Minimum Spanning Tree
Algoritmos de Prim e Kruskal

Fernando Lobo

Algoritmos e Estrutura de Dados II

Algoritmo de Prim

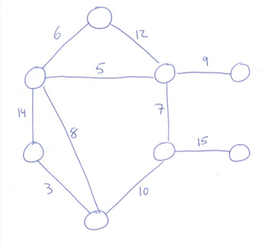
- Inicialmente $A = \emptyset$
- Mantemos $V - A$ numa fila com prioridade Q.
- A chave de cada nó na fila indica o custo mínimo de juntar esse nó a um qualquer nó de A.
- Quando o algoritmo termina, a fila Q está vazia. A MST A é dada por:

$$A = \{(v, v.\pi) : v \in V - \{s\}\}$$

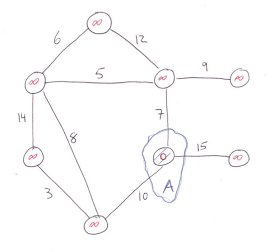
Pseudocódigo

MST-PRIM(G, w, s)
    for each $u \in G.V$
        $u.key = \infty$
        $u.\pi = \text{NIL}$
    $s.key = 0$
    $Q = G.V$
    while $Q \neq \emptyset$
        $u = \text{EXTRACT-MIN}(Q)$
        for each $v \in G.Adj[u]$
            if $v \in Q$ and $w(u, v) < v.key$
                $v.\pi = u$
                $v.key = w(u, v)$

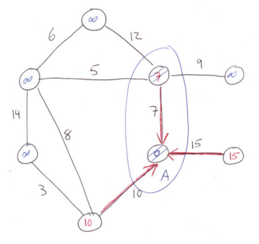
Exemplo



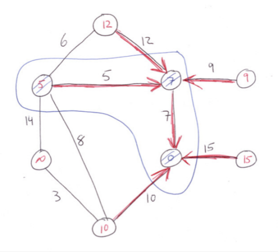
Inicialização



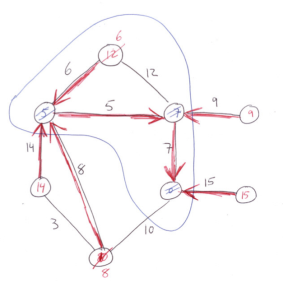
1ª iteração do ciclo **while**



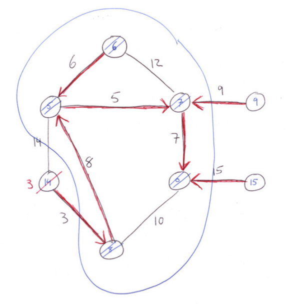
2ª iteração do ciclo **while**



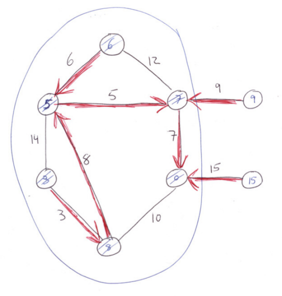
3ª iteração do ciclo **while**



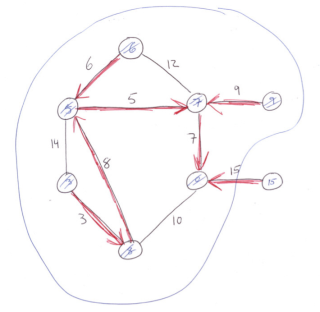
4ª iteração do ciclo **while**



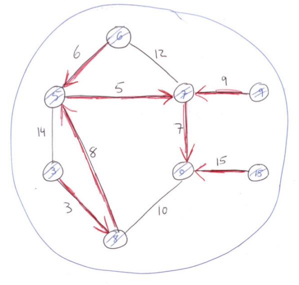
5ª iteração do ciclo **while**



6ª iteração do ciclo **while**



7ª iteração do ciclo **while**



Complexidade do Algoritmo de Prim

- Depende da implementação da fila com prioridade.
- Se implementarmos com um heap binário:
  - Inicialização → BUILD-HEAP → $O(V)$
  - Ciclo **while** é executado $|V|$ vezes.
  - $|V|$ EXTRACT-MINS → $O(V \lg V)$
  - No máximo, $|E|$ DECREASE-KEYS → $O(E \lg V)$
  - Total $= O(V \lg V + E \lg V) = O(E \lg V)$
- A verificação **if** $v \in Q$ pode ser obtida em tempo constante se mantivermos um bit vector com indicação dos nós que estão em Q.

