Practical construction of Maximum Bandwidth Paths

Shiva Kumar Pentyala
Texas A&M University
UIN: 127003995
shivakumar.pentyala@gmail.com

***Abstract*—** Network optimization has been an important area in the current research in computer science and computer engineering. Constructing maximum bandwidth paths has been a basic operation in the study of network routing, in particular in the recent study of network QoS routing. In the literature, it has been proposed that a maximum bandwidth path be constructed by a modified Dijkstra's algorithm or by a modified Kruskal's algorithm. In this project, I have implemented 2 types of Dijkstra Algorithm which are simple and heap based, 2 types of Kruskal Algorithm which are simple and path-compressed based. In addition, I have evaluated the performance of these algorithms on dense and sparse graphs and observed that Kruskal's approach is simpler, easier in implementation, more flexible, and faster than the Dijkstra algorithm.

Keywords— Kruskal's algorithm, Dijkstra's algorithm

I. Introduction

As the Internet evolves into a global communication infrastructure, there is a growing need to support more sophisticated service than the traditional "besteffort" service. In particular, Quality of Service (QoS) routing has recently received substantial attention in the context of its possible use in an integrated services IP network. It has been recognized that the establishment of an efficient QoS routing scheme poses several complex challenges. A QoS connection request in general consists of a source node s, a destination node t, and a set of QoS requirements. Each QoS requirement can be either a "bottleneck" constraint, typically the path bandwidth constraint, or an "additive" constraint, such as path delay, packet loss, or jitter. This is well known that constructing a routing path subject to more than one additive constraint is NP-hard. Therefore, many researchers have turned their attention to designing either heuristic algorithms or approximation algorithms for the QoS routing problem. In many proposed QoS routing algorithms, the most important subproblem is as below:

Maximum Bandwidth Path Problem: Given a source node s and a destination node t in a network G, in which each edge e is associated with a value w(e), construct a path from s to t in G whose bandwidth is maximized (the bandwidth of a path is equal to the minimum edge bandwidth over all edges in the path).

The MAX-BANDWIDTH PATH problem is not new. This have been studied for more than 40 years. Modification of Dijkstra [Corman et al, 2009] and Kruskal [Malpani et al, 2002] algorithms have been used for this problem for several years. In this course project I have been implemented these algorithm in Java. At first I will illustrate about implementation methods I have used for constructing random graphs. Then, I will explain about Heap structure and my implementation. Section 4 is about Dijkstra and Kruskal algorithms and their modifications and their implementation details. At last, there is a result section that will show the results of my Java implementation.

II. Random Graph Generation

Networks can be shown with graph in which every node in networks are vertices in graphs and every links in networks are edges in graphs. I have implemented 2 types of random undirected graphs with 5000 vertices. To make sure that our graphs are connected, I have started with a cycle that contains all vertices of the graphs, then added rest of the edges randomly. Positive weights to edges are assigned randomly.

• Sparse Graph(G1): In this type of graph G1, the average vertex degree is 8. In general, average vertex degree is 2|E|/|V| where E are number of Edges and V are number of Vertices in the Graph. So our Graph has 4|V| edges (Equal to 20000) and all neighbors of a vertex are different and pairs of vertices that are adjacent are chosen randomly. Below figure shows the source code of my sparse graph implementation.

```
//vertexcounter to note down the number of neighbors
for(int i=1;i<=vertex;i++)
   vertexcounter[i]=0;
int counter=0;
//start with a cyclic graph that contains all vertices of the graph
for(int i=1;i<=vertex;i++){
 int w2=rand()%100+1;
 if (i==5000){
     addedge(1,5000,w2);
     graph[1][5000]=1;
     graph[5000][1]=1;
 }
 else{
     addedge(i,i+1,w2);
     graph[i][i+1]=1;
     graph[i+1][i]=1;
 }
 counter++;
}
//here counter=5000 as we made graph cyclic above
//Total 4|V| edges, but already added 5000, add rest 20,000-5,000=15,000
int random1,random2;
while(counter<(vertex*deg)/2)
{
   random1=rand()%vertex+1;
   random2=rand()%vertex+1;
  if((vertexcounter[random1]<8)&&(vertexcounter[random2]<8)&&(random1!=random2)&&(graph[random1][random2]==0))
    {
        int w= rand()%100+1;
        addedge(random1,random2,w);
        graph[random1][random2]=1;
```

