Signal Flow Graph

Names/ IDs/

Toqa Alaa Ahmed 14

Nada Aalama Mohamed 55

1) Problem Statement.

Given:

Signal flow graph representation of the system. Assume that total number of nodes and numeric branches gains are given.

Required:

1- Graphical interface.
2- Draw the signal flow graph showing nodes, branches, gains, …
3- Listing all forward paths, individual loops, all combination of n non-touching loops.
4- The values of Δ , Δ_1 , …, Δ_m where m is number of forward paths.
5- Overall system transfer function.

2) Main Features of the program and additional options if exists.

- We can select any node and make it source node or sink node
- We can delete any selected node and if it is part of edge then also this edge will be removed
- When select node to be source node, its color will change to green
- When select node to be sink node , its color will change to cyan

3) Data Structure.

- Edge class: it contain number of start node of edge (from) and number of end node of edge (to) and gain of this edge.
  We can get them by getter methods.
- Array of array list of edges: we use it to save adjacent list of every nodes.

Where the size of this array is equal to number of our nodes and every node have array list.

- Array list of array list of edges: we used it to save paths and individual loops.
  Where the size of main array list is equal to number of paths (or loops) and the size of sub list is equal number of edges which form any path (or loop).
- Array list of array list of array of integer: we use it to save combination of non-touched loops.
- Array of Boolean: it help us to know any node visited or not to either calculate paths or loops.
- Array of double numbers: it contain all deltas either overall delta or deltas of paths.

4) Main modules.
   - We use singleton design pattern in class called base.
     Where base contain all information (array lists of paths & adjacent lists & individuals loops & combination of non-touched loops & deltas).
     And we can use this information in any class by setter and getter methods.
   - We use MVC design pattern to organize our code in 3 packages:

   1) View package: contains the classes related to the Gui.
   2) Control package: contains one class which responsible for transferring data between the view with the model.
   3) Model package: contains all required logic and calculation.

5) Algorithms used.

```
static void calculateLoops(){
    base.setArLoops(new HashSet< ArrayList<Integer> >());
    arrLoops = base.getArLoops();

    base.setLoops(new ArrayList<ArrayList<Edge>>());
    loops = base.getLoops();

    Stack<Integer> s = new Stack<Integer>();

    for(int i = 0 ; i < adjList.length ; ++i){
        if(visited[i] == false ){
            checkLoop( i, s );
        }
    }
}
```

- Calculate loops create two things:

First arranged loops which contain all loops but each loop is arranged to can check if add any loop before or not.

Second array list of loops which contain all actual loops

And then loop on array of adjacent list and take each node to check loop in its array of list.

```
static void checkLoop( int node , Stack<Integer> s ){
    visited[node] = true ;
    s.push(node);

    for(int i = 0 ; i < adjList[node].size() ; ++i){
        int to = adjList[node].get(i).getTo() ;
        if(visited[to] == true){
            addLoop(to ,s);
        }
        else{
            checkLoop( to , s );
        }
    }

    s.pop();
    visited[node] = false ;
}
```

- Check loop take a node and then pass on all adjacent of it and find ‘’to’’ of this edge if it was true this means we go and come back to original node.

Then call add loop else.

- Add loop take repeated node (come back node) and stack which contains all path of this loop except come back node or contain number of each node which pass in loop.

  Then we do for loop to add edges of this loop in array list of edges called cycle.

