Comparison of algorithms for finding Maximum Bandwidth Path

Introduction:

In this report, I mainly want to focus on the analysis of 3 different approaches of finding the maximum bandwidth path between the two vertices of a graph. So I will give a brief introduction about the environment setup that I used for carrying out the experiment and how I carried out those experiments.

Random Generation of Graph:

To have the entire graph connected, I have connected all the nodes in a cycle. To maintain the randomness, I created an array of 1 to N nodes and after random shuffling of the array, I connected each vertex a[i] with a[i+1] for i = 1 to N-1 and finally a[N] to a[1].

For sparse graphs, the degree of each vertex is between 7 to 9, while for that in dense graph is 900 to 1100 approximately. The graph is generated by connecting each vertex to some other vertices randomly until it reaches the acceptable degree range. The graph is undirected. The weight of each edge is chosen randomly between the range 100 to 5000.

Heap structure:

I have created two heaps for two algorithms. In the Dijkstra's modified version of shortest path using heap, the heap supports search and update operation efficiently. For that I have maintained an extra array to keep track of each vertex where it is located. The heap stores nodes and maximum capacity of heap is 50 thousand nodes. The max-heap is maintained based on bandwidth array passed to each heap operation as a parameter.

Another heap required for finding Kruskal's maximum spann ing tree has a little different structure. It stores edges in it in the form (u, v, w) i.e.d edge from u to v with weight w. This is also a max-heap based on weight of the edge.

Algorithms (Short description here. In the analysis part, we again talk about each):

1. The modification of Dijkstra's shortest path algorithm without heap - Here I use an extra array for storing the bandwidth possible so far from s to each other vertex which is initialized to -1. For ease of finding path, we also maintain a parent array.
2. The modification of Dijkstra's shortest path algorithm with heap - Here I use a heap structure described above and an extra array for storing the bandwidth possible so far from s to each other vertex which is initialized to -1. Again, for the ease of finding path, we also maintain a parent array.
3. The modification of Kruskal's algorithm - Here again I use the heap structure described above. For checking if addition of an edge creates a cycle, I use Union-Find algorithm.

Comparison of algorithms for finding Maximum Bandwidth Path

Testing:

1. I have created 10 dense and 10 sparse graphs. For each graph, I chose 10 pairs of vertices to find the maximum bandwidth path between them and then take average of the timings.
2. Out of interest, I have analyzed 3 dense and 3 sparse graphs, where I chose 500 and 2000 pairs of vertices per graph to find the maximum bandwidth path between them.
3. All the timings measured are in seconds.

Implementation Details:

1. I have implemented the project in C++ language.
2. For storing the heap in Kruskal's approach, I use vector in STL which are equivalent to dynamically allocated arrays. It was not possible to use static arrays to maintain a heap of ~2.5 million edges.
3. I store the graph in unordered_map where I map each vertex to a vector which contain a connected node and the edge weight pair.
4. Union Find algorithm uses union by rank as well as path compression heuristics in the implementation.
5. The s-->t path in maximum spanning tree is found using BFS traversal.

Analysis:

There are 3 different algorithms which I used for the analysis of finding maximum bandwidth path given a graph and pair of (source, destination). The results of the experiment were interesting in multiple ways. As discussed in class, the three approaches are -

1. Dijkstra's Naive Approach - 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^2+m)**.
2. Dijkstra's Heap Approach - Another version of Dijkstra's algorithm can be implemented using the heap. Now, with the help of heap, max-heap to be precise, the operation of finding minimum is comparatively very efficient which takes O($lg(n)$); but the insertion operation which was happening in O(1) in the $(1)^{st}$ approach, now takes O($lg(n)$). As there are in total 'n' nodes, 'm' edges again in the graph, the running time of the algorithm is **O(($n+m$)*$lg(n)$)** in the worst case.
3. Kruskal's Maximum Spanning Tree Approach - The third approach is very different from the above two which 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 edges into the heap and there are 'm' such edges. So it will take **O(*m*lg(m)*)** time to insert. Later to detect the cycle, I have used Union-Find algorithm which has amortized complexity of O(*lg*(n)*). Once we have the tree, we can find the source to destination path in linear time **O(*n+m*)** using BFS (or DFS, I have used BFS).

Looking at the theoretical running time of the algorithms, we can conclude, or mis-conclude, that the approach (1) is worst among all and approaches (2) and (3) are comparable. But the interesting observation is that it really depends on the application. I have done the analysis on two categories of graphs - Sparse and Dense. Here is the simple summary of the observations.

