Faculty Of Engineering Alexandria University

Signal Flow Graph 5 / 25 / 2020

Signal Glow Graph is a systematic method to solve the overall transfer function of any systems type, the project allows you to Enter information of your graph number of nodes and edges with their weights, Project is implemented in Java.

Code Implementation by:

Name : احمد حمدى ابراهيم رضوان

ID : 4

Github Link for code implementation:

https://github.com/Ahmed1Radwan/Signal_Flow-Graph-Solver

Problem Statement

Given

Given a general signal flow graph, write a program to calculate the overall transfer function using Mason Formula.

Requirements

- An interactive GUI enable user to deal with the program
- User enter all information about the graph like number of nodes and the edges with their weights.
- Program calculate the signal flow graph with mason's rule and list all forward paths, individual loops & all combination of non-touching loops.
- And show the overall system transfer function.

Data Structure

- Arrays

Class MasonAlgoritm:

In MasonAlgorithm class all information stored like gains of the edges, forward paths, loops, non-touching loops, gains of them and the overall transfer function.

- adjacencyMatrix[][] 2D array : contain information about edges starting node and ending node and edge's weight.
- forwardPaths `ArrayList<ArrayList<Integer>>` contain all forward paths label with their nodes number.
- Loops `ArrayList<ArrayList<Integer>>`: contain all loops label with their nodes number.
- nonTouchingLoops `ArrayList<Integer[]>`: contain all the non-touching loops label with their nodes number.
- forwardPathsGain`ArrayList<Double>`: contain the result gain of each path in the graph.
- loopsGain `ArrayList<Double>`: contain the result gain of each loop in the graph.
- nonTouchingLoopsGain `ArrayList<Double>` : contain the result gain of each non-touching loop in the graph.

Algorithms Used

Class MasonAlgorithm:

Class contain all methods that used to calculate the overall transfer function.

```
private void generateForwardPathsAndLoops(ArrayList<Integer> path, boolean[] visited, int node) {
    path.add(node);
    visited[node] = true;
    // forward path case
    if (path.size() > 1 && node == numOfNodes - 1) {
        addToFP(new ArrayList<>(path));
        return;
    }
    for (int neighbour = 0; neighbour < numOfNodes; neighbour++) {
        if (adjacencyMatrix[node][neighbour] != 0) {
            if (!visited[neighbour]) {
                generateForwardPathsAndLoops(path, visited, neighbour);
                path.remove(path.size() - 1); // here back tracking for new forward path
                visited[neighbour] = false;
                // loop case
            } else {
                int index = path.indexOf(neighbour);
                if (index != -1) {
                    List<Integer> temp = path.subList(index, path.size());
                    addToLoops(new ArrayList<Integer>(temp));
                }
            }
        }
    }
}
```

Algorithm Description: -

- Generate all forward paths in the graph from the first node (source) to the last node (sink).
- Generate all loops in the graph.

Note : while generate loops it's repeated while backtracking so method addToLoops only add loops which isn't exist in the loops array.

```
// generate all possible combination of non touching loops
private void generateNonTouching(ArrayList<ArrayList<Integer>> arrList, int n) {
    Set<List<Integer>> hasGenerated = new HashSet<List<Integer>>();
    boolean flag = false;
    ArrayList<ArrayList<Integer>> nextArrList = new ArrayList<ArrayList<Integer>>();
    for (int i = 0; i < arrList.size(); i++) {
        for (int j = i + 1; j < arrList.size(); j++) {
            for (int k = 0; k < arrList.get(j).size(); k++) {
                int x = arrList.get(j).get(k);
                ArrayList<Integer> temp = new ArrayList<Integer>();
                temp.addAll(arrList.get(i));
                temp.add(x);
                if (isNonTouching(temp)) {
                    Collections.sort(temp);
                    if (!hasGenerated.contains(temp)) {
                        hasGenerated.add(temp);
                        flag = true;
                        nextArrList.add(new ArrayList<Integer>());
                        nextArrList.get(nextArrList.size() - 1)
                                .addAll(temp);
                        nonTouchingLoops.add(temp.toArray(new Integer[temp
                                .size()]));
                        nonTouchingLoopGains.add(getNonTouchingGain(temp));
                    }
                }
            }
        }

    }
    if (flag) {
        generateNonTouching(nextArrList, ++n);
    }
}
```

Algorithm Description: -

- Generate all possible combination between our loops and check the non-touching loops of them and store them.

```java
public double getOvalAllTF() {
    double current = 0;
    double delta = 1;
    int e = 1;
    int nth = 2;

    double sum = 0;
    for(int i=0;i<loops.size();i++) {
        sum += loopGains.get(i);
    }
    delta = delta - sum;
    for (int i = 0; i < nonTouchingLoops.size(); i++) {
        if (nonTouchingLoops.get(i).length == nth) {
            delta += e * nonTouchingLoopGains.get(i);
        } else {
            e *= -1;
            ++nth;
            i--;
        }
    }
    double nominatorTerm = 0;
    double deltaN;
    for(int i = 0;i < forwardPaths.size();i++) {
        LinkedList<Integer> indcies = getValidLoopsWithPath(forwardPaths.get(i), loops);
        deltaN = 1;
        double loopSumGains = 0;
        for(int  j = 0;j < indcies.size();j++) {
            loopSumGains += loopGains.get(indcies.get(j));
        }
        deltaN = deltaN - loopSumGains;
        nominatorTerm = nominatorTerm + (forwardPathGains.get(i) * deltaN);
    }
    return nominatorTerm / delta;
}
```

Algorithm Description: -

- ❖ Using Mason's rules of calculating transfer function

The gain formula is as follows:

$$G = \frac{y_{\text{out}}}{y_{\text{in}}} = \frac{\sum_{k=1}^{N} G_k \Delta_k}{\Delta}$$

$$\Delta = 1 - \sum L_i + \sum L_i L_j - \sum L_i L_j L_k + \cdots + (-1)^m \sum \cdots + \cdots$$

where:

- Δ = the determinant of the graph.
- y_{in} = input-node variable
- y_{out} = output-node variable
- G = complete gain between y_{in} and y_{out}
- N = total number of forward paths between y_{in} and y_{out}
- G_k = path gain of the kth forward path between y_{in} and y_{out}
- L_i = loop gain of each closed loop in the system
- L_iL_j = product of the loop gains of any two non-touching loops (no common nodes)
- $L_iL_jL_k$ = product of the loop gains of any three pairwise nontouching loops
- Δ_k = the cofactor value of Δ for the k^{th} forward path, with the loops touching the k^{th} forward path removed. *

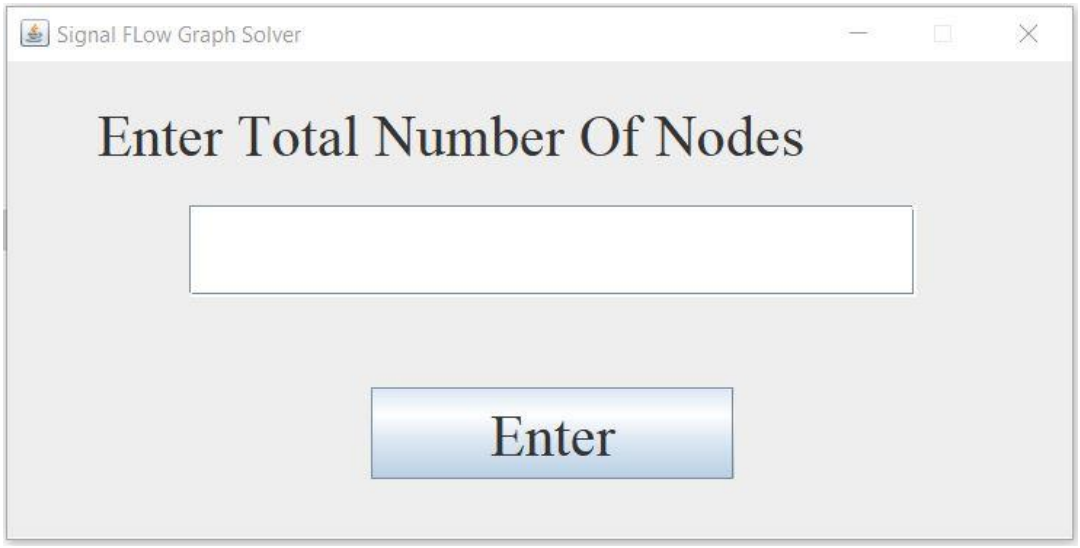
```java
// get the Non touching loops with forward paths
private LinkedList<Integer> getValidLoopsWithPath(ArrayList<Integer> path, ArrayList<ArrayList<Integer>> loops) {

    LinkedList<Integer> ans = new LinkedList<Integer>();

    for(int i=0;i<loops.size();i++) {
        int flag = 0;
        for(int j = 0;j < path.size();j++) {
            if(loops.get(i).contains(path.get(j))) {
                flag++;
                break;
            }
        }
        if(flag == 0 ) {
            System.out.println("----------------------");
            System.out.println("delta indcies - " + i);
            System.out.println("----------------------");
            ans.add(i);
        }
    }

    return ans;

}
```

```java
    //  getters and setters

    public void setLoops(ArrayList<ArrayList<Integer>> newLoops) {
        this.loops= newLoops;
    }

    public String[] getLoops() {
        String loopsString[] = new String[loops.size()];
        int itr = 0;
        for (ArrayList<Integer> arr : loops) {
            loopsString[itr] = "";
            for (int i = 0; i < arr.size(); i++) {
                loopsString[itr] += (arr.get(i) + 1) + " ";
            }
            itr++;
        }
        return loopsString;
    }

    public void setNonTouchingloops(ArrayList<Integer[]> newNonTouchingloops) {
        this.nonTouchingLoops= newNonTouchingloops;
    }
```

