O problema do caminho mínimo

Guilherme Lage Albano[1]

[1]Instituto de ciências exatas e aplicadas – Universidade Federal de Ouro Preto (UFOP)
Caixa Postal 24 CEP 35.930-970 – João Monlevade – MG – Brasil

{Albano,Guilherme }Guilherme.albano@aluno.ufop.edu.br

Abstract. *This article describes the shortest path problem in graphs, that sum up in finding the shortest path beetwen a initial vértice 'S' and another final vértice 'T' of a weighted directed graph. Dijsktra, Bellman-Ford and Floyd-Warshall are the three more common algorithims in this situation and were used to solve this problem, all of them were build-up using python.*

Resumo. *Este artigo descreve o problema do caminho mínimo em grafos, que consiste em encontrar o caminho de menor custo entre um vértice inicial 'S' e outro vértice final 'T' de um grafo ponderado, para a solução foi utilizado os três algoritmos mais comuns para esse tipo de situação, que são o algoritmo de Dijkstra, Bellman-Forde e Floyd-Warshall, todos implementados em python.*

1. Introdução sobre o problema

O problema do caminho mínimo em grafos, consiste em encontrar o menor caminho entre dois vertices, um inicial 'S' e um final 'T', o caminho mais curto é aquele que a soma dos pesos das arestas é menor em relação as outras opções. Os vértices se torna, adjacentes quando possuem uma aresta interligando-os, para que seja possível se chegar de um vértice 'S' a um vértice 'T' é necessário que todos os vértices entre estes possuam ao menos duas arestas.

2. Aplicações reais

Várias situações reais podem ser representadas em grafos, como localização e mapeamento por GPS, usuários e seguidores em redes sociais, cabeamento e transmissão de dados, problema da inspeção de rotas (popularmente conhecido como o problema do carteiro chinês) etc.

3. A resolução do problema

Para a resolução deste problema foram utilizados os três algoritmos mais conhecidos para a situação que são Dijkstra, Bellman-Ford e Floyd-Warshall, todos os três foram implementados em python e utilizando a IDE PyCharm.

Para a representação dos grafos em pyhton, foi utilizado lista de adjacências, onde cada vértice é representado pela posição [i] na lista, suas arestas são representadas por sub-listas [j][k] contidas em cada posição, onde j é o vértice adjacente e k o peso da aresta.

O gráfico ao lado é representado pela seguinte lista.

G = [[(1 , 2) , (2 , 6)] , [(2 , 5) , (4 , 3)] , [(1 , 5) , (3 , 4)] , [(0 , 2) , (4 , 7)] , [(0 , 1) , (3 , 7)] , [ ]]

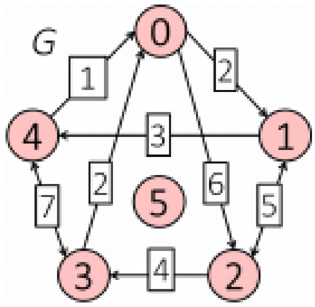

Figure 1. Grafo G

4. Algoritmos

Em cada subseção uma explicação sobre cada algoritmo, pseudocódigo e desempenho.

4.1. Dijsktra

Proposto inicialmente em 1959, O algoritmo de Dijkstra desenvolvido recebe como parâmetro um lista representando um grafo possuindo arestas adjancentes e pesos e um vértice de origem denomidado ‘s’.

Primeiramente, criamos três listas:

- Lista dist que possui as distâncias minímas para cada vértice
- Lista pred que possui os antecessores de cada vértice
- Lista Q que possui todos os vértices do grafo

Para cada elemento contido no grafo parâmetro da função, insere-se os seguintes valores em cada lista:

- Lista dist recebe ∞ para cada elemento em grafo
- Lista pred recebe ‘Nulo’ para cada elemento em grafo
- Lista Q recebe todos os vértices em grafo, em ordem crescente de valor

Após iniciar as três listas, a distância (lista dist) de Origem ‘s’ recebe 0

Então para cada elemento em Q é feito os seguintes passos

- Recebemos a menor distância em dist
- Recebemos o vértice correspondente a esta distância
- Adicionamos este vértice em uma lista de vértices já processados

Então para cada aresta do vértice de menor distância atual:

Se o peso da aresta do vértice atual for maior que o peso da aresta de menor distãncia encontra, acrescido do peso da aresta entre o vértice atual e o vértice de menor peso, atualizamos caminho para chegar na vértice analisado.

