



Resumo

prova 2

Grafos



Materia:

+ caminho critico ✓
+ excentricidade ✓

Conectividade em Grafos Direcionados

- Fracamente Conexo: Existe um caminho entre quaisquer dois vértices se as arestas forem consideradas não direcionadas. (*Busca por Largura / Profundidade*)
- Semi-Fortemente Conexo: Para qualquer par de vértices u e v, existe um caminho unidirecional, ou de u para v ou de v para u. (*Dij Kruskal / Bellman-Ford*)
- Fortemente Conexo: Existe um caminho em ambas as direções entre quaisquer dois vértices do grafo. (*Kosaraju*)

Algoritmo de Kosaraju



- Encontra componentes fortemente conexos.
- Utiliza duas buscas em profundidade (DFS): uma para obter a ordem topológica e uma para cada componente fortemente conexo no grafo transposto.

Algoritmos de Caminho Mínimo (Shortest Path)



- Dijkstra: Encontra o caminho mais curto de um vértice para todos os outros (não funciona com arestas negativas).
- Bellman-Ford: Resolve o problema de caminho mínimo, mesmo com arestas negativas, mas com maior complexidade.

Algoritmos de Fluxo (Mínimo e Máximo)



- Fluxo Min-Max: Encontra o caminho que maximiza o fluxo enquanto minimiza gargalos.
- Fluxo Max-Min: Encontra o caminho que minimiza o fluxo máximo.

Caminho Máximo (Longest Path)



- Ordenação Topológica por largura (ideia de pegar as bases do grafo): Utilizada para encontrar o caminho mais longo em DAGs (grafos direcionados acíclicos).

Árvores Geradoras Mínimas / Máximas (Minimum/Maximum Spanning Trees)



- Prim: Expande uma árvore começando por um vértice e adicionando arestas de menor custo.
- Kruskal: Adiciona as arestas de menor custo sem formar ciclos, até conectar todos os vértices.

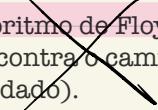
Algoritmo de Johnson



- Usa Dijkstra repetidamente para todos os pares de vértices, mesmo com arestas negativas, ao transformar pesos negativos em não negativos.

Algoritmo de Floyd-Warshall

- Encontra o caminho mínimo entre todos os pares de vértices. (Ainda a ser estudado).



Conectividade em Grafos Direcionados

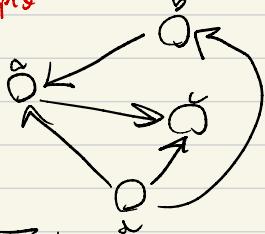
► Fracamente conexo: Quando existe um caminho entre quaisquer dois vértices ao ignorar a direção dos arcos. Para identificar esse tipo de grafo, podemos:

- 1- criar o grafo S subjacente de G (em que todos os arcos de G estão presentes e tratados como não-direcionais) vértice
- 2- realizar uma busca em profundidade a partir de um v qualquer
- 3- se todos os vértices forem visitados, o grafo é Fracamente Conexo.

► Semi-Fortemente conexo: Quando, para qualquer par de vértices (u, v) existe um caminho uni direcional (de u p/ v ou de v p/ u). Podemos identificá-lo com os passos:

- 1- aplicamos uma busca a partir de um vértice. Se todos os vértices forem alcançáveis, o grafo é semi-fortemente conexo

* um exemplo:



existem:
a → c ab db
a → c d → a ac dc
b → a d → b ad cbX
não é
semi. fort. conexo.

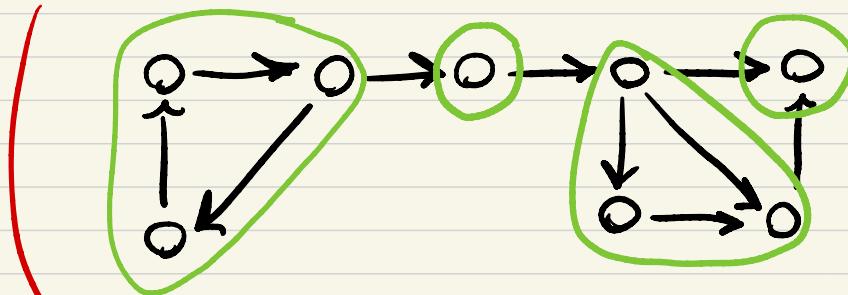
► Fortemente conexo: Quando existe um caminho em ambos os direções para qualquer par de vértices. Identificamos esse grafo por meio do algoritmo Kosaraju com algumas adições, da seguinte forma:

- 1- realizamos uma DFS no grafo marcando os tempos de início e fim da visitação.
- 2- geramos o grafo transposto T de G .
- 3- aplicamos uma nova DFS a partir de T na ordem não-crescente dos tempos de término.
- 4- se todo o grafo for visitado nessa única DFS em T , o grafo é fortemente conexo.

