UNIVERSITAT DE BARCELONA

Facultat de Matemàtiques i Informàtica

GRAU DE MATEMÀTIQUES

Treball final de grau

ELECTRICAL CIRCUIT SIMULATION

Autora: Julie Isabel Arago Delage

Director: Dr. Antoni Benseny

Realitzat a: Departament de Matemàtiques i Informàtica

Barcelona, June 21, 2020

Acknowledgments

I would like to thank my tutor, Dr. Antoni Benseny, for sharing his enthusiasm for this subject and allow me to join him on a project he has being working on for a long time.

I would also like to thank my parents, they have been showing unconditional support trough all these years. Even when I was discouraged they gave me the strength to keep going and I cannot thank them enough. Merci beaucoup.

Last but not least, to David, thank you for your pacience during all this process and for helping me staying sane.

Abstract

Given an electrical circuit, formed by wires, nodes, batteries and resistors, the main goal of this work is to solve this electrical circuit. This means finding the electric current in each wire. The graph theory and its use in computer science through data structures, like trees, and hierarchical traversing algorithms, like the Breadth-first search algorithm, will be used to solve the problem. The first part of the project will intend to solve an electrical circuit using C++, the main algorithms used have been described in this paper. The second part of the project will simulate an electrical circuit using graphics and control libraries allowing to interact with the electrical circuit.

2010 Mathematics Subject Classification. 68R10, 90C35, 05C85, 94C15

Contents

Chapter 1

Introduction

The main goal of this paper is implementing a simulator which will visualise the representation of an electrical circuit as well as solve it efficiently. Graph theory, linear algebra as well as computer science, more specifically programming in C++, are the main subjects that will be used.

This paper is composed of seven chapters. The first chapter is this introduction, which gives a historical approach of electrical circuits as well as introduces the most important physicists who discovered the laws that allow its solving. Some basic concepts of the elements of an electrical circuits as well as said laws are also explained in it. Once some basic concepts of graph theory are explained, the second chapter describes the Breadth-first search algorithm, which is a hierarchical algorithm for graphs that allows to find a spanning tree. In the first part of the third chapter the data structure needed to represent an electrical graph and a meshed tree are explained. In the second part, two crutial functions of the program are described. The first one is an expansion of the Breadth-first search algorithm and it allows to find a meshed tree and its fundamental cycles. The second one is the function that solves the electrical circuit. In the fourth chapter the visual part of the simulator is described and some of the main functions of the graphics and control libraries are explained. Then an overview of the implementation of the program is given by means of a flowchart, that gives a visual description of the program with its functions and the relation between them. In the fifth chapter a detailed example of how to solve an electrical circuit is given. The results of the program are shown and compared to the results found when the electrical circuit is solved by hand. Finally, the conclusions are given in the sixth chapter.

1.1 Historical introduction

An electrical circuit includes devices that give energy to the charged particles constituting the current, such as a battery, devices that use current, such as resistors, and the connecting wires. Two of the basic laws that mathematically describe the performance of electric circuits are Ohm's law and Kirchhoff's laws.[1]

Georg Ohm was a German physicist who discovered the law, named after him, which states that the current flow through a conductor is directly proportional to the potential difference (voltage) and inversely proportional to the resistance (described in 1.1). His work greatly influenced the theory and applications of current electricity [2], twenty years later, in 1845, Gustav Kirchhoff, who also was a German physicist, first announced Kirchhoff's laws (described in 1.2 and 1.3), which allow calculation of the currents from the voltages and resistances in electrical circuits.[3].

1.2 Main idea

Graph theory will allow us to model an electrical circuit into a graph, where the nodes of the electrical circuit correspond to the vertices of the graph and the wires of the electrical circuit correspond to the edges of the graph. Therefore the information regarding wires, i.e. voltage, resistance and electric current in each wire, will be kept as information related to an edge.

To give a general idea on how to solve an electrical circuit, we will need to apply the Kirchhoff laws. Kirchhoff's first law (1.2) will give us nodal linear equations and Kirchhoff's second law (1.3) will give us meshed linear equations. A linear sistem can be written by joining these equations. Finally, solving the system of linear equations will give the intensities which traverse each wire of the electrical circuit.

To sum up, the data that will be given is the voltages and the resistances of an electrical circuit, once we solve the electrical circuit, the intensities of each wire will be found.

We need an efficient way of traversing graphs and an efficent way to keep all the data. Having the structure of a graph is essencial because hierarchical algorithms will be crucial to solve a general electrical circuit, mainly to write down the meshed linear equations.

1.3 Electrical Circuits: basic concepts

This section aims to remind briefly the fundamental concepts regarding electrical circuits so the reader is familiarized with them.

An **electrical circuit** is an interconnection of electrical elements which are interconneting wires that contain **direct current** (current that remains constant with time). There are two types of elements found in electric circuits: passive elements and active elements. An active element is capable of generating energy while a passive element is not. Resistors are an example of passive elements and generators and batteries are examples of active elements. The associated magnitudes to these elements are:

- The **electromotive force** or voltage is the energy required to move a unit charge through an element like generators or batteries. It is measured in volts (V).
- The **resistance** of an element, like a resistor, denotes its ability to resist the flow of electric current. It is measured in ohms (Ω).
- The **electric current** or current intensity is the time rate of change of charge in a wire. It is measured in amperes (A).

A **branch** represents a single element such as a voltage source (which can be positive or negative) or a resistor; a **node** is the point of connection between two or more branches and a **cycle** or a **mesh** is any closed path in a circuit [5].

Circuit analysis is the process of determining the currents through the elements of the electrical circuit. To determine these values the following laws will be used:

Ohm's law. Ohm's law states that the voltage V across a resistor is directly proportional to the current I flowing through the resistor.

$$V = IR \tag{1.1}$$

Notice that equation 1.1 gives the following equality:

$$\Omega = \frac{V}{A}$$

Kirchhoff's current law. Kirchhoff's current law (or Kirchhoff's first law) states that the algebraic sum of currents entering a node is zero. Specifically, the sum of the currents entering a node is equal to the sum of the currents leaving the node. Equivalently:

$$\sum_{k=1}^{n} I_k = 0 \tag{1.2}$$

Where n is the number of branches which are connected to the node and I_k is the $k-$th current which is entering or leaving the node.

Kirchhoff's voltage law. Kirchhoff's voltage law (or Kirchhoff's second law) states that the algebraic sum of all electromotive forces around any cycle is zero. Specifically, the sum of voltage drops is equal to the sum of voltage rises. This means that, for each mesh of the electrical circuit, the algebraic sum of the electromotive forces given by the batteries has to be the same as the algebraic sum of the voltage drops in the resistances R traversed by the current intensity I (this will be given by 1.1). Equivalently:

$$\sum_{k=1}^{n} I_k R_k = \sum_{k=1}^{n} \mathcal{E}_k \tag{1.3}$$

Where n is the number of branches forming the cycle and $\mathcal{E}_k$ is the $k-$th electromotive force on the $k-$th branch.

The following figure (1.1) is an example of how an electrical circuit with one cycle is represented. The circles represent the nodes, the lines represent the wires, the arrows represent the intensities, the couple of perpendicular lines represent the batteries and the zig-zag line represents the resistors.

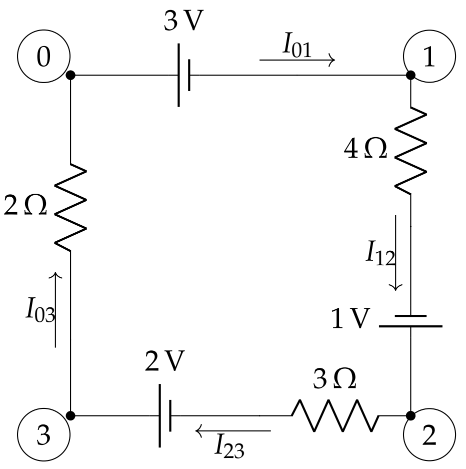


Figure 1.1: Example of an electrical circuit

Chapter 2

Graph traversal algorithms for spanning trees

The modelization of an electrical circuit is done by means of graphs. How this is exactly done will be explained in the next chapter. This chapter aims to illustrate how we will implementate graph hierarchical traversal algorithms to find a spanning tree. An extended version of this algorithm will allow us to find a meshed tree, which will allow us to retrieve the information of the cycles in an electrical circuit. The first section aims to remind briefly the fundamental concepts regarding graphs so the reader is familiarized with them. The second section describes the algorithm that will be used to traverse a graph and extract the required information.

2.1 Basic concepts

A **graph** $G(V, E)$ is a finite set of **vertices** $V = \{v_1, ..., v_{v_n}\}$ and a finite set of **edges** $E = \{e_1, ..., e_{e_n}\}$ where each edge connects two vertices. If those two vertices are the same then the edge will be named a **loop**. Two vertices are **neighbours** if they are joined by an edge, i.e. the vertices are **incident** on the edge.

A graph is **connected** if there exists a path between every pair of vertices. The largest connected subgraphs of a graph $G(V, E)$ are called **connected components**. A **closed walk** of length k is a sequence of edges $e_1, ..., e_k$ starting and finishing at the same vertex, a **cycle** is a closed walk where all edges and intermediate vertices are different. A graph will be named **tree** if it is a connected graph without cycles. Trees are some of the most frequent graphs used in mathematical modeling and computation [6].

Given a graph $G(V, E)$, a **spanning tree** is subgraph which is a tree with all vertices of $G(V, E)$ covered with the minimum possible number of edges. It will be explained in the next section how to find a spanning tree given a graph using a graph algorithm.

Since we will use graph algorithms we need to know how a graph is represented. An **adjacency list** is a collection of unordered lists used to represent a finite graph. Therefore an adjacency list is an array of v_n lists, one for each vertex $v \in V$, where each list contains the set of neighbours of v.

2.2 Breadth-first search

The *Breadth-first search* algorithm, BFS to abbreviate, is a graph search algorithm. Applying BFS one is able to find the shortest path between two points. This algorithm has many applications like GPS navigation systems or social networking websites among others. Breadth means width, this indicates how the graph and its vertices will be visited when applying the BFS algorithm. Note that we do not need to have a tree shaped graph to begin with, however, given any type of graph the outcome will be a tree. We call spanning trees the trees generated using the *Breadth-first search* algorithm.

As it will be explained in Section 3.3 we will need to apply a modified version of the BFS algorithm to a given graph in order to have a meshed tree.

2.2.1 BFS function

This section aims to explain how the function *BFS* works. *BFS* will be called in the *BFS_Meshed_Trees* function which will be explained in Section 3.3. The function takes a graph G as a parameter and it returns a component Tn which is an integer with the number of connected components of the graph G.

Firstly an array of vertices of size v_n will be declared and an array of booleans of size v_n will be initialized to false. The first one will store the vertices by order of visit. The second one, *visited*, will store whether a vertex has been visited or not, i.e. if a vertex v has been visited then $visited[v] = true$ if it has not been visited then $visited[v] = false$. There will also be an index $n = 0$.

Secondly a loop that iterates all the vertices will take place. If the actual vertex has not been visited then the following steps will be made:

- First step: Set a second index $i = n$. Update the number of components Tn. Set the actual vertex v as the root. Mark the root as visited and add the root to the array of visited vertices in the position $n + 1$. If the actual vertex has adjacent vertices then we will do:

- Second step: The vertex in the position $i+1$ will be chosen. A loop will iterate every adjacent vertex of v_i, if it has not been visited then it will be marked as visited and added to the array of visited vertices in the position v_{n+1}. This will be done while $i < n$.

