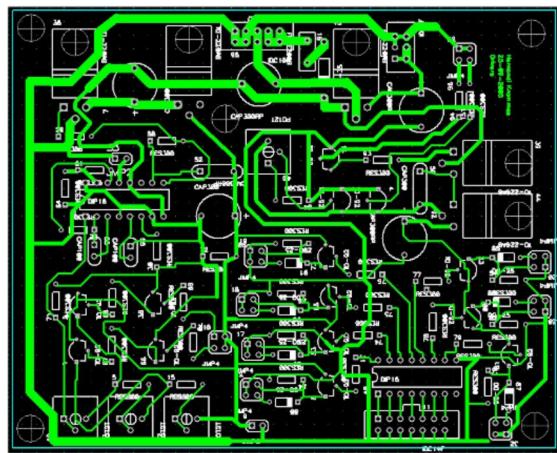
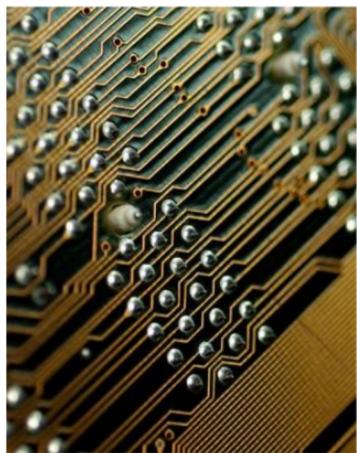

Algoritmos de Árvores Espalhadas Mínimas

"Pelos seus frutos os conhecereis. É possível alguém colher uvas de um espinheiro ou figos das ervas daninhas? Assim sendo, toda árvore boa produz bons frutos, mas a árvore ruim dá frutos ruins. Toda árvore que não produz bons frutos é cortada e atirada ao fogo."

(Mateus 7:16,17,19)

Revisão – Aplicação

- No projeto de circuitos eletrônicos
 - Interligar conjunto de pinos
 - Gastar o mínimo de fios



Revisão – Árvore Espalhada Mínima

- Por que “espalhada”?
 - Porque é um subgrafo espalhado do grafo original (tem o mesmo conjunto de vértices)
- Por que “árvore”?
 - Grafo conectado com o menor número possível de arestas
- Por que “mínima”?
 - Dentre todas as árvores espalhadas, é a que possui menor custo (soma das arestas)
- Nome alternativo: árvore espalhada de custo mínimo

Algoritmos

- Nesta aula, veremos dois algoritmos
 - Kruskal's
 - Prim's
- E veremos uma ideia genérica de algoritmo, que justifica a corretude dos dois
- Também veremos uma nova estrutura de dados
 - Disjoint-set forest

Algoritmo Genérico

Algoritmo Genérico

- Aqui mostramos uma estrutura genérica de algoritmo, que tem comprovação matemática de que é correto
 - Sempre retorna a árvore espalhada mínima
- Qualquer algoritmo que opere com as mesmas propriedades, tem a garantia formal/matemática de que está correto

Algoritmo Genérico

- Mantém um conjunto de arestas A
 - Propriedade: A é subconjunto das arestas de alguma árvore espalhada mínima
- É um algoritmo guloso, que opera adicionando uma aresta por vez (sem removê-la depois)
- Uma aresta que pode ser adicionada A assim (de modo a formar parte de alguma árvore espalhada mínima) será chamada de aresta segura

Algoritmo Genérico

GENERIC-MST(G)

A = vazio;

while (A não cobrir todos os vértices)
 encontra aresta segura;
 adiciona-a a A;

retorna A;

Algoritmo Genérico

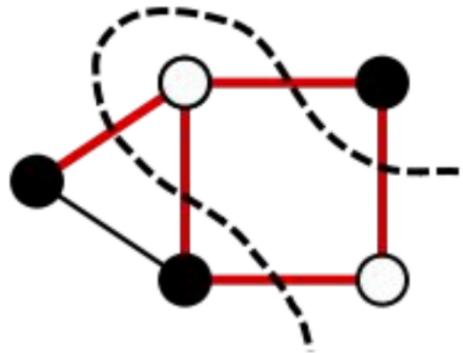
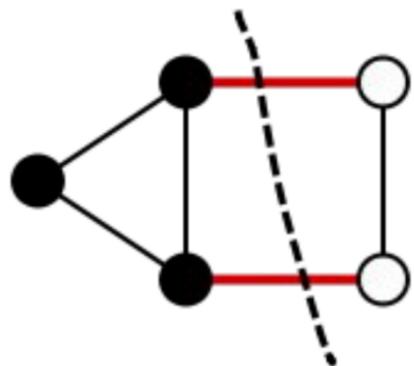
- O maior problema dos algoritmos é identificar arestas seguras a cada iteração
- Veremos propriedades para identificar arestas seguras com base
 - no grafo G
 - e no conjunto A
- Começando com definições auxiliares...

Definições Auxiliares

- Um corte ($S, V-S$)
 - Uma partição de V (conjunto de vértices) em dois subconjuntos disjuntos: S e $(V-S)$
- Uma aresta que cruza o corte
 - Tem uma extremidade em cada partição: S e $(V-S)$
- Um corte respeita um conjunto de arestas X
 - É quando nenhuma aresta de X cruza o corte

Exemplo

- Dois cortes possíveis:



Condições Suficientes

- Condição suficiente para a aresta (u,v) ser segura para A:
 - A tem que ser subconjunto de alguma árvore espalhada mínima (pode ser vazio)
 - Existe um corte $(S, V-S)$ que respeita A
 - Tal que aresta (u,v) é a aresta de menor peso que cruza esse corte

Exemplo

- Um grafo valorado
- Um corte
- Arestas que cruzam o corte
- Menor aresta que cruza o corte

Algoritmos

- Os algoritmos que veremos, de maneiras indiretas, seguem a ideia apresentada:
 - A cada iteração, eles definem um corte (implicitamente),
 - E, com base nele, escolhem a menor aresta que o cruza

