Fall
2021

CSCE 629: Analysis of Algorithms (Course Project)

UIN: 730007771

FARDEEN HASIB MOZUMDER

Introduction

This project aims to implement network routing protocols using various algorithms and data structures learnt in CSCE 629 course. In this project, we have solved the maximum bandwidth path problem which is an important topic in Computer science and Electrical Engineering and considered a significant network optimization problem. Usually, for a specific graph, which consists of many vertices and undirected edges between two vertices with arbitrary weight, we may able to find different paths from one source vertex to terminal we choose. However, since the bandwidth of one path depends on its smallest weight throughout the path, and many paths can share edges, the maximal bandwidth path from a certain source to a terminal may not be unique. On the other hand, finding a path that contains the maximum bandwidth between the source and destination can be achieved in multiple ways, but the performances can vary based on the type of structures used and how the algorithms are implemented. For our problem, we used two well-known algorithms to find maximum bandwidth path: Dijkstra's and Kruskal's. However, they each were modified and tuned to either use a heap structure, heapsort for edges, or not use the heap structure at all. With these three different versions of implementations, we measured the performance on two types of random, weighted undirected graphs which are mathematically represented as a sparse or a dense graph.

Implementation Details

1. Random Graph Generation

The first part of the problem was to generate two types of graphs: sparse and dense. Each of the graphs were implemented using adjacency lists and with 5000 vertices. In our implementation, the class "node" was initiated with a name and adjacent list of edges. And, the "edge" class was initiated with Source (src), Destination (dist) and a Weight (wt)/Bandwidth. Using these two classes, we were able to implement graphs of different types, the sparse and dense graphs. The sparse graph has a vertex degree of 6, which means that on average, the number of edges of a vertex is 6 and they must be randomly assigned. The edge weights were assigned a value from 1 to 10000 at random. On the other hand, the dense graph must ensure that each vertex is adjacent to about 20% of the other vertices, which means each vertex must be connected to at least 1000 other vertices on average. The edge weights for dense graphs were also assigned a value from 1 to 10000 randomly.

2. Heap Structure

There are many sorting algorithms, such as quick sort, insertion sort, merge sort etc. Heap sorting (non-increasing order) is also a kind of sorting which depends on a special data structure called MaxHeap. Here is an example of constructing a Max Heap data structure for an input of {35 33 42 10 14 19 27 44 26 31},

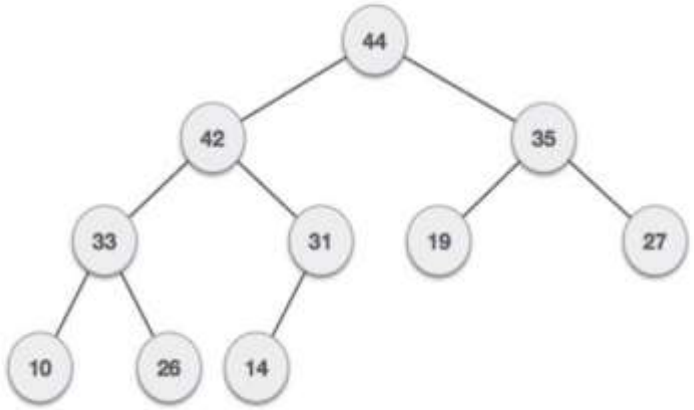

Figure: MaxHeap Structure

The property of max heap is that, the parent element would always be larger than its children. That means, value at MaxHeap[i] > MaxHeap[2i+1] (Left_Chlid) and MaxHeap[i] > MaxHeap[2i+2] (Right_Child). To build the MaxHeap structure and utilize it in our problem, we need to use insert operation, delete operation and PopMax operation, which in turn can violate the MaxHeap property. To fix this violation and maintain the property of Parent-Child relationship, I implemented swap, fix_heap_up and fix_heap_down operations, which allocate elements in their appropriate position. The workflow of these operations is discussed briefly in the following.

a) Insert:

Firstly, the new element is put at the last unoccupied position. Then, it is compared with its parent element. If its value is greater than its parent, we do a swap operation. We repeat this procedure, until its value is not greater than its parent. I named this process "fix_heap_up" in my implementation. Using this "fix_heap_up" process, we maintain the property of MaxHeap after an insertion operation.

b) Delete:

For deletion, we swap the MaxHeap index's value, where we want to perform the delete operation, with the last element of the heap and then assign a negative/invalid value to the last element. This in turn will violate the MaxHeap property, as the last element in the MaxHeap structure was smaller. So, we need to fix the heap by calling the "fix_heap_down" operation. The "fix_heap_down" operation swaps the parent with its largest child, if the parent's value is smaller than any of its children's values. The "fix_heap_down" operation will be repeated until the property of the MaxHeap is restored.

c) PopMax:

The Popmax pops the 1st element/largest element of the heap. This operation is quite useful if we want to implement HeapSort algorithm. The Popmax operation is similar to delete operation, except we want to delete the 1st element and return it. Then, we need to call the "fix_heap_down" operation to restore the MaxHeap property.

