Algoritmos em Grafos: Caminho mais curto

R. Rossetti, A.P. Rocha, J. Pascoal Faria

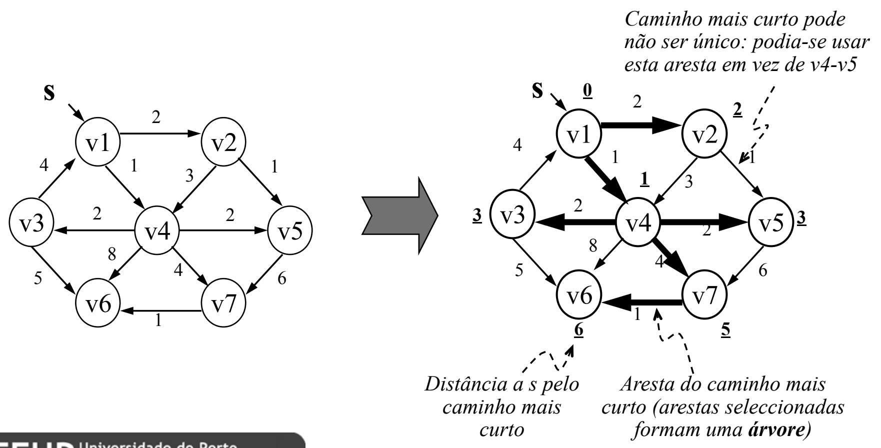
FEUP, MIEIC, CAL, 2013/2014

Índice

- Caminhos mais curtos dum vértice para todos os outros
  - Caso de grafos dirigidos não pesados
  - Caso de grafos dirigidos pesados
  - Caso de grafos dirigidos com arestas de peso negativo
  - Caso de grafos dirigidos acíclicos
- Caminhos mais curtos entre todos os pares de vértices
- Caminho mais curto entre dois vértices
- Optimizações para redes viárias
- Aplicação à gestão de projectos

Caminhos mais curtos dum vértice para todos os outros

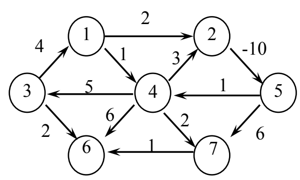
Dado um grafo pesado G = (V, E) e um vértice *s*, obter o caminho mais "curto" (de peso total mínimo) de *s* para cada um dos outros vértices em G



Variantes

(em relação ao caso base: grafo dirigido, fortemente conexo, pesos >0)

- Grafo não dirigido
  - Mesmo que grafo dirigido com pares de arestas simétricas
- Grafo não conexo
  - Pode não existir caminho para alguns vértices, ficando distância infinita
- Grafo não pesado
  - Mesmo que peso 1 (mais curto = com menos arestas)
  - Existe um algoritmo mais eficiente para este caso do que p/ caso base
- Arestas com custos negativos
  - Ciclos com custo negativo tornam o caminho mais curto indefinido (de v4 a v7 o custo pode ser 2 ou -1 ou -4 ou …)
  - Exige algoritmo menos eficiente para este caso do que para o caso base



Aplicações

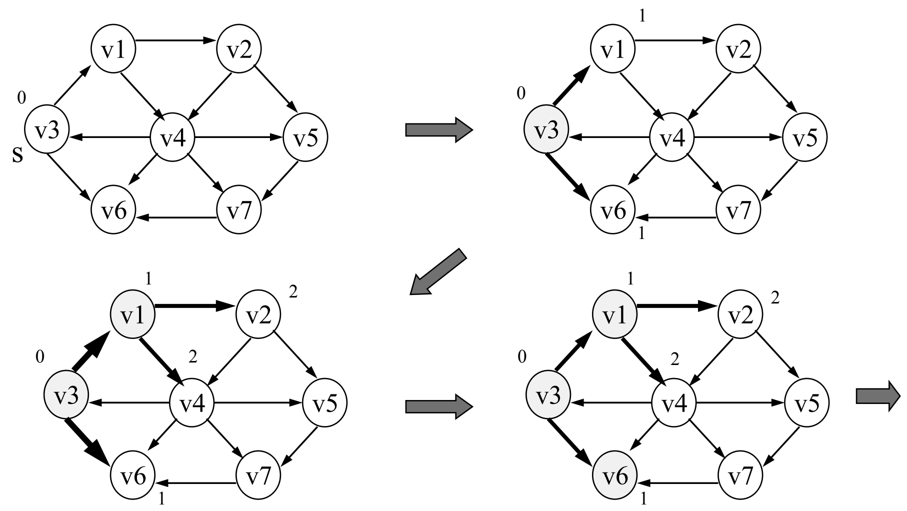
- Problemas de encaminhamento (*routing*)
  - Encontrar o melhor percurso numa rede viária
  - Encontrar o melhor percurso de avião
  - Encontrar o melhor percurso de metro
  - Encaminhamento de tráfego em redes informáticas
- Problemas de planeamento:
  - Por onde passar cabos nas ruas, desde uma sede até um conjunto de filiais, por forma a minimizar a distância de cada filial até à sede

