CSCE 629 Analysis of Algorithms
Course Project

Lu Sun 228002579

November 2019

1 Implementations

In the project, two kinds of different Graphs are generated. They both have 5000 vertices which are named by integers 0,1,...,4999

1. The sparse one (Graph1,Edge1): it is with fixed total number of edges.
2. The dense one (Graph2, Edge2): it is with fixed probability for the number of adjacent other vertices form each vertex. Weights of each edge are given by random.

Three kinds of methods for solving the Max-Bandwidth-Path problem whose source-destination vertices (s,t) selected by random are given.

1. The first one is Dijkstra's Algorithm without max-heap structure.
2. The second one is Dijkstra's Algorithm with max-heap structure.
3. The third one is Kruskal's Algorithm with heap-sort order.

In this section, implementation of the algorithm details is discussed in Python language.

1.1 Random Graph Generation

The vertices of the graphs are named by integers 0,1,...,4999. According to Professors' explanation that the range of weight can't be too small, integer between 0 and 100000(include 0 and 100000, where 0 stands for edge(i,i) or -1 stands for no edge between different pair (i j)) for edge weights are chosen. First of all, to make sure that the both Graph mentioned above are connected, a cycle is generated with the following formula: edge(0,1), edge(1,2),...,edge(4998,4999) exist in Graphs with random weights($0 < wt <= 100000$). Considered the difference between Dijkstra's algorithm (matrix is better) and Kruskal's algorithm (edge is better), Graph and Edge are constructed for recording the graphs respectively.

1. Graph: a 5000×50000 (numpy) matrix, Graph[i,j] stands for the weight of edge(i,j).
2. Edge : a list for triple(vertex1, vertex2, wt), Edge[i] stands for the ith edge(vertex1,vertex2) (and edge(vertex2,vertex2),for the graphs aren't directed) with weight wt. (See function 'Edge' in Code)

The details for the above two kinds of graphs are shown as follow:

1.1.1 The Sparse Graphs (Graph1,Edge1)

In this kinds of graphs, the average vertex degree needs to be 6, which means the number of edges m in the graphs should be 15000 by the formula $m = 6 \times n/2 = 3 \times 5000$. 4999 edges have been constructed. We only need to choose 10001 (=15000-4999) edges out of $(1 + 4998) \times 4998/2$ edges left. So the following edges are labeled as $\{0, 1, 2, ..., (1+4998) \times 4998/2 - 1\}$ and only 10001 different numbers are chosen randomly from it. After selecting all the 10001 edges, weights for those edges between 1 and 100000 are added by random. (See function 'ConstructNewGraph1' in code)

1.1.2 The Dense Graphs (Graph2,Edge2)

Each vertex has edges going to about 20% of other points can be viewed as every outgoing edges from this vertex has 20% probability to be exist. So on every edge(i,j) ($i! = j$), in order to assign it a 20% probability to be exist, a random integer is chosen between $\{0, 1, 2, 3, 4\}$. If the integer equal to 0, a $wt > 0$ is assigned to the edge(i,j); otherwise, -1 is assigned to the edge(i,j). Then the probability is between $1/5 = 20\%$ and $(1/5 * 4999 + 1)/4999 = 20.020004\%$(for edge(i,i+1) is chosen by fixed) which is still around 20%. (See function 'ConstructNewGraph2' in code)

1.2 Heap Structure

Here a max-heap is constructed whose sub-index starts from 0 to n-1 (0,1,...,4999). 'heap' is used for recording the name of a vertex in the graph; 'value' is used for recording the value of vertex(edge weight or brand width); 'fringe2heap' is used for recording the heap's index; 'idx' is the node needed to be manipulated. So left and right children of node idx is $2 \times idx$ and $2 \times idx + 1$. As in Dijsktra's algorithm, the heap of fringes are built by adding or deleting or changing one vertex per time. So function 'heapInsert' is used for adding the vertex one by one; function 'heapDelete' is used for deleting vertex one by one; function 'heapifyDown' is used for changing in deleting action and function 'heapifyUp' is used for changing in adding action, which means adjust an element in the heap down to up or up to down. The insert process is simply appending the new vertex to the back of the heap and heapify up at that vertex. The delete process is also delete the exact vertex and use the last vertex in the heap to fill the position of the deleted one and then heapify down at this vertex. Cause this is max-heap, so the Maximum is just the first element of the heap.

