1

Algoritmos em Grafos: Pesquisa e Ordenação Topológica

R. Rossetti, A.P. Rocha, J. Pascoal Faria
CAL, MIEIC, FEUP
Março de 2014

Algoritmos em Grafos: Introdução • CAL – MIEIC/FEUP, Março de 2014

2

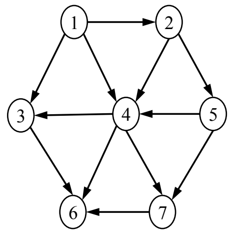
Representação

Algoritmos em Grafos: Introdução • CAL – MIEIC/FEUP, Março de 2014

Matriz de adjacências

1	
2	
3	
4	
5	
6	
7	

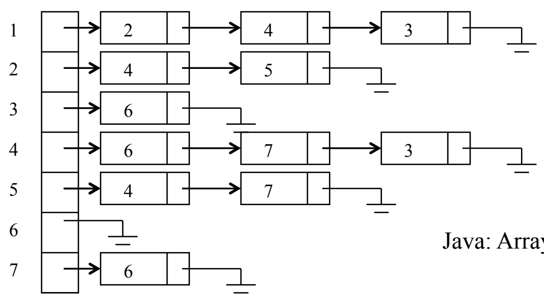
G_1



- a[u][v] = 1 sse (u, v) $\in$ E
- elementos da matriz podem ser os pesos (0 – não há aresta)
- grafo não dirigido – matriz simétrica
- apropriada para grafos densos
  - 3000 cruzamentos x 12 000 troços de ruas (4 por cruzamento) = 9 000 000 de elementos na matriz!

Lista de adjacências

- Estrutura típica para grafos esparsos
  - para cada vértice, mantém-se a lista dos vértices adjacentes
  - vector de cabeças de lista, indexado pelos vértices
  - espaço é O(|E| + |V|)
  - pesquisa de adjacentes em tempo proporcional ao número destes
- Grafo não dirigido: lista com dobro do espaço



Java: ArrayList<LinkedList<Integer>>

Representação

- Normalmente precisamos de guardar informação adicional em cada vértice e em cada aresta (nome, peso, etc.), pelo que se opta por uma representação mais complexa, como por exemplo:

```
class Graph {
   ArrayList<Vertex> vertexSet;
}

class Vertex {
   String name;
   LinkedList<Edge> adj; //arestas a sair deste
vértice
}

class Edge {
   Vertex dest;
   double weight;
}
```


Pesquisa em profundidade e em largura

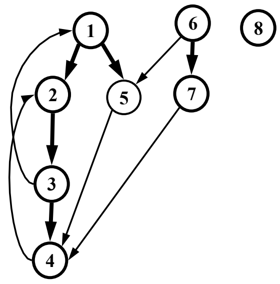
Pesquisa profundidade (depth-first search)

- Arestas são exploradas a partir vértice v mais recentemente descoberto que ainda tenha arestas a sair dele.
- Quando todas as arestas de v forem exploradas, retorna para explorar arestas que saíram do vértice a partir do qual v foi descoberto.
- Se se mantiverem vértices por descobrir, um deles é seleccionado como a nova fonte e o processo de pesquisa continua a partir daí.
- Todo o processo é repetido até todos os vértices serem descobertos.

Pesquisa em profundidade

```
class Graph { …
 void dfs() {
   for (Vertex v : vertexSet)
      v.visited = false;
   for (Vertex v : vertexSet)
       if (! v.visited)
          dfs(v);
          //v passa a ser a raiz de uma
          //árvore na floresta dfs.
 }
  void dfs( Vertex v ) {
   v.visited = true;
     // fazer qualquer coisa c/ v aqui
   for(Edge e : v.adj)
      if( ! e.dest.visited )
         dfs( e.dest );
     // ou aqui
   }
}
```

Exemplo em grafo dirigido (vértices numerados por ordem de visita e dispostos por profundidade de recursão):



Arestas a traço forte: árvore de expansão em profundidade

Na DFS podem ser produzidas várias árvores, porque a pesquisa pode ser repetida a partir de várias fontes (ao contrário da BFS que só produz uma). O conjunto das várias árvores é conhecido como Floresta DFS.

