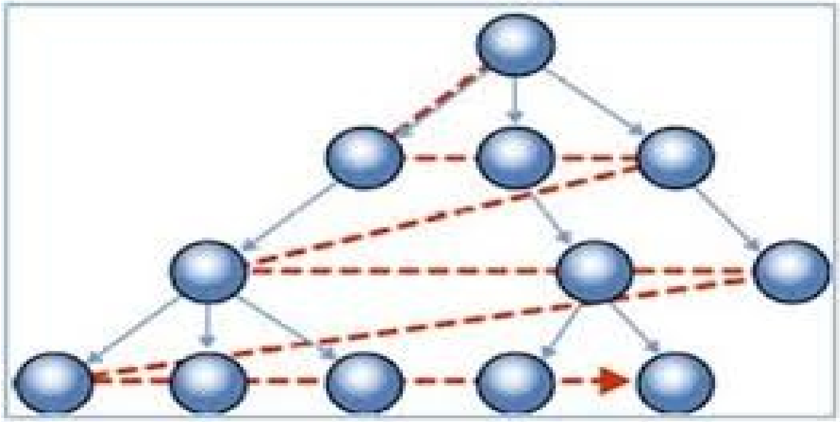
ACH2024

Aula 9 – Grafos:
Árvore Geradora Mínima
Algoritmo de Prim

Profa. Ariane Machado Lima

Aula anterior



Que estrutura de dados de suporte precisamos para gerenciar a ordem de vértices sendo processados?

FILA!

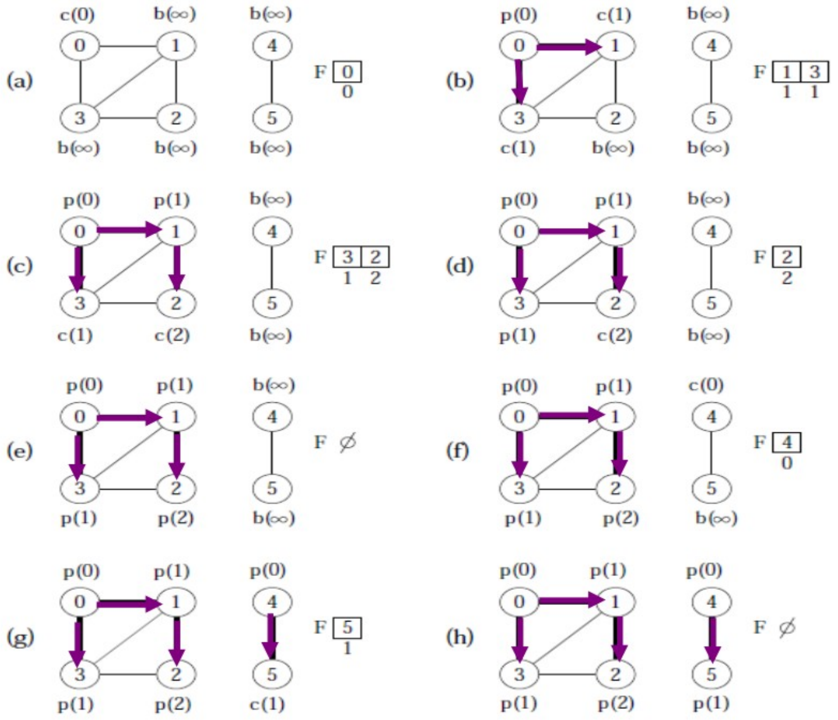
Busca em Largura

- Expande a fronteira entre vértices descobertos e não descobertos uniformemente através da largura da fronteira.
- O algoritmo descobre todos os vértices a uma distância k do vértice origem antes de descobrir qualquer vértice a uma distância $k + 1$.