This means once one of the vertices of the graph has been picked as the root, then all the adjacent vertices that are on the same level of depth will be visited. Once all the adjacent vertices have been visited we will be able to visit the next level of depth. The same steps will be followed recursively until all the vertices which are reachable from the root are visited. If all vertices belong to the same connected component then the loop will stop after the first vertex. This is because all the vertices are reachable from the root therefore there will be just one connected component of the graph G.

2.2.2 BFS pseudocode

There are many ways of programming a BFS algorithm. The following pseudocode for the BFS algorithm gives a general idea of the steps needed to program it:

Algorithm 1 BFS (G,r)

```
    Input: graph $G(V,E)$, root $r \in V$
    Output: none
  Let $V$ be an array
  Add root $r$ in $V$
  while $V$ is not empty do
      remove first vertex, $u$, of $V$
      for all edges $(u,v) \in E$
      if $v$ not visited then
          visit vertex $v$
          $u$ is $v$'s parent
          add $v$ in $V$
      end if
  end while
```

2.2.3 BFS Example

The next pages are an example of how to implement the basic BFS algorithm. We will use orange to mark the vertex being processed, pink to mark the unvisited vertices, blue to mark the visited vertices and yellow for the edges.

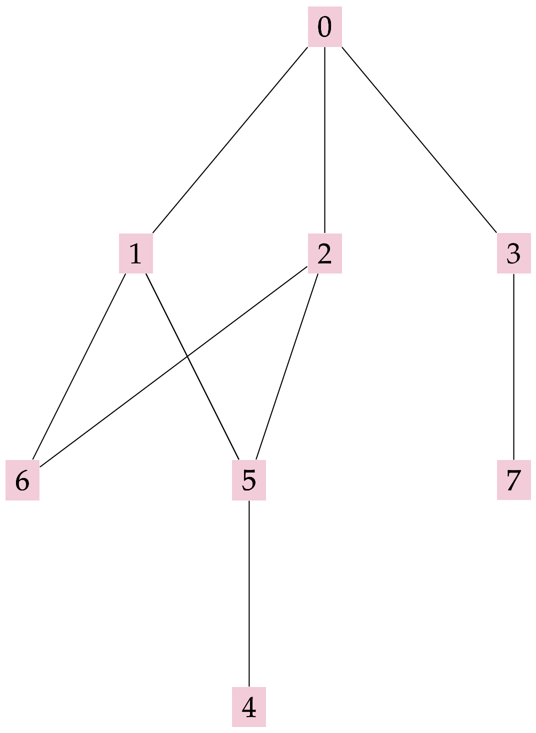


Figure 2.1: Given a graph G

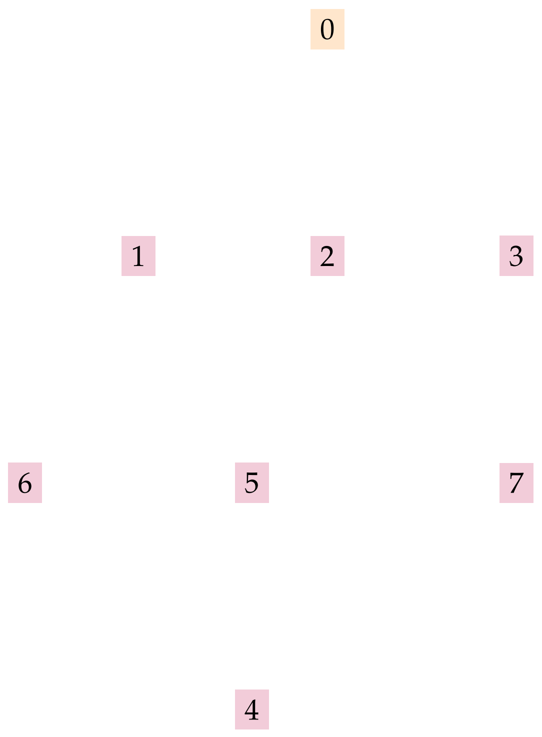


Figure 2.2: First iteration: we set the vertex 0 as the root. Therefore it has no parent and it will be added to the array. $V = [0]$.

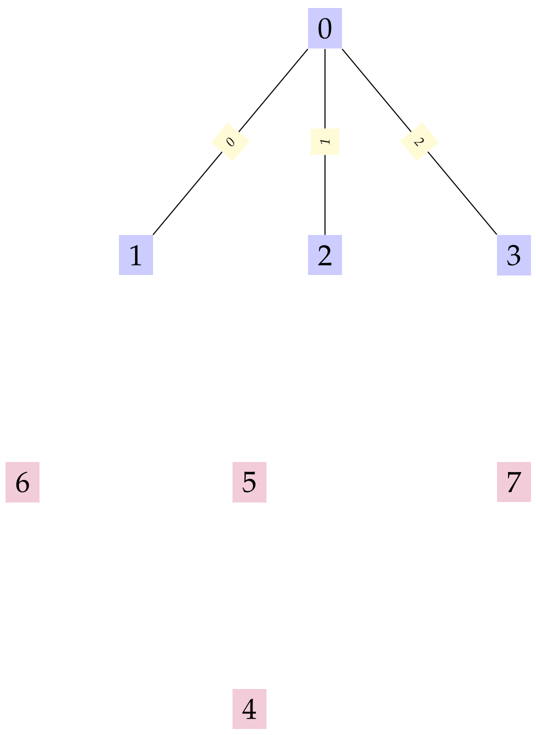


Figure 2.3: Second iteration: we will visit the roots' neighbours, i.e. the vertices 1,2 and 3. Therefore the parent of those vertices will be the root 0. We will add those vertices to the array, $V = [0, 1, 2, 3]$. Since all the neighbours of 0 have been visted the root is popped out of the array, $V = [1, 2, 3]$.

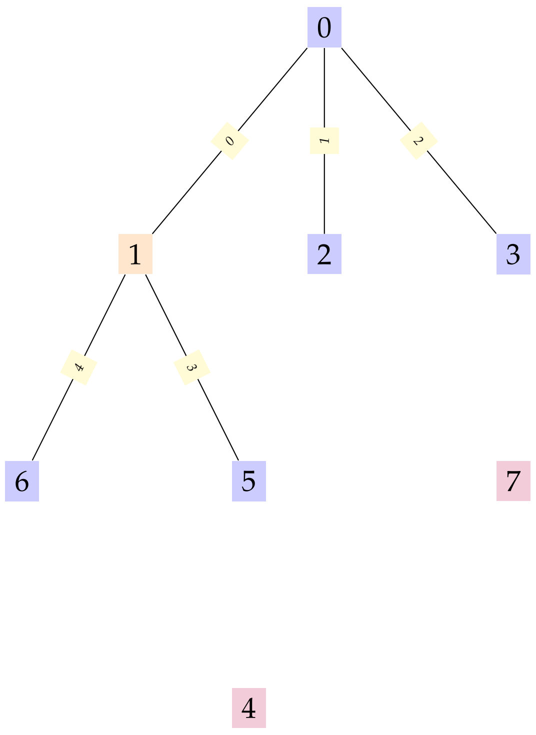


Figure 2.4: Third iteration: we will visit the neighbours of the vertex 1, i.e. the vertices 5 and 6. Therefore the parent of those vertices will be the vertex 1. We will add those vertices to the array, $V = [1, 2, 3, 5, 6]$. Since all the neighbours of 1 have been visted the vertex 1 is popped out of the array, $V = [2, 3, 5, 6]$.

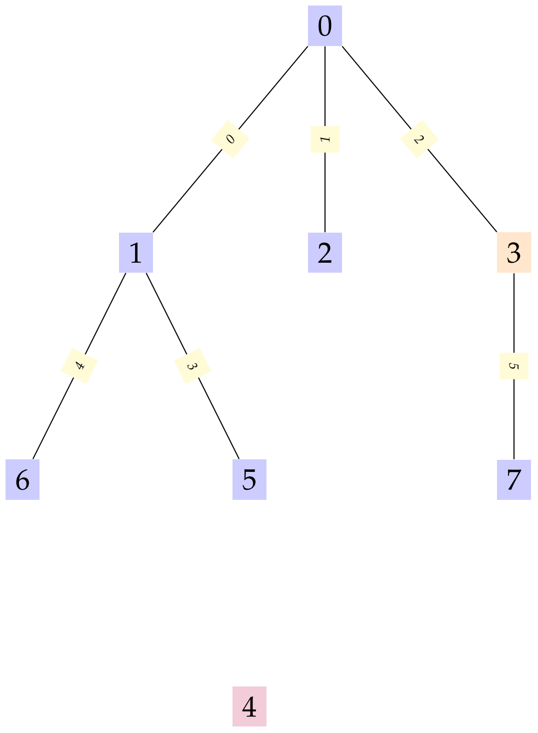


Figure 2.5: Fourth iteration: we will visit the neighbours of the vertex 2, i.e. the vertices 5 and 6. Since all the neighbours of 2 have already been visted the root is popped out of the array, $V = [3, 5, 6]$. The next vertex to visit is the vertex 3 which has the vertex 7 as a neighbour, since it has not been visited yet it will be added to the array $V = [3, 5, 6, 7]$. Therefore the parent of the vertex 7 is the vertex 3 and this vertex is popped out of the array, $V = [5, 6, 7]$.

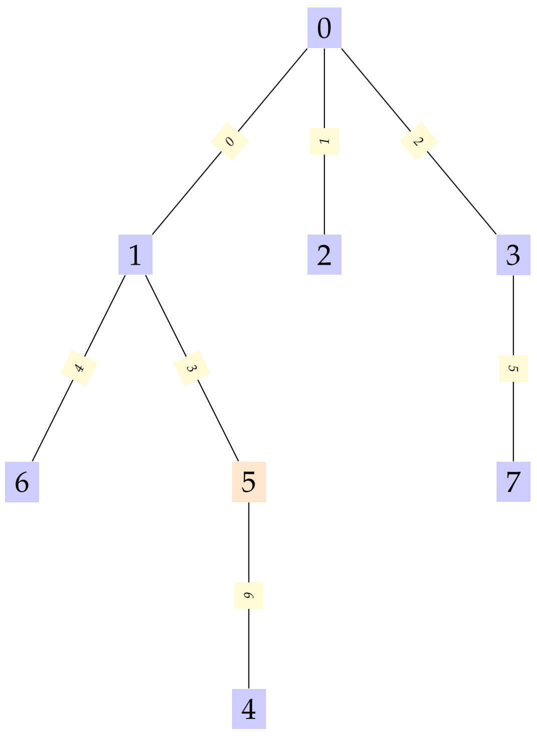


Figure 2.6: Fifth iteration: we will visit the neighbour of the vertex 5, which is the vertex 4. Therefore the parent of this vertex is the vertex 5. We will add this vertex to the an array, $V = [5, 6, 7, 4]$. Sixth iteration: since the neighbours of the vertices 5, 6, 7 and 4 have already been visited those vertices will pop out of the an array. Implementing the BFS algorithm to our graph gives us the spanning tree from above. Note that the edge from the vertex 2 to the vertex 5 and from the vertex 2 to the vertex 6 have been removed since we are looking for a tree, i.e a graph with no cycles.