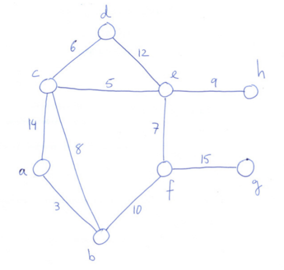
Complexidade do Algoritmo de Prim

- É possível obter uma complexidade melhor se usarmos heaps de Fibonacci para implementar a fila.
  (Não demos estes heaps, mas se tiverem curiosidade poder ler o capítulo do vosso livro que fala sobre eles.)
  - Consegue-se fazer com que $|E|$ DECREASE-KEYS tenha complexidade $O(E)$, amortizado.
    $\implies T_{Prim} = O(V \lg V + E)$
  - É um ganho substancial para grafos esparsos.

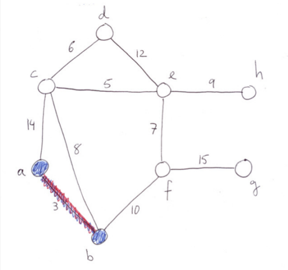
Algoritmo de Kruskal

- Inicialmente $A = \emptyset$. No final A será uma MST.
- Ordenamos os arcos do grafo por ordem crescente de peso.
- Percorre-se a lista ordenada dos arcos, um a um, e adicionamos o arco a A desde que não produza ciclo.
- Ao contrário do algoritmo de Prim, o algoritmo de Kruskal mantém ums floresta. Inicialmente a floresta tem $|V|$ árvores, uma para cada nó. No final, a floresta só tem uma árvore que é uma Minimum Spanning Tree.

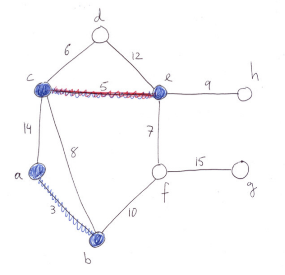
Exemplo: Inicialização



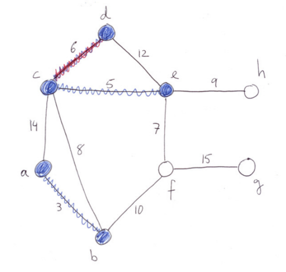
Iteração 1



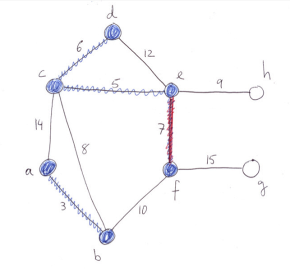
Iteração 2



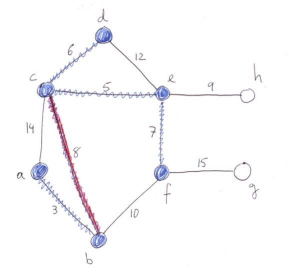
Iteração 3



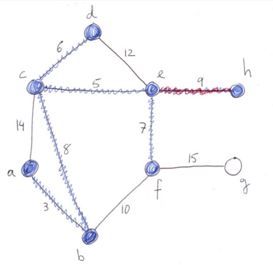
Iteração 4



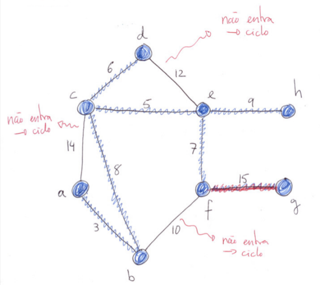
Iteração 5



Iteração 6

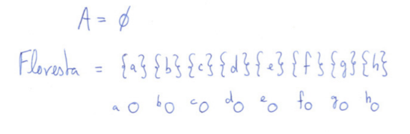


Iteração 7, 8, 9, 10



Evolução da floresta de árvores

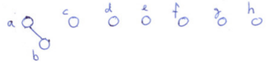
Inicialização



Iteração 1

$A = \{(a,b)\}$

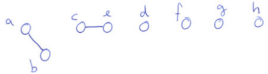
Floresta $= \{a,b\}\ \{c\}\ \{d\}\ \{e\}\ \{f\}\ \{g\}\ \{h\}$



