Algoritmos em Grafos: Caminho mais curto

J. Pascoal Faria, R. Rossetti, L. Ferreira

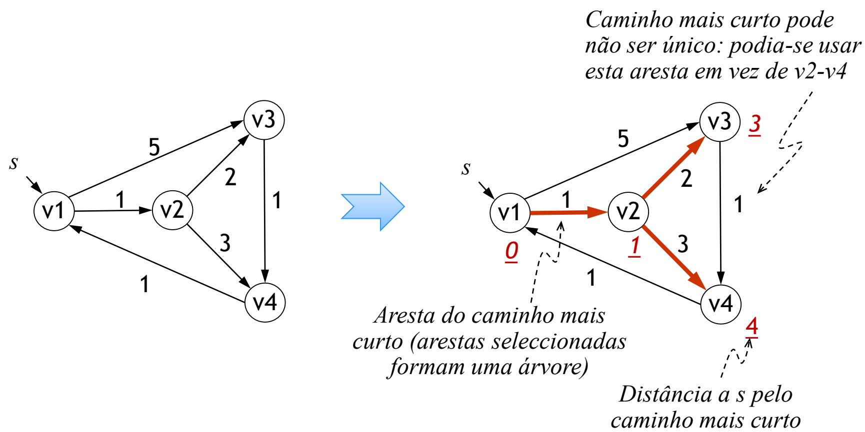
FEUP, MIEIC, CAL

Índice

Caminho mais curto de um vértice para todos os outros

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

- 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 algoritmo mais eficiente para este caso do que p/ caso base
- Arestas com pesos negativos
  - 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:
  - 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 (**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 definitivam/ 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
```

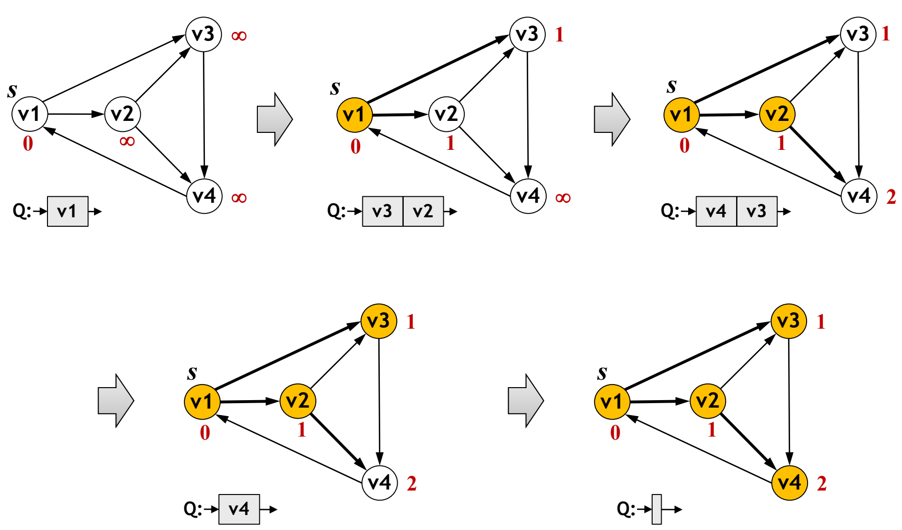
Tempo de execução:

O(|E| + |V|)

Espaço auxiliar:

O(|V|)

Evolução da marcação do grafo



Caso de grafo dirigido pesado (pesos≥0)

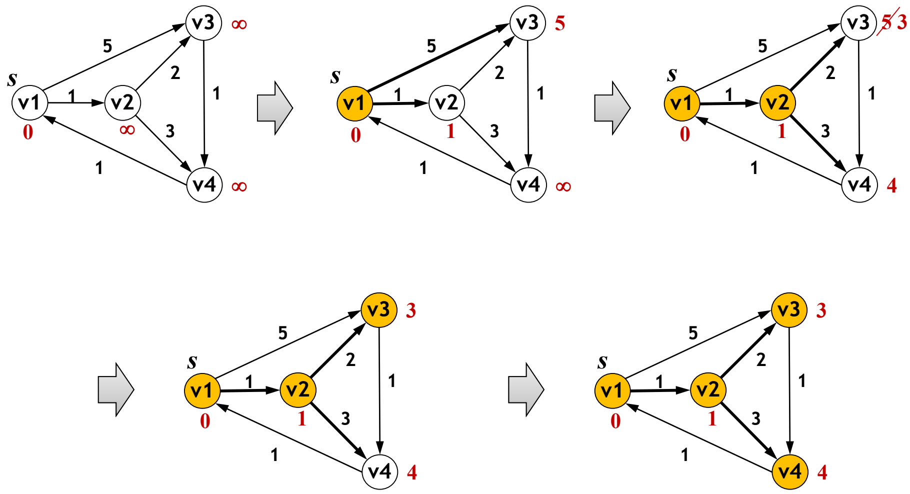
- Método básico semelhante ao caso de grafo não pesado
- Distância obtém-se somando pesos das arestas em vez de 1
- Próx. vértice a processar continua a ser o de distância mínima
