Informatics Institute of Technology

Algorithms:Thoery and Implementation

5SENG002C.2

Coursework-01

Module Leader's Name –Mr.Sudarshana Walihinda

Name : Sajeban Antonyrex

UoW Number : w1698405

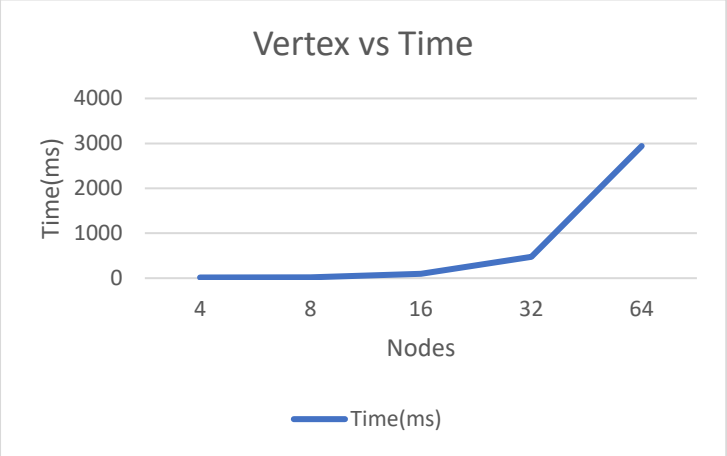
IIT Number: 2017472

1 Overview

Finding the maximum flow in a network has been implemented by many Algoritms.Here Edmond-Karp Algorithm in Ford-Fulkerson Implementaion is used. Path with minimum edges are picked because of the Breadth First Search Approach thereby reducing the time.

2 Performance Analysis

The following diagram represents the Empirical study on the Algorithm. The time increases with the increase of vertex. Because of the differences in capacity and edges connectivity there were different outcomes of time for the same vertex count.Therfor More than 5 times a vertex was tested and the average is represented in the below figure.



Nodes	Test I	Test II	Test III	Test IV	Test V	Avg Time
4	46	0	15	16	0	15.4
8	15	16	31	15	16	18.1
16	78	141	87	79	93	97.8
32	593	390	463	432	511	477.8
64	2776	3140	3443	2732	2611	2940.4

The table represents the tests in Milli Seconds and the average time is plotted in the graph illustrated with the time factor.

The above diagram suggests that the capacity of edges and number of edges play a huge role in this maximum flow finding algorithm. Time accelerates according to the increase of vertex count. By considering the acceleration, we can define that the graph represents a quadratic function(n^2).

*The Performance of the Computer affects in the running of Algorithm.

3 Order of Growth Classification

3.1 Segment by Segment Classification on Critical Places

There were 5 critical places found on the implementation of Algorithm. They are tabulized below.

The IDs and the relevant code can be referred in the Appendix.

ID	Description	Big O	Type
A	Multi Dimensional Array Insertion Assigning values to residual graph	O(n2)	Quadratic Looping through An array within an array
B	HashMap Iteration BFS function to find augmented path	O(n)	Linear Hashmap.get: O(1) Looping through all elements:O(n)
C	Multi Dimensional Array Search Finding bottleneck capacity	O(n2)	Quadratic Traversing through loop with loop to find the bottleneck capacity
D	Multi dimensional Array search and insertion Assigning the Flow in the residual graph	O(n2)	Quadratic Traversing through loop with loop to assign the bottleneck capacity to the residual graph

E	Queue Traversing	O(n)	Linear Looping until queue is empty O(n) Queue Insertion :O(1) HashMap Insertion:O(1) List insertion: O(n) Queue Deletion: O(1)

3.2 Overall Order of Growth Analysis of the Algorithm

The case scenario can be addressed giving the Big O notation.

These factors affect in the running of this Algorithm

1. Adjacent Matrix Representaiton
2. Breadth First Search

Traversing through the matrix representing graph through V vertices give **$O(V^2)$**.

The worst case sceneraio FF EK implementaion is going from the Source to sink passing through all the vertex flowing through all edges .The complexity to do a BFS **O(E).**The complexity of FF implementation is going to be O(E*No Of Augmented Path). Every Augmented will have a bottleneck capacity edge. Then,the number of times an edge can be critical(Edge holding the bottle neck capacity) should be found out.The BFS returns a shortest path first then gives longer paths. An edge can become a critical edge V times. If all edges becomes a critical edge then it will be VE . Therefore we can say that **O(E*VE).**

Since we are doing a BFS search it will be **$O(V^2)$.** Thus giving the final notation as **$O(EV^3)$**.

