ENHSP - SJR Visualizer

Romain André - Clément Chamayou - Joseph Thibault

January 20th 2022

1 Introduction

The project is about creating a visualizer to graphically display problem resolutions via a solver called ENHSP. It is a search-based planning algorithm that uses heuristic search to find a plan for a problem described in PDDL. The planner takes the problem description and converts it into a graph-search problem, where each node represents a state of the system. The tool is a website that takes a file with the extension .sp_log as an input and displays it as a collapsible tree with a panel to visualize the data of each state.

First of all we are going to detail how you can access this tool. Then we are going to detail how you can use it to visualize your own PDDL solvings. Finally we will document the structure and the key parts of the code, in order to make it easily maintainable.

2 Installation

To use the visualizer, since it is not a software there is no need for any kind of installation as long as the user provides an *.sp_log* file to visualize. The generation of such files is detailed in the ENHSP User Manual whom you will find the link below. Once the user has a *.sp_log* to visualize, he can go to this Github page. Once on this page, he can visualize the search tree by uploading an *.sp_log* file after pressing on the following button :

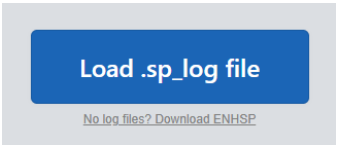

FIGURE 1 – Area to load a .sp_log file in the top left corner

Please refer to ENHSP Visualizer User Manual for further explanations on the different features of the visualizer.

3 Code documentation

In this document the different parts of the code are explained. You will first find the structure of the code and then the different functions implemented. For coding this visualizer, the D3.js library had been used and it can be found here.

3.1 Structure of the code

This visualizer is composed of three files : **index.html** and **stylesheet.css** respectively for the main architecture of the website and its style, **SearchSpaceTree.js** for the different functionalities. In the **SearchSpaceTree.js** file, the code is organised using the following steps :

1. Creating listeners on the first clickable items
2. Loading the tree
3. Retrieving data about tree's size
4. Updating the tree and its display

3.1.1 Creating listeners on the first clickable items

The first listener to be created is the one waiting for HTML to finish loading. Inside this listener, another one is created waiting for the user to upload a *.sp_log* file. After loading the file, the tree is created (function **loadTree()**) and parsed for the first time in order to calculate its dimensions and then to display them.

```
document.addEventListener('DOMContentLoaded', function() {
    zoom = d3.zoom().on("zoom", zoomed);

    svgbg = d3.select("#treePanel")
        .append("svg")
        .attr("width", "100%")
        .attr("height", "100%")
        .call(zoom)
        .on("dblclick.zoom", null)
        .on("contextmenu", function(d, i) {
            d3.event.preventDefault();
            // react on right-clicking
        });

    svg = svgbg
        .append("g")

    treemap = d3.tree().nodeSize([25, 25]);

    // Select the file input element
    document.getElementById('fileInput')
        .addEventListener('change', function() {
            // Read the file
            var fr = new FileReader();
            fr.onload = function() {
                treeData = fr.result.replaceAll("descendants", "children");
                console.log(treeData);
                loadTree();
            }

            fr.readAsText(this.files[0]);

            var appBanners = document.getElementsByClassName('hideUntilLoad');
            for (var i = 0; i < appBanners.length; i++) {
                appBanners[i].style.visibility = 'visible';
            }

            var appBanners = document.getElementsByClassName('hideAfterLoad');
            for (var i = 0; i < appBanners.length; i++) {
                appBanners[i].style.visibility = 'hidden';
            }
```

```
        })
})
```

3.1.2 Loading the tree

Before creating the tree, the predicate table is emptied (it is useful if another file had been loaded before and the user loads a new one). Then the tree is created with the d3 function **hierarchy()** and the tree's dimensions are calculated. Once done, the final path leading to the goal is identified with the function **getPlanPath()** detailed in the section 3.4. To be more user friendly, the tree is then reduced to its root node and its first set of children as well as being centered in the screen.

```
function loadTree() {
    // Empty predicateTable
    var table = document.getElementById("predicateTable");
    table.innerHTML = "";

    // Assigns parent, children, height, depth
    root = d3.hierarchy(JSON.parse(treeData), function(d) { return d.children; });
    root.x0 = height / 2;
    root.y0 = 0;

    // Getting and displaying information about the tree

    var treeSpecs = treeInfo(root)
    var width = document.getElementById("width")
    width.innerHTML = treeSpecs[0]
    var height = document.getElementById("height");
    height.innerHTML = treeSpecs[1];
    var averageBF = document.getElementById("averageBF");
    averageBF.innerHTML = treeSpecs[2]

    // Get the main plan path for the problem
    solutionNode = getLastVisitedNode();
    planPath = getPlanPath(solutionNode);

    resetZoom();
    closePanel();
    foldAllNodes();
}
```

