Project Report

Maximum Bandwidth Path problem is an important network optimization problem. The project compares the performance of different algorithms that solve the problem when applied to different types of networks. The number of nodes in the network is 5000. A random graph generation algorithm is used to generate 10 graphs. In 5 of these, the degree of each node is approximately 8 i.e. the graphs can be categorized as sparse graphs. In the remaining 5, the probability of an edge between any 2 nodes is 0.2 The expected degree of each node is thus 1000 and the graph is a categorized as a dense graph.
The performance of 3 algorithms: Dijkstra without Heap, Dijkstra with Heap and Kruskal (with HeapSort) is observed on the 2 categories of graphs.

1. **Implementation Details**

   The project is implemented in *python*. It is divided into 4 parts.
   - Graph Generation
   - Dijkstra's algorithm without heap to find maximum bandwidth path
   - Dijkstra's algorithm with Heap to find maximum bandwidth path
   - Kruskal's algorithm to find maximum bandwidth path

   Graph Generation:

   In this project, two graphs are generated one is a sparse graph and other is a dense graph. Both the graphs contain 5000 nodes. Sparse graph has a degree of approximately 8 whereas dense graph has a degree of around 1000.

   The graphs are implemented using an **adjacency list**. Adjacency list is created using a default dictionary in python. The key in the dictionary is the name of the vertices represented by the index and the value is a **list of vertex node.** Vertex node is a custom data structure that stores the vertex name and the bandwidth of the edge connecting the two vertices in the adjacency list.

   Algorithm to create sparse graph:
   1. For i= 0 to V/2 // where V is the number of vertices
   2. While length(graph[i]<8)
   3. j = random(0,V-1) //generates a random number between 0 and V-1
   4. if j != i and there is no edge between i and j and length(graph[j]<8)
   5. w = random(1,1000) // w is randomly generated weight of the edge
   6. add edge from i to j with weight w
   7. add edge from j to i with weight w

Algorithm to create dense graph:

1. For i= 0 to V/2 // where V is the number of vertices
2. While length(graph[i]<0.2*V)
3. j = random(0,V-1) //generates a random number between 0 and V-1
4. if j != i and there is no edge between i and j and length(graph[j]<0.2*V)
5. w = random(1,1000) // w is randomly generated weight of the edge
6. add edge from i to j with weight w
7. add edge from j to i with weight w

Find more details about the classes below:

Vertex: It is a node to store destination and bandwidth

```
class Vertex:
        def __init__(self,dst=0,bw=0):
            self.dst = dst
            self.bw = bw
```

Edge: It is a node to store edges with three attributes source, destination and bandwidth

```
class Edge:
        def __init__(self,src,dst,bw):
            self.src = src
            self.dst = dst
            self.bw = bw
```

Graph: It is an adjacency list containing vertex objects. The size of the adjacency list is 5000.

```
class Graph:
        def __init__(self):
                self.graph = defaultdict(list)

        def neighbors(self,u): This returns the list of vertex objects in the adjacency list
                                at  index u

        def exists(self,i,j): This function checks if an edge already exists between node i
                              and j

        def addEdge(self,u,v,bw): Adds an edge in the graph from u to v with bandwidth
                                  bw

        def createGraphG1(self):  Creates a sparse graph G1

        def createGraphG2(self): Creates a dense graph G2
```

Dijkstra's algorithm for maximum bandwidth path:

Dijkstra's algorithm is implemented in as taught in class. Three arrays have been initialized to store bandwidth, status and parent of every node. So, to get the maximum fringe we will have to iterate over the entire array. This makes the time complexity of getting the fringe with maximum bandwidth linear.

Dijkstra's algorithm with heap:

This algorithm is also similar to the one taught in class. The only difference between this Dijkstra's algorithm above is the inclusion of heap. Heap is implemented using a list. The heap implemented in *heap.py* is a max heap that supports INSERT, DELETE and MAX operation. To improve the performance of delete operation an auxiliary list is used to store the position of vertices in the heap.

To make the heap faster, a list is implemented that stores the index of vertices in the heap. This makes delete operations faster as we don't have to search for the vertex in the heap. Heap implementation can be found in *heap.py.* The heap class creates two lists, *heap* and *position*. Heap stores all the vertices and position stores indices of the vertices.

Whenever the position of a vertex changes due to insert or delete operation, the position of the vertex is updated in the *position* list.

Details about the class heap.

```
class Heap:
        def __init__(self):
            self.heap = list()
            self.position = [-1 for i in range(V)]

        def heapify(self,i):
            largest = i
            left = 2*i+1
            right = 2*i+2

            if left<len(self.heap) and self.heap[left].bw > self.heap[largest].bw:
                    largest = left
            if right<len(self.heap) and self.heap[right].bw > self.heap[largest].bw:
                    largest = right
            if largest != i:
                    self.heap[largest], self.heap[i] = self.heap[i], self.heap[largest]
                    self.position[self.heap[i].dst] = i
                    self.position[self.heap[largest].dst] = largest
                    self.heapify(largest)

    def insert(self, v, bw):
            if len(self.heap) == V:
                    return
            vertex = Vertex(v,bw)
            self.heap.append(vertex)
            i = len(self.heap)-1
            self.position[v] = i
            while i!=0 and self.heap[parent(i)].bw < self.heap[i].bw:
                    self.heap[i],self.heap[parent(i)]=self.heap[parent(i)], self.heap[i]
                    self.position[self.heap[i].dst] = i
                    self.position[self.heap[parent(i)].dst] = parent(i)
                    i = parent(i)

    def delete(self, v):
            positionInHeap = self.position[v]
            if positionInHeap > V:
                    return
```

```
                i = len(self.heap)-1
                self.heap[i],self.heap[positionInHeap]=self.heap[positionInHeap], self.heap[i]
                self.position[self.heap[positionInHeap].dst] = positionInHeap
                self.heap.pop()
                self.position[v] = -1
                self.heapify(positionInHeap)
                return

        def maximum(self):
                return self.heap[0].dst
```

