CSCE 629 Analysis of Algorithms

Project Report

Name: Tian Lan
UIN: 922009685

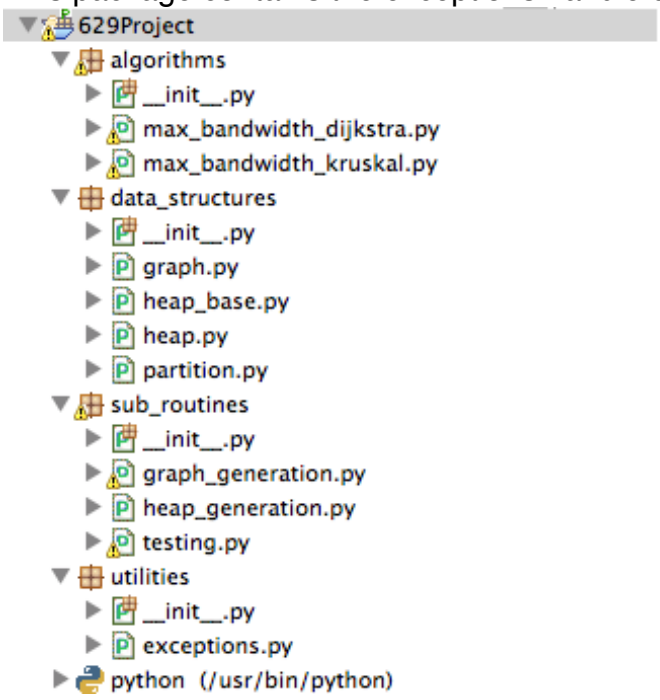
Date: 04/07/2014

1.Project outline:

1. Programming Language: Python 2.7
2. Machine: Macbook Pro 15, 2.7 GHz Intel Core i7, 16GB 1600 MHz DDR3
3. Code structure:

The project is managed in four packages

- "algorithms": This package contains the Max-Bandwidth-Path algorithms using Dijkstra's algorithm or Kruskal's algorithm.
- "data_structures": This package contains the three data structures used for the project: graph, heap and partition which could be used for MakeSet-Find-Union operations.
- "sub_routines": This package contains the main programs for the project
- "utilities": This package contains the exceptions handle class.



1-1

2. Implementations and Analysis:

2.1 Data structures:

a) Graph ADT (Abstract Data Type):

A graph is a collection of vertices and edges. We model the abstraction as a combination of three data types: Vertex, Edge, and Graph. A Vertex is a lightweight object that stores an arbitrary element provided by the user and an Edge also stores an associated object, retrieved with the weight( ) method. Below are the functions for Vertex() and Edge() class:

Vertex()		
Type	Name	Description

Attributes	“element”	The name of the vertex
Functions	element()	Return the name of the vertex

2-1

Edge()		
Type	Name	Description
Attributes	“origin”	store the name of the vertex
Functions	endpoints()	Return a tuple (*u*,*v*) such that vertex *u* is the origin of the edge and vertex *v* is the destination; for an undirected graph, the orientation is arbitrary.
	opposite(v)	Assuming vertex *v* is one endpoint of the edge (either origin or destination), return the other endpoint.
	weight()	Return the weight of the edge

2-2

As we have the basic data structures of vertex and edge, we could implement the Graph class now. As the project only requires undirected graph, so we only implement the undirected graph class. There are many kinds of data structures for graphs, I use the adjacency map structure to implement the graph. The ADT is as follows:

Graph()			
Type	Name	Description	Time
Attributes	“outgoing”	store the name of the vertex	N/A
	"vertices"	store all the vertices in the graph	N/A
	validate_vertex()	Verify that v is a Vertex of this graph	O(n)
	vertex count()	Return the number of vertices of the graph.	O(1)
	vertices()	Return an iteration of all the vertices of the graph	O(n)
	edge count()	Return the number of edges of the graph	O(1)
	remove_vertex(v)	Remove vertex *v* and all its incident edges	O(n)

