PONTIFÍCIA UNIVERSIDADE CATÓLICA DE MINAS GERAIS
Instituto de Ciências Exatas e de Informática

Trabalho de Projeto e Análise de Algoritmos - Caminho Mais Curto em um Grafo com Aresta de Peso Negativo*

Luiz Gustavo Bragança dos Santos[1]
Pedro Augusto Prosdocimi Resende[2]
Pedro Henrique Silva Xavier[3]

*Artigo apresentado ao Instituto de Ciências Exatas e Informática da Pontifícia Universidade Católica de Minas Gerais como pré-requisito para obtenção do título de Bacharel em Ciência da Computação.

[1]Aluno, Ciência da Computação, Brasil, luiz.braganca@sga.pucminas.br.
[2]Aluno, Ciência da Computação, Brasil, pedro.prosdocimi@sga.pucminas.br.
[3]Aluno, Ciência da Computação, Brasil, phsxavier@sga.pucminas.br.

Sumário

Lista de Figuras

1 INTRODUÇÃO

Um problema clássico na área da Teoria de grafos trata-se de encontrar o caminho mais curto em grafos dirigidos ou não dirigidos. O presente trabalho objetiva apresentar o Algoritmo de Dijkastra e Bellman-Ford para encontrar o caminho mais curto tanto em grafos com arestas com peso positivo quanto em grafos com arestas possuindo peso negativo, respectivamente. Os algoritmos desenvolvidos neste trabalho, primeiramente verificam a existência de arestas com peso negativo, caso não exista é aplicado tanto Dijkstra quanto Bellman- Ford para resolver o problema, e caso exista é utilizado apenas Bellman-Ford em que também é feito a verificação de existência de ciclos, pois caso ocorram há comprometimento do resultado.Será também apresentado no decorrer deste trabalho, a classe grafo a qual o trabalho utiliza, análise de complexidade e testes realizados para verificação do funcionamento dos Algoritmo de Dijkastra e Bellman-Ford.

2 ALGORITMO DE DIJKSTRA

O algoritmo de Dijkstra[2], objetiva encontrar o caminho mínimo partindo- se de um vértice inicial, aos demais vértices. O algoritmo pode ser utilizado seja para grafos orientado ou não orientado, e se assemelha a outro algoritmo aplicado em grafos, Busca em largura[3]. É um algoritmo guloso, pois no momento da execução, toma a provável decisão ótima. Possui ordem de complexidade $O([m + n] \log n)$ onde m é o número de arestas e n o número de vértices.

2.1 Restrições

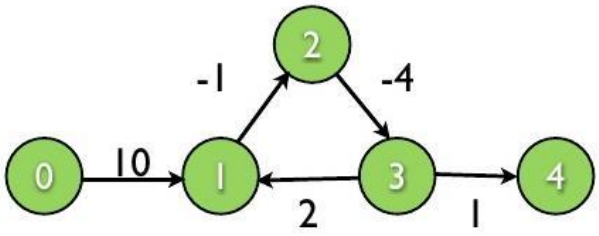
Mesmo sendo um algoritmo que solucione o problema de encontrar o menor caminho entre um vértice inicial e os demais tanto para grafos direcionado quanto não direcionado, no entanto as arestas do grafo devem ter custos não negativos.

3 ALGORITMO DE BELLMAN-FORD

O algoritmo de Bellman-Ford[4], assim como Dijkstra, encontra o menor caminho de um vértice a outro, porém resolve o problema quando existem arestas com peso negativo. O que torna-se necessário, no entanto, com esse algoritmo é encontrar ciclos negativos, caso existam. A existência de um ciclo negativo compromete o algoritmo, pois cada vez que se entra no ciclo, o somatório dos pesos diminui e então nunca alcançará um limite inferior para esse somatório. Na imagem 2 abaixo, segue um exemplo de como se tornará impossível encontrar um caminho mínimo caso existam *loops* negativos.

3.1 Restrições

Encontrar ciclos negativos, caso existam. Sendo encontrados não haverá como determinar o caminho mínimo.



No exemplo desta imagem, é possível perceber que ao tentar descobrir um caminho de custo mínimo entra os vértices 0 e 4, entramos em ciclo negativo cuja a cada passagem o valor do somatório dos pesos das arestas é decrementado, de forma que não conseguirmos uma valor final.

No pseudocódigo acima, a função recebe como parâmetro uma lista de vértices e arestas. Utilizando os atributos distância e anterior, os vértices serão modificados para armazenar o caminho mais curto.

