Algoritmos em Grafos: Caminho mais curto (Parte I)

R. Rossetti, A. P. Rocha, L. Ferreira, J. P. Fernandes, F. Ramos, G. Leão

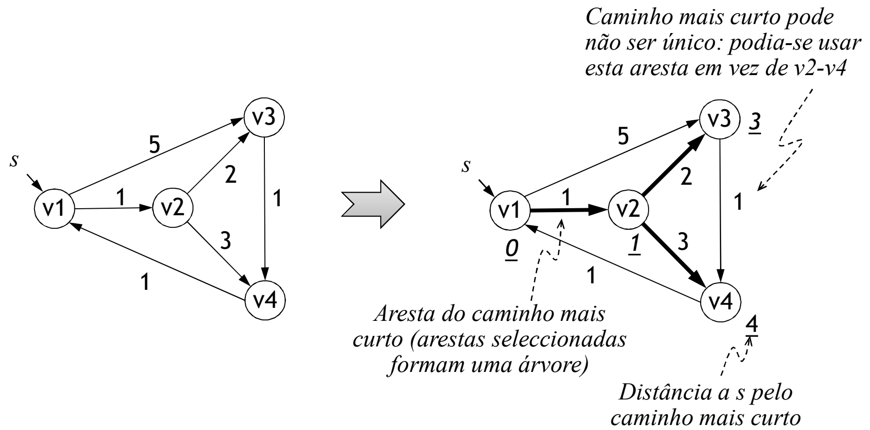
FEUP, MIEIC, CAL

Índice

- Caminhos mais curtos de um vértice para todos os outros
  - Caso de grafos dirigidos não pesados
    - baseado em pesquisa em largura, O(|V| + |E|)
  - Caso de grafos dirigidos pesados
    - Dijkstra, algoritmo ganancioso, O((|V| + |E|) log |V|)
  - Caso de grafos dirigidos com arestas de peso negativo
    - Bellman-Ford, programação dinâmica, O(|E| |V|)
  - Caso de grafos dirigidos acíclicos
    - baseado em ordenação topológica, O(|V| + |E|)

Problema

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

- <u>Caso base</u>: grafo dirigido, fortemente conexo, pesos >=0
- <u>Grafo não dirigido</u>
  - Mesmo que grafo dirigido com pares de arestas simétricas
- <u>Grafo não conexo</u>
  - Pode não existir caminho para alguns vértices, ficando distância infinita
- <u>Grafo não pesado</u>
  - Mesmo que peso 1 (mais curto = com menos arestas)
  - Existe algoritmo mais eficiente para este caso do que p/ caso base
- <u>Arestas com pesos negativos</u>
  - Existe algoritmo menos eficiente para este caso do que p/ caso base
  - Ciclos com peso negativo tornam o caminho mais curto indefinido

Aplicações

- Problemas de encaminhamento (*routing*)
  - Encontrar o melhor percurso numa rede viária
    - Nota: Algoritmo para encontrar caminho mais curto entre 2 pontos é baseado no algoritmo para encontrar caminhos mais curtos do ponto de partida para todos os outros
  - Encontrar o melhor percurso de avião
  - Encontrar o melhor percurso de metro
  - Encaminhamento de tráfego em redes informáticas
- Problemas de planeamento:
  - Planeamento de tarefas e respectivas dependências
  - Problemas operacionais, como minimização da cablagem necessária à ligação de pontos numa rede organizacional

Caso de grafo dirigido não pesado

- Método básico (**pesquisa em largura** + cálculo de distâncias):
  1. Marcar o vértice *s* com distância 0 e todos os outros com distância ∞
  2. Entre os vértices já alcançados (distância $\neq \infty$) 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. Voltar 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 definitiva / definida ao alcançar um vértice pela 1ª vez; ao alcançar por um 2º caminho, distância nunca diminui

Estruturas de dados

- Usando uma fila (FIFO) para inserir os novos vértices alcançados e extrair o próximo vértice a processar (ver bfs), 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 c/ nil)

