Texas A&M University

CSCE 629 **Analysis of Algorithms**

Course Project

Author: **S Venkata Satish Kumar Pasumarthi**

UIN: 726006526

Prof. Jianer Chen

Nov 30th 2017

Sections Page Number

1. Abstract:

In this project, we were asked to analyse the performance of maximum Bandwidth Path algorithms we studied in the class namely Dijkstra and Kruskal. I have implemented two variants of Dijkstra i.e., one without any data structure for storing fringes and other with Heap for storing the fringes. For Kruskal, I have implemented the Path compression version of Union-Find. I have tested all these 3 variants of algorithms on two kinds of graphs i.e., sparse and dense.

2. Introduction:

The maximum bandwidth problem finds its applications in various domains such as pathway analysis and computation of maximum flows. Here I present the problem statement for this project and its terminology.

Problem statement: Given a source node ‘s’ and a destination node ‘t’ in a network G, in which each edge is associated with a weight (w) termed as bandwidth, construct a maximum bandwidth path from s to t in G.
Bandwidth of a path is equal to the minimum edge bandwidth (weight) along the path.

I chose C++ as my language of implementation for this project since it is runtime efficient compared with many other high level languages like Python, Java etc. I will discuss the implementation details as well as the run time comparisons for these algorithms in detail in the following sub-sections.

Nomenclature:
In this project, I had used the following notations for vertices whose definition is explained below:

Vertices Notations:
UNSEEN : Any vertex not visited by the algorithm
INTREE : Vertex visited but bandwidth calculation at the vertex is not finalized
FRINGE : Vertex visited and bandwidth calculation at the vertex is completed.

Color Notations (for DFS):
WHITE : Vertex on which DFS has not run yet.
GREY : Vertex on which DFS has started but not yet completed.
BLACK : Vertex on which DFS has completed.

3. Graph Implementation:

For comparison between algorithms A and B, we cannot just simply conclude that A is better than B or vice-versa. The performance of the network routing algorithms depends on the type of the graph the algorithm it is dealing with. So, we were asked to analyse the performance of 3 different algorithms on two variants of graphs namely sparse and dense graphs. These graphs are all generated randomly; let us have a look at the graph definitions and their implementation now.

The following conditions are met while creating graphs:

i. Number of vertices are 5000.
ii. The graph should be undirected.
iii. The graph should be Simple i.e., multiple edges and loops are not allowed.
iv. The graph should be connected.
v. The graph should be random.
vi. The edge weights are chosen in between 1-10000 randomly.

Handling the above requirements:

Since (i) is a very straight forward thing, I will discussing how I handled the conditions (ii) to (vi) in the project. I am using list of pairs to store my graph. A Pair is nothing but a representation of vertex, weight.

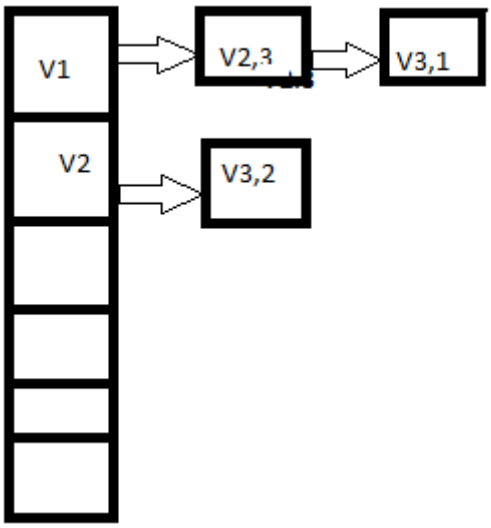

Fig 1: Graph Representation

Undirected:

While adding edge with bandwidth (wt) between two vertices (u,v), in the list I add the pair (v,wt) to u as well as the pair (u,w) to v.

