Network Flow and Circulation with Demands Problem

Sainadh Chilukamari (sc249)

Project-1 Report

Advance Algorithms

Algorithms Description

Adjacency List Creation: Graph is a data structure that consists of two components that is a finite set of vertices and a finite set of ordered pair of the form (u, v) called as edge The pair of the form (u, v) indicates that there is an edge from vertex u to vertex v and this edges contain weight. This graph is represented using adjacency list. An array of LinkedList is used to create an adjacency list. An '*.txt' file figure shown below (left) has edges and weights for each node. Each line is read from the text file and the i^{th} value is taken as to the node and edge is created with $(i+1)^{th}$ value as weight. The adjacency list figure created for the '*.txt' file is shown below (right). This particular graph has 8 nodes with the source (0) and sink (7).

```
1 10 2 15 3 5
2 4 4 9 5 15
3 4 5 8

5 15 7 10
6 15 7 10
2 6 7 10
```

```
adj[0] -> |1 | 10| -> |2 | 15| -> |3 | 5| ->
adj[1] -> |2 | 4| -> |4 | 9| -> |5 | 15| ->
adj[2] -> |3 | 4| -> |5 | 8| ->
adj[3] ->
adj[4] -> |5 | 15| -> |7 | 10| ->
adj[5] -> |6 | 15| -> |7 | 10| ->
adj[6] -> |2 | 6| -> |7 | 10| ->
adj[7] ->
```

Breadth-First Search: The Breadth-First Search algorithm is used to traverse the graph and to find all possible paths from source to sink. The adjacency list is created from the graph file (as shown in the above right figure) is given to the graph program. Along with this adjacency, the source and sink nodes are also given to the program. The traversal starts with the source node by pushing the source node into the queue data structure. We have also used two arrays, one is a boolean array named visited_arr and another array is of type integer named parent which is used to store the parents of the current vertex while traversing the graph. At first, the node in the queue is popped and the visited array is flagged true for the current node (source node). The adjacent nodes to the current node are stored in an array list from the adjacency list. For each adjacent node, when the adjacent vertex is not in the visited array and the edge to it is greater than zero, then the node is added into the visited array (flagged true for that node) and the node is added to the queue. Again, after all the adjacent nodes for the source are added to the queue, the same process is done again with the first adjacent node, i.e., the first input to the queue is popped, and so on until the sink node is popped. This is done because, when the sink node is approached, the adjacent node to be inserted into the queue doesn't exist and the traversal ends. Below is the pseudo-code that we have used for the BFS algorithm. BFS also returns the shortest path from source to sink if the sink node is discovered in the traversal.

```
BFS (G)                //Where G is the graph - adjacency list
    let Q be queue.
    let visited_arr be boolean array //To avoid processing more than once
    let parent be int array //To store parent of particular vertex
    Q.add( source ) //Inserting source in queue until all its neighbour vertices are marked.
    mark source as visited.
    while ( Q is not empty)
       //Removing that vertex from queue,whose neighbour will be visited now
       v  = Q.poll( )
      //processing all the neighbours of v
      for all neighbours w of v in Graph G
         if w is not visited
              Q.add( w )          //Stores w in Q to further visit its neighbour
              mark w as visited. //Put true in visited_arr
Store v as w's parent.
```

Ford Fulkerson algorithm: For this algorithm, adjacency list is given to Ford Fulkerson function and this is a directed graph involving a source(s), sink(t) and several other nodes connected with edges, where each edge has an individual capacity that is the maximum flow allowed through that particular edge. For any node, the in-flow should be equal to the out-flow. For any edge, the flow should be greater than or equal to zero and less than or equal to the maximum flow of that edge. This algorithm is used to calculate maximum flow in a graph.

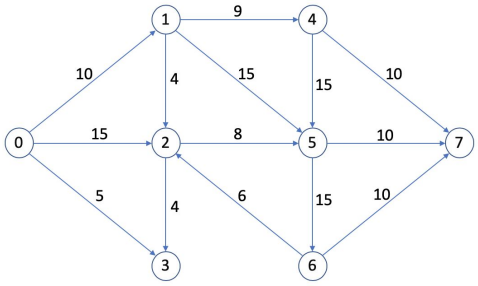
```
FordFulkerson(Graph G):   //Where G is the original graph - adjacency list
    residualGraph = G          //store original graph as residual graph
   maxFlow =0             //initialize maxFlow to zero
   while (BFS(residualGraph)):   //check for augmenting path between S and T
in residual network graph
      Augment flow between S to T along the path P
      maxFlow += pathFlow //store min path flow and add it to maxFlow
      Update residual network graph
   return maxFlow
```

According to the pseudo-code (above), the algorithm requires a graph as an input, the original graph is stored as a residual graph and then modified using the BFS output, to obtain the final residual graph. The residual graph is obtained by augmenting paths from source to sink. The residual graph shows the additional flow through the edges after the maximum flow through a particular path. After all the paths are explored, the final residual graph is created where there is no additional possible flow from source to sink. Therefore, if there is a path from source to sink in the residual graph, it is possible to add flow from source to sink. The final residual graph, help us to calculate the maximum flow and the paths required for maximum flow.