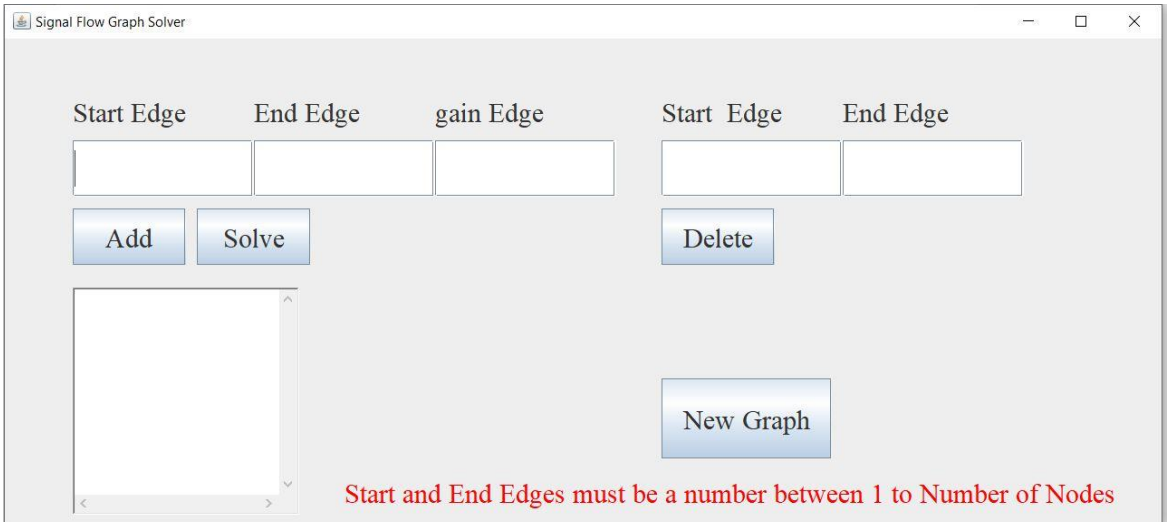
```java
public void setNonTouchingloops(ArrayList<Integer[]> newNonTouchingloops) {
    this.nonTouchingLoops= newNonTouchingloops;
}

public String[] getNonTouchingloops() {
    String[] temp = getLoops();
    String nonString[] = new String[nonTouchingLoops.size()];
    int itr = 0;
    for (Integer[] arr : nonTouchingLoops) {
        nonString[itr] = "";

        if (arr.length > 0)
            nonString[itr] += temp[arr[0]];

        for (int i = 1; i < arr.length; i++)
            nonString[itr] += " , " + temp[arr[i]];

        itr++;
    }
    return nonString;
}

public void setForwardPaths(ArrayList<ArrayList<Integer>> newForwardPaths) {
    this.forwardPaths= newForwardPaths;
}


public String[] getForwardPaths() {
    String fbString[] = new String[forwardPaths.size()];
    int itr = 0;
    for (ArrayList<Integer> arr : forwardPaths) {
        fbString[itr] = "";
        for (int i = 0; i < arr.size(); i++) {
            fbString[itr] += (arr.get(i) + 1) + " ";
        }
        itr++;
    }
    return fbString;
}

public void setLoopsGain(ArrayList<Double> inputLoopsGain) {
    this.loopGains = inputLoopsGain;
}
public Double[] getLoopsGain() {
    return loopGains.toArray(new Double[loopGains.size()]);
}

public void setNonTouchingloopsGain(ArrayList<Double> inputNonTouchingloopsGain) {
    this.nonTouchingLoopGains = inputNonTouchingloopsGain;
}
public Double[] getNonTouchingloopsGain() {
    return nonTouchingLoopGains.toArray(new Double[nonTouchingLoopGains.size()]);
}

public void setForwardPathsGain(ArrayList<Double> inputForwardPathsGain) {
    this.forwardPathGains= inputForwardPathsGain;
}
public Double[] getForwardPathsGain() {
    return forwardPathGains.toArray(new Double[forwardPathGains.size()]);
}

public void setAdjacencyMatrix(double[][] inputAdjacencyMatrix) {
    this.adjacencyMatrix= inputAdjacencyMatrix;
}
```

User Guide

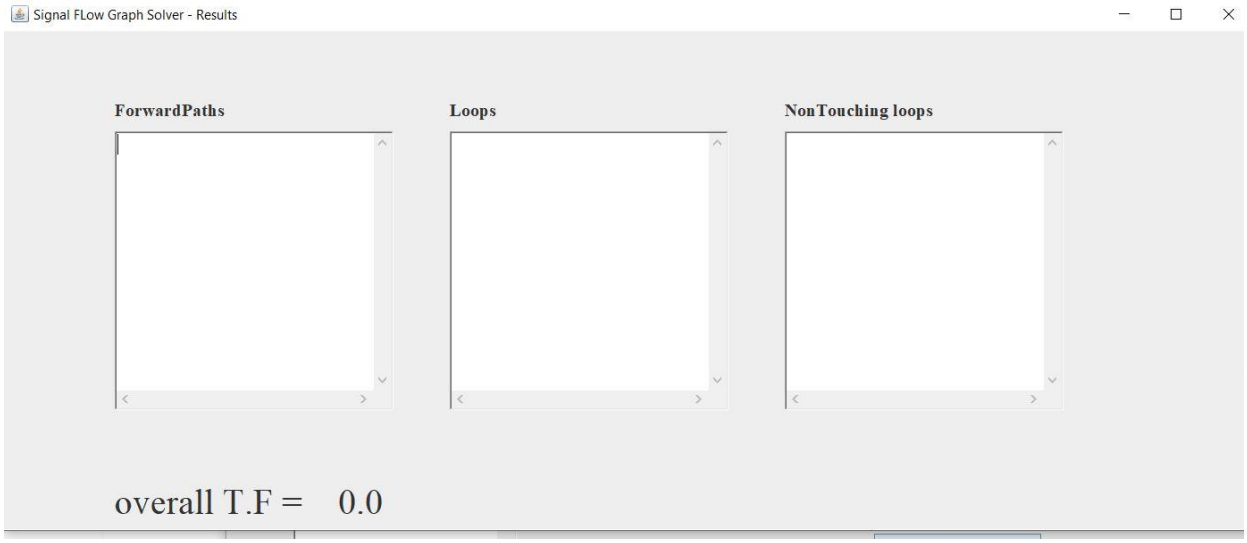
- First must Enter the total Number of nodes in the graph.



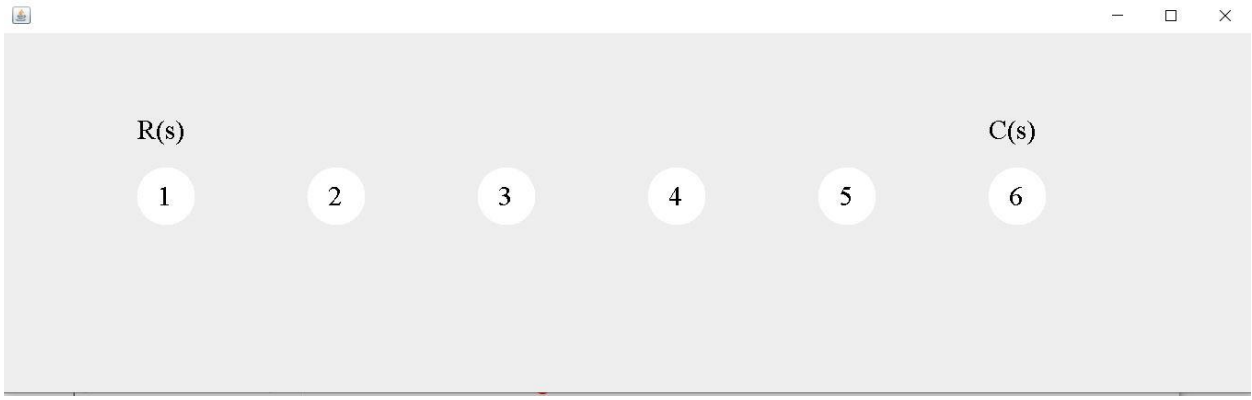
- Second adding edges between nodes so user must identify three things:
  - The starting node of the edge (Start Edge).
  - The ending node of the edge (End Edge).
  - The gain of the Edge
  - And then press the Add button.
- User also can remove any of edges by identify two things:
  - The starting node of the edge (Start Edge).
  - The ending node of the edge (End Edge).
  - And press the Delete button.
- User also can generate new Graph by pressing the New Graph Button.



❖ After Pressing the Solve button

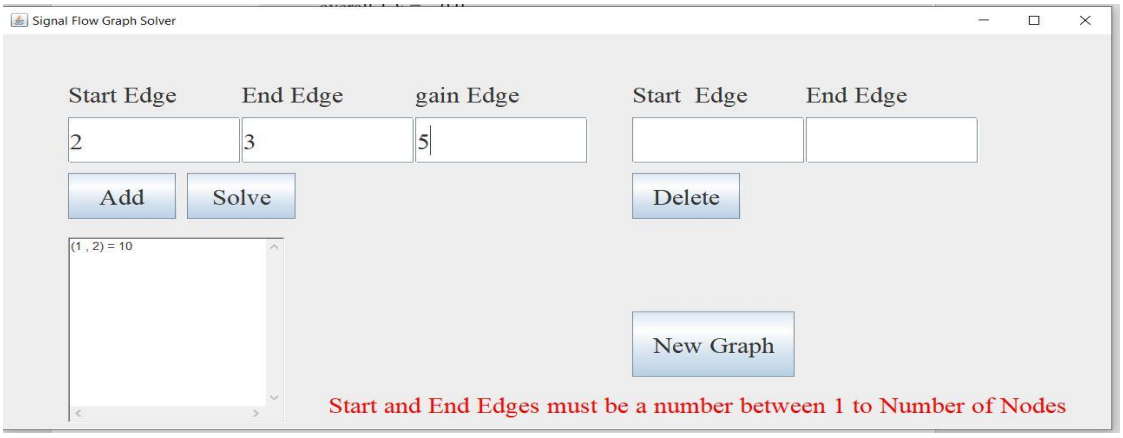


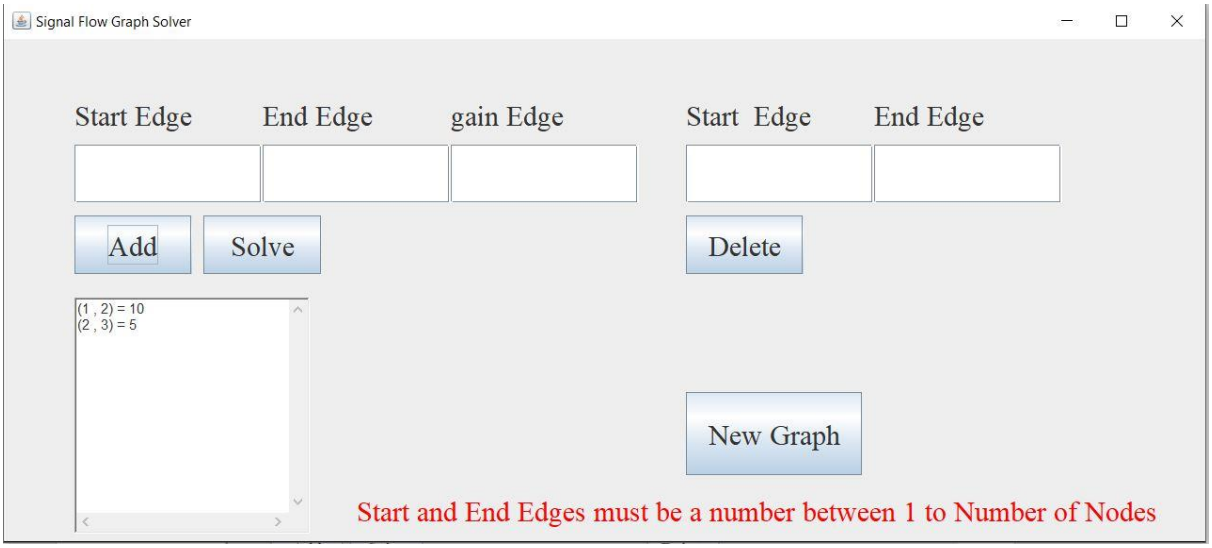
Here will appear all the information about the graph and the overall transfer function and the graph.