Caso de grafo dirigido não pesado

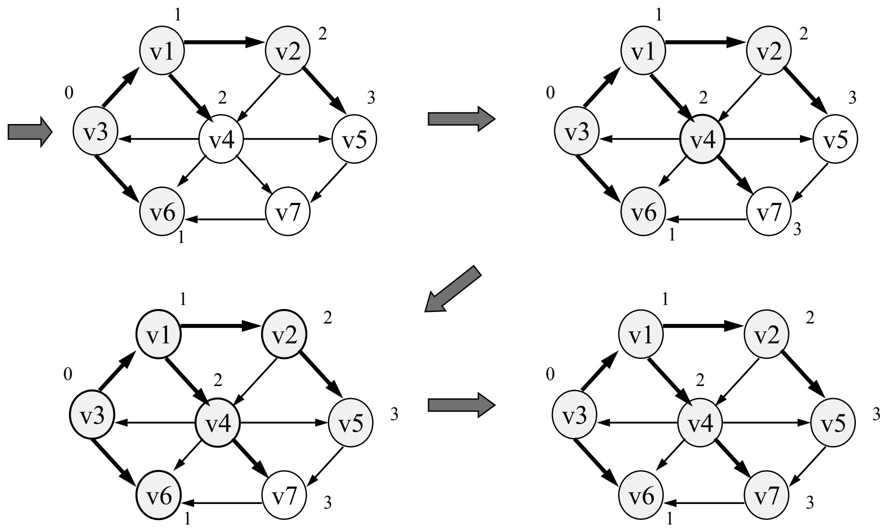
- Método básico
  1. Marcar o vértice *s* com distância 0 e todos os outros com distância ∞
  2. Entre os vértices já alcançados (distância ≠ ∞) e não processados (no passo 3), escolher para processar o vértice *v* marcado com distância mínima
  3. Processar vértice v: analisar os adjacentes de v, marcando os que ainda não tinham sido alcançados (distância ∞) com distância de *v* mais 1
  4. Passar ao passo 2, se existirem mais vértices para processar

- Esta ordem de progressão por distâncias crescentes (1º vértices a distância 0, depois a distância 1, ...) é crucial para garantir eficiência
  - Distância fica definitivamente definida ao alcançar um vértice pela 1ª vez; ao alcançar por um 2º caminho, a distância nunca é inferior
  - Este tipo de pesquisa em grafos designa-se por **pesquisa em largura**

Evolução da marcação do grafo (1/2)



Evolução da marcação do grafo (2/2)



Estruturas de dados

- Usando uma fila (FIFO) para inserir os novos vértices alcançados e extrair o próximo vértice a processar, garante-se a ordem de progressão pretendida
- Associa-se a cada vértice a seguinte informação:
  - *dist*: distância ao vértice inicial
  - *path*: vértice antecessor no caminho mais curto (inicializado com null)

Implementação

```
void shortestPathUnweighted(Vertex s) {
      for (Vertex v : vertexSet) {
            v.path = null;
            v.dist = INFINITY;
      }
      s.dist = 0;
      Queue<Vertex> q = new LinkedList<Vertex>();
      q.add(s); // na cauda
      while( ! q.isEmpty() ) {
            v = q.remove(); // da cabeça
            for (Edge e : v.adj) {
                  Vertex w = e.dest;
                  if (w.dist == INFINITY) {
                        w.dist = v.dist + 1;
                        w.path = v;
                        q.add(w);
                  }
            }
      }
}
```