Implementação

```
buscaEmLargura(grafo){
    Aloca vetores cor, antecessor, distancia com tamanho grafo->nrVertices
      Para cada vertice v
            cor[v] ← branco;  antecessor[v] ← -1; distancia[v] ← ∞;
      Para cada vertice v
            se cor[v] = branco
            visitaLargura(v, grafo, cor, antecessor, distancia);
}

visitaLargura(s, grafo, cor, antecessor, distancia){
    cor[s] ← cinza;
    distancia[s] ← 0;
    F ← Ø;
    insereFila(F, s);
    enquanto F ≠ Ø
          w ← removeFila(F)
          para cada vertice u da lista de adjacência de w
               se cor[u] = branco
                     cor[u] ← cinza;
                     antecessor[u] ← w;
                     distancia[u] ← distancia[w] + 1;
                     insereFila(F, u);
          cor[w] ← preto;
}
```



Complexidade: O(V+A)

Profa. Ariane Machado Lima

Aplicações

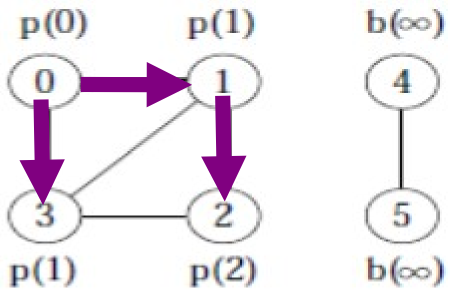
- Para encontrar os vértices vizinhos dentro de um certo raio (por exemplo, em sistemas de computação móvel, navegação GPS, etc)
- Pessoas a uma certa distância em uma rede social
- Coleta de lixo em memória (melhor localidade de referência do que se usar busca em profundidade)
- Caminhos mais curtos (NÃO é o mesmo que caminho mínimo)

Caminhos Mais Curtos

- A busca em largura obtém o **caminho mais curto** de u até v.
- O procedimento *VisitaBfs* contrói uma árvore de busca em largura que é armazenada na variável *Antecessor*.

Como uso a busca em largura?

Ex: caminho de 0 a 2



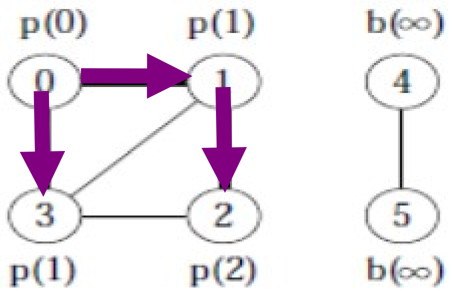
Caminhos Mais Curtos

- A busca em largura obtém o **caminho mais curto** de u até v.
- O procedimento *VisitaBfs* contrói uma árvore de busca em largura que é armazenada na variável *Antecessor*.

Como uso a busca em largura?

Chamo o visitaLargura em u (origem) até encontrar v

Ex: caminho de 0 a 2



Caminhos Mais Curtos

- A busca em largura obtém o **caminho mais curto** de u até v.
- O procedimento *VisitaBfs* contrói uma árvore de busca em largura que é armazenada na variável *Antecessor*.

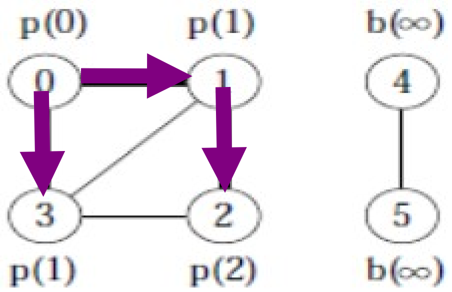
Imprimindo o caminho: (u tendo sido a origem da **visitaLargura**)

```
if (d[v] == ∞)
        printf ("Nao existe caminho de %d ate %d" , u, v) ;
else imprimeCaminho(u, v, antecessor);


void imprimeCaminho(int u, int v, int antecessor[])
{
```

DICA: usando recursão!

