CSCE-629 Analysis of Algorithms

Course Project – Network Optimization

Satya Kesav G, UIN: 426009269

Instructions for program execution

The program is written in C++. It is recommended to use C++ 12 for executing the program since it involves few libraries that are supported with C++ 12 or above. Currently, the program only prints the max BW of the path between the source and destination. But, if you intend to see the path for any algorithm, please uncomment the stdout lines that are present at the end of the program. If you intend to change the network parameters, they are present at the beginning of the program.

Implementation Details

1. **Network Parameters:**

N	/* Number of Nodes/ Vertices in the Graph */
degree	/* Degree of each vertex in Type-1 Graph (Sparse if N is large) */
per100	/* percentage of two vertices being adjacent in Type-2 Graph*/
max_weight	/* denotes the maximum weight of an edge in the graph */
min_weight	/* denotes the minimum weight of an edge in the graph */

The above parameters decide the anatomy of each graph that is generated in this project. The weights in the graph are randomly generated for each edge but they are bounded by the minimum and maximum values hard coded in the program which are reported above. They may be changed to any values of interest but within the range of the data type INT in C++. Positive values of the weights of edges are recommended since it is a maximum bandwidth problem and hence the algorithms might not work properly for any other values.

2. **Graph Generation:**

There are two kinds of graphs that are generated in this program which are described below:

a. **<u>Type-1 Graph:</u>**

The main parameter that distinguishes this graph from the other type of graphs is the degree which is reported above. The degree denotes the number of edges each vertex has on an average in the graph. Based on the degree of each vertex, the graph may become a sparse graph or dense graph. For an average vertex degree of "8", the graph turns out to be a sparse graph since the number of nodes assumed for this project are 5000. If the number of nodes are very less in the graph, but with vertex degree 8, the graph may appear dense because the density of the graph depends on the ratio between number of edges (which depends on vertex degree) and the square of number of nodes.

In this implementation, to make the graph connected, a simple cyclic graph is created first with all nodes included. Later, for each vertex, adjacent nodes are added randomly in order to make the degree of each vertex 8. Self-loops and multiple edges are avoided during the process deliberately which can be seen from the code.

void CreateCycle(vector<vector<pair<int,int>>>& adjlist, int n, vector<vector<int>>& maplist) is the function that creates a simple cyclic graph. '*maplist*' is used to avoid repeated edges in the graph.

void CreateType1Graph(vector<vector<pair<int,int>>>& adjlist, int n, int degree)
is the function in the program that creates the graph described above.

The '*adjlist*' is a 2-D vector which holds the adjacency list representation of the graph generated and '*n*' denotes the number of vertices and '*degree*' denotes the vertex degree.

Each cell in the '*adjlist*' is of type pair<int,int>. The first field of the pair describes the vertex number and the second field describes the weight of the edge in the adjacency list representation. For clear understanding, see the example below:

Adjlist[0] = [(1,5), (2, 10)]

Adjlist[1] = [(0,5)]

Adjlist[2] = [(0,10)]

The sample graph given above has three vertices 0, 1 & 2. Vertex-0 is connected to vertex-1 with an edge of weight 5 and connected to vertex-2 with an edge of weight 10.

b. Type-2 Graph:

The parameter that effects the Type-2 Graph is the percentage of two vertices being adjacent to each other. The percentage considered in this graph is 20%. That means, every node in the graph in connected to 20% of the total nodes in the graph. Considering 5000 nodes, the graph becomes a dense graph.

In this implementation, a cyclic graph is created first using the same function given above to make sure the graph is connected and later other adjacent vertices are added in order to make the number of adjacent nodes as about 20% of total, thus creating a dense graph for the given scenario.

void CreateType2Graph(vector<vector<pair<int,int>>>& adjlist, int n, int per100)
is the function that creates the graph described above in the program.

Note: The graph generation is not analysed since it is not accounted for any performance measures.

3. Heap Implementation:

The implementation of Max Heap which is made compatible to work with Dijkstra's Algorithm is presented below.
The following are the basic functions created for the implementation:

1. *void Insert(vector<pair<int,int>>& maxheap, int& heaplen, vector<int>& pos, pair<int,int> input)*
2. *pair<int,int> extractMax(vector<pair<int,int>>& maxheap, int& heaplen, vector<int>& pos)*
3. *void Delete(vector<pair<int,int>>& maxheap, int& heaplen, vector<int>& pos, int index)*
4. *pair<int,int> getmax(vector<pint>& maxheap, int& heaplen, vector<int>& pos)*

The heap array (maxheap) is an array of pairs for which each pair contains the first field as the vertex number and second field as the max bandwidth of node from a given source. The 'pos' array contains the position of each node 'i' in the max heap. 'heaplen' is the length of the max heap.

