November 30, 2017

CSCE 629: Analysis of Algorithms

Implementation of Network Routing Protocol

(Language: Java)

Submitted By:
Anjali Chadha
626008062

Introduction:

This project is the implementation of a network routing protocol. We are using 3 different approaches to calculate the maximum bandwidth path. The performance of these approaches is evaluated on both dense and sparse networks.

Graph Generation:

For the purpose of evaluating algorithms, we are generating two types of graphs: sparse graphs and dense graphs. Also, as mentioned in the project, we are assuming a fixed number of nodes, 5000. Sparse graphs and dense graphs will differ on the basis of the number of edges present in the graph.

As mentioned in the project description, our sparse graphs should have average vertex degree of 8. This graph can be generated using the following algorithm:

1) Given n = 5000, average vertex degree = 8.
2) Calculate the total number of edges that should be present in the sparse graph.
   Total number of edges (m) = 4 * n = 20,000.
3) In order to ensure that the graph is connected, first create a cycle connecting all the nodes of the graph. This process will use 5000 (n) edges.
4) Remaining edges left = 20,000 - 5000 = 15,000.
5) Now, until number of edges left = 0, we will randomly select two vertices out of the 5000 vertices, and connect them using an edge if they are not already connected.

To create a dense graph, in which every node is connected to about 20% of other vertices, we use an algorithm similar to the sparse graphs.

1. Given n = 5000
2. In order to ensure that the graph is connected, first create a cycle connecting all the nodes of the graph.
3. Now, for every pair of nodes, we will randomly select an integer between 1 and 100. If the number is less than or equal to 20, we will put an edge between the two nodes, if they are not already connected. This will require two nested **for** loops iterating through all the nodes of the graph.

We are using adjacency list representation of the graph throughout this project. Also, all the graphs used here are undirected. In both algorithms described for generating sparse and dense graphs, we perform a step that requires us to check whether an edge already exists in a pair of nodes.

```
private List<Edge>[] adjList;
```

1.1 Code Snippet of adjacency list representation of the graph

One approach to find whether an edge already exists in the graph is to iterate through the array list corresponding to the given node in the adjacency list of the graph. However, this approach may be time-consuming as it requires us to search all the edge list of the node. Time Complexity of this approach is O(m).

Improvement to the above approach would be using a HashSet, which stores a unique representation of a pair of nodes as the key. Now, when we want to check whether the edge already exists or not, we simply need to calculate the unique key for the pair of the nodes and look up in the HashSet. The time complexity of this approach is O(1), constant time.

```
private int calculateEdgeKey(int s, int t) {
    return (int)Math.pow(10, 8) * s + t;
}
```

1.2 Code Snippet calculating the hashkey for an edge.

Using second approach instead of the first one, reduced our time to create the dense graph by almost half.

Algorithms:

We are using two famous algorithms - Dijkstra's shortest path algorithm and Kruskal's Minimum Spanning Tree algorithm in this project. We have modified these algorithms to calculate the maximum bandwidth in a given graph.

1) ***Dijkstra's algorithm without the use of heap data structure.***
   The implementation details are very similar to the algorithm discussed in the class. We are storing all the fringe vertices in a simple array list. Therefore, whenever we want to find a vertex with highest edge weight, we need to traverse entire list of fringe vertices. The time complexity of finding the fringe vertex with maximum weight is O(n). Using a list data structure ultimately leads to quadratic running time of Dijkstra algorithm
   For more details, look *DijkstraWithoutHeap.java* in source code.

2) ***Dijkstra's algorithms using heap data structure.***
   Here we are using a maximum heap data structure to store fringe vertices. Therefore, time complexity of finding the node with maximum weight is constant, O(1).

As mentioned in the project description, we are using two arrays, H[] and D[] to store the vertices and their values (bandwidths) respectively.

Also, we have two variables - size and n.
size - the actual size of the heap.
n - the length of the array representing the heap array.

Since we already know the maximum number of elements that can be present in the heap, we initialize its size to *n*, in order to avoid any array resizing we may need to perform later.

We have implemented Insert(vertex), delete(vertex), deleteMaximum(), maximum() methods for Heap.

To efficiently implement delete(vertex) method of the Heap data structure, we used another array reversePQ[]. *Why do we need another array?*
delete(vertex) method requires us to first find the position of the vertex in the heap and then delete it from the heap. If we don't used reversePQ[], we need to search the entire heap for the required vertex. Since heaps are not efficient for searching, worst case time complexity of search in heap would be O(n).

Instead, if we use a reverse indexed array, reversePQ[] which stores the heap location of each vertex, delete(vertex) operation can be performed in O(log n) time.
For more details, look *DijkstraWithHeap.java* and *HeapDijkstra.java* in source code.

