Árvore de cobertura mínima

Algoritmo de Prim

$G = (V, E)$ – grafo pesado não orientado conexo

MST-PRIM(G, w, s)

```
for each vertex u in G.V do
    u.key <- INFINITY                 // cost of adding u
    u.p <- NIL
s.key <- 0
Q <- G.V                  // priority queue, with key u.key
while Q != EMPTY do
    u <- EXTRACT-MIN(Q)
    for each vertex v in G.adj[u] do
        if v in Q and w(u,v) < v.key then
            v.p <- u
            v.key <- w(u,v)           // decrease key in Q
```

Análise da complexidade do algoritmo de Prim

Grafo representado através de listas de adjacências

Linhas

1–3 Ciclo executado $|V|$ vezes

5 Construção da fila com prioridade (*heap*): $O(V)$

6–11 Ciclo executado $|V|$ vezes

7 Remoção do menor elemento da fila: $O(\log V)$

8–11 Ciclo executado $2|E|$ vezes **no total**

11 Alteração da prioridade de um elemento na fila: $O(\log V)$

Operação executada, no pior caso, $|E|$ vezes

Complexidade temporal do algoritmo

$$O(V + V + V \log V + 2E + E \log V) = O(E \log V)$$

Restantes operações com complexidade temporal constante

Árvore de cobertura mínima

Algoritmo de Kruskal

$G = (V, E)$ – grafo pesado não orientado conexo

MST-KRUSKAL(G, w)

```
n <- |G.V|
A <- EMPTY                          // set with the MST edges
P <- MAKE-SETS(G.V)                 // partition of G.V
Q <- G.E              // priority queue, key is weight w(u,v)
e <- 0
while e < n - 1 do
    (u,v) <- EXTRACT-MIN(Q)
    if FIND-SET(P, u) != FIND-SET(P, v) then
        A <- A + {(u,v)}
        UNION(P, u, v)
        e <- e + 1
return A
```

Análise da complexidade do algoritmo de Kruskal (1)

Linhas

3 Construção da partição

MAKE-SETS

4 Construção da fila com prioridade (*heap*)

$$O(E)$$

6–11 Ciclo executado entre $|V| - 1$ e $|E|$ vezes

7 Remoção do menor elemento da fila (*heap*)

$$O(\log E) = O(\log V)$$

$(|E| < |V|^2$ e $\log |E| < log|V|^2 = 2 \log |V| = O(\log V))$

8 $2 \times$ FIND-SET

10 Executada $|V| - 1$ vezes

UNION

Restantes operações com complexidade temporal constante

Análise da complexidade do algoritmo de Kruskal (2)

Juntando tudo, obtém-se

$$\text{MAKE-SETS} + O(E) + |E| \times O(\log V) +$$
$$|E| \times 2 \times \text{FIND-SET} + (|V| - 1) \times \text{UNION}$$

ou

$$O(E) + |E| \times O(\log V) + f(V, E)$$

com

$$f(V, E) = \text{MAKE-SETS} + 2 \times |E| \times \text{FIND-SET} + (|V| - 1) \times \text{UNION}$$

Partição

Conjuntos disjuntos (*Disjoint sets*)

Abstracção da implementação de conjuntos disjuntos com os elementos do conjunto $\{1, 2, \ldots, n\}$

Operações suportadas

MAKE-SETS(n)
: Cria conjuntos singulares com os elementos $\{1, 2, \ldots, n\}$

FIND-SET(i)
: Devolve o representante do conjunto que contém o elemento i

UNION(i, j)
: Reúne os conjuntos a que pertencem os elementos i e j

Também é conhecido como *Union-Find*

Partição

Implementação em vector

MAKE-SETS(n)

```
let P[1..n] be a new array
for i <- 1 to n do
  P[i] <- -1   // i is the representative for set {i}
return P
```

FIND-SET(P, i)

```
while P[i] > 0 do
     i <- P[i]
return i
```

UNION(P, i, j)

```
P[FIND-SET(P,j)] <- FIND-SET(P, i)
```

Partição

Implementação em vector

Reunião por tamanho

Se $P[i] = -k$, o conjunto de que i é o representante contém k elementos

UNION-BY-SIZE(P, i, j)

```
ri <- FIND-SET(P, i)    // get i's set representative
rj <- FIND-SET(P, j)    // get j's set representative
if (P[rj] < P[ri])      // j's set is larger than i's
    P[rj] <- P[rj] + P[ri]
    P[ri] <- rj
else        // i's is larger or both have the same size
    P[ri] <- P[ri] + P[rj]
    P[rj] <- ri
```

