2017

CSCE-629-Course Project

MAX-BANDWIDTH-PATH PROBLEM

Yang Li

Uin:126004157

CSCE 629 Analysis of Algorithm

Course Project

Yang Li

UIN:126004157

1 IMPLEMENTATION

1.1 IMPLEMENTATION OF GRAPH

1.1.1 GRAPH 1, IN WHICH EVERY VERTEX HAS DEGREE EXACTLY 6.

Analysis of this graph:

In order to build this graph, I first need to figure how could I build a graph using programming language. So, firstly I found I need to figure out which data structure should I use to build a graph.

Generally, we have several ways to represent a graph. One of those is by using adjacent matrix and another one is by using adjacent list.

Adjacency Matrix:
Adjacency Matrix is a 2D array of size V x V where V is the number of vertices in a graph. Let the 2D array be adj[][], a slot adj[i][j] = 1 indicates that there is an edge from vertex i to vertex j. Adjacency matrix for a undirected graph is always symmetric. Adjacency Matrix is also used to represent weighted graphs. If adj[i][j] = w, then there is an edge from vertex i to vertex j with weight w.

Pros: Representation is easier to implement and follow. Removing an edge takes O(1) time. Queries like whether there is an edge from vertex 'u' to vertex 'v' are efficient and can be done O(1).

Cons: Consumes more space O(V^2). Even if the graph is sparse(contains less number of edges), it consumes the same space. Adding a vertex is O(V^2) time

Adjacency List: An array of linked lists is used. The size of the array is equal to the number of vertices. Let the array be array[]. An entry array[i] represents the linked list of vertices adjacent to the ***i***th vertex. This representation can also be used to represent a weighted graph. The weights of edges can be stored in nodes of linked lists. Following is adjacency list representation.

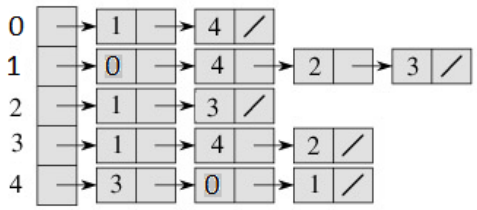

fig 1. adjacency list representation

For graph1, every vertex has degree exactly 6. That means each vertex connect with other 6 vertices. When compared with total vertices number(5000), Graph 1 is sparse(contains less number of edges). Actually, we could use any data structure to represent a graph. However, The choice of the graph representation is situation specific. It totally depends on the type of Graph. It will be not so appropriate to use an adjacent matrix to represent Graph1 Since Graph1 is sparse and use adjacent matrix may waste a lot of space. Therefore I choose to use an adjacent list to represent Graph1. That leads to follow analysis.

Implementation of an adjacent list:

From above we know that adjacent list is kind of key-value pair. Key is vertex and key's value is all vertices that connect with it. According to that, we could implement adjacent list by an adj[], in which an entry adj[i] represents the linked list of vertices adjacent to the *i*th vertex. However, I think there is a better way to realize *adjacent list* that is I could use Map. In JAVA, the language I use to finish this project, a Map is a structure that store Key and its value, this property match our need perfectly. For this reason, I finally choose to use HashMap to represent Graph 1.

After choosing data structure, I need to figure out how to represent a weighted graph. Unlike unweighted graph, which we can use Integer as an element for ArrayList, we need elements in ArrayList contain two kinds of information at the same time. One is vertex's name, the other is the weight that between this vertex and the vertex it connects with. Therefore, I defined a Class named Edge to store that information. Then, I could represent Graph 1 by using

Map<Integer(vertex name),ArrayList<Edge>(vertices that connected with it and weight between them)>.

Generation of Graph:

After addressing how to represent a graph I need to figure out how to build a graph based on method given above. Since we need to generate a random graph that has 5000 vertices and every vertex has exactly degree 6 then a problem that some vertices may can't connect to any other vertices for the reason that all other vertices already have degree 6 may occur. In order to address this problem, I use shuffle.

First, I build a graph by connecting each vertex with its previous three vertices and next three vertices(that makes this graph has exactly degree of 6). After this action, a graph that each vertex has exactly degree 6 is built and assign weight randomly.

Then based on this graph, I made vertices unordered by applying shuffle rule to each vertex. After that, I still face a problem that since Graph1 is undirected, then weight from a to b need to be same as b to a. Nevertheless, the graph that I have built yet do not have that property. Hence I need to change the weight. To solve this, I used a loop.