1.3 Routing Algorithm

In this part, three different methods are implemented to solve Max-Bandwidth-Path problem: Dijkstra's Algorithm without heap, Dijkstra's Algorithm with heap and Kruskal's Algorithm with heap sort. In each algorithm, same graph with same sourse-destination pair is chosen as input. The same output, bandwidth for the path, is shown. (Choose Data 3, Graph1, s1, t1) The code for generating same things is as follow:

```
############Construct Same Graph with the sparse one##############
    Graph1 = -1*np.ones((VERT_SIZE,VERT_SIZE))
    Edge1 = []
    startime10 = time.process_time()
    ConstructNewGraph1(Graph1,Edge1)
    print('Time for constructing Graph 1:',time.process_time()-
                                        startime10)
    st1 = random.sample(range(0,VERT_SIZE),2)
    #make sure s not equal t
    s1 = st1[0]# random.randint(0,VERT_SIZE)
    t1 = st1[1]#random.randint(0,VERT_SIZE)
```

```
Results: Time for constructing Graph 1: 14.109375
```

1.3.1 Dijkstra's Algorithm without Heap

In this algorithm, it Starts from source vertex s, searches at the fringe vertex with max capacity and updates known nodes till destination vertex t (end point) is in-tree. This algorithm does not use heap structure to store fringe. So, each time the fringe vertex with maximum wt (in code 'capacity' stands for fringe weight or brandwidth and 'G[i][j]' stands for edge weight) needed to be find. Therefore, for each loop, all vertices needed to be visited in order to find the largest one. In code, 'status' for each vertex have 3 different values. '0', '1' and '2' stand for 'unseen', 'fringe' and 'in-tree' respectively. Within each loop, 'maxCapIdx' records the maximum vertex mentioned above and 'maxCap' is the corresponding maximum capacity. The other things mentioned in code are the same as that in class.

Moreover, learned form class, the time complexity of this algorithm is $O(|V|^2)$. (The method is in function 'DijkstraNoHeap') The exmaple is as follow:

```
###################NoHeap#########################
    startime11 = time.process_time()
    print('\n s=',s1,',t=',t1,',bandwidth=',DijkstraNoHeap(Graph1,
                                        s1, t1))
    endtime11 = time.process_time()
    time11.append(endtime11-startime11)
```

```
Results:  s= 4198 ,t= 158 ,bandwidth= 71171.0
```

1.3.2 Dijkstra's Algorithm with Heap

It's a modified version of algorithm in the section above. In this method, fringes are stored in a max-heap, so the fringe vertex with max capacity can be easily found which is just the first element in heap. There are three parts needed to be token care.

1. The insert and delete command affect the heap.
2. Updating an existing fringe's capacity affects the heap
3. When Updating fringe vertex w, the exacted w position in heap need to be find

For the first part, it has been discussed in the above section 'Heap Structure' where 'insert' needs heaptifyUp and 'delete' needs heaptifyDown. For the second part, as this capacity will only be updated larger, so only heaptifyUp is needed for updating fringe's vertex. For the last part, if just scan the heap to find w, it'll be O(n)'s complexity, not linear. So 'fringe2heap' is used in code to record every fringe's location in the heap and update these records of each heapifyUp and heapifyDown process. This will not increase the complexity of the algorithm and the location of each fringe vertex can be readed in $O(1)$. The leaf code is the same as that mentioned in class.

1. When a vertex changes into fringe, insert it into heap
2. At the beginning of each loop, choose the first element of heap whose capacity is largest
3. When the capacity of a fringe vertex changes, delete the vertex in heap, then insert it with new capacity.

Moreover, as mentioned in class, the time complexity of this algorithm is $O(|E|log|V|)$. (The method is in function 'DijkstraHeap')The example is as follows:

```
####################Heap#############################
    startime12 = time.process_time()
    print('\n s=',s1,',t=',t1,',bandwidth=',DijkstraHeap(Graph1, s1
                                        , t1))
    endtime12 = time.process_time()
    time12.append(endtime12-startime12)
```

```
Results:  s= 4198 ,t= 158 ,bandwidth= 71171.0
```

1.3.3 Kruskal's Algorithm with HeapSort

This method can be divided into 2 parts: Kruskal part and Kruskal search part.

