# Programação Dinâmica: Algoritmo de Bellman-Ford

César Garcia Daudt
Caio Licks Pires de Miranda
*Instituto de Informática*
*Universidade Federal do Rio Grande do Sul*

22/11/2010

**Resumo**

Este artigo se propõe a fazer uma apresentação do algoritmo de Bellman-Ford, exemplificar algumas de suas aplicações, enquadrá-lo como um algoritmo de programação dinâmica e desenvolver o cálculo de sua complexidade. Apresentará também um exemplo simples de execução do algoritmo.

# 1 Introdução

O algortimo de Bellman-Ford, criado por Richard Bellman e Lester Ford, Jr.[2], computa, para um dado dígrafo (grafo orientado) $G$ com arestas ponderadas, o menor caminho de um nodo de origem $s$ até cada um dos outros nodos de $G$[4]. Vale ressaltar que este algoritmo, ao contrário do largamente conhecido algoritmo de Dijkstra, não impõe nenhuma restrição sobre o sinal do peso das arestas (usando o procedimento de Dijkstra, o problema antes referido é solucionado apenas se garantirmos que o peso de todas arestas é positivo), o que o torna uma solução mais genérica.[2]

A existência e cálculo do caminho mais curto são garantidos caso não haja a presença de ciclos negativos durante o caminho da origem até um nodo $v$ do grafo. Entende-se aqui por ciclo negativo todo caminho fechado pertencente a $G$ tal que a soma de seus pesos é negativa. De fato, caso seja necessário passar por um ciclo negativo $C$ para atingir um nodo, na i-ésima vez que completarmos $C$ estaremos realizando um caminho menor que na (i-1)-ésima vez, o que caracteriza a inexistência de um menor caminho. No entanto, o algoritmo de Bellman-Ford é capaz de indicar a presença de tais ciclos, o que nos mostra uma segunda utilidade do mesmo.

Algumas das aplicações deste algoritmo são:

- Protocolos de Roteamento Vetor-Distância.[2]
- Problema *"Triangular Arbitrage"* [3]

O último problema citado é bastante interessante e útil para o ramo de economia e investimentos. A idéia é sabermos se, dadas, no mínimo, três moedas diferentes (e.g. Reais, Dólares e Euros) e as taxas de conversão entre elas, há como realizarmos uma seqüência de câmbio da moeda original para as outras tal que, na hora em que voltarmos para moeda original, teremos uma quantia maior que a inicial. Esta aplicação vale-se da presença de um ciclo negativo no grafo que representa o problema. Uma explicação detalhada do problema e sua solução podem ser vistos em [3] e na apresentação que acompanha o artigo e será mostrada em aula.

# 2 Formalização

## 2.1 Idéia

Por serem feitos para solucionar problemas semelhantes, os algoritmos de Dijkstra e Bellman-Ford possuem estruturas similares[2]. No entanto, enquanto o primeiro busca exaustivamente o nodo com menor peso ainda não computado, o último apenas usa o procedimento de relaxamento[1] (descrito na seqüência do parágrafo seguinte) $V - 1$ vezes, onde $V$ representa a quantidade de vértices de $G$.

Basicamente o algoritmo de Bellman-Ford propaga as distâncias mínimas por $G$, repetindo os passos a seguir $V - 1$ vezes:

1. Para toda aresta $(u,v)$ de $G$ fazemos:
   (a) Se a soma *distância(s,u) + peso(u,v)* for menor que o valor de *distância(s,v)*, então temos um novo menor caminho de $s$ até $v$. Assim, atualizamos para $v$ o seu nodo predecessor - agora o nodo $u$ - e sua distância mínima - *distância(s,u) + peso(u,v)*.
   (b) Caso a hipótese em (a) seja falsa, proseguimos para o próximo passo de 1.

De fato, se assumirmos a ausência de ciclos negativos, podemos considerar que o menor caminho entre $s$ e um nodo $t$ de $G$ não terá nenhuma repetição de arestas ou nodos. Daí é que decorre a necessidade de executarmos o procedimento $V - 1$ vezes, pois devemos considerar a participação de todos os nodos menos a origem $s$. Caso queiramos ver a presença de ciclos negativos, basta executarmos a seqüência do relaxamento mais uma vez. Assim, se conseguirmos minimizar uma distância para um nodo $v$, estaremos repetindo arestas/nodos durante o caminho, o que indica a presença de um ciclo negativo e a inexistência de um menor caminho entre $s$ e $c$.