Algoritmo de Kruskal

Algoritmo de Kruskal

- Ordena as arestas por peso, para analisar as arestas da menor para a maior
- Adicionando a aresta em A (a árvore mínima) se ela não criar ciclo
 - Pois não seria árvore
- Em iterações intermediárias, A é uma floresta
 - Pode estar desconectada

Algoritmo de Kruskal

- O teste do ciclo é indireto:
 - Guarda, para cada vértice do grafo, o componente conectado do qual ele faz parte, na saída A
 - Um ciclo em A acontece sse uma aresta ligar vértices do mesmo componente
 - Assim, testa se a aresta liga componentes diferentes (e adiciona, neste caso)

Algoritmo de Kruskal

- Funções auxiliares

- MAKE-SET(): associa cada vértice com um "set" diferente (onde o set representa o componente)
 - Cada vértice, sozinho, é um componente
- FIND-SET(): descobre de qual set (componente) um vértice faz parte
- UNION(): une os componentes de dois vértices, associando todos os vértices dos dois componentes a um mesmo set

Algoritmo de Kruskal

MST-KRUSKAL (G)

A = VAZIO;

para cada vértice v

 MAKE-SET (v);

ordena as arestas;

para cada aresta (u,v), em ordem crescente

 if (FIND-SET(u) ≠ FIND-SET(v))

 adiciona (u,v) a A

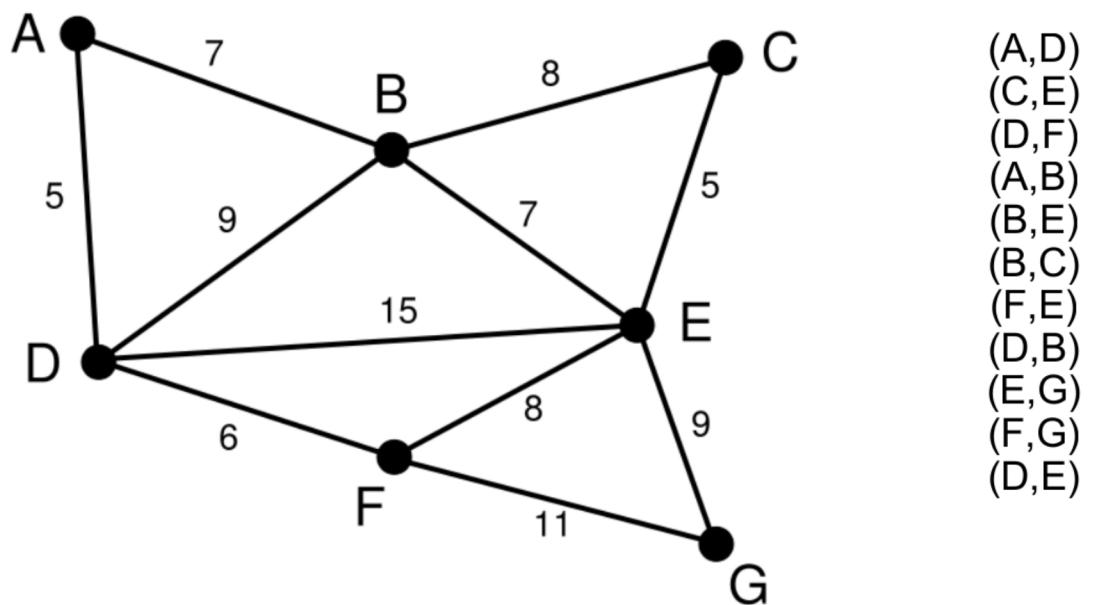
 UNION(u,v)

Explicação

- Cada componente, individualmente, pode ser visto como um corte no grafo
 - Separa ele do restante do grafo
- Se uma aresta liga componentes diferentes, ela cruza o corte
- Como as arestas são analisadas da menor para a maior, o algoritmo adiciona sempre a menor aresta que cruza um corte

Exemplo

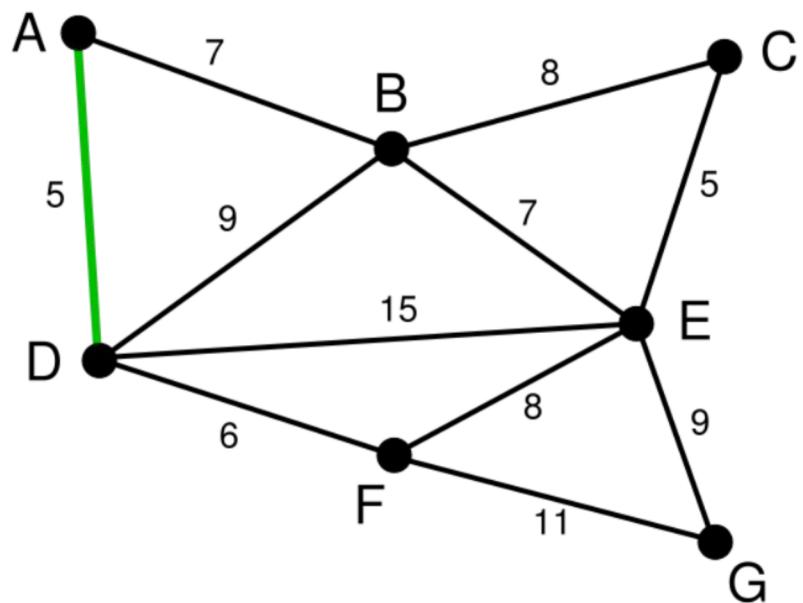
- Ordem de análise das arestas (por peso):



