3 Outcomes

The primary goal of this assignment is to implement a Control Flow Graph simulator/calculator and implement the following algorithms:

- Shortest Path
- Simple Path
- Prime Path
- Reachability
- Graph Splitting
- etc

4 Introduction

Complete the task below. Certain classes have been provided for you alongside this specification in the Student files folder. A very basic main has been provided. **Please note this main is not extensive and you will need to expand on it**. Remember to test boundary cases. Submission instructions are given at the end of this document.

5 Background

5.1 Graphs

A Graph is a collection of vertices and connections between them. The vertices are nodes and the connections are known as edges. Each edge connects a pair of vertices. If the edges are directional, the graph is known as a directed graph.

Each edge can be assigned a number that represents values such as cost, distance, length or weight. Such a graph is then called a weighted directed graph.

5.2 Control Flow Graphs

Developed by Frances E. Allen in the 1970s [All70], Control Flow Graphs (CFG) are a static analysis of software code used in software testing. Blocks of code in the source code represent nodes in the graph. Sequential flows between blocks of code represent edges between nodes. CFG have designated entry points and exit points. A CFG is known as a Single Entry Single Exit (SESE) graph if it only has a single entry node and a single exit node. A problem that arose during research on static analysis is how to deal with loops in the CFG. In an attempt to solve these issues, Simple and Prime paths were introduced. These will be discussed in the next section. As an example of how a CFG graph can be constructed from code, the following code snippet can be represented as the SESE CFG below the code snippet.

```
int func(){
    int a = 5;
    int i=0;
    while(i != a){
        i++;
    }
    return i;
}
```

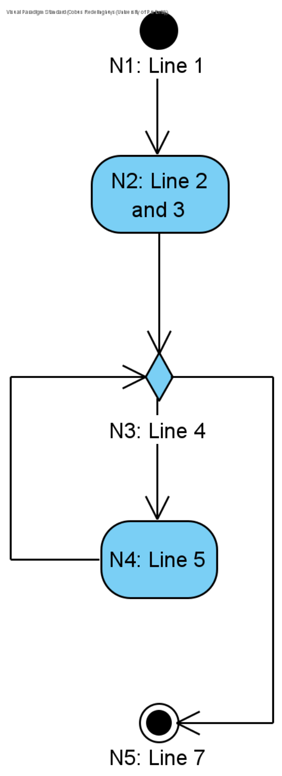

Figure 1: CFG of the above code snippet

Paths within the CFG can be used to create software tests. Let's assume using the above example, we have a Path consisting of the following nodes: {N1, N2, N3, N5}. N4 is not included as in this example, we do not enter the loop. Now it would be up to the test engineer to determine if such a path exists and what variables need to be set so that the path is executed.

Luckily this step of the static analysis is beyond the scope of this assignment.

5.3 Simple Paths

From theory, it should be clear that loops are problematic. It is not always possible to know at compile time or even before then how many times the loop would be executed. This problem is similar to that of the Halting Problem (which is algorithmically unsolvable) [Bur87].
In an attempt to approach this problem, CFGs make use of something called a

Simple Path. A Simple Path is a path of any length that satisfies the following properties [Gol10]:

- Only the first and last node in the path may be repeated.
- All edges used in the path needs to exist in the graph.

Hint: The theoretical maximum length of Simple Paths is the number of nodes in the graph

Simple Paths of length 0 consist of just a Node. Using the CFG in Figure 1 the following Simple Paths can be created:

- Length 0:
  - N1
  - N2
  - N3
  - N4
  - N5
- Length 1:
  - {N1, N2}
  - {N2, N3}
  - {N3, N4}
  - {N3, N5}
  - {N4, N3}
- Length 2:
  - {N1, N2, N3}
  - {N2, N3, N4}
  - {N2, N3, N5}
  - {N3, N4, N3}
  - {N3, N3, N4}
  - {N4, N3, N5}
- Length 3:
  - {N1, N2, N3, N4}
  - {N1, N2, N3, N5}

5.4 Prime Paths

To reduce the possible paths created by Simple Paths, Prime Paths were introduced. A Prime Path is a Simple Path that is not a sub-path of another Simple Path. In other words, a Prime Path is the longest Simple Path that does not form part of another Simple Path [Gol10]. Using Figure 1 and the Simple Paths listed in the previous section, the following Prime Paths can be determined:

- {N3, N4, N3}
- {N4, N3, N4}
- {N4, N3, N5}
- {N1, N2, N3, N4}
- {N1, N2, N3, N5}

Interestingly, Prime Paths also locate loops in the graph as can be seen by the first two paths listed above.

