CTC-12

Projeto e Análise de Algoritmos

Carlos Alberto Alonso Sanches

CTC-12

9) Algoritmos em grafos

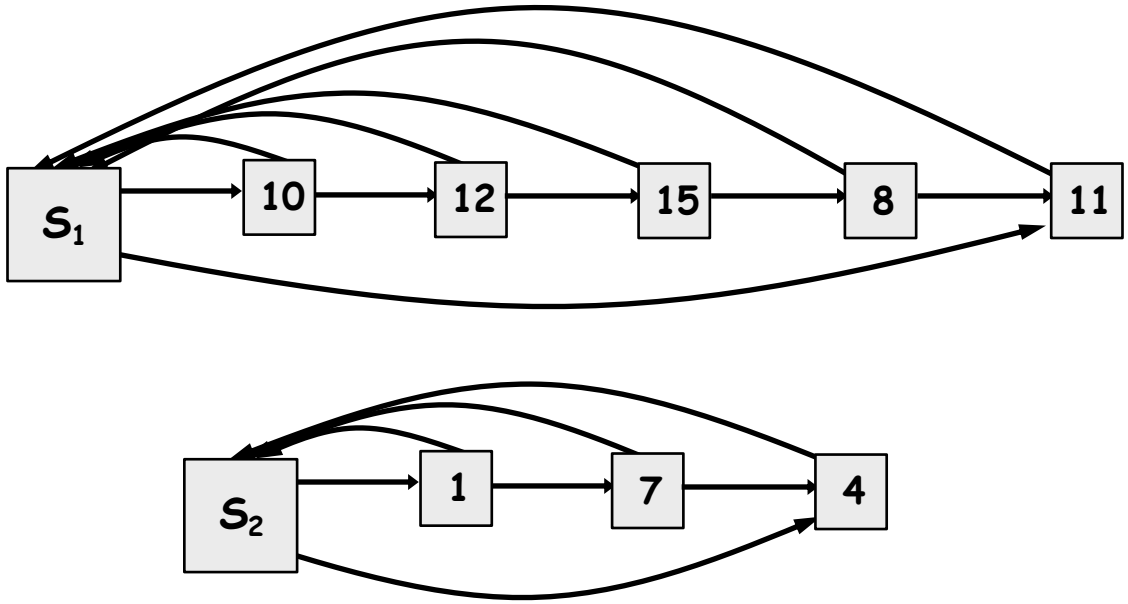
Union-find, Kruskal, Prim

Conjuntos disjuntos dinâmicos

- Em diversas aplicações, é necessário representar uma partição dinâmica de um conjunto S com n elementos.
- Concretamente, esta partição corresponde a uma coleção de subconjuntos S_i disjuntos tais que:
  - $\forall S_i \subseteq S$
  - $US_i = S$
  - $S_i \cap S_j = \phi$
- Deseja-se uma estrutura de dados com três operações:
  - MakeSet(x): cria um subconjunto unitário com o elemento x
  - Union(x,y): une os subconjuntos que contêm os elementos x e y
  - Find(x): retorna o representante do subconjunto que contém x (importante: cada subconjunto possui um único representante)
- Alguns exemplos de aplicação: implementação eficiente de conjuntos, obtenção de árvores de espalhamento, algoritmos de *clustering*, alcance e conectividade, soluções de jogos e quebra-cabeças, etc.

Implementação com listas

- Uma possível implementação seria utilizar uma lista para cada subconjunto, onde o representante é o seu primeiro nó.
- Exemplo:



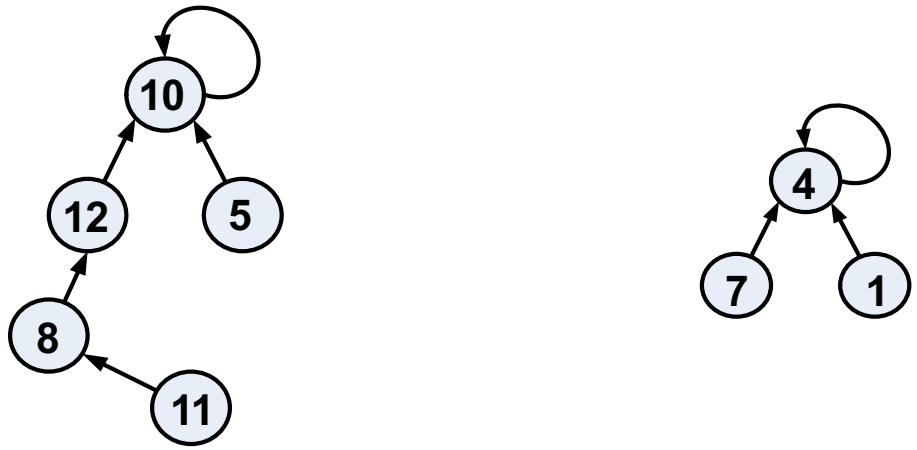
- Os ponteiros para o nó líder permitem que Find gaste tempo constante.
- Se o nó líder armazenar também o tamanho corrente do subconjunto, Union gastará tempo proporcional ao tamanho do menor subconjunto.

Análise amortizada de tempo

- Se uma sequência de q operações gasta tempo $T(q)$, dizemos que o custo amortizado por operação é $T(q)/q$.
- Vamos analisar a eficiência da implementação com listas através do custo amortizado de q operações MakeSet, Union ou Find:
  - Devem incluir necessariamente n operações MakeSet, ou seja, $q \geq n$.
  - Cada MakeSet ou Find gasta tempo $\Theta(1)$.
  - O número máximo de operações Union é $n-1$. Em cada Union, o tamanho do menor subconjunto é ao menos dobrado: desse modo, qualquer elemento terá seus ponteiros atualizados no máximo $\lg n$ vezes. Portanto, o tempo total das operações Union é $O(n.\log n)$.
- Nesta implementação com listas, $q \geq n$ operações MakeSet, Union ou Find gastam tempo $O(q + n.\log n)$. Portanto, seu custo amortizado por operação é $O((n.\log n)/q)$.

Implementação com árvores

- Outra possível implementação é através de árvores, onde a raiz é o representante do subconjunto.
- Exemplo:



- A operação Find exige um percurso até a raiz.
- Na operação Union, a raiz da árvore mais baixa apontará para a raiz da mais alta, que será a representante do novo subconjunto.

Pseudocódigos com uso de rank

Em cada nó, o campo **rank** armazena a altura da sua sub-árvore

```
MakeSet(x) {
    x.pai = x;
    x.rank = 0;
}
```

```
elemento Find(x) {
    while (x != x.pai) {
        x = x.pai;
    }
    return x;
}
```

```
Union(x,y) {
    x1 = Find(x);
    y1 = Find(y);
    if (x1 != y1)
        if (x1.rank > y1.rank)
                y1.pai = x1;
        else
                if (x1.rank < y1.rank)
                        x1.pai = y1;
                else {
                        y1.pai = x1;
                        x1.rank++;
                }
}
```

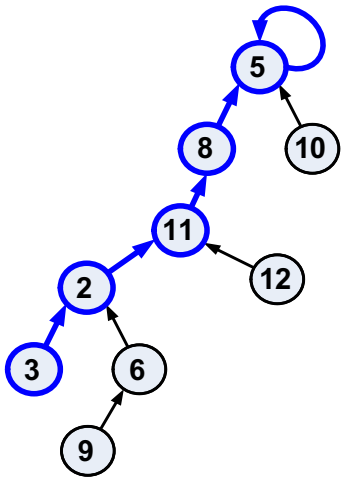
Propriedades desta implementação

1) x.rank < (x.pai).rank para todo nó x que não é raiz.
2) Quando uma raiz x passa a ter um pai, então x.rank < (x.pai).rank.
3) x.rank não será mais alterado após x deixar de ser raiz.
4) Uma raiz com rank r tem no mínimo 2^r nós na sua árvore.
   - Prova por indução em r:
     - Verdadeiro para r = 0: $1 \geq 2^0$.
     - Uma raiz com rank r+1 é criada pela união de duas árvores com rank r: portanto, o total de nós da sua árvore será no mínimo $2^r + 2^r = 2^{r+1}$.
5) O valor máximo de rank será $\lfloor \lg n \rfloor$, onde n é número total de nós.
   - Propriedade (4): $n \geq 2^{r_{máx}} \Leftrightarrow r_{máx} \leq \lfloor \lg n \rfloor$
   - Bastam $\Theta(n.\log \log n)$ bits para armazenar os ranks de todos os n nós.
6) Há no máximo $n/2^r$ nós com rank r, para qualquer $r \geq 0$.
   - Se for raiz: segue pela propriedade (4).
   - Se for nó interno: tinha essa propriedade quando era raiz; depois que passou a ter um pai, seus descendentes permanecem e seu rank não muda devido à propriedade (3).