Functions	remove_edge(e)	Remove edge *e* from the graph	O(1)
	edges()	Return an iteration of all the edges of the graph	O(m)
	get_vertex(x)	Return the vertex with name x	O(1)
	get_edge(u,v)	Return the edge from vertex *u* to vertex *v*, if one exists; otherwise return None.	O(1)
	degree(v)	Return the number of edges incident to vertex *v*.	O(1)
	incident_edges(v)	Return an iteration of all edges incident to vertex *v*.	O(n)
	insert_vertex(x)	Create and return a new Vertex named by *x*	O(1)
	insert_edge(u, v, x)	Create and return a new Edge from vertex *u* to vertex *v*, with weight x.	O(1)

2-3

The following is the code snippet for the Graph() class:

```
def incident_vertices(self, v):
        """Return all (outgoing) vertices incident to vertex v in the graph."""
        #self._validate_vertex(v)
        adj = self._outgoing
        for vertex in adj[v].keys():
            yield vertex

    def insert_vertex(self, x=None):
        """Insert and return a new Vertex with element x."""
        v = self.Vertex(x)
        self._outgoing[v] = {}
        self._vertices[x] = v
        return v

    def insert_edge(self, u, v, x=None):
        """Insert and return a new Edge from u to v with auxiliary element x.
        Raise a ValueError if u and v are not vertices of the graph.
        Raise a ValueError if u and v are already adjacent.
        """
        if self.get_edge(u, v) is not None:      # includes error checking
```

```
            raise ValueError('u and v are already adjacent')
        if u == v:      # includes error checking
            raise ValueError('u and v are the same')
        e = self.Edge(u, v, x)
        self._outgoing[u][v] = e
        self._outgoing[v][u] = e
```

b) Heap:

The Heap module is built upon a base class HeapBase(), where the class has a nested "_Item" class that store the items in a heap. Moreover, this class uses the feature of Python's comparative patterns, so we could compare the items in a heap more easily.

HeapBase()		
Type	Name	Description
Nested class	"_Item"	object store heap items
Functions	is_empty	Return True if the priority queue is empty.
	_parent(j)	Return the parent of item at j
	_right(j)	Return right item of item at j
	_left(j)	Return left item of the item at j
	_has_left(j)	Return True if item at j has left son
	_has_right(j)	Return True if item at j has right tre
	_swap(i, j)	Swap the elements at indices i and j of array

2-4

The Heap module has the basic operations for a heap. In our case, I build two classes for the module: A MaxHeap and a MinHeap. Below is the list of functions for the MaxHeap:

MaxHeap()			
Type	Name	Description	Time
Attribute	_data	It is an array to store all the elements in a heap.	N/A
	add(key, value)	Add a key-value pair to the heap.	O(logn)
	remove(key, value)	Remove the key-value pair in the heap.	O(n)
	max()	Return the maximum key-value pair	O(1)

Functions	remove_max()	Remove and return the maximum key-value pair	O(logn)
	_upheap(j)	The upward movement of the newly inserted entry at position j by means of swaps	O(logn)
	_downheap(j)	Opposite of _upheap(j)	O(logn)

2-5

I use a dictionary (map) in the class to store each item in the heap. So each item in the heap is represented in the form {vertex: value}, where value could be the bandwidth for the vertex. Moreover, the _data[1…5000] is the array to index all the items in the heap. So _data[0] is the root item of the heap, which is the maximum item.
For my case, the remove() function is the one that takes much time. As we firstly have to find the position i of the item by scanning the data array which is in O(n) time, then we swap the last item and this item, remove the last item and use down-heap method from i to the last to restore the heap property. This is in O(logn) time. So the total time for removing an element is O(n). Below is the code snippet for remove():

```
def remove(self, key, value):
        #search for index
        i = 0
        while i < len(self) and self._data[i]._key != key:
            i += 1
        #if the key is not existed
        if i == len(self):
            raise BaseException("Not a valid key!")
        else:
            self._swap(i, len(self._data) - 1)    # put deleting item at the end
            self._data.pop()                      # and remove it from the list;
            self._downheap(i)                     # then fix order
```

The add() and remove_max() are simple and both executed in O(logn) time.

c) Partition:

We consider a data structure for managing a partition of elements into a collection of disjoint sets. My initial motivation is in support of Kruskal's minimum spanning tree algorithm, in which a forest of disjoint trees is maintained, with occasional merging of neighboring trees.
Below is the ADT for Partition():

Partition()		
Type	Name	Description
Nested class	Position	store the position of a partition
	make_group(e)	Makes a new group containing element e, and returns its Position.