Prime Paths **are not** required to start at the entry node and finish at the exit node. Only the last Prime Path listed actually has this property.

6 Task 1: Control Flow Graph

Your task for this assignment would be to implement the following class diagram as described in later sections.

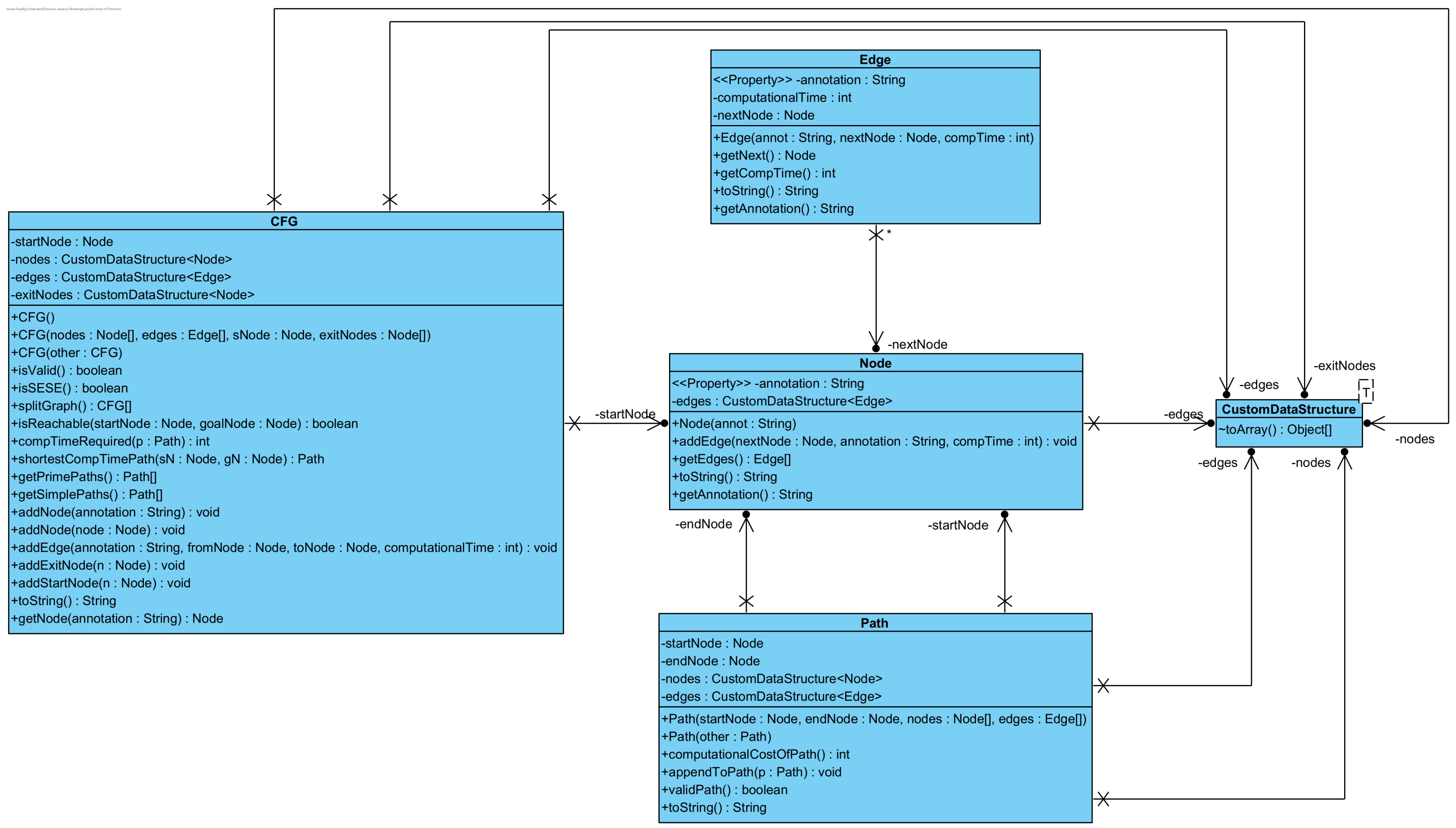
Visual Paradigm Standard(Francois Jacobus Redelinghuys(University of Pretoria))
Edge
<<Property>> -annotation : String
-computationalTime : int
-nextNode : Node
+Edge(annot : String, nextNode : Node, compTime : int)
+getNext() : Node
+getCompTime() : int
+toString() : String
+getAnnotation() : String
*
-nextNode
CFG
-startNode : Node
-nodes : CustomDataStructure<Node>
-edges : CustomDataStructure<Edge>
-exitNodes : CustomDataStructure<Node>
+CFG()
+CFG(nodes : Node[], edges : Edge[], sNode : Node, exitNodes : Node[])
+CFG(other : CFG)
+isValid() : boolean
+isSESE() : boolean
+splitGraph() : CFG[]
+isReachable(startNode : Node, goalNode : Node) : boolean
+compTimeRequired(p : Path) : int
+shortestCompTimePath(sN : Node, gN : Node) : Path
+getPrimePaths() : Path[]
+getSimplePaths() : Path[]
+addNode(annotation : String) : void
+addNode(node : Node) : void
+addEdge(annotation : String, fromNode : Node, toNode : Node, computationalTime : int) : void
+addExitNode(n : Node) : void
+addStartNode(n : Node) : void
+toString() : String
+getNode(annotation : String) : Node
-startNode
Node
<<Property>> -annotation : String
-edges : CustomDataStructure<Edge>
+Node(annot : String)
+addEdge(nextNode : Node, annotation : String, compTime : int) : void
+getEdges() : Edge[]
+toString() : String
+getAnnotation() : String
-edges
-exitNodes
-edges
T
CustomDataStructure
~toArray() : Object[]
-nodes
-edges
-nodes
-endNode
-startNode
Path
-startNode : Node
-endNode : Node
-nodes : CustomDataStructure<Node>
-edges : CustomDataStructure<Edge>
+Path(startNode : Node, endNode : Node, nodes : Node[], edges : Edge[])
+Path(other : Path)
+computationalCostOfPath() : int
+appendToPath(p : Path) : void
+validPath() : boolean
+toString() : String

