Comentários da aula anterior

Componentes Fortemente Conectados (algoritmo)

1. Chama BuscaEmProfundidade (G) para obter os tempos de término (t[u], ou f[u]) para todos os vértices de G, isto é, enquanto existirem vértices ´brancos´ em G.
2. Obtém G^T.
   - $G = (V, E)$, $G^T = (V, E^T)$, onde $E^T = \{(u, v) : (v, u) \in E\}$
3. Chama BuscaEmProfundidade (G^T) em ordem decrescente de t[u] obtido no passo 1, enquanto existirem vértices u ´brancos´ em G^T.
4. Retorne todas as árvores obtidas no passo 3.

Comentários da aula anterior

Componentes Fortemente Conectados

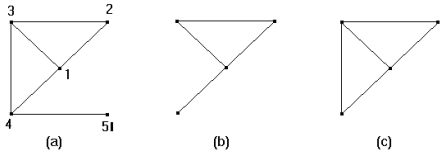
- Complexidade
  - Executa duas vezes o DFS
  - $O(|V| + |E|)$

Árvore Geradora Mínima

Originais da Profa. Rosane Minghin

Sub-grafo

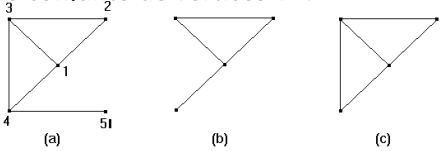
- Um sub-grafo **G2(V2, E2)** de um grafo **G1(V1, E1)** é um grafo tal que **V2** está contido em **V1** e **E2** está contido em **E1.**



b e c são subgrafos de a

Sub-grafo induzido

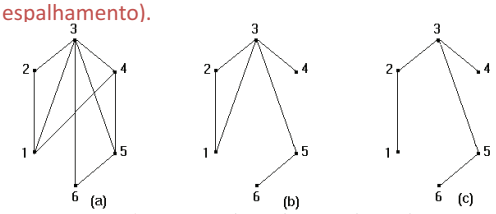
- Se o sub-grafo **G2** de **G1** satisfaz: para quaisquer **(v,w)** pertencentes a **V2**, se **(v,w)** pertence a **E1**, então **(v,w)** também pertence a **E2**. Dessa forma, **G2** é dito sub-grafo induzido pelo conjunto de vértices **V2.**



b e c são sub-grafos de a, mas apenas c é sub-grafo induzido

Sub-grafo gerador

- **Sub-grafo Gerador** (ou sub-grafo de espalhamento) de um grafo **G1(V1, E1)** é um sub-grafo **G2(V2, E2)** de **G1** tal que **V1=V2**. Quando o sub-grafo gerador é uma árvore, ele recebe o nome de árvore geradora (ou de espalhamento).



b e c são sub-grafos geradores de a
c é árvore geradora de a e b

Sub-grafo gerador de custo mínimo

- Formalmente...
- Dado um grafo não orientado G(V, E)
  - onde *w*: E→$\Re^+$ define os custos das arestas.
  - queremos encontrar um sub-grafo gerador conexo T de G tal que, para todo sub-grafo gerador conexo T' de G.

$$\sum_{e \in T} w(e) \leq \sum_{e \in T'} w(e)$$

Árvore geradora mínima (MST)

- Claramente o problema só tem solução se G é conexo.
- A partir de agora, assumimos G conexo.
- Também não é difícil ver que a solução para esse problema será sempre uma árvore...
  - Basta notar que T não terá ciclos pois, poderíamos obter um outro sub-grafo T', ainda conexo e com custo menor que o de T, removendo o ciclo!

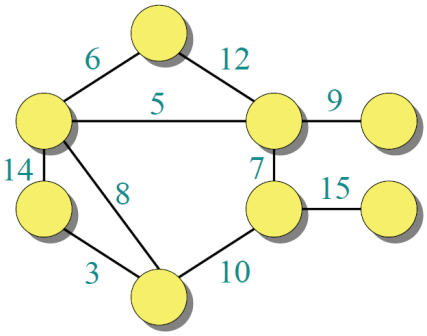
Árvore geradora mínima (MST)

- Árvore Geradora (*Spanning Tree*) de um grafo G é um sub-grafo de G que contém todos os seus vértices e, ainda, é uma árvore.
- Árvore Geradora Mínima (*Minimum Spanning Tree*, MST) é a árvore geradora de um grafo valorado cuja soma dos pesos associados às arestas é mínimo, *i.e.*, é uma árvore geradora de custo mínimo.