  Then we sort these number nodes to can check if this Loop added before or not.

```
static void addLoop(int to, Stack<Integer> s) {
    Stack<Integer> temp = new Stack<Integer>() ;
    Stack<Integer> path = new Stack<Integer>() ;
    ArrayList<Integer> test = new ArrayList<Integer>() ;

    path.push(to);
    test.add(to);

    while( s.peek() != to ){
        test.add(s.peek());
        path.push(s.peek());
        temp.push(s.pop());
    }

    while(!temp.empty()){
        s.push( temp.pop() );
    }

    ArrayList<Edge> cycle = new ArrayList<Edge>();

    int f = -1 ; //from
    int t = path.pop(); //to
    int src = t ;//save to in another place to check

    //check if any node have two paths to the same way to form loop
    while(!path.empty()){
        f = t ;
        t = path.pop();

        for(int i = 0; i < adjList[f].size(); ++i){
            if(adjList[f].get(i).getTo() == t ){
                cycle.add(adjList[f].get(i));
                break ;
            }
        }
```

```
            }
        }

        //get the second repeated loop node from previous node
        for(int i=0 ; i < adjList[t].size() ; ++i){
            if(adjList[t].get(i).getTo() == src ){
                cycle.add(adjList[t].get(i));
                break ;
            }
        }

        Collections.sort(test);

        //check if this loop exists in our loops or not
        if(!arrLoops.contains(test)){
            arrLoops.add(test);
            loops.add(cycle);
        }

        base.setLoops(loops);
        base.setArLoops(arrLoops);
    }


public static double calculateTF(int source, int sink) {
        Base base = Base.getInstance(null);
        Loops.calculateLoops();
        Paths.calculatePaths(source, sink);
        Delta.calculateDelta();
        double TF = 0 ;
        for (int i = 1; i < base.getDeltas().length; i++) {
            TF += base.getDeltas()[i] * Paths.pathGain(i-1);
        }
        return TF / base.getDeltas()[0];
}
```

- Calculate TF : take all calculation (loops & deltas & paths) where

Mason's Rule:

- The transfer function, *C(s)/R(s),* of a system represented by a signal-flow graph is;

$$\frac{C(s)}{R(s)} = \frac{\sum_{i=1}^{n} P_i \Delta_i}{\Delta}$$

```
static void calculatePaths(int source, int sink){
    base.setPaths(new ArrayList<ArrayList<Edge>>());
    paths = base.getPaths();
    LinkedList<Integer> pathNodes = new LinkedList<Integer>();
    pathNodes.add(source);
    checkPath(source, sink, pathNodes);
    base.setDeltas(new double[paths.size() + 1]);
}

static void checkPath(int source ,int sink , LinkedList<Integer> pathNodes){
    //check finish and arrive to sink node
    if (source == sink){
        addPath(pathNodes);
        return ;
    }

    for (int j = 0; j < adjList[source].size(); j++) {
        int to = adjList[source].get(j).getTo();
        if (visited[to] == false){
            pathNodes.add(to);
            visited[to] = true ;
            checkPath(to, sink, pathNodes);
            visited[to] = false ;
            pathNodes.removeLast();
        }
    }

}
```

- Calculate paths: it create paths which is array of list of array of list & set deltas double array of size of paths+1
- Check path: take as parameters source and sink nodes then do for loop and recursion on adjacent list of each called source node ,then if source equal to sink this create path and call add path method

```
static void addPath(LinkedList<Integer> pathNodes)
{
    final long uniquePath = pathNodes.stream().distinct().count();
    //check to not add new path result in self_loop
    if (uniquePath == pathNodes.size()) {
    paths.add(new ArrayList<Edge>());
    Iterator<Integer> i = pathNodes.iterator();
    int from , to =i.next();
    while(i.hasNext())
    {
        from = to;
        to = i.next();

        for (int j = 0; j < adjList[from].size(); j++) {
            if(adjList[from].get(j).getTo() == to) {
                paths.get(paths.size()-1).add(adjList[from].get(j));
            }
        }
    }
    base.setPaths(paths);
    }
}

static double pathGain(int i)
{
    double gain = 1 ;
    for (int j = 0; j < paths.get(i).size(); j++) {
        gain *= paths.get(i).get(j).getGain();
    }

    return gain ;
}
```

- Add path: first check unique path to prevent calculate self-loops in any path.
  Then do iterator on linked list of paths to get every edge form this path and add at end.
- Path gain: It is equal to multiplication of all gain of edges.

