Polytopes and Consecutive Patterns on Trees

Christina Nguyen

December 20, 2020

In this report we want to explore the feasible region for consecutive patterns on trees. First, we will begin with a discussion on trees and patterns. From there, we will discuss a a classic example of patterns on permutations. Then, we will return to the topic of trees and the question that began this project which is whether or not the feasible region for patterns of depth 2 on trees is a polytope.

1 Background

1.1 Trees

A tree T is a graph with no cycles and a rooted tree $T(x)$ is a tree T with a specified vertex x as its root as shown in Figure 2, see [3] for a complete definition in the context of graph theory. Let v be a vertex of T and u be another vertex. Then, we can call u a child of v if there is a path from the root to u that passes through v and u and v are neighbors. Furthermore, we will call any vertex of degree one a leaf if it's not the root. For any tree T, the maximum number of children is given by δ and its depth as r.

A rooted planar tree is a rooted tree where we order the vertices. The tree in Figure 2 is also a rooted planar tree. Figure 5 illustrates how ordering the vertices changes how we identify different trees. We will focus on rooted planar trees for this project and we will simply refer to them as trees.

1.2 Patterns

Let X be a set of size $n \in \mathbb{N}$ with a natural order of $x_1, x_2, \ldots, x_n$ and we will consider the set S_X to be the set of permutations of size X. Let $\sigma \in S_x$ be a permutation in one line notation so $(\sigma(x_1), \sigma(x_2), \ldots, \sigma(x_n))$ and $\pi = (y_1, \ldots, y_k) \in S_k$ be a pattern with $k \leq n$ [6]. We will denote the number of classical occurrences of π in σ as

$$\mathrm{occ}(\pi, \sigma) = |\{I \subseteq [n] \colon \mathrm{pat}_I(\sigma) = \pi\}|$$

where I is a subset of $[n]$ [4].

We define $\widetilde{\mathrm{occ}}$ as the proportion of classical occurrences of a permutation π in σ as

$$\widetilde{\mathrm{occ}}(\pi, \sigma) = \frac{\mathrm{occ}(\pi, \sigma)}{\binom{n}{k}} \in [0, 1].$$

In the limit for large σ we can obtain the feasible region for classical occurrences, or the vectors that encode the possible proportion of patterns, is

$$\mathrm{FR}_{classical} = \left\{\vec{v} \in [0,1]^{S_k} \colon \exists\, (\sigma^m)_{m \in \mathbb{N}} \in S_n \text{ s.t. } |\sigma^m| \to \infty \text{ and } \widetilde{\mathrm{occ}}(\pi, \sigma^m) \to \vec{v}_\pi, \forall \pi \in S_k\right\}.$$

Classical occurrences are well studied, see [2], [7], so we want to look at a similar concept. We want to study what happens when we constrain our patterns to occurrences where the pattern can be found in consecutive indices, or consecutive occurrences. We'll denote the number of consecutive occurrences of π as

$$\text{c-occ}(\pi, \sigma) = |\{I \subseteq [n] \colon I \text{ is an interval } \mathrm{pat}_I(\sigma) = \pi\}|.$$

Similarly, we define the proportion of consecutive occurrences as

$$\widetilde{\text{c-occ}}(\pi, \sigma) = \frac{\text{c-occ}(\pi, \sigma)}{n} \in [0, 1].$$

The feasible region for consecutive occurrences is

$$\text{FR} = \left\{ \vec{v} \in [0,1]^{S_k} : \exists \left(\sigma^m\right)_{m \in \mathbb{N}} \in S_n \text{ s.t. } |\sigma^m| \to \infty \text{ and } \widetilde{\text{c-occ}}(\pi, \sigma^m) \to \vec{v}_\pi, \forall \pi \in S_k \right\}.$$

We claim that the description of the feasible region gives us points within the convex hull of a polytope [4]. We know that the vertices correspond to linear functionals and we also know that the feasible region corresponds to a convex set. For permutations it's known that the feasible region is a polytope. However, in general it's not known that the feasible region is a polytope. In this report we will investigate the feasible region for trees.