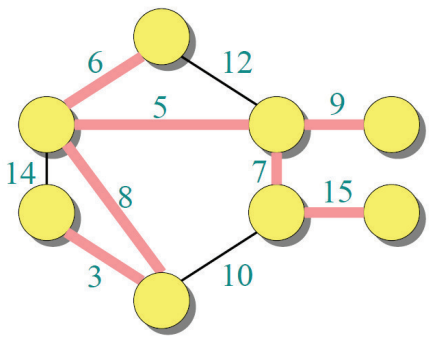
Porque é um problema interessante?

- Suponha que queremos construir estradas para interligar n cidades
  - Cada estrada direta entre as cidades i e j tem um custo associado.
  - Nem todas as cidades precisam ser ligadas diretamente, desde que todas sejam acessíveis...
- Como determinar eficientemente quais estradas devem ser construídas de forma a minimizar o custo total de interligação das cidades?

Exemplo de uma MST (MIT OpenCourseWare)



Exemplo de uma MST (MIT OpenCourseWare)



Árvore geradora mínima (MST)

- Como encontrar a árvore geradora mínima de um grafo G?
  - Algoritmo Genérico;
  - Algoritmo de Prim;
  - Algoritmo de Kruskal.

Algoritmo Genérico

```
Generic-MST(G)
A = ∅
While A não define uma spanning tree
   encontre uma aresta (u,v) segura para A
   A = A ∪ {(u,v)}
Return A
```

A – conjunto de arestas.

G conexo, não direcionado, ponderado.

Abordagem 'gulosa' -> MST cresce uma aresta por vez.

Aresta é 'segura' se mantém a condição de que, antes de cada iteração, A é um sub-conjunto de alguma MST.

Algoritmo de Prim

```
{ Gera uma Minimum Spanning Tree do
  grafo ponderado G – Algoritmo de Prim}

Prim-MST (G)
   Escolha um vértice s para iniciar a árvore
   enquanto "Há vértices que não estão na árvore"
      Selecione a aresta com menor peso adjacente
      a um vértice pertencente à árvore e a outro
      não pertencente à árvore

      Insira a aresta selecionada e o respectivo
      vértice na árvore
   fim-enquanto
```

•Inicia em um determinado vértice e gera a árvore, uma aresta por vez

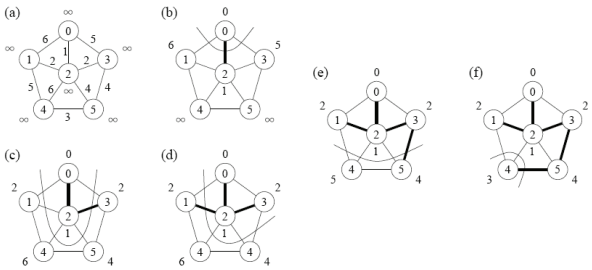
Algoritmo Prim (MIT OpenCourseWare)

- Idéia: mantém V-A como uma fila de prioridade Q. A chave de cada vértice em Q é o peso do vértice mais leve conectando-o a um vértice em A.

$Q \leftarrow V$
$key[v] \leftarrow \infty$ for all $v \in V$
$key[s] \leftarrow 0$ for some arbitrary $s \in V$
while $Q \neq \varnothing$
  do $u \leftarrow$ EXTRACT-MIN(Q)
    for each $v \in Adj[u]$
      do if $v \in Q$ and $w(u, v) < key[v]$
        then $key[v] \leftarrow w(u, v)$ ▷ DECREASE-KEY
          $\pi[v] \leftarrow u$

- No final, $\{(v,\pi[v])\}$ formam a MST

Exemplo Prim



Exemplo 2 Prim

- Quadro...

Análise do Prim (MIT OpenCourseWare)

$O(V)$ total:
- $Q \leftarrow V$
- $key[v] \leftarrow \infty$ for all $v \in V$
- $key[s] \leftarrow 0$ for some arbitrary $s \in V$

$|V|$ times:
- **while** $Q \neq \varnothing$
- **do** $u \leftarrow$ EXTRACT-MIN(Q)

$degree(u)$ times:
- **for** each $v \in Adj[u]$
- **do if** $v \in Q$ and $w(u, v) < key[v]$
- **then** $key[v] \leftarrow w(u, v)$
- $\pi[v] \leftarrow u$

- Tempo: $O(V)T_{\text{Extract-Min}} + O(E)T_{\text{Decrease-Key}}$

Análise do Prim

- Maneira mais eficiente de determinar a aresta de menor peso a partir de um dado vértice
  - Mantem todas as arestas que ainda não estão na árvore em uma fila de prioridade (*heap*).
  - Prioridade é dada à aresta de menor peso adjacente a um vértice na árvore e outro fora dela.