4 IMPLEMENTAÇÃO

4.1 A Classe Grafo

Para a realização deste trabalho, foi criada uma classe Grafo, para grafos orientados, cuja representação utilizada foi matriz de adjacência. A classe Grafos, elabora o grafo da seguinte forma:

- Para N vértices, é criada uma matriz NxN;
- O Número de vértices é indicado pela variável numVertice
- Matriz[i][j] contém o peso relacionado a aresta i para j e,
- Caso não exista aresta entre i e j o peso é 0.

Apresentação de algumas partes da classe Grafo e suas respectivas funções:

```
1 //A classe Grafo possui os atributos: numVertice e grafo
2 //numVertice serve para saber a dimens o da matriz.
3 //o construtor abaixo    utilizado para construir a matriz de Adjac ncia.
4 //o desconstrutor, para liberar o espa o que foi alocado.
5 class Grafo {
```

```
6       private:
7           int numVertice;
8           int **grafo;
9
10      public:
11          bool negAresta;
12
13      Grafo(int n){
14          numVertice = n;
15          negAresta = false;
16          grafo = new int*[numVertice];
17          for(int i = 0; i < numVertice; i++){
18              grafo[i] = new int[numVertice];
19          }
20      }
21
22      ~Grafo(){
23          for(int i = 0; i < numVertice; i++){
24              delete grafo[i];
25          }
26      }
27
28      //M todo para preencher a matriz conforme entrada:
29      void criarGrafo(){
30          int tmp = 0;
31
32          for(int i = 0; i < numVertice; i++){
33              for(int j = 0; j < numVertice; j++){
34                  scanf("%d,", &(tmp));
35                  grafo[i][j] = tmp;
36                  if(tmp < 0) negAresta = true;
37          }
38      }
39  }
```

4.2 Algoritmo de Dijkstra

```
1   void dijkstra(int verticeIni,Grafo* grafo) {
2       int n_v = grafo->getnumVertice();
3       bool visitado[n_v];
4       int prox[n_v];
5       int distancia[n_v];
6
7       // inicializando os vetores.
8       for(int i = 0;i < n_v; i++)
9           distancia[i] = INT_MAX, visitado[i] = false, prox[i] = -1;
```

```
10
11      // Distancia de "verticeIni" ate ele mesmo    0
12      distancia[verticeIni] = 0;
13
14      // procurar o menor caminho para cada vertice
15      for (int count = 0; count < n_v -1; count++) {
16          //procurar a menor distancia , apos encontrar marca o vertice como
                ↪ visitado
17          int x = minDistance(distancia, visitado, n_v);
18          visitado[x] = true;
19
20          // atualizar a distancia dos vizinhos do vertices "u"
21          for (int M = 0; M < n_v; M++){
22              int peso = grafo->getPeso(x, M);
23
24              // atualizar a distancia se encontrar outro caminho menor
25              // atualizar so os vertices nao visitados, pois no caso seu
                    ↪ menor caminho ja esta definido
26              if (!visitado[M] && grafo->aresta(x, M) && distancia[x] !=
                    ↪ INT_MAX && distancia[x]+ grafo->getPeso(x, M) < distancia
                    ↪ [M]){
27                  distancia[M] = distancia[x] + grafo->getPeso(x, M);
28                  prox[M] = x;
29              }
30          }
31      }
32      printF(distancia, n_v, prox);
33  }
```

O método acima recebe como parâmetro um vértice inicial e um grafo, e utilizando 3 vetores estabelece então um caminho deste vértice inicial recebido á todos os outros. Sobre os vetores:

- O vetor visitado, é utilizado apenas quando se tem certeza que o caminho do vértice inicial, até o vértice analisado é o menor possível. Então o vértice em questão é marcado como visitado;
- O vetor de distancia, armazena as distancias do vetor inicial a todos os outros. Apenas é armazenada a distancia caso esta seja a menor;
- O vetor prox, guarda o menor caminho entre o vértice inicial a todos os outros.

O algoritmo de Dijkstra acima implementado no trabalho, utiliza também da seguinte função:

```
1  int minDistance(int dist[], bool visited[], int V)
2  {
3      // inicializar o min
4      int min = INT_MAX, min_index;
5
6      for (int v = 0; v < V; v++)
7          if (visited[v] == false && dist[v] <= min)
8              min = dist[v], min_index = v;
9
10     return min_index;
11 }
```

Esta função fornece o vértice ainda não visitado e cuja distancia seja menor do que a distancia já conhecida, ou seja dado um vértice que já se sabe sua distancia utiliza-se esta função para retornar o seu vértice adjacente ainda não visitado e como distancia menor do que a conhecida.