After all above actions are taken, Graph1 will be built correctly.

1.1.2 GRAPH 2, IN WHICH EACH VERTEX HAS EDGES GOING TO ABOUT 20% OF THE OTHER VERTICES

Since this graph is an undirected, weighted graph and each vertex has edges going to about 20% of the other vertices and this graph has 5000 vertices. So that means each vertex in this graph has about 1000 edges. That property make this graph dense. Hence, I plan to use adjacent matrix to build this graph. After those actions the graph has already been correctly build.

Implementation of an adjacent matrix:

As discussed above we could use 2-D matrix to present adjacent matrix. If adj[i][j] = w, then there is an edge from vertex I to vertex j with weight w.

Generation of Graph:

There are several problems need to be pay attention to when building the graph. The first thing is I need to grantee adj[i][j] = adj[j][i]. That could be achieved by give value to adj[i][j] and adj[j][i] at same time. Next problem is that I need to generate weight randomly and give weight to about the other 1000 vertices. I accomplished this problem by writhing a function that could generate -1 with probability 80% and generate random weight with probability 20%.

```
public int randomWeight(){
    int w = 0;
    Map<String, Integer> keyChanceMap = new HashMap<String, Integer>();
    keyChanceMap.put("with_weight", 20);
    keyChanceMap.put("without_weight", 80);
    String key = RandomChance.chanceSelect(keyChanceMap);
    if (key=="with_weight"){
       w = new Random().nextInt(200);
    }
    if(key=="without_weight"){
       w = -1;
   }
    return w;
    }
```

1.2 IMPLEMENTATION OF HEAP

Implementation of Insert:

When I need to use insert action I need to insert element to appropriate place in heap. I always insert this element in the last position and after that use function pushup to move this element to the right place in heap.

Implementation of pushUp:

In this function, if this is the first element then I directly add it to heap. When it is not the first element, I compare it with its parent, if it is greater than its parents, then switch their elements.

Do it recursively until it is in the right position.

Implementation of delete:

When implement delete I need to remove this element in heap and make other elements in right position. Therefore, firstly I switch the element I want to delete and the last element in heap, then delete the last position's element and after that use function pushdown to make all elements in right position then decrease the heap size by 1.

Implementation of pushDown:

In this function, I compare the element in position i with its child. If this element is greater than both of its child then find the smaller one of its child then switch this element and the smaller child.

Implementation of Max:

Return the first element of the heap.

Implementation of heapSort:

Create a ArrayList to store sorted element. Adding the first element of heap(max) then use delete function to delete it. Keep doing this until heap is empty.

1.3 IMPLEMENTATION OF ROUTING ALGORITHM

For grapgh 1:
Since graph 1 is built by HashMap, I use iterator to iterate it.
For gragph 2:
Since graph 2 is built by 2-D matrix, I use double for-loop to traverse it.

1.3.1 MAX-BANDWIDTH-PATH BASED ON DIJKSTRA'S ALGORITHM WITHOUT USING A HEAP STRUCTURE

I need to find a data structure to store fringes vertices since I need to find max bandwidth path without using heap structure. I choose to use ArrayList to store all fringes.
Unlike using heap, to find the vertex with largest weight i need use for-loop traverse all edges in ArrayList and find one with largest Bandwidth. Then find all vertices linked to this vertices make those vertices fringe, if those vertices are already fringe then change those fringe vertices' bandwidth to the smallest between its father's bandwidth and the weight it linked to its father.

1.3.2 MAX-BANDWIDTH-PATH BASED ON DIJKSTRA'S ALGORITHM USING A HEAP STRUCTURE

Since we could use heap, then we can use heap sort to get the vertex with largest weight. After that we just do Dijkstra.

1.3.3 MAX-BANDWIDTH-PATH BASED KRUSKAL'S ALGORITHM

Overall Describe:
First I need to sort all edges by their weights and then pick from ordered edges add them to "tree" which I use arraylist to represent. The tree is a maximum spanning tree. Then in order to find max bandwidth path we could find a path in the maximum spanning tree that go from start vertex to end vertex. That will be achieved by doing BFS on maximum spanning tree.

Generation of maximum spanning tree:

Firstly, sorted all edges in the given graph and add them to a ArrayList to store their edge information. When add edges we need guarantee that we only add an edge once. As for the reason that we use Map< vertex name ,ArrayList<Edeg>> to show graph we have an edge, eg. "a-b", represented both as Map<a , ArrayList<b>> and Map<b , ArrayList<a>>. That means if we just go through Map and add all to ArrayList we will add each edge twice, one is from "a-b", the other one is from "b-d". So, I solved this problem by just add edges whose father's name is less than its name. After that, edges are sorted and add to ArrayList.

