Generating all ideals of an arborescence.

This is a description of an algorithm for generating all ideals of an arborescence, where $\mathcal{I}$ is the number of ideals in the tree, with a constant amortized time of $O(1)$ per transition from $ideal_n$ to $ideal_{n+1}$ and an overall complexity of $O(\mathcal{I})$. It is believed this algorithm presents a novel approach not written about previously. The algorithm is short and simple to understand consisting of a pop, jump and push per ideal.

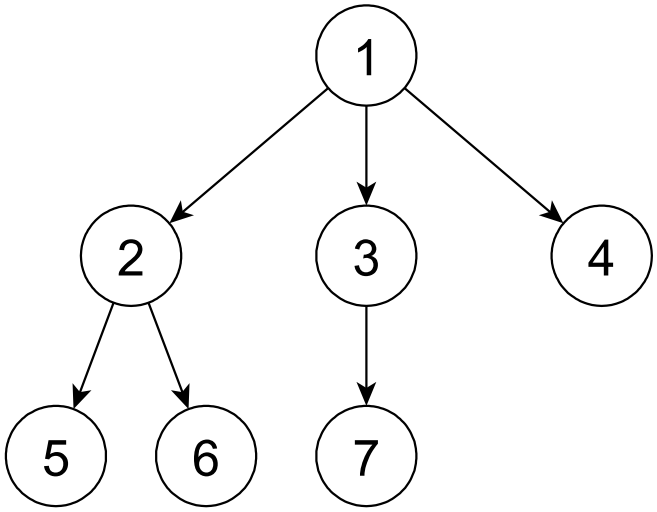
Algorithm 1 (Pop, jump and push ideals generation). Given an arborescence whos nodes are the sequence $(0, \ldots, n-1)$ when arranged in preorder, this algorithm visits all tuples $(s_1, \ldots, s_n)$ where $s_p \leq s_c$ whenever p is the parent of c. Rightmost subtrees are pushed or popped between one visit and the next. A single array of jump pointers $[j_1, j_2, \ldots, j_n]$ indicates the first node of the next right subtree or n if one doesn't exist.

Steps:

while s:
    visit s
    $i = j[s.pop()]$
    $s.push([i..n))$

How it works...

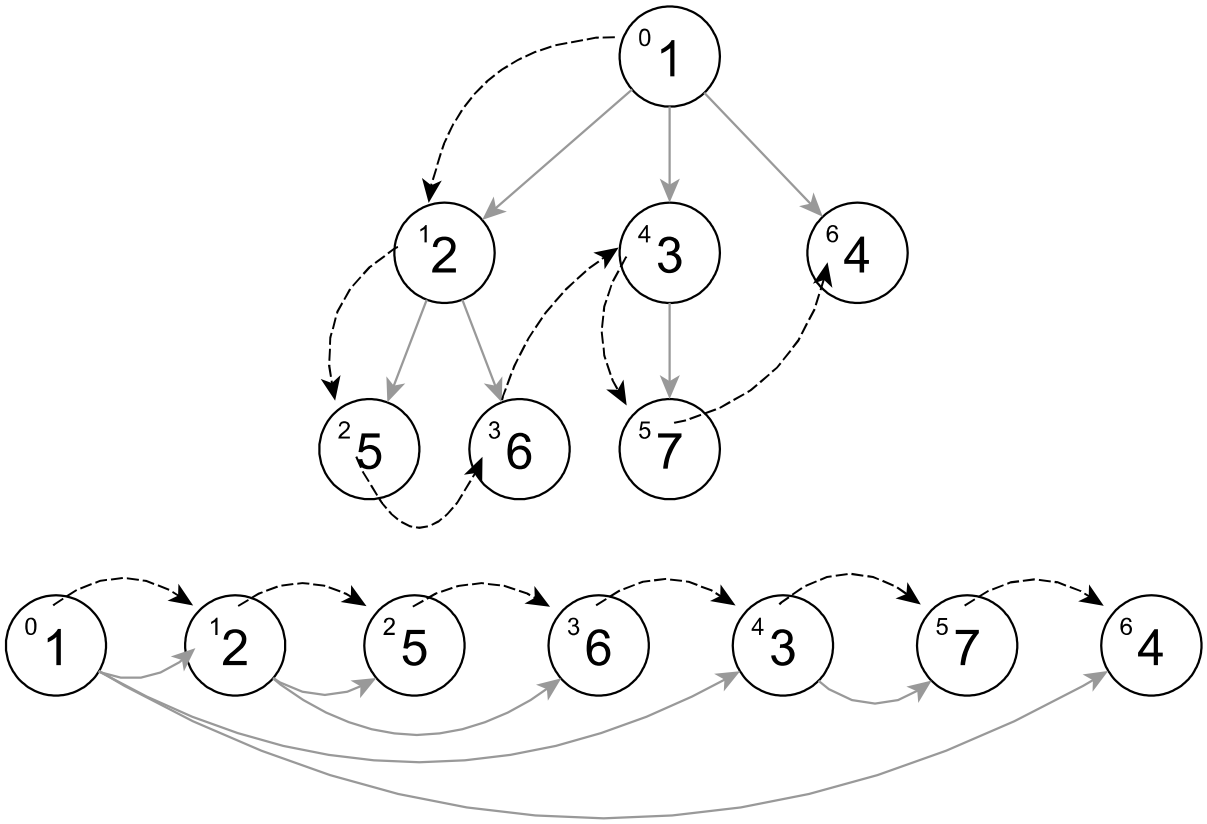
a) Begin with a tree:



Labels

$$root = 1$$
$$parents = [\backslash, 1, 1, 1, 2, 2, 3]$$
$$children = [1, 2, 3, 4, 5, 6, 7]$$

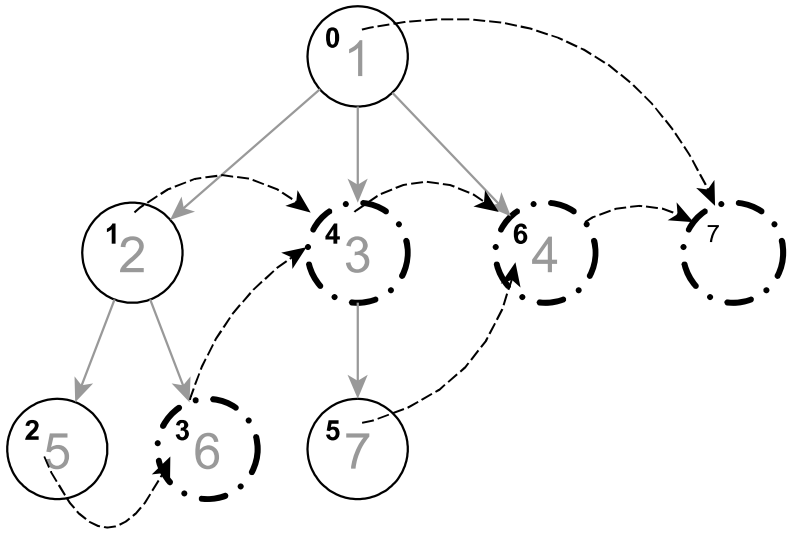
b) Sort the tree's node labels/objects/pointers to pre-order.



Pre-Order

$$root = 1$$
$$parents = [\backslash, 1, 2, 2, 1, 3, 1]$$
$$children = [1, 2, 5, 6, 3, 7, 4]$$
$$indices = [0, 1, 2, 3, 4, 5, 6]$$

c) Get the pre-order index of the first node of the next right subtree for each node. Use 'n' for the missing rightmost at the end.



Next Subtree Root

$$
\begin{aligned}
root &= 1 \\
parents &= [\diagdown, 1, 2, 2, 1, 3, 1] \\
children &= [1, 2, 5, 6, 3, 7, 4] \\
indices &= [0, 1, 2, 3, 4, 5, 6] \\
jump\_indices &= [7, 4, 3, 4, 6, 6, 7]
\end{aligned}
$$

d) Consume the end of the indices and replenish from the root of the next subtree to the last descendant of the tree.

$f(j = jump\_indices)$
  $n = |j|$
  $s = [0..n-1]$
  While s:
    visit s
    $i = j[s.pop()]$
    $s.push([i..n))$