Figure 2: Class diagram

6.1 Node

This will be the class that represents the nodes (code blocks) in the CFG.

- Members:
  - annotation: String
    - This is the "label" associated with this Node.
    - Within the same CFG, it can be assumed that each annotation is unique.
  - edges: *Own data structure*<Edge>
    - This is the list of edges for this node.
    - You should create your own data structure that will be a list of edges.
    - Please see Section 9 for more information about the restrictions and implementation of your own data structure
- Functions:
  - Node(annot: String)
    - This is the constructor for the Node class.
    - Initialize the annotation member with the passed-in parameter and the edges with a new instance of the custom data structure.
  - addEdge(nextNode:Node, annotation:String, compTime:int): void
    - This function adds an edge to the current node.
    - Use the passed-in parameters to create a new edge.
  - getEdges(): Edge[]
    - This function should return the edges member variable as an array.
    - You can utilize the toArray function of your own data structure and then cast accordingly.
    - If no edges exist, an empty array should be returned. Note empty and **not** null.

  - toString(): String
    * This is a provided function.
    * Do not change this function.
    * Please note your custom data structure should have a fully working toArray function for this function to work correctly.
  - getAnnotation(): String
    * This function should return the annotation member variable.

6.2 Edge

This is the class that will represent the edges in the CFG.

- Members:
  - annotation: String
    * This is the "label" associated with this Edge.
    * Within the same CFG, it can be assumed that each annotation is unique.
  - computationalTime: int
    * This is the "weight" of the edge.
    * The value represents the computational time to move to the next node via this edge.
    * It can be assumed that computationalTime will always be positive.
  - nextNode: Node
    * This is the node this edge is pointing to.
    * It **can** be assumed that this will never be null.
- Functions:
  - Edge(annot: String, nextNode: Node, compTime: int)
    * This is the constructor for the Edge class.

    - Populate the respected member variables with the passed-in parameters.
    - Make use of shallow copies to populate the nextNode member variable.
  - getNext(): Node
    - This function should return the nextNode member variable.
  - getCompTime(): int
    - This function should return the compTime member variable.
  - toString(): String
    - This is a provided function.
    - Do not change this function.
  - getAnnotation(): String
    - This function should return the annotation member variable.