For the first part, Kruskal with edges sorted by heap-sort according to weight, which is different from that in class. So in each loop, the first element in

heap (with largest edge weight) get out of the heap if it connects two connected components. Since the process is deleting one element in heap, 'heaptifyDown' need to be used afterwards.This is the process of building a "Max-Spanning Tree". It's obvious that when the graph is really dense, this part will be slow. The results can be seen in the Table 2. In this part, only 'Union' and 'Find' function are constructed. For the funcition Makeset, it replaced by arrays constructed by numpy package.

For the second part, the path in the tree (constructed in part 1) from source vertex s to destination vertex t is the max-bandwidth-path and pick the smallest edge in the path as the max capacity. As mentioned above, list of triple (v1,v2,wt) is used as graph structure. Since (v2,v1,wt) and (v1,v2,wt) are only record for once in the list, in this part vertex1 need to be v1 first and then to be v2. The corresponding code is as follow:

```
def kruskalsearch(T,capacity, s, t):
    for i in range(0,len(T)):
        e = T[i]
        if (e.vertex1 == s) and (capacity[e.vertex2] == -1):
            capacity[e.vertex2] = min(capacity[s], e.weight)
            if(e.vertex2 == t):
                return 1
            elif (kruskalsearch(T, capacity, e.vertex2, t)!=0):
                return 1

        if (e.vertex2 == s) and(capacity[e.vertex1] == -1):
            capacity[e.vertex1] = min(capacity[s], e.weight)
            if(e.vertex1 == t):
                return 1
            elif(kruskalsearch(T, capacity, e.vertex1, t)!=0):
                return 1

    return 0
```

All the other things are the same as that mentioned in class, including Union and Find functions

The time complexity of the whole algorithm is $O(|E|log|V|)$ as mentioned in class. The example is shown as follow:

```
####################Kruskal############################
    startime13 = time.process_time()
    print('\n s=',s1,',t=',t1,',bandwidth=',Kruskal(Edge1, s1, t1))
    endtime13 = time.process_time()
    time13.append(endtime13-startime13)
```

```
Results:  s= 4198 ,t= 158 ,bandwidth= 71171.0
```

We get the final resutls:

```
 Dijkstra no heap time, Dijkstra with heap time, kruskal time:
10.5781 9.0781 11.9688
```

1.4 Testing

In this part, 5 sparse graphs and 5 dense graphs (G,E) are constructed. For each graphs, 5 pair source-destination vertex (s,t) are randomly token. For each graph (G,E) and each pair (s,t), run Dijkstra's Algorithm without Heap, Dijkstra's Algorithm with Heap and Kruskal's Algorithm with HeapSort. The time for three methods above in $(5 + 5) \times 5 = 50$ different groups of data is recorded in this part.

2 Discusses and Analyzes

2.1 Results

The final results in Testing part are shown in the following Table 1 and Table2.

	t	s	Dijkstra(no heap)	Dijkstra(heap)	kruskal
	634	915	15.6406	8.5312	1.2031
	2015	261	7.5781	4.8125	10.8594
G1	353	996	21.3594	11.3906	10.6719
	3409	1218	17.5469	9.4375	3.1094
	3253	4621	24.2344	13.9375	4.7344
	1799	1	3.2031	2.0625	4.2969
	4700	3421	13.5938	7.0156	8.9531
G2	1820	3081	6.5938	5.3125	8.7500
	1609	4894	4.3125	0.3281	7.1562
	3317	2874	17.7031	9.2969	6.6562
	158	4198	10.5781	9.0781	11.9688
	2342	1653	2.1406	10.3906	8.5781
G3	3215	4698	19.6562	14.0156	9.9219
	1979	3619	9.1875	5.6875	7.8594
	2845	3759	14.625	8.4531	7.8438
	4332	960	10.5781	2.4219	7.1719
	3659	401	11.9062	6.4062	6.4844
G4	3166	257	12.0156	6.4531	4.1250
	4510	905	14.2500	7.9375	8.3594
	1399	727	1.4219	1.0312	1.4531
	1549	3416	3.9844	9.8281	1.0312
	3978	3925	0.7656	0.5156	6.9375
G5	4227	3805	14.4844	7.8281	8.9219
	3263	4115	17.1250	9.7344	5.5781
	4780	2463	14.8125	4.4219	1.3125

