Árvores geradoras mínimas

Marco A L Barbosa
malbarbo.pro.br

Departamento de Informática
Universidade Estadual de Maringá

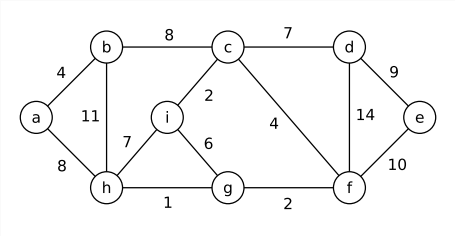
Problema

O prefeito de uma cidade decidiu conectar todas as escolas através de uma rede de fibra ótica a internet. Uma das escolas (qualquer uma) servirá como *gateway*, as demais devem ser ligadas a esta escola de alguma forma.

Em um estudo preliminar foi feito um levantamento das possível ligações entre as escolas e determinado o custo de cada ligação. O prefeito quer gastar o mínimo possível e garantir que todas as escolas estejam conectas a internet. Como decidir quais ligações devem ser feitas?

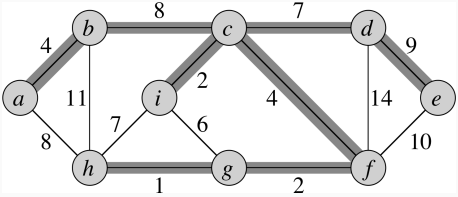
Vamos criar um grafo com estas informações para nos ajudar a entender o problema:

- Cada escola é representada por um vértice
- Cada possível ligação entre as escolas é representada por uma aresta com um valor (custo da ligação)



Fazer apenas a ligação (i, g) conecta todas as escolas? Não!

Fazer apenas as ligações (i, g), (g, h) e (h, i) conecta todas as escolas? Não!



Quais características uma solução deve ter?

- Deve conectar todos os vértices
- Deve ter $|V| - 1$ arestas, ou seja, ser uma árvore
- Ter custo mínimo entre todas as possíveis árvores (pode haver mais que uma)

Uma forma de confirmar que entendemos o problema é fazer uma definição precisa dele. Então vamos fazer isso.

Dado um grafo conexo não orientado $G = (V, E)$ e uma função peso $w : E \to \mathbb{R}$, queremos encontrar um subconjunto acíclico $T \subseteq E$ que conecte todos os vértices de G e cujo peso total

$$w(T) = \sum_{(u,v) \in T} w(u, v)$$

seja mínimo.

Como T é acíclico e conecta todos os vértices, T forma uma árvore, que chamamos de **árvore geradora mínima** (AGM ou MST em inglês).

No contexto de árvores geradoras, vamos usar T para nos referir tanto ao conjunto de arestas quanto ao subgrafo induzido pelo conjunto de arestas.

Tipo do problema

Agora que entendemos o problema, vamos construir exemplos e pensar no processo que utilizamos para encontrar as respostas. Conhecer o tipo do problema que estamos lidando pode nos ajudar nesta etapa.

Tipos de problemas computacionais

Decisão (sim ou não). Exemplo: Verificar se existe um caminho simples entre dois vértices em um grafo.

Busca (saída arbitrária). Exemplo: Encontrar um caminho simples entre dois vértices em um grafo.

Contagem (número de soluções para um problema de busca). Exemplo: Contar quantos caminhos simples existem entre dois vértices de um grafo.

Otimização (melhor solução entre todas as soluções para um problema de busca). Exemplo: Encontrar um caminho simples com o menor número de arestas entre dois vértices em um grafo.

Tipo do problema

Que tipo de problema é encontrar uma árvore geradora de custo mínimo? De otimização.

Quais técnicas de projeto de algoritmos são comumente utilizadas para problemas de otimização?

- Algoritmo guloso
- Programação dinâmica
- Melhoramento iterativo
- Etc

Vamos tentar utilizar estas técnicas para derivar hipóteses de algoritmos.

Discutimos em sala como chegamos nessas ideias.

Melhoramento iterativo

- Iniciar como uma árvore geradora qualquer e tentar derivar uma árvore com peso menor substituindo uma aresta que está na árvore por outra que não está
    - Inserir uma aresta formando um ciclo na árvore e remover a aresta de maior peso do ciclo
    - Remover uma aresta da árvore desconectando a árvore e inserir uma aresta de menor peso que reconecta a árvore

Algoritmo Guloso 0

- Tentar remover as arestas do grafo em ordem decrescente de peso evitando remover as arestas que desconectam o grafo

Algoritmo Guloso 1

- Iniciar sem nenhuma aresta e ir adicionando arestas em ordem crescente de peso evitando formar ciclos

Algoritmo Guloso 2

- Iniciar com um vértice na árvore e ir ligando novos vértices à árvore usando as arestas de menor peso

A ideia do Algoritmo Guloso 1 foi proposta primeiramente por Kruskal e a ideia do Algoritmo Guloso 2 por Prim.

