste exemplo, escolhemos o vértice `0`. A tabela abaixo registra, no início de cada iteração, o conjunto de vértices da subárvore T, o custo de T, e as arestas da franja de T.

```
T                 custo   franja

0                   0     0-2 0-4 0-6 0-7
0 7                 0+16  0-2 0-4 0-6 7-1 7-2 7-4 7-5
0 7 1              16+19  0-2 0-4 0-6 7-2 7-4 7-5 1-2 1-3 1-5
0 7 1 2            35+26  0-4 0-6 7-4 7-5 1-3 1-5 2-3 2-6
0 7 1 2 3          61+17  0-4 0-6 7-4 7-5 1-5 2-6 3-6
0 7 1 2 3 5        78+28  0-4 0-6 7-4 2-6 3-6 5-4
0 7 1 2 3 5 4     106+35  0-6 2-6 3-6 4-6
0 7 1 2 3 5 4 6   141+40
```

A aresta da franja que será acrescentada a T na próxima iteração está pintada de vermelho. A MST resultante tem custo 181. Ela pode ser exibida apagando as colunas apropriadas da descrição do grafo:

```
5-4     7-5 0-7         2-3 7-1 0-2             2-6
 35      28  16          17  19  26              40
```

Exercícios 1

1. Para que serve o algoritmo de Prim? Que problema resolve?
2. ★ Como começa uma iteração genérica do algoritmo de Prim? (Cuidado! Não se trata de listar as *ações* que ocorrem no início de uma iteração! Trata-se de listar as informações disponíveis no início da iteração, antes que a execução da iteração comece.)
3. Suponha que nosso grafo é uma árvore. É verdade que em cada iteração do algoritmo de Prim a franja tem apenas uma aresta?
4. ★ [Sedgewick Property 20.3] *Correção e invariantes.* Prove que o algoritmo de Prim está correto, ou seja, produz uma MST quando aplicado a um grafo não-dirigido conexo com custos nas arestas. (Sugestão: Use o critério de minimalidade baseado em cortes.) [Solução]
5. [Sedgewick, figura 20.1] Qual o custo de uma MST do grafo não-dirigido com custos descrito a seguir?

```
0-6  0-1  0-2  4-3  5-3  7-4  5-4  0-5  6-4  7-0  7-6  7-1
 51   32   29   34   18   46   40   60   51   31   25   21
```

6. Considere o grafo não-dirigido completo cujos vértices são os pontos `(2,1) (2,5) (1,6) (6,6) (3,3) (3,4) (5,2) (5,7)` do plano. O custo de cada aresta é o comprimento do segmento de reta que une os pontos no plano. (Para que os custos sejam inteiros, multiplique os comprimentos por `1000` e tome o piso do resultado.) Escreva as arestas de uma MST na ordem em que elas são escolhidas pelo algoritmo de Prim.
7. [Sedgewick 20.33] Considere o grafo não-dirigido conexo cujos vértices são os pontos no plano dados abaixo. Suponha que as arestas do grafo são `1-0 3-5 5-2 3-4 5-1 0-3 0-4 4-2 2-3` e o custo de cada aresta é igual ao comprimento do segmento de reta que liga as pontas da aresta. (Para que os custos sejam inteiros, multiplique os comprimentos por `1000` e tome o piso do resultado.) Aplique o algoritmo de Prim a esse grafo. Exiba uma figura do grafo e da subárvore no início de cada iteração.

```
  0     1     2     3     4     5
(1,3) (2,1) (6,5) (3,4) (3,7) (5,3)
```

8. [Sedgewick 20.24] Mostre que a seguinte estratégia pode não encontrar uma MST de um grafo não-dirigido conexo. Comece cada iteração com uma subárvore T do grafo. Em cada iteração, (1) seja v o vértice que foi acrescentado a T mais recentemente; (2) seja e uma aresta de custo mínimo dentre as que incidem em v e estão na franja de T; (3) comece nova iteração com $T+e$ no papel de T.
9. *Grafo desconexo.* Que acontece se o algoritmo de Prim for aplicado a um grafo não-dirigido desconexo? É realmente necessário verificar se o grafo é conexo antes de invocar o algoritmo de Prim?
10. Compare o algoritmo de Prim com o algoritmo de Dijkstra para o problema da CPT (árvore de caminhos baratos). Quais a principal diferença? Quais as semelhanças? Comece por apontar as diferenças entre os problemas que os dois algoritmos resolvem.