3.1.3 Retrieving the dimensions of the tree

When loaded, the tree is not totally displayed in case it is too big. It is then useful to have the dimensions of the tree in order to predict its size. The function **treeInfo** does that by parsing the tree. It consists in a loop creating levels for each children generation. For instance : the root node is level 0, its children are level 1, their children are level 2, etc. For each level, the number of nodes in this level (represented by **currentLevel.length**) is compared to the variable **width**. The max between the two becomes the new width of the tree. The **average branching factor** is calculated by dividing the total number of nodes by the total of levels. Finally, the height of the tree is given by the d3 attribute *height* when applied to a root node.

```
function treeInfo(root) {

    if (!root) return 0

    var currentLevel = [root]
    var nextLevel = []
    var width = 0
    var totalLevels = 0
    var n_nodes = 0

    while (currentLevel.length > 0) {
        totalLevels++
        width = Math.max(width, currentLevel.length)
        for (let i = 0; i < currentLevel.length; i++) {
            let node = currentLevel[i]
            n_nodes++
            if (node.children) nextLevel = nextLevel.concat(node.children)
        }
        currentLevel = nextLevel
        nextLevel = []
    }

    var averageBF = n_nodes / totalLevels

    return [width, root.height,  Math.round(averageBF * 10) / 10]

}
```

3.1.4 Updating the tree and its display

Here comes the main function regarding the display of the tree. This **update()** function is composed of the following parts :

1. Setting up the tree
2. Setting the nodes ID
3. Updating new nodes
4. Removing nodes not needed
5. Setting the style of the nodes and the links
6. The **action()** function
7. The **propertiesPanel()** function

3.1.4.1 Setting up the tree

Here, the root node is taken as input and a new layout is computed using the treemap function. The position of the nodes in the tree is then updated by assigning the y position based on their depth.

```
// Assigns the x and y position for the nodes
var treeData = treemap(root);

// Compute the new tree layout.
var nodes = treeData.descendants(),
```

```
    links = treeData.descendants().slice(1);

// Normalize for fixed-depth.
nodes.forEach(function(d) { d.y = d.depth * 180 });
```

3.1.4.2 Setting the nodes ID

This code is updating the nodes in an SVG (Scalable Vector Graphics) element. It first selects all 'g' elements with class 'node', binds data to them, and sets the class attribute to 'node'. The data binding is done based on each node's unique identifier, which is generated if it does not already exist.

```
var node = svg.selectAll('g.node')
    .data(nodes, function(d) { return d.id || (d.id = ++i); })
    .attr("class", "node");
```

3.1.4.3 Updating new nodes

It then updates the attributes and styles of the nodes based on their data, adding new elements to the nodes if needed. The nodes are also updated to transition smoothly to their new position.

```
var nodeEnter = node.enter().append('g')
        .attr('class', 'node')
        .attr("transform", function(d) {
            posX0 = isNaN(source.x0) ? 0 : source.x0;
            posY0 = isNaN(source.y0) ? 0 : source.y0;
            return "translate(" + posY0 + "," + posX0 + ")";
        })
        .on("contextmenu", function(d) {
            d3.event.preventDefault();
            // react on right-clicking
            action(this, d, true);
        })
        .on("click", function(d) {
            //if shift key is pressed, is action2
            if (d3.event.shiftKey) {
                action(this, d, true);
            } else {
                action(this, d, false);
            }
        })
        .on("mouseover", function(d) {
            //change size timed
            d3.select(this).select('circle.node')
                .transition()
                .duration(100)
                .attr('r', 13);
        })
        .on("mouseout", function(d) {
            //change size
            d3.select(this).select('circle.node')
                .transition()
```

```
                .duration(100)
                .attr('r', 10);
        });
```