Todos os algoritmos funcionam fazendo repetidamente a inserção e/ou remoção de arestas.

Dessa forma, é interessante se perguntar quando é “seguro” inserir ou remover uma aresta da AGM.

Vamos ver novamente as duas forma de melhorar uma árvore no algoritmo melhorativo que descrevemos:

- Inserir uma aresta formando um ciclo na árvore e remover a aresta de maior peso do ciclo
- Remover uma aresta da árvore desconectando a árvore e inserir uma aresta de menor peso que reconecta a árvore

Hipóteses

Se todos os pesos forem diferentes, então

- a aresta de maior peso de um ciclo não pertence a AGM;
- a aresta de menor peso que conecta um vértice de $S \subset V$ e um vértice de $V - S$ pertence a AGM;

Remoção segura

A aresta de maior peso de um ciclo não pertence a AGM.

Prova por contradição.

Seja (u, v) a aresta de maior peso de um ciclo C qualquer e suponha que ela esteja na AGM T.

O que acontece se removermos (u, v) de T? Separamos a árvore em duas componentes, onde u está em uma componente e v está em outra.

Existe uma forma de reconectar as duas componentes com uma outra aresta que não seja (u, v)? Sim. Para encontrarmos a aresta seguimos o ciclo C, mas ao invés de seguir a aresta (u, v), começamos com u e vamos para a outra "direção" do ciclo. Para alguma aresta (x, y) em C, x vai estar no mesmo componente de u e y no mesmo componente de v. Usamos essa aresta para criar uma nova árvore $T' = T \cup \{(x, y)\} - \{(u, v)\}$.

Quem tem menor peso, (x, y) ou (u, v)? (x, y), pois (u, v) é a aresta de maior de C. Quem tem menor peso, T' ou T? T'. Então T não pode ser um AGM, o que é uma contradição.

Inserção segura

A aresta de menor peso que conecta um vértice de $S \subset V$ e um vértice de $V - S$ pertence a AGM.

Prova por contradição.

Seja (u, v) uma aresta com $u \in S$ e $v \in (V - S)$ e seja T uma AGM que não contenha (u, v).

Existem um caminho de u para v em T? Sim. Seja (x, y) uma aresta qualquer desse caminho de maneira que $x \in S$ e $y \in (V - S)$.

A aresta (x, y) pode ser aresta (u, v)? Não, pois (u, v) não está em T e (x, y) está.

O que acontece se removermos a aresta (x, y) de T? Separamos a árvore em duas componentes, um componente com os vértices de S e outra com os vértices de $V - S$. Podemos reconectar a árvore com a aresta (u, v)? Sim. Obtemos uma árvore $T' = T \cup \{(u, v)\} - \{(x, y)\}$.

Quem tem menor peso, (x, y) ou (u, v)? (u, v), pois é aresta de menor peso que conecta um vértice de S e outro de $V - S$. Quem tem menor peso, T' ou T? T'. Então T não pode ser um AGM, o que é uma contradição.

Como construir uma árvore geradora mínima?

O livro CLRS apresenta os algoritmos de Kruskal e Prim em uma abordagem unificada. Além disso, para desenvolver o conceito de aresta segura, ele não requer que todas as arestas do grafo tenha pesos distintos.

Vamos ver como o livro apresenta esse assunto.

Como construir uma árvore geradora mínima? Uma aresta por vez.

Começamos com um conjunto vazio A. Em cada iteração determinamos um aresta (u, v) que pode ser adicionada a A, de forma a manter a seguinte invariante:

- Antes de cada iteração, A é um subconjunto de alguma árvore geradora mínima. Chamamos a aresta (u, v) de **aresta segura** para A.

No final, temos uma árvore geradora mínima!

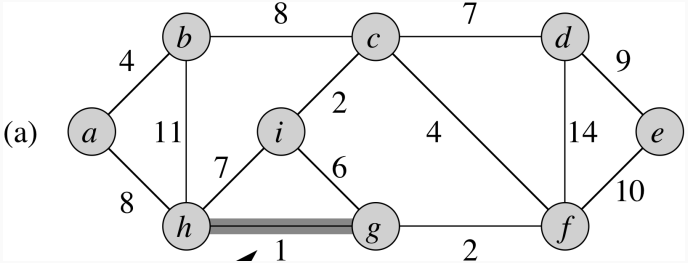
GENERIC-MST(G, w)

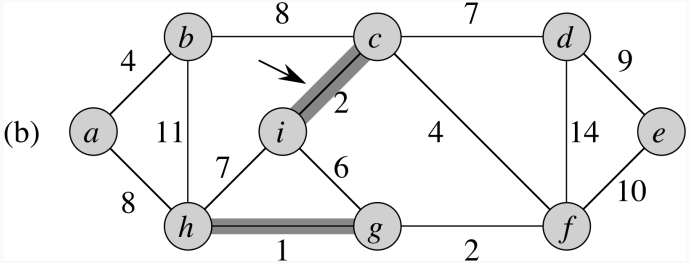
1 $A = \emptyset$
2 **while** A não forma uma árvore geradora
3 encontre uma aresta (u, v) que seja segura para A
4 $A = A \cup \{(u, v)\}$
5 **return** A

