Jayashree Mallipudi 633002212
Department of computer science & Engineering,TexasA&M University

Analysis of Algorithms
Course Project Report
JAYASHREE MALLIPUDI
UIN:633002212

Introduction
In this course project, I have implemented 2 network routing protocols for the construction of a maximum bandwidth path which are Kruskal and 2 types of Dijkstra Algorithms which are simple and heap-based on sparse and dense graphs. These implementations are tested on 5 different randomly generated pairs of graphs (dense and sparse). Each graph is tested on 5 randomly generated source and destination vertices and tested the running time of all these different algorithm implementations.

1.Graph Generation
For the graph construction, I have used the adjacency list array and the weights of edges are randomly generated between 1 to 20.

Sparse graph construction:
First of all final generated graph need to be connected so to make sure of that I connected all vertices with a single edge between them and the last vertex to the first vertex so as to form a cycle and then add edges randomly based on the property of having average vertex degree 6.
For an undirected graph, each edge includes 2 vertices which count in the degree of both vertices so the average degree of an undirected graph is 2 * no of edges/no of vertices
To construct a graph G1 with an average vertex degree of 6 total number of edges needed will be 6* no of vertices/2 =3* no of vertices =3(5000)=15000. Therefore I generate 150000 edges for 5000 vertices and randomly pick 2 different vertices and add an edge between them
Finally, I am able to generate a random graph of average vertex degree 6

Dense graph construction:
In order to have a connected graph add an edge between the adjacent node and connect the edge between the last node and the first node so as to form a cycle.
Now, in order to have each vertex be adjacent to 20% of other vertices
Each pair of vertices are chosen randomly with a probability of 20% and add a random weight for the edge between them.

Jayashree Mallipudi 633002212
Department of computer science & Engineering,TexasA&M University

2.Heap Implementation

I implemented a max heap for Dijkstra's with a heap algorithm
The heap structure is as follows:
In the H heap array, we skip index 0 for convenience and the vertices start with 0 to 4999. The root is the second element in the array. For the kth element of the array, the left child will be at the 2*k index and the right child at the 2*k+1 index, and the parent is located at the k/2 index.
The following operations were implemented in the max heap:
Max: It returns the root element of Maxheap. Time complexity is O(1)
Insert: we add the new element at the end of the heap tree, and then we traverse the element towards the top based on the conditions like if the new value is greater than its parent no need to do anything otherwise traverse up and fix the heap. Since we traverse towards the root of the tree max worst time complexity will be O(logn)
If an element already exists then insert will update the value.
Heapify: In order to fix the element add/delete any case to fix the tree back to its normal properties we use heapify.
Delete: Deleting at an index also in the worst case need to traverse the whole tree Therefore time complexity is O(logn)

3.Routing Algorithms:

As part of the project, I have implemented modified Dijkstra and Kruskal algorithms in order to find the maximum bandwidth.
Below are the implementation details:
Dijkstra:
I have implemented two versions of Dijkstra's
1. Simple Dijkstra's -Implementation of fringes without any special data structure.
Time complexity is $O(n^2)$.
2. With heap: In this version implemented fringes using MaxHeap its corresponding heap implementation is explained in the earlier section.
The time complexity of this algorithm is O(mlogn)
Algorithm is as follows:
Dijkstra(G,s,t)
1. for each node v in G do status[v] = unseen
2. status[s]=in-tree; bw[s] = +INF dad[s]=-1 F=empty
3. for each edge [s,w] do #O(n) without heap O(nlogn) with heap(since insert takes logn)
    (a) status[w]=fringe
    (b) bw[w]=weight[s,w] Insert(F,w)
    (c) dad[w]=s
4. while status[t] != in-tree do:#there are fringersO(n^2) without heap and O(mlogn) with heap
    (a) pick a fringe v of the max bw[v] v=Max(F); Delete(F,v)
    (b) status[v] = in-tree
    (c) for each edge [v,w] do

i. if status[w] = un-seen then
A. status[w]=fringe
B. bw[w]=min{bw[v],weight[v,w]}
C. dad[w]=v
ii. else if status[w] = fringe and bw[w]<min{bw[v], weight[v,w]} then
A. bw[w]=min{bw[v], weight[v,w]}
B. dad[w]=v Insert(F,w) //it updates value if already exists
(d) return dad[1...n]

For connected graph mlogn >nlogn therefore with heap time is mlogn and without heap is $O(n^2)$
From the source trying to find all fringers along the path updating the bandwidth value.
In the case of Dijkstra's using heap, we store the fringers in Max heap and a few modifications are done to the algorithm like insert and delete functions to the algorithm as specified in the pseudo-code.