  - Mas já não é necessariamente o mais antigo ⇒ Obriga a usar **fila de prioridades** (com mínimo à cabeça) em vez duma fila 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)

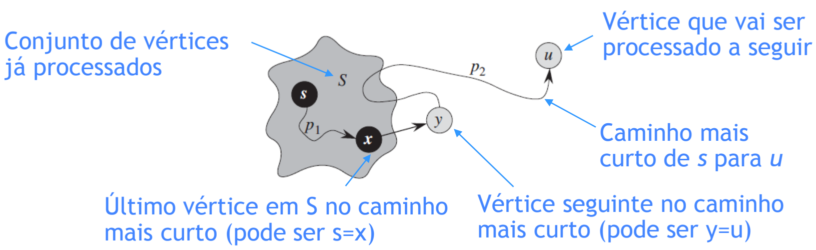
Eficiência do algoritmo de Dijkstra

- 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)

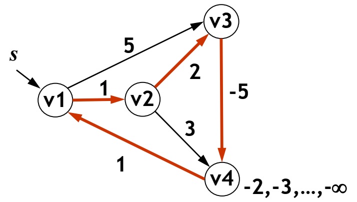


- Uma vez que o algoritmo funciona por relaxamento sucessivo, $d_{su} \geq \delta_{su}$ (1)
- Sendo um caminho mais curto, e estando *y* antes de *u*, tem-se $\delta_{sy} \leq \delta_{su}$ (2)
- Sendo *u* o próximo vértice escolhido para processar, tem-se $d_{su} \leq d_{sy}$ (3)
- Uma vez que a aresta (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 $\boldsymbol{\delta_{su} = d_{su}} = d_{sy} = \delta_{sy} = \delta_{su}$, c.q.d.

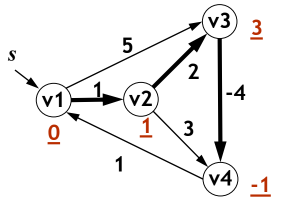
Caso de arestas com peso negativo

- 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.

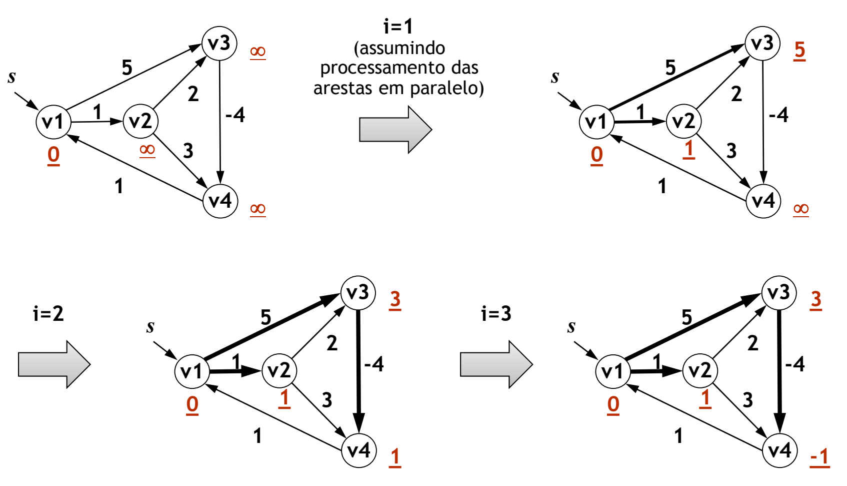


Algoritmo de Bellman-Ford

```
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



FEUP Universidade do Porto Faculdade de Engenharia
CAL, Algoritmos em Grafos: Caminho mais curto