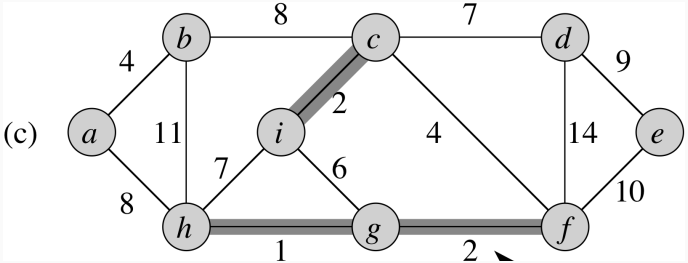
Agora vamos ver com detalhes como os algoritmo de Krukal e Prim constroem uma AGM uma aresta por vez.

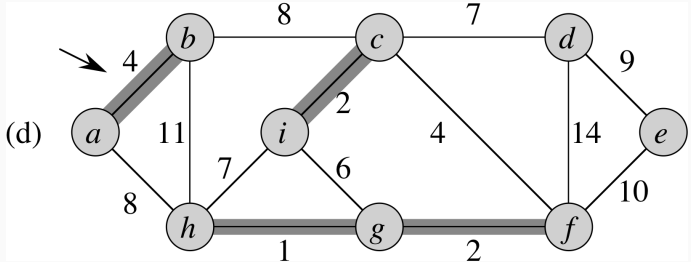
Baseia-se diretamente no algoritmo genérico

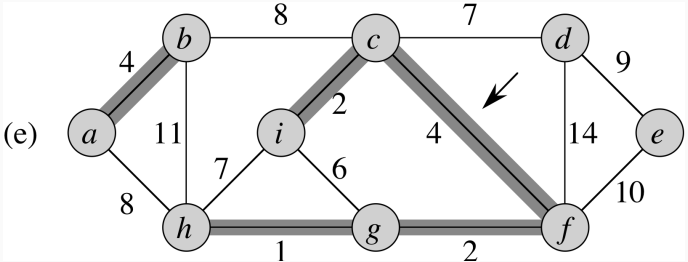
- Inicialmente cada vértice está em sua própria componente (árvore)
- De todas as arestas que conectam duas árvores quaisquer na floresta, uma aresta (u, v) de peso mínimo é escolhida. A aresta (u, v) é segura para alguma das duas árvores (vamos provar isso depois).