Kruskal's algorithm for Maximum Spanning Tree:

In Kruskal's algorithm, a new list E is created from the graph G that contains all the edges of the graph. This list of edges is sorted in non-decreasing order. After sorting the edges, the edges are picked starting from the maximum to create a spanning tree. To implement the spanning tree, MakeSet, Find, Union operations are used as taught in class. This functionality is contained in a class named *SetUnionFind.* The SetUnionFind class creates an object that contains information about *parent, rank and size* for each node in the tree. The find operation is iterative, it traverses the entire path. This makes it linear time.

To create the spanning tree, a new graph T is created and edges are added to it by iterating over the list of edges starting from length(E) – 1.

Details about the class SetUnionFind

```
Class SetUnionFind:
        def __init__(self):
                self.parent = [i for i in range(0,V)]
                self.rank = [1 for i in range(0,V)]
                self.max_size = V

        def __init__(self,size):
                self.parent = [i for i in range(0,V)]
                self.rank = [1 for i in range(0,V)]
                self.max_size = size

        def MakeSet(self,i):
                self.parent[i] = i
```

```
            self.rank[i] = 1

        def Union(self,i,j):                // Combines different nodes into a single subtree
            if self.rank[i] > self.rank[j]:
                self.parent[j] = i
            elif self.rank[i] < self.rank[j]:
                self.parent[i] = j
            else:
                self.parent[j] = i
                self.rank[i] += 1

        def Find(self,i):                   // Finds the root of a node
            w = i
            temp = list()
            while self.parent[w] != w:
                temp.append(w)
                w = self.parent[w]
            while len(temp)>0:
                i = temp[len(temp)-1]
                temp.pop()
                self.parent[i] = w
            return w
```

2. Performance of the algorithms

Sparse Graph:

Source, Destination	Time to run		
	Dijkstra's algorithm	**Dijkstra with Heap**	**Kruskal's algorithm**
3246, 726	0:00:00.534924	0:00:00.536495	0:00:00.255377
1932, 1354	0:00:01.801110	0:00:01.788919	0:00:00.259360
1029, 4116	0:00:02.304125	0:00:02.260387	0:00:00.246018
1511, 3222	0:00:02.082543	0:00:02.084962	0:00:00.249031
3865, 2495	0:00:00.278861	0:00:00.252293	0:00:00.255156

Dense Graph:

Source, Destination	Time to run		
	Dijkstra's algorithm	**Dijkstra with Heap**	**Kruskal's algorithm**
1977, 2444	0:00:00.888564	0:00:00.828745	0:00:47.429537
1276, 2398	0:00:00.654699	0:00:00.497206	0:00:47.525748
1008, 3672	0:00:05.160293	0:00:04.237169	0:00:49.608761
3990, 1934	0:00:01.296324	0:00:01.176892	0:00:51.126040
4649, 1877	0:00:01.180375	0:00:00.704987	0:00:49.620178

3. Analysis of the algorithms

For sparse graph, the observed performance of the algorithms is

Kruskal's algorithm > Dijkstra's algorithm with heap > Dijkstra's algorithm

Kruskal works marginally faster than Dijkstra with heap and much faster than Dijkstra. Dijkstra takes O(n^2) time because of the complexity of finding maximum fringe. So, this was already known. However, the surprising thing is the difference in the running time of Kruskal and Dijkstra with heap because the time complexity of both the algorithms is the same O(m log n). This difference is due to constants in Big-O notation.

This can happen due to various reasons. Kruskal sorts the edges once and it can take less time if there are patterns in the data. Kruskal doesn't insert or remove edge from the heap multiple times. On the other hand, many insert, delete operations take place in Dijkstra with heap. This takes longer time.

For dense graph, the observed performance of the algorithms is

Dijkstra's algorithm with heap > Dijkstra's algorithm > Kruskal's algorithm

Dijkstra with heap works faster than Dijkstra and both of them are more than 10 times faster than Kruskal. We know the difference in complexity in Dijkstra and Dijkstra with heap. So, this was an expected result. The reason Kruskal is so slow is because it works on the number of edges. For dense graphs, the number of edges m>>n. To show things in perspective for a dense graph that is connected to 20% of its neighbors, if the number of vertices is 5000 the number of edges would be around 1875538. As the number is so large it takes a lot of time to sort all the edges and then create a maximum spanning tree.

4. Improvements

Sorting the edges using heapsort takes a lot of time as the number of edges is extremely large. We can improve upon this sorting technique and select the one that performs better. We can try other sorting algorithms such as quick sort or merge sort to look for performance improvements.

The find operations in the creating a maximum spanning tree can be improved by using techniques described in class. The algorithm in the project takes linear time. This makes the overall algorithm a lot slower. If we implement the method taught in class that reduces the time complexity of Find operation from O(mn) to O(n + mlog*n).