Functions	find(p)	Finds the group containing p and return the position of its leader.
	union(p,q)	Merges the groups containing elements p and q (if distinct).

2-6

The code snippet is below as an implementation of a Partition() class using union-by- size and path compression.

```
    def make_group(self, e):
        """Makes a new group containing element e, and returns its Position."""
        return self.Position(self, e)

    def find(self, p):
        """Finds the group containing p and return the position of its leader."""
        self._validate(p)
        if p._parent != p:
            p._parent = self.find(p._parent)    # overwrite p._parent after recursion
        return p._parent

    def union(self, p, q):
        """Merges the groups containing elements p and q (if distinct)."""
        a = self.find(p)
        b = self.find(q)
        if a is not b:                          # only merge if different groups
            if a._size > b._size:
                b._parent = a
                a._size += b._size
            else:
                a._parent = b
                b._size += a._size
```

Time complexity:
When using the tree-based partition representation with both union-by-size and path compression, performing a series of k make group, union, and find operations on an initially empty partition involving at most n elements takes $O(k \log^* n)$ time.

2.2 Random Graph Generation:

This part requires us to build two kinds of graphs of 5000 vertices: in the first graph G1, every vertex has degree exactly 6; in the second graph G2, each vertex has edges going to about 20% (1000) of the other vertices. Randomly assign positive weights to edges in the graphs. I design a GraphGeneration class to generate this 2 kinds of graphs.

a) generateG1() is a function to build the first kind of graph. The basic idea of this function is:

1. Insert N(5000) vertices into this graph.
2. Run three times of building a cycle that runs through all vertices once. Start from a randomly chosen vertex and choose the next vertex randomly as long as it is insertable.

After this, the graph G1 will be an 6 degree normal graph. Below is the code snippet:

```
for i in range(0,3):
    vertices = G1.vertices()
    v = random.sample(vertices,1)[0] #randomly choose a source
    vertices.remove(v)
    initial = v
    while len(vertices) > 0:
        u = random.sample(vertices,1)[0] #randomly choose a vertex
        try:
            G1.insert_edge(v, u, random.randrange(1,10000)) #if insertable
        except:
            continue #try another
        vertices.remove(u)
        v = u
    #link the head and tail
    try:
        G1.insert_edge(v, initial, random.randrange(1,10000))
    except:
        continue
```

b) generateG2() is a function to build the second kind of graph. The basic idea of this function is:

1. Insert N(5000) vertices into this graph
2. for every vertex, randomly choose about another 500 vertices (or a few more), add edges connecting this vertex and all other randomly chosen vertices. If the edge has already been added, just ignore it.

After this, every vertex in the graph G2 will go to about 20% (1000) of the other vertices.

```
for v in vertices:
    random_vertices = random.sample(vertices,520)
    for u in random_vertices:
        try:
            G2.insert_edge(v, u, random.randrange(1,10000))
        except:
            continue
```

Both of graphs have the edges with positive integer weight randomly chosen from [1,10000).

2.3 Routing Algorithms

a) Max-Bandwidth-Path using Dijkstra's algorithm:

As required by the project, we should implement the Dijkstra's algorithm using or not using heap. So I implement a MBPDijkstra() class to complete this task. This class has two inner classes: HeapFringes() and MapFringes(), respectively serving for using heap to store all vertices in the fringe, or using map in the form of {vertex:bandwidth}. Both class has the following four operations implemented specifically for its data structure. Below is the code snippet and function description and the time for both kind of implementations:

```
class HeapFringes:
    def __init__(self):
        self._fringes = MaxHeap()

    def add(self, v, bandwidth):
```

```
            self._fringes.add(v, bandwidth)

        def update(self, v, bandwidth):
            self._fringes.remove(v, bandwidth)
            self._fringes.add(v, bandwidth)

        def removeMaximum(self):
            return self._fringes.remove_max()[0]

    class MapFringes:
        def __init__(self):
            self._fringes = defaultdict()

        def add(self, v, bandwidth):
            self._fringes[v] = bandwidth

        def update(self, v, bandwidth):
            self._fringes[v] = bandwidth

        def removeMaximum(self):
            max_vertex = max(self._fringes.iteritems(), key=lambda x:x[1])[0]
            self._fringes.pop(max_vertex)
            return max_vertex
```