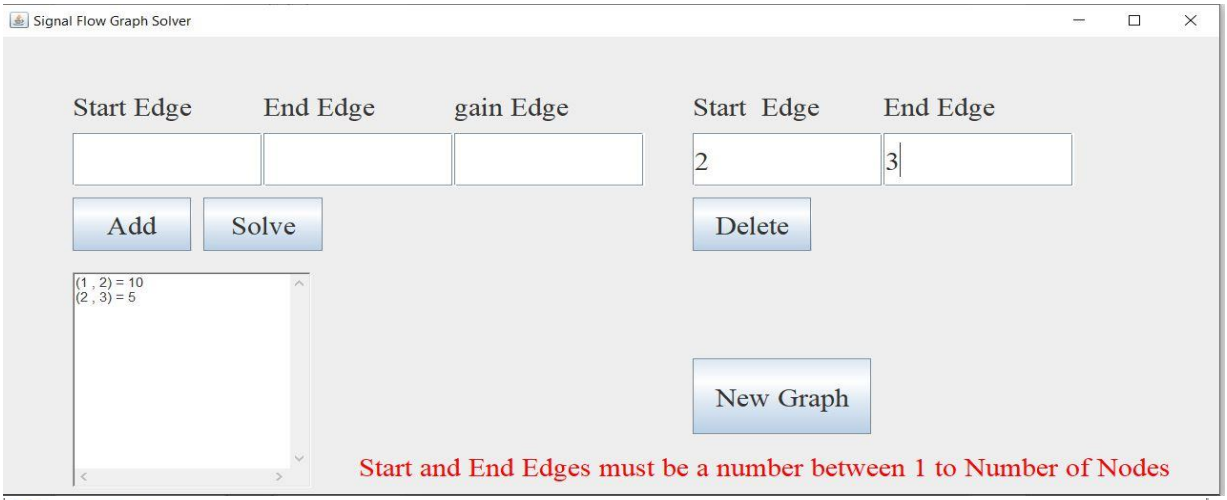
➢ Example of Adding Edge between two nodes:

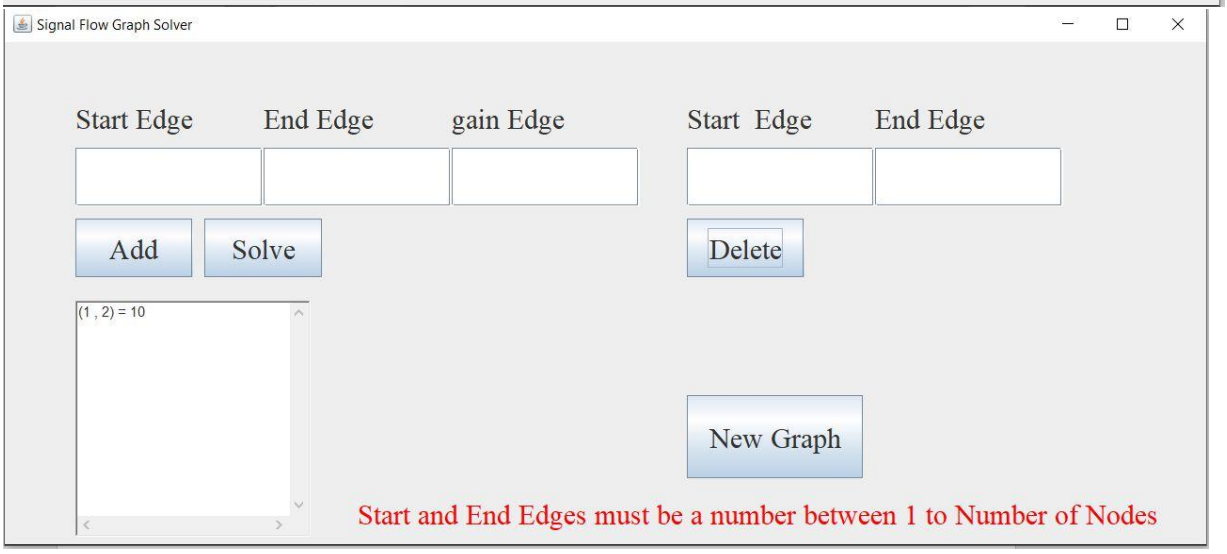
- Write information of the edge in the text boxes then press add





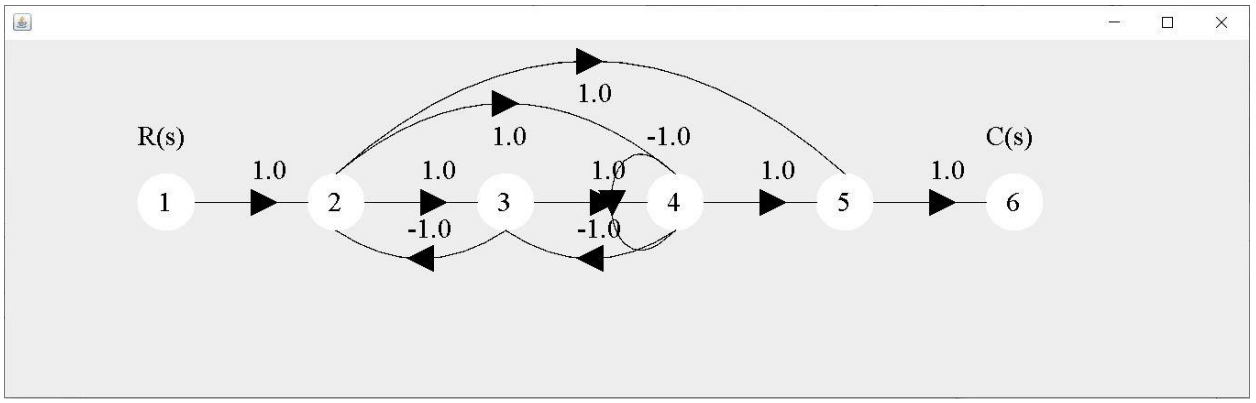
- Example of deleting Edge between two nodes:
  - Write information of the edge in the text boxes in the right and press delete

