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

Trabalho Prático - Caminho mais curto em um Grafo com Aresta de Peso Negativo*

Gabriel Lopes Ferreira[1]
Giovanni Carlos Guaceroni[2]
Leonardo Viana Won Dollinger[3]

*Trabalho apresentado a disciplina de Projeto de Análise de Algoritmos da Pontifícia Universidade Católica de Minas Gerais.

[1]Aluno de graduação do Curso de Ciência da Computação, Brasil – gabriel.ferreira.1116674@sga.pucminas.br.
[2]Aluno de graduação do Curso de Ciência da Computação, Brasil – gcguaceroni@sga.pucminas.br.
[3]Aluno de graduação do Curso de Ciência da Computação, Brasil – ldollinger@sga.pucminas.br.

1 INTRODUÇÃO

Neste trabalho prático, três problemas serão abordados, são estes, o problema do caminho mais curto em um grafo com peso não negativo, determinar se existe um ciclo negativo em um grafo e por fim, o caminho mais curto em um grafo com aresta negativa sem ciclo negativo usando de programação dinâmica.

O problema do caminho mais curto em um grafo se baseia em, dado um grafo com pesos nas arestas, encontrar o menor caminho (caminho de menor peso) entre dois vértices x e y. Muitas vezes o peso representa a distância entre os vértices, e por esse motivo, este problema passou a ser conhecido como problema do caminho mínimo. Tal problema, pode ser facilmente resolvido usando do algoritmo de Dijkstra.

O algoritmo de Dijkstra funciona da seguinte maneira: Dado um grafo G com custos positivos nos arcos e um vértice s, o algoritmo, faz crescer uma subárvore presente em G, a partir do vértice s, até que ela conecte todos os vértices que estão ao alcance de s. Assim, ao final da execução do algoritmo, a subárvore torna-se geradora.

O problema do caminho mais curto em um grafo pode acabar se complicando quando existem arestas com peso negativo, já que nesses casos é possível que existam ciclos negativos. Para determinar se tais ciclos existem e para encontrar o menor caminho, usaremos o algoritmo de Bellman-Ford.

Ao receber um grafo G com custos nos arcos e um vértice s, o algoritmo de Bellman-Ford procura um caminho de custo mínimo de G a partir de um vértice s. Para simplificar a discussão do algoritmo, suporemos que todos os vértices de G estão ao alcance de s. O algoritmo calculará o vetor de pais de um caminho de custo mínimo ou anunciará que o grafo tem um ciclo negativo.

2 IMPLEMENTAÇÃO E ANÁLISE DE COMPLEXIDADE

2.1 Algoritmo de Dijkstra

O Algoritmo de Dijkstra resolve o problema do menor caminho entre dois vértice utilizando de uma estratégia gulosa, aonde ele espera que um ótimo local o levará a um ótimo global. Utilizamos a implementação do algoritmo de Dijkstra disponível no livro do Cormen. O código completo está diponível no arquivo Dijkstra.cpp.

O algoritmo usa uma PriorityQueue que foi implementada utilizando dois vetores, por ser um programa com proposito educacional, optamos por esse implementação por ser mais fácil e pratica de se implementar. Abaixo podemos ver os dois vetores, onde o primeiro é a fila propriamente dita, e o segundo é um vetor auxiliar para guardarmos os valores das chaves que foram removidas da fila.

```
    private:
        vector<int> priorityQueue;
        vector<int> valuesOfRemovedKeys;
```

Abaixo temos a função getMinKey() da PriorityQueue, como foi dito antes utilizamos um vetor para representar a fila, onde realizamos uma busca sequencial para encontrar o menor valor. Considerando a quantidade de operações if realizadas a complexidada do procedimento abaixo no melhor e pior caso é T(n) = n, ou seja ela é O(n)

```
    int getMinKey(){
        int min  = INT_MAX;
        for(int i = 0;i< priorityQueue.size(); i++){
            if(priorityQueue[i]<min&&priorityQueue[i]!=-1){
                min = priorityQueue[i];
            }
        }
        return min;
    }
```

