IMPLEMENTAÇÃO DO ALGORITMO DE DIJKSTRA

Data: 27 de outubro de 2017

Matheus Francisco Batista Machado Matrícula : 14202492

Disciplina: Projeto e Análise de Algoritmos

---

1 Introdução

Em muitas das vezes em problemas que há uma sequencia de passos sendo cada passo há diversas alternativas, busca usar algoritmos baseado em otimização para solucionar estes problemas.A técnica que é usada para resolver este problema é chamada de algoritmo guloso, ou ganancioso, pois em cada passo faz uma escolha que é considerada ótima no momento, baseado apenas nas informações do momento.Também podemos ressaltar que algoritmos guloso nem sempre chegam a uma solução ótima porém são muito rápidos e eficientes.

Um dos algoritmos cujo é guloso é chamado de Algoritmo de Dijkstra é um dos algoritmos concebido pelo holandês Edsger Dijkstra em 1965 e publicado em 1959, usado para encontrar o menor caminho em um grafo dirigido ou não dirigido com pesos não negativos, ou seja baseado em técnica gulosa ira encontrar o menor caminho de uma aresta até outras dado que entre as aresta contenha um custo não negativo.

2 Algoritmo de Dijkstra

O algoritmo de Dijkstra é um dos algoritmos que calcula o caminho de custo mínimo entre vértices de um grafo. Escolhido um vértice como raiz da busca, este algoritmo calcula o custo mínimo deste vértice para todos os demais vértices do grafo.

- O algoritmo funciona para grafos direcionados e não direcionados.
- Arestas com custos não negativos ou seja arestas negativas o algoritmo não funciona.

Passos para implementar algoritmo

- Numerar os vértices do grafo
- Atribuir os valores o vértice de saída ”raiz” recebe 0 e os demais vértices infinito
- Começando na raiz (u), consultar todo (v) adjacente a (u). Calcula-se sua distância entre os vértice, somar o peso das arestas

- Dentre todos os vértices adjacentes a (u) "raiz", escolher o que tem menor distância e recalcular os vértices adjacentes a ele.
- Escolha um vértice dentro os não escolhidos.(repita o passo anterior)
- Escolha o de menor distância e busca esta finalizada.

Algoritmo em pseudo-código

```
1  – Para todo u E V
2  –     dista(u) = infinito
3  –      prev(u) = null
4  – dist(s) =0
5  – H = construir-fila(V) /*Fila com prioridade contendo os vertices de G */
6  – /*(prioridade menor distancia a seus filhos)*/
7  – Enquanto H e nao-vazio faca
8  –     u= extrair-min(H)
9  –             Para todas arestas (u,v) pertence E /* arestas u->v */
10 –      /*sao ordenadas em ordem decrescente das distancias*/
11 –             se dist(v) > dist(u) + l(u,v)
12 –             dist(v) = dist(u) + l(u,v)
13 –             prev(v) = u
14 –             diminuir-chave(H,v) /*v ira ser recolocado*/
15 –                     /*na fila com sua prioridade ajusatada*/
```

2.1 Análise de Complexidade

Podemos analisar que nas linhas 1, 2, 3 temos um custo c1 e c2 constante **O(**$|V|$**)** vezes.

```
1  – Para todo u E V
2  –     dista(u) = infinito
3  –      prev(u) = null
```

Na linha 4 o custo é c3 constante é executado 1 vez, já na linha 5 iremos chamar de X = **Custo de inserir elemento na fila executado** $|V|$ **vezes**.

```
5  – H = construir-fila(V) /*Fila com prioridade contendo os vertices de G */
6  – /*(prioridade menor distancia a seus filhos)*/
```

Na linha 7 **O(**$|V|$**)** vezes, na linha 8 iremos chamar **custa Y executados O(**$|V|$**) vezes**

```
7  – Enquanto H e nao-vazio faca
8  –     u= extrair-min(H)
9  –             Para todas arestas (u,v) pertence E /* arestas u->v */
10 –      /*sao ordenadas em ordem decrescente das distancias*/
```