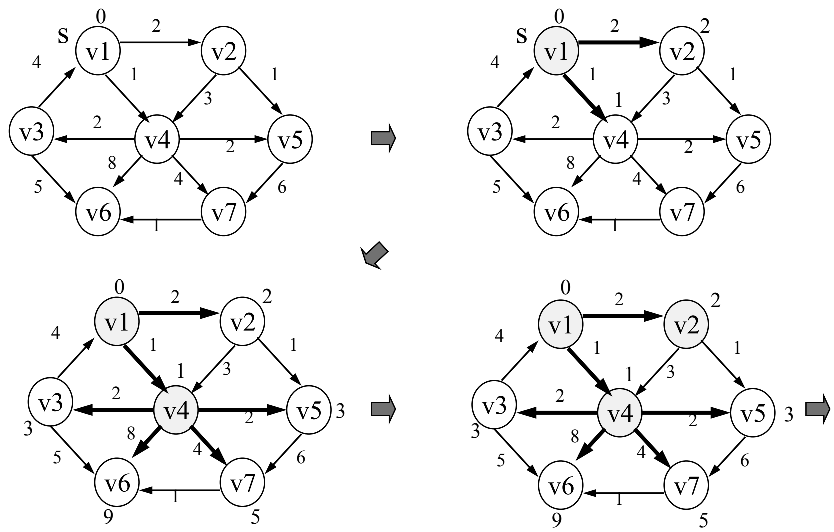
Tempo de execução:
O(|E| + |V|)

Espaço auxiliar:
O(|V|)

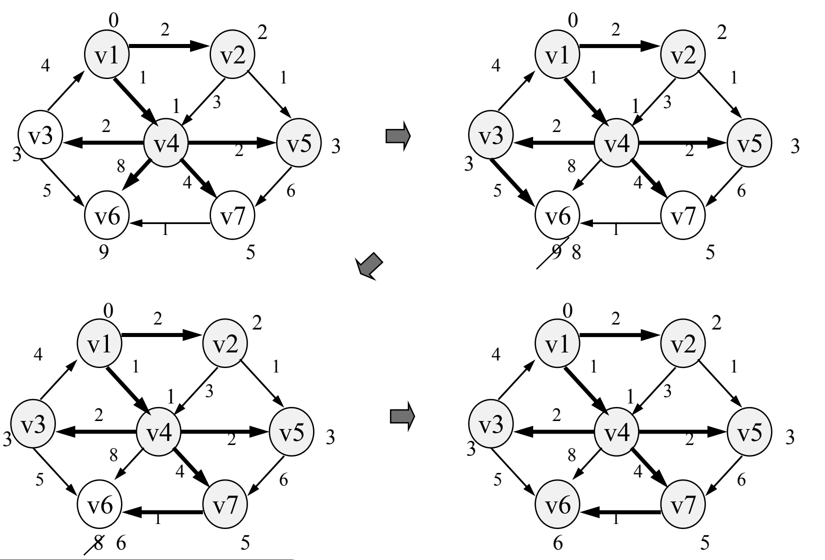
Caso de grafo dirigido pesado

- Método básico semelhante ao caso de grafo não pesado
  - Distância obtém-se somando pesos das arestas em vez de 1
  - Próximo vértice a processar continua a ser o de distância mínima
    - Mas já não é necessariamente o mais antigo ⇒ Obriga a usar fila de prioridades (com mínimo à cabeça) em vez duma fila simples
    - A ordem é crucial para garantir que a distância (e caminho mais curto) dos vértices já processados ao vértice inicial não é mais alterada, assumindo que não há pesos negativos
    - Mas já pode ser necessário rever em baixa a distância de um vértice alcançado e ainda não processado (vértice na fila) ⇒ Obriga a usar fila de prioridades alteráveis
  - Restrição: só é válido se não existirem custos negativos
  - É um algoritmo ganancioso: em cada passo procura maximizar o ganho imediato (neste caso, minimizar a distância)