4.1.2 Pseudocódigo:

```
função dijkstra entrada(i) grafo; (ii) Origem s:

    i recebe 0
    para cada v em grafo:
        dist recebe ∞
        pred recebe Nulo
        Q recebe i
        i recebe i + 1

    dist[s] recebe 0
    Escreve: "Algoritmo de Dijkstra"
    Escreve: "Origem: " + Origem s
    Escreve: "Destino: " +  tamanho(grafo) -1
    Escreve: "Processando..."


    para cada q em Q faça:
        menor recebe x | min{dist, ∀i em Q}
         u recebe y | posição x em Q
        excluidos[] recebe u
        i recebe 0

        para cada aresta em grafo[u] faça:

            se  dist[aresta[0]] > dist[u] + aresta[1] faça:
                dist[aresta[0]] recebe dist[u] + aresta[1]
                pred[aresta[0]] recebe u
            i recebe i + 1


    Escreve "Tempo: " + tempo
    retorna dist, pred
```

4.1.3 Desempenho

A tabela a seguir representa os testes feitos com o algoritmo e avaliação de seu tempo de execução para cada tipo de grafo, onde |V | é o número de vértices, |E| é o número de arestas, w^{min} é o peso mínimo de uma aresta e w^{max} é o peso máximo

|V|	|E|	W^{min}	W^{max}	Tempo de execução
50	50	1	50	0,001 s
50	1.500	1	5	0,001 s
100	2.000	1	50	0,008 s
100	8.000	1	5	0,008 s
500	10.000	1	50	0,7 s
500	100.000	1	5	0,9 s
1.000	50.000	1	50	5,8 s
1.000	500.000	1	5	5,9 s

4.2 Bellman-Ford

Outra solução para o problema do caminho mínimo foi proposto pelos matemáticos R. Bellman e L. R. Ford e publicado em 1956-58.

Neste algoritmo, recebe-se como entrada uma lista representando um grafo ponderado e um vértice origem, assim como o algoritmo de Dijkstra.

O funcionamento deste algoritmo resume-se em inspecionar cada aresta sequencialmente, procurando por caminhos mínimos e isso repete até que não se encontrem mais nenhum caminho melhor. Com esta estrategia é possível calcular o caminho mínimo em grafos com arestas de peso negativo.

No algoritmo implementado em python, recebemos como parametro uma lista de adjacências representando o grafo e o valor origem s, após isso os passos de inicialização são parecidos com o algoritmo de Dijkstra descrito anteriormente.

Primeiramente, criamos três listas:

- Lista dist que possui as distâncias minímas para cada vértice
- Lista pred que possui os antecessores de cada vértice
- Lista Q que possui todos os vértices do grafo

Para cada elemento contido no grafo parâmetro da função, insere-se os seguintes valores em cada lista:

- Lista dist recebe ∞ para cada elemento em grafo
- Lista pred recebe 'Nulo' para cada elemento em grafo
- Lista Q recebe todos os vértices em grafo, em ordem crescente de valor

Após iniciar as três listas, a distância (lista dist) de Origem 's' recebe 0

Então repetimos os seguintes passos x - 1 vezes, onde x é o número de vértices no grafo

A variável trocou recebe falso e só é atualizada caso haja alguma troca de distância

E então para cada aresta contida no grafo se é feita uma inspeção, se caso o peso da aresta atual for maior que o peso da aresta do vértice adjacente acrescido do valor do peso para se chegar neste outro vértice adjacente, atualiza-se as listas de predescessores e distâncias para os novos valores encontrados e a variável "Trocou" recebe "verdadeiro" pois houve uma atualização nas listas de distância e predescessores.

Caso não haja atualizações no loop atual, encerramos o laço de repetição prematuramente pois já temos o caminho mínimo.