The 'getmax' function just returns the maximum value in the heap whereas 'extractMax' function returns the maximum value while deleting it from the heap.

Maximum of Heap:

The 'getmax function returns the first element in the max heap which is the maximum element in the array. Here the maximum value corresponds to the maximum bandwidth value. This takes only an order of O(1).

Insertion in Heap:

For Insertion into the max heap, we first append the element at the end of the heap array and recursively compare the current element's bandwidth (BW) value with the parent's bandwidth value. We execute it until we either reach the root of the heap i.e. the first element of the heap or when the parent's BW is already greater than the current element's BW.

While updating the max heap array, the position array ('pos') is also updated accordingly for each swap operation between child and parent i.e. the positions are also swapped. The entire operation takes an order of O(log n) since in a binary tree of size n nodes, the maximum height is log n. In the worst case, we travel up from the bottom of the heap array to the root.

Deletion in Heap:

Deletion can be done in several ways. In this approach, the 'extractMax' function is utilized (function description is detailed above). An element which is given to be deleted is first updated with BW value of 'infinity' and then the element is bottled up until the root. This takes an order of O(log n) for the same reason as Insertion since the heavier element has to reach the root from the bottom of the tree in worst case.

After the element is bottled up to the root, the 'extractMax' function is called which removes the root of the max heap array. This operation also takes time O(log n). Thus the entire delete operation in heap takes an order of O(log n).

These functions will be used for the implementation of Dijkstra's Algorithm later in this program.

4. Routing Algorithms for Maximum Bandwidth path:

a. Dijkstra's Algorithm without Heap:

In this algorithm, a bandwidth array is created to store the maximum bandwidth of all nodes starting from a given node. At the end of the algorithm, all the nodes have their maximum BW values updated (if run for the entire graph, but the algorithm is usually stopped after reaching the destination). Also, for tracing the path from the source to destination, a 'dad' array is maintained and a status array is used to record the status of each node in the graph. The status can be 'unvisited', 'fringe' or 'in_tree' in the algorithm.

Unvisited	- The node is still not explored.
Fringe	- The node is in fringe now.
In_tree	- The node is explored and the maximum BW value is obtained.

Initially, the status, BW and dad arrays are reset. In the next step, the values from the source node are assigned to these arrays. Later, the explored tree is expanded until the destination node is recorded as 'in_tree'.

The initialization step in the algorithm takes time O(n). The expansion of the explored set takes an order of O(n^2) since every node has to be explored in the worst case in order to reach the destination. And during each expansion, the entire bandwidth array has to be traversed to find the maximum bandwidth node which is still a fringe. These two operations taking O(n) each and nested make the complexity O(n^2) (ignoring O(n) initialization time). The actual complexity would be O(m+n^2) since in the worst case all the 'm' edges are iterated. But, m is always less than or equal to n^2 in a non-repeating edges and no self-loops connected graph.

int Dijkstra(vector<vector<pair<int,int>>>& adjlist, int& source, int& destination, vector<int>& path)
is the function that computes the maximum bandwidth from source to destination.

The function returns the max BW value and the '*path'* array lists the path from source to the destination.

b. Dijkstra's Algorithm with Heap:

In this implementation, a fringe array is separately maintained and is stored as a heap to reduce the cost of finding the maximum BW fringe element from O(n) to O(log n). This makes the initialization process described above take time O(n*log n) since all initial fringe (adjacent to source) nodes have to be inserted into the heap (n nodes with O(log n) insert operation cost).

In the next step where the explored set is expanded to find the destination node, the complexity now becomes O(m*log n + n*log n). During the expansion, all the operations for making a node fringe in the above algorithm are replaced here by the heap insertion operations. Also, the operation where the bandwidth of a fringe is updated by a new maximum is replaced by a deletion followed by insertion into the heap. The m*log n factor arises because for every edge (from an explored vertex) if there is an unexplored vertex, it is inserted into the heap which takes O(log n) time. Thus the complexity becomes O((m+n)*log n).

int Dijkstra(vector<vector<pair<int,int>>>& adjlist, int& source, int& destination, vector<int>& path)
is the function that computes the maximum bandwidth from source to destination.

The function returns the max BW value and the '*path'* array lists the path from source to the destination.

