Technical Documentation
Drzewa++

Piotr Mikstacki,
Błażej Pudlik,
Maciej Skrabski,
Michał Wojtaszek,
Marcin Wróblewski

January 2020

1 Introduction

The aim of this project is to visualize and provide data modification in tree data structures. JavaScript is the main language of our project and web page is the final product.
The first phase of the project has been written in Python, because it was easier for our developers to begin with. With the main idea of the project coded in Python and working well, we decided to rewrite it to JavaScript.
Thanks to our program being lightweight, we can work at real time and use only the interpreter of user's browser.

2 Data types

In order to provide wanted functionalities we need some complex data types. Not only must they keep the values and references of the data, but also be able to modify them. We chose arrays as the lightest and fastest data type that met our requirements.
The next section describes how we managed working with tree structures on arrays.

3 Tree generation and swapping operation

3.1 *Swap* function

takes three arguments:

1. the tree in which we want the swapping to take place,
2. the first node (as list of arguments, which are this node's coordinates),
3. the second node (just like the first one) (You always swap just two branches at a time, swapping of more elements can always be divided into combination of swapping just two elements at the time, in other words, any cycle of length r is a product of $r - 1$ transpositions).

How does the function work?

1. if the nodes are the same - don't change the tree. Just return it. Skip the rest of the function,
2. if both nodes passed as arguments have *different* parents:
   (a) print 'Different parents! Cannot swap.' to screen
   (b) return tree
   (c) skip rest of the function

   (now comes the tricky part - recursion)
3. if the first coordinate is the same for both nodes:

```
swap(tree[first_node[first_coordinate]],
    first_node[/enter the "first node's tree" at first coordinate/],
    second_node[/enter the "second node's tree" at first coordinate/]).
```

4. else:

```
a = first_node[second_to_last_coordinate]
b = second_node[second_to_last_coordinate]
tree[a], tree[b] = tree[b], tree[a] (this is an actual swap of the nodes)
return tree, end function
```

3.2 How many nodes are there on the n-th level?

number of vertices deriving from each node (except for the final leaves, for obvious reasons) raised to the n-th level's power

3.3 How many nodes are there in total in the tree?

Sum the number of nodes on each level using the function from previous point.

3.4 Tree generation

on input:

- n — number of levels;
- p — number of vertices deriving from each node (except for the final leaves, for obvious reasons)

3.4.1 Optimizing trivial cases

```
if _n_ < 1 or _p_ < 1:
    print 'bad input data' information to the screen at exit of the function.
    There is no point in generating such a tree, obviously.

if _n_ is equal to 1:
    Return a single _node_ with value 1, as it is the only node in the tree,
    therefore it will always have the value of 1.
```

3.4.2 Actual generation

1. Creating two empty arrays:
    - arr = [ ] - for holding all nodes that haven't been appended to the tree yet
    - tree = [ ] - the tree itself, pretty self-explanatory
2. Initiating an array of all nodes in the tree

```
for i in range(totalNodes(n, p), 0, -1):
    arr.append(i)
```

3. generating a layer of leafs and their parents, by picking them up from the array from point 1., and appending it to a tree
   (a) temporary array that will come in handy
   ```
   temp = []
   ```
   (b) loop iterating on the range from 0 to the number of nodes on the last level, the leaf level
   ```
   for i in range(0, nodesOnLvl(n-1, p)):
   ```
   (c) append the temporary array with a node from the array holding all the nodes, remove that node from the array holding all the nodes.
   ```
   temp.append(arr.pop())
   ```
   (d) if the number of nodes in the temporary array corresponds with p (which is number of vertices deriving from each node) append those nodes to the tree and append a value next to that group of nodes - this will be their parent. So, to put it in layman's terms – if there are enough nodes to form a child – do so and attach a father to them
   ```
   if ((i+1) % p == 0 and i != 0):
     tree.append([arr.pop(), temp])
     temp = [] (clearing the temporary array)
   ```