Table 1: The sparse graphs (Graph1,Edge1)

	t	s	Dijkstra(no heap)	Dijkstra(heap)	kruskal
	3578	4427	14.6719	5.0625	193.8906
	823	1157	18.3750	11.0469	178.7031
G1	1556	3659	2.1875	1.2500	176.3281
	1293	4422	21.9062	14.0781	179.1562
	565	810	11.0781	6.8438	177.9219
	2133	341	30.3281	18.7344	226.1719
	1455	4759	14.7969	9.7500	195.6719
G2	3035	921	29.2812	17.8281	194.0000
	4592	2228	26.3750	3.0469	202.3594
	3749	1276	15.3125	4.4531	193.1562
	1066	4346	2.2969	13.5938	201.9375
	4993	2843	21.0469	15.1562	204.2500
G3	2573	3632	15.3281	11.8125	186.3906
	709	2970	19.5156	13.7656	207.5312
	1519	37	25.9062	15.9062	184.8594
	2833	3036	16.3750	11.7188	144.6250
	1493	3401	21.6250	13.9375	138.5312
G4	2381	2528	12.9688	7.9219	129.0625
	2321	1678	2.8438	1.8906	140.5312
	1546	1433	20.7656	13.9844	137.1562
	3638	216	5.0312	1.1406	142.5625
	4423	125	20.6875	12.9219	134.5156
G5	4469	880	13.8281	8.4375	132.6094
	4504	3084	13.3438	8.0938	132.5469
	4243	3753	18.3125	13.3438	140.8750

Table 2: The dense graphs (Graph2,Edge2)

By comparing that data in Table 1 and Table 2, following conclusions can be got:

1. For the sparse graph (in Table 1), Dijkstra's algorithm with heap and Kruskal's algorithm work better than Dijkstra's algorithm without heap.

2. For the dense graph (in Table 2), Kruskal's algorithm works much worse than the other two.

2.2 Analysis

1. For the first conclusion, Dijkstra's algorithm without heap has $O(|V|^2)$ while the others are both $O(|E|log|V|)$. In spare graph 1, $|E|$ is small , which results in the better performing.

2. For the second conclusion, the reason may be that heap sort is not suitable for Krustal's part. When $|E|$ goes to really large (in dense graph2), insert

one by one wastes lots of time. Moreover, in each loop, deleting the first one in heap and using heaptifyDown to find the largest one in heap for next loop also waste lots of time. Finally, it results in the worst perform among the three method.

3. The code is finished on surface which much slow than that on Desktop or Laptop (cause the problem of GPU)

4. In kruskal's Algorithm, if replaced heap sort by sorting the edges in non-increasing order directly, it will be much faster for Graph2. Namely, replace the 'kruskal' part in function 'Kruskal(E,s,t)' by following code

```
E.sort(key=lambda  Edge: Edge.weight ,  reverse  = True)
T = []
# ############ makeset ###########
rank = np.random.randint(0,1,VERT_SIZE)
dad = np.random.randint(-1,0,VERT_SIZE)
for i in  range(0,len(E)):
    vert1 = E[i]. vertex1
    vert2 = E[i]. vertex2
    r1 = Find(vert1 ,dad)
    r2 = Find(vert2 ,dad)
    if r1 != r2:
        T.append(E[i])
        Union(r1,r2,dad ,rank)
```

The complexity time for the required one by using heap sort is over 150 second, which is shown in Table2. However, the complexity time by using directly sort order can reduce the time within 30 second.

5. All three methods share the same bandwidth in each 50 different groups which shows the correctness of the above coding