Exemplo

- Iteração 1

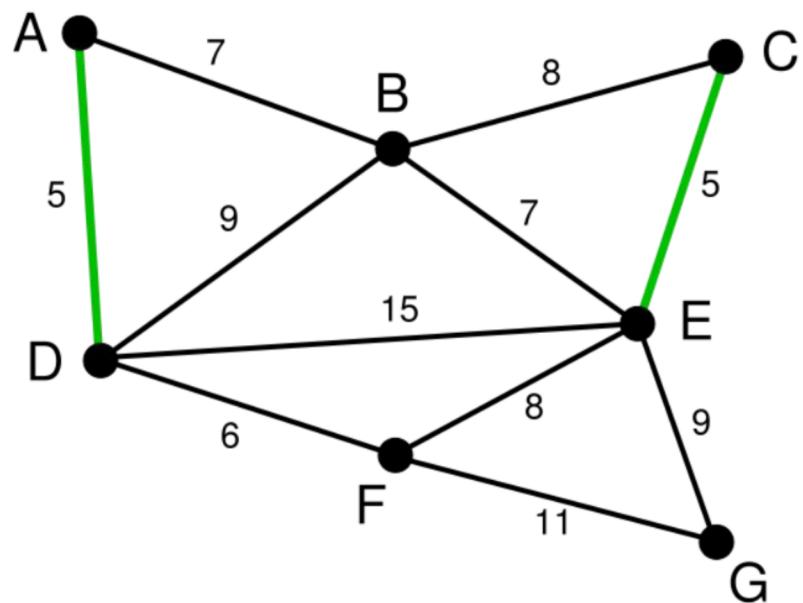
- Adiciona aresta (A,D)



Exemplo

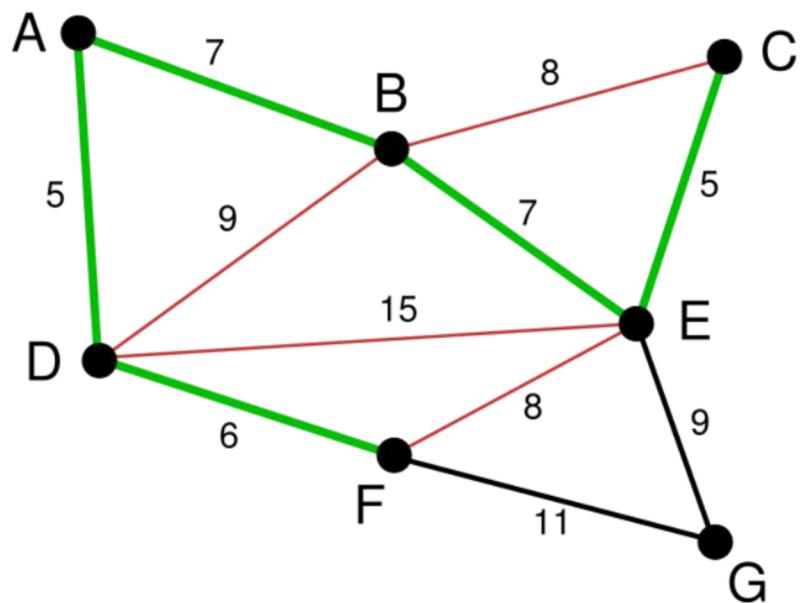
- Iteração 2

- Adiciona aresta (C,E)



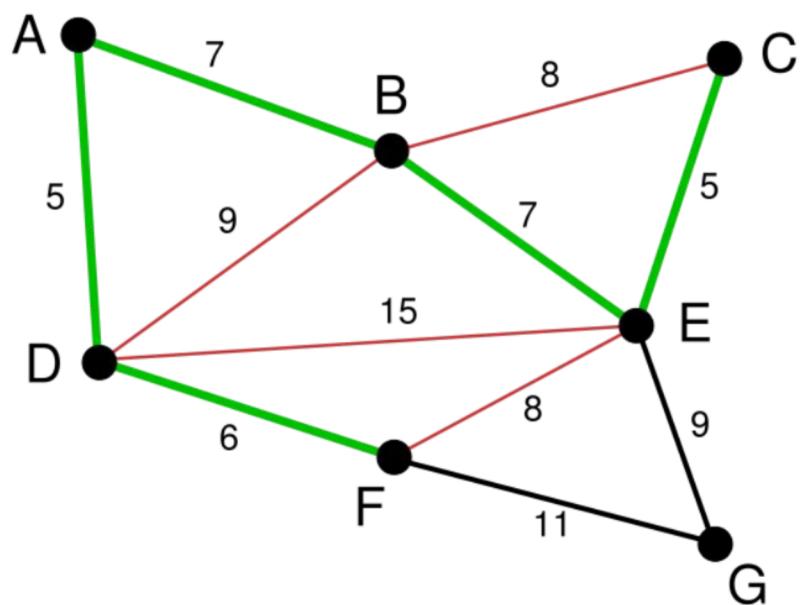
Exemplo

- Iterações 3 a 5:
 - Adiciona (D,F), (A,B), (B,E)



Exemplo

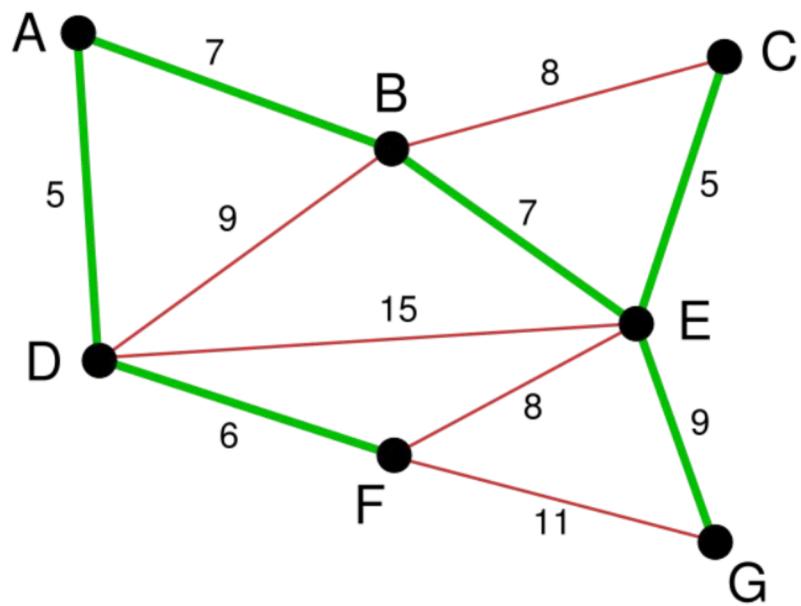
- Iterações 6 a 8
 - Ignora (B,C), (F,E), (D,B)



Exemplo

- Iteração 9

- Adiciona aresta (E,G) – pode parar (por quê?)



