Tree Visualization: The Math, The Biology, The CS

Bill Z. Qin

Last Updated on January 13, 2020

*This document is a supplement to Bill Z. Qin's Tree Visualization project. This document is **not a guide** on how to use the app. If you would like a guide, please visit the project's Github Repository or navigate to the guide page directly on the app.*

This document, while not of the most scholarly kind, is also not a cakewalk. There will be mathematics, statistics, and computer science that cater towards the average college student. If you feel unable or do not want to understand this content, you may find this document boring, confusing, and/or frustrating. If you are unsure about your readiness for this document, check out the Critical Concepts section for a comprehensive list of technical concepts used in this project.

Modelling trees is no simple task. Apart from creating a suitable UI and full-stack application, the growth and managing of trees becomes an increasingly difficult task. This document will outline the mathematics and strategies used to model trees and their components in this project. It is important to note that as the project undergoes further development that these strategies will inevitably change and improve and that this document will reflect such changes. On the other hand, the mathematics and strategies presented may not be the most efficient and may sometimes be serving as placeholders until a breakthrough is made in that area of the project.

Contents

1 Appetizers

Much like ordering an appetizer at a restaurant, it is by all means optional but may prove beneficial to read before the rest of the document.

1.1 Critical Concepts

As stated in the notice at the beginning of the document, this document is not a cakewalk. This is a (hopefully) comprehensive list of all the technical concepts that I employed while making this project. **You do not need to know all concepts listed here to view this document.**

- Frontend App Development (written in ERB)
  - HTML/CSS (though not explicitly mentioned)
  - Javascript (though not explicitly mentioned)
  - SVGs
- Backend App Development (written in Ruby w/ Sinatra)
  - Model-View-Controller Framework
  - Databases and SQL
- Algorithm Development
  - Recursion
  - Memoization
  - Simple Graph Theory
  - Complexity Theory and Notation (Big-O, Amortization)
- Simple Statistical Distributions (normal distribution, uniform distribution)
- Mathematics
  - 2D Cartesian Geometry and Trigonometry
  - The Logistic Model of Cell Growth
  - Advanced Functions and Limits
  - Advanced Math Notation (Sums, Sets, etc.)
  - Probability Density Functions
- Trees (I'm not sure what you'll understand if you don't know what trees look like)

1.2 Lingo, Jargon, etc.

This project involves the growing of trees, which are complicated. As such, it is important that we first develop some terminology that allow us to break down trees into more barebone components that will allow us to use mathematics and biology to understand.

Important Vocabulary:

- A **tree** is an entity, developed as a model in the application, that consists of many **branches**.
  - Every tree keeps a record of how many branches it has through a `last_branch_id`, which also serves as the identification for the next new branch introduced to the tree.
  - Every tree has an **age** associated with it, measured in days.
  - Every tree belongs to a **user** and only one user. A user may, of course, own multiple trees.
- A **branch** is an entity, developed as a model in the application, that serves as the fundamental building block for all trees.
  - Every branch has a specific **identification number** associated with it. Within a tree, this number will be unique to each branch, but will not necessarily be so across other trees.
  - Every branch also has an **age** associated with it, measured in days.
  - Every branch has a **length**, measured in pixels. **This unit of measurement will likely be changed in order to make the application's UI more stable on different window resolutions.**
  - (*UNIMPLEMENTED*) Every branch has a **width**, measured in pixels. **This unit of measurement will likely be changed in order to make the application's UI more stable on different window resolutions.**
  - Every branch, in the database, is given a **parent branch** (which can be NULL), a **position scalar** that ranges from the open/closed interval of real numbers (0,1], and an **angle of deviation**. Together, this allows the front-end to compute the exact position of every branch on the tree through trigonometry. For more information, see how we compute branch endpoints in the frontend.
  - Every branch has a **generation**, which is a non-negative integer. The generation of a branch is 0 if it has no parent and is the generation of its parent branch plus 1 if it does have one.
- During a **day**, every branch of the tree will grow a little, increasing in length and (*UNIMPLEMENTED*) width.
  - Every branch b has a chance (possibly zero) of producing an **offshoot**, which is a branch b' that will have b as its parent, with a certain minimum angle of deviation. Note that b' will NOT grow on the day it was created, and will start the next day at age 0.
  - (*CURRENTLY UNIMPLEMENTED*) Every branch b has a chance (possibly zero) of **splitting**. This will result in the creation of two branches b_1 and b_2 that will have b as their parent and will

have a position scalar of 1, while marking b as having been split and thereby **unable to split again**. Similar to offshoots, b_1 and b_2 will NOT grow on the day they were created and will start the next day at age 0.