Implementações do algoritmo

Para transformar o algoritmo de Prim num programa, precisamos tomar algumas *decisões de projeto*. Suporemos que nosso grafo é conexo. A árvore geradora T do grafo será representada por uma árvore radicada. Para isso, basta escolher um vértice de T para fazer o papel de raiz e eliminar um dos dois arcos de cada aresta de T. A árvore radicada será representada por um vetor de pais `pa[]` alocado pelo usuário.

Suporemos que nosso grafo é representado por listas de adjacência com custos. Para cada vértice `v` e cada `a` em `G->adj[v]`, o custo do arco que liga `v` a `a->w` será `a->c` e esse número poderá ser positivo ou negativo. Suporemos também que temos uma constante

`INFINITY` MAX_INT

de valor maior que o custo de qualquer aresta. Finalmente, suporemos que os dois arcos que compõem cada aresta têm o mesmo custo.

Discutiremos a seguir uma implementação ingênua do algoritmo de Prim e duas implementações mais sofisticadas e eficientes: uma para grafos densos e uma para grafos esparsos.

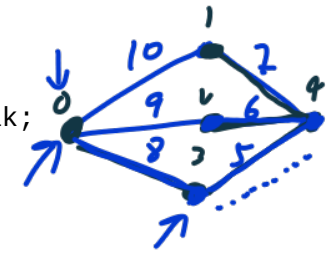
Implementação ingênua

Nossa primeira implementação transforma o algoritmo de Prim em código de maneira direta e literal. O resultado é simples mas ineficiente:

```
#define UGraph Graph

void UGRAPHmstP0( UGraph G, vertex *pa)
{
   for (vertex w = 0; w < G->V; ++w) pa[w] = -1;
```

```
    pa[0] = 0;
    while (true) {
        int min = INFINITY;
        vertex x, y;
        for (vertex v = 0; v < G->V; ++v) {
            if (pa[v] == -1) continue;
            for (link a = G->adj[v]; a != NULL; a = a->next)
                if (pa[a->w] == -1 && a->c < min) {
                    min = a->c;
                    x = v, y = a->w;
                }
        }
        if (min == INFINITY) break;
        // A
        pa[y] = x;
    }
}
```



pa 0,4,4,0,3,1

No ponto `A`, `x-y` é uma aresta de custo mínimo dentre as que estão na franja da árvore. O custo dessa aresta é `min`.

Desempenho. A função `UGRAPHmstP0()` é quadrática: quando aplicada a um grafo com V vértices e E arestas, a função consome tempo proporcional a VE no pior caso. Pode-se dizer que o consumo de tempo é proporcional a V vezes o tamanho do grafo.

Exercícios 2

1. Verifique que a função `UGRAPHmstP0()` implementa corretamente o algoritmo de Prim. Dê especial atenção à instância em que `G` tem um só vértice e à instância em que `G` não tem arestas.
2. *Caminho.* Aplique a função `UGRAPHmstP0()` ao grafo com custos descrito a seguir. Faça o rastreamento da execução da função.

```
0-1  1-6  6-5  5-3  3-2  2-4
 99   99   99   99   99   99
```

3. *Desempenho.* Mostre que a função `UGRAPHmstP0()` consome VE unidades de tempo no pior caso.
4. Modifique o código de `UGRAPHmstP0()` de modo a imprimir um rastreamento da execução da função. Cada linha do rastreamento deve exibir o estado de coisas no início de uma iteração: o conjunto de vértices da árvore, o conjunto de todas as arestas da franja, a aresta mais barata da franja, e o custo dessa aresta.
5. Escreva uma versão simplificada da função `UGRAPHmstP0()` que receba um grafo não-dirigido conexo e devolva o custo — apenas o custo — de uma MST do grafo. Escreva código "enxuto", sem variáveis supérfluas.
6. [Sedgewick 20.54] Escreva uma implementação do algoritmo de Prim que começa por colocar as arestas do grafo em ordem crescente de custo e depois tira proveito dessa ordem.

