Project Report

An Implementation Of:

Fractional Cascading Algorithm on N-ary Trees

Thinh Truong

13 December 2023

1 Introduction

Implementation of an algorithm helps us observe its efficiency and behavior in practice. With the goal to improve running time of naive algorithm on a N-ary tree problem, we explore fractional cascading algorithm. The theoretical information in this report fully refers to the work of M. Smid, see [1].

Consider an undirected connected graph $G = (V, E)$ with k vertices. Let d be the maximal degree of any vertex, assumed to be a constant. Each vertex v contains a sorted list $C(v)$ of real numbers. The type of query we want to solve is: Given a real number y and a connected subgraph (V', E') of G, locate y in all lists $C(v)$ for $v \in V'$.

Let n be the sum of the lengths of all lists $C(v)$. By performing a binary search in $C(v)$ for each $v \in V'$, we can solve the query in $O(|V'| \log n)$ time, and the amount of space used is $O(n + k)$.

Fractional cascading can improve this running time to $O(|V'| + \log n)$ while still using $O(n + k)$ space. The technique was initially introduced by Chazelle and Guibas, but the original version is deterministic and quite complicated. Hence, this report will focus on an easier approach of the algorithm which is the randomized fractional cascading technique, due to Kurt Mehlhorn in 1991. In this report, I will briefly explain each part of the randomized fractional cascading technique on N-ary tree, show the implementation along with the practical running time analysis.

2 The Implementation

I will briefly describe the fractional cascading algorithm in this section, thanks to the work of M. Smid [1]. Since the implementation will be performed with a target integer number on the sorted list $C(v)$ of integers, the explanation of the algorithm will also be updated accordingly.

In order to perform fractional cascading, we first need to generate augmented list $A(v)$ from each list $C(v)$. The list $A(v)$ contains $C(v)$, the minimum and maximum element, and some elements that are copied from the lists of the neighbors of v in G.

For each vertex $u \in V$, $A(u)$ will be a sorted sequence of integers. With each element x of $A(u)$, we store the following additional information:

1. pred: a pointer to the predecessor of x in $A(u)$
2. next: a pointer to the successor of x in $A(u)$
3. proper: a pointer to the smallest element in $C(u)$ that is at least equal to x
4. for each vertex v that is a neighbor of u there is at most one pointer to the list $A(v)$. This pointer is called a bridge from x in $A(u)$ to $A(v)$.

Steps to construct augmented list:

Initially, for each vertex u, we initialize $A(u)$ with the minimum and maximum element. For each neighboring pair u and v of vertices, we connect the occurrences of these elements in $A(u)$ and $A(v)$ by bridges.

We then perform repetition step. For each vertex $u \in V$ and each element $x \in C(u)$, do the following: Let $x_0, x_1 \in A(u)$ such that $x_0 < x < x_1$. call the following procedure:

$Insert(x, x_0, x_1, u)$:

Insert x into $A(u)$ between x_0 and x_1 and update the appropriate pred and next pointers. Store with x a bit indicating whether x belongs to $C(u)$ or $A(u) \backslash C(u)$.

for *all neighbors* v *of* u **do**
  determine y_0 and y_1 in $A(v)$ such that $y_0 < x <= y_1$;
  if $x = y_1$ **then**
    (x already occurs in $A(v)$)
    connect the occurrences of x in $A(u)$ and $A(v)$ by bridges
  else
    call $Insert(x, y_0, y_1, v)$ with probability $\frac{1}{2d}$
    and connect the occurrences of x in $A(u)$ and $A(v)$ by bridges
  end
end

We then update proper pointer of each element in $A(u)$ list for each vertex $u \in V$ in postprocessing step.

Steps to find the smallest element that is at least equal to the target:

Let u and v be two neighboring vertices and let target $y \in \mathbb{Z}$. Assume we have located y in $A(u)$. Then we locate y in $A(v)$ and $C(v)$ as follows:

1. let x be the smallest element in $A(u)$ that is at least equal to y
2. starting in x, follow next pointers until we reach the first element z that contains a bridge to $A(v)$
3. traverse the bridge to $A(v)$
4. starting in z in $A(v)$, follow pred pointers until we are at the smallest element x' that is at least equal to y
5. follow the proper pointer of x' to the element x'' of $C(v)$

x'' is the smallest element of $C(v)$ that is at least equal to y.

