CSCE 629 Analysis of Algorithms

Project report

Name: Yanxiang Yang

UIN: 324006110

Introduction

This project is focusing on network routing algorithms. The goal is to find a maximum bandwidth path of a network. In this project, famous algorithm like Dijkstra's Algorithm and Kruskal's Algrithm are modified to solve this routing problem. Three different approaches are implemented. They are tested in 5 pairs of different random graphs. Details analysis and comparison are made to verify the concepts we learnt in class.

Implementation and Design

1. Graph

The first problem is to implement graph ADT. There several data structures for representing a graph. No matter how we are going to implement this graph, we have to maintain two collections for vertices and edges respectively. However, different data structures to organize edges differ a lot in performance. Edge List (edge array) is the easiest way to organize and store edges. However, this data structure is not efficient to locate a particular edge(u, v), or the set of all edges incident to a vertex v. Another way to organize edges is called Adjacency Matrix. It provides worst-case O(1) access to a specific edge(u, v). However, it have to generate and maintain an n*n matrix. It takes O(n*n) time to insert or remove a vertex. This data structure is more suitable for a dense graph(the number of edges is proportional to n*n). One very popular way to organize edges is Adjacency List. It maintains a separate list containing those edges that are incident to the vertex. This way is very efficient to get all edges incident to a given vertex. Adjacency Map is a very similar way as Adjacency List, but the secondary container of all edges incident to a vertex is organized as a map, rather than as a linked list, with the adjacent vertex serving as a key. The advantage of this way is that we can access to a specific edge(u, v) in O(1) expected time.

In this project, I utilize the adjacency map representation. Different from the adjacency list structure, we use dictionaries to maintain incidence collections rather than use unordered linked lists. For adjacency list structure, such an incidence collection I(v) uses space proportional to O(deg(v)), allows an edge to be added or removed in O(1) time, and allows an iteration of all edges incident to vertex v in O(deg(v)). However, the best implementation of getting an edge(u, v) requires O(min(deg(u), deg(v))) time. The advantage of the adjacency map, relative to an adjacency list, is that getting an edge(u, v) can be solved in expected O(1) time by searching for vertex u as a key in incidence collection I(v), or vice versa.

The integer numbers from 1 to 5000 are used to denote the 5000 vertices. A tuple in python, like (u, v, weight), is used to denote an edge. A simple example, like {‘1’:{‘2’: (1, 2, 100)}, ‘2’:{‘1’: (1, 2, 100)}}}, indicates that there two vertices 1 and 2, and an edge between 1 and 2 with weight 100.

2. Random graph generation

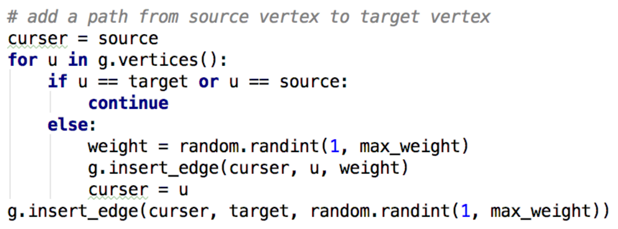
The first step and basic requirement of this project is to generate two different types of random graph. Both two have 5000 vertices. For type one, every vertex should have exactly 6 degree. For the other one, every vertex has edges going to 20% of the other vertices.

a. Source vertex and target vertex generation:

   A random number between 1 and 2000 is chosen as a source vertex. A random number between 3000 and 4999 is chosen as a target vertex.

b. Adding a path from source to target that goes through all vertices in the graph G.

   The algorithm is: (In Python3)

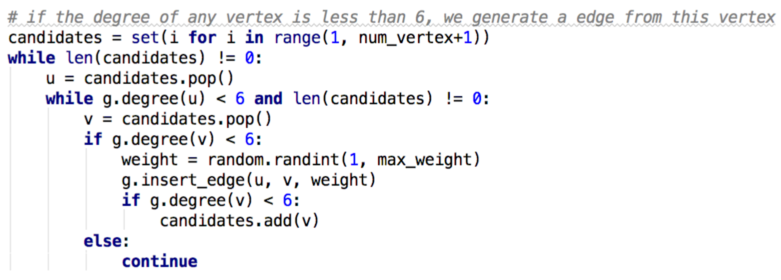


```python
# add a path from source vertex to target vertex
curser = source
for u in g.vertices():
    if u == target or u == source:
        continue
    else:
        weight = random.randint(1, max_weight)
        g.insert_edge(curser, u, weight)
        curser = u
g.insert_edge(curser, target, random.randint(1, max_weight))
```

   The basic idea: I used a variable named curser to indicate current vertex we are handling. Curser is initialized to source. Choosing a vertex from vertices that has not been seen in the graph. If this vertex is source or target, then we skip it as it has been seen. Otherwise, we generate an edge between curser and this vertex with random opposite weight. Then, we replace curser with this vertex. In the end, we add target to this path by generating an edge between curser and target. This algorithm takes linear time to generate this path, O(n) where n is the number of vertices in the graph.

c. Generating a graph with 5000 vertices, and each vertex has exactly 6 degree.

   The algorithm is: (In Python3)

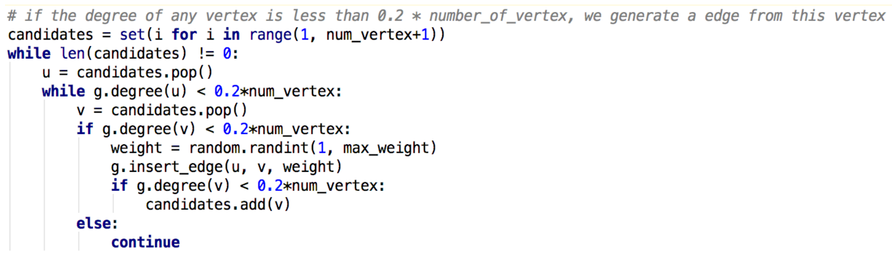


```python
# if the degree of any vertex is less than 6, we generate a edge from this vertex
candidates = set(i for i in range(1, num_vertex+1))
while len(candidates) != 0:
    u = candidates.pop()
    while g.degree(u) < 6 and len(candidates) != 0:
        v = candidates.pop()
        if g.degree(v) < 6:
            weight = random.randint(1, max_weight)
            g.insert_edge(u, v, weight)
            if g.degree(v) < 6:
                candidates.add(v)
        else:
            continue
```

   At the beginning, we have a set of candidates to insert edges between them. In python, set.pop() will randomly return an element and remove it from that set. We first randomly select one vertex A from candidates set. If the degree of this vertex A is less than 6, we randomly select one another vertex B from the rest of candidates set with degree less than 6(otherwise we delete this vertex B from candidates set), we add an edge between vertex A and vertex B. If the degree of vertex B is still less than 6, put back vertex B to candidates set. We keep doing these steps until there is no element in candidates set.

Because it takes O(1) to pop an random item from the set. And every time we delete vertices that have degree not less than 6 from candidates set when meeting them. Therefore the overall execution times is about 6*n. The total running time of this algorithm is O(n).

d. Generating a graph with 5000 vertices, and each vertex has edges going to about 20% of the other vertices.
The algorithm is: (In Python3)



```
# if the degree of any vertex is less than 0.2 * number_of_vertex, we generate a edge from this vertex
candidates = set(i for i in range(1, num_vertex+1))
while len(candidates) != 0:
    u = candidates.pop()
    while g.degree(u) < 0.2*num_vertex:
        v = candidates.pop()
        if g.degree(v) < 0.2*num_vertex:
            weight = random.randint(1, max_weight)
            g.insert_edge(u, v, weight)
            if g.degree(v) < 0.2*num_vertex:
                candidates.add(v)
        else:
            continue
```

The basic idea of this graph generation algorithm is the same as the previous one. The main difference is that each vertex has edges going to 20% of the other vertices. I simply changed 6 to 0.2*n, where n is the number of vertices. The overall execution times is about 0.2*n*n, so the running time of this algorithm is O(n*n).

3. Heap

A heap is a specialized tree-based data structure that satisfies the heap property: if A is a parent of node B, then the key of node A is order with respect to the key of node B with the same ordering applying across the heap. And in this project, we only use max heap, which means the keys of parent nodes are always greater than or equal to those of children. And the largest key is stored at root. I utilize heap based priority queue to store fringes for modified Dijskra's algorithm and also all the edges of the graph when it comes to modified Kruskal's algorithm. The main advantage of heap is that we can retrieve max item in O(1), and remove max item in O(log n).