Iteração 2

$A = \{(a,b), (c,e)\}$

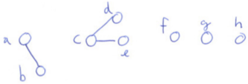
Floresta $= \{a,b\}\ \{c,e\}\ \{d\}\ \{f\}\ \{g\}\ \{h\}$



Iteração 3

$A = \{ (a,b), (c,e), (c,d) \}$

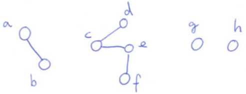
$Floresta = \{a,b\} \{c,d,e\} \{f\} \{g\} \{h\}$



Iteração 4

$A = \{ (a,b), (c,e), (c,d), (e,f) \}$

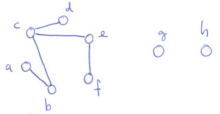
$Floresta = \{a,b\} \{c,d,e,f\} \{g\} \{h\}$



Iteração 5

$A = \{(a,b), (c,e), (c,d), (e,f), (c,b)\}$

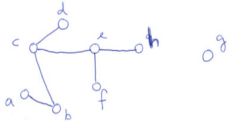
Floresta = $\{a, b, c, d, e, f\}$ $\{g\}$ $\{h\}$



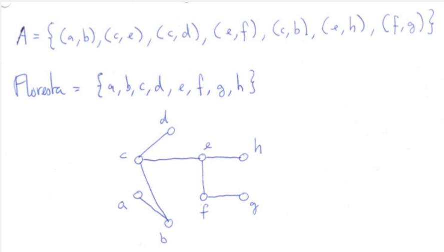
Iteração 6

$A = \{(a,b), (c,e), (c,d), (e,f), (c,b), (e,h)\}$

Floresta = $\{a, b, c, d, e, f, h\}$ $\{g\}$



Iteração 7, 8, 9, 10



Implementação do Algoritmo de Kruskal

- Necessitamos de uma estrutura de dados que permita manter dinamicamente uma floresta de árvores.
- Inicialmente temos $|V|$ árvores.
- A cada passo do algoritmo juntamos duas árvores e ficamos com menos uma árvore na floresta.
- Na realidade, não é necessário manter explicitamente as árvores.
  - Basta manter os nós que cada árvore contém.
  - Nota: As árvores são disjuntas. Um nó é elemento de uma e uma só árvore.

Implementação do Algoritmo de Kruskal

- O que necessitamos é de uma estrutura de dados para manter uma colecção S de conjuntos disjuntos.
- Essa estrutura de dados tem o nome UNION-FIND (vamos estudá-la em detalhe na próxima aula).
- Suporta estas operações: $\textsc{Make-Set}(x)$, $\textsc{Find-Set}(x)$ e $\textsc{Union}(x, y)$
  - $\textsc{Make-Set}(x)$: cria $\{x\}$ e acrescenta-o a S.
  - $\textsc{Find-Set}(x)$: retorna um identificador do conjunto que contém x.
  - $\textsc{Union}(x, y)$: faz a união do conjunto que contém x com o conjunto que contém y e acrescenta-o a S, eliminado os conjuntos originais que continham x e y.

Pseudocódigo

```
MST-KRUSKAL(G, w)
    A = ∅
    for each v ∈ G.V
        MAKE-SET(v)
    sort G.E in ascending order of weight w
    for each (u, v) ∈ G.E, taken in ascending order of weight
        if FIND-SET(u) ≠ FIND-SET(v)
            A = A ∪ {(u, v)}
            UNION(u, v)
    return A
```

Complexidade do Algoritmo de Kruskal

- Primeiro ciclo **for**: O(V) MAKE-SETS
- Ordenar E: O($E \lg E$)
- Segundo ciclo **for**: O(E) FIND-SETS e UNIONS
  Na realidade só fazemos O(V) UNIONS. Porquê?
- A complexidade vai depender da dorma como se implementa a estrutura de dados UNON-FIND.
- Mas é possível obter uma complexidade de O($E \lg V$).
  Veremos como na próxima aula.