Grafos e seus Algoritmos

Prof. Tiago Massoni

Engenharia da Computação

Poli - UPE

Grafo

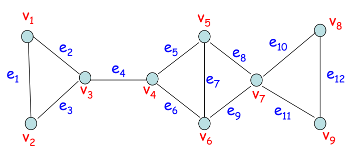
- Conjunto de pontos (vértices ou nós) conectados por linhas (arestas)

G = (V, E)

vértices arestas

$V = \{v_1, v_2, ..., v_n\}$ $|V| = n$
$E = \{e_1, e_2, ..., e_m\}$ $|E| = m$

Grafo



$V = \{v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8, v_9\}$
$n = 9$
$E = \{e_1, e_2, e_3, e_4, ..., e_9, e_{10}, e_{11}, e_{12}\}$
$m = 12$

Conceitos de Grafos

- Cada aresta é definida por um par não-ordenado de nós, que são suas extremidades:

$e = (v_i , v_j)$

$e = (u, v)$ ➡ u e v são adjacentes
e é incidente a v
e é incidente a u

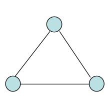
$d(v)$: grau do nó v = número de arestas incidentes a v (nós adjacentes)

$d(v_1) = d(v_2) = d(v_8) = d(v_9) = 2$
$d(v_3) = d(v_4) = d(v_5) = d(v_6) = 3$
$d(v_7) = 4$

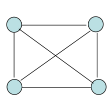
$d(v) = 0$ ➡ vértice isolado

Conceitos de Grafos

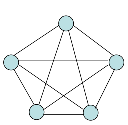
K_n: grafo completo com n nós
número de arestas: $n(n-1)/2$



K_3



K_4



K_5

Grafo k-regular: todos os nós têm grau k

K_n é (n-1)-regular

Conceitos de Grafos

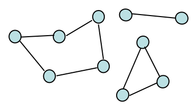
- Caminho de v_i a v_j
  - Seqüência P de vértices e arestas alternados, tais que cada aresta é incidente ao nó anterior e ao nó posterior
  - Se $v_i = v_j$, P é um ciclo ou circuito
- Caminho simples: cada vértice aparece exatamente uma vez
- Comprimento de um caminho: número de arestas
- Caminhos disjuntos em vértices/arestas: não têm vértices/arestas em comum

Conceitos de Grafos

Vértices v_i e v_j são conectados se existe um caminho de v_i a v_j.

Dois vértices v_i e v_j estão na mesma componente conexa se existe um caminho entre eles.

Um grafo é conexo se possui uma única componente conexa, ou seja, se existe um caminho entre qualquer par de nós



É conexo?

Problema importante: determinar se um grafo é conexo ou não

Grafo orientado

Grafo no qual são associadas direções aos seus arcos

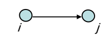
$G = (V, A)$

$V = \{v_1, \dots, v_n\}$

$A = \{a_1, \dots, a_m\}$

$a = (i, j) \in V \times V$ par ordenado

Início ou origem Fim ou destino

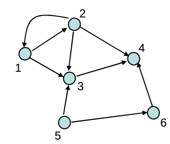


j é sucessor de i

i é predecessor de j

$d^-(i)$ = grau de entrada de i = número de predecessores de i

$d^+(i)$ = grau de saída de i = número de sucessores de i

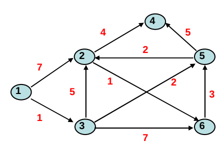


$d^-(1) = 1$ $d^+(4) = 0$

$d^-(4) = 3$ $d^+(6) = 1$

Grafos com pesos

- Arestas podem ter um outro componente – peso (custo)
  - (v,w,p)
- No caso serve para medir alguma grandeza em relação ao caminho entre dois vértices



Exemplos de grafos

- Sistema de aeroportos
- Fluxo de tráfego
- Grade de disciplinas de um curso

Representação de grafos

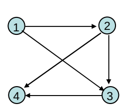
Matriz de adjacência:

