Algoritmos para Grafos, via Sedgewick | Índice remissivo

Algoritmo de Prim

Nosso problema nesta página é o mesmo da página anterior: encontrar uma MST (árvore geradora mínima) de um grafo *G* com custos nas arestas.

(Esta página é um resumo da seção 20.3 (Prim's Algorithm and Priority-First Search), p.235-245, do livro de Sedgewick.)

O algoritmo

O algoritmo de Prim (publicado em 1961) se apoia nas condições de otimalidade de MSTs para encontrar uma MST de um grafo *G* com custos nas arestas. (Os custos são números reais arbitrários, não necessariamente todos positivos.)

Para descrever o algoritmo, convém recorrer ao conceito de franja. A franja (= *fringe*) de uma subárvore não geradora *T* de *G* é o conjunto de todas as arestas de *G* que têm uma ponta em *T* e outra ponta fora. Se denotarmos por *X* o conjunto dos vértices de *T* e por *Y* o conjunto dos vértices fora de *X*, podemos dizer que a franja é o conjunto das arestas que pertencem ao corte (*X*,*Y*).

No início de cada iteração do algoritmo de Prim temos uma árvore *T*. (No início da primeira iteração, *T* consiste em um único vértice.) Cada iteração consiste no seguinte:

> **se** a franja de *T* não é vazia
> > **então** seja *e* uma aresta de custo mínimo na franja
> > comece nova iteração com *T*+*e* no papel de *T*
> >
> > **senão** pare

Se *G* for conexo, o algoritmo produz uma MST de *G*. Caso contrário, o algoritmo produz uma MST de uma das componentes de *G*.

Exercícios

1. [IMPORTANTE] Prove que o algoritmo de Prim produz uma MST de qualquer grafo conexo com custos nas arestas. (Sugestão: use as propriedades da troca de arestas.)
2. Mostre que a seguinte estratégia pode não encontrar uma MST de um grafo *G*. Cada iteração começa com uma subárvore (não necessariamente geradora) *T* de *G*. Cada iteração consiste no seguinte: (1) tome o vértice *v* que foi acrescentado a *T* mais recentemente e escolha uma aresta de custo mínimo *e* dentre as que incidem em *v* e estão na franja de *T*; (2) comece nova iteração com *T*+*e* no papel de *T*.

Implementação grosseira do algoritmo

Nossa primeira implementação do algoritmo de Prim é simples e óbvia mas ineficiente. A função abaixo recebe um grafo `G` com custos nas arestas e calcula uma MST do componente de `G` que contém o vértice `0`. A MST é tratada como uma arborescência com raiz `0` e armazenada no vetor de pais `parent`.

A função supõe que o grafo é representado por sua matriz de adjacência e o custo de cada aresta é estritamente menor que `maxCOST`.

```
void bruteforcePrim (Graph G, Vertex parent[]) {
   Vertex v, w;
   for (w = 0; w < G->V; w++) parent[w] = -1;
   parent[0] = 0;
   while (1) {
      double mincost = maxCOST;
      Vertex v0, w0;
      for (w = 0; w < G->V; w++)
         if (parent[w] == -1)
            for (v = 0; v < G->V; v++)
               if (parent[v] != -1 && mincost > G->adj[v][w])
                  mincost = G->adj[v0=v][w0=w];
      if (mincost == maxCOST) break;
      /* A */
      parent[w0] = v0;
   }
}
```

No ponto *A*, `v0-w0` é uma aresta de custo mínimo dentre as que estão na franja da árvore. O custo da aresta `v0-w0` é `mincost`.

Exercícios

3. Qual o consumo de tempo da função bruteforcePrim?
4. Quanto tempo consome a função bruteforcePrim quando aplicada a um grafo completo com custos nas arestas?
5. Qual o custo de uma MST do grafo 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 cujos vértices são os seguintes pontos no plano:

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

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. Aplique o algoritmo de Prim a esse grafo. Exiba uma figura do grafo e da árvore no início de cada iteração.