4. generating higher branches of the tree
   ```
   l = n-2 (because we've already created the first
   two layers in the leaf-parent generation explained above)
   for j in range(l, 0, -1): (iterating down)
       temp2 = [] (another temporary array
       that will come in handy)
       for i in range(0, nodesOnLvl(j, p)):
           temp2.append(tree[i])
           if ((i+1) % p == 0 and i != 0):
               temp.append([arr.pop(), temp2])
               temp2 = []
       if(nodesOnLvl(j, p) != 0):
           tree = temp
       temp = []
       (pretty self explanatory after explaining
           the leaf-parent generation)
   ```

5. return tree[0]

4 Tests

We are fans of test-driven development. Majority of tests have been written in the first phase of project, when we were working on the structure of the tree. When finished, our tests were focused on testing the UI and UX on our page. Some structural tests:

- First example - hardcoded test:

```
a = [1, 0]
b = [1, 2]

drzewo = [13, [[4, [1, 2, 3]], [8, [5, 6, 7]], [12, [9, 10, 11]]]]
swap(drzewo, a, b)
expected = [13, [[12, [9, 10, 11]], [8, [5, 6, 7]], [4, [1, 2, 3]]]]
print(drzewo == expected)


t1 = generateTree(6, 5)
c1 = t1[1][4][1][3]
c2 = t1[1][4][1][0]

a = [1, 4, 1, 3]
b = [1, 4, 1, 0]
swap(t1, a, b)

print(t1[1][4][1][3] == c2 and t1[1][4][1][0] == c1)

t1 = generateTree(2, 2)
print(t1)
swap(t1, [1, 0], [1, 1])
print(t1 == [3, [2, 1]])
```

- Another example: the function to test results of modification in the tree.

```
export function compareArrays (array1, array2) {
    if(typeof array1 !== 'object' || typeof array2 !== 'object'){
        return array1 === array2
    }
    return array1.length === array2.length &&
        array1.sort().every(function (value, index) {
            return value === array2.sort()[index]
        })
}
```

Our application positively passes most of automatic tests, although we reported some problems during 'production' tests. We noticed that whenever a new tree was created, old functions existed, so they were not suitable for new tree.

5 Graphical scheme of project structure

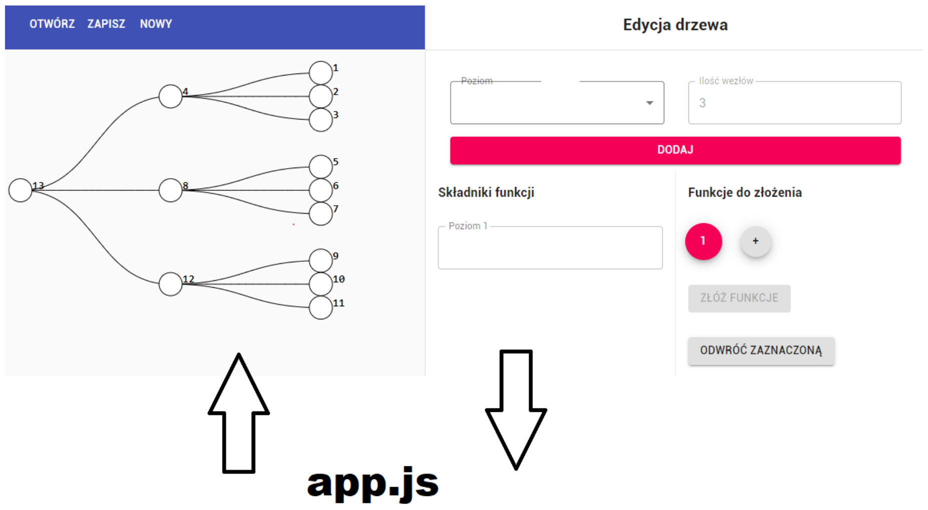

Figure 1: Project structure

As shown in the picture we work in one direction flow:

1. get the data from user
2. use the data and modify the tree structure
3. graphically show efects of the modification made in 2.

- Navigation bar is implemented to save/open/create new Tree.
- We use react Redux to manage control data in our app. Therefore after any changes from user we can redraw the visualization.
- Before visualization we need to generate tree in code due to data collected from user. It is all described in code inside the "tree" folder, both generating and modifying the tree included.
- For graphical visualization we used react's library called "react-tree-graph". It's a package to display tree structures in SVG format. We normalized our data to match the requirements of the library.