SIGNAL-FLOW GRAPH SOLVER

https://github.com/ZyadSamy/signal-flow-graph

TEAM MEMBERS

Name	ID
Zyad Samy Ramadan	19015720
Ramy Ahmed El Sayed	19015649
Basel Ahmed Awad	19015513
Abdelmoneim Hany Abdelmoneim Elsayed	19017359
Youssef Ahmed Saeed Zaki	19016903

PROBLEM STATEMENT

Given:
Signal flow graph representation of the system. Assume that the 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 combinations of *n* non-touching loops.
4. The values of Δ, Δ_1,.....Δ_m where *m* is the number of forward paths.
5. Overall system transfer function.

MAIN FEATURES

- Dynamically add nodes and branches, the number of nodes or branches doesn't have to be specified at the start.
- GUI to create and display the graph, and the ability to zoom in and out to fit the screen.
- Users could add branches by simply clicking on the nodes connecting it on the graph.
- Provides a listing of all forward paths, individual loops and all combinations of *n* non-touching loops.

- Calculates The values of Δ, Δ_1,…..Δ_m , and the overall system transfer function.

DATA STRUCTURES

Five compound data structures were created to ease the process of manipulating the data sent from the front-end.
The five data structures are as follows:

1. **Node**
   The main data structure used, that is sent from the front-end to the back-end.
   It is sent in the form of an `ArrayList<Node>`.

```
public class Node {
      String name;
      ArrayList<Edge> edgeArrayList;
}
```

   Name: Name of the node.
   edgeArrayList: Edges of the node.

2. **Edge**
   The edges of the nodes that are used to simulate an adjacency-list representation of our graph.

```
public class Edge {
    final int gain;
    private Node toNode;
}
```

   Gain: Gain on the edge.
   toNode: Node being pointed at by the edge.

3. **ForwardPaths**
   The path that is evaluated from the `ForwardFinder` class and represents the forward path with its gain.

```
public class ForwardPaths {
    ArrayList<String> path;
    double gain;
    public double delta;
}
```

   Path: List of node names that belong to the path.
   Gain: Product of gains on the path.
   Delta: Calculated path Δ.

4. **Loop**

   The loop that is evaluated from the `LoopFinder` class and represents a loop with its backward gain.

```
public class Loop {
    ArrayList<String> loopNodes;
    double gain;
}
```

   loopNodes: List of node names that belong to the loop.
   Gain: Backward gain of the loop.

5. **NTLoopsCombination**

   The combination of non-touching loops evaluated from the LoopFinder class and contains the nodes of non-touching loops after merging and the product of their gains.

```
public class NTLoopsCombination {
    double gain;
    Set<Loop> NTLoops;
    Set<String> nodesAfterJoining;
}
```

   Gain: Products of gains of non-touching loops.
   NTLoops: Set of loops that are non-touching.
   nodesAfterJoining: Set of loops after merging.

MAIN MODULES

Two primary modules were used to evaluate all the values of Mason’s law:

1. **ForwardFinder**
   A class which evaluates and returns all the forwards nodes from a given start node to an end node.
   It has a major method that returns an ArrayList<ForwardPaths> containing all the forward paths.

```
 public ArrayList<ForwardPaths> getAllPaths(ArrayList<Node> graph,
String start, String end)
```

2. **LoopFinder**
   A class which evaluates and returns all loops, all non-touching loops, all path deltas, overall delta, and the graph transfer function.
   The major methods used are as follows:

```
public ArrayList<Loop> findAllLoops()
public ArrayList<LinkedList<NTLoopsCombination>>
findNTLs(ArrayList<Loop> loops)
public double getOverallDelta(ArrayList<Loop> loops,
ArrayList<LinkedList<NTLoopsCombination>> nonTouching)
public void getPathDelta(ArrayList<Loop> loops,
ArrayList<LinkedList<NTLoopsCombination>> nonTouching,
ArrayList<ForwardPaths> paths)
public double getTF(double delta, ArrayList<ForwardPaths> paths)
```

ALGORITHMS USED

1. DFS Algorithm

- Purpose:
    - Traversing nodes and iterating over each node in graph
    - Helps in finding forward paths by taking input node as start and output node as end
    - Helps in finding loops
- Data Structures used:
    - Used on an array of nodes which represent our graph
- Technique (Recursive):
    - Enters current node
    - Checks if node is visited
        - Marks it as visited
        - Iterate over node edges
        - Gets toNode from each edge
    - If not
        - Returns to previous node and continues iteration over edges
- Analysis:
    - Time Complexity: O(N^2)
        - N: number of nodes