Análise do algoritmo de Bellman-Ford

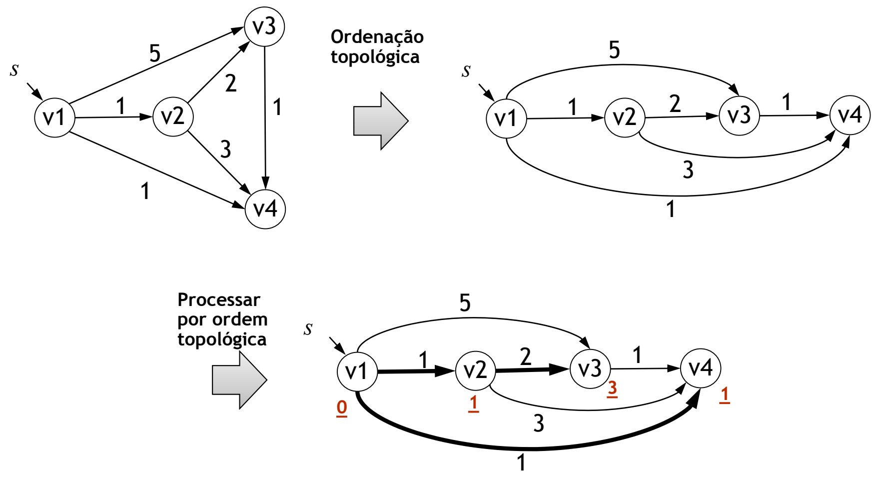
- Em cada iteração *i*, o algoritmo processa todas as arestas e garante que encontra todos os caminhos mais curtos com até *i* arestas (e possivelmente alguns mais longos) (invariante do ciclo principal).
- Uma vez que o caminho mais comprido, sem ciclos, tem |V|-1 arestas, basta executar no máximo |V|-1 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



Caminho mais curto entre todos os pares de vértices

Caminho mais curto entre todos os pares de vértices

- Execução repetida do algoritmo de Dijkstra (ganancioso): $O(|V|(|V|+|E|) \log|V|)$
  - Bom se o grafo for esparso ($|E| \sim |V|$)
- 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 o código é mais simples
  - Usa matriz de adjacências W[i,j] com pesos (∞ quando não há aresta; 0 quando i = j)
  - Calcula matriz de distâncias mínimas D[i,j] e matriz P[i,j] que indica o vértice anterior a *j* no caminho mais curto de *i* para *j*

Algoritmo de Floyd-Warshall

- 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}
- Inicialização:
  $D[i,j]^{(0)} = W[i,j]$, P[i,j](0) = nil
- Fórmula de recorrência:
  $D[i, j]^{(k)} = \min(D[i, j]^{(k-1)}, D[i, k]^{(k-1)} + D[k, j]^{(k-1)})$, $k = 1, \dots, |V|$
- Valor de $P[i,j]^{(k)}$ é atualizado conforme o termo mínimo escolhido
- Para minimizar memória, pode-se atualizar a matriz (por ordem apropriada) em cada iteração k, em vez de criar nova

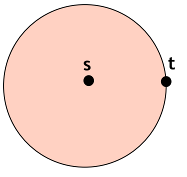
Caminho mais curto entre dois pontos numa rede viária

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

Método base (rede viária)

- Rede viária pode ser representada por um grafo dirigido em que
  - os vértices representam interseções
  - as arestas representam vias (possivelmente de sentido único)
  - os pesos representam distâncias, tempos, custos, etc.
- O algoritmo de Dijkstra é a base para encontrar o caminho mais curto entre dois pontos *s* e *t*, parando-se a pesquisa quando o próximo nó a processar é o nó *t*.
- Uma vez que o algoritmo processa os vértices por distâncias crescentes ao vértice de partida, é inspecionado um círculo em torno de *s* de "raio" igual à distância entre *s* e *t* (na métrica escolhida)

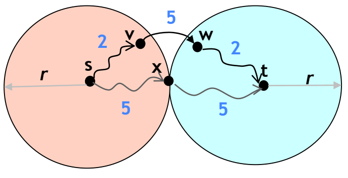


Otimizações

- Mas os mapas de estradas são muito grandes
  - Mapa de 17 países da Europa ocidental, do 9º desafio DIMACS sobre caminhos mais curtos (2009), tinha cerca de 19 milhões de nós e 23 milhões de arestas.
  - Em Fev. de 2018, open street maps (disponíveis publicamente) tinha cerca de 4.3×10^9 nós
- Algoritmo de Dijkstra pode demorar muitos segundos ou mesmo minutos a encontrar o caminho mais curto em trajetos de longa distância, o que não é apropriado para sistemas interativos
- Diversas otimizações conseguem ganhos (*speedup*) da ordem de 10^3 ou mesmo 10^6 (com pré-processamento mais ou menos complexo), reduzindo tempo de pesquisa para ms ou μs!

Pesquisa bidirecional

- Executar o algoritmo de Dijkstra no sentido de *s* para *t* e em sentido inverso de *t* para *s* (no grafo invertido), alternando passo a passo
- Manter a distância μ do caminho mais curto conhecido entre *s* e *t*: ao processar uma aresta (*v*, *w*) tal que *w* já foi processado na outra direção, verificar se o correspondente caminho *s*-*t* melhora μ
- Terminar quando se vai processar um vértice *x* já processado na outra direção (podendo o caminho mais curto passar por *x* ou não)
- Retornar a distância μ e o caminho correspondente



Área processada ~ $2\pi r^2$, em vez de ~ $4\pi r^2$ na pesquisa unidirecional.

