Catalan Toolbox (Draft)

Last Updated on June 15, 2024

Name Redacted

Table of Contents

1. Foreword	2
2. Overview	2
3. Random Full k-ary Tree Generation	3
4. r-dyck Path	7
4.1. Pre-Order	
4.2. Mirrored Pre-Order	8
5. Triangulation	9
5.1. Tree to Triangulation	9
5.2. Triangulation to Tree	9
5.3. Triangulation Flip	10
6. Arc Graph	11
7. Chords Graph	12
8. Coin Stack	13
9. Pattern-Avoiding Enumeration	14
10. Lexicographical Enumeration	15
11. Visualization	16

1. Foreword

This documentation explores the ideas behind CATALAN TOOLBOX repo. It is currently written on a best-effort basis.

Mistakes and grammar issues are to be expected. Feel free to reach out if you have spot any errors.

2. Overview

CATALAN TOOLBOX strives to provide an efficient way to convert between well-known Catalan structures such as binary tree, dyck path, polygon triangulation, and so on. Additionally, CATALAN TOOLBOX aims to provide visualization and support common operations such as triangulation flip as well.

In an effort to pursue efficiency, C++ and Rust programming languages were considered. C++ won out due to the possibility of having self-referential data, in particular, having the child node referring back to the parent node, which Rust isn't good at. For visualization, Python was chosen for its extensive plotting libraries.

Intentionally left blank

3. Random Full k-ary Tree Generation

Let's think about it in terms of available spaces for a k-ary tree. k new spaces is created with each new internal node, and 1 space is lost with each node. There is originally 1 available space before the first node, root, is placed. This is to say:

- Internal node: k-1 new spaces.
- Leaf node: -1 space.

Let's define T to be the *per-order* traversal of a k-ary tree where 1 denotes an internal node and 0 denotes a leaf node. Let's also define S to be the *score* for T.

For example, in Figure 1, the tree nodes are labeled by the order in which it is traversed, and we obtain the sequence $S = \{1, 3, 5, 4, 3, 2, 1, 0\}$ with the corresponding $T = \{1, 1, 0, 0, 0, 0, 0, 0\}$.

Proposition 1. For any given, well-formed, full k-ary tree, it has a *unique*, *pre-order* traversal sequence S, $\{1, ..., 0\}$, which maps to the number of available spaces.

Proof.

Because the k-ary tree is well-formed and full, it must be the case that the available space reaches 0 after inserting all the tree nodes in a pre-order traversal fashion.

It also must be unique because

- Whether a node is a leaf or an internal node can be determined by whether the next number in the sequence increases or decreases.
- Each internal node must have k child nodes.
- Order of traversal is given: pre-order traversal.

2 3 4

Figure 1: A 3-ary tree. Nodes are labeled in accordance to their pre-order traversal order.

Proposition 2. At no point in sequence S, except the last number in S, reaches 0.

Proof.

Assume that it reaches 0 before the last number in S and it has traversed through p internal nodes. Then, because it reaches 0, it must have traversed through at least 1 + pk nodes. It is a contradiction because the only way for this to happen, the whole tree needs to be traversed.

Proposition 3. Any m rotation of sequence $T_m = \{t_m, t_{1+m}, ..., t_{n-1+m}\} \neq T$ cannot form a valid full k-ary tree.

Proof.

PROPOSITION 2 shows that $S = \{s_0, s_1, ..., s_{n-1}, s_n\}$, the subarray $\{s_0, ..., s_p\}$ where p < n results in $s_p \ge s_0$.

For any m rotation where $m \neq 0 \bmod n$, $T_m = \{t_m, ..., t_n, t_0, t_1, ..., t_{m-1}\}$. We know that $\{t_0, t_1, ..., t_{m-1}\}$ corresponds to $\{s_0, s_1, ..., s_m\}$ with $s_m \geq s_0$. Let's denote $s_m - s_0 = k$.

If S_m is valid, then it must end with 0 per Proposition 2; it follows that there are two possible cases:

- k=0: There must be a 0 in S_m before the last element.
- k > 0: There must be a negative element in S_m .

Both cases are in direct contradiction with Proposition 2.

Proposition 4. Given a random T_r that consists of n number of 1 and $(k-1) \cdot n + 1$ number of 0. Exactly one of its rotation results in a valid k-ary tree sequence.

Proof.

In Proposition 3, we proved that all rotation of a valid T, except T itself, is invalid. Here, we only need to prove that we can always get a valid T to begin with. A valid T is one where the number of available spaces never falls to 0 except at the last node. Because we are able to freely rotate T_r , the least advantageous arrangement would be to have 1s evenly spaced out.