7. 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

Toda implementação eficiente do algoritmo de Prim depende do conceito de custo de um vértice em relação a uma árvore. Dada uma árvore não geradora do grafo, o custo de um vértice `w` que está fora da árvore é o custo de uma aresta mínima dentre as que incidem em `w` e estão na franja da árvore. Em outras palavras, o custo de `w` é o custo de uma aresta mínima dentre as que têm uma ponta na árvore e outra em `w`. Se nenhuma aresta da franja incide em `w`, o custo de `w` é `maxCOST` (que é maior que o custo de qualquer aresta e portanto tem o sabor de ∞).

Nas implementações que examinaremos abaixo, os custos dos vértices e as arestas que justificam esses custos são representados pelos vetores

`cost` e `frj`.

O custo do vértice `w` em relação à árvore é `cost[w]`. Para cada vértice `w` fora da árvore, o vértice `frj[w]` está na árvore e a aresta que liga `w` a `frj[w]` tem custo `cost[w]`. Cada iteração do algoritmo de Prim escolhe um vértice `w` fora da árvore e adota `frj[w]` como valor de `parent[w]`.

Implementação eficiente para grafos densos

A implementação abaixo é ótima para grafos densos. É apropriado, portanto, representar o grafo por uma matriz de adjacência:

```
/* Recebe grafo G com custos nas arestas e calcula uma MST do
componente de G que contém o vértice 0.  A função armazena a MST
no vetor parent, tratando-a como uma arborescência com raiz 0. */

/* O grafo G é representado por sua matriz de adjacência. A
função supõe que maxCOST > 0 e e.cost < maxCOST para cada
aresta e.  Supõe também que o grafo tem no máximo maxV vértices.
O código abaixo é uma versão melhorada do Programa 20.3, p.238,
```

```
de Sedgewick. */

void GRAPHmstP1 (Graph G, Vertex parent[]) {
   double cost[maxV]; Vertex v0, w, frj[maxV];
   for (w = 1; w < G->V; w++) {
      parent[w] = -1;
      frj[w] = 0;
      cost[w] = G->adj[0][w];
   }
   parent[0] = 0;
   while (1) {
      double mincost = maxCOST;
      for (w = 0; w < G->V; w++)
         if (parent[w] == -1 && mincost > cost[w])
            mincost = cost[v0=w];
      if (mincost == maxCOST) break;
      parent[v0] = frj[v0];
      for (w = 0; w < G->V; w++)
         if (parent[w] == -1 && cost[w] > G->adj[v0][w]) {
            cost[w] = G->adj[v0][w];
            frj[w] = v0;
         }
   }
}
```

O fragmento de código

```
if (cost[w] > G->adj[v0][w]) {
    cost[w] = G->adj[v0][w];
}
```

é característico do algoritmo de Prim. A operação que ele executa é conhecida como relaxação (da aresta `v0-w`). Essa operação aparece em toda implementação do algoritmo.

Desempenho. No pior caso, o consumo tempo da função `GRAPHmstP1` é proporcional a

$$V^2.$$

Portanto, a função `GRAPHmstP1` é linear para grafos densos (pois o tamanho de tais grafos é proporcional a V^2).

Exercícios

8. [BOM!] Considere o grafo com custos nas arestas definido abaixo:

```
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
1.5 1.5 2.5 2.5 1.5 1.5 3.5 2.5 1.5 4.5 2.5 4.5 6.5
```

Suponha que certa iteração de `GRAPHmstP1` começa com a árvore cujas aresta são `0-1` e `0-2`. Dê o estado dos vetores `frj` e `cost`. (Dica: Não é preciso executar a função passo a passo; basta conhecer as propriedades de `frj` e `cost`.)