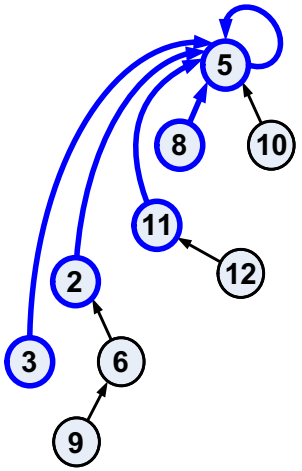
Compressão de caminhos

- Nesta estrutura baseada em árvores, é possível adotar uma melhoria: na execução de cada operação Find, todos os nós percorridos passam a apontar para a sua correspondente raiz.
- Exemplo:

Antes de Find(3)



Após Find(3)



Novo pseudocódigo

```
elemento Find(x) {
    if (x != x.pai)
        x.pai = Find(x.pai);
    return x.pai;
}
```

- Rank passa a ser um limite superior para a altura da sub-árvore.
- Caso contrário, Find precisaria atualizar esse campo em cada nó percorrido. Veremos que não vale a pena fazer isso...

As propriedades continuam válidas

- A compressão de caminhos não altera a implementação das operações MakeSet e Union.
- As operações Find afetam somente nós internos, enquanto as operações Union afetam somente as raízes.
- A compressão de caminhos:
  - não cria novas raízes;
  - não altera os ranks de cada nó;
  - não move nós de uma árvore de raiz x para outra de raiz y.
- Todas as propriedades anteriores continuam válidas:
  - Propriedades (1) e (2): quando um nó muda de pai, seu novo pai será um ancestral da mesma árvore, que possui um rank ainda maior.
  - Propriedades (3), (4) e (5): trivial.
  - Propriedade (6): na contagem dos nós internos, bastaria supor que seus descendentes permanecessem como filhos, com o acréscimo de um ponteiro para a raiz da árvore.

Logaritmo binário iterado

$\log^* n = k$, onde k é o menor inteiro tal que $\lg^k n \leq 1$

$$\lg^k n = \underbrace{\lg (\lg (\lg (\ldots (\lg n) \ldots)))}_{k \text{ vezes}}$$

Definição formal:

$$\log^* n = \begin{cases} 0 & \text{se } n \leq 1 \\ 1 + \log^* (\lg n) & \text{se } n > 1 \end{cases}$$

n	$\log^* n$
1	0
2	1
[3, 4]	2
[5, 16]	3
[17, 65536]	4
[65537, 2^{65536}]	5

$2^{65536} \approx 10^{19728}$

Total de átomos no universo $< 10^{83}$

Para valores práticos de n, $\log^* n \leq 5$

Tempo das operações Find

- O tempo de cada operação Find é proporcional ao tamanho do percurso nas árvores através do ponteiro pai. Devido à propriedade (1), esse percurso sempre leva a nós com valores maiores de rank.
- Lembrando que o maior valor possível de rank é $\lfloor \lg n \rfloor$, vamos distribuir todos esses valores em até log* n intervalos: [1], [2], [3, 4], [5, 16], ..., $[r+1, 2^r=R]$, $[R+1, 2^R=S]$, $[S+1, \lfloor \lg n \rfloor]$.
- A quantidade de nós com rank no intervalo $[r+1, 2^r=R]$ é no máximo $n/2^r$:
  - Basta aplicar a propriedade (6) a cada valor deste intervalo, e observar que a primeira parcela é maior que o restante da somatória.
  - $n/2^{r+1} + n/2^{r+2} + \ldots + n/2^R \leq 2n/2^{r+1} = n/2^r$
- Quando um nó com rank no intervalo $[r+1, 2^r]$ deixar de ser raiz, ele receberá 2^r *créditos* para serem consumidos em operações Find.
  - Portanto, cada intervalo $[r+1, 2^r]$ recebe até $2^r.(n/2^r) = n$ *créditos*.
  - Como há até log* n intervalos, são distribuídos no máximo n.log* n *créditos*.