A função a seguir extractMinKey(), remove logicamente a chave de menor valor, colocando o valor -1 na posição dela e ao mesmo tempo salvamos no nosso vetor auxiliar valuesOfRemovedKeys o antigo valor na posição removida. Por fim retornarmos a posição onde estava o menor valor. Considerando a quantidade de operações if realizadas temos que no pior caso T(n) = 2n, e no melhor caso T(n) = 2n - 1. Ou seja ela é O(n)

```
    int extractMinKey(){
        int min  = 0;
        for(int i = 0;i< priorityQueue.size(); i++){
            //Esse if e necessario para caso a chave 0 ja tenha sido
                removida da fila
            //somente entraremos ele caso a chave 0 tenha sido removida
            if(priorityQueue[min]==-1){
                min = i;
            }
            else if(priorityQueue[i]<priorityQueue[min]&&priorityQueue[i
                ]!=-1){
                min = i;
            }
        }
        valuesOfRemovedKeys[min] = priorityQueue[min];
        priorityQueue[min] = -1;
        return min;
    }
```

A função abaixo, descreseKey(), altera o valor na posição "vertex"para o valor passado pelo parâmetro "value". Se considerarmos como operação relevante o a escrita no vetor, teremos que T(n) = 1, ou seja O(1)

```
void decreaseKey(int vertex, int value){
    priorityQueue[vertex] = value;
}
```

A função abaixo, keyIsOnQueue(), retorna um booleano que representa se a chave passado por parâmetro já foi removida ou não da fila, como remoção realizada é lógica verificamos se o valor é diferente de -1. Considerando a comparação como operação relevante teremos que T(n) = 1, ou seja O(1)

```
bool keyIsOnQueue(int vertex){
    return priorityQueue[vertex]!=-1;
}
```

A função abaixo, insertKey(), insere na fila o valor passado por parâmetro, ao mesmo tempo fazemos uma inserção no vetor que guarda o valor das chaves removidas, como o valor inserido nesse segundo vetor não interessa no momento, passamos o valor fixo de 0. Considerado a inserção no vetor como operação relevante temos que T(n) = 2, logo O(1).

```
void insertKey(int value){
    priorityQueue.push_back(value);
    valuesOfRemovedKeys.push_back(0);//inicializa o vetor de chaves
        removidas
}
```

Na função abaixo, getKeyValue() retornamos o valor da chave passada por parâmetro para isso fazemos uma chamada do metodo keyIsOnQueue(), e verificamos se a chave está na fila, caso esteja retornamos o seu valor da fila, caso contrário retornamos o valor do array de removidos. Considerando a operação relevante como a verificação se o elemento está na fila, logo temos T(n) = 1, ou seja O(n) = 1

```
int getKeyValue(int vertex){
    return keyIsOnQueue(vertex) ? priorityQueue[vertex] :
        valuesOfRemovedKeys[vertex];//se a chave j  foi removida,
        retorna seu antigo valor
}
```

No método abaixo retornarmos se a fila está vazia ou não, como fazer uma remoção lógica da fila, para checarmos isso é necessário fazermos N chamadas ao método keyIsOnQueue,

caso algum retorne verdadeiro, retornarmos falso já que a chave em questão está na fila. onsiderando a operação relevante como a verificação se o elemento está na fila, logo temos no melhor caso T(n) = 1, e no pior caso T(n) = n.

```
bool isEmpty(){
    for(int i=0;i<priorityQueue.size() ; i++){
        if(keyIsOnQueue(i)){
            return false;
        }
    }
    return true;
}
```

Construtor da classe Grafo, nele atribuímos ao atributo numOfVertex, o parâmetro recebido +1, em seguida alocamos um array de ponteiros e seguida percorrendo esse array e para cada posição alocando um array de inteiros. Se considerarmos a alocação de memoria como operação relevante temos no melhor e pior caso T(n) = n+1, logo O(n).

```
Graph(int numOfVertex){
    this->numOfVertex = numOfVertex+1;
    this->adjMatrix = new int*[this->numOfVertex];
    for(int i = 0 ; i < this->numOfVertex ; i++){
        this->adjMatrix[i] = new int[this->numOfVertex];
        memset(adjMatrix[i],0,sizeof(int)*this->numOfVertex);
    }
}
```

A função abaixo insere uma aresta bi direcionada no grafo, que vai de v1 até v2 e de v2 até v1 com peso weight, considerando a atualização dos valores como operação relevante, temos que T(n) = 2, logo O(1)

```
void insertBiEdge(int v1 , int v2,int weight){
    this->adjMatrix[v1][v2] = weight;
    this->adjMatrix[v2][v1] = weight;
}
```