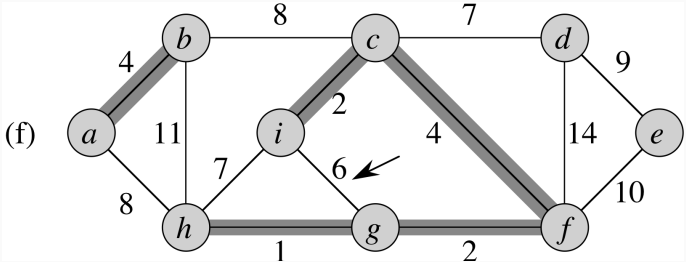

(a)
a
b
c
d
e
f
g
h
i
4
8
11
8
7
2
4
14
9
6
10
1
2

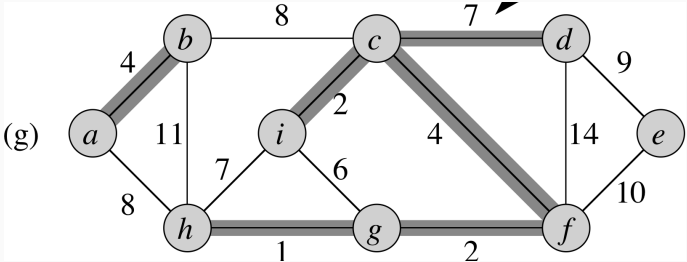

(b)
a
b
c
d
e
f
g
h
i
8
7
4
9
2
11
4
14
7
6
8
10
1
2

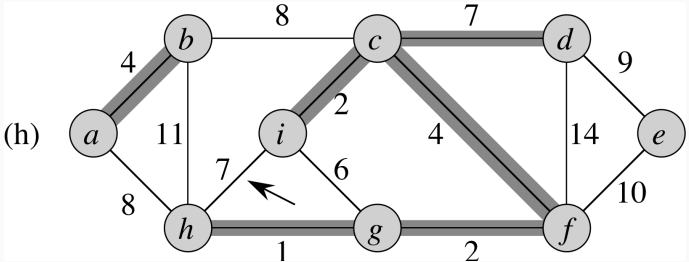

(c)
a
b
c
d
e
f
g
h
i
8
7
4
9
2
11
4
14
7
6
8
10
1
2

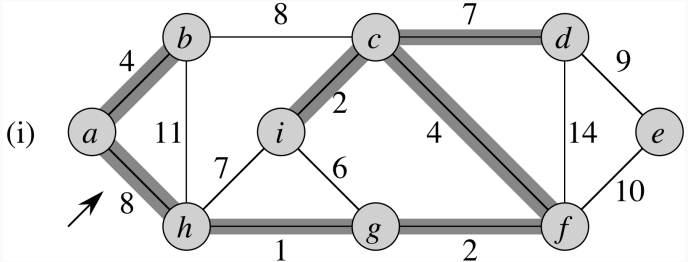

(d)
a
b
c
d
e
f
g
h
i
4
8
7
9
11
2
4
14
7
6
8
10
1
2

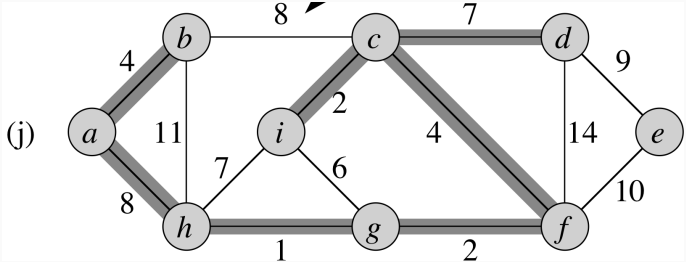

(e)
a
b
c
d
e
f
g
h
i
8
7
4
9
2
11
4
14
7
6
8
10
1
2

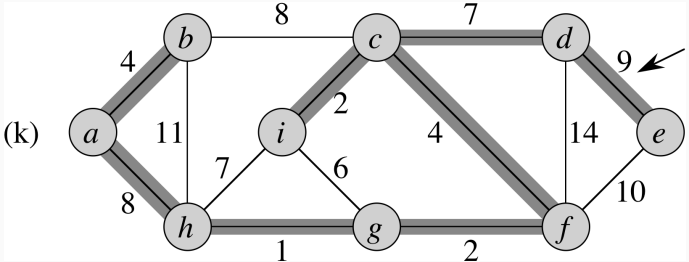

(f)
a
b
c
d
e
f
g
h
i
8
7
4
9
2
11
4
14
7
6
8
10
1
2

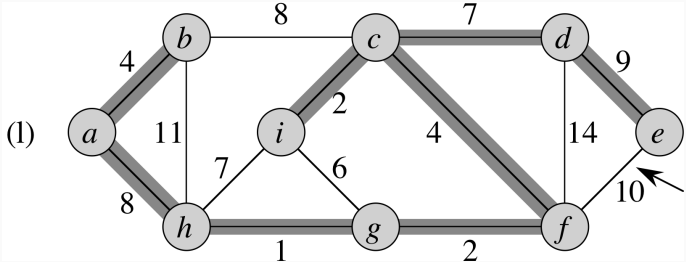

(g)
a
b
c
d
e
f
g
h
i
4
8
8
7
9
11
2
4
14
7
6
10
8
1
2

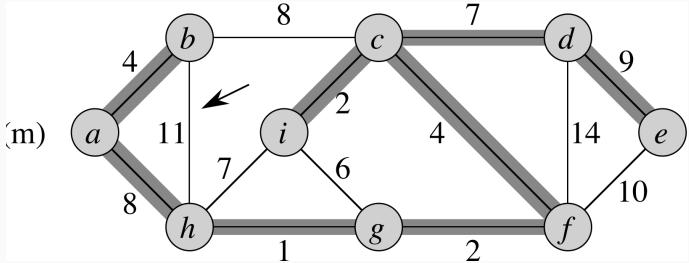

(h)
a
b
c
d
e
f
g
h
i
8
7
4
9
2
11
4
14
7
6
8
10
1
2

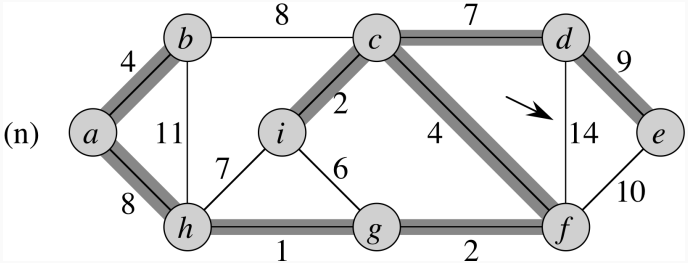

(i)
a
b
c
d
e
f
g
h
i
4
8
8
7
9
11
2
4
14
7
6
10
8
1
2


(j)
a
b
c
d
e
f
g
h
i
8
7
4
9
2
11
4
14
7
6
8
10
1
2


(k)
a
b
c
d
e
f
g
h
i
8
7
4
9
2
11
4
14
7
6
8
10
1
2


(l)
a
b
c
d
e
f
g
h
i
8
7
4
9
2
11
4
14
7
6
8
10
1
2


(m)
a
b
c
d
e
f
g
h
i
8
7
4
9
11
2
4
14
7
6
8
10
1
2


(n)
a
b
c
d
e
f
g
h
i
4
8
8
7
9
11
2
4
14
7
6
10
8
1
2

Qual é o "desafio" para implementar este algoritmo de forma eficiente? É o gerenciamento das árvores: verificar se dois vértices estão na mesma árvore e juntar duas árvores.

