



UNIVERSIDADE CATÓLICA DE PELOTAS
ENGENHARIA DE COMPUTAÇÃO
Análise e projeto de algoritmos

Gabriel Harter Zoppo e Guilherme Moura Baccarin

Trabalho de Implementação II

Pelotas,
26/11/2021

Sumário

Sumário	1
Introdução	2
Fundamentação Teórica	3
Algoritmo de Dijkstra	3
Algoritmo de Bellman-Ford	3
Algoritmo da mochila fracionária	4
Algoritmo da mochila booleana	4
Desenvolvimento	5
Algoritmo de Dijkstra	5
Algoritmo de Bellman-Ford	7
Algoritmo da mochila Fracionária	8
Algoritmo da mochila Booleana	9
Resultados:	10
Algoritmos de menor caminho	10
Algoritmos do problema da mochila	12
Conclusão:	14

Introdução

O trabalho refere-se a implementação de quatro algoritmos e fazer uma análise detalhada do funcionamento e mostrar de maneira gráfica os resultados usando entradas de tamanhos diferentes.

Os algoritmos em serão implementados na linguagem de programação python pela simplicidade e pela ampla utilização atualmente, os algoritmos em questão são Algoritmo de Dijkstra, Algoritmo de Bellman-Ford, algoritmo da mochila fracionária e algoritmo da mochila booleana.

Fundamentação Teórica

Algoritmo de Dijkstra

É um algoritmo que usa a estratégia gulosa para calcular o caminho de custo mínimo entre os vértices de um grafo, ou seja ele vai percorrer todos os vértices do grafo calculando o caminho que tem menos custo para chegar até ele. O custo menor fica na Aresta e continua até chegar ao final do grafo.

Algoritmo de Bellman-Ford

É similar ao algoritmo anterior porém ele aceita pesos negativos, que o algoritmo de Dijkstra não aceita além de usar a programação dinâmica para achar o menor caminho, abaixo temos os dois rodando simultaneamente.

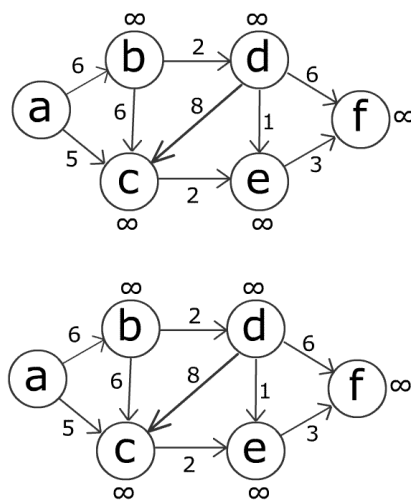


Figura 1 : Dijkstra contra Bellman-Ford

Algoritmo da mochila fracionária

É um algoritmo na qual o objetivo é encher uma mochila de objetos que maximize o valor e não seja mais pesado que a mochila possa carregar, é possível pegar frações de objetos, este algoritmo utiliza a estratégia gulosa para encontrar a solução.

Algoritmo da mochila booleana

É o mesmo problema anterior porém com diferença de abordagem, que no caso da mochila booleana é o fato de não podermos dividir os itens e usar a programação dinâmica para encontrar a solução.

