

Programação Dinâmica

prof. Fábio Luiz Usberti

MC521 - Desafios de Programação I

Instituto de Computação - UNICAMP - 2018

- 1 Introdução
- 2 Algoritmo de Dijkstra
- 3 Algoritmo de Bellman-Ford
- 4 Referências

Caminhos mínimos de um para todos

- **Problema de caminhos mínimos de um para todos:** Dado um grafo ponderado G , com comprimentos c_{ij} associados a cada aresta (i, j) , e um vértice de origem s . Quais são os caminhos mínimos de s para todos os outros vértices?
- **Aplicações:**
 - 1 Menor percurso entre duas cidades;
 - 2 Planejamento de sistemas de transporte;
 - 3 Transmissão de dados em uma rede de computadores;
 - 4 Desenho de placas de circuito integrado VLSI;

Caminhos mínimos de um para todos

- Nesta aula serão apresentados algoritmos eficientes (polinomiais) para o problema de caminhos mínimos de um para todos;
- Para um **grafo não ponderado** (ou se todas as arestas possuem o mesmo comprimento), é possível usar o algoritmo de busca em largura $O(V + E)$;
- Para **caminhos mínimos em grafos ponderados**, serão apresentados os algoritmos de **Dijkstra** em $O((V + E)\log V)$ e de **Bellman Ford** em $O(VE)$;
- O algoritmo de *Dijkstra* pressupõe que $c_{ij} \geq 0$, para toda aresta (i, j) .
- O algoritmo de *Bellman Ford* pode ser aplicado para grafos com **arestas de comprimento negativo**, contanto que não ocorram ciclos de comprimento negativo no grafo.

Visão geral

- Seja $dist[i]$ um **limitante superior** para a distância de caminho mínimo entre os vértices i e s .
- O algoritmo de Dijkstra é um algoritmo iterativo onde a cada iteração são realizados processos denominados de **relaxações** que procuram aproximar $dist[i]$ para a distância de caminho mínimo entre os vértices i e s .
- **Inicialização:** $dist[s] \leftarrow 0$ e $dist[i] \leftarrow \infty$ ($\forall i \neq s$). Anote todos os vértices como não visitados.
- **Iteração:** Enquanto existirem vértices não visitados.
 - 1 Escolha como vértice corrente um vértice não visitado i , tal que $i \leftarrow \arg \min_i [dist[i]]$.
 - 2 Anote o vértice i como visitado.
 - 3 Para cada vértice v vizinho de i :
 - ★ **Relaxação da aresta** (i, v) : Faça $dist[v] \leftarrow \min[dist[v], dist[i] + c_{iv}]$.

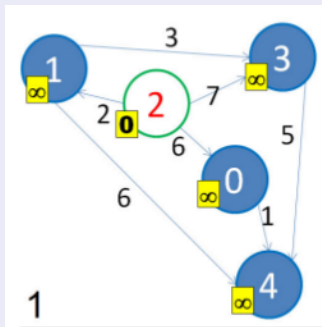
Fila de prioridades

- A cada iteração o algoritmo de *Dijkstra* busca pelo vértice não visitado i com o menor valor de $dist[i]$.
- Uma boa estrutura de dados para armazenar os vértices não visitados é uma **fila de prioridades** (heap de mínimo), adotando como prioridade de cada vértice i o valor de $dist[i]$.
- Considere que a fila de prioridades armazena para cada vértice i o par $\{dist[i], i\}$.
- Desse modo, as operações de **inserção** e de **extração do menor elemento** na fila passam a ter complexidade $O(\log n)$.

Algoritmo de Dijkstra

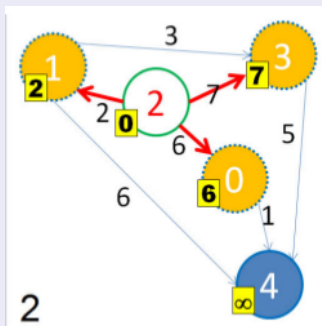
Exemplo

- Considere o grafo a seguir, com vértice de origem $s = 2$.
- **Inicialização:** $dist[2] \leftarrow 0$. Insira na fila de prioridades (pq) o par $\{dist[2], 2\}$, ou seja, $pq \leftarrow [\{0, 2\}]$.



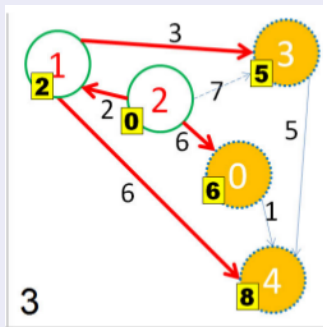
Exemplo

- **ITERAÇÃO 1:** $pq = [\{0, 2\}]$. Vértice corrente 2.
- **Relaxação:** arestas $(2, 0)$, $(2, 1)$ e $(2, 3)$; $dist[0] \leftarrow 6$, $dist[1] \leftarrow 2$ e $dist[3] \leftarrow 7$.
- **Fila de prioridades:** $pq \leftarrow [\{2, 1\}, \{6, 0\}, \{7, 3\}]$.