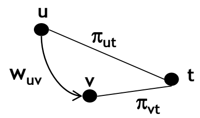
Ganho (*speedup*) ~2x

Pesquisa orientada ao objetivo (1/3)

- **Algoritmo A***: escolher para processar o vértice *v* com valor mínimo de $d_{sv} + \pi_{vt}$, parando quando se vai processar o vértice *t*
  - d_{sv} - distância mínima conhecida de *s* a *v*
  - π_{vt} - estimativa por baixo (*lower bound*) da distância mínima de *v* a *t* (função potencial)
- Em geral, não garante o ótimo
- Em certos casos, garante o ótimo, por exemplo:
  - Pesos das arestas são as distâncias em km
  - π_{vt} é a distância Euclidiana (em linha reta) de *v* a *t*
  - Equivale então a aplicar o algoritmo de Dijkstra com pesos das arestas modificados $w'_{uv} = w_{uv} - \pi_{ut} + \pi_{vt}$, somando-se no final π_{st} à distância mínima obtida de *s* para *t* (ver justificação a seguir).
  - Pode ser combinado com pesquisa bidirecional
  - Ganho (*speedup*) na prática é moderado

Pesquisa orientada ao objetivo (2/3)

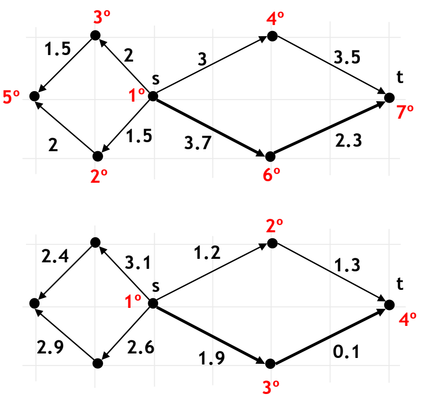
- Justificação:
  - Pela desigualdade triangular, garante-se $\pi_{ut} \leq w_{uv} + \pi_{vt}$, logo $w'_{uv} = w_{uv} - \pi_{ut} + \pi_{vt} \geq 0$.



  - O peso ao longo de um caminho $(s, v_1, v_2, \ldots, v_k)$, fica igual ao do grafo original, acrescido de $\pi_{st} - \pi_{v_k t}$ (outros termos intermédios cancelam-se)
  - Logo, escolher o vértice v com menor $d_{sv} + \pi_{vt}$ no grafo original (A*), é o mesmo que escolher o vértice com menor d_{sv} no grafo modificado (Dijkstra)

Pesquisa orientada ao objetivo (3/3)

- Ilustração



Distância mínima: 6

Expande 6 nós

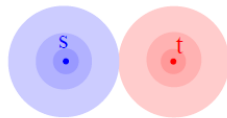
Somar/subtrair dist. geom. a *t* do vértice final/inicial

Distância mínima: 2+4=6
(4 = dist. geom. de *s* a *t*)

Expande 3 nós (50%)

Otimizações com pré-processamento

- Ganhos de desempenho muito significativos só c/pré-processamento
  - Compromisso entre: tempo de pré-processamento, tempo de pesquisa, espaço de armazenamento, facilidade de atualização c/ info. dinâmica
- Redes hierárquicas (*highway networks*)
  - Pré-processamento decompõe a rede em vários níveis hierárquicos
  - Pesquisa usa rede mais densa próximo de *s* e *t* e mais esparsa longe de *s* e *t*
  - A pesquisa é realizada em tempo ~1ms



- Nós de trânsito (*transit nodes*)
  - Pré-processamento calcula os nós de trânsito (cerca de 10^4 na Europa Ocidental) - tal que o caminho mais curto entre nós da rede que não estão “muito perto" entre si passa por pelo menos um dos nós de trânsito
  - Calcula para cada nó da rede os nós de trânsito próximos (nós de acesso)
  - A pesquisa é reduzida a poucos *table lookups*, realizada em tempo ~10μs

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)
- “Data Structures and Algorithm Analysis in Java”, Second Edition, Mark Allen Weiss, Addison Wesley, 2006
- Sanders P., Schultes D. (2007) Engineering Fast Route Planning Algorithms. In: Demetrescu C. (eds) Experimental Algorithms. WEA 2007. Lecture Notes in Computer Science, vol 4525. Springer, Berlin, Heidelberg
- Efficient Point-to-Point Shortest. Path Algorithms. Andrew V. Goldberg (Microsoft Research). Chris Harrelson (Google). Haim Kaplan (Tel Aviv University). Renato F. Werneck (Princeton University).