2. Jonathan’s Algorithm

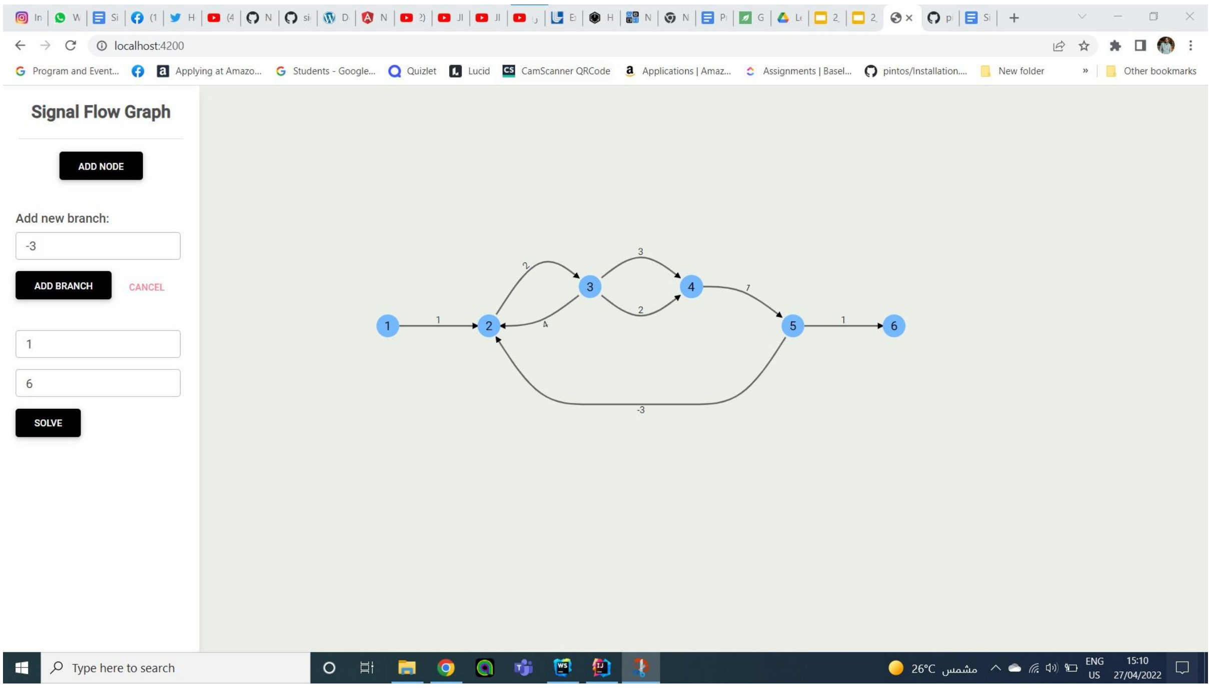
- Purpose:
  - Getting all loops (circuits) in a graph
- Data Structures used:
  - Array of nodes: our graph
  - Set of nodes: blockedSet
    - where we keep the blocked nodes while iterating
  - Map<Node,Set<Nodes>>: blockedMap
    - where we manage blocking nodes dependencies
    - Which nodes need to be unblocked when current node is unblocked.
  - Stack: our recursive DFS stack
  - Array of loops which will contain the final result
- Technique:
  - Use Tarjan’s Algorithm on our graph
    - Discussed further
  - Get the component which has the least node in the graph
  - Iterate over all nodes in component
  - Look for a cycle that begins and ends with current node:
    - Use DFS to traverse and use stack accordingly
    - If node is blocked DFS tracks back
    - If no cycle at some nodes, nodes are added to blocked set
    - If node contains cycle from start node, don’t clock it
    - If node ‘a’ can have cycle only if node ‘b’ is not blocked
      - Place them on blockedMap
    - Unblock nodes when a new direction of paths opens
      - Unblock depending nodes on it in blockedMap
    - If cycle found add it to our loops array
  - When we got all cycles starting from start node
    - we delete start node from graph
    - We delete any toEdges having start node as end node
  - Continue
- Complexity:
  - Analysis:
    - Time Complexity: O(N^3)
      - N: number of nodes

