A Guide to JavaScript Algorithms — Graph and Tree Traversal

A Quick Guide to Writing JavaScript Graph and Tree Traversal Algorithms

Rajat S Follow
Jan 1, 2019 · 9 min read

Algorithms and data structures have always been connected together. Without data structures, we would not have any objects by which we can implement our logic to solve the problem statement, and without algorithms, our objects cannot be consumed.

An algorithm is a function that takes in some data structure as input and manipulates it into some kind of output. The type of output that we get is based on the logic of the algorithm. To do all this, it is important that the developer clearly understands the problem statement and what is asked of him.

Understanding the logic behind the algorithm is fairly easy. things get more complicated when you are required to turn the logic into actual code.

In this post, we will take a look at some the common traversal algorithms in JavaScript. Some of these are very commonly asked in coding interviews, while others are not so common and will usually make you scratch your head in order to solve them.

Tip: Discover and share JS components in Bit's component platform, and use components like Leko to build applications faster with your team. Give it a try.

. . .

Breadth First Searching Algorithm

Breadth first is an algorithm that is used to search for a node in a graph data structure. The algorithm starts at one node, then goes to its neighboring nodes. If the node we are searching for is not found, then it will go to the next node look at its neighbors.

This algorithm also uses the queue data structure to make note of all the nodes that it has visited. This way, the algorithm will save time by skipping the already visited nodes.

Let's start by writing the queue, node, and graph creator functions as shown below:

```
queueCreator = () => {
  const queue = []
  return {
    add(x) {
      queue.unshift(x)
    },
    remove() {
      if (queue.length === 0) {
```

```
        return undefined
      }
      return queue.pop()
    },
    next() {
      if (queue.length === 0) {
        return undefined
      }
      return queue[queue.length - 1]
    },
    get length() {
      return queue.length
    },
    empty() {
      return queue.length === 0
    }
  }
}

nodeCreator = (id) => {
  const neighbors = []
  return {
    id,
    neighbors,
    addNeighbors(node) {
      neighbors.push(node)
    }
  }
}

graphCreator = (uni = false) => {
  const nodes = []
  const edges = []
  return {
    uni,
    nodes,
    edges,
    addNode(id) {
      nodes.push(nodeCreator(id))
    },
    searchNode(id) {
      return nodes.find(n => n.id === id)
    },
    addEdge(idOne, idTwo) {
      const a = this.searchNode(idOne)
```

```
        const a = this.searchNode(idOne)
        const b = this.searchNode(idTwo)

        a.addNeighbors(b)
        if (!uni) {
          b.addNeighbors(a)
        }
        edges.push(`${idOne}${idTwo}`)
      },
      display() {
        return nodes.map(({neighbors, id}) => {
          let output = `${id}`
          if (neighbors.length) {
            output += ` => ${neighbors.map(node => node.id).join(' ')}`
          }
          return output
        }).joing('\n')
      },
    }
  }
```

breadthFirst.js hosted with ♡ by GitHub view raw

Inside the `graphCreator` object, we will create a `breadthFirst` method that takes in two arguments. The first argument will be the `id` of the starting node and the second argument will be a function that gets called everytime we visit the node's neighbors.

We will use the `reduce` function on the `nodes` array to reduce it to an object where each `id` is the current node's `id` and the entire value is set to false. It will be set to true when the algorithm visits the corresponding node.

After that, we will keep make a note of all the nodes that we need to visit by using the `queueCreator` function. The traversal algorithm will only work `while` this queue is not empty. So we will fill the queue with the `startingNode` and its neighbors.

Every time the `while` loop ends an iteration, we will remove a node from the queue and set it as the `currentNode`. If the algorithm has not visited that node before, then we will call the `neighborVisit` function and set its value to `true`.

