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.

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

The implementation of the algorithm and its components will be described and explained throughout this section.

2.1 Node classes and functions

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

```python
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])

    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])
```

```python
            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 [1]: 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"] = []
```

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

```
        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, 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.

2.2 Path functions

We want to the the algorithm using two 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. Throughout the paper, the root to leaf path will be referred as path 1, and the other path will be referred as path 2. These paths are generated by the following functions.

```
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

```

```python
        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

def leaf_node_leaf_path(root: TreeNode, height: int):
    leaf_nodes = get_leaf_nodes(root)

    if not leaf_nodes:
        return []

    # Select a random starting leaf node
    start_leaf = random.choice(leaf_nodes)

    path = []
    current = start_leaf
    path.append(current)

    mid_point = height//2

    # Path going up
    for _ in range(mid_point):
        if current.parent:
            current = current.parent
            path.append(current)

    # Path going down
    for _ in range(mid_point):
        if current.children:
            current = random.choice(current.children)
            path.append(current)
```

```python

    return 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.

2.3 Algorithm functions

```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

```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
```

```python
        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

2.4 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:

```python
def runtime_test(max_degree: int, max_n: 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)
        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

3 Running time analysis

We fix the value of d and perform the tests on the two paths. Results are captured in the following graphs:

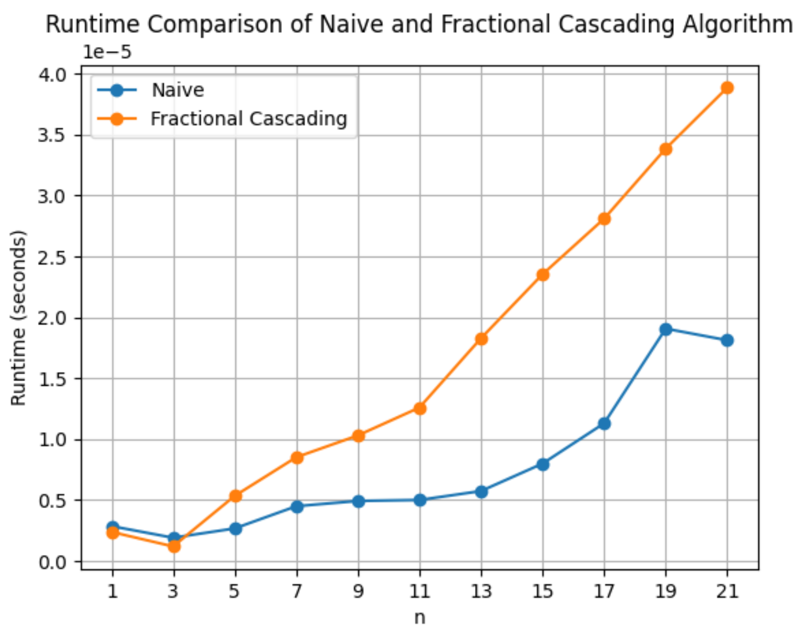

Figure 1: Running time of path 1 on $d = 3$.

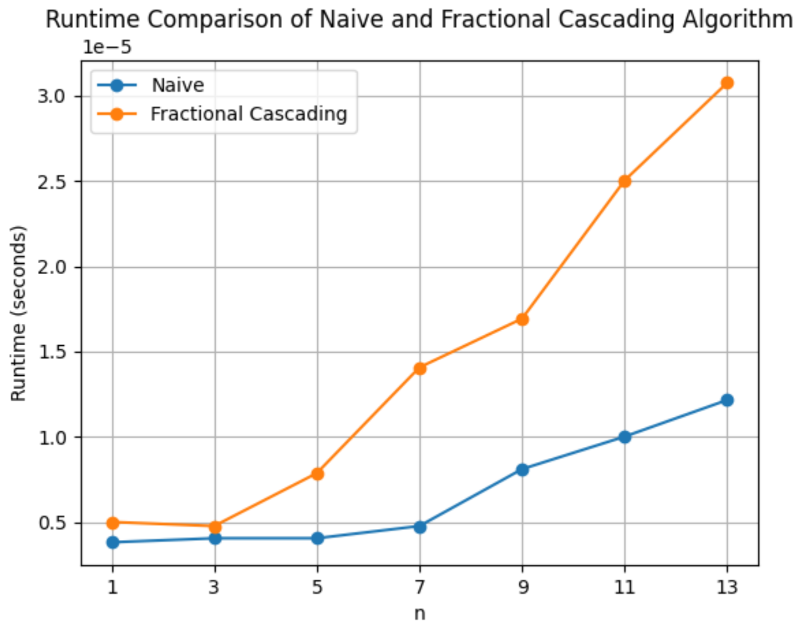

Figure 2: Running time of path 1 on $d = 4$.

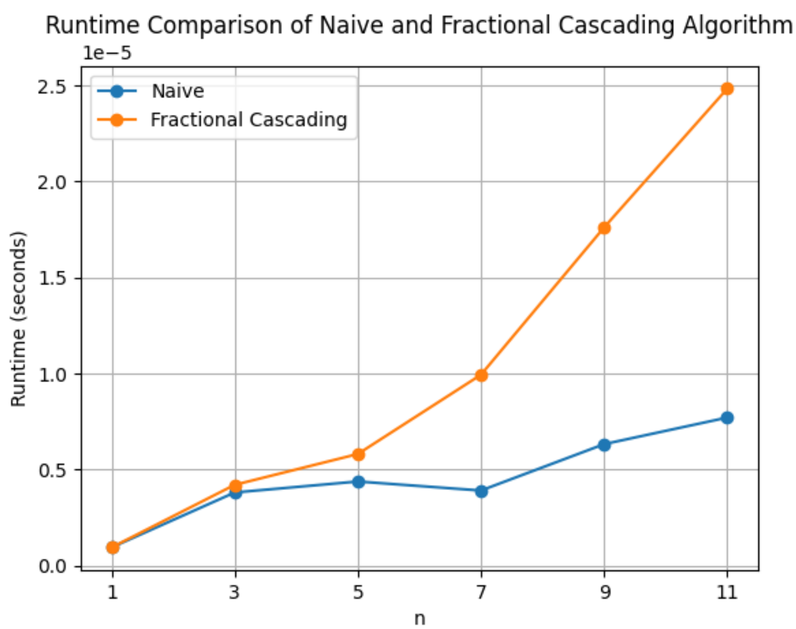

Figure 3: Running time of path 1 on $d = 5$.

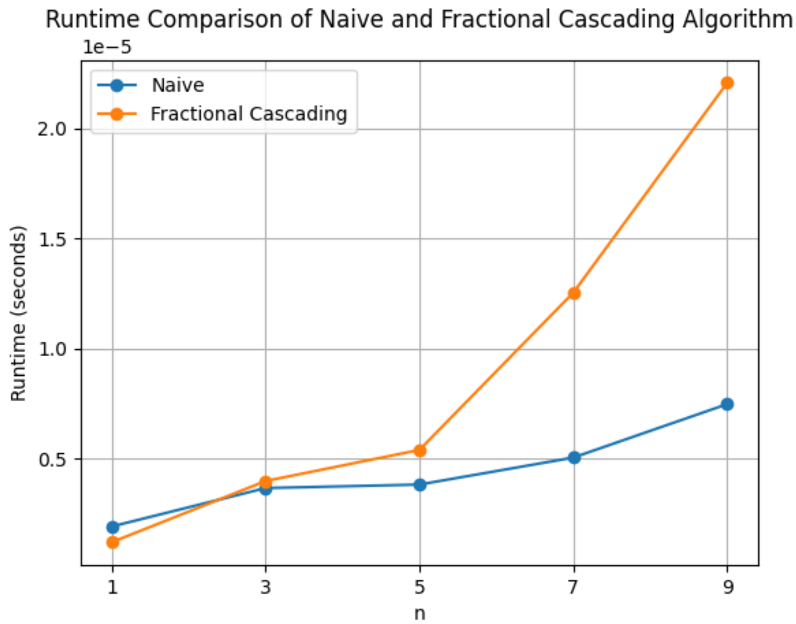