3) Modified Kruskal Algorithm

We modified the Kruskal's Minimum Spanning Tree algorithm to find the Maximum Spanning Tree for the given graph. We are required to implement Union Find and Heap data structure for the efficient implementation of this algorithm.

The implementation details of Union Find data structure is same as discussed in class where we perform union by rank and do path compression.

In the first step of Kruskal algorithm, we require all the edges in decreasing order. To achieve this, we again use a heap data structure which allows us to perform heap-sorting on all the edges efficiently. In every iteration of the Kruskal, we can remove the edge with highest weight and perform heapify() in O(log n) time.

Since we are using a 0-based index heap, our parent and child indices are as follows:

1.3 Code Snippet find parent's and children's position in heap.

```
private int parent(int i) {
    if(i == 0) return -1;
    if((i - 1)/2 >= 0) {
        return (i - 1)/2;
    }
    return -1;
}

private int leftChild(int i) {
  int j = 2 * i + 1;
  return (j < size) ? j: -1;
}

private int rightChild(int i) {
    int j = 2 * i + 2;
    return (j < size) ? j : -1;
}
```

For performing heapify() procedure to restore the order in the heap, we have written two subroutines --float(index) and sink(index). float subroutine compares the index element with its ancestors and take the position which satisfies the heap property. sink subroutine compares the index element with its descendants and take the correct position.

Performance of Routing Algorithms:

For the purpose of evaluating the algorithm's performance, we will randomly pick 5 source and destination vertices for a graph, and run all three algorithms.

The following times are calculated on Intel i7- 3.00GHZ

A. Sparse Graphs

1) Graph 1

(Source, Destination)	Time Taken(sec)		
	Dijkstra without Heap	Dijkstra with Heap	Kruskal (MST creation + Path finding)
3290, 3989	0.091	0.002	0.008
3263, 169	0.002	0.003	0.008
2751, 4132	0.003	0.001	0.009

4313, 2113	0.01	0.002	0.01
3929, 4100	0.113	0.008	0.016

2) Graph 2

(Source, Destination)	Time Taken(sec)		
	Dijkstra without Heap	Dijkstra with Heap	Kruskal (MST creation + Path finding)
621, 4686	0.132	0.004	0.009
2737, 2942	0.023	0.002	0.007
2178, 3510	0.11	0.003	0.008
3328, 2184	0.122	0.012	0.015
4954, 166	0.092	0.005	0.011

3) Graph 3

(Source, Destination)	Time Taken(sec)		
	Dijkstra without Heap	Dijkstra with Heap	Kruskal (MST creation + Path finding)
4607, 2496	0.128	0.004	0.008
1000, 694	0.105	0.005	0.008
4664, 911	0.084	0.002	0.007
4087, 605	0.109	0.01	0.017
2092, 2880	0.13	0.01	0.014

4) Graph 4

(Source, Destination)	Time Taken(sec)		
	Dijkstra without Heap	Dijkstra with Heap	Kruskal (MST creation + Path finding)
4807, 4577	0.006	0.0	0.007
1819, 1200	0.136	0.008	0.009

1663, 4362	0.139	0.005	0.011
3854, 4994	0.115	0.004	0.009
3917, 957	0.132	0.012	0.019

5) Graph 5

(Source, Destination)	Time Taken(sec)		
	Dijkstra without Heap	Dijkstra with Heap	Kruskal (MST creation + Path finding)
1141, 214	0.033	0.001	0.007
4554, 2483	0.109	0.002	0.009
4999, 3351	0.116	0.006	0.01
2700, 2399	0.099	0.003	0.008
256, 2830	0.142	0.013	0.016

B. ***Dense Graphs***

1) Graph 1

(Source, Destination)	Time Taken(sec)		
	Dijkstra without Heap	Dijkstra with Heap	Kruskal
4998, 2036	0.163	0.045	5.092
3628,3217	0.003	0.002	5.072
3657, 3040	0.187	0.072	4.885
1872, 2224	0.038	0.008	5.199
4543, 3897	0.151	0.032	5.37

2) Graph 2

(Source, Destination)	Time Taken(sec)		
	Dijkstra without Heap	Dijkstra with Heap	Kruskal (MST creation + Path finding)
4390, 1541	0.108	0.102	4.89

1050, 1584	0.166	0.052	4.96
2200, 3472	0.196	0.066	4.818
2073, 1427	0.133	0.01	4.872
358, 3279	0.134	0.064	5.225

3) Graph 3

(Source, Destination)	Time Taken(sec)		
	Dijkstra without Heap	Dijkstra with Heap	Kruskal (MST creation + Path finding)
3871, 319	0.122	0.043	5.938
4323, 3268	0.142	0.029	5.047
2641, 1701	0.209	0.027	5.901
4095, 4353	0.186	0.033	5.676
2045, 817	0.057	0.03	6.25