Complexidade

- O tempo de execução do algoritmo de Kruskal depende de como são implementados MAKE-SET, FIND-SET e UNION
- Estrutura de dados para partições disjuntas (disjoint-set data structure)

Estruturas para Partições Disjuntas

- Representam um universo dividido em conjuntos
 - Sem interseção comum
 - Cuja união resulta no universo
- Por questões didáticas, veremos
 - Uma alternativa simplista
 - E a estrutura Disjoint-Set Forest !
- No Kruskal, o universo é V e os conjuntos são os componentes conectados

Abordagem Simplicista

- Guarda, para cada elemento, um identificador (e.g. um número) que identifica o seu conjunto
 - Array set
- Exemplo: representar os conjuntos
 - $C_1 = \{ 1, 2, 3, 7 \}$
 - $C_2 = \{ 4, 6, 8, 9 \}$
 - $C_3 = \{ 5 \}$

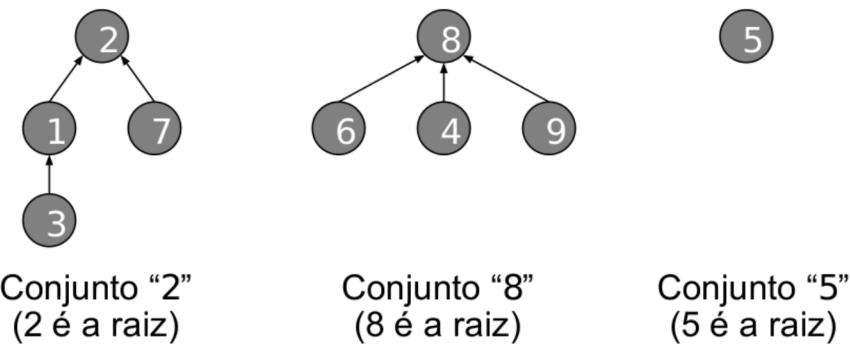
```
set[1] = 1 (C1)
set[2] = 1
set[3] = 1
set[4] = 2 (C2)
set[5] = 3 (C3)
set[6] = 2
set[7] = 1
set[8] = 2
set[9] = 2
```

Abordagem Simplicista

- Custos de tempo (para V elementos):
 - FIND-SET(v): $O(1)$
 - UNION(v,w): $O(V)$
- Custo do Kruskal: $O(V^2 + E \log V)$
 - Inicialização: $O(V)$
 - Ordenação: $O(E \cdot \log E) = O(E \log V)$
 - FIND-SET (2E chamadas), custo total: $O(E)$
 - UNION ($V-1$ chamadas), custo total: $O(V^2)$

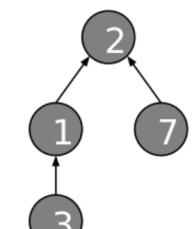
Disjoint-Set Forest

- Uma floresta, onde a raiz de cada árvore representa todos os demais vértices
 - Cada vértice guarda referência para seu pai
 - Se não tiver pai, o vértice é uma raiz

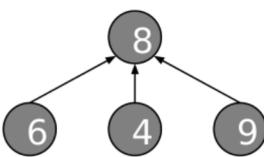


Disjoint-Set Forest

- Pode ser implementado como um array, em que as posições são os vértices e o conteúdo é o pai do vértice na árvore
 - Raízes têm pai -1



Conjunto “2”
(2 é a raiz)



Conjunto “8”
(8 é a raiz)



Conjunto “5”
(5 é a raiz)

pai-set[1] = 2
pai-set[2] = -1
pai-set[3] = 1
pai-set[4] = 8
pai-set[5] = -1
pai-set[6] = 8
pai-set[7] = 2
pai-set[8] = -1
pai-set[9] = 8

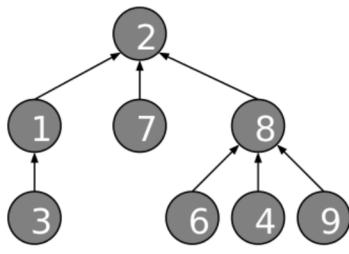
Disjoint-Set Forest

- Para descobrir o conjunto de um vértice, basta seguir pelos pais até chegar a uma raiz
- Qual o conjunto do vértice 3?
 - O pai do vértice 3 é 1
 - O pai do vértice 1 é 2
 - O pai do vértice 2 é -1,
 - Raiz
 - Resposta: “conjunto 2”

pai-set[1] = 2
pai-set[2] = -1
pai-set[3] = 1
pai-set[4] = 8
pai-set[5] = -1
pai-set[6] = 8
pai-set[7] = 2
pai-set[8] = -1
pai-set[9] = 8

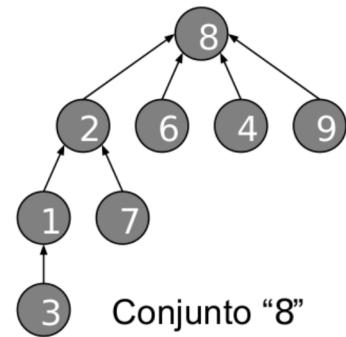
Disjoint-Set Forest

- A união de dois conjuntos (componentes) é feita inserindo-se uma raiz como filha da outra
- União dos conjuntos “2” e “8”



Conjunto “2”

OU



Conjunto “8”

Disjoint-Set Forest

- Existem algumas melhorias simples:
 - Redução da profundidade da árvore
 - Balanceamento da árvore
- Com apenas a última delas, o custo combinado de todas as operações MAKE, FIND e UNION usadas no Kruskal é
 - $O(E \log V)$
- Complexidade total do algoritmo de Kruskal:
 $O(E \log V)$