3 Code

```
import time
import random
import numpy as np
import math

VERT_SIZE = 5000

class Edge:
    def __init__(self,vertex1,vertex2,weight):
        self.vertex1 = vertex1
        self.vertex2 = vertex2
        self.weight = weight
    def __repr__(self):
        return repr((self.vertex1, self.vertex2, self.weight))

def ConstructNewGraph1(G1,E1):
    E1.clear()
```

```
    for i in range(0,VERT_SIZE-2):
        G1[i,i] = 0
        wt = random.randint(1,100000)
        e = Edge(i,i+1,wt)
        E1.append(e)
        G1[i,i+1] = wt
        G1[i+1,i] = wt

    NewSize = int((VERT_SIZE-2)*(VERT_SIZE-1)/2)
    NewLen = int(VERT_SIZE*6/2-(VERT_SIZE-1))
    RandEdge = random.sample(range(0,NewSize),NewLen)
    RandEdge.sort()
    for i in range(0,VERT_SIZE-2):
        for j in range(i+2,VERT_SIZE):
            RandPoint = i*VERT_SIZE - (i+1)*(i+2)/2 + 1 + j-(i+2)
            if (len(RandEdge)!=0)and(RandPoint==RandEdge[0]):
                wt = random.randint(1,100000)
                e = Edge(i,j,wt)
                E1.append(e)
                G1[i,j] = wt
                G1[j,i] = wt
                RandEdge.pop(0)

def ConstructNewGraph2(G2,E2):
    E2.clear()
    for i in range(0,VERT_SIZE-1) :
        G2[i,i] = 0
        wt = random.randint(1,100000)
        e = Edge(i,i+1,wt)
        E2.append(e)
        G2[i,i+1] = wt
        G2[i+1,i] = wt

    for i in range(0,VERT_SIZE-1):
        for j in range(i+1,VERT_SIZE):
            perct = random.randint(0,4)%5
            if(perct==0):
                wt = random.randint(1,100000)
                e = Edge(i,j,wt)
                E2.append(e)
                G2[i,j] = wt
                G2[j,i] = wt
```

```
def DijkstraNoHeap(G,s,t):#0 unseen;1 fringe;2 intree
    if s==t:
        return -1

    status = np.random.randint(0,1,VERT_SIZE)
    dad = np.random.randint(-1,0,VERT_SIZE)
    capacity = np.zeros(VERT_SIZE)
    status[s] = 2
    for j in range(0,VERT_SIZE):
        if G[s,j]>0:
            status[j] = 1
            capacity[j] = G[s,j]
            dad[j] = s
```

```
    while status[t]!=2 :
        maxCap = 0
        maxCapIdx = -1
        for i in range(0,VERT_SIZE):
            if (status[i] ==1) and (capacity[i]>maxCap):
                maxCap = capacity[i]
                maxCapIdx = i

        v = maxCapIdx
        status[v] = 2
        for w in range(0,VERT_SIZE):
            if G[v,w]>0 :
                if status[w] ==0 :
                    status[w] =1
                    dad[w] = v
                    capacity[w] = min(capacity[v],G[v,w])
                elif (status[w] ==1) and (capacity[w]< min(capacity
                                                    [v],G[v,w])):
                    dad[w] = v
                    capacity[w] = min(capacity[v],G[v,w])

    return capacity[t]
```

```
#def heapMax(heap):
#    return heap[0]

def heapifyUp(heap, value, fringe2heap, idx):
    if idx >= 1 :
        idxHalf = math.floor(idx/2)
        if value[heap[idx]] > value[heap[idxHalf]]:
            temp = heap[idxHalf]
            heap[idxHalf] = heap[idx]
            heap[idx] = temp
            fringe2heap[heap[idxHalf]] = idxHalf
            fringe2heap[heap[idx]] = idx
            heapifyUp(heap, value, fringe2heap, idxHalf)

def heapifyDown(heap, value, fringe2heap, idx):
    if idx * 2 >= len(heap):
        return

    childIdx = 2 * idx
    if idx * 2 + 1 < len(heap):
        if value[heap[2*idx+1]]>value[heap[2*idx]]:
            childIdx = 2 * idx + 1

    if value[heap[childIdx]] > value[heap[idx]]:
        temp = heap[childIdx]
        heap[childIdx] = heap[idx]
        heap[idx] = temp
        fringe2heap[heap[childIdx]] = childIdx
        fringe2heap[heap[idx]] = idx
        heapifyDown(heap, value, fringe2heap, childIdx)

def heapInsert(heap, value, fringe2heap, vert):
    fringe2heap[vert] = len(heap)
    heap.append(vert)
```

```
    heapifyUp(heap, value, fringe2heap, len(heap) - 1)

def heapDelete(heap, value, fringe2heap, idx):
    if len(heap) > 1:
        heap[idx] = heap.pop()
        fringe2heap[heap[idx]] = idx
        if idx < len(heap)-1:
            heapifyDown(heap, value, fringe2heap, idx)
    else:
        heap = []

def DijkstraHeap( G, s, t):
    if s == t:
        return -1
    status = np.random.randint(0,1,VERT_SIZE)