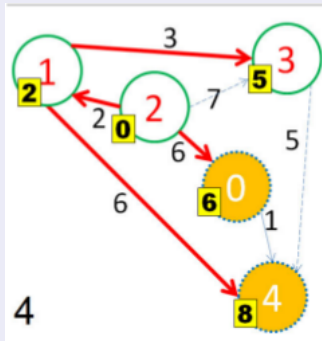
Exemplo

- **ITERAÇÃO 2:** $pq = [\{2, 1\}, \{6, 0\}, \{7, 3\}]$. Vértice corrente 1.
- **Relaxação:** arestas $(1, 3)$ e $(1, 4)$;
 $dist[3] \leftarrow \min(dist[3], dist[1] + c_{13}) = \min(7, 2 + 3) = 5$ e $dist[4] \leftarrow 8$.
- **Fila de prioridades:** $pq \leftarrow [\{5, 3\}, \{6, 0\}, \{7, 3\}, \{8, 4\}]$. **Obs.** Há duas ocorrências do vértice 3 na fila de prioridade. Isso não é um problema contanto que o algoritmo verifique se um vértice extraído da fila já foi visitado.



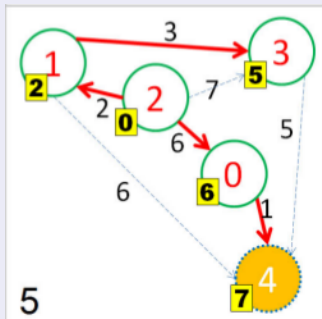
Exemplo

- **ITERAÇÃO 3:** $pq = [\{5, 3\}, \{6, 0\}, \{7, 3\}, \{8, 4\}]$. Vértice corrente 3.
- **Relaxação:** aresta $(3, 4)$; $dist[4] \leftarrow \min(dist[4], dist[3] + c_{34}) = \min(8, 10) = 8$.
- **Fila de prioridades:** $pq \leftarrow [\{6, 0\}, \{7, 3\}, \{8, 4\}]$.



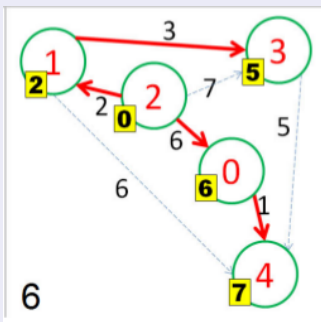
Exemplo

- **ITERAÇÃO 4:** $pq = [\{6, 0\}, \{7, 3\}, \{8, 4\}]$. Vértice corrente 0.
- **Relaxação:** aresta $(0, 4)$; $dist[4] \leftarrow \min(dist[4], dist[0] + c_{04}) = \min(8, 7) = 7$.
- **Fila de prioridades:** $pq \leftarrow [\{7, 3\}, \{7, 4\}, \{8, 4\}]$.



Exemplo

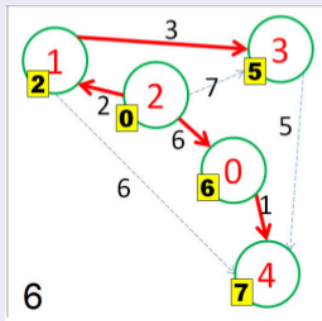
- **ITERAÇÃO 5:** $pq = [\{7, 3\}, \{7, 4\}, \{8, 4\}]$. O vértice 3 é o próximo da fila de prioridade, porém já foi visitado. Vértice corrente 4.
- **Relaxação:** não há arestas a serem relaxadas.
- **Fila de prioridades:** $pq \leftarrow [\{8, 4\}]$.



Algoritmo de Dijkstra

Exemplo

- $pq = [\{8, 4\}]$. O vértice 4 é o próximo da fila de prioridade, porém já foi visitado.
- O algoritmo se encerra, pois **fila de prioridades está vazia**.



Código-fonte

- Uma implementação do algoritmo de Dijkstra é apresentada a seguir.
- *pred[v]* armazena o **vértice predecessor** à *v* no caminho mínimo de *s* até *v*.

```
vi dist(V, INF);    // INF = 1E9 to avoid overflow
vi pred(V, -1);
priority_queue< ii , vector<ii > , greater<ii > > pq;
dist[s] = 0;
pq.push(ii(dist[s], s));

while (!pq.empty()) { // main loop
    ii front = pq.top(); pq.pop();    // greedy: get shortest unvisited vertex
    int d = front.first, u = front.second;
    if (d > dist[u]) continue;    // check for duplicates
    for (int j = 0; j < (int)AdjList[u].size(); j++) {
        ii v = AdjList[u][j];    // all outgoing edges from u
        if (dist[u] + v.second < dist[v.first]) {
            dist[v.first] = dist[u] + v.second;    // relax operation
            pred[v.first] = u;    // save predecessor
            pq.push(ii(dist[v.first], v.first));
        }
    }
}
```