Kosaraju (SCC)

→ componentes fortemente conexos.

↳ algoritmo usado para encontrar componentes fortemente conexos em um grafo (cada excentricidade)



Coyunto máximo de vértices tal que p/ todo par de vértice $U \in V$, existe um caminho de U p/ V e de V p/ U .

↳ Passo a passo do algoritmo:

1- busca em profundidade (DFS) em $G(V, E)$ gravando os tempos de início e fim da visitação

2- gerar o grafo transposto T de G .

3- na ordem mais-crescente (inversa) dos tempos de término, realizar uma busca em profundidade no grafo T .

4- para cada conjunto de vértices visitados em T , marque-o como um componente fortemente conexo em G .

→ (caminho mínimo)

Algoritmos de Shortest-Path

↳ algoritmos para encontrar o menor caminho em um grafo.

Dijkstra: Encontra o menor caminho de um vértice para todos os outros.

↳ Não funciona com **arestas negativas** (já que o Dijkstra não resiste aos já precedentes, ele não considera que existe um caminho mais curto através de uma aresta negativa, pois já marcou o nó como finalizado quando encontrou o caminho mais curto até ele).

↳ É um algoritmo de **busca gulosa** (greedy algorithm), já que ele realiza uma decisão local repetidamente com objetivo de encontrar a melhor solução. → é inversível)

↳ Algoritmo:

Dijkstra (G, l):

$$\forall v \in V \rightarrow d[v] = \infty$$

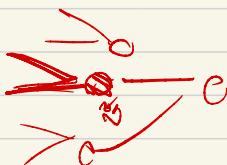
$$d[u] = 0$$

$$\text{visitados} = \{u\}$$

seleciona, dentre todos os dists, a menor.

(atribui a v^*) $\forall v \in V \rightarrow d[v] = \min(d[v^*] + w(v^*, v), d[v])$

adicionamos o nó com a menor dist. acumulada aos visitados



para todos os

nós que ainda não foram visitados

atualiza a distância de cada vizinho só se a nova dist. for menor que a dist. já registrada.

$$d[v] = \min(d[v^*] + w(v^*, v), d[v])$$

peso
 ↓
 distância de u até v^*
 + o peso da aresta (v^*, v)

dist. acumulada
 de u para v

↳ Passo a Passo:

1- initialize as distâncias como ∞ e o do nó inicial como 0, e crie o conjunto de nós visitados vazio.

2- entre todos os nós ainda não visitados, escolha o nó com a menor distância acumulada e atribuimos à variável v^* . Atribuimos v^* aos visitados.

3- para cada vizinho de v^* , calcule a nova distância potencial passando por v^* . Se essa nova distância for menor que a distância registrada anteriormente para o vizinho, atualize a distância do vizinho.

4- repita o passo de escolher o vértice de menor distância acumulada até que não existam mais nós não visitados.

Definição da
Eva Tardos

Dijkstra's Algorithm (G, ℓ)

Let S be the set of explored nodes

For each $u \in S$, we store a distance $d(u)$

Initially $S = \{s\}$ and $d(s) = 0$

While $S \neq V$

Select a node $v \notin S$ with at least one edge from S for which

$d'(v) = \min_{e=(u,v): u \in S} d(u) + \ell_e$ is as small as possible

Add v to S and define $d(v) = d'(v)$

EndWhile

Bellman - Ford: Pode ser implementado em grafos com pesos negativos (só não com ciclos negativos, ou seja, ciclos em que a soma de seus pesos é negativa).