Esta situação é comum em outros contextos e pode ser resolvida com um estrutura de dados para conjuntos disjuntos. Três operações são definidas para este tipo de estrutura:

- MARK-SET(v), coloca o vértice v no seu próprio conjunto (árvore)
- FIND-SET(v), identifica em qual conjunto o vértice v está
- UNION(u, v), junta os vértices do conjunto de u e v em um único conjunto

A seção 21.3 descreve uma implementação bastante eficiente, onde o tempo de execução de m operações em n elementos é $O(m\alpha(n))$, sendo α é uma função que cresce lentamente.

Baseado no exemplo de funcionamento e nas operações de conjuntos disjuntos, vamos escrever o pseudo código do algoritmo.

MST-KRUSKAL(G, w)

1 $A = \emptyset$
2 **for** each vertex $v \in G.V$
3 MAKE-SET(v)
4 ordenar em ordem não decrescente as arestas de G pelo peso w
5 **for** each edge $(u, v) \in E^*$
6 **if** FIND-SET$(u) \neq$ FIND-SET(v)
7 $A = A \cup \{(u, v)\}$
8 UNION(u, v)
9 **return** A

* Em ordem não decrescente de peso

Análise do tempo de execução

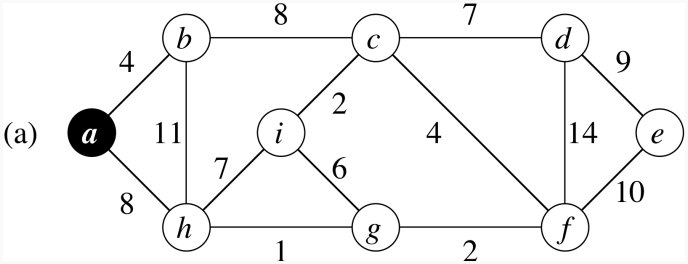
- A ordenação das arestas na linha 4 tem tempo $O(E \lg E)$
- Operações com conjuntos disjunto
  - O laço das linhas 5 a 8 executa $O(E)$ vezes FIND-SET e UNION. Juntamente com as $|V|$ operações MAKE-SET, elas têm tempo $O((V + E)\alpha(V))$. Pelo fato de G ser conexo, temos que $|E| \geq |V| - 1$ e portanto o tempo com operações com conjuntos disjuntos é $O(E\alpha(V))$. Além disso, $\alpha(|V|) = O(\lg V) = O(\lg E)$, e portanto o tempo total das operações com conjuntos disjuntos é $O(E \lg E)$.
- Somando o custo de ordenação e o custo das operações com conjuntos disjuntos, temos $O(E \lg E)$. Observando que $|E| < |V^2|$, temos que $\lg |E| = O(\lg V)$, e portanto, o tempo de execução do algoritmo é $O(E \lg V)$.

Também baseado diretamente no algoritmo genérico

- Começa com uma árvore com uma raiz arbitrária r
- Expande a árvore até alcançar todos os vértices, sempre escolhendo conectar um vértice da árvore com outro que não está na árvore usando a aresta de menor peso

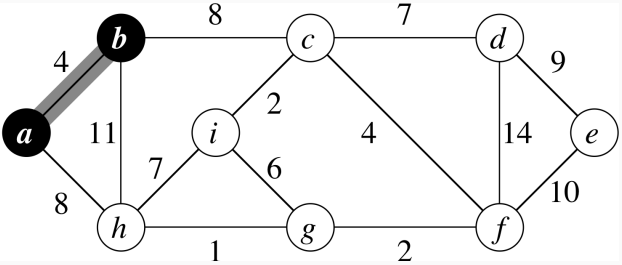
Como apenas uma árvore é mantida, podemos representá-la da mesma forma que no BFS e DFS.

- Para cada vértice v, mantemos o atributo $v.\pi = u$, onde (u, v) é uma aresta de peso mínimo que conecta v a um vértice da árvore (o vértice u)
- Também precisamos armazenar o peso da aresta (u, v), para isso mantemos em cada vértice v um atributo $v.chave = w(v, u)$. Se v não pode ser ligado a árvore, então $v.\pi = \text{NIL}$ e $v.chave = \infty$

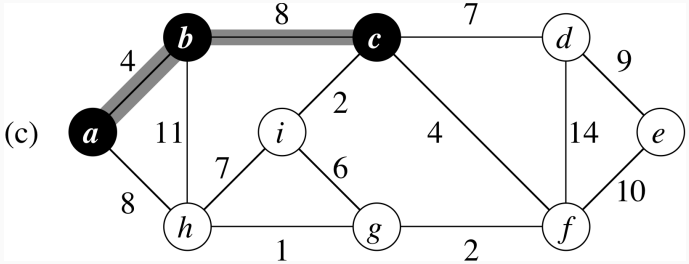


Vértice	***a***	*b*	*c*	*d*	*e*	*f*	*g*	*h*	*i*
chave	0	4	∞	∞	∞	∞	∞	8	∞
π		*a*						*a*	