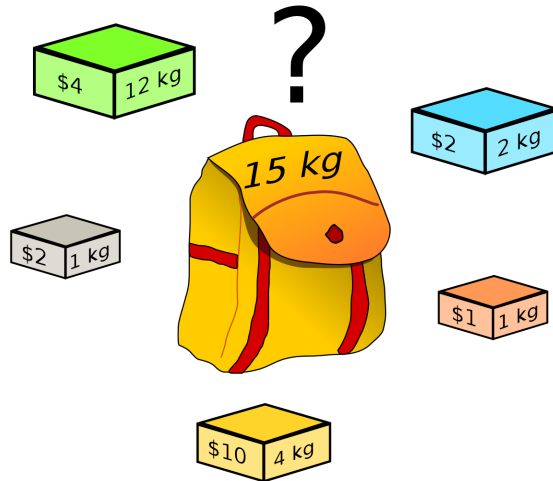


Figura 2: Problema da mochila

Desenvolvimento

Algoritmo de Dijkstra

Abaixo temos o código referente a função do algoritmo dijkstra que percorre o grafo verificando a distância até o nodo e assim encontrando o menor custo para cada nodo do grafo e por fim temos a chamada da função e o cálculo do tempo de execução do algoritmo.

```
# declaração das bibliotecas
from collections import defaultdict
from heapq import *
import time
import random

# Setando o tempo inicial para calcularmos o tempo total posteriormente
antes = time.time_ns() / (10 ** 9)

# Declarando a função do dijkstra
def dijkstra(edges, f, t):

    # preparação das variáveis pro algoritmo
    g = defaultdict(list)
    for l, r, c in edges:
        g[l].append((c, r))

    q, seen, mins = [(0, f, ())], set(), {f: 0}

    # loop para percorrer a lista
    while q:
        (cost, v1, path) = heappop(q)
        # verifica se o nodo já foi visto ou não e se ele não ele é colocado na lista de visto
        if v1 not in seen:
            seen.add(v1)
            path = (v1, path)
            # se o nodo atual for o nodo final buscado ele retorna o custo e caminho até ele.
            if v1 == t:
                return (cost, path)
```

```
    # loop para percorrer o grafo e modificar valores que ja foram vistos se tem um caminho mais curto até eles
    for c, v2 in g.get(v1, ()):
        if v2 in seen:
            continue

        # Pega o valor do nodo anterior
        prev = mins.get(v2, None)
        next = cost + c

        # verifica se o nodo anterior tem valor menor e coloca no nodo
        if prev is None or next < prev:
            mins[v2] = next
            heappush(q, (next, v2, path)) # joga o nodo pro final da fila

    return float("inf")
```

```

▶ if __name__ == "__main__":

# Fazendo a declaração do grafo utilizado, chamar a função e printar sua distância
n = 10
grafo1 = [
    ('Argentina', 'Brasil', random.randrange(0, n)),
    ('Brasil', 'Uruguay', random.randrange(0, n)),
    ('Uruguay', 'Argentina', random.randrange(0, n)),
    ('Chile', 'Bolivia', random.randrange(0, n)),
    ('Bolivia', 'Venezuela', random.randrange(0, n)),
    ('Venezuela', 'Brasil', random.randrange(0, n)),
    ('Equador', 'Peru', random.randrange(0, n)),
    ('Paraguai', 'Argentina', random.randrange(0, n)),
    ('Peru', 'Brasil', random.randrange(0, n)),
    ('Brasil', 'Venezuela', random.randrange(0, n)),
    ('Argentina', 'Chile', random.randrange(0, n)),
    ('Chile', 'Peru', random.randrange(0, n)),
    ('Bolivia', 'Brasil', random.randrange(0, n)),
    ('Peru', 'Equador', random.randrange(0, n)),
    ('Peru', 'Colombia', random.randrange(0, n))
]

print("Argentina -> Venezuela,:")
print("A menor distancia para o grafo um e: ",
      dijkstra(grafo1, "Argentina", "Venezuela"))

# Calcula o tempo de execução e printa na saída
depois = time.time_ns() / (10 ** 9)
total = (depois - antes)
print(total)

```

Algoritmo de Bellman-Ford

Abaixo temos o algoritmo de bellman-Ford que também vai percorrer o grafo e encontrar o caminho com menor custo possível até cada um dos nodos, ele possui uma vantagem sobre o algoritmo de dijkstra que é o fato de aceita nodos com peso negativo, e por fim temos a chamada da função e o cálculo do tempo de execução do algoritmo.

```
[7] # importando bibliotecas necessárias
import time
import random

# declaração da função bellman-ford
def bellman_ford(graph, source):

    # Setando o caso base
    distance, predecessor = dict(), dict()
    for node in graph:
        distance[node], predecessor[node] = float('inf'), None
    distance[source] = 0

    # Uma loop para percorrer todo o grafo
    for _ in range(len(graph) - 1):
        for node in graph:
            for neighbour in graph[node]:

                # Verifica a distância entre o nodo e todos os nodos vizinhos e coloca na variável distancia, juntamente com o nodo que a distância foi calculada
                if distance[neighbour] > distance[node] + graph[node][neighbour]:
                    distance[neighbour], predecessor[neighbour] = distance[node] + \
                        graph[node][neighbour], node

    # um loop para percorrer o grafo quando temos pesos negativos
    for node in graph:
        for neighbour in graph[node]:
            assert distance[neighbour] <= distance[node] + \
                graph[node][neighbour], "Negative weight cycle."

    #retorna os resultados encontrados que são a distancia e o nodo
    return distance, predecessor
```

```
if __name__ == '__main__':

    # declarando a variável para calcular o tempo de execução do código
    antes = time.time_ns() / (10 ** 9)

    # Fazendo a declaração do grafo utilizado, chamar a função e printar sua distância
    n = 10
    grafo = {
        'Argentina': {'Chile': random.randrange(0, n), 'Bolivia': random.randrange(0, n), 'Uruguay': random.randrange(0, n),
                      'Brasil': random.randrange(0, n), 'Paraguay': random.randrange(0, n) },
        'Brasil': {'Venezuela': random.randrange(0, n), 'Bolivia': random.randrange(0, n)},
        'Uruguay': {'Brasil': random.randrange(0, n)},
        'Bolivia': {'Brasil': random.randrange(0, n), 'Peru': random.randrange(0, n)},
        'Chile': {'Bolivia': random.randrange(0, n), 'Peru': random.randrange(0, n)},
        'Peru': {'Ecuador': random.randrange(0, n)},
        'Venezuela': {'Colombia': random.randrange(0, n)},
        'Ecuador': {'Peru': random.randrange(0, n)},
        'Colombia': {'Ecuador': random.randrange(0, n)},
        'Paraguay': {'Brasil': random.randrange(0, n)},
    }

    distance1, predecessor = bellman_ford(grafo, source='Argentina')
    print(distance1, '\n')

    # Calcula o tempo de execução e printa na saída
    depois = time.time_ns() / (10 ** 9)
    total = (depois - antes)
    print(total)
```