	HeapFringes()	MapFringes
add(v, bandwidth)	O(logn)	O(1)
update(v, bandwidth)	O(n)	O(1)
removeMaximum()	O(1)	O(n)

2-7

The algorithm I used is the same as the solution in homework #3:

```
    def initialization(self):
        for v in self.g.vertices():
            self.parent[v] = None
            self.bandwidth[v] = -1
            self.removed[v] = False

        self.bandwidth[self.s] = float("inf")

        if self.using_heap:
            self.fringes = self.HeapFringes()
        else:
            self.fringes = self.MapFringes()

        for w in self.g.incident_vertices(self.s):
            self.parent[w] = self.s;
            self.bandwidth[w] = self.g.get_edge(self.s, w).weight()
            self.fringes.add(w, self.bandwidth[w])
            #print "add", w, self.bandwidth[w]

    def find_mbp(self):
        while True:
```

```
            u = self.fringes.removeMaximum()
            self.removed[u] = True #mark it as removed
            #print "remove", u
            if u == self.t:
                break
            for w in self.g.incident_vertices(u):
                if self.bandwidth[w] == -1:
                    self.parent[w] = u
                    self.bandwidth[w] = min(self.bandwidth[u],self.g.get_edge(u, w).weight())
                    self.fringes.add(w, self.bandwidth[w])
                    #print "add", w, self.bandwidth[w]
                elif not self.removed[w] and self.bandwidth[w] <
                                        min(self.bandwidth[u],self.g.get_edge(u, w).weight()):
                    self.parent[w] = u
                    self.bandwidth[w] = min(self.bandwidth[u],self.g.get_edge(u, w).weight())
                    self.fringes.update(w, self.bandwidth[w])
```

The time complexity for this algorithm is :

1. for the initialization(), it takes O(n)
2. for the find_mbp():
   -if using heap: as shown in 2-7, as update() takes O(n) and add() takes O(logn) time, so the relaxation part takes O(mn+mlogn) = O(mn). Each removeMaximum() takes O(1) time, so totally O(n) time. Thus the total time is O(n+mn) = **O(mn)**.
   -if using map (not using heap): as shown in 2-7, as update() and add() takes O(1) time, so the relaxation part takes O(m) time. Each removeMaximum() takes O(n) time, so totally O(n^2) time. Thus the total time is **O(n^2)**.

Thus we could see that when m ≫ n, especially when the graph is a dense graph, the Dijkstra with Heap method will be much slower than Dijkstra with Map method.

b) Max-Bandwidth-Path using Kruskal's algorithm:

This is also solved by the solution in Homework #3. With the help of the Partition() class, I implement Kruskal's algorithm by using MakeSet-Find-Union operations on the disjoint sets. After getting the MST for the given graph, I use DFS to find the unique path from source s to destination t.

```
    def initialization(self):
        for e in self.g.edges():
            self.heap.add(e, e.weight())
        for v in self.g.vertices():
            self.position[v] = self.forest.make_group(v)
        self.tree = []
        self.path = list()

    def find_mbp(self):
        size = self.g.vertex_count()
        while len(self.tree) != size - 1 and not self.heap.is_empty():
            # tree not spanning and unprocessed edges remain
            edge,weight = self.heap.remove_max()
            u,v = edge.endpoints()
            a = self.forest.find(self.position[u])
```

```
            b = self.forest.find(self.position[v])
            if a != b:
                self.tree.append(edge)
                self.forest.union(a,b)

        self.adj_dict = defaultdict(list)
        for edge in self.tree:
            u,v = edge.endpoints()
            self.adj_dict[u].append(v)
            self.adj_dict[v].append(u)

        self.color = defaultdict()
        for v in self.g.vertices():
            self.color[v] = "white"
        path = [self.s]
        weights = [float("inf")]
        self.dfs(self.s, path, weights)

    def dfs(self, v, path, weights):
        if v == self.t:
            self.path = deepcopy(path)
            self.max_bandwidth = min(weights)
        self.color[v] = "grey"
        for w in self.adj_dict[v]:
            if self.color[w] == "white":
                path.append(w)
                weights.append(self.g.get_edge(v, w).weight())
                self.dfs(w, path, weights)
        path.pop()
        weights.pop()
```