In the next chapter meshed trees and their data structure will be explained. In advance, the following figure illustrates how the meshed tree corresponding to the graph G (figure 2.2.3) looks like:

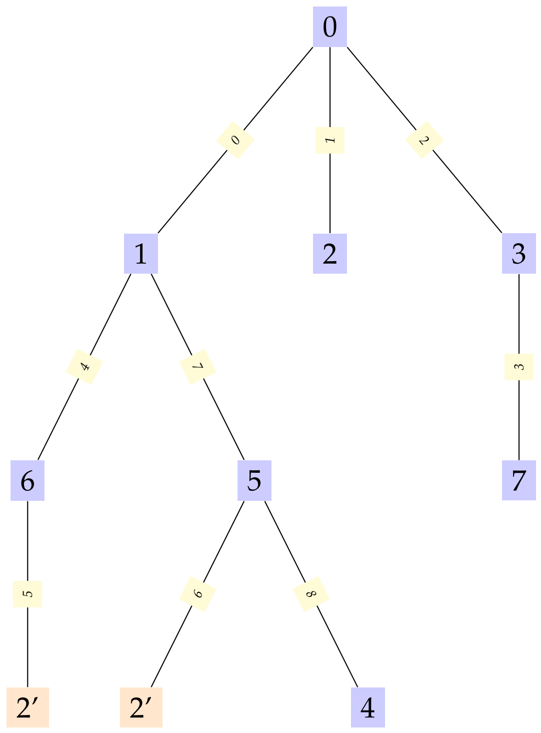


Figure 2.7: Meshed tree with 2′ as an alias node and 5 and 6 as closure branches. This means 6-1-0-2 and 5-1-0-2 form two fundamental cycles.

Chapter 3

Data structures and relevant functions

3.1 Kirchhoff graph

An electrical circuit can be seen as a connected graph where the nodes of the electrical circuit are the vertices of the graph and the wires of the electrical cricuit are the edges of the graph. This will be named **Kirchhoff** or **electrical graph**.

The aim of this chapter is to explain how the data is structured. An electrical circuit will be given and we will need to be able to restructure it in order to have a graph, more specifically, a tree with extra information called meshed tree. Therefore we need to know how the data file will look like in order to know how to read it with the data of the electrical circuit.

The aim of this chapter is to explain how an electrical circuit is transformed into a meshed tree. In this section an example will be used to explain the steps needed.

Given the following electrical circuit in 3.1.

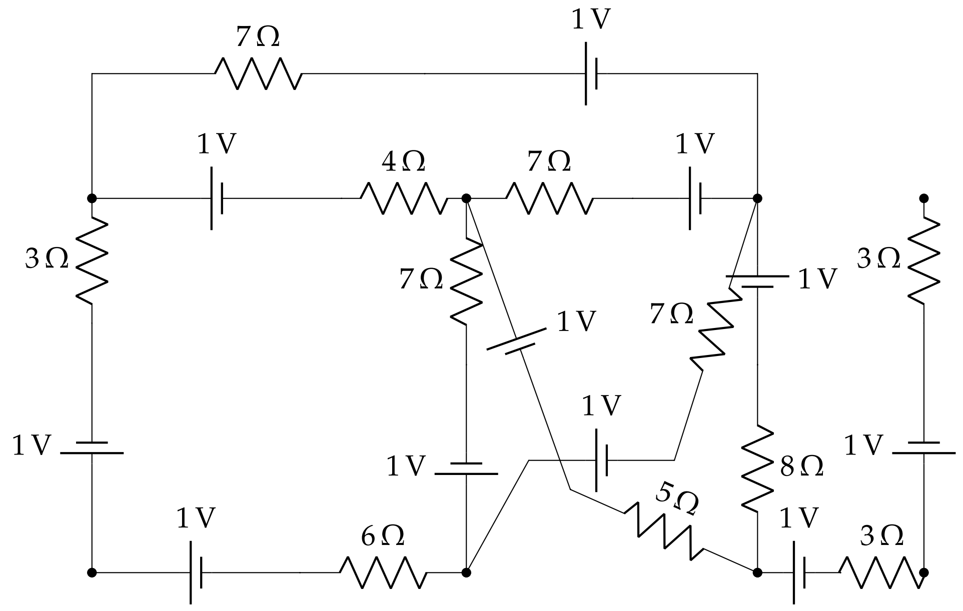


Figure 3.1: Electrical circuit

Firstly we will take all the data given in the file and read it. For example, the file corresponding to the figure 3.1 will have the following information:

```
8   11
0    2   4
1    2   22
2   22   4
3   22   22
4   32   0
5   38   22
6   38   12
7   38   4
0    0   1   3   1
1    0   2   4   1
2    5   2   5   1
3    1   3   6   1
4    2   3   7   1
5    2   4   7   1
6    3   4   7   1
7    0   4   7   1
8    4   5   8   1
9    5   6   3   1
10   7   6   2   1
```

Let us analyse what the numbers in each line of the file mean:

1. 8 is the number of nodes, 11 is the number of edges.
2. Node 0 is in the position (2,4).
3. Node 1 is in the position (2,22).
4. Node 2 is in the position (22,4).
5. Node 3 is in the position (22,22).
6. Node 4 is in the position (32,0).
7. Node 5 is in the position (38,22).
8. Node 6 is in the position (38,12).
9. Node 7 is in the position (38,4).

10. Edge 0 connects Node 0 and Node 1, there is one resistor of 3 Ω and one battery of $1V$.
11. Edge 1 connects Node 0 and Node 2, there is one resistor of 4 Ω and one battery of $1V$.
12. Edge 2 connects Node 5 and Node 2, there is one resistor of 5 Ω and one battery of $1V$.
13. Edge 3 connects Node 1 and Node 3, there is one resistor of 6 Ω and one battery of $1V$.
14. Edge 4 connects Node 2 and Node 3, there is one resistor of 7 Ω and one battery of $1V$.
15. Edge 5 connects Node 2 and Node 4, there is one resistor of 7 Ω and one battery of $1V$.
16. Edge 6 connects Node 3 and Node 4, there is one resistor of 7 Ω and one battery of $1V$.
17. Edge 7 connects Node 0 and Node 4, there is one resistor of 7 Ω and one battery of $1V$.
18. Edge 8 connects Node 4 and Node 5, there is one resistor of 8 Ω and one battery of $1V$.
19. Edge 9 connects Node 5 and Node 6, there is one resistor of 3 Ω and one battery of $1V$.
20. Edge 10 connects Node 7 and Node 6, there is one resistor of 2 Ω and one battery of $1V$.

Every file will have v_n (number of vertices) and e_n (number of edges), followed by v_n lines with the positions (x, y) of each vertex and, finally, e_n lines with the vertices that are connected in pairs by edges and the resistance and voltage corresponding to each edge.

Secondly, this information will be saved using matrices and arrays. For each file of an electrical graph the following data will be available:

- The graph will be saved using a matrix, i.e. using adjacency lists: KG
- Positions will be saved using an array of points: Kp
- Resitances will be saved using an array of resistances: Kr

- Voltages will be saved using an array of voltages: Kv
- Currents will be saved using an array of currents: Kc

Let us call v the actual vertex, then $KG[v]$ will be the adjacency list of size n of the actual vertex, this means from $i = 0$ to $i < n$ the vertices $KG[v][i]$ are the neighbours of v. Let $vip(v, i)$ be a map which, given a vertex v and an index i, gives an edge $e = KE[vip(v, i)]$ that joins the vertex v with its neighbour $KG[v][i]$.

In our previous example this data will look like:

v	**KG[v]**
0	$[1, 2, 4]$
1	$[0, 3]$
2	$[0, 5, 3, 4]$
3	$[1, 2, 4]$
4	$[2, 3, 0, 5]$
5	$[2, 4, 6]$
6	$[5, 7]$
7	$[6]$

e=KE[vip(v,i)]	**v-KG[v][i]**	**Kr[e]**	**Kv[e]**
0	0-1	3	1
1	0-2	4	1
2	5-2	5	1
3	1-3	6	1
4	2-3	7	1
5	2-4	7	1
6	3-4	7	1
7	0-4	7	1
8	4-5	8	1
9	5-6	3	1
10	7-6	2	1

3.2 Meshed tree

Once we have the information saved we will be able to apply the modification of the *Breadth-first search* algorithm explained in the next section 3.3. Therefore a Kirchhoff graph will be transformed into a meshed tree and we will have access to the fundamental cycles information, which is crucial to solve the main problem and find the intensities of the electrical circuit.

The modifications done to the standard BFS algorithm will allow us to introduce three new concepts, **alias nodes**, **closure branches** and **fundamental cycles**. These concepts will help us avoiding having a graph with cycles but instead a tree shaped graph will be built but maintaining all the information of the cycles from the initial electrical circuit.

When a graph is being traversed, if the current node being processed has a neighbour that has already been visited it means they form a cycle. To avoid it, a closure branch will be added to join the current node to an alias node (i.e. it will be denoted with the same number but adding a $'$). Therefore the meshed tree will have the same amount of edges as the original electrical graph, e_n, and it will have $e_n + 1$ nodes.

There can be many loops in an electrical graph, the key is to find the most convenient in order to solve the lineal equations using the Kirchhoff laws. This allow us to introduce the concept of **fundamental cycle** or **fundamental mesh**. Given a meshed tree found using the BFS modified algorithm, there will be a fundamental cycle for every closure branch that has been added to the meshed tree, therefore there will be $e_n - v_n + 1$ fundamental cycles in a graph.

Finally all the information of the **Kirchhoff graph** will be printed in a file "example.out":

- Graph information:
  - Number of nodes and branches
  - Adjacency list for every node (its adjacent nodes with the number of the branch connecting them and resistance and voltage of named branch)
  - Information of every branch (the nodes it connects, the number of the branch and the resistance and voltage of the branch)
- BFS information:
  - Vertices of the meshed tree (its alias, its visited order whilst doing the breadth-first search, its depth and its parent)
  - Edges of the meshed tree (if the vertex does not correspond to its alias then the edge will be printed as "out tree" otherwise it will be printed as "in tree")
  - Paths to the root for every vertex
  - Fundamental cycles (i.e. the vertices forming every cycle)

3.3 BFS_Meshed_Trees function

As it has been explained in the previous section, the *Breadth-first* algorithm will be slightly modified in order to find meshed trees. The aim of this section is to explain how the *BFS_Meshed_Trees* function finds the fundamental cycles of a meshed tree. Knowing which are the fundamental cycles will allow us to write a part of the linear equations needed to solve an electrical circuit. The function takes the following parameters:

- graph G
- edges GE
- vector<vertex> $BFSv$
- vector<index> $BFSind$
- vector<vertex> $BFSa$
- vector<vertex> $BFSp$
- vector<edge> $BFSe$
- vector<length> $BFSd$

and it returns a component Tn.