Implementações eficientes

A implementação ingênua do algoritmo de Prim é lenta e ineficiente porque cada iteração recalcula toda a franja da árvore, mesmo sabendo que a franja mudou pouco desde a iteração anterior. Para obter uma implementação mais eficiente, é preciso começar cada iteração com a franja pronta e *atualizá-la* no fim da iteração. Mas é difícil fazer isso se a franja for tratada como uma simples lista de arestas; é preciso inventar uma representação mais eficiente e tomar algumas decisões de projeto adicionais.

A *fronteira* de uma árvore T é o conjunto de todos os vértices do grafo que não pertencem a T mas são vizinhos de vértices de T. O *preço* de um vértice `w` da fronteira de T é o custo de uma aresta de custo mínimo dentre as que estão na franja de T e incidem em `w`. Se a aresta da franja que determina o preço de `w` é `v-w`, diremos que `v` é o *gancho* de `w`.

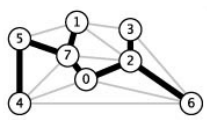
Podemos agora reescrever o algoritmo de Prim em termos de preços e ganchos. Cada iteração começa com uma árvore T e com os preços e ganchos dos vértices que estão na fronteira de T. O processo iterativo consiste no seguinte:
⚠ enquanto a franja de T não estiver vazia,

1. escolha um vértice `y` de preço mínimo na fronteira de T,
2. seja `x` um gancho de `y`,
3. acrescente o arco `x-y` e o vértice `y` a T,
4. atualize os preços e ganchos fora de T.

Para armazenar os preços dos vértices usaremos um vetor `preco[]` indexado pelos vértices. Os ganchos poderiam ser armazenados num vetor alocado especialmente para esse fim, mas é melhor armazená-los na parte "ociosa" do vetor de pais de T, ou seja, nas posições do vetor `pa[]` indexadas pelos vértices da fronteira de T. Com isso, os elementos de `pa[]` terão a seguinte interpretação: se `v` está em T então `pa[v]` é o pai de `v`, se `v` está na fronteira de T então `pa[v]` é o gancho de `v`, e nos demais casos `pa[v]` está indefinido. Poderíamos dizer que os elementos de `pa[]` indexados pelos vértices da fronteira são "pais provisórios", estando sujeitos a alterações nas próximas iterações.

As próximas seções usarão essas ideias para produzir duas implementações rápidas do algoritmo de Prim, uma para grafos densos e uma para grafos esparsos.

Exemplo B. Voltamos a examinar o grafo não-dirigido do exemplo A. Veja abaixo, mais uma vez, a lista de arestas e seus custos.



```
5-4 7-4 7-5 0-7 1-5 0-4 2-3 7-1 0-2 1-2 1-3 7-2 2-6 3-6 0-6 6-4
 35  37  28  16  32  38  17  19  26  36  29  34  40  52  58  93
```

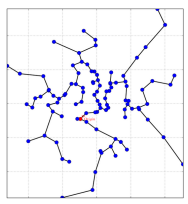
Aplique o algoritmo de Prim começando com o vértice `0`. Veja o conjunto de vértices de T e o estado dos vetores `pa[]` e `preco[]` no início de sucessivas iterações, com "`.`" indicando valores indefinidos e "`*`" indicando infinito.

```
                 pa[]             preco[]
T                0 1 2 3 4 5 6 7  0  1  2  3  4  5  6  7

0                0 . 0 . 0 . 0 0  0  * 26  * 38  * 58 16
0 7              0 7 0 . 7 7 0 0  0 19 26  * 37 28 58 16
0 7 1            0 7 0 1 7 7 0 0  0 19 26 29 37 28 58 16
0 7 1 2          0 7 0 2 7 7 2 0  0 19 26 17 37 28 40 16
0 7 1 2 3        0 7 0 2 7 7 2 0  0 19 26 17 37 28 40 16
0 7 1 2 3 5      0 7 0 2 5 7 2 0  0 19 26 17 35 28 40 16
0 7 1 2 3 5 4    0 7 0 2 5 7 2 0  0 19 26 17 35 28 40 16
0 7 1 2 3 5 4 6  0 7 0 2 5 7 2 0  0 19 26 17 35 28 40 16
```

Os ganchos e os preços dos vértices da fronteira estão pintados de cinza. Observe como os valores em cada coluna de `pa[]` mudam de uma iteração para a seguinte. A MST calculada pelo algoritmo tem arestas `0-7 7-1 0-2 2-3 7-5 5-4 2-6`. O custo dessa MST é `181`.

