**Network Flow and Circulation with Demands Problem**

Prudhvi Kishan Kotamarthy (pk100)

Project-1 Report

Advance Algorithms

**Algorithms Detailed**

**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 these edges contain weight. This graph is represented using adjacency list. An array of LinkedList is used to create an adjacency list. A ‘\*.txt’ file figure shown below (left) has edges and weights for each node. Each line is read from the text file and the ith 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 graph has 10 nodes with the source (0) and sink (9).

Calendar

Description automatically generated with medium confidence Text

Description automatically generated with low confidence

**Breadth-First Search:** 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 as input. 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 nodeVisited 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 Implementation:**

BFS (G)               //Where G is the graph - adjacency list

      let Q be queue.

      let nodeVisited be Boolean array //To avoid visiting a node more than once

      let parent be an Integer array //To store parent of a node

      Q.add (source edge) //Inserting source in queue && //Mark source as visited

      while (Q is not empty and size of Q is greater than zero)

           //Removing that vertex from Q, whose neighbors will be visited now

           currentNode =Q.poll()    
 currentNode value is taken from the source

//processing all the neighbors of currentNode

for all neighboring edges of currentNode in Graph G //destination edge is assigned to nodeNumber

               if edge is not visited

                        Q.add (edge details) //Stores the new edge in the Q to further visit its neighbor

                        mark the above edge as visited. //Put true in visited at the end.

store the edge as currentNode’s parent.

return (nodeVisited[sink]). //when the destination is reached

**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 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.

The source is calculated by calculating the node whose count of No. of inbounds is 0. We traverse through the entire adjacency list in a for loop and break it when the inbound value becomes true. Then we will return the value of the source.

The sink is calculated by traversing the entire adjacency list and returning the target when we reach the end of the adjacency list.

**Ford Fulkerson Implementation:**

FordFulkerson (Graph G):   //Where G is the original graph

      maxFlow = 0 //initialize maxFlow to zero

s = getSource and t = getTarget Nodes // Source Inbound is Zero. Target Outbound is zero

residualGraph = G         //store original graph as residual graph

     while (BFS (residualGraph, s, t)):  //check for augmenting path from source to sink

        maxFlow += pathFlow // store minimum capacity path flow

        Update residual network graph

    return maxFlow

According to the pseudo-code (above), we calculate the source and sink, and 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:

* 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.

**Test Case Generation Algorithm:** This algorithm generates graph files for the programs, which is a random graph generator. The number of vertices starts from 5 and the loop runs till 10000 generating graphs for every case. The program picks a random number of edges for the graph at around 50 percent of the number of edges for a complete graph i.e., (numOfVertices\*(numOfVertices - 1)/2).

The above formula is obtained by below statement:  
**A complete graph has an edge between any two vertices. You can get an edge by picking any two vertices. So if there are n vertices, then there are nC2=n(n−1)/2 edges**.

Below conditions are coded for the graph generation:

* 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)
* No self-loop to a vertex.

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.

**Test Cases and Implementation Results:**

A total of 2328 test cases are considered for BFS and FFA, all of which are generated from the graph generator program. These 2328 test cases consist of nodes 1 to 2328 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 50% 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 are plotted in a line graph (as the figure is shown below).

The runtimes of FFA in s (seconds) are plotted with (V\*E2) and results in a line graph (as the figure is shown below).

We have also generated graphs of 2000, 10000 and 20000 nodes and tested BFS and FFA algorithms.

FFA algorithm took nearly 6 hours to give the max flow of 20000 nodes and it took about 1 hour for 10000 nodes. This is evident that it is exponentially increasing with the increase of no. of nodes.

**Sample Run Results:**

**Table

Description automatically generated**

**BFS graph – (Run time in milli-seconds vs E+V) :**

**Ford Fulkerson - (Run time in seconds vs V\*E2):**

**Data Structures Used:**

For all modules/algorithms, we have used data structures like Linked List, ArrayList, Queue and Arrays.

We have used Adjacency list instead of Adjacency Matrix because of below two reasons:

* + Space complexity is improved while using Adjacency List as we don’t need to traverse the list repeatedly which we do in Adjacency Matrix.
  + Time complexity by using Adjacency Matrix is O(|V|2). For Adjacency List it is linear. So, we chose Adjacency List.

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 a visited array to avoid processing a node more than once.

**Contribution in the project:**

● Written executable module for generating an adjacency list from the given text file.

● Implemented the Breadth First Search, Ford-Fulkerson algorithm, and Circulation problem in a different approach.

● Created generic algorithm for multiple test cases generation in a single run.