The main reference function in our heap-based priority queue is:

```python
def _heapify(self):
    start = self._parent(len(self) - 1)
    for j in range(start, -1, -1):
        self._downheap(j)

def __len__(self):
    return len(self._data)

def add(self, key, value):
    self._data.append(self._Item(key, value))
    self._upheap(len(self._data) - 1)

def max(self):
    """ return but do not remove (k, v) tuple with minimun key.
    raise empty exception if empty.
    """
    if self.is_empty():
        raise Empty('Priority queue is empty.')
    item = self._data[0]
    return (item._key, item._value)

def remove_max(self):
    """ remove and return (k, v) tuple with minimum key. """
    if self.is_empty():
        raise Empty('Priority queue is empty.')
    self._swap(0, len(self._data) - 1)
    item = self._data.pop()
    self._downheap(0)
    return (item._key, item._value)
```

4. Modified Dijkstra's Algorithm without heap structure

To find a maximum path of a graph, the first approach is to modify Dijkstra's algorithm. Instead of finding a shortest path, now we are trying to find maximum bandwidth path. The main difference between these two is the edge relaxation.

For shortest path problem:

If D[v] > D[u] + weight(u, v) then:

D[v] = D[u] + weight(u, v)

For max bandwidth problem:

If D[v] < min (D[u], weight(u, v)) then:

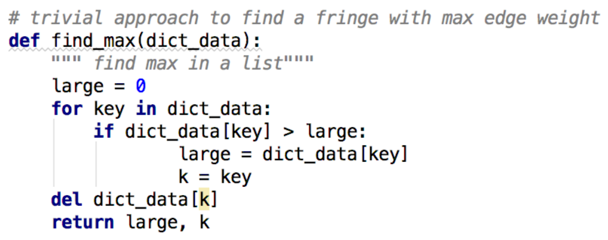
D[v] = min (D[u], weight(u, v))

The algorithm: (In Python3)

```python
def max_bandwidth_no_heap(g, src, target):

    status = {}  # status map
    parent = {}  # store the parent of current vertex
    fringe = {} # fringes are stored in a list
    bw = {}  # bandwidth associated with each vertex

    for v in g.vertices():
        status[v] = 'unseen'

    bw[src] = float('inf')
    parent[src] = None
    status[src] = 'intree'

    for edge in g.incident_edges(src):
        v = get_opposite(edge, src)
        parent[v] = src
        status[v] = 'fringe'
        bw[v] = edge[2]
        fringe[v] = bw[v]

    while (len(fringe) != 0) and (not status[target] == 'intree'):
        weight, vertex = find_max(fringe)
        bw[vertex] = weight
        status[vertex] = 'intree'
        for edge in g.incident_edges(vertex):
            v = get_opposite(edge, vertex)
            w = edge[2]
            if status[v] == 'unseen':
                status[v] = 'fringe'
                parent[v] = vertex
                bw[v] = min(bw[vertex], w)
                fringe[v] = bw[v]
            elif status[v] == 'fringe' and bw[v] < min(bw[vertex], w):
                parent[v] = vertex
                bw[v] = min(bw[vertex], w)
                fringe[v] = bw[v]

    if not status[target] == 'intree':
        print('No such path from source to target!')
    maxbandwidth = bw[target]

    path =[]
    path.append(target)
    find_path(parent,src, target, path)
    path.reverse()
    return path, maxbandwidth
```

Because for the very first try, we do not use heap data structure to store fringes. We have to design a trivial algorithm to find maximum value among fringes.

The algorithm: (In Python3)



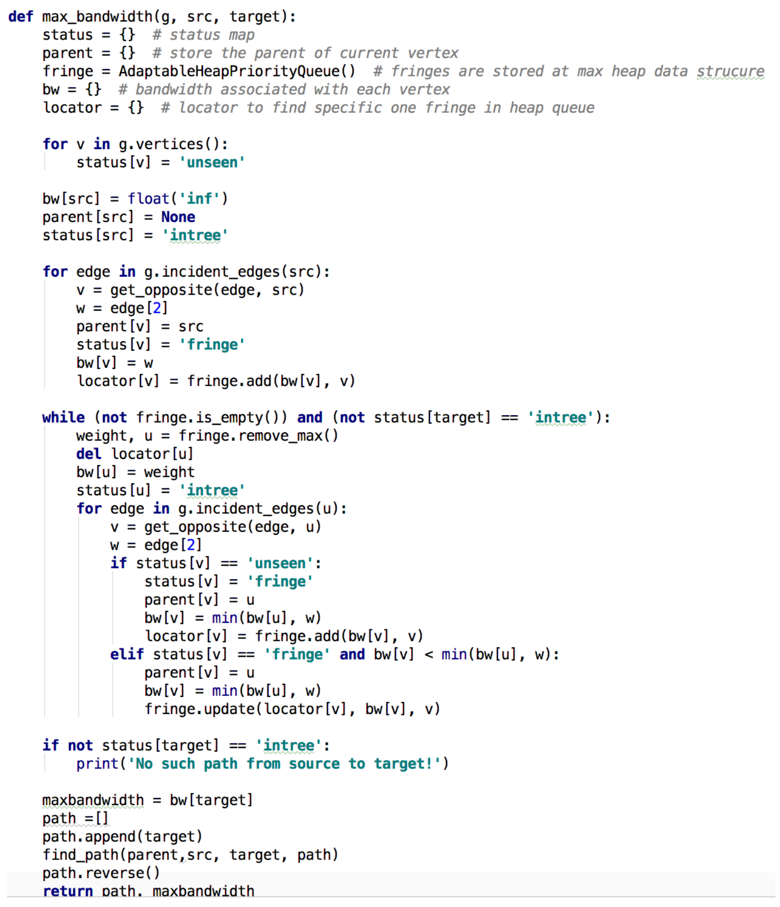
```
# trivial approach to find a fringe with max edge weight
def find_max(dict_data):
    """ find max in a list"""
    large = 0
    for key in dict_data:
        if dict_data[key] > large:
                large = dict_data[key]
                k = key
    del dict_data[k]
    return large, k
```

This algorithm will execute the-number-of-fringes times for the worst case, and fringes can be up to n where n is the number of vertices in the graph. Thus, this subroutine takes O(n) time. At the section of analysis and discussion, I will explain this in detail and why the whole algorithm runs in O(n*n) time.

5. Modified Dijkstra's Algorithm with heap structure

This is the second try with Dijkstra's algorithm, this time we are trying to use heap data structure. By using this data structure may somewhat improve our algorithm. Instead of organizing fringes in un-ordered dictionary, we store them in a heap-based priority queue.

The algorithm is: (In Python3)



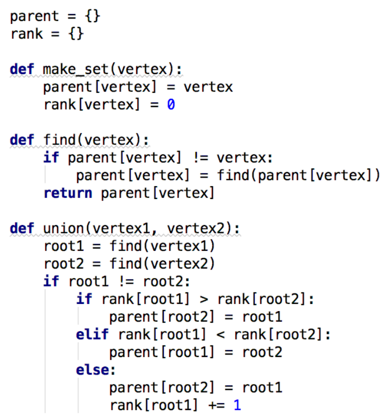
```
def max_bandwidth(g, src, target):
    status = {}  # status map
    parent = {}  # store the parent of current vertex
    fringe = AdaptableHeapPriorityQueue()  # fringes are stored at max heap data strucure
    bw = {}  # bandwidth associated with each vertex
    locator = {}  # locator to find specific one fringe in heap queue

    for v in g.vertices():
        status[v] = 'unseen'

    bw[src] = float('inf')
    parent[src] = None
    status[src] = 'intree'

    for edge in g.incident_edges(src):
        v = get_opposite(edge, src)
        w = edge[2]
        parent[v] = src
        status[v] = 'fringe'
        bw[v] = w
        locator[v] = fringe.add(bw[v], v)

    while (not fringe.is_empty()) and (not status[target] == 'intree'):
        weight, u = fringe.remove_max()
        del locator[u]
        bw[u] = weight
        status[u] = 'intree'
        for edge in g.incident_edges(u):
            v = get_opposite(edge, u)
            w = edge[2]
            if status[v] == 'unseen':
                status[v] = 'fringe'
                parent[v] = u
                bw[v] = min(bw[u], w)
                locator[v] = fringe.add(bw[v], v)
            elif status[v] == 'fringe' and bw[v] < min(bw[u], w):
                parent[v] = u
                bw[v] = min(bw[u], w)
                fringe.update(locator[v], bw[v], v)

    if not status[target] == 'intree':
        print('No such path from source to target!')

    maxbandwidth = bw[target]
    path =[]
    path.append(target)
    find_path(parent,src, target, path)
    path.reverse()
    return path, maxbandwidth
```

6. Modified Kruskal's Algorithm

Kruskal's algorithm is the basic algorithm to find a minimum spanning tree in a graph. To solve maximum bandwidth path problem, we need to modify it a little bit. Rather than to find a minimum-spanning tree, we need to find maximum spanning tree first. The paths between any pair of nodes in this max spanning tree are the max bandwidth paths.