9. Discuta a seguinte variante da função `GRAPHmstP1`:

```
void GRAPHmstP1 (Graph G, Vertex parent[]) {
   double cost[maxV]; Vertex v0, w, frj[maxV];
   for (w = 0; w < G->V; w++) {
      parent[w] = -1;
      cost[w] = maxCOST;
   }
   v0 = 0;
   frj[v0] = v0;
   cost[v0] = 0.0;
   while (1) {
      double mincost = maxCOST;
      for (w = 0; w < G->V; w++)
         if (parent[w] == -1 && mincost > cost[w])
            mincost = cost[v0=w];
      if (mincost == maxCOST) break;
      parent[v0] = frj[v0];
      for (w = 0; w < G->V; w++)
         if (parent[w] == -1 && cost[w] > G->adj[v0][w]) {
            cost[w] = G->adj[v0][w];
            frj[w] = v0;
         }
   }
}
```

10. Discuta e critique a seguinte variante da função `GRAPHmstP1`:

```
void GRAPHmstP1(Graph G, Vertex parent[]) {
   double cost[maxV], mincost;
   Vertex v, w, v0, frj[maxV];
   for (v = 0; v < G->V; v++) {
      parent[v] = -1;
      cost[v] = maxCOST;
   }
   v0 = 0; parent[v0] = v0;
   while (1) {
      for (w = 0; w < G->V; w++)
         if (parent[w] == -1)
            if (cost[w] > G->adj[v0][w]) {
               cost[w] = G->adj[v0][w];
               frj[w] = v0;
            }
      mincost = maxCOST;
      for (w = 0; w < G->V; w++)
         if (parent[w] == -1 && mincost > cost[w])
            mincost = cost[v0=w];
      if (mincost == maxCOST) break;
      parent[v0] = frj[v0];
   }
}
```

11. Discuta e critique o programa 20.3, p.238, de Sedgewick, reproduzido abaixo. Trata-se de uma redação alternativa da função `GRAPHmstP1` da seção anterior. (O código trata `G->V` como um vértice fictício e define `cost[G->V] == maxCOST`.)

```
static Vertex frj[maxV];
void GRAPHmstP(Graph G, Vertex parent[], double cost[]) {
   Vertex v, w, v0;
   for (v = 0; v < G->V; v++) {
```

```
            parent[v] = -1;
            frj[v] = v;
            cost[v] = maxCOST;
         }
         parent[0] = 0;
         cost[G->V] = maxCOST;
         for (v0 = 0; v0 != G->V; ) {
            parent[v0] = frj[v0];
            v = v0;
            for (w = 0, v0 = G->V; w < G->V; w++)
               if (parent[w] == -1) {
                  if (G->adj[v][w] < cost[w]) {
                     cost[w] = G->adj[v][w];
                     frj[w] = v;
                  }
                  if (cost[w] < cost[v0]) v0 = w;
               }
         }
      }
```

12. Escreva uma versão simplificada da função `GRAPHmstP1` que receba um grafo conexo e devolva o custo de uma MST do grafo sem construir a MST explicitamente. Escreva código "enxuto", sem variáveis supérfluas.

13. [INVARIANTES] Enuncie as propriedades que valem no início de cada iteração de `GRAPHmstP1` e explicam o funcionamento da função. Prove essas propriedades.