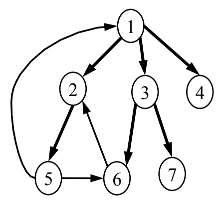
Pesquisa largura (breadth-first search)

- Dado um vértice s, explora-se sistematicamente o grafo descobrindo todos os vértices a que se pode chegar a partir de s. Só depois é que passa para outro vértice.
- Calcula a distância (número mais pequeno de arestas) de s para qualquer outro vértice.
- Produz uma árvore BFS com raiz s. Para qualquer vértice v a que se possa chegar a partir de s, o caminho na árvore BFS é o mais curto no grafo.

Pesquisa em largura

```
1: void bfs( Vertex v ) {
2:    Queue<Vertex> q = new Queue<Vertex>();
3:    q.add(v); // insere root fila (FIFO)
4:    v.discovered = true;
5:    while (q.size() > 0) {
6:       Vertex v = q.remove();
// Procura vértices adjacentes ao obtido na
   linha 6 e adiciona-os à fila
7:    for(Edge e : v.adj) {
8:          Vertex w = e.dest;
9:       if( ! w.discovered) {
10:            w.discovered = true;
11:           q.add(w);
12:          }
13:      }
// marca vértice v como explorado
14:       v.explored = true;
15:    }
16: }
```

Exemplo em grafo dirigido (vértices numerados por ordem de visita e dispostos por níveis de distância ao vértice inicial):



Arestas a traço forte: árvore de expansão em largura

BFS é um dos métodos mais simples e é o arquétipo para muitos algoritmos importantes de grafos. Exemplo: Prim's Minimum-Spanning Tree e Dijkstra Single-Source Shortest-paths.

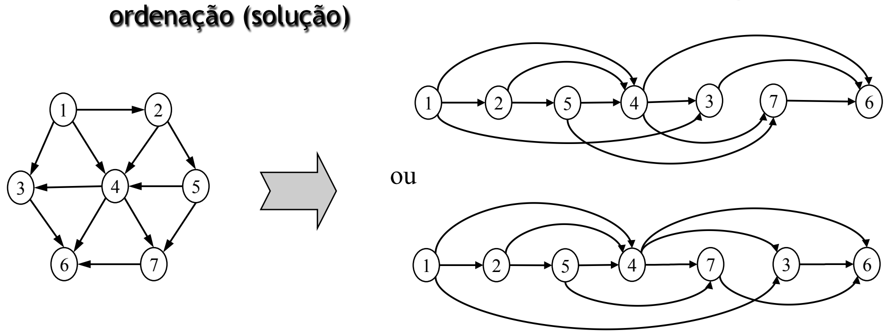
Funciona em grafos dirigidos e não dirigidos.

Algoritmos em Grafos: Introdução • CAL – MIEIC/FEUP, Março de 2014

Ordenação topológica

Problema

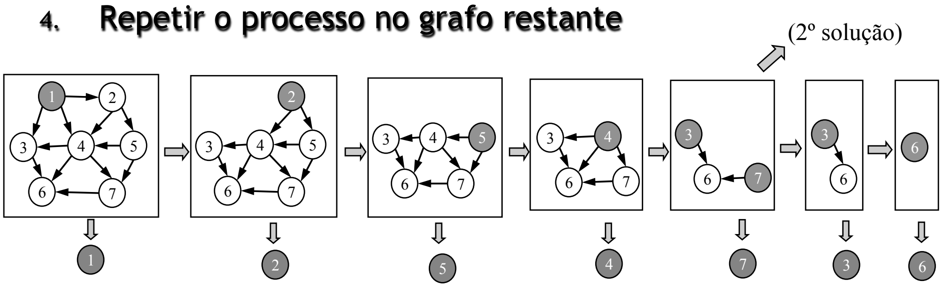
- Ordenação linear dos vértices de um DAG (Grafo Acíclico Dirigido) tal que, se existe uma aresta (v, w) no grafo, então v aparece antes de w
  - **Impossível se o grafo for cíclico**
  - **Não é necessariamente única. Pode existir mais do que uma ordenação (solução)**