3.1.4.4 Removing nodes not needed

The following code is removing any nodes that are exiting the visualization. It animates them to transition to the location of their parent node, reduces their circle size to 0, and reduces the opacity of their text labels to 0. Once the animation is complete, the nodes are removed from the visualization.

```
// Remove any exiting nodes
var nodeExit = node.exit().transition()
    .duration(duration)
    .attr("transform", function(d) {
        return "translate(" + source.y + "," + source.x + ")";
    })
    .remove();

// On exit reduce the node circles size to 0
nodeExit.select('circle')
    .attr('r', 1e-6);

// On exit reduce the opacity of text labels
nodeExit.select('text')
    .style('fill-opacity', 1e-6);
```

3.1.4.5 Setting the style of the nodes and the links

The following lines are setting the style of the nodes. First, a circle around the nodes is added. If it has children, an arrow is added inside the node. Then labels are added : the integer "visited_step" indicating whether the node had been visited and the action done on that node. The text for the action is split in two lines to gain some space.

```
// Add Circle for the nodes
nodeEnter.append('circle')
    .attr('class', 'node')
    .attr('r', 1e-6)
    .style("fill", function(d) {
        return d._children ? "lightsteelblue" : "#fff";
    });

// If node has children
nodeEnter.append('text')
    .attr("dy", ".35em")
    .attr("text-anchor", "middle")
    .text(function(d) {
        return d._children ? "->" : d.children ? "<-" : "";
    })
    .attr("fill", function(d) {
        return d._children ? "black" : d.children ? "#1B65B6" : "black";
    });

```

```
// Add labels for the nodes
nodeEnter.append('text')
    // .attr("dy", ".35em")
    .attr("dy", "-.35em")
    .attr("x", -12)
    .attr("text-anchor", "end")
    .text(function(d) {
        var text = d.data.visit_step;
        return text;
});

// Add label fo the action done at this node if it's part of the problem's solution (first half of the action)
nodeEnter.append('text')
    .attr("dy", "-.25em")
    .attr("x", 15)
    .text(function(d) {
        var text = "";
        if (d.data.action != null) {
            var posSpace = d.data.action.substring((d.data.action.length / 2) - 1, d.data.action.length).indexOf(" ");
            if (posSpace != -1) text = d.data.action.substring(0, (d.data.action.length / 2) - 1 + posSpace);
            else text = d.data.action;
        }
        return text;
    });
// Second half
nodeEnter.append('text')
    .attr("dy", ".95em")
    .attr("x", 15)
    .text(function(d) {
        var text = "";
        if (d.data.action != null) {
            var posSpace = d.data.action.substring((d.data.action.length / 2) - 1, d.data.action.length).indexOf(" ");
            if (posSpace != -1) text = d.data.action.substring((d.data.action.length / 2) - 1 + posSpace + 1, d.data.action.length);
        }
        return text;
    });
```

Once new nodes are added and initialized they are added to an array with all the other nodes already on the screen. Then their graphic visualization is updated :

```
var nodeUpdate = nodeEnter.merge(node);

// Transition to the proper position for the node
nodeUpdate.transition()
    .duration(duration)
    .attr("transform", function(d) {
        return "translate(" + d.y + "," + d.x + ")";
    });
```

```
9
10 // Update the node attributes and style
11 nodeUpdate.select('circle.node')
12     .attr('r', 10)
13     //outline
14     .style("stroke", function(d) {
15         //black if visited
16         return d.data.visited ? "black" : "#1B65B6";
17     })
18     .style("fill", function(d) {
19         //fill if visited
20         return d.data.visited ? "lightsteelblue" : "#fff";
21     })
22     .style("stroke-width", "2px")
23     .attr('cursor', 'pointer');
```