6.3 Path

- Members
  - startNode: Node
    - This is the starting Node for this path.
    - It can be assumed that this variable will never be null.
  - endNode: Node
    - This is the ending Node for this path.
    - It can be assumed that this variable will never be null.
  - nodes: *Own data structure*<Node>
    - This is an **ordered** list of nodes that are in the path.
    - The nodes in the list will be ordered in the same order as the path will traverse.
    - You should create your own data structure that will accomplish this task.

    - Please see Section 9 for more information about the restrictions and implementation of your own data structure
  - edges: *Own data structure*<Edge>
    - This is an **ordered** list of edges that are in the path.
    - The edges in the list will be ordered in the same order as the path will traverse.
    - You should create your own data structure that will accomplish this task.
    - Please see Section 9 for more information about the restrictions and implementation of your own data structure

- Functions:
  - Path(startNode:Node, endNode:Node, nodes:Node[], edges:Edge[])
    - This is the constructor for the Path class.
    - Populate the startNode and endNode member variables with the appropriate passed-in parameters. Make use of shallow copies.
    - Use the passed-in arrays to populate the appropriate member variables. The elements in the data structure should be shallow copies of the elements in the array.
    - If the passed-in arrays are null, the respective member variable should be initialized with a new empty data structure of your own creation.
  - Path(other: Path)
    - This is the copy constructor for the Path class.
    - Populate the appropriate members with shallow copies.
    - For your own data structure members, create a new instance data structure and populate it with shallow copies.
  - computationalCostOfPath(): int
    - This function should calculate how "expensive" this path is by summing all the computation times of all the edges used to get from the start to the end node.

        * If there are no edges then the function should return 0.
    - validPath(): boolean
        * This function should validate the path.
        * A path is valid if and only if:
            · For every ith node in the nodes array, the ith edge can be used to go to the $\boldsymbol{i}$**th** $+$ **1** node.
            · The final node that is reached will be the endNode member variable.
    - toString(): String
        * This is a provided function.
        * Do not change this function.
        * Please note your custom data structure should have a fully working toArray function for this function to work correctly.

6.4 CFG

- Members:
    - startNode: Node
        * This is the starting node for the CFG.
        * The starting node can be null.
    - nodes: *Own data structure*<Node>
        * This is a list of all the nodes in the CFG including the starting node and all the exit nodes.
        * You should create your own data structure that will accomplish this task.
        * Please see Section 9 for more information about the restrictions and implementation of your own data structure
    - edges: *Own data structure*<Edge>
        * This is a list of all the edges in the CFG.

    - You should create your own data structure that will accomplish this task.
    - Please see Section 9 for more information about the restrictions and implementation of your own data structure
  - exitNodes: *Own data structure*<Node>
    - This is a list of all the nodes that act as exit points for the CFG.
    - You should create your own data structure that will accomplish this task.
    - Please see Section 9 for more information about the restrictions and implementation of your own data structure

- Functions:
  - CFG()
    - This is the default constructor for the CFG class.
    - Initialize each of the custom data structure member variables to a new instance of the data structure.
  - CFG(nodes: Node[], edges: Edge[], sNode: Node, exitNodes: Node[])
    - This is a parameterized constructor for the CFG class.
    - Use the passed-in arrays to populate the respected member variables. Make use of shallow copies to populate the array lists.
    - If any of the passed-in arrays are empty, initialize the respected member variable to a new instance of the data structure.
  - CFG(other: CFG)
    - This is a copy constructor for the CFG class.
    - This constructor should create a **deep copy** of the passed-in parameter.
    - All the elements in the data structure should also be **deep copies**. *Hint: remember, the annotations for the edges and nodes are unique to each node and edge.*

    * If the passed-in parameter is null, initialize all the appropriate members to a new instance of the custom data structure.

- isValid(): boolean
    * This function should determine if the CFG is valid.
    * A CFG is valid if and only if the following properties hold:
        · The start node is not null.
        · There is at least one exit node.
        · For every non-exit node in the CFG, at least one of the exit nodes is reachable from this node. In other words, there exists a path from said node to at least one exit node.
    * If the CFG is valid, the function should return true, otherwise false.
- isSESE(): boolean
    * This function should determine if the CFG is a SESE CFG.
    * A SESE CFG can be described by the following properties:
        · Is a valid CFG.
        · Only has one exit node.
- splitGraph(): CFG[]
    * This function will attempt to turn any valid CFG into a set of SESE graphs.
    * If the CFG is already a SESE graph, return an array populated with just the current graph. (In other words, the array will have a size of 1)
    * Use the following pseudo-code as a guide on how to split the CFG:

```
For each exit node EN, make a new CFG with the
    current CFG s start node as the start node and EN
    as the exit node.
For every node N in the current CFG check if N can
    reach each of the exit nodes. If it can then add
    that node to the appropriate new CFG.
Remember to also add all the appropriate edges.
```