    dad = np.random.randint(-1,0,VERT_SIZE)
    capacity = np.zeros(VERT_SIZE)
    fringe2heap = np.random.randint(-1,0,VERT_SIZE)
    heap = []
    status[s] = 2
    for j in range(0,VERT_SIZE):
        if G[s,j] > 0:
            status[j] = 1
            capacity[j] = G[s,j]
            dad[j] = s
            heapInsert(heap, capacity, fringe2heap, j)

    while status[t] != 2:
        v = heap[0]#heapMax(heap)
        status[v] = 2
        heapDelete(heap, capacity, fringe2heap, 0)
        for w in range(0,VERT_SIZE):
            if G[v,w] > 0:
                if status[w] == 0:
                    status[w] = 1
                    dad[w] = v
                    capacity[w] = min(capacity[v], G[v,w])
                    heapInsert(heap, capacity, fringe2heap, w)
                elif (status[w] == 1) and (capacity[w] < min(
                                                capacity[v], G[v,
                                                w])):
                    dad[w] = v
                    capacity[w] = min(capacity[v], G[v,w])
                    heapifyUp(heap, capacity, fringe2heap,
                                                    fringe2heap[w
                                                    ])

    return capacity[t]
```

```
def kruskalsearch(T,capacity, s, t):
    for i in range(0,len(T)):
        e = T[i]
        if (e.vertex1 == s) and (capacity[e.vertex2] == -1):
            capacity[e.vertex2] = min(capacity[s], e.weight)
            if(e.vertex2 == t):
                return 1
            elif (kruskalsearch(T, capacity, e.vertex2, t)!=0):
```

```
                return 1

        if (e.vertex2 == s) and(capacity[e.vertex1] == -1):
            capacity[e.vertex1] = min(capacity[s], e.weight)
            if(e.vertex1 == t):
                return 1
            elif(kruskalsearch(T, capacity, e.vertex1, t)!=0):
                return 1

    return 0

#def MakeSet(vertex,dad,rank):
#    dad[vertex] = -1
#    rank[vertex] = 0

def Union(r1,r2,dad,rank):
    if r1 ==r2:
        return -1

    if rank[r1] > rank[r2] :
        dad[r2] = r1
    elif rank[r1] < rank[r2] :
        dad[r1] = r2
    else : #rank[r1] = rank[r2]
        dad[r2] = r1
        rank[r1] += 1

def Find(vertex,dad):
    w = vertex
    while dad[w] != -1:
        w = dad[w]

    return w

def Kruskal(E, s, t):
    heap = []
    cap = np.zeros(len(E))
    fringe2heap = np.random.randint(-1,0,len(E))
    for i in range(0,len(E)):
        cap[i] = E[i].weight
        heapInsert(heap, cap, fringe2heap, i)

    #E.sort(key=lambda Edge: Edge.weight, reverse = True)
    T = []
    ############makeset###########
    rank = np.random.randint(0,1,VERT_SIZE)
    dad = np.random.randint(-1,0,VERT_SIZE)

    for i in range(0,len(E)):
        if len(heap)!=0:
            v = heap[0]#heapMax(heap)
            heapDelete(heap, cap, fringe2heap, 0)
            vert1 = E[v].vertex1#E[i].vertex1
            vert2 = E[v].vertex2#E[i].vertex2
            r1 = Find(vert1,dad)
            r2 = Find(vert2,dad)
            if r1 != r2:
```

```
                T.append(E[v])#(E[i])
                Union(r1,r2,dad,rank)

    capacity = -1*np.ones(VERT_SIZE)
    capacity[s] = 100000
    kruskalsearch(T, capacity, s, t)
    return capacity[t]
```

```
time11 = []
time12 = []
time13 = []
time21 = []
time22 = []
time23 = []
s1 = np.random.randint(-1,0,25)
t1 = np.random.randint(-1,0,25)
s2 = np.random.randint(-1,0,25)
t2 = np.random.randint(-1,0,25)
for i in range(0,5):
    print('Data ',i+1,':\n')
##################################################
####################Graph1#######################
##################################################
    Graph1 = -1*np.ones((VERT_SIZE,VERT_SIZE))
    Edge1 = []
    startime10 = time.process_time()
    ConstructNewGraph1(Graph1,Edge1)
    print('Time for constructing Graph 1:',time.process_time()-
                                        startime10)
    for j in range(0,5):
        st1 = random.sample(range(0,VERT_SIZE),2)
        indx = 5*i+j
        s1[indx] = st1[0]# random.randint(0,VERT_SIZE)
        t1[indx] = st1[1]#random.randint(0,VERT_SIZE)
####################NoHeap#############################
        startime11 = time.process_time()
        print('\n s=',s1[indx],',t=',t1[indx],',bandwidth=',