A função abaixo insere uma aresta direcionada no grafo, que vai de v1 até v2 com peso weight, considerando a atualização dos valores como operação relevante, temos que T(n) = 1, logo O(1)

```
void insertEdge(int v1,int v2,int weight){
    this->adjMatrix[v1][v2] = weight;
}
```

A função abaixo executa o algoritmo de Dijkstra, a função recebe dois valores inteiros, indicando o vértice de inicio(start) e o vértice destino(end), e retorna a menor distância entre os dois.

```
int dijkstra(int start, int end){
    PriorityQueue priorityQueue;
    for(int i = 0; i< numOfVertex ; i++){
        priorityQueue.insertKey(INT_MAX);
    }
    priorityQueue.decreaseKey(start,0);
    int sizePath = 0;
    while(!priorityQueue.isEmpty()&&priorityQueue.keyIsOnQueue(end)){
        int u = priorityQueue.extractMinKey();
        for(int i=0 ; i< this->numOfVertex ; i++){
            int weightUI = adjMatrix[u][i];
            int keyValueI = priorityQueue.getKeyValue(i);
            int keyValueU = priorityQueue.getKeyValue(u);
            int possiblePath = keyValueU+weightUI;
            if(i!=u&&priorityQueue.keyIsOnQueue(i)&&keyValueI>
                possiblePath&&weightUI!=0){
                 priorityQueue.decreaseKey(i,possiblePath);
            }
        }
    }
    return priorityQueue.getKeyValue(end);
}
```

Em seguida iremos dividir a função de vários pontos importante para podermos calcular a complexidade da mesma.

Inicializamos a fila de prioridade para cada vertice no grafo, com valor inifinito para cada um. Como a funcao realiza n chamadas ao insertKey e o mesmo tem função de complexida igual a T(n) = 2 temos que a função de complexidade deste for será de T(n) = 2n.

```
for(int i = 0; i< numOfVertex ; i++){
    priorityQueue.insertKey(INT_MAX);
}
```

Dentro do while extraímos a menor chave, com o custo de T(n) = 2n no pior caso.

```
 int u = priorityQueue.extractMinKey();
```

Em seguida executamos um for que irá repetir N vezes, se considerarmos como operação relevante o if dentro do for, sempre teremos o custo de 1 execução do if. Logo o custo do for será T(n) = n.

```
1       for(int i=0 ; i< this->numOfVertex ; i++){
              int weightUI = adjMatrix[u][i];
3             int keyValueI = priorityQueue.getKeyValue(i);
              int keyValueU = priorityQueue.getKeyValue(u);
5             int possiblePath = keyValueU+weightUI;
              if(i!=u&&priorityQueue.keyIsOnQueue(i)&&keyValueI>possiblePath
                 &&weightUI!=0){
7                priorityQueue.decreaseKey(i,possiblePath);
              }
9         }
```

A cada execucao do while fazemos a chamada de dois métodos, o keyIsOnQueue, que tem uma complexidade de T(n) = 1, e o isEmpty(), como o isEmpty() com o passar do tempo vai tendo sua complexidade sendo aumentada, e não conseguimos prever exatamente o valor, consideraremos sempre o pior caso ou seja T(n) = n. Por fim o while é encerrado caso o valor logico seja verdadeiro. Fato que corre quando o vértice de destino foi alcançado e o mesmo for removida da fila. Dentro do while, temos o custo de n+2n+n totalizando um custo de 5n a cada execução, como no pior caso o while executa n vezes, temos que o custo dele será de, $n * (5n)$, logo $5n^2$

```
1     while(!priorityQueue.isEmpty()&&priorityQueue.keyIsOnQueue(end)){
         int u = priorityQueue.extractMinKey();
3        for(int i=0 ; i< this->numOfVertex ; i++){
              int weightUI = adjMatrix[u][i];
5             int keyValueI = priorityQueue.getKeyValue(i);
              int keyValueU = priorityQueue.getKeyValue(u);
7             int possiblePath = keyValueU+weightUI;
              if(i!=u&&priorityQueue.keyIsOnQueue(i)&&keyValueI>possiblePath
                 &&weightUI!=0){
9                priorityQueue.decreaseKey(i,possiblePath);
              }
11        }
    }
```

Por fim ao somarmos todas as complexidades parciais da função,chegamos que $T(n) = 5n^2 + 2n$, ou seja a função será $O(n^2)$.

