CSCE 629

Project Report

FALL 2015

By:

Divyesh M Tekale

NetID: tekale2

UIN: 923004428

M.S Computer Engineering

1. INDROCUTION

Many challenging problems exist today in Network Optimization techniques. Maximum Bandwidth/Maximum flow is one such interesting problem. For a given network Graph, the Edges represent the capacity of the links in between the nodes. For a communication on a path from a source vertex to destination vertex, the total link capacity is limited by the edge with the minimum link capacity. Hence the goal of maximum bandwidth path is to find an optimal path between the source vertex and the destination vertex, such that the total bandwidth of that path is maximized.

Hence, in this project I implemented the maximum Bandwidth Path Algorithm by modifying three of the well-known algorithms: Dijkstra's, Dijkstra's with heaps and Kruskal's algorithm. The whole development phase went as follows:

1. Implement shortest path with the aforementioned algorithms.
2. Validate the correctness with small generated graph with help of sample graph [4].
3. Modify the existing algorithms for calculating maximum bandwidth path.
4. Validate the correctness with small generated graph.
5. Implement Random Graph generation and test it.
6. Run and obtain results.

The rest of the report is organized as follows: First I discuss the details of my implementation of the above mentioned algorithms. Then I discuss how I validate my algorithms, with standardized and non-standardized methods (solving the maximum bandwidth path problem by hand and validate it against the output of my algorithms). Then finally, I discuss how performed the testing to generate results. The whole project is developed in C++ with some C++11 standards for the used STL containers in the program.

2. IMPLEMENTATION

The first and foremost decision to make was the choice of data structure to use for holding the graph. Adjacency matrix uses O (n*n) memory, and has O (1) lookup if it needs to check the existence of an edge between the given two vertices. On the other hand Adjacency List uses memory proportional to number of edges in the graph. Adjacency list is better when listing out vertices adjacent to a given vertex. Since both the Dijsktra algorithms requires looking up adjacent edges; adjaceny list data structure was chosen for this purpose. Kruskal required sorting of edges. Hence for Kruskal's algorithm I used a vector of structures to hold the values of edge weights and the connected pair of vertices. Finally to get the shortest path from the generated MST, I had to use adjacency matrix for marking the edges that are in the MST and the use DFS to find the path from s to t. In the following subsections, I will be discussing about my implementation of each of the mentioned algorithms.

2.1: *Dijkstra without out heaps*: Dijkstra without the use of heaps was pretty straight forward implementation from the pseudo code given in the class:

Dijkstra(G,s,t):

1. For v=1 to n
   status[v] = UNSEEN

cap[v] = 0
parent[v] = v

2. status[s] = IN_TREE
3. For each edge w adjacent to s:
   status[w] = FRINGE
   parent [w] = s
   cap[w] = weight[s,w]
4. While there are fringes to process //implemented using in_tree counter
   v = max_capacity_fringe() //O(V) implementation
   status[v] = IN_TREE
   for each edge w adjacent to v do:
      if status[w] == UNSEEN then do:
         status[w] = FRINGE
         dad[w] =v
         cap[w] = min{cap[v],weight[v,w]}
      else if status[w] == FRINGE and cap[w] < min{cap[v],weight[v,w]}
         dad[w] = v;
         cap[w] = min{cap[v],weight[v,w]}

By analysis we can see that outer loop runs V time while finding maximum capacity edge takes O(V) time looking up into the cap[] array. Hence the whole algorithm runs in $O(V^2)$ time.