Visão geral

- O **algoritmo de Bellman-Ford** é um algoritmo de programação dinâmica **bottom-up** que determina os caminhos mínimos de um vértice de origem **s** até os demais vértices, considerando **caminhos com até i arestas**.
- O **caso base** do algoritmo consiste nos caminhos com até **$i = 0$ arestas**.
- Em seguida, são obtidos os caminhos mínimos acrescidos de uma aresta, até a **última iteração**, onde o algoritmo obtém os caminhos mínimos com até **$i = |V| - 1$ arestas**.
- Os caminhos mínimos com até **$i = |V| - 1$ arestas** correspondem às **soluções do problema**, uma vez que um caminho simples possui no máximo **$|V| - 1$ arestas**.

Subestrutura ótima

- Considerando que $dist[v, k]$ é a distância mínima de um caminho com até k arestas da origem s até o vértice v , então a **subestrutura ótima** do problema pode ser expressa pela seguinte função recursiva:

$$dist[s, 0] = 0$$

$$dist[v, 0] = \infty \quad \text{se } v \neq s$$

$$dist[v, i] = \min \left\{ \min_{(u,v) \in E} \{dist[u, i-1] + c_{uv}\}, dist[v, i-1] \right\} \quad \text{se } i > 0$$

- Note que é possível **reduzir o uso de memória** armazenando somente as duas últimas colunas da tabela de programação dinâmica, uma vez que o cálculo da coluna i depende somente de valores armazenados na coluna $i-1$.

Redução do uso de memória

- Particularmente para esse problema, é possível **reduzir ainda mais o uso de memória** armazenando somente a última coluna da tabela de programação dinâmica.
- Nesse caso, o cálculo da coluna (iteração) i **sobrescreve diretamente** os valores armazenados na coluna (iteração) $i - 1$. Desse modo, a **subestrutura ótima** do problema pode ser expressa como:

$$\begin{aligned} dist[s] &= 0 && \text{se iteração } i = 0 \\ dist[v] &= \infty && \text{se iteração } i = 0 \text{ e } v \neq s \\ dist[v] &= \min\left\{ \min_{(u,v) \in E} \{dist[u] + c_{uv}\}, dist[v] \right\} && \text{se iteração } i > 0 \end{aligned}$$

Código-fonte

- Uma implementação **bottom-up** do algoritmo de Bellman-Ford é apresentada a seguir.
- O laço principal do algoritmo se repete $V - 1$ vezes, onde em cada iteração aplica-se a relaxação para todas as arestas do grafo.
- **pred[v]** armazena o **vértice predecessor** à **v** no caminho mínimo de **s** até **v**.

```
vi dist(V, INF);
vi pred(V, -1);
dist[s] = 0;

for (int i = 1; i <= V - 1; i++) // relax all E edges V-1 times
    for (int u = 0; u < V; u++) // these next two loops = O(E), overall O(VE)
        for (int j = 0; j < (int)AdjList[u].size(); j++) {
            ii v = AdjList[u][j];
            if (dist[u] + v.second < dist[v.first]) {
                dist[v.first] = dist[u] + v.second; // relax
                pred[v.first] = u; // save predecessor
            }
        }
```

Detecção de ciclo negativo

- O algoritmo de Bellman-Ford pode ser usado para **detectar a presença de ciclo negativo**.
- Após $|V| - 1$ relaxações de todas as arestas, o algoritmo obtém as distâncias de caminhos mínimos de **s** para todos os outros vértices.
- Um ciclo negativo pode ser detectado aplicando-se uma relaxação adicional em cada aresta.
- Se após a relaxação adicional for possível reduzir a distância de qualquer vértice até **s**, conclui-se que há pelo menos um ciclo de custo negativo.

```
// after running the O(VE) Bellman Ford's algorithm
bool hasNegativeCycle = false;

for (int u = 0; u < V; u++) // one more pass to check
    for (int j = 0; j < (int) AdjList[u].size(); j++) {
        ii v = AdjList[u][j];
        if (dist[v.first] > dist[u] + v.second) // if this is still possible
            hasNegativeCycle = true; // then negative cycle exists!
    }
printf("Negative Cycle Exist? %s\n", hasNegativeCycle ? "Yes" : "No");
```

Referências

- 1 S. Halim e F. Halim. Competitive Programming 2, Second Edition Lulu (www.lulu.com), 2011. (IMECC – 005.1 H139c)
- 2 S. S. Skiena, M. A. Revilla. Programming Challenges: The Programming Contest Training Manual, Springer, 2003.
- 3 T.H. Cormen, C.E. Leiserson, R.L.Rivest e C. Stein. Introduction to Algorithms. 2nd Edition, McGraw-Hill, 2001. (no. chamada IMECC – 005.133 Ar64j 3.ed.)
- 4 U. Manber. Introduction to Algorithms: A Creative Approach. Addison-Wesley. 1989. (no. chamada IMECC – 005.133 Ec53t 2.ed.)