CSCE–629 FINAL PROJECT

Mehdi Gorjian
626001122

Fall 2021

1 Introduction

In this network optimization project, I implemented a network routing protocol using the data structures and algorithms. The project divided into several phases. First, creating two types of Graph, one with the average degree of 6 and the other with the adjacency of 20% of the total vertices. Second, defining a Max Heap data structure, with Insert, Delete functions. Third, the algorithm should solve a Max–Bandwidth–Path problem for each two given vertices utilizing 3 different methods, one with a modification of Dijkstra's algorithm without using a heap structure, the second method by modifying Dijkstra's algorithm using a heap structure for fringes, and finally a Kruskal's algorithm modified, in which the edges are sorted by Heapsort. In the end, I tested the routing algorithms on 5 pairs of graphs $G1$ and $G2$, randomly generated using the generated subroutines.

2 Random Graph Generation

The Graph was created using the adjacency list and the adjacency matrix. To make sure the generated graph is connected, the Graph starts with a cycle containing all the edges, then adds random edges to it to satisfy the propositions of $G1$ and $G2$. The Graph assigns a random weight within the range of $[1, 20]$ and the whole Graph is completely random in generating. Graph $G1$ was defined with the average vertex degree of 6, which means `num of edges = num of vertices * average vertex degree * 0.5` it means that for the total number of vertex of 5000 around 15000 edges exist. In Graph $G2$, each vertex is adjacent to around 20% of the number of edges. In this case, for each pair of vertices connected with the probability of 20% after generating the Graph vertices and the connection based on the cycle connection strategy.

3 Heap Structure

The Max Heap structure implementation includes subroutines for Insert, and Delete. The structure utilizes Zero indexing in which it holds the vertices structure in the range of $[0, 4999]$. The heap is given by an array $H[5000]$, where each element $H[i]$ gives the name of a vertex in the graph and the vertex values are stored in a different array $D[5000]$. Thus, to find the value of a vertex $H[i]$ in the heap, we can use $D[H[i]]$.

4 Routing Algorithms

The algorithms get two vertices as the `start` and the `destination` to find the Max–Bandwidth–Path using 3 modified algorithms based on Dijkstra and Kruskal.

1. *Dijkstra's algorithm without heap:* I used array for storing the possible bandwidth from s to each other vertex which is initialized to ∞.

2. *Dijkstra's algorithm with heap:* I used heap structure and an array for storing the bandwidth possible so far from s to each other vertex which is initialized to ∞. For path finding, I maintain a parent array.

3. *Kruskal's algorithm:* I used heap structure and to check whether addition of an edge creates a cycle or not, I used Union–Find algorithm.

4.1 Dijkstra without Heap

Algorithm 1 Dijkstra without Heap

```
1:  function DijkstraWithoutHeap(G)
2:      for v = 1 to n do
3:          status[v] = unseen
4:      end for
5:      status[s] = in − tree; bw[s] = ∞
6:      for edge(s, w) do
7:          status[s] = fringe
8:          bw[w] = weight(s, w)
9:          dad[w] = s
10:     end for
11:     while status[t] ≠ in − tree do
12:         pick fringe v from max − bw[v]
13:         status[v] = in − tree
14:         for each edge (v, w) do
15:             if status[w] = unseen then
16:                 status[w] = fringe
17:                 bw[w] = min{bw[v], weight[v, w]}
18:                 dad[w] = v
19:             else if status[w] = fringe and bw[w] < min{bw[v], weight[v, w]} then
20:                 bw[w] = min{bw[v], weight[v, w]}
21:                 dad[w] = v
22:             end if
23:         end for
24:     end while
25:     Return dad[1...n]
26: end function
```