• Dense graph(G2): In this type of graph G2, each vertex is adjacent to about 20% of the other vertices. As each vertex needs to be adjacent to about 20% of remaining vertices, so every vertex need not have a degree of 1000. Hence I pick each pair of vertices, and calculate a random number between 1 and 100. If the random number is below 20, I kept the edge else discarded it. This can also be interpreted as connecting them by an edge with a probability 20%. Below figure shows the source code of my dense graph implementation.

```
//Initialization
 for(int i=1;i<=vertex;i++)
   for(int j=1;j<=vertex;j++)
       graph[i][j]=0;
//start with a cyclic graph that contains all vertices of the graph
for(int i=1;i<=vertex;i++){
 int w2=rand()%100+1;
 if (i==5000){
     addedge(1,5000,w2);
     graph[1][5000]=1;
     graph[5000][1]=1;
 }
 else{
     addedge(i,i+1,w2);
     graph[i][i+1]=1;
     graph[i+1][i]=1;
 }
}

for(int i=1;i<=vertex;i++){
   for(int j=1;j<=vertex;j++){
       if((i!=j) && (graph[i][j]==0)){
           int r=rand()%100+1;
               if(r<20){
                   seed();
                   int w1= rand()%100+1;
                   addedge(i,j,w1);
                   graph[i][j]=1;
                   graph[j][i]=1;
           }
```

III. HEAP STRUCTURE

I have implemented Max Heap which is used in Dijkstra algorithm and Heap Sort for Kruskal algorithm:

• Max Heap: In my Dijkstra implementation, I used this heap. One of the tricky part for using this heap is whenever Dijkstra want to change the capacity of its fringes we should have the place of it. If we do not have every vertices place, it would take O(n) to find every vertices. Therefore an array named heap1 was used to store the vertices and another array named heap2 was used to store the values for these vertices.

• Heap Sort: For constructing Maximum bandwidth path based on Kruskal in the sorting part, I used this.

Heap structure algorithms are:

Algorithm 1 Max Heapfy

```
1: procedure MAX HEAPFY(A,i)
2:     l ← LEFT(i)
3:     r ← RIGHT(i)
4:     if l ≤ A.heap_size and A[l] > A[i] then
5:         laregest ← l
6:     else largest ← i
7:     if r ≤ A.heap_size and A[r] > A[largest] then
8:         laregest ← r
9:     if largest ≠ i then
10:        exchange A[i]with A[largest].
11:        MAX HEAPFY(A,largest).
```

Algorithm 2 BUILD MAX HEAP

```
1: procedure BUILD MAX HEAP(A)
2:     A.heap_size ← A.length
3:     for i ← [A.lenght/2] to 1 do
4:         MAX HEAPFY(A,i).
```

Algorithm 3 HEAP SORT

```
1: procedure HEAP SORT(A)
2:     BUILD MAX HEAP(A).
3:     for i ← A.lenght to 2 do
4:         exchange A[1] with A[i].
5:         A.heap_size ← A.heap_size − 1
6:         MAX HEAPFY(A,1).
```

Algorithm 4 INSERT

```
1: procedure INSERT(A,key)
2:     A.heap_size ← A.heap_size + 1
3:     A[A.heap_size]key
4:     MAX HEAPFY(A,A.heap_size).
```

Algorithm 5 HEAP MAX

```
1: procedure HEAP MAX(A)
2:     return A[1]
```

IV. DIJKSTRA AND KRUSKAL ALGORITHMS

I have implemented modification of Dijkstra and Kruskal Algorithms. In this section, I would explain these 2 algorithms and their implementation tricks:

• Dijkstra: Dijkstra is one of the most popular algorithm for finding shortest path. I have implemented 2 versions of this algorithm. These two algorithms different in their implementation part.

a) Simple(without heap): Simple Dijkstra refer to implementation of Dijkstra without using a heap structure. In this version, time complexity is equal to $O(n^2)$. The graph was implemented as an adjacency list for faster processing. An array called set was used to see which vertices were added and which vertices were not. A function called maxdistance was used to find the vertex with largest capacity among the vertices which are not visited. Once the vertex u is found, then the adjacency list of u contains the fringes. The capacity for each vertex is calculated and array Dist is used to store the capacity of the vertices from the source. Dist[final] gives us the max band width from the source to the final vertex.

b) Using Heap: Heap Dijkstra refer to implementation based on Max Heap which I explained in the previous section. Because of using better structure for fringes which is Max heap, this algorithm's complexity would be O(m log(n)).Here instead of storing the capacity in an array, we store it in a heap. heap1 contains the vertices names and heap2[heap1[i]] will give the weight of the vertex present at heap1[i]. The vertex with the max capacity at every iteration is selected using extract max function of the heap. Once the vertex with highest capacity is found, the adjacency list of that vertex is checked.