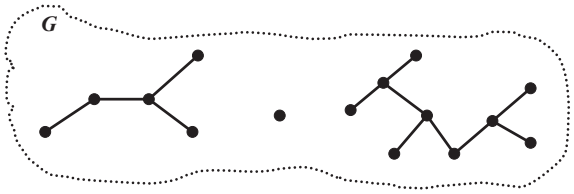
Análise do Prim (MIT OpenCourseWare)

- Tempo: $O(V)T_{\text{Extract-Min}} + O(E)T_{\text{Decrease-Key}}$

Q	$T_{\text{EXTRACT-MIN}}$	$T_{\text{DECREASE-KEY}}$	Total
array	$O(V)$	$O(1)$	$O(V^2)$
binary heap	$O(\lg V)$	$O(\lg V)$	$O(E \lg V)$

Algoritmo de Kruskal

- Floresta
  - Uma Floresta é um conjunto de árvores.



Algoritmo de Kruskal

- Não inicia em nenhum vértice em particular
  - Considera se cada aresta individualmente pode ou não pertencer à árvore geradora mínima, analisando-as em ordem crescente de custo.
  - As árvores que compõem a floresta são identificadas pelos conjuntos S_i, que contém os vértices que a compõem.

Algoritmo de Kruskal

- Basicamente o algoritmo consiste de
  - Estado inicial: corresponde a floresta formada por |V| árvores triviais (um só vértice cada), *i.e.*, A = ∅.
  - "Incluir em A todas as arestas de E em ordem crescente de peso, rejeitando, contudo, cada uma que forma ciclos com as arestas já em A."
  - Ao final do processo, o conjunto A contém a solução do problema, *i.e.*, a MST.
  - Pode ser interpretado como sendo a construção de uma árvore geradora a partir de uma floresta.

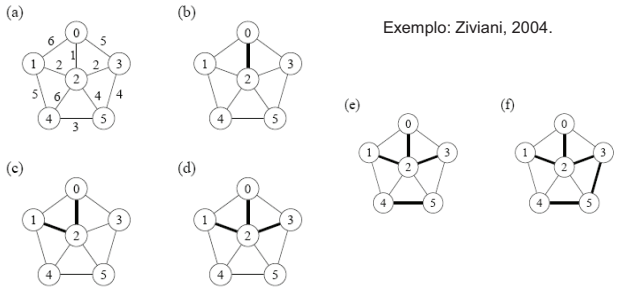
Algoritmo de Kruskal

- Sejam S_1 e S_2 duas árvores conectadas por (u, v):
  - Como (u, v) tem de ser uma aresta leve conectando S_1 com alguma outra árvore, (u, v) é uma aresta segura para A.
  - Testa se uma dada aresta a ser adicionada ao conjunto solução A forma um ciclo.
  - É guloso porque, a cada passo, ele adiciona à floresta uma aresta de menor peso.

Algoritmo de Kruskal (Cormen, 2001)

```
MST-KRUSKAL(G, w)
1  A ← ∅
2  for each vertex v ∈ V[G]
3      do MAKE-SET(v)
4  sort the edges of E into nondecreasing order by weight w
5  for each edge (u, v) ∈ E, taken in nondecreasing order by weight
6      do if FIND-SET(u) ≠ FIND-SET(v)
7          then A ← A ∪ {(u, v)}
8               UNION(u, v)
9  return A
```


Algoritmo de Kruskal



Exemplo: Ziviani, 2004.

Algoritmo de Kruskal (Cormen, 2001)

- Complexidade:

```
MST-KRUSKAL(G, w)
1  A ← ∅
2  for each vertex v ∈ V[G]
3      do MAKE-SET(v)
4  sort the edges of E into nondecreasing order by weight w
5  for each edge (u, v) ∈ E, taken in nondecreasing order by weight
6      do if FIND-SET(u) ≠ FIND-SET(v)
7          then A ← A ∪ {(u, v)}
8               UNION(u, v)
9  return A
```


Algoritmo de Kruskal (Cormen, 2001)

- Complexidade:

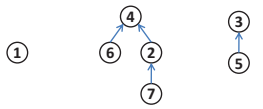
```
                MST-KRUSKAL(G, w)
O(V)        →   1  A ← ∅
                2  for each vertex v ∈ V[G]
                3      do MAKE-SET(v)
O(E lg E)   →   4  sort the edges of E into nondecreasing order by weight w
                5  for each edge (u, v) ∈ E, taken in nondecreasing order by weight
O(E).(T_FIND-SET →  6      do if FIND-SET(u) ≠ FIND-SET(v)
+T_UNION)       7          then A ← A ∪ {(u, v)}
                8               UNION(u, v)
                9  return A
```


Algoritmo de Kruskal

- Complexidade:
  - Depende de como FIND-SET e UNION forem implementados.
  - Esse teste garante que a inclusão de *e* em E_T não introduz um ciclo.
  - Trata conjuntos disjuntos: maneira eficiente de verificar se uma dada aresta forma um ciclo.
  - Teste $S_p \cap S_q = \varnothing$

Algoritmo de Kruskal

- Estrutura de dados Union-Find (Skiena, 2008)



1	2	3	4	5	6	7
0	4	0	0	3	4	2

FIND-SET(i): encontrar a raiz da árvore contendo i, caminhando pelos ponteiros para o pai, até que não há mais para onde ir. Retornar o rótulo da raiz.