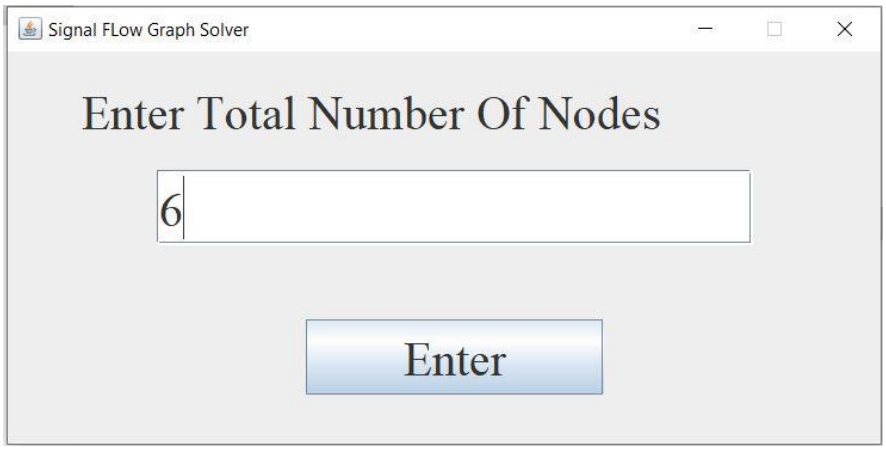


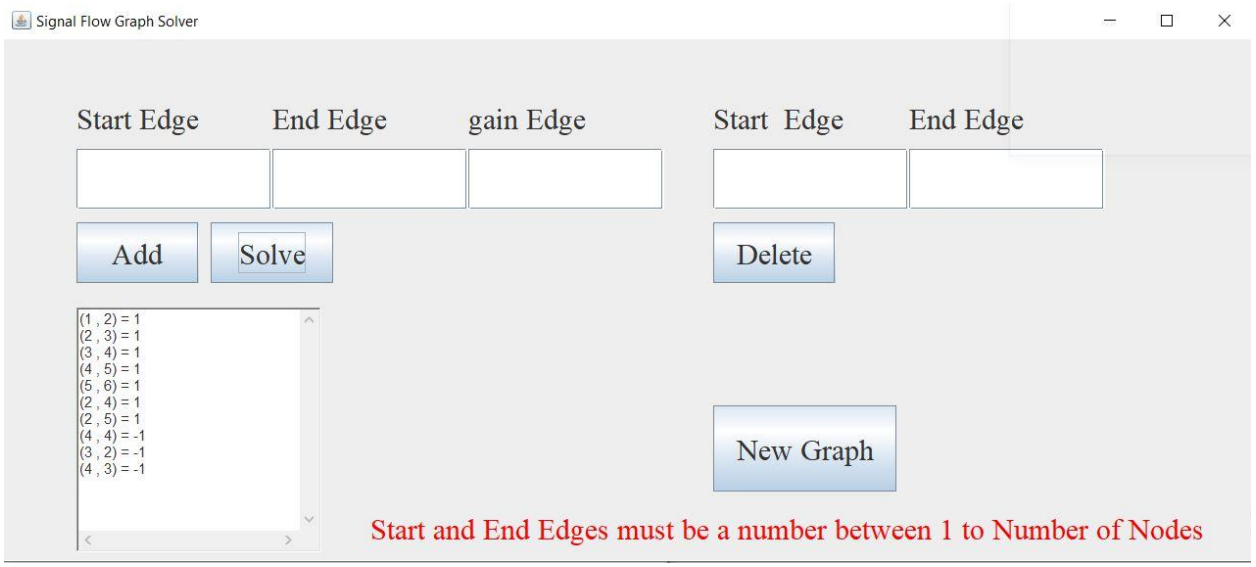


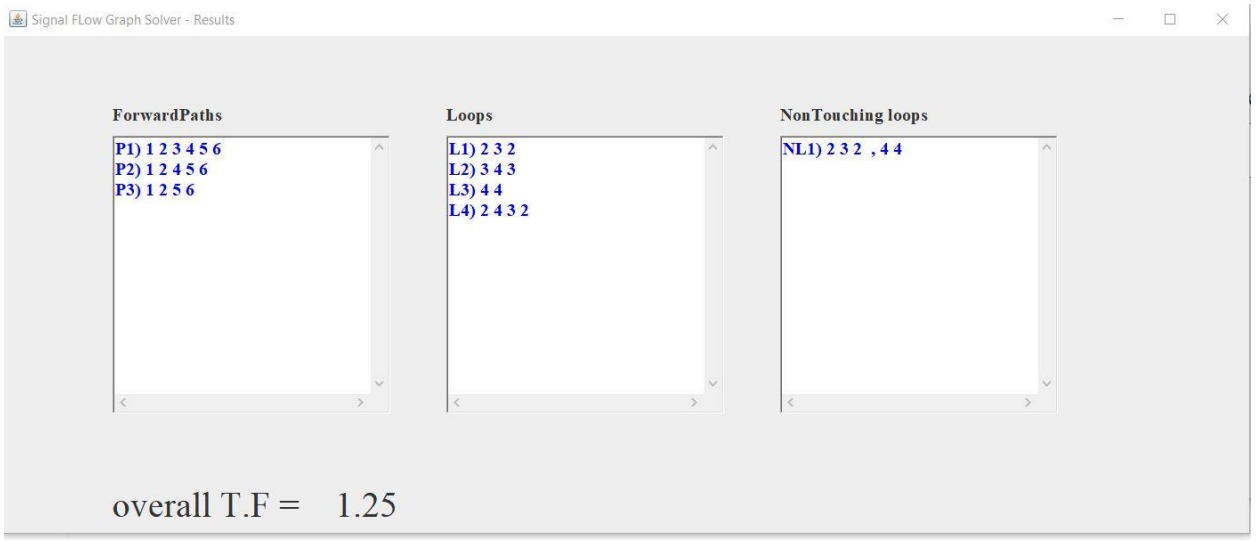
Sample Runs

- First Example

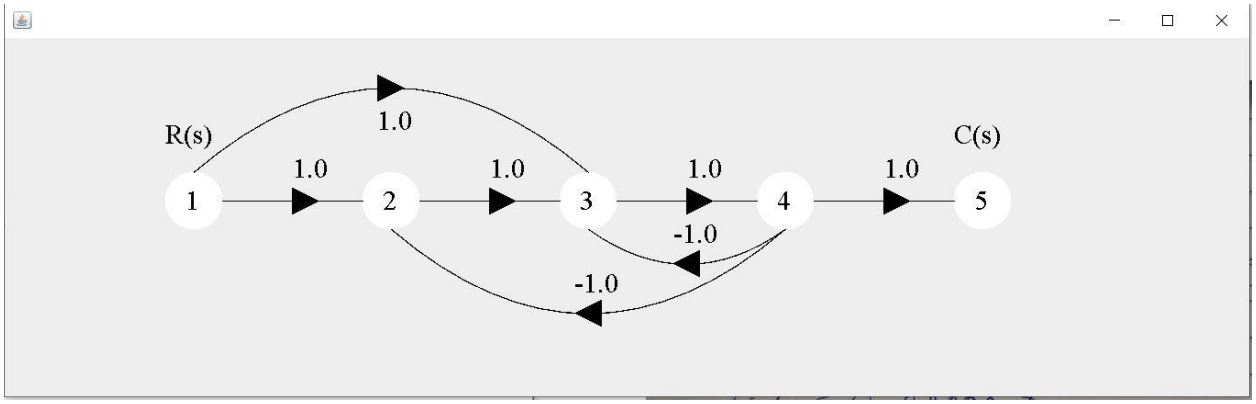


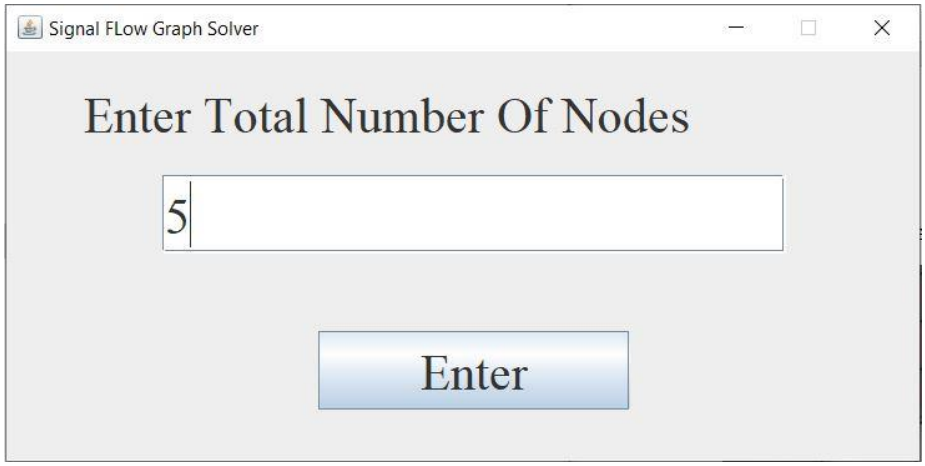


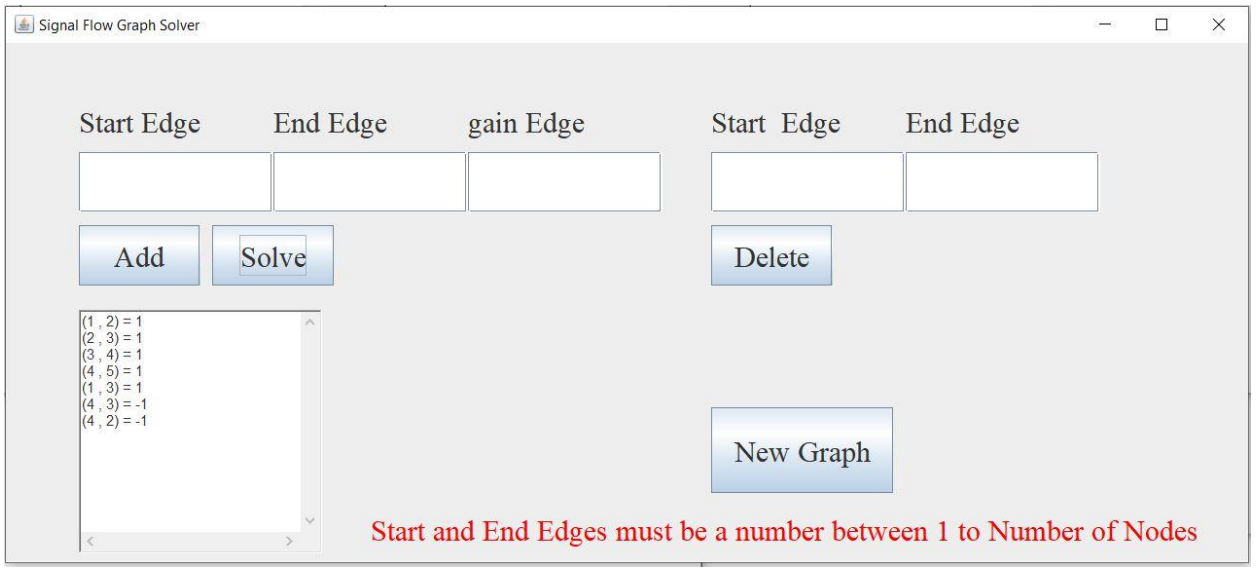


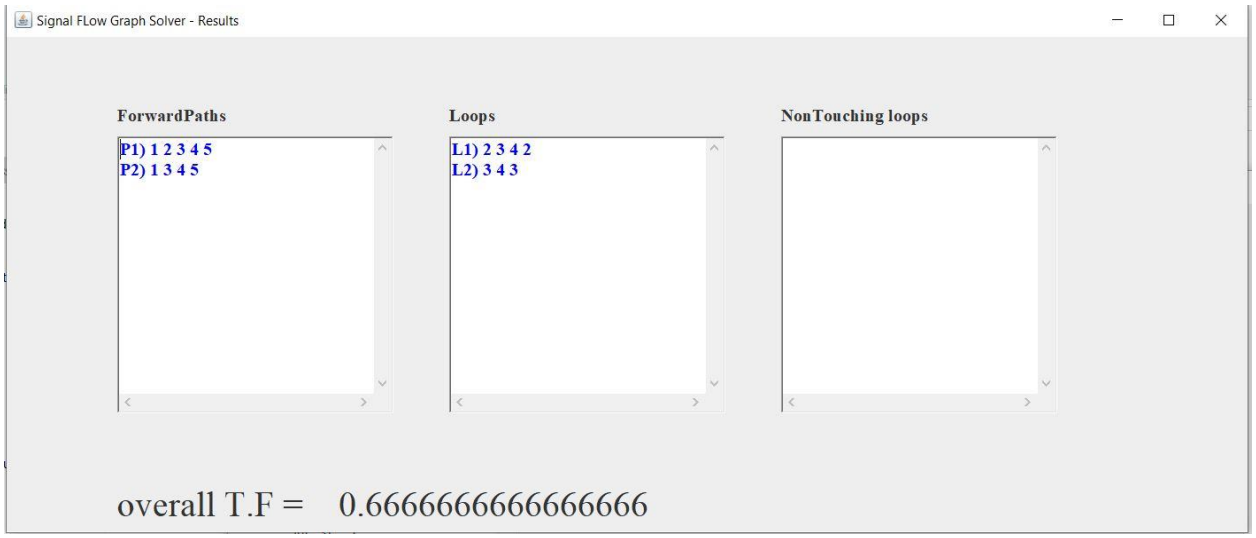


❖ Second Example

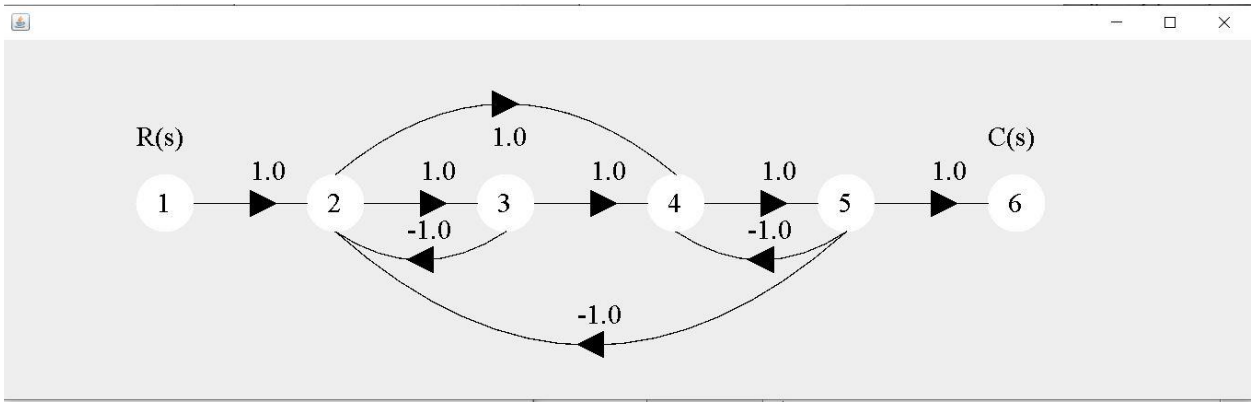


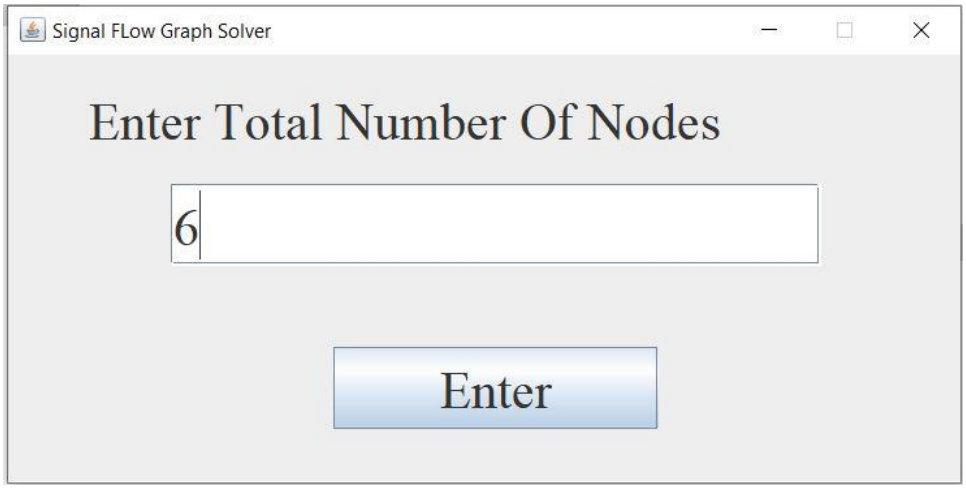


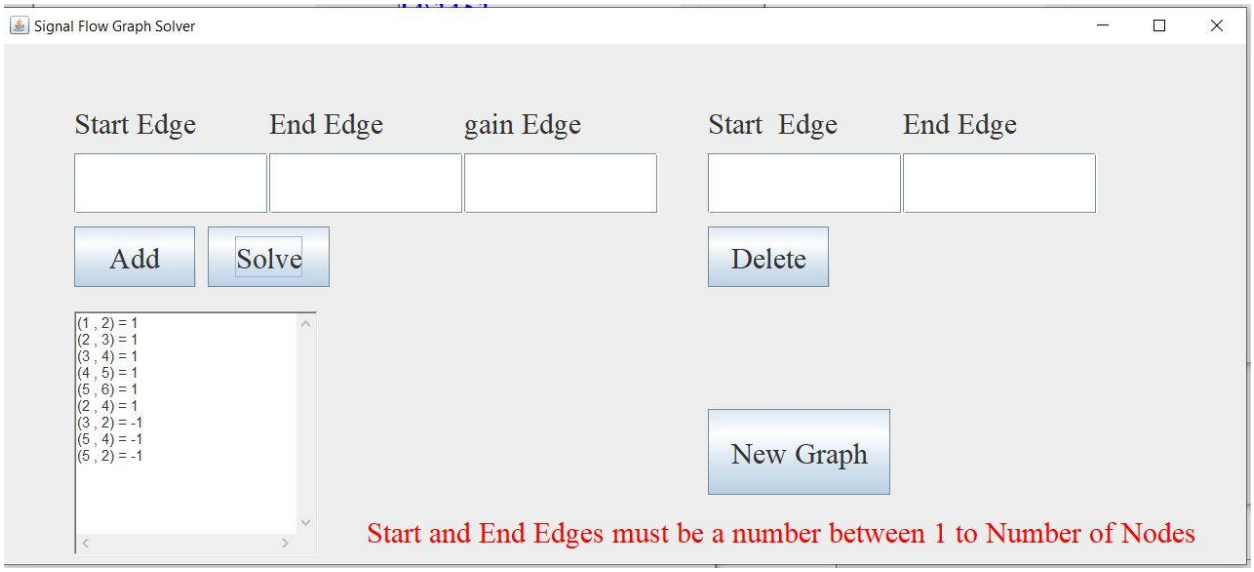