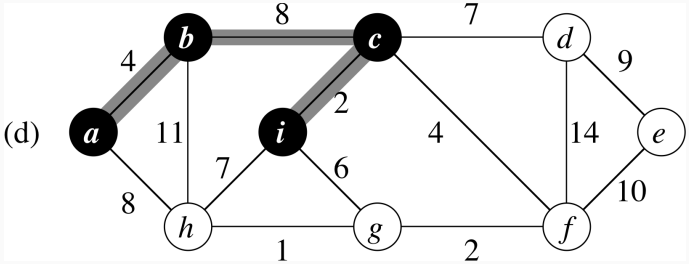
(b)



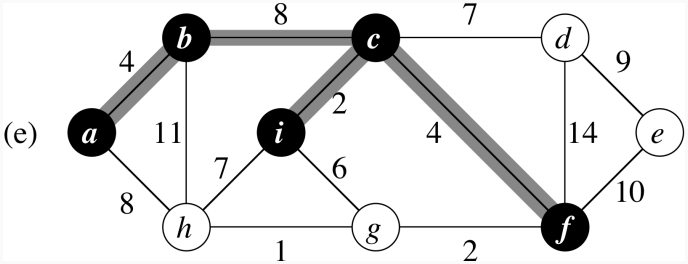
Vértice	**a**	**b**	c	d	e	f	g	h	i
chave	0	4	8	∞	∞	∞	∞	8	∞
π		a	b					a	



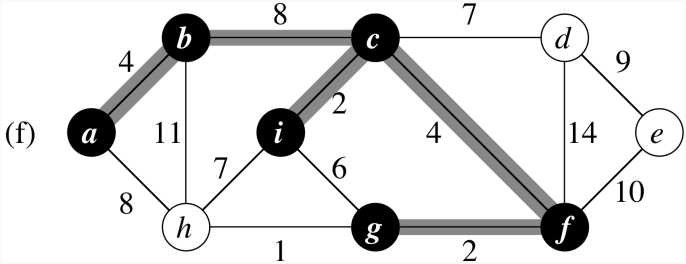
Vértice	**a**	**b**	**c**	d	e	f	g	h	i
chave	0	4	8	7	∞	4	∞	8	2
π		a	b	c		c		a	c



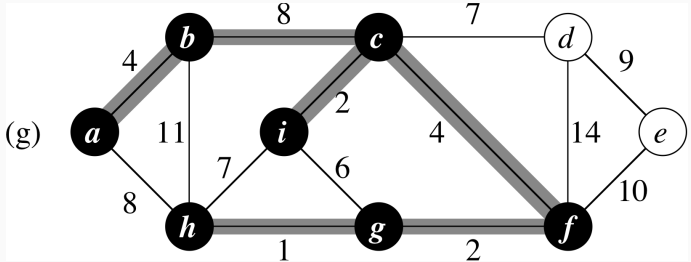
Vértice	**a**	**b**	**c**	d	e	f	g	h	**i**
chave	0	4	8	7	∞	4	6	7	2
π		a	b	c		c	i	i	c



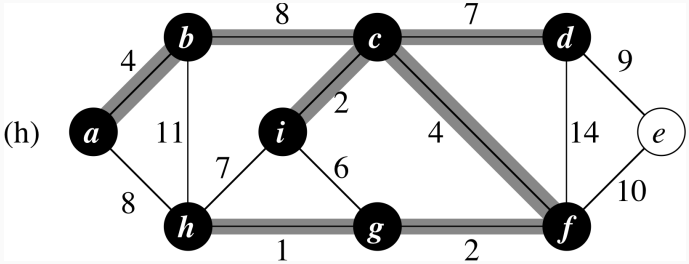
Vértice	**a**	**b**	**c**	d	e	**f**	g	h	**i**
chave	0	4	8	7	10	4	2	7	2
π		a	b	c	f	c	f	i	c



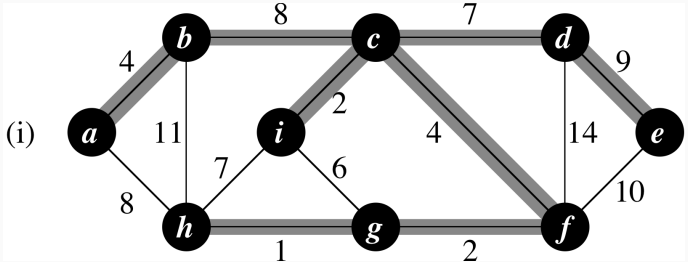
Vértice	**a**	**b**	**c**	d	e	**f**	**g**	h	**i**
chave	0	4	8	7	10	4	2	1	2
π		a	b	c	f	c	f	g	c