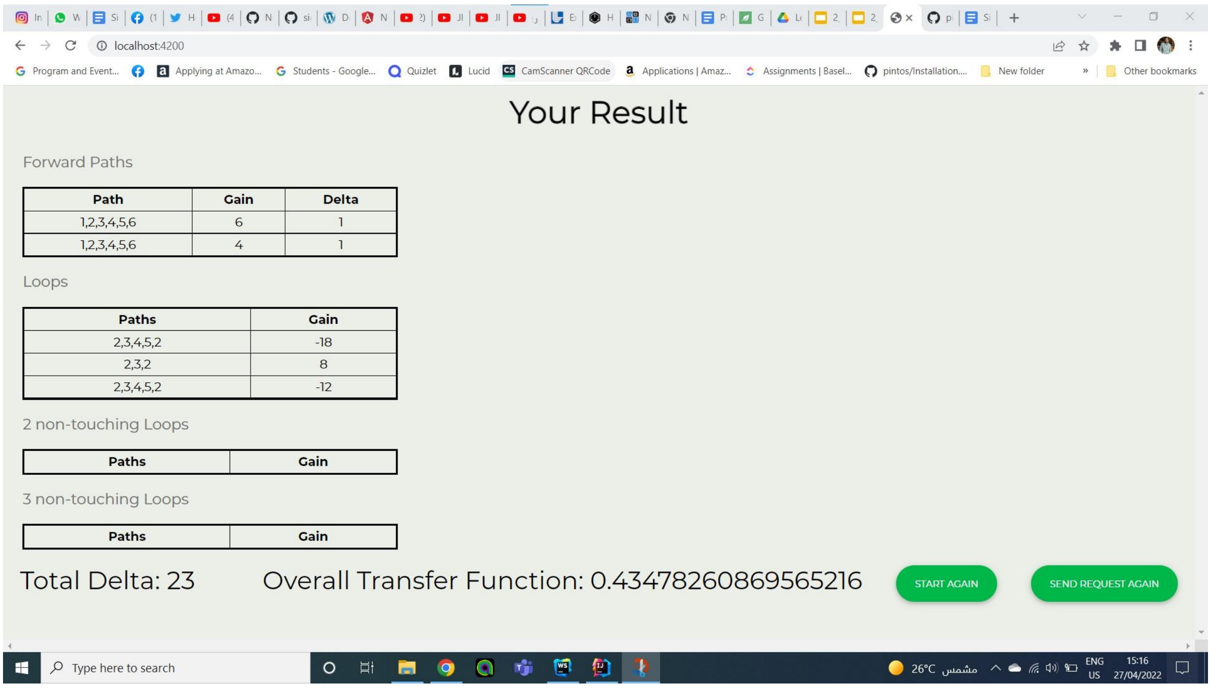
3. Tarjan’s Algorithm

- Purpose:
    - Gets strongly connected components of a given graph
- Important Definitions:
    - Lo link of a node: least node that could be reached from current node
- Data Structures used:
    - graph
    - Recursive stack of nodes as we use DFS
    - 2 node-integer maps
        - indexMap: maps each node to its DFS index
        - lolinkMap: maps each node to its lo link
- Technique:
    - We DFS traverse graph
    - Update indexMap value of current node
    - Update lolinkMap value of current node
        - Minimum of all upcoming nodes indices visited from current and current index
        - Done by head recursion
    - After traversal is done
        - We make m groups on components
            - where each component has all its nodes having same lolink
- Complexity:
    - Analysis:
        - Time Complexity: O(N+E)
            - N: number of nodes
            - E: number of edges