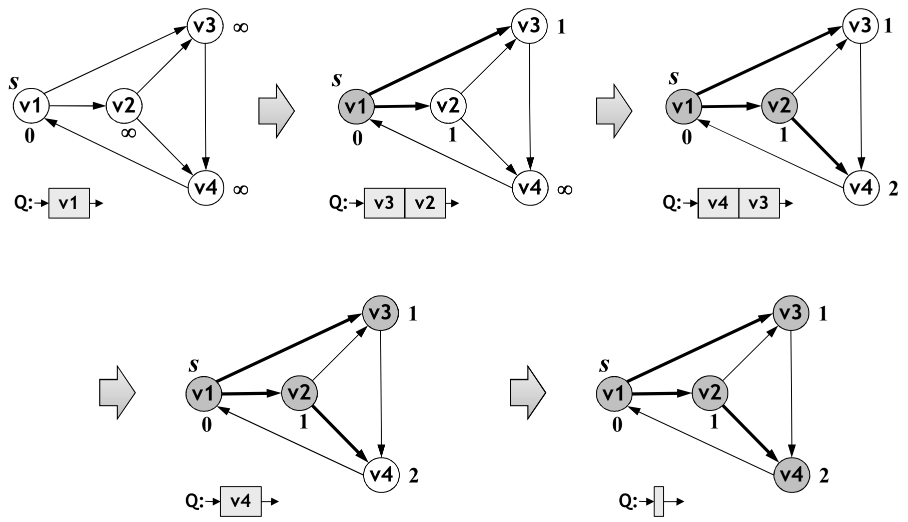
Pseudo-código

```
SHORTEST-PATH-UNWEIGHTED(G=(V,E), s):
    for each v ∈ V do
        dist(v) ← ∞
        path(v) ← nil
    dist(s) ← 0
    Q ← ∅
    ENQUEUE(Q, s)
    while Q ≠ ∅ do
        v ← DEQUEUE(Q)
        for each w ∈ Adj(v) do
            if dist(w) = ∞ then
                ENQUEUE(Q, w)
                dist(w) ← dist(v) + 1
                path(w) ← v
```

Tempo de execução: $O(|E| + |V|)$

Espaço auxiliar: $O(|V|)$

Evolução da marcação do grafo



Caso de grafo dirigido pesado (pesos≥0)

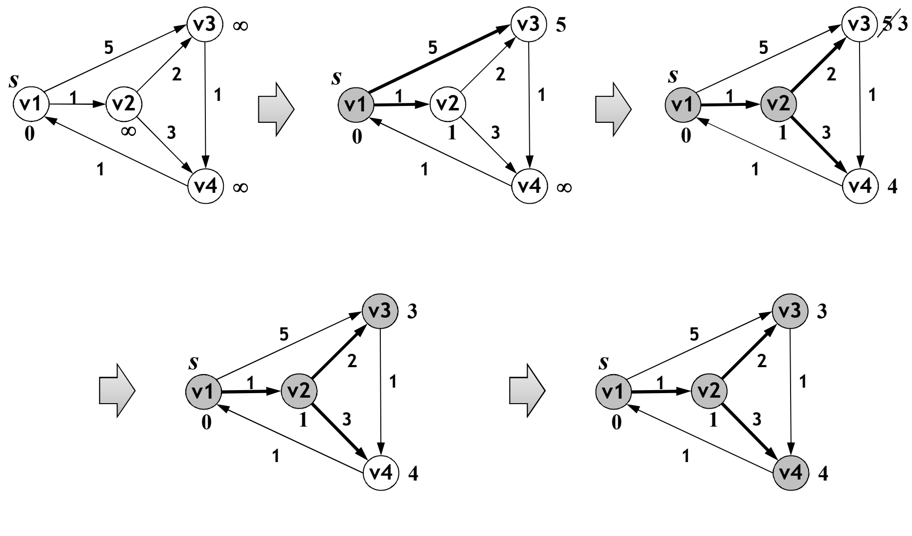
- Método básico semelhante ao caso de grafo não pesado
- Distância obtém-se somando pesos das arestas em vez de 1
- Próx. vértice a processar continua a ser o de distância mínima
  - Mas já não é necessariamente o mais antigo ⇒ Obriga a usar **fila de prioridades** (com mínimo à cabeça) em vez duma fila simples
  - Mas 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**
  - Nota: A ordem é crucial para garantir que a distância ao vértice de partida dos vértices já processados não é mais alterada, assumindo que não há pesos negativos (ver análise adiante)
- É um algoritmo **ganancioso**: em cada passo procura maximizar o ganho imediato (neste caso, minimizar a distância)

Algoritmo de Dijkstra (adaptado)

```
DIJKSTRA(G, s):  // G=(V,E), s ∈ V
    for each v ∈ V do
       dist(v) ← ∞
       path(v) ← nil
    dist(s) ← 0
    Q ← ∅   // min-priority queue
    INSERT(Q, (s, 0)) // inserts s with key 0
    while Q ≠ ∅ do
       v ← EXTRACT-MIN(Q) // greedy
       for each w ∈ Adj(v) do
          if dist(w) > dist(v) + weight(v,w) then
             dist(w) ← dist(v)+ weight(v,w)
             path(w) ← v
             if w ∉ Q then  // old dist(w) was ∞
                INSERT(Q, (w, dist(w)))
             else
                DECREASE-KEY(Q, (w, dist(w)))
```