Exemplo C. O vídeo *Prim's Algorithm Animation* no YouTube aplica o algoritmo de Prim a um grafo cujos vértices são pontos aleatórios no plano. O grafo é completo e o custo de cada aresta é a distância geométrica entre suas pontas. O vídeo mostra muito bem os preços e os ganchos dos vértices que estão fora da árvore T. (Infelizmente, o vídeo usa a mesma cor para pintar as arestas da árvore e as arestas da franja.)



Exercícios 3

1. ★ Como começa uma iteração genérica da implementação eficiente do algoritmo de Prim? (Cuidado! Não se trata de dizer o que acontece no início de uma iteração. Trata-se, isto sim, dizer que informações estã disponíveis no início de uma iteração genérica, antes que a execução da iteração comece.)

2. *Caminho.* Calcule uma MST no grafo não-dirigido com custos descrito a seguir. Use as ideias de precos e ganchos.

```
0-1  1-6  6-5  5-3  3-2  2-4
 99   99   99   99   99   99
```

3. ★ Considere o grafo não-dirigido com custos nas arestas descrito abaixo. Seja T a árvore cujos arestas são `0-1 1-2 1-3` . Calcule os preços e os ganchos de todos os vértices da fronteira de T.

```
0-1 0-5 1-2 1-3 2-3 2-5 2-6 3-6 5-1 5-4 6-4
 10  60  10  15  20  50  40  50  15   0  20
```

4. Considere o grafo não-dirigido com custos nas arestas descrito abaixo. Use o algoritmo de Prim para encontrar uma MST. Use o vértice `1` como raiz. No começo de cada iteração, dê a árvore T bem como o preço e o gancho de cada vértice na fronteira de T.

```
0-1 0-4 1-5 2-0 2-3 2-4 4-3 5-0 5-2 6-4
 10  30  10  10  60  50  10  40  20   0
```

Primeira implementação eficiente

No início de cada iteração de nossa primeira implementação eficiente do algoritmo de Prim temos