2.2 Bellman-Ford

Ao contrário do algoritmo de Dijkstra, o algoritmo Bellman-Ford pode ser aplicado para grafos contendo arestas com pesos negativos para tentar encontrar o menor caminho entre dois vértices. Entretanto, se o grafo tiver um ciclo negativo, então, os caminhos mais curtos para

alguns vértices podem não existir já que o peso do caminho mais curto deve ser igual a menos infinito. entretanto, o algoritmo pode ser modificado para sinalizar a presença de um ciclo negativo e/ou encontrar o menor caminho entre dois vértices.

Segue abaixo a implementação:

Construtor da classe Grafo, nele atribuímos ao atributo numOfVertex, o parâmetro recebido, em seguida alocamos um array de ponteiros e seguida percorrendo esse array e para cada posição alocando um array de inteiros. Se considerarmos a alocação de memória como operação relevante temos no melhor e pior caso T(n) = n+1, logo O(n).

```
Graph::Graph(int numOfVertex){
    this->numOfVertex = numOfVertex;
    this->adjMatrix = new int*[this->numOfVertex];
    dist = new int[this->numOfVertex];
    for(int i = 0 ; i < this->numOfVertex ; i++){
        this->adjMatrix[i] = new int[this->numOfVertex];
        memset(adjMatrix[i],0,sizeof(int)*this->numOfVertex);
    }
}
```

Esta é a função de adicionar uma aresta ao nosso grafo, como o nosso grafo é ponderado na verdade adicionamos o peso na nossa matriz de adjacência além disso adicionamos uma struct de Edge ao nosso vetor de Edges para o funcionamento do Bellman-Ford. Se consideramos a operação relevante a atribuição na nossa matriz temos no melhor e pior caso T(N) = 1, logo O(1);

```

void Graph::addEdge(int v1, int v2,int weight){
    adjMatrix[v1][v2] = weight;
    Edge edge;
    edge.src = v1;
    edge.dest = v2;
    edge.weight =  weight;
    edges.push_back(edge);
}
```

Essa função faz a inicialização do vetor de distâncias, ele recebe um parâmetro "src"onde todas as distancias para a origem terão valor "infinity"e a distancia para ela mesmo será 0. Considerando a operação relevante a atribuição do valor "infinity"ao nosso vetor temos no melhor e no pior caso T(N) = n+1 , logo O(n).

```
void Graph::initilizeDist(int src){
    for(int i = 0;i<this->numOfVertex;i++){
        dist[i] = infinity;
    }
    dist[src] = 0;
}
```

Essa função consiste em verificar se pode ser encontrado um caminho mais curto v(do que aquele encontrado até o momento), passando pelo vértice corrente u. Em caso positivo ele atualiza dist(v). Considerando o pior e melhor caso temos T(n) = 1 ou seja O(1).

```
void Graph::relax(int u, int v){
    int weightUV = adjMatrix[u][v];
    if(dist[v]>dist[u] + weightUV){
        dist[v] = dist[u] + weightUV;
    }
}
```

Esta função realiza de fato o Bellman-Ford que consiste em achar a menor distancia entre dois vértices podendo conter arestas com pesos negativos. Tendo como operação relevante a realização da função relaxa, temos T(n) = (n*m - m) sendo m = tamanho do vector de edges. Ou seja O(n*m)

```
int Graph::bellmanFord(int src, int dest){
    initilizeDist(src);
    for(int i = 0;i<this->numOfVertex-1;i++){
        for(Edge &edge : edges){
            relax(edge.src,edge.dest);
        }
    }
    return dist[dest];
}
```

A função abaixo, tem como finalidade, encontrar ciclos negativos no dado grafo de aresta negativa. Tomando como operação relevante, o teste entre as distancias ("if(dist[v]>dist[u] + weightUV)"), temos o pior e melhor caso como T(n) = n + (n*m - m) ou seja O(n*m).

```
1
bool Graph::hasNegativeCycle(){
3     bellmanFord(0);
      for(Edge &edge : edges){
5         int u = edge.src;
          int v = edge.dest;
7         int weightUV = adjMatrix[u][v];
          if(dist[v]>dist[u] + weightUV){
9             return true;
          }
11    }
      return false;
13 }
```

