Modernizing Legacy Software (Concluder Web Application)

Hashem Alsaggaf

April 14, 2018

Contents

1 Introduction

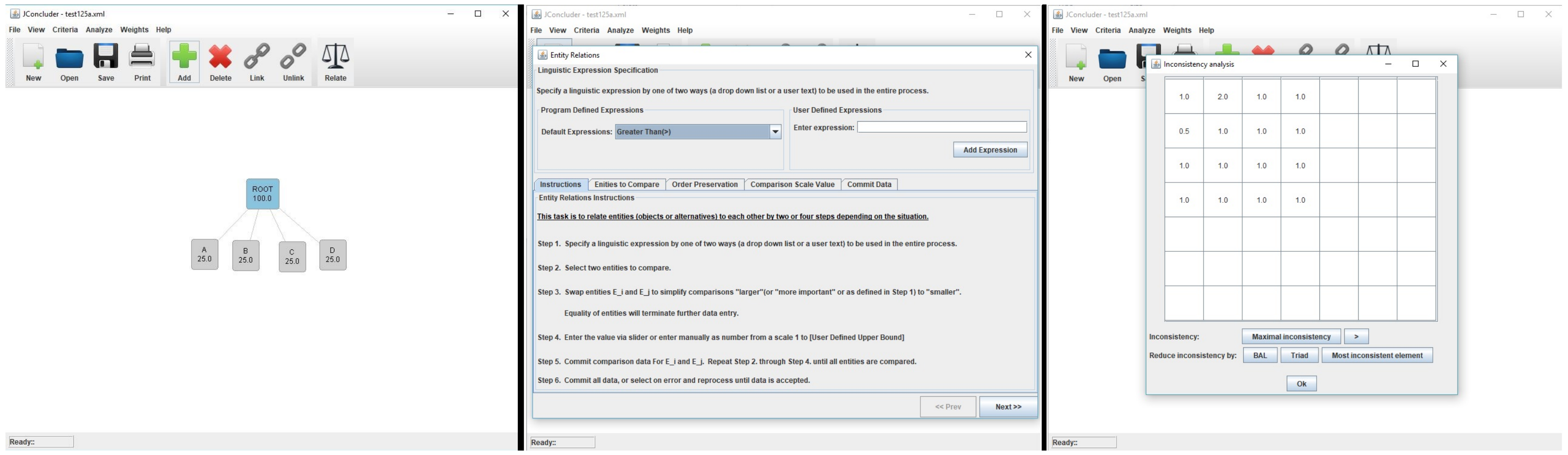
The purpose of this paper is to document the process of developing the first version of a web application.

Subjects mentioned or discussed and attachments in this paper:

- Pairwise Comparison Method
- Normalization
- Matrix calculations
- Tree structure
- html codes
- javascript codes
- jQuery codes
- JSON Objects
- Design sketches
- Developing method
- Implementation highlights
- Logic Diagram
- Screenshots
- Current Issues
- Room for enhancement

1.1 Project Background

In 2016, a team of developers and researchers contributed to create *Concluder*. An open source java software, developed to calculate inconsistency in order to improve decision making and knowledge management by employing Pairwise Comparison method for weight distribution. Screenshots of the software can be seen in Figure 1.


JConcluder - test125a.xml
File View Criteria Analyze Weights Help
New Open Save Print Add Delete Link Unlink Relate
ROOT 100.0
A 25.0
B 25.0
C 25.0
D 25.0
Ready::
Entity Relations
Inconsistency analysis
New Open
Ready::

Linguistic Expression Specification

Specify a linguistic expression by one of two ways (a drop down list or a user text) to be used in the entire process.

Program Defined Expressions

Default Expressions: Greater Than(>)

User Defined Expressions

Enter expression:

Add Expression

Instructions | Enities to Compare | Order Preservation | Comparison Scale Value | Commit Data

Entity Relations Instructions

This task is to relate entities (objects or alternatives) to each other by two or four steps depending on the situation.

Step 1. Specify a linguistic expression by one of two ways (a drop down list or a user text) to be used in the entire process.

Step 2. Select two entities to compare.

Step 3. Swap entities E_i and E_j to simplify comparisons "larger"(or "more important" or as defined in Step 1) to "smaller".

Equality of entities will terminate further data entry.

Step 4. Enter the value via slider or enter manually as number from a scale 1 to [User Defined Upper Bound]

Step 5. Commit comparison data For E_i and E_j. Repeat Step 2. through Step 4. until all entities are compared.

Step 6. Commit all data, or select on error and reprocess until data is accepted.

<< Prev | Next >>

1.0	2.0	1.0	1.0			
0.5	1.0	1.0	1.0			
1.0	1.0	1.0	1.0			
1.0	1.0	1.0	1.0			

Inconsistency: Maximal inconsistency >

Reduce inconsistency by: BAL Triad Most inconsistent element

Ok

Figure 1: Screenshots of the java version of Concluder

1.2 Goal

The goal of this project is to design and implement a web application version of Concluder. This version should be accessible from any device with modern internet browser.

2 Developing Method

The developing method of this project consists of five phases:

- System requirements
- UI/UX design
- Implementation
- testing
- publishing

3 System requirements

In this phase, a list of Concluder web application features will be identified. To understand the system requirements, we looked at the java version of Concluder, then highlighted most of its features, keeping in mind that more features will be added as the development cycle iterate. Some of the features are:

- Elements representation
- Editing elements
- Elements relations
- Calculating and reducing Inconsistency

3.1 Elements representation

One of the most important requirement is a way to represent elements. In the java version of Concluder, elements are represented as a tree (see Figure 1). The user would likely expect to see elements represented as a tree in the web application as well. Therefore, elements should be represented as a tree in the web app version of Concluder.

3.2 Editing elements

Another required feature is to be able to manipulate the elements. That includes adding new elements to the tree, editing existing nodes in the tree, and deleting nodes from the tree.

3.3 Elements relations

Another main feature is to be able to relate elements by weights. Elements relation need to be represented as weights in this version as well.

3.4 Calculating and reducing Inconsistency

The Pairwise Comparison Method calculates inconsistency using matrices. The minimum required calculations are:

- Presenting Pairwise Comparison (PC) matrix.
- Weight distribution.
- Relation inconsistency.
- Reducing inconsistency.
- Normalization.

4 UI/UX Design

In this phase, a number of sketches for the user interface and experience is produced to guide the implementation phase. After that, the user interface will be designed with Webflow.io to produce clean HTML and CSS files to be used as a wire-frame for the application.

4.1 webflow.io

Webflow.io is a cloud-based software as a service (SaaS). It is used to visually make responsive web page designs.
The use of such a tool is necessary to be able to design and edit the user interface rapidly and in a fast way. One main advantage of using webflow.io is the ability to export the design to HTML/CSS/JS code. The exported code is simple, organized, and very easy to read and understand. Once the code is exported, it is treated as a wire-frame for the UI.

4.2 Design details

To accommodate an organized user interface, the design of different windows (called screens) is proposed. The main screens are Edit Screen, Relation Screen, and Analyzing Screen. As seen in Figure 2 and in figure 6, in each screen, there is different control buttons and/or panels. To navigate between screens, a slide panel on the left side of the application is designed (called Main Panel), look at Figure 3.
For user feedback, a pop-up window, with a theme that matches the overall theme is designed, see Figure 4. This window will provide help and tips to the user when needed.