Ex: caminho de 0 a 2



Caminhos Mais Curtos

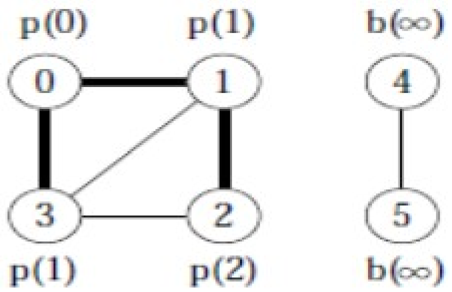
- A busca em largura obtém o **caminho mais curto** de u até v.
- O procedimento *VisitaBfs* contrói uma árvore de busca em largura que é armazenada na variável *Antecessor*.

Imprimindo o caminho: (u tendo sido a origem da **visitaLargura**)

```
if (d[v] == ∞)
        printf ("Nao existe caminho de %d ate %d" , u, v) ;
else imprimeCaminho(u, v, antecessor);

void imprimeCaminho(int u, int v, int antecessor[])
{
        if (u == v) { printf ( "%d " , u ); return; }
        else {
                imprimeCaminho(u, antecessor[v], antecessor);
                printf ( "%d " , v);
        }
}
```

Ex: caminho de 0 a 2



Exercício de programação

Implementem Busca em Largura **com nossa interface**

EP 1

Aula de hoje

Árvore geradora mínima *(Minimum Spanning Tree)*

Veremos dois algoritmos distintos para resolver esse problema:

- Algoritmo de Prim
- Algoritmo de Kruskal (na aula que vem)

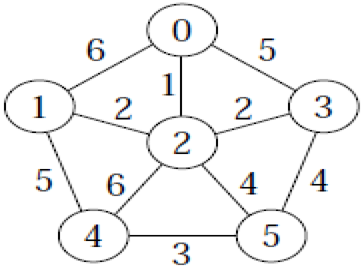
Árvore Geradora Mínima - Motivação

- Projeto de redes de comunicações conectando n localidades.
- Arranjo de $n - 1$ conexões, conectando duas localidades cada.
- Objetivo: dentre as possibilidades de conexões, achar a que usa menor quantidade de cabos.
- Modelagem:
  - $G = (V, A)$: grafo conectado, não direcionado.
  - V: conjunto de cidades.
  - A: conjunto de possíveis conexões
  - $p(u, v)$: peso da aresta $(u, v) \in A$, custo total de cabo para conectar u a v.
- Solução: encontrar um subconjunto $T \subseteq A$, acíclico, que conecta todos os vértices de G e cujo peso total $p(T) = \sum_{(u,v)\in T} p(u, v)$ é minimizado.

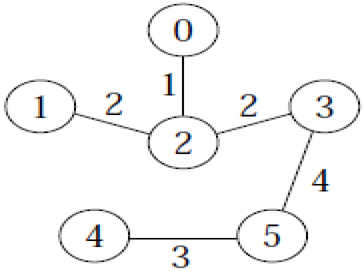
Árvore Geradora Mínima

- Como $G' = (V, T)$ é acíclico e conecta todos os vértices, T forma uma árvore chamada **árvore geradora** de G.
- O problema de obter a árvore T é conhecido como **árvore geradora mínima** (AGM).

Ex.: Árvore geradora mínima T cujo peso total é 12. T não é única, pode-se substituir a aresta $(3, 5)$ pela aresta $(2, 5)$ obtendo outra árvore geradora de custo 12.



(a)



(b)

AGM - Algoritmo Genérico

```
void GenericoAGM()
{  S = ∅;          No final será o conjunto de arestas que formam a AGM
   while(S não constitui uma árvore geradora mínima)
   { (u,v) = seleciona(A);
      if(aresta (u,v) é segura para S) S = S+ {(u,v)} }
   return S;
}
```

- Uma estratégia **gulosa** permite obter a AGM adicionando uma aresta de cada vez.
- Invariante: Antes de cada iteração, S é um subconjunto de uma árvore geradora mínima.
- A cada passo adicionamos a S uma aresta (u, v) que não viola o invariante. (u, v) é chamada de uma **aresta segura**.
- Dentro do **while**, S tem que ser um subconjunto próprio da AGM T, e assim tem que existir uma aresta $(u, v) \in T$ tal que $(u, v) \notin S$ e (u, v) é seguro para S.