4.2 Dijkstra with Heap

Algorithm 2 Dijkstra with Heap

```
1: function DijkstraWithHeap(G)
2:     for v = 1 to n do
3:         status[v] = unseen
4:     end for
5:     status[s] = in − tree; bw[s] = ∞
6:     for edge(s, w) do
7:         status[s] = fringe
8:         bw[w] = weight(s, w)
9:         dad[w] = s
10:    end for
11:    while status[t] ≠ in − tree do
12:        v = max(fringe)
13:        Delete(fringe)
14:        status[v] = in − tree
15:        for each edge (v, w) do
16:            if status[w] = unseen then
17:                status[w] = fringe
18:                bw[w] = min{bw[v], weight[v, w]}
19:                fringe.Insert(w, bw[w])
20:                dad[w] = v
21:            else if status[w] = fringe and bw[w] < min{bw[v], weight[v, w]} then
22:                bw[w] = min{bw[v], weight[v, w]}
23:                fringe.Insert(w, bw[w])
24:                dad[w] = v
25:            end if
26:        end for
27:    end while
28:    Return dad[1...n]
29: end function
```

4.3 Kruskal with Heapsort Edges

Algorithm 3 Kruskal with Heapsort Edges

```
1: function KruskalWithHeapSort(G)
2:     define arrays of Parent and Rank
3:     apply HeapSort to Edges
4:     G′ = G without Edges
5:     for each v in G′ do
6:         MakeSet(v)
7:     end for
8:     for each e in Edges (u, v) do
9:         Rank_u = Find(u)
10:        Rank_v = Find(v)
11:        if Rank_u ≠ Rank_v then
12:            G′.AddEdge(u, v)
13:            Union(Rank_u, Rank_v)
14:        end if
15:    end for
16:    Return G′
17: end function


18: function MakeSet(v)
19:    p[v] = −1
20:    Rank[v] = 0
21: end function


22: function Union(r_1, r_2)
23:    if Rank[r_1] < Rank[r_2] then
24:        p[r_1] = r_2
25:    else if Rank[r_2] < Rank[r_1] then
26:        p[r_2] = r_1
27:    else
28:        p[r_1] = r_2
29:        Rank[r_2] + +
30:    end if
31: end function


32: function Find(v)
33:    w = v
34:    while p[w] ≠ 0 do
35:        w = p[w]
36:    end while
37:    Return w
38: end function
```

5 Testing

There are three different algorithms I utilized for the analysis of finding maximum bandwidth path given a graph and a pair of $(source, destination)$.

- *Dijkstra's without heap:* is a modified version of Dijkstra's shortest path algorithm for finding maximum bandwidth path which does not use the heap, requires linear time to find the vertex at maximum bandwidth among all the neighbours. As there can be n nodes, m edges in the graph, the worst case complexity of the algorithm becomes $O(n + m)$.
- *Dijkstra's with heap:* can be utilized using heap. With the help of heap, max–heap is fast and precise, the operation is efficient which takes $O(\log n)$ however, the insertion operation takes $O(1)$ in the forst approach, now takes $O(\log n)$. With total n nodes and m edges, the running time of the algorithm is $O((n + m) \log n)$ in the worst case.
- *Kruskal with heap:* is different from the above methods uses Kruskal's algorithm to find the maximum spanning tree and it is guaranteed that the maximum bandwidth path from any pair of vertices lies in this spanning tree. Kruskal's algorithm inserts all the m edges into the heap which takes $O(m \log m)$. For cycle detection, I have applied Union–Find algorithm which has amortized complexity of $O(\log^* n)$. Once we have the tree, we can find the source to destination path in linear time $O(n + m)$.

5.1 Graph G1 time comparison

	Dijkstra without Heap	Dijkstra with Heap	Kruskal with Heapsort
graph 1	2.69210195541	0.172005176544	2.54193210602
graph2	0.03339695930	0.024281024932	2.46159410477
graph 3	0.39941906929	0.030972003936	2.41584300995
graph 4	4.02639222145	0.285075187683	3.03517580032
graph 5	3.24632811546	0.173991918564	2.57944917679
Average	2.07952766418	0.137265062331	2.60679883956

Running Time: Dijkstra with Heap < Dijkstra without Heap < Kruskal without Heapsort

5.2 Graph G2 time comparison

	Dijkstra without Heap	Dijkstra with Heap	Kruskal with Heapsort
graph 1	2.31743502617	4.11292004585	423.561660051
graph2	8.53429794312	1.29516100883	422.908943176
graph 3	8.84093999863	4.80018806458	423.089514971
graph 4	4.55925488472	2.95114088058	422.802627087
graph 5	8.85581207275	2.84955406189	415.444319963
Average	6.62154798507	3.20179281234	421.561413049