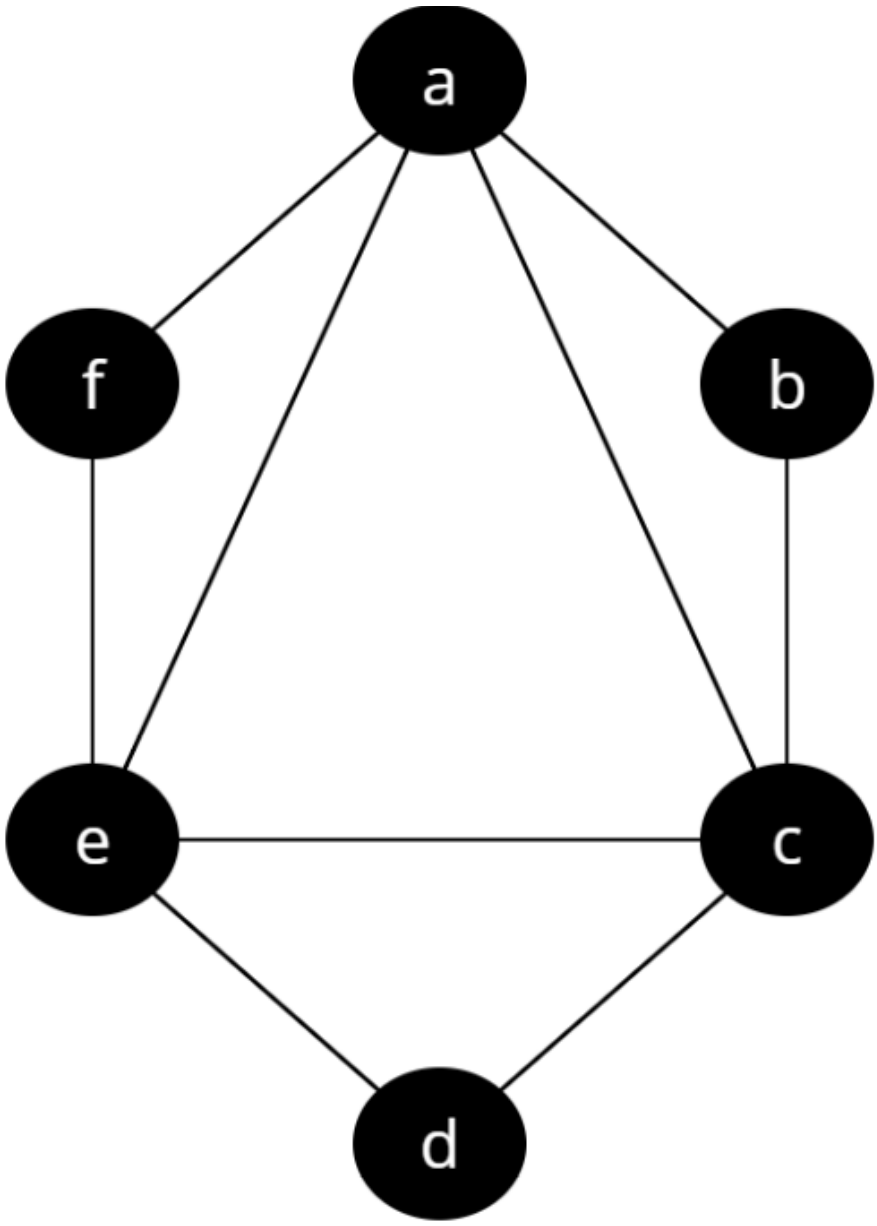
Enough talking, lets write the code as shown below:

```
breadthFirst(startingNode, neighborVisit) {
  const firstNode = this.searchNode(startingNode)
  const visitedNode = nodes.reduce((sum, node) => {
    sum[node.id] = false
    return sum
  }, {})
  const queue = queueCreator()
  queue.add(firstNode)
  while (!queue.empty()) {
    const temp = queue.remove()
    if (!visitedNode[temp.id]) {
      neighborVisit(temp)
      visitedNode[temp.id] = true
    }
    temp.neighbors.forEach(node => {
      if(!visitedNode[node.id]) {
        queue.add(node)
      }
    })
  }
}
```