Then I build maximum spanning tree on the base of ArrayList's edges. When we add edges to tree, we need to guarantee that the two vertices of this edge in different pieces, which means their root need to be different. That is because we don't want form a circle in the maximum spanning tree. We use Find sub function to find the root of vertices' root.
There is still another problem need to pay attention to. When I build ArrayList, I just add one direction of edges to it. However, since we have an undirected graph, we actually need an edge of two direction, eg "a-b" need to be presented as "a->b" and "b-a". So, when we build a maximum spanning tree which built based on a ArrayList which contains all information of Edges. When add Edges to ArrayList, both two path need to be added to it.

Do BFS to find shortest path:

Based on maximum spanning tree I do BFS to find all vertex linked to start vertex then build an array that store each vertex's father. When I need to find a path I can just go back from the dad array. Then we can find a path.

2 TESTING

Time counting: use System time to calculate code running time.

2.1 GRAPH 1 TEST: (WILL JUST SHOW ONE RESULT FOR ONE PAIR OF START-END VERTICES. ALL OTHER TESTS WILL JUST GIVE THE RUNNING TIME) MAX WEIGHT FOR GRAPH(1) IS 100.

1) from vertex 233 to 1685:

max bandwidth with heap for graph1

```
maximum bandwidth is 36

from 233 to 1685 we need go through follow vertices
3290 4255 4291 2488 4740 1958 2387 3483 220 1437 3676 4941 4658 2300 1103 4852 4319 3182 252 3566
2493 952 110 755 317 2003 3684 2481 65 1558 1104 293 191 3855 3919 1468 2210 3257 4943 2819 3129
3254 4312 2921 2373 2272 39 2671 3060 4379 1662 3677
run time : 32 ms
```

max bandwidth without heap for graph1

```
maximum bandwidth is 36

from 233 to 1685 we need go through follow vertices
3290 4255 4291 2488 4740 1958 2387 3483 220 1437 3676 4941 4658 2300 1228 1730 2235 4893 3182 252
3566 2493 1065 2802 2315 2657 4184 1584 3684 2481 65 1558 1104 293 191 3855 3919 1715 1986 3257
4943 2819 3129 3254 4312 2921 2373 282 3471 833 1149 2135 559 1662 3677
run time: 39 ms
```

max bandeidth kruskal :

```
from 233 to 1685 we need go through follow vertices
3290 4255 4291 2488 4740 1958 2387 3128 3483 220 4640 2245 1437 3676 4941 4658 2300 1515 2931 4518
2235 644 4319 3182 252 1165 3566 2493 1065 952 110 2802 2315 755 2657 317 4184 1584 3684 2481 2754
1698 1558 1104 293 191 1869 4691 1706 3288 1986 3257 367 4943 2819 1921 3992 3254 4231 3025 4312
3822 642 2921 2373 748 2272 282 3471 833 1149 2135 4379 1662 2603 1890 1616 4970
run time:  1072 ms
```

Test graph1 #1			
Start and end	max bandwidth with heap & run time	max bandwidth without heap & run time	max bandwidth kruskal run time
Form 5 to 2888	max bandwidth :22 run time : 32ms	max bandwidth :22 run time : 28ms	Run time: 968 ms
Form 50 to 168	max bandwidth :29 run time : 18ms	max bandwidth :29 run time : 21ms	Run time: 962 ms
Form 10 to 166	max bandwidth :24 run time : 25ms	max bandwidth :24 run time : 39ms	Run time: 918 ms
Form 66 to 2666	max bandwidth :29 run time : 28ms	max bandwidth :29 run time : 27ms	Run time: 935 ms

Test graph1 #2

Start and end	max bandwidth with heap & run time	max bandwidth without heap & run time	max bandwidth kruskal run time
Form 336 to 777	max bandwidth :30 run time : 26ms	max bandwidth :30 run time : 28ms	Run time: 1081 ms
Form 150 to 268	max bandwidth :26 run time : 20ms	max bandwidth :26 run time : 29ms	Run time: 988 ms
Form 100 to 1666	max bandwidth :28 run time : 22ms	max bandwidth :28 run time : 30ms	Run time: 969 ms
Form 166 to 3666	max bandwidth :30 run time :29ms	max bandwidth :30 run time : 41ms	Run time: 1064 ms
Form 366 to 2786	max bandwidth :26 run time : 23ms	max bandwidth :26 run time : 27ms	Run time: 985 ms