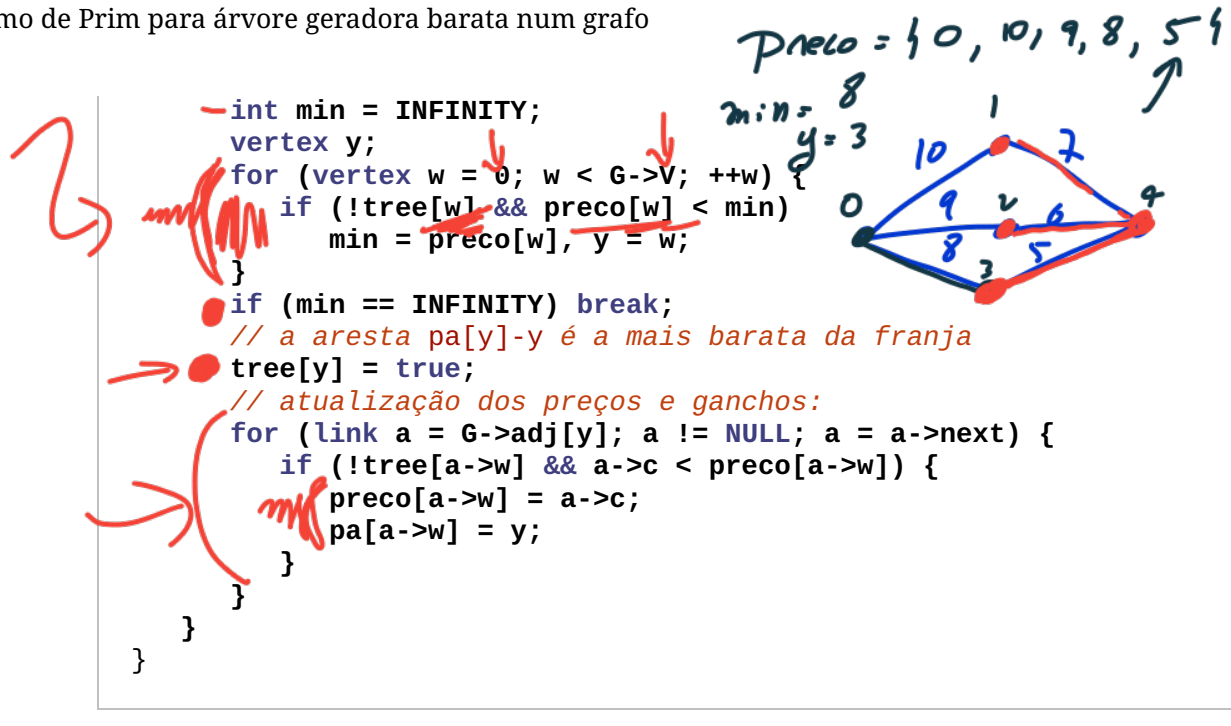
- o vetor característico `tree[]` do conjunto de vértices da árvore T,
- um vetor `preco[]` que contém o preço de cada vértice na fronteira de T,
- um vetor `pa[]` que contém os pais dos vértices de T e os ganchos dos vértices da fronteira de T.

```
#define UGraph Graph

/* Recebe um grafo não-dirigido conexo G com custos arbitrários nas arestas e calcula uma MST de G. A função trata
a MST como uma árvore radicada com raiz 0 e armazena a árvore no vetor de pais pa[0..V-1] alocado pelo usuário. (A
função é uma implementação do algoritmo de Prim. O código é uma versão melhorada do Programa 20.3 de
Sedgewick.) */

void UGRAPHmstP1( UGraph G, vertex *pa)
{
   bool tree[1000];
   int preco[1000];
   // inicialização:
   for (vertex v = 0; v < G->V; ++v)
      pa[v] = -1, tree[v] = false, preco[v] = INFINITY;
   pa[0] = 0, tree[0] = true;
   for (link a = G->adj[0]; a != NULL; a = a->next)
      pa[a->w] = 0, preco[a->w] = a->c;

   while (true) {
```




```
    int min = INFINITY;
    vertex y;
    for (vertex w = 0; w < G->V; ++w) {
       if (!tree[w] && preco[w] < min)
          min = preco[w], y = w;
    }
    if (min == INFINITY) break;
    // a aresta pa[y]-y é a mais barata da franja
    tree[y] = true;
    // atualização dos preços e ganchos:
    for (link a = G->adj[y]; a != NULL; a = a->next) {
       if (!tree[a->w] && a->c < preco[a->w]) {
          preco[a->w] = a->c;
          pa[a->w] = y;
       }
    }
 }
}
```

Podemos usar uma versão simplificada da inicialização se estivermos dispostos a aceitar que a primeira iteração do `while` não comece com os preços e ganchos definidos na fronteira da árvore, como isso acontece nas demais iterações:

```
// inicialização simplificada:
for (vertex v = 0; v < G->V; ++v)
   pa[v] = -1, tree[v] = false, preco[v] = INFINITY;
pa[0] = 0, preco[0] = INFINITY - 1;
```

Desempenho. Quando aplicada a um grafo não-dirigido com V vértices e E arestas, a função `UGRAPHmstP1()` consome tempo proporcional a $V^2 + E$. Como $E < V^2$, o consumo de tempo da função é proporcional a

$$V^2.$$

Como o tamanho de grafos densos é proporcional a V^2, podemos dizer que a função `UGRAPHmstP1()` é linear para grafos densos.

Exercícios 4

1. Como começa cada iteração do `while` na função `UGRAPHmstP1()`? Quais as informações de que a iteração dispõe?
2. ★ *Instâncias extremas.* A aplicação da função `UGRAPHmstP1()` a um grafo que tem apenas um vértice produz o resultado correto? E a aplicação a um grafo com apenas 2 vértices? E a aplicação a um grafo que consiste em um caminho apenas? Justique suas respostas diretamente a partir do código da função.
3. Verifique que a função `UGRAPHmstP1()` implementa corretamente o algoritmo de Prim. Verifique, em particular, que a inicialização simplificada está correta.
4. *Desempenho.* Mostre que `UGRAPHmstP1()` consome no máximo $V^2 + E$ unidades de tempo.
5. ★ Considere o grafo não-dirigido com custos nas arestas definido abaixo. Suponha que certa iteração de `UGRAPHmstP1()` começa com a árvore cujas arestas são `0-1` e `0-2`. Dê o estado dos vetores `pa[]` e `preco[]` no começo da iteração. (Não é necessário executar a função passo a passo a partir da primeira iteração; basta conhecer as definições de gancho e preço.)

```
0-1 0-2 1-2 3-4 3-5 3-6 4-1 4-2 4-6 5-1 6-0 6-1 6-2
 15  15  25  25  15  15  35  25  15  45  25  45  65
```