Figure 4: Running time of path 1 on $d = 6$.

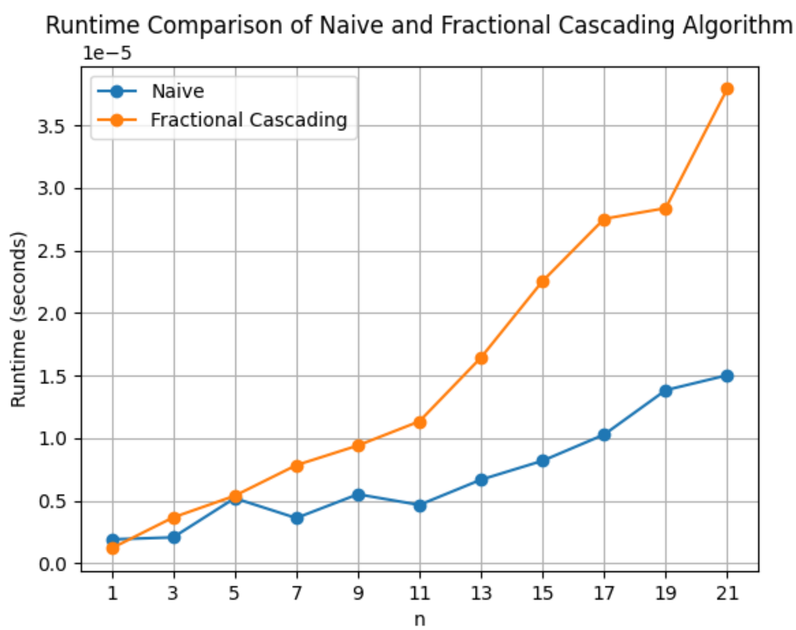

Figure 5: Running time of path 2 on $d = 3$.

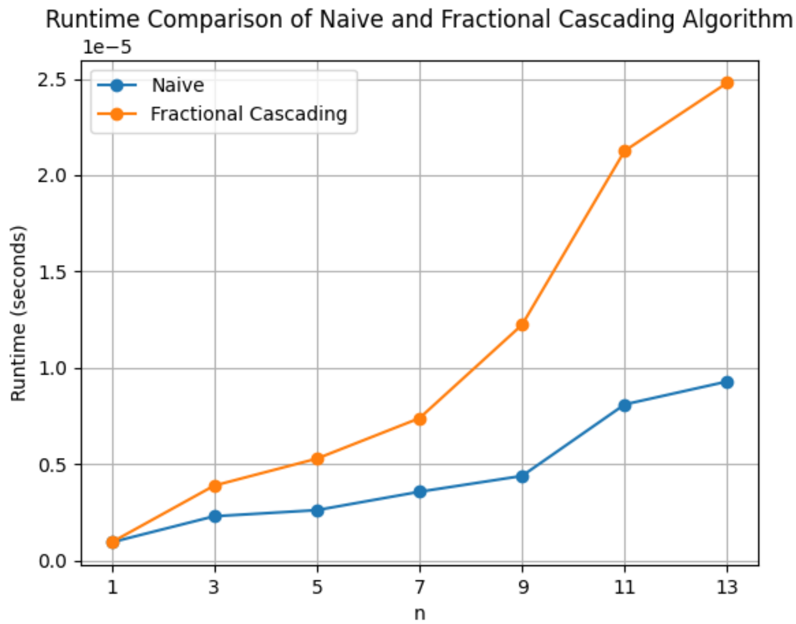

Figure 6: Running time of path 2 on $d = 4$.

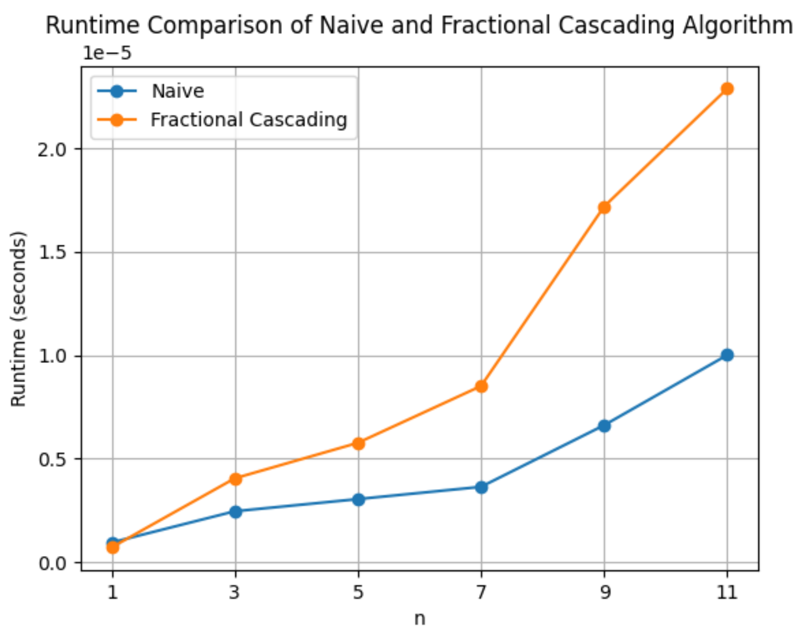

Figure 7: Running time of path 2 on $d = 5$.

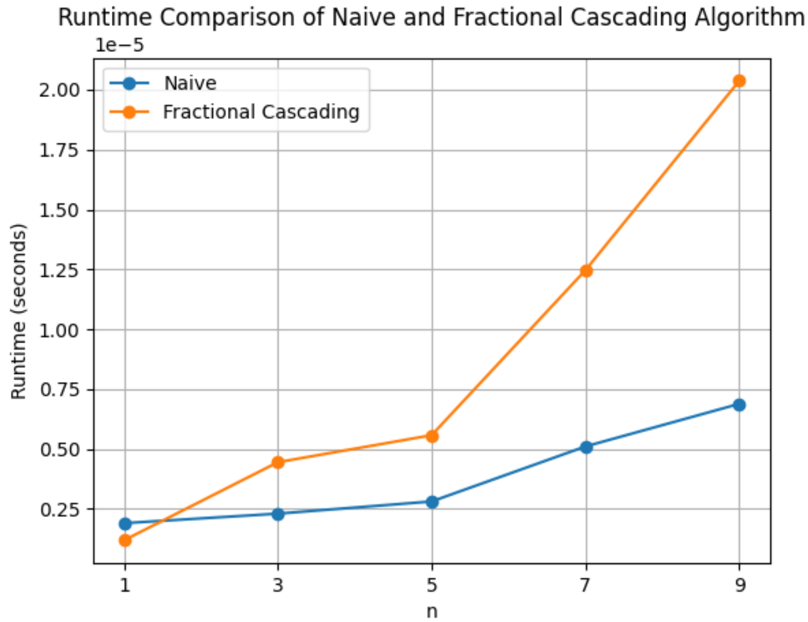

Figure 8: Running time of path 2 on $d = 6$.

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 computing power, the maximum number of nodes that I can get to is just over 2 million nodes, which is equivalent to $n = 21$ when $d = 3$.

The question is as n gets larger, will there ever be the point that fractional cascading outperforms the naive algorithm. Based on the graphs that we acquired, it is unlikely that growth rate of fractional cascading running time seems so much higher than that of the naive algorithm.

4 Conclusion

The algorithm did not work as expected.