2 Permutations

First, we will discuss a concrete example. Consider $\sigma \in S_8$ such that $\sigma = (7, 1, 4, 5, 2, 3, 6, 8)$ and we will look at patterns π of $(1, 2)$ and $(2, 1)$.

To find the number of classical occurrences, we can count the times the patterns $(1, 2)$ and $(2, 1)$ occur in σ. We get that $\text{occ}((1, 2), \sigma) = 18$ and $\text{occ}((2, 1), \sigma) = 10$. From this we get

$$\widetilde{\text{occ}}((1, 2), \sigma) = \frac{\text{occ}((1, 2), \sigma)}{28} = \frac{18}{28} \in [0, 1],$$
$$\widetilde{\text{occ}}((2, 1), \sigma) = \frac{\text{occ}((2, 1), \sigma)}{28} = \frac{10}{28} \in [0, 1].$$

If we were to compute this for all sequences in S_8 we could plot those points to obtain a discrete set close to the feasible region. In order to approximate the feasible region we need to compute these values for a larger S_n. Or, we can maximize $\widetilde{\text{occ}}((1, 2), \sigma)$ and $\widetilde{\text{occ}}((2, 1), \sigma)$ over a vector $\vec{v} \in [0, 1]$ to obtain the vertices of the feasible region.

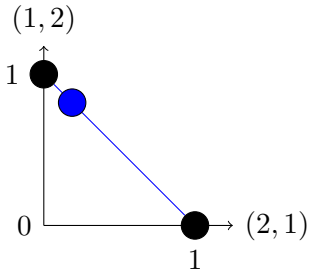

Figure 1: The feasible region for $(1, 2)$ and $(2, 1)$ is a line.

3 Trees

Now we'll look at trees. Why should we study patterns in trees? Trees can bridge between the linearity of permutations and the non-linearity of graphs. Let $\mathcal{F}_\delta^{(r)}$ be a set of trees of depth at most r with at most δ children. Let T, S be rooted trees, then the number of consecutive occurrences of S in T is denoted by $\text{c-occ}_\delta^{(r)}(S, T)$

$$\widetilde{\text{c-occ}}^{(r)}(S, T) = \frac{\text{c-occ}^{(r)}(S, T)}{|T|}.$$

We also define a vector, where δ indicates the maximum number of children in the patterns,

$$\widetilde{\text{c-occ}}_{\delta}^{(r)}(S,T) \in [0,1]^{|\mathcal{F}_{\delta}^{r}|}$$

since the depth of the pattern changes how we count them on the tree. We want to introduce a slightly different definition for the feasible region

$$\text{FR}_{\delta}^{(r)} = \left\{ \nu \in [0,1]^{|\mathcal{F}_{\delta}^{r}|} \;\middle|\; \exists \left(T^{(n)}\right) \text{ s.t. } \widetilde{\text{c-occ}}_{\delta}^{(r)}\left(T^{(n)}\right) \to \nu, \left|T^{(n)}\right| \to \infty \right\}.$$

This definition is very similar to the definition of the feasible region for consecutive occurrences as seen earlier. However, we introduce δ and r since we need to consider the maximum degree of the tree along with the depth of the pattern.

For trees with a maximum degree of 2 and for patterns of depth 2, we propose that $\text{FR}_{\delta}^{(r)}$ is a polytope.

3.1 Trees of Depth 1

Now we can put it all together into a concrete example for a tree with $\delta = 2$ and $r = 1$. Consider the following rooted tree

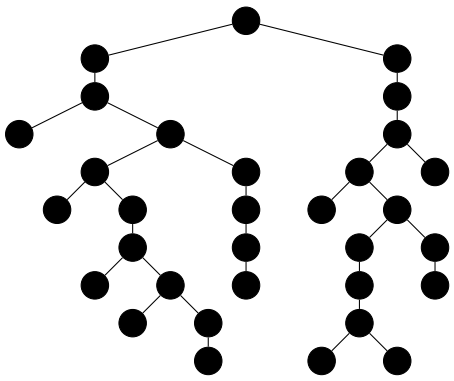

Figure 2: An example of a tree with $0 \leq \delta \leq 2$.