Signal Flow Graph Solver
Start Edge
End Edge
gain Edge
Start Edge
End Edge
Add
Solve
Delete
(1 , 2) = 1
(2 , 3) = 1
(3 , 4) = 1
(4 , 5) = 1
(1 , 3) = 1
(4 , 3) = -1
(4 , 2) = -1
New Graph
Start and End Edges must be a number between 1 to Number of Nodes

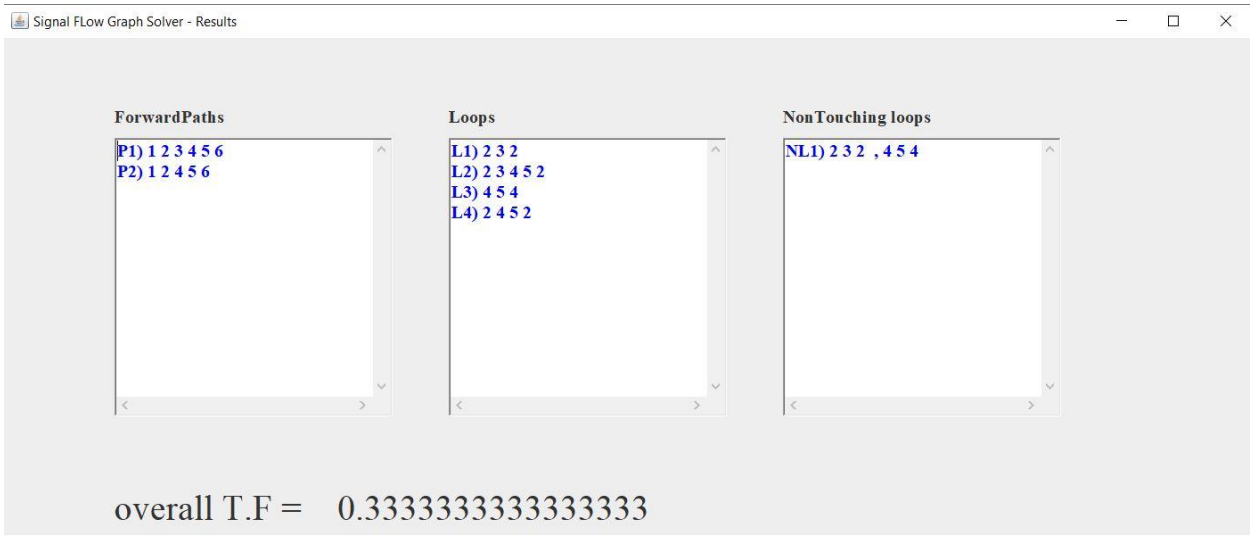

Signal FLow Graph Solver - Results
ForwardPaths
P1) 1 2 3 4 5
P2) 1 3 4 5
Loops
L1) 2 3 4 2
L2) 3 4 3
NonTouching loops
overall T.F = 0.6666666666666666