The time complexity for this algorithm is:

1. for the initialization(), it takes O(n) to make groups and O(mlogm) to complete the heap sort of all the edges. As normally m is larger than n, so the time is O(mlogm)
2. for the find_mbp(): it takes O(mlog*n) to get the MST and takes O(n+n-1) time to get the path. So total time is O(mlog*n)

Thus the total time for Kruskal's algorithm is **O(mlogm)**

Below is the time complexity for three methods:

Algorithm	Time
Dijkstra-Heap	O(mn)
Dijkstra-Map	$O(n^2)$
Kruskal	O(mlogm)

2.4 Testing

The implementation for this part is basically easy:

1. Test you routing algorithms on 5 pairs of graphs, randomly generated using your subroutines implemented in Step 1.
2. For each generated graph, pick at least 5 pairs of randomly selected source-destination vertices. For each source-destination pair (s, t) on a graph G2, construct a path that goes through all vertices in the graph (We do not have to add such a path in G1 as we construct the graph by adding 3 such paths).
3. Run three algorithms on each pair and record their running time. Output the maximum bandwidth, time and path for each algorithm.

Below is the example code snippet for the main program.

```
for i in range(5):
        print "################################################################"
        print "######################## Round: ",i+1,"############################"
        print "################################################################"
        gg = GraphGeneration()
        gg.N = 5000
        #generating G1 and G2
        G1 = gg.generateG1()
        G2 = gg.generateG2()

        graphs = [G1,G2]
        graph_names = ["G1","G2"]
        for c in range(2):
            G = graphs[c]
            #G = G2
            print "****************************************************************"
            print "****************",graph_names[c],"*****************************"
            print "****************************************************************"
            #5 pairs of source-sink vertices
            for j in range(5):
                s = None
                t = None
                #s and t should be different
                while s == t:
                    s = G.get_vertex(random.randint(1,gg.N))
                    t = G.get_vertex(random.randint(1,gg.N))
                print "==============Count:", j+1, "Pair:(", s, t, ") ==============="
                print "Constructing path..."
                construct_path(G)
                #Dijkstra-No-Heap
                print "Dijkstra-No-Heap Finding Max-Bandwidth-Path in G..."
                start_time = time.time()
                mbpd = MBPDijkstra(G, s, t, False)
                mbpd.find_mbp()
                print "Done! Time:", time.time() - start_time
                print mbpd.str_mbp()
                #Dijkstra-Heap
                print "Dijkstra-Heap Finding Max-Bandwidth-Path in G..."
                start_time = time.time()
                mbpd_heap = MBPDijkstra(G, s, t, True)
```

```
mbpd_heap.find_mbp()
print "Done! Time:", time.time() - start_time
print mbpd_heap.str_mbp()
#Kruskal-HeapSort
print "Kruskal-HeapSort Finding Max-Bandwidth-Path in G..."
start_time = time.time()
mbpk = MBPKruskal(G, s, t)
mbpk.find_mbp()
print "Done! Time:", time.time() - start_time
print mbpk.str_mbp()
```

3.Example of Result:

There will be 5 rounds, each round generate G1 and G2 once, and for each graph generate 5 pairs of source-destination vertices. “Count” means the number of pairs have been checked.
Here as an example, I only give the first two pairs for G1 and G2 in the first round:

```
################################################################
######################### Round:  1 ############################
################################################################
Generating G1...
Generating G2...
************************************************************
************************* G1 *********************************
************************************************************
==============Count: 1 Pair:( 822 523 ) ================
Constructing path...
Dijkstra-No-Heap Finding Max-Bandwidth-Path in G...
Done! Time: 0.524725914001
Maximum bandwidth = 7730, The path is: [822, 1966, 2195, 894, 4767, 3606, 3206, 4792, 495, 2862, 3731, 4574,
2829, 1031, 2501, 4831, 4473, 2117, 1598, 1137, 3659, 2222, 1005, 4451, 2729, 238, 4039, 4613, 2978, 2966, 4962,
3741, 3202, 359, 523]

Dijkstra-Heap Finding Max-Bandwidth-Path in G...
Done! Time: 22.8738601208
Maximum bandwidth = 7730, The path is: [822, 1966, 2195, 894, 4767, 3606, 3206, 1672, 1016, 3126, 4233, 1286,
4787, 2722, 4598, 2496, 3787, 587, 477, 718, 2591, 3939, 4611, 204, 3566, 3995, 4372, 3471, 4197, 3735, 2264,
2046, 1723, 1767, 3679, 3087, 4148, 3741, 3202, 359, 523]

Kruskal-HeapSort Finding Max-Bandwidth-Path in G...
Done! Time: 1.17287802696
Maximum bandwidth = 7730, The path is: [822, 1966, 2195, 894, 4767, 3606, 3206, 1672, 1016, 3126, 2435, 4791,
53, 3827, 3819, 3870, 1261, 1068, 3262, 2067, 2511, 3847, 1674, 2792, 3465, 618, 434, 3765, 1490, 3692, 3195,
4025, 2219, 3582, 1124, 3255, 2918, 1365, 4204, 1721, 1844, 2721, 2081, 1963, 4772, 4378, 396, 384, 3531, 3853,
1478, 3422, 2991, 2156, 4913, 4601, 1614, 1437, 308, 4930, 4343, 1725, 176, 4283, 4538, 3066, 2297, 4572, 1769,
1413, 2826, 2780, 4949, 4007, 1139, 3236, 1836, 3129, 1168, 1820, 1669, 981, 486, 359, 523]

==============Count: 2 Pair:( 1378 4060 ) ================
Constructing path...
Dijkstra-No-Heap Finding Max-Bandwidth-Path in G...
Done! Time: 1.27094197273
Maximum bandwidth = 7913, The path is: [1378, 1835, 2097, 157, 3472, 4016, 2474, 4416, 1120, 2883,
2764, 4908, 1360, 1060, 2130, 4048, 3632, 3601, 3011, 1094, 4869, 2350, 4060]

Dijkstra-Heap Finding Max-Bandwidth-Path in G...
```

Done! Time: 23.866877079
Maximum bandwidth = 7913, The path is: [1378, 1835, 4467, 2169, 379, 4694, 4839, 851, 434, 700, 2701, 4407, 3515, 3183, 4835, 3030, 3740, 939, 2592, 1133, 2289, 1742, 1966, 4060]

Kruskal-HeapSort Finding Max-Bandwidth-Path in G...
Done! Time: 1.34926486015
Maximum bandwidth = 7913, The path is: [1378, 1835, 2097, 157, 3472, 4016, 4696, 1982, 3579, 3608, 3474, 1489, 1932, 740, 2823, 2565, 1909, 716, 969, 3858, 1767, 4898, 179, 2144, 4877, 2258, 3836, 2012, 1999, 3113, 4702, 1593, 3277, 2734, 4950, 111, 3687, 4101, 3278, 3593, 698, 4411, 1369, 4889, 4240, 425, 3112, 565, 3804, 3082, 2640, 1080, 3327, 1830, 421, 1569, 540, 938, 1880, 2422, 3403, 3769, 234, 891, 2065, 341, 2622, 1014, 1859, 2619, 1729, 2051, 3547, 4618, 3750, 1190, 3480, 1543, 3592, 4788, 946, 1102, 2586, 3528, 356, 3892, 3293, 3650, 884, 3582, 2219, 865, 1509, 1868, 663, 1829, 1128, 153, 3115, 2368, 4598, 525, 15, 1167, 2383, 354, 353, 2350, 4060]
……
****
********** G2 **********
****
==========Count: 1 Pair:(2130 4517) ==========
Constructing path...
Dijkstra-No-Heap Finding Max-Bandwidth-Path in G...
Done! Time: 19.2752799988
Maximum bandwidth = 9971, The path is: [2130, 3340, 1974, 4721, 3465, 2082, 3259, 1618, 3395, 1674, 2901, 407, 3482, 1905, 243, 1277, 1291, 3973, 1024, 2547, 3639, 474, 2120, 4763, 938, 3842, 1977, 4517]

Dijkstra-Heap Finding Max-Bandwidth-Path in G...
Done! Time: 243.649236917
Maximum bandwidth = 9971, The path is: [2130, 3340, 1916, 4209, 4185, 1283, 803, 4963, 1462, 2519, 305, 4863, 2954, 625, 3777, 841, 4285, 983, 319, 3544, 3048, 855, 4795, 1369, 3526, 3842, 1977, 4517]

Kruskal-HeapSort Finding Max-Bandwidth-Path in G...
Done! Time: 28.5181949139
Maximum bandwidth = 9971, The path is: [2130, 3340, 1916, 4209, 4185, 2464, 2829, 4838, 3390, 4760, 2263, 2828, 4898, 1385, 485, 1997, 3140, 1031, 1192, 1146, 4371, 1454, 4342, 1068, 1638, 1100, 2929, 601, 2646, 4626, 2181, 4811, 2577, 3696, 3854, 1458, 1180, 280, 4622, 2730, 889, 3871, 1095, 179, 4763, 938, 3842, 1977, 4517]

==========Count: 2 Pair:(1246 1534) ==========
Constructing path...
Dijkstra-No-Heap Finding Max-Bandwidth-Path in G...
Done! Time: 19.9205341339
Maximum bandwidth = 9953, The path is: [1246, 509, 1949, 3010, 3193, 754, 4794, 3651, 4370, 3089, 2403, 3379, 3818, 3936, 2036, 546, 2506, 3039, 4994, 784, 4177, 4500, 1452, 2157, 3399, 2491, 1180, 1458, 3854, 3696, 467, 3589, 2618, 617, 1944, 4471, 878, 1023, 4902, 328, 3034, 4432, 1925, 2477, 2295, 132, 2222, 1096, 769, 252, 3966, 2190, 2156, 4222, 576, 2678, 346, 4693, 2188, 253, 4773, 4843, 1534]

Dijkstra-Heap Finding Max-Bandwidth-Path in G...
Done! Time: 256.717787981
Maximum bandwidth = 9953, The path is: [1246, 4607, 733, 2029, 873, 3854, 3696, 2577, 4811, 2181, 4626, 2646, 2077, 132, 2222, 1096, 769, 252, 3966, 2190, 2156, 4222, 576, 2678, 346, 4693, 2188, 253, 4773, 4843, 1534]

Kruskal-HeapSort Finding Max-Bandwidth-Path in G...
Done! Time: 27.6678318977
Maximum bandwidth = 9953, The path is: [1246, 4607, 733, 2029, 873, 3854, 3840, 4215, 4512, 4069, 1915, 1154, 45, 229, 3568, 2228, 4862, 4330, 1257, 1943, 4247, 3515, 4138, 3904, 2019, 4567, 2107, 839, 364, 81, 1876, 1462, 4963, 803, 4412, 2628, 4166, 3480, 2080, 3980, 4372, 3796, 1096, 769, 252, 3966, 2190, 2156, 4222, 576, 2678, 346, 4693, 2188, 253, 4773, 4843, 1534]
……

4.Conclusion:

1. The value of the maximum bandwidth generated from the three algorithms are the same. The path each algorithm generated may differ.
2. For my case, the time reported by the program shows that Dijkstra's algorithm using map is the fastest, the second is Kruskal's algorithm and the last is Dijkstra's algorithm using heap.
3. The graph of time-input size for three algorithms:

5.Future improvements:

1. It seems that python is not a efficiency programming language to focus on data structures and algorithms. It is easy to implement but actually much slower than C++/C/Java. So the transformation of code into more basic language in the future will shorten the time.
2. Design a different data structure for the heap so that the time of removal of any element (by key, not by index) will be reduced from O(n) to O(logn), which will definitely decrease the time spent on Dijkstra heap will be shorter than Dijkstra map.
3. We could use some existing python library which provides faster implementation of data structures and algorithms. The data structures in Python could be made more light-weight by using this libraries which include features of C/C++ into implementation.