The main change of Kruskal's algorithm is that instead of considering edges in non-decreasing order, we need to organize and handle edges in non-increasing order.

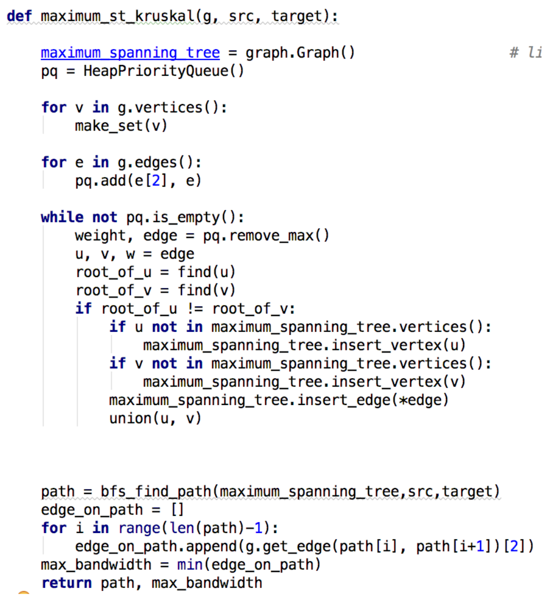
The following is the Union-Find subroutines:



```
parent = {}
rank = {}

def make_set(vertex):
    parent[vertex] = vertex
    rank[vertex] = 0

def find(vertex):
    if parent[vertex] != vertex:
        parent[vertex] = find(parent[vertex])
    return parent[vertex]

def union(vertex1, vertex2):
    root1 = find(vertex1)
    root2 = find(vertex2)
    if root1 != root2:
        if rank[root1] > rank[root2]:
            parent[root2] = root1
        elif rank[root1] < rank[root2]:
            parent[root1] = root2
        else:
            parent[root2] = root1
            rank[root1] += 1
```

To achieve Union-Find in time O(logn), we attach the shorter tree to the taller tree by introducing another array(I utilized dictionary in this project). If r is a root, then rank[r] is the height of the tree.

The modified Kruskal's algorithm is: (In Python3)

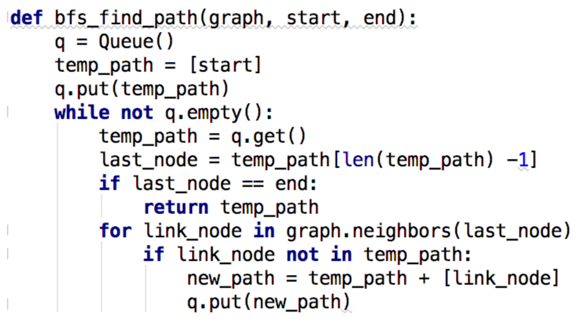


```
def maximum_st_kruskal(g, src, target):

    maximum_spanning_tree = graph.Graph()                # li
    pq = HeapPriorityQueue()

    for v in g.vertices():
        make_set(v)

    for e in g.edges():
        pq.add(e[2], e)

    while not pq.is_empty():
        weight, edge = pq.remove_max()
        u, v, w = edge
        root_of_u = find(u)
        root_of_v = find(v)
        if root_of_u != root_of_v:
            if u not in maximum_spanning_tree.vertices():
                maximum_spanning_tree.insert_vertex(u)
            if v not in maximum_spanning_tree.vertices():
                maximum_spanning_tree.insert_vertex(v)
            maximum_spanning_tree.insert_edge(*edge)
            union(u, v)


    path = bfs_find_path(maximum_spanning_tree,src,target)
    edge_on_path = []
    for i in range(len(path)-1):
        edge_on_path.append(g.get_edge(path[i], path[i+1])[2])
    max_bandwidth = min(edge_on_path)
    return path, max_bandwidth
```

The basic idea is to find maximum spanning tree first and then find the path between source and target in the maximum spanning tree.

7. BFS to find path on maximum spanning tree

Breadth-first search is an algorithm for traversing tree or graph. It starts at the tree root and explores the neighbor nodes first, before moving to the next level neighbors.



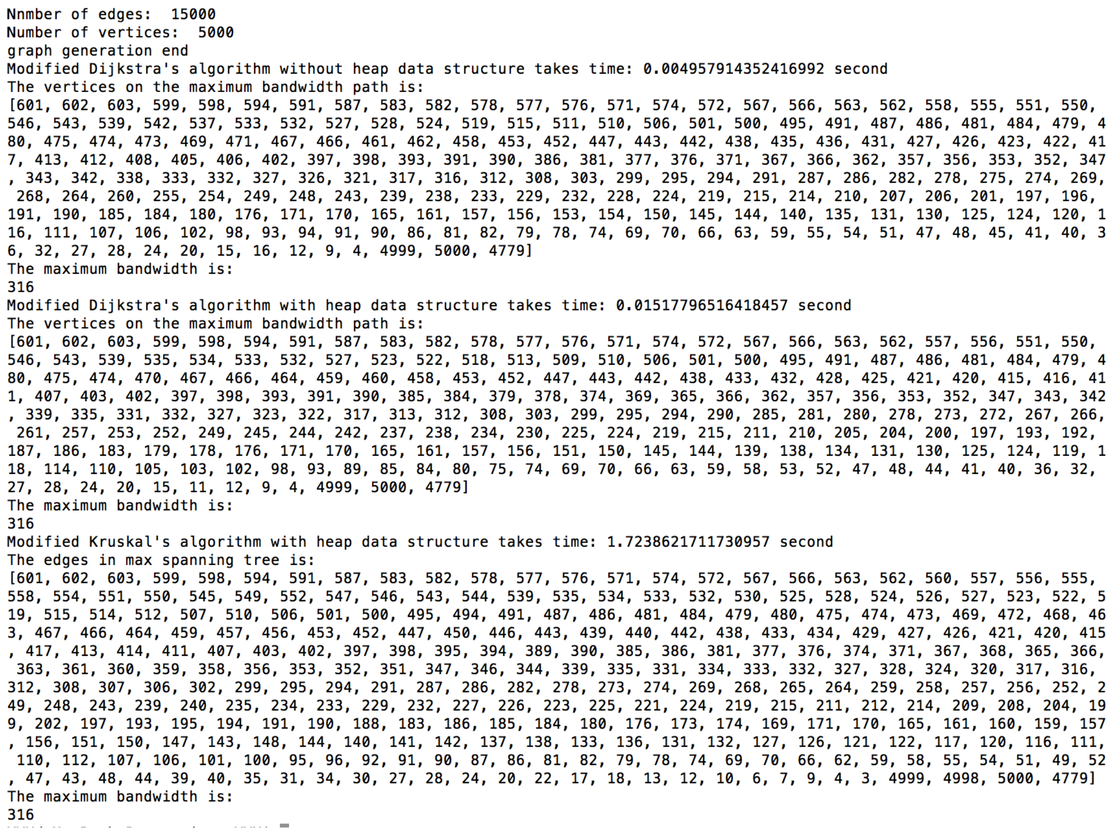
```
def bfs_find_path(graph, start, end):
    q = Queue()
    temp_path = [start]
    q.put(temp_path)
    while not q.empty():
        temp_path = q.get()
        last_node = temp_path[len(temp_path) -1]
        if last_node == end:
            return temp_path
        for link_node in graph.neighbors(last_node):
            if link_node not in temp_path:
                new_path = temp_path + [link_node]
                q.put(new_path)
```

This algorithm takes linear time to find the path from the source to the target.