Running Time: Dijkstra with Heap < Dijkstra without Heap < Kruskal without Heapsort

5.3 Time comparison conclusion

- $G2 \approx 3.0$ times slower than $G1$ while using Dijkstra without Heap
- $G2 \approx 23.0$ times slower than $G1$ while using Dijkstra with Heap
- $G2 \approx 162.0$ times slower than $G1$ while using Kruskal without Heapsort

5.4 Comparison note

According to the results, for $G1$ (sparse graphs), the heap method is a proper solution that does not rely on queries amount, and for $G2$ (dense graphs), there exist some performances. However, using the Kruskal method does not show a performance in results especially if the graph changes most often with few amount of queries for path finding. Meanwhile, for dense graphs is the overhead of MST is large and have lower efficiency due to the insertion of a large amount of edges into the heap structure which takes $O(\log n)$ per each insertion. Finally, Dijkstra without heap method is not recommended to be chosen compare to the Dijkstra with heap which is the best option due to it dependency of the graph and the amount of the queries.

6 System metrics

Macbook Pro (Mid-2012)
Processor: 2.5 GHz Dual–Core Intel Core i5
Memory: 8 GB 1600 MHz DDR3
Programming Language: Python 3.70

7 Graph G1 Log File

```
----------INITIALIZING GRAPH----------
Selection options:
    [1] G1 Graph
    [2] G2 Graph
Enter option: 1
-------------------------------------------
Enter Graph type G1 Nodes Number: 5000
[result] Number of Edges in Graph G1: 15000
[result] Total Time for creating Graph G1: 9.81101107597 s
-------------------------------------------
[info] Testing Process Started...
-------------------------------------------

------------iteration pair graph 1 -------------
[Solution 1] Max Bandwidth Path, Dijkstra without Heap:
    Max Bandwidth: 14.0
    Max Bandwidth Path:4433 → 166 → 165 → 4384 → 755 → 3031 → 3030 →
    3544 → 296 → 4256 → 4255 → 3197 → 3487 → 2313 → 2314 → 62 → 1195
     → 3133 → 3134 → 3135 → 3136 → 3275 → 3274 → 66 → 67 → 3723 →
    3724 → 2935 → 1000 → 2133 → 2951 → 2843 → 1072 → 125 → 126 → 127
     → 747 → 3257 → 3256 → 1616 → 3023 → 3022 → 2276 → 1050 → 1049
    → 275 → 3130 → 3129 → 3621 → 2744 → 2743 → 2610 → 2015 → 1538 →
     1537 → 3250 → 1304 → 2061 → 1021 → 2174 → 2173 → 2102 → 317 →
    3731 → 3730 → 3581 → 3300 → 3299 → 417 → 4445 → 4933 → 2710 →
    1076 → 1075
    S-T path: (4433 → 1075)
[Solution 2] Max Bandwidth Path: Dijkstra with Heap:
    Max Bandwidth: 14.0
    Max Bandwidth Path:4433 → 4432 → 4777 → 4776 → 1369 → 815 → 816 →