14. Escreva uma versão da função `GRAPHmstP1` para grafos representados por listas de adjacência.

Implementação eficiente para grafos esparsos

Esta seção discute uma implementação mais sofisticada do algoritmo de Prim. Ela usa uma fila de prioridades (= *priority queue*) para escolher, eficientemente, a próxima aresta a ser acrescentada à árvore.

```
static double cost[maxV];

/* Recebe grafo G com custos nas arestas e calcula uma MST do
componente de G que contém o vértice 0.  A função armazena a MST
no vetor parent, tratando-a como uma arborescência com raiz 0. */

/* O grafo G é representado por listas de adjacência.  (O código
abaixo foi copiado do Programa 20.4, p.242, de Sedgewick.) */

void GRAPHmstP2 (Graph G, Vertex parent[]) {
   Vertex v0, w, frj[maxV]; link p;
   PQinit();
   for (w = 0; w < G->V; w++)
      parent[w] = frj[w] = -1;
   parent[0] = 0;
   for (p = G->adj[0]; p != NULL; p = p->next) {
      cost[p->w] = p->cost;
```

```
        PQinsert(p->w);
        frj[p->w] = 0;
    }
    while (!PQempty()) {
        v0 = PQdelmin();
        parent[v0] = frj[v0];
        for (p = G->adj[v0]; p != NULL; p = p->next) {
            w = p->w;
            if (parent[w] == -1) {
                if (frj[w] == -1) {
                    cost[w] = p->cost;
                    PQinsert(w);
                    frj[w] = v0;
                }
                else if (cost[w] > p->cost) {
                    cost[w] = p->cost;
                    PQdec(w);
                    frj[w] = v0;
                }
            }
        }
    }
}
```

(Note a operação de relaxação `if (cost[w] > p->cost) {cost[w] = p->cost;}` característica do algoritmo de Prim.)

A função `GRAPHmstP2` usa uma fila com prioridades. (Veja capítulo 9 (Priority Queues and Heapsort), p.389, do volume 1 do livro de Sedgewick.) A fila é manipulada pelas seguintes funções:

- `PQinit()`: inicializa uma fila de vértices em que cada vértice `v` tem prioridade `cost[v]`.
- `PQempty()`: devolve `1` se a fila estiver vazia e `0` em caso contrário.
- `PQinsert(v)`: insere o vértice `v` na fila.
- `PQdelmin()`: retira da fila um vértice de prioridade mínima.
- `PQdec(w)`: reorganiza a fila depois que o valor de `cost[w]` foi decrementado.

A implementação clássica da fila de prioridades usa uma estrutura de *heap*. O heap é armazenado num vetor `pq[1..N]` (a posição `0` do vetor não é usada). A prioridade de um vértice `pq[k]` no heap é `cost[pq[k]]`. Propriedade fundamental do heap:

`cost[pq[k/2]] ≤ cost[pq[k]]`

para `k=2,..,N`. Portanto, o vértice `pq[1]` minimiza `cost`.

```
/* O código abaixo é uma adaptação do programa 9.12, p.391, do
```

```
volume 1 do livro de Sedgewick.  Supõe-se que N ≤ maxV. */

/* O vetor qp é o "inverso" de pq:  para cada vértice v, qp[v] é
o único índice tal que pq[qp[v]] == v.  É claro que qp[pq[i]] ==
i para todo i. */

static Vertex pq[maxV+1];
static int N;
static int qp[maxV];

void PQinit(void) {
  N = 0;
}
int PQempty(void) {
   return N == 0;
}
void PQinsert(Vertex v) {
   qp[v] = ++N;
   pq[N] = v;
   fixUp(N);
}
Vertex PQdelmin(void) {
   exch(1, N);
   --N;
   fixDown(1);
   return pq[N+1];
}
void PQdec(Vertex w) {
   fixUp(qp[w]);
}
static void exch(int i, int j) {
   Vertex t;
   t = pq[i]; pq[i] = pq[j]; pq[j] = t;
   qp[pq[i]] = i;
   qp[pq[j]] = j;
}
static void fixUp(int k) {
   while (k > 1 && cost[pq[k/2]] > cost[pq[k]]) {
      exch(k/2, k);
      k = k/2;
}
static void fixDown(int k) {
   int j;
   while (2*k <= N) {
      j = 2*k;
      if (j < N && cost[pq[j]] > cost[pq[j+1]]) j++;
      if (cost[pq[k]] <= cost[pq[j]]) break;
      exch(k, j);
      k = j;
   }
}
```

(O código de `GRAPHmstP2` pode parecer um pouco assustador porque depende de um grande número de funções auxiliares. É um bom exercício escrever uma versão "compacta" da função `GRAPHmstP2`, que incorpore, tanto quanto razoável, o código das funções de manipulação da fila de prioridades.)