For trees with at most depth 1, there are three possible trees. We will introduce them with the following notation, where each tree is described by the degree of its root.

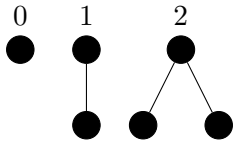

Figure 3: The trees in $\mathcal{F}_2^{(1)}$.

For a fixed tree S let $a_S = \widetilde{\text{c-occ}}_{\delta}^{(1)}(S,T)$ where $S \in \mathcal{F}_2^{(1)}$ as described in Figure 3. If $a_S \in \text{FR}_2^{(1)}$ then, in the limit, we claim that the following equations are satisfied

$$a_S \geq 0 \text{ for } S \in \mathcal{F}_{\delta}^{(r)}, \tag{1}$$

$$\sum_{S \in \mathcal{F}_{\delta}^{(r)}} a_S = 1, \tag{2}$$

$$\sum_{S \in \mathcal{F}_{\delta}^{(r)}} \deg(S) a_S = 1. \tag{3}$$

These equations do not easily follow from the definitions and a proof is provided in [1].

For a tree of this size, it wouldn't be too difficult to find the feasible vector by calculating c-occ and $\widetilde{\text{c-occ}}$ by hand. But in order to generate the feasible region, we would need more trees and larger trees so we used python to compute these values. For the tree in Figure 2, the values are found using the code in Appendix A.3.

$$\widetilde{\text{c-occ}}(0,T) = \frac{\text{c-occ}(0,T)}{|T|} = \frac{11}{32},$$
$$\widetilde{\text{c-occ}}(1,T) = \frac{\text{c-occ}(0,T)}{|T|} = \frac{11}{32},$$
$$\widetilde{\text{c-occ}}(2,T) = \frac{\text{c-occ}(0,T)}{|T|} = \frac{10}{32}.$$

There is a discrepancy between the values we expect from Equations 2 and 3 since we are calculating the values with a smaller tree. If we were to do the same calculation for a much larger tree we would eventually obtain the desired equalities. We can obtain the feasible region from the above equations which is shown in the figure below.

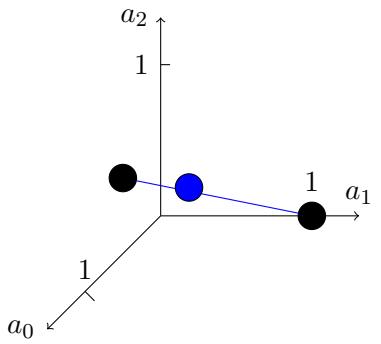

Figure 4: The feasible region for patterns of depth 1.

3.2 Trees of Depth 2

For patterns, of depth 2 the problem gets more interesting (aka harder) since we have 13 possible trees of depth 2 where each vertex has at most 2 children. Below, we will introduce the trees with the following notation where the trees are identified according to the figure below.

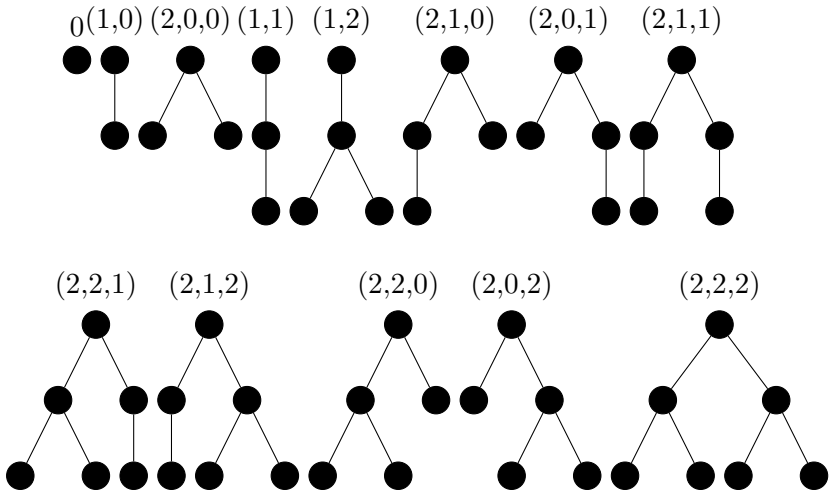

Figure 5: The trees in $\mathcal{F}_2^{(2)}$.