Referências

- Simulação do Algoritmo de Kruskal
 - <http://visualgo.net/mst.html>
- Simulação da Disjoint-Set Forest
 - <http://visualgo.net/ufds.html>

Algoritmo de Prim

Algoritmo de Prim

- O algoritmo inicia em um vértice r
 - Será a raiz (root) da árvore
- Vai aumentando A como uma “árvore parcial”
 - Diferente do Kruskal, A nunca fica desconectada
- Cada aresta adicionada vai conectar um novo vértice a esta árvore parcial

Algoritmo de Prim (Versão 1)

MST-PRIM(G, r)

apenas o vértice r é considerado conectado a A;

enquanto (não conectar todo vértice)

 encontra o vértice v mais barato de ser
 ligado a A;

 adiciona em A, a aresta que liga v;
 atualiza custo de ligar A aos vizinhos de v;

retorna A;

Refinando o Algoritmo...

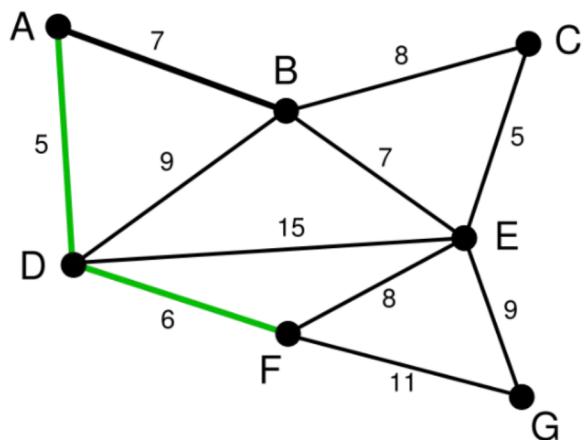
- Para cada vértice v , o algoritmo mantém informações da menor aresta conhecida que pode conectar v
 - $\pi[v]$: vértice da árvore de onde sai a menor aresta que conecta v
 - $\text{chave}[v]$: o peso da menor aresta que conecta v
- Se não for conhecida nenhuma aresta que liga v a algum vértice de A , temos:
 - $\pi[v] = -1$
 - $\text{chave}[v] = \infty$

Refinando o Algoritmo...

- Para escolher o vértice mais barato de ser adicionado à árvore, basta analisar os valores de $\text{chave}[]$
 - Custo para ligar v à arvore
- O vértice de menor valor de $\text{chave}[v]$ é o escolhido a cada iteração
 - Subentende-se que a aresta $(\pi[v], v)$ é adicionada

Exemplo

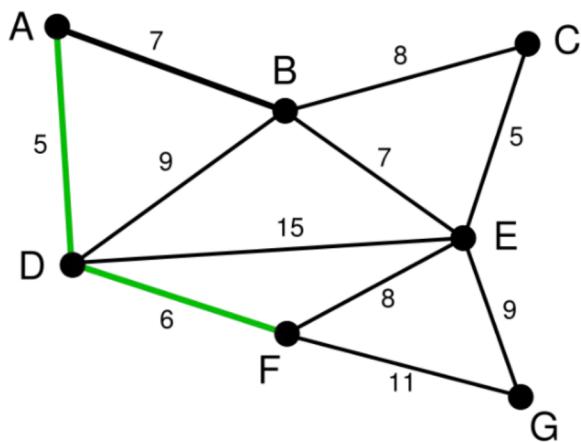
- Um passo intermediário do algoritmo...
 - Supondo inicio no vértice ‘A’



Para cada vértice v fora da árvore, qual o vértice da árvore que tem a aresta mais barata até v ?

Exemplo

- Um passo intermediário do algoritmo...



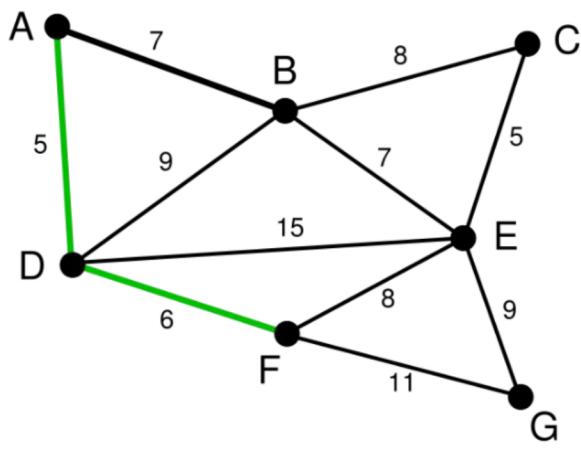
De onde partem as arestas que conectam B, C, E e G com menor custo:

$\pi[B] = A$
 $\pi[C] = -1$ (o vértice C ainda não pode ser conectado à árvore)
 $\pi[E] = F$
 $\pi[G] = F$

Quais os custos dessas ligações?

Exemplo

- Um passo intermediário do algoritmo...



Custos de conectar os vértices B, C, E e G à árvore:

chave[B] = 7
chave[C] = ∞
chave[E] = 8
chave[G] = 11

Qual o melhor vértice para ser unido à árvore agora?
Resposta: B!

Refinando o Algoritmo...