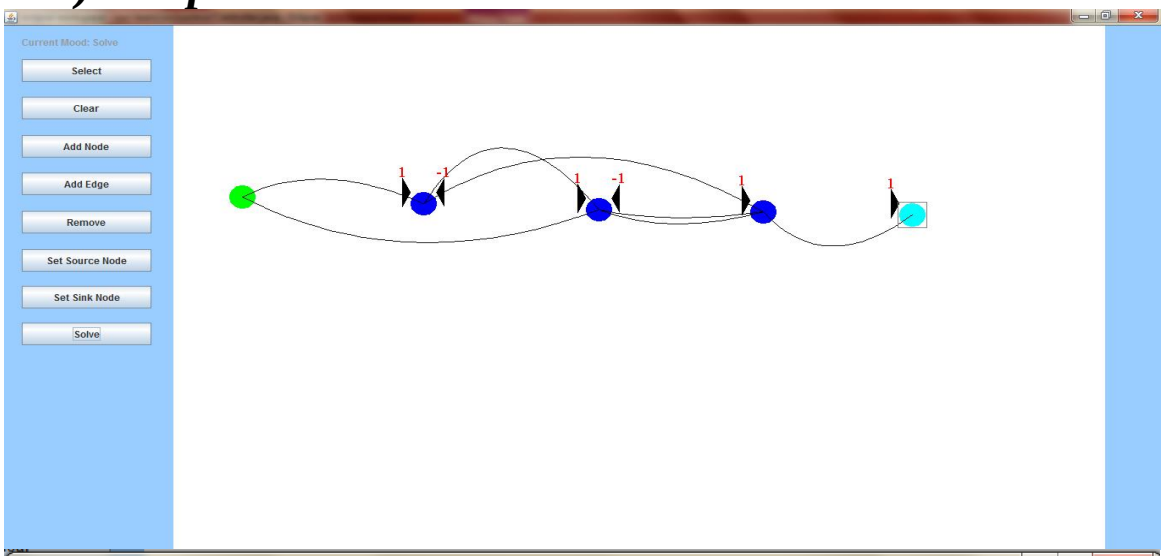
```
private static double getGain(int[] takenCycles)
{
    double gain  = 1 ;

    for (int i = 0; i < takenCycles.length; i++) {
        for (int j = 0; j < loops.get(takenCycles[i]).size(); j++) {

            gain *= loops.get(takenCycles[i]).get(j).getGain();
        }
    }
    return gain ;
}
```

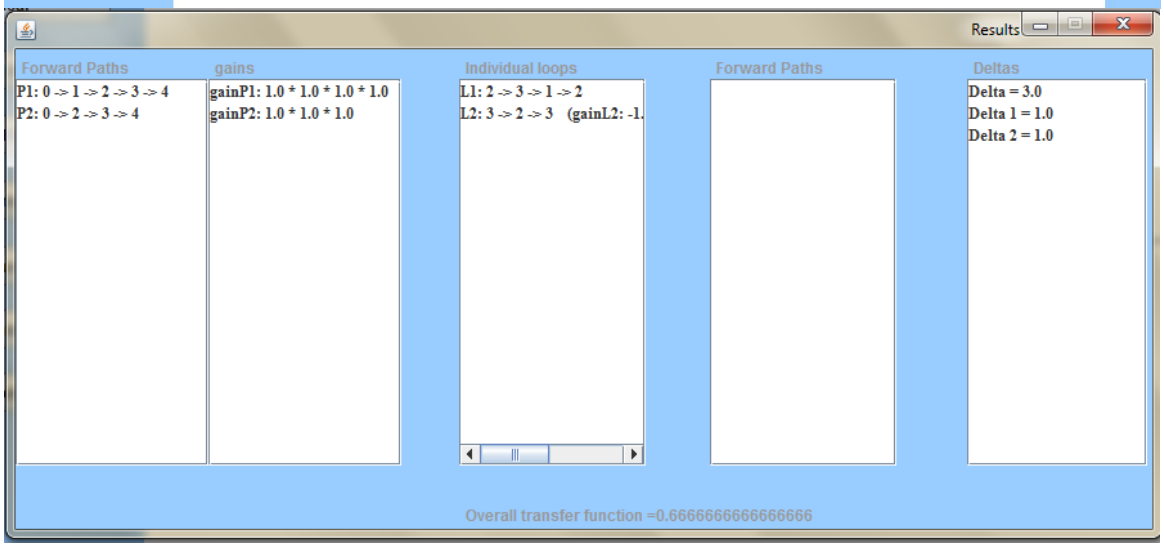
- Get gain: calculate the gain of multiply loops which are combination of non-touched loops.