2.2: *Dijkstra with max heap*: Dijkstra with max heap was almost the same implementation as of normal Dijkstra's algorithm. Vector (vector<unsigned int>) was used for maintain the queue because it already had pop_front, push_back, pop_back functions implemented. Moreover a position array has been used to keep track of positions of the vertices in the Queue. Finally Extract_Max (Remove the vertex with maximum weight from the top), Increase_Key(bubble up the vertex to correct parent position), and heapify(bubble the current vertex down the heap to correct child position) were needed to be implemented. The algorithm is as follows:

Heap_Dijkstra(G,s,t):

1. For v=1 to n
   status[v] = UNSEEN
   cap[v] = 0
   parent[v] = v
   add v to Q if v!=s
   update_position()
2. status[s] = IN_TREE
3. For each edge w adjacent to s:
   status[w] = FRINGE
   parent [w] = s
   cap[w] = weight[s,w]
   increase_key(cap, position[w])
4. While there are fringes to process //implemented using in_tree counter
   v = Extract_max(cap[])//log(V) implemntation
   status[v] = IN_TREE

```
for each edge w adjacent to v do:
        if status[w] == UNSEEN then do:
                status[w] = FRINGE
                dad[w] =v
                cap[w] = min{cap[v],weight[v,w]}
                increase_key(cap, position[w])
        else if status[w] == FRINGE and cap[w] < min{cap[v],weight[v,w]}
                dad[w] = v;
                cap[w] = min{cap[v],weight[v,w]}
                heapify(cap, position[w])
```

Heapify(cap[], curr): //O(log(n))

1. largest =curr
2. while(curr<Q.size) do:

```
left = 2*curr
right = 2*curr+1
if cap[Q[curr]] < cap[Q[left]]  do:
        largest = left
if cap[Q[largest]] < cap[Q[right]]  do:
        largest = right
if largest != curr
        swap (Q[curr],Q[largest])
        update_position() //update the positions in the position array in O(1)
        curr = largest
```

Extract_max(cap[]):://O(log(n))

1. max = Q[0]
2. Q[0] = Q.back()
3. update_pos()
4. Q.pop_back()
5. Heapify(cap , 0)

Increase_Key(cap[], curr): //O(log(n))

1. parent = curr/2
2. While cap[Q[parent]] < cap[Q[curr]] do:

```
swap( Q[parent], Q[curr] )
update_position()
curr = parent
parent = curr/2
```

From HW 2 we know that heaps take atmost log n steps. Hence all the heap operations run in log(n) time. The outer loop runs for number of vertices V and inner loop runner for E steps. Hence It takes O((E+V) log(v)) which is O(E log(V)).

2.3: ***Kruskal's Algorithm***: Kruskal's algorithm needed setup. During the graph generation phase, build_heap is called to build the heap of edges. Hence when actual Kruskal is called, Heapsort can directly be called. Since Kruskal is independent of starting/ending vertex, Kruskal is called only once to create a MST, and then DFS is called for every source to get the path.

Kruskal(G):

1. HeapSort(G->Kruskal_list)
2. For vertex v=1 to n do;
   make_set(v)
3. For i in range (1,m) do:
   $e_i \rightarrow [v_i, w_i]$
   if $(r1 = Find(v_i)) \neq (r2 = Find(w_i))$ do:
   in_mst[v_i][w_i] = true
   in_mst[w_i][v_i] = true
   Union (r1,r2)
4. Return

Union-Find Algorithm is pretty basic, primitive and was provided in class, also min_heapify is a basic primitive algorithm. Hence skipping the pseudo code for those.

Make_Heap():

1. For i ranging (arr_size-1)/2 down to 1 do:
   Min_heapify(arr,i, arr_size-1)

Heap_sort():

1. For i ranging arr_size-1 down to 1 do:
   Swap(arr[0],arr[i])
   Heapify(arr, 0, i-1)

Finally, using the in_mst matrix, we ran DFS to find the path from s to t.

DFS(in_mst[] [],v)

1. init(parent[],visited[])
2. visited[v] =true
3. for all vertices w adjacent to v do:
   if visited[w] == false
   parent[w] = v
   DFS(in_mst[] [], w)

As discussed in class heapify takes log(E) steps, heapsort takes E*log(E) steps. Also the union and makeset each take constat time, while find operation takes log(n) steps with implemented with rank. The outer loop runs over all the edges hence the total time it takes is O(E Log V) from [5]. The DFS runs in linear O(E+V) time since it iterates over all the edges and all the vertices.