4) Graph 4

(Source, Destination)	Time Taken(sec)		
	Dijkstra without Heap	Dijkstra with Heap	Kruskal (MST creation + Path finding)
1786, 4551	0.08	0.015	4.258
4760, 2388	0.088	0.077	5.7
4666, 1476	0.028	0.045	4.356
3433, 4656	0.187	0.048	4.562
1456, 1364	0.19	0.038	5.203

5) Graph 5

(Source, Destination)	Time Taken(sec)		
	Dijkstra without Heap	Dijkstra with Heap	Kruskal (MST creation + Path finding)
1456, 2898	0.041	0.058	5.456

1356, 1203	0.018	0.028	5.379
4420, 3508	0.27	0.023	4.987
1456, 2898	0.041	0.058	5.456
775, 827	0.117	0.079	5.103

Terminal Output:

```
Source, Destination 2499, 2459
Dijkstra without heap: MaxBW- 172
2459, 4528, 4527, 3417, 4935, 4565, 4566, 4043, 3014, 3013, 1372,
2484, 1333, 1332, 2021, 4983, 1185, 1184, 556, 1017, 1366, 1365, 757,
758, 759, 3152, 194, 193, 4911, 236, 4240, 4241, 4987, 2257, 4357,
3065, 3236, 2773, 2981, 2384, 736, 737, 1575, 4614, 2499,

Dijkstra Algorithm Without Heap completed in  0.049 sec
Dijkstra with heap: MaxBW- 172
2459, 4528, 4527, 3417, 4935, 4565, 4566, 4043, 3014, 3013, 1372,
2484, 436, 4419, 4418, 4621, 4622, 4623, 1763, 1762, 2965, 1588,
2557, 985, 3334, 3333, 4584, 2374, 2375, 2376, 2377, 2928, 3064,
3065, 3236, 2773, 2981, 2384, 736, 737, 1575, 4614, 2499,

Dijkstra Algorithm Using Heap completed in 0.006 sec
Kruskal: MaxBw - 172
2459, 4528, 4527, 3417, 4935, 4565, 4566, 4043, 3014, 3013, 1372,
2484, 436, 4419, 4418, 4621, 4622, 3387, 578, 579, 1952, 4162, 4270,
4269, 1096, 1097, 4252, 2628, 948, 949, 950, 2913, 2912, 3239, 2719,
2720, 1700, 4274, 1429, 534, 664, 3711, 1207, 4852, 659, 124, 123,
3298, 2582, 4857, 4243, 4151, 4203, 3152, 194, 193, 4911, 236, 4240,
4241, 4987, 2257, 4357, 3065, 3236, 2773, 2981, 2384, 736, 737, 1575,
4614, 2499,

Kruskal Algorithm completed in 0.019 sec
```

Analysis and Discussion:

As per the times calculated above, the performance of Dijkstra's algorithm with and without heap is aligned with the concepts discussed in class.. Whether the graph is dense or sparse, the Dijkstra's algorithm using heap almost always performs better than the algorithm without heap. This is because the time complexity of the algorithm not using heap is quadratic in nature and that of Dijkstra's algorithm using heap is O(m log n). In case, we increase the number of vertices in the graph, the performance difference of both the algorithms will be more apparent.

For sparse graphs, the performance of the Kruskal and Dijkstra with heap is comparable and better than Dijkstra's without heap.

However, in dense graphs, If we look at the execution time of Kruskal algorithm, the performance of Kruskal is worst of all three. However, there is more than meets the eye. Kruskal algorithm has two steps. First, it calculates the maximum spanning tree of the graph, followed by the finding the bandwidth of the two input vertices on the MST. In our performance analysis above, for every source, destination pair of vertices, we are performing both steps. However, in practise, we need to create the Maximum Spanning Tree only once per graph. Once that is done, we can find the bandwidth using that tree in linear time by performing the Depth First Search. Therefore, if we average the running time of Kruskal accounting the factors explained above, we can observe that Kruskal's performance is surely better than the Dijkstra's algorithm without using heap. And Kruskal performance is comparable to Dijkstra with heap algorithm as in case of sparse graphs.

Theoretically, the time complexity of the Kruskal and Dijkstra is O(m log n), which is dependant on the number of edges in the graph. Therefore, it is expected that dense graph algorithms will take longer to complete than the sparse graphs algorithm. This pattern can be observed in our data findings above.

If our graph is a static network, in which no nodes and edges are being added or removed dynamically, then Kruskal's algorithm will be the most efficient to use out of all the three. This is because our network is static, we need to calculate the Maximum Spanning Tree only once. After the MST creation, path finding will take linear time only for any pair of vertices. However, in real life networks, the static assumption is rarely applicable, and in that case Dijkstra's algorithm with heap may be our best choice.