SAMPLE RUNS

Sample 1





Your Result

Forward Paths

Path	Gain	Delta
1,2,3,4,5,6	6	1
1,2,3,4,5,6	4	1

Loops

Paths	Gain
2,3,4,5,2	-18
2,3,2	8
2,3,4,5,2	-12

2 non-touching Loops

Paths	Gain

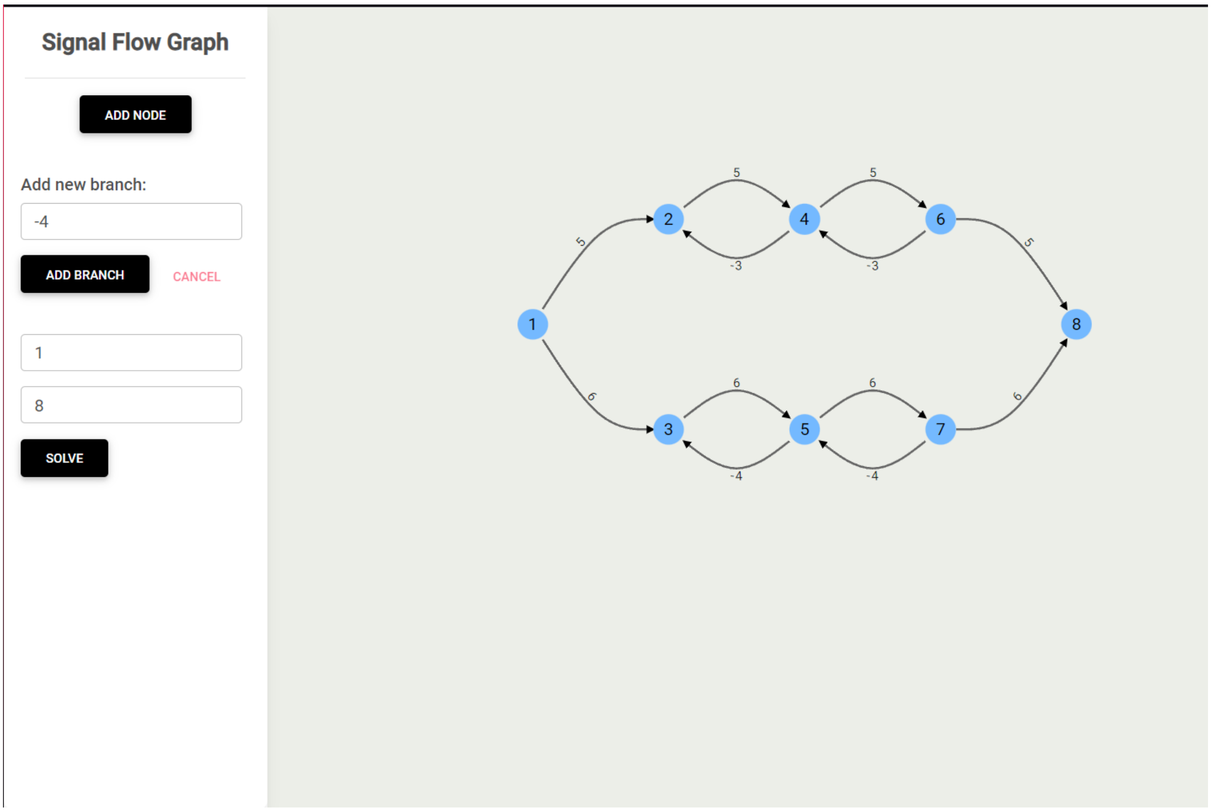
3 non-touching Loops

Paths	Gain

Total Delta: 23

Overall Transfer Function: 0.43478260869565216

Sample 2

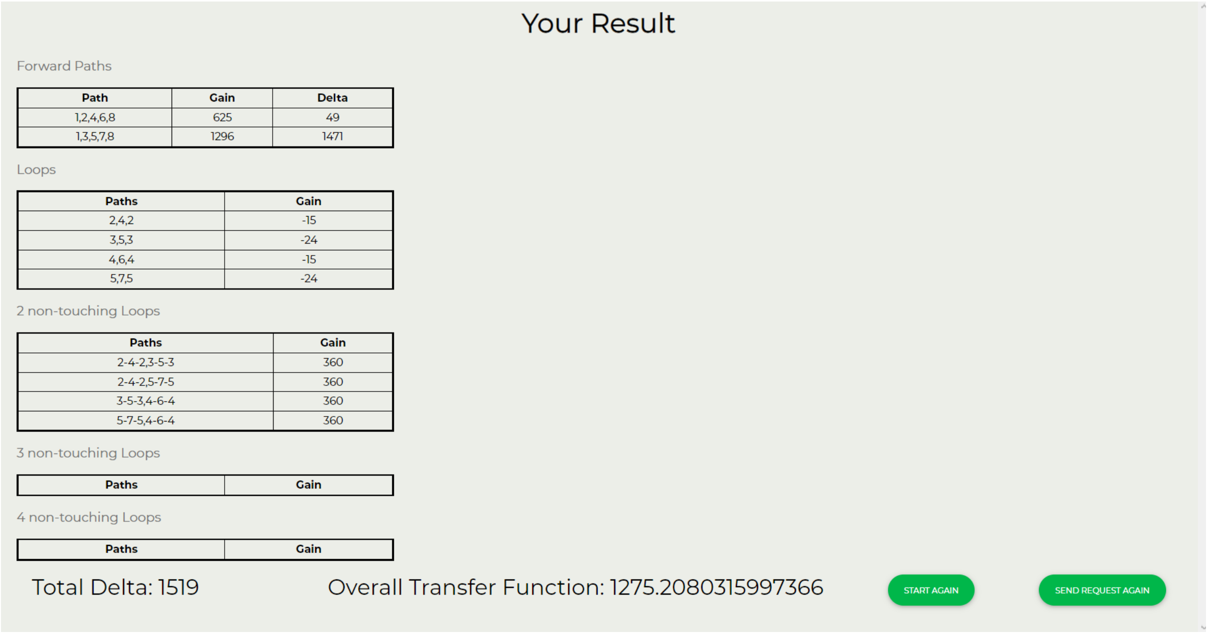


Add new branch:

-4

1

8



Your Result

Forward Paths

Path	Gain	Delta
1,2,4,6,8	625	49
1,3,5,7,8	1296	1471

Loops

Paths	Gain
2,4,2	-15
3,5,3	-24
4,6,4	-15
5,7,5	-24

2 non-touching Loops

Paths	Gain
2-4-2,3-5-3	360
2-4-2,5-7-5	360
3-5-3,4-6-4	360
5-7-5,4-6-4	360

3 non-touching Loops

Paths	Gain

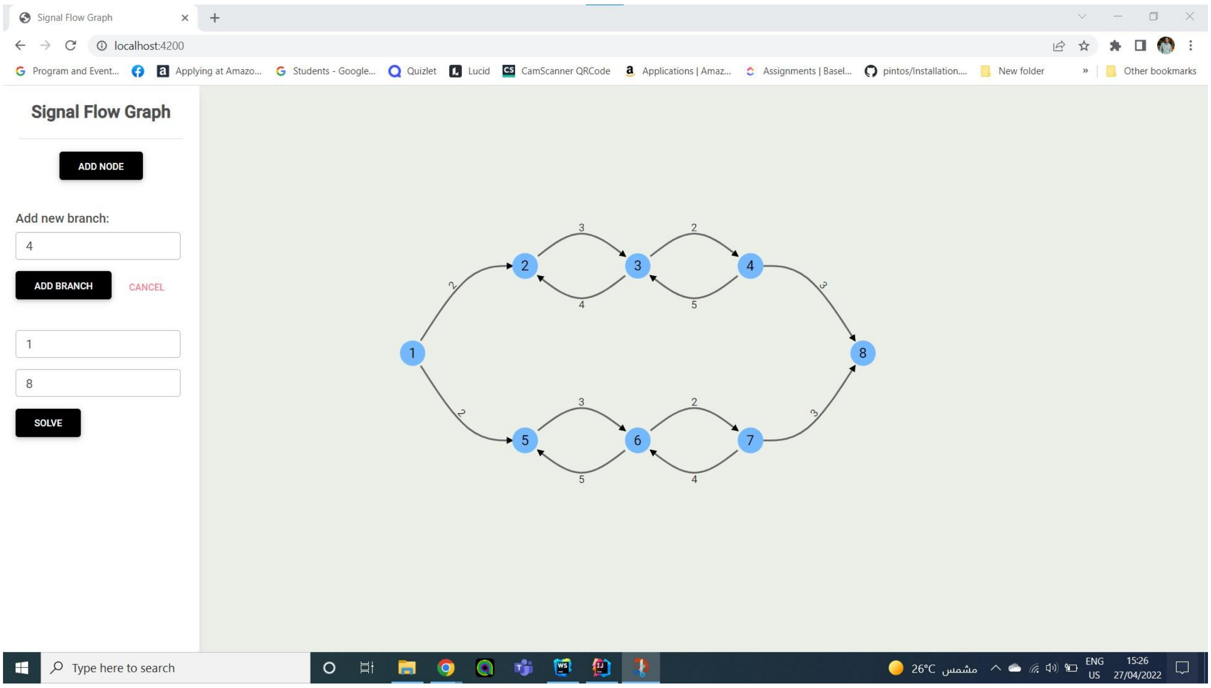
4 non-touching Loops

Paths	Gain

Total Delta: 1519

Overall Transfer Function: 1275.2080315997366

Sample 3



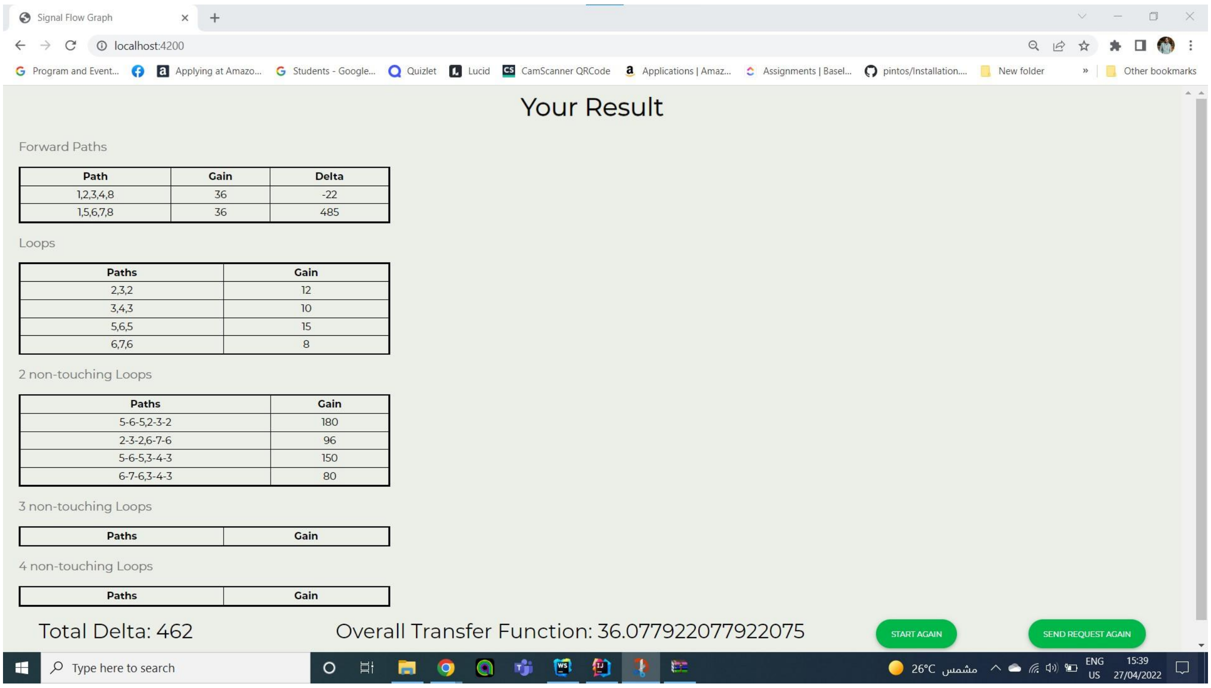
Signal Flow Graph

Add new branch:

4

1

8



Your Result

Forward Paths

Path	Gain	Delta
1,2,3,4,8	36	-22
1,5,6,7,8	36	485

Loops

Paths	Gain
2,3,2	12
3,4,3	10
5,6,5	15
6,7,6	8

2 non-touching Loops

Paths	Gain
5-6-5,2-3-2	180
2-3-2,6-7-6	96
5-6-5,3-4-3	150
6-7-6,3-4-3	80

3 non-touching Loops

Paths	Gain

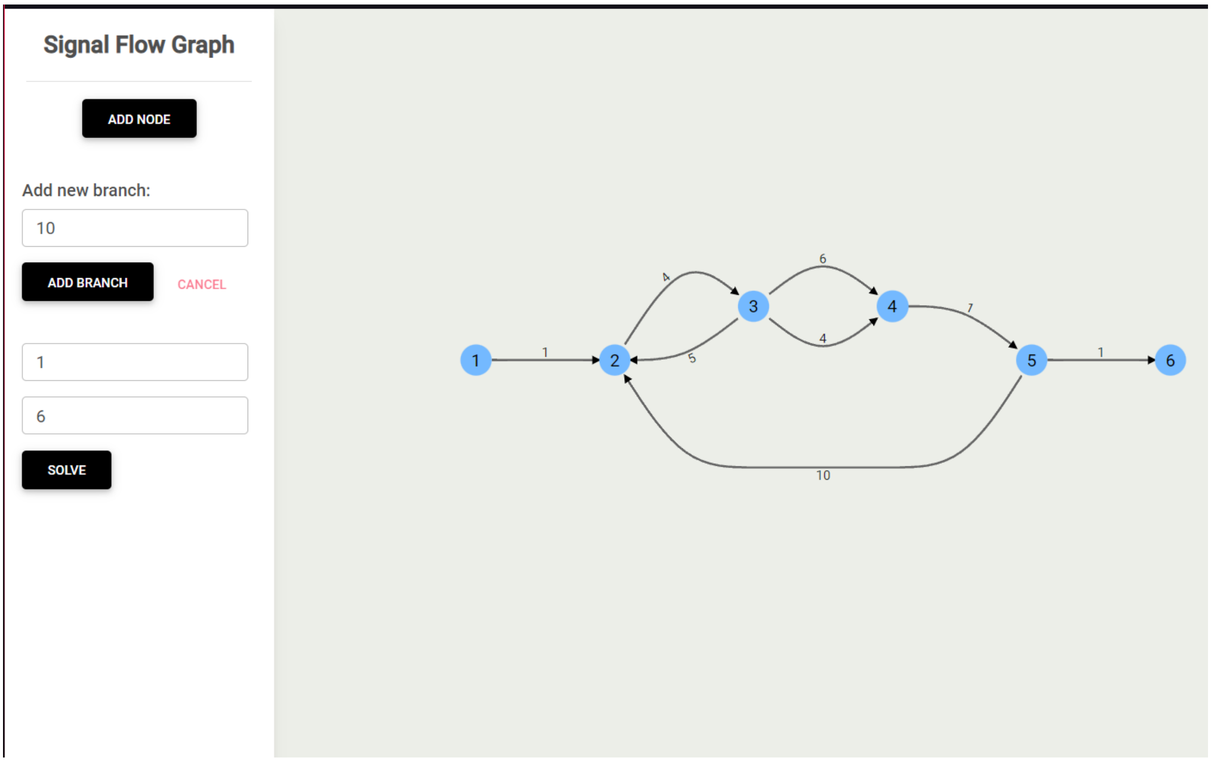
4 non-touching Loops

Paths	Gain

Total Delta: 462

Overall Transfer Function: 36.077922077922075

Sample 4



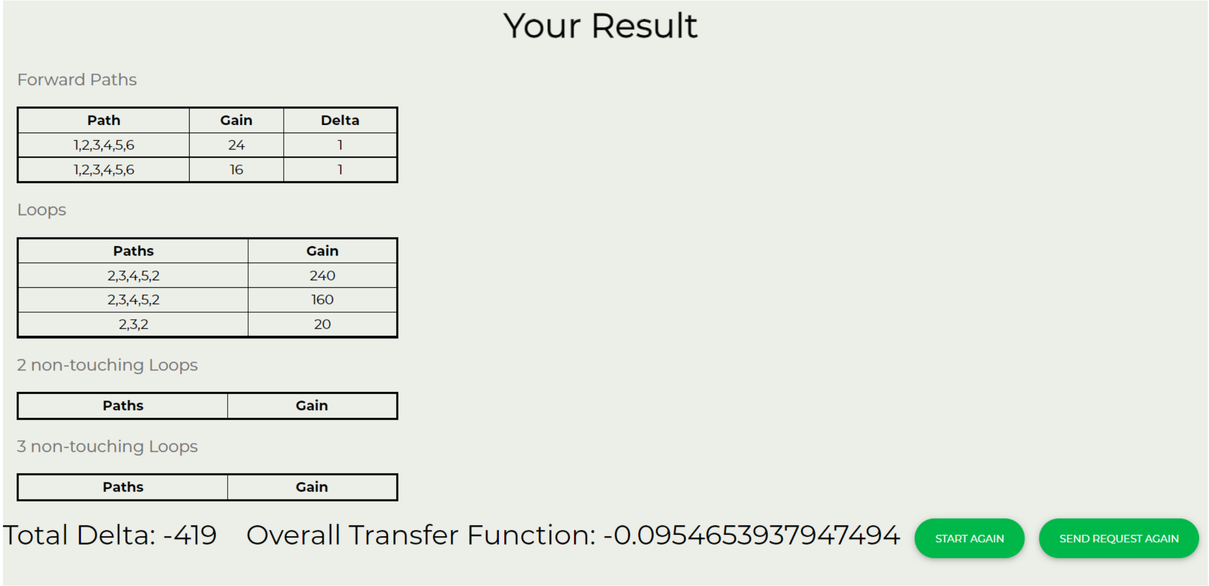
Signal Flow Graph

Add new branch:

10

1

6



Your Result

Forward Paths

Path	Gain	Delta
1,2,3,4,5,6	24	1
1,2,3,4,5,6	16	1

Loops

Paths	Gain
2,3,4,5,2	240
2,3,4,5,2	160
2,3,2	20

2 non-touching Loops

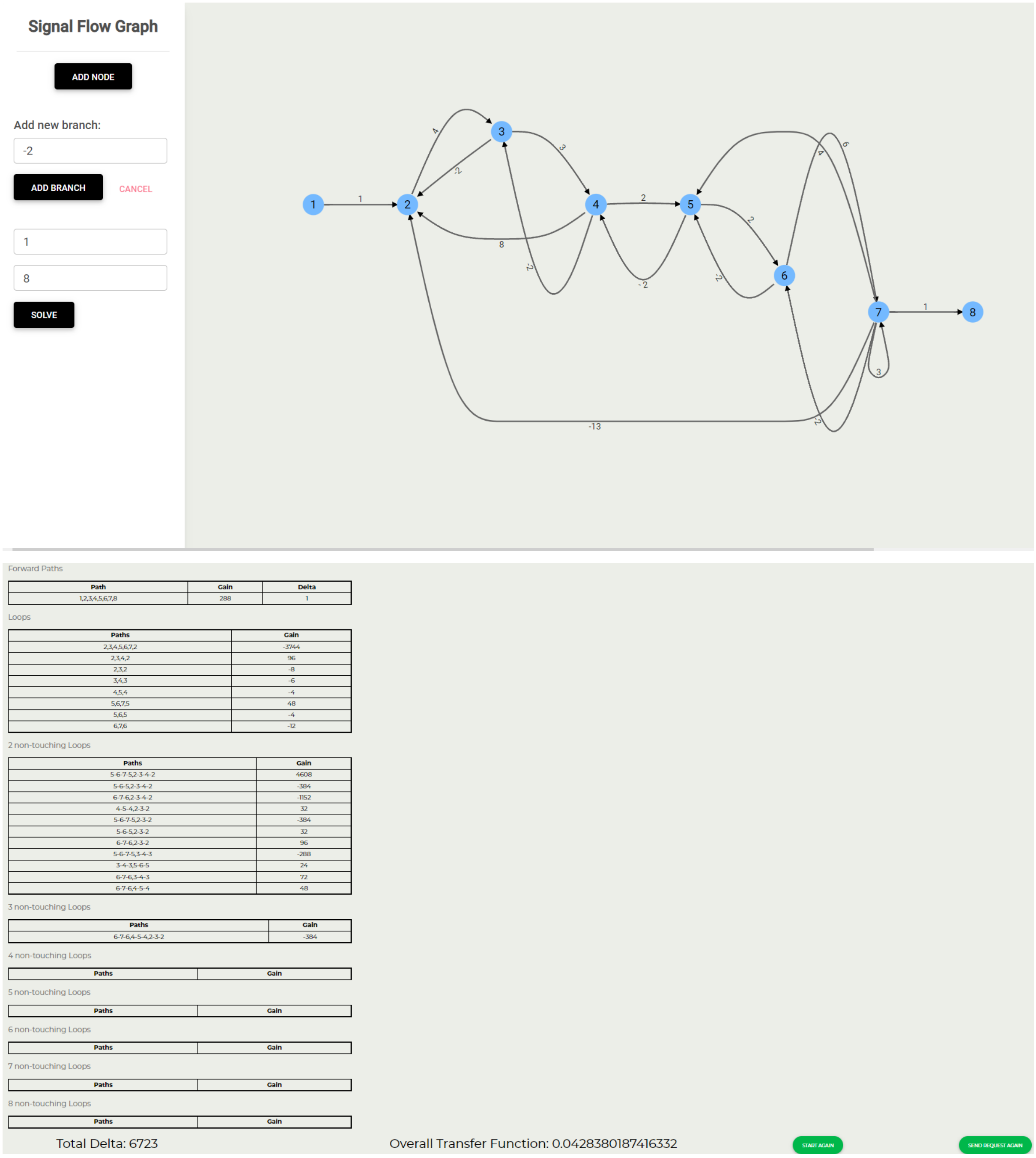
Paths	Gain

3 non-touching Loops

Paths	Gain

Total Delta: -419   Overall Transfer Function: -0.0954653937947494

Sample 5



Add new branch:

-2

1

8

Forward Paths

Path	Gain	Delta
1,2,3,4,5,6,7,8	288	1

Loops

Paths	Gain
2,3,4,5,6,7,2	-3744
2,3,4,2	96
2,3,2	-8
3,4,3	-6
4,5,4	-4
5,6,7,5	48
5,6,5	-4
6,7,6	-12

2 non-touching Loops

Paths	Gain
5-6-7-5,2-3-4-2	4608
5-6-5,2-3-4-2	-384
6-7-6,2-3-4-2	-1152
4-5-4,2-3-2	32
5-6-7-5,2-3-2	-384
5-6-5,2-3-2	32
6-7-6,2-3-2	96
5-6-7-5,3-4-3	-288
3-4-3,5-6-5	24
6-7-6,3-4-3	72
6-7-6,4-5-4	48

3 non-touching Loops

Paths	Gain
6-7-6,4-5-4,2-3-2	-384

4 non-touching Loops

Paths	Gain

5 non-touching Loops

Paths	Gain

6 non-touching Loops

Paths	Gain

7 non-touching Loops

Paths	Gain

8 non-touching Loops

Paths	Gain

Total Delta: 6723

Overall Transfer Function: 0.0428380187416332

SIMPLE USER GUIDE

HOW TO RUN

- Front-end

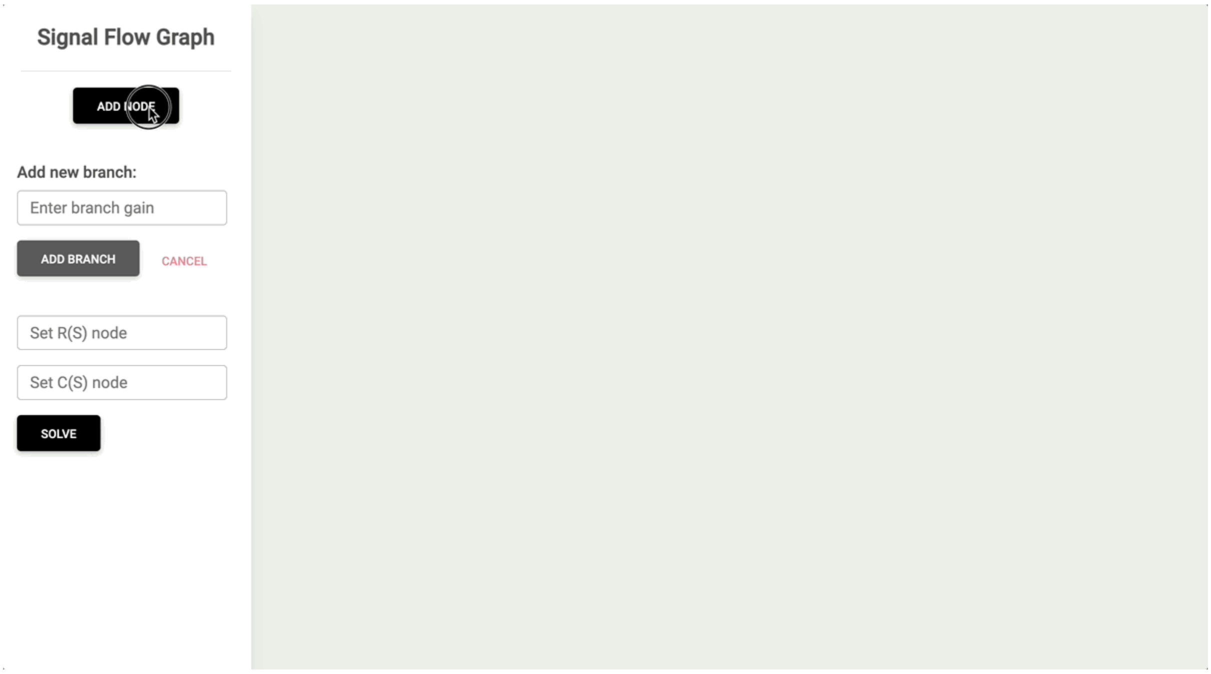
```
git clone https://github.com/ZyadSamy/signal-flow-graph.git
cd signal-flow-graph
npm install --legacy-peer-deps
ng serve -o
```

- Back-end

  Import project as maven project and solve dependencies and run it using a Java IDE

```
cd backend
mvn install
```

HOW TO USE



Steps

1. Add nodes needed by clicking on “Add node”, the added nodes will be numbered in an ascending order starting from number 1.
2. Add branches between nodes by doing the following:
   a. Add branch gain
   b. Click on “Add branch” button
   c. Click on the graph on the node to be drawn from
   d. Click on the node to be drawn to
3. Keep adding nodes and branches according to your problem.
4. When done with the graph, type in “input node” the number of the node desired to calculate the transfer function from, and do the same for the output node.
5. Click on “Solve”.

Google Docs Link

https://docs.google.com/document/d/1Fgwmge5O-30Ymzyq2Cq24B2hoqBJAR7KoWEcAwe6Jjs/edit#heading=h.m7ozziruvq7t