Testing and results

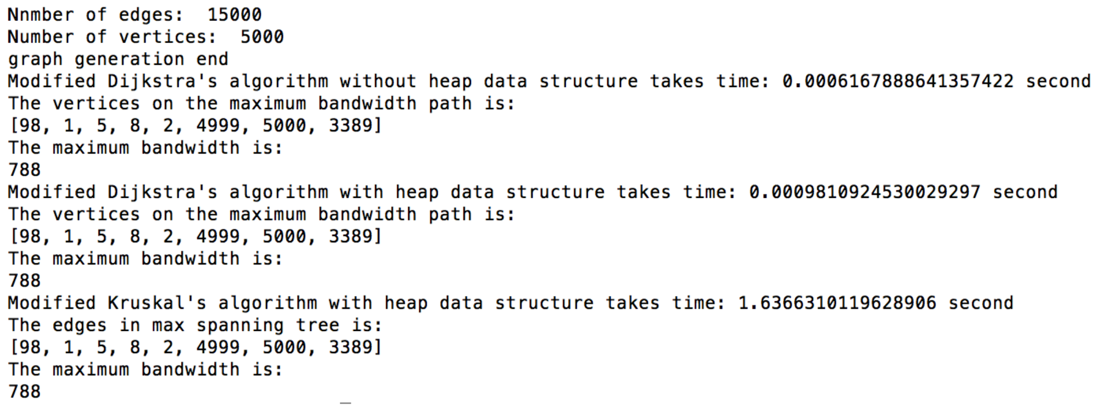
1. Test for graph 1:

The first time to running the entire program, the result is showing below (program output):



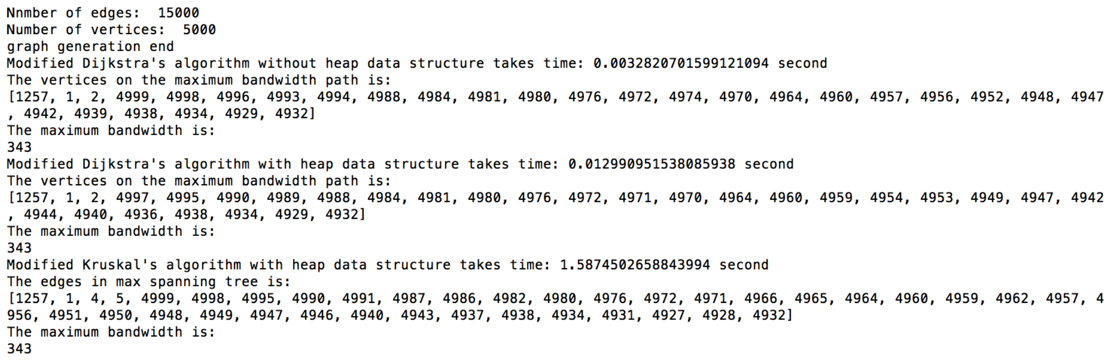
```
Nnmber of edges:  15000
Number of vertices:  5000
graph generation end
Modified Dijkstra's algorithm without heap data structure takes time: 0.004957914352416992 second
The vertices on the maximum bandwidth path is:
[601, 602, 603, 599, 598, 594, 591, 587, 583, 582, 578, 577, 576, 571, 574, 572, 567, 566, 563, 562, 558, 555, 551, 550,
546, 543, 539, 542, 537, 533, 532, 527, 528, 524, 519, 515, 511, 510, 506, 501, 500, 495, 491, 487, 486, 481, 484, 479, 4
80, 475, 474, 473, 469, 471, 467, 466, 461, 462, 458, 453, 452, 447, 443, 442, 438, 435, 436, 431, 427, 426, 423, 422, 41
7, 413, 412, 408, 405, 406, 402, 397, 398, 393, 391, 390, 386, 381, 377, 376, 371, 367, 366, 362, 357, 356, 353, 352, 347
, 343, 342, 338, 333, 332, 327, 326, 321, 317, 316, 312, 308, 303, 299, 295, 294, 291, 287, 286, 282, 278, 275, 274, 269,
 268, 264, 260, 255, 254, 249, 248, 243, 239, 238, 233, 229, 232, 228, 224, 219, 215, 214, 210, 207, 206, 201, 197, 196,
191, 190, 185, 184, 180, 176, 171, 170, 165, 161, 157, 156, 153, 154, 150, 145, 144, 140, 135, 131, 130, 125, 124, 120, 1
16, 111, 107, 106, 102, 98, 93, 94, 91, 90, 86, 81, 82, 79, 78, 74, 69, 70, 66, 63, 59, 55, 54, 51, 47, 48, 45, 41, 40, 3
6, 32, 27, 28, 24, 20, 15, 16, 12, 9, 4, 4999, 5000, 4779]
The maximum bandwidth is:
316
Modified Dijkstra's algorithm with heap data structure takes time: 0.01517796516418457 second
The vertices on the maximum bandwidth path is:
[601, 602, 603, 599, 598, 594, 591, 587, 583, 582, 578, 577, 576, 571, 574, 572, 567, 566, 563, 562, 557, 556, 551, 550,
546, 543, 539, 535, 534, 533, 532, 527, 523, 522, 518, 513, 509, 510, 506, 501, 500, 495, 491, 487, 486, 481, 484, 479, 4
80, 475, 474, 470, 467, 466, 464, 459, 460, 458, 453, 452, 447, 443, 442, 438, 433, 432, 428, 425, 421, 420, 415, 416, 41
1, 407, 403, 402, 397, 398, 393, 391, 390, 385, 384, 379, 378, 374, 369, 365, 366, 362, 357, 356, 353, 352, 347, 343, 342
, 339, 335, 331, 332, 327, 323, 322, 317, 313, 312, 308, 303, 299, 295, 294, 290, 285, 281, 280, 278, 273, 272, 267, 266,
 261, 257, 253, 252, 249, 245, 244, 242, 237, 238, 234, 230, 225, 224, 219, 215, 211, 210, 205, 204, 200, 197, 193, 192,
187, 186, 183, 179, 178, 176, 171, 170, 165, 161, 157, 156, 151, 150, 145, 144, 139, 138, 134, 131, 130, 125, 124, 119, 1
18, 114, 110, 105, 103, 102, 98, 93, 89, 85, 84, 80, 75, 74, 69, 70, 66, 63, 59, 58, 53, 52, 47, 48, 44, 41, 40, 36, 32,
27, 28, 24, 20, 15, 11, 12, 9, 4, 4999, 5000, 4779]
The maximum bandwidth is:
316
Modified Kruskal's algorithm with heap data structure takes time: 1.7238621711730957 second
The edges in max spanning tree is:
[601, 602, 603, 599, 598, 594, 591, 587, 583, 582, 578, 577, 576, 571, 574, 572, 567, 566, 563, 562, 560, 557, 556, 555,
558, 554, 551, 550, 545, 549, 552, 547, 546, 543, 544, 539, 535, 534, 533, 532, 530, 525, 528, 524, 526, 527, 523, 522, 5
19, 515, 514, 512, 507, 510, 506, 501, 500, 495, 494, 491, 487, 486, 481, 484, 479, 480, 475, 474, 473, 469, 472, 468, 46
3, 467, 466, 464, 459, 457, 456, 453, 452, 447, 450, 446, 443, 439, 440, 442, 438, 433, 434, 429, 427, 426, 421, 420, 415
, 417, 413, 414, 411, 407, 403, 402, 397, 398, 395, 394, 389, 390, 385, 386, 381, 377, 376, 374, 371, 367, 368, 365, 366,
 363, 361, 360, 359, 358, 356, 353, 352, 351, 347, 346, 344, 339, 335, 331, 334, 333, 332, 327, 328, 324, 320, 317, 316,
312, 308, 307, 306, 302, 299, 295, 294, 291, 287, 286, 282, 278, 273, 274, 269, 268, 265, 264, 259, 258, 257, 256, 252, 2
49, 248, 243, 239, 240, 235, 234, 233, 229, 232, 227, 226, 223, 225, 221, 224, 219, 215, 211, 212, 214, 209, 208, 204, 19
9, 202, 197, 193, 195, 194, 191, 190, 188, 183, 186, 185, 184, 180, 176, 173, 174, 169, 171, 170, 165, 161, 160, 159, 157
, 156, 151, 150, 147, 143, 148, 144, 140, 141, 142, 137, 138, 133, 136, 131, 132, 127, 126, 121, 122, 117, 120, 116, 111,
 110, 112, 107, 106, 101, 100, 95, 96, 92, 91, 90, 87, 86, 81, 82, 79, 78, 74, 69, 70, 66, 62, 59, 58, 55, 54, 51, 49, 52
, 47, 43, 48, 44, 39, 40, 35, 31, 34, 30, 27, 28, 24, 20, 22, 17, 18, 13, 12, 10, 6, 7, 9, 4, 3, 4999, 4998, 5000, 4779]
The maximum bandwidth is:
316
```

The second time to running the entire program, the result is showing below (program output):