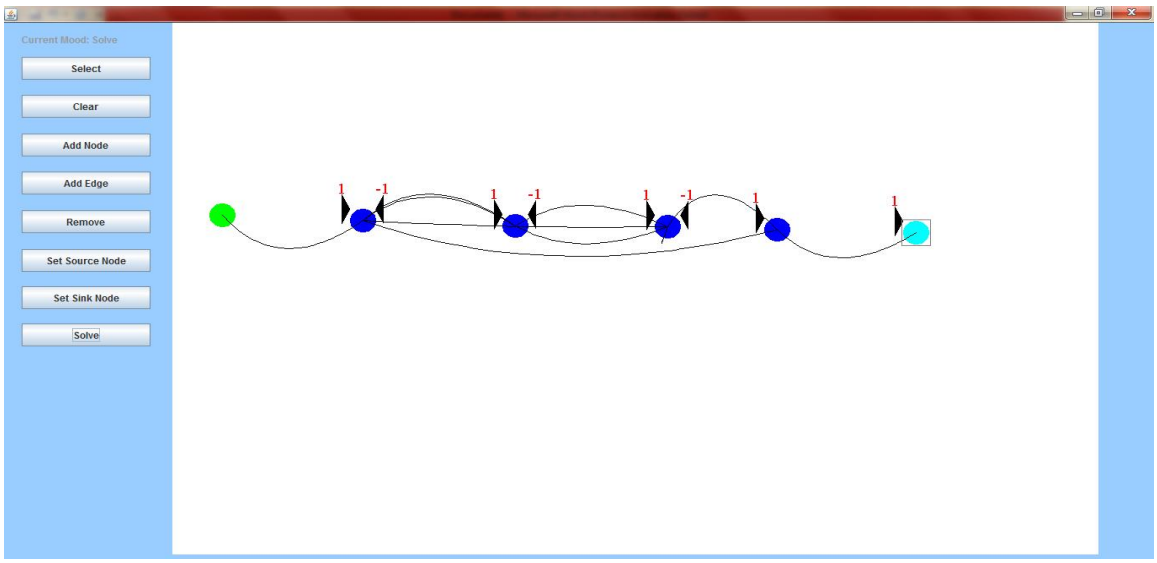
```
private static boolean isNonTouching(int cycle , int[] takenCycles , int index) {
       for (int i = 0; i < index; i++) {
           if(isTwoTouchedLoop(loops.get(cycle),loops.get(takenCycles[i])))
               return false;
       }
       return true;
   }

private static boolean isTwoTouchedLoop( ArrayList<Edge> loop1 , ArrayList<Edge> loop2 ){
       Set<Integer> nodes = new HashSet<Integer>();

       //put all nodes in loop1 in set
       for(int i=0 ; i < loop1.size() ; ++i){
           nodes.add(loop1.get(i).getTo());
       }
       //check if any node in loop one equal any node in loop2
       for(int i=0 ; i < loop2.size() ; ++i){
           if( nodes.contains(loop2.get(i).getTo())){
               return true ;
           }
       }
       //if we finish for loop and didn't return any thing, then they are be non touched
       return false ;
   }
```