3 The Implementation

The implementation of the algorithm and its components will be described and explained throughout this section. Full source code can be found in *GitHub repository*.[1]

3.1 Node classes and functions

I use the `TreeNode` class to represent each node in the tree.

```
class TreeNode:
    def __init__(self, value, max_list_len: int):
        self.value = value
        self.parent : TreeNode = None
        self.children : list[TreeNode] = []
        self.node_list = sorted([random.randint(MIN_NUMBER_IN_LIST,
    MAX_NUMBER_IN_LIST) for _ in range(random.randint(1, random.randint(1,
    max_list_len)))])
        self.augmented_list= [ListNode(MIN_BOUNDARY, self)] + [ListNode(
    MAX_BOUNDARY, self)]     #initialize the list with boundary values

        # set prev/next pointers for augmented_list
        for i in range(len(self.augmented_list)):
            if i > 0:
                self.augmented_list[i].set_prev(self.augmented_list[i-1])
            if i < len(self.augmented_list) - 1:
                self.augmented_list[i].set_next(self.augmented_list[i+1])

```

[1] GitHub repository of the implementation. *https://github.com/ThinhTTruong/fractional-cascading*

```
    def add_child(self, child_node: "TreeNode"):
        self.children.append(child_node)

    # generate augmented list without having proper pointer set
    def generate_augmented_list(self):
        # connect boundaries by bridges to neighbors
        for child in self.children:
            self.augmented_list[0].add_bridges(child.augmented_list[0])
            child.augmented_list[0].add_bridges(self.augmented_list[0])
            self.augmented_list[-1].add_bridges(child.augmented_list[-1])
            child.augmented_list[-1].add_bridges(self.augmented_list[-1])
        if self.parent:
            self.augmented_list[0].add_bridges(self.parent.augmented_list
[0])
            self.parent.augmented_list[0].add_bridges(self.augmented_list
[0])
            self.augmented_list[-1].add_bridges(self.parent.augmented_list
[-1])
            self.parent.augmented_list[-1].add_bridges(self.augmented_list
[-1])
        # perform insert on neighbors
        for value in self.node_list:
            list_node = insert_own_list(self.augmented_list, value, self)
            if list_node != None:
                insert_recursive(list_node, self)

    def post_processing(self):
        # set proper pointers
        start, end = 0, 0
        for i in range(len(self.augmented_list)):
            if self.augmented_list[i].tree_node:
                end = i
                current_proper = self.augmented_list[i]

                while start <= end:
                    self.augmented_list[start].set_proper(i, current_proper)
                    start += 1

        # add boundary values on node_list for naive algorithm
        self.node_list.insert(0, MIN_BOUNDARY)
        self.node_list.append(MAX_BOUNDARY)

    def print_augmented_list(self):
        values = list(map(lambda node: node.value, self.augmented_list))
        return values
```

Source [2]: fractional-cascading/src/node.py

Each `TreeNode` v contains `parent` and `child` attributes. It also stores `node_list` which represents a sorted list $C(v)$ of real numbers, and `augmented_list` which represents the augmented list $A(v)$. The `generate_augmented_list()` function can be called to generate $A(v)$ list from $C(v)$ list, and `post_processing()` can be used to set the proper pointers and after $A(v)$ list is generated.

Each element in $A(v)$ list is represented by a `ListNode`.

```python
class ListNode:
    def __init__(self, value, tree_node: TreeNode = None):
        self.value = value
        self.tree_node = tree_node
        self.bridges : list["ListNode"] = []
        self.prev = None
        self.next = None
        self.proper = None

    def add_bridges(self, list_node: "ListNode"):
        self.bridges.append(list_node)

    def set_prev(self, list_node: "ListNode"):
        self.prev = list_node

    def set_next(self, list_node: "ListNode"):
        self.next = list_node

    # Save proper as (index, list_node)
    def set_proper(self, index: int, list_node: "ListNode"):
        self.proper = (index, list_node)
```

Source: fractional-cascading/src/node.py

Each element `ListNode` in $A(v)$ stores the pred, next, proper and bridge pointers. Using `TreeNode` and `ListNode`, with the help of helper functions to perform *Steps to construct augmented list* from Section 2, we can create a balanced N-ary tree with `create_whole_tree()` which takes the size of the tree and maximum degree of a node as parameters.

[2]Path to the source file in the implementation's GitHub repository.

