Informatics Institute of Technology

Department of Computing

(B.Eng.) in Software Engineering

Module: 5SENG01W: Algorithms: Theory, Design and Implementation

Module Leader: Sudharshana Welihindha

Academic lecturer - Achala Aponso

Coursework

Name: Dhanasekara Mudiyanselage Akshaan Dileesha Bandara

UOW ID: w1743055

IIT ID: 2018597

Tutorial Group: G

Table of Contents

Algorithmic approach

1) Algorithmic Strategy

Ford–Fulkerson algorithm is used to find the maximum flow of the graph. It also uses Bread First Search (BFS). BFS is used to find the path from source to sink.

The graph is based on 2D matrix.

2) Data Structure

The data structures used in the algorithm is **Array** (1D and 2D), **LinkedList** and **ArrayList**.

- The graph matrix (graph and the residual graph) is stored in 2D Array, whilst 1D Array is used to store the path and the visited nodes in the breadFirstSearch method.

  int[] paths and **boolean**[] visited = **new boolean**[**vertices**];
  depicts the code snippets of path array and visited nodes array.

- In breadFirstSearch method LinkedList is used to store the nodes.

  LinkedList<Integer> nodes = **new** LinkedList<>();
  depicts the code snippets of LinkedList.

- Also, Array List is used to store the augmented path of maximum flow.
  private ArrayList<Integer> **visitedNode** = **new** ArrayList<>();
  depicts the code snippets of Array List to store the augmented path of maximum flow.

  Furthermore, the source and the sink read by GeneralData.txt is stored in an Array List.
  ArrayList<Integer> generalData = **new** ArrayList<>();
  depicts the code snippets of Array List to store the source and sink.

3) Pseudo-code

Note: The pseudo-code is only written to breadthFirstSearch(**int**[][] residualGraph, **int** source, **int** sink, **int**[] paths) and maxFlow(**int**[][] graph, **int** source, **int** sink) methods.

Declare vertices variable to store the number of vertices (/nodes)
Create an ArrayList to store the augmented path an name it as visitedNode

Constructor
MaximumFlow(noOfVertices)
    Initialize vertices to noOfVertices

Breadth_First_Search
breadthFirstSearch(residual graph 2d array, source, sink, paths array)
    pathFound initialized to false
    create an array to store whether the node is visited or not

    create LinkedList to store the nodes
    added the source to the LinkedList

```
set visited[source] to true
set paths[source] to -1

WHILE size of nodes LinkedList not equal to 0
    retrieve and return the first element of the LinkedList nodes and assign it to variable u
    FOR v=0 to v < no of vertices
        IF visited[v] is false AND residual graph [u] [v] > 0
            Add value v to nodes LinkedList
            Set paths[v] to u
            Set visited[v] to true
        ENDIF
        v++
    ENDFOR
ENDWHILE
Set pathFound to visited[sink]
Return pathFound
```

Max_Flow

```
maxFlow(graph 2d array, source, sink)
    Create residual graph and fill it with given capacities in the original graph as residual capacities in
    residual graph

    FOR u=0 to u < no of vertices

        FOR v=0 to v < no of vertices

            residualGraph[u][v] initialized to graph[u][v]

            v++

        ENDFOR
    u++
    ENDFOR

    create an array of length no of vertices to store the path
    initially no flow (maxflow initialized to 0)
    WHILE there is path from source to sink
        Find the maximum flow through the path found
        FOR v = sink to v not equal to source
            add value v to visitedNode ArrayList
            u initializes to path[v]
            Find minimum residual capacity of the edges along the path filled by BFS and
            assign to pathFlow

            u = path[v]

        ENDFOR
```

```
FOR v = sink to v not equal to source
      u initializes to path[v]
      update the residual graph [u][v] -= pathFlow
      update the residual graph [v][u] -= pathFlow
ENDFOR
maxflow += pathflow
Reverse the visitedNode ArrayList
DISPLAY "Path: "+visitedNode+" Flow: "+pathFlow
Clear the visitedNode ArrayList
ENDWHILE
Return maxflow
```

Analyzing the performance of the algorithm

1) Input data sizes

It was created by my own based on doubling hypothesis (i.e. doubling the no of nodes and doubling the no of edges).

- **Data6x6.txt** is 6x6 (6 rows and 6 columns). So, it contains 6 nodes and 12 edges.
- **Data12x12.txt** is 12x12 (12 rows and 12 columns). So, it contains 12 nodes and 24 edges.
- **Data24x24.txt** is 24x24 (24 rows and 24 columns). So, it contains 24 nodes and 48 edges.
- **Data48x48.txt** is 48x6 (48 rows and 48 columns). So, it contains 48 nodes and 96 edges.

2) Measurements of Times

There are 2 packages in the Project (Code_1 and Code_2).

1) The implementation in the Code_1 package is where the user can select an option whether to find the maximum flow, delete node, modify capacity and exit the program. Object from Stopwatch class is created at the beginning of the main method. So, the time complexity for the maximum flow will also have the time which went for the user to enter the options.