3. Routing Algorithms

The input for each of these algorithms is a graph G, a source vertex s, and a destination vertex t. Each of these algorithms will then return the maximum bandwidth path from s to t.

a. Dijkstra’s Algorithm without Heap Structure

Dijkstra’s algorithm was implemented based on the same pseudo code that was provided in class. Firstly, three arrays are initialized to store the dad, bandwidth, and status of each node. The status value was implemented using the numerical value of 2 for when a node is in-tree, 1 for when a node is a fringe, and 0 for when a node is still unseen. As, we are using an array to keep track of the fringes, we will need to iterate through the entire array in order to find the maximum fringe, increasing the overall time complexity of the algorithm. The pseudo code of Dijkstra’s algorithm with the use of an array to find the maximum bandwidth path is shown below:

Max Bandwidth path using Dijkstra’s algorithm without heap:

```
a) for each vertex v = 1 to n do
        status[v] = unseen
b) status[s] = intree
        BW[s] = inf
        dad[s] = -1
c) for each edge[s,v] do
        status[v] = fringe
        dad[v] = s
        BW[v] = wt(s,v)
d) while there are fringes do
        let v be the fringe with max(BW[v])
        status[v] = intree
        for each edge[v,w] do
                if status[w] == unseen
                        then status[w] = fringe
                                dad[w] = v
                                BW[w] = min(d[v], wt(v,t))
                else if status[w] == fringe and BW[w] < min(BW[v], wt(v,w))
                        then BW[w] = min(BW[v], wt(v,w))
                                dad[w] = v
```

Time Complexity for finding Max Bandwidth path using Dijkstra’s algorithm without heap:

In step d, the inner loop to find max runs n times at worst case and the outer loop can also run maximum of (n-1) times, as there can be at n-1 fringes in worst case. Hence, the time complexity of the algorithm is $O(n^2)$.

b. Dijkstra’s Algorithm with Heap Structure

This implementation of Dijkstra’s Algorithm is almost same as the last implementation, except in this case, we manage the fringe nodes in a MaxHeap structure. We insert all fringes using insert operation and by calling the fix_heap_up method, we bubble up the fringe with maximum bandwidth at the top. So, finding the maximum fringes take constant time only. But the insertion operation depends on the height of the MaxHeap, which can be highest (log n). So, every insertion can take O(logn) time at worst case. Again, after each PopMax operation, as we need to call “fix_heap_down” method to restore the MaxHeap property. Similar as insertion, the PopMax or Delete operations take O(logn) times as well. The pseudo code of Dijkstra’s algorithm with the use of a MaxHeap to find the maximum BW in a path is shown below:

Max Bandwidth path using Dijkstra’s algorithm with heap:

```
a) for each vertex v = 1 to n do
        status[v] = unseen
b) status[s] = intree
        BW[s] = inf
        dad[s] = -1
c) for each edge[s,v] do
        status[v] = fringe
        dad[v] = s
        BW[v] = wt(s,v)
        Heap.insert(v)
d) while there are fringes do
        v = Heap.Popmax()
        status[v] = intree
        for each edge[v,w] do
                if status[w] == unseen
                        then status[w] = fringe
                                dad[w] = v
                                BW[w] = min(d[v], wt(v,t))
                                Heap.insert(w)
                else if status[w] == fringe and BW[w] < min(BW[v], wt(v,w))
                              Heap.delete(w)
                              BW[w] = min(BW[v], wt(v,w))
                              Heap.insert(w);  dad[w] = v
```

Time Complexity of Max Bandwidth path using Dijkstra’s algorithm with heap:

The inner loop in step d runs for total number of edges m in worst case , and the insert in the heap takes logn times. Hence, this process can take O(mlogn). The outer loop can run maximum of (n-1) times and the PopMax operation in heap takes O(logn). Hence this process can take O(nlogn). Thus overall time complexity of this algorithm is O(nlogn + mlogn). As in our case, m>>n, so the total time complexity is O(mlogn).