4.2.2 Pseudocódigo

```
função bellmanFord entrada(i) grafo; (ii) Origem s:

    i recebe 0
    j recebe 0

    para cada vértice em grafo faça:
       dist recebe ∞
       pred recebe Nulo
       Q recebe i
       i recebe i + 1

    dist[s] ecebe 0
    i recebe 0
    b recebe 0

    Escreve "Algoritmo de Bellman-Ford"
    Escreve "Origem: " + Origem s
    Escreve: "Destino: " + tamanho(grafo) - 1
    Escreve: "Processando..."

    enquanto b <= tamanho(grafo):
        trocou recebe Falso
        i recebe 0
        para cada vértice em Q:
            j recebe 1

            para cada aresta em grafo[i]:
                    adjacente recebe aresta[0]

                    se dist[adjacente] > dist[i] + aresta[1]:
                        dist[adjacente] recebe dist[i] + aresta[1]
                        pred[adjacente] recebe i

                    trocou recebe Verdadeiro
                    j recebe  j + 1
            i recebe i + 1
        b recebe b + 1



        se trocou é igual a  Falso:
            Fim enquanto

    Escreve: "Tempo: " + Tempo
    returna dist, pred
```

4.2.3 Desempenho

A tabela a seguir representa os testes feitos com o algoritmo e avaliação de seu tempo de execução para cada tipo de grafo, onde |V | é o número de vértices, |E| é o número de arestas, w^{min} é o peso mínimo de uma aresta e w^{max} é o peso máximo

|V|	|E|	W^{min}	W^{max}	Tempo de execução
50	50	1	50	0,001 s
50	1.500	1	5	0,01 s
100	2.000	1	50	0,04 s
100	8.000	1	5	0,1 s
500	10.000	1	50	1,1 s
500	100.000	1	5	9 s
1.000	50.000	1	50	13 s
1.000	500.000	1	5	87 s

4.3. Floyd-Warshall

O algoritmo proposto por Robert Floyd em 1962 é baseado no algoritmo de Stephen Warshall do mesmo ano para cálculo de fechos transitivos em grafos. O algoritmo de Floyd-Warshall recebe como parâmetro apenas a lista de adjacências que representa o grafo e também diferentemente dos algoritmos de Dijkstra e Bellman-Ford, o retorno do algoritmo de Floyd-Warshall são duas matrizes, uma de distâncias e a outra de predecessors. O funcionamento deste algoritmo consiste em examiner todos os caminhos possíveis e atualizar as matrizes de distâncias e predecessors quando encontrar um novo caminho de menor custo. O algoritmo compara os caminhos entre os vértices i e j passando por k vértices intermediários, k = 1, . . ., n.

Inicialmente no algoritmo cria-se as matrizes de distâncias e predecessors, atualizamos essas matrizes com os valores-peso das arestas da lista de adjacências passada no parâmetro.

Após a implementação inicial, entramos em um laço de repetição onde examinamos todos os possíveis caminhos para cada vértice, este laço é repetido até que sejam feitas verificações em todos os vértices e em todos os caminhos possíveis, a desvantagem desse algoritmo é seu elevado custo computacional.

4.3.2 Pseudocódigo:

```
função floydWarshall entrada(i)grafo:
    i recebe 0
    j recebe 0
    k recebe 0

    enquanto i < tamanho(grafo):
        j recebe 0

        enquanto j < tamanho(grafo):

            se i for igual a j:
                dist[i] recebe 0

            se não:
                dist[i] recebe ∞
               pred[i] recebe Nulo
            j recebe j + 1
        i recebe i + 1

    Escreve: "Algoritmo de Floyd-Warshall"
    Escreve: "Origem: 0"
    Escreve: "Destino: " + tamanho(grafo) - 1
    Escreve: "Processando..."

    i recebe 0
    para cada vértice em grafo faça:

        para cada aresta neste vértice faça:
            dist[i][aresta[0]] recebe aresta[1]
            pred[i][aresta[0]] recebe i
        i recebe i + 1



    enquanto k < tamanho(grafo):
        i recebe 0
        enquanto i < tamanho(grafo):
            j recebe 0
            enquanto j < tamanho(grafo):
                se dist[i][j] > dist[i][k] + dist[k][j]:
                    dist[i][j] recebe dist[i][k] + dist[k][j]
                    pred[i][j] recebe pred[k][j]


                j recebe j + 1
            i recebe i + 1
        k recebe k + 1



    returna dist, pred
```

4.3.3 Desempenho

A tabela a seguir representa os testes feitos com o algoritmo e avaliação de seu tempo de execução para cada tipo de grafo, onde |V | é o número de vértices, |E| é o número de arestas, w^{min} é o peso mínimo de uma aresta e w^{max} é o peso máximo

|V|	|E|	W^{min}	W^{max}	Tempo de execução
50	50	1	50	0,03 s
50	1.500	1	5	0,03 s
100	2.000	1	50	0,3 s
100	8.000	1	5	0,3 s
500	10.000	1	50	36 s
500	100.000	1	5	36 s
1.000	50.000	1	50	288 s
1.000	500.000	1	5	295 s