Simple Graph:

Loop detection: While adding edges to my graph, I make sure that the vertices are not same (u != v)

Multiple edge detection: This is one of the hurdles we have to deal with while creating graphs. To handle this situation, I am creating a unique hash value out of the vertex pair using unordered pairing function as below:

$<x,y> = x*y+trunc((|x-y|-1)^2/4) = <y,x>$

While adding an edge I generate the hash value as shown above and stored it in a map. Every time I add an edge, I check if the hash value exists in the map, if it doesn't exist, if it exists

then I will skip adding the edge which implies the vertex pair has already an edge connecting them.

Picking edge weights:

I am using random number generator (rand()) which generates the random numbers and I modulo this with 10000 and add 1 to make sure my edge weights are lying between 1 to 10000. Since, I am using a random generator, the edge weights are randomized.

The connectivity and randomness are explained in the below sections.

a) **Sparse Graph:** We were asked to generate a sparse graph (SG) in which the average vertex degree is 8. We know that average degree of a vertex is given by 2*|E| / |V| . So, the number of edges for the graph turns out to be 20000. This graph generation is done in two steps.

Step 01: In order to achieve (iii) I am traversing all the vertices in pairs (1,2) (3,4) …(5000,1) and adding an edge between them with random weight (bandwidth) generated by rand() as explained above. This step adds 5000 edges to the graph.

Step 02: Since we have to add the rest 15000 edges, I randomly generate 15000 pairs of random numbers for vertices and then randomly pick a weight which is again generated randomly in the range 1-10000.

This completes the generation of sparse graph which has 20000 edges which has the average degree as exactly 8.

b) **Dense Graph**: The generation of dense graph is not that straight forward since we don't have the exact degree of the vertex nor the edge count. Instead we have the probability of vertex connecting to other vertices. It is given to us that each vertex is adjacent to about 20% of the other randomly chosen vertices. I tackled this problem by using Erdős–Rényi model which deal with probability of creating an edge between two vertices in a graph given by P_E. This graph generation is done in two steps.

Step 01: This step is same as the step 01 in sparse graph generation.In order to achieve (iii) I am traversing all the vertices in pairs (1,2) (3,4) …(5000,1) and adding an edge between them with random weight (bandwidth) generated by rand() function in C++. This step adds 5000 edges to the graph.

Step 02: In this step, I traverse through all the possible pairs of the vertices and for each case I generate a random number in the range of 0-1 and compare it with the probability of edge selection (P_{ES}), whose definition is explained in the next few lines.

For Erdős-Rényi graphs, the expected probability of creating an edge is |E|/C(|V|,2) which is nothing but the number of edges in graph divided by the total number of possible edges(possibilities).Calculating this probability (call it P_E) allows us to deterministically visit all pairs of edges.

Probability = Expected Outcomes/Possibilities.

Rule of Thumb: If the number of expected outcomes remain same; doubling the number of possibilities would halve the probability.

Since the vertex is connected to 20% of other vertices i.e., a vertex is connected to 1000 vertices roughly. So, the total possible edges with this condition is |E| = 5000*1000/2 = 2.5million. However, the number of possible edges in a fully connected graph is C(|V|,2) which comes out ~ 12.5million. So calculating the probability using the above formula gives us a probability of 0.2.

P_E = Probability of creating an edge between two vertices.

P_{ES} (Edge Selection Probability) = Probability of selecting the edge between two vertices.

Relation between P_{ES} and P_E:

I am running two for loops each iterating from 1 to 5000 and generating a random number (say r) and comparing it with the probability of having an edge between the vertices. But here we cannot use P_E directly as we have increased our possibilities to 25million even though the maximum possible edges are 12.5million. Since the number of desired edges is same (since the condition that each vertex is connected to 20% of other vertices hasn't changed) I had to use the probability P_{ES} , (Probability of Edge selection) instead of the default P_E which is $0.5*P_E$ (using Rule of Thumb stated above)

```
for(int u=1; u <= 5000 ; u++)
        for (int v = 1; v <= 5000; v++)
                if (r < P) then
                        Pick the edge
```

Below are some statistics of one of my dense graph generation instance:

Avg. degree : 951.488

Edge Count : 2378721

The above values may vary from the graph to graph since we are dealing with random numbers and probability.