- ↳ Tem uma complexidade maior → enquanto o Dijkstra usa uma fila de prioridade p/ selecionar os vrs a serem relaxados, o Bellman and Ford simplesmente relaxa todos os arestas
- ↳ $O(V \times E)$

↳ Algoritmo:

Bellman-Ford($G, pesos$):

$$\forall v \in V \rightarrow d[v] = \infty$$

visitados = {}

$$d[u] = 0$$

FOR EACH v in V :

\forall aresta $= (e, v) \in E$ (relaxamento) que a soma da dist $[e]$ + peso da aresta, encontramos um caminho + curto para v passando por e .

se $d[v] > d[e] + peso(e, v)$

$d[v] \leftarrow d[e] + peso(e, v)$

$v^* \leftarrow e$

atualiza o predecessor.

→ não escolhi a menor p/ relaxar, relaxa todos

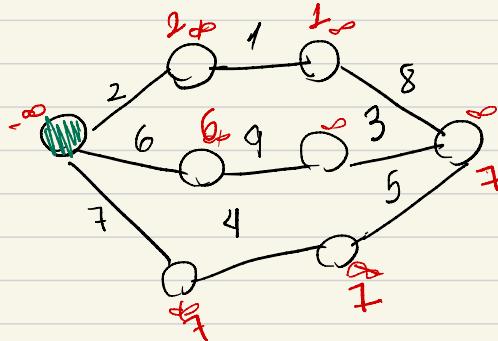
```

procedure shortest-paths( $G, l, s$ )
Input:   Directed graph  $G = (V, E)$ ;
         edge lengths  $\{l_e : e \in E\}$  with no negative cycles;
         vertex  $s \in V$ 
Output:  For all vertices  $u$  reachable from  $s$ ,  $dist(u)$  is set
         to the distance from  $s$  to  $u$ .
for all  $u \in V$ :
   $dist(u) = \infty$ 
   $prev(u) = \text{nil}$ 
 $dist(s) = 0$ 
repeat  $|V| - 1$  times:
  for all  $e \in E$ :
    update( $e$ )
procedure update( $(u, v) \in E$ )
 $dist(v) = \min\{dist(v), dist(u) + l(u, v)\}$ 

```

Algoritmos de Fluxo:

Min-max: encontra o caminho que maximiza o fluxo enquanto minimiza os gorgulhos. (Menor dos maiores)



→ maximiza os recursos p/ os de custo ao longo do caminho.

Max-min: encontra o caminho que minimiza os fluxos máximos. (Maior dos menores)

Max-Min ($G(V, E), G, W$)

$$F \leftarrow \{\}$$

valores $\leftarrow \infty$ p/ a origem, $-\infty$ p/ os demais

$$\bar{F} \leftarrow F \cup \{\text{origem}, \text{valores(origem)}\}$$

while $F \neq E$:

$$u \leftarrow \text{extratoMaiorValor}(F)$$

for each v de u :

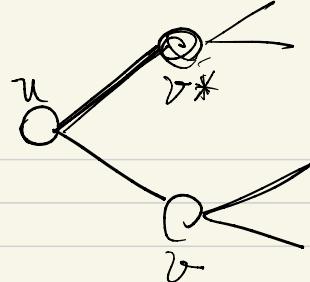
$$\text{menor_custo} \leftarrow \min(\text{valores}[u], W(u, v))$$

se menor_custo > valores[v]:

$$\text{valores}[v] \leftarrow \text{menor_custo}$$

$$F \leftarrow \bar{F} \cup \{v, \text{valores}[v]\}$$

Max-Min



Dijkstra (G, l):

$$\forall v \in V \rightarrow d[v] = \infty$$

$d[u] = \infty$

$$\text{visitados} = \{u\}$$

while $V \neq \text{visitados}$:

$$v^* = \arg \min \{d[w]\} \text{ para } v \in V \setminus \text{visitados}$$

$$\text{visitados} = \text{visitados} \cup \{v^*\}$$

$$\forall v \in V \rightarrow d[v] = \min(d[v], d[v^*] + w(v^*, v))$$

visit v

$$\max(d[v], \min(d[u], w(u, v)))$$

Algoritmo Widest Path

Max-Min

↳ Usado p/ encontrar o caminho máximo de valor em um grafo não direcionado $G = (V, E)$.