Três possíveis casos

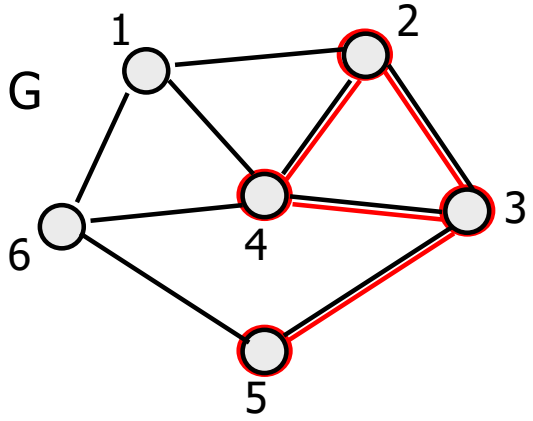
- Na operação Find(x), há 3 possíveis casos:
  - x.pai é uma raiz: gasta tempo constante.
  - (x.pai).rank está em algum intervalo posterior: isso pode ocorrer até log* n vezes, que é a quantidade de intervalos.
  - (x.pai).rank está no mesmo intervalo $[r+1, 2^r]$ de x.rank:
    - Suponha que o nó x pague um *crédito* por ponteiro percorrido.
    - Depois de gastar seus 2^r *créditos*, o nó x chegará necessariamente ao próximo intervalo. Em outras palavras, os *créditos* de x permitem que ele se torne filho de uma raiz de algum intervalo posterior.
    - Deste modo, os n.log* n *créditos* distribuídos garantem que todos os nós passem a ter como pai uma raiz de algum intervalo posterior.
- Portanto, depois que n.log* n ponteiros forem percorridos em operações Find, o tamanho de qualquer percurso nas árvores se tornará O(log* n).

Análise amortizada de tempo

- Sejam q ≥ n operações MakeSet, Union ou Find aplicadas sobre n elementos, utilizando árvores com compressão de caminhos.
- Como há n operações MakeSet e o tempo de uma operação Union é da ordem de duas operações Find, basta contabilizarmos o tempo das operações Find.
- Portanto, essas q operações MakeSet, Union ou Find aplicadas sobre n elementos gastam tempo O(q.log* n).
- Para valores práticos de n, este tempo é O(q), ou seja, o custo amortizado dessas q operações é praticamente constante.
- A demonstração apresentada se deve a Hopcroft e Ullman (1973). Em 1975, Tarjan provou que este custo é $O(q.\alpha(n))$, onde $\alpha(n)$ é o inverso da função de Ackermann: uma função que cresce ainda mais lentamente que log* n.

Subgrafos

- O grafo $G'=(V',E')$ é um subgrafo de $G=(V,E)$ se $V' \subseteq V$ e $E' \subseteq E$, e todas arestas de E' têm seus vértices em V'.
- Quando $V'=V$, G' é chamado de subgrafo gerador de G.
- Seja $X \subseteq V$ e $E(X)$ o subconjunto das arestas de E com ambos os vértices em X. Dizemos que $G(X)=(X,E(X))$ é o subgrafo de G induzido por X.
- Exemplo:



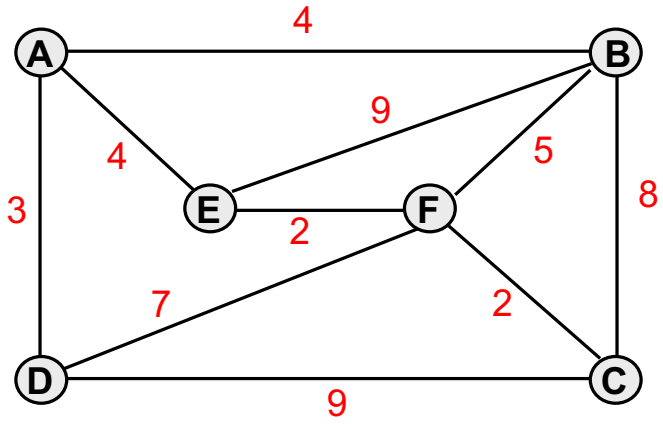
$X = \{2, 3, 4, 5\}$

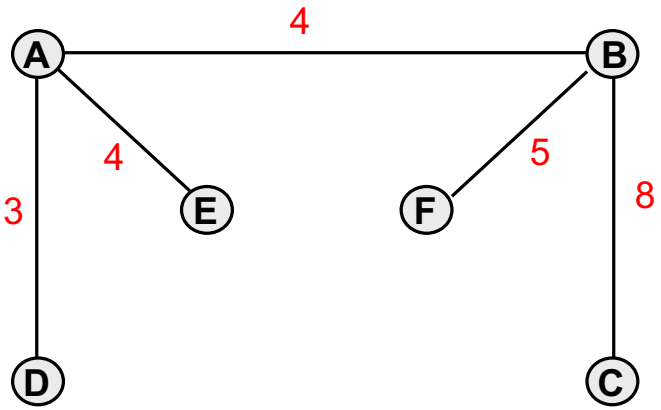
$G(X)$

Árvore geradora de custo mínimo (MST)

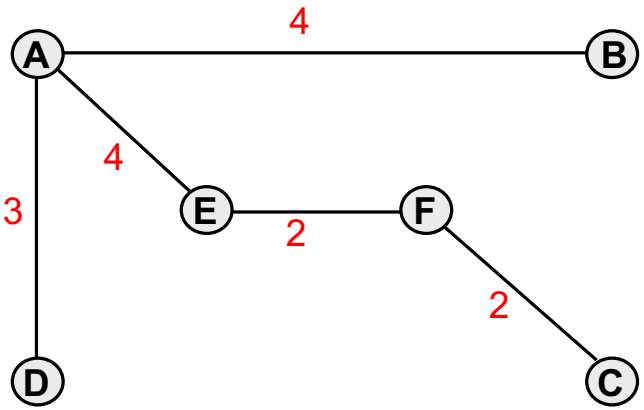
- Dado um grafo $G=(V,E)$, conexo e ponderado, com custo associado $c(e)$, $e \in E$, deseja-se encontrar um subgrafo T tal que:
  - seja gerador de G (isto é, possua todos os vértices);
  - seja acíclico e conexo (isto é, uma árvore);
  - tenha custo total $c(T) = \Sigma_{e \in E} c(e)$ que seja mínimo.
- T costuma ser chamado de *árvore de espalhamento de custo mínimo.*
- Em um grafo conexo, sabemos que $m \geq n-1$.
- Alguns exemplos de aplicação: projeto de redes e sistemas de transportes, determinação de rotas eficientes, análise da estrutura de redes sociais, representação de árvores genealógicas, etc.

Exemplo





Árvore geradora
com custo 24

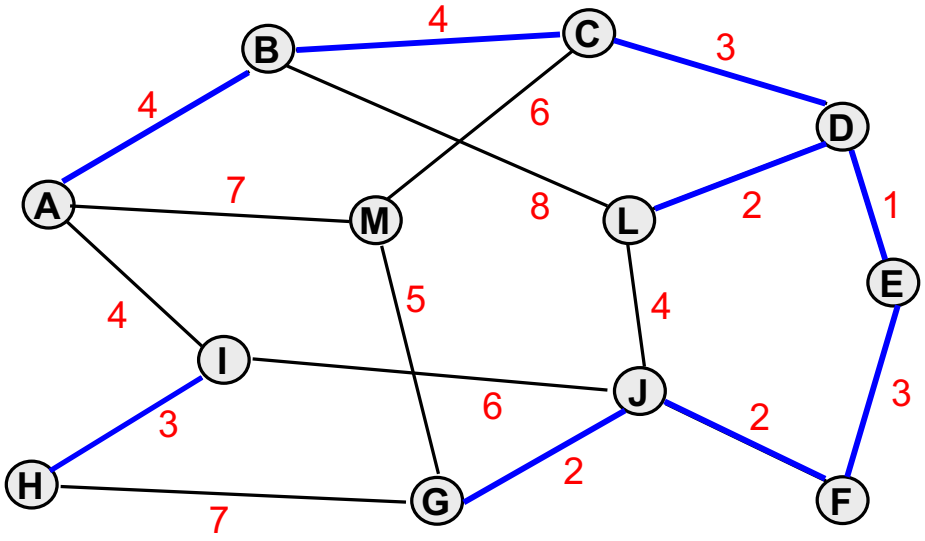


Árvore geradora
com custo 15

Ideia de Kruskal (1956)

- Princípio: a aresta de menor custo sempre pertence à árvore geradora de custo mínimo.
- Demonstração:
  - Suponha, por absurdo, que a aresta de custo mínimo não esteja na solução ótima.
  - A inserção desta aresta na solução ótima gera um ciclo.
  - Removendo-se a aresta de maior custo neste ciclo (que não é a inserida), obtém-se uma nova árvore geradora.
  - Essa nova árvore tem um custo inferior à solução ótima inicial: contradição.