Assim, vemos claramente a solução usando programação dinâmica:

- Os resultados parciais - obtidos para cada uma das $V - 1$ iterações e registrados numa memória extra para cada nodo - são combinados para se conseguir uma solução ótima.
- Sempre se resolvem os problemas menores antes dos maiores - incluímos um nodo a cada iteração para construir os caminhos mínimos.

- A solução é obtida ascendentemente (*bottom-up*).

Deduz-se em 2.3 a complexidade do algoritmo e mostra-se que, ao usarmos a estratégia dinâmica, temos uma complexidade inferior do que se analizássemos todos as combinações de nodos - que residiria claramente numa tentativa com custo exponencial.

## 2.2 Pseudocódigo

```
1  func Bellman_Ford(vertices, arestas, origem)
2          //inicializacao
3          for each vertice in vertices
4                  if v == origem then v.distancia = 0;
5                  else v.distancia = infinito;
6                  v.predecessor = null;
7
8          //relaxamento
9          for i=1 to size(vertices)-1
10                 for each aresta uv in arestas
11                         u = uv.origem;
12                         v = uv.destino;
13                         if u.distancia + uv.peso < v.distancia
14                                 v.predecessor = u;
15                                 v.distancia = u.distancia + uv.peso
16
17         //checagem de ciclos negativos
18         for each aresta uv in arestas
19                 u = uv.origem
20                 v = uv.destino
21                 if u.distancia + uv.peso < v.distancia
22                         error "ciclo negativo"
```

## 2.3 Dedução da complexidade

Para deduzir a complexidade deste algoritmo, utilizaremos o pseudocódigo disponível em 2.2, tomando como operações elementares a comparação e atribuição. A análise será num caso pessimista.

A estratégia para dedução será considerar, separadamente, o custo de três partes distintas do algoritmo: *inicialização*, *relaxamento* e *checagem de ciclos negativos*. Estas podem ser vistas após as linhas com comentários homônimos.

Para *inicialização* temos um laço que executa $V$ vezes, onde V é o número de vertices do grafo fornecido. Além disso, no pior caso, faremos 1 comparação e 2 atribuições. Logo:

$$\sum_{i=1}^{V} 3 = 3 \sum_{i=1}^{V} 1 = 3V \tag{1}$$

Para *relaxamento* temos dois laços encadeados, sendo que o externo é executado *V - 1* vezes e o interno $E$ vezes, onde $E$ é o número de arestas. Novamente, pensando no pior caso, teremos 2 atribuições, 1 teste e mais 2 atribuições, o que resulta em:

$$\sum_{i=1}^{V-1} \sum_{j=1}^{E} 5 = 5 \sum_{i=1}^{V-1} \sum_{j=1}^{E} 1 = 5 \sum_{i=1}^{V-1} E = 5E \sum_{i=1}^{V-1} 1 = 5E(V-1) \tag{2}$$

Para *checagem de ciclos negativos* temos um laço que executa $E$ vezes 2 atribuições e 1 teste. Assim:

$$\sum_{i=1}^{E} 3 = 3 \sum_{j=1}^{E} 1 = 3E \tag{3}$$

Para concluir o cálculo da complexidade, basta observarmos que as 3 partes anteriormente calculadas são conjuntivas. Assim, se somarmos 1, 2 e 3 temos:

$$3V + 5EV - 5E + 3E = 3V + 5EV - 2E \tag{4}$$

Claramente este algoritmo possui ordem **O**(V E). De fato, basta lembrarmos que, para um grafo com no máximo uma aresta com origem em $u$ e destino em $v$, temos, no máximo, $V^2$ arestas. Logo, o termo que dominará a soma sera $5EV$, cujo limite assintótico superior é dado por **O**(V E).

# 3 Exemplo de Execução

Na figura em [3] (encontrado originalmente em [1]) temos um exemplo de execução do algoritmo. No passo *(a)*, temos o grafo fornecido, tendo o nodo $s$ como origem e $t$, $x$, $y$ e $z$ como destinos. Nos passos *(b)-(e)* temos a etapa

*relaxamento*, onde está representada a distância entre a origem e todos nodos (dentro dos círculos) e as arestas em destaque são aquelas que, naquele passo, são responsáveis por termos um caminho ótimo.