Modified Dijkstra's algorithm for MAX-BANDWIDTH PATH is shown as below.

Algorithm 6 DIJKSTRA

```
1: procedure DIJKSTRA(G,s,t)
2:     for for each node v in G do
3:         P[v] ← 0.
4:         B[v] ← −∞.
5:     B[s] ← 0
6:     F ← ∅
7:     for for each neighbor w of s do
8:         P[w] ← s.
9:         B[w] ← weight(s, w).
10:        add w to F.
11:    repeat
12:        remove the node u of maximum B[u] from F
13:        for each neighbor w of u do
14:            if B[w] == −∞ then
15:                P[w] ← u.
16:                B[w] ← minB[u], weight(u, w).
17:                add w to F.
18:            else if (w is in F) & (B[w] < minB[u], weight(u, w)) then
19:                P[w] ← u.
20:                B[w] ← minB[u], weight(u, w).
21:    until (B[t] ≠ −∞) & (t is not in F)
```

• Kruskal: This algorithm is useful for finding maximum spanning tree in a given graph G. Finally, once the maximum spanning tree T is constructed, the unique path from s to t in the tree T can be easily constructed in linear time by using BFS or DFS. Implementation of Kruskal is based on Union, Find and Make Set Operations. There is two version of these Operations which are: A modification of kruskal algorithm is used to find the max capacity. The union, find and makeset functions are implemented. The changes that we do to the kruskal is that we arrange the edges in the decreasing order. We use an array to a structure called edge. Edge contains the following entities: from, to, weight and edgeindex. For each edge, we give these values. Heap sort is used to arrange the edges in the decreasing order. Once we get the Max spanning tree. Breadth First search is used to find the max band width between two vertices in the tree. The array mstpath is used to store the final tree structure and BFS is included to construct the actual path from the source node to the destination node after the maximum spanning tree is constructed.

There is two version of these Operations which are:

a) *simple Rank based*: In these implementation, we are just using two array of Rank and dad. Rank would show the height of each node.

b) *Path compression based*: This is a path compression approach which reduces total Find operations time.

Implementation of these Operations are shown in following:

Algorithm 7 MAKE SET

```
1: procedure MAKE SET(v)
2:     Dad[v] ← 0.
3:     Rank[w] ← 0.
```

Algorithm 8 FIND RANK

```
1: procedure FIND RANK(v)
2:     w ← v.
3:     while Dad[w] ≠ 0 do
4:         w ← Dad[w].
       return w
```

Algorithm 9 FIND COMPRESSION

```
1: procedure FIND COMPRESSION(v)
2:     w ← v.
3:     let q be a queue.
4:     while Dad[w] ≠ 0 do
5:         q ← w.
6:         w ← Dad[w].
7:     while q ≠ ∅ do
8:         y ← q.
9:         Dad[y] ← w.
       return w
```

Algorithm 10 UNION

```
1: procedure UNION(r1,r2)
2:     if Rank[r1] > Rank[r2] then
3:         Dad[r2] ← r1.
4:     else if Rank[r2] > Rank[r1] then
5:         Dad[r1] ← r2.
6:     else
7:         Dad[r1] ← r2.
8:         Rank[r2] + +.
```

General Kruskal algorithm based on these operations would be O(m log(n)) and its algorithm is shown in following:

Algorithm 11 MAX BANDWIDTH KRUSKAL

```
1: procedure MAX BANDWIDTH KRUSKAL(G)
2:     sort the links of G in terms of link bandwidth in non-increasing order: {e1, e2, ...., em}.
3:     for each node v of G do
4:         MAKE SET(v).
5:     T ← ∅.
6:     for i = 1 to m do
7:         let ei = [ui, vi].
8:         r1 ← FIND(ui)
9:         r2 ← FIND(vi)
10:        if r1 ≠ r2 then
11:            add ei to T
12:            UNION(r1,r2)
```