Exemplo



c(T) = 24

Componentes

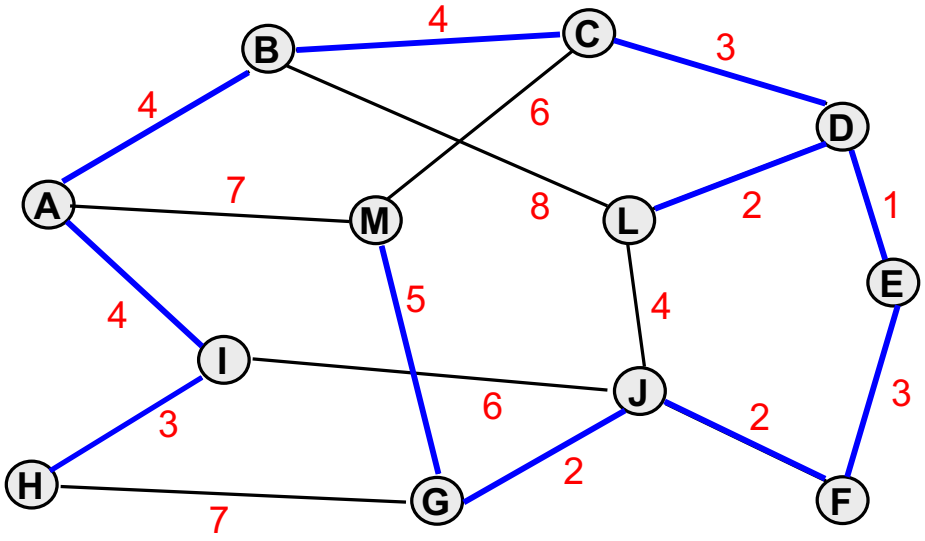
{ A, B, C, D, E, F, G, J, L }

{ H, I } { M }

Lista ordenada

e	c(e)
(D,E)	1
(D,L)	2
(F,J)	2
(G,J)	2
(C,D)	3
(E,F)	3
(H,I)	3
(A,B)	4
(B,C)	4
...	...

Exemplo



Lista ordenada

e	c(e)
...	...
(A,I)	4
(J,L)	4
(G,M)	5
(C,M)	6
(I,J)	6
(A,M)	7
(G,H)	7
(B,L)	8

c(T) = 33

Componentes

{ A, B, C, D, E, F, G, H, I, J, L, M }

Algoritmo de Kruskal

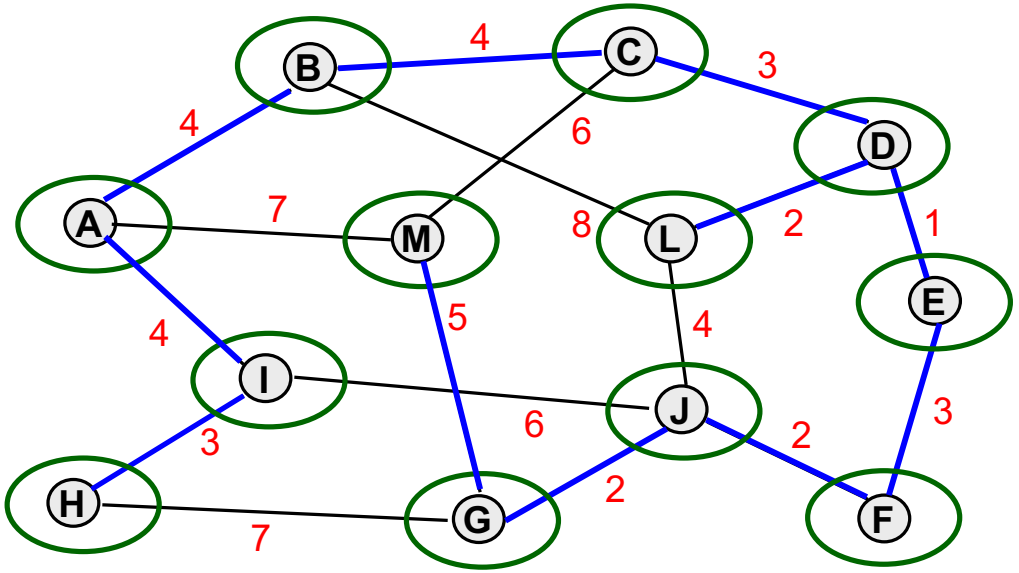
```
Kruskal(G) {
    T ← ∅;
    A ← vetor com as arestas em ordem crescente de custo;
    for (i=1; i<=n; i++)
        MakeSet(i);
    for (i=1; i<=m && |T|<n-1; i++) {
        <u,v> = A[i];
        if (Find(u) != Find(v)) {
            T ← T ∪ {<u,v>};
            Union(u,v);
        }
    }
}
```