Appendix

```
public static int[][] findMaxFlow(int[][] graph) {

    int maxflow = 0;
    int[][] rGraph = new int[graph.length][graph.length];
    for (int i = 0; i < graph.length; i++) {             //   multidimensional Insertion
        for (int j = 0; j < graph.length; j++) {

            rGraph[i][j] = graph[i][j];         //Generating the inital residual graph
        }
    }
    HashMap<Integer, Integer> hashMap = new HashMap<~>();
    // while(
    while (bfs(rGraph,  s: 0,  t: rGraph.length - 1, hashMap)) {  //passing the arguments residual graph,initial node, final node
        boolean x = true;
        LinkedList linkedList = new LinkedList();
        linkedList.add(rGraph.length - 1);    //this list holds the path flow
        int y = rGraph.length - 1;
        while (x) {

            y = hashMap.get(y);                 //linkedlist insertion
            linkedList.add(y);          //Populating the path list by hashmap values
            if (y == 0) {
                x = false;
            }
        }
        int d = 0;
        int size = linkedList.size() - 1;
        System.out.println("LINK LIST : " + linkedList);  // printing path of flow....
        int bottleneckCapacity = 25;
        for (int i = 0; i < graph.length; i++) {
            for (int j = 0; j < graph.length; j++) {

                if (size != 0) {//multidimensional array search

                    d = rGraph[(int) linkedList.get(size--)][(int) linkedList.get(size)];
                    System.out.println(d);  //Calculating the bottleneck capacity to send the flow through the route
                    if (d < bottleneckCapacity) {

                        bottleneckCapacity = d;
                    }
                }
            }
        }
        System.out.println(bottleneckCapacity);
        maxflow += bottleneckCapacity;
          size = linkedList.size() - 1;
          for (int i = 0; i < graph.length; i++) {
              for (int j = 0; j < graph.length; j++) {                        //multidimensional array search and insertion
                  if (size != 0) {                                    //Updating the residual capacities in the residual graph
                      d = rGraph[(int) linkedList.get(size--)][(int) linkedList.get(size++)];        //Getting the present value
                      rGraph[(int) linkedList.get(size--)][(int) linkedList.get(size)] = d - bottleneckCapacity;//Subtracting the flow value
                      d2 = rGraph[(int) linkedList.get(size++)][(int) linkedList.get(size--)];                //sent through the path and
                      rGraph[(int) linkedList.get(size++)][(int) linkedList.get(size--)] = d2 + bottleneckCapacity; //Updating the backward edge value
                  }
              }
          }
          hashMap.clear();            //Clearing the hashmap for new augmented path calculation

    }
    int[][] maxflowgraph = new int[graph.length][graph.length];
    for (int i = 0; i < graph.length; i++) {  //multi dimensionnal plotting final max flow graph insertion
        for (int j = 0; j < graph.length; j++) {

            if (graph[i][j] > 0) {
                maxflowgraph[i][j] = graph[i][j] - rGraph[i][j];
            }
        }
    }
    return maxflowgraph;
}

static boolean bfs(int rGraph[][], int s, int t, HashMap<Integer, Integer> hashMap) {
    ArrayList<Integer> visitedList = new ArrayList<~>( initialCapacity: 6);  //list to hold the visited elements
    Queue<Integer> q = new LinkedList<>();      //Queue to store the values visted to do breadthfirst search
    q.add(0);
    visitedList.add(0);
    while (!q.isEmpty()) {   //queue search
        for (int i = 0; i < rGraph.length; i++) { //array
            if (rGraph[q.peek()][i] > 0 && !visitedList.contains(i)) { //if the node is already visted it wont be visted again
                q.add(i);                   //Getting the neighbours of the node and updating them in visitedlist and queue
                hashMap.put(i, q.peek());
                visitedList.add(i);
            }
        }
        q.remove();

    }
    return visitedList.contains(t);//If no more path found this value will be returned thus terminating the search and returning max flow value
}
}
```

A

B

C

D

E