Algoritmo de Boruvka

Esta página discute mais um algoritmo para o problema de encontrar uma MST (árvore geradora mínima) em um grafo com custos nas arestas.

(Esta página é um resumo da seção 20.5 (Boruvka's Algorithm), p.252-255, do livro de Sedgewick.)

O algoritmo

O algoritmo de Boruvka (publicado em 1926) se apoia nas condições de otimalidade de MSTs para encontrar uma MST de um grafo com custos nas arestas.

Vamos nos restringir a grafos conexos. Essa restrição simplifica a discussão, embora seja irrelevante para o algoritmo. (Se for aplicado a um grafo desconexo, o algoritmo produzirá uma MST em cada componente.)

Para descrever o algoritmo, é importante lembrar que cada aresta um grafo consiste em dois arcos anti-paralelos. Vamos supor (em consonância com nossa estrutura de dados) que o custo de cada um desses dois arcos é igual ao custo da aresta. O algoritmo manipula os dois arcos de cada aresta independentemente. Assim, durante a execução do algoritmo, pode ocorrer que apenas um dos arcos de algumas arestas é escolhido para fazer parte da árvore geradora. No fim, entretanto, ambos os arcos de cada aresta da árvore estarão presentes.

Será necessário fazer uma adaptação previsível do conceito de franja. Diremos que a franja (=*fringe*) de uma subárvore T de nosso grafo G é o conjunto de todos os arcos que saem de T, ou seja, têm ponta inicial em T e ponta final fora de T. Diremos, ainda, que um arco é externo a uma floresta geradora F de G se tiver pontas em duas componentes diferentes de F. É claro que cada arco externo a F está na franja de exatamente uma das componentes de F.

Cada iteração do algoritmo de Boruvka começa com uma floresta geradora F de G. (No início da primeira iteração, cada componente de F tem apenas um vértice.) Cada iteração consiste no seguinte:

1. **se** existe algum arco externo a F
    1. **então** para cada componente T de F
    2. escolha um arco mínimo na franja de T
    3. seja B o conjunto de todos os arcos escolhidos
    4. seja B' um subconjunto maximal de B tal que
    5. $F+B'$ não tem ciclos de comprimento $\geqslant 3$
    6. seja B'' o conjunto dos arcos anti-paralelos aos de B'
    7. comece nova iteração com $F+B'+B''$ no papel de F
    8. **senão** pare

Esse algoritmo produz uma MST de G (uma vez que estamos supondo G conexo).

Diremos que B é um conjunto de Boruvka. Cada componente da floresta F contribui um arco de sua franja para esse conjunto. (É curioso o sabor de processamento paralelo desse algoritmo.)

Se o grafo $F + B$ não tiver ciclos de comprimento $\geqslant 3$, então B' será igual a B. Caso contrário, será preciso descartar alguns arcos de B para obter B'. O cálculo de B' e B'' e a substituição de F por $F+B'+B''$ são realizados, simultaneamente, pelo seguinte processo iterativo: examine os arcos de B um a um; se um arco for externo à floresta corrente, acrescente-o à floresta juntamente com seu arco anti-paralelo; senão, descarte-o.

EXEMPLO: A tabela abaixo define um grafo e os custos de suas arestas.

0-1	0-2	1-2	1-3
0.1	0.1	0.1	0.2

No início da primeira iteração do algoritmo de Boruvka, cada componente da floresta F tem um único vértice. Suponha que o algoritmo escolhe os seguintes arcos:

componente	0	1	2	3
arco na franja	0-2	1-0	2-1	3-1

O grafo $F + B$ contém o ciclo `0-2-1-0`, de comprimento $\geqslant 3$. Para obter B', basta eliminar qualquer um dos três arcos do ciclo. Se eliminarmos o arco `2-1`, a próxima iteração começará com a floresta que tem arestas `0-2`, `0-1` e `1-3`. (Se o algoritmo tivesse escolhido o conjunto de arcos abaixo, o grafo $F+B$ não teria ciclos de comprimento $\geqslant 3$, e nesse caso B' seria igual a B.)

componente	0	1	2	3
arco na franja	0-2	1-0	2-0	3-1

Exercícios

1. Aplique o algoritmo de Boruvka ao grafo abaixo. Em cada iteração, diga qual arco você escolheu na franja de cada componente da floresta.

```
0-1  0-2  0-3  1-2  2-3  3-1
0.3  0.5  0.5  0.7  0.7  0.7
```

2. Aplique o algoritmo de Boruvka ao grafo abaixo. Em cada iteração, diga qual arco você escolheu na franja de cada componente da floresta.

```
0-4  0-1  5-4  4-2  0-3  1-6  6-2  2-7  7-3  3-5  5-1
0.0  0.3  0.4  0.5  0.5  0.7  0.0  0.7  0.0  0.7  0.0
```

3. Aplique o algoritmo de Boruvka ao grafo abaixo. Em cada iteração, diga qual arco você escolheu na franja de cada componente da floresta.

```
0-1  0-2  0-3  1-2  2-3  3-1
0.2  0.5  0.2  0.2  0.2  0.7
```

4. Seja F uma floresta geradora de um grafo com custos nas arestas. Para cada componente T de F, seja b_T um arco de custo mínimo na franja de T. Seja B o conjunto de todos os arcos b_T. Seja C um ciclo de comprimento $\geqslant 3$ no grafo $F+B$. Mostre que todos os arcos de C que estão em B têm o mesmo custo. Tire daí a seguinte conclusão: se os custos das arestas do grafo são distintos dois a dois então o grafo $F+B$ é uma floresta.