Figure 2: Edit Tree Sketch

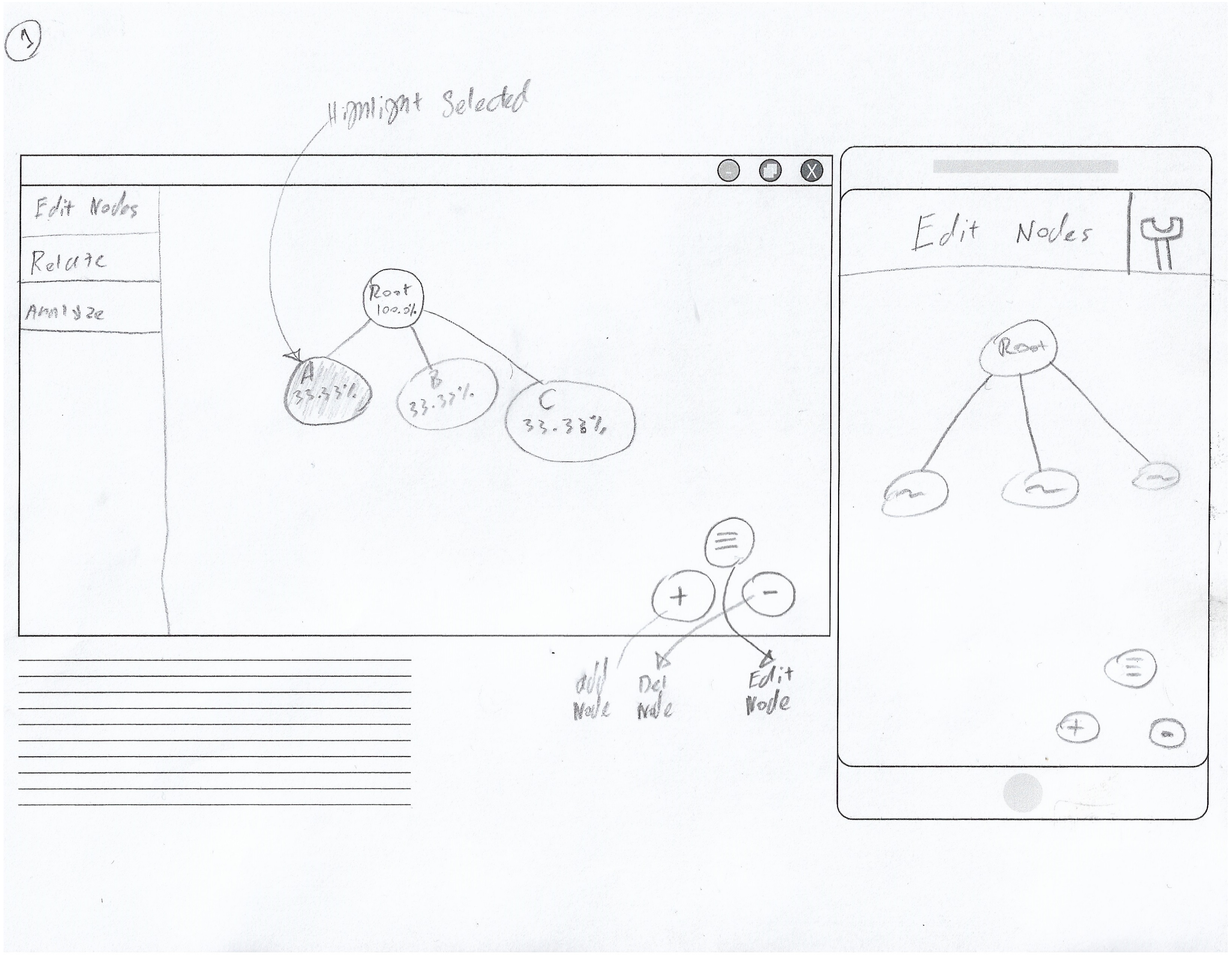


Figure 3: Tree Editing screenshot

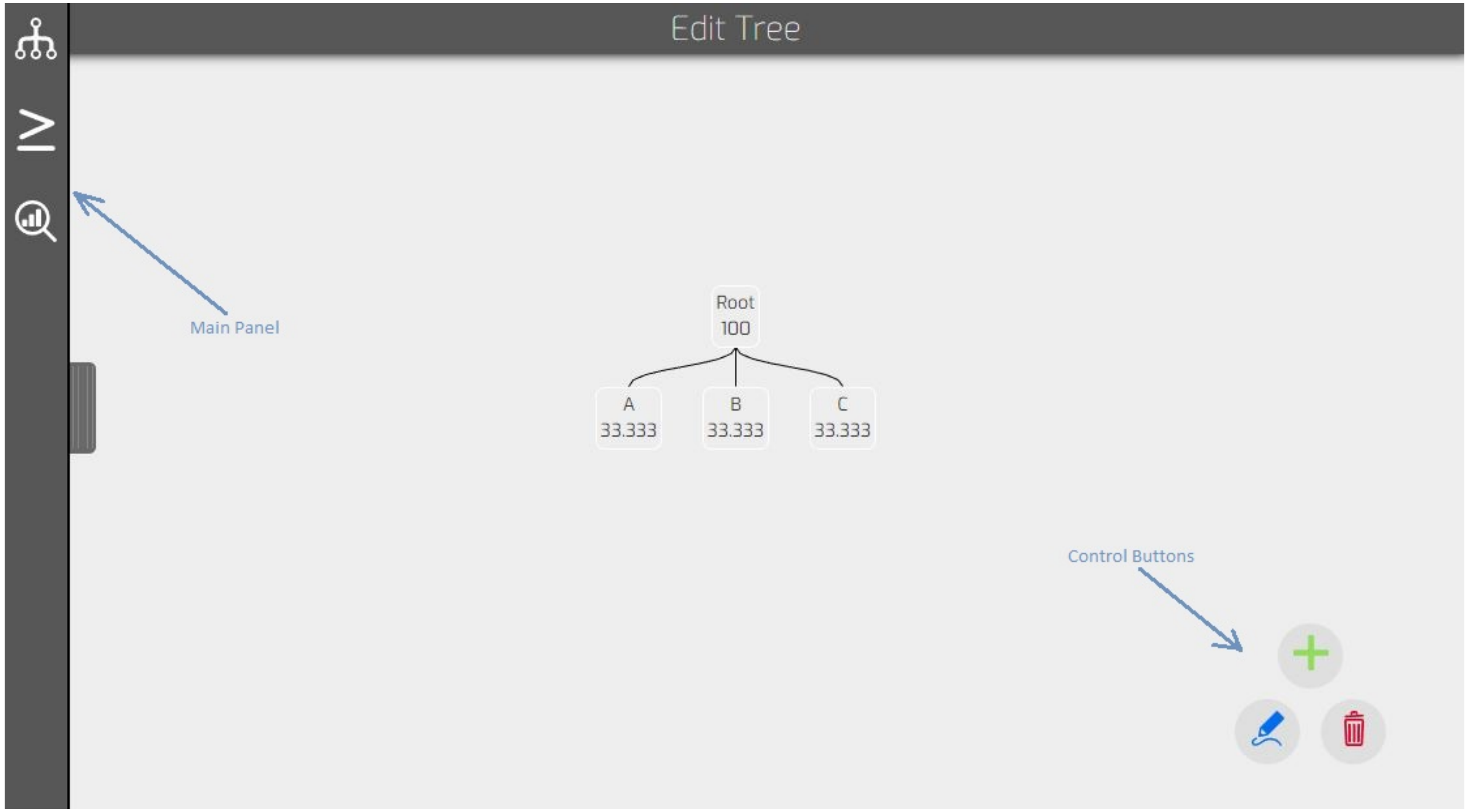
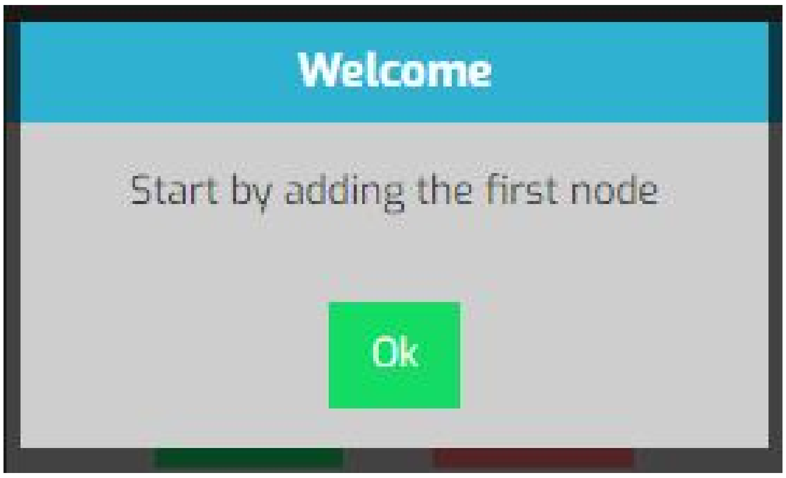


Figure 4: Messaging Example


Welcome

Start by adding the first node

4.2.1 Edit Screen

This is where the user interact with the tree structure in terms of adding, editing, or deleting nodes from the tree. As the system requires the manipulating of a tree structured data, we need to design a layout where the user can view the tree and manipulate it at the same time. At the bottom right of this screen, the user can see three buttons with declarative icons; add, delete and edit. See Figure 3. Before any action is committed, the user confirmation is needed to avoid accidental actions. See Figure 5 for an example of the user flow in terms of editing the tree.
The same flow principle applies on deleting and editing nodes.

4.2.2 Relation Screen

A sketch for this screen is presented in Figure 6. In this screen, the user is able to see the tree and select two nodes to set the relation between them, see Figure 7. When two nodes are selected, the Relation Wizard will show the two selected nodes, and will ask the user to choose which node is bigger and by how much, see Figure 8. After applying the relation between two nodes, the relation will be saved in the Nodes Relations tab. To access the Nodes Relations tab, the user would click on the Relations Tab Button at the bottom right-corner of the screen, see Figure 7. In the Nodes Relations tab, the user have the ability to Edit or Delete a relation, see Figure 9. See the diagram in Figure 10 for the user flow in Relation Screen.

Figure 5: Add a node flow diagram

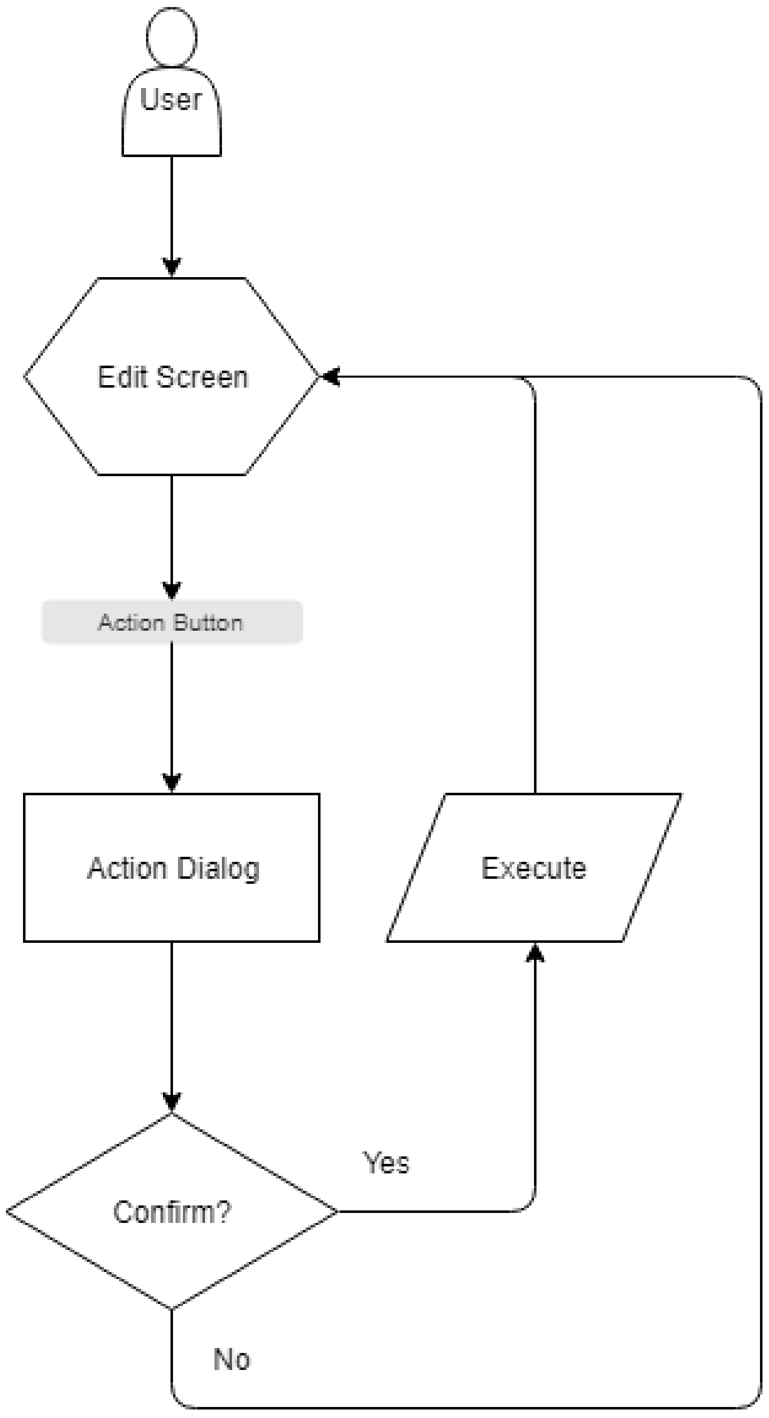
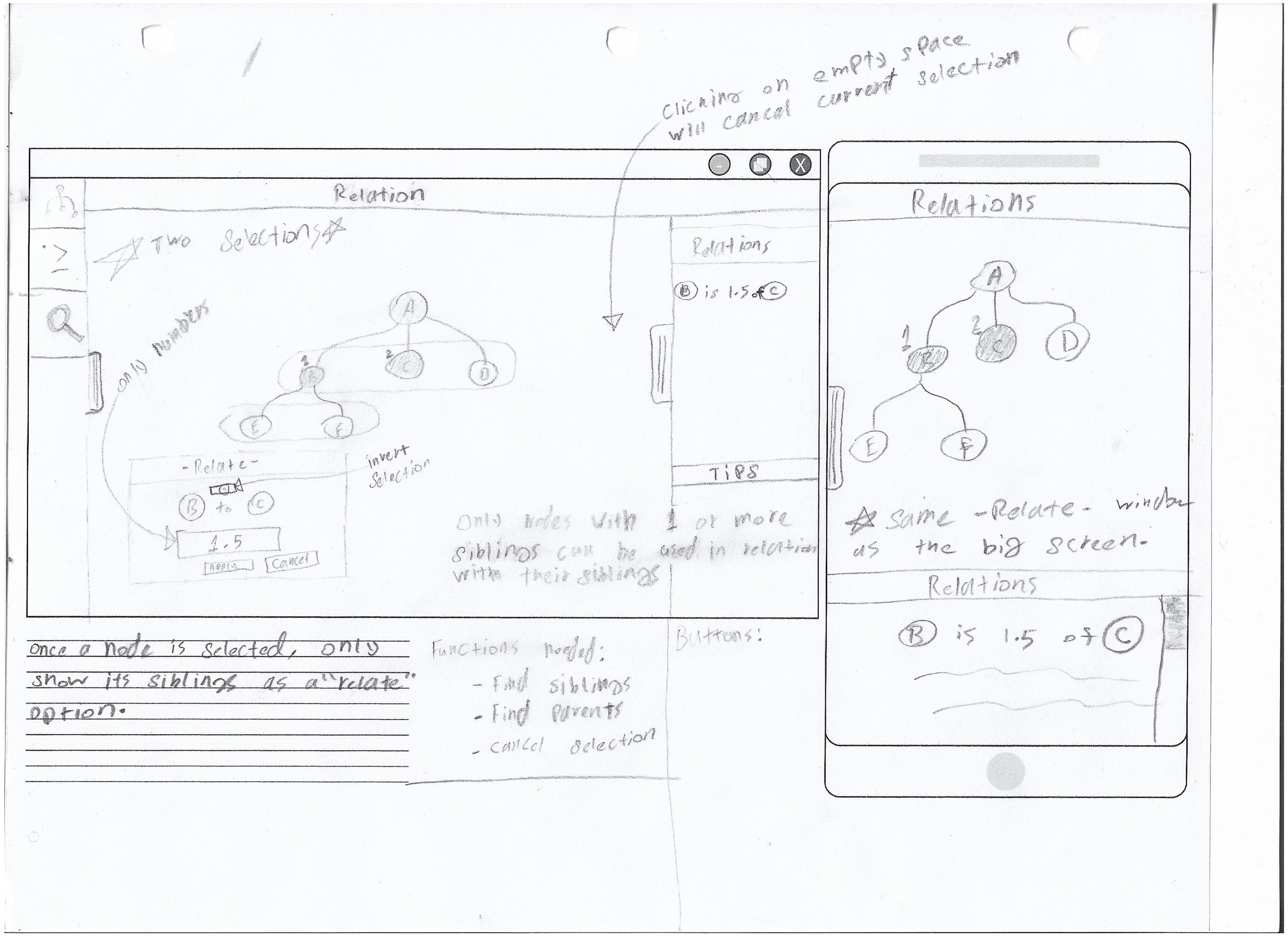


