PONTIFÍCIA UNIVERSIDADE CATÓLICA DE MINAS GERAIS

CAMINHO MAIS CURTO EM UM GRAFO COM ARESTA DE PESO NEGATIVO

Izabela Costa Gonçalves

Belo Horizonte Novembro de 2016

Sumário

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[1] ,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 [2].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[3],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.

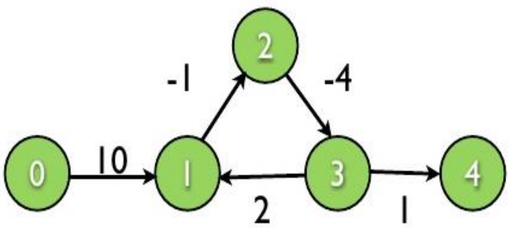

Imagem 2.Retirada:< https://pt.wikipedia.org/wiki/Algoritmo_de_Bellman-Ford>.Acesso em novembro de 2016.

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 num_vet
- 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:

```
//A classe Grafo possui os atributos: numVetrtice e grafo

//numVertice serve para saber a dimensão da matriz.
//o construtor abaixo é utilizado para construir a matriz de Adjacencia.
//o desconstrutor, para liberar o espaço que foi alocado.
class Grafo {
  private:
        int numVertice;
          int **grafo;
  public:
          bool negAresta;
    Grafo(int n){
              numVertice = n;
             negAresta = false;
              grafo = new int*[numVertice];
              for(int i=0; i<numVertice; i++){
                  grafo[i] = new int[numVertice];
              }
     }
    ~Grafo(){
         for(int i=0; i<numVertice; i++){
           delete grafo[i];
         }
     }
//Método para preencher a matriz conforme entrada:
   void criarGrafo(){
   int tmp = 0;
   for(int i=0; i<numVertice; i++){
       for(int j=0; j<numVertice; j++){
            scanf("%d,", &(tmp));
            grafo[i][j] = tmp;
             if(tmp < 0) negAresta = true;
       }
    }
 }
```

4.2 Algoritmo de Dijkstra

```
void dijkstra(int verticeIni,Grafo* grafo) {
        int n_v = grafo->getnumVertice();
        bool visitado[n_v];
        int prox[n_v];
        int distancia[n_v];

// inicializando os vetores.
        for(int i=0;i<n_v;i++)
                distancia[i] = INT_MAX, visitado[i] = false, prox[i] = -1;

// Distancia de "verticeIni" ate ele mesmo é  0
        distancia[verticeIni] = 0;

// procurar o menor caminho para cada vertice
        for (int count = 0; count < n_v -1; count++) {

//procurar a menor distancia , apos encontrar marca o vertice como visitado
                int x = minDistance(distancia, visitado, n_v);
                visitado[x] = true;
 // atualizar a distancia dos vizinhos do vertices "u"
                for (int M = 0; M < n_v; M++){
                        int peso = grafo->getPeso(x, M);

// atualizar a distancia se encontrar outro caminho menor
// atualizar so os vertices nao visitados, pois no caso seu menor caminho ja
esta definido
                        if (
        !visitado[M] && grafo->aresta(x, M) && distancia[x] != INT_MAX &&
distancia[x]+ grafo->getPeso(x, M) < distancia[M]
                        ){

                                distancia[M] = distancia[x] + grafo->getPeso(x, M);
                        prox[M] = x;

                        }
                }

        }

printF(distancia, n_v, prox);

}
```

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

- O vetor visitado, é utilizado apenas quando se tem certeza que o caminho do vértice incial, ate 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 incial 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:

```
int minDistance(int dist[], bool visited[], int V)
{
  // inicializar o min
        int min = INT_MAX, min_index;

        for (int v = 0; v < V; v++)
                if (visited[v] == false && dist[v] <= min)
                        min = dist[v], min_index = v;

        return min_index;
}
```