Kruskal's algorithm
In this algorithm we would find the max bandwidth between source and destination by constructing a maximum spanning tree.
This construction is based on the union, find make set operations.
The time complexity of the algorithm is ElogE or(ElogV if $E<V^2$)

Kruskal(G, Edges,s,t)
Sort the edges of G in non-increasing order using heapsort(G){e1,e2,...em}
for each node v of G do
Makeset(v)
T<-empty
For i=1 to m do
Let ei=[ui, vi]
r1<-FIND(ui)
r2<-FIND(vi)
If r1!=r2 then
Add ei to T
UNION(r1,r2)

Min Heap sort:
This is based on the binary heap data structure
1. We first build a min heap from the input data edges
2. The smallest item is stored at the root now replace it with an item of the heap, also reduce the size of the heap by 1 filled by heapify the root of the tree, fixing the heap to satisfy heap properties.
3. Repeat these steps until the tree becomes empty.
This heapsort uses min heap sorts in descending order.

Jayashree Mallipudi 633002212
Department of computer science & Engineering,TexasA&M University

Kruskal's Algorithm:

1. Firstly we need to sort all the edges in the graph using heapsort I used min-heap for this purpose. Min heap sort implementation is mentioned above.
2. For each of the sorted edges find its root and if both ends of the edges are in a different tree then combine them otherwise continue the process
3. For finding the root I used compressed find in which we change the parent of all the nodes along the path to the root.
4. In the union we add the shorter tree to a larger one and make a single tree.
5. We use a make set to create an empty graph with all vertices.

4.Testing Results:

I have tested all 3 algorithms on the 5 pairs of graphs in which 5 (s,t) source and destination pair s are randomly chosen. Therefore, Every algorithm is tested on 25 samples for dense and sparse graphs, I calculated the average running time of each pair.
For each graph, I calculated the average running time of all 5 samples and listed them separately.

All the running times mentioned were in seconds
Exact amount is added in the text file along with the code.

Graph1

Algorithm	Sparse	Dense
Naive Dijkstra	1.37	1.98
Dijkstra with heap	0.13	1.08
Kruskal	0.65	42.70

Graph 2

Algorithm	Sparse	Dense
Naive Dijkstra	0.91	1.92
Dijkstra	0.07	1.24
Kruskal	0.63	56.05

Jayashree Mallipudi 633002212
Department of computer science & Engineering,TexasA&M University

Graph 3

Algorithm	Sparse	Dense
Naive Dijkstra	0.77	1.97
Dijkstra	0.08	0.59
Kruskal	0.66	45.84

Graph 4

Algorithm	Sparse	Dense
Naive Dijkstra	1.12	1.65
Dijkstra	0.15	1.02
Kruskal	0.68	48.00

Graph 5

Algorithm	Sparse	Dense
Naive Dijkstra	1.36	1.88
Dijkstra	0.06	1.07
Kruskal	0.69	51.63

Final Average of 5 graphs for different algorithms

Algorithm	Sparse	Dense
Naive Dijkstra	1.11	1.88
Dijkstra	0.10	1.00
Kruskal	0.66	48.84

Analysis

Time complexities of algorithms

Dijkstra	Dijkstra with heap	Kruskal
$O(\|V\|^2)$	O((|E|+|V|)log V) =O(ElogV) if E>V mostly	O(E log E) Or

	for all connected graphs	O(E log V) if $E<V^2$

Sparse graph
We can conclude from the above 5 graph analysis Kruskal and Dijkstra with max heap perform well
For the sparse graph with average node degree of 6 has fewer edges relatively where $|V|^2 > |E|\log V$
$V^2 > E\log E$ for kruskals
Between dijkstra with heap and kruskal log E>log V in our scenario of average degree 6 therefore ElogE>ElogV
Test results satisfies the time complexities order
Dijkstra with heap<Kruskal<Dijkstra without heap

Dense graph
Each node connects to 20% other nodes and has large number of edges
Therefore kurskal has higher impact since we need to sort all edges first and then add them one by one so kruskal runs extremely slow in dense graph since its time is bounded by E logE
Then for the two dijkstra algorithms without heap is relatively 2 times slower to with heap dijkstra and from time complexities for large E we can simplify max heap verison time as Elog V therefore its better than without heap naive version.

Between Kruskal and Dijkstra without heap we can say $E\log E >> V^2$
Therefore we can say that Kruskal is not suitable for dense graphs.

Conclusion:

1. Data structure indeed plays a vital role in the actual running time of algorithms
2. Dijkstra without heap is not ideal in any situation generally
3. Kruskal is efficient on sparse but far worse on dense graph
4. Finally, it's always better to use Dijkstra with a heap version with or without the knowledge of the graph.

Further Research:

There were different variations of kruskal to further improve it like by avoiding duplicate edges during sorting its an unnecessary waste of space and time for the heap and there is also a two branch kruskal algorithm which is improved to choose a middle value suitable for any scenario and also we can split into subgraphs and construct a spanning tree for each and combine.In each iteration of subgraph there will be lesser edges so the time complexity might reduce.