Algoritmo da mochila fracionária:

Abaixo temos o código da mochila fracionária que percorrerá a lista de itens em busca de encher a mochila com itens que maximizem o valor do roubo, podemos fracionar os itens para levar caso for necessário, devemos levar em consideração que não podemos levar mais peso que a capacidade da mochila e não podemos repetir itens e por fim temos a chamada da função e o cálculo do tempo de execução do algoritmo.

```
[ ] # importando as bibliotecas necessárias
import time

# declarando a função da mochila fracionária
def fractional_knapsack(value, weight, capacity):

    # organizando os dados
    index = list(range(len(value)))
    ratio = [v/w for v, w in zip(value, weight)]
    index.sort(key=lambda i: ratio[i], reverse=True)

    # inicialização dos valores de saída
    max_value = 0
    fractions = [0]*len(value)

    # loop para percorrer o grafo
    for i in index:
        # verificar se é necessário particionar o item ou não
        if weight[i] <= capacity:
            fractions[i] = 1
            max_value += value[i]
            capacity -= weight[i]
        # se for necessário ele faz partição para dar o maior valor possível
        else:
            fractions[i] = capacity/weight[i]
            max_value += value[i]*capacity/weight[i]
            break

    # retorno dos valor máximo obtido e as partições
    return max_value, fractions

# inicia as variáveis
n = 600
weight1 = []
value1 = []
capacity = 300

# fazendo a criação dos itens e colocando valores aleatórios para as variáveis valor e peso do item
for i in range(n):
    value1 = [x*3 for x in range(1, n)]
    weight1 = [x*2 for x in range(1, n)]

# printa o valor máximo obtido usando as variáveis dadas
antes = time.time_ns() / (10 ** 9)
max_value, fractions = fractional_knapsack(value1, weight1, capacity)
print('Maximo de valor que pode ser carregado:', max_value)

# printa o tempo final de execução do algoritmo
depois = time.time_ns() / (10 ** 9)
total = (depois - antes)
print(total)
```

Algoritmo da mochila booleana

Abaixo temos o código referente a mochila booleana que percorre o vetor de valor e peso e vai definir o número de itens que maximizam o valor que a mochila vai carregar, o peso deve ser menor ou igual ao peso máximo carregado pela mochila em questão e por fim temos a chamada da função e o cálculo do tempo de execução do algoritmo.

```
[6] # importando as bibliotecas necessárias
import time

# declarando a função da mochila booleana
def knapSack(W, wt, val, n):

    # verificando se a capacidade e o número de itens são nulos
    if n == 0 or W == 0:
        return 0

    # se o peso passar da capacidade da mochila ele uma chamada recursiva da função
    if (wt[n-1] > W):
        return knapSack(W, wt, val, n-1)

    # senão passar a capacidade ele retorna o valor que maximiza o lucro.
    else:
        return max(
            val[n-1] + knapSack(
                W-wt[n-1], wt, val, n-1),
            knapSack(W, wt, val, n-1))

# inicializando as variáveis
val = []
wt = []
n = 500
```

```
# senão passar a capacidade ele retorna o valor que maximiza o lucro.
else:
    return max(
        val[n-1] + knapSack(
            W-wt[n-1], wt, val, n-1),
        knapSack(W, wt, val, n-1))

# inicializando as variáveis
val = []
wt = []
n = 500

# fazendo a criação dos itens e colocando valores aleatórios para as variáveis valor e peso do item
for i in range(n):
    val = [x*3 for x in range(0, n)]
    wt = [x*2 for x in range(0, n)]

# printa o valor máximo obtido usando as variáveis dadas
antes = time.time_ns() / (10 ** 9)
print("Valor máximo:", knapSack(100, wt, val, n))

# printa o tempo final de execução do algoritmo
depois = time.time_ns() / (10 ** 9)
total = (depois - antes)
print(total)
```