- Vértices não ligados à árvore ficam em uma fila de prioridade mínima Q (queue)
- Remove de Q sempre o de menor chave
 - Subentende-se que o vértice removido foi adicionado à árvore
- Depois, atualiza a chave dos vértices vizinhos
 - Vai ser necessário ajustar a fila Q

Algoritmo de Prim (Versão 2)

MST-PRIM(G, r)

para cada vértice u

chave[u] = ∞ ;

π [u] = -1;

chave[r] = 0;

Q = fila de prioridade com todos os vértices,
ordenados pelos valores chave[]

Algoritmo de Prim (Versão 2)

MST-PRIM(G, r)

```
para cada vértice u
    chave[u] = ∞;
    π [u] = -1;
chave[r] = 0;
Q = fila de prioridade com todos os vértices,
    ordenados pelos valores chave[]
while (Q ≠ ∅)
    u = REMOVE-MENOR(Q);
    para cada v de Adj[u]
        if (v ∈ Q e pesoAresta(u,v) < chave[v] )
            π [v] = u;
            chave[v] = pesoAresta(u,v);
            MUDA-ESTIMATIVA(Q, v, chave[v]);
```

Algoritmo de Prim (Versão 2)

- A saída do algoritmo pode ser o próprio array π
[]
 - Árvore enraizada com raiz em r
 - Para cada vértice, guarda o pai dele na árvore espalhada mínima
- Observação:
 - A escolha da raiz tem pouco importância
 - A árvore de saída pode ser tratada como sem raiz

Explicação

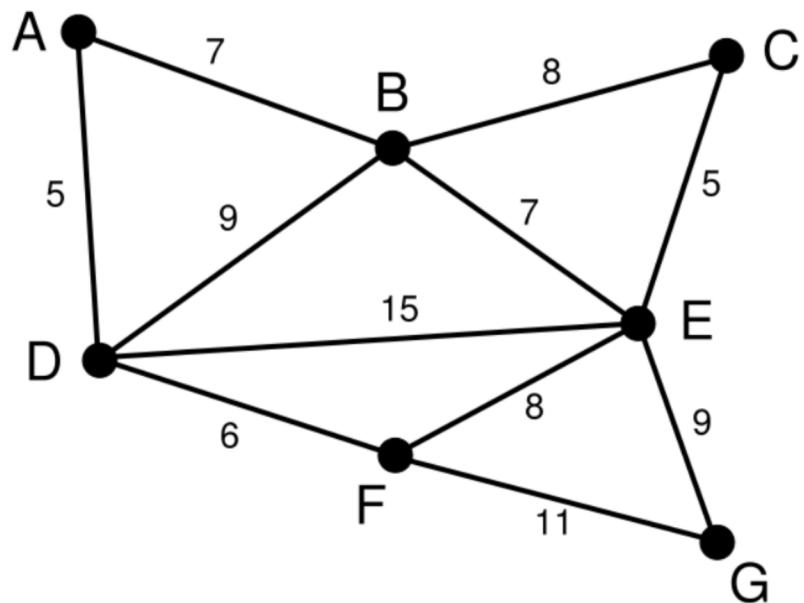
- Q guarda os vértices que não estão na árvore
 - Logo, Q induz um corte no grafo
 - Corte ($Q, V-Q$) ou (Q, A)
- O vértice escolhido a cada iteração:
 - Está em Q
 - Tem a aresta de menor custo até um vértice de A
 - Logo, esta é uma aresta leve que cruza o corte!

Semelhança com o Dijkstra

- O algoritmo de Prim é muito parecido com o algoritmo de Dijkstra (para menores caminhos)
 - Vértices mantidos em uma fila de prioridade
 - A cada iteração remove o de menor estimativa
 - Depois atualiza os custos dos vizinhos
- Operacionalmente, a diferença é o cálculo dos custos (as chaves)

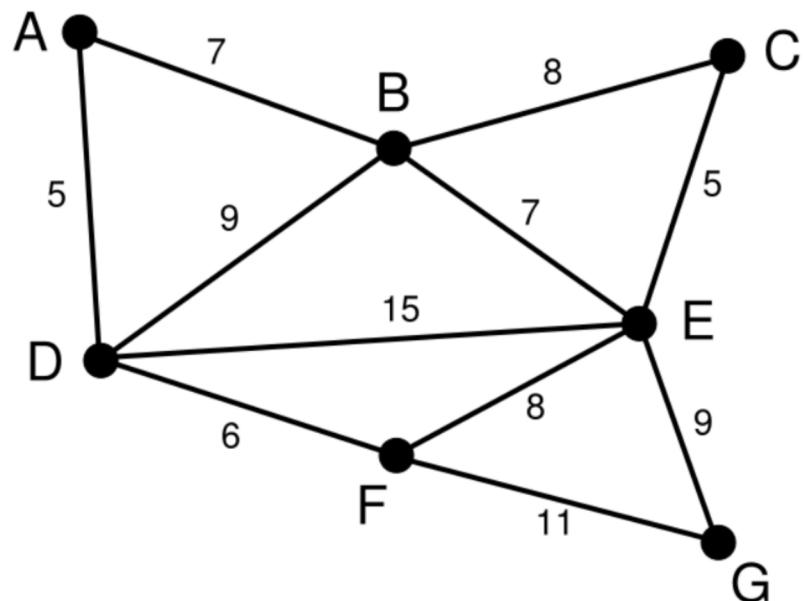
Exemplo

- O grafo original



Exemplo

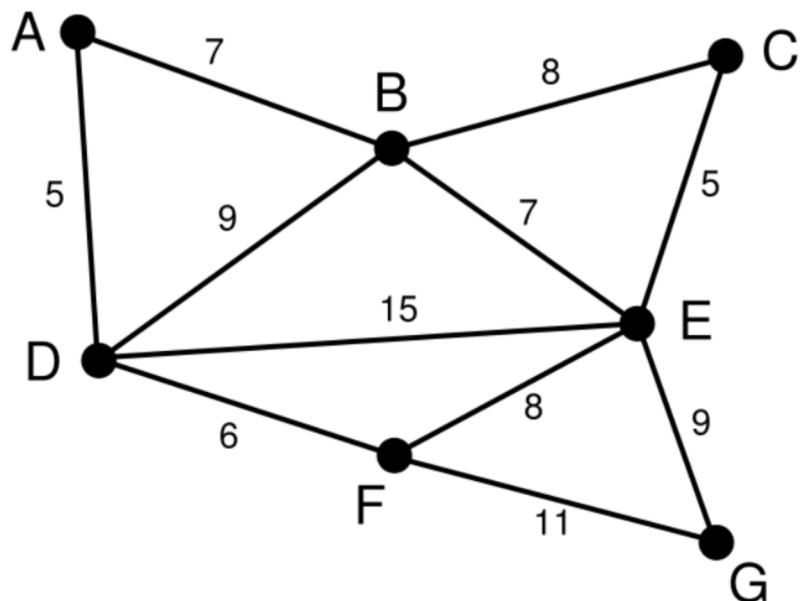
- Iniciando no vértice “A”
 - Inicia com chave[A] = 0
 - Com isso, ele será o primeiro da fila Q