Here each set is given by a Union-Find tree such that a sequence of m MakeASet, Union, and Find operations takes time O(mlog∗ n), where log∗ n ≤ 6 for all practical numbers n. Therefore, the time complexity of Kruskal's algorithm is t (m) + O(mlog∗ n), where t (m) = O(mlogn) is the time of step 1 for sorting m elements. Based on a Dijkstra's style algorithm, with a subtle data structure and analysis, the maximum spanning tree problem can be solved in time O(m + n logn). Further investigation actually shows that the maximum spanning tree problem can be solved in O(mlogβ(n, m)) time, where β(n, m) = min{i | $\log^i n \le m/n$} ≤ log∗ n. Finally, once the maximum spanning tree T is constructed, the unique path from s to t in the tree T can be easily constructed in linear time O(n). Therefore, the maximum spanning tree problem can be solved at least as efficiently as the Dijkstra's algorithm. Moreover, since logβ(n, m) ≤ 3 for all practical values n, and m = O(n) for most practical applications. Hence the maximum spanning tree problem can be solved more efficiently (in time O(mlogβ(n, m))) than the best implementation of the Dijkstra's algorithm (in time O(m + n logn)) in network applications. Although the best theoretical bounds for Dijkstra's algorithm and the algorithms for the maximum spanning tree problem are better than O(mlogn), these best algorithms are difficult to understand and the implementations of these best algorithms are in general subtle and involved.

V. Advantages of kruskal over Dijkstra

• Kruskal's algorithm is simpler. From the view of programming, Kruskal's algorithm requires a much shorter program than that for Dijkstra's algorithm. From the view of data structure, Kruskal's algorithm uses two simple arrays (a "dad" array and a "rank" array) to implement the Union-Find trees, while Dijkstra's algorithm requires at least three arrays (for the "dad" relations, "weights", and "status") plus a priority queue.

• Kruskal's algorithm has time complexity O(mlog∗ n) plus the time t (m) for sorting the m edges. Since there are many very well-studied sorting algorithms, which have been either written as standard software packages or even coded in hardware, the time t (m) is mlogn times a very small constant. On the other hand, for a simple priority queue used in Dijkstra's algorithm, each of the minimum, insertion, and deletion operations takes time c log n with a relatively large constant c. So, intuitively we can see that Kruskal's algorithm can be expected to be several times faster than Dijkstra's algorithm.

• Since in many practical cases, sorting can be done in linear time, this makes Kruskal's algorithm run in time O(mlog∗ n), which is essentially linear. This approach we have seen in the class under approach-3 to solve Maximum Bandwidth problems.

• Maximum spanning tree is not particularly with respect to any referred source node and destination node, therefore, a maximum spanning tree actually provides a maximum bandwidth path for any pair of source node and destination node. On the other hand, Dijkstra's algorithm only provides maximum bandwidth paths from a fixed source node to other nodes.

VI. RESULTS

I have tested my four algorithms on the 5 pair of randomly generated graphs. For each generated graph, 5 pairs of randomly selected source-destination vertices (s,t) are picked, For each source-destination pair (s, t) on a graph G, testing of all these algorithms is done and recorded the running time. The programs were run in Intel(R) Core(TM) i7 Processor @ 2.50GHz with 8 GB ram and observed the following.

Algorithm	Average time (s) for Dense Graph	Average time (s) for Sparse Graph
Simple Dijkstra	0.6459	0.1079
Heap Dijkstra	0.3947	0.061
Simple Kruskal	0.212	0.052
Compression Kruskal	0.198	0.053

Observatons:

• Running time of dense graph is a lot more than running time of sparse graph.

• Running time of heap Dijkstra is always better than simple Dijkstra.

•Running time of Kruskal is better than Dijkstra.

• Running time of compression Kruskal is better than simple Kruskal in dense graph and sparse graph.

VII. Analysis

The experimental result exactly matches the theoretical results. Theoretically modified Dijkstra without heap runs in $0(n^2)$. Dijkstra with Heap and modified kruskal runs in O(mlogn) time. The constant value in the complexity expression of modified kruskal is lesser in value as compared to modified dijkstra with heap. Theoretically Kruskal should be the fastest followed by modified dijkstra with heap and then modified dijkstra without heap. Practically, from the above table we can see that the modified kruskal is the fastest followed by the modified dijkstra with heap implementation for the maximum bandwidth path. Modified dijkstra without using a heap is the slowest among all three. Hence the theoretical results matches the practical implementation.
Dijkstra without heap runs in $O(n^2)$. The step where we pick the fringe with the largest capacity runs in O(n) time. Going through the neighbors of the vertex and then calculating the max capacity takes O(n) when we do not put it as a heap but it takes O(logn) if we put it as a heap. Theoretically, these algorithms should run on faster on sparse graph than in dense graph due to the lesser number of edges. Practically, we see from the table that the sparse graph implementation is faster than dense graph implementation as the m value is lesser for the sparse graphs. Hence these algorithms run faster on a sparse graph than in a dense graph. Hence we can see that the theoretical results matches the experimental results