- The **window** for the tree is a hard limit for all tree growth, and *no tree can grow past it.* As the unit of measurement is still pixels, the window, too, is measured as a 800 pixel by 600 pixel range.

2 Data Management

With this much information, it is critical to think about how to store this information efficiently and effectively.

2.1 User Information

This classic feature of applications involves the storage of user information securely. Users are, of course, created as models (found on the Github Repository under `/models/user.rb`). This application has opted to employ SHA256 encryption with salting as its method of security. This method of encryption may eventually be overhauled for a more powerful method using MD5 and private keys.

2.2 Trees

Trees are created as models (found on the Github Repository under `/models/tree.rb`). Every tree then belongs to a user and contains many branches. There's nothing too notable about how trees are stored, except that they are identified through the usage of UUIDs, or Universally Unique IDs, that are near-guaranteed to be uniquely generated. This document will not go into UUIDs, but you can visit the Wikipedia page on them for more information.

2.3 Branches

In the initial state of the project, every tree had a `branches_and_roots` JSON string attached to them, which would contain the information on all branches in order to initially test the front-end's capabilities. However, this method is inefficient both in memory and mutability, and was deprecated for also making branches also as models (found on the Github Repository under `/models/branch.rb`).

As noted earlier, branches store information such as their length, width, parent branch, etc. Apart from length and width, most of these variables will remain constant throughout a branch's lifetime. As such, these variables were selected to be stored under the branch model as opposed to simply recording the positions of the branches in 2D space, which may make computations far more tedious and take longer to compute.

For example, suppose branch a has child branch b. a currently has length 30 and angle 0, where as branch b has length 10, angle 30, and position scalar 0.6. This means that if we let branch a start at (0,0) that a becomes the branch $((0,0),(0,-30))$, which means that branch b starts at $(0,-18)$, which means that branch b becomes $((0,-18),(5,-18-5\sqrt{3}))$. We note the simplicity of calculating each branch's actual location from this information. If we move to the next day and branch a grows 5, it follows that when computing b's growth, we need only adjust the length of b in the database, as b's new location will simply be recomputed.

If we had instead stored only the branches as $((0,0),(0,-30))$ and $((0,-18),(5,-18-5\sqrt{3}))$, it would be

	Current Memory Model	Position Memory Model
Load Tree	~ $25n$	$2n$
Go Forward 1 Day	~ $25n^*$	~ $48n^*$

Table 1: Computations required to perform operations on trees.
Here, n is the number of branches on the tree in question.
*: These values are amortized to include tree behaviour (i.e. splitting, offshoots).

extremely tedious to recalculate b's new location, as we would first have to recompute each length, then recalculate b's starting point and performing the same computation as before.

Regarding Table 1, we note that the number of computations that are required to load a tree and go forward 1 day both sum to approximately $50n$ computations. However, because you may grow trees by up to 999 days at a time, and you can only request to load a tree "once", it follows that the current memory model gives more consistent results and having a better worst-case scenario.

3 Growth of Trees

Simulating how these trees will grow is arguably the most complex part of the project, as this involves developing an algorithm for tree growth that will result in believable trees while sustaining a sufficient amount of randomness so that every tree will grow differently. **The algorithm that stands is nowhere near perfect, and will change constantly. Furthermore, it is also incomplete at the moment.**

The growth of trees, on paper, is very simple. Currently, the only way to grow a tree is by selecting some x days to grow it, where $x \in \mathbb{Z}^*_{1000}$, or that x is an integer ranging from 1 to 999, inclusive. The application then simply runs a function `grow_tree()` on the tree x number of times.

We note that this is not the most efficient way to go about this task. As we will see in the upcoming subsection, `grow_tree()` works by branch, simply iterating through branches. This proves to be a $O(nx)$ computation, where n is the number of branches on the tree. This can be brought down to an amortized $O(n)$ operation, as will be shown in the Other and To-be-implemented section.