↳ é uma adaptação do Dijkstra

↳ Passo a Passo:

1- inicializaremos um conjunto $P = \emptyset$ e um vetor W que apresenta tinf para V e $-\infty$ para todo vértice pertencente a $P \setminus V$.
ponto + forte N

conjunto de V que ainda não foram visitados

ainda não sabemos o melhor caminho p/ eles.

2- iniciamos uma busca em largura partindo de V usando uma fila de prioridade ordenada pelo vetor W de modo que o vértice com peso mais alto em W seja removido primeiro (utlizando expandindo o caminho que tem a maior largura possível).

3- enquanto $P \neq \emptyset$

↳ removemos o vértice w c/ maior valor de W da fila d.p.e de P .
↳ ✓ vizinho u de w em P , altere o valor de u para $\max(\min(W[w], l(w, u)), W[u])$, e adicione u na fila de prioridade.



armazena o valor máximo entre o caminho que estamos expandindo e o valor de $W[u]$,

$$\max(\min(W[w], l(w, u)), W[u])$$

gongozo { calcula o menor valor entre:
 $W[w] =$ largura do caminho + larg. atígora p/ w .
 $l(w, u) =$ o peso da aresta entre w e o vizinho u

Minimum Spanning Trees:

↳ algoritmos para gerar a MST (uma árvore (sem ciclos) que conecta todos os vértices do grafo original com o menor peso total possível). ou seja
(dos custos)

Prim: análise local (verifica os custos mais vindos vindo das vizinhanças).

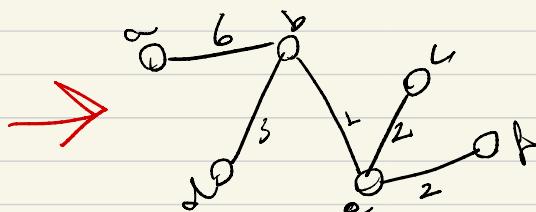
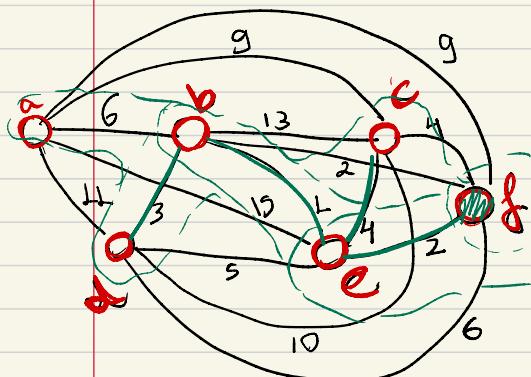
↳ Passo a Passo: \rightarrow Grafo (V, E)

1- começa em um vértice do conjunto S , que pertence ao V , que inicialmente contém v . (v é o vértice inicial que pertence a V).

2- para adicionarmos uma aresta $\{w, u\}$, w deve pertencer a S , e u deve pertencer a $V \setminus S$, de modo que essa aresta $\{w, u\}$ pertencente a E deve ser a aresta de menor peso no conjunto de corte de S .

se esses requisitos forem cumpridos, adicionamos $\{w, u\}$ ao conjunto Solução e w à S .

3- o algoritmo se encerra quando $|S| = |V|$



Kruskal: análise global (analisar o grafo todo). Ao invés de um só conjunto S , temos vários.

e custos R^+

↳ Passo a Passo: (Grafo conexo, não-direcionado, ponderado $G(V, E)$)

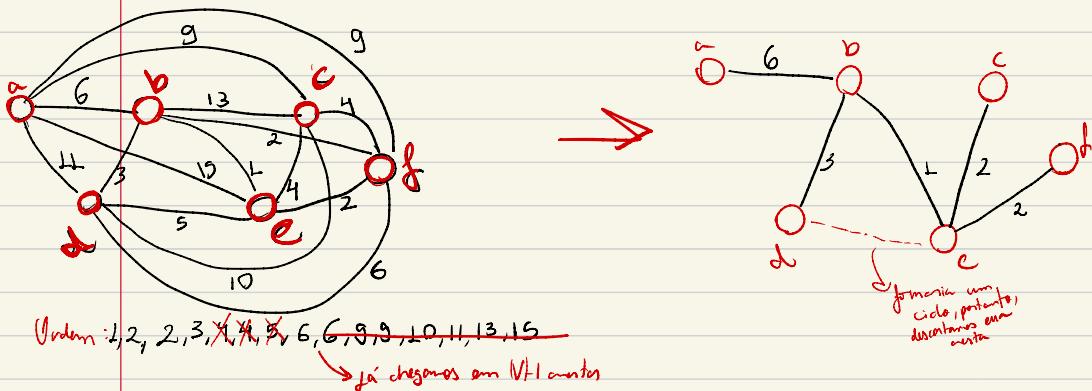
1- ordenamos todos os custos por peso em ordem não-decrescente

2- criamos um grafo nulo T com todos $v \in V$.