Evolução da marcação do grafo (1/2)



Evolução da marcação do grafo (2/2)



Algoritmo de Dijkstra (modificado)

```
void Dijkstra(Vertex s) {
for (Vertex v : vertexSet) {v.path = null; v.dist = INFINITY;}
s.dist = 0;
PriorityQueue<Vertex> q = new PriorityQueue<Vertex>();
q.insert(s);
while ( ! q.isEmpty() ) {
   Vertex v = q.extractMin(); // remove vértice com dist mínimo
   for(Edge e: v.adj) {
      Vertex w = e.dest;
      if (v.dist + e.weight < w.dist) {
         w.dist = v.dist + e.weight;
         w.path = v;
         if (w.queueIndex == -1) // w ∉ q (ver a seguir)
              q.insert(w);
         else
              q.decreaseKey(w);
      }
   }
}
}
```

Não serve versão da biblioteca do Java

Assumindo que as arestas não têm pesos negativos, equivale a testar que w.dist antes da actualização era `INFINITY`

O(|E|*log |V|)

Eficiência de decreaseKey

- Suponhamos a fila de prioridades implementada com um heap (array) com o mínimo à cabeça e seja *n* o tamanho do heap
- Método naive:
  1. Procurar sequencialmente no array objeto cuja prioridade se quer alterar - O(n)
  2. Subi-lo (ou descê-lo) na árvore até restabelecer o invariante da árvore (cada nó menor ou igual que os filhos) - O(log n)
  - Total: O(n)   - Mau!
- Método otimizado:
  - Cada objeto colocado no heap guarda a sua posição no heap (queueIndex)
    - (Min)Heap: [v2 v3 v5 v4]
    - Objetos: v2(pri=3,idx=0),v3(pri=6,idx=1),v4(pri=7,idx=3),v5(pri=4,idx=2),v6(..,idx=-1)
  - Não é necessário o passo 1), logo o tempo total é O(log n)
- Introduz um *overhead* mínimo nas inserções e eliminações (quando se insere/move um objeto no heap, queueIndex tem de ser atualizado)
- Com Fibonacci Heaps consegue-se fazer decreaseKey em tempo amortizado O(1) (vide referências)

Eficiência do algoritmo de Dijkstra*

- Tempo de execução é

  O(|V| + |E| + |V|*log|V| + |E|*log |V|), ou simplesmente

  O(|E|*log |V|),   se |E|>|V|
  - O(|V|*log |V|) - extração e inserção na fila de prioridades
    - O nº de extrações e inserções é |V|
    - Cada operação destas pode ser feita em tempo logarítmico no tamanho da fila, que no máximo é |V|
  - O(|E|*log |V|) – decreaseKey
    - Feito no máximo |E| vezes (uma vez por cada aresta)
    - Cada operação destas pode ser feita em tempo logarítmico no tamanho da fila, que no máximo é |V|
- Pode ser melhorado para O(|V|*log |V|) com Fibonacci Heaps (ver 2ª referência)
- Nota: O algoritmo proposto inicialmente por Dijkstra não mencionava filas de prioridades, e tinha uma eficiência O(|V|^2)

Caso de arestas com peso negativo

- Pode ser necessário processar cada vértice mais do que uma vez
- **Algoritmo de Bellman-Ford**: semelhante ao algoritmo de Dijkstra, mas com uma fila simples (como no caso de grafo sem pesos) em vez duma fila de prioridades (logo não precisa de *decreaseKey*)
- Se não existirem ciclos com peso negativo, cada vértice é processado no máximo |V| vezes e o tempo de execução no pior caso é O(|V|*|E|)
- Se existirem ciclos com peso negativo, o problema não tem solução
- Assim que um vértice é processado mais do que |V| vezes, podemos concluir que existem ciclos com peso negativo e parar o algoritmo