isto é, assim que se entra no while,

AGM - Algoritmo Genérico

```
void GenericoAGM()
{   S = ∅;
    while(S não constitui uma árvore geradora mínima)
    { (u,v) = seleciona(A);
      if(aresta (u,v) é segura para S) S = S+ {(u,v)} }
    return S;
}
```

No final será o conjunto de arestas que formam a AGM

A principal diferença entre os algoritmos de Prim e de Kruskal é como eles definem mais especificamente o que é uma **aresta segura**.

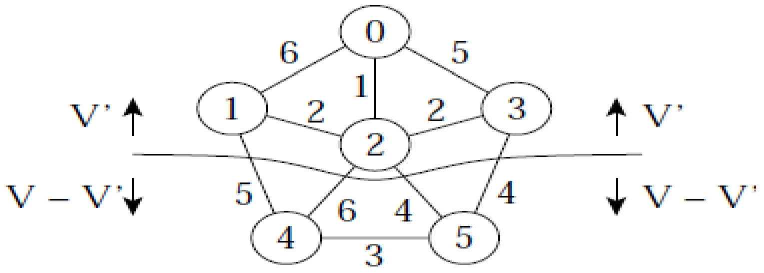
- Uma estratégia **gulosa** permite obter a AGM adicionando uma aresta de cada vez.
- Invariante: Antes de cada iteração, S é um subconjunto de uma árvore geradora mínima.
- A cada passo adicionamos a S uma aresta (u, v) que não viola o invariante. (u, v) é chamada de uma **aresta segura**.
- Dentro do **while**, S tem que ser um subconjunto próprio da AGM T, e assim tem que existir uma aresta $(u, v) \in T$ tal que $(u, v) \notin S$ e (u, v) é seguro para S.

isto é, assim que se entra no while,

Algoritmo de Prim

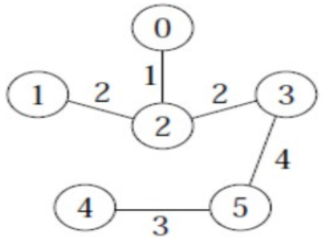
AGM - Definição de Corte

- Um **corte** $(V', V - V')$ de um grafo não direcionado $G = (V, A)$ é uma partição de V.
- Uma aresta $(u, v) \in A$ *cruza* o corte $(V', V - V')$ se um de seus vértices pertence a V' e o outro vértice pertence a $V - V'$.
- Um corte *respeita* um conjunto S de arestas se não existirem arestas em S que o cruzem.
- Uma aresta cruzando o corte que tenha custo mínimo sobre todas as arestas cruzando o corte é uma *aresta leve*.

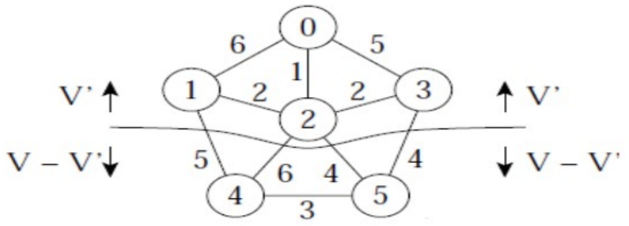


AGM - Teorema para Reconhecer Arestas Seguras

- Considere $G = (V, A)$ um grafo conectado, não direcionado, com pesos p sobre as arestas V.
- Considere S um subconjunto de A que está incluído em alguma AGM para G.
- Considere $(V', V - V')$ um corte qualquer que respeita S.
- Considere (u, v) uma aresta leve cruzando $(V', V - V')$.
- Satisfeitas essas condições, (u, v) é uma aresta segura para S.