3.2 Path functions

We want to run the algorithm using three different paths. The first one is a random path from the root to a leaf. The second one is a random path from a leaf going up to a random node in the middle of the tree, and then going down to a random leaf. The third path is the first path but repeated up and down multiple times, producing a very long path. Throughout the paper, these paths will be referred as path 1, path 2, and path 3 respectively. These paths are generated by the following functions.

```python
# path 1
def root_to_leaf_path(root: TreeNode, node_degree: int):
    if not root:
        return []

    path = []
    current = root

    while current:
        path.append(current)

        if not current.children:
            break

        child_index = random.randint(0, node_degree - 2)
        current = current.children[child_index]

    return path

def get_leaf_nodes(root: TreeNode):
    leaf_nodes: list[TreeNode] = []

    def dfs(node: TreeNode):
        if not node:
            return
        if not node.children:
            leaf_nodes.append(node)
        for child in node.children:
            dfs(child)

    dfs(root)
    return leaf_nodes

# path 2
def leaf_node_leaf_path(root: TreeNode, height: int):
    leaf_nodes = get_leaf_nodes(root)
```

```
37
38      if not leaf_nodes:
39          return []
40
41      # Select a random starting leaf node
42      start_leaf = random.choice(leaf_nodes)
43
44      path = []
45      current = start_leaf
46      path.append(current)
47
48      mid_point = height//2
49
50      # Path going up
51      for _ in range(mid_point):
52          if current.parent:
53              current = current.parent
54              path.append(current)
55
56      # Path going down
57      for _ in range(mid_point):
58          if current.children:
59              current = random.choice(current.children)
60              path.append(current)
61
62      return path
63
64  # path 3
65  def up_and_down_path(root: TreeNode, node_degree: int, repetition_time: int
        ):
66      path = root_to_leaf_path(root, node_degree)
67      new_path = []
68
69      for node in path:
70          new_path.append(node)
71
72      for node in reversed(path[:-1]):
73          new_path.append(node)
74      for i in range(repetition_time -1):
75          for node in path[1:]:
76              new_path.append(node)
77
78          for node in reversed(path[:-1]):
79              new_path.append(node)
80      return new_path
```

Source: fractional-cascading/src/paths.py

Given a N-ary tree of size n where n is even, path 1 will produce a path of $n/2$ nodes while path 2 will produce a path of $(n/2) + 1$ nodes. For example, with $n = 4$, the path produced by path 1 has length of 4 while the path produced by path 2 has length of 5. When n is odd, the length of the paths produced by the two techniques will be the same.

For path 3, if the original path is repeated x times, then the length of the path is $xn - (x - 1)$. For example, when $n = 5$, and path 1 is repeated 10 times to produce path 3, then the length of path 3 is 41.

3.3 Algorithm functions

Given a target integer and a path of nodes, for any node v, we check if the list $C(v)$ contains the target, and return the smallest value in the list that is greater than or equal to the target.

When performing naive algorithm on the path, we simply do binary search on `node_list` of each node independently.

```python
def naive_algorithm(path: list[TreeNode], target: int):
    result = []
    for tree_node in path:
        # binary seach
        search_result = binary_search_naive(tree_node.node_list, target)
        result.append(search_result)
    return result

def binary_search_naive(lst, target):
    left, right = 0, len(lst) - 1

    while left < right:
        mid = (left + right) // 2

        if lst[mid] == target:
            return target
        elif lst[mid] < target:
            left = mid + 1
        else:
            right = mid

    return lst[right]
```

Source: fractional-cascading/src/algorithms.py

For fractional cascading, we perform binary search on `node_list` of the first node in the path. Then we follow *Steps to find the smallest element that is at least equal to the target* from Section 2 by using `helper()` function to get the smallest element that is at least equal to the target in `node_list` of subsequent nodes.

```python
def fractional_cascading(path: list[TreeNode], target: int):
    result = []

    # Binary search on first tree node
    current_node = path[0]
    (first_index, first_value) = binary_search_fc(current_node.augmented_list, target)
    if first_value == target and current_node.augmented_list[first_index].tree_node:
        result.append(first_value)
    else:
        result.append(current_node.augmented_list[first_index].proper[1].value)

    cur_index = first_index
    # fractional cascading on other nodes
    for i in range(len(path)-1):
        cur_index, res = helper(cur_index, path[i], path[i+1], target)
        result.append(res)

    return result

def binary_search_fc(lst: list[ListNode], target: int):
    #return (index, value)
    left, right = 0, len(lst) - 1

    while left < right:
        mid = (left + right) // 2

        if lst[mid].value == target:
            return (mid, target)
        elif lst[mid].value < target:
            left = mid + 1
        else:
            right = mid
    return (right, lst[right].value)

```

```
def helper(cur_index: int, cur_node: TreeNode, next_node: TreeNode, target: int):
    cur_list = cur_node.augmented_list
    current = cur_list[cur_index]
    found = False
    value = 0
    while current:
        if current.bridges:
            for bridge in current.bridges:
                if bridge.tree_node == next_node: #step 2
                    current = bridge #step 3
                    found = True
                    break
        if found:
            break
        else:
            current = current.next
    value = current.proper[1].value
    while current and current.prev.value >= target: #step 4
        current = current.prev
        if current.proper:
            value = current.proper[1].value

    index = next_node.augmented_list.index(current)

    if current.proper and value == target:
        return (index, target) # return index of current element in A(v), and matching value in C(v)

    return (index, value) # return index of element in A(v), and smallest value in C(v) that is greater target
```