❖ Third Example

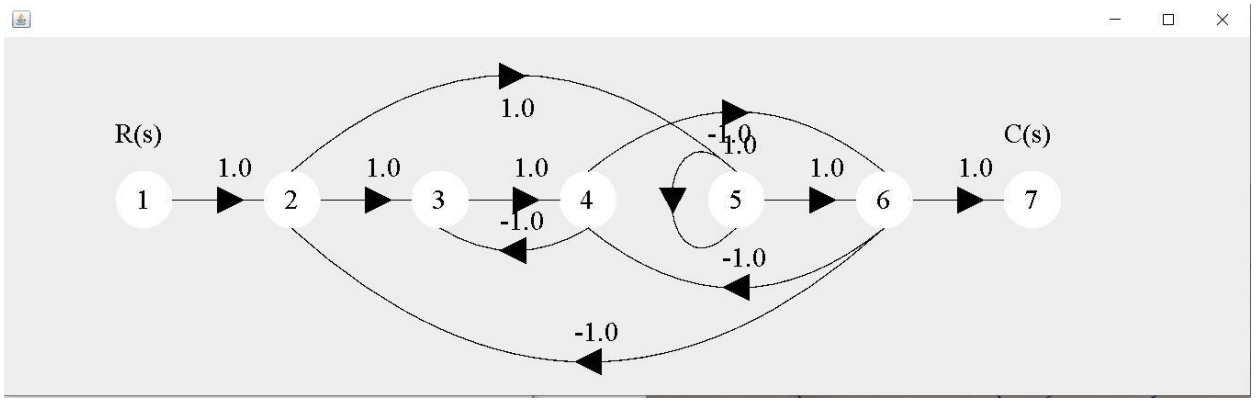


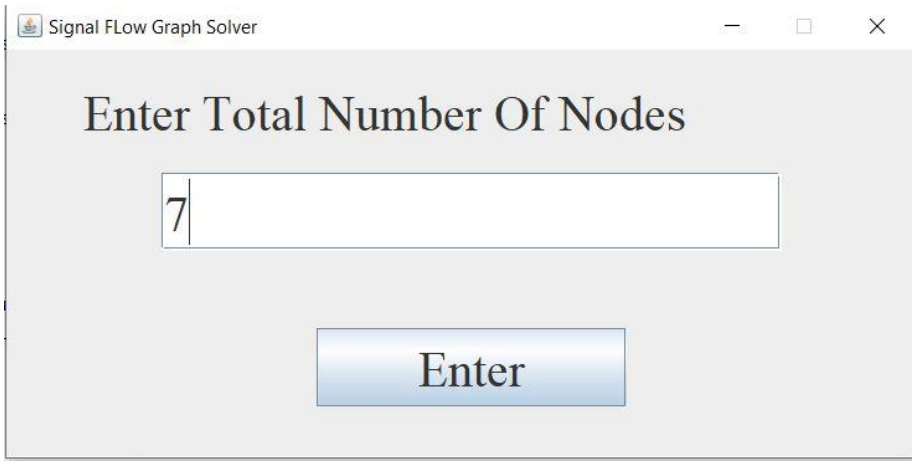


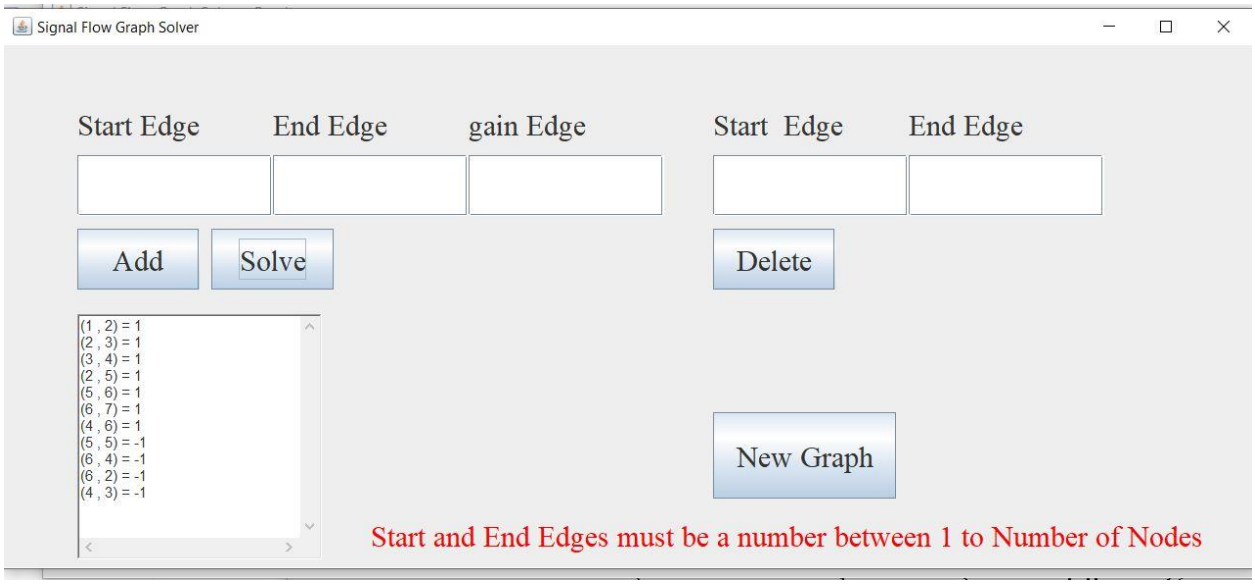


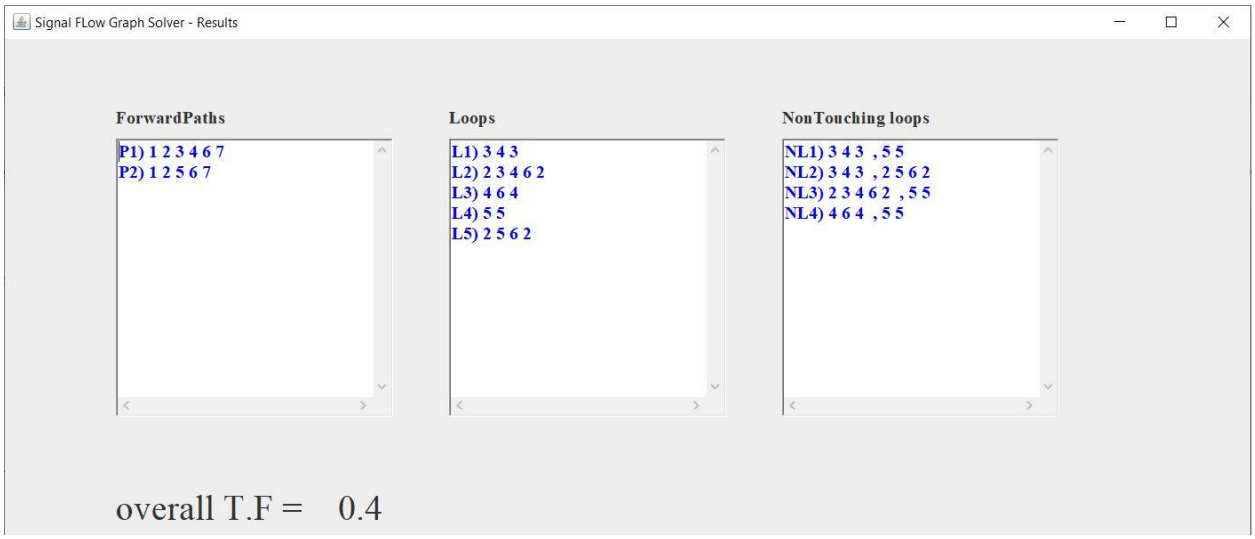


❖ Fourth Example

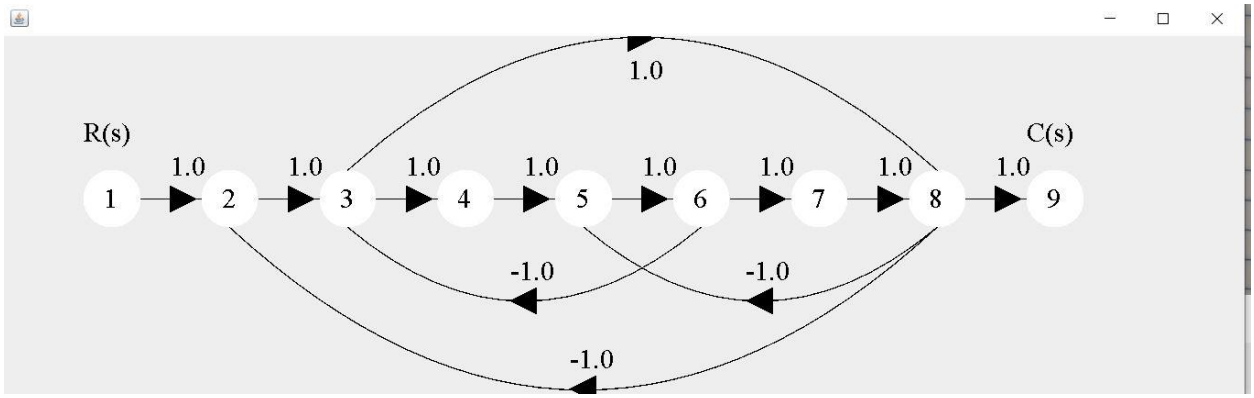


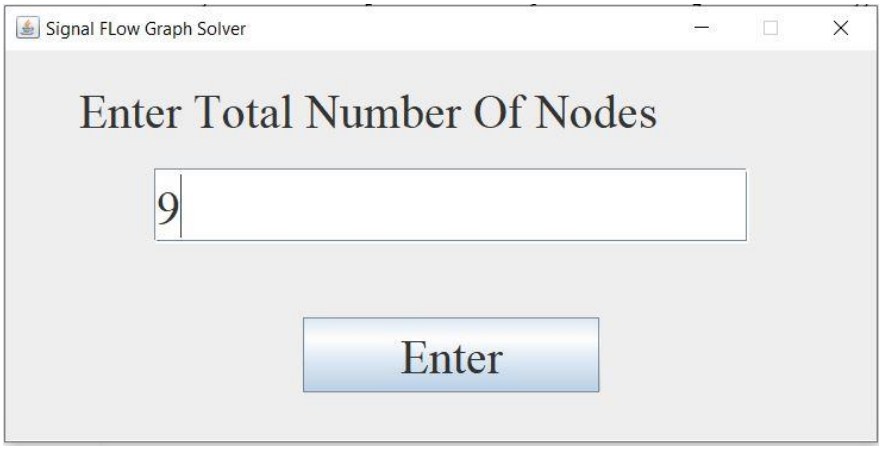


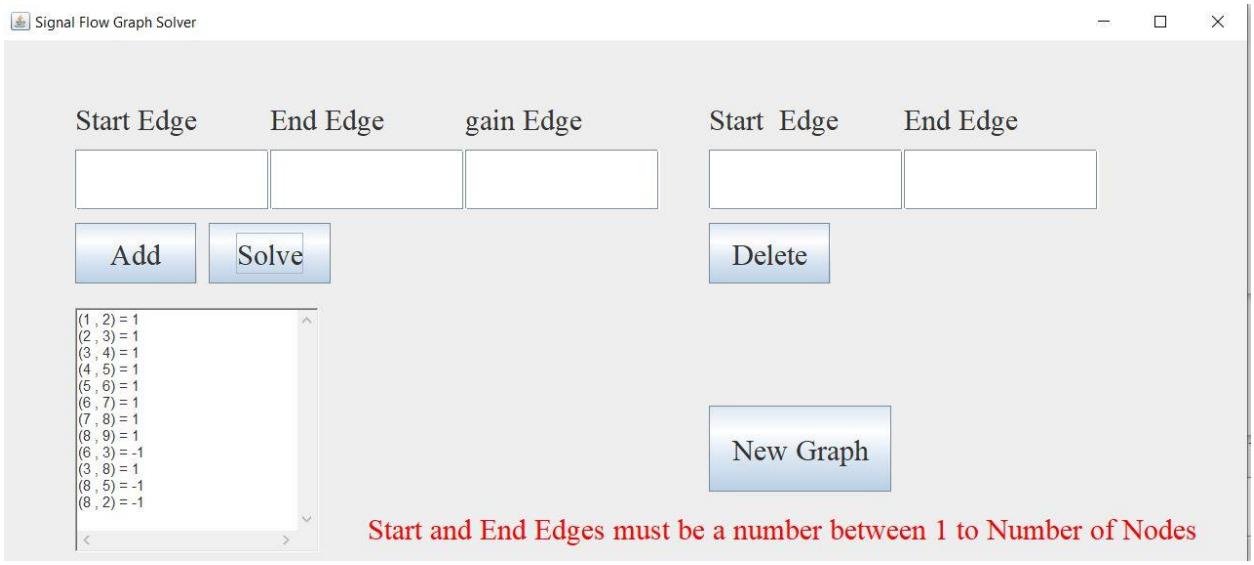

Signal Flow Graph Solver
Start Edge
End Edge
gain Edge
Start Edge
End Edge
Add
Solve
Delete
(1 , 2) = 1
(2 , 3) = 1
(3 , 4) = 1
(2 , 5) = 1
(5 , 6) = 1
(6 , 7) = 1
(4 , 6) = 1
(5 , 5) = -1
(6 , 4) = -1
(6 , 2) = -1
(4 , 3) = -1
New Graph
Start and End Edges must be a number between 1 to Number of Nodes

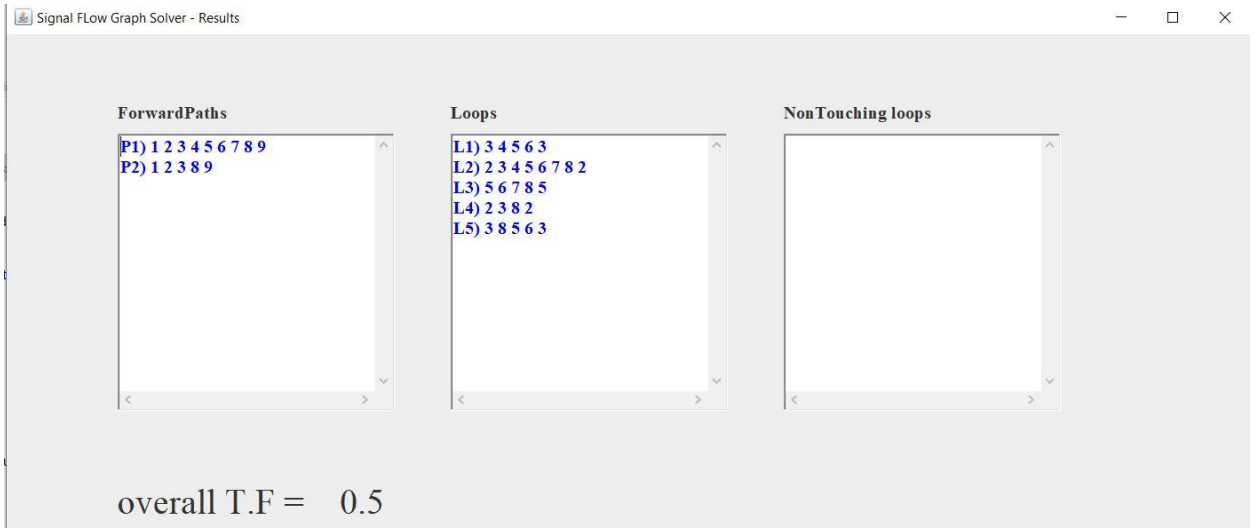

Signal FLow Graph Solver - Results
ForwardPaths
P1) 1 2 3 4 6 7
P2) 1 2 5 6 7
Loops
L1) 3 4 3
L2) 2 3 4 6 2
L3) 4 6 4
L4) 5 5
L5) 2 5 6 2
NonTouching loops
NL1) 3 4 3 , 5 5
NL2) 3 4 3 , 2 5 6 2
NL3) 2 3 4 6 2 , 5 5
NL4) 4 6 4 , 5 5
overall T.F = 0.4