The function also modifies the parameters given. First of all we will explain the data structures of the parameters that will be filled in during the execution of the function:

- $BFSv$ is an array of vertices which stores the vertices of the tree, the original vertices will be stored in the positions $0, ..., v_{n-1}$ and the alias vertices in the positions $v_n, ..., e_n$.
- $BFSind$ is an array of indexs which stores the index of each vertex.
- $BFSa$ is an array of vertices. If the vertex in the given position belongs to an original vertex then the array will store the same vertex. On the other hand, if it belongs to an alias vertex then the array will store the original vertex.
- $BFSp$ is an array of vertices which stores the upper vertex (i.e. the parent) of each vertex.
- $BFSe$ is an array of edges which stores the upper edge of each vertex (i.e. the edge that connects the actual vertex to its parent).
- $BFSd$ is an array of lengths which stores the depth of each vertex.

Firstly, the arrays will be declared, their size will be $en + 1$ and $BFSp$ will be initialised to $en + 1$. Then the program will roughly follow the same structure as the *BFS* function (2.1.2) until the Second step.

Let us call v the actual vertex, then $G[v]$ will be the adjacency list of size n of the actual vertex, this means from $i = 0$ to $i < n$ the vertices $G[v][i]$ are the neighbours of v. Let $vip(v,i)$ be a map which, given a vertex v and an index i, gives an edge $e = GEvip(v,i)$ that joins the vertex v with its neighbour $G[v][i]$.

If the adjacency list of v is not empty then two cases will be considered:

- The vertex being processed, $G[v][i]$, is an original vertex. Then the following arrays will be updated:
  - $BFSa[G[v][i]] = G[v][i]$, i.e. the alias array updates in the position of the vertex the vertex itself.
  - $BFSv[n+1] = G[v][i]$, i.e. the vertices array updates the vertex in the $n+1$ position, notice that $n+1 < v_n$ because $G[v][i]$ is an original vertex.
  - $BFSd[G[v][i]] = BFSd[v] + 1$, i.e. the depth array updates the vertex's depth as the parent's depth+1.
  - $BFSp[G[v][i]] = v$, i.e. the parents array updates the vertex's parent as v.
  - $BFSe[G[v][i]] = GE[vip(v,i)]$, i.e. the edges array updates the vertex's upper edge as the edges that joins v with i.
- The vertex being processed, av, is an alias vertex. Then the following arrays will be updated:
  - $BFSa[av] = G[v][i]$, i.e. the alias array updates in the position of the vertex the original vertex.
  - $BFSv[n+1] = av$, i.e. the vertices array updates the vertex in the $n+1$ position, notice that $n+1 > v_n - 1$ because av is an alias vertex.
  - $BFSd[av] = BFSd[v] + 1$, i.e. the depth array updates the vertex's depth as the parent's depth+1.
  - $BFSe[av] = GE[vip(v,i)]$, i.e. the edges array updates the vertex's upper edge as the edges that joins v with i.
  - $BFSp[av+1] = v$, i.e. the parents array updates the next vertex of the actual vertex parent as v.

The following example will show how the meshed tree corresponding to the electrical circuit 3.1 is:

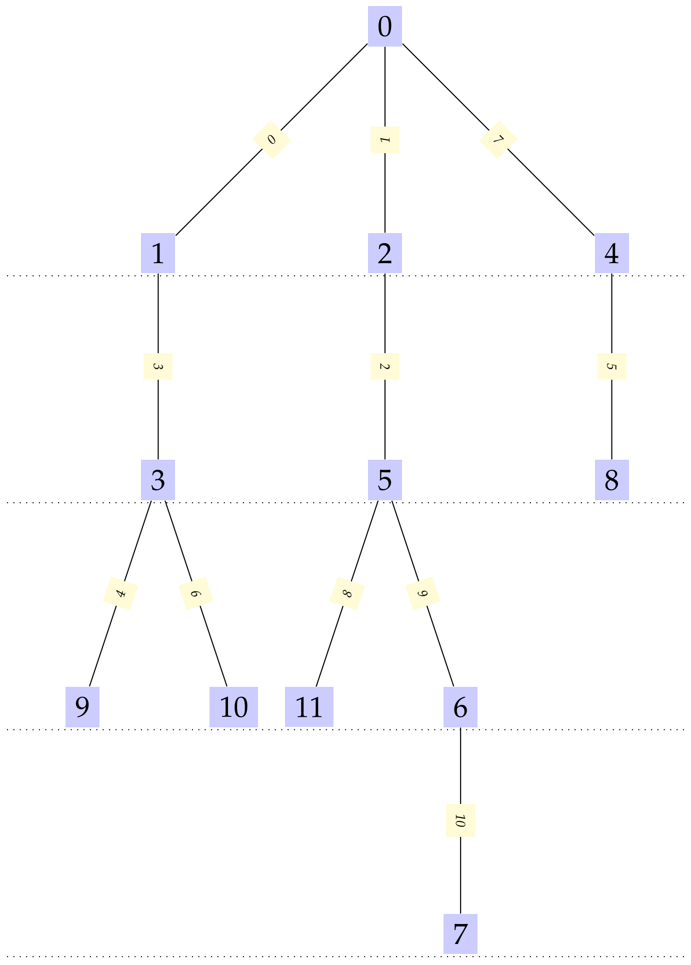


Figure 3.2: Meshed tree with 8, 9, 10 and 11 as alias nodes and 4, 5, 6 and 8 as closure branches.

Let us see how the data of this meshed tree is kept:

v	BFSa[v]	BFSind[v]	BFSd[v]	BFSp[v]	BFSe[v]
0	0	0	0	-	-
1	1	1	1	0	0
2	2	2	1	0	1
3	3	4	2	1	3
4	4	3	1	0	7
5	5	5	2	2	2
6	6	10	3	5	9
7	7	11	4	6	10
8	2	6	2	4	5
9	2	7	3	3	4
10	4	8	3	3	6
11	4	9	3	5	8

3.3.1 Fundamental cycles

With the knowledge of how the *BFS_Meshed_Trees* function works, this section aims to illustrate how the fundamental cycles are found. Given the meshed tree from 3.2 the corresponding fundamental cycles are 3.3, 3.4, 3.5 and 3.6.

Firstly, there will be a loop that iterates all alias nodes, i.e. the vertices from v_n to e_n, in this example 4 alias vertices will be iterated and each one of them will produce a fundamental cycle. The actual alias vertex (painted in orange) will ascend by substituting the actual vertex for its parent until the depth (depth is pictured as a dotted horizontal line) of the actual vertex is the same as the depth (this is pictured with a solid line arrow) of the original vertex (painted in blue).

Secondly, once the depth is equal, both sides will ascend by substituting the actual vertex for its parent until a common vertex (painted in red) is found (this is pictured with a dashed line arrow).

Notice that in the next example the common vertex is the root, the vertex 0, however this is not a generalization as it can happen that the first common vertex found differs from the root.

Finally, in each step the actual vertex is being saved in an array, this array will represent the fundamental cycle. The arrays corresponding to the fundamental cycles are: $[2', 4, 0, 2]$, $[2', 3, 1, 0, 2]$, $[4', 3, 1, 0, 4]$ and $[4, 5, 2, 0, 4]$.

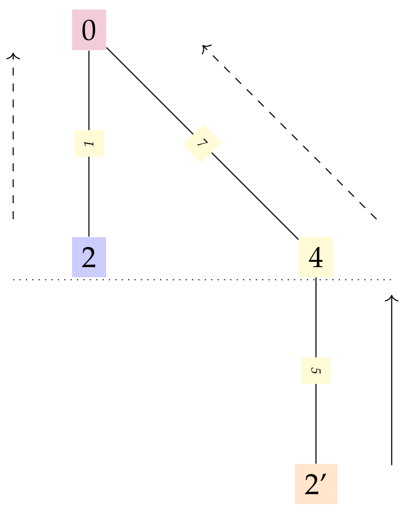


Figure 3.3: Fundamental cycle 1.

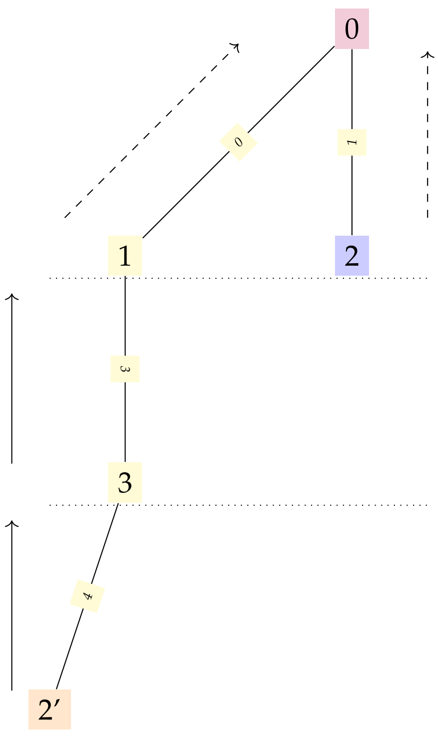


Figure 3.4: Fundamental cycle 2.

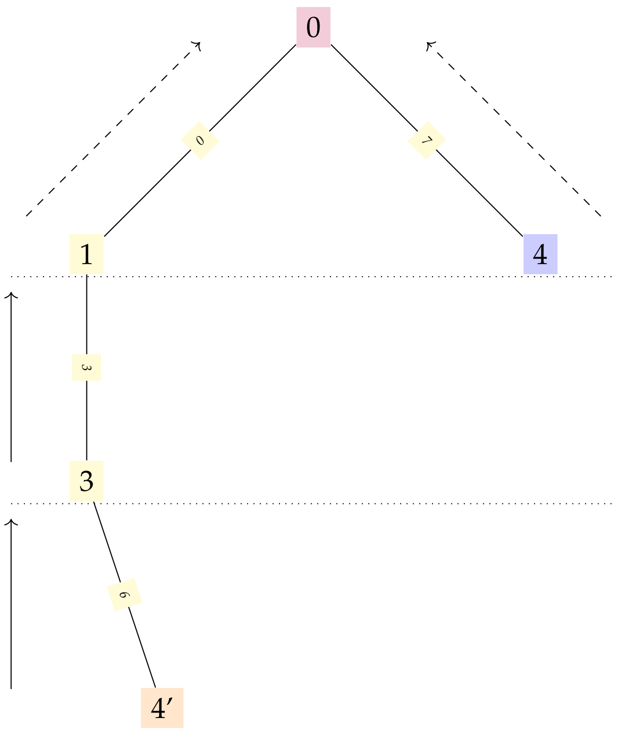


Figure 3.5: Fundamental cycle 3.

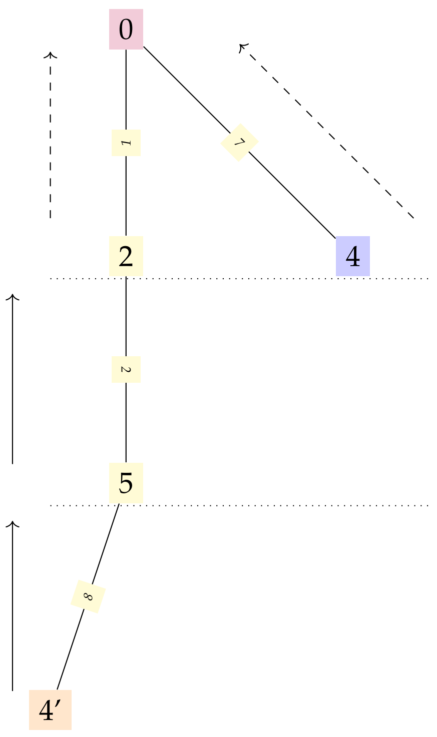


Figure 3.6: Fundamental cycle 4.