Graph category --> Algorithm	Sparse	Dense
Dijkstra's Naive Approach	Too bad	Good
Dijkstra's Heap Approach	Very good	Very good
Kruskal's Maximum Spanning Tree Approach	Too bad	Too bad

(First Impression : 10 graphs, 10 queries per graph)

We didn't expect the 3rd approach to perform worse than the 1st approach, did we? Lets also have a closer look at the average time that it took for each algorithm for each category of graphs. The experiment was done on 10 different randomly generated graphs with 10 pairs of vertices per graph chosen randomly as source and destination, for each category of graph. I wanted to make sure that I have as accurate analysis as possible.

Graph category --> Algorithm	Sparse	Dense
Dijkstra's Naive Approach	0.0786	0.1177
Dijkstra's Heap Approach	0.0041	0.0635
Kruskal's Maximum Spanning Tree Approach	0.0798	2.902
Kruskal's : Naive	1:1	25:1
Kruskal's : Heap	20:1	45:1
Naive : Heap	20:1	2:1

(Runtime in seconds : Average of 10 graphs, 10 queries per graph : Compute MST everytime)

Comparison of algorithms for finding Maximum Bandwidth Path

What? The 3rd approach is so worse than the other two! It almost takes 25 times more time than the 1st approach. This was not expected. Let's take an even closer look at the timing below.

Graph category --> Algorithm	Sparse	Dense
Dijkstra's Naive Approach	0.0786	0.1177
Dijkstra's Heap Approach	0.0041	0.0635
Kruskal's Maximum Spanning Tree Creation	0.076	2.8981
Finding the s-->t path from the Max. Spanning Tree	0.0037	0.0039
Path Finding : Naive	0.04:1	0.03:1
Path Finding : Heap	1:1	0.06:1
MST Creation : Path Finding	20:1	743:1

(Average of 10 graphs, 10 queries per graph : Compute time for MST & path separately)

Now the things start getting very interesting. The entire time that the 3rd approach was taking, was highly contributed by just the Maximum Spanning Tree creation. Finding the maximum bandwidth path from the maximum spanning tree is so quick. To be precise, approximately the MST creation takes over 99% of the total time required for running the entire algorithm compared to just pathfinding.

Anyway, the 3rd approach is still bad, why not just use the other ones which are performing so better than this one?
Wait a second, I sense something here. The key observation is that Finding MST is not dependent on what s-->t path we want, right? So why not just try one thing - compute the MST only once per graph and then use it for all s-->t pairs on that graph. Lets see what numbers has to say.
(The observations mentioned below are the outcome of an experiment done considering the above fact. The MST is computed only once and then used multiple times directly)

Comparison of algorithms for finding Maximum Bandwidth Path

Graph category --> Algorithm	Sparse	Dense
Dijkstra's Naive Approach	0.0786	0.1177
Dijkstra's Heap Approach	0.0041	0.0635
Kruskal's Maximum Spanning Tree Approach	0.0113	0.2937
Kruskal's : Naive	0.14:1	2.5:1
Kruskal's : Heap	2.75:1	4.6:1

(Average of 10 graphs, 10 queries per graph: Compute MST once per graph, use it per query)

The difference is reduced significantly. Now the 3rd approach is behaving comparable to the 1st approach and 2nd approach too. Remember, I had only 10 vertices per graph in the above experiment. So to take it further, I decided to carry out an extra experiment, where for the same graph- I chose 500 different pairs of vertices. There are 5000 nodes in the graph, so total possible pairs are approximately 1.25 million. Out of which I am only taking 500. Now, the numbers depict totally different story.
(I carried out this experiment with 2 sparse and 2 dense graphs)

Graph category --> Algorithm	Sparse	Dense
Dijkstra's Naive Approach	0.0746	0.129
Dijkstra's Heap Approach	0.0039	0.0676
Kruskal's Maximum Spanning Tree Approach	0.004	0.01
Kruskal's : Naive	0.05:1	0.07:1
Kruskal's : Heap	1:1	0.14:1

(Average of 2 graphs, 500 queries per graph - MST once per graph, use result per query)

Now, the 3rd approach performed way better than earlier. For both sparse as well as dense graphs, the algorithm works very quickly. For sparse graphs, heap approach is still a very good choice. Just out of curiosity, I increased my number of pairs of vertices from 500 to 2000. Carried out the experiment once with dense and once with sparse graph. As expected, the 3rd approach outperformed the other two by huge margin.