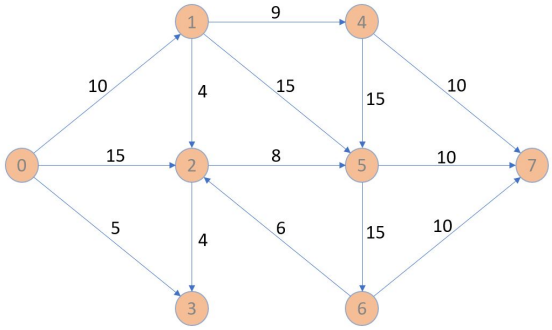
Circulation Problem: In the circulation problem, the Max-flow problem is considered again, but this time there will be no source and sink. Each node will have a demand or supply if $d(v) > 0$ it demands and if $d(v) < 0$ it supplies Since we are considering multiple nodes with demands and supplies, a common source and a common sink are added to the graph or adjacency list where all supplies are connected to the source and all demands to sink. The circulation problem is only possible when the sum of demands is equal to the sum of supplies in a graph. It is feasible if and only if it is equal to the max-flow value f which is obtained by the Ford-Fulkerson algorithm.

Graph Generation: This algorithm generates graph files for the programs, which is a random graph generator. The user is prompted to enter the number of vertices, and based on this the program picks a random number of edges for the graph at around 60 percent of the number of edges for a complete graph i.e., (numOfVertices*(numOfVertices - 1)/2). There are some conditions given for the graph generation, such as no edge is to be created once an edge exists between two vertices, no outgoing edges from the destination vertex(last vertex), no incoming edges to the source vertex (first vertex) and there is no self-loop to a vertex. All these conditions must be satisfied and the graph is generated. The graph is written to a .txt file and later converted to adjacency lists by a graphing program to be used in BFS and FFA algorithms.

Visualization Example



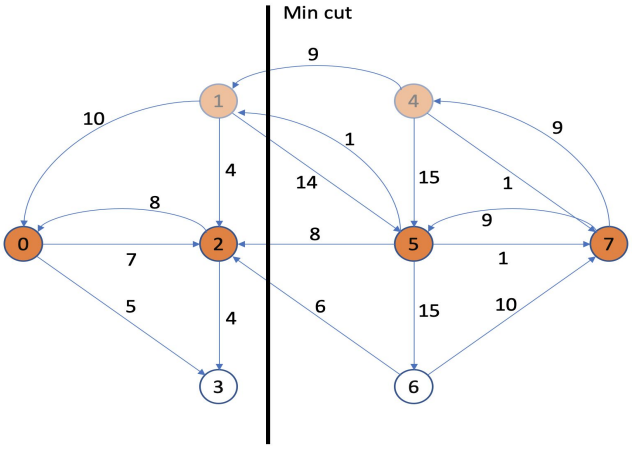
Below is an original graph of 8 nodes and 14 edges, with weights assigned between the nodes. There are many different paths that can lead to the sink (7) from the source (0). As per the BFS algorithm, the BFS traversal for the above graph, 0 1 2 3 4 5 7 6, as shown in the figure below. Every node is stored in a queue and polled out of queue to check if there are any adjacent nodes to it. For eg: Node 0 has 1, 2, 3 as adjacent nodes. So, these nodes are pushed into queue to check for other adjacent nodes, this process repeats that there are no other adjacent nodes. Every time the node that was popped out from the queue was tracked for BFS traversal.



BFS Traversal:	0	1	2	3	4	5	7	6

As per the Ford-Fulkerson algorithm the above graph with a maximum flow of 18 as shown in the figure below. There are only 3 paths that can be traversed from source (0) to sink (7) as per the residual graph. For every iteration, the minimum capacity of that path will be subtracted and new

edges are added called residual edges (backflow with minimum path capacity). As per the below figure, 0-2-5-7 is the last path in the graph and then there are no paths from source (0) to sink (7), here the maximum flow attained is 18.