For trees of depth at most 2, in the feasible region, the following equations are satisfied

$$\sum_{S\in\mathcal{F}_\delta^{(r)}} a_S = 1, \tag{4}$$

$$\sum_{S\in\mathcal{F}_\delta^{(r)}} \deg(S)a_S = 1, \tag{5}$$

$$a_0 = a_{(1,0)} + a_{(2,0,1)} + a_{(2,0,2)} + a_{(2,2,0)} + a_{(2,1,0)} + 2a_{(2,0,0)} \tag{6}$$

$$a_{(1,0)} + a_{(1,2)} = a_{(2,0,1)} + a_{(2,2,1)} + a_{(2,1,0)} + a_{(2,1,2)} + 2a_{(2,1,1)}, \tag{7}$$

$$a_{(1,2)} + a_{(2,2,2)} = a_{(2,0,0)} + a_{(2,0,1)} + a_{(2,1,0)} + a_{(2,1,1)}. \tag{8}$$

with $a_S \geq 0$ for all $S \in \mathcal{F}_\delta^{(r)}$.

For patterns of depth 2, we will use python to find the feasible vector for the tree in Figure 2. Using the code in Appendix A.4 we found the values for a_T

$$\begin{array}{lll} a_0 = 0.34375, & a_{(1,0)} = 0.09375, & a_{(2,0,0)} = 0.03125, \\ a_{(1,1)} = 0.125, & a_{(1,2)} = 0.125, & a_{(2,1,0)} = 0.0, \\ a_{(2,0,1)} = 0.0625, & a_{(2,1,1)} = 0.0625, & a_{(2,2,0)} = 0.03125, \\ a_{(2,0,2)} = 0.09375, & a_{(2,2,2)} = 0.0, & a_{(2,1,2)} = 0.0, \\ a_{(2,2,1)} = 0.03125 & & \end{array}$$

We also checked these values against Equations 6-8 using the code in Appendix A.5.

$$(6) : 0.34375 = 0.34375,$$
$$(7) : 0.21875 = 0.21875,$$
$$(8) : 0.125 = 0.15625$$

We expect that the equation would hold in the limit because the margin of error would shrink as the trees get larger. If we were to iterate the code in Appendix A.5 for a set of large randomly generated trees we expect to get a set of points that would approximate the feasible region.

4 Conclusion

We were able to compute the feasible vector for randomly generated trees and to verify that Equations 6-8 hold in the feasible region. As a result we are one step closer to proving that $\mathrm{FR}_2^{(2)}$ is a polytope. Since we know that the feasible region is convex and is in a $9-$dimensional vector space we can approximate its vertices. Afterwards, we will be able to check if the points generated by the code lie within the convex hull.

References

[1] David Aldous. "Asymptotic Fringe Distributions for General Families of Random Trees". In: *Ann. Appl. Probab.* 1.2 (May 1991), pp. 228–266. DOI: 10.1214/aoap/1177005936. URL: https://doi.org/10.1214/aoap/1177005936.

[2] Miklos Bona. *Combinatorics of Permutations*. Chapman and Hall/CRC, Apr. 2016. ISBN: 9780429107245. DOI: 10.1201/b12210.

[3] Adrian Bondy and U.S.R. Murty. *Graph Theory*. Springer-Verlan, 2008.

[4] Jacopo Borga and Raul Penaguiao. "The feasible region for consecutive patterns of permutations is a cycle polytope". In: (Oct. 2019). URL: http://arxiv.org/abs/1910.02233.

[5] Michael T. Goodrich, Roberto Tamassia, and Michael H. Goldwasser. *Data Structures and Algorithms in Python*. Wiley Global Education, 2013.

[6] Richard Kenyon et al. "Permutations with fixed pattern densities". In: *Random Structures and Algorithms* 56 (1 2020). ISSN: 10982418. DOI: 10.1002/rsa.20882.