6. Aplique a função `UGRAPHmstP1()` ao grafo não-dirigido com custos nas arestas descrito abaixo. Faça o rastreamento da execução da função.

```
0-2 2-4 4-1 1-3 3-0 2-3 0-4
  8   2   3   0   8   0   4
```

7. *Variantes de código.* Analise, discuta, e critique as variantes de código da função `UGRAPHmstP1()` descritas na capítulo anexo.
8. Escreva uma versão simplificada da função `UGRAPHmstP1()` que receba um grafo não-dirigido conexo e devolva o custo de uma MST do grafo sem construir a MST explicitamente. Escreva código "enxuto", sem variáveis supérfluas.
9. Escreva uma versão da função `UGRAPHmstP1()` para grafos não-dirigidos conexos representados por matriz de adjacências. [Solução]
10. Compare o código da função `UGRAPHmstP1()` com o da função `GRAPHcptD1()` (que calcula uma CPT). Quais as diferenças? Quais as semelhanças?

Segunda implementação eficiente

A primeira implementação do algoritmo de Prim começa cada iteração examinando os vértices da fronteira da árvore, um por um, à procura do mais barato. Para acelerar esse processo, a implementação seguinte mantém os vértices da fronteira em ordem crescente de preços (ou quase isso), para que não seja preciso *procurar* o vértice mais barato.

Tal como na primeira implementação do algoritmo de Prim, cada iteração da segunda implementação começa como

- o vetor característico `tree[]` do conjunto de vértices da árvore T,

- um vetor `preco[]` que contém o preço de cada vértice da fronteira de T,
- um vetor `pa[]` que contém os pais dos vértices de T e os ganchos dos vértices da fronteira de T.

Mas, diferentemente da primeira implementação, os vértices que não pertencem a T são mantidos numa fila priorizada "de mínimo" (= *min priority queue*). (Veja capítulo 9 (Priority Queues and Heapsort) do volume 1 do livro de Sedgewick.) (Seria suficiente manter na fila priorizada os vértices da fronteira, mas é mais simples colocar na fila todos os vértices que estão fora de T.)

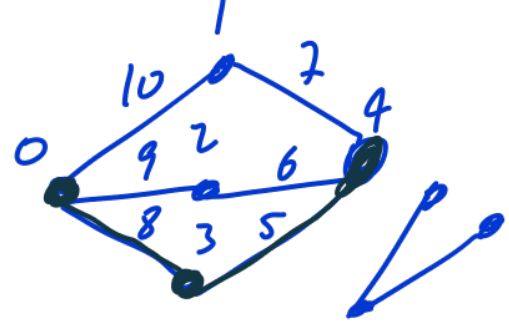
```
#define UGraph Graph

/* Recebe um grafo não-dirigido conexo G com custos arbitrários nas arestas e calcula uma MST de G.  A função tra-
ta a MST como uma árvore radicada com raiz 0 e armazena a árvore no vetor de pais pa[].  (A função implementa o
algoritmo de Prim. O grafo G e os custos são representados por listas de adjacência. O código abaixo foi inspirado
no Programa 21.1 de Sedgewick.) */

void UGRAPHmstP2( UGraph G, vertex *pa)
{
   bool tree[1000];
   int preco[1000];
   // inicialização:
   for (vertex v = 1; v < G->V; ++v)
      pa[v] = -1, tree[v] = false, preco[v] = INFINITY;
   pa[0] = 0, tree[0] = true;
   for (link a = G->adj[0]; a != NULL; a = a->next)
      pa[a->w] = 0, preco[a->w] = a->c;

   PQinit( G->V);
   for (vertex v = 1; v < G->V; ++v)
      PQinsert( v, preco);

   while (!PQempty( )) {
      vertex y = PQdelmin( preco);
      if (preco[y] == INFINITY) break;
      tree[y] = true;
      // atualização dos preços e ganchos:
      for (link a = G->adj[y]; a != NULL; a = a->next)
         if (!tree[a->w] && a->c < preco[a->w]) {
            preco[a->w] = a->c;
            PQdec( a->w, preco);
            pa[a->w] = y;
         }
   }
   PQfree( );
}
```