Note: The analysis of the algorithms for various kinds of graph inputs are mentioned in the Results section to avoid repetition.

c. Kruskal's Algorithm:
In this algorithm a maximum spanning tree is constructed using Kruskal's algorithm first and then DFS is run on the spanning tree to find the maximum BW between the path from source to destination.
In the Kruskal's algorithm, the edges are sorted in descending order using heap sort in order to create a fair comparison between Dijkstra's algorithm with heap. After the sort, each edge is extracted from the max heap array and the add it to the spanning tree if it doesn't form a cycle. To find whether an edge forms a cycle, Union find algorithm has been used. Also, union by rank and path compression are used to optimize the complexity of the 'find' operation.
Implementation of Union Find:
The following are the main functions used for implementing the union find data structure.
void union12(int r1, int r2, vector<int>& parent, vector<int>& rank)
int findpc(int v, vector<int>& parent)
The '*findpc'* function finds the parent of a vertex v given. The '*union12*' function joins the two roots '*r1'* and '*r2*' i.e. either r1 is attached to r2 or r2 is attached to r1 based on the rank of the vertex in order to grow the height of the tree (union) logarithmically. In the '*findpc'* function, path compression technique is used and thus attaches all the vertices directly to the root. The detailed analysis is taken from the class notes and hence not mentioned here.

The complexity of the Kruskal's algorithm therefore takes O(m*log m) time for sorting and then creating the MST. Later, for finding the max BW by applying DFS takes O(m+n) time. Hence, the algorithm takes O(m*log m + n).

Testing and Results

For the tables reported below, the following are the assumptions:
Number of vertices – 5000, Minimum weight – 10, Maximum weight – 99.

The first five are of Graph type-1 i.e. sparse graphs and the rest are of Graph type-2 i.e. dense graphs.

Graph-1 (Sparse)

Source – Destination (BW)	Dijkstra's (ms)	Dijkstra's, Heap (ms)	Kruskal's (ms)
4993 – 4067 (84)	78	0	Total time for building
2892 – 64 (81)	109	0	maximum spanning tree
4312 – 2651 (79)	141	15	is 15 ms. It is redundant
4543 – 520 (79)	110	15	to construct spanning
3419 – 3701 (77)	156	16	tree for each pair.
Average time:	**118.8**	**9.2**	**3**

Graph-2 (Sparse)

Source – Destination (BW)	Dijkstra's (ms)	Dijkstra's, Heap (ms)	Kruskal's (ms)
172 – 15 (71)	172	15	Total time for building
782 – 3862 (76)	78	16	maximum spanning tree
2776 – 4071 (76)	109	15	is 11 ms. It is redundant
298 – 4541 (85)	141	15	to construct spanning
4083 – 3064 (60)	78	0	tree for each pair.
Average time:	**115.6**	**12.2**	**2.2**

Graph-3 (Sparse)

Source – Destination (BW)	Dijkstra's (ms)	Dijkstra's, Heap (ms)	Kruskal's (ms)
4170 – 1905 (83)	94	0	Total time for building
4929 – 505 (83)	109	0	maximum spanning tree
3608 – 3697 (84)	93	0	is 31 ms. It is redundant
3103 – 2853 (72)	94	0	to construct spanning
2206 – 1337 (86)	62	16	tree for each pair.
Average time:	**90.4**	**3.2**	**6.2**

Graph-4 (Sparse)

Source – Destination (BW)	Dijkstra's (ms)	Dijkstra's, Heap (ms)	Kruskal's (ms)
674 – 2831 (83)	109	0	Total time for building

2638 – 2756 (83)	109	0	maximum spanning tree
4020 – 1085 (88)	143	15	is 16 ms. It is redundant
471 – 3010 (75)	157	0	to construct spanning
1794 – 2831 (76)	170	0	tree for each pair.

Average time: 137.3 3.0 3.2

Graph-5 (Sparse)

Source – Destination (BW)	Dijkstra's (ms)	Dijkstra's, Heap (ms)	Kruskal's (ms)
2316 – 1034 (70)	156	16	Total time for building
4510 – 3887 (82)	110	0	maximum spanning tree
458 – 4633 (49)	93	0	is 16 ms. It is redundant
4395 – 1364 (68)	156	16	to construct spanning
1127 – 1883 (86)	47	0	tree for each pair.

Average time: 112.4 6.4 3.2

By observing the results of the performance times for the three algorithms on the type-1 graph i.e. sparse graph, it is quite evident that Dijkstra's algorithm without heap takes more time compared to the algorithm with heap. This also can be seen theoretically that the complexity of Dijkstra's without heap is of order O(n^2), whereas the complexity for the algorithm with heap is O(n*log n). When compared to Kruskal's algorithm, it should be almost similar to Dijkstra's with heap as the edges are sorted using the heap sort algorithm which takes the time O(m*log m) since m is proportional to n in sparse graphs. Also, there are few points to note here. Since Kruskal's algorithm builds the maximum spanning tree first, we can find the maximum bandwidth path from one node to any other node in linear time. Hence, using the Kruskal's algorithm for a multiple source/destination maximum bandwidth problem is very efficient. For some of the results that are indicated by '0ms', it means that the algorithms run in fraction of milli seconds for which C++ could not compute.