Source: fractional-cascading/src/algorithms.py

3.4 Test functions

Given the tree size n and maximum degree d, we generate a tree with height of n. Each node v in the tree has list $C(v)$ of length n, where the minimum value in the list is 1 and the maximum value in the list is n^2. For each n, we fix the value of d and run the query n times and get the average running time. We then plot the running time of the two algorithms in a graph for comparison. The `runtime_test()` function is expressed as follow:

```
def runtime_test(max_degree: int, max_n: int, path_option: int):
    n_values = list(range(1, max_n, 2))
```

```
    naive_runtimes = []
    fc_runtimes = []
    for n in n_values:
        naive_runtime, fc_runtime = get_runtime(n, max_degree, path_option)
        naive_runtimes.append(naive_runtime)
        fc_runtimes.append(fc_runtime)
    plt.plot(n_values, naive_runtimes, marker='o', linestyle='-', label='Naive')
    plt.plot(n_values, fc_runtimes, marker='o', linestyle='-', label='Fractional Cascading')

    plt.title('Runtime Comparison of Naive and Fractional Cascading Algorithm')
    plt.xticks(range(min(n_values), math.ceil(max(n_values))+1, 2))
    plt.xlabel('n')
    plt.ylabel('Runtime (seconds)')
    plt.legend()
    plt.grid(True)
    plt.show()
```

Source: fractional-cascading/src/tests.py

4 Running time analysis

4.1 Tests on increasing n

We fix the value of d and perform the tests on the path 1 and path 2. Results are captured in the following graphs:

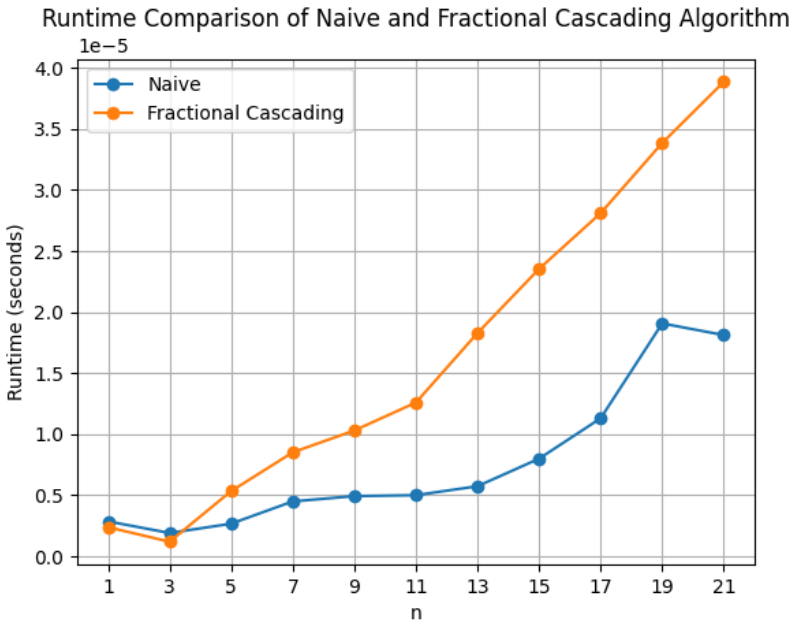

Figure 1: Running time of path 1 on $d = 3$.

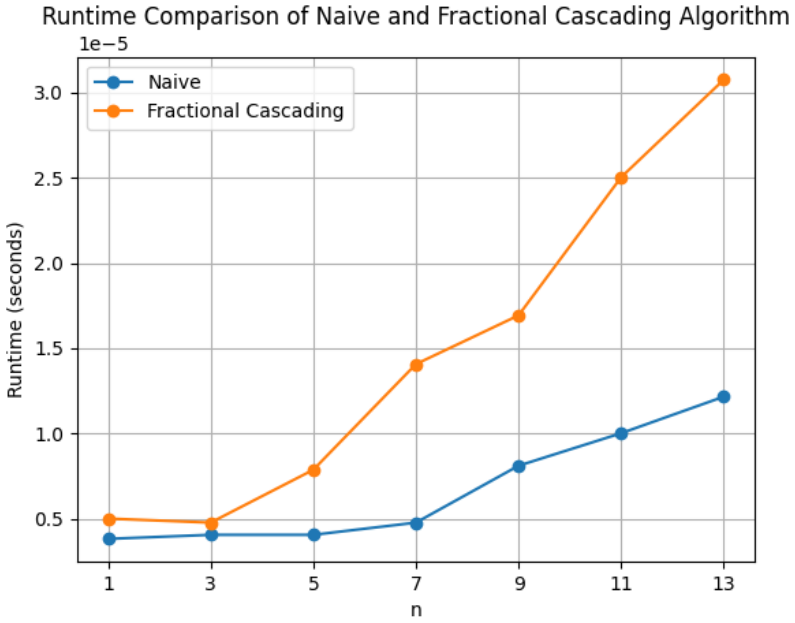

Figure 2: Running time of path 1 on $d = 4$.