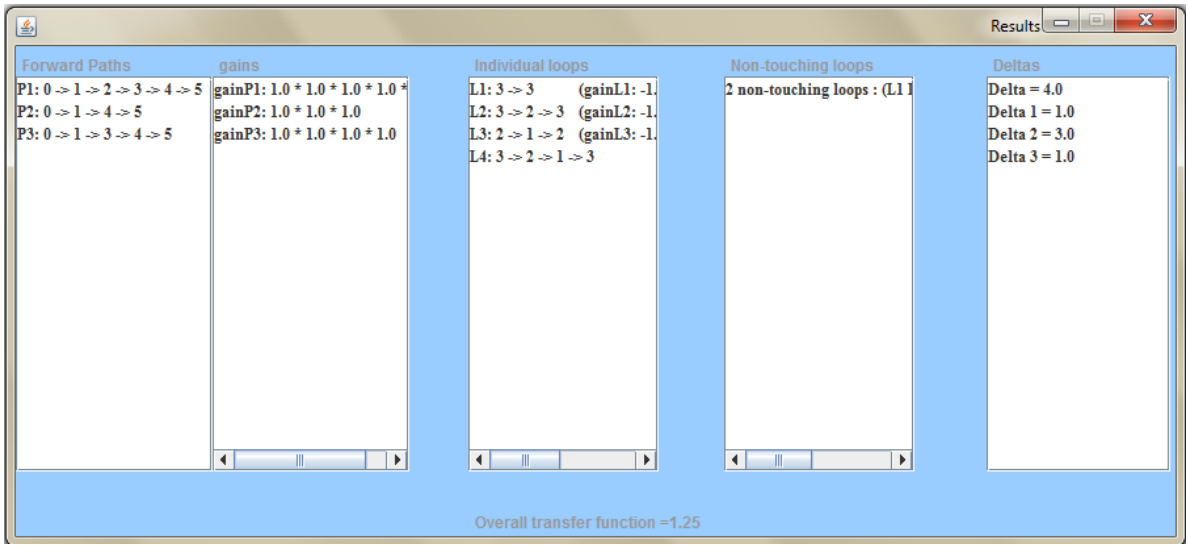
- This check if taken loops be non-touched with each other or not to can calculate non-touched loops of any path

