CSCE-629 Analysis Of Algorithms
Project Report

Overview

In this project we discuss different routing algorithms also known as Max-Bandwidth Problems .We implement these algorithms in three different ways and analyze their performance on different graphs which are generated randomly but are connected so that there is possible route between any two vertices.We generate random graphs majorly in two different categories one sparse and other dense graph to study the behaviour of these algorithms .By repeating these exercises for different graphs and different values of source(s) and destination(t) we can calculate average time taken by each algorithm to compute max bandwidth path and can have good idea how these algorithms behave and what are reasons behind good or bad performance of any algorithm under specific conditions.

Description

This project is divided into main 4 sub tasks:

1.) Generating Random Graph
2.) Implementing Heap
3.) Implementation of 3 different routing algorithms
4.) Testing

1.Generating Random Graph

Here we generate random graphs with 5000 vertices in two major categories:

a)Sparse Graphs

We create a random graph of degree 6 for all its vertices.Technically it is not possible to have all vertices to have exactly degree 6. so we join as many vertices as possible with degree exactly 6. Here the connectivity of graph is roughly .1% as one vertex is connected to 6/5000 vertices which is ~.1% .So this is very sparsely connected graph.

b) Dense Graph

Another random graph of connectivity % of 20 is created for 5000 vertices. we use probability and random function to achieve this.So roughly one vertex is connected to 20% of 5000 i.e 1000 vertices.It is based on probability so not exaclty all vertices are connected to exactly 1000 other vertices but it is pretty close on average which leads to overall connectivity of graph to 20%.

2. Implementation of Heap

we implement heap with all the heap operations to create heap which can be used in Dijkstra's Algorithm for picking up fringe with max capacity or to heap sort the edges in non increasing order for Kruskal's Algorithm.Operations required for heap are Insert(),ExtractMax(),Delete(),Heapify() and HeapSort().We maintain two arrays D[] and H[] to maintain one to one relationship between distances and corresponding vertices associated with that distance.

3. Routing

a)Max-Bandwidth without using heap

All vertices are divided into 3 categories intree,fringe,unseen. s is included in intree with all adjoining vertices included in fringe.fringe with maximum capacity is picked up and included in intree. While picking up fringe we go through all fringe and pick up fringe with maximum capacity.

b)Max-Bandwidth with using heap

It is similar to above algorithm in all respect other than that we maintain heap of all fringes so that we can extract fringe with maximum capacity within O(1) time.Insertion of fringe into heap takes O(logn) time.

<u>Max-Bandwidth Dijkstra's Algorithm</u>

```
1. For each vertex v =1 … n do
        status[v] = unseen
2. status[s] = intree
3. for each edge [s,w] do
        status[w] = fringe ; cap[w] = wt[s,w] ; dad[w] =s
4. while status[t]!=intree do
        pick a fringe v with max capacity
        status[v] = intree
        for each edge [v,w] do
            if status[w] = unseen
                status[w] = fringe
                dad[w] = v
                cap[w]  = min{cap(v) , wt(v,w)}
            else if status[w] = fringe & cap[w] <min{cap[v],weight[v,w]}
```

```
                then dad[w] = v
                cap[w]  = min{cap[v],weight[v,w]}

                Time Complexity = nlog(n)
```

c)Kruskal's Algorithm

By Kruskal's Alg. we form a maximum spanning tree by sorting all edges in non increasing order. picking all edges in this order till all vertices are connected and discarding edges which create cycles.Then connecting the path from s to t in maximum spanning tree will give us maximum bandwidth path between s and t.

Kruskal's Algorithm

```
  1. Sort the edges in graph G in non increasing order(e1,e2,........em)
  2. For each node v of graph G do
             MakeSet(v);
3. T = null;
4. For all edges 1 to m do
         let edge ei is between {v1,v2}
         r1 = Find(v1)
         r2 = Find(v2)
         if(r1=r2)
         then { T=T U ei ; UNION(r1,r2) ; }

             Time Complexity = mLog(n)
```

4) Testing

As mentioned in project report we create 5 pairs of both graphs sparse as well dense. For each pair 5 pairs of s and t are generated and all vertices are connected to make sure it is connected.for each instance all the 3 algorithms are applied to compute the result to make sure they give the same result . and time elapsed in this execution of these algorithms is recorded . so that we can analyze the complexity and their performance.

Implementation Details

Implementation of Entire Project is divided into 5 public class.