❖ Fifth Example

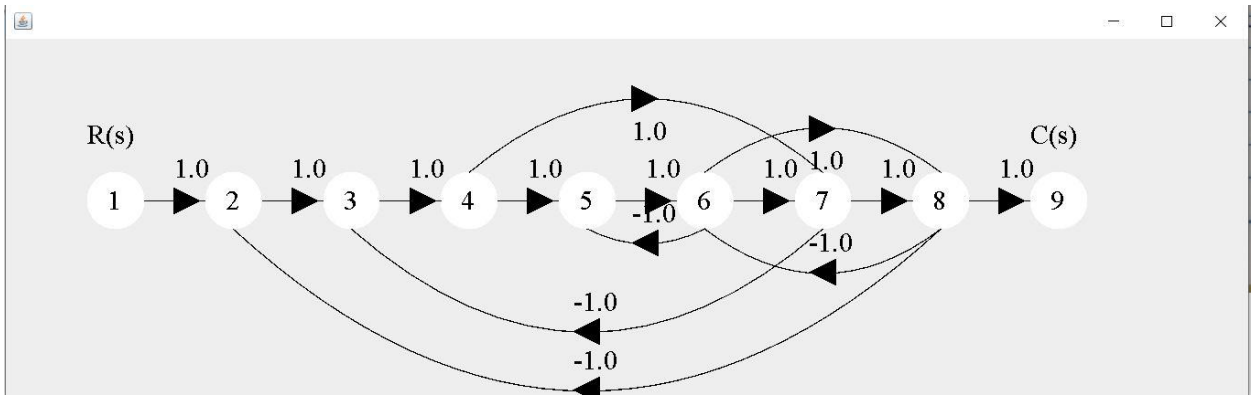


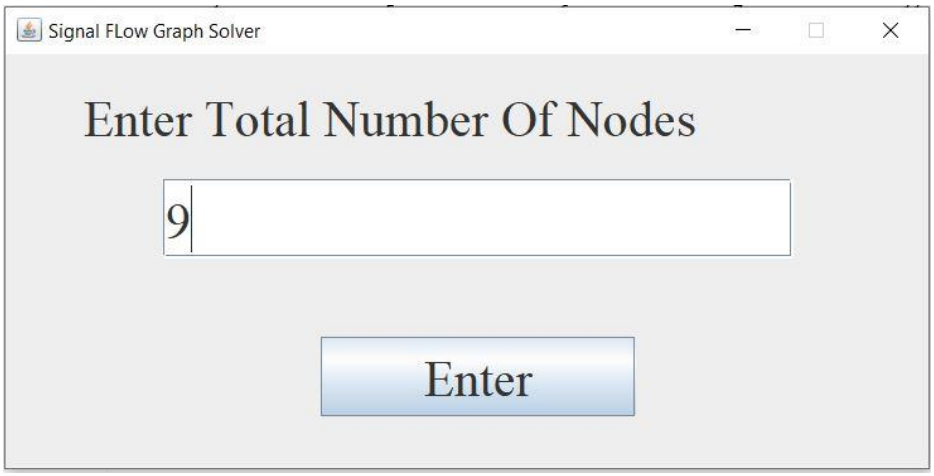


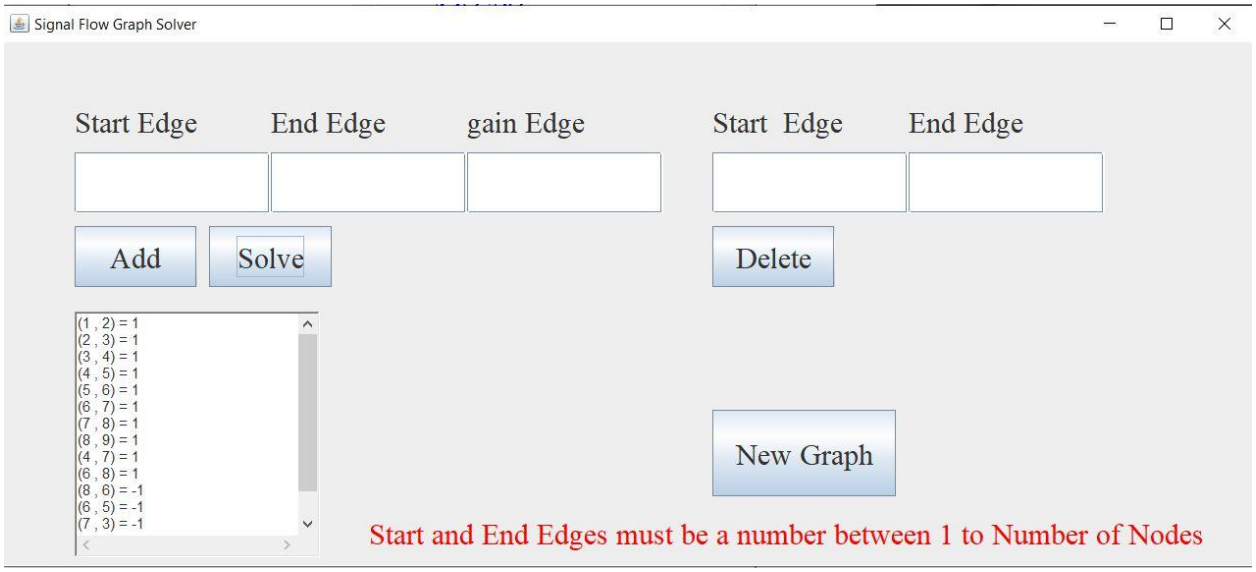


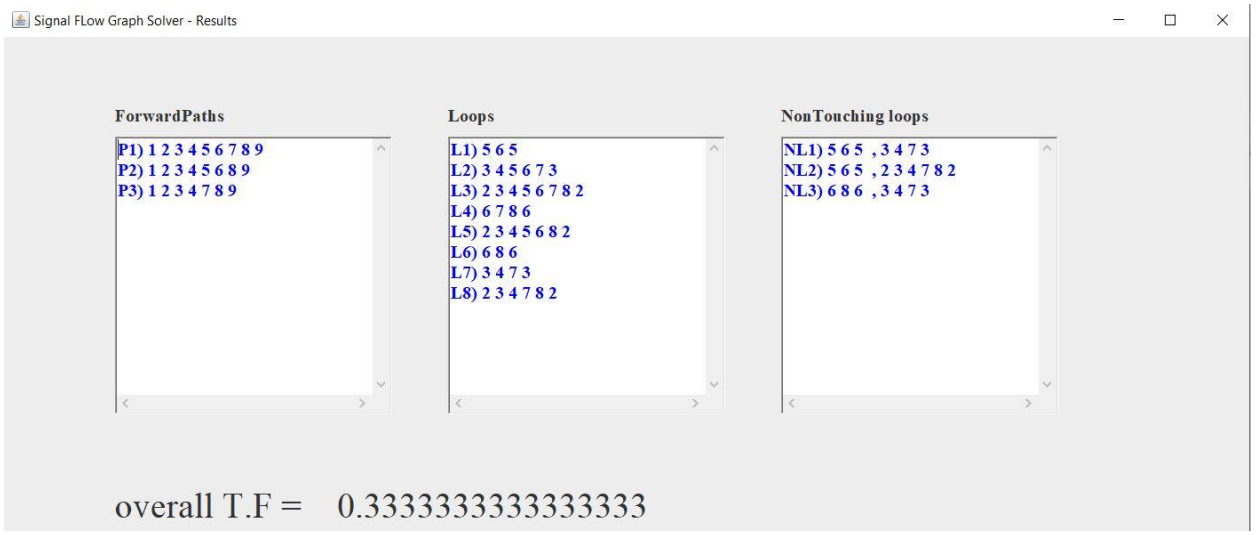


❖ Sixth Example

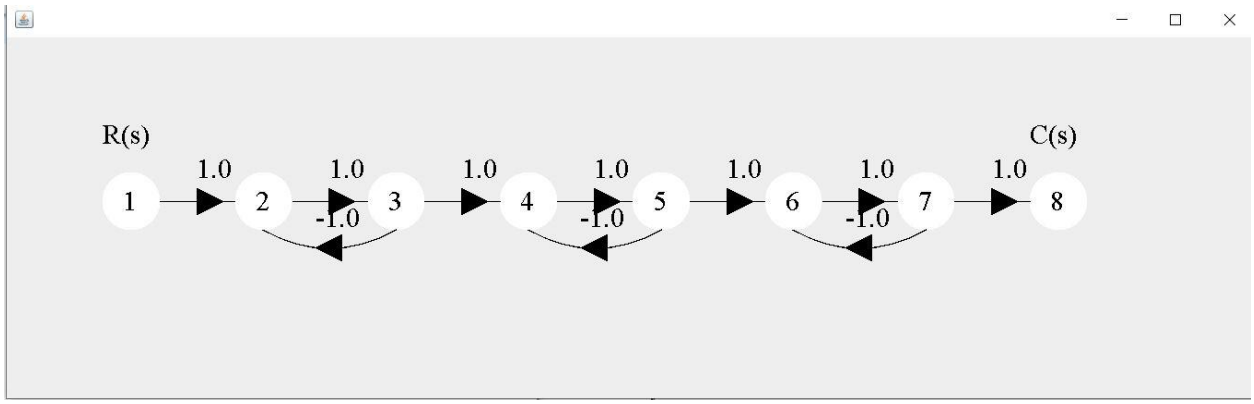


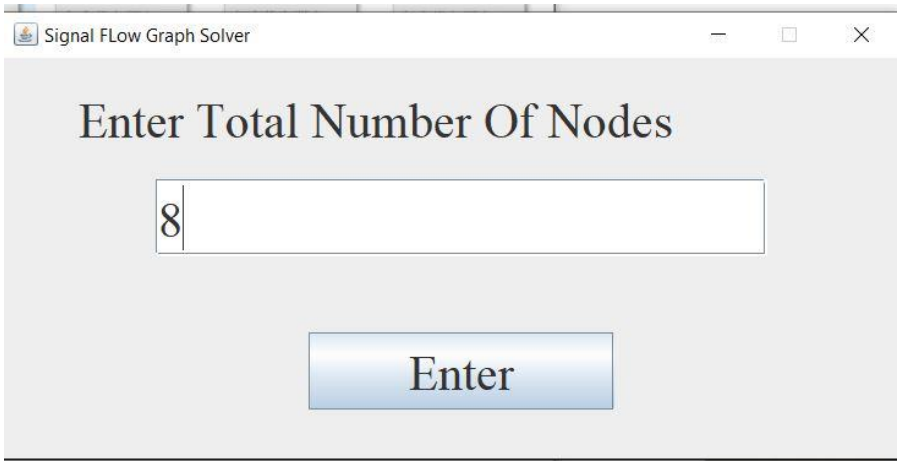


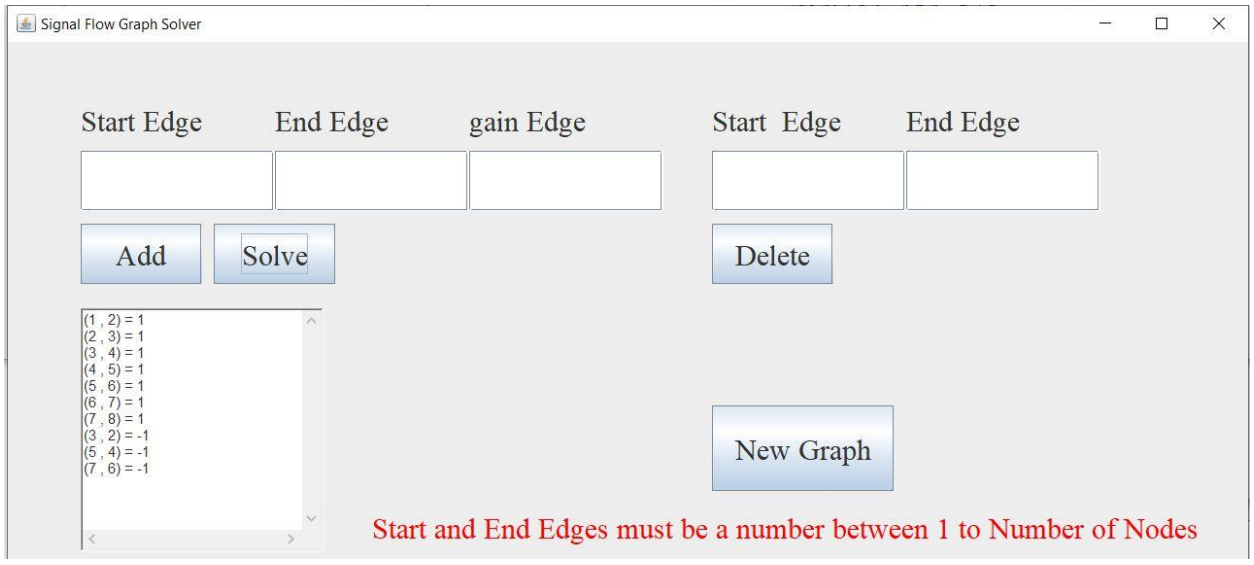

Signal Flow Graph Solver
Start Edge
End Edge
gain Edge
Start Edge
End Edge
Add
Solve
Delete
(1 , 2) = 1
(2 , 3) = 1
(3 , 4) = 1
(4 , 5) = 1
(5 , 6) = 1
(6 , 7) = 1
(7 , 8) = 1
(8 , 9) = 1
(4 , 7) = 1
(6 , 8) = 1
(8 , 6) = -1
(6 , 5) = -1
(7 , 3) = -1
New Graph
Start and End Edges must be a number between 1 to Number of Nodes