Esta função fornece o vértice anda 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

```
bool bellmanFord(int verticeIni,Grafo* grafo){
      int n_v = grafo->getnumVertice();
       int prox[n_v];
       int distancia[n_v];
       bool resposta = false;
// inicializando vetor de distancia :
       for(int i=0;i <n_v ;i++)
                 distancia[i] = INT_MAX, prox[i] = -1;
//distancia do vertice inicial a ele mesmo Ã© 0
         distancia[verticeIni] = 0;

       for( int i=0; i< n_v-1;i++){
```

```
      for( int j=0 ; j <n_v; j++){
            for(int k=0; k<n_v ;k++){
                  int x, v, peso;
                 x = j, v = k, peso = grafo->getPeso(j, k);

// verifica se existe aresta direcionada j->k,
//caso tenha menor distancia passando por estas arestas atualiza vetor
distancia.
               if( grafo->aresta(j, k) && distancia[x] != INT_MAX &&
distancia[x] + peso < distancia[v]){
                  distancia[v] = distancia[x] + peso;
                  prox[v] = x;
                }
            }


         }
      }
//verificando se existe ciclo negativo,
//a cada passagem sera obtido um valor menor.

    for(int j=0; j<n_v;j++){

         for(int k=0; k<n_v;k++){
               int x = j, v = k, peso = grafo->getPeso(j, k);
               if( grafo->aresta(j, k) && distancia[x] != INT_MAX &&
distancia[x] + peso < distancia[v]){

                  resposta =  true;
               }
         }
      }

      if(!resposta) {

         printF(distancia, n_v, prox);
      }
     else{
       printf(" Existe ciclo negativo!!");

     }

      return resposta;
}
```

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 15.05. O programa foi desenvolvido no editor Sublime Text. O compilado utilizado G++.
Para executar o programa basta: g++ CaminhoM.cpp <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,e
- 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.

Dijkastra

Operações relevantes definidas são:

- Comparação entre elementos do vetor, e
- Atribuição ao vetor.

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

Função: minDistance

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

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 vertices "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.

Bellman-Ford

A seguintes operações foram definidas como relevantes:

- Comparação com elementos do vetor, e
- Atribuição ao vetor.

Os acessos aos métodos da classe grafo possuem função de custo T(n) = θ(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 :

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

Para a estrutura de repetição mais interna :

Acessa duas vezes os métodos da classe Grafo e duas comparações:

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

6.2 Pior caso

Dijkstra

Operações relevantes definidas são:

- Comparação entre elementos do vetor, e
- Atribuição ao vetor.

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

Função: minDistance

O número de comparações que ocorre é 2n, logo :

T(n) = $\theta(\mathrm{n})$.

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 minDistancia, 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:

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

Bellman-Ford

A seguintes operações foram definidas como relevantes:

- Comparação com elementos do vetor, e
- Atribuição ao vetor.

Os acessos aos métodos da classe grafo possuem função de custo T(n) = θ(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 :

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

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:

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

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 = θ(0).

O método printF fora da classe grafo possui função de custo: T(n) = n+1 = θ(n).

O construtor e desconstrutor possui função de custo: T(n) = n = θ(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 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.

```
8
0
0 4 4 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 4 -2 0 0
3 0 2 0 0 0 0 0
0 0 0 1 0 0 -2 0
0 3 0 0 -3 0 0 0
0 0 0 0 0 2 0 2
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.

```
6
0
0 5 0 0 9 0
0 0 12 15 0 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 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

Ulitizando-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. Referências Bibliográficas

[1]- Dijkstra, Edsger; Thomas J. Misa, Editor (2010-08). «An Interview with Edsger W. Dijkstra». Communications of the ACM [S.l.: s.n.] **53** (8): 41–47.

[2]-Busca em Largura. Disponível em: <http://www.professeurs.polymtl.ca/michel.gagnon/Disciplinas/Bac/Grafos/Busca/busca.html#Larg> Acessado em: Novembro de 2016.

[3]- Belman-Ford. In Wikipedia. Disponível em: <https://pt.wikipedia.org/wiki/Algoritmo_de_Bellman-Ford> Acessado em:Novembro de 2016.

[4] Dynamic Programming In geeksforgeeks. Disponível em: http://www.geeksforgeeks.org/dynamic-programming-set-23-bellman-ford-algorithm/> Acessado em :Novembro de 2016.