UNION(i,j): Ligar a raiz da árvore contendo i a raiz da árvore contendo j.

Algoritmo de Kruskal

- Estrutura de dados Union-Find (Skiena, 2008)
  - Minimizar o tamanho da árvore:
  - UNION(i,j): decidir qual das duas raizes será a nova raíz → a da maior árvore.
  - Aumento da altura da árvore em função do número de nós
    - 2 árvores de 1 nó → árvore de altura 2
    - Unir uma árvore de 1 nó não aumenta a altura
    - 2 árvores de altura 2 → árvore de altura 3
    - É necessário dobrar o número de nós para aumentar a altura em 1
  - FIND-SET e UNION → $O(\lg n)$

Algoritmo de Kruskal (Cormen, 2001)

- Complexidade:

MST-KRUSKAL(G, w)

$O(V)$ →	1	$A \leftarrow \emptyset$
	2	**for** each vertex $v \in V[G]$
	3	**do** MAKE-SET(v)
$O(E \lg E)$ →	4	sort the edges of E into nondecreasing order by weight w
	5	**for** each edge $(u, v) \in E$, taken in nondecreasing order by weight
$O(E).(T_{\text{FIND-SET}}$ →	6	**do if** FIND-SET(u) $\neq$ FIND-SET(v)
$+T_{\text{UNION}})$	7	**then** $A \leftarrow A \cup \{(u, v)\}$
	8	UNION(u, v)
	9	**return** A

$= O(E \lg E)$
se FIND-SET e UNION forem implementados com Union-Find

Algoritmo de Kruskal

- Complexidade: $O(E \lg E)$
  - Trata conjuntos disjuntos: maneira eficiente de verificar se uma dada aresta forma um ciclo. Utiliza estruturas dinâmicas. Sempre unindo árvores disjuntas, árvores são obtidas.

Caminhos mínimos

Profa. Debora Medeiros

Baseado nos slides da Profa. Rosane Minghin

Caminho mínimo

- Problema: encontrar o caminho de menor custo (ou o menor caminho) entre dois vértices em um grafo valorado
  - Algoritmo de Djikstra;
  - Algoritmo de Floyd-Warshall.

Caminho mínimo

- Grafo dirigido G(V, E) com função peso w: E→ℜ que mapeia as arestas em pesos.
- Peso (custo) do caminho p = <v0, v1, ..., vk>

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

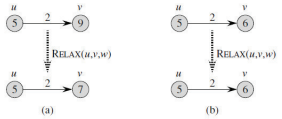
- Custo do caminho de menor peso entre u e v:

$$\delta(u,v) = \begin{cases} \min\{w(p) : u \overset{p}{\Rightarrow} v\} \; se \exists \; rota \; de \; u \; p/ \; v \\ \infty \; cc \end{cases}$$

Dijkstra (Cormen, 2001)

INITIALIZE-SINGLE-SOURCE(G, s)
1 **for** each vertex $v \in V[G]$
2 **do** $d[v] \leftarrow \infty$
3 $\pi[v] \leftarrow$ NIL
4 $d[s] \leftarrow 0$

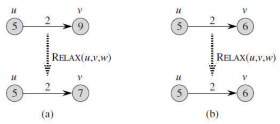
RELAX(u, v, w)
1 **if** $d[v] > d[u] + w(u, v)$
2 **then** $d[v] \leftarrow d[u] + w(u, v)$
3 $\pi[v] \leftarrow u$



Algoritmo Dijkstra (Cormen, 2001)

INITIALIZE-SINGLE-SOURCE(G, s)
1 **for** each vertex $v \in V[G]$
2 **do** $d[v] \leftarrow \infty$
3 $\pi[v] \leftarrow$ NIL
4 $d[s] \leftarrow 0$

RELAX(u, v, w)
1 **if** $d[v] > d[u] + w(u, v)$
2 **then** $d[v] \leftarrow d[u] + w(u, v)$
3 $\pi[v] \leftarrow u$



DIJKSTRA(G, w, s)
1 INITIALIZE-SINGLE-SOURCE(G, s)
2 $S \leftarrow \emptyset$
3 $Q \leftarrow V[G]$
4 **while** $Q \neq \emptyset$
5 **do** $u \leftarrow$ EXTRACT-MIN(Q)
6 $S \leftarrow S \cup \{u\}$
7 **for** each vertex $v \in Adj[u]$
8 **do** RELAX(u, v, w)

Exemplo Dijkstra (Cormen, 2001)