[7] Vincent Vatter and Nik Ruskuc. *Permutation Patterns*. Ed. by Steve Linton, Nik Ruskuc, and Vincent Vatter. Cambridge University Press, 2010. ISBN: 9780511902499. DOI: 10.1017/CBO9780511902499.

[8] Günter M. Ziegler. *Lectures on Polytopes*. Vol. 152. Springer New York, 1995. ISBN: 978-0-387-94365-7. DOI: 10.1007/978-1-4613-8431-1.

A Code

A.1 Tree Generation

```
import random as rand #to create random children

tree = [2] #root is 2 for variety
children = [] #empty list to store children
amount = 2 #we start with two children, since we start at 2

"""
Generates the trees, but sometimes makes small trees
Note! r generates r additional levels and a row of 0s is appended
    so a tree with r=4 will actually have 6 levels
"""
def tree_gen(d,r): #d is number of children, r is depth
    tree=[2]
    amount = 2
    x = 0 #placeholder var for random numbers
    for i in range(r): #iterates over the depth
        children = []
        for j in range(amount): #makes sure we don't get trees that don't work
            x = rand.randint(0,d)
            children.append(x)
            temp = sum(children)  #biases random numbers so we get bigger trees
            if temp == 0:
                while temp == 0:
                    for k in range(amount):
                        x = rand.randint(0, d)
                        children.append(x)
                        temp = sum(children)
                amount = sum(children)
            else:
                amount = sum(children)
        tree.append(children)
    children = []
    for k in range(amount):
        children.append(0)
    tree.append(children)
    return tree
```

A.2 Size of Tree

```
def size():
    size = 1
    for i in range(1,len(tree)):
        for j in range(len(tree[i])):
            size += 1
    return size
```

A.3 Pattern Checker for Depth 1

```
#checks for depth 1 patterns
def pat_check_1(n):
    count = 0 #finds c-occ

    if tree[0] == n: #checks the root
        count += 1
    for i in range(1,len(tree)): #this is to make python happy
        if len(tree[i]) >=2:
            for j in range(len(tree[i])):
                if tree[i][j] == n:
                    count += 1
        else: #not the most efficient way to get rid of a bug but it works
            if tree[i][0] == n:
                count +=1

    return count
```

A.4 Pattern Checkers for Depth 2

```
"""
Checks for patterns of depth 2 with a root of degree 1.
This is messy but it works.
"""
def pat_check_n1(n,a):
    count = 0
    pos = 0 #temp var for the position of the root of the pattern tree
    row = 0
    if tree[0] == n: #checks the root
        if tree[1][0] == a:
            count += 1
            print("row: 0", "count: ", count)

    row += 1
    while row <= len(tree)-2: #keeps index in length of the tree
                              #index n-2 is actually the 2nd to last bc python
       if len(tree[row]) == 1:
           if tree[row][0] == n:
               if tree[row+1][0] == a:
                   count += 1
                   print("row: ", row, "col: ", "0", "count: ", count)
```

```
        if len(tree[row]) == 2: #if I leave this out it over counts
                                    #NO IDEA WHY THOUGH
            if tree[row][0] == n:
                if tree[row+1][0] == a:
                    count += 1
                    print("row: ", row, "col: ", "0", "count: ", count)

            if tree[row][1] == n:
                pos = tree[row][0]
                if tree[row+1][pos] == a:
                    count += 1
                    print("row: ", row, "col: ", "1", "count: ", count)

        if len(tree[row]) > 2:
            i = 0
            while i <= len(tree[row])-1:
                if tree[row][i] == n:
                    pos = sum(tree[row][0:i])
                    if tree[row+1][pos] == a:
                        count += 1
                        print("row: ", row, "col: ", i, "count: ", count, "pos: ", pos
                i +=1

        row += 1

    return count

"""
Checks for patterns of depth 2, with a root of degree 2.
This isn't my best bit of code.
It works so that's enough for now.
"""
def pat_check_n2(n,a,b):
    count = 0
    pos = 0 #temp var for the position of the root of the pattern tree
    row = 0
    if tree[0] == n: #checks the root
        if tree[1][0] == a:
            if tree[1][1] == b:
                count += 1
                #print("row: 0", "count: ", count)

    row += 1
    while row <= len(tree)-2: #keeps index in length of the tree
                              #index n-2 is actually the 2nd to last bc python
        if len(tree[row]) == 1:
            if tree[row][0] == n:
                if tree[row+1][0] == a:
                    if tree[row+1][1] == b:
```

```
                    count += 1
                    #print("row: ", row, "col: ", "0", "count: ", count)

        if len(tree[row]) == 2: #over counts if I leave it out. no idea why
            if tree[row][0] == n:
                if tree[row+1][0] == a:
                    if tree[row+1][1] == b:
                        count += 1
                        #print("row: ", row, "col: ", "0", "count: ", count)
            if tree[row][1] == n:
                pos = tree[row][0]
                if tree[row+1][pos] == a:
                    if tree[row+1][pos+1] == b:
                        count += 1
                        #print("row: ", row, "col: ", "1", "count: ", count)

        if len(tree[row]) > 2:
            i = 0
            while i <= len(tree[row])-1:
                if tree[row][i] == n:
                    pos = sum(tree[row][0:i])
                    if tree[row+1][pos] == a:
                        if tree[row+1][pos+1] == b:
                            count += 1
                            #print("row: ", row, "col: ", i, "count: ", count, "pos: "
                i += 1

        row += 1
    return count
```