Figure 6: Relations Sketch



Clicking on empty space will cancel current selection
Relation
Two Selections
only numbers
A
B
C
D
E
F
-Relate-
B to C
1.5
Apply
Cancel
invert Selection
Relations
B is 1.5 of C
TIPS
Only nodes with 1 or more siblings can be used in relation with their siblings
Once a node is selected, only show its siblings as a "relate" option.
Functions needed:
- Find siblings
- Find parents
- cancel selection
Buttons:
Relations
same -Relate- window as the big screen.
Relations
B is 1.5 of C

Figure 7: Relation Screen screenshot

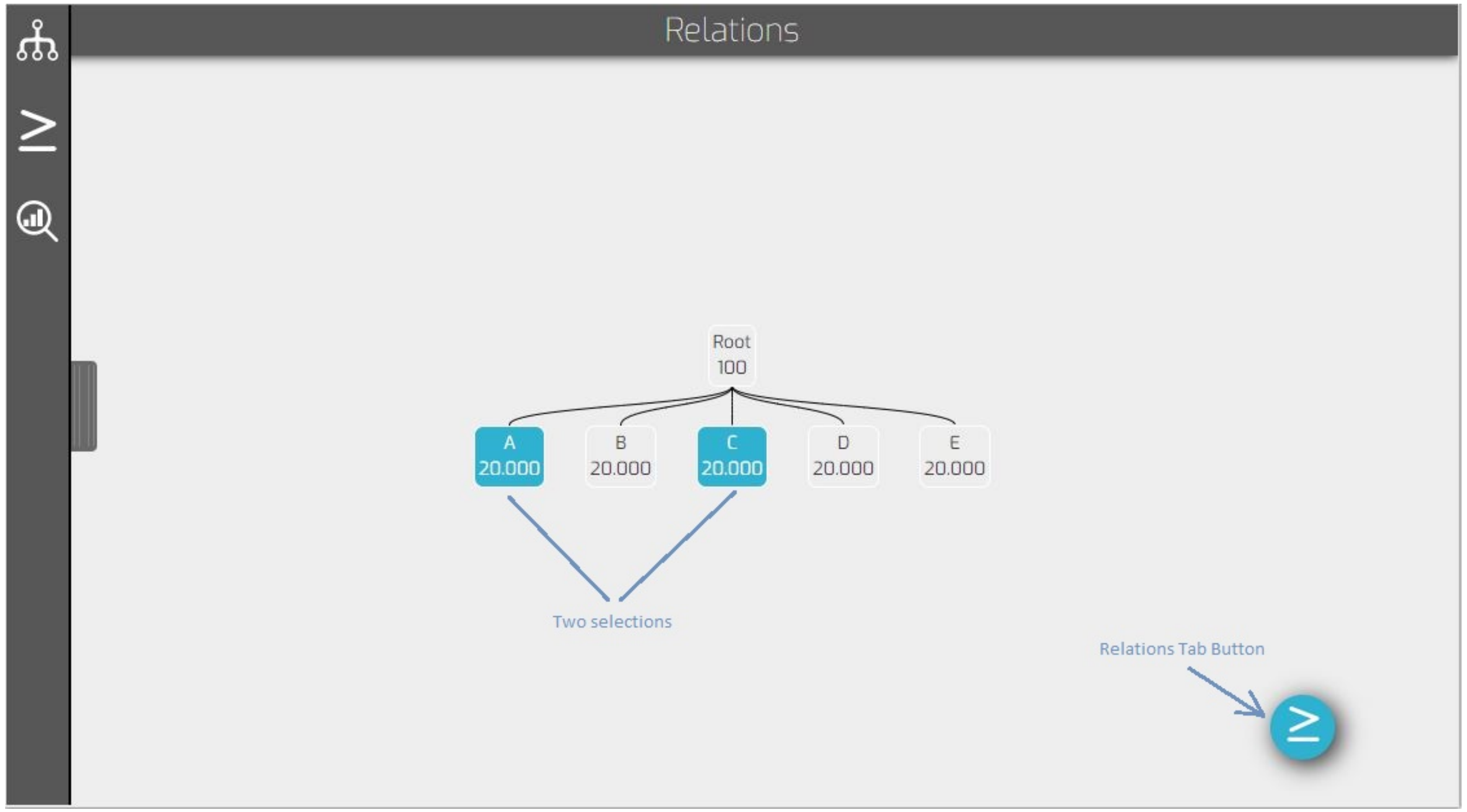
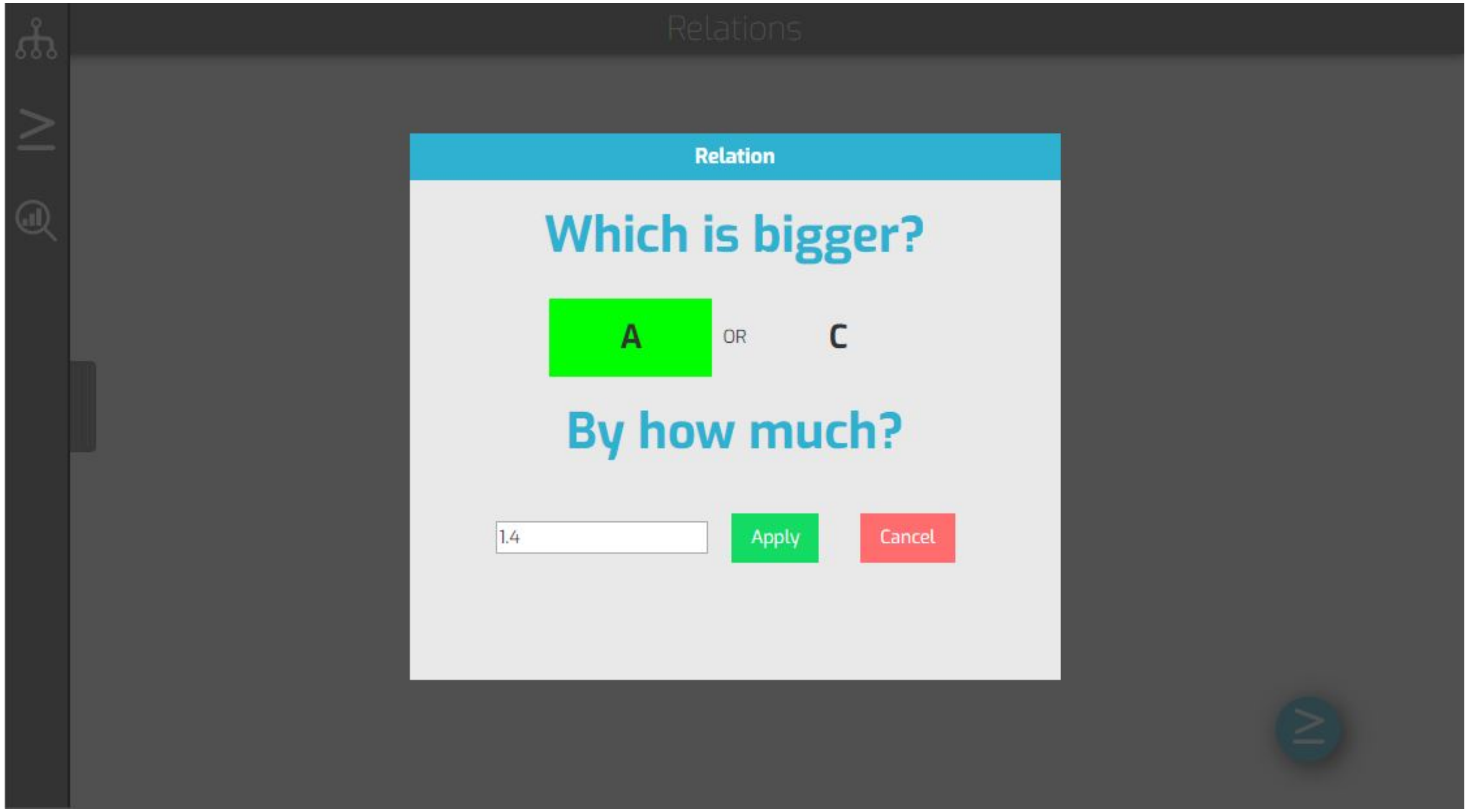


Figure 8: Relation Wizard


Relations

Relation

Which is bigger?

A OR C

By how much?