Exemplo

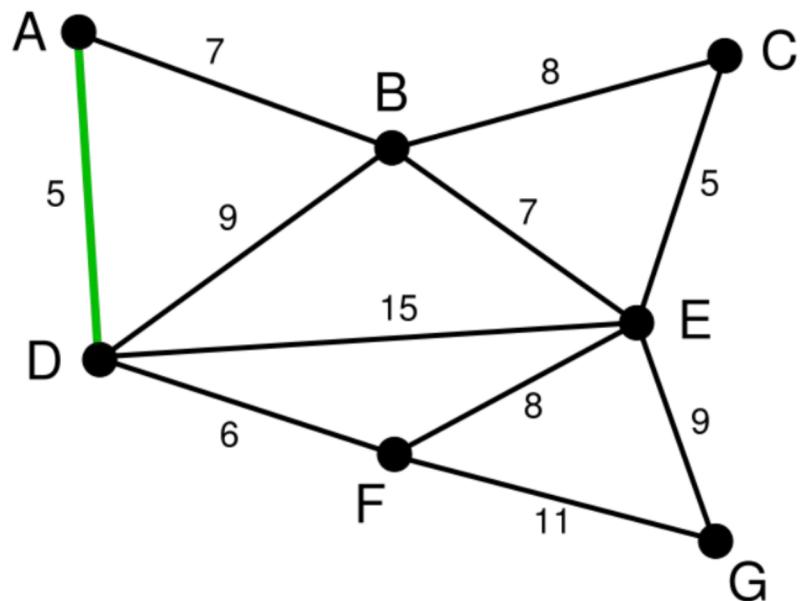
- Iteração 1

- Vértice atual (removido de Q): A
- Atualiza as chaves de: B e D



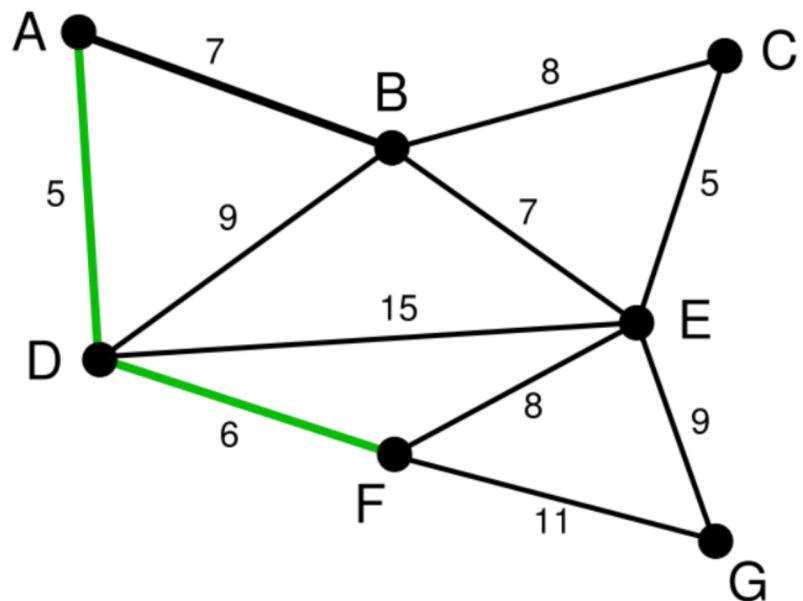
Exemplo

- Iteração 2
 - Vértice atual: D
 - Atualiza as chaves de: E e F



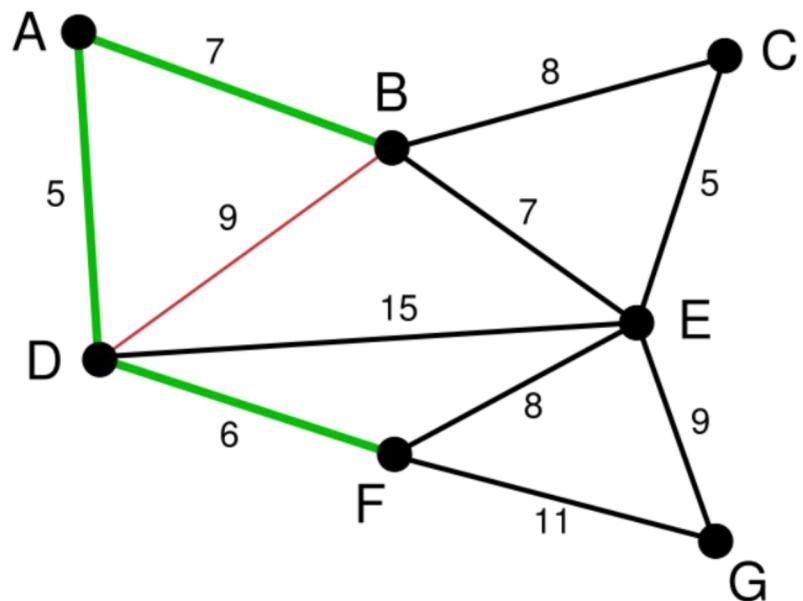
Exemplo

- Iteração 3
 - Vértice atual: F
 - Atualiza as chaves de: E e G



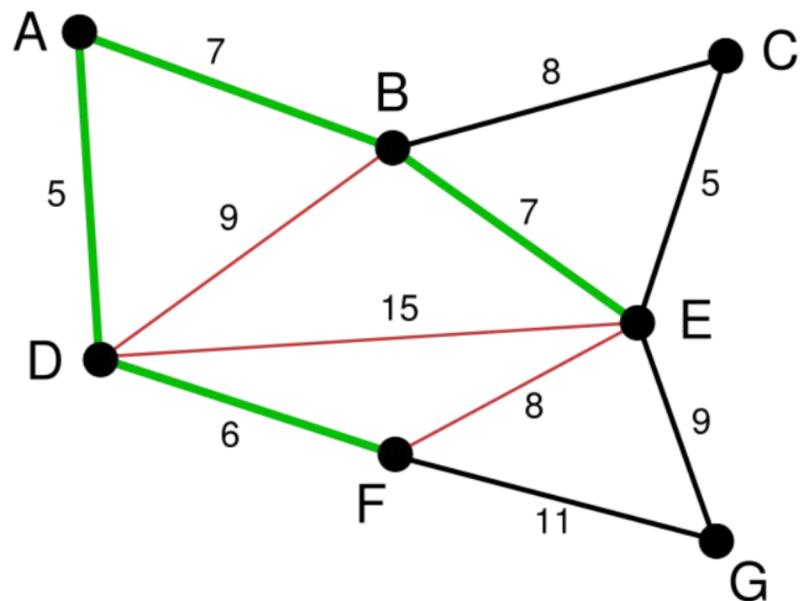
Exemplo

- Iteração 4
 - Vértice atual: B
 - Atualiza as chaves de: C e E



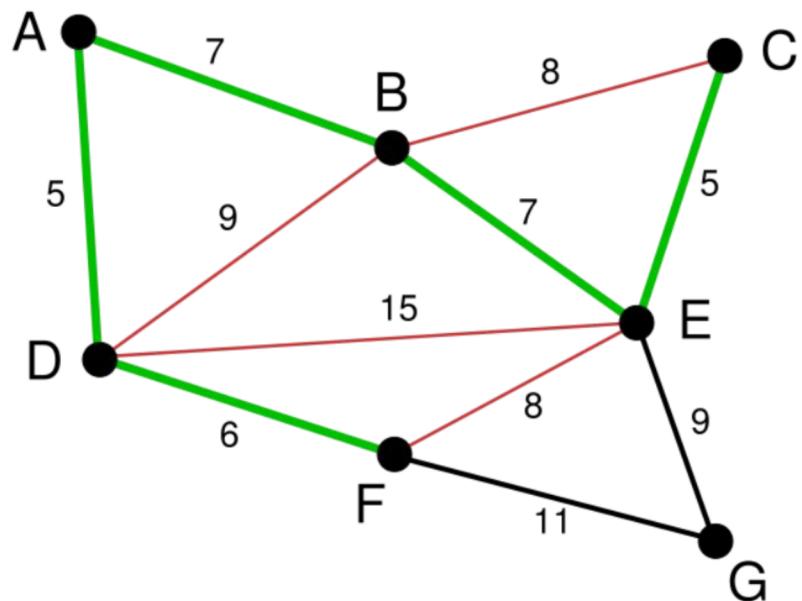
Exemplo

- Iteração 5
 - Vértice atual: E
 - Atualiza as chaves de: C e G



Exemplo

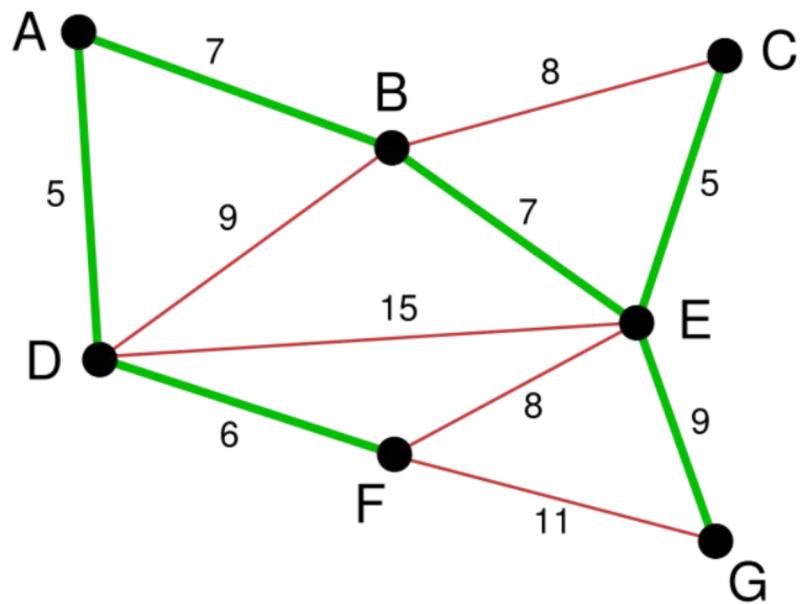
- Iteração 6
 - Vértice atual: C
 - Não atualiza chaves



Exemplo

- Iteração 7

- Vértice atual: G
- Não atualiza chaves



Complexidade de Tempo

- Depende, principalmente, da implementação da fila de prioridades
- Operações críticas
 - Criar a fila, inicialmente
 - Remover o primeiro
 - Reajustes por mudanças da chave
- O ideal é implementar a fila com heap

Complexidade de Tempo

- Implementado com “Heaps de Fibonacci”, o algoritmo de Prim atinge a sua melhor performance:

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

Referências

- Simulações do algoritmo de Prim
 - <http://visualgo.net/mst.html>
 - <http://www-b2.is.tokushima-u.ac.jp/~ikeda/suuri/dijkstra/Prim.shtml>

Comentários Finais

Complexidade – Comparação

	Prim	Kruskal
Grafos quaisquer	$O(E + V \log V)$	$O(E \log V)$
Grafos esparsos ($E=O(V)$)	$O(V \log V)$	$O(V \log V)$
Grafos densos ($E=O(V^2)$)	$O(V^2)$	$O(V^2 \log V)$

Outros Algoritmos

- Há vários outros algoritmos gulosos e com complexidade semelhantes
 - Boruvka's, etc
- Algoritmos recentes são marginalmente melhores, teoricamente (Chazelle, 2000)
 - Na prática, pode não fazer muita diferença...