$visit$	$\_ = pop()$	$i = j[\_]$	$[i..n)$	s_{result}
$0, 1, 2, 3, 4, 5, 6$	6	7	$\{\}$	$0, 1, 2, 3, 4, 5$
$0, 1, 2, 3, 4, 5$	5	6	$\{6\}$	$0, 1, 2, 3, 4, 6$
$0, 1, 2, 3, 4, 6$	6	7	$\{\}$	$0, 1, 2, 3, 4$
$0, 1, 2, 3, 4$	4	6	$\{6\}$	$0, 1, 2, 3, 6$
$0, 1, 2, 3, 6$	6	7	$\{\}$	$0, 1, 2, 3$
$0, 1, 2, 3$	3	4	$\{4, 5, 6\}$	$0, 1, 2, 4, 5, 6$
$0, 1, 2, 4, 5, 6$	6	7	$\{\}$	$0, 1, 2, 4, 5$
$0, 1, 2, 4, 5$	5	6	$\{6\}$	$0, 1, 2, 4, 6$
$0, 1, 2, 4, 6$	6	7	$\{\}$	$0, 1, 2, 4$
$0, 1, 2, 4$	4	6	$\{6\}$	$0, 1, 2, 6$
$0, 1, 2, 6$	6	7	$\{\}$	$0, 1, 2$
$0, 1, 2$	2	3	$\{3, 4, 5, 6\}$	$0, 1, 3, 4, 5, 6$
$0, 1, 3, 4, 5, 6$	6	7	$\{\}$	$0, 1, 3, 4, 5$
$0, 1, 3, 4, 5$	5	6	$\{6\}$	$0, 1, 3, 4, 6$
$0, 1, 3, 4, 6$	6	7	$\{\}$	$0, 1, 3, 4$
$0, 1, 3, 4$	4	6	$\{6\}$	$0, 1, 3, 6$
$0, 1, 3, 6$	6	7	$\{\}$	$0, 1, 3$
$0, 1, 3$	3	4	$\{4, 5, 6\}$	$0, 1, 4, 5, 6$
$0, 1, 4, 5, 6$	6	7	$\{\}$	$0, 1, 4, 5$
$0, 1, 4, 5$	5	6	$\{6\}$	$0, 1, 4, 6$
$0, 1, 4, 6$	6	7	$\{\}$	$0, 1, 4$
$0, 1, 4$	4	6	$\{6\}$	$0, 1, 6$
$0, 1, 6$	6	7	$\{\}$	$0, 1$
$0, 1$	1	4	$\{4, 5, 6\}$	$0, 4, 5, 6$
$0, 4, 5, 6$	6	7	$\{\}$	$0, 4, 5$
$0, 4, 5$	5	6	$\{6\}$	$0, 4, 6$
$0, 4, 6$	6	7	$\{\}$	$0, 4$
$0, 4$	4	6	$\{6\}$	$0, 6$
$0, 6$	6	7	$\{\}$	0
0	0	7	$\{\}$	$\{\}$

The visited indices can be used to lookup the original node labels/objects/pointers from step b without further processing.

Another option is to manipulate values alongside the indices sequence.

Let each node be given a weight in $W = [w_0, \ldots, w_{n-1}]$ and have $w = \sum W$. Then

while s:
    visit s, w
    $i = s.pop()$
    $w = w - W[i]$
    $i = j[i]$

$s.push([i..n))$
$w = w + sum(W[i..n))$

would generate a weight for each ideal without requiring further iteration over the visited indices sequence.

e) terminate

Analysis

Overall Growth

The primary constraint in the time complexity of this problem is directly related to the combinatorial explosion in the number of ideals ($\mathcal{I}$) as the number of nodes (n) increases. The value of $\mathcal{I}$ is directly related to the structure of the tree.

Regardless of the structure the tree the total number of elements pushed, including populating the initial array, is equal to the number of elements popped which is equal to $\mathcal{I}$.

$\mathcal{I}$ for a rooted tree structure

- when wide and shallow is 2^{n-1}.
- when narrow and deep is n.
- in the balanced tree case is given in A004019 and is

$$\mathcal{I}(n+1) = 1 + 2\mathcal{I}(n) + \mathcal{I}(n)^2 = (1 + \mathcal{I}(n))^2$$

Worst Case (wide and shallow)

In the worst case this algorithm is equivalent to counting in binary in descending order while tracking the last '1' bit. The last '1' bit is flipped off and the remaining bits are flipped on which is equivalent to popping the last element of our indices sequence array and pushing the remaining elements with a $\geq$ index.

This means the number of elements pushed in the worst case tree is equal to the number of bits *enabled* while counting in binary which is $2^{n-1} = \mathcal{I}$ and the count of push instances is 2^{n-2}. As such the average number of elements pushed per push instance is 2 and the overall is amortized to 1.

This results in a worst case constant amortized cost of $O(1)$ per ideal and an overall complexity of $O(\mathcal{I})$.

Balanced Tree Case

In the balanced tree case, the number of ideals is given by the OEIS A004019 sequence and grows exponentially with n. The number of push instances is the product of the number of binary trees of height less than or equal to $n-1$ as given by OEIS A003095.

Where n is the number of levels in the balanced tree we have

$$\lim_{n \to 5+} \text{average pushed per instance} = \frac{\text{\# of pushed elements}}{\text{\# of push instances}} = \frac{A004019(n)}{\prod_{i=1}^{n} A003095(i)} = 2.60385\ldots$$

The limit, to 512 bits of precision, is reached by $n = 10$.

This results in a balanced tree case constant amortized cost of $O(1)$ per ideal and an overall complexity of $O(\mathcal{I})$.

Best Case (narrow and deep)

In the best case, the number of ideals is equal to the number of nodes (n) and there is only one push instance per node which takes place while initializing the array. No pushes take place during processing.

This results in a best case constant time cost of $O(1)$ per ideal and an overall complexity of $O(\mathcal{I})$.

Conclusion

The time complexity of the pop jump push algorithm for generating the ideals of an arborescence is directly related to the structure of the tree. In the worst case, the balanced tree case and the best case the algorithm has a constant amortized time complexity of $O(1)$ per ideal and an overall complexity of $O(\mathcal{I})$.