1.4

Figure 9: Relation Tab

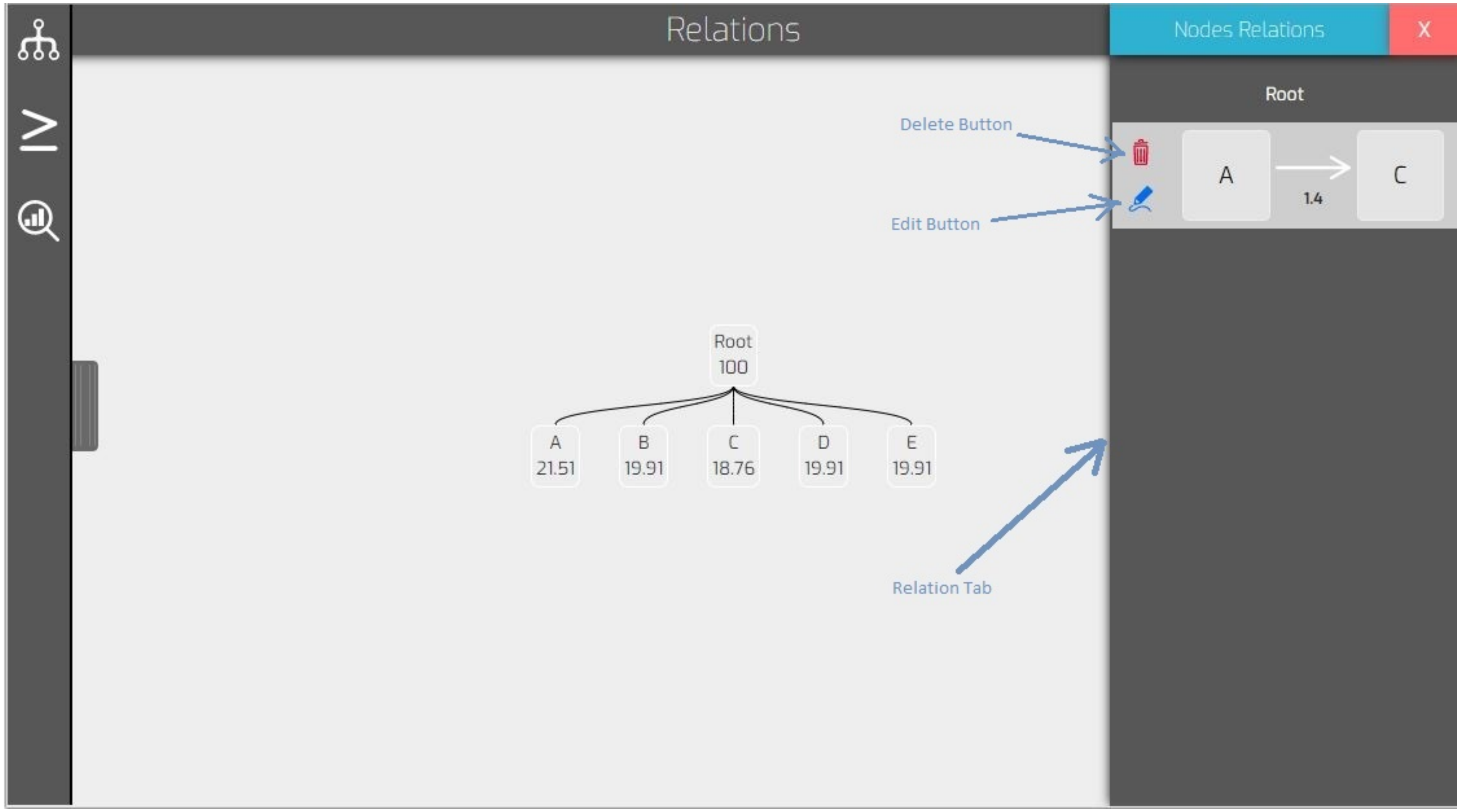
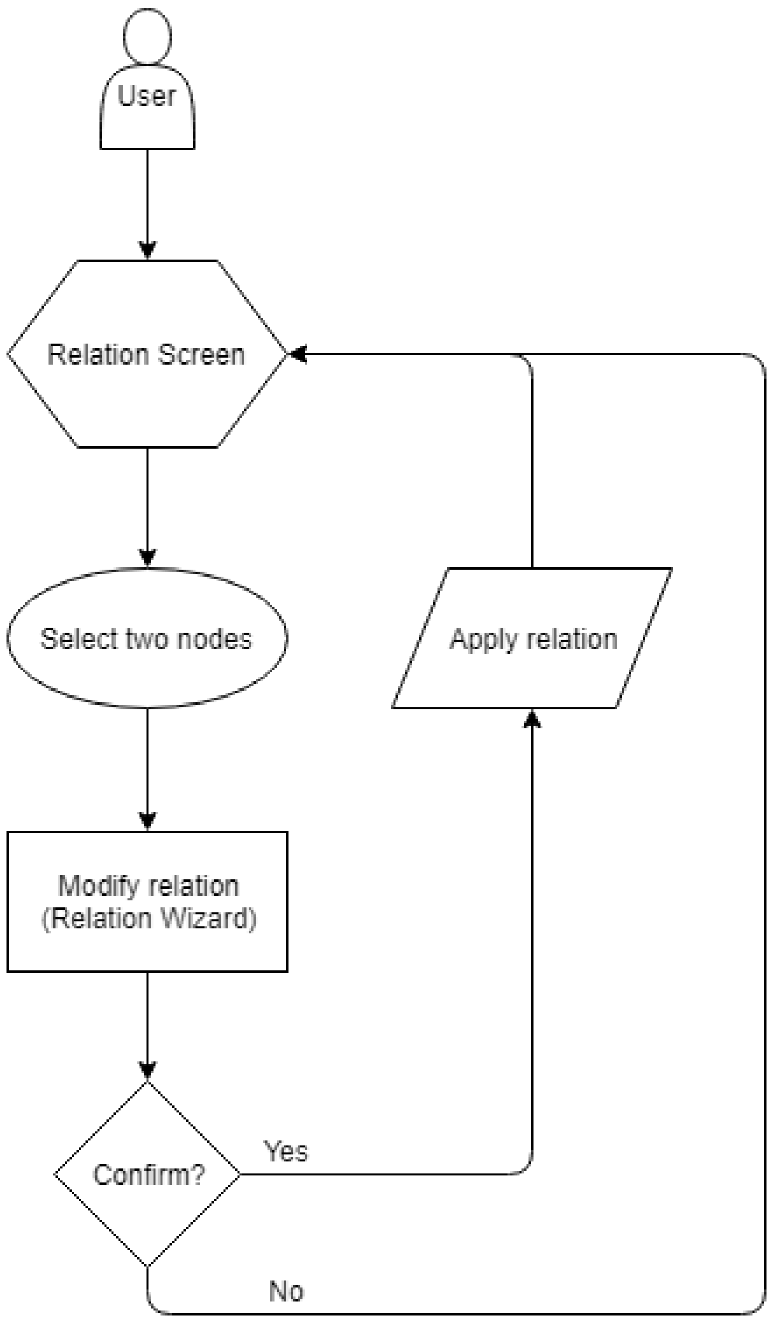


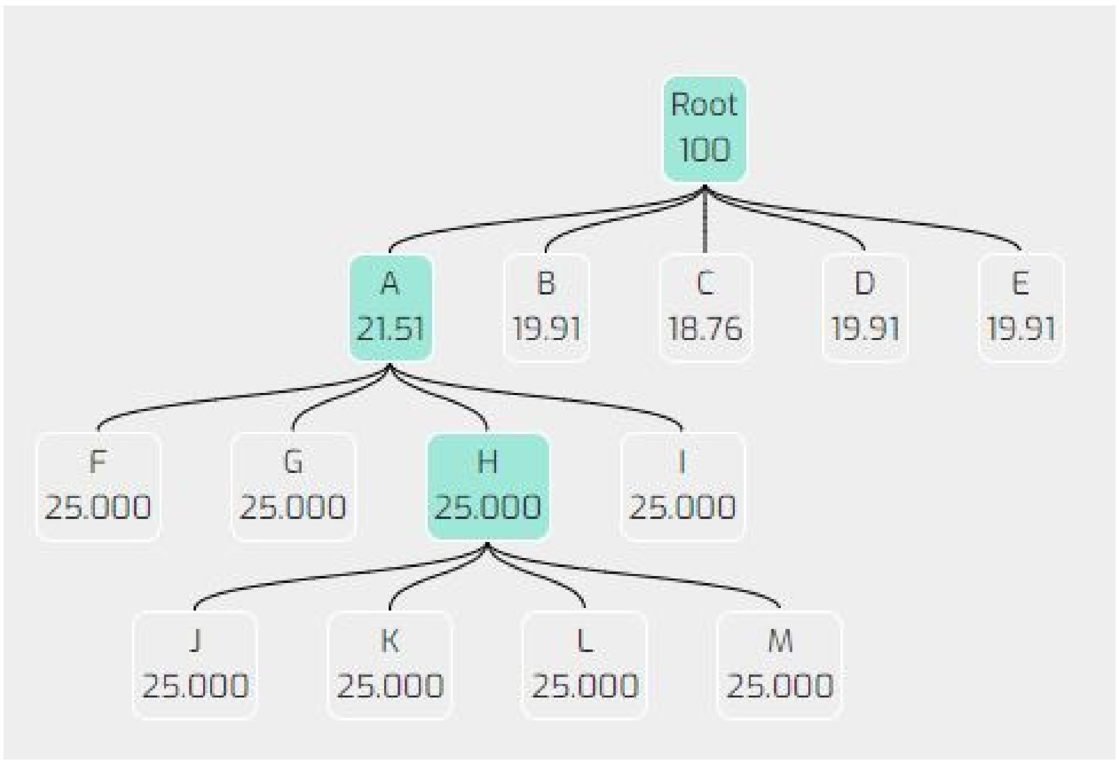
Figure 10: Relation (User Flow)



4.2.3 Analyzing Screen

When the user navigate to this screen, the user will see the tree where all nodes with three or more children are highlighted, see Figure 11. When the user click on a highlighted node, the Analyze Wizard will show and present an editable relation matrix (Editable Pairwise Comparison Matrix) for that group, see Figure 12.