3 TESTES

3.1 Ambiente Computacional

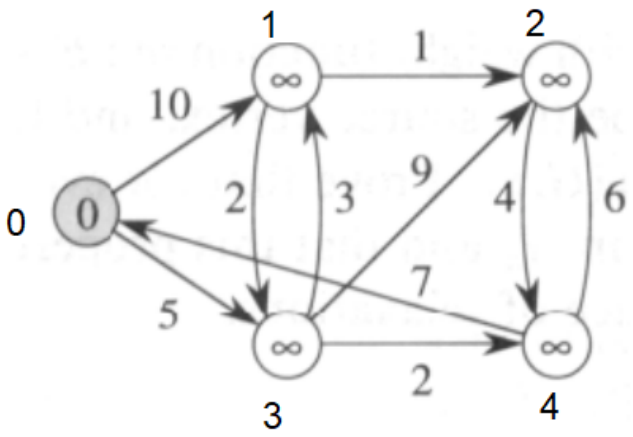
Para os testes foi utilizado uma maquina com o Sistema Operacional Windows 10 Education 64bit, com um processador Intel(R) Core(TM) i5-3470S CPU @ 2.90GHz, com memória RAM de 16GB DDR3 1600MHz. Os testes foram realizados utilizando o WSL(*Windows Subsystem for Linux*) com o sistema operacional Ubuntu 20.04.1 LTS. Foi utilizado o utilitário do *make* para criação de scripts para facilitar o build e a execução do programa;

3.2 Algoritmo de Dijkstra

Nesta subseção será detalhada a execução do algoritmo de Dijkstra. Para compilar o código utilize o comando *make build_Dijkstra*, para executar utilize *make run_Dijkstra* ou *make run_Dijkstra_Default* para executar com a entrada exemplo.

Para os testes foi utilizado uma entrada simples com 5 vértices e 10 arestas. Essa entrada representa o seguinte grafo:

Figura 1 – Grafo utilizado como entrada



Segue abaixo a entrada utilizada:

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

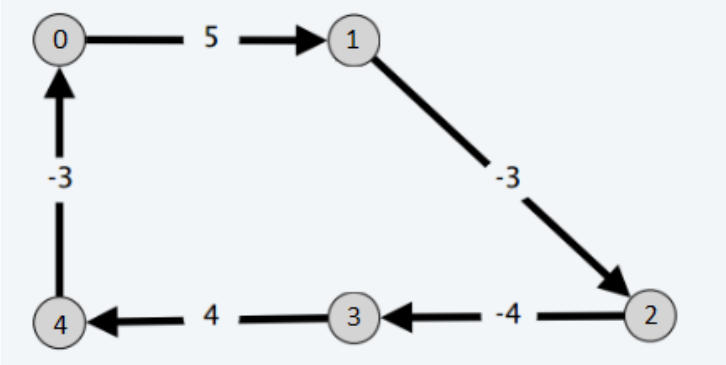
A entrada começa com dois valores V, E representando os vértices e as arestas do grafos respectivamente. Em seguida deve-se entrar com E linhas, contendo três valores, V U W indicando que há uma resta do vértice V ao vértice U com peso W. Em seguida deve-se entrar com dois valores Start e End, representando o vértice de começo e o vértice destino.

Para a entrada utilizada a saída do programa foi o valor 7. Indicando que a distância entre o vértice 0 e o vértice 3 é 7.

3.3 Bellman-Ford Detecção ciclo negativo

Nesta subseção será detalhada a execução do algoritmo de Bellman-Ford para detecção de ciclo negativo em um grafo. Para compilar o código utilize o comando *make build_Bellman_Ford_Negative_Cycle*, para executar utilize *make run_Bellman_Ford_Negative_Cycle* ou *make run_Bellman_Ford_Negative_Cycle_Default* para executar com a entrada exemplo.

Figura 2 – Grafo de ciclo negativo



5 5
0 1 5
1 2 -3
2 3 -4
3 4 4
4 0 -3

A entrada começa com dois valores V, E representando os vértices e as arestas do grafos respectivamente. Em seguida deve-se entrar com E linhas, contendo três valores, V U W indicando que há uma resta do vértice V ao vértice U com peso W.