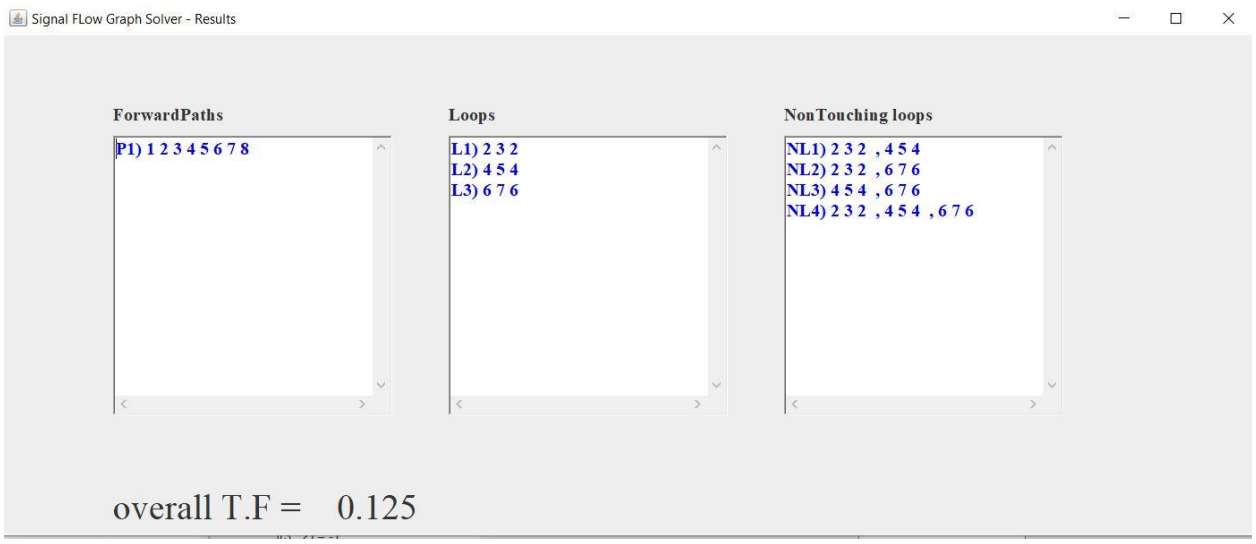

Signal FLow Graph Solver - Results
ForwardPaths
P1) 1 2 3 4 5 6 7 8 9
P2) 1 2 3 4 5 6 8 9
P3) 1 2 3 4 7 8 9
Loops
L1) 5 6 5
L2) 3 4 5 6 7 3
L3) 2 3 4 5 6 7 8 2
L4) 6 7 8 6
L5) 2 3 4 5 6 8 2
L6) 6 8 6
L7) 3 4 7 3
L8) 2 3 4 7 8 2
NonTouching loops
NL1) 5 6 5 , 3 4 7 3
NL2) 5 6 5 , 2 3 4 7 8 2
NL3) 6 8 6 , 3 4 7 3
overall T.F = 0.3333333333333333

❖ Seventh Example

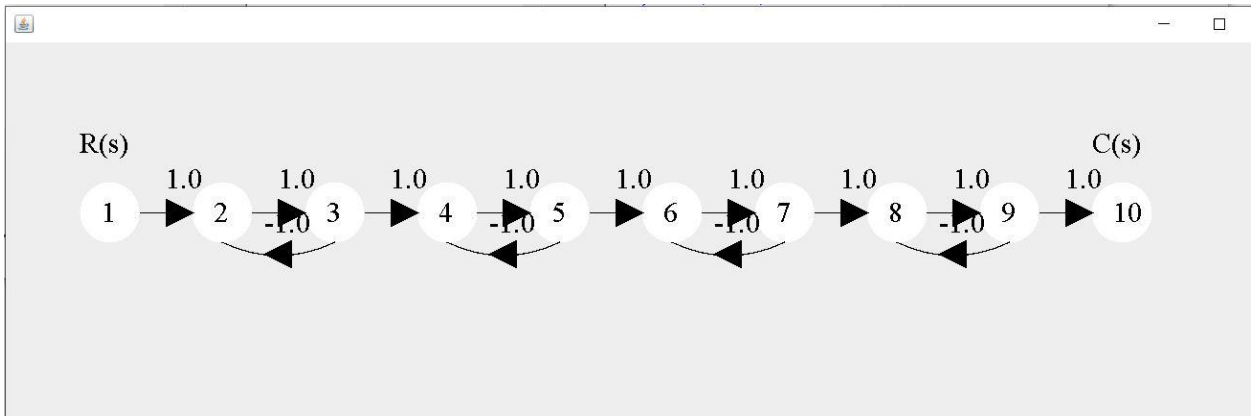


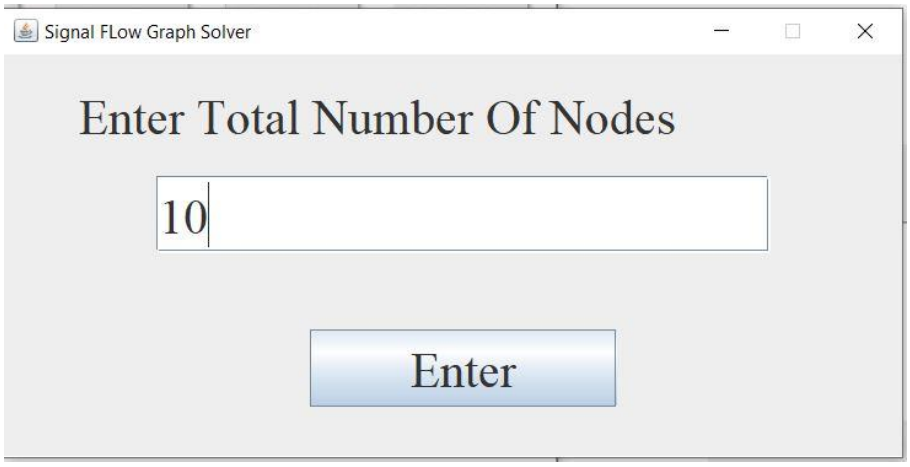


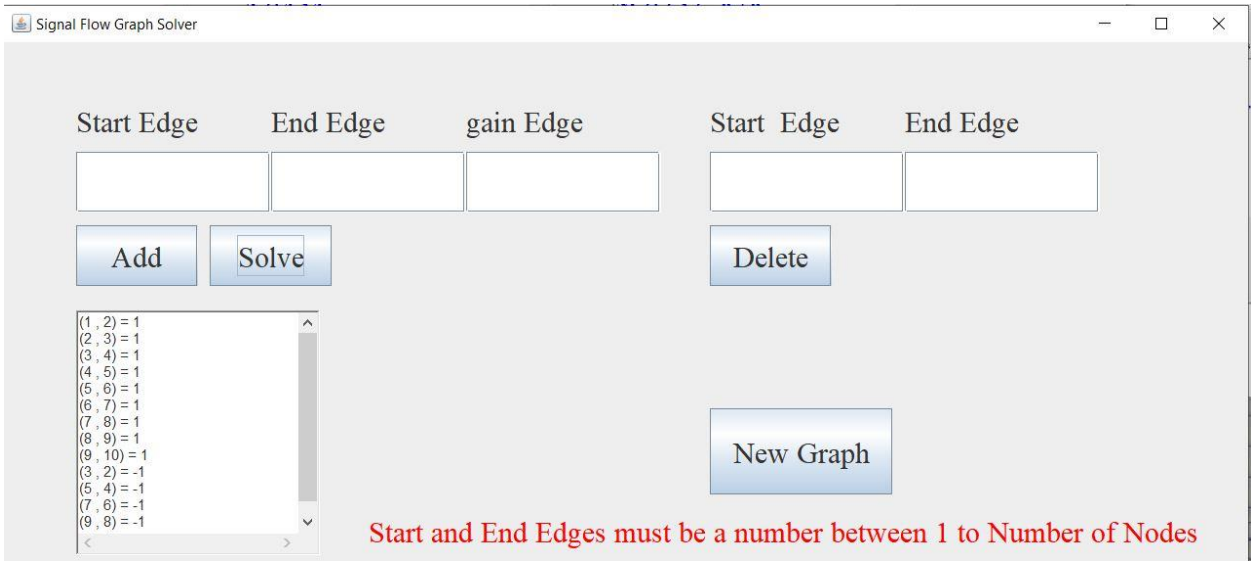


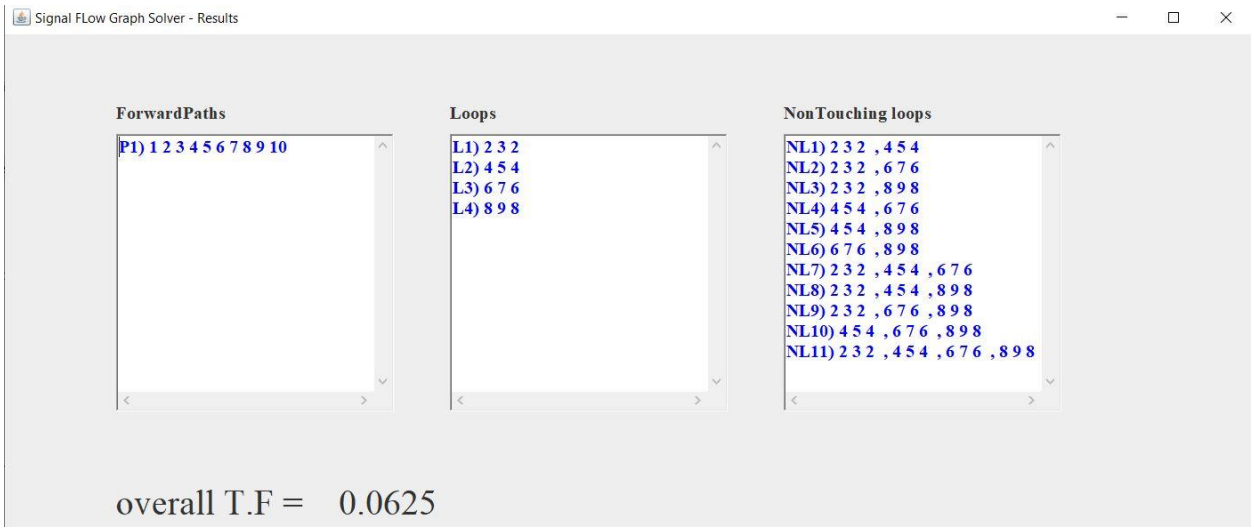


❖ Eight Example






Signal Flow Graph Solver
Start Edge
End Edge
gain Edge
Start Edge
End Edge
Add
Solve
Delete
(1 , 2) = 1
(2 , 3) = 1
(3 , 4) = 1
(4 , 5) = 1
(5 , 6) = 1
(6 , 7) = 1
(7 , 8) = 1
(8 , 9) = 1
(9 , 10) = 1
(3 , 2) = -1
(5 , 4) = -1
(7 , 6) = -1
(9 , 8) = -1
New Graph
Start and End Edges must be a number between 1 to Number of Nodes


Signal FLow Graph Solver - Results
ForwardPaths
P1) 1 2 3 4 5 6 7 8 9 10
Loops
L1) 2 3 2
L2) 4 5 4
L3) 6 7 6
L4) 8 9 8
NonTouching loops
NL1) 2 3 2 , 4 5 4
NL2) 2 3 2 , 6 7 6
NL3) 2 3 2 , 8 9 8
NL4) 4 5 4 , 6 7 6
NL5) 4 5 4 , 8 9 8
NL6) 6 7 6 , 8 9 8
NL7) 2 3 2 , 4 5 4 , 6 7 6
NL8) 2 3 2 , 4 5 4 , 8 9 8
NL9) 2 3 2 , 6 7 6 , 8 9 8
NL10) 4 5 4 , 6 7 6 , 8 9 8
NL11) 2 3 2 , 4 5 4 , 6 7 6 , 8 9 8
overall T.F = 0.0625