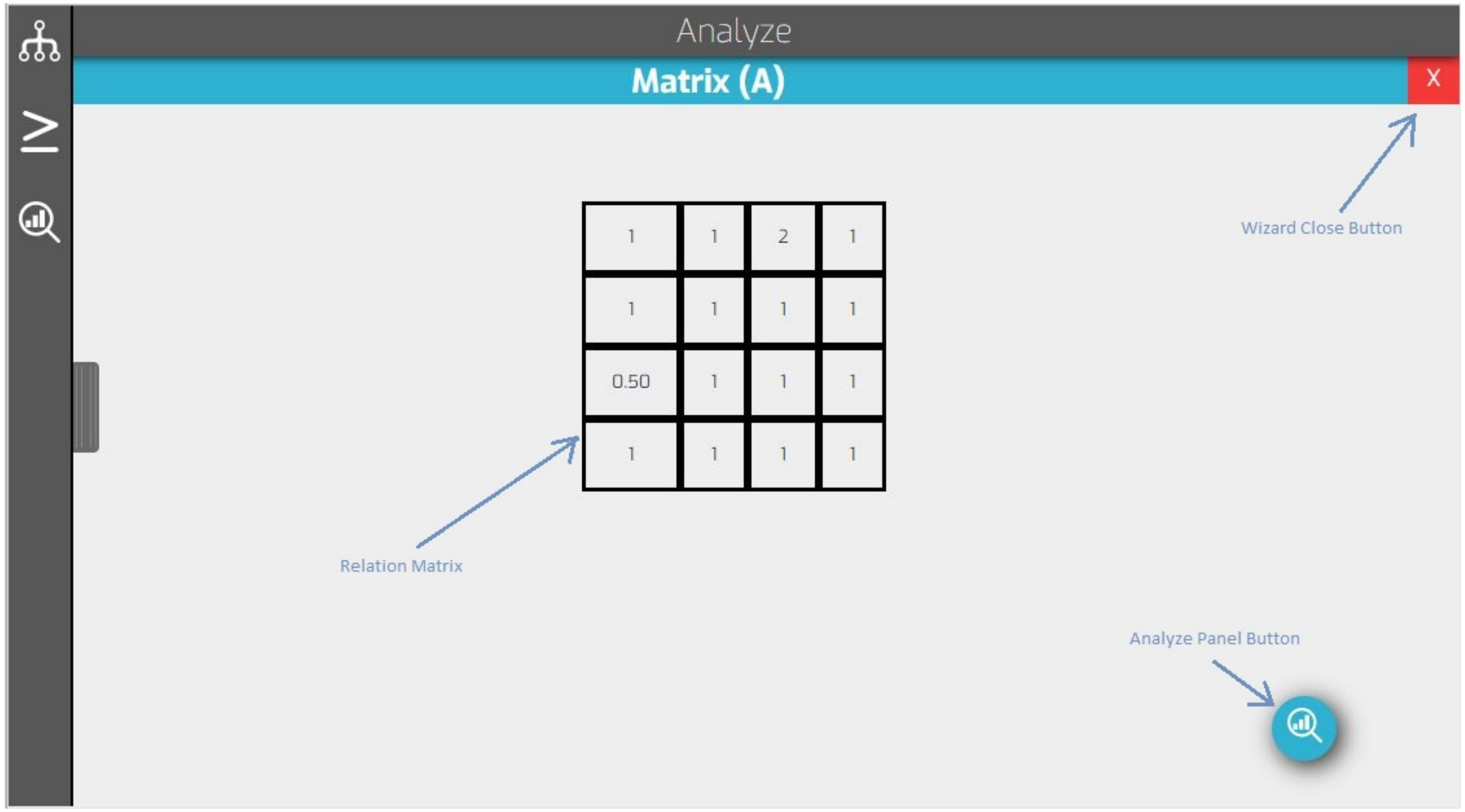
Figure 11: Nodes with children



In the Analyze Wizard, the user is able to analyze the matrix by using the Analyze Panel. The Analyze Panel is accessible by clicking on the Analyze Panel button at the bottom-right corner of the Analyze Wizard, see Figure 12.
See diagram in Figure 13 to get an overview of the user flow in the Analyze Screen.
The analyze panel has two sections; General and Traids. See Figure 14.

- General section has general analyzing functions, like showing the most inconsistent elements and reducing inconsistency button.
- Traid section deals with traids analyzing operations, like showing all traids, reducing a traid, and showing the most inconsistent traid.

Figure 12: Analyze Wizard



Matrix (A)

1	1	2	1
1	1	1	1
0.50	1	1	1
1	1	1	1

Figure 13: Analyze Screen User Flow

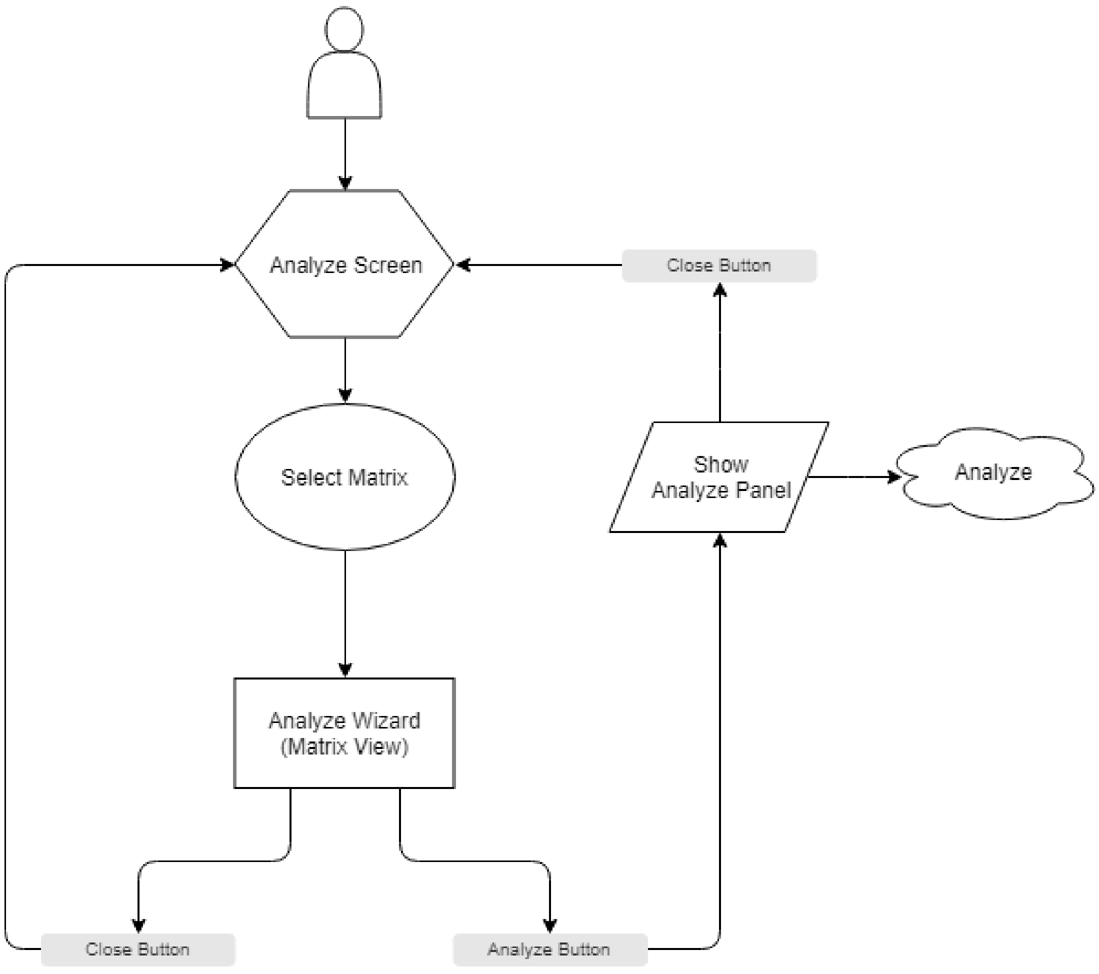
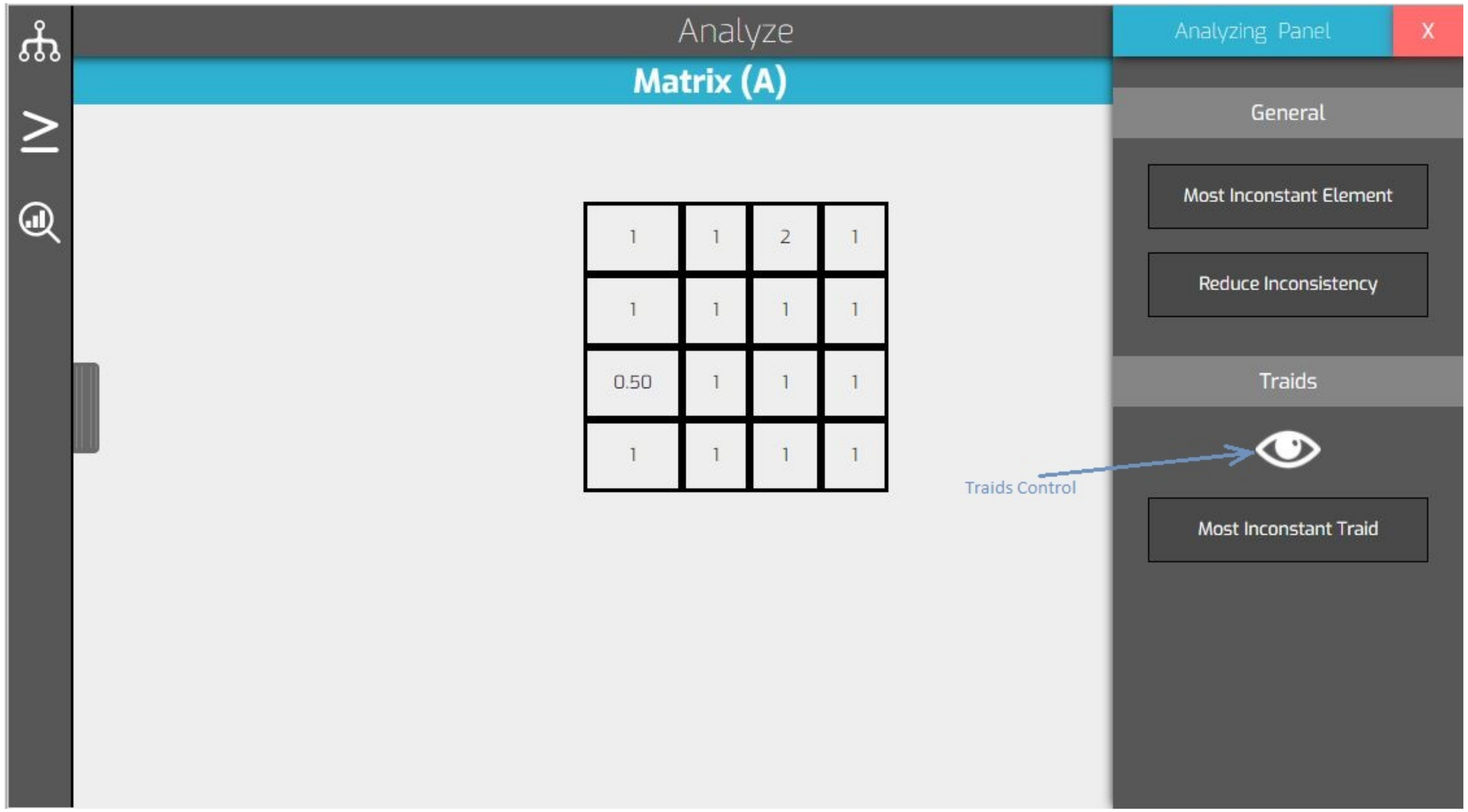