Vértice	**a**	**b**	**c**	d	e	**f**	**g**	**h**	**i**
chave	0	4	8	7	10	4	2	1	2
π		a	b	c	f	c	f	g	c



Vértice	**a**	**b**	**c**	**d**	*e*	**f**	**g**	**h**	**i**
chave	0	4	8	7	9	4	2	1	2
π		*a*	*b*	*c*	*d*	*c*	*f*	*g*	*c*



Vértice	*a*	*b*	*c*	*d*	*e*	*f*	*g*	*h*	*i*
chave	0	4	8	7	9	4	2	1	2
π		*a*	*b*	*c*	*d*	*c*	*f*	*g*	*c*

Qual é o desafio para implementar o algoritmo de Prim de forma eficiente? É escolher qual o próximo vértice a ser conectado na árvore.

Vamos usar o conceito de fila de prioridades. Uma fila de prioridades tem as seguintes operações

- EXTRACT-MIN(Q), remove o primeiro elemento da fila Q (de acordo com a prioridade)
- DECREASE-KEY(Q, v), atualiza a prioridade de v na fila Q

Baseado no exemplo de funcionamento e nas operações de fila de prioridades, vamos escrever o pseudo código do algoritmo.

Algoritmo de Prim

MST-PRIM(G, w, r)

```
1  for u ∈ G.V
2      u.chave = ∞
3      u.π = NIL
4  r.chave = 0
5  Q = G.V
6  while Q ≠ ∅
7      u = EXTRACT-MIN(Q)
8      for v ∈ G.Adj[u]
9          if v ∈ Q e w(u, v) < v.chave
10             v.π = u
11             v.chave = w(u, v)
```

Análise do tempo de execução

- A inicialização nas linhas de 1 a 3 tem tempo $O(V)$
- A inicialização da fila na linha 4 requer $|V|$ operações INSERT na fila (ou pode ser feito com uma única operação CREATE)
- A operação EXTRACT-MIN é executada uma vez para cada vértice
- O laço for das linhas 8 a 11 é executado no total $O(E)$ vezes
  - O teste de pertinência da linha 9 pode ser implementa em tempo constante usando um atributo no vértice
  - A atribuição na linha 11 envolve uma operação implícita de DECREASE-KEY
- Portanto, o tempo de execução total é $O(V) \times O(\text{INSERT}) + O(V) \times O(\text{EXTRACT-MIN}) + O(E) \times O(\text{DECREASE-KEY})$

Como podemos implementar uma fila de prioridades?

- Usando um arranjo simples
  - CREATE inicializa um arranjo com todos os vértices, $O(V)$
  - EXTRACT-MIN, faz uma busca linear no arranjo e remove o vértice com menor chave, tempo $O(V)$
  - DECREASE-KEY não faz nada, portanto $O(1)$
- Usando um Heap (Seção 6.5)
  - CREATE inicializa um arranjo com todos os vértices, $O(V)$
  - EXTRACT-MIN, faz uma busca linear no arranjo e remove o vértice com menor chave, tempo $O(\lg V)$
  - DECREASE-KEY não faz nada, portanto $O(\lg V)$
- Usando um Heap de Fibonacci (Capítulo 19)
  - Interesse teórico, na prática é superado pelo Heap

Análise do algoritmo de Prim

Tempos das operações

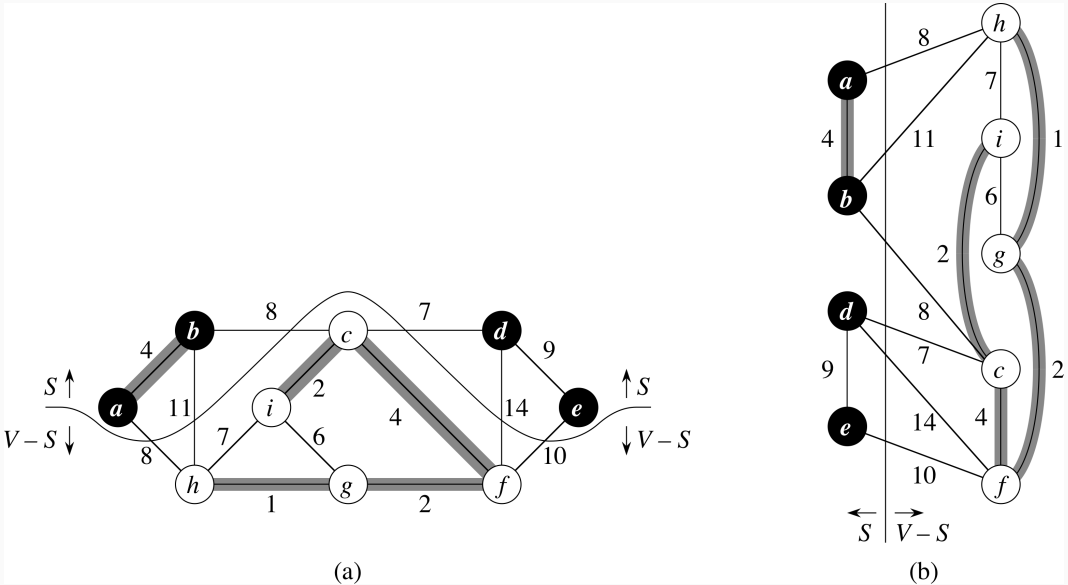
Operação	Arranjo	Heap	Heap de Fibonacci
CREATE	$O(V)$	$O(V)$	$O(V)$
EXTRACT-MIN	$O(V)$	$O(\lg V)$	$O(\lg V)$ (amortizado)
DECREASE-KEY	$O(1)$	$O(\lg V)$	$O(1)$ (amortizado)