4. **Heap Structure:**

I have implemented 2 different types of heap structure. One of them is Max Heap which is exactly as it is recommended in the course project which is used in Dijkstra algorithm and the other one is just for the heap sort for Kruskal algorithm.

MaxHeap: In my Dijkstra implementation, I used the MaxHeap data structure to store the fringes. But the tricky part is when we have to update the bandwidth of a fringe. Since, we don't know the position of the vertex and also as heap doesn't support search I maintained another array POS[] to keep track of my heap elements position. If we don't have this additional array it would take O(n) to search for the vertex. My MaxHeap variables are as below:

VERT array – To store the fringe vertices

BW array – To store the fringe bandwidth

POS array – To track the vertex's position in the heap

My MaxHeap class has the following sub-routines:

insert_into_heap(vertex) : Populates the heap by adding the vertex to the heap data structure. Calls buildmaxheap function on every invocation of this.

delete_from_heap(vertex): Delete fringe vertex from the heap. Calls buildmaxheap function on every invocation of this.

getmaxvertex_from_heap(): Returns the fringe with maximum bandwidth

buildmaxheap(index) : Pushes the max elements to the top.

HeapSort: I used this class for constructing Maximum bandwidth path based on edges sorted. Since the implemented MaxHeap doesn't satisfy the requirement of Kruskal. I had to create this class which sorts the edges and return the edge where in MaxHeap it would return the vertex. Here I need the complete edge information including both the vertices connecting to it. So I push the edge element tuple i.e., pair of pairs (u,(v,wt)) into the heap and pop one by one while constructing the maximum spanning tree. I used only one array since all the needed information is present in the tuple itself. [edge_vector – Array to store the edges]. My HeapSort class contains the following sub-routines:

buildHeapEdgeSort(index) – Heapify from the index

insertedge_into_heap(edge) – Inserts edge into the heap and runs the buildHeapEdgeSort function.

deleteedge_from_heap() – delete the first element (max edge) from the heap

getmaxedge_from_heap() – return the first element (max edge) from the heap

5. Algorithms:

a) Dijkstra: I have implemented the two versions of Dijkstra i.e., one without using any data structure to store the fringes and other with using MaxHeap to store the fringes and pop them up when required.

- Without Heap: When we are looking at fringes, we need to pick fringe with maximum bandwidth and this is achieved by linear search on all the fringes in each iteration. This implementation helps us to compare this naïve method with other implementations which use efficient heap data structure. This implementation has a runtime of $O(n^2)$
- With Heap: I used MaxHeap to store the fringe vertices and pop the fringe with maximum vertex in each iteration. Since the popping element from heap requires O(log(n)) time (since heap contains n vertices), the total runtime for Dijkstra is O(mlog(n)). Because of the use of efficient data structure like MaxHeap, the we could

see the very good performance of this version of Dijkstra compared to without heap implementation. I will discuss those details in the results section.

b) Kruskal:

Kruskal algorithm involves two steps. First is creating maximum spanning tree and second is running Depth First Search to trace the path from source to destination.

Maximum spanning tree creation: For this, I first build the heap using construct_heap() sub routine which adds the edges into the heap. Sorting however is strictly not needed here since we only need the next biggest edge in the graph. This property of producing the maximum of all the left out elements is perfectly delivered by HeapSort class which does the MaxHeap operations but with edges as its elements. After adding the edges, I pick edge with maximum bandwidth by extracting the max edge from the heap and add it to the tree if it doesn't form a cycle. For checking the cycle, I have implemented the modified version of Union-Find which was discussed in the class.