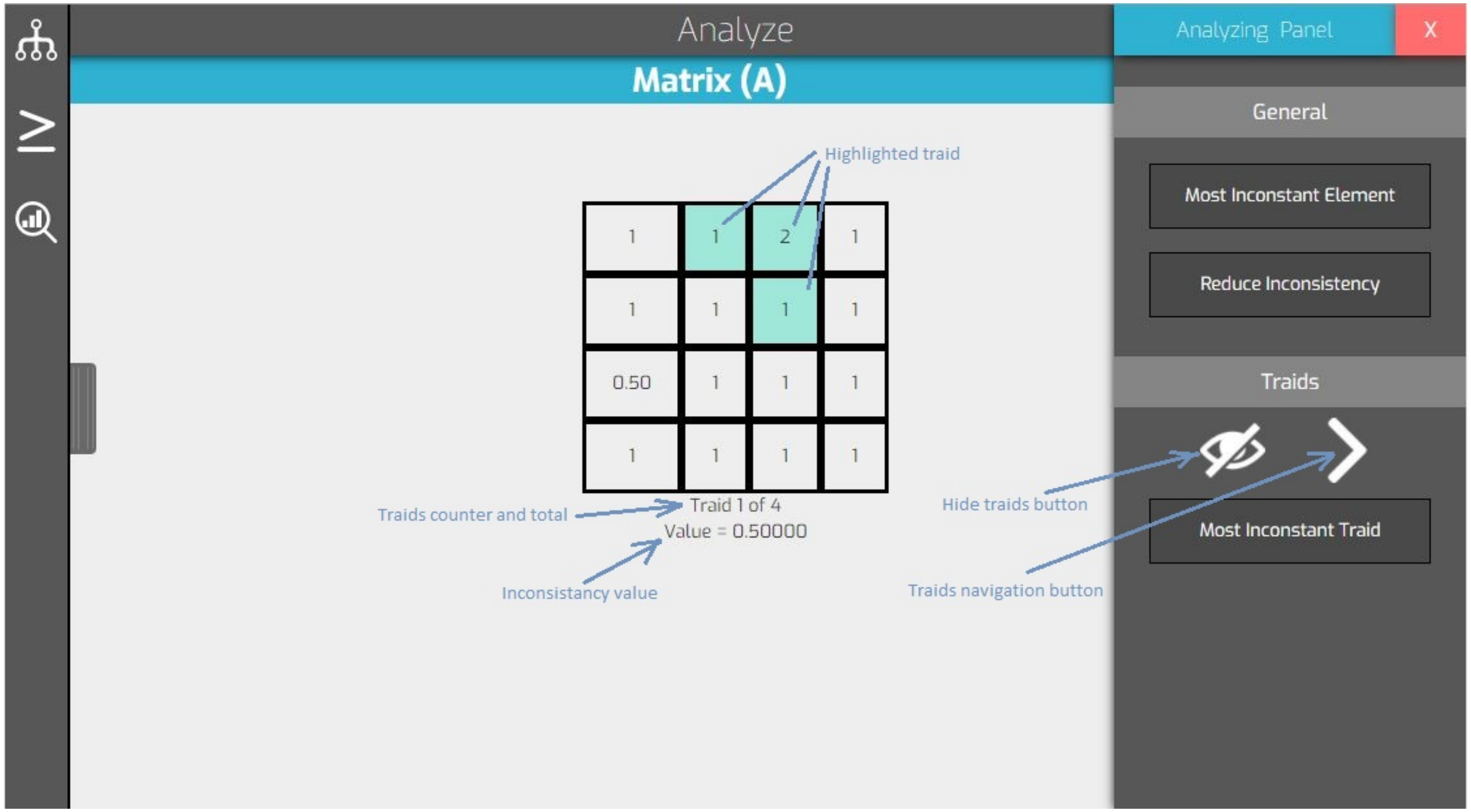
Figure 14: Analyze Panel



1	1	2	1
1	1	1	1
0.50	1	1	1
1	1	1	1

When clicking on the Traid Control button, the first traid will be highlighted in the matrix, the traids counter and total number of traids will be presented, the inconsistency value will also be presented, and the traid navigation button and the Hide traid button will show as well. See Figure 15.

Figure 15: Analyze Panel - Traid Control



1	1	2	1
1	1	1	1
0.50	1	1	1
1	1	1	1

Traid 1 of 4
Value = 0.50000

5 Implementation

In this phase, The UI is taken from sketches and designed in webflow.io. after the UI is designed, a clean html and css code is produced and used as a wire-frame for the project.
After producing the html and css codes, some html elements are given an ID to be accessible by javascript. The ID assignments is critical to be able to do things like showing different screens and wizards, assign actions to buttons, etc...
For this project, all screens are hidden by default (the css display property is set to 'none'). To show a screen, the screen and the button related to that screen is given an html id. Then an action to show the screen is programmed to be triggered when the button is clicked. Lets consider the Relation Screen for example. Take a look at the html code for the Relation Screen and its button, See Code 1.

Code 1: Relation Button html code

```
<div id="relationIcon" class="menu-btn">
    <img src="images/relation.svg" class="menu-icon" />
</div>
...
...
...
<div id="relationScreen" class="relations-screen">
     ...
     ...
     ...
</div>
```

Notice **id="relationIcon"** and **id="relationScreen"**. Also, take a look at the javascript (js) code (Code 2)[1] to see how we are showing the Relation Screen when the Relation icon/button is clicked.

Code 2: Relation Button html code

```
var relationIcon = $('#relationIcon');
var relationScreen = $('#relationScreen');

relationIcon.click(function(){
    relationScreen.show();
}
```

This technique is implemented for most buttons in this project.

Sometimes we have to hide and show screens based on events too. For example, we have to hide the Analyzing Screen when the user navigates away from it.

[1]jQuery library is used here.

To ensure easier readability of coeds and easier maintenance, the code for each feature is placed in a separate file. We have the following files:

- nodes.js - deals with trees and nodes operations
- matrix.js - deals with matrix operations
- relation.js - deals with relation's operation
- analyze.js - deals with analyzing tasks
- control.js - deals with UI buttons and control
- index.js - deals with screen navigation, shortcuts, etc...

The js files are listed in 'index.html' as a script resources, in a sequence where the bottom js files have access to all variables and function in the js files above it. However, those are not the only js files in the application, we are also using jQuery and Treant.js libraries.

5.1 Elements Representation (tree structure)

Elements representation is one of the most important aspect of this system. Elements are represented as a tree structure, we used Treant.js[2]. With Treant.js, we need an HTML div with ID (where the tree will be represented) and a JSON object (the tree we want to represent). The JSON object should have the following structure.

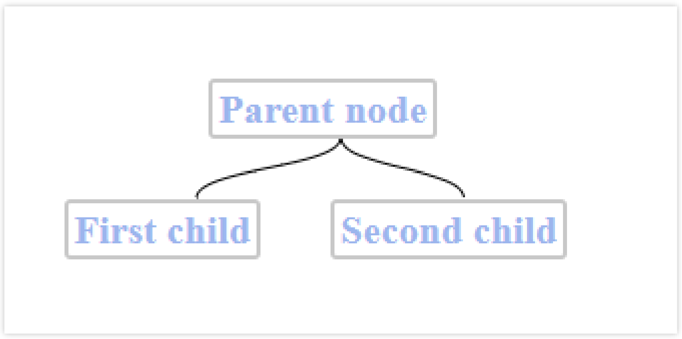
Code 3: Tree Example

```
Tree = {
    chart: {
        container: "#tree-simple"
    },

    nodeStructure: {
        text: { name: "Parent node" },
        children: [
            {
                text: { name: "First child" }
            },
            {
                text: { name: "Second child" }
            }
        ]
    }
};
```

The above JSON tree structure (Code 3) will produce the following chart (Figure 16). Note: each node in the tree chart is an HTML div element, this is important for tree manipulation.

[2]Treant.js is a javascript library that create tree structure charts from JSON objects.

Figure 16: Simple tree output



5.1.1 Tree Configuration

The two main attribute for a tree object (as described in treant-js library website) are:

- chart (JSON pair values)
- nodeStructure (JSON pair values)

In **chart** attribute, we set the chart configuration. For the purpose of this software, we made use of the following attributes for chart configuration:

- container
- node
- callback

To access the container attribute, we use **Tree.chart.container**. In the **container** attribute, we declare the html div ID where the tree will be drawn. Example: To change the div where the tree is drawn (Code 4):

Code 4: Tree container example

```
Tree.chart.container = "#treeDiv";
```

In the above example (Code 4), the tree will be drawn in an html div with the ID ‘treeDiv’ (notice the number sign before the ID)

In the **node** attribute, we made use of the key ‘HTMLclass’.

- HTMLclass is used to apply a CSS class to every node in the tree (remember, all nodes are html elements)

Example:

Code 5: Tree.node Example

```
node: {
    HTMLclass: "nodeClick",
},
```

In the above example (Code 5), a class called ‘nodeClick‘ is applied to every node.
In **callback** attribute, we can set javascript functions and events. All events are listed in the Treant-js websites. We made use of the event ‘onTreeLoaded‘ to apply a function that is triggered when a node is clicked (Code 6).

Code 6: Node click trigger function

```
callback: {
    onTreeLoaded :   function(){
        $('.nodeClick').click(function(){
            ...
        });
    }
},
```

A complete chart configuration would look like this (Code 7)

Code 7: Chart configuration

```
treeConfig = {
    chart: {
        container: "#treeDiv",
        node: {
            HTMLclass: "nodeClick",
        },
        callback: {
            onTreeLoaded : function(){
                $('.nodeClick').click(function(){
                    ...
                    ...
                    ...
                });
            }
        }
    }
}
```

A complete code for tree configuration is shown in Code 8.

Code 8: Complete Tree Configuration

```
var treeConfig = {
        chart: {
        container: "",
        node: {
            HTMLclass: "nodeClick"
        },
        callback: {
            onTreeLoaded : function(){
                $('.nodeClick').click(function(){
                    clearSelection();
                    $(this).css({'background-color': '#2fb1d1', 'color': 'white'})
                        ↪ ;
                    selectedNode = $(this).attr('id');
                    findNode(selectedNode);// for debugging
                });
            }
        }
    },

    nodeStructure: {
        HTMLid: "node_root",
        text: { name: "Root",  weight: 100},
        parent: "",
        children: [

        ],
        matrix: [

        ]
    }
};
```

5.1.2 Tree Structure

In the **'nodeStructure'**, we define the structure of the tree, in terms of children and descriptive text. A very simple structure could be like the one shown in (Example - Code 4). Every node has parents excepts the root node. All children of a node are listed in the children attribute (as an array), and a descriptive text is set in the text attribute. For this project, some extra attributes has been introduced; **HTMLid, parent, and matrix**, see Code 9.

Code 9: nodeStructure code

```
nodeStructure: {
    HTMLid: "node_root",
    text: { name: "Root",  weight: 100},
    parent: "",
    children: [

    ],
    matrix: []
}
```

5.2 Editing elements - Tree operations

First thing we need to do is to draw the tree using treant.js library functions. The basic code to draw the tree is shown in Code 11. To keep the tree configuration intact and to make sure we can always reset a tree configuration, we clone the tree configuration ('treeConfig') as our tree (see Code 10). Note that the variable 'treeConfig' is already defined as a tree configuration with chart settings and nodes structure (See Code 7).

'$.extend' is a jQuery function that is used to merge tow objects together. By merging any object with an empty object, we get a cloned object. The overhead of cloning is necessary to keep our tree settings intact, in case we need to reset the tree.

Code 10: $.extend syntax

```
var Tree = $.extend(true, {}, treeConfig);
```

Code 11: Basic Tree Drawing

```
chart = new Treant(Tree, function() { }, $);
```

However, the tree will need to be drawn in multiple locations across the software. Therefore, a function called '**drawTree**' is implemented, see Code 12.

Code 12: drawTree Function

```
function drawTree(target ,tree){
    tree.chart.container = target;
    chart = new Treant(tree, function() { }, $);
}
```

drawTree function takes tow arguments; **target** and **tree**. **target** is the div id where the tree will be drawn, and **tree** is the tree we want to draw.
Now we can use this function to draw trees in different screens.