Os vértices que não pertencem à árvore T ficam armazenados numa fila priorizada "de mínimo" com prioridade ditada pelo preço de cada vértice. A fila é manipulada pelas seguintes funções:

- `PQinit(G->V)`: inicializa uma fila priorizada com capacidade para `G->V` vértices.
- `PQempty()`: devolve `true` se e somente se a fila está vazia.
- `PQinsert(w,preco)`: insere o vértice `w` na fila com prioridade `preco[w]`.
- `PQdelmin(preco)`: retira da fila um vértice `y` que minimiza `preco[]`.
- `PQdec(w,preco)`: reorganiza a fila depois que o valor de `preco[w]` diminuiu.

Desempenho. A implementação clássica da fila priorizada usa uma estrutura de *heap*. Com essa implementação da fila, a função `UGRAPHmstP2()` consome tempo proporcional a $(V+E) \log V$ no pior caso, sendo V o número de vértices e E o número de arestas de `G`. Como `G` é conexo, temos $E \geq V-1$ e portanto o consumo de tempo é proporcional a

$$E \log V$$

no pior caso. Portanto, `UGRAPHmstP2()` é apenas um pouco pior que linear. Podemos dizer que `UGRAPHmstP2()` é linearítmica.

Como se compara o consumo de tempo de `UGRAPHmstP2()` com o de `UGRAPHmstP1()`? Se nos restringirmos a grafos esparsos, `UGRAPHmstP2()` é assintoticamente mais rápida que `UGRAPHmstP1()`. Se nos restringirmos a grafos densos, a relação se inverte: `UGRAPHmstP1()` é assintoticamente mais rápida que `UGRAPHmstP2()`.

Exercícios 5

1. ★ *Instâncias extremas.* A aplicação da função `UGRAPHmstP2()` a um grafo que tem apenas um vértice produz o resultado correto? E a aplicação a um grafo com apenas dois vértices? E a aplicação a um grafo que consiste em um caminho apenas? Justique suas respostas diretamente a partir do código da função.
2. Verifique que a função `UGRAPHmstP2()` implementa corretamente o algoritmo de Prim.
3. Como começa cada iteração da função `UGRAPHmstP2()`? Quais as informações de que a iteração dispõe?
4. *Inicialização.* Escreva uma versão mais simples da inicialização, como já fizemos para `UGRAPHmstP1()`.
5. ★ Para que serve a linha "`if (preco[y] == INFINITY) break`" no código de `UGRAPHmstP2()`?

6. ★ Considere o grafo não-dirigido com custos nas arestas definido abaixo. Suponha que certa iteração de `UGRAPHmstP2()` começa com a árvore cujas aresta são `0-1` e `0-2`. Dê o estado dos vetores `pa[]` e `preco[]` no início da iteração. Supondo que a fila priorizada está implementada em um heap, dê o estado do vetor `pq[]` no início da iteração. (Não é preciso executar a função passo a passo a partir da primeira iteração; basta conhecer as definições de preço e gancho.)

```
0-1 0-2 1-2 3-4 3-5 3-6 4-1 4-2 4-6 5-1 6-0 6-1 6-2
 15  15  25  25  15  15  35  25  15  45  25  45  65
```

7. *Desempenho.* Suponha que a fila priorizada é implementada em um heap. Mostre que `UGRAPHmstP2()` consome no máximo $(V + E) \log V$ unidades de tempo.

8. [Sedgewick 20.34] Descreva uma família de grafos não-dirigidos conexos que force a função `UGRAPHmstP2()` com fila priorizada implementada em um heap a consumir $E \log V$ unidades de tempo, sendo V o número de vértices e E o número de arestas.