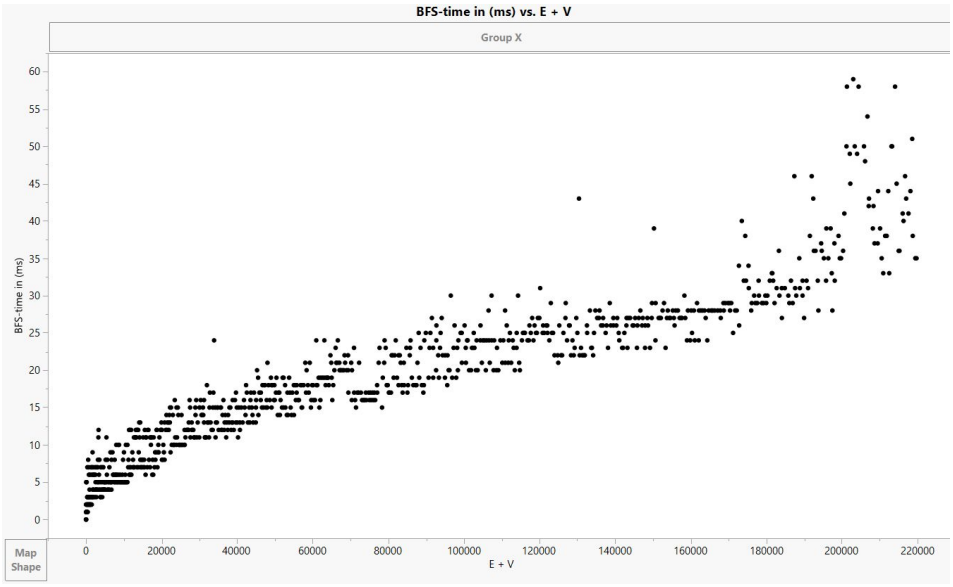
Max Flow = 9 + 1 + 8 = 18

Test Cases and Implementation Results

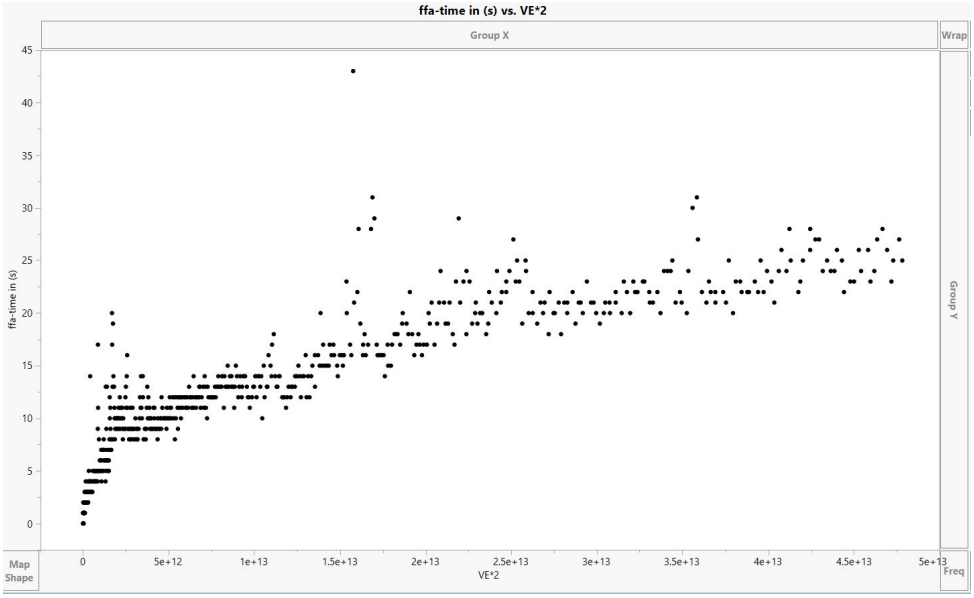
A total of 1000 test cases are considered for BFS and FFA, all of which are generated from the graph generator program. These 1000 test cases consist of nodes 1 to 1000 respectively. For each test case, a random number of edges are assigned, and the algorithms are run for these graphs. The random number of edges is 60% of edges from a complete graph. For all these test cases, the run times of BFS and FFA algorithms are calculated. The runtimes of BFS in ms (milliseconds) are plotted with (E + V) and results in a graph (as the figure is shown below) which contradicts the linearity. This is because some of the randomly generated nodes may or may not have a path from source to sink, so runtime here is 0ms, and large graphs which have path took more time to run (difference in the last path of the graph). We have also generated graphs of 5000, 10000 and 20000 nodes and tested BFS and FFA algorithms. FFA algorithm took nearly 5.5 hours to give the max flow of 20000 nodes, but it took only about 1 hour for 10000 nodes. This shows it is exponentially increasing with the increase of nodes.

Data Structures Used

For all modules/algorithms, we have used data structures like Linked List, ArrayList, Queue and Arrays. We have used different data structures because in linked list insertion and deletion are faster and in arrays, there will be fast data retrieval. For example, to create a graph from the graph file we have used Linked List data structure for adjacency lists and in the BFS algorithm, we have used a queue for BFS traversal, parent array to store the parent information of a particular node and also a visited array to avoid processing a node more than once.



Whereas the runtimes of FFA in s (seconds) are plotted with (E*2)V and we observed that we are also calculating the runtimes of FFA algorithm with no max flow or path from source to destination. As we can see the graph below results in an graph close to the exponential graph (as the figure is shown below).



Contributions

- Worked on generating an adjacency list from the given text file.
- Worked on implementing the Ford-Fulkerson algorithm and Circulation problem.
- Worked on creating test cases.