Regarding the links, the structure is very similar : new links are added to the visualization, styled whether it represents a part of the solution or not, useless links are removed. The shape of the path is made by the function **diagonal()**.

```
1  // Update the links...
2  var link = svg.selectAll('path.link')
3      .data(links, function(d) { return d.id; });
4
5  // Enter any new links at the parent's previous position.
6  var linkEnter = link.enter().insert('path', "g")
7      .attr("class", "link")
8      .attr('d', function(d) {
9          var o = { x: source.x0, y: source.y0 }
10         return diagonal(o, o)
11     });
12
13 // Make link thicker and darker if visited, colored if in planPath
14 linkEnter.style("stroke", function(d) {
15         //if d is in planPath
16         if (planPath.includes(d)) {
17             return "#1B65B6";
18         } else if (d.data.visited) {
19             return "#444";
20         } else {
21             return "#777";
22         }
23     })
24     .style("stroke-width", function(d) {
25         //if d is in planPath
26         if (planPath.includes(d)) {
27             return "3px";
28         } else if (d.data.visited) {
29             return "2px";
30         } else {
31             return "1px";
32         }
33     });
```

```
34
35
36 // UPDATE
37 var linkUpdate = linkEnter.merge(link);
38
39 // Transition back to the parent element position
40 linkUpdate.transition()
41     .duration(duration)
42     .attr('d', function(d) { return diagonal(d, d.parent) });
43
44 // Remove any exiting links
45 var linkExit = link.exit().transition()
46     .duration(duration)
47     .attr('d', function(d) {
48         var o = { x: source.x, y: source.y }
49         return diagonal(o, o)
50     })
51     .remove();
52
53 // Store the old positions for transition.
54 nodes.forEach(function(d) {
55     d.x0 = d.x;
56     d.y0 = d.y;
57     d.data.state = parseData(d.data.state);
58 });
59
60
61 // Creates a curved (diagonal) path from parent to the child nodes
62 function diagonal(s, d) {
63
64     path = `M ${s.y} ${s.x}
65         C ${(s.y + d.y) / 2} ${s.x},
66           ${(s.y + d.y) / 2} ${d.x},
67           ${d.y} ${d.x}`
68
69     return path
70 }
```

3.1.4.6 The action() function

When a node is pressed, the code checks whether the user pressed with left or right click (or shift key + left click). It will then call the action function which will determine based on the input, and if we have inverted controls or not, which behavior to execute. Its main behavior is to update and open the properties panel, and change the selected node color to red and makes its border thicker. If its the secondary behavior, it folds or unfolds the node (expands its children), updates its text label (change the direction of its arrow), and updates the tree. The **focus** parameter is a reference to the clicked node element in the DOM (Document Object Model). The **d** parameter represents the data associated with the selected node. The **action2** parameter is an optional boolean value that indicates whether the action executed was the main one or the secondary one.

```
1 function action(focus, d, action2 = false) {
2     if ((!reversedActions && !action2) || (reversedActions && action2)) {
3         propertiesPanel(d);
```

```
        d3.selectAll(".node").classed("selected", false);
        d3.select(focus).classed("selected", true);
        //change all other nodes outline color
        d3.selectAll(".node").select('circle.node')
            .style("stroke", function(dd) {
                //black if visited
                return dd.data.visited ? "black" : "#1B65B6";
            })
            .style("stroke-width", "1.5px");
        //change node outline color and border thickness animation
        d3.select(focus).select('circle.node')
            .style("stroke", "red")
            .style("stroke-width", "3px");
    } else {
        if (d.children) {
            fold(d);
        } else {
            unfold(d);
        }
        d3.select(focus).select("text").text(function(d) {
                return d._children ? "" : d.children ? "" : "";
            })
            .attr("fill", function(d) {
                return d._children ? "black" : d.children ? "#1B65B6" : "black";
            });
        update(d);
    }
}
```

3.1.4.7 The propertiesPanel() function

This function is a part of the **update()** function but is described in the section 3.7.3.

3.2 Fold/Unfold

3.2.1 Fold node(s)

The method **fold** only one node

```
function fold(d) {
    d._children = d.children;
    d.children = null;
}
```

Collapse will fold the node and all it's children recursively

```
function collapse(d) {
    if (d.children) {
        fold(d);
        d._children.forEach(collapse)
    }
}
```

FoldAllNodes will call the previous method to collapse the tree starting from the root and then it will visually update the graph. The user will be able to only see the root node and its first children. It is called when pressed on the corresponding button on the interface.

```
function foldAllNodes() {
    root.children.forEach(collapse);
    update(root)
}
```

3.2.2 Unfold node(s)

This function **unfold** only one node by displaying its children

```
    function unfold(d) {
    d.children = d._children;
    d._children = null;
}
```

We created the function **uncollapse** to expand all nodes starting at the node d. It will expand it and the function recursively called itself for each children.

```
function uncollapse(d) {
    if (d._children) {
        unfold(d);
    }
    if (d.children) {
        d.children.forEach(uncollapse);
    }
}
```

DisplayAllNodes will call the previous method to uncollapse the tree starting with the root and then it will visually update the graph. The user will be able to see the full tree. It is called when pressed on the corresponding button on the interface.

```
function displayAllNodes() {
    uncollapse(root)
    update(root)
}
```