3.4 KC_Compute function

This section aims to explain how the function *KC_Compute*, which computes the intensities of a given electrical circuit, works. A linear system of equations can be represented as an augmented matrix, the main goal of the *KC_Compute* function is to fill in a matrix R and an array E that together will compose an augmented matrix. As it has been discussed in the introduction (1.1), the linear system of equations that solves an electrical circuit will be $RI = E$. The augmented matrix will look like the following matrix:

$$\underbrace{\left(\begin{array}{cccc|c} R_{11} & R_{12} & \dots & R_{1e_n} & 0 \\ \vdots & \ddots & \ddots & \vdots & \vdots \\ R_{v_n1} & R_{v_n2} & \dots & R_{v_ne_n} & E_{v_n} \\ \vdots & \ddots & \ddots & \vdots & \vdots \\ R_{e_n1} & R_{e_n2} & \dots & R_{e_ne_n} & E_{e_n} \end{array}\right)}_{R \qquad\qquad\qquad E} \begin{array}{l} \left.\begin{array}{c} \\ \\ \end{array}\right\}\text{Nodal equations} \\ \left.\begin{array}{c} \\ \\ \\ \\ \end{array}\right\}\text{Meshed equations} \end{array}$$

Where E's coefficients corresponding to the nodal equations are 0 (1.2) whereas the coefficients corresponding to the meshed equations are the algebraic sums of the electromotive force in the mesh (1.3).

The function *KC_Compute* takes the following parameters, those are the same arrays used in *BFS_Meshed_Trees*, Section3.3, but with a changed notation to emphasise that the data structure is from a Meshed Tree:

- graph *KG*
- edges *KE*
- resistances *Kr*
- voltages *Kv*
- vector<vertex> *MTv*
- vector<vertex> *MTa*
- vector<vertex> *MTuv*
- vector<edge> *MTue*
- vector<length> *MTd*

First of all, we will describe the data structures of those parameters:

- *Kr* is an array of resistances which stores the resistance of each edge.
- *Kv* is an array of voltages which stores the voltage of each edge.

- *MTv* is an array of vertices which stores the vertices of the tree, the original vertices will be stored in the positions $0, ..., v_{n-1}$ and the alias vertices in the positions $v_n, ..., e_n$.
- *MTa* is an array of vertices. If the vertex in the given position belongs to an original vertex then the array will store the same vertex. On the other hand, if it belongs to an alias vertex then the array will store the original vertex.
- *MTuv* is an array of vertices which stores the upper vertex (i.e. the parent) of each vertex.
- *MTue* is an array of edges which stores the upper edge of each vertex (i.e. the edge that connects the actual vertex to its parent).
- *MTd* is an array of lengths which stores the depth of each vertex.

The function returns a vector<double> E of length e_n. This array will have the intensity related to the i-th edge in the i-th position, i.e:

$$E[0] = I_0, ..., E[k] = I_k, ..., E[e_n - 1] = I_{e_n - 1}$$

Given all this data, the aim of the *KC_Compute* function is to:

1. Fill a $e_n \times e_n$ matrix R and an array E of length e_n that correspond to the system of linear equations of the electrical circuit.
2. Solve the system of linear equations, i.e. find the intensities traveling through each wire of the electrical circuit.

Firstly, the first $v_n - 1$ rows will be filled using the Kirchhoff first law(1.2). This rows correspond to the nodal equations therefore the array E will be null in the positions $0, ..., v_n - 1$. Notice that: even if we have v_n nodes, i.e, v_n nodal equations, only $v_n - 1$ nodal equations are used. This is because nodal equations are not all linearly independent, therefore we will remove one of them to ensure the linear independency.

Let v be the i-th vertex and e the j-th edge. Let u and w be the vertices joined by e where $u < w$. Then the matrix R will be filled using the following process:

$$R[v][e] = \begin{cases} 1 & \text{if } v = w \\ -1 & \text{if } v = u \\ 0 & v \neq u, v \neq w \end{cases}$$

Secondly, the rows v_n-th till the e_n-th will be filled using the Kirchhoff second law(1.3). Those rows correspond to the mesh equations, i.e. for each fundamental mesh belonging to the electrical circuit there will be its corresponding mesh equation. As we have seen in the previous section 3.3, *BFS_Meshed_Trees* computes the fundamental cycles of a spanning tree. A similar approach will be used in order to find the linear equations linked to the fundamental cycles.

In the first instance there will be a loop which will start setting the actual vertex, *av*, to the smaller alias vertex and will loop through all the alias vertices. Each loop will fill the row corresponding to the alias vertex for the matrix R and the position corresponding to the alias vertex for the array E. Then the following steps will be made:

The first step is a loop which will keep running until the depth of the actual vertex and the depth of the original vertex, v, are the same, i.e. $MTd[v] = MTd[av]$. In this step the matrix R will be filled with the resistance value corresponding to the upper edge, *ue*, of the actual vertex, i.e. $R[av][ue] = \pm Kr[ue]$, and the array E will be filled with the voltage value corresponding to the upper edge of the actual vertex, i.e. $E[av] = Kv[ue]$. Finally, the actual vertex will be substituted by its upper vertex, *uv*, which will make the depth diminish by one, i.e. $av = uv$.

We will imagine the next step as a mountain, at the end of one of the sides there will be the original vertex and at the end of the other side there will be the actual vertex, both of them will have the same depth since it has been equalized in the first step. The aim is to go on top of the mountain which is the first vertex in common.

The second step is a loop that will stop when the actual vertex is the same as the original vertex. Analogously to the first step, the position in the matrix R corresponding to the upper edge of the actual vertex will be updated with its corresponding value of the resistance and the position in the array E corresponding to the upper edge of the actual vertex will be updated with its corresponding value of the voltage. Then the actual vertex will be substituted by its upper vertex.

The original vertex's side will follow the same steps, update its postion in R and E corresponding to the upper edge with the corresponding resistance and voltage and finally the original vertex will be substituted by its upper vertex.

Finally, once the matrix R and the array E are filled, they will be given as parameters of a function called *Gauss*.

The function *Gauss* takes a matrix A , an array b, a dimension n and a tolerance tol as parameters. Then it applies the Gaussian elimination to the augmented matrix composed by the matrix A and the array b. The solution of the linear system of equations represented by the augmented matrix will be in b.

This means that once we apply *Gauss* the solution of the system of equations will be in E, i.e. the array E will be the array of intensities I. This same array, E, will be the one that *KC_Compute* returns.

Chapter 4

Implementation of the simulator

Now that the main functions of the program have been explained this chapter aims to explain how the implementation has been done and how the simulator works.

4.1 OpenGL, GLUT and FLTK

4.1.1 Definitions

Firstly we will introduce what OpenGl and FLTK are.

OpenGL stands for Open Graphics Library. It is a cross-language, cross-platform API, which stands for application programming interface, for rendering 2D and 3D vector graphics. The API is typically used to interact with a GPU, which stands for graphics processing unit, to achieve hardware-accelerated rendering.[7]

It should not be considered a library, it is a specification that gives the information of what is it achievable with the API but there is no implementation, hence why it is not an actual source code.[8]

A typical program that uses OpenGL begins with calls to open a window into the framebuffer into which the program will draw. Then, calls are made to allocate a GL context and associate it with the window. Once a GL context is allocated,the programmer is free to issue OpenGL commands.[9]

GLUT is the OpenGL Utility Toolkit, a window system independent toolkit for writing OpenGL programs. It implements a simple windowing application programming interface (API) for OpenGL. GLUT makes it considerably easier to learn about and explore OpenGL Programming. [10]

The Fast Light Tool Kit (FLTK) is a cross-platform C++ GUI Toolkit. FLTK provides modern GUI functionality without the bloat and supports 3D graphics via OpenGL and its built-in GLUT emulation. [11]

4.1.2 Functions

We will give some examples so the reader can see how OpenGL, GLUT and FLTK work:

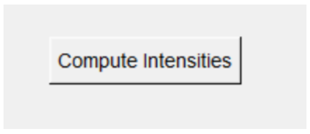
- ***GL_Line*(int x1, int y1, int x2, int y2, rgb colour)**: the function creates a line of colour *colour* which joins the vertex $(x1, y1)$ to the vertex $(x2, y2)$.
- ***GL_Box*(int x1, int y1, int x2, int y2)**: the function creates a rectangle with vertices $(x1, y1)$, $(x1, y2)$, $(x2, y1)$ and $(x2, y2)$.
- ***GL_String*(int xp, int yp, char* text)**: the function writes the characters $text$ of colour $colour$ in the starting position given by the vertex (xp, yp). The function ***CText*(int xp, int yp, char* text, rgb colour)** uses *GL_String* to write the $text$ in a given $colour$.

For example, in the functions *KC_vis* and *MT_vis*, which are in charge of drawing the Kirchhoff graph and the corresponding meshed tree respectively, *GL_Line* will be used to draw the edges and the arrows, *GL_Box* will be used to draw the boxes around the number that represents an edge and a node and *CText* will be used to draw the number of the edge and the node.

- ***Fl_Window*(int w, int h, char* title)** : the function creates a widget of width w and hight h with the string *title* on the top left of the window.
- ***Fl_Output*(int x, int y, int w, int h, char* text)** : the function creates a widget of width w and hight h in the starting position (x, y) with the string $text$.
- ***Fl_Button*(int x, int y, int w, int h, char* text)** : the function creates a widget of width w and hight h in the starting position (x, y) with the string $text$.
- ***Fl_Counter*(int x, int y, int w, int h, char* text)** : the function creates a widget of width w and hight h in the starting position (x, y) with the string $text$.

The function *Control()*, which is in charge of creating all the control buttons, will use the previous functions to achieve it. Take into consideration that buttons generate callbacks when they are clicked by the user[12], as it is illustrated in the next example:

The *KC_Compute* function, explained in Section 3.4, will be called only if the "Compute Intensities" (4.1.2) button is pressed.