Another way to look at it is by construction, say, $T_r = \{t_0, t_1, ..., t_{n-1}\}$, find the prefix score of $y = \{t_0, ..., t_p\}$ that results in the first minimum score. Then $T_{p+1} = \{t_{p+1}, ..., t_{n-1}, t_0, ..., t_p\}$ is valid.

- The score for $\left\{t_{p+1},...,t_{b}\right\}$ where $p+1\leq b\leq n-1$ must be non-decreasing, otherwise it would have been included earlier.
- The score for $\{t_0, ..., t_c\}$ where $0 \le c < p$ must also be greater than y, otherwise y would not have been the *first* minimum.

Combined the two points above, one can conclude that the minimum score for T_{p+1} happens at the last element, which as we know, is 0 because $1 + (k-1) \cdot n - ((k-1) \cdot n + 1) = 0$.

This proposition is directly related to Proposition 6.

Proposition 5. Number of well-formed, full k-ary tree is

$$\frac{\binom{kn+1}{n}}{kn+1}$$

Proof.

- There are $\binom{kn+1}{n}$ ways to choose n internal nodes from $k \cdot n + 1$ nodes.
- From Proposition 3 and 4, every kn + 1 rotation only results in exactly one full k-ary tree.

Hence, it follows that the total number of full k-ary is $\frac{\binom{kn+1}{n}}{kn+1}$.

Proposition 6. For any given rotated T_m , the root node immediately follows T_{\min} where T_{\min} corresponds to the first minimum s in S_m . If the aforementioned node is the last node, then the root wraps around to be the first node. In short, $Root = t_{p \bmod n}$ where s_p is the first minimum in S_m .

Proof.

This proposition is related to Proposition 4.

Any S_m comes from moving a segment in S at the front to be in the back; there are two cases here to discuss.

Case m=n: The minimum in S_m is found at the last element

This only happens when the entire S is rotated into itself. The minimum occurs at the last element when p = n. Hence, the root node is the first element.

This checks out because root = $t_{p \bmod n} = t_{n \bmod n} = t_0$ and $s_p = s_n = 0$, which is the minimum.

Case $m \neq n$: The minimum in S_m is found not at the last element

This happens when an overall non-decreasing segment in S at the front moves to be at the back. From Proposition 2, it follows that any valid S cannot end with a non-decreasing segment. If the root node does not immediately follow the node that leads to the *first* minimum s in S_m , it would be a contradiction due to having a non-decreasing segment at the end. Side Note:

By non-decreasing segment, I mean $\{s_v, ..., s_b\}$ where $s_b \geq s_v$.

It is entirely possible for S_m to have more than 1 minimum value. For example: $T_m = \{0,0,1,0,0,1,0\}$, then its corresponding $S_m = \{1,0,-1,1,0,-1,1,0\}$ where -1 appears twice.

With the 6 propositions above, we can generate a random full k-ary tree via a shuffle algorithm that guarantees each permutation on the indexes has an equal chance of being selected. Luckily, C++23 has exactly that with its std::shuffle() function.

Reorders the elements in the given range [first, last) such that each possible permutation of those elements has equal probability of appearance.

— CPPREFERENCE.COM

Intentionally left blank

4. r-dyck Path

4.1. Pre-Order

Given the following r-dyck path as shown in Figure 2, where all paths, except the one to the right-most child, is labeled 1 and the path to the right-most child is labeled 0. A wicked dyck path is defined as the bit string formed by performing a pre-order traversal to gather all the 01. In the case of Figure 2, it's 111010.

Given any valid wicked dyck path, output its tree structure in no worse than $O(n \log n)$ time.

4.1.1. A Quick Glance

If scanning forward from the beginning, we won't know if the next 1 is the left-most child of the current node or its sibling.

4.1.2. Strategy

By utilizing prefix sum, if we look at only the root and its children, we notice this pattern

$$\overbrace{1...1}$$
 $\overbrace{2...2}$... $\overbrace{n-1...n-1}$ $\overbrace{0...0}$

where each ... in y...y never dips below y, but can go arbitrarily high. An idea is then born: We can reconstruct it by scanning backward as follows:

- 1. Determine where the right-most subtree is by finding the last 0, say, index m, with the identical prefix sum and then recurse on subtree (m + 1, current index).
- 2. Set current index to m-1.
- 3. If current index is 1, then this path does not have a subtree.
- 4. If current index is 0, then find the *first* 1, say, index w, with the identical prefix sum and then recurse on subtree(w + 1, current index).
- 5. Combine Step 4 and Step 5 into 1 step; that is, for step 3. index w is itself.
- 6. Set current index to w-1.
- 7. Go to Step 3 until the left-most subtree has been located.