4.3 Algoritmo de Bellman-Ford

```
1  bool bellmanFord(int verticeIni,Grafo* grafo){
2      int n_v = grafo->getnumVertice();
3      int prox[n_v];
4      int distancia[n_v];
5      bool resposta = false;
6
7      // inicializando vetor de distancia :
8      for(int i = 0;i < n_v ;i++)
9          distancia[i] = INT_MAX, prox[i] = -1;
10
11     //distancia do vertice inicial a ele mesmo e' 0
12     distancia[verticeIni] = 0;
13     for( int i = 0; i < n_v-1; i++){
14         for( int j = 0 ; j < n_v; j++){
15             for(int k = 0; k < n_v; k++){
16                 int x, v, peso;
17                 x = j, v = k, peso = grafo->getPeso(j, k);
18
19                 // verifica se existe aresta direcionada j->k,
20                 //caso tenha menor distancia passando por estas arestas
                       ↪ atualiza vetor distancia.
21                 if(grafo->aresta(j, k) && distancia[x] != INT_MAX &&
                       ↪ distancia[x] + peso < distancia[v]){
22                     distancia[v] = distancia[x] + peso;
23                     prox[v] = x;
24                 }
25             }
26         }
27     }
28     //verificando se existe ciclo negativo,
29     //a cada passagem sera obtido um valor menor.
30     for(int j = 0; j < n_v; j++){
31         for(int k = 0; k < n_v; k++){
32             int x = j, v = k, peso = grafo->getPeso(j, k);
33             if( grafo->aresta(j, k) && distancia[x] != INT_MAX && distancia
                   ↪ [x] + peso < distancia[v]){
34                 resposta = true;
35             }
36         }
37     }
38     if(!resposta) {
39         printF(distancia, n_v, prox);
40     }
41     else{
42         printf("Existe ciclo negativo!!");
43     }
```

```
44      return resposta;
45  }
```

A função bellmanFord() acima, recebe como parâmetro o vértice inicial a partir do qual irá determinar a menor distância para todos os outros utilizando a estratégia de programação dinâmica. Utilizando-se de dois vetores prox e distancia acompanha-se o menor caminho do vértice inicial a todos os outros.

Após as inicializações, verifica-se a existência de aresta direcionada e caso esta possui menor distancia é então colocado este valor no vetor distância. Após todo esse processo é verificado a existência de ciclo negativo, caso exista a cada passagem um valor menor será obtido e a variável controladora resposta, que informará a existência de ciclo, assumirá o estado verdadeiro, caso contrario permanece como instanciada indicando que não há ciclos.

5 AMBIENTE COMPUTACIONAL UTILIZADO

Todo o trabalho foi desenvolvido utilizando o sistema operacional Linux versão 16.04. O programa foi desenvolvido no editor Sublime Text. O compilado utilizado g++.

Para compilar o programa basta:

```
1 $ g++ CaminhoM.cpp -o CaminhoM
```

Para executar o programa basta:

```
1 $ ./CaminhoM < entradaX.txt > saidaX
```

O X se refere a qual entrada padrão é utilizada, gerando então uma saída.

5.1 Entrada e saída Padrão

A entrada padrão utilizado é um arquivo de texto, com as seguintes especificações:

- A primeira linha se refere ao tamanho da matriz quadrada(vértices);
- A segunda linha ao vértice inicial utilizado para a busca;
- O restante se refere a composição da matriz utilizada.

A saída padrão é feita de acordo com o algoritmo utilizado para obter o menor caminho dado o vértice inicial. Caso utilizado o Dijkstra, é apresentado na parte superior do arquivo de saída que utilizou-se dijkstra, caso contrário Bellman, ou ambos.

6 ANÁLISE DE COMPLEXIDADE

6.1 Melhor Caso

Ocorre quando o grafo é nulo.

6.1.1 Dijkastra

Operações relevantes definidas são:

- Comparação entre elementos do vetor;
- Atribuição ao vetor.

Os acessos aos métodos da classe grafo possuem função de custo $T(n) = \theta(n)$.
Função: minDistance()

$$\sum_{i=0}^{n}(1 + 2n + \sum_{j=1}^{n} 4) = 6n^2 - 5n = \theta(n^2)$$

Análise de complexidade para o for que procura o menor caminho para cada vértice:

- Esta estrutura de repetição ocorre n-1 vezes(o numero de vértices menos 1), no melhor caso, ocorreria uma única chamada á função minDistancia(), uma única atribuição ao vetor e então a estrutura de repetição que se segue seria executada.

Análise de complexidade do melhor caso da estrutura de repetição(*for*) cuja função é atualizar a distancia dos vizinhos do vértices "u":

Esta estrutura de repetição ocorre n vezes (n = numero de vértices), sendo que no melhor caso ocorrerá 3 comparações com elementos do vetor e 3 chamadas a métodos da classe grafo.