5. Prove que o algoritmo de Boruvka produz uma MST de qualquer grafo conexo com custos nas arestas.

Implementação grosseira do algoritmo

Nossa primeira implementação do algoritmo de Boruvka é simples mas um tanto ineficiente. Como no algoritmo de Kruskal, um dos vértices de cada componente da floresta será o representante da componente. Denotaremos por `id[v]` o representante da componente que contém o vértice `v`. Se `i` é o representante de uma componente então `id[i]` é igual a `i`.

O código abaixo supõe que o grafo é representado por sua matriz de adjacência, que o custo de cada aresta é estritamente menor que `maxCST` e que `adj[v][w]` é igual a `maxCST` se e somente se `v-w` não é aresta.

Cada iteração começa com uma floresta F e tem duas fases. Na primeira fase, o algoritmo escolhe um conjunto de Boruvka B e armazena os arcos desse conjunto num vetor `bvka`. Na segunda, o algoritmo incorpora uma parte B' de B, juntamente com os arcos anti-paralelos aos de B', à floresta F.

```
#define maxCST 1000000.0

void bruteforceBoruvka (Graph G) {
   Arc bvka[maxV];
   Vertex i, j, u, v, w, z, idv, idw;
   int AA;
   for (i = 0; i < G->V; i++) id[i] = i;
   while (1) {
      for (u = 0; u < G->V; u++)
         bvka[u] = ARC(u, u, maxCST);
      AA = 0;
      for (v = 0; v < G->V; v++) {
         for (w = 0; w < G->V; w++) {
            double cst = G->adj[v][w];
            if (cst == maxCST || v == w) continue;
            i = id[v]; j = id[w];
            if (i != j) {
               AA++;
               if (bvka[i].cst > cst)
                  bvka[i] = ARC(v, w, cst);
            }
         }
      }
      if (AA == 0) break;
      for (u = 0; u < G->V; u++) {
         Arc e = bvka[u];
         idv = id[e.v]; idw = id[e.w];
         if (idv != idw) {
            printf("%d-%d ", e.v, e.w);
            for (z = 0; z < G->V; z++)
               if (id[z] == idv)  id[z] = idw;
         }
      }
   }
}
```

No fim da primeira fase (ou seja, na 22ª linha do código),

- `AA` é o número de arcos externos e
- para cada vértice `i` que é o representante de uma componente da floresta F, o arco `bvka[i]` é mínimo dentre os que estão na franja da componente.

A figura abaixo tenta ilustrar a primeira fase de uma iteração. Estamos supondo que `v-w` é uma aresta de custo `55`, que `v` e `w` estão nas componentes representadas pelos vértices `i` e `j` respectivamente, que a componente representada por `i` escolheu o arco `v-w`, e que a componente representada por `j` escolheu o arco `w-v`. O símbolo "`*`" representa `maxCST`.

u	0	1	i	j	v		w	V-1
bvka[u].v			v	w	v		w	
bvka[u].w			w	v	v		w	
bvka[u].cst			55	55	*		*	

Exemplo

Aplique a função `bruteforceBoruvka` ao grafo definido pela lista de arestas abaixo.

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

Veja o estado do vetor `id` no início de sucessivas iterações:

```
0 1 2 3 4 5 6 7
2 7 2 5 5 5 7 7
7 7 7 7 7 7 7 7
```

Veja as arestas impressas no fim de sucessivas iterações,

```
0-2 1-7 3-5 4-3 6-7
                   0-7 4-7
```

Exercícios

6. Qual o consumo de tempo da função `bruteforceBoruvka`?

Implementação eficiente do algoritmo de Boruvka

Uma implementação eficiente do algoritmo de Boruvka usa as idéias da função `bruteforceBoruvka` combinadas com uma estrutura *union-find*. No código abaixo, a função `DIGRAPHarcs` armazena os arcos do grafo `G` no vetor `a[0..A-1]`, em ordem arbitrária.

```
/* Vamos supor que o grafo tem no máximo maxV vértices e no máximo maxE arestas. */

#define maxV 10000
#define maxE 100000000

/* A função abaixo recebe um grafo conexo G com custos nas arestas e calcula uma MST de G.  A
função armazena as arestas das MSTs no vetor mst[0..k-1] e devolve k. */

/* A função supõe que o custo de cada aresta é estritamente menor que maxCST.  O código é uma
versão corrigida do programa 20.6, p.254, de Sedgewick. */

int GRAPHmstB(Graph G, Edge mst[]) {
   Arc bvka[maxV], a[2*maxE];
   Vertex i, j, u, v, w;
   int A = G->A, k = 0;
   DIGRAPHarcs(a, G);
   UFinit(G->V);
   while (1) {
      int h, AA;
      for (u = 0; u < G->V; u++)
```

```
         bvka[u] = ARC(u, u, maxCST);
      for (AA = 0, h = 0; h < A; h++) {
         i = find(a[h].v); j = find(a[h].w);
         if (i != j) {
            a[AA++] = a[h];
            if (bvka[i].cst > a[h].cst)
               bvka[i] = a[h];
         }
      }
      A = AA;
      if (A == 0) break;
      for (u = 0; u < G->V; u++) {
         Arc e = bvka[u];
         v = e.v; w = e.w;
         if (v != w && find(v) != find(w)) {
            UFunion(v, w);
            mst[k++] = e;
         }
      }
   }
   return k;
}
```

