Componentes fortemente conexas

Strongly connected components

G – grafo orientado

SCC(G)

1. Executar DFS(G) para calcular o instante `u.f` em que termina o processamento de cada vértice `u`
2. Calcular G^T
3. Executar DFS(G^T), processando os vértices por ordem decrescente de `u.f` (calculado em 1), no ciclo principal de DFS (linha 5)
4. Devolver os vértices de cada árvore da floresta da pesquisa em profundidade (construída em 3) como uma componente fortemente conexa distinta

Árvore de cobertura mínima (1)

Minimum(-weight) spanning tree (MST)

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

Uma árvore é um grafo não orientado conexo acíclico

(Retirando qualquer arco de uma árvore, obtém-se um grafo não conexo)

Uma árvore de cobertura de G é um subgrafo $G' = (V', E')$ de G tal que

- $V' = V$
- $E' \subseteq E$
- G' é uma árvore

Árvore de cobertura mínima (2)

Minimum(-weight) spanning tree (MST)

Seja o peso de um grafo $w(G)$ a soma dos pesos dos arcos de G

$$w(G) = \sum_{e \in E} w(e)$$

Uma árvore de cobertura mínima de G é uma árvore de cobertura G' de peso mínimo:

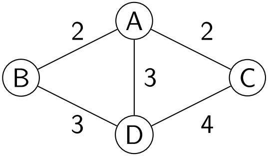
Para qualquer árvore de cobertura G'' de G tem-se

$$w(G') \leq w(G'')$$

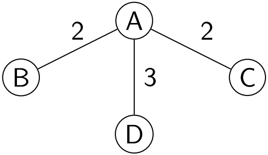
Árvore de cobertura mínima (3)

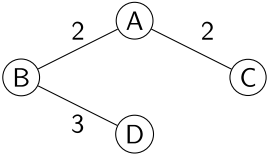
Minimum(-weight) spanning tree (MST)

Exemplo



Árvores de cobertura mínimas





O peso das árvores de cobertura mínimas deste grafo é $2+2+3 = 7$

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       // custo de juntar u à MST
    u.p <- NIL              // nó a que u é ligado
s.key <- 0
Q <- G.V                    // fila com prioridade, por
                            // mínimos, chave é 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        // arco (u,v) é candidato
            v.key <- w(u,v) // pode alterar Q!
```

Filas com prioridade (1)

Uma fila com prioridade é uma fila em que a cada elemento está associado um valor (`key`), que determina a sua prioridade

Numa fila organizada por mínimos, a prioridade é maior quando o valor é menor

Numa fila organizada por máximos, a prioridade é maior quando o valor é maior

A operação DEQUEUE retira da fila um elemento com maior prioridade

Para tornar explícita a disciplina da fila, no pseudo-código, a operação DEQUEUE é denotada por EXTRACT-MIN, para filas organizadas por mínimos, e por EXTRACT-MAX, para filas organizadas por máximos

Filas com prioridade (2)

Implementação, para um conjunto limitado de elementos

Vector ou lista (duplamente) ligada

- Inserção de elemento com complexidade temporal constante
- Remoção de elemento com complexidade temporal linear no número de elementos na fila
- Ou vice-versa, se a fila for mantida ordenada por prioridade

Heap binário

- Ambas as operações com complexidade temporal logarítmica no número de elementos na fila
- Criação, a partir de um conjunto de elementos, com complexidade temporal linear no número de elementos

Filas com prioridade com actualização da prioridade (1)

Por vezes, como acontece no algoritmo de Prim, é necessário aumentar a prioridade de um elemento presente na fila

Operação denotada por DECREASE-KEY, numa fila organizada por mínimos, e por INCREASE-KEY, numa fila organizada por máximos

Nas implementações em vector ou em lista, essa operação tem complexidade temporal constante (ou linear, se a fila for mantida ordenada)

Heap binário

Na implementação normal com um *heap* binário, a operação tem complexidade temporal linear no número de elementos na fila, devido a ser preciso localizar o elemento

Se, além do *heap*, a implementação mantiver um mapa, com a posição de cada elemento, a operação pode ser realizada com complexidade temporal logarítmica no número de elementos na fila

Filas com prioridade com actualização da prioridade (2)

Poor man's approach

Uma alternativa à actualização da prioridade de um elemento presente na fila é uma nova inserção do elemento, com o novo valor associado

- Podem existir várias cópias do mesmo elemento na fila, com diferentes prioridades
- É necessário associar uma *flag* a cada elemento, que diga se ele já saiu da fila, e foi processado, ou não

A complexidade temporal das várias operações mantém-se

Aumenta o número de elementos que a fila pode conter

Análise da complexidade do algoritmo de Prim (1)

Operação da fila com prioridade

Fila implementada através de um *heap* binário

- Construção de uma fila com n elementos: $\Theta(n)$
- Ver se está vazia: $\Theta(1)$
- Remoção do elemento mínimo: $O(\log n)$
- Determinar se contém um dado elemento: $O(n)$

  Associando uma *flag* a cada vértice, pode-se reduzir a complexidade temporal desta operação para $\Theta(1)$
- Alterar o valor associado a um elemento: $O(n + \log n)$

  Mantendo, para cada vértice, o índice da posição em que se encontra, pode-se reduzir a complexidade temporal desta operação para $O(\log n)$

A seguir, considera-se uma implementação em que cada operação é realizada da maneira mais eficiente

Análise da complexidade do algoritmo de Prim (2)

Grafo representado através de listas de adjacências

Linhas

1–3 Ciclo executado $|V|$ vezes

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

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                     // conjunto dos arcos da MST
P <- MAKE-SETS(G.V)            // partição de G.V, floresta
Q <- G.E                       // fila com prioridade, por
                               // mínimos, chave é w(u,v)
e <- 0                         // número de arcos na MST
while e < n - 1 do
    (u,v) <- EXTRACT-MIN(Q)
    if FIND-SET(P, u) != FIND-SET(P, v) then
        A <- A + {(u,v)}       // novo arco da MST
        UNION(P, u, v)
        e <- e + 1
return A
```