3.3 Zooming on the tree and resetting the zoom

The function **zoomed()** allows the user to zoom in and out on the tree, it is called by the d3.js on zoom callback. The **resetZoom()** will recenter the tree on the first node. It is called when pressed on the corresponding button on the interface, or when a new tree is loaded.

```
function zoomed() {
    //limit zoom
    if (d3.event.transform.k > 5) {
        d3.event.transform.k = 5;
    } else if (d3.event.transform.k < 0.1) {
```

```
        d3.event.transform.k = 0.1;
    }
    svg.attr("transform", d3.event.transform)
}
```

```
function resetZoom() {
    var transform = d3.zoomIdentity;
    //centers transform
    transform.x = (d3.select("svg").node().getBoundingClientRect().width / 3);
    transform.y = (d3.select("svg").node().getBoundingClientRect().height / 2);

    transform.k = 2;

    svgbg.transition()
        .duration(750)
        .call(zoom.transform, transform);
}
```

3.4 Finding the solution path

In order to find the solution path and to highlight it, the first thing to do is to find the last node of the solution, which has the highest visit_step, it is done by **getLastVisitedNode()**. Then the function **getPlanPath()** will recursively finds the path all the way to the root from the given node.

```
// Function to find the node with the highest visited step
function getLastVisitedNode() {
    //get all nodes and put them in a list
    var nodesList = treemap(root).descendants();

    //sort list by visit_step
    nodesList.sort(function(a, b) {
        return a.data.visit_step - b.data.visit_step;
    });

    //get the node with the highest visit_step
    var lastNode = nodesList[nodesList.length - 1];

    //return this node
    return lastNode;
}

// Function to return the path to the last visited node
function getPlanPath(lastNode) {
    //work our way up from the last visited node
    var path = [];
    var node = lastNode;
    while (node.parent) {
        path.unshift(node);
        node = node.parent;
    }
```

```
    return path;
}
```

3.5 Toggling reversed actions

The function **toggleReversedActions()** toggles the value of the global variable reversedActions between true and false. It then updates the text of the HMTL to display the actions that correspond to right-clicking or shift+left-clicking a node, depending if they are inverted or not. It is called when pressed on the corresponding button on the interface.

```
function toggleReversedActions() {
    reversedActions = !reversedActions;
    //get id actionText
    var actionText = document.getElementById("actionText");
    if (reversedActions) {
        actionText.innerHTML = "<b>Node Properties: </b>Right Click/Shift+Left
        ↪  Click<br><b>Expand Node: </b>Left Click";
    } else {
        actionText.innerHTML = "<b>Node Properties: </b>Left Click<br><b>Expand Node:
        ↪  </b>Right Click/Shift+Left Click";
    }

    //get class actionbtn
    var actionbtn = document.getElementsByClassName("actionbtn");
    if (reversedActions) {
        //set background color
        actionbtn[0].style.backgroundColor = "#1B65B6";
        //set text color
        actionbtn[0].style.color = "#FFFFFF";
    } else {
        //set background color
        actionbtn[0].style.backgroundColor = "#FFFFFF";
        //set text color
        actionbtn[0].style.color = "#222222";
    }
}
```

3.6 Reaching the last node

When the tree is too large to manually explore, it can be fastidious to reach the end of it. Here is the main purpose of the two following functions : reaching the last node of the solution with only one click. The function **focusLastNode()** displays all the nodes and then focuses the view and zoom on the last node of the problem solution given its position. This position is retrieved with the function **findNodePosition(node)**. It allows the user to easily see the state of each properties the problem once it is solved. The function **focusLastNode()** is called when the corresponding button is pressed.

```
function focusLastNode() {
    displayAllNodes();

    // Put the root node back in the center of the screen
```

```
    var transform = d3.zoomIdentity;

    // Set focus position
    var pos = findNodePosition(solutionNode);
    transform.x = (d3.select("svg").node().getBoundingClientRect().width / 2) - pos[1];
    transform.y = (d3.select("svg").node().getBoundingClientRect().height / 2) - pos[0];

    // Reset zoom
    transform.k = 2;

    svgbg.transition()
        .duration(750)
        .call(zoom.transform, transform);
}

//function to find node position in canvas
function findNodePosition(node) {
    var transform = d3.zoomTransform(svg.node());
    //accounts for zoom (default) = 2
    var x = node.x * 2;
    var y = node.y * 2;
    return [x, y];
}
```