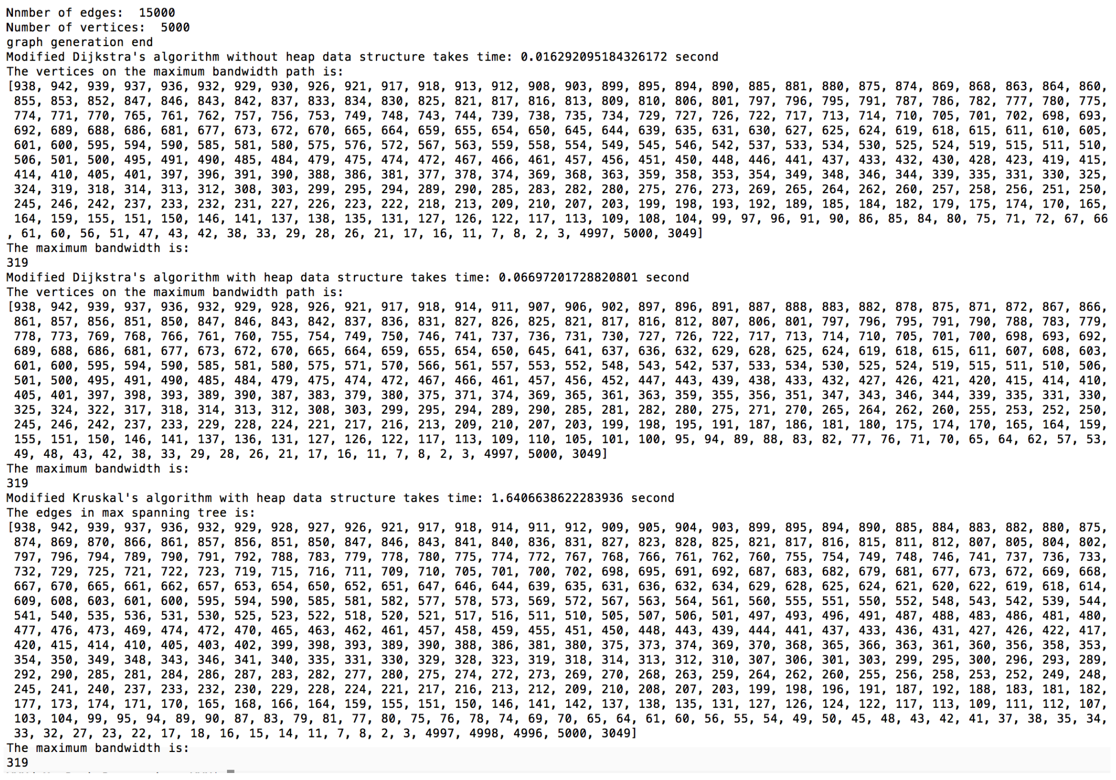
```
Nnmber of edges:  15000
Number of vertices:  5000
graph generation end
Modified Dijkstra's algorithm without heap data structure takes time: 0.0006167888641357422 second
The vertices on the maximum bandwidth path is:
[98, 1, 5, 8, 2, 4999, 5000, 3389]
The maximum bandwidth is:
788
Modified Dijkstra's algorithm with heap data structure takes time: 0.0009810924530029297 second
The vertices on the maximum bandwidth path is:
[98, 1, 5, 8, 2, 4999, 5000, 3389]
The maximum bandwidth is:
788
Modified Kruskal's algorithm with heap data structure takes time: 1.6366310119628906 second
The edges in max spanning tree is:
[98, 1, 5, 8, 2, 4999, 5000, 3389]
The maximum bandwidth is:
788
```

The third time to running the entire program, the result is showing below (program output):



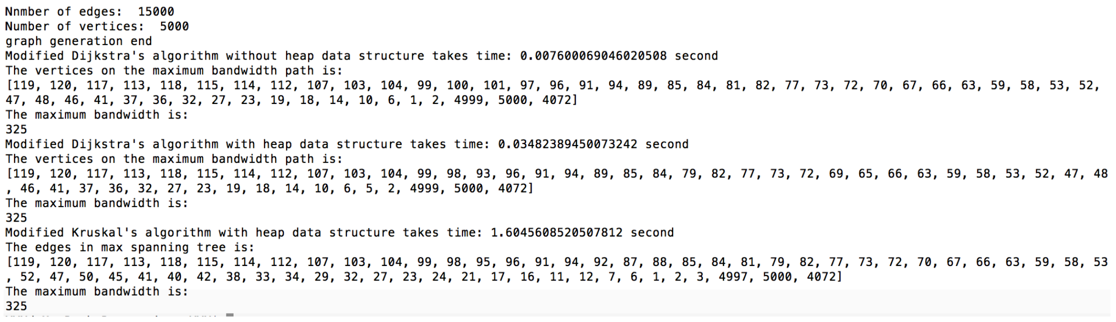
```
Nnmber of edges:  15000
Number of vertices:  5000
graph generation end
Modified Dijkstra's algorithm without heap data structure takes time: 0.0032820701599121094 second
The vertices on the maximum bandwidth path is:
[1257, 1, 2, 4999, 4998, 4996, 4993, 4994, 4988, 4984, 4981, 4980, 4976, 4972, 4974, 4970, 4964, 4960, 4957, 4956, 4952, 4948, 4947
, 4942, 4939, 4938, 4934, 4929, 4932]
The maximum bandwidth is:
343
Modified Dijkstra's algorithm with heap data structure takes time: 0.012990951538085938 second
The vertices on the maximum bandwidth path is:
[1257, 1, 2, 4997, 4995, 4990, 4989, 4988, 4984, 4981, 4980, 4976, 4972, 4971, 4970, 4964, 4960, 4959, 4954, 4953, 4949, 4947, 4942
, 4944, 4940, 4936, 4938, 4934, 4929, 4932]
The maximum bandwidth is:
343
Modified Kruskal's algorithm with heap data structure takes time: 1.5874502658843994 second
The edges in max spanning tree is:
[1257, 1, 4, 5, 4999, 4998, 4995, 4990, 4991, 4987, 4986, 4982, 4980, 4976, 4972, 4971, 4966, 4965, 4964, 4960, 4959, 4962, 4957, 4
956, 4951, 4950, 4948, 4949, 4947, 4946, 4940, 4943, 4937, 4938, 4934, 4931, 4927, 4928, 4932]
The maximum bandwidth is:
343
```