, na linha 9 **O(**$|E|$**)** vezes , nas linhas 11 , 12 13, temos c4,c5,c6 custos constantes. Na linha 14 **custo Z executada O(**$|V|+|E|$**) vezes**.

```
14 –             diminuir-chave(H,v) /*v ira ser recolocado*/
15 –                     /*na fila com sua prioridade ajusatada*/
```

Somando temos então **O(**$|V|$**)+ c3 + X.(O(**$|V|$**))+ Y.O(**$|V|$**) + (Z+c4+c5+c6).O(**$|E|+|V|$**)** , temos então **O(**$|V|$**)+X+Y.O(**$|V|$**)+Z.O(**$|E|+|V|$**)**. Com isso podemos perceber que a análise de complexidade depende da implementação da fila com prioridades.

2.2 Fila de prioridade

2.3 Vetor

Analisando a fila de prioridade para um vetor, neste caso temos que nas linhas 4-6 consome tempo proporcional a n e na linha 8 extrair-min no pior caso temos que olhar todo o vetor logo consome um tempo n. Nas linhas 7-14 será repetido no máximo n vezes, pois um vértice ira sair da fila em cada iteração, logo o consumo de tempo é **O(n^2)**.

- Custo X= insere elemento no vetor O(1)
- Custo Y = extrai min(H) = O(V)
- Custo Z = ajustar as distancias O(1)

logo para nosso algoritmo visto na seção acima

$$O(|V|) + X.O(|V|) + Y.O(|V|) + Z.O(|V| + |E|) = O(|V|) + O(|V|) + O(|V|)O(|V|) + O(|E| + |V|) = O(|V|^2 + |E|)$$

O($|V|^2$) já que $|E| <= |V|^2$.

2.4 Heap

Também pode ser estruturado uma fila de prioridade com um min-heap , com este caso as linhas 4-6 consomem um tempo proporcional a **nlgn** com isso cada execução de extrair o menor ira consumir um tempo constante.
E a função que ira diminuir o valor das arestas consomem tempo proporcional a **lg n**. As linhas 7-8 serão executadas O(n) e as linhas 9-11 é limitadas pelo número de arcos tendo um consumo **O(mlgn)** ou seja o algoritmo de Dijkstra é **O(n lg n + n + mlg n = O((n+m)lgn)**.
Para nosso algoritmo

- Custo X = insere no vetor O(log|V|)
- Custo Y = extra o minimo O(log|V|)
- Custo Z = Arruma a distância dos vertices O(log|V|)

Complexidade total

$$O(|V|) + X.O(|V|) + Y.O(|V|) + Z.O(|V| + |E|) = O(|V|) + O(log|V|).O(|V|) + O(log|V|).O(|V|) + O(log|V|).O(|V| + |E|) = O(log|V|).O(|V| + |E|)$$