2.4 ***Graph_Generation:*** The random Graph generation was pretty straight forward implementation. For each vertex in the array, generate a unique random vertex of given degree. Assign random weight>0 to all the edges. Also a path from start vertex to all the other vertices is added so that the whole graph is a single connected component.

For debugging purposes graph generation from file was also made. The first line of the file contained the number of edges in the graph and then the next lines contained the two vertices and the weight of each edge.

3. VALIDATION

For the purpose of verifying correctness, validation was done using known sparse graph of small size as shown in the following figure:

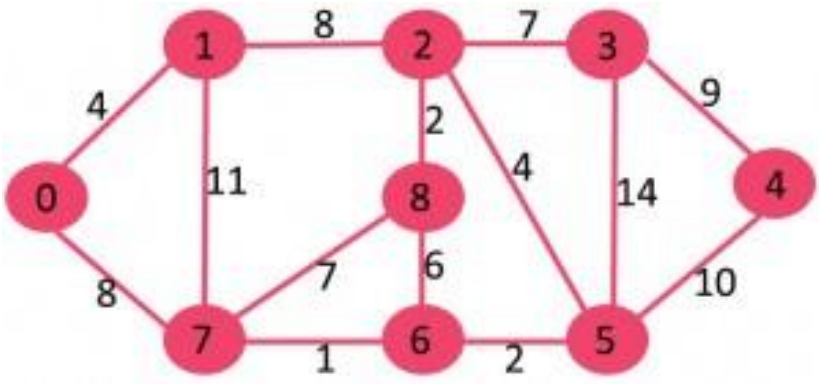

Fig 1 (adapted from geekforgeeks.com)

So for the given graph maximum bandwidth path from vertex 0 to 8 is 0 -> 7 -> 8. Hence running the algorithm against this graph on the above algorithms we confirm the correctness of algorithm.

Also during the testing phase a unique peculiarity was found. Suppose we find maximum bandwidth from 0 to 4. The Kruskal algorithm outputs 0->7->1->2->3->5->4. While the dijkstra's output 0->7->1->2->3->4. After careful inspection we find that both the paths are correct since the minimum bandwidth is limited by edge 2->3 with capacity 7. Hence with further such test I validated all my algorithms for correctness.

4. RESULTS

Finally, step counting was added at each critical place to calculate the number of required algorithmic steps. Although in big O notation the lower order polynomials are dropped, the step counter I added counted each and every critical step. To check the correctness, I calculated the system time it took to run both the Algorithms. Hence after running for 10 such runs and averaging them the plots are shown in the fig 2 and fig 3. From the graph we can see peculiar behavior for Kruskal's algorithm. In dense graph the Krukal's algorithm reaches almost $V^2 \log V$ (where V=5000 in this case) since number of edges approaches near V^2 number of steps while Dijkstra's still V^2. Hence we observe the same pattern in the graph below, we can see that run time of Kruskal is almost double of that of normal Dijkstra's algorithm. The results were obtained on a core2duo machine running Ubuntu 15.10.