Tempo de execução:
O((|V|+|E|) * log |V|)

Evolução da marcação do grafo



Eficiência de DECREASE-KEY

- Suponhamos a fila de prioridades implementada com um *heap* (array) com o mínimo à cabeça e seja ***n*** o tamanho do *heap* (no máximo |V|)
- Método naïve: O(n)
  1. Procurar sequencialmente no array objeto cuja chave se quer alterar: O(n)
  2. Subir (ou descer) o objeto na árvore até restabelecer o invariante da árvore (cada nó menor ou igual que os filhos): O(log n)
  - Total: O(n) - Mau!
- Método melhorado: O(log n)
  - Cada objeto colocado no *heap* guarda a sua posição (índice) no *array*
  - 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, o seu índice tem de ser atualizado)
- Método otimizado: O(1)
  - Com Fibonacci Heaps consegue-se fazer DECREASE-KEY em tempo amortizado O(1) (ver referências)

Eficiência do algoritmo de Dijkstra

- Tempo de execução é

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

  ou simplesmente

  O((|V|+|E|) * log |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|) – DECREASE-KEY
    - 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
- *Nota: O algoritmo proposto inicialmente por Dijkstra não mencionava filas de prioridades, e tinha uma eficiência $O(|V|^2)$

Análise do algoritmo de Dijkstra (1/2)

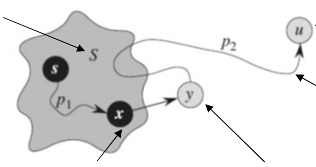
- Prova-se por indução o **invariante** do ciclo principal:
  - **A distância conhecida dos vértices já processados ao vértice de partida é a distância mínima** (logo não é mais alterada)
  - Mais precisamente, no momento em que se selecione um vértice *u* para processamento no início da ciclo, tem-se $d_{su} = \delta_{su}$
  - Notação: d – distância conhecida, δ - distância mínima

- Base (1º vértice) (inicialização do invariante)
  - O 1º vértice processado é o vértice de partida (s), com distância 0
  - Esta distância não pode ser reduzida (assumindo arestas de peso ≥ 0)

- Passo indutivo (k-ésimo vértice, k > 1) (manutenção do invariante)
  - Assume-se que o invariante se verifica para k-1
  - Ver slide seguinte

Análise do algoritmo de Dijkstra (2/2)

Conjunto de vértices já processados



Vértice que vai ser processado a seguir

Caminho mais curto de *s* para *u*

Último vértice em S no caminho mais curto (pode ser s=x)

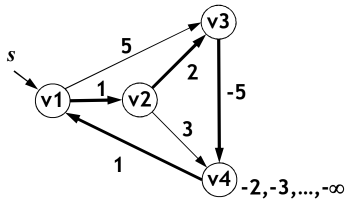
Vértice seguinte no caminho mais curto (pode ser y=u)

- Uma vez que o algoritmo funciona por relaxamento sucessivo, $d_{su} \geq \delta_{su}$ (1)
- Sendo um caminho mais curto, e estando *y* antes de *u*, tem-se $\delta_{sy} \leq \delta_{su}$ (2)
- Sendo *u* o próximo vértice escolhido para processar, tem-se $d_{su} \leq d_{sy}$ (3)
- Uma vez que a aresta (x,y) foi analisada quando *x* foi processado com $d_{sx} = \delta_{sx}$ (pelo invariante), e (x,y) está num caminho mais curto, tem-se $d_{sy} = \delta_{sy}$ (4)
- Combinando (1), (2), (3) e (4), tem-se $\delta_{su} \leq d_{su} \leq d_{sy} = \delta_{sy} \leq \delta_{su}$ o que só é possível com $\delta_{su} = \mathbf{d_{su}} = d_{sy} = \delta_{sy} = \delta_{su}$, c.q.d.

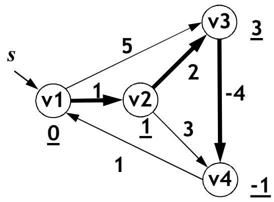
Caso de arestas com peso negativo

- Neste caso pode ser necessário processar cada vértice mais do que uma vez.
- Se existirem ciclos com peso negativo, o problema não tem solução.
- Não existindo ciclos com peso negativo, o problema é resolúvel em tempo O(|V||E|) pelo **algoritmo de Bellman-Ford** (a seguir).

Sem solução, pois tem um ciclo de peso negativo (-1). Percorrendo o ciclo várias vezes, diminui-se o peso do caminho.



Com solução, pois não tem ciclos de peso negativo.