Comparison of algorithms for finding Maximum Bandwidth Path

Graph category --> Algorithm	Sparse	Dense
Dijkstra's Naive Approach	0.0692	0.1305
Dijkstra's Heap Approach	0.0035	0.0686
Kruskal's Maximum Spanning Tree Approach	0.0035	0.006
Kruskal's : Naive	0.05:1	0.04:1
Kruskal's : Heap	1:1	0.08:1

(1 graph, 2000 queries - Compute MST once, use result per query)

If we see the above two tables carefully, we observe that for sparse graphs, there is no much improvement in the performance in 3rd approach as compared to the other approaches with 500 queries. We also observed that the heap approach behaved the best among other approaches since the beginning. So I will conclude here that, **for sparse graphs, the heap approach is the best one irrespective of how many queries would be there per graph.** Secondly, for dense graphs, we observed some improvement again. So **if we know that there will be a dense graph which won't change oftenly i.e. there are many queries per graph, then the spanning tree approach is the one to go with.** I would like to conclude one more thing about the 3rd approach - Finding Maximum Spanning Tree with Kruskal's algorithm and then finding the path. As we saw in the beginning, it does not make a sense to use this approach for this problem if we know that the graph is changing frequently and there are very few queries per graph to find s-->t path. The overhead of finding MST is too huge to get compensated by the efficiency of the BFS in finding the s-->t path in linear time. This is especially true when the graph is dense because we need to insert those many edges inside the heap which takes O(lg(n)) time per insertion.

What we didn't talk about much is the 1st and 2nd approach. If we observe all the data again, we will observe that 1st approach performs better or even close to 2nd approach in very rare cases. On average, it performs pretty bad - almost 20 times slower - compared to 2nd algorithm. I have some consolidated & averaged data about these two approaches. Let's have a look.

Comparison of algorithms for finding Maximum Bandwidth Path

	Sparse		Dense	
	Naive	Heap	Naive	Heap
Graph 1	0.0990	0.0038	0.1151	0.0733
Graph 2	0.0682	0.0037	0.1515	0.0731
Graph 3	0.0692	0.0028	0.1493	0.0668
Graph 4	0.0853	0.0042	0.1109	0.0663
Graph 5	0.0766	0.0043	0.1239	0.0681

(Average of 10 queries per graph)

I think this data is sufficient to support my earlier claim. So another conclusion is that, **naive approach without heap is not the one that we should choose in any scenario. Only possible scenario I can think of is that the graph is sparse, graph is changing frequently i.e. there are very less queries per graph and for some reason we can't implement the heap. In that case its reasonable to use this approach**. So, in general cases the 2nd approach - with heap - is the best choice because it does not depend on how the graph is and how many queries per graph are there. In average scenario, it performs better than both the other approaches. The real trouble with this approach is the practical implementation. The implementation of the heap with a mapping of each vertex using another array or map, is highly error prone, but we have to have that to support the efficient (O(lgn)) time search & update operation. Having implemented this myself, I think it is difficult to get it error free. The implementation bugs are very subtle, the bugs do not cause the algorithm to fails everytime. Once in some hundred instances the issue occurs, most of the times, errors do not occur on smaller graphs because there is a very little amount of decision making, and then debugging it becomes difficult to debug on bigger graphs, bigger heaps. But well, we want to have efficient algorithms, so the efforts are worth taking.

Finally, my recommendation from the entire analysis would be to use 2nd - heap based approach if we are completely unsure about how frequently the graph will change, how many s, t queries will be there per graph AND we can assure that the implementation is correct by testing it rigorously. If we have some idea about inputs - graph will change or not, if so - how frequently? How many queries approximately per graph will be there? - then its best to use 3rd MST based approach. The 1st approach should always be avoided unless we have smaller graphs and ease of implementation is the concern, then it is ok to go with 1st approach keeping in mind that we loose on running time for keeping the code maintainable (easy).

Comparison of algorithms for finding Maximum Bandwidth Path

The pseudocode to choose the algorithm among three depending on the application-

```
ALGORITHM Choose-Maximum-Bandwidth-Path-Approach:
      if (I know how the input will be):
            if (the graph is sparse):
                  if (the implementation is of big concern):
                        USE "WITHOUT-HEAP-APPROACH"
                  else:
                        USE "HEAP-APPROACH"
            else: // the graph is dense
                  if (there will be many queries per graph):
                        // approx more than 1% of the total number of nodes
                        USE "SPANNING-TREE-APPROACH"
                  else: // very few queries per graph
                        USE "HEAP-APPROACH"
      else: // no idea about input
            USE "HEAP-APPROACH"
```

(Algorithm to choose the right approach for particular application)