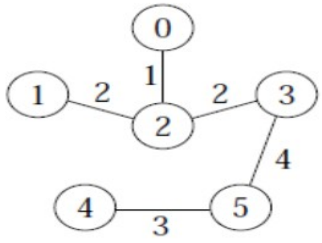
AGM



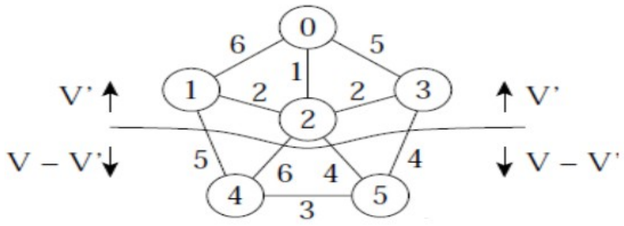
Ex: S = {(0,2), (1,2), (2,3)}
aresta(s) leve(s): ?

AGM - Teorema para Reconhecer Arestas Seguras

- Considere $G = (V, A)$ um grafo conectado, não direcionado, com pesos p sobre as arestas V.
- Considere S um subconjunto de A que está incluído em alguma AGM para G.
- Considere $(V', V - V')$ um corte qualquer que respeita S.
- Considere (u, v) uma aresta leve cruzando $(V', V - V')$.
- Satisfeitas essas condições, (u, v) é uma aresta segura para S.



AGM



Ex: S = {(0,2), (1,2), (2,3)}

aresta(s) leve(s): {(2,5), (3,4)}

Algoritmo de Prim para Obter Uma AGM

- O algoritmo de Prim para obter uma AGM pode ser derivado do algoritmo genérico.
- O subconjunto S forma uma única árvore, e a aresta segura adicionada a S é sempre uma aresta de peso mínimo conectando a árvore a um vértice que não esteja na árvore.
- A árvore começa por um vértice qualquer (no caso 0) e cresce até que "gere" todos os vértices em V.

No primeiro corte V' = {0},
e S é tal que respeita esse corte

- A cada passo, uma aresta leve é adicionada à árvore S, conectando S a um vértice de $G_S = (V, S)$.
- De acordo com o teorema anterior, quando o algoritmo termina, as arestas em S formam uma árvore geradora mínima.

```
void GenericoAGM()
{  S = ∅;  ⟶  No final será o conjunto de arestas que formam a AGM
   while(S não constitui uma árvore geradora mínima)
   { (u,v) = seleciona(A);
     if(aresta (u,v) é segura para S) S = S+ {(u,v)} }
   return S;
}
```

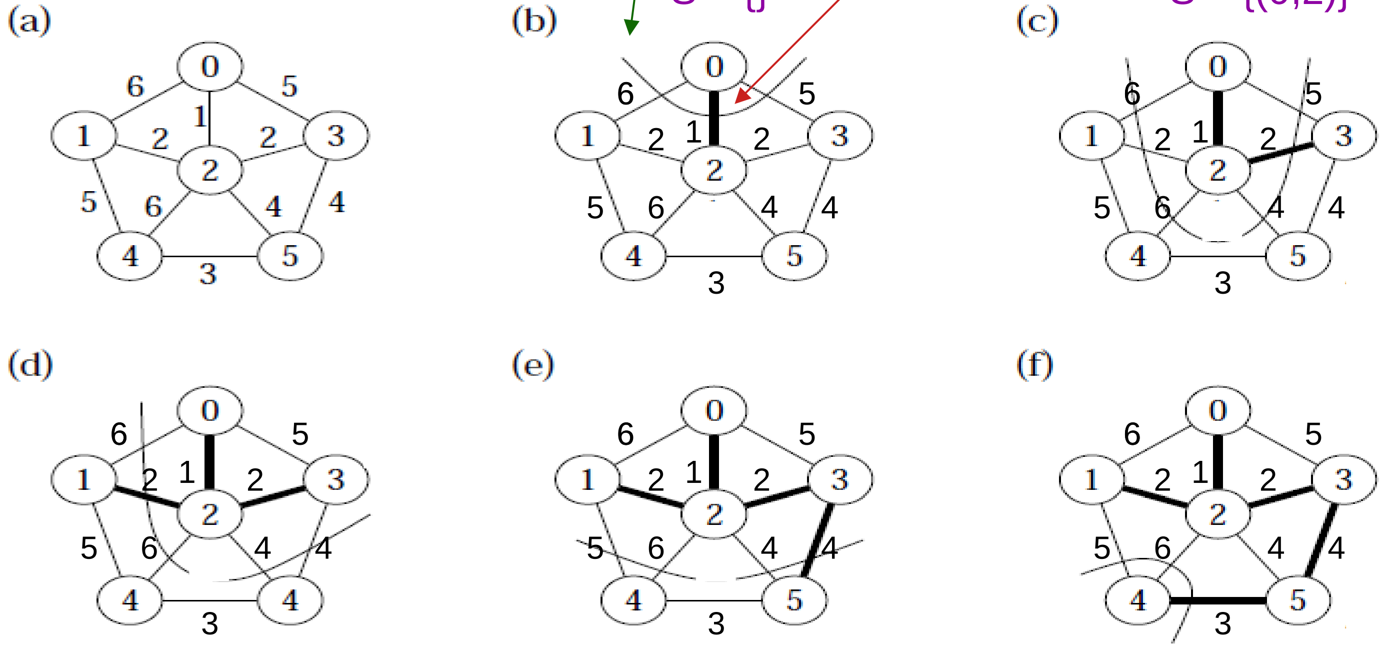
Corte separa vértices que pertencem à AGM sendo construída (ou seja, são extremos das arestas de S) dos demais vértices do grafo (inicialmente só o vértice 0)