Graph-1 (Dense)

Source – Destination (BW)	Dijkstra's (ms)	Dijkstra's, Heap (ms)	Kruskal's (ms)
2546 – 781 (99)	91	148	Total time for building
2059 – 4360 (99)	122	147	maximum spanning tree
3655 – 2897 (99)	78	89	is 2719 ms. It is redundant
4505 – 3043 (99)	94	110	to construct spanning
4980 – 1687 (99)	122	178	tree for each pair.

Average time: 101.4 134.4 543.8

Graph-2 (Dense)

Source – Destination (BW)	Dijkstra's (ms)	Dijkstra's, Heap (ms)	Kruskal's (ms)
4895 – 1632 (99)	62	78	Total time for building
4392 – 3993 (99)	141	144	maximum spanning tree
3979 – 4501 (99)	78	157	is 2860 ms. It is redundant
3451 – 738 (99)	94	93	to construct spanning
1290 – 192 (99)	110	132	tree for each pair.

Average time: 97.0 120.8 572

Graph-3 (Dense)

Source – Destination (BW)	Dijkstra's (ms)	Dijkstra's, Heap (ms)	Kruskal's (ms)
1986 – 4301 (99)	172	47	Total time for building
1419 – 2311 (99)	94	78	maximum spanning tree
4260 – 2639 (99)	110	109	is 2859 ms. It is redundant
3686 – 446 (99)	47	93	to construct spanning
3045 – 2873 (99)	94	109	tree for each pair.

Average time: 103.2 87.2 571.8

Graph-4 (Dense)

Source – Destination (BW)	Dijkstra's (ms)	Dijkstra's, Heap (ms)	Kruskal's (ms)
2383 – 1061 (99)	102	125	Total time for building
2249 – 364 (99)	86	131	maximum spanning tree
303 – 1126 (99)	96	147	is 2984 ms. It is redundant
1644 – 2181 (99)	90	123	to construct spanning
4856 – 2956 (99)	110	188	tree for each pair.

Average time: 96.8 142.8 596.8

Graph-5 (Dense)

Source – Destination (BW)	Dijkstra's (ms)	Dijkstra's, Heap (ms)	Kruskal's (ms)
3041 – 896 (99)	81	113	Total time for building
1658 – 742 (99)	95	149	maximum spanning tree
2850 – 3326 (99)	140	125	is 2906 ms. It is redundant
4128 – 4438 (99)	110	188	to construct spanning
2141 – 894 (99)	109	178	tree for each pair.

Average time: 107.0 150.6 581.2

Observing the results of the performance times of the three algorithms for a dense graph, it is clear that the Dijkstra’s algorithm with heap take on average more time than the algorithm without heap. The understanding is as follows: theoretically we know that the running time of the algorithm without heap and with heap is O(n^2) and O(m*log n + n*log n) respectively where m denotes the number of edges. For a dense graph, the number of edges are proportional to n^2 hence making the algorithm with heap costlier than the algorithm without heap. The same is observed in most of the cases tabulated above which is practical.

While observing the Kruskal’s algorithm, it is seen that it performs at least 10 times slower than the Dijkstra’s algorithm with heap. The reason is that the algorithm sorts all the edges according to their weights and hence takes time O(m*log m) and m covers all the edges. But, in the Dijkstra’s algorithm with heap, although the theoretical complexity is O(m*log n), m doesn’t include all the edges. It includes only the edges that are in and around the path from source to destination. For e.g:- If we are measuring the maximum BW in the path between vertices 1 and 2, since the graph is dense, the vertices would be only few nodes away, maybe less than 100 nodes away. Then the running time becomes 100*log n instead of 5000*log n which is given theoretically. Thus, there is a difference between the theoretical complexity and the practical running time. This is a great instance of such problems.

Hence, Kruskal’s algorithm is always a bad choice for dense networks. Dijkstra’s with heap also performs slightly slower compared to the Dijkstra’s algorithm without heap for dense graphs.

Further improvements

For sparse graphs, the algorithm-1 i.e. Dijkstra’s without heap may be improved using the following approach. During every iteration while expanding the explored set, the node that is a fringe with maximum bandwidth is searched among all 5000 (in this case) vertices. Instead of searching over all vertices, if the fringe nodes are separately maintained in a linked list, the search can be optimized from O(n) to O(size of fringe array). Since the graph is assumed to be sparse, the size of fringe array will be very small compared to the size of the graph (If the weight distribution of edges is closer). Also, note that there will be deletions and insertions into the fringe array which is only O(size of fringe array). Thus, better performance may be achieved using a separate fringe array.