6.1.2 Bellman-Ford

A seguintes operações foram definidas como relevantes:

- Comparação com elementos do vetor;
- Atribuição ao vetor.

Os acessos aos métodos da classe grafo possuem função de custo $T(n) = \theta(n)$.

A parte do código direcionada ao relaxamento. Para a primeira e segunda estrutura de repetição relacionada ao relaxamento. Ambas estruturas estão aninhadas, e possuem o mesmo

número de repetições, sendo que fazem duas comparações com elementos do vetor e dois acessos a métodos get e set da classe grafo :

$$T(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} 3 = 3n^2 = \theta(n^2)$$

Para a estrutura de repetição mais interna, acessa duas vezes os métodos da classe Grafo e duas comparações:

$$T(n) = \sum_{i=0}^{n} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} 3 = \theta(n^3)$$

6.2 Pior Caso

6.2.1 Dijkstra

Operações relevantes definidas são:

- Comparação entre elementos do vetor;
- Atribuição ao vetor.

Os acessos aos métodos da classe grafo possuem função de custo $T(n) = \theta(n)$.
Função: minDistance()
O número de comparações que ocorre é 2n, logo:

$$T(n) = \theta(n)$$

Análise de complexidade para o for que procura o menor caminho para cada vértice:

Esta estrutura de repetição ocorre n-1 vezes(o numero de vértices menos 1), no pior caso, ocorreria uma chamada á função minDistance(), uma atribuição ao vetor e então a estrutura de repetição que se segue seria executada.

Análise de complexidade do pior caso da estrutura de repetição(for) cuja função é atualizar a distancia dos vizinhos do vertices "u":

Esta estrutura de repetição ocorre n vezes (n = numero de vértices), sendo que no pior caso ocorrerá 3 comparações com elementos do vetor e 3 chamadas a métodos da classe grafo e duas atribuições ao vetor:

$$\sum_{i=0}^{n}(1 + 2n + \sum_{j=1}^{n} 4) = 6n^2 - 5n = \theta(n^2)$$

6.2.2 Bellman-Ford

A seguintes operações foram definidas como relevantes:

- Comparação com elementos do vetor;
- Atribuição ao vetor.

Os acessos aos métodos da classe grafo possuem função de custo $T(n) = \theta(n)$.

A parte do código direcionada ao relaxamento. Para a primeira e segunda estrutura de repetição relacionada ao relaxamento. Ambas estruturas estão aninhadas, e possuem o mesmo número de repetições, sendo que fazem duas comparações com elementos do vetor e dois acessos a métodos get e set da classe grafo, resultando na função de custo:

$$T(n) = \sum_{i=0}^{n}\sum_{j=0}^{n} 4 = 4n^2 = \theta(n^2)$$

Para a estrutura de repetição mais interna, realiza duas comparações com elementos do vetor, duas atribuições e duas chamadas a métodos da classe Grafo, resultando na seguinte função de custo:

$$T(n) = \sum_{i=0}^{n}\sum_{j=0}^{n-1}\sum_{k=0}^{n-1} 5 = \theta(n^3)$$

Análise de complexidade para classe Grafo:

Para a análise de complexidade, foram definidas as operações relevantes:

- Comparação com os elementos do vetor;
- Atribuição de valores ao vetor;
- Operação de leitura;
- Operação de exibição em tela(printF).

Os métodos get set utilizados na classe Grafo possui função de curso: $T(n) = 0 = \theta(0)$.
O método printF() fora da classe grafo possui função de custo: $T(n) = n + 1 = \theta(n)$.
O construtor e desconstrutor possui função de custo: $T(n) = n = \theta(n)$.
O método CriarGrafo() possui função de custo: $T(n) = n^2 = \theta(n^2)$.

7 TESTES REALIZADOS

Os Testes foram realizados utilizando-se a entrada padrão. Foram utilizadas quatro entradas que seguem a baixo:

Entrada 1: Corresponde a um Grafo simples. Para encontrar o menor caminho, pode-se utilizar qualquer um dos dois algoritmos.

```
6
0
0  5  0  0  9  0
0  0  12  15  0
0  0  0  3  0  0
0  0  0  0  0  0
0  2  0  0  0  4
0  0  1  0  0  0
```