c. Kruskal’s Algorithm with HeapSort

Kruskal’s algorithm was implemented using the pseudo code presented in class with a few modifications in order to take advantage of the HeapSort algorithm. Generally, we use Kruskal algorithm to find maximum/minimum spanning tree. But we can use Kruskal to find maximum BW path from a certain source to terminal by first building the maximum spanning BW tree of the original graph, and then apply DFS traversal method to find maximum BW path for that certain source/terminal pair. To find the maximum BW spanning tree of the original graph, we first sort the edges of the graph in non-increasing order using HeapSort. Then, using the MakeSet, Union, Find operations on the sorted edges, we keep adding edges in our spanning tree, as long as it does not create a cycle. In this way, we form the Max BW spanning graph of the input graph. After this step, by calling a simple DFS from the source vertex and by maintaining a dad array, we can find the maximum BW path of our desired path as well as the BW of that path. The final implementation of this modified version of Kruskal's algorithm with the use of heap sort is shown below:

Max Bandwidth path using Kruskal's algorithm with heapsort:

```
1.   Sort the edges in non-increasing order using heapsort:
         a.   $e_1, e_2, \dots, e_m$;
2.   For (v = 0; v < n; v++) MakeSet(v);
3.   for (k = 1; k <= m; k++)
3.1        let $e_k = [u_k, v_k]$;
3.2        $r_1$ = Find($u_k$); $r_2$ = Find($v_k$);
3.3        if ($r_1 \neq r_2$)
3.3           Union($r_1$, $r_2$);
4.   output(T).
```

```
MakeSet(v);
  p[v] = -1;
  h[v] = 0.
```

```
Find(v)
1. w = v;
2. while (p[w] $\neq$ -1)
      w = p[w];
3. return(w).
```

```
Union($r_1$, $r_2$);
  if (h[$r_1$] > h[$r_2$]) p[$r_2$] = $r_1$;
  else if (h[$r_2$] > h[$r_1$]) p[$r_1$] = $r_2$;
  else \\ h[$r_2$] = h[$r_1$])
      p[$r_2$] = $r_1$; h[$r_1$] = h[$r_1$] + 1.
```

Time Complexity of Max Bandwidth path using Kruskal's algorithm with heapsort:

Sorting m edges using Heapsort takes O(mlogm) and m Find operations with path compression takes O(mlog*n) time. But, as for our case, m>>n, so total time complexity for this algorithm is O(mlogm).

<u>Testing</u>

For testing, we generate 5 different pairs of sparse graph and dense graph. For each pair of graphs, we generate 5 different source and terminal pairs. We then apply our three different algorithms on each graph with each source-terminal pairs and record the time taken by each of the algorithms to find the maximum BW path for all the cases. We then evaluate and compare the results.

<u>Results</u>

Here, we show the performance of our implemented algorithms on both type of graphs on a same source/terminal pair. Other results will be shown in a tabular format in following.

Source = 4418**, Terminal =** 3336

Sparse Graph:

```
4418->2024->4838->1643->2989->1778->4112->3153->3434->2316->3267->3248->4664->3433->361
5->2392->3929->3197->4362->3294->4498->3957->3542->3554->4620->4018->2802->2829->3877->3
555->3619->4701->4691->2080->3690->2430->3480->3059->4583->2280->3535->4413->4287->1841-
>4067->2921->3850->3965->4123->2225->2029->4062->2254->2855->2489->4372->1655->3336
Maximum Bandwidth with dijkstra:  5424
Time taken:  1.2712302207946777
```

```
4418->2024->3385->283->926->2314->4607->3863->3537->2266->1137->710->4017->4560->3582->3
348->159->2074->1655->3336
Maximum Bandwidth with dijkstra with heap:  5424
Time taken:  0.15621304512023926
```

```
4418->2024->3385->283->926->2314->4607->3863->3537->2266->1646->2225->4123->1595->3229->
4540->3071->2455->4871->4793->1943->1114->1108->167->2611->51->3342->1401->92->1324->138
3->4548->116->4930->2424->1226->581->3119->758->2382->3303->284->2748->3955->245->1990->
3822->4073->1017->618->3633->108->905->4210->2766->4845->4166->1874->353->328->3336
Maximum Bandwidth with kruskal:  5424
Time taken:  1.3703923225402832
```