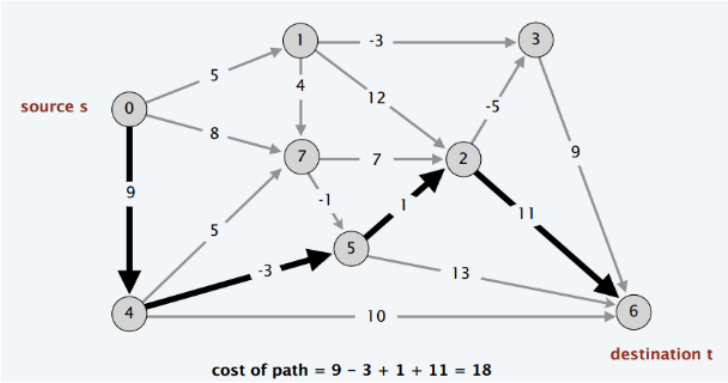
A saída falará se existe ou não um ciclo negativo, mostrando respectivamente a mensagem "Input graph has Negative Cycle"ou "Input graph do not have Negative Cycle".

3.4 Bellman-Ford cálculo de distância com aresta negativa

Nesta subseção será detalhada a execução do algoritmo de Bellman-Ford. Para compilar o código utilize o comando *make build_Bellman_Ford_Distance*, para executar utilize *make run_Bellman_Ford_Distance* ou *make run_Bellman_Ford_Default* para executar com a entrada exemplo.

Para os testes foi utilizado uma entrada simples com 8 vértices e 16 arestas. A entrada representa o seguinte grafo:

Figura 3 – Grafo de aresta negativa



Segue abaixo a entrada utilizada:

8 16
0 4 9
0 1 5
0 7 8
1 3 -3
1 7 4
1 2 12
2 6 11
2 3 -5
3 6 9
4 5 -3
4 7 5
4 6 10
5 2 1
5 6 13
7 2 7
7 5 -1
0 6

A entrada começa com dois valores V, E representando os vértices e as arestas do grafos respectivamente. Em seguida deve-se entrar com E linhas, contendo três valores, V U W indicando que há uma resta do vértice V ao vértice U com peso W. Em seguida deve-se entrar com dois valores Start e End, representando o vértice de começo e o vértice destino.

Para a entrada utilizada a saída do programa foi o valor 11. Indicando que a distância entre o vértice 0 e o vértice 6 é 11 (0-4 = 9, 4-5 = -3, 5-2 = 1, 2-3 = -5 e 3-6 = 6).

4 CONCLUSÃO

Neste documento, foi falado sobre o problema do caminho mais curto em um grafo com peso não negativo, em um grafo de aresta negativa sem ciclo negativo e sobre determinar se

existe um ciclo negativo em um grafo. Além disso, foi também foi abordada a implementação e análise de complexidade dos algoritmos de Dijkstra e Bellman-Ford visando a solução dos problemas.

Como citado, na busca da solução de tais problemas, implementamos e dissecamos os algoritmos de Dijkstra e Bellman-Ford, para assim, compreendermos melhor sobre o problema em si e entendermos também a complexidade de cada função dos dados algoritmos. Após analisarmos a complexidade dos algoritmos, percebemos que tanto o Dijkstra quanto o Bellman-Ford são soluções viáveis, já que ambas são funções polinomiais, sendo sendo o Dijkstra $O(n^2)$. e o Bellman-Ford $O(n * m)$. sendo n o número de arestas e m o numero de vértices.

No desenvolvimento do artigo, encontramos pequenas dificuldades em entender o funcionamento do algoritmo de Bellman-Ford e implementar o mesmo, adaptando-o para atender separadamente ao problema do caminho mais curto em um grafo de aresta negativa sem ciclo negativo e para determinar se existe um ciclo negativo em um grafo de aresta negativa. Outro problema enfrentado, foram os testes, havendo certa dificuldade de pensar no tamanho ideal para as entradas dos problemas e nas medições de performance.

REFERÊNCIAS

ALGORITMO de Bellman-Ford. Disponível em: <https://cp-algorithms-brasil.com/grafos/bellmanford.html>.

FEOFILOFF, Paulo. Algoritmo de dijkstra. Março 2020.

PROBLEMA do Caminho Mínimo. Disponível em: <http://www.dainf.ct.utfpr.edu.br/petcoce/wp-content/uploads/2011/06/DiscreteMathFloydWarshall.pdf>.