VIII. Further Research

I think the following points would hold in some scenarios.
• In heap Dijkstra implementation, I have used an additional array for knowing the place of each vertex, which is unfair to compare it to Kruskal in my opinion. So, one can construct heap dijkstra without that additional information.
• In Kruskal implementation, after finding maximum spanning tree, for constructing best path between s and t, I used BFS, which would be linear time. However, one can construct spanning tree by storing additional information and doing that in O(1) like heap Dijkstra that I mentioned.
• There is median Maximum bandwidth algorithm with linear running time which would be much better than Dijkstra and Kruskal. However, implementing that would need much more time for us.

Acknowledgment

Deepest gratitude would like to be delivered for Dr. Jianer Chen and Qin Huang for giving valuable advices on Piazza which helped alot to make this work possible.

References

[1] Ellison, Nicole B. "Social network sites: Definition, history, and scholarship." Journal of ComputerMediated Communication 13.1 (2007): 210-230.

[2] Malpani, Navneet, and Jianer Chen. "A note on practical construction of maximum bandwidth paths." Information Processing Letters 83.3 (2002): 175-180.

[3] Cormen, Thomas H. Introduction to algorithms. MIT press, 2009.

[4] Olifer, Natalia. Computer networks. John Wiley & Sons, 2005.

APPENDIX

```
the graph generation is done!
Start the test:
1 graph pair:
Algorithm 1 on G1 for vertices (12,300) is 112 ms
Algorithm 2 on G1 for vertices (12,300) is 68 ms
Algorithm 3 on G1 for vertices (12,300) is 48 ms

Algorithm 1 on G2 for vertices (12,300) is 965 ms
Algorithm 2 on G2 for vertices (12,300) is 573 ms
Algorithm 3 on G2 for vertices (12,300) is 204 ms

2 graph pair:

Algorithm 1 on G1 for vertices (689,2690) is 142 ms
Algorithm 2 on G1 for vertices (689,2690) is 76 ms
Algorithm 3 on G1 for vertices (689,2690) is 56 ms

Algorithm 1 on G2 for vertices (689,2690) is 1026 ms
Algorithm 2 on G2 for vertices (689,2690) is 652 ms
Algorithm 3 on G2 for vertices (689,2690) is 451 ms
```

```
3 graph pair:

Algorithm 1 on G1 for vertices (2412,56) is 104 ms
Algorithm 2 on G1 for vertices (2412,56) is 39 ms
Algorithm 3 on G1 for vertices (2412,56) is 41 ms

Algorithm 1 on G2 for vertices (2412,56) is 602 ms
Algorithm 2 on G2 for vertices (2412,56) is 395 ms
Algorithm 3 on G2 for vertices (2412,56) is 142 ms

4 graph pair:

Algorithm 1 on G1 for vertices (3688,4126) is 99 ms
Algorithm 2 on G1 for vertices (3688,4126) is 36 ms
Algorithm 3 on G1 for vertices (3688,4126) is 29 ms

Algorithm 1 on G2 for vertices (3688,4126) is 652 ms
Algorithm 2 on G2 for vertices (3688,4126) is 241 ms
Algorithm 3 on G2 for vertices (3688,4126) is 169 ms
```

```
4 graph pair:

Algorithm 1 on G1 for vertices (3688,4126) is 99 ms
Algorithm 2 on G1 for vertices (3688,4126) is 36 ms
Algorithm 3 on G1 for vertices (3688,4126) is 29 ms

Algorithm 1 on G2 for vertices (3688,4126) is 652 ms
Algorithm 2 on G2 for vertices (3688,4126) is 241 ms
Algorithm 3 on G2 for vertices (3688,4126) is 169 ms

5 graph pair:

Algorithm 1 on G1 for vertices (91,1101) is 112 ms
Algorithm 2 on G1 for vertices (91,1101) is 68 ms
Algorithm 3 on G1 for vertices (91,1101) is 51 ms

Algorithm 1 on G2 for vertices (91,1101) is 421 ms
Algorithm 2 on G2 for vertices (91,1101) is 264 ms
Algorithm 3 on G2 for vertices (91,1101) is 121 ms
**********************************************
```