The fourth time to running the entire program, the result is showing below (program output):



```
Nnmber of edges:  15000
Number of vertices:  5000
graph generation end
Modified Dijkstra's algorithm without heap data structure takes time: 0.016292095184326172 second
The vertices on the maximum bandwidth path is:
[938, 942, 939, 937, 936, 932, 929, 930, 926, 921, 917, 918, 913, 912, 908, 903, 899, 895, 894, 890, 885, 881, 880, 875, 874, 869, 868, 863, 864, 860,
 855, 853, 852, 847, 846, 843, 842, 837, 833, 834, 830, 825, 821, 817, 816, 813, 809, 810, 806, 801, 797, 796, 795, 791, 787, 786, 782, 777, 780, 775,
 774, 771, 770, 765, 761, 762, 757, 756, 753, 749, 748, 743, 744, 739, 738, 735, 734, 729, 727, 726, 722, 717, 713, 714, 710, 705, 701, 702, 698, 693,
 692, 689, 688, 686, 681, 677, 673, 672, 670, 665, 664, 659, 655, 654, 650, 645, 644, 639, 635, 631, 630, 627, 625, 624, 619, 618, 615, 611, 610, 605,
 601, 600, 595, 594, 590, 585, 581, 580, 575, 576, 572, 567, 563, 559, 558, 554, 549, 545, 546, 542, 537, 533, 534, 530, 525, 524, 519, 515, 511, 510,
 506, 501, 500, 495, 491, 490, 485, 484, 479, 475, 474, 472, 467, 466, 461, 457, 456, 451, 450, 448, 446, 441, 437, 433, 432, 430, 428, 423, 419, 415,
 414, 410, 405, 401, 397, 396, 391, 390, 388, 386, 381, 377, 378, 374, 369, 368, 363, 359, 358, 353, 354, 349, 348, 346, 344, 339, 335, 331, 330, 325,
 324, 319, 318, 314, 313, 312, 308, 303, 299, 295, 294, 289, 290, 285, 283, 282, 280, 275, 276, 273, 269, 265, 264, 262, 260, 257, 258, 256, 251, 250,
 245, 246, 242, 237, 233, 232, 231, 227, 226, 223, 222, 218, 213, 209, 210, 207, 203, 199, 198, 193, 192, 189, 185, 184, 182, 179, 175, 174, 170, 165,
 164, 159, 155, 151, 150, 146, 141, 137, 138, 135, 131, 127, 126, 122, 117, 113, 109, 108, 104, 99, 97, 96, 91, 90, 86, 85, 84, 80, 75, 71, 72, 67, 66
, 61, 60, 56, 51, 47, 43, 42, 38, 33, 29, 28, 26, 21, 17, 16, 11, 7, 8, 2, 3, 4997, 5000, 3049]
The maximum bandwidth is:
319
Modified Dijkstra's algorithm with heap data structure takes time: 0.06697201728820801 second
The vertices on the maximum bandwidth path is:
[938, 942, 939, 937, 936, 932, 929, 928, 926, 921, 917, 918, 914, 911, 907, 906, 902, 897, 896, 891, 887, 888, 883, 882, 878, 875, 871, 872, 867, 866,
 861, 857, 856, 851, 850, 847, 846, 843, 842, 837, 836, 831, 827, 826, 825, 821, 817, 816, 812, 807, 806, 801, 797, 796, 795, 791, 790, 788, 783, 779,
 778, 773, 769, 768, 766, 761, 760, 755, 754, 749, 750, 746, 741, 737, 736, 731, 730, 727, 726, 722, 717, 713, 714, 710, 705, 701, 700, 698, 693, 692,
 689, 688, 686, 681, 677, 673, 672, 670, 665, 664, 659, 655, 654, 650, 645, 641, 637, 636, 632, 629, 628, 625, 624, 619, 618, 615, 611, 607, 608, 603,
 601, 600, 595, 594, 590, 585, 581, 580, 575, 571, 570, 566, 561, 557, 553, 552, 548, 543, 542, 537, 533, 534, 530, 525, 524, 519, 515, 511, 510, 506,
 501, 500, 495, 491, 490, 485, 484, 479, 475, 474, 472, 467, 466, 461, 457, 456, 452, 447, 443, 439, 438, 433, 432, 427, 426, 421, 420, 415, 414, 410,
 405, 401, 397, 398, 393, 389, 390, 387, 383, 379, 380, 375, 371, 374, 369, 365, 361, 363, 359, 355, 356, 351, 347, 343, 346, 344, 339, 335, 331, 330,
 325, 324, 322, 317, 318, 314, 313, 312, 308, 303, 299, 295, 294, 289, 290, 285, 281, 282, 280, 275, 271, 270, 265, 264, 262, 260, 255, 253, 252, 250,
 245, 246, 242, 237, 233, 229, 228, 224, 221, 217, 216, 213, 209, 210, 207, 203, 199, 198, 195, 191, 187, 186, 181, 180, 175, 174, 170, 165, 164, 159,
 155, 151, 150, 146, 141, 137, 136, 131, 127, 126, 122, 117, 113, 109, 110, 105, 101, 100, 95, 94, 89, 88, 83, 82, 77, 76, 71, 70, 65, 64, 62, 57, 53,
 49, 48, 43, 42, 38, 33, 29, 28, 26, 21, 17, 16, 11, 7, 8, 2, 3, 4997, 5000, 3049]
The maximum bandwidth is:
319
Modified Kruskal's algorithm with heap data structure takes time: 1.6406638622283936 second
The edges in max spanning tree is:
[938, 942, 939, 937, 936, 932, 929, 928, 927, 926, 921, 917, 918, 914, 911, 912, 909, 905, 904, 903, 899, 895, 894, 890, 885, 884, 883, 882, 880, 875,
 874, 869, 870, 866, 861, 857, 856, 851, 850, 847, 846, 843, 841, 840, 836, 831, 827, 823, 828, 825, 821, 817, 816, 815, 811, 812, 807, 805, 804, 802,
 797, 796, 794, 789, 790, 791, 792, 788, 783, 779, 778, 780, 775, 774, 772, 767, 768, 766, 761, 762, 760, 755, 754, 749, 748, 746, 741, 737, 736, 733,
 732, 729, 725, 721, 722, 723, 719, 715, 716, 711, 709, 710, 705, 701, 700, 702, 698, 695, 691, 692, 687, 683, 682, 679, 681, 677, 673, 672, 669, 668,
 667, 670, 665, 661, 662, 657, 653, 654, 650, 652, 651, 647, 646, 644, 639, 635, 631, 636, 632, 634, 629, 628, 625, 624, 621, 620, 622, 619, 618, 614,
 609, 608, 603, 601, 600, 595, 594, 590, 585, 581, 582, 577, 578, 573, 569, 572, 567, 563, 564, 561, 560, 555, 551, 550, 552, 548, 543, 542, 539, 544,
 541, 540, 535, 536, 531, 530, 525, 523, 522, 518, 520, 521, 517, 516, 511, 510, 505, 507, 506, 501, 497, 493, 496, 491, 487, 488, 483, 486, 481, 480,
 477, 476, 473, 469, 474, 472, 470, 465, 463, 462, 461, 457, 458, 459, 455, 451, 450, 448, 443, 439, 444, 441, 437, 433, 436, 431, 427, 426, 422, 417,
 420, 415, 414, 410, 405, 403, 402, 399, 398, 393, 389, 390, 388, 386, 381, 380, 375, 373, 374, 369, 370, 368, 365, 366, 363, 361, 360, 356, 358, 353,
 354, 350, 349, 348, 343, 346, 341, 340, 335, 331, 330, 329, 328, 323, 319, 318, 314, 313, 312, 310, 307, 306, 301, 303, 299, 295, 300, 296, 293, 289,
 292, 290, 285, 281, 284, 286, 287, 283, 282, 277, 280, 275, 274, 272, 273, 269, 270, 268, 263, 259, 264, 262, 260, 255, 256, 258, 253, 252, 249, 248,
 245, 241, 240, 237, 233, 232, 230, 229, 228, 224, 221, 217, 216, 213, 212, 209, 210, 208, 207, 203, 199, 198, 196, 191, 187, 192, 188, 183, 181, 182,
 177, 173, 174, 171, 170, 165, 168, 166, 164, 159, 155, 151, 150, 146, 141, 142, 137, 138, 135, 131, 127, 126, 124, 122, 117, 113, 109, 111, 112, 107,
 103, 104, 99, 95, 94, 89, 90, 87, 83, 79, 81, 77, 80, 75, 76, 78, 74, 69, 70, 65, 64, 61, 60, 56, 55, 54, 49, 50, 45, 48, 43, 42, 41, 37, 38, 35, 34,
 33, 32, 27, 23, 22, 17, 18, 16, 15, 14, 11, 7, 8, 2, 3, 4997, 4998, 4996, 5000, 3049]
The maximum bandwidth is:
319
```

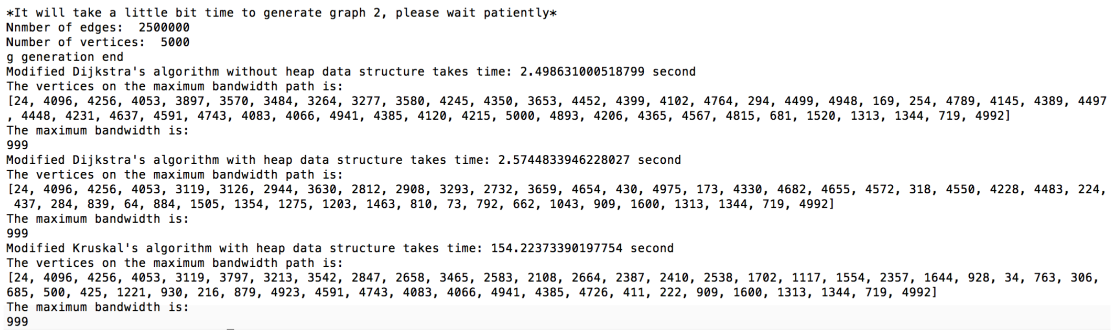
The fifth time to running the entire program, the result is showing below (program output):



```
Nnmber of edges:  15000
Number of vertices:  5000
graph generation end
Modified Dijkstra's algorithm without heap data structure takes time: 0.007600069046020508 second
The vertices on the maximum bandwidth path is:
[119, 120, 117, 113, 118, 115, 114, 112, 107, 103, 104, 99, 100, 101, 97, 96, 91, 94, 89, 85, 84, 81, 82, 77, 73, 72, 70, 67, 66, 63, 59, 58, 53, 52,
47, 48, 46, 41, 37, 36, 32, 27, 23, 19, 18, 14, 10, 6, 1, 2, 4999, 5000, 4072]
The maximum bandwidth is:
325
Modified Dijkstra's algorithm with heap data structure takes time: 0.03482389450073242 second
The vertices on the maximum bandwidth path is:
[119, 120, 117, 113, 118, 115, 114, 112, 107, 103, 104, 99, 98, 93, 96, 91, 94, 89, 85, 84, 79, 82, 77, 73, 72, 69, 65, 66, 63, 59, 58, 53, 52, 47, 48
, 46, 41, 37, 36, 32, 27, 23, 19, 18, 14, 10, 6, 5, 2, 4999, 5000, 4072]
The maximum bandwidth is:
325
Modified Kruskal's algorithm with heap data structure takes time: 1.6045608520507812 second
The edges in max spanning tree is:
[119, 120, 117, 113, 118, 115, 114, 112, 107, 103, 104, 99, 98, 95, 96, 91, 94, 92, 87, 88, 85, 84, 81, 79, 82, 77, 73, 72, 70, 67, 66, 63, 59, 58, 53
, 52, 47, 50, 45, 41, 40, 42, 38, 33, 34, 29, 32, 27, 23, 24, 21, 17, 16, 11, 12, 7, 6, 1, 2, 3, 4997, 5000, 4072]
The maximum bandwidth is:
325
```