3.1 Creation of Trees

As this project currently stands, when a tree is created, it will be generated with one single branch b. Branch b will have a length of 5, an angle of 0, and will be situated so that it appears centered at the very bottom of the window.

This will, of course, eventually be improved upon.

3.2 Branch-wise Growth

Upon calling `grow_tree()`, the function will take every existing branch of the tree and increase its length. The function manages to target every branch because given any tree t, the IDs of the set of branches B of t are guaranteed to be the set $\{0, 1, \ldots, n-1\}$, where n is the number of branches of t. This means we can perform a simple SQL query on all of these IDs within the tree t to get its information and update it accordingly.

To compute the new length of the tree, the application opts for a **logistic model with deviation**. The logistical model is commonly employed in biological and chemical simulations as a simple way to mimic cell birth and cell death in one neat and tidy formula.

$$l(t) = \frac{c(G)}{1 + ae^{-bt}}$$

Equation 4.1: The Logistic Model

In our case, the variables are defined as such:

- t is the current day.
- $l(t)$ is the length of the branch when it has age t.
- a is the scaling factor that dictates how fast a branch will grow and meet half its carrying capacity. We note that we can expect $l(t) = 0.5c$ at roughly $t = \frac{\ln(a)}{b}$.
- b is the growth factor that will also dictate how steeply the branch will grow, and is known as the **logistic growth rate**.
- $c(G)$ is the carrying capacity of a branch, and G is the generation of this branch. That is, the branch will never exceed a length of $c(G)$, and $\lim_{t\to\infty} l(t) = c(G)$.

These variables are tweaked constantly to best mimic tree growth, but will never be perfect. Furthermore, c is currently based on the branch's generation, while a and b are constant throughout all trees and all branches.

Of course, simply using the logistic model would give us the same lengths for branches of the same generation and the same age always. As such, the application employs this formula:

$$l'(t) = l_0 + D(l(t) - l(t-1))$$

Equation 4.2: The Logistic Model w/ Deviation

where

- t is the current day.
- $l'(t)$ is the length of the branch when it has age t.
- l_0 is the length of the branch before the day had started.
- $l(t), l(t-1)$ are computed using the old logistic model.
- $D(x)$ is a zero-capped standard deviation function.

3.2.1 Zero-capped Standard Deviation

Given some positive real x, the function $D(x)$ first picks a value L from $\mathcal{N}(x, (Zx)^2)$ and then returns

$$D(x) = \begin{cases} L & \text{if } l > 0 \\ 0 & \text{otherwise} \end{cases}$$

where $\mathcal{N}(x, (Zx)^2)$ is the normal distribution with mean x and standard deviation Zx, where Z is a constant (much like a and b, that is being tweaked to make tree growth as clean as possible). As a reminder, the normal distribution $\mathcal{N}(\mu, \sigma^2)$ has probability density

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-0.5\left(\frac{x-\mu}{\sigma}\right)^2}$$

for all $x \in \mathbb{R}$. In Ruby, we can pick from the normal distribution thanks to the `Normal.rng` function found in the `distribution` gem that specializes in all sorts of statistical distributions, which can be found here.

3.3 Offshoots

To be completed soon. Stay tuned!

3.4 Splitting (*UNIMPLEMENTED*)

To be completed soon. Stay tuned!

4 Displaying Trees

When trees grow, they are stored and altered in the database, but a table with a bunch of numbers is not helpful to any user. Just as important is being able to display the information in the table in a meaningful way.

4.1 Implementation as SVG

SVGs, or Scalable Vector Graphics, is an image format that specializes in changing and animated 2D graphics, and has been selected as the way trees will be displayed for users.

Every branch is represented as an SVG line on the canvas that holds the entire tree with some width and some endpoints within their canvas, and are colored the same color: a nice shade of brown with RGB value `#654321`. As the project progresses, branches may take a different form and be displayed differently.

4.2 Branch Endpoints

The width of the SVG line for a branch is currently very simple to find, it's currently constant and will eventually simply be a data point on every branch in the database. However, the more challenging part comes from locating the branch's endpoints. Fortunately, every branch, as you may recall, has been given a select number of variables as defined in the Lingo and Jargon section, which will make this task all the more feasible.