A.5 Tree Relation Checker

```
"""
Generates a random tree and then checks the patterns
User inputs a pattern depth and then the depth of the random tree.
"""

#import the two files with all the other functions
from treegen import *
from depth2 import pat_check_n1, pat_check_n2

def tree_checker(pat, depth): #the pattern type (depth 1 or 2)

    #tree = [2]
    size = 0

    #depth = int(input("Depth of the tree : "))
    tree = tree_gen(2,depth)
    size = tree_size()

    #tree = [2, [1, 1], [2, 1], [0, 2, 2], [2, 1, 2, 0], [0, 1, 1, 0, 2],
```

```
#[2, 1, 1, 1], [0, 2, 0, 1, 0], [0, 1, 2],[0,0,0]]

if pat == 1:
    a0 = pat_check_1(0) / size
    a1 = pat_check_1(1) / size
    a2 = pat_check_1(2) / size

    print("a_0 :", a0, "a_1 :", a1, "a_2 :", a2)

elif pat == 2:
    a0 = pat_check_1(0) / size
    a1 = pat_check_1(1) / size
    a2 = pat_check_1(2) / size

    a10 = pat_check_n1(1,0) / size
    a11 = pat_check_n1(1,1) / size
    a12 = pat_check_n1(1,2) / size

    a200 = pat_check_n2(2,0,0) / size
    a210 = pat_check_n2(2,1,0) / size
    a201 = pat_check_n2(2,0,1) / size
    a211 = pat_check_n2(2,1,1) / size
    a220 = pat_check_n2(2,2,0) / size
    a202 = pat_check_n2(2,0,2) / size
    a221 = pat_check_n2(2,2,1) / size
    a212 = pat_check_n2(2,1,2) / size
    a222 = pat_check_n2(2,2,2) / size

    print("the occurences: ")
    print("a_0 :", a0, "a_1 :", a1, "a_2 :", a2)
    print("a_10 :", a10, "a_11 :", a11, "a_12 :", a12)
    print("a_200 :", a200, "a_210 :", a210, "a_201 :", a201)
    print("a_211 :", a211, "a_220 :", a220, "a_202 :", a202)
    print("a_222 :", a222, "a_212 :", a212, "a_221 :", a221)

    #a1, a2 are already accounted for by a10 and a200
    eq1 = a10 + a201+ a202 + a220 + a210 + 2*a200
    eq2_lhs = a10 + a12
    eq2_rhs = a201 + a221 + a210 + a212 + 2*a211
    eq3_lhs = a12 + a222
    eq3_rhs = a200 + a201 + a210 + a211

    print("Equations: ")
    print("1st :", a0, "=", eq1)
    print("2nd :", eq2_lhs, "=", eq2_rhs)
    print("3rd :", eq3_lhs, "=", eq3_rhs)

else:
    print(":(")
```