Second, we need a mechanism to select a node in the tree, meaning we need to find a way for the software to know what node the user is selecting.
The steps for selecting a node are:

- A mapping table is created to store references of each node
- When a node is created, an id will be assigned for it
- A reference to the new node is stored in the mapping table
- A variable called '**selectedNode**' is created to store the id of a selected node
- The user clicks on a node
- The software then look for that node in the mapping table by its id
- The node id will be stored in '**selectedNode**'

- As a feedback, the background of the selected node is changed.

These steps involves many aspects. We already implemented the click trigger for all nodes (see Code 6), now we need to define the actions for the trigger. However, we have different screens, and each screen uses the click trigger for different actions. For example, in the Edit Screen, we just select the node to specify where the new node will be listed or which node to delete, but in the Relation Screen, we can select two nodes to relate between them. To accommodate different use for the same trigger, we can access the callback attribute and change the action. Refer to 5.1.1 for tree configuration.

When the Edit Screen is visible, the callback attribute holds the code shown in Code 13.

Code 13: callback code for Edit Screen

```
callback: {
    onTreeLoaded : function(){
        $('.nodeClick').click(function(){
            clearSelection();
            $(this).css({'background-color': 'gray', 'color': 'white'});
            selectedNode = $(this).attr('id');
            findNode(selectedNode);
        });
    }
}
```

The function **clearSelection()** will simply reset the background of all nodes in the tree. See Code 14.

Code 14: clearSelection Function

```

function clearSelection(){
    $('.node').each(function(){
        $(this).css({'background-color': 'gray', 'color': 'black'});
    });
}
```

We might need to access the root of our tree in some situations. Therefore, we need to assign a root variable as shown in Code 15.

Code 15: Root assignment

```
var root = Tree.nodeStructure;
```

Editing elements now becomes a matter of accessing and manipulating the tree structure. To access any node in the tree, we can use the mapping table. The mapping table, is initialized with the root reference (see Code 16), and every time a node is added, a refrence to that node is added in the mapping table as well.

Code 16: Mapping the tree root

```
var map = {
    'node_root': root
};
```

We have three main operations for manipulating the tree;

- Add Node
- Edit Node
- Delete Node

5.2.1 Adding nodes

To add a new node, we first need to have an id counter to make sure all ids are unique. Then we need to specify where to add the new node, meaning under which node will the new node be listed. That can be done by clicking on the node, which will store the selected node in '**selectedNode**'. The function to add a node is shown in Code 17

Code 17: addNode Function

```
function addNode(target, name){

    if(target == null || name == ""){
        return;
    }

    target.children.push({
        HTMLid: newId,
        text: {name: name, weight: 100},
        parent: target.HTMLid,
        children: [],
        matrix: []
    });
    map[newId] = target.children[target.children.length - 1];
    distributeWeight(target, 100)
    drawTree("#editTreeChart", Tree);
    nodeIdCounter++;
    newId = idPrefix + nodeIdCounter;
}
```

Notice **nodeIdCounter**, **idPrefix**, and **newId**; **nodeIdCounter** starts with one and increments every time we add a node, **idPrefix** is set as the string 'node_', so **newId** is always unique. addNode function takes two arguments; '**target**' and '**name**', **target** is the target node where the new node will be listed under (this is usually done by using the function **findNode**), and **name** is the descriptive text for the new node. Also notice the use of 'target.children.push'. The push function is a built-in function for javascript arrays.
After pushing the new node, the new weight is distributed using the function **distributeWeight**. Then we draw the tree to see the new structure.

5.2.2 Weight Distribution

Weight distribution is very straight forward when we are editing the tree. The function used for weight distribution is called '**distributeWeight**' and is shown in Code 18.

Code 18: distributeWeight Function

```
function distributeWeight(parent, weight){
    var factor = parent.children.length;
    parent.children.forEach(function (child){
        newWeight = weight/factor;
        child.text.weight = newWeight.toFixed(3);
    });
}
```

As can be seen in Code 18, **distributeWeight** function takes two arguments, **parent** and **weight**. **parent** is the parent node where the weight will be distributed among its children. **weight** is the value that need to be distributed (usually it is set to 100).
We store the weight factor as the children number. Then we iterate through each child and assign the weight for each node as weight/factor.

5.2.3 Deleting Nodes

Deleting nodes from tree structure is done by employing the built-in function in javascript **splice**. The function that is implemented to delete nodes is called **deleteNode**, see Code 19.

Code 19: deleteNode Function

```
function deleteNode(id){

    var parentId = map[id].parent;
    var parent = map[parentId];

    var i = 0;
    parent.children.forEach(function (child){
        if(child.HTMLid == id){
            parent.children.splice(i, 1);
        }else{
            i++;
        }
    });
}
```

The function **deleteNode** takes the id of the node we want to delete, finds its parent node, then loop through its children to find the index of the node we want to delete, then delete it by using the array function **splice()**.

5.2.4 Editing Nodes

Editing nodes is done using the implemented function **editNodeName**. See Code 20

Code 20: editNodeName Function

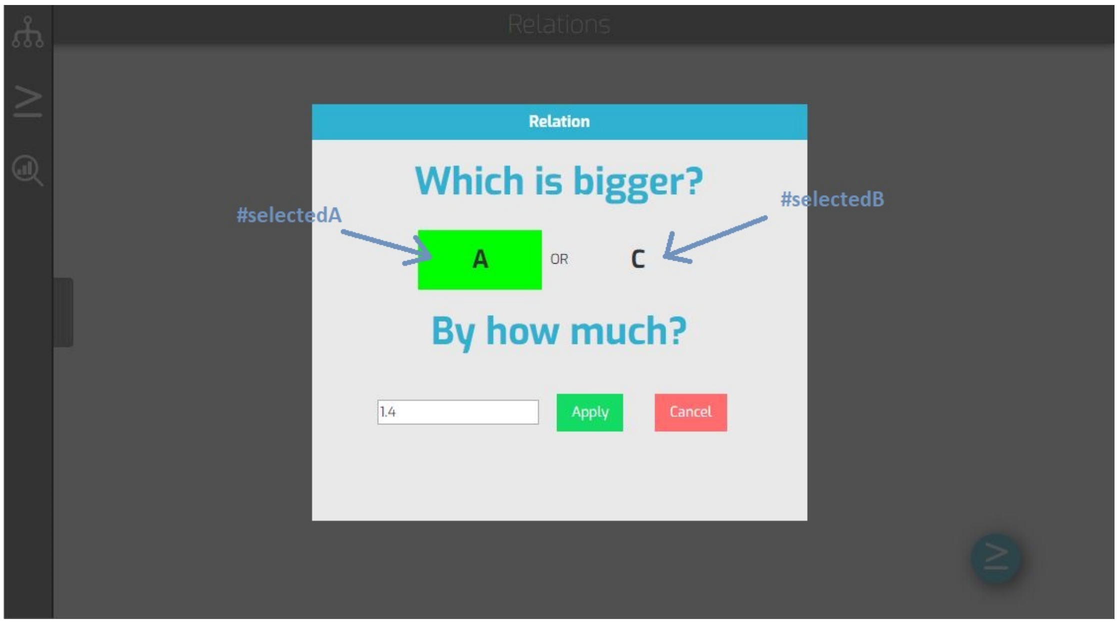
```
function editNodeName(id, name){
    map[id].text.name = name;
}
```

The function **editNodeName** takes two arguments, **id** and **name**. It will locate the target node by its id using the mapping table, then assign the new name.

5.3 Relations

As mentioned in Section 4.2.2, the user should be able to select two nodes in this screen. So the selection mechanism will have to change. To change the selection mechanism we have to access the chart configuration and change the action of clicking on a node.
To read about accessing the chart configuration, refer to section 5.1.1.
First, two variables are assigned for nodes selection; **nodeA** and **nodeB**, both are initialized as null. We already assigned the html elements ids where the selection will be shown for the user; **selectedA** and **selectedB**, see Code 21 and Figure 17.

Figure 17: Relation Wizard



Code 21: Node Selection Variables

```
var nodeA = null;
var nodeB = null;

var selectedA = $('#selectedA');
var selectedB = $('#selectedB');
```

Then we need to know the order of selection (first selection or second selection). To determine the order of selection, If statement is used. See Code 22.

Code 22: Detect the order of selection

```
        if(nodeA == null && nodeB == null){
            \\ this is the first selection
        }else if(nodeA != null && nodeB == null){
            \\ this is the second selection
        }else if(nodeA != null && nodeB != null){
            \\ the user already selected two nodes (a selection reset is required)
        }
```

Referring to Code 22:

- In the first selection, we simply find the selected node by id and store it in the variable **nodeA**, then change the selected node's background color for visual indication.
- In the second selection, we find the selected node and store it in **nodeB**, then change the background color for the second node. After that, we change the text in **selectedA** and **selectedB** accordingly, then show the relation wizard.
- In case two nodes are already selected, we need to reset the selections. To do so, we simply clear the selection variables, and re-draw the tree.

The selecting function for this screen is stored in a variable called '**relationSelectionFunction**' then it is assigned to the callback attribute in chart configuration. To read about accessing the chart configuration, refer to section 5.1.1. The complete code for Relation Screen selection function is shown in Code 23.

Code 23: Relation selection function

```
var relationSelectionFunction = function(){
    $('.nodeClick').click(function(){
        if(nodeA == null && nodeB == null){
            findNode($(this).attr('id'), root);
            nodeA = targetNode;
            $(this).css({'background-color': '#2fb1d1', 'color': 'white'});
        }else if(nodeA != null && nodeB == null){
            $(this).css({'background-color': '#2fb1d1', 'color': 'white'});
            findNode($(this).attr('id'), root);
            nodeB = targetNode;

            selectedA.text(nodeA.text.name);
            selectedB.text(nodeB.text.name);

            relationWizard.show(250);
        }else if(nodeA != null && nodeB != null){
            nodeA = null;
            nodeB = null;
            drawTree("#relateTreeChart", Tree);
        }
    });
}
```

Note: '**relateTreeChart**' is the id for the html div element where the tree will be drawn.

After the user selects two nodes, the relation wizard is shown, and the user is prompt to chose which node is bigger (in terms of weights). The user is also prompt to indicate 'by how much is one node bigger than the other'. For example, lets assume the user have two properties; property A and property C, and lets assume that property A is bigger than property C by forty percent (for example, if property C is 100 percent, then property A is 140 percent the size of property C). In this case we say that property A is 1.4 the size of property C. See Figure 17.
Once this relation is applied. It will be shown in the **Relation Tab**, and the user will have the ability to edit the relation or delete it, See Figure 9. For organization, all relations that are made under the same parents are grouped together.

5.4 Analyzing

In this screen, we want to highlight only the nodes that can be analyzed (nodes with more than two children), and the user should be able to click on the highlighted nodes only. Two features are implemented here, highlighting and selecting:

- The function to highlight nodes with more than two children is shown in Code 24.

  Code 24: Highlighting Matrix Nodes (nodes with more than 2 children)

  ```
  function highlightMatrixNodes(currentNode){
      if (countChildren(currentNode) > 2) {
          var node = $('#'+currentNode.HTMLid);
          node.css({'background-color': '#9EE7D9'});
      }
      currentNode.children.forEach(function (currentChild) {
          highlightMatrixNodes(currentChild);
      });

  }
  ```

  The function '**countChildren**' counts the number of children in a specific node. The code for the function **countChildren** is shown in Code 25.