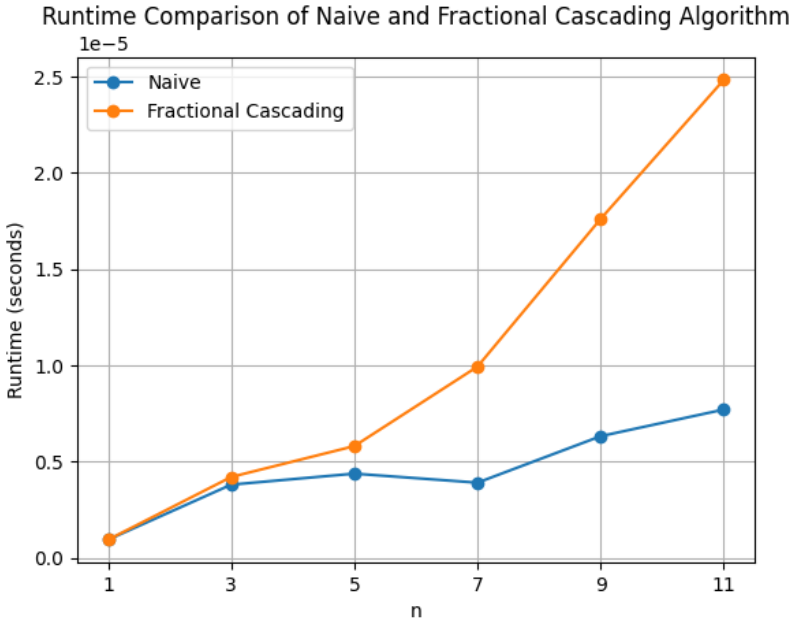

Figure 3: Running time of path 1 on $d = 5$.

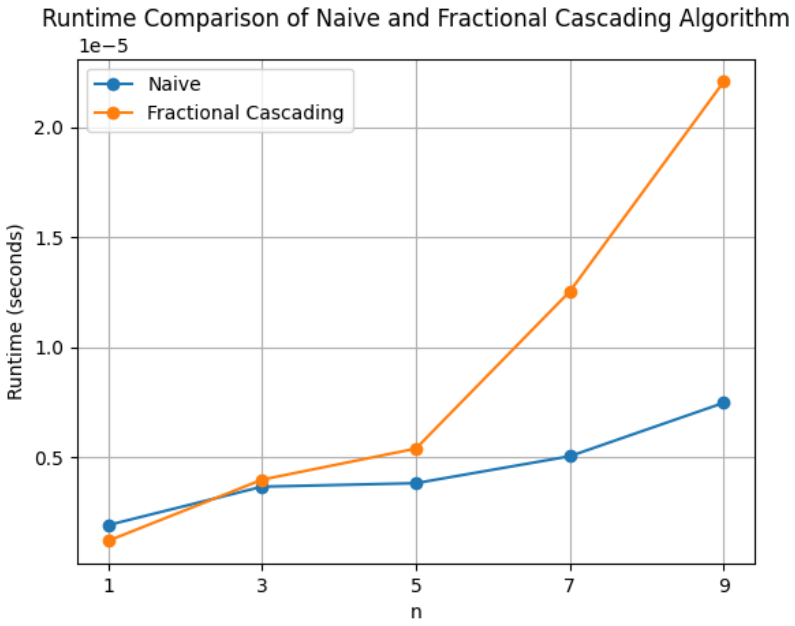

Figure 4: Running time of path 1 on $d = 6$.

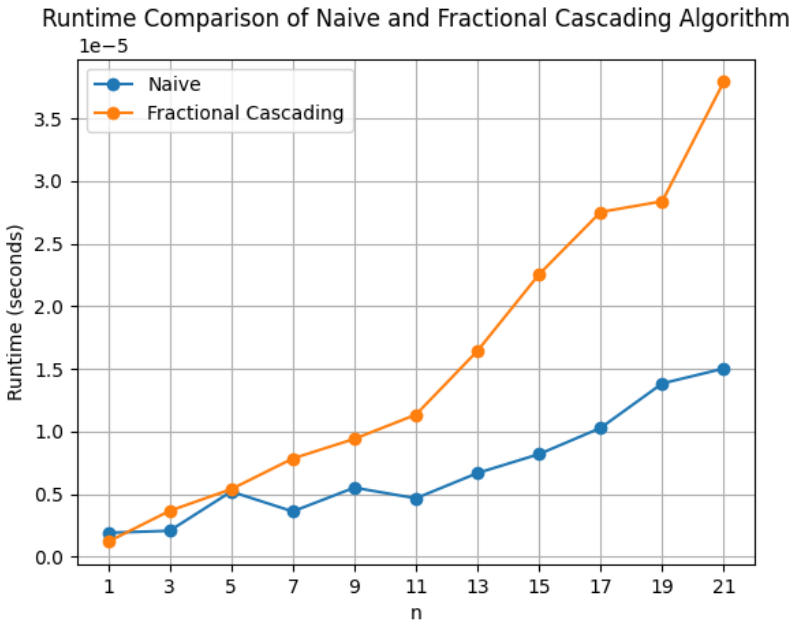

Figure 5: Running time of path 2 on $d = 3$.