Hint: there are alternatives to this method

    * Each resulting SESE CFG should only contain nodes that can reach its exit node and only edges that connect nodes in the SESE CFG together.
    * Remove all unnecessary edges from the nodes in the SESE CFG.
    * The original CFG should remain unchanged.
    * Return all the new SESE CFGs in an array.
    * Note the order of the newly created SESE CFGs does not matter as the FitchFork main will sort them.
- isReachable(startNode: Node, goalNode: Node): boolean
    * This function should determine if the goalNode is reachable from the startNode.
    * If it is reachable, the function should return true and false if not.
    * If either the startNode or the goalNode is null, the function should return false.
- compTimeRequired(p: Path): int
    * This function should return the computational of the passed-in path.
    * If the path is null, the function should return -1.
- shortestCompTimePath(sN: Node, gN: Node): Path
    * This function should determine the shortest path, with respect to computational time, between the sN and the gN.
    * It can be assumed that there exists at least one path between sN and gN
    * This function should return a Path object, containing the shortest path from sN to gN.
    * The resulting path object should be populated with both the edges and the nodes needed to traverse the shortest path.
- getPrimePaths(): Path[]
    * This function should return all the Prime Paths in the CFG as an array.

    - It can be assumed that the CFG will be a SESE CFG.
- getSimplePaths(): Path[]
    - This function should return all the simple paths in the CFG as an array.
    - It can be assumed that the CFG will be a SESE CFG.
- addNode(annotation: String): void
    - This function should create a new Node with the passed-in parameter and add it to the appropriate list.
    - If a node already exists with the passed-in annotation no new node should be added to the current CFG
- addNode(node: Node): void
    - This function should add the passed-in node as a shallow copy into the nodes list.
    - If the passed-in parameter is null the function should not alter the nodes list.
- addEdge(annotation: String, fromNode: Node, toNode: Node, computationalTime: int): void
    - This function should add a new edge to the fromNode which will connect to the toNode.
    - Use the passed-in parameters to create the new edge.
    - Remember to add the edge to the edges member.
    - If an edge with the same annotation already exists in the CFG, do not add it to the nodes member or the edges member.
    - If the computationalTime is negative, do not add it to the nodes member or the edges member.
- addExitNode(n: Node): void
    - This function will add the passed-in node to the exit nodes member, if it is not already contained in the member.
    - This function should also add the node to the nodes member, if the nodes member does not contain the passed-in node.

    * This function should make use of shallow copying to perform its functionality.
    * If the passed-in parameter is null, the function should not add anything to any of the lists.
  - addStartNode(n: Node)
    * This function will set the start node member variable to the passed-in parameter.
    * If the start node is already set, the function should not set it.
    * If the node is not in the nodes member, add it.
    * Make use of shallow copy.
  - toString(): String
    * This is a provided function.
    * Do not change this function.
  - getNode(annotation: String): Node
    * This function should return the node that is associated with the passed-in annotation.
    * If no node has the annotation then the function should return null.

7 Note on edges

Due to the nature of shallow copies, your code will need to account for when an edge is added to a node outside of the CFG object. Thus, before you use the edges member variable of the CFG class it is recommended that you ensure that the edges member variable is up to date.

8 Suggested Helper function(s)

The following functions are suggested as helper functions:

- Get all the paths of a certain length.

- Check if a path is a simple path.
- Reset function to restore member variables to default values.
- Append function to append a path onto another path.
- Remove node and edge function.

Note this list is not exhaustive. Helper functions and member variables need to be added to complete some of the functionalities.

9 Own Data Structure

This assignment needs some additional data structure to accomplish a few of the implementations. You need to create your own custom data structure. You may decide which custom data structure you would like to implement from any of the data structures learnt in COS110 and COS212. The data structure must implement the following function:

```
    public Object[] toArray()
```

This function should add the elements contained in the data structure in the order they were inserted into the data structure. This function should be fully working as the toString functions use this function. The resulting array should be exactly sized and should not contain any nulls.

The following are some of the recommended functional requirements of your custom data structure:

- Dynamically sized
- Insert
- Remove
- Search
- Contains