9. Escreva uma versão simplificada da função `UGRAPHmstP2()` que receba um grafo não-dirigido conexo e devolva apenas o custo de uma MST do grafo. Escreva código "enxuto", sem variáveis supérfluas.

10. Escreva uma versão all-in-one da função `UGRAPHmstP2()` que incorpore, tanto quanto razoável, o código das funções de manipulação da fila priorizada.

11. [Sedgewick 20.39] *Fila priorizada simplória.* Escreva uma implementação da fila priorizada em que a fila é representada por um vetor `pq[i..k]` de vértices que tem preços crescentes (ou seja, `preco[pq[i]]` ≤ ... ≤ `preco[pq[k]]`). Estime o consumo de tempo de cada uma das funções `PQinit()`, `PQempty()`, `PQinsert()`, `PQdelmin()` e `PQdec()`. Repita o exercício com vetor `pq[1..k]` de preços decrescentes.

12. *Variantes de código.* Analise, discuta, e critique as variantes de código da função `UGRAPHmstP2()` descritas na capítulo anexo.

13. [Sedgewick 20.36] Adapte o código da função `UGRAPHmstP2()` para grafos não-dirigidos conexos representados por matriz de adjacências.

14. Compare o código da função `UGRAPHmstP2()` com o da função `GRAPHcptD2()` (que calcula uma CPT). Quais as diferenças? Quais as semelhanças?

Exercícios 6

1. *UGRAPHmstP3( ).* Escreva uma variante da função `UGRAPHmstP2()` em que apenas os vértices da fronteira de T são colocados na fila priorizada.

2. *Atualize suas bibliotecas.* Acrescente as implementações do algoritmo de Prim discutidas neste capítulo à biblioteca `GRAPHlists`. Também acrescente as versões apropriadas das funções à biblioteca `GRAPHmatrix`. Atualize os correspondentes arquivos-interface. Use `malloc()` para alocar os vetores `tree[]` e `preco[]`.

3. Que acontece se as funções `UGRAPHmstP0()`, `UGRAPHmstP1()` ou `UGRAPHmstP2()` forem aplicadas a um grafo não-dirigido desconexo?

4. [Sedgewick 20.40] Seja G um grafo não-dirigido conexo com custos nas arestas. Uma aresta e de G é *crítica* se o custo de uma MST de $G - e$ é maior que o custo de uma MST de G. Escreva uma função que determine todas as arestas críticas de G em tempo proporcional a $E \log V$.

5. [Sedgewick 20.46] Faça testes empíricos para determinar até que ponto o consumo de tempo do algoritmo de Prim depende do primeiro vértice escolhido pelo algoritmo. (As implementações acima escolhem `0` para primeiro vértice.) Vale a pena escolher o primeiro vértice aleatoriamente?

6. Seja G um grafo não-dirigido conexo com custos nas arestas e seja S uma subárvore (não necessariamente geradora) de G. Dê um algoritmo eficiente que encontre uma subárvore geradora de G que tenha custo mínimo *dentre as que contêm* S.

7. *Desafio.* Encontre um algoritmo *linear* para o problema da MST, ou seja, um algoritmo que consuma tempo proporcional a $V+E$ no pior caso para calcular uma MST de um grafo não-dirigido conexo com V vértices e E arestas.

Perguntas e respostas

- PERGUNTA: Os processos iterativos das funções `UGRAPHmstP0()`, `UGRAPHmstP1()` e `UGRAPHmstP2()` são controlados por um `while (true)`. Como o grafo é conexo, não seria mais natural controlá-los por um `for (int i = 0; i < G->V-1; ++i)`?

  RESPOSTA: É verdade. Só estou usando `while (true)` para que não seja necessário verificar se o grafo é conexo antes de invocar as funções. Se forem invocadas com um grafo desconexo, as funções produzirão uma árvore geradora *na componente conexa do grafo que contém o vértice* `0`.

Veja o vídeo *Prim's Algorithm Animation* no YouTube.

Veja #4.3 na página Lecture Slides do livro *Algorithms, 4th.ed.* de Sedgewick e Wayne.

---

www.ime.usp.br/~pf/algoritmos_para_grafos/
Atualizado em 2019-10-15
© *Paulo Feofiloff*
IME-USP