1)Random Graph

This class is used to generate random graph either dense or sparse.
To store graph we use arraylist of arrays of Integer to store the adjacency list of Graph.
Vector<Vector<Integer>> is used for this purpose. Vector at ith position represents vertices with edges from vertex i.To store the weights of corresponding weights of edges we use two dimensional array weight[n][n]. n is declared as public static class variable . other static variables are p,max_weight and f.
f represents the fixed degree we require each vertex to have in sparse graph.
p represents the percentage of connectivity for dense graph.
max_weight is max weight and edge can have.weight is randomly selected from range [1 … max_weight].
graphWithfDegrees()
There is API to generate graph with fixed degree .
for each vertex i random vertex v is generated from 0 to 4999 and edge is created between them untill degree of i reaches 6. we keep on doing until maximum vertex achieves 6 degree. if any vertex can not find any random vertex with degree less than 6 in 100 retries algo skips that vertex.weight array and arraylist of arrays are updated in symmetrical order to suggest undirected nature of graph.
graphWithFixedConnectivity()
This API to generate dense graph considers each possible edge one time. then random number is generated between 1 to 100 . if that number is less than 21 we place edge between two vertices.out of 5000 possibilities for single node edge will be connected only 20% of time that is approx. 1000 times. which ensures the connectivity of graph will be 20%.top loop i goes from 1 to 5000 and inside loop goes from i+1 to 5000.
This ensures each edge between node is only considered once for connection.
ConnectAll() API connects all vertices in this way 0->1->2->3 … ->n-1 to ensure that graph remains connected.There are getter APIs to receive graph in form of arraylist of arrays and weight array.

2) Edge
It is simple class which just holds attribute variables of edge such as both endpoints e1,e2 and weight of edge.

3)Heap
Heap class is OOPs presentation of heap with all basic heap operations.
it has two H[n] and D[n] arrays to represent vertices or fringes and their capacity respectively.H[] array is also changed according to D[] array so distance of vertex at H[i] will be D[i].
API Insert is to insert into Heap.
ExtractMax() returns fringe on top of heap with max capacity.

HeapifyUp() is to arrange node if it moved upwards. similar function is for HeapifyDown().There are other useful operations such as parent(),left() and right() to traverse array in heap.
size stores the size of heap.HeapSort() sorts the heap in increasing order.

4) CalcMaxBandwidthPath
THis public class has three major APIs to find solution using three algorithms we discussed above.
Its constructors take graph,s and t values as params.It populates its own class variablles from params in constructors.

WithoutUsingHeap() calculates the Max Bandwidth using Dijkstra's Algorithm . it traverse all nodes using array of arraylists and uses their weights from weight arrays. We declare enum of intree,fringe and unseen as public static ints in class.
status[n] array is maintained to keep of track all vertices.
cap[n] array stores the capacity of each node for path starting from s.
cap[s] is initialized with 0. dad[n] array stores the parent of every node from which that node is reachable. it is updated when new node is discovered or new path has more bandwidth than old path in fringe.

UsingHeap() uses all implementation of above algorithm except fringe picking part. fringes are stored in heap . we use our heap class . all distances and vertices are stored in separate array.Using ExtractMax() API of Heap Class we pick up the fringe with max cap and turn its status[] to intree.While inserting into heap any fringe we provide the capacity and vertex for which we are inserting fringe into heap.
Max Value is stored in cap[t] and path from s to t is in dad[] array.

UsingKruskal() solves the problem using kruskal's algorithm. private functions such as MakeSet(v),Find(v),UNION(v) are defined to carry on the set operations.we use heap sort to sort the vector of edges in non decreasing order.D[] array stores the weight of edge and H[] array stores the index of edge in original vector of edges. Maximum spanning tree is stores in weight array and Vector of edges i.e Vector<edges>.

findPath() traverses the resultant tree using DFS starting from s until t and populates dad array to construct path from s to t.
We require another traversal of dad[] from t to s to find Max Bandwidth of Graph.

Getdad() and GetMaxBandwidthValue() returns path and solution respectively.

5)Routing
This is the main class which holds the public static void main(String[] args) function.

Here we uses all the above classes and their public API to carry on testing as mentioned in project report.
In main loop which runs 5 times we generate a pair of graph i.e graph1 and graph2. one sparse and another dense.
In inside loop which also runs 5 times we generate value of s and t and connect graphs using connectAll() API.
For these pair of sand t values all 3 algorithms are applied on both graph1 and graph2.we generate new CalcMaxBandWidth class each time for different procedure.Once we get dad[] and solution we print the path from t to s for each procedure as well as Max bandwidth Value.
for each instance we measure time starting from beginning of algorithms to until we get path from class object.then path of these are also printed alongside each result.
We add up the time for each algo on different type of graph and divide by 25 i.e 5*5 .
5 times graph is generated and 5 times same algo is run on it using diff. values of s and t.