Given the following instructions:

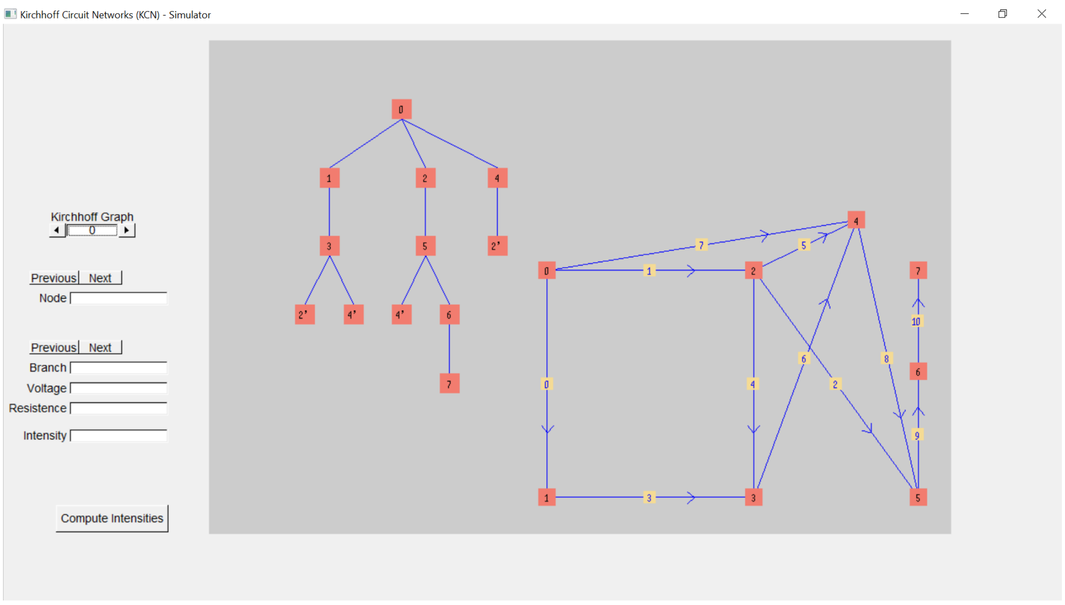
```
Fl_Button *Compute = new Fl_Button(80, 700, 170, 40, "Compute Intensities");

Compute->callback(Compute_cb);
```

The first instruction creates de "Compute Intensities" button in the starting position $(80, 700)$ with width 170 and hight 40. The second instruction connects the button to the function *Compute_cb*, and this functions calls *Compute*. *Compute* will update the *Kc* array by means of the following instruction:

```
Kc = KC_Compute(KG, Kp, KE, Kr, Kv, MTv, MTa, MTuv, MTue, MTd);
```

4.2 Kirchhoff Circuit Networks Simulator

It can be seen in the next screenshot how the functions described in the previous section are implemented. When the program is executed the following window will pop up:

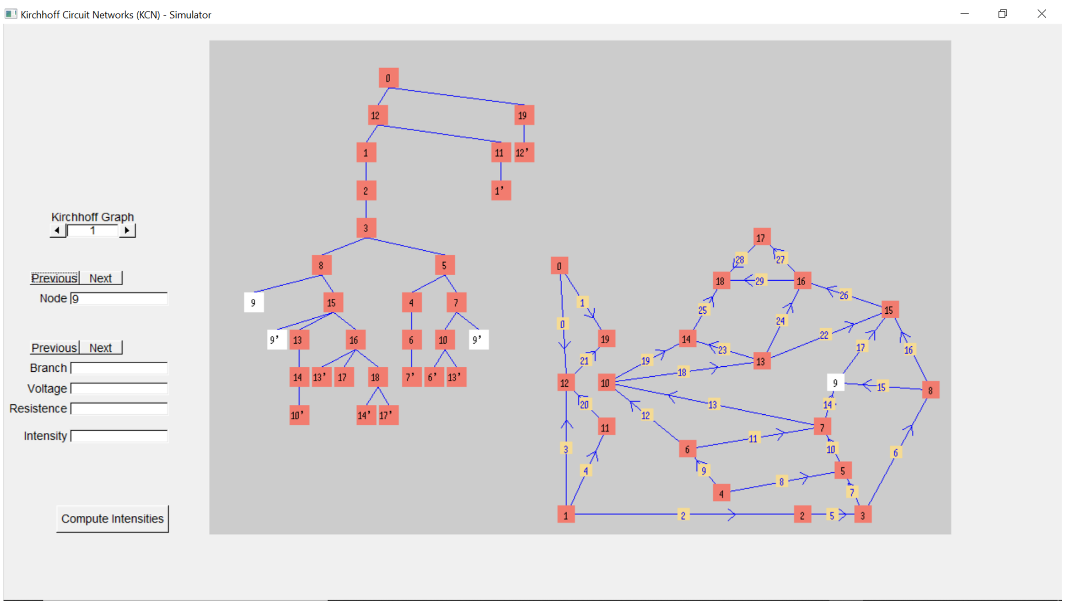


As it can be seen on the left hand side the following features are available:

- Change between different examples of Kirchhoff Graph: press the arrows underneath "Kirchhoff Graph".
- Move from one node to another: press the previous or next buttons above "Node".
- Move from one edge to another: press the previous or next buttons above "Branch".
- Compute the intensites of the electrical circuit: press the button "Compute Intensities".

On the left side of the dark coloured window there is the spanning tree of the graph being analysed and on the right side there is the Kirchhoff graph.

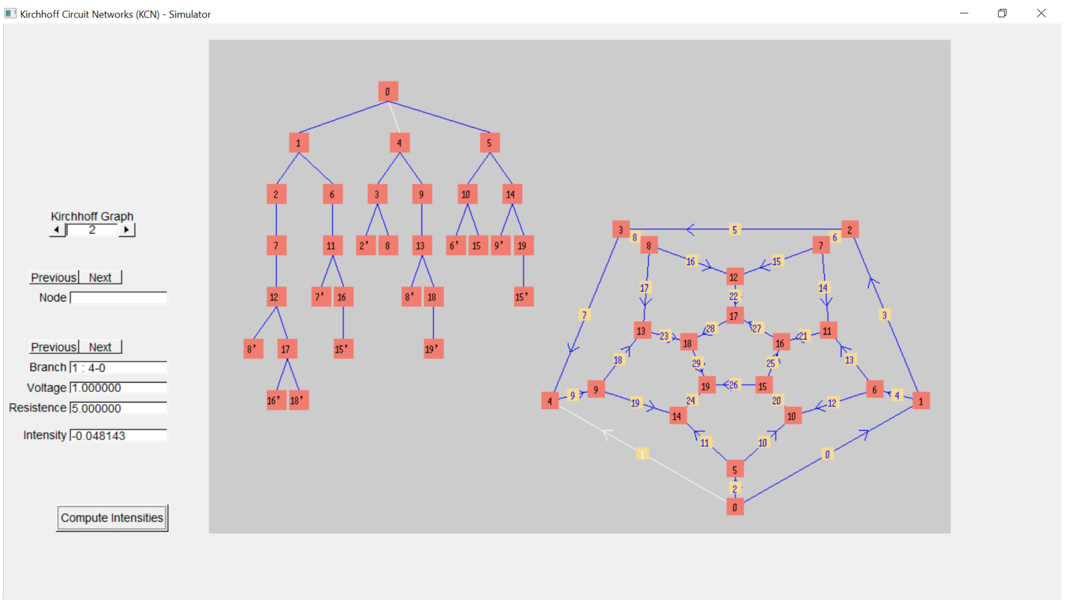
If we hover over a node the node of both the spanning tree (including the alias nodes) and the Kirchhoff graph will change to white, the number of the node will appear in the buttons zone on the left hand side of the screen:



Node 9
Branch
Voltage
Resistence
Intensity

Note that it is not needed to hover over a node to highlight it. If the previous and next buttons over "Node" are pressed the original node and its alias will also turn into white.

If we hover over an edge the edge of both the spanning tree and the Kirchhoff graph will change to white, the number of the edge as well as its voltage and resistance will appear in the buttons zone on the left hand side. The intensity of each edge will always be null if we do not press the "Compute Intensities" button. Once we push it each value will change and will appear as in the next screenshot.



Branch 1 : 4-0
Voltage 1.000000
Resistence 5.000000
Intensity -0.048143

Note that it is not needed to hover over an edge to highlight it. If the previous and next buttons over "Branch" are pressed the edges will also turn into white.

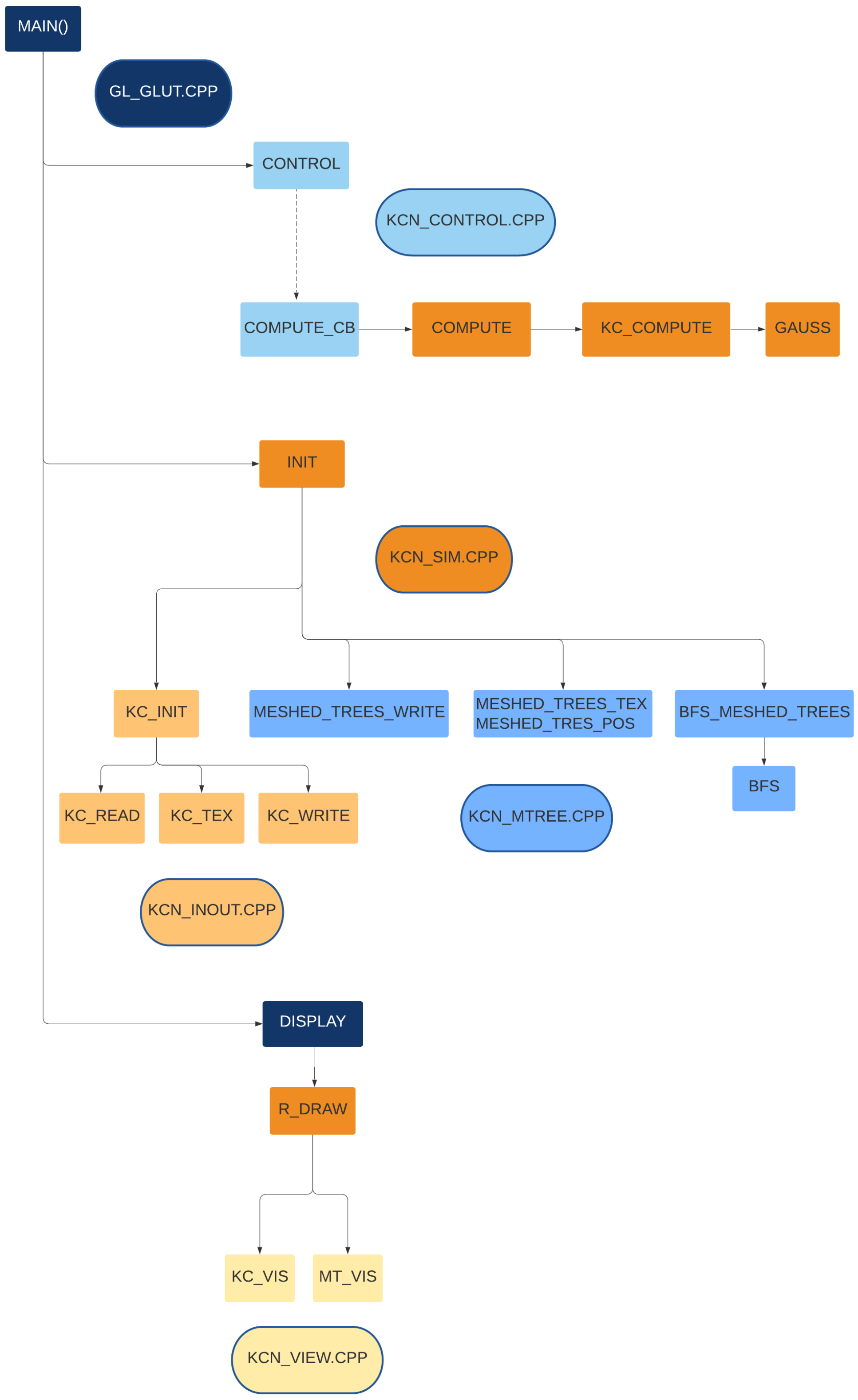
4.3 Implementation

4.3.1 Flowchart

This section aims to provide an insight into the program execution by means of a flowchart as well as give a direct access to the functions explained in this paper.