    817 → 4890 → 327 → 326 → 1549 → 4423 → 1258 → 2529 → 3881 → 4052
     → 4051 → 558 → 4645 → 4646 → 4647 → 4648 → 4649 → 1666 → 1665
    → 1688 → 4887 → 3730 → 3581 → 3300 → 3299 → 417 → 4445 → 4933 →
     2710 → 1076 → 1075
    S-T path: (4433 → 1075)
[Solution 3] Max Bandwidth Path: Kruskal with Heap Sort:
    Max Bandwidth: 14.0
    Max Bandwidth Path:4433 → 166 → 165 → 4384 → 755 → 3031 → 3030 →
    3544 → 296 → 4256 → 4255 → 3197 → 3487 → 2313 → 2314 → 62 → 1195
     → 4676 → 4123 → 4124 → 842 → 900 → 901 → 4155 → 999 → 998 →
```

```
    3965 → 1805 → 1804 → 3816 → 1150 → 1890 → 1891 → 1468 → 120 →
    3220 → 4373 → 4374 → 2785 → 241 → 240 → 851 → 4515 → 4514 → 4985
     → 4984 → 3065 → 385 → 572 → 127 → 126 → 125 → 3870 → 3871 →
    1574 → 3444 → 3443 → 1861 → 994 → 2189 → 4333 → 319 → 318 → 317
     → 3731 → 3730 → 3581 → 3300 → 3299 → 417 → 4445 → 4933 → 2710 →
     1076 → 1075
    S-T path: (4434 → 1076)

------------iteration pair graph 2 -------------
[Solution 1] Max Bandwidth Path, Dijkstra without Heap:
    Max Bandwidth: 16.0
    Max Bandwidth Path:108 → 2512 → 3323 → 3324 → 662 → 2176 → 4035 → 42
     → 4483 → 64 → 65 → 917 → 1843 → 1557 → 2141 → 2140 → 818 →
    255 → 423
    S-T path: (108 → 423)
[Solution 2] Max Bandwidth Path: Dijkstra with Heap:
    Max Bandwidth: 16.0
    Max Bandwidth Path:108 → 2512 → 3323 → 3324 → 662 → 2176 → 4035 →
    943 → 942 → 1123 → 3063 → 1701 → 1702 → 1703 → 3310 → 268 → 3998
     → 3907 → 1664 → 2180 → 829 → 830 → 4414 → 2725 → 307 → 306 →
    305 → 4350 → 2147 → 2146 → 2145 → 3136 → 3275 → 3274 → 66 → 998
     → 3965 → 3142 → 3143 → 545 → 544 → 1517 → 423
    S-T path: (108 → 423)
[Solution 3] Max Bandwidth Path: Kruskal with Heap Sort:
    Max Bandwidth: 16.0
    Max Bandwidth Path:108 → 2512 → 3323 → 3324 → 662 → 2176 → 4035 →
    943 → 942 → 1123 → 3063 → 3064 → 3604 → 1644 → 1909 → 213 → 214
     → 1090 → 1734 → 3630 → 3631 → 4786 → 2882 → 2883 → 2964 → 408 →
     2770 → 2769 → 2868 → 1021 → 4515 → 851 → 240 → 241 → 2785 →
    4374 → 4373 → 3220 → 3219 → 1516 → 1517 → 423
    S-T path: (109 → 424)

------------iteration pair graph 3 -------------
[Solution 1] Max Bandwidth Path, Dijkstra without Heap:
    Max Bandwidth: 14.0
    Max Bandwidth Path:63 → 2817 → 2818 → 1527 → 1526 → 1781 → 1782 →
    739 → 738 → 501 → 500 → 1992 → 1991 → 2252 → 1558 → 435 → 436 →
     1241 → 878 → 1733 → 1732 → 1731 → 510 → 511 → 1347 → 896 → 897
     → 1049 → 275 → 2021 → 2020 → 2019 → 2018 → 572 → 127 → 126 →
    125 → 996 → 767 → 1432 → 1431 → 1430 → 1957 → 1462 → 1463 → 1878