Desempenho. Eis uma estimativa do consumo de tempo no pior caso de cada uma das funções de manipulação da fila de prioridades quando aplicada a um grafo com `V` vértices:

- `PQinit`: constante, ou seja, não depende de `V`;
- `PQempty`: constante, ou seja, não depende de `V`;
- `PQinsert`: proporcional a `lg(V)`;
- `PQdelmin`: proporcional a `lg(V)`;
- `PQdec`: proporcional a `lg(V)`.

Assim, o consumo de tempo da função `GRAPHmstP2` é proporcional a `V lg(V) + E lg(V)` no pior caso. Para grafos conexos, essa expressão se reduz a

`E lg(V)`.

Portanto, a função `GRAPHmstP2` é um pouco pior que linear. Mesmo assim, esse desempenho é melhor que o da função `GRAPHmstP1` quando os grafos são esparsos.

Exercícios

15. Analise e discuta a seguinte versão de `GRAPHmstP2`:

```
void GRAPHmstP2 (Graph G, Vertex parent[]) {
   Vertex v0, w, frj[maxV]; link p;
   PQinit();
   for (w = 0; w < G->V; w++)
      parent[w] = frj[w] = -1;
   v0 = 0;
   frj[v0] = v0;
   cost[v0] = 0.0;
   PQinsert(v0);
   while (!PQempty()) {
      v0 = PQdelmin();
      parent[v0] = frj[v0];
      for (p = G->adj[v0]; p != NULL; p = p->next) {
         w = p->w;
         if (parent[w] == -1) {
            if (frj[w] == -1) {
               cost[w] = p->cost;
               PQinsert(w);
               frj[w] = v0;
            }
            else if (cost[w] > p->cost) {
               cost[w] = p->cost;
               PQdec(w);
               frj[w] = v0;
            }
         }
      }
   }
}
```

16. Discuta e critique a seguinte versão de `GRAPHmstP2`:

```
void GRAPHmstP2 (Graph G, Vertex parent[]) {
   Vertex v0, w, frj[maxV]; link p;
   PQinit();
   for (w = 0; w < G->V; w++)
      parent[w] = frj[w] = -1;
   v0 = 0; parent[v0] = v0;
   while (1) {
      for (p = G->adj[v0]; p != NULL; p = p->next) {
         w = p->w;
         if (parent[w] == -1) {
            if (frj[w] == -1) {
               cost[w] = p->cost;
               PQinsert(w);
               frj[w] = v0;
            }
            else if (cost[w] > p->cost) {
               cost[w] = p->cost;
               PQdec(w);
               frj[w] = v0;
            }
         }
      }
      if (PQempty()) break;
      v0 = PQdelmin();
      parent[v0] = frj[v0];
   }
}
```

17. [BOM!] Considere o grafo com custos nas arestas definido abaixo:

```
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
1.5 1.5 2.5 2.5 1.5 1.5 3.5 2.5 1.5 4.5 2.5 4.5 6.5
```

Suponha que certa iteração de `GRAPHmstP2` começa com a árvore cujas aresta são `0-1` e `0-2`. Dê o estado dos vetores `frj` e `cost`. Dê o estado do vetor `pq`, supondo que a fila de prioridades está implementada como um *heap*. (Dica: Não é preciso executar a função passo a passo; basta conhecer as propriedades de `frj` e `cost`.)

18. [INVARIANTES] Enuncie as propriedades que valem no início de cada iteração de `GRAPHmstP2` e explicam o funcionamento da função. Prove essas propriedades.

19. Descreva uma família de grafos com `V` vértices e `E` arestas que force a função `GRAPHmstP2` a consumir tempo proporcional a `E log(V)`.