Aresta leve a ser adicionada a S

Algoritmo de Prim: IDEIA GERAL

S = {}

S = {(0,2)}



S = {(0,2), (2,3)}

S = {(0,2), (2,3), (1,2)}

S = {(0,2), (2,3), (1,2), (3,5)}

S = {(0,2), (2,3), (1,2), (3,5),(4,5)}

Exemplo

Como gerar a AGM a partir do grafo?

Como seria a implementação desse algoritmo?

Algoritmo de Prim

Uma importante questão é: como selecionar essa aresta leve…

Como faço isso de modo eficiente?

Algoritmo de Prim

Uma importante questão é: como selecionar essa aresta leve…para isso:

Q: fila de prioridade contendo os vértices de G que ainda estão fora da AGM.

Qual deveria ser o vértice de maior prioridade?

Algoritmo de Prim

Uma importante questão é: como selecionar essa aresta leve…para isso:

Q: fila de prioridade contendo os vértices de G que ainda estão fora da AGM.

Qual deveria ser o vértice de maior prioridade?

Aquele que é a ponta de uma aresta leve naquele dado momento

Algoritmo de Prim

Uma importante questão é: como selecionar essa aresta leve…para isso:

Q: fila de prioridade contendo os vértices de G que ainda estão fora da AGM.

Qual deveria ser o vértice de maior prioridade?

Aquele que é a ponta de uma aresta leve naquele dado momento

Então o que deveria ser a chave desses vértices?

Algoritmo de Prim

Uma importante questão é: como selecionar essa aresta leve…para isso:

Q: fila de prioridade contendo os vértices de G que ainda estão fora da AGM.

Qual deveria ser o vértice de maior prioridade?

Aquele que é a ponta de uma aresta leve naquele dado momento

Então o que deveria ser a chave desses vértices?

key[v]: peso da aresta de menor peso que conecta o vértice v (que ainda não está na AGM parcial) a um vértice que já se encontra nela. Quanto menor key[v] maior a prioridade nesta fila

π[v]: (antecessor) vértice da outra ponta desta aresta (que já está na AGM)

Quando um vértice u sai de Q (porque tem o menor key),(u, π[u]) é a aresta leve que acaba de entrar

Minimum Spanning Tree

G: grafo

w: pesos das arestas

r: raiz

Referência: Cormen, Rivest, Leiserson: Introduction to Algorithms

```
MST-PRIM(G, w, r)
    for each u ∈ V[G]
        do key[u] ←
           π[u] ←
    key[r] ←
    Q ←
```