Tempo de MST-PRIM

Tempo	Arranjo	Heap	Heap de Fibonacci
Grafo qualquer	$O(V^2)$	$O(E \lg V)$	$O(E + V \lg V)$
Grafo denso	$O(V^2)$	$O(V^2 \lg V)$	$O(V^2)$

Vamos fornecer uma regra para reconhecer arestas seguras, mas antes precisamos de algumas definições. Seja $G = (V, E)$ um grafo não orientado, $S \subset V$ e $A \subseteq E$

- Um **corte** $(S, V - S)$ de G é uma partição de V
- Uma aresta $(u, v) \in E$ **cruza** o corte $(S, V - S)$ se um de seus extremos está em S e o outro em $V - S$
- Um corte **respeita** o conjunto A de arestas se nenhuma aresta em A cruza o corte
- Uma aresta é uma **aresta leve** cruzando um corte se seu peso é o mínimo de qualquer aresta que cruza o corte


a
b
c
d
e
f
g
h
i
S
V − S
(a)
(b)

Teorema 23.1

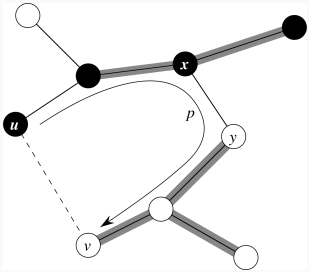
Seja $G = (V, E)$ um grafo conexo não orientado com uma função peso w de valor real definido em E. Seja A um subconjunto de E que está incluído em alguma árvore geradora mínima de G, seja $(S, V - S)$ qualquer corte de G que respeita A e seja (u, v) uma aresta leve cruzando $(S, V - S)$. Então a aresta (u, v) é segura para A.

Prova

Prova por construção.

Seja T uma AGM que contém A. T contém (u, v)?

- Se sim, é claro que (u, v) é segura para A
- Senão, construímos outra AGM T' que contém $A \cup \{(u, v)\}$



T é a AGM que inclui A.

$(S, V - S)$ é qualquer corte que respeita A. (u, v) é uma aresta leve cruzando o corte.

Temos que mostrar que (u, v) é segura para A construindo uma árvore T'.

A aresta (u, v) forma um ciclo com as arestas no caminho simples p de u para v em T. Tem alguma aresta do caminho p que cruza o corte? Sim! Porque u e v estão em lados opostos do corte. Seja (x, y) esta aresta.

A aresta (x, y) está em A? Não, pois o corte respeita A.

Como criar a árvore T'? Removendo a aresta (x, y) quebra T em dois componentes, adicionando (u, v) reconecta os componentes formando a nova árvore geradora $T' = (T - \{(x, y)\}) \cup \{(u, v)\}$.

Qual é a relação entre os pesos de (x, y) e (u, v)? Tanto (x, y) e (u, v) cruzam o corte $(S, V - S)$, como (u, v) é uma aresta leve para este corte, então $w(u, v) \leq w(x, y)$.

Qual é a relação entre os pesos de T e T'? Como $w(u, v) \leq w(x, y)$, então $w(T') = w(T) - w(x, y) + w(u, v) \leq w(T)$. Além disso, como T é uma AGM então $w(T) \leq w(T')$, então $w(T) = w(T')$. Então T' também é uma AGM.

Corolário 23.2

Seja $G = (V, E)$ um grafo conexo não orientado com uma função peso w de valor real definido em E. Seja A um subconjunto de E que está incluído em alguma árvore geradora mínima de G, e seja $C = (V_C, E_C)$ um componente conexo (árvore) na floresta $G_A = (V, A)$. Se (u, v) é uma aresta leve conectando C a algum outro componente em G_A, então (u, v) é segura para A.

Prova

Tomamos $S = V_C$ no Teorema 23.1.

23.1-3 Mostre que se uma aresta (u, v) está contido em alguma árvore geradora mínima, então ela é uma aresta leve cruzando algum corte do grafo.

Exercícios

Veja a lista de exercícios e algumas soluções na página da disciplina.

Referências

Thomas H. Cormen et al. Introduction to Algorithms. 3rd edition. Capítulo 23.