Saída 1: Vértice inicial 0

```
        Dijkstra

Vertice          Distancia       Caminho
0                0               0
1                5               1 < para 0
2                14              2 < para 5 < para 4 < para 0
3                17              3 < para 2 < para 5 < para 4 < para 0
4                9               4 < para 0
5                13              5 < para 4 < para 0


        Bellman-Ford

Vertice          Distancia       Caminho
0                0               0
1                5               1 < para 0
2                14              2 < para 5 < para 4 < para 0
3                17              3 < para 2 < para 5 < para 4 < para 0
4                9               4 < para 0
5                13              5 < para 4 < para 0
```

Entrada 2: Corresponde a um Grafo simples, que possui aresta negativa. Para encontrar o caminho mínimo pode-se utilizar apenas Bellman-Ford.

```
4
0
0  5  2  8
0  0  −1  2
0  0  0  0
1  1  1  0
```

Saída 2: Vértice inicial 0

```
 Para Dijkstra existe peso negativo!


        Bellman-Ford

Vertice          Distancia       Caminho
0                0               0
1                5               1 < para 0
2                2               2 < para 0
3                7               3 < para 1 < para 0
```

Entrada 3: Corresponde a um Grafo simples, com aresta negativa e também possui um ciclo negativo, logo mesmo utilizando Bellman-Ford não encontrará o menor caminho.

```
1  8
2  0
3  0  4  4  0  0  0  0  0
4  0  0  0  0  0  0  0  0
5  0  0  0  0  4  −2  0  0
6  3  0  2  0  0  0  0  0
7  0  0  0  1  0  0  −2  0
8  0  3  0  0  −3  0  0  0
9  0  0  0  0  0  2  0  2
10 0  0  0  0  −2  0  0  0
```

Saída 3: Vértice inicial 0

```
Para Dijkstra existe peso negativo!


       Bellman-Ford
Existe ciclo negativo!!
```

Entrada 4: Corresponde a um grafo simples, sem arestas negativas.
Pode-se utilizar Dijkstra, assim como Bellman-Ford.

```
1  6
2  0
3  0  5  0  0  9  0
4  0  0  12  15  0
5  0  0  0  3  0  0
6  0  0  0  0  0  0
7  0  2  0  0  0  4
8  0  0  1  0  0  0
```

Saída 4: Vértice inicial 0

```
        Dijkstra

Vertice          Distancia       Caminho
0                0               0
1                3               1 < para 0
2                6               2 < para 4 < para 0
3                5               3 < para 1 < para 0
4                5               4 < para 0


        Bellman-Ford

Vertice          Distancia       Caminho
0                0               0
1                3               1 < para 0
2                6               2 < para 4 < para 0
3                5               3 < para 1 < para 0
4                5               4 < para 0
```

8 CONCLUSÃO

Utilizando-se os algoritmos Dijkstra e Bellman-Ford, é possível resolver problemas de caminho mínimo desde que não possuam ciclos negativos. O algoritmo de Djkstra é possível apenas resolver os problemas cujo Grafo não possua arestas negativa, isso ocorre porque Dijkstra toma a melhor decisão do momento e não volta atrás, por tanto pode ainda haver caminhos menores porém não serão encontrados. O algoritmo de Bellman-Ford é capaz de resolver problemas de caminho mínimo com arestas de peso negativo, pois possibilita a mudança da distância caso encontre um caminho menor do que o já encontrado,ao contrário de Dijkstra, porém a presença de ciclos negativos ainda é um problema.

Conclui-se também nesta trabalho, que caso o grafo possua ciclo negativo, mas esse não seja atingido, o funcionamento do algoritmo Bellman-Ford é garantido, pois ciclos negativos comprometem o resultado visto que o algoritmo pode entrar no ciclo diversas vezes e sempre obtendo valores menores.

9 ANEXOS

O programa está devidamente incluído em sua pasta com o nome "codigo", onde possui o arquivo chamado CaminhoM.cpp, o qual possui a implementação dos algoritmos: Dijkstra e Bellman-Ford, e dentro da pasta "entrada", possui os exemplos de entradas para o código, com aresta com peso positivo e negativo.

Referências

GEEKSFORGEEKS. **Bellman–Ford Algorithm | DP-23**. Disponível em: <http://www.geeksforgeeks.org/dynamic-programming-set-23-bellman-ford-algorithm/>.

MISA, Thomas J; FRANA, Philip L. An interview with edsger w. dijkstra. **Communications of the ACM**, ACM, v. 53, n. 8, p. 41–47, 2010.

PROFESSEURS. **Busca em Largura**. 2001. Disponível em: <http://www.professeurs.polymtl.ca/michel.gagnon/Disciplinas/Bac/Grafos/Busca/busca.html#Larg>.

WIKIPÉDIA. **Bellman-Ford**. 2019. Disponível em: <https://pt.wikipedia.org/wiki/Algoritmo_de_Bellman-Ford>.