Antes da 11ª linha do código, `a[0..A-1]` contém todos os arcos externos (em relação à floresta geradora `mst[0..k-1]`) e possivelmente alguns arcos não-externos. No fim da primeira fase (20ª linha do código), o vetor `bvka[0..V-1]` contém um conjunto de Boruvka: para cada vértice `u` que é representante de uma componente, `bvka[u]` é um arco mínimo da franja da componente.

No início de cada iteração, o conjunto de arestas `mst[0..k-1]` define uma floresta geradora F de `G` e as funções `find` e `UFunion` têm o seguinte papel:

- `find(v)` é o representante da componente que contém `v`.
- `UFunion(v, w)` promove a união das componentes de F que contêm `v` e `w` respectivamente.

Para dar uma implementação eficiente às funções `find` e `UFunion`, o vetor `id` é definido de maneira um pouco diferente da que usamos na seção anterior. Cada componente da floresta é representada por uma *union-find tree* (que não é um subdigrafo de nosso grafo `G`). Na *union-find tree*, cada vértice `i` tem um pai `id[i]`. Se `i` é a raiz de uma *union-find tree* então `id[i] = i`. (Veja a seção 1.3, p.11-19, do volume 1 do livro de Sedgewick.) Se `i` é raiz de uma union-find tree, então `sz[i]` é o número de nós na union-find tree.

A implementação da estrutura *union-find tree* exibida abaixo é mais eficiente, no sentido amortizado, que a que usamos na implementação do algoritmo de Kruskal: ela inclui o "truque" *path compression* no código da função `find`.

```
/* (O código abaixo é uma versão adaptada dos programas 1.3, 1.4 e 4.8 (p.17, 19 e 152
respectivamente) do volume 1 do livro de Sedgewick). */

static Vertex id[maxV];
static int sz[maxV];

void UFinit(int N) {
   Vertex i;
   for (i = 0; i < N; i++) {
      id[i] = i;
      sz[i] = 1;
   }
}

/* A função find devolve a raiz da union-find tree que contém o vértice x. */

Vertex find(Vertex x) {
   Vertex i;
   for (i = x; i != id[i]; i = id[i])
      id[i] = id[id[i]];  /* path compression */
   return i;
}

/* A função UFunion faz a união das union-find trees que contêm os vértices v e w. */

void UFunion(Vertex v, Vertex w) {
   Vertex i = find(v), j = find(w);
   if (i == j) return;
   if (sz[i] < sz[j]) {
      id[i] = j;
      sz[j] += sz[i];
   }
   else {
      id[j] = i;
      sz[i] += sz[j];
   }
```

```
}
```

(O código de GRAPHmstB pode parecer um pouco assustador porque depende de várias funções auxiliares. É um bom exercício escrever uma versão "compacta" da função GRAPHmstB que incorpore, tanto quanto razoável, o código das funções de manipulação da estrutura *union-find*.)

Desempenho

Graças ao *path compression*, o consumo *amortizado* de tempo de cada execução da função `find` é proporcional a $\log^*(E)$, sendo E o número de arestas do grafo.

Cada iteração reduz o número de componentes da floresta F pelo menos à metade. Assim, o número de iterações não passa de $\log(V)$. Com isso, o consumo de tempo da implementação GRAPHmstB do algoritmo de Boruvka é proporcional a

$$E \log(V) \log^*(E)$$

no pior caso. Na prática, isto é essencialmente o mesmo que $E \log(V)$. Mesmo essa estimativa é muito pessimista: na prática, o consumo de tempo é, em geral, proporcional a E.

Exercícios

7. Considere o grafo cujos vértices são pontos no plano:

   ```
      vértices     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 `v-w` é igual ao comprimento do segmento de reta que liga os pontos `v` e `w` no plano. Aplique o algoritmo de Boruvka a esse grafo. Exiba uma figura do grafo e da floresta no início de cada iteração.

8. O teste "`find(v) != find(w)`" na parte final do código da função GRAPHmstB é realmente necessário? (Dica: Considere arestas de mesmo custo.)

9. Escreva uma implementação do algoritmo de Boruvka que começa por colocar as arestas do grafo em ordem crescente de custos.

URL of this site: http://www.ime.usp.br/~pf/algoritmos_para_grafos/
Last modified: Mon Mar 8 09:47:48 BRT 2010
Paulo Feofiloff
IME-USP