Inicialização

Minimum Spanning Tree

G: grafo
w: pesos das arestas
r: raiz

Referência: Cormen, Rivest, Leiserson: Introduction to Algorithms

```
MST-PRIM(G, w, r)
  for each u ∈ V[G]
       do key[u] ← ∞
          π[u] ← NIL
  key[r] ← 0
  Q ← V[G]
```

Inicialização

Minimum Spanning Tree

G: grafo

w: pesos das arestas

r: raiz

Referência: Cormen, Rivest, Leiserson: Introduction to Algorithms

```
MST-PRIM(G, w, r)
for each u ∈ V[G]
        do key[u] ← ∞
           π[u] ← NIL
key[r] ← 0
Q ← V[G]
while Q ≠ Ø
        do u ← EXTRACT-MIN(Q)
           for each v ∈ Adj[u]
                do if v ∈ Q and w(u, v) < key[v]
                      then π[v] ← u
                           key[v] ← w(u, v)
```

Isso significa que eu vou adicionar u à AGM parcial, com a aresta (u,Π[u]) .

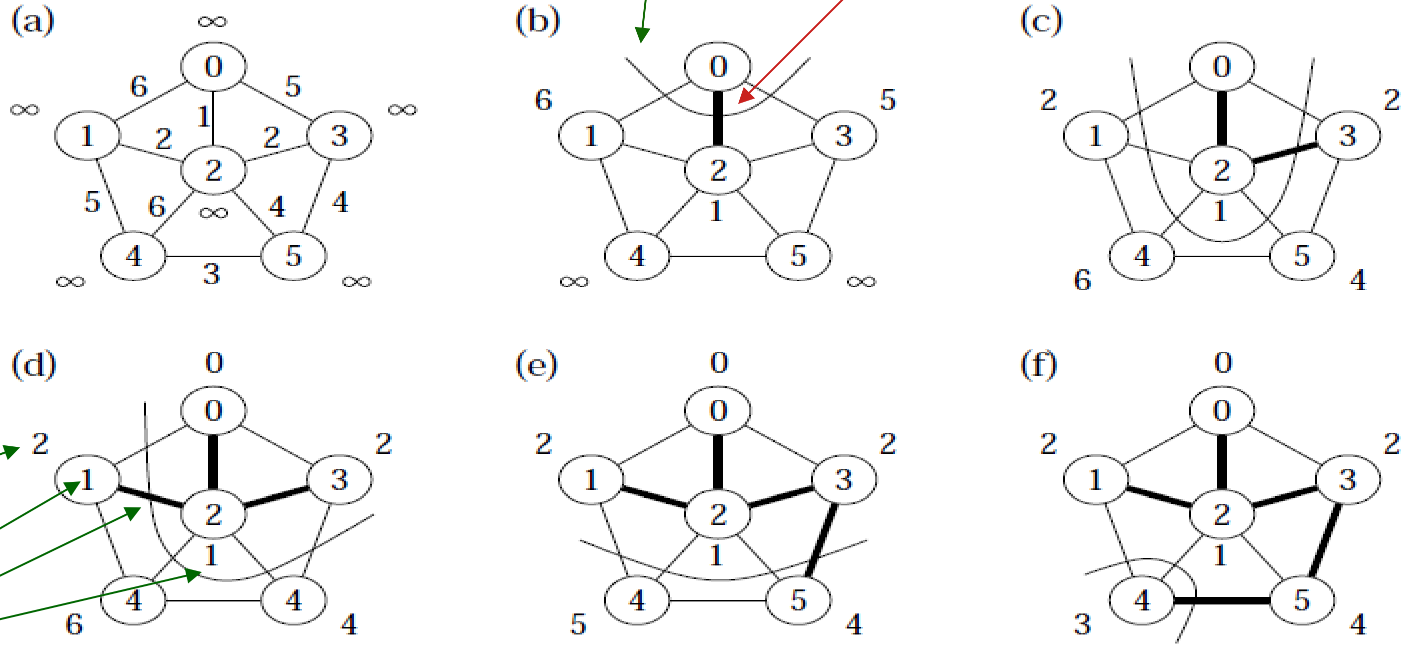
Já que u agora faz parte da AGM parcial, todas as arestas que conectam u a um vértice em Q (ie, que estão fora da AGM parcial) são “candidatas” a arestas leves, por isso preciso atualizar o key dos vértices v adjacentes a u de forma a considerar a aresta (u, v) como possível aresta leve

```
MST-PRIM(G, w, r)
  for each u ∈ V[G]
      do key[u] ← ∞
         π[u] ← NIL
  key[r] ← 0
  Q ← V[G]
  while Q ≠ Ø
      do u ← EXTRACT-MIN(Q)
         for each v ∈ Adj[u]
             do if v ∈ Q and w(u, v) < key[v]
                   then π[v] ← u
                        key[v] ← w(u, v)
```