We can now test the algorithm out by creating a graph data structure first. As shown below:

```
const graph = graphCreator(true)
graph.addNode('a')
graph.addNode('b')
graph.addNode('c')
graph.addNode('d')
graph.addNode('e')
graph.addNode('f')
graph.addEdge('a', 'c')
graph.addEdge('a', 'e')
graph.addEdge('b', 'a')
graph.addEdge('b', 'c')
graph.addEdge('c', 'd')
graph.addEdge('c', 'e')
graph.addEdge('d', 'e')
graph.addEdge('e', 'f')
graph.addEdge('f', 'e')
```

The graph created here would look something like this:



We can now use the `breadthFirst` method to print the node ids as in the order in which the algorithm arrives at the node, as shown below.

```
graph.breadthFirst('c', node => {
  console.log(node.id)
})

// Output
c
d
e
f
```

Here the algorithm starts with the node `c`. This node is connect to nodes `d` and `e`. So, the algorithm will print out the nodes `c` followed by `d` and `e`. Next, the algorithm will look at the neighbors of node `d`. Node `d` is only connected to `e`, which will be ignored by the algorithm since it is already visited. The algorithm will now look at the neighbors of `e`. Node `e` if only connected to node `f`, which the algorithm will print out in the console.

. . .

Depth First Searching Algorithm

Depth first algorithm is a traversal algorithm that starts searching at one node, and then goes down the path of its neighboring nodes before it goes to the other paths.

Lets go to back into the `graphCreator` object and create a new method named `depthFirst`. Similar to the `breadthFirst` method, this method will also take two arguments, one to define which node to start the traversal from and the other argument is a function that will be called as the algorithm visits each node for the first time.

```
depthFirst(startingNode, neighborVisit) {
  const firstNode = this.searchNode(startingNode)
  const visitedNode = nodes.reduce((sum, node) => {
    sum[node.id] = false
    return sum
  }, {})
  // Write the next code here
}
```

In the depth first algorithm, if there is another level to go down, then the algorithm will `travel` down that path first. We will create a function named `travel` that will check if the node in that path was already visited. If it was, then the algorithm will return nothing. Else, the algorithm will call the `neighborVisit` function and mark the node as visited by setting the `sum[node.id]` value to `true`. We will also loop through all the neighbors of the node and call `travel` on them, and we will start the algorithm by calling `travel` on the `firstNode` as shown below:

```
travel = (node) => {
  if (visitedNode[node.id]) {
    return
  }
  neighborVisit(node)
  visitedNode[node.id] = true

  node.neighbors.forEach(neighbor => {
    travel(neighbor)
  })
}
travel(firstNode)
```

Our algorithm is now ready! Let's test it out on the same graph that we used for the breadth first algorithm.

```
const graph = graphCreator(true)

graph.addNode('a')
graph.addNode('b')
graph.addNode('c')
graph.addNode('d')
graph.addNode('e')
graph.addNode('f')

graph.addEdge('a', 'c')
graph.addEdge('a', 'e')
graph.addEdge('b', 'a')
graph.addEdge('b', 'c')
graph.addEdge('c', 'd')
graph.addEdge('c', 'e')
graph.addEdge('d', 'e')
graph.addEdge('e', 'f')
graph.addEdge('f', 'e')
```

Calling the `depthFirst` method on the graph with node `a` as the `startingNode` will give it the following output

```
graph.depthFirst('a', node => {
  console.log(node.id)
})

// Output
a
```

```
c
d
e
f
```

Node `a` is first connected to node `c`, so the algorithm will travel down this path first. The node `c` is then connected to node `d`, which is connected to node `e`, which is finally connected to node `f`.

. . .

Binary Tree Traversal Algorithms

A binary tree is a tree data structure where each node can only have upto two child nodes.