20. Escreva uma implementação da fila de prioridade em que a fila é, simplesmente, um vetor crescente `pq[1..N]`. Estime o consumo de tempo de cada uma das funções `PQinit`, `PQempty`, `PQinsert`, `PQdelmin` e `PQdec`. Repita tudo com vetor decrescente.

21. Escreva uma implementação trivial da fila de prioridade em que a fila é um vetor `pq [1..N]` cujos elementos estão em ordem arbitrária. Estime o consumo de tempo de cada uma das funções `PQinit`, `PQempty`, `PQinsert`, `PQdelmin` e `PQdec`. Faça testes para comparar o desempenho dessa implementação com o desempenho de `GRAPHmstP2`.

22. Adapte o código da função `GRAPHmstP2` para grafos representados por matriz de

adjacência.

Outra implementação para grafos esparsos

O código abaixo é uma variante da função `GRAPHmstP2`. Nessa variante, os vértices são todos colocados na fila de prioridades antes do início do processo iterativo. O vetor `parent` usurpa o papel de `frj` e `frj` é dispensado. Com isso, o valor de cada elemento de `parent` pode ser alterados várias vezes ao longo do processo iterativo (ao contrário do que acontece em `GRAPHmstP2`).

O código dessa variante é mais curto que o de `GRAPHmstP2` (embora não seja mais eficiente). Por isso, há quem considere essa variante mais atraente.

```
/* (Código inspirado no Programa 21.1, p.284, de Sedgewick.) */

static double cost[maxV];

void GRAPHmstP3 (Graph G, Vertex parent[]) {
   Vertex v0, w; link p;
   for (w = 1; w < G->V; w++) {
      parent[w] = -1;
      cost[w] = maxCOST;
   }
   parent[0] = 0;
   for (p = G->adj[0]; p != NULL; p = p->next)
      cost[p->w] = p->cost;
   PQinit();
   for (w = 1; w < G->V; w++)
      PQinsert(w);
   while (!PQempty()) {
      v0 = PQdelmin();
      if (cost[v0] == maxCOST) break;
      for (p = G->adj[v0]; p != NULL; p = p->next) {
         w = p->w;
         if (cost[w] > p->cost) {
            cost[w] = p->cost;
            PQdec(w);
            parent[w] = v0;
         }
      }
   }
}
```

Exercícios

23. [INVARIANTES] Enuncie as propriedades que valem no início de cada iteração de `GRAPHmstP3` e explicam o funcionamento da função. Prove essas propriedades.

24. Discuta e critique a seguinte variante da função `GRAPHmstP3`:

```
static double cost[maxV];
void GRAPHmstP3 (Graph G, Vertex parent[]) {
   Vertex v0, w; link p;
   PQinit();
   for (w = 0; w < G->V; w++) {
      parent[w] = -1;
      cost[w] = maxCOST;
      PQinsert(w);
   }
   v0 = 0; parent[v0] = v0;
   while (1) {
      for (p = G->adj[v0]; p != NULL; p = p->next) {
         w = p->w;
         if (cost[w] > p->cost) {
            cost[w] = p->cost;
            PQdec(w);
            parent[w] = v0;
         }
      }
      if (PQempty()) break;
      v0 = PQdelmin();
      if (cost[v0] == maxCOST) break;
   }
}
```

Mais exercícios

25. Uma aresta *e* de um grafo *G* é *crítica* se o custo de uma MST de $G-e$ é estritamente menor que o custo de uma MST de *G*. Escreva uma função que determine todas as aresta críticas de *G* em tempo proporcional a $E \log(V)$.

26. Faça testes empíricos para determinar até que ponto o consumo de tempo do algoritmo de Prim depende do primeiro vértice escolhido pelo algoritmo. Vale a pena escolher esse vértice aleatoriamente?

URL of this site: http://www.ime.usp.br/~pf/algoritmos_para_grafos/
Last modified: Fri Feb 3 08:18:09 BRST 2012
Paulo Feofiloff
IME-USP