Partição

Implementação em vector

Reunião por altura

Se $P[i] = -h$, a árvore do conjunto de que i é o representante tem altura h

UNION-BY-RANK(P, i, j)

```
ri <- FIND-SET(P, i)
rj <- FIND-SET(P, j)
if (P[rj] < P[ri])     // j's set tree taller than i's
    P[ri] <- rj
else
    if (P[rj] == P[ri])     // both heights are equal
        P[ri] <- P[ri] - 1
    P[rj] <- ri
```

Partição

Implementação em vector

Compressão de caminho

FIND-SET-WITH-PATH-COMPRESSION(P, i)

```
if P[i] < 0 then
    return i
P[i] <- FIND-SET-WITH-PATH-COMPRESSION(P, P[i])
return P[i]
```

Análise da complexidade do algoritmo de Kruskal (3)

$$O(E) + |E| \times O(\log V) + f(V, E)$$

$$f(V, E) = \text{MAKE-SETS} + 2 \times |E| \times \text{FIND-SET} + (|V| - 1) \times \text{UNION}$$

Implementação da Partição	Básica	União por tam./altura	+ Compressão de caminho
MAKE-SETS	$O(V)$	$O(V)$	$O((V+E)\,\alpha(V))$ [Tarjan 1975]
$2 \times \lvert E \rvert \times$ FIND-SET	$O(EV)$	$O(E \log V)$	
$(\lvert V \rvert - 1) \times$ UNION	$O(V^2)$	$O(V \log V)$	
$f(V, E)$	$O(EV)$	$O(E \log V)$	$O(E\alpha(V))$
Algoritmo de Kruskal	$O(EV)$	$O(E \log V)$	$O(E \log V)$

$\alpha(n) \leq 4$ para $n < 10^{80}$

Análise da complexidade do algoritmo de Kruskal (4)

$$\alpha(n) = \min\{k \mid A_k(1) \geq n\}$$

onde

$$A_k(j) = \begin{cases} j + 1 & \text{se } k = 0 \\ A_{k-1}^{(j+1)}(j) & \text{se } k \geq 1 \end{cases}$$

$$\begin{aligned} A_0(1) &= 2 \\ A_1(1) &= A_0(A_0(1)) = 3 \\ A_2(1) &= A_1(A_1(1)) = 7 \\ A_3(1) &= 2047 \\ A_4(1) &\gg 2^{2048} \gg 10^{80} \end{aligned}$$

Iteração de uma função

$A_{k-1}^{(0)}(j) = j$ e $A_{k-1}^{(i)}(j) = A_{k-1}(A_{k-1}^{(i-1)}(j))$, para $i \geq 1$

Caminho mais curto

Num grafo pesado, com pesos w, o peso do caminho

$$p = v_0\, v_1\, \ldots\, v_k$$

é a soma dos pesos dos arcos que o integram

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

O caminho p é mais curto que o caminho p' se o peso de p é menor que o peso de p'

Cálculo dos caminhos mais curtos

Algoritmos

Cálculo dos caminhos mais curtos num grafo orientado acíclico (DAG), com pesos possivelmente negativos

Algoritmo de Dijkstra, para grafos sem pesos negativos

Algoritmo de Bellman-Ford, para quaisquer grafos pesados

Estes algoritmos calculam os caminhos mais curtos de um nó s para os restantes nós do grafo (*single-source shortest paths*)

Caminhos mais curtos

Subrotinas comuns aos diversos algoritmos

O peso do caminho mais curto de s a qualquer outro nó é inicializado com ∞

INITIALIZE-SINGLE-SOURCE(G, s)

```
for each vertex v in G.V do
    v.d <- INFINITY   // peso do caminho mais curto de s a v
    v.p <- NIL        // predecessor de v nesse caminho
s.d <- 0
```

Se o caminho de s a v, que passa por u e pelo arco (u, v), tem menor peso do que o mais curto anteriormente encontrado, é esse o mais curto encontrado até ao momento

RELAX(u, v, w)

```
if u.d + w(u,v) < v.d then
    v.d <- u.d + w(u,v)
    v.p <- u
```

Caminhos mais curtos a partir de um vértice

DAGs

$G = (V, E)$ – DAG pesado (pode ter pesos negativos)

DAG-SHORTEST-PATHS(G, w, s)

```
topologically sort the vertices of G
INITIALIZE-SINGLE-SOURCE(G, s)
for each vertex u, taken in topologically sorted
                                                  order do
    for each vertex v in G.adj[u] do
        RELAX(u, v, w)
```