  Code 25: countChildren function

  ```
  function countChildren(node){
      counter = 0;
      node.children.forEach(function(child){
          counter++;
      });
      return counter;
  }
  ```

  The function **highlightMatrixNodes** is recursive. It iterates through each node, and highlight it if it has more than two children. The highlighting technique in Code 24 works by first obtaining the html id of a node, then changing the background-color property of that node.

- The selecting mechanism is stored in a variable called '**analyzeSelectionFunction**'.

The complete code for the selecting mechanism for this screen is shown in Code 26.

Code 26: analyzeSelectionFunction function

```
var analyzeSelectionFunction = function(){
    $('.nodeClick').click(function(){
        findNode($(this).attr('id'), root);
        numberOfChildren = countChildren(targetNode);
        if(numberOfChildren > 2){
            matrixTitle.text("Matrix␣(" + targetNode.text.name+")");
            printMatrix(targetNode.matrix, '#matrixView');
            matrixStatistics.text('');
            removeHighlights(targetNode.matrix)
            matrixWizard.show();

            // a list of inconsistencies is created (inconsList)
            calcInconsistancy(targetNode.matrix);
        }
    });
}
```

As can be seen in Code 26, only nodes with more than two children are clickable. The **printMatrix** function will be explained later.

5.5 Matrix Operations

Many matrix operations are implemented, like printing and editing a matrix, deleting an element from a matrix, summing a matrix, and calculating elements weights from a matrix.
We know that any matrix is a 2D array. Take for example the 5-by-5 matrix shown in Code 27.

Code 27: Matrix Example

```

Matrix = [
             [A,B,C,D,E],
             [F,G,H,I,J],
             [K,L,M,N,O],
             [P,Q,R,S,T],
             [V,W,X,Y,Z]
        ]
```

In this system, every node has a matrix. Every time a child is added to a parent, that child is also added to the parent's matrix. See Code 28.

Code 28: Matrix construction

```
for(i = 0 ; i < target.children.length ; i++){
        target.matrix[i] = [];
        for(j = 0 ; j < target.children.length ; j++){
            target.matrix[i][j] = 1;
        }
}
```

The code in Code 28 runs every time a node is added under a parent. It is a nested loop that is used to create a matrix of ones with a length that is equal to the number of children. So, if a node has four children, it would have a 4-by-4 matrix. See example in Code 29.

Code 29: Matrix Example 2

```
map['node_root'].matrix = [
    [1,1,1,1],
    [1,1,1,1],
    [1,1,1,1],
    [1,1,1,1]
]
```

5.5.1 Printing a matrix

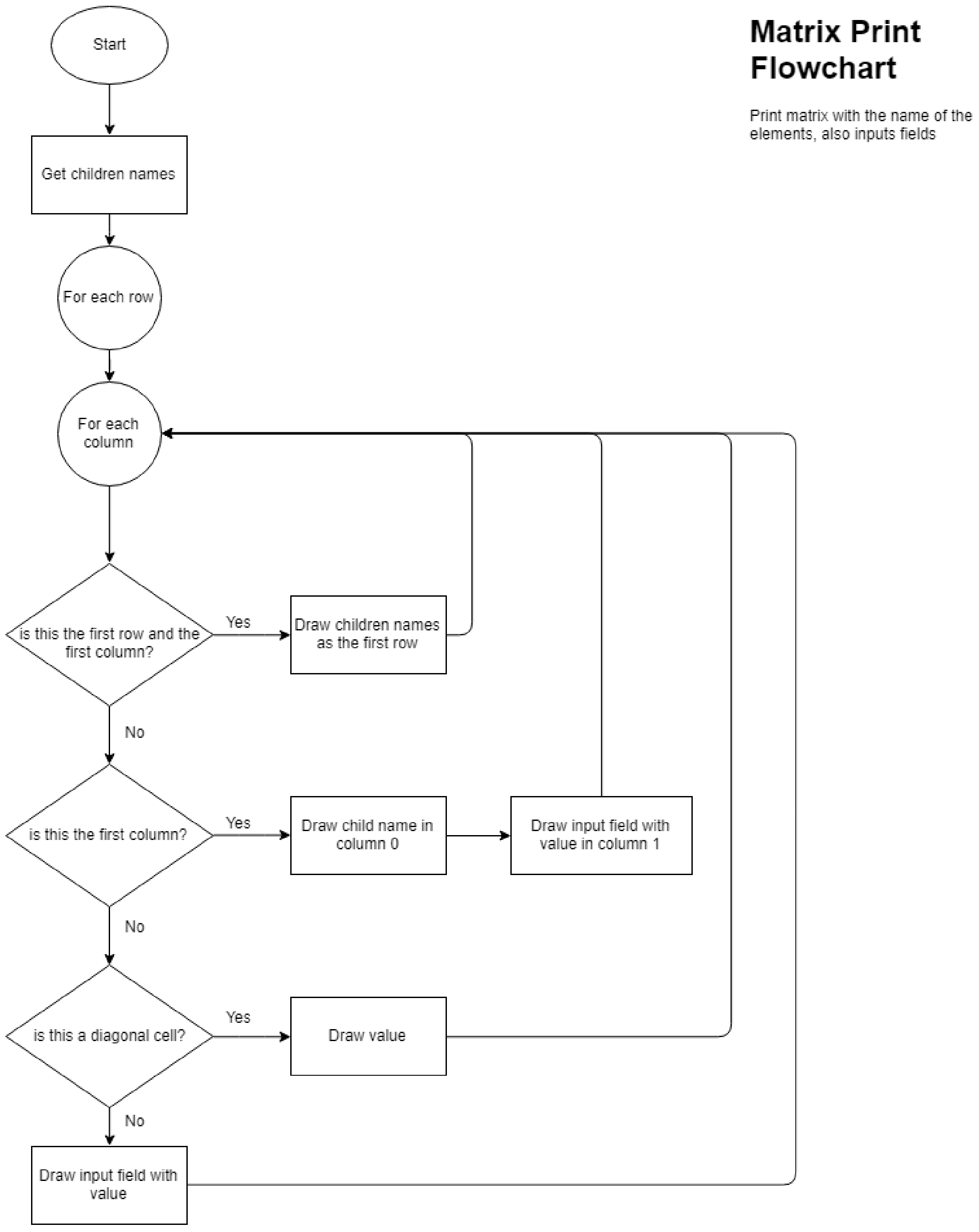
To represent a matrix as a table, we first need to know the length of its dimensions. For instance, a 4-by-4 matrix would have a length of four. If we look at a matrix as a table, its length would be the number of rows or columns, because every matrix has the same number of rows and columns. Then we can use a simple nested loop to draw the rows and columns of the matrix. Rows are arrays, and columns are elements. The outer loop should go through each row, and the inner loop should go through each column (element in an array). A simple nested loop to draw a matrix can look like the code shown in Code 30.

Code 30: Matrix nested loop

```
for(i = 0; i < matrix.length; i++){
    for(j = 0; j < matrix.length; j++){
        //draw matrix[i][j]
    }
    //new line
}
```

For this project however, we need more than just a simple draw. We need to draw elements names. We also need to draw the matrix as an html table. Plus we need to be able to access the values in the matrix as input fields. This adds a bit of complexity, but the concept of nested loop still holds. Take a look at the complexity of printing a matrix in this system in the flowchart in Figure 18. Also see the JS function **printMatrix** shown in Code 31.

Figure 18: Matrix Printing Flowchart


Matrix Print Flowchart
Print matrix with the name of the elements, also inputs fields
Start
Get children names
For each row
For each column
is this the first row and the first column?
Yes
Draw children names as the first row
No
is this the first column?
Yes
Draw child name in column 0
Draw input field with value in column 1
No
is this a diagonal cell?
Yes
Draw value
No
Draw input field with value

Code 31: printMatrix Function

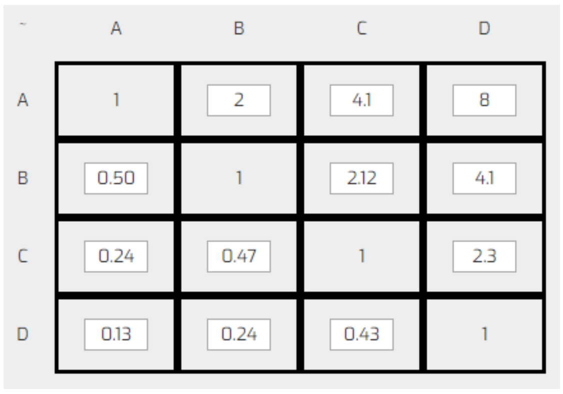
```
function printMatrix(node, targetDiv){
    var matrix = node.matrix;
    var children = node.children;
    var elementsNames = [];

    var table = "";

    //get elements names in an array
    children.forEach(function(child){
        elementsNames.push(child.text.name);
    });

    for(i = 0 ; i < matrix.length; i++){
        for(j = 0 ; j < matrix.length; j++){
            if(i == 0 && j == 0){
                table += "<tr><td class='elementName'>~</td>";
                elementsNames.forEach(function(element){
                    table += "<td class='elementName'>"+element+"</td>";
                });
                table += "</tr><tr>";
            }
            if(j == 0){
                table += "<td class='elementName'>"+elementsNames[i]+"</td>";
            }
            if(i != j){
                //input id = cell_ij_input
                table += "<td id='cell_"+i+j+"' data-i='"+i+"' data-j='"+j+"'
 onClick='cellClick(this.id)' ><input id='cell_"+i+j+"_input' class='cellInput'
    ↪ type='text' value='"+matrix[i][j]+"' readonly></td>";
            }else{
                table += "<td>"+matrix[i][j]+"</td>";
            }
        }
        table += "</tr>";
    }
    targetDiv = $(targetDiv);
    targetDiv.text('');
    targetDiv.append(table);
}
```

printMatrix function takes two arguments:

- **node**: is the node that we want to print its matrix.
- **targetDiv**: is the html div id of where we want the matrix to be drawn.

An output example of the function **printMatrix** would look similar to the matrix shown in Figure 19.

Figure 19: Matrix print example



-	A	B	C	D
A	1	2	4.1	8
B	0.50	1	2.12	4.1
C	0.24	0.47	1	2.3
D	0.13	0.24	0.43	1

Notice the input fields and the corresponding names for elements at the top of the table and the left side.

When the user click on a node, the inconsistency for that node will be calculated and stored, along with all the traids for the matrix associated with it. This is broken into the following steps:

First, we know that every relation produces a traid with three values (X, Y, and Z). We also know that each traid can be calculated using the formula described by Koczkodaj in [1] and shown in Figure 20. Therefore a JSON structure to save each consistency is implemented as shown in Code 32.

Figure 20: Inconsistency Value Formula

$$f(x, y, z) = 1 - \min\left\{\frac{y}{xz}, \frac{xz}{y}\right\}$$

Code 32: inconsistency JSON structure

```
{
    incons: (1-Math.min((x*z)/y,y/(x*z))).toFixed(5),
    x:[i,j],
    y:[i,k],
    z:[j,k]
}
```

6 Testing

7 Publishing

References

[1] Waldemar W. Koczkodaj and Ryszard Szwarc. “On Axiomatization of Inconsistency Indicators in Pairwise Comparisons”. In: *CoRR* abs/1307.6272 (2013). arXiv: `1307.6272`. URL: `http://arxiv.org/abs/1307.6272`.