Corte separa vértices que são extremos das arestas de S dos demais vértices do grafo (inicialmente só o vértice 0), **ou seja, vértices que estão fora de Q dos que estão dentro de Q**

Aresta leve a ser adicionada a S

Algoritmo de Prim: Exemplo



key
rótulo do vértice
Π (antecessor)
corte

Complexidade

Depende da implementação de Q….

```
MST-PRIM(G, w, r)
for each u ∈ V[G]
    do key[u] ← ∞
       π[u] ← NIL
key[r] ← 0
Q ← V[G]
while Q ≠ Ø
    do u ← EXTRACT-MIN(Q)
       for each v ∈ Adj[u]
           do if v ∈ Q and w(u, v) < key[v]
                 then π[v] ← u
                      key[v] ← w(u, v)
```

Complexidade

Depende da implementação de Q….

```
MST-PRIM(G, w, r)
1  for each u ∈ V[G]
2      do key[u] ← ∞
3         π[u] ← NIL
4  key[r] ← 0
5  Q ← V[G]
6  while Q ≠ Ø
7      do u ← EXTRACT-MIN(Q)
8         for each v ∈ Adj[u]
9             do if v ∈ Q and w(u, v) < key[v]
10                then π[v] ← u
11                     key[v] ← w(u, v)
```

Se Q for uma lista linear simples não ordenada:

Linhas 1 a 5: O(V)
Loop da linha 6: V vezes
Linha 7: O(V)
    Linha 6-7: O(V^2)
Linhas 8-11: O(A) no total (assumindo lista de adjacência)

Complexidade: O(V) + O(V^2) +O(A) = O(V^2)

Complexidade

Depende da implementação de Q….

```
MST-PRIM(G, w, r)
1   for each u ∈ V[G]
2       do key[u] ← ∞
3          π[u] ← NIL
4   key[r] ← 0
5   Q ← V[G]
6   while Q ≠ Ø
7       do u ← EXTRACT-MIN(Q)
8          for each v ∈ Adj[u]
9              do if v ∈ Q and w(u, v) < key[v]
10                    then π[v] ← u
11                         key[v] ← w(u, v)
```

Se Q for uma lista linear simples não ordenada:

Linhas 1 a 5: O(V)
Loop da linha 6: V vezes
Linha 7: O(V)
  Linha 6-7: $O(V^2)$
Linhas 8-11: O(A) no total (assumindo lista de adjacência)

Complexidade: $O(V) + O(V^2) + O(A) = O(V^2)$

Arg! Precisa melhorar….

Algoritmo de Prim

Uma árvore, inicialmente vazia, cresce até chegar a ser uma AGM

A cada passo um vértice é acrescentado a essa árvore

Referências

Livro do Cormen cap 23 (3ª ed) - AGM

– e sobre Heaps binários nas seções 6.1 a 6.3

– e sobre Heaps Fibonacci – cap 19

Livro do Ziviani cap 7 (seção 7.8)