3 Implementação

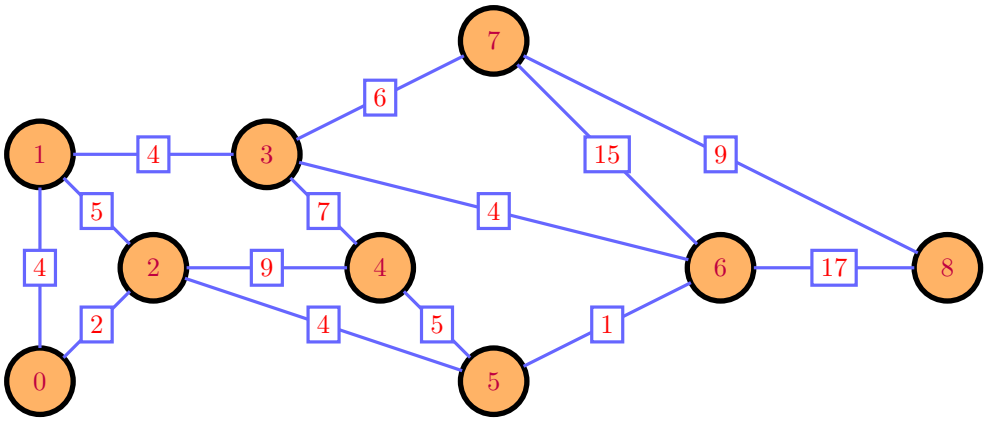
Leitura de arquivo

O algoritmo consiste em uma entrada representada por um arquivo cuja primeira linha do arquivo contém o número de vértices (V), na segunda linha contém o número de arestas

Exemplo de arquivo de entrada

```
9
14
3
0   1   4
0   2   2
1   2   5
1   3   4
2   4   9
2   5   4
3   4   7
3   6   4
3   7   6
4   5   5
5   6   1
6   8   17
7   6   15
7   8   9
```

O arquivo de entrada é a representação do grafo ilustrado abaixo.



Entrada Grafo aleatorio

Foi implementada uma função que recebe o numero de vértices e o número de arcos sendo assim ela gera um grafo aleatório com vértices 0..V-1 e número esperado de arcos igual a A. A função supõe que $V >= 2eA <= V * (V - 1)$.

Estrutura do grafo

Foi utilizado uma lista de adjacência é a representação de todas arestas ou arcos de um grafo em uma lista. Para grafos não direcionados cada uma das entradas é um conjunto de dois nós contendo as duas extremidades da aresta correspondente, se não for dirigido as entradas são uma tupla, um indica o nó de origem e outro o nó de destino

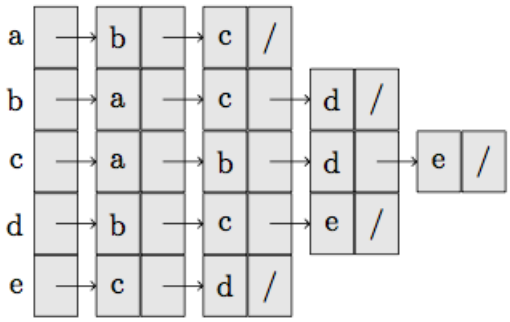


Figure 1: Representação de um grafo não dirigido em lista de adjacência

Cada nó na lista possui pelo menos dois campos, vértice que ira armazena o indice do vértice que é adjacente i e próximo que é um ponteiro para o próximo nó adjacente. As cabeça da lista são armazenadas em um vetor de ponteiros para facilitar o acesso aos vértices.

Vantagens de lista de adjacência

A principal alternativa para a lista de adjacência é a matriz de adjacência. Para um grafo com uma matriz de adjacência esparsa uma representação de lista de adjacências do grafo ocupa menos espaço, porque ele não usa nenhum espaço para representar as arestas que não estão presentes. Usando uma implementação de listas de adjacência com um simples array, uma lista de adjacência de um grafo não direcionado requer cerca de **8**e bytes de armazenamento, onde e é o número de arestas: cada aresta dá origem a entradas nas duas listas de adjacência e usa quatro bytes cada uma.

Heap binário

- Criar um min Heap de tamanho V onde é o número de vértices no gráfico fornecido. Toda celula do min heap ira conter número de vértice e valor da distância do vértice.
- Inicialize min heap com o vértice de origem "raiz" o valor de distância atribuído ao vértice de origem é zero, para todos os demais vértices atribuímos um valor INF (alto).
- Enquanto o nosso heap não estiver vazio, extraia o vértice com a menor distância do min heap. O vértice extraido é **u**, para cada vértice adjacente **v** de **u** verifique se v está no min heap. Caso esteja e o valor da distância for maior do que o peso u**–** **v** então atualize o valor da distância de v.