To start, we will write a node creating function that will take the `id` as an argument. It will also have a `left` and `right` properties that are initially set to `null`. Also, we will write methods to add child nodes, one to the left path and the other to the right path as shown below.

```
nodeCreator(id) {
  return {
    id,
    left: null,
    right: null,
    addToLeft(leftId) {
      const newLeftNode = nodeCreator(leftId)
      this.left = newLeftNode
      return newLeft
    },
    addToRight(rightId) {
      const newRightNode = nodeCreator(rightId)
      this.right = newRightNode
      return newRight
    }
  }
}
```

We now get to the traversal part of this section. A binary tree can be traversed in three ways:

- In-order
- Pre-order
- Post-order

These three are actually quite similar to each other. Each of them will receive a starting node and a recurring function that the algorithm will use to traverse through the binary tree. First, let's create an object that will store the code for these traversal processes.

```
const algorithms = {
  IN: () => {},
  PRE: () => {},
  POST: () => {},
}
```

The in-order traversal algorithm first travels down the left branch, then goings to the current node, and then goes down the right branch. To code this algorithm, we will call the `IN` function recursively as shown below:

```
IN: (node, func) => {
  if (node !== null) {
    processes.IN(node.left, func)
    func(node)
    processes.IN(node.right, func)
  }
},
```

The pre-order algorithm first visits the current node, then travels down the left path, and then travels down the right path.

```
PRE: (node, func) => {
  if(node !== null) {
    func(node)
    processes.PRE(node.left, func)
```

```
      processes.PRE(node.right, func)
    }
  }
```

The post-order algorithm is the complete opposite of the pre-order algorithm. It travels down the left path first, then it travels down the right path, and finally visits the current `node`.

```
  POST: (node, func) => {
    if (node !== null) {
      processes.POST(node.left, func)
      processes.POST(node.right, func)
      func(node)
    }
  }
```

We still need to write the binary tree creating function. All trees have a root node. So, this function will first receive a `rootId` as an argument, using which it will create a node named `root`.

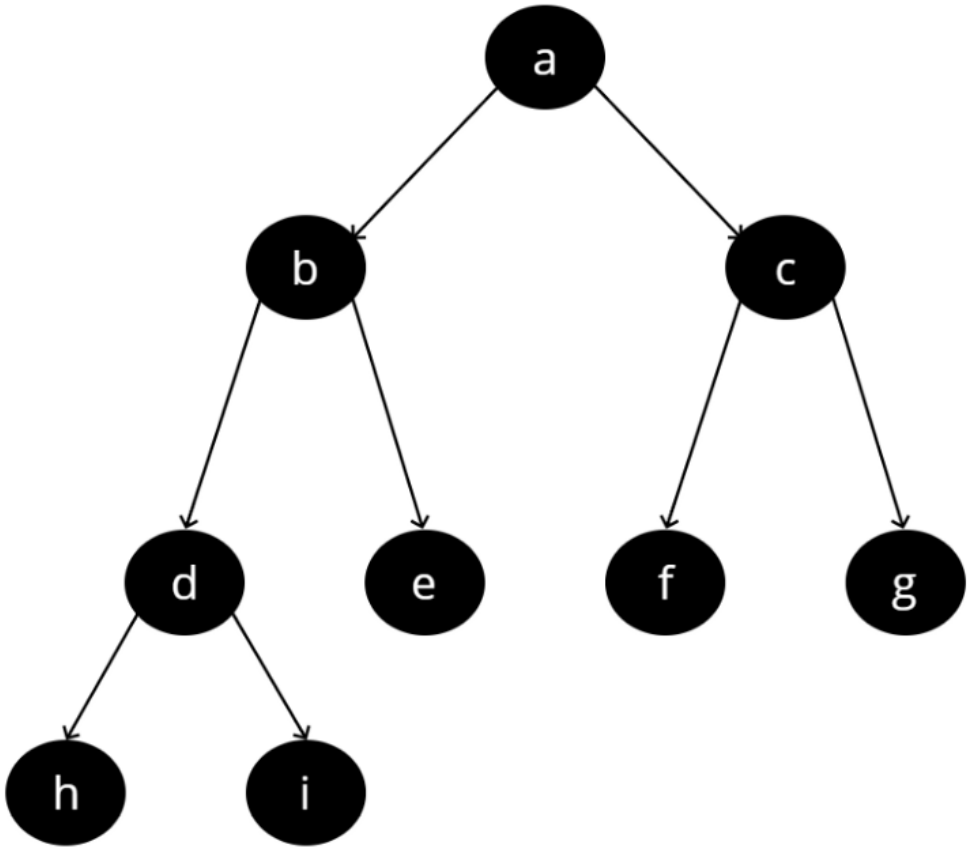
The `display` function is going to take the traversal `type` as an argument, with its value set to `IN` by default. The function will have a variable named `output` initially set to an empty string. We will then create the function that traversing algorithm use to go through the tree. Basically, when the algorithm visits a node, its `id` will get added to the `output` string.

```
  binaryTreeCreator = (rootId) => {
    const root = nodeCreator(rootId)
    return {
      root,
      display(type = 'IN') {
        let output = ''
        const func = node => {
          output += output.length === 0 ? node.id : ` => ${node.id}`
        }
        processes[type](this.root, func)
        return output
      }
    }
  }
```