Uma linha para cada nó

Uma coluna para cada nó

$a_{ij} = 1 \rightarrow (i, j) \in A$

$a_{ij} = 0 \rightarrow (i, j) \notin A$

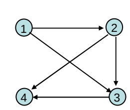


$$A_{n \times n} = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

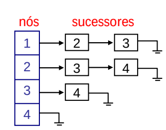
Representação de grafos

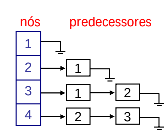
Lista de adjacências

Cada nó aponta para a lista de seus sucessores (ou nós adjacentes)



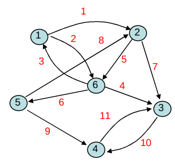
n nós
m arestas $\Rightarrow$ n +m posições

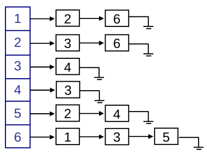




Exemplos

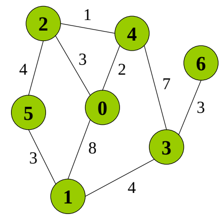
$$A = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \end{bmatrix}$$





Matriz de Adjacências com Pesos

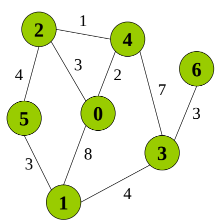
- Para cada par, o custo da aresta se são vizinhos, infinito em caso contrário
- Neste caso, não-direcionado

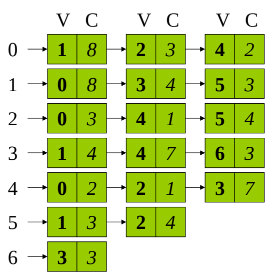


	0	1	2	3	4	5	6
0	∞	8	3	∞	2	∞	∞
1	8	∞	∞	4	∞	3	∞
2	3	∞	∞	∞	1	4	∞
3	∞	4	∞	∞	7	∞	3
4	2	∞	1	7	∞	∞	∞
5	∞	3	4	∞	∞	∞	∞
6	∞	∞	∞	3	∞	∞	∞

Lista de Incidências com Pesos

- Cada nó da lista contém o vértice vizinho e o custo da aresta (sem infinitos)
- Não-direcionado



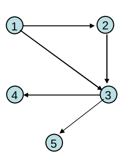


Representação de grafos

- Matriz de adjacências
  - Requisito de espaço: ordem de $|V|^2$
  - Bom para grafos densos
- Lista de adjacências
  - Requisito de espaço: $O(|E|+|V|)$
  - Bom para grafos esparsos
  - Mais usado em geral

Ordenação Topológica

- Ordenação nos vértices de forma que todas as arestas vão da esquerda para a direita
  - Se (v,w), então w vem depois de v na ordem
  - Várias ordenações são possíveis
  - Se houver ciclo, não é possível fazer



Ordem: 1, 2, 3, 4, 5

Ordem: 1, 2, 3, 5, 4

Ordenação topológica 1

```
void topSort() throws CycleException{
 Vertex v,w;

 for (int c=0; c < NUMVERTICES; c++){
   v = findVertexOfInDegreeZero();
   if (v == null) throw new CycleException();

   v.topOrder = c;

   for each w adjacent to v
     w.inDegree--;
 }
}
```

- inDegree: d-(v)
- Por causa da função chamada em cada laço, custo é $O(|V|^2)$
- Podemos melhorar isso registrando em cada passo os vértices que tem indegree = 0, e procurando apenas neste registro
  - Fila!

Ordenação topológica 2

```
void topSort() throws CycleException{
 Queue q = new Queue();
 int c = 0; Vertex v,w;

 for each vertex v
   if (v.inDegree == 0) q.enqueue(v);

 while (!q.isEmpty()){
   v = q.dequeue();
   v.topOrder = ++c;
   for each w adjacent to v
     if (--w.inDegree==0) q.enqueue(w);
 }

 if (counter != NUM_VERTICES)
   throw new CycleException();
}
```

- Se lista de adjacência for usada, custo é O(|E|+|V|)

Caminho mais Curto

Dados: grafo $G=(V,A)$ orientado e
distância c_{ij} associada à aresta $(i,j) \in A$.
Problema: Obter o caminho mais curto entre dois nós *s* e *t*

A "distância" pode ter diversas interpretações dependendo da aplicação: custos, distâncias, consumo de combustível, etc.

