Problem3

November 11, 2022

[28]:
```
import math
import numpy as np
import time
import matplotlib.pyplot as plt
```

0.0.1 Generate random graph with weight

[2]:
```
# generate graph
def graph(N,p=0.7):
    """
    input:
        matrix size N
        edge probability p
    output:
        weight matrix W

    This function takes as the input martrix size N and edge
    probability p (default value 0.7) to generate a N by N
    matrix with weights between each node with no self-routing,
    i.e. W[i][j] = 0 if i=j.
    """
    W = np.zeros((N,N))

    for i in range(N-1):
        w_binary = np.random.choice([0,1], size=(1,N-i-1), p=[0.3,0.7])
        w = np.where(w_binary==1,
                     ([1]*(N-i-1) + np.random.randint(5, size=(1,N-i-1))),
                     w_binary)
        W[i,i+1:] = w
        W[i+1:,i] = w
    return W
```

0.0.2 Part (a) Dijkstra algorithm

[3]:
```
# function 1 Dijkstra algorithm
def func1(T,W):
    """
    input:
        target node T
        weight matrix W
    output:
        minimum weights path vector D

    This function takes as the input weight matrix W,
    and the target node T to implement Dijkstra
    algorithm to generate the minimum weight paths D
    from all other nodes to node T.
    """

    # initialization
    N = W.shape[1]
    T -= 1
    D = [math.inf] * N
    D[T] = 0
    steps = [False] * N

    for _ in range(N):
        minDist = math.inf

        for i in range(N):
            if D[i] < minDist and steps[i] == False:
                minDist = D[i]
                minIdx = i
        steps[minIdx] = True

        for j in range(N):
            if(W[minIdx][j] >0 and steps[j] == False
              and D[j] > (W[minIdx][j]+D[minIdx])):
                D[j] = W[minIdx][j]+D[minIdx]

    return D
```

0.0.3 Part (b)

In this part, since the task is to find the minimum weight path to node T from all other nodes by listing out all paths and sorting them, I choose to implement the Bellman-Ford algorithm. The algorithm is similar to this method while it looks for the minimum weight path when going through each node rather than sorting paths from all nodes at the end. If we analyze the complexity of this algorithm is O(V*E), where V is the number of vertices and E is the number of edges. Further, for a complete graph with N vertices, it has N^2 edges, the complexity is O(N^3), which should

be them same as the complexity of the method we required to implement. On the other hand, the complexity of Dijkstra algorithm is O(V^2) which equals to O(N^2) under the same conditions.

[4]:
```
# function 2 Bellman-Ford algorithm
def func2(T,W):
    """
    input:
        target node T
        weight matrix W
    output:
        minimum weights path vector D

    This function takes as the input weight matrix W,
    and the target node T to implement Bellman-Ford
    algorithm to generate the minimum weight paths D
    from all other nodes to node T.
    """
    # initialization
    N = W.shape[1]
    T -= 1
    D = [math.inf] * N
    D[T] = 0
    for _ in range(N-1):
        for i in range(N):
            for j in range(N):
                if(D[i] < math.inf and W[i][j] >0
                   and D[i]+W[i][j] < D[j]):
                    D[j] = D[i]+W[i][j]
    return D
```

0.0.4 Part (c)

In this part, I simply reimplement the Dijkstra algorithm while adding a parents list to store the parent node of each node to keep track of the minimum weight path. In my view, the weight vector D is unncessary, since the path can be found by re-implement what we have done in part a. On the other hand, if we want to implement the Bellman-Ford algorithm to reconstrcu the weight matrx D in a certain certain level to build the shortest path, then only the weight vector D of the starting node T is insufficient, since the algorithm needs the weight vectors D's of all nodes.