And we are done! Now to test the traversal algorithms, lets first create a binary tree as shown below:

```
const binaryTree = binaryTreeCreator('a')
const b = binaryTree.root.addToLeft('b')
const c = binaryTree.root.addToRight('c')
const d = b.addToLeft('d')
const e = b.addToRight('e')
const f = c.addToLeft('f')
const g = c.addToRight('g')
const h = d.addToLeft('h')
const i = d.addToRight('i')
```

The binary tree that built from the above code will look something like this:



The traversal algorithm can be used in three ways. The first way, we don't pass any argument to the display method. That way, the algorithm will run the in-order traversal

as shown below.

```
console.log(binaryTree.display())
// output
h => d => i => b => e => a => f => c => g
```

If we pass the `type` argument to the display method and set its value to `IN`, we will get the same output.

The second way will require us to pass the `type` argument with its value set to `PRE`. The algorithm will then run the pre-order traversal as shown below:

```
console.log(binaryTree.display(type = 'PRE'))
// output
a => b => d => h => i => e => c => f => g
```

The final way will have us pass the `type` argument with a value of `POST`. The algorithm will then run the post-order traversal as shown below:

```
console.log(binaryTree.display(type = 'POST'))
// output
h => i => d => e => b => f => g => c => a
```

. . .

Conclusion

In this post, we took a look at some of the most common traversal algorithms in JavaScript.

We started with the breadth-first algorithm, where the algorithm starts with a node, then explores its neighbors, and then moves to the next level of nodes.

Then we took a look at the depth-first algorithm, where the algorithm starts with a node, explores each of its branches, then backtracks to other nodes.

Finally, we saw how to build a binary tree and the various ways we can traverse through it.

In order to keep this post short and to a point, I have only focused on traversal related algorithms in JavaScript. But even then, there are many other algorithms in JavaScript. You can check out this repository maintained by Oleksii Trekhleb to learn about other kinds of algorithms that you can code in JavaScript.

trekhleb/javascript-algorithms

📝 Algorithms and data structures implemented in JavaScript with explanations and links to further readings ...

github.com

Thanks for reading this long post! I hope this post helped you understand JavaScript Algorithms a little better.If you liked this post, then please do give me a few 👏 and feel free to comment below and chat! Cheers

. . .

Learn more

The Most In-Demand JavaScript Frameworks for Developers in 2019

2018 "State of JS" report is here- which frameworks will rule 2019?

blog.bitsrc.io

6 JavaScript User Authentication Libraries for 2019

"Build me a user-authentication in two weeks!" — Useful ways to get the job

done, quick and effective.

blog.bitsrc.io

5 Tools for Faster Development in React

5 tools to speed the development of your React application, focusing on components.

blog.bitsrc.io