An implementation that uses binary search is provided in CATALAN TOOLBOX. Keep in mind that we don't actually need to store prefix sum separately because it can be updated as the algorithm unfolds. Additionally, finding the *first* index backward in STEP 4 can be pre-computed for all indexes in linear time with O(1) access time; finding the *last* index backward in STEP 1, on the other hand, is not that straight forward and is handled with binary search, though a O(1) query solution does exist.

Regarding time complexity, for each subtree, one binary search is carried out on a list containing only indexes that have the same prefix sum and is 0. Number of 0 in any wicked r-dyck path is $\frac{n}{r}$. Hence, in the worst case, the running time complexity is $O(n \log \frac{n}{r})$.

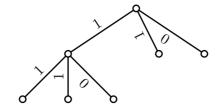


Figure 2: A 3-dyck path in 3-ary tree.

4.2. Mirrored Pre-Order

In figure 2, via *mirrored* pre-order traversal, we obtain 011011. It is immediately clear how to turn such sequence back into its tree form: Scanning forward, if the current index is 0, then go right; otherwise, go to its left sibling.

5. Triangulation

5.1. Tree to Triangulation

Figure 3 and 4 tell the whole story.

- Blue Roman numeral: It is the order in which the lines are drawn by the algorithm.
- Red lines: The lines that were drawn by the algorithm.
- Numbers: Each side corresponds to a matching leaf.
- **Direction**: Left node returns the *lower* English alphabet; while right node returns the *higher* English alphabet. For example, node 2 returns B because it's a left node, and it connects to E because the right subtree, denoted by $\{C, E\}$, returns E.
- Order: Post-order because leaf nodes need to be processed first.

This algorithm, as seen in CATALAN TOOLBOX, runs in O(n) time.

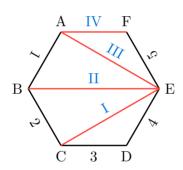


Figure 3: Hexagon

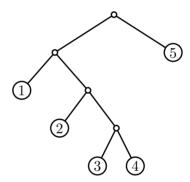


Figure 4: Corresponding tree

5.2. Triangulation to Tree

In a similar fashion to Section 5.1, process it in a bottom-up approach to ensure runtime being O(n). We consider the lines formed by two adjacent sides, as known as ears, are the closest to each other and have a distance of 2 units; in Figure 3, $\{C, E\}$ is one such case.

The maximum distance in any polygon is capped at the total number of sides -1. In the case of Figure 3, it's $\{A, F\}$, which has 5 units of distance; therefore, to process lines in a bottom-up fashion, one only needs to gather all lines in accordance to its distance, then process all the lines within bucket with distance 2, followed by bucket with distance 3, and so on.

Then, at any given line, say $\{x,y\}$ where x < y, the lower x would look up to get its left child, and the higher y would look down to get its right child; and in the end, update itself to be the node that x would look up to and y would look down to. For example, $\{C,E\}$, from C, we get that node 3 is the left child, and from E, we get that node 4 is the right child. For $\{B,E\}$, from B, we get that node 2 is the left child, and internal node created by line I to be the right child because it was updated previously when processing $\{C,E\}$.

5.3. Triangulation Flip

In addition to keeping track of what Section 5.2 tracks, we can keep a separate array that tracks the parent-child relationship. This can be done while doing triangulation to tree in Section 5.2 to avoid paying the O(n) pre-processing time. Catalan Toolbox chooses to **not** do this because there may not be a guarantee that the algorithm in Section 5.2 would be called first.

Once we obtain all needed info with the pre-processing, we can deduce the points that it flips to. Say, $\{x,y\}$ is selected to be flipped, then it must be the case that one vertex u is the *higher* vertex of its left child edge, and the other vertex v is from the parent edge which does not overlap with both x and y. Note that it does not guarantee that u is higher or lower than v, hence in Catalan Toolbox, there is a check to swap u with v if u > v.

To update the relationship between edges after a flip, there are two cases:

- Left child remains to be a child.
- Left child is not a child anymore.

In either cases, the update codes can be summarized into one case as seen in Catalan Toolbox

where it toggles between 2 cases with XOR operator. It is arguably not very readable. It achieves running time of O(1) per flip.

6. Arc Graph

7. Chords Graph

8. Coin Stack

9. Pattern-Avoiding Enumeration

10. Lexicographical Enumeration

11. Visualization