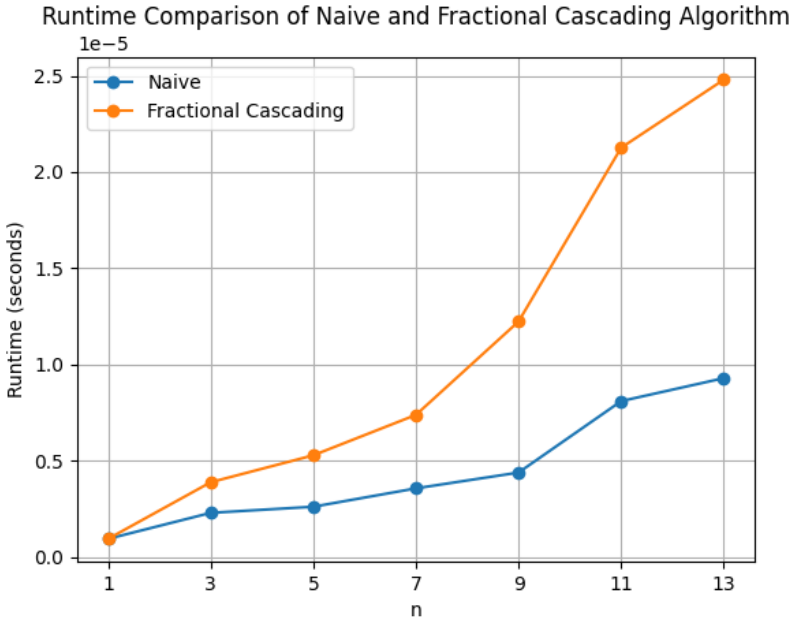

Figure 6: Running time of path 2 on $d = 4$.

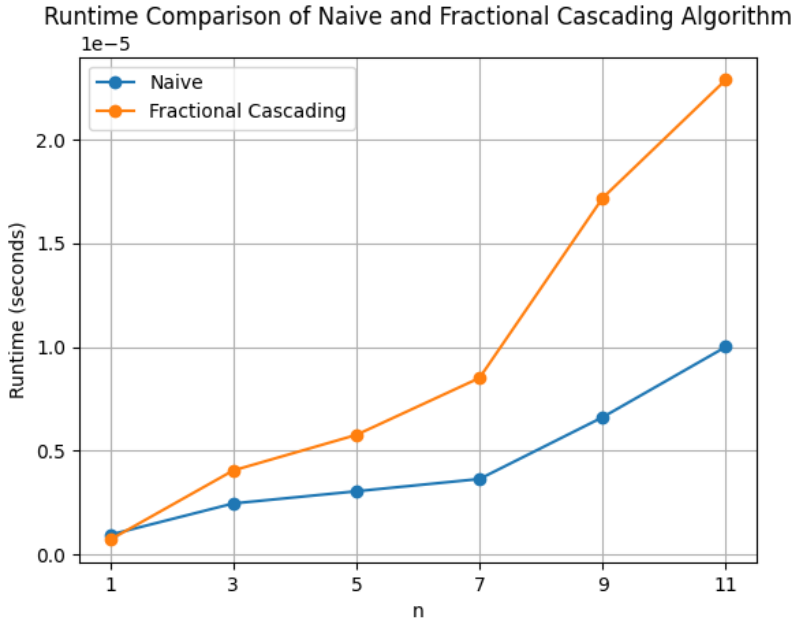

Figure 7: Running time of path 2 on $d = 5$.