Test graph1 #3

Start and end	max bandwidth with heap & run time	max bandwidth without heap & run time	max bandwidth kruskal run time
Form 35 to 2688	max bandwidth :28 run time : 29ms	max bandwidth :28 run time : 32ms	Run time: 1068 ms
Form 560 to 2168	max bandwidth :23 run time : 22ms	max bandwidth :23 run time : 25ms	Run time: 992 ms
Form 170 to 1466	max bandwidth :24 run time : 29ms	max bandwidth :24 run time : 37ms	Run time: 1018 ms
Form 636 to 2566	max bandwidth :30 run time : 23ms	max bandwidth :30 run time : 29ms	Run time: 937 ms
Form 6886 to 3666	max bandwidth :23 run time : 21ms	max bandwidth :23 run time : 27ms	Run time: 1023 ms

Test graph1 #4

Start and end	max bandwidth with heap & run time	max bandwidth without heap & run time	max bandwidth kruskal run time
Form 52 to 2778	max bandwidth :24 run time : 25ms	max bandwidth :24 run time : 29ms	Run time: 932 ms
Form 550 to 1698	max bandwidth :29 run time : 28ms	max bandwidth :29 run time : 31ms	Run time: 1032 ms
Form 57 to 573	max bandwidth :25 run time : 27ms	max bandwidth :25 run time : 36ms	Run time: 938 ms
Form 566 to 3266	max bandwidth :27 run time : 28ms	max bandwidth :27 run time : 33ms	Run time: 978 ms
Form 578 to 3466	max bandwidth :24 run time : 27ms	max bandwidth 24 run time : 30ms	Run time: 997 ms

Test graph1 #5			
Start and end	max bandwidth with heap & run time	max bandwidth without heap & run time	max bandwidth kruskal run time
Form 64 to 2168	max bandwidth :25 run time : 29ms	max bandwidth :25 run time : 29ms	Run time: 995 ms
Form 30 to 168	max bandwidth :24 run time : 18ms	max bandwidth :24 run time : 21ms	Run time: 934 ms
Form 1560 to 1626	max bandwidth :28 run time : 25ms	max bandwidth :28 run time : 39ms	Run time: 1034 ms
Form 656 to 1236	max bandwidth :26 run time : 30ms	max bandwidth :26 run time : 30	Run time: 987 ms
Form 24 to 2632	max bandwidth :27 run time : 25ms	max bandwidth :27 run time : 27ms	Run time: 634 ms

2.2 **GRAPH 2 TEST:** (SINCE MAX-BANDWIDTH-PATH IS NOT SUCH LONG, I WILL PRESENT PATH) MAX WEIGHT FOR GRAPH(2) IS 100000.

1) from vertex 5 to 4998:

max bandwidth with heap for graph2:

```
maximum bandwidth is 99886
from 5 to 4998 we need go through follow vertices
1774 4858 3101 2193 2460 1230 4435 2268 2800 2643 1651 4382 4395 4232 4303 3832 1076 196
run time: 4511 ms
```

max bandwidth without heap for graph2:

```
maximum bandwidth is 99886
from 5 to 4998 we need go through follow vertices
3909 4156 4609 1232 942
run time: 312 ms
```

max bandeidth kruskal :

```
from 5 to 4998 we need go through follow vertices
3456 4449 3035 3901 3208 3463 3009 3040 3093 1235 564 4003 1236 4983 1273 1578 1285 3007 3189 3310
3483 67 3306 1225 4126 283 3841 3325 135 2458 576 3060 1323 3806 1278 4947 3552 4991 589 77 4697
1385 328
run time: 4806 ms
```

2) from vertex 8 to 676:

max bandwidth with heap for graph2:

maximum bandwidth is 99764
from 8 to 676 we need go through follow vertices

1002 1457 3283 3444 3976 3759

run time: 4500 ms

max bandwidth without heap for graph2:

maximum bandwidth is 99764
from 8 to 676 we need go through follow vertices

2153 2728 4422 154

run time: 366 ms

max bandwidth kruskal :

from 8 to 676 we need go through follow vertices
707 330 2847 1401 1890 644 3675 287 4303 3002 3216 3094 293 2538 1246 4164 1232 4689 573 3821 3160
3282 3227 4012 3093 4861 1227 1708 1216 3610 3163 283 2044 1381 1847 154
run time: 4709 ms