     → 1877 → 1876 → 1481 → 1480 → 1479 → 1124 → 1125 → 1641 → 1807
     → 2169 → 2168
    S-T path: (63 → 2168)
[Solution 2] Max Bandwidth Path: Dijkstra with Heap:
    Max Bandwidth: 14.0
    Max Bandwidth Path:63 → 2817 → 4452 → 4260 → 4259 → 4000 → 4001 →
    1250 → 2920 → 2555 → 2556 → 105 → 106 → 2151 → 2869 → 1242 →
    1243 → 21 → 20 → 3447 → 2339 → 2013 → 4630 → 493 → 492 → 2169 →
     2168
    S-T path: (63 → 2168)
[Solution 3] Max Bandwidth Path: Kruskal with Heap Sort:
    Max Bandwidth: 14.0
    Max Bandwidth Path:63 → 2817 → 2818 → 1527 → 1526 → 429 → 2188 →
    2187 → 3133 → 1195 → 4676 → 4123 → 4124 → 842 → 900 → 901 → 4155
     → 999 → 998 → 3965 → 1805 → 1804 → 3816 → 1150 → 1890 → 1891 →
     1468 → 120 → 3220 → 4373 → 4374 → 4842 → 1126 → 1125 → 1641 →
    1807 → 2169 → 2168
    S-T path: (64 → 2169)

------------iteration pair graph 4 -------------
[Solution 1] Max Bandwidth Path, Dijkstra without Heap:
    Max Bandwidth: 13.0
    Max Bandwidth Path:1809 → 1808 → 1807 → 1641 → 1125 → 1124 → 4797 →
    3598 → 771 → 772 → 2843 → 487 → 1429 → 377 → 420 → 421 → 3288 →
     2280 → 227 → 1820 → 1275 → 1274 → 1767 → 4443 → 3256 → 1616 →
```

```
    3023 → 3022 → 2276 → 1050 → 4394 → 4395 → 3938 → 3937 → 3936 →
    3967 → 2324 → 2596 → 899 → 900 → 901 → 4155 → 999 → 998 → 3965
    → 1805 → 1804 → 3816 → 1150 → 1890 → 1891 → 1468 → 120 → 3220 →
     4373 → 4374 → 2785 → 241 → 240 → 851 → 4515 → 1021 → 2868 →
    2769 → 2770 → 408 → 2964 → 2883 → 2882 → 4786 → 3631 → 3630 →
    1734 → 1090 → 214 → 213 → 1909 → 1644 → 3604 → 3064 → 3063 →
    1123 → 942 → 943 → 2588 → 2589
    S-T path: (1809 → 2589)
[Solution 2] Max Bandwidth Path: Dijkstra with Heap:
    Max Bandwidth: 13.0
    Max Bandwidth Path:1809 → 1808 → 1807 → 1641 → 1125 → 1126 → 4842 →
    4374 → 4373 → 3220 → 120 → 1468 → 1891 → 1890 → 1150 → 1149 →
    1148 → 2392 → 2937 → 1417 → 1418 → 82 → 4138 → 3085 → 4840 →
    4839 → 3122 → 1190 → 2436 → 1281 → 1280 → 498 → 499 → 500 → 1992
     → 4857 → 1681 → 3920 → 4021 → 1485 → 1484 → 1483 → 1912 → 1913
    → 267 → 3262 → 3261 → 865 → 2944 → 3704 → 4683 → 4724 → 65 →
    64 → 4483 → 42 → 4035 → 943 → 2588 → 2589
    S-T path: (1809 → 2589)
[Solution 3] Max Bandwidth Path: Kruskal with Heap Sort:
    Max Bandwidth: 13.0
    Max Bandwidth Path:1809 → 1808 → 1807 → 1641 → 1125 → 1126 → 4842 →
    4374 → 2785 → 241 → 240 → 851 → 4515 → 1021 → 2868 → 2769 → 2770
     → 408 → 2964 → 2883 → 2882 → 4786 → 3631 → 3630 → 1734 → 1090
    → 214 → 213 → 1909 → 1644 → 3604 → 3064 → 3063 → 1123 → 942 →
    943 → 2588 → 2589
    S-T path: (1810 → 2590)