Sample Output

Sample output for one such instance of output showcasing time elapsed ,path from s to t and bandwidth value:

```
Max Bandwidth Value  is 702976
3555 <--- 2776 <--- 3797 <--- 3798 <--- 4599 <--- 4041 <--- 4040 <--- 2226 <--- 2225
<--- 3010 <--- 4659 <--- 4807 <--- 4047 <--- 2547 <--- 386
The time elapsed is :9.473877 ms
```

Performance Analysis

Time Complexity for Dijkstra’s Algorithm without heap is n*n.
As picking up fringe also takes n time with maximum capacity.
While using heap extracting max takes O(1) time fro heap of fringes while insertion into heap takes O(logn) time. Kruskal algorithm considers each edge at time and insert into resultant tree if it does not create cycle.searching vertex in set takes logn time.

So Dijkstra’s is expected to perform better when using heap as picking fringe from heap takes logn time when compared to dijkstra’s algorithm without heap.Kruskal Algorithm is mlogn. It performs better when there are lower number of edges. more dense the graph more number of edges and connectivity more time complexity and worse the performance.In Sparse graph kruskal performs great as there are so less number of edges compared to vertices then the dene graph.

we run our project of 3 different Max bandwidth algorithms on both types of graph sparse as well dense and record avg time elapsed in Max bandwidth calculation.

	Dijkstra's without Heap	Dijkstra's with heap	Kruskal's Algorithm
sparse	123.79376	71.132569	27.207194
dense	113.585413	334.580977	1550.237356
sparse	1.971866	9.42347	13.429503
dense	16.858718	150.628815	1644.979177
sparse	12.136181	0.864989	13.182129
dense	476.426368	178.7322	1543.46583
sparse	56.741173	9.374839	12.110769
dense	541.434014	375.35398	2146.74919
sparse	31.322933	4.993493	7.349281
dense	599.374933	453.643906	1553.834522

Fig 1.0

Shows 5 Iterations of one graph through different s and t

	Dijkstra’s without Heap	Dijkstra’s with heap	Kruskal’s Algorithm
Sparse	61.96(ms)	10.08(ms)	8.96(ms)
Dense	215.56(ms)	156.4(ms)	1399.0(ms)

Fig 2.0

Fig 2.0 shows average time in milliseconds(ms) for all scenarios.

As we can see in case of Sparse graphs Dijkstra’s Alg. without heap time complexity is n2 which takes 61ms. Time elapsed goes down sharply in case of heap as time complexity is nlogn where n =5000 so logn is much lower than n.

for n = 5000 logn = ~3. So this reduces time elapsed very drastically.

In Case of kruskal mlogn logn is still very lower but m is very less for sparse graph. no of edges or connectivity is less which leads to speed up time in kruskal and avg time is 8.96.

In case of dense graph there are more possible paths so avg. time for all algorithms increase significantly. Without heap it takes 215.56 ms for calculation of path. it is expected as it is worst possible algorithm without any optimization for dense graph.

using heap reduces the time complexity by factor of logn instead of n .

So time is expected to go down as it happens time Using heap for dense graph is 156.4 seconds.

Kruskal is affected pretty badly as it has factor of m(edges) instead of n multiplied to logn. SO more dense graph means more no. of edges and connectivity which increases time elapsed very significantly. no of edges in dense graph is much more than that of sparse graph. there are 1000 edges for every vertex i.e

20/100 *5000*5000/2 ~= 500000 edges .

So Avg. time increase almost 7 times to 1399.0ms for Kruskal for dense Graph.

Conclusion

We can safely conclude that Dijkstra’s Without heap is worse than using heap as it performs badly in almost cases as maintaining heap causes extraction of fringe in constant time and insertion in log(n) time. For larger value of n(no. of vertices) Dijkstra’s is expected to perform better with heap.

Kruskal algorithm performs really well for graphs with less no. of edges or less connectivity.As connectivity goes up the perfomance detiorates as we saw in above readings. so it is not preferable to use in case of dense graphs.

In conclusion we can suggest that Dijkstra’s Algorithm using heap is preferable to use in case of Dense graphs with more connectivity. In case of sparse graphs with low connectivity Kruskal Alg. outperforms Dijkstra’s Alg. and is more preferable.

Possible Improvements

1.Space complexity is not much bigger issue with modern machines but still we can save some space if we use similar array of arraylists to store weight for edges rather than 2-D array which leaves most of spaces redundant and wastes lot of memory.array of arraylists stores only weights of edges which exists thus saving lot of space in process.
2. We can possibly use QuickSort to sort the edges in non decreasing order for Kruskal as it can perform better in some scenarios but we will have to create another class to achieve so.
3. Path Compression can be used while finding a vertex from a set which speed ups the alg. even greater than logn . we can use additional stack to store dad of each vertex and then traverse along the array saving many iterations in process.