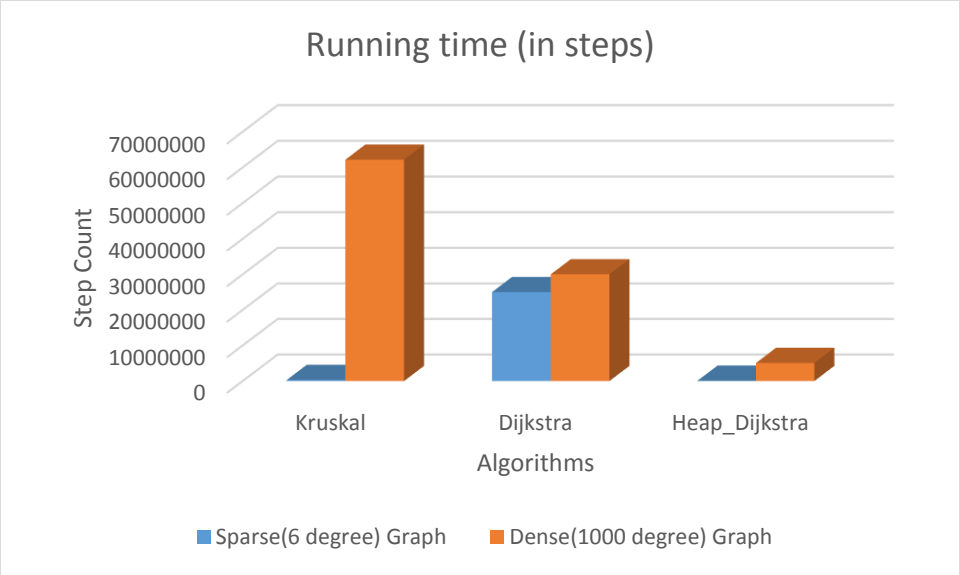

Fig2: Number of counted steps, averaged across 10 test cases

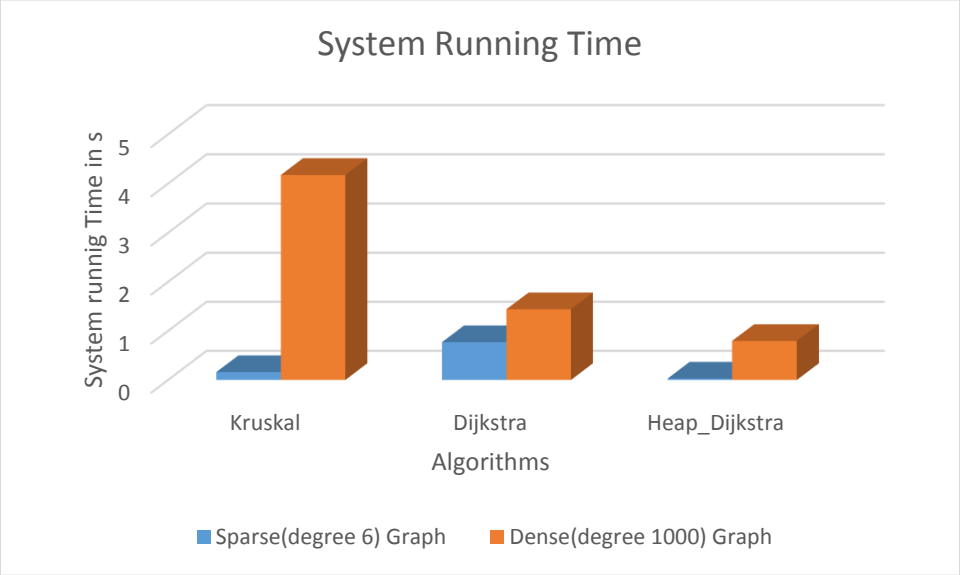

Fig3: System running time, averaged across 10 test cases

5. CONCLUSION

As assigned, I implemented, validated and tested the Max_bandwidth path algorithm, and produced results. Many improvements can be made in terms of space complexity. Use of list container instead of vectors where O(1) lookup did'nt matter. Second improvement can be made based on number of hops, we can run the dijkstra's algorithm switching s,t so that minimum bandwidth path is obtained, with minimum number of hops between the source and the destination.

REFERENCES

1. http://mat.gsia.cmu.edu/classes/QUANT/NOTES/chap11.pdf
2. https://en.wikipedia.org/wiki/Flow_network
3. CSCE 629 Course Notes.
4. https://www.cs.usfca.edu/~galles/visualization/Dijkstra.html
5. https://en.wikipedia.org/wiki/Kruskal%27s_algorithm
6. https://www.geekforgeeks.com