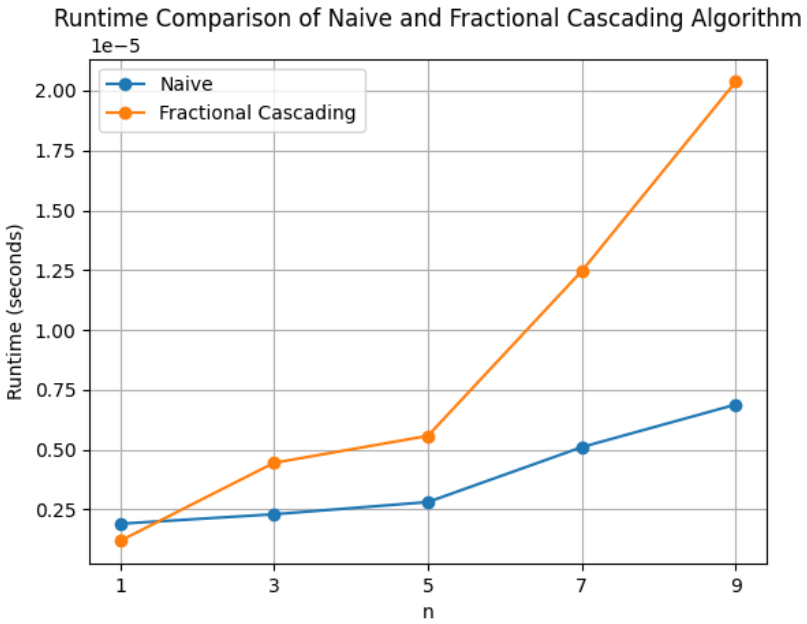

Figure 8: Running time of path 2 on $d = 6$.

When $n = 1$, the tree is a single vertex v with $C(v) = A(v)$. It is expected that the runtime of both algorithms should be the same because we do binary search on the same list. However, the actual running times slightly vary between the two algorithms, and this could be due to the computing power at the time of measurement.

We observe that the graphs of fractional cascading always have higher slope than the graph of naive algorithm. As n gets large, the running time of fractional cascading increases much faster than the running time of naive algorithm. This is not what we expected because in a path, which is a subgraph (V', E') of the N-ary tree (V, E), naive algorithm runs in $O(|V'| \log n)$ time while fractional cascading has running time of $O(|V'| + \log n)$. Given limited memory space, the maximum number of nodes that I can get to is just over 2 million nodes, which is equivalent to $n = 21$ when $d = 3$.

4.2 Tests on increasing $|V'|$

As the memory space is limited and it is hard to continue increasing n, we attempt to increase $|V'|$ by increasing the length of the path. We fix $d = 3$ and vary n from 1 to 9, and test the running time on long paths.

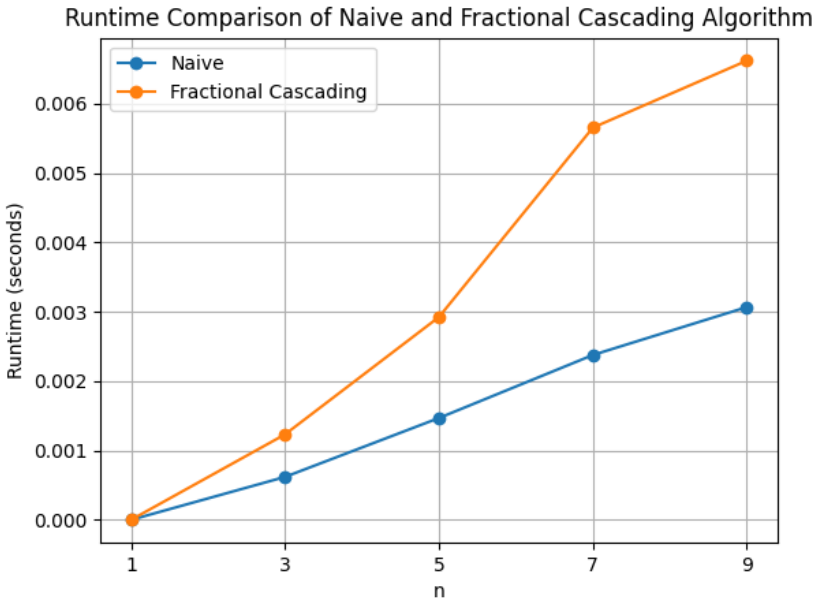

Figure 9: Running time of path 3 where the original path is repeated $1,000$ times.

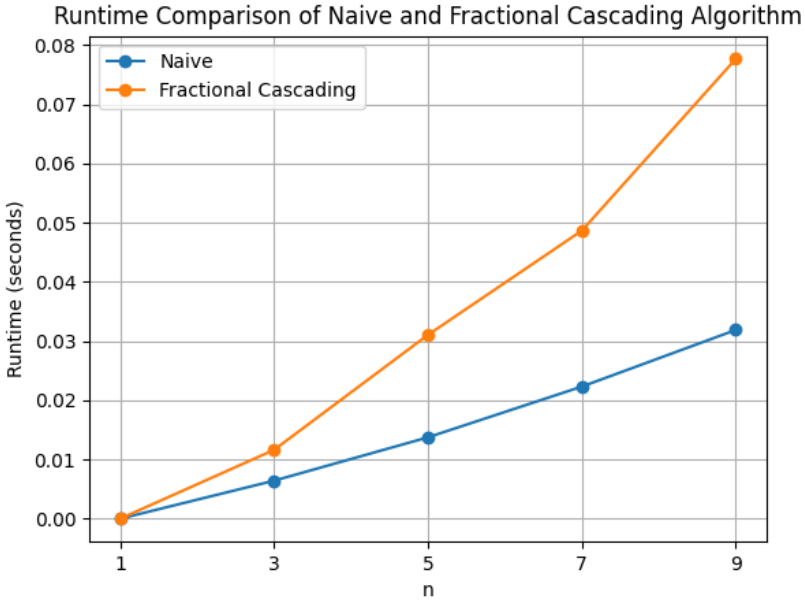

Figure 10: Running time of path 3 where the original path is repeated $10,000$ times.