- Complexidade de tempo:
  - Ordenação dos custos: $\Theta(m.\log m)$
  - n MakeSet, 2m Find, n-1 Union: $O((n+m).\log^* n)$
- Total: $\Theta(m.\log m)$, supondo $m > n$

Ideia de Prim (1957)

- Inicialmente, T será um vértice arbitrário de G.
- Critério de inclusão de vértices e arestas em T:
  - Dentre todas as arestas de G incidentes em T, escolhe-se a de menor custo.
  - Essa nova aresta e seu vértice adjacente serão incluídos em T somente se esse novo vértice ainda não estiver em T.
  - O processo termina quando T ficar com n vértices.
- É preciso utilizar uma estrutura de dados que armazene em ordem crescente de custo os vértices ainda não incluídos na árvore.

Exemplo



c(T) = 33

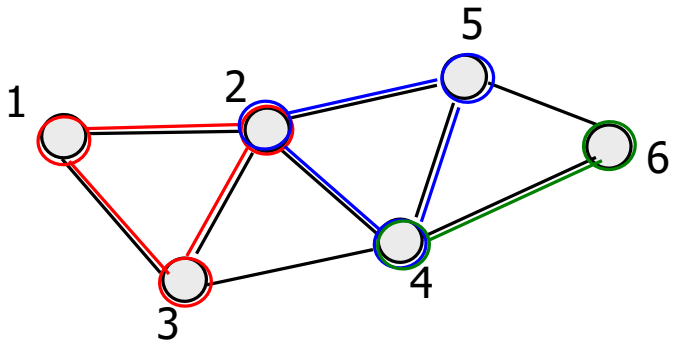
Algoritmo de Prim

```
Prim(r) {
        T ← ∅;          // árvore de espalhamento de custo mínimo
        U ← {r};        // vértices que já estão na árvore
        while (U ≠ V) {
                <u,v> = aresta de custo mínimo | u∈U e v∈V-U;
                T ← T ∪ {<u,v>}; // aqui, T contém só arestas
                U ← U ∪ {v};
        }
}
```

- Com *heap* de mínimo, o tempo de pior caso será Θ(m.log n):
  - O *heap* possuirá apenas os vértices vizinhos da árvore em construção, cada um com sua distância corrente (começará com o vértice r, com distância nula).
  - Quando um vértice é retirado desse *heap*, modificam-se as distâncias que seus vizinhos têm em relação à árvore (será preciso manter um vetor auxiliar que armazena a posição corrente de cada vértice no *heap*).
  - No total, são realizadas n extrações de mínimo e até m modificações de valor.

Outros problemas intratáveis

- Clique é um subconjunto de V que induz um grafo completo.
- Exemplos:



$C_1 = \{1, 2, 3\}$

$C_2 = \{2, 4, 5\}$

$C_3 = \{4, 6\}$

$C_4 = \{3\}$

- Não se conhece um algoritmo de tempo polinomial no tamanho do grafo que encontre seu clique máximo.
- Há ainda muitos problemas em grafos que são intratáveis, ou seja, se desconhecem resoluções de tempo polinomial: coloração, caixeiro viajante, cobertura de vértices, conjunto de vértices dominantes, etc.