Complexidade do algoritmo

Pode-se perceber que os loops são O(V+E), também o loop da função de diminuir o valor das arrestas para arrumar o peso leva um tempo O(logV). Então a complexidade do tempo total é O(E+V)*O(logV) que é O((E+V)*logV) = O(ElogV) levando em consideração que nosso algoritmo utiliza uma fila de prioridade Heap binário.

Podemos então melhorar o tempo de execução do algoritmo utilizando um Fibonacci Heap para O(E + VlogV). O motivo é que o Fibonacci Heap leva tempo O(1) tempo para a operação de diminuir o valor, enquanto nosso algoritmo leva um tempo O(log n).

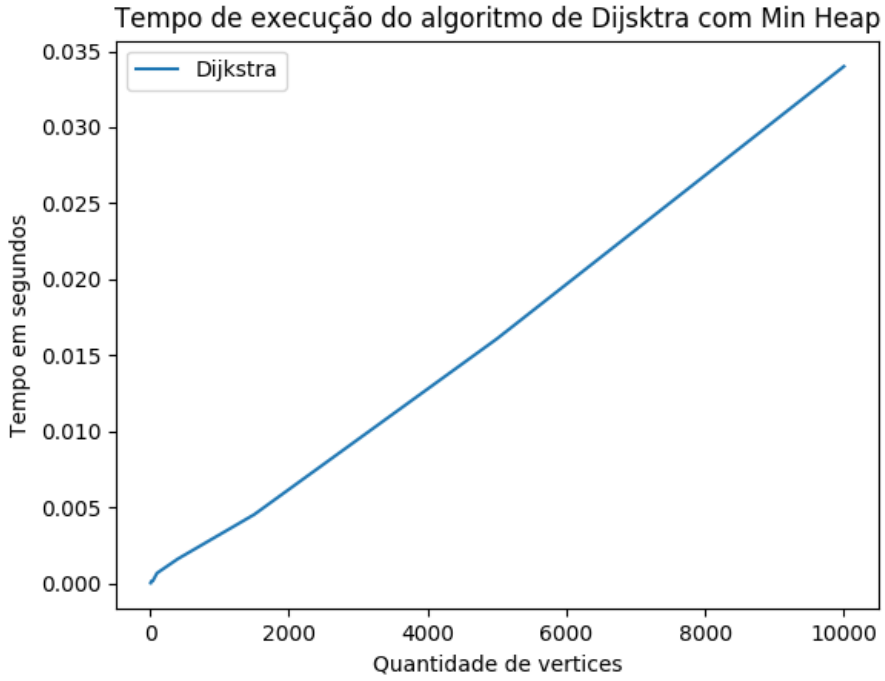


Figure 2: Tempo de execução do algoritmo

4 Limitações do algoritmos

O algoritmo pode apresentar problemas :

- A quantidade de arestas esperadas na função de gerar números aleatórios $V >= 2$ e $A <= V * (V - 1)$.
- A busca inicialmente estava fixa no vértice 0 como raiz, pode ser escolhido outro desde que esta aresta seja conexa com as demais ou seja tenha um (v) , tal que (u)—>(v) onde u é a raiz escolhida.
- Para grafos aleatórios muito grandes o algoritmo demora um pouco.

5 Referencias

[CLRS3] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, Introduction to Algorithms, 3rd edition, MIT Press, 2009.

Steven Skiena, Implementing Discrete Mathematics: Combinatorics and Graph Theory with Mathematica, Addison-Wesley, 1990.

$[CLRS]T.H.Cormen, C.E.Leiserson, R.L.Rivest, C.Stein, IntroductiontoAlgorithms, 2ndedition, MITPress\&McGra$
$Hill$, 2001.

$https://www.ime.usp.br/\ pf/algoritmos_para_grafos/aulas/random.html$

$SlidesLuciana$

$https://www.ime.usp.br/\ pf/analise_de_algoritmos/index.html$

$https://www.ime.usp.br/\ pf/algoritmos_para_grafos/aulas/graphs.html$

$https://www.ime.usp.br/\ pf/algoritmos_para_grafos/aulas/dijkstra.html$