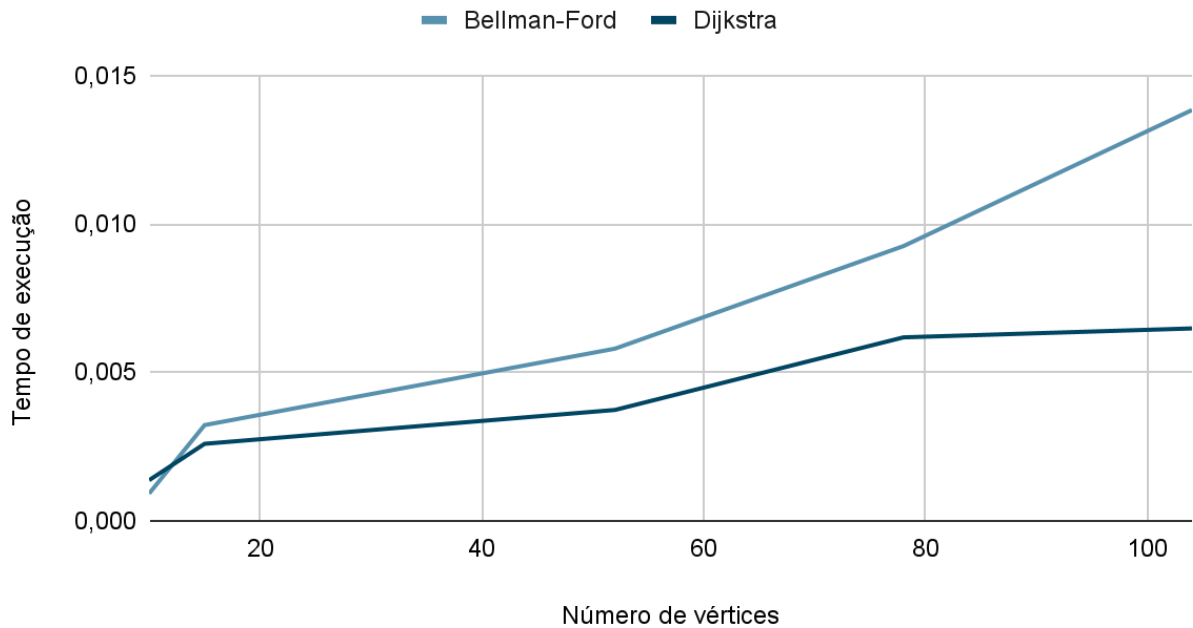
Resultados:

Algoritmos de menor caminho:

Algoritmo de Dijkstra	Vértices	Arestas	Distância Até o último vértice	Tempo de reação
1	10	17	7	0.001369476318
2	15	25	9	0.002594470977783203
3	52	78	41	0.003735065460205078
4	78	106	135	0.006178617477416992
5	104	134	155	0.00273895263671875

Algoritmo de Bellman-Ford	Vértices	Arestas	Distância Até o último vértice	Tempo de reação
1	10	17	16	0.0009210109710693359
2	15	25	14	0.003224611282348633
3	52	78	110	0.005798816680908203
4	78	106	127	0.009251832962036133
5	104	134	128	0.013831377029418945