5. Análise de desempenho

5.1 Análise geral

As seguites configurações foram utilizadas para realizar os testes:

Processador: Celeron G1820 2 núcleos/2 threads 2.6 ghz 2mb de cache

Placa-mãe: H81-1M-CS/BR chipset Haswell

Memória RAM: 8gb DDR3 1333hz dual-channel

Placa de video: GTX 750 Overclock 1Gb GDDR5

Os resultados dos testes podem variar de acordo com o hardware.

Observando os gráficos conclui-se que os melhores desempenhos gerais são respectivamente os algoritmos de Dijkstra, Bellman-Ford e Floyd-Warshall.

5.2 Análise individual

Dijkstra

O algoritmo de Dijkstra possui o menor custo computacional entre os três e retorna duas listas como resultado da função, lista de predecessores e de distância, a sua desvantagem está na sua incapacidade de retornar resultados corretos para grafos com arestas de peso negative, porém para grafos simples e resultados em lista é uma boa opção.

Bellman-Ford

O algoritmo de Bellman-Ford apesar de ser o Segundo com melhor desempenho, possui resultados semelhantes ao algoritmo de Dijkstra e seu retorno é o mesmo, lista de predecessores e de distâncias, sua vantagem em relação ao algoritmo anterior é que ele é capaz de processor corretamente grafos com arestas de peso negativo.

Floyd-Warshall

Em comparação aos outros algoritmos, o de Floyd-Warshall tem um custo relativamente maior quando se é trabalhado com grafos consideravelmente grandes, porém seu retorno é uma matriz de predecessores e de distâncias, sendo mais completo que o retorno dos algoritmos anteriores, além de processor corretamente grafos de aresta negative. O algortimo de Floyd-Warshall é uma boa escolha quando se precisa de retorno mais detalhado e uma análise mais complete dos grafos.

Referências

Algoritmo de Floyd-Warshall, Acesso em 07 de Março de 2021.

Disponível em: https://cp-algorithms-brasil.com/grafos/floyd.html

Algoritmo de Dijkstra para cálculo do Caminho de Custo Mínimo, Acesso em 07 de Março de 2021.

O problema do caminho mínimo, acesso em: 07 de Março de 2021. Disponível em:

http://www.dainf.ct.utfpr.edu.br/petcoce/wpcontent/uploads/2011/06/DiscreteMathFloydWarshall.pdf

Feofilloff, Paulo. Algoritmo de Bellman-Ford. Acesso em 07 de Março de 2021

Disponível em: https://www.ime.usp.br/~pf/algoritmos_para_grafos/aulas/bellman-ford.html

Floyd-Warshall algorithm, Acesso em 07 de Março de 2021.

Disponível em: https://www.geeksforgeeks.org/floyd-warshall-algorithm-dp-16/

M. Carvalho, Marco Antonio. Algoritmo de Dijkstra. Acesso em 07 de Março de 2021
Disponível em: http://www.decom.ufop.br/marco/site_media/uploads/bcc204/05_aula_05.pdf

M. Carvalho, Marco Antonio. Algoritmo de Bellman-Ford.
Acesso em 07 de Março de 2021
Disponível em:http://www.decom.ufop.br/marco/site_media/uploads/bcc204/06_aula_06.pdf

M. Carvalho, Marco Antonio. Algoritmo de Floyd-Warshall.
Acesso em 07 de Março de 2021
Disponível em: http://www.decom.ufop.br/marco/site_media/uploads/bcc204/07_aula_07.pdf
Disponível em**: http://www.inf.ufsc.br/grafos/temas/custo-minimo/dijkstra.html**

Problema do caminho mínimo, acesso em 07 de Março de 2021.
Disponível em: https://docs.ufpr.br/~volmir/PO_II_10_caminho_minimo.pdf

Rodrigues Bueno, Letícia. Algoritmos em grafos: caminho mínimo.
Acesso em 07 de Março de 2021. Disponível em:
http://professor.ufabc.edu.br/~leticia.bueno/classes/aa/materiais/caminhominimo.pdf