3- seguindo a ordem não-decrescente de pesos, adicionaremos os custos de G em T somente se, ao adicioná-la uma aresta, o grafo T continue aciclico (não pode gerar ciclos em T).

→ se os dois pertencerem ao mesmo componente conexo

4- assim que o n.º de arestas de T for igual a $|V| - 1$, temos a resposta da MST.



Cut Property:

(4.17) Assume that all edge costs are distinct. Let S be any subset of nodes that is neither empty nor equal to all of V , and let edge $e = (v, w)$ be the minimum-cost edge with one end in S and the other in $V - S$. Then every minimum spanning tree contains the edge e .

Algoritmo de Johnson:

1- técnica p/ encontrar o menor caminho de m para m vértices do Grafo.

↳ Passo a Passo:

1- transformamos o grafo adicionando um novo vértice S^* que tem uma aresta de peso 0 p/ todo v de V .

2- aplicamos o Bellman-Ford nesse grafo alterado a partir do vértice S p/ calcular a menor distância $d[v]$ de s p/ cada v de V .

3- se o Bellman-Ford encontra ciclos negativos, o algoritmo se encerra.

4- com os valores $d[v]$, ajustamos o peso original de cada aresta $\{u, v\}$ no G original com a fórmula $\text{peso}'(u, v) = \text{peso}(u, v) + d[u] - d[v]$. Essas alterações garantem que todos os pesos sejam não-negativos (para usar Dijkstra).

5- aplicamos o Dijkstra p/ cada v em V do novo Grafo.

6- p/ obter as distâncias corretas no grafo original, reinvertimos a reponderença dos pesos com $d[u, v] = d'[u, v] - d[u] + d[v]$.

Maior caminho (em m de arestas)

0) $G(V, E) \rightarrow (b, w)$, $w: E \rightarrow \{\pm\}$

1) $w' = -w$

2) aplicamos o Dijkstra seguindo uma ordem

3) resposta de Dijkstra é "invertida".

→ acha a base
- começamos no v
que tem o maior fator
transitivo.

Excentricidade:

Excentricidade(G, u):

$$d[u] = 0$$

$$S = \emptyset$$

for each $v \in V$

$$d[v] = \infty$$

while $S \neq \emptyset$

$$v^* = \text{extrain} \min(S)$$

for each $v^* \in \text{vizinhos}(v^*)$

$$d[v^*] = \min(d[v^*], d[v^*])$$

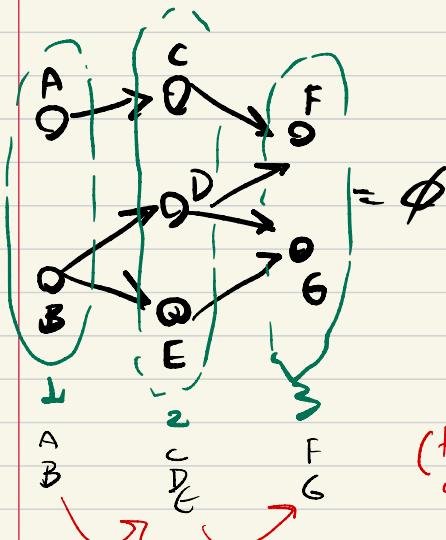
excentricidade = max ($d[v]$ for $v \in V$)

semelhante ao Dijkstra, mas
sem utilizar pesos (contando
os vértices).

Caminho Crítico

↳ usado p/ alcançar o vértice mais distante no menor tempo possível.

vértices v/
gan de entrada 0



► identificamos a base

► removemos a base

► fazemos isso até o grafo f. ficar vazio
(quanto tiver componentes de base diferentes
adiciona-os em um caminho).

► isso garante a ordem topológica.

B A DCE GF

(tem vérticos faltantes)
de encontrar ele