We first observe that if we know one endpoint, the length of the branch, and the angle at which the line must be drawn, it can be computed by way of trigonometry the second endpoint. Since the length of the branch is a data point on the branch, we need only compute the first endpoint and the angle.

4.2.1 The First Endpoint

Recall that branches have a "position scalar" that ranges from the open/closed interval of real numbers (0,1]. This position scalar is given to a branch at birth and remains constant for every branch, and usually ranges from 0.3 to 0.7 (currently assigned to uniformly at random). This dictates where along its parent branch it branches off from.

For some branch b, suppose it has scalar k. Now, suppose we know the endpoints of its parent branch to be (x_1, y_1) and (x_2, y_2) on the canvas. It follows that we can compute the first endpoint (x, y) of the branch as

$$(x, y) = (x_1 k + y_1(1 - k), x_2 k + y_2(1 - k))$$

Of course, this means we first have to compute the parent branch's endpoints. To alleviate this problem, the algorithm will run recursively and employ memoization, or storing previous answers, to complete the computation.

4.2.2 The Angle

The angle is actually far simpler to determine. We recall that every branch has an "angle of deviation" from its parent branch. Using another recursive function, we can simply use the following function to compute the angle:

$$f(id) = \begin{cases} f(P(id)) + \theta_{id} & \text{if branch with this id has a parent} \\ 0 & \text{otherwise (when id = 0 and the branch is the root)} \end{cases}$$

where θ_{id} is the angle of deviation of the branch with that id and $P(id)$ is the ID of the parent of said branch. Once again, memoization is used to reduce the time complexity from $O(n \log n)$ to $O(n)$.

4.2.3 Finishing Up

Once you have the first endpoint (x_1, y_1), the length l, and the angle θ of our branch, we can quickly compute the second endpoint (x_2, y_2) through some trigonometry:

$$(x_2, y_2) = (x_1 + l\sin(\theta), y_1 + l\cos(\theta))$$

Note that the $\sin(\theta)$ and $\cos(\theta)$ are flipped from the conventional use of polar coordinates because our 0 degree angle is actually pointing directly up, and our measurement of angle is clockwise.

Once we have the two endpoints, we simply draw it in as an SVG, rinse and repeat for all branches!

5 Other and To-be-implemented

5.1 Improving Growth Complexity

As mentioned previously in this paper, it is possible to greatly reduce the time complexity of growing trees from $O(nx)$ to $O(n)$ amortized, where n is the number of branches in a tree and x is the number of days that it will grow by.

The best way to describe this scenario is with an example. Suppose we have a tree with 4 branches (which must have IDs 0, 1, 2, and 3) and wish to grow the tree by 3 days. Currently, the algorithm will essentially call a `grow_branch()` function like this:

```
grow_branch(0), grow_branch(1) ... grow_branch(3),
grow_branch(0), grow_branch(1) ... grow_branch(3),
 grow_branch(0), grow_branch(1) ... grow_branch(3)
```

for a total of 12 calls. If a branch needs to split or offshoot, then more calls to `grow_branch` appear. For example, if the first call of `grow_branch(2)` causes a split and creates branches with ID 4 and 5, the call queue will now look like

```
                grow_branch(3),
grow_branch(0), grow_branch(1) ... grow_branch(5),
 grow_branch(0), grow_branch(1) ... grow_branch(5)
```

and will continue like this. However, there is a way to optimize it by adjusting our length/width increase methods and create a `new_grow_branch` method that takes in both a branch argument and the number of days.

```
new_grow_branch(0,3),new_grow_branch(1,3)...new_grow_branch(3,3)
```

medskip This new method would lump the length/width increase method (which can easily be generalized to a multi-day algorithm) to only run once. The only issue now, however, is how to determine splitting/offshoots and **still know what day they occurred**. After all, if branch 2 split into ID 4 and 5 on day 1, it would require us to add `new_grow_branch(4,2)` and `new_grow_branch(5,2)` to the queue, but if it happened on day 2, it would require us to add the same thing with the second argument being 1. Fortunately, there exists a less-simple tweak (in theory, I haven't implemented it yet) that can more precisely use a probability distribution function to pinpoint an exact day even when computing in groups like this.

Because these tweaked algorithms should have very similar runtimes to existing ones, it follows that this new algorithm would have run 6 times and the old one would have run 16 times. In theory, new branches will very rarely split, but it is still