13

Método básico

1. Descobrir um vértice sem arestas de chegada (indegree=0)
2. Imprimir o vértice
3. Eliminá-lo e às arestas que dele saem
4. Repetir o processo no grafo restante



14

Algoritmo de ordenação topológica

- Refinamento do método básico:
  - simular eliminação actualizando indegree dos vértices adjacentes
  - memorizar numa estrutura auxiliar vértices por imprimir c/ indegree=0
- Dados de entrada:
  - V - conjunto de vértices
  - adj(v) - conjunto (ou lista) de vértices adjacentes a cada vértice v
    - ou conj. de arestas que saem de v, que por sua vez indicam vértices adj.
- Dados de saída
  - T – sequência (ou lista) dos vértices por ordem topológica
    - ou numTop(v) – número atribuído a cada vértice v por ordem topológica
- Dados temporários
  - indegree(v) - nº de arestas que chegam a v, partindo de vértices por visitar
  - C - conjunto de vértices por visitar cujo indegree é 0 (candidatos)

Algoritmo de ordenação topológica

- Método:

```
1.  for each v∈V do indegree(v) ← 0
2.  for each v∈V do for each w∈adj(v) do indegree(w) ← indegree(w) +1
3.  C ← { } // Pode ser uma fila (queue)
4.  for each v∈V do if indegree(v) = 0 then C ← C ∪ {v}
5.  T ← [ ] // Pode ser uma lista  LinkedList
6.  while C ≠ {} do
7.     v ← remove-one(C)
8.     T ← T concatenado-com [v]
9.     for each w ∈ adj(v) do
10.        indegree(w) ← indegree(w) - 1
11.        if indegree(w) = 0 then C ← C ∪ {w}
12. if |T| ≠ |V| then Fail("O grafo tem ciclos")
```


Análise do algoritmo

- As diferentes escolhas do próximo vértice no ponto 7 dão as diferentes soluções possíveis
- Se as inserções e eliminações em C forem efectuadas em tempo constante (usando por exemplo uma fila FIFO), algoritmo pode ser executado em tempo O(|V|+|E|)
  - o corpo do ciclo de actualização do indegree (passos 9, 10, 11) é executado no máximo uma vez por aresta
  - as operações de inserção e remoção na fila (nos passos 4, 7 e 11) são executadas no máximo uma vez por vértice
  - a inicialização leva um tempo proporcional ao tamanho do grafo

Implementação (1/2)

```
public LinkedList<Vertex> topsort()
                          throws CycleFoundException
{
    // Inicializações
    for (Vertex v : vertexSet)
      v.indegree = 0;
    for (Vertex v : vertexSet)
      for (Edge e : v.adj)
        e.dest.indegree++;

    Queue<Vertex> C = new LinkedList<Vertex>();
    for (Vertex v : vertexSet)
      if (v.indegree == 0)
        C.add(v);

    LinkedList<Vertex> T = new LinkedList<Vertex>();
```


Implementação (2/2)

```
    while (!C.isEmpty()) {
      Vertex v = C.remove();
      T.add(v);
      for (Edge e : v.adj) {
        Vertex w = e.dest;
        w.indegree--;
        if (w.indegree == 0)
          C.add(w);
      }
    }
    // Terminação
    if (T.size() != vertexSet.size())
      throw new CycleFoundException();
    return T;
}
```

Aplicações

- Grafos Acíclicos Dirigidos (DAG) são usados em aplicações onde é necessário indicar a precedência entre eventos.
- Exemplo: Escalonamento de Sequências de tarefas.
- A ordenação topológica de um DAG dá-nos uma ordem (sequência) dos eventos (tarefas, trabalhos, etc.) representados nesse DAG.

Referências e mais informação

- "Data Structures and Algorithm Analysis in Java", Second Edition, Mark Allen Weiss, Addison Wesley, 2006
- "Introduction to Algorithms", Second Edition, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, The MIT Press, 2001