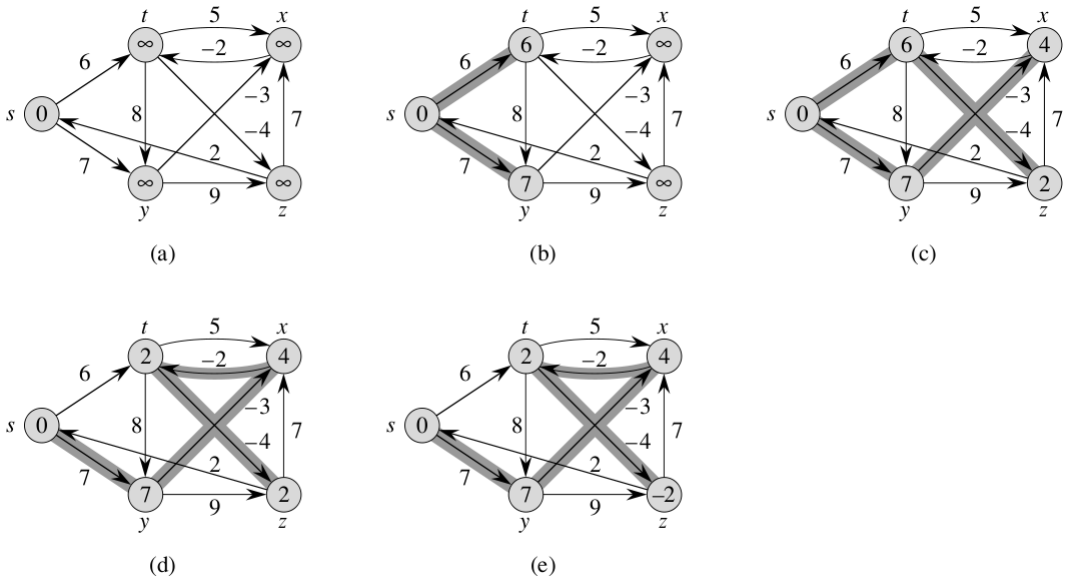


Figura 1: Passos do Algotimo de Bellman-Ford

# 4 Exemplo de implementação

```
1  #!/usr/bin/env python
2
3  distances = []
4  parents = []
5
6  def Initialize(graph, start):
7      #incializando
8      for vertex in graph.V:
9          if graph.E.has_key((start, vertex)):
10             distances[vertex] = INFINITY;
11
```

```
12      distances[start] = 0
13      parents[start] = None
14
15  def Bellman_Ford(graph, start):
16
17      Initialize(graph, start)
18
19          #relaxamento
20      for i in xrange(len(graph.V) - 1):
21          for edge in graph.E.keys():
22                  #0 representa a origem e 1 representa o destino
23              if (distances[edge[0]] + graph.E[edge]) < distances[edge[1]]:
24                  distances[edge[1]] = graph.E[edge] + distances[edge[0]]
25                  parents[edge[1]] = edge[0]
26
27      #checagem de ciclos negativos
28      for edge in graph.E.keys():
29          if (graph.E[edge] + distances[edge[0]]) < distances[edge[1]]:
30              distances[edge[1]] = graph.E[edge] + distances[edge[0]]
31              parents[edge[1]] = edge[0]
32              return "Error"
33
34      #Caso haja sucesso
35      return None
```

# 5 Referências

## Referências


[1] Thomas H. Cormen, Charles E Leiserson, Ronald L Rivest, Clifford Stein .*Introduction to Algorithms* The MIT Press 2nd edition, 2002.

[2] *Bellman-Ford Algorithm - Wikipedia, the free encyclopedia*, disponível em `http://en.wikipedia.org/wiki/Bellman-Ford_algorithm`.

[3] *Bellman-Ford Algorithm's Applications - Triangular Arbitrage*, disponível em `http://shriphani.com/blog/2010/07/02/bellman-ford-algorithms-applications-triangular-arbitrage/`.

[4] *One Source Shortest Path: The Bellman-Ford Algorithm*, disponível em `http://compprog.wordpress.com/2007/11/29/one-source-shortest-path-the-bellman-ford-algorithm/`.