[23]:
```
# function 3 find minimum weight path by D
def func3(W,D,T,i):
    """
    input:
        weight matrix W
        minimum weights path vector D
        target node T
        start node i
    output:
```

```
        minimum weights path: path

    This function takes the input weight matrix W,
    the minimum weights path vector D,the target
    node T, and the start node i to implement
    Dijkstra algorithm to generate the minimum weight paths
    from node i to node T.
    """
    N = W.shape[1]
    T -= 1
    i -= 1
    d = [math.inf] * N
    d[T] = 0
    steps = [False] * N
    parent = [T]*N
    path = [i+1]
    for _ in range(N):
        minDist = math.inf

        for j in range(N):
            if d[j] < minDist and steps[j] == False:
                minDist = d[j]
                minIdx = j
        steps[minIdx] = True

        for k in range(N):
            if(W[minIdx][k] >0 and steps[k] == False
              and d[k] > (W[minIdx][k]+d[minIdx])):
                d[k] = W[minIdx][k]+d[minIdx]
                parent[k] = minIdx

    idx = i

    while idx != T:
        idx = parent[idx]
        path.append(idx+1)

    return path
```

0.0.5 Part (d)

[8]:
```
graph1 = graph(8,0.7)
print(graph1)
```

```
[[0. 2. 0. 2. 4. 2. 2. 2.]
 [2. 0. 2. 1. 0. 4. 2. 1.]
 [0. 2. 0. 1. 2. 2. 5. 0.]
```

```
 [2. 1. 1. 0. 2. 4. 2. 0.]
 [4. 0. 2. 2. 0. 2. 0. 3.]
 [2. 4. 2. 4. 2. 0. 3. 0.]
 [2. 2. 5. 2. 0. 3. 0. 5.]
 [2. 1. 0. 0. 3. 0. 5. 0.]]
```

[11]:
```
start_a = time.process_time()
D_a = func1(8,myGraph)
time_a = time.process_time() - start_a
print("The process time is: ",time_a,'\n')
print("The weights of minimum weight path from all nodes to node 8 is: \n",D_a)
```

```
The process time is:  0.0002814059999998175

The weights of minimum weight path from all nodes to node 8 is:
 [1.0, 5.0, 3.0, 2.0, 6.0, 5.0, 5.0, 0]
```

[12]:
```
start_b = time.process_time()
D_b = func2(8,myGraph)
time_b = time.process_time() - start_b
print("The process time is: ",time_b,'\n')
print("The weights of minimum weight path from all nodes to node 8 is: \n",D_b)
```

```
The process time is:  0.0009267370000001218

The weights of minimum weight path from all nodes to node 8 is:
 [1.0, 5.0, 3.0, 2.0, 6.0, 5.0, 5.0, 0]
```

[21]:
```
graph2 = graph(20)
print(graph2)
```

```
[[0. 5. 5. 0. 5. 1. 0. 1. 0. 3. 0. 0. 4. 0. 5. 3. 1. 2. 1. 4.]
 [5. 0. 2. 0. 2. 2. 0. 0. 1. 5. 1. 5. 5. 5. 5. 5. 5. 0. 5. 4.]
 [5. 2. 0. 4. 0. 1. 2. 0. 5. 0. 1. 2. 0. 0. 4. 3. 0. 5. 0. 3.]
 [0. 0. 4. 0. 4. 5. 3. 5. 2. 1. 0. 0. 0. 5. 3. 3. 5. 1. 5. 3.]
 [5. 2. 0. 4. 0. 3. 1. 0. 4. 3. 0. 5. 1. 1. 5. 0. 5. 0. 3. 0.]
 [1. 2. 1. 5. 3. 0. 2. 0. 3. 0. 2. 2. 0. 0. 1. 0. 0. 2. 3. 0.]
 [0. 0. 2. 3. 1. 2. 0. 5. 1. 4. 0. 3. 5. 1. 2. 3. 0. 5. 4. 5.]
 [1. 0. 0. 5. 0. 0. 5. 0. 3. 4. 2. 0. 0. 0. 0. 5. 2. 2. 0. 5.]
 [0. 1. 5. 2. 4. 3. 1. 3. 0. 2. 3. 5. 1. 0. 2. 4. 4. 4. 4. 0.]
 [3. 5. 0. 1. 3. 0. 4. 4. 2. 0. 1. 3. 5. 5. 5. 3. 0. 3. 2. 0.]
 [0. 1. 1. 0. 0. 2. 0. 2. 3. 1. 0. 2. 5. 3. 0. 3. 1. 4. 5. 3.]
 [0. 5. 2. 0. 5. 2. 3. 0. 5. 3. 2. 0. 2. 2. 0. 2. 5. 3. 1. 5.]
 [4. 5. 0. 0. 1. 0. 5. 0. 1. 5. 5. 2. 0. 4. 5. 4. 5. 2. 3. 0.]
 [0. 5. 0. 5. 1. 0. 1. 0. 0. 5. 3. 2. 4. 0. 5. 1. 0. 1. 5. 0.]
 [5. 5. 4. 3. 5. 1. 2. 0. 2. 5. 0. 0. 5. 5. 0. 2. 1. 5. 1. 2.]
 [3. 5. 3. 3. 0. 0. 3. 5. 4. 3. 3. 2. 4. 1. 2. 0. 2. 5. 0. 5.]
 [1. 5. 0. 5. 5. 0. 0. 2. 4. 0. 1. 5. 5. 0. 1. 2. 0. 1. 1. 0.]
```

```
 [2. 0. 5. 1. 0. 2. 5. 2. 4. 3. 4. 3. 2. 1. 5. 5. 1. 0. 3. 4.]
 [1. 5. 0. 5. 3. 3. 4. 0. 4. 2. 5. 1. 3. 5. 1. 0. 1. 3. 0. 1.]
 [4. 4. 3. 3. 0. 0. 5. 5. 0. 0. 3. 5. 0. 0. 2. 5. 0. 4. 1. 0.]]
```