Bellman-Ford Vs Dijkstra



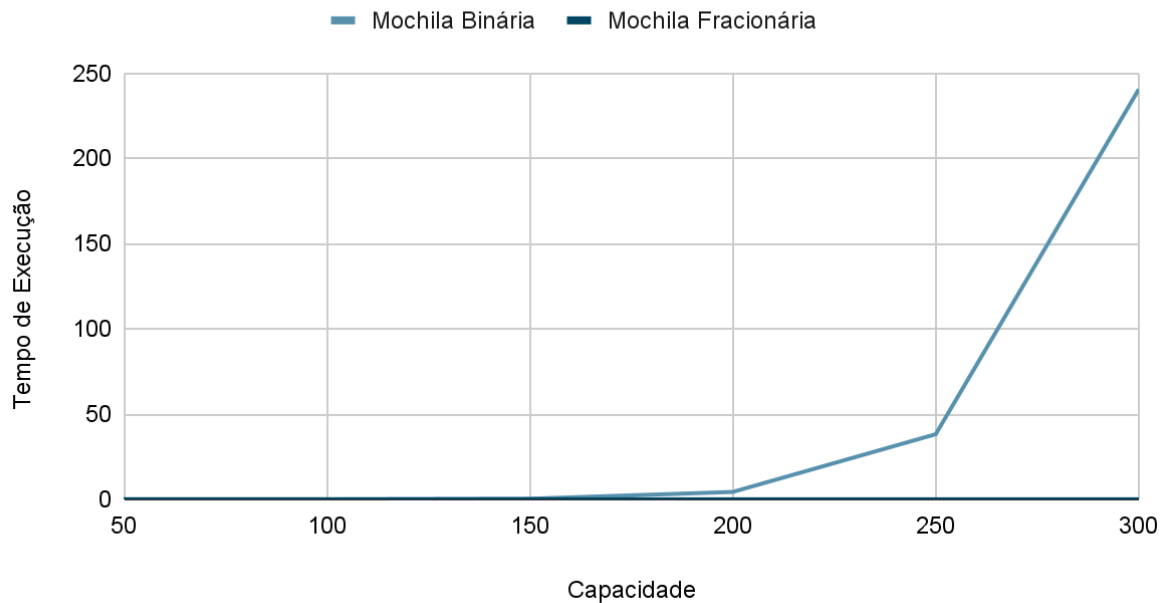
Foi executado 5 grafos diferentes com valores de arestas e vértices diferentes e a partir dele foi feito o gráfico acima na qual temos o número de vértices pelo tempo de reação, percebemos assim que o Dijkstra é mais rápido de modo geral que o bellman-ford, mais pelo fato de que o último usa programação dinâmica que é um pouco mais lento do que usando programação gulosa.

Algoritmos do problema da mochila:

Mochila Booleana	Quantidade de itens	Capacidade	Valor	Tempo de reação
1	700	50	75	0.0023593902587890625
2	700	100	150	0.023139476776123047
3	700	150	225	0.3559987545013428
4	700	200	300	4.334496974945068
5	700	250	375	38.20192074775696
6	700	300	450	240.58322620391846

Mochila Fracionária	Quantidade de itens	Capacidade	Valor	Tempo de reação
1	700	50	75	0.0007014274597167969
2	700	100	150	0.0006489753723144531
3	700	150	225	0.0006470680236816406
4	700	200	300	0.0007271766662597656
5	700	250	375	0.0005545616149902344
6	700	300	450	0.0007309913635253906

Mochila Binária Vs Mochila Fracionária



Acima temos um gráfico referente ao tempo de reação dependendo da capacidade de cada uma das mochilas, todos com 700 itens no total, porém variando os valores de peso que cada um poderia carregar de 50 a 300.

Percebemos que o algoritmo guloso teve melhor desempenho com a capacidade de peso na mochila aumentou, diferentemente do problema da mochila binária que usa programação dinâmica que piorou quanto mais peso a mochila carregar. A mochila fracionária se mantém estável durante um grande tempo, mas a mochila binária por outro lado nem roda dependendo da capacidade da mochila.

Conclusão:

De modo geral os algoritmos que tiveram o maior desempenho foram os algoritmos gulosos que são o Dijkstra e o problema da mochila fracionária que ambos tiveram um tempo baixo para todos os tamanhos de entrada.

Os algoritmos do problema da mochila booleana e de Bellman-Ford que usam programação dinâmica foram mais lentos do que os que usam estratégia gulosa, apesar de que o Bellman-Ford tem uma vantagem sobre o Dijkstra que é o fato dele aceitar arestas de peso negativo.

Os resultados foram esperados pois os algoritmos de programação dinâmica testa todas as possibilidades que podem ser mais demorado além de gastar recursos resolvendo sub problemas que já foram resolvidos, já os algoritmos gulosos podem não encontrar a solução ótima porém seu tempo de execução é consideravelmente menor.