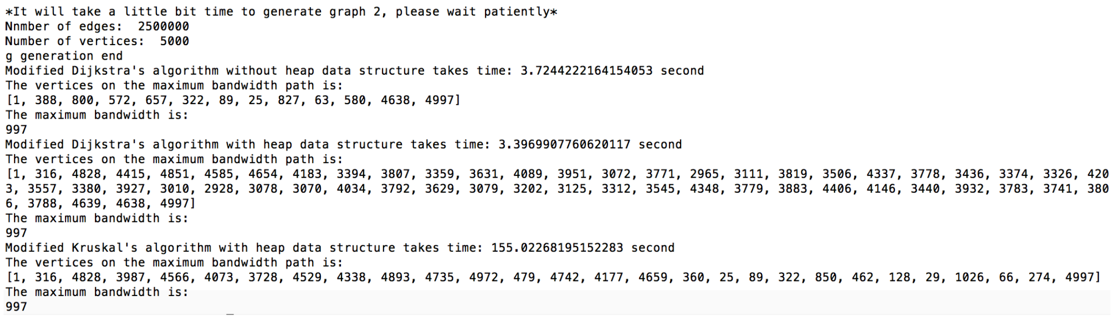
2. Test for graph 2:

The first time to running the entire program, the result is showing below (program output):



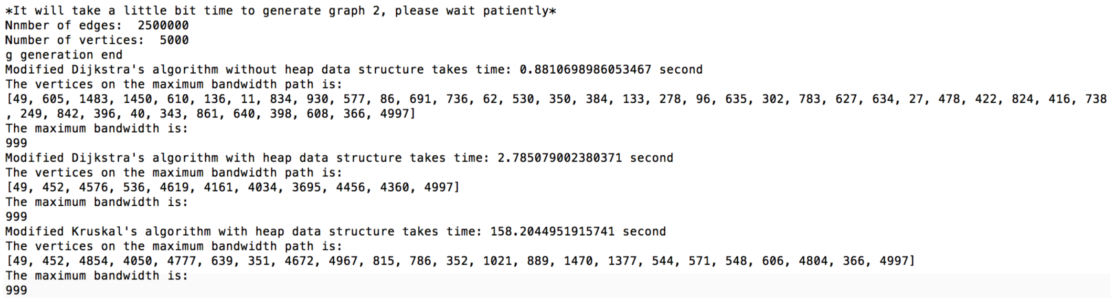
```
*It will take a little bit time to generate graph 2, please wait patiently*
Nnmber of edges:  2500000
Number of vertices:  5000
g generation end
Modified Dijkstra's algorithm without heap data structure takes time: 2.498631000518799 second
The vertices on the maximum bandwidth path is:
[24, 4096, 4256, 4053, 3897, 3570, 3484, 3264, 3277, 3580, 4245, 4350, 3653, 4452, 4399, 4102, 4764, 294, 4499, 4948, 169, 254, 4789, 4145, 4389, 4497
, 4448, 4231, 4637, 4591, 4743, 4083, 4066, 4941, 4385, 4120, 4215, 5000, 4893, 4206, 4365, 4567, 4815, 681, 1520, 1313, 1344, 719, 4992]
The maximum bandwidth is:
999
Modified Dijkstra's algorithm with heap data structure takes time: 2.5744833946228027 second
The vertices on the maximum bandwidth path is:
[24, 4096, 4256, 4053, 3119, 3126, 2944, 3630, 2812, 2908, 3293, 2732, 3659, 4654, 430, 4975, 173, 4330, 4682, 4655, 4572, 318, 4550, 4228, 4483, 224,
 437, 284, 839, 64, 884, 1505, 1354, 1275, 1203, 1463, 810, 73, 792, 662, 1043, 909, 1600, 1313, 1344, 719, 4992]
The maximum bandwidth is:
999
Modified Kruskal's algorithm with heap data structure takes time: 154.22373390197754 second
The vertices on the maximum bandwidth path is:
[24, 4096, 4256, 4053, 3119, 3797, 3213, 3542, 2847, 2658, 3465, 2583, 2108, 2664, 2387, 2410, 2538, 1702, 1117, 1554, 2357, 1644, 928, 34, 763, 306,
685, 500, 425, 1221, 930, 216, 879, 4923, 4591, 4743, 4083, 4066, 4941, 4385, 4726, 411, 222, 909, 1600, 1313, 1344, 719, 4992]
The maximum bandwidth is:
999
```

The second time to running the entire program, the result is showing below (program output):



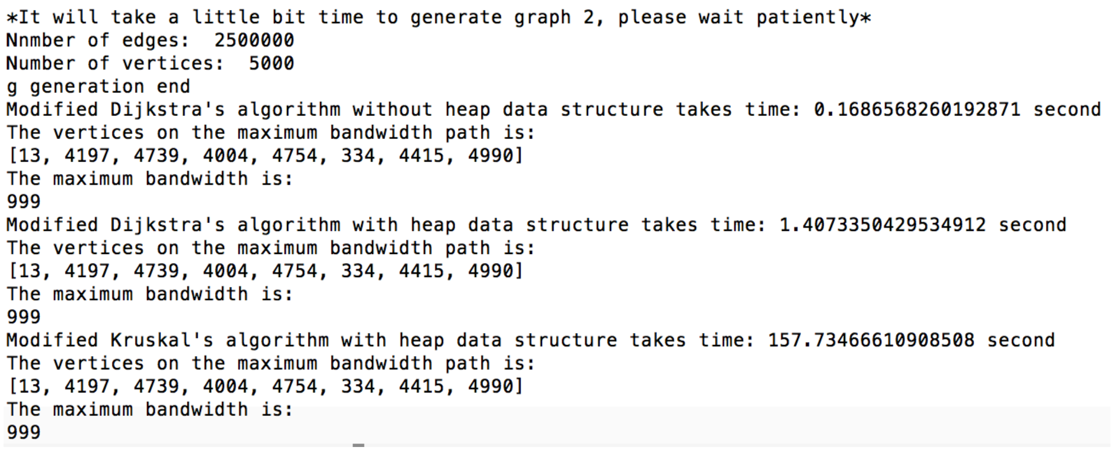
```
*It will take a little bit time to generate graph 2, please wait patiently*
Nnmber of edges:  2500000
Number of vertices:  5000
g generation end
Modified Dijkstra's algorithm without heap data structure takes time: 3.7244222164154053 second
The vertices on the maximum bandwidth path is:
[1, 388, 800, 572, 657, 322, 89, 25, 827, 63, 580, 4638, 4997]
The maximum bandwidth is:
997
Modified Dijkstra's algorithm with heap data structure takes time: 3.3969907760620117 second
The vertices on the maximum bandwidth path is:
[1, 316, 4828, 4415, 4851, 4585, 4654, 4183, 3394, 3807, 3359, 3631, 4089, 3951, 3072, 3771, 2965, 3111, 3819, 3506, 4337, 3778, 3436, 3374, 3326, 420
3, 3557, 3380, 3927, 3010, 2928, 3078, 3070, 4034, 3792, 3629, 3079, 3202, 3125, 3312, 3545, 4348, 3779, 3883, 4406, 4146, 3440, 3932, 3783, 3741, 380
6, 3788, 4639, 4638, 4997]
The maximum bandwidth is:
997
Modified Kruskal's algorithm with heap data structure takes time: 155.02268195152283 second
The vertices on the maximum bandwidth path is:
[1, 316, 4828, 3987, 4566, 4073, 3728, 4529, 4338, 4893, 4735, 4972, 479, 4742, 4177, 4659, 360, 25, 89, 322, 850, 462, 128, 29, 1026, 66, 274, 4997]
The maximum bandwidth is:
997
```

The third time to running the entire program, the result is showing below (program output):