3.7 Sliding node properties panel

3.7.1 Parse state data

The following function **parseData(data)** takes as a parameter the data present in the "state" variable of the node and order it in a dictionnary. Each predicate in the sp_log file respect the following format in the state attribute : (predicate_name)=predicate_value. We gather the predicate name between the parenthesis and gather the predicate value after the equal until the next open parenthesis. We crop the data string to remove the parsed part. The loop ends when we can't find another opening parenthesis in the remaining data. The returned dictionnary is really easy to read and will be used to fill the predicate table.

```
function parseData(data) {
    if (typeof data === 'string' || data instanceof String) {
        var dict = {};
        var notTheEnd = true
        var value;
        while (notTheEnd) {
            var variable = data.substring(data.indexOf("(") + 1, data.indexOf(")"));
            data = data.substring(data.indexOf("=") + 1);
            if (data.indexOf("(") != -1) {
                value = data.substring(0, data.indexOf("("));
            } else {
                value = data;
                notTheEnd = false;
            }
            dict[variable] = value;
        }
        return dict;
```

```
    }
    return data;
}
```

3.7.2 Filling of the predicate Table

The function **fillpredicateTable(data, parentData)** bellow takes as parameters the previous dictionnary and another one from its parent node. First of all we are emptying the predicate table before adding anything. For each predicate we insert a new row in the predicate table. If the value of the predicate in the dictionnary is not equal to the value of the parent node, the old value is shown in bold and the new one in red. If the current value is equal to the parent value then we display it normally. For each float value we are displaying only 2 digits after the decimal, to avoid large numbers on screen.

```
function fillpredicateTable(data, parentData = null) {

    //Emptying the table
    var table = document.getElementById("predicateTable");
    table.innerHTML = "";

    Object.keys(data).forEach(element => {
        var row = table.insertRow(-1);

        if (parentData != null && data[element] != parentData[element]) {
            row.insertCell(-1).innerHTML = "<b>" + element + "</b>";
            //keep only 2 digits after .
            var indexOfDot = data[element].indexOf(".");
            if (indexOfDot == -1) {
                row.insertCell(-1).innerHTML = "<b>" + parentData[element] + " => <span
                ↪  style=\"color: red;\">" + data[element] + "</span></b>";
            } else {
                row.insertCell(-1).innerHTML = "<b>" + parentData[element].substring(0,
                ↪  indexOfDot + 3) + " => <span style=\"color: red;\">" +
                ↪  data[element].substring(0, indexOfDot + 3) + "</span></b>";
            }
        } else {
            row.insertCell(-1).innerHTML = element;
            //keep only 2 digits after .
            var indexOfDot = data[element].indexOf(".");
            if (indexOfDot == -1) {
                row.insertCell(-1).innerHTML = data[element];
            } else {
                row.insertCell(-1).innerHTML = data[element].substring(0, indexOfDot + 3);
            }
        }
    });
}
```

3.7.3 Displaying the predicate table

The function **propertiesPanel(d)** takes as a parameter the information of the node. First of all, the function gathers the parent data of the parent node. Then we initialize the 3 fields (distance, action and actionCost) that

appear in the above part of the predicate table. Next it fills the predicate table using the former function. Finally, we are closing the predicate panel if the node pressed is the same as the last one pressed. Otherwise it will opens it.

```
    // Open the predicate table
    function propertiesPanel(d) {

        var parentData = null;
        var nodeData = d.data;

        if (d.parent != null) {
            parentData = d.parent.data;
        }

        var distance = document.getElementById("distance");
        distance.innerHTML = nodeData.distance;

        var action = document.getElementById("action");
        action.innerHTML = nodeData.action;

        var actionCost = document.getElementById("actionCost");
        actionCost.innerHTML = nodeData.action_cost_to_get_here;

        if (parentData == null) {
            fillpredicateTable(nodeData.state);
        } else {
            fillpredicateTable(nodeData.state, parentData.state);
        }

        const sliderPanel = document.getElementById("sliderPanel");
        if (lastClicked === d) {
            sliderPanel.style.bottom = "-65%";
            lastClicked = null;
        } else {
            sliderPanel.style.bottom = "20px";
            lastClicked = d;
        }
    }
```

3.7.4 Closing the panel

This small function gathers the panel by its id. Then it changes its position by editing the CSS with a negative value to hide the panel. Finally, since the panel is closed, no node is selected so the variable lastClicked is set to null. This function is triggered either by pressing the already selected node, or by pressing the "Close Tab" button of the node properties panel.

```
function closePanel() {
    const fullAssignmentPanel = document.getElementById("sliderPanel");
    fullAssignmentPanel.style.bottom = "-65%";
    lastClicked = null;
}
```