                                            DijkstraNoHeap(Graph1, s1
                                            [indx], t1[indx]))
        endtime11 = time.process_time()
        time11.append(endtime11-startime11)
####################Heap###############################
        startime12 = time.process_time()
        print('\n s=',s1[indx],',t=',t1[indx],',bandwidth=',
                                            DijkstraHeap(Graph1, s1[
                                            indx], t1[indx]))
        endtime12 = time.process_time()
        time12.append(endtime12-startime12)
####################Kruskal############################
        startime13 = time.process_time()
        print('\n s=',s1[indx],',t=',t1[indx],',bandwidth=',Kruskal
                                            (Edge1, s1[indx], t1[indx
                                            ]))
        endtime13 = time.process_time()
        time13.append(endtime13-startime13)
    #print ('\n Dijstra no heap time:',endtime-startime)
    #print ('\n Dijstra with heap time:',endtime-startime)
```

```
    #print ('\n kruskal time:',endtime-startime)
        print('\n Dijkstra no heap time, Dijkstra with heap time,
                                            kruskal time:')
        print(endtime11-startime11,endtime12-startime12,endtime13-
                                            startime13)
        print('\n')
    print('\n\n')
################################################
###################Graph2#####################
################################################
    Graph2 = -1*np.ones((VERT_SIZE,VERT_SIZE))
    Edge2 = []
    startime20 = time.process_time()
    #Graph1, Edge1 =
    ConstructNewGraph2(Graph2,Edge2)
    print('Time for constructing Graph 2:',time.process_time()-
                                        startime20)
    for j in range(0,5):
        indx = 5*i+j
        st2 = random.sample(range(0,VERT_SIZE),2)
        s2[indx] = st2[0]# random.randint(0,VERT_SIZE)
        t2[indx] = st2[1]#random.randint(0,VERT_SIZE)
###################NoHeap#########################
        startime21 = time.process_time()
        print('\n s=',s2[indx],',t=',t2[indx],',bandwidth=',
                                            DijkstraNoHeap(Graph2, s2
                                            [indx], t2[indx]))
        endtime21 = time.process_time()
        time21.append(endtime21-startime21)
###################Heap###########################
        startime22 = time.process_time()
        print('\n s=',s2[indx],',t=',t2[indx],',bandwidth=',
                                            DijkstraHeap(Graph2, s2[
                                            indx], t2[indx]))
        endtime22 = time.process_time()
        time22.append(endtime22-startime22)
###################Kruskal#########################
        startime23 = time.process_time()
        print('\n s=',s2[indx],',t=',t2[indx],',bandwidth=',Kruskal
                                            (Edge2, s2[indx], t2[indx
                                            ]))
        endtime23 = time.process_time()
        time23.append(endtime23-startime23)
    #print ('\n Dijstra no heap time:',endtime-startime)
    #print ('\n Dijstra with heap time:',endtime-startime)
    #print ('\n kruskal time:',endtime-startime)
        print('\n Dijkstra no heap time, Dijkstra with heap time,
                                            kruskal time:')
        print(endtime21-startime21,endtime22-startime22,endtime23-
                                            startime23)
        print('\n')
    print('\n\n\n')

print('Graph1: t, s, Dijkstra no heap time, Dijkstra with heap time
                                    , kruskal time\n')
for i in range(0,25):
    print('& ',t1[i],'&',s1[i],'&',round(time11[i],4),'&',round(
```

```
                                        time12[i],4),'&',round(time13
                                        [i],4),'\\\\','\n')

print('Graph2: t, s, Dijkstra no heap time, Dijkstra with heap time
                                    , kruskal time\n')
for i in range(0,25):
    print(' &',t2[i],'&',s2[i],'&',round(time21[i],4),'&',round(
                                        time22[i],4),'&',round(time23
                                        [i],4),'\\\\','\n')
```