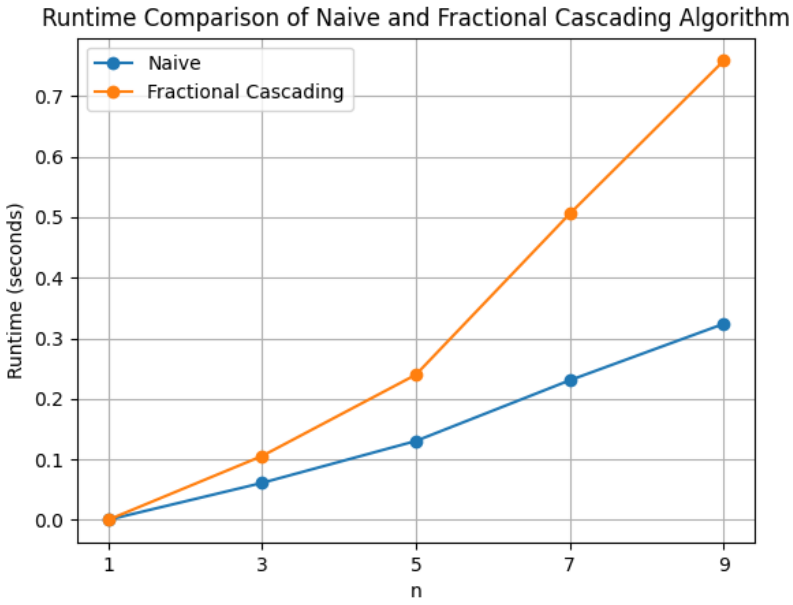

Figure 11: Running time of path 3 where the original path is repeated $100,000$ times.

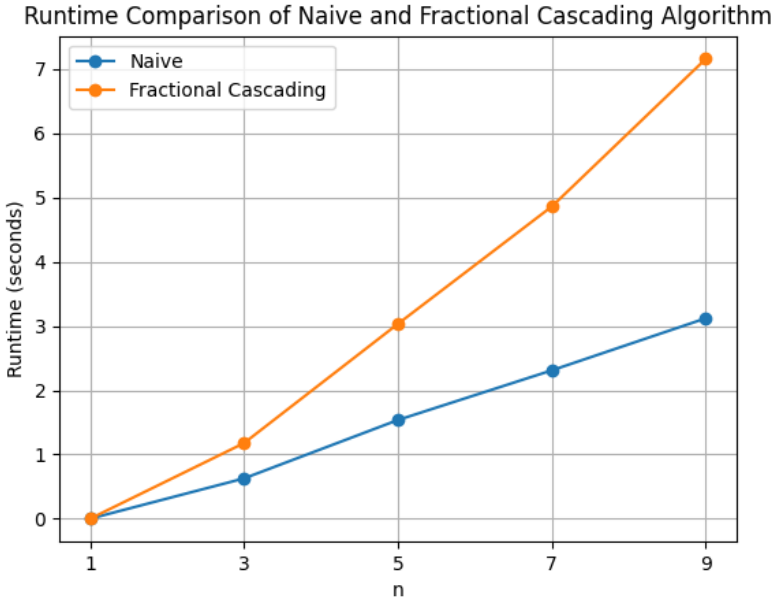

Figure 12: Running time of path 3 where the original path is repeated $1,000,000$ times.

Similar to when we increase n on a short path, we observe that the slope of the fractional cascading graph is larger than that of the naive graph. The graphs of the algorithms seem to be identical between iterations, only the running time increases by 10 when we repeat the path 10 more times.

5 Conclusion

Based on the results, the fractional cascading algorithm does not work as expected in practice. As the naive algorithm runs in $O(|V'| \log n)$ time and fractional cascading has running time of $O(|V'| + \log n)$, I attempt to increase n, and increase $|V'$, and both result in fractional cascading takes longer to run than the naive way. This could suggest that the base value of fractional cascading in big O notation is significantly larger than that of naive algorithm, and we did not reach that threshold of n and $|V'|$ to see the impact of fractional cascading algorithm on running time. Further tests may need to be performed where we can test on a very large n and $|V'$ so that we can assess the performance of fractional cascading.

References

[1] M. Smid. *Lecture Notes on Fractional Cascading.* https://people.scs.carleton.ca/~michiel/lecturenotes/fractionalcascading.pdf