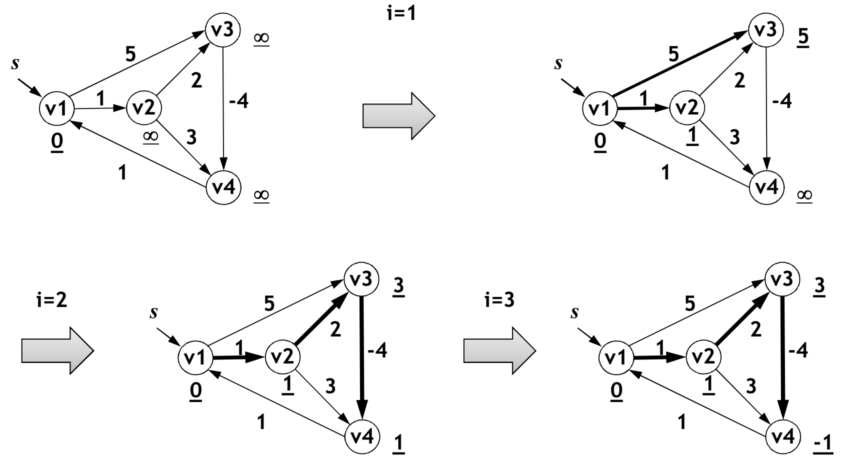


Algoritmo de Bellman-Ford

```
BELLMAN-FORD(G, s):  // G=(V,E), s ∈ V
    for each v ∈ V do
       dist(v) ← ∞
       path(v) ← nil
    dist(s) ← 0
    for i = 1 to |V|-1 do
       for each (v, w) ∈ E do
          if dist(w) > dist(v) + weight(v,w) then
             dist(w) ← dist(v)+ weight(v,w)
             path(w) ← v
    for each (v, w) ∈ E do
       if dist(v) + weight(v,w) < dist(w) then
          fail("there are cycles of negative weight")
```

Tempo de execução: O(|E| |V|)

Evolução da marcação do grafo



Análise do algoritmo de Bellman-Ford

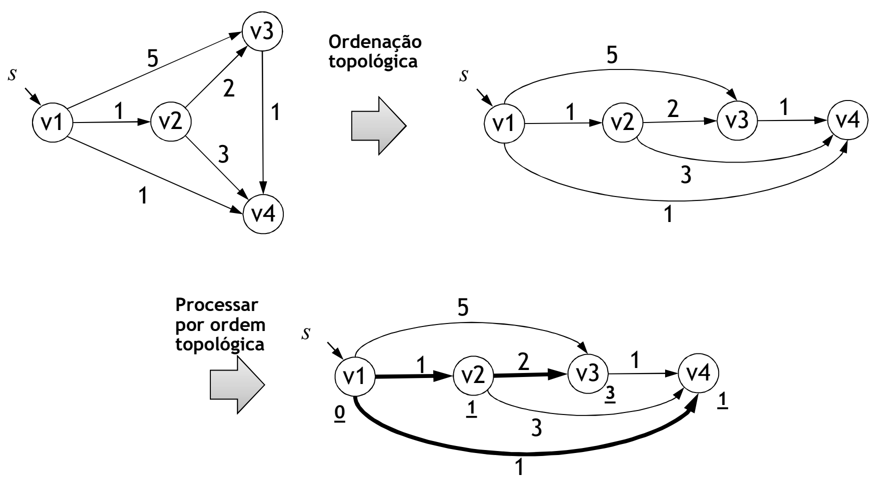
- Em cada iteração *i*, o algoritmo processa todas as arestas e garante que encontra todos os caminhos mais curtos com até *i* arestas (e possivelmente alguns mais longos) (invariante do ciclo principal).
- Uma vez que o caminho mais comprido, sem ciclos, tem |V|-1 arestas, basta executar no máximo |V|-1 iterações do ciclo principal para assegurar que todos os caminhos mais curtos são encontrados.
- No final é executada mais uma iteração para ver se alguma distância pode ser melhorada; se for o caso, significa que há um caminho mais curto com |V| arestas, o que só pode acontecer se existir pelo menos um ciclo de peso negativo.

- Podem ser efetuadas algumas melhorias ao algoritmo, mas que mantêm a complexidade temporal de O (|V| |E|).
- É um caso de aplicação de programação dinâmica (Porquê?)

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 'novas' a entrar
  - Pode-se combinar a ordenação topológica com a atualização das distâncias e caminhos numa só passagem
  - Tempo de execução é o da ordenação topológica: 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)

Exemplo



Referências e mais informação

- "Introduction to Algorithms", 3rd Edition, T.H. Cormen, C. E. Leiserson, R. L. Rivest , C. Stein., MIT Press, 2009
  - Capítulo 24 (Single-Source Shortest Paths)
  - Capítulo 19 (Fibbonacci Heaps)