Exemplo 1: Dado um mapa rodoviário, determinar a rota mais curta de uma cidade a outra
(rota mais rápida,
rota com menor consumo de combustível, ...)

Menor caminho

- Dado um grafo G = (V,E) e um vértice distinto s, achar o caminho mais curto de s para todos os outros vértices em G
- Algoritmo mais conhecido: Dijkstra

Dijkstra

- Algoritmo capaz de descobrir o menor caminho existente entre um nó de origem e um nó de destino
- É um algoritmo completo
  - caso exista o menor caminho entre os nós de origem e destino, o algoritmo sempre* encontra o menor caminho
- Ótimo
  - não há nenhum caminho menor do que o encontrado
- e eficiente

*O algoritmo de Dijkstra não é capaz de lidar com arestas de peso negativo

Dijkstra

- O algoritmo de Dijkstra identifica, a partir de um vértice do grafo, qual é o custo mínimo entre esse vértice e todos os outros
- No início, o conjunto S contém somente esse vértice, chamado origem
  - A cada passo, selecionamos no conjunto de vértices sobrando, o que é o mais perto da origem
- Depois atualizamos, para cada vértice sobrando, a sua distância em relação à origem
  - Se passando pelo novo vértice acrescentado, a distância fica menor, é essa nova distância que será memorizada

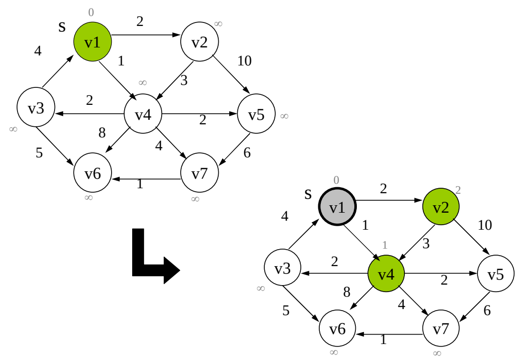
Dijkstra

```
class Vertex
{
  LinkedList adj;     //vert. adjacentes
  boolean known;  //jah visitado
  int dist;   //Custo deste em relacao a s
  Vertex path; //vert. Anterior no menor cam.
...
}
```

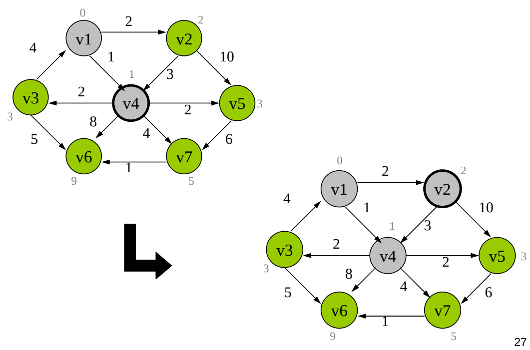
Dijkstra

```
void dijkstra(Vertex s){
 Vertex v,w;
 for each w em G
   w.dist = INFINITY;
 s.dist=0;

 while(true){
   v= vertice ainda não visitado de menor cam.
   if (v==null) break; //acabou o algoritmo
   v.known = true;

   for each w adjacent to v
     if (!w.known)
       if (v.dist + costVtoW < w.dist){
          w.dist = v.dist + costVtoW;
          w.path = v;
       }
 }
}
```

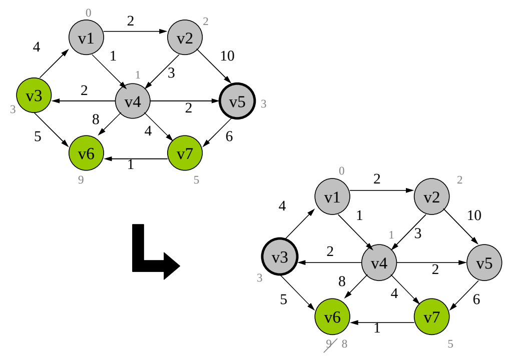
Dijkstra



Dijkstra



Dijkstra



Dijkstra