Caso de grafos acíclicos

- Simplificação do algoritmo de Dijkstra
  - Processam-se os vértices por ordem topológica
  - Suficiente para garantir que um vértice processado jamais pode vir a ser alterado, pois não há arestas a entrar
  - Combina-se a ordenação topológica com a atualização das distâncias e caminhos numa só passagem
  - Basta usar a fila simples da ordenação topológica, não é necessário usar uma fila de prioridades
  - Tempo de execução O(|V|+|E|)
- Aplicações
  - Processos irreversíveis
    - não se pode regressar a um estado passado (certas reações químicas)
    - deslocação entre dois pontos "em esqui" (sempre descendente)
  - Gestão de projetos
    - Projeto composto por atividades com precedências acíclicas (não se pode começar uma atividade sem ter acabado uma precedente) - ver adiante

Caminho mais curto entre todos os pares de vértices*

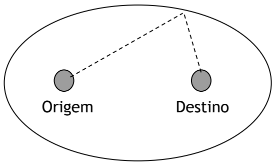
- Execução repetida do algoritmo de Dijkstra (ganancioso): O(|V|*|E|*log|V|)
  - Bom se o grafo for esparso (|E| ~ |V|)
- Algoritmo de Floyd-Warshall, baseado em programação dinâmica: $O(|V|^3)$
  - Melhor que o anterior se o grafo for denso ($|E| \sim |V|^2$)
  - Mesmo em grafos pouco densos pode ser melhor porque código é mais simples
  - Usa <u>matriz de adjacências</u> W[i,j] com pesos (∞ quando não há aresta; 0 quando i=j)
  - Calcula <u>matriz de distâncias</u> mínimas D[i,j]
  - Em cada iteração k (de 0 a |V|), D[i,j] tem a distância mínima do vértice i a j, podendo usar vértices intermédios apenas do conjunto {1, …, k}
  - Fórmula de recorrência: $D[i,j]^{(k)} = \min(D[i,j]^{(k-1)}, D[i,k]^{(k-1)}+D[k,j]^{(k-1)})$, k=1…|V|
    Começando em: $D[i,j]^{(0)} = W[i,j]$
  - Minimizar memória: atualizar matriz em cada iteração k, em vez de criar nova
  - Mantém-se ao mesmo tempo uma <u>matriz P</u>[i,j] que indica o vértice anterior a *j* no caminho mais curto de *i* para *j*

Caminho mais curto entre dois vértices

- Problema muito importante na prática
  - Exemplo: caminho mais curto entre dois pontos num mapa de estradas
- Não se conhece algoritmo mais eficiente a resolver este problema do que a resolver o mais geral (de um vértice para todos os outros)
- Portanto, acha-se o caminho mais curto da origem para todos os outros, e seleciona-se depois o caminho da origem para o destino pretendido
- Otimização: parar assim que chega a vez de processar o vértice de destino
  - Num mapa de estradas, ajuda para distâncias curtas, mas não para distâncias longas

Optimizações para redes viárias (1)*

- Tirar partido da natureza geométrica da rede
  - Se a rede estiver bem concebida, a distância mais curta por estrada entre dois pontos não difere da distância Euclidiana (em linha recta) a menos de um certo valor (aditivo ou multiplicativo)
  - Pré processa-se o mapa (achando caminhos mais curtos entre todos os pares de vértices) para obter este valor
  - Ao procurar o caminho mais curto entre dois pontos, usa-se o valor anterior para determinar a distância máxima por estrada, e limita-se a zona de procura a uma elipse de corda igual a essa distância máxima



Optimizações para redes viárias (2)*

- Tirar partido da natureza hierárquica da rede
  - Vias de trânsito local (dentro de cidade), vias de trânsito de longa distância (entre cidades), etc.
  - Pré processa-se o mapa (achando caminhos mais curtos entre todos os pares de vértices) para determinar o grau de localidade *L(x)* de cada vértice (e aresta) *x*:

    L(x) = max { min(dist-euclidiana(s, x), dist-euclidiana(x, t)) | s, t $\in$ V $\wedge$ x $\in$ shortest-path(s,t) }

  - Ao procurar o caminho mais curto entre dois vértices *s* e *t*, ignoram-se todos as arestas e vértices *x* tais que

    min(dist-euclidiana(s,x), dist-euclidiana(x,t)) > L(x)

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