The heap insertion and deletion of all the edges would take mlogm, where m is the number of edges. Once the maximum spanning tree is computed, we still need to find the maximum bandwidth for the destination. For this from the source, I start DFS. There is a single DFS tree generated, since the max spanning tree generated is never disjoint. This only takes O(n). The overall time complexity of Kruskal is given by O(mlogm+mlogn).

During Kruskal implementation, I faced two challenges:

1) How to handle duplicate edges?
2) How to cut down on runtime while handling dense graphs?

#1 In the graph data structure we have the following pairs

V1 -> (V2,wt)

V2-> (V1,wt)

While inserting the edges from the Graph data structure into the heap, we have to make sure that we are not inserting the edge twice. Let me explain my implementation here with an example:

V1 -> (V2,wt1) -> (V3,wt2)

V2 -> (V1,wt1)

V3 -> (V1,wt2)

Above is a sample representation of the graph. I traverse the list of edges vertex by vertex. While traversing the list of edges connected to V2, I will encounter (V1,wt) but we don't want to insert this since we already inserted the edge when we traversed V1 edges. So, if the vertex in the pair is less than the vertex it's connected, I skip adding this edge to my edge. Below is the snippet of my code which explains this

```
for(int i=1; i<=VERTICES_COUNT; i++){
        for(node=G1.edge[i].begin(); node!=G1.edge[i].end(); node++){
                if(i < (*node).first){
                        //insert the edge
                }
}
```

#2 since we know that maximum spanning tree with n vertices contain n-1 edges, while constructing the Maximum spanning tree if at any point my tree contains n-1 edges, I skip iterating through the rest of the edges. This helps to avoid iterating through all edges unnecessarily leading to runtime overhead.

6. **Results and Conclusions:**

I have tested my algorithms on the 5 pair of graphs in which 5 pairs of source and destination (s,t) are randomly chosen. For every algorithm, I have 25 samples for each of dense and sparse graphs. I calculated the average of these running times and tabulated them. The running time of the algorithms is relative to the machine on which the algorithms are run, since these vary a lot with machine load averages.

Algorithm	**Sparse Graph**	**Dense Graph**
Dijkstra without Heap	79.6ms	312.8ms
Dijkstra with Heap	2.8ms	212.2ms
Kruskal with HeapSort	16.4ms	2868.4ms

Execution hostname: linux2.cs.tamu.edu
Number of test samples: 25

Run time for sparse graph generation is much lower than dense graph since the number of edges increases by ~ 125x in dense graph. This is expected.

Run time of Dijkstra with heap is always better than simple Dijkstra. This proves that using efficient data structure with the algorithm is a must for faster run time.

Sparse Graph: From the experiments on 5 varieties of sparse graph and looking at the runtimes, below are my observations:

Running time of Kruskal is better than Dijkstra with heap in sparse graph.

The increasing order of run time for these 3 algorithms are as shown below:
DijkstraWithHeap < Kruskal < DijkstraWithoutHeap

Dense Graph:

Run time of both variants of Dijkstra is better than Kruskal in dense graph.

In dense graphs, Kruskal gets a poor performance because it has to scan all the edges to construct the spanning tree and check if they form a cycle. This becomes a bottle neck in dense graph. Also the constant factor in run time analysis grows bigger.

The increasing order of run time for these 3 algorithms is as shown below:
DijkstraWithHeap < DijkstraWithoutHeap < Kruskal

7. Further Research:

- I could have incorporated more optimization in Dijkstra's without heap algorithm to make it run faster for dense graphs.
- In Kruskal, we can stop scanning the edges if we figure out the source and destination are in the same set.
- There is a linear version of Maximum bandwidth algorithm which would be a better competitor for Dijkstra with heap. I couldn't work on that due to time constraint.

8. References:

Random Graph Models:
https://en.wikipedia.org/wiki/Erdős-Rényi_model

Pairing functions:
https://en.wikipedia.org/wiki/Pairing_function

Pseudo code:
Class Notes