The Time in the below table is calculated for the implementation of Code_1 Package (MaximumFlow.java)

N	Time(ms)	ratio	lg ratio
6	1401		
12	1508	1.0764	0.0320
24	1755	1.1638	0.0659
48	2022	1.1521	0.0615

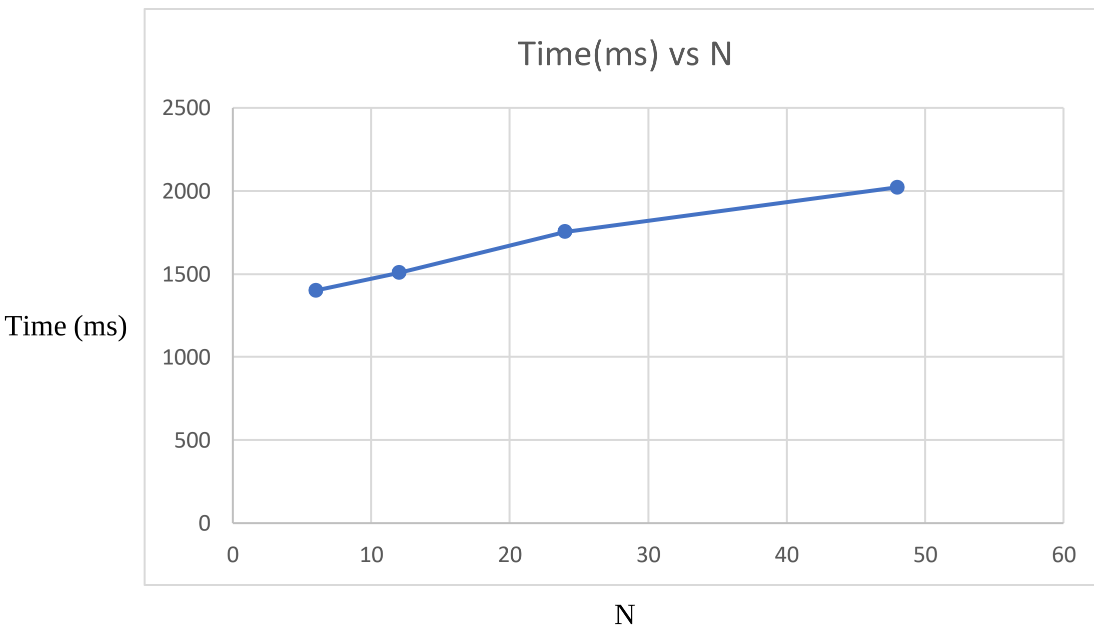
2) The implementation in the Code_2 package is done for measuring the exact running time of the algorithm. It'll first display the maximum flow for the existing graph and it's running time. Then it displays whether to delete a node, modify capacity along with finding the maximum flow.

The Time in the below table is calculated for the implementation of Code_2 Package (MaximumFlow.java)

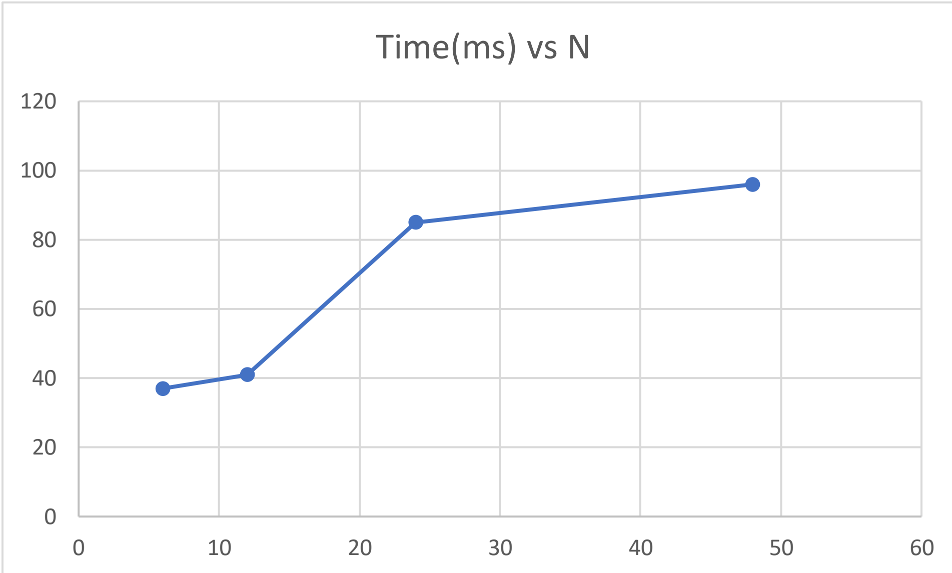
N	Time(ms)	ratio	lg ratio
6	37		
12	41	1.1081	0.0446
24	85	2.0732	0.3166
48	96	1.1294	0.0528

Conclusion

Graph for 1)



Graph for 2)



Time (ms)

N

Algorithmic performance in terms of Big-O *order-of-growth classifications*

The implementation is done based on Breadth-First Search (BFS). Big-O notation of BFS is **$O(N^2)$**. BFS always uses the path with minimum no of edges. So, since BFS is used in the algorithm, the worst-case time complexity can be reduced to O(EN). Thus, the Time Complexity is $O(EN^3)$.

The rules of Big-O notation:

1. Get rid of constants
2. Get rid of lower order values

- Big-O of breadthFirstSearch(int[][] residualGraph, int source, int sink, int[] paths) method: **$O(N^2)$**
- Big-O of maxFlow(int[][] graph, int source, int sink) method:

$O(N^2) + O(N^*2N)$

$= O(N^2) + O(2N^2)$

$= O(3N^2)$

After getting rid of constants Big-O is: **$O(N^2)$**

Thus, the Big-O of the algorithm is: **$O(N^2) + O(N^2)$**

$= O(2N^2)$

$= O(N^2)$

So, **$O(N^2)$** is the Big-O of the algorithm.

The order of growth table:

Order of growth	Name
1	constant
Log N	logarithmic
N	Linear
N log N	Linearithmic
N^2	Quadratic
N^3	Cubic
2^N	Exponential

So, the order of growth for the algorithm is **Quadratic**.

Note:

Guidelines that should be adhered when running the project is written in the README.md file submitted along with code submission.