3) from vertex 8 to 676:

max bandwidth with heap for graph2:

maximum bandwidth is 99896
from 8 to 676 we need go through follow vertices
1766 262 75 3087 822 1452 381 1479 3267 1165 950 648

run time: 5040ms

max bandwidth without heap for graph2:

maximum bandwidth is 99896

from 16 to 686 we need go through follow vertices
495 1265 2610 648

run time: 387ms

max bandwidth kruskal :

from 16 to 686 we need go through follow vertices

2609 1624 1338 3369 4693 3105 3338 3271 3211 3805 3248 3855 3062 3957 1257 271 4784 1241 1446 586
2632 274 3312 3304 3578 3264 3321 4042 1255 4743 3046 3098 3133 4552
run time: 4498 ms

4) from vertex 68 to 1676:

max bandwidth with heap for graph2:

maximum bandwidth is 99788
from 68 to 1676 we need go through follow vertices
607 3320 4109 1799 2362 2331 2694 4591 2189 558 1913 970 2970 4833 3872
4948 ms

max bandwidth without heap for graph2:

maximum bandwidth is 99788

from 68 to 1676 we need go through follow vertices
3309 421 3271 552
337 ms

max bandwidth kruskal :

from 68 to 1676 we need go through follow vertices
3309 3915 3289 4059 3031 4584 3188 1241 2486 574 575 2658 1214 2485 1245 1196 556 4069 3006 3080
3172 3115 4976 275 594 2514 288 1614 1222 3370 1275 1440
4825 ms

5) from vertex 168 to 2676:

max bandwidth with heap for graph2:

maximum bandwidth is 99886
from 168 to 2676 we need go through follow vertices
2505 1477 454 1521 707 4949 4959
309 ms

max bandwidth without heap for graph2:

maximum bandwidth is 99886
from 168 to 2676 we need go through follow vertices
464 3907 3922 2065 1351
446 ms

max bandwidth kruskal :

from 168 to 2676 we need go through follow vertices
81 3338 4153 1215 3097 4204 590 134 4034 3279 4566 1249 4635 1258 1307 1252 3014 4100 3485 4496
1351
4700 ms

Test graph2 #2

Start and end	max bandwidth with heap & run time	max bandwidth without heap & run time	max bandwidth kruskal run time
Form 232 to 3566	max bandwidth:99776 run time : 253ms	max bandwidth :99776 run time : 348ms	Run time: 6178 ms
Form 336 to 777	max bandwidth :99819 run time : 261ms	max bandwidth :99819 run time : 426ms	Run time: 6489 ms
Form 326 to 783	max bandwidth:99873 run time : 440ms	max bandwidth :99873 run time : 515ms	Run time: 10613 ms
Form 636 to 2436	max bandwidth :99871 run time : 303ms	max bandwidth :99871 run time : 360ms	Run time: 8766 ms
Form 163 to 4366	max bandwidth :99761 run time : 287ms	max bandwidth :99761 run time : 475ms	Run time: 7009 ms

Test graph2 #3

Start and end	max bandwidth with heap & run time	max bandwidth without heap & run time	max bandwidth kruskal run time
Form 35 to 3248	max bandwidth :99879 run time : 274ms	max bandwidth :99879 run time : 413ms	Run time: 8629 ms
Form 354 to 1688	max bandwidth : 99867 run time : 268ms	max bandwidth :99867 run time : 372ms	Run time: 10000 ms
Form 160 to 1634	max bandwidth :99845 run time : 25ms	max bandwidth :99845 run time : 39ms	Run time: 7157 ms
Form 753 to 2645	max bandwidth :99813 run time : 260ms	max bandwidth :99813 run time : 412ms	Run time: 11267 ms
Form 668 to 966	max bandwidth :99861 run time : 270ms	max bandwidth :99861 run time : 367ms	Run time: 7370 ms

Test graph2 #4

Start and end	max bandwidth with heap & run time	max bandwidth without heap & run time	max bandwidth kruskal run time
Form 234 to 2836	max bandwidth :99677 run time : 306ms	max bandwidth :99677 run time : 345ms	Run time: 10834ms
Form 345 to 2456	max bandwidth :99803 run time : 285ms	max bandwidth :99803 run time : 438ms	Run time: 8388 ms
Form 243 to 435	max bandwidth :99831 run time : 274ms	max bandwidth :99831 run time : 381ms	Run time: 7733ms
Form 324 to 523	max bandwidth :99778 run time : 251ms	max bandwidth :99778 run time : 453ms	Run time: 7453 ms
Form 43 to 2634	max bandwidth :99893 run time : 263ms	max bandwidth :99893 run time : 436ms	Run time: 8191 ms