------------iteration pair graph 5 -------------
[Solution 1] Max Bandwidth Path, Dijkstra without Heap:
    Max Bandwidth: 15.0
    Max Bandwidth Path:356 → 151 → 3192 → 4750 → 1481 → 1480 → 4340 →
    2275 → 2276 → 3022 → 3023 → 1616 → 3256 → 3257 → 747 → 127 → 126
     → 125 → 1072 → 2843 → 2951 → 2133 → 1000 → 2935 → 3724 → 3723
    → 67 → 66 → 3274 → 3275 → 3136 → 3135 → 3134 → 3133 → 1195 → 62
     → 2314 → 2313 → 3487 → 3197 → 4255 → 4256 → 296 → 3544 → 3030
    → 3031 → 755 → 4384 → 4383 → 3823 → 1061 → 1062 → 1595 → 1596 →
     1597 → 340 → 37 → 36 → 351
    S-T path: (356 → 351)
[Solution 2] Max Bandwidth Path: Dijkstra with Heap:
    Max Bandwidth: 15.0
    Max Bandwidth Path:356 → 151 → 3192 → 4750 → 1481 → 1480 → 3316 →
    3315 → 4861 → 4606 → 4605 → 4435 → 4758 → 3906 → 3907 → 1664 →
    2180 → 829 → 830 → 4414 → 2725 → 307 → 306 → 305 → 3327 → 4935
    → 4934 → 273 → 320 → 1962 → 4532 → 4531 → 1245 → 4557 → 4896 →
    4791 → 4583 → 3065 → 4984 → 4985 → 4514 → 4515 → 4214 → 4215 →
    232 → 377 → 4853 → 4852 → 250 → 251 → 1959 → 4910 → 4909 → 4307
     → 586 → 587 → 588 → 4676 → 1195 → 62 → 2314 → 2313 → 3487 →
    3197 → 4255 → 4256 → 296 → 3544 → 3030 → 3031 → 755 → 4384 →
    4383 → 3823 → 1061 → 1062 → 1595 → 1596 → 1597 → 340 → 37 → 36
    → 351
    S-T path: (356 → 351)
[Solution 3] Max Bandwidth Path: Kruskal with Heap Sort:
    Max Bandwidth: 15.0
    Max Bandwidth Path:356 → 151 → 3192 → 4750 → 1481 → 1581 → 1580 →
    4787 → 4786 → 2882 → 2883 → 2964 → 408 → 2770 → 2769 → 2868 →
    1021 → 4515 → 4514 → 4985 → 4984 → 3065 → 385 → 572 → 127 → 126
     → 125 → 3870 → 3871 → 1574 → 1573 → 3722 → 2879 → 4610 → 1144 →
      4632 → 344 → 343 → 1062 → 1595 → 1596 → 1597 → 340 → 37 → 36
    → 351
    S-T path: (357 → 352)

------------------------Algorithm Run Time--------------------------
Dijkstra without Heap | Dijkstra with Heap | Kruskal with Heapsort
2.69210195541         0.172005176544         2.54193210602
Dijkstra without Heap | Dijkstra with Heap | Kruskal with Heapsort
```

```
0.0333969593048          0.0242810249329          2.46159410477
Dijkstra without Heap | Dijkstra with Heap | Kruskal with Heapsort
0.39941906929          0.0309720039368          2.41584300995
Dijkstra without Heap | Dijkstra with Heap | Kruskal with Heapsort
4.02639222145          0.285075187683          3.03517580032
Dijkstra without Heap | Dijkstra with Heap | Kruskal with Heapsort
3.24632811546          0.173991918564          2.57944917679
-----------------------------Run Ended!-------------------------------
```