The program will start with the execution of the **main()** which will call the following functions:

- *Control*, as it has been explained in Subsection 4.1.2 the *Compute_cb* function will be called only when the "Compute Intensities" button is pressed, hence why it is shown as a dotted arrow in the flowchart.
  - *KC_Compute* which has been explained in Section 3.4 it uses the *Gauss* function to calculate the intensities.
- *Init* will call the following functions:
  - *BFS_Meshed_Trees* which has been explained in Section 3.3. It will call the following function:
    - *BFS* which has been explained in Subsection 2.1.2.
- *Display* will call the following functions:
  - *KC_vis* and *MT_vis* both of them have been explained in Subsection 4.1.2.


MAIN()
GL_GLUT.CPP
CONTROL
KCN_CONTROL.CPP
COMPUTE_CB
COMPUTE
KC_COMPUTE
GAUSS
INIT
KCN_SIM.CPP
KC_INIT
MESHED_TREES_WRITE
MESHED_TREES_TEX
MESHED_TRES_POS
BFS_MESHED_TREES
BFS
KC_READ
KC_TEX
KC_WRITE
KCN_MTREE.CPP
KCN_INOUT.CPP
DISPLAY
R_DRAW
KC_VIS
MT_VIS
KCN_VIEW.CPP

Chapter 5

Results

The aim of this chapter is to prove that the program works properly.

For this purpose, an example on how to find the intensities of each wire of the electrical circuit will be explained step by step. Given an electrical circuit this will be modeled into an electrical graph, then its corresponding spanning tree will be found. Lastly, the linear equations associated to the electrical cicuit will be solved using an online linear equations solver. At the end of the chapter the program will be run using the data of the same electrical circuit and both results will be compared.

5.1 Example

Given the electrical circuit from the figure 3.1. The corresponding Kirchhoff graph is the figure 5.1.

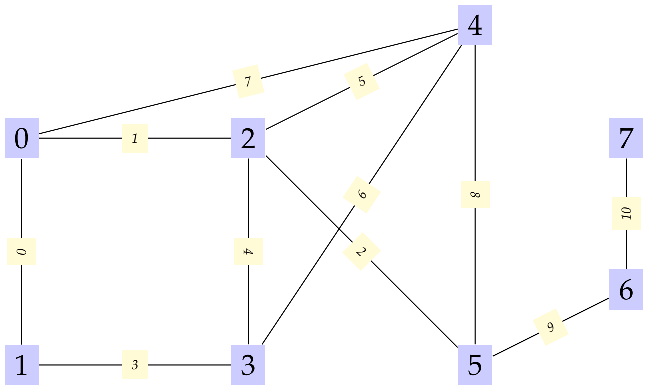


Figure 5.1: Kirchhoff graph

Now the directions of the currents need to be set. The intensity of the branches will be set as follows: the intensity will be moving from the node with the smallest index to the other node. This means we will have an electrical circuit like in the figure 5.2:

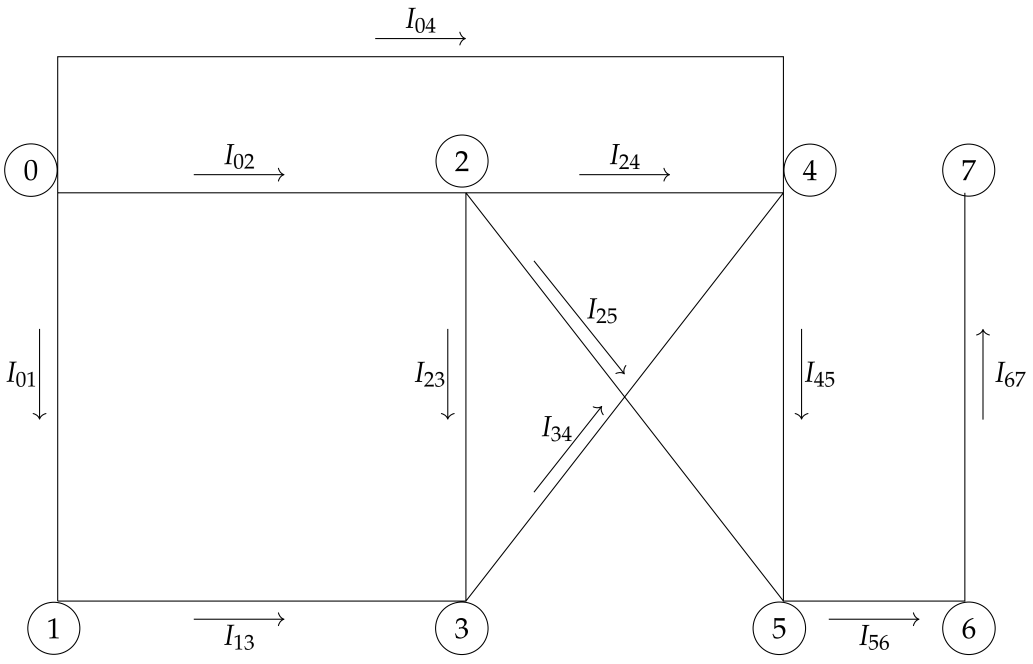


Figure 5.2: Electrical circuit with nodes and intensities

5.1.1 Step 1

The first step will be applying the *BFS_Meshed_Trees* algorithm to the electrical graph. This will give us the corresponding meshed tree as we can see in 5.3. As it has been explained in Section 3.3, when we compute the meshed tree, the fundamentals cycles of the initial Kirchhoff graph can be found.

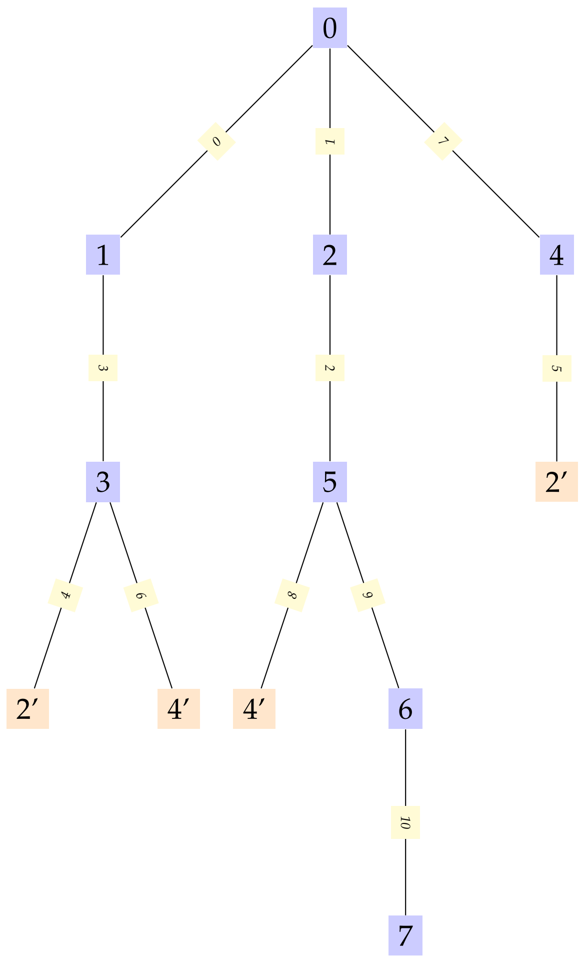


Figure 5.3: Meshed tree with 2′ and 4′ as alias nodes and 4, 5, 6 and 8 as closure branches.

The following figures 5.4, 5.5, 5.6 and 5.7 represent the fundamental cycles found after computing the meshed tree pictured in the figure 5.3.