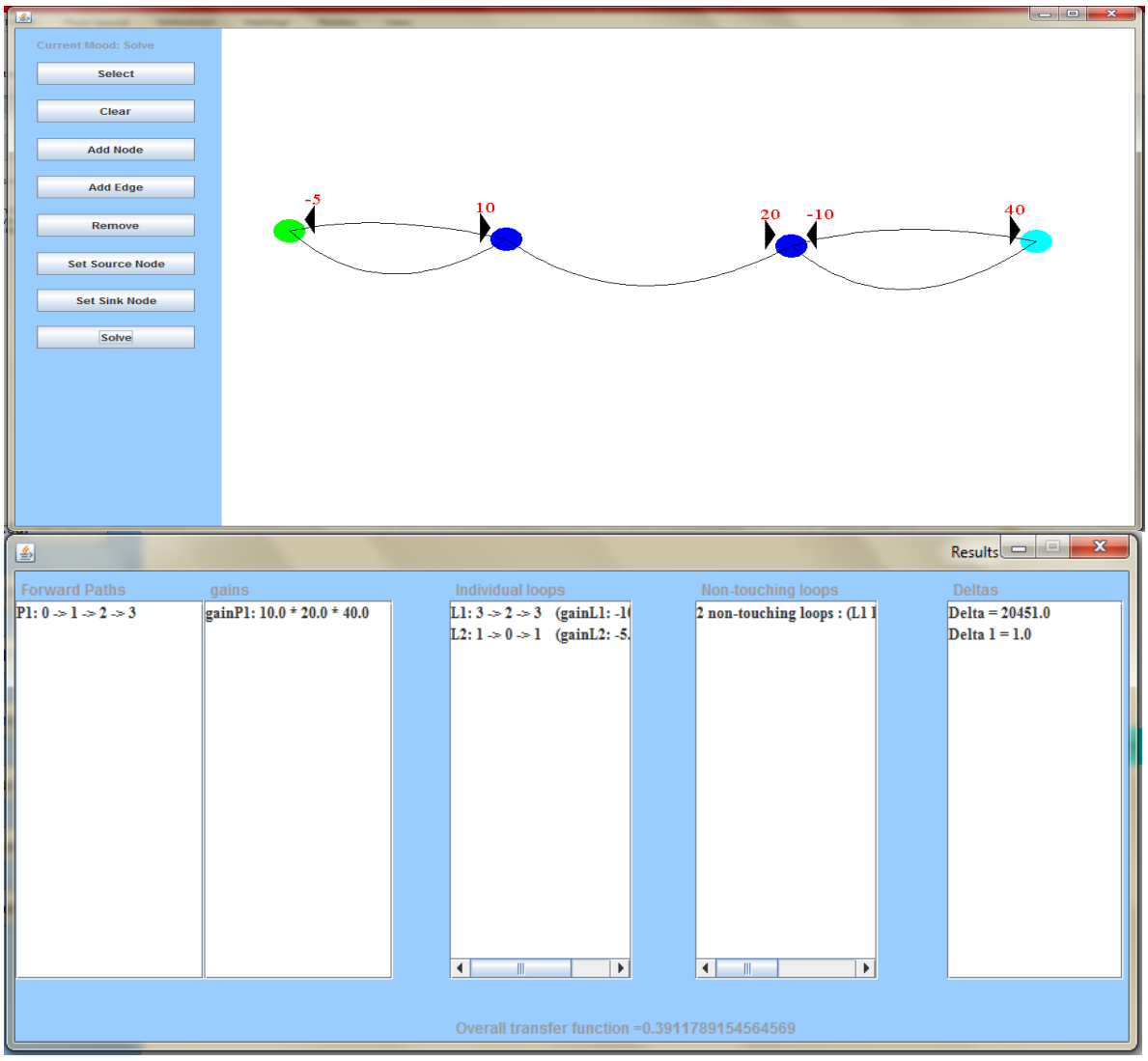
6) Sample runs.








Results
Forward Paths
gains
Individual loops
Non-touching loops
Deltas
P1: 0 -> 1 -> 2 -> 3 -> 4 -> 5
P2: 0 -> 1 -> 4 -> 5
P3: 0 -> 1 -> 3 -> 4 -> 5
gainP1: 1.0 * 1.0 * 1.0 * 1.0 *
gainP2: 1.0 * 1.0 * 1.0
gainP3: 1.0 * 1.0 * 1.0 * 1.0
L1: 3 -> 3 (gainL1: -1.
L2: 3 -> 2 -> 3 (gainL2: -1.
L3: 2 -> 1 -> 2 (gainL3: -1.
L4: 3 -> 2 -> 1 -> 3
2 non-touching loops : (L1 I
Delta = 4.0
Delta 1 = 1.0
Delta 2 = 3.0
Delta 3 = 1.0
Overall transfer function =1.25


Current Mood: Solve
Select
Clear
Add Node
Add Edge
Remove
Set Source Node
Set Sink Node
Solve
-5
10
20
-10
40
Results
Forward Paths
gains
Individual loops
Non-touching loops
Deltas
P1: 0 -> 1 -> 2 -> 3
gainP1: 10.0 * 20.0 * 40.0
L1: 3 -> 2 -> 3 (gainL1: -1(
L2: 1 -> 0 -> 1 (gainL2: -5.
2 non-touching loops : (L1 I
Delta = 20451.0
Delta 1 = 1.0
Overall transfer function =0.3911789154564569

7) Simple user guide.

- First, user starts add nodes by click on add node button then choose any point in draw area and click it.
- Second, click on add edge button and choose from node first and click it then node to be the second clicked node
- It will appear window to enter gain of this edge then enter ok.
- When finish added nodes and edges, user select two nodes to be source and sink nodes.
- Finally, user click on button solve to get solution.