Dense Graph:

```
4418->949->1071->4353->1437->3592->3827->3014->3778->3609->2857->1275->283->4993->1829->
3852->3336
Maximum Bandwidth with dijkstra:  9987
Time taken:  3.7282705307006836
```

```
4418->949->1071->4353->1437->581->4448->2357->2081->4268->267->1212->4021->627->1282->25
67->3611->1473->1040->4715->2201->2330->538->1030->4220->3715->1107->2041->50->3030->361
8->4139->3551->3330->3641->1717->1695->222->2599->1363->3609->2857->1275->283->4993->182
9->3852->3336
Maximum Bandwidth with dijkstra with heap:  9987
Time taken:  2.4568209648132324
```

```
4418->949->1071->4353->1437->581->4448->2357->2081->4268->267->1212->4021->627->1282->25
67->3611->1473->1040->4715->2201->2330->538->1030->4220->3715->1107->2041->573->3241->29
74->2934->4663->3240->3137->3551->3330->3641->1717->1695->222->2599->1363->3609->2857->1
275->283->4993->1829->3852->3336
Maximum Bandwidth with kruskal:  9987
Time taken:  244.76609659194946
```

Sparse Graph			**Dense Graph**		
Dijkstra's with Heap (s)	**Dijkstra's without Heap (s)**	**Kruskal's with Heap Sort (s)**	**Dijkstra's with Heap (s)**	**Dijkstra's without Heap (s)**	**Kruskal's with Heap Sort (s)**
0.015	0.079	1.404	3.636	6.700	270.03
0.239	2.022	1.645	1.489	4.597	252.84
0.156	1.271	1.370	2.456	3.728	244.76
0.140	0.927	1.355	1.551	3.890	246.07
0.338	2.801	1.333	3.453	6.371	264.26
0.055	2.424	1.255	2.726	3.945	250.50
0.013	1.497	1.285	3.161	3.933	235.29
0.132	0.714	1.301	2.712	4.647	230.34
0.232	1.296	1.253	1.157	1.516	231.04
0.124	0.805	1.350	2.945	5.235	230.60
0.125	2.282	1.322	0.699	2.999	229.00
0.158	1.201	1.358	1.395	2.538	234.79
0.090	0.446	1.283	1.921	4.828	238.01
0.171	1.55	1.317	1.660	3.104	247.67
0.0	1.839	1.432	3.218	6.003	235.95
0.203	1.709	1.269	1.188	5.501	236.86
0.187	0.659	1.253	2.611	4.629	235.63
0.312	2.305	1.286	3.3121	4.6904	235.71
0.097	0.652	0.826	2.965	0.254	235.63
0.220	1.758	1.282	1.6495	0.9195	238.49
0.203	0.0312	1.298	0.412	1.113	259.27

0.133	0.880	1.460	3.465	5.112	242.11
0.340	2.749	1.340	3.065	5.120	240.83
0.337	2.469	1.365	2.215	2.584	249.87
0.309	2.506	1.228	3.142	5.676	246.20

Conclusion

- From the table, we observed that the Dijkstra's with Heap outperforms both Dijkstra's without Heap and Kruskal's algorithm in both sparse and dense graphs. This is due to the Maxheap Structure which enables us to find max fringe in constant time.

- Kruskal's performance for sparse graph is good and on average takes less time than Dijkstra's without MaxHeap. This is because for sparse graph the number of edges is small and sorting them takes lesser amount of time. But for dense graph, Kruskal's performance degrades drastically, as the number of edges in a dense graph is roughly 2.5 million. And, as Heapsort takes O(mlogm) time, which determines the total complexity of the algorithm, is not trivial at all.

- Dijkstra's (both with and without Heap) performance depends on the number of edges between the source and terminal vertex, hence, we get erratic performances as the number of edges between the source and sink is chosen randomly. But for Kruskal's algorithm, the number of edges or how the source and terminal is connected does not affect its performance. Because, it has to sort all the edges anyway. Hence, for Kruskal's, we get consistence performance throughout for both types of graphs.

- **Summary of the results:**
  Time required by the algorithms for Sparse graphs
  Dijkstra's with heap < Kruskal's with heap sort < Dijkstra's without heap.

  Time required by the algorithms for Dense graphs
  Dijkstra's with heap < Dijkstra's without heap < Kruskal's with heap sort.

Future Improvement

We can try using Mergesort instead of Heapsort as Mergesort is faster than Heapsort for larger sets. But the drawback is that Mergesort requires twice the memory of heapsort because of the second array. Since, Merge sort will perform sequential reads from two runs and sequential writes to a single merged run, a merge sort is faster than a heapsort.