Test graph2 #5			
Start and end	max bandwidth with heap & run time	max bandwidth without heap & run time	max bandwidth kruskal run time
Form 654 to 3454	max bandwidth :99873 run time : 256ms	max bandwidth :99873 run time : 348ms	Run time: 6786 ms
Form 76 to 4545	max bandwidth :99861 run time : 288ms	max bandwidth :99861 run time : 483ms	Run time: 7261 ms
Form 3253 to 2435	max bandwidth :99834 run time : 414ms	max bandwidth :99834 run time : 454ms	Run time: 10796 ms
Form 345 to 2546	max bandwidth :99863 run time : 263ms	max bandwidth :99863 run time : 378ms	Run time: 7099 ms
Form 43 to 643	max bandwidth :99850 run time : 263ms	max bandwidth :99850 run time : 468ms	Run time: 6947 ms

3 ANALYSIS

Result analysis:

From above we can see that all three methods spend less time to find max-bandwidth-path for graph 1 than doing the same thing to graph 2.

That is because for graph 1, it's a sparse graph so we use adjacent list to represent graph while graph 2 is a dense graph and we use adjacent matrix to represent graph2. Different data structure can cause different cost time.

What is more, if we use heap to store fringes time complexity will be decreased.

Kruskal cost much more time than both kinds of dijstra.

Analysis of algorithms for graph 1:

For Dijkstra:

1) with heap

   First, we need to go through all vertices and make their status unseen, that takes O(n).

   Second, we need to find all vertices that linked to start vertex, which are total 6 vertices. Therefore, time is O(1). Action insert to heap takes O(log N).

   Third, we need to find max bandwidth of all fringes. Since we use heap, this Max operation will takes O(1)

   Since we need do Max until heap is empty, total time take is O(N)

   Fourth, total time to do bandwidth change part is O(mlog N)

   Therefore, total time is O(mlog N)

2) without heap

First, we need to go through all vertices and make their status unseen, that takes O(n).

Second, we need to find all vertices that linked to start vertex, which are total 6 vertices. Therefore, time is O(n). add to arrraylist take O(1).

Third, we need to find max bandwidth of all fringes. Since we use heap, this Max operation will takes O(N)

Since we need do Max until fringe is empty, total time take is O(N^2)

Fourth, total time to do bandwidth change part is O(N). Therefore, total time is O(N^2)

For kruskal:

First, we need to sort edges increasingly, that will takes O(mlog N)

Second, for all sorted edges find their vertices, time is O(m)

Third, we need to find root of vertices. Time is O(log N)

Fourth, union is O(1);

Fifth, making MaxST takes O(NM)

Finally, BFS, takes O(M+N);

Therefore, total time is O(NM)

Analysis of algorithms for graph 2:

For Dijkstra:

1) with heap

First, we need to go through all vertices and make their status unseen, that takes O(n).

Second, we need to find all vertices that linked to start vertex, which are total 1000 vertices. Therefore, time is O(N). Action insert to heap takes O(log N). total time O(Nlog N)

Third, we need to find max bandwidth of all fringes. Since we use heap, this Max operation will takes O(1)

Since we need do Max until heap is empty, total time take is O(N)

Fourth, total time to do bandwidth change part is O(mlog N)

Therefore, total time is O(mlog N)

2) without heap:

First, we need to go through all vertices and make their status unseen, that takes O(N).

Second, we need to find all vertices that linked to start vertex. time is O(N). add to arrraylist take O(1).

Third, we need to find max bandwidth of all fringes. Since we use heap, this Max operation will takes O(N)

Since we need do Max until fringe is empty, total time take is O(N^2)

Fourth, total time to do bandwidth change part is O(N)

Therefore, total time is O(N^2)

For kruskal:

First, we need to sort edges increasingly, that will takes O(mlog N)

Second, for all sorted edges find their vertices, time is O(m)

Third, we need to find root of vertices. Time is O(log N)

Fourth, union is O(1);

Fifth, making MaxST takes O(NM)

Finally, BFS, takes O(N^2);

Therefore, total time is O(N^2)