[24]:
```
D_c = func1(20,graph2)
path = func3(graph2,D_c,20,1)
print("The weights of minimum weight path from all nodes to node 20 is: \n",D_c)
print("The minimum weight path from node 1 to node 20 is: \n",path)
```

```
The weights of minimum weight path from all nodes to node 20 is:
 [2.0, 4.0, 3.0, 3.0, 4.0, 3.0, 4.0, 3.0, 4.0, 3.0, 3.0, 2.0, 4.0, 4.0, 2.0,
4.0, 2.0, 3.0, 1.0, 0]
The minimum weight path from node 1 to node 20 is:
 [1, 19, 20]
```

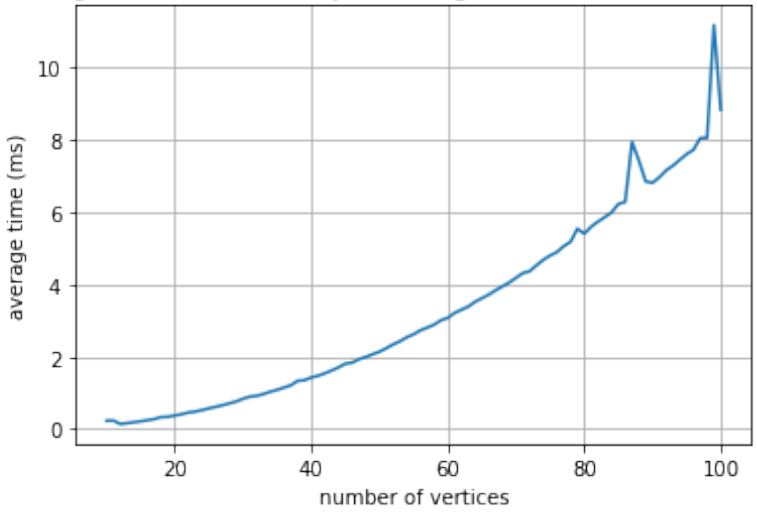
[49]:
```
times = []
for n in range(10,101):
    avg_time = 0
    for i in range(10):
        graph_N = graph(n)
        start_a = time.process_time()
        D = func1(1,graph_N)
        time_a = time.process_time() - start_a
        avg_time += time_a

    times.append(avg_time/10*1000)
```

[51]:
```
plt.plot([i for i in range(10,101)],times)
plt.title("Plot 1: The Average Process Time of Dijkstra's Algorithm vs Number␣
 ↪of Vertices of a Graph")
plt.xlabel("number of vertices")
plt.ylabel("average time (ms)")
plt.grid()
plt.show
```

[51]:
```
<function matplotlib.pyplot.show(close=None, block=None)>
```

Plot 1: The Average Process Time of Dijkstra's Algorithm vs Number of Vertices of a Graph



From the plot of the average process time versus the number of vertices of a graph, we can see that the time growth basically matches the time complexity of O(V^2)