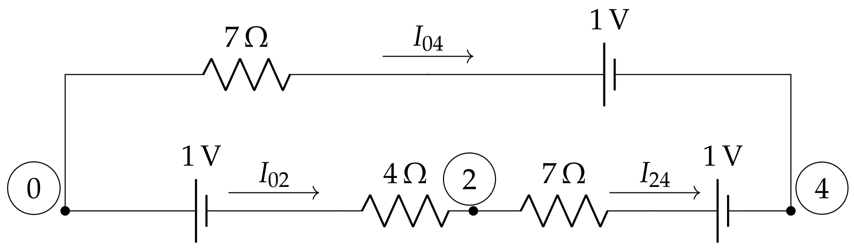


Figure 5.4: Fundamental cycle 1

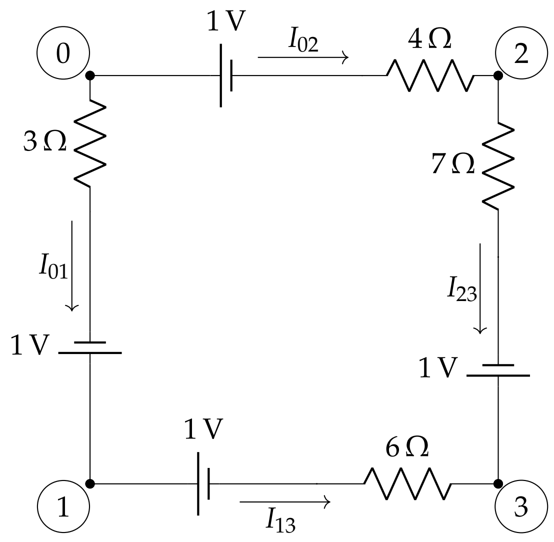


Figure 5.5: Fundamental cycle 2

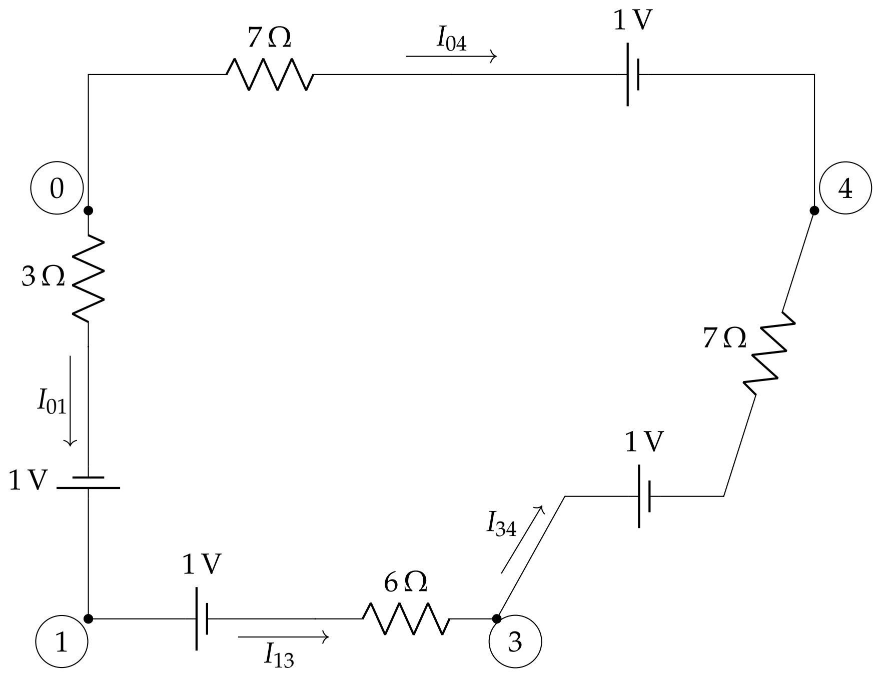


Figure 5.6: Fundamental cycle 3

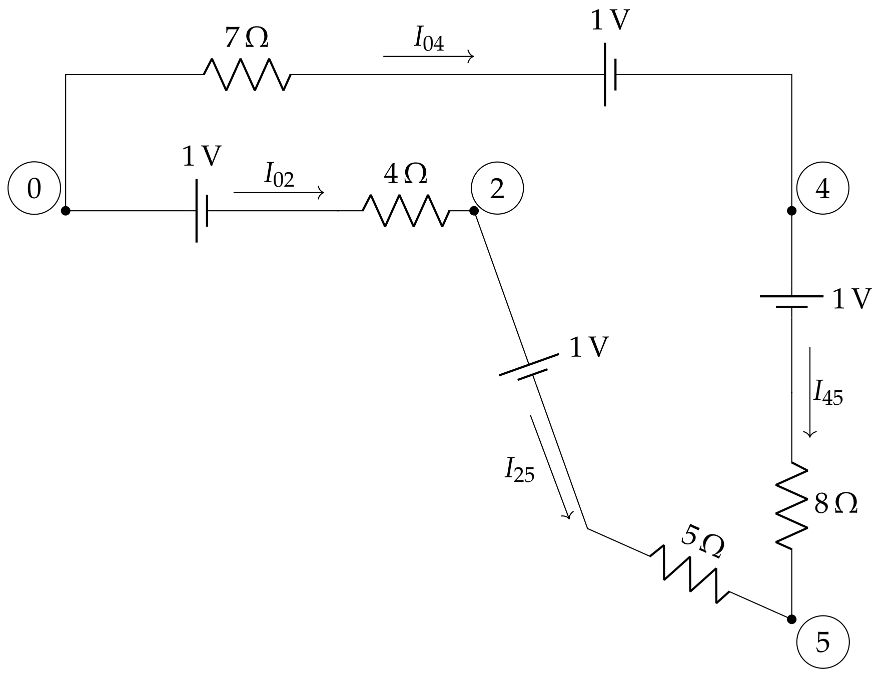


Figure 5.7: Fundamental cycle 4

5.1.2 Step 2

The second step will be writing the nodal equations. A nodal equation can be written for each node using the Kirchhoff first law(1.2) but since the nodal equations are not linearly independent we will write just $v_n - 1$ nodal equations to ensure the linear independency.

Since there are 8 nodes in the electrical graph given there will be 7 nodal equations, which are the following:

$$N_0 : -I_{01} - I_{02} - I_{04} = 0 \tag{5.1}$$
$$N_1 : I_{01} - I_{13} = 0 \tag{5.2}$$
$$N_2 : I_{02} - I_{23} - I_{24} - I_{25} = 0 \tag{5.3}$$
$$N_3 : I_{13} + I_{23} - I_{34} = 0 \tag{5.4}$$
$$N_4 : I_{04} + I_{24} + I_{34} - I_{45} = 0 \tag{5.5}$$
$$N_5 : I_{25} + I_{45} - I_{56} = 0 \tag{5.6}$$
$$N_6 : I_{56} - I_{67} = 0 \tag{5.7}$$

5.1.3 Step 3

The third step will be writing the mesh equations. A mesh equation can be written for each fundamental cycle using the Kirchhoff second law(1.3). Therefore there will be $e_n - v_n + 1$ mesh equations.

Since there are 4 fundamental cycles (5.4, 5.5, 5.6, 5.7) in the electrical graph given there will be 4 mesh equations, which are the following:

$$C_1 : R_{24}I_{24} - R_{04}I_{04} + R_{02}I_{02} = \mathcal{E}_{24} - \mathcal{E}_{04} + \mathcal{E}_{02} \tag{5.8}$$
$$C_2 : R_{23}I_{23} - R_{13}I_{13} - R_{01}I_{01} + R_{02}I_{02} = \mathcal{E}_{23} - \mathcal{E}_{13} - \mathcal{E}_{01} + \mathcal{E}_{02} \tag{5.9}$$
$$C_3 : -R_{34}I_{34} - R_{13}I_{13} - R_{01}I_{01} + R_{04}I_{04} = -\mathcal{E}_{34} - \mathcal{E}_{13} - \mathcal{E}_{01} + \mathcal{E}_{04} \tag{5.10}$$
$$C_4 : R_{45}I_{45} - R_{25}I_{25} - R_{02}I_{02} + R_{04}I_{04} = \mathcal{E}_{45} - \mathcal{E}_{25} - \mathcal{E}_{02} + \mathcal{E}_{04} \tag{5.11}$$

Specifically:

$$C_1 : 7I_{24} - 7I_{04} + 4I_{02} = 1 \tag{5.12}$$
$$C_2 : 7I_{23} - 6I_{13} - 3I_{01} + 4I_{02} = 0 \tag{5.13}$$
$$C_3 : -7I_{34} - 6I_{13} - 3I_{01} + 7I_{04} = -2 \tag{5.14}$$
$$C_4 : 8I_{45} - 5I_{25} - 4I_{02} + 7I_{04} = 0 \tag{5.15}$$

A system of linear equations can be represented in matrix form, joining the nodal equations with the mesh equations. The corresponding matrix we want to solve is:

$$\begin{pmatrix} -1 & -1 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\ 1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & -1 & 0 & -1 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 \\ 0 & 4 & 0 & 0 & 0 & 7 & 0 & -7 & 0 & 0 & 0 \\ -3 & 4 & 0 & -6 & 7 & 0 & 0 & 0 & 0 & 0 & 0 \\ -3 & 0 & 0 & -6 & 0 & 0 & -7 & 7 & 0 & 0 & 0 \\ 0 & -4 & -5 & 0 & 0 & 0 & 0 & 7 & 8 & 0 & 0 \end{pmatrix} \begin{pmatrix} I_{01} \\ I_{02} \\ I_{25} \\ I_{13} \\ I_{23} \\ I_{24} \\ I_{34} \\ I_{04} \\ I_{45} \\ I_{56} \\ I_{67} \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ -2 \\ 0 \end{pmatrix}$$

5.1.4 Step 4

The fourth and final step will be applying the Gauss elimination method to the following 11x12 augmented matrix:

$$\left(\begin{array}{ccccccccccc|c} -1 & -1 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & -1 & 0 & -1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & -1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 \\ 0 & 4 & 0 & 0 & 0 & 7 & 0 & -7 & 0 & 0 & 0 & 1 \\ -3 & 4 & 0 & -6 & 7 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -3 & 0 & 0 & -6 & 0 & 0 & -7 & 7 & 0 & 0 & 0 & -2 \\ 0 & -4 & -5 & 0 & 0 & 0 & 0 & 7 & 8 & 0 & 0 & 0 \end{array}\right)$$

Once we have applied the Gauss elimination method to the matrix, the resulting intensities will be the following:

$$I_{01} = \frac{229}{3849} = 0.059496 \qquad I_{24} = \frac{257}{7786} = 0.033008$$

$$I_{02} = \frac{173}{5399} = 0.032043 \qquad I_{34} = \frac{207}{1759} = 0.117681$$

$$I_{25} = -\frac{683}{11547} = -0.0591496 \qquad I_{04} = -\frac{185}{2021} = -0.0915389$$

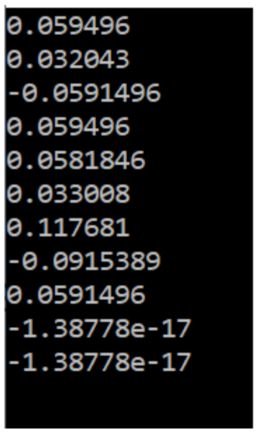
$$I_{13} = \frac{229}{3849} = 0.059496 \qquad I_{45} = \frac{683}{11547} = 0.0591496$$

$$I_{56} = 0$$

$$I_{23} = \frac{541}{9298} = 0.0581846 \qquad I_{67} = 0$$

Note that if an intensity has a negative value it means it travels oppositely, thus if we have $I_{ab} < 0$ this means the intensity travels from the node b to the node a.

The following picture 5.8 is a screenshot of the results given when the program is executed:



```
0.059496
0.032043
-0.0591496
0.059496
0.0581846
0.033008
0.117681
-0.0915389
0.0591496
-1.38778e-17
-1.38778e-17
```

Figure 5.8: Program output

Therefore the results given by the program match with the results found whilst resolving the previous example by hand.

Chapter 6

Conclusions

The aim of this paper is to set out and solve problems in electrical circuits and do a simulation where it can clearly be seen how an electrical graph and its corresponding meshed tree look like.

The program works properly no matter the size of the graph, as it has been tested for various type of graphs, and the intensities of each wire of an electrical circuit given are computed in a very efficient way. However, the success of this program clearly lays in the way the data is given. An electrical circuit is modeled into a graph so basically this whole work depends on the way we model the electrical circuit so we can apply the graph hierarchical algorithms to it.

As it has been seen in Chapter 3 the *BFS_Meshed_Trees* function is entirely based on converting a graph into a meshed tree. The fundamental cycles found via the meshed tree are crucial to find the meshed linear equations corresponding to the electrical graph and without those it cannot be solved. This means a graph is needed to find the intensities of an electrical circuit applying the program described in this paper.

Finally, this program can be improved in many ways. For example, the positions for each node of the electrical circuit that are given in the lecture file "example.dat" could be given in a 3-dimensional space, then the program could be modified so the visualisation of its corresponding electrical graph and meshed tree is in 3D. Another way to improve it could be that the information given in the lecture file "example.dat" would only be the number of nodes and edges, the position of the nodes and the information regarding edges and which vertices they join. Then the resistance and the voltage of each edge could be filled using the simulation, this would allow more interaction with the program. Lastly, there could be a modification allowing to remove or add an edge, this would allow to visualise how it would modify the intensities corresponding to other edges.

Bibliography

[1] ENCYCLOPÆDIA BRITANNICA website. Retrieved June 19 from
https://www.britannica.com/technology/electric-circuit

[2] ENCYCLOPÆDIA BRITANNICA website. Retrieved June 19 from
https://www.britannica.com/biography/Georg-Ohm

[3] ENCYCLOPÆDIA BRITANNICA website. Retrieved June 19 from
https://www.britannica.com/biography/Gustav-Robert-Kirchhoff

[4] Antoni Benseny, *Càlculs en xarxes elèctriques*, Materials Matemàtics, 2007 (6), pp. 29, (2007).

[5] Charles K. Alexander and Matthew N.O. Sadiku, *Fundamentals of Electric Circuits*, New York: McGraw-Hill, (2013).

[6] F. Javier Soria de Diego, *Graphs*, Department of Applied Mathematics and Analysis University of Barcelona, (2015).

[7] OpenGl. Wikipedia. Retrieved June 17 from
https://en.wikipedia.org/wiki/OpenGL

[8] The Cherno. Welcome to OpenGl. YouTube. Retrieved June 17 from
https://www.youtube.com/watch?v=W3gAzLwfIP0&list=PLlrATfBNZ98foTJPJ_Ev03o2oq3-GGOS2

[9] The OpenGL R Graphics System: A Specification (Version 3.0 - September 23, 2008) Mark Segal and Kurt Akeley. Retrieved June 17 from
https://www.khronos.org/registry/OpenGL/specs/gl/glspec30.pdf

[10] OpenGL website. Retrieved June 17 from
https://www.opengl.org/resources/libraries/

[11] FLTK website. Retrieved June 17 from
`https://www.fltk.org/doc-1.4/intro.html`

[12] FLTK website. Retrieved June 17 from
`https://www.fltk.org/doc-1.3/classFl_Button.html`