8 Graph G2 Log File

```
----------INITIALIZING GRAPH----------
Selection options:
    [1] G1 Graph
    [2] G2 Graph
Enter option: 2
------------------------------------------
Enter Graph type G2 Nodes Number: 5000
[result] Number of Edges in Graph G2: 2602637
[result] Total Time for creating Graph G2: 144.103569031 s
------------------------------------------
[info] Testing Process Started...
------------------------------------------

------------iteration pair graph 1 -------------
[Solution 1] Max Bandwidth Path, Dijkstra without Heap:
    Max Bandwidth: 20.0
    Max Bandwidth Path:1983 → 155 → 105 → 20 → 28 → 71 → 8 → 9 → 63 → 865
    S-T path: (1983 → 865)
[Solution 2] Max Bandwidth Path: Dijkstra with Heap:
    Max Bandwidth: 20.0
    Max Bandwidth Path:1983 → 1281 → 4938 → 1358 → 865
    S-T path: (1983 → 865)
[Solution 3] Max Bandwidth Path: Kruskal with Heap Sort:
    Max Bandwidth: 20.0
    Max Bandwidth Path:1983 → 509 → 4041 → 4039 → 4033 → 1045 → 4030 → 2654 → 2014 → 2388 → 4031 → 865
    S-T path: (1984 → 866)

------------iteration pair graph 2 -------------
[Solution 1] Max Bandwidth Path, Dijkstra without Heap:
    Max Bandwidth: 20.0
    Max Bandwidth Path:2430 → 11 → 6 → 112 → 21 → 8 → 71 → 15 → 129 → 97 → 102 → 58 → 222 → 4735
    S-T path: (2430 → 4735)
[Solution 2] Max Bandwidth Path: Dijkstra with Heap:
    Max Bandwidth: 20.0
    Max Bandwidth Path:2430 → 178 → 916 → 4829 → 4735
    S-T path: (2430 → 4735)
[Solution 3] Max Bandwidth Path: Kruskal with Heap Sort:
    Max Bandwidth: 20.0
    Max Bandwidth Path:2430 → 4087 → 78 → 4052 → 7 → 4047 → 110 → 4044 → 390 → 1009 → 148 → 504 → 4031 → 2388 → 2014 → 2654 → 4030 → 1020 → 4735
    S-T path: (2431 → 4736)

------------iteration pair graph 3 -------------
[Solution 1] Max Bandwidth Path, Dijkstra without Heap:
    Max Bandwidth: 20.0
    Max Bandwidth Path:2972 → 44 → 9 → 8 → 71 → 28 → 104 → 126 → 127 → 128 → 4587
```

```
    S-T path: (2972 → 4587)
[Solution 2] Max Bandwidth Path: Dijkstra with Heap:
    Max Bandwidth: 20.0
    Max Bandwidth Path:2972 → 343 → 1003 → 4587
    S-T path: (2972 → 4587)
[Solution 3] Max Bandwidth Path: Kruskal with Heap Sort:
    Max Bandwidth: 20.0
    Max Bandwidth Path:2972 → 1020 → 4030 → 2654 → 2014 → 873 → 1011 → 4069 → 4090 → 4587
    S-T path: (2973 → 4588)

------------iteration pair graph 4 -------------
[Solution 1] Max Bandwidth Path, Dijkstra without Heap:
    Max Bandwidth: 20.0
    Max Bandwidth Path:3869 → 194 → 138 → 14 → 40 → 81 → 1579
    S-T path: (3869 → 1579)
[Solution 2] Max Bandwidth Path: Dijkstra with Heap:
    Max Bandwidth: 20.0
    Max Bandwidth Path:3869 → 669 → 3923 → 4890 → 1579
    S-T path: (3869 → 1579)
[Solution 3] Max Bandwidth Path: Kruskal with Heap Sort:
    Max Bandwidth: 20.0
    Max Bandwidth Path:3869 → 4037 → 503 → 1955 → 4031 → 2388 → 2014 → 4693 → 4029 → 4038 → 16 → 2050 → 1579
    S-T path: (3870 → 1580)

------------iteration pair graph 5 -------------
[Solution 1] Max Bandwidth Path, Dijkstra without Heap:
    Max Bandwidth: 20.0
    Max Bandwidth Path:2929 → 14 → 40 → 81 → 118 → 67 → 4722
    S-T path: (2929 → 4722)
[Solution 2] Max Bandwidth Path: Dijkstra with Heap:
    Max Bandwidth: 20.0
    Max Bandwidth Path:2929 → 209 → 4722
    S-T path: (2929 → 4722)
[Solution 3] Max Bandwidth Path: Kruskal with Heap Sort:
    Max Bandwidth: 20.0
    Max Bandwidth Path:2929 → 4079 → 248 → 2017 → 485 → 1008 → 1514 → 4031 → 2388 → 2014 → 123 → 508 → 4722
    S-T path: (2930 → 4723)

------------------------Algorithm Run Time--------------------------
Dijkstra without Heap | Dijkstra with Heap | Kruskal with Heapsort
2.31743502617         4.11292004585         423.561660051
Dijkstra without Heap | Dijkstra with Heap | Kruskal with Heapsort
8.53429794312         1.29516100883         422.908943176
Dijkstra without Heap | Dijkstra with Heap | Kruskal with Heapsort
8.84093999863         4.80018806458         423.089514971
Dijkstra without Heap | Dijkstra with Heap | Kruskal with Heapsort
4.55925488472         2.95114088058         422.802627087
Dijkstra without Heap | Dijkstra with Heap | Kruskal with Heapsort
8.85581207275         2.84955406189         415.444319963
----------------------------Run Ended!-----------------------------
```