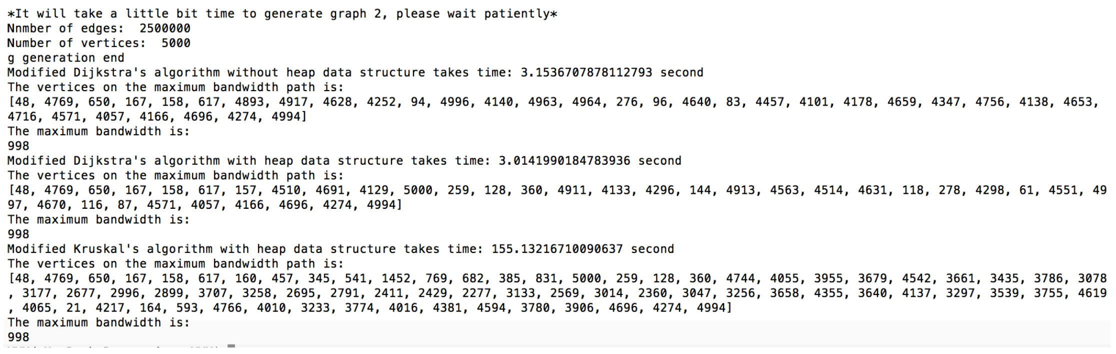
```
*It will take a little bit time to generate graph 2, please wait patiently*
Nnmber of edges:  2500000
Number of vertices:  5000
g generation end
Modified Dijkstra's algorithm without heap data structure takes time: 0.8810698986053467 second
The vertices on the maximum bandwidth path is:
[49, 605, 1483, 1450, 610, 136, 11, 834, 930, 577, 86, 691, 736, 62, 530, 350, 384, 133, 278, 96, 635, 302, 783, 627, 634, 27, 478, 422, 824, 416, 738
, 249, 842, 396, 40, 343, 861, 640, 398, 608, 366, 4997]
The maximum bandwidth is:
999
Modified Dijkstra's algorithm with heap data structure takes time: 2.785079002380371 second
The vertices on the maximum bandwidth path is:
[49, 452, 4576, 536, 4619, 4161, 4034, 3695, 4456, 4360, 4997]
The maximum bandwidth is:
999
Modified Kruskal's algorithm with heap data structure takes time: 158.2044951915741 second
The vertices on the maximum bandwidth path is:
[49, 452, 4854, 4050, 4777, 639, 351, 4672, 4967, 815, 786, 352, 1021, 889, 1470, 1377, 544, 571, 548, 606, 4804, 366, 4997]
The maximum bandwidth is:
999
```

The fourth time to running the entire program, the result is showing below (program output):



```
*It will take a little bit time to generate graph 2, please wait patiently*
Nnmber of edges:  2500000
Number of vertices:  5000
g generation end
Modified Dijkstra's algorithm without heap data structure takes time: 0.1686568260192871 second
The vertices on the maximum bandwidth path is:
[13, 4197, 4739, 4004, 4754, 334, 4415, 4990]
The maximum bandwidth is:
999
Modified Dijkstra's algorithm with heap data structure takes time: 1.4073350429534912 second
The vertices on the maximum bandwidth path is:
[13, 4197, 4739, 4004, 4754, 334, 4415, 4990]
The maximum bandwidth is:
999
Modified Kruskal's algorithm with heap data structure takes time: 157.73466610908508 second
The vertices on the maximum bandwidth path is:
[13, 4197, 4739, 4004, 4754, 334, 4415, 4990]
The maximum bandwidth is:
999
```

The fifth time to running the entire program, the result is showing below (program output):



```
*It will take a little bit time to generate graph 2, please wait patiently*
Nnmber of edges:  2500000
Number of vertices:  5000
g generation end
Modified Dijkstra's algorithm without heap data structure takes time: 3.1536707878112793 second
The vertices on the maximum bandwidth path is:
[48, 4769, 650, 167, 158, 617, 4893, 4917, 4628, 4252, 94, 4996, 4140, 4963, 4964, 276, 96, 4640, 83, 4457, 4101, 4178, 4659, 4347, 4756, 4138, 4653,
4716, 4571, 4057, 4166, 4696, 4274, 4994]
The maximum bandwidth is:
998
Modified Dijkstra's algorithm with heap data structure takes time: 3.0141990184783936 second
The vertices on the maximum bandwidth path is:
[48, 4769, 650, 167, 158, 617, 157, 4510, 4691, 4129, 5000, 259, 128, 360, 4911, 4133, 4296, 144, 4913, 4563, 4514, 4631, 118, 278, 4298, 61, 4551, 49
97, 4670, 116, 87, 4571, 4057, 4166, 4696, 4274, 4994]
The maximum bandwidth is:
998
Modified Kruskal's algorithm with heap data structure takes time: 155.13216710090637 second
The vertices on the maximum bandwidth path is:
[48, 4769, 650, 167, 158, 617, 160, 457, 345, 541, 1452, 769, 682, 385, 831, 5000, 259, 128, 360, 4744, 4055, 3955, 3679, 4542, 3661, 3435, 3786, 3078
, 3177, 2677, 2996, 2899, 3707, 3258, 2695, 2791, 2411, 2429, 2277, 3133, 2569, 3014, 2360, 3047, 3256, 3658, 4355, 3640, 4137, 3297, 3539, 3755, 4619
, 4065, 21, 4217, 164, 593, 4766, 4010, 3233, 3774, 4016, 4381, 4594, 3780, 3906, 4696, 4274, 4994]
The maximum bandwidth is:
998
```

Analysis and discussion

1. Modified Dijkstra’s algorithm with or without using heap

One important thing you may notice is that the total execution time of the algorithm without using heap is less than algorithm using heap. This might be different from the theoretical analysis. The reasons are following:

a. For the algorithm without heap, I utilized dictionary structure rather than simple array to organize our fringes. That means the insertion and deletion only take O(1). The inner loop of Dijkstra’s algorithm will execute at most n+m times, where n is the number of vertices and m is the number of edges. Thus the total running time of the inner loop block is O(n+m). The critical module is the function to find max value to Although using the simple way (by comparison) to search the max value in a dictionary takes linear time which is proportional to the current number of fringes, this number is usually quite smaller than n (the total number of vertices). Therefore, although the total running time is bounded by c*n*n, the constant c will be quite small. However, for the algorithm with heap structure, I utilized the heap-based priority queue ADT to organize the fringes. The advantage is that we can retrieve max value in O(1). However, to insert or delete or update this heap-based queue will take O(logn). The inner loop of Dijkstra's algorithm will execute at most n+m times, where n is the number of vertices and m is the number of edges. The total running time for the inner loop block is (n+m)logn.

b. The number of vertices is much less than edges. The number of edges can be up to n*n. In this situation, O((n+m)logn) can be much worse than O(n*n). For our case, the number of edges is much larger than the number of vertices. Therefore, m dominates the running time.

In theoretical, the modified Dijkstra's algorithm takes quadratic time. However, in practice, if we use dictionary (map) data structure to store the fringes, it will achieve a relatively good performance.

2. Modified Kruskal's algorithm

The running time of Kruskal's algorithm for a graph depends on how we implement the disjoint-set data structure. I utilized the union-by-rank and path-compression heuristics. To put edges into the heap-based queue will take O(mlogm). The while loop performs FIND and UNION operations taking O((n+m)logn). Because m<=n*n, O(logm) = O(logn). The total running time is O(mlogn).

We can see that the execution time of this modified kruskal's algorithm on graph 1 is much less than on graph 2. That is because the running time of this algorithm heavily depends on the number of edges, as we discussed before, O(mlogm).

Actually, to solve the max-bandwidth problem, Kruskal's algorithm is not enough. It only returns us a max-spanning tree. The remaining problem is how to find the path from source to target. Luckily, there are two very simple algorithms to be able to get this job done, depth-first search (DFS) and breadth-first search (BFS). I utilized BFS in my implementation. This subroutine will take linear time that is proportional to the number of vertices on the max-spanning tree.

Conclusion and future works

After implementation and analysis, we can see that different algorithms differ a lot in performance to solve the practical problem. Dijkstra's algorithm and Kruskal's algorithm, although they has almost the same time complexity in theoretical, O((n+m)logn) and O(mlogn), they have a large difference in running time when faced with different situations. For instance, we prefer to use modified Dijkstra's algorithm when the number of edges is much larger than the number of vertices. However, when dealing with a graph has very few edges, we might better use modified Kruskal’s algorithm.

Not only the algorithms matter a lot, the data structures also matter a lot in performance. For example, for dictionary, it only takes constant time o(1) to insert or delete an element. However, for array, it needs to take linear time to insert or delete an element. It helps us a lot to efficiently store and manipulate the fringes for modified Dijkstra’s algorithm 1. Heap structure helps us a lot in retrieving the max item of a collection, by taking O(1) time.

There are still a lot of alternative ways to solve the max bandwidth path. Also, there are a lot of data structures we can use to improve our algorithm. Like we can organize fringes as a 2-3 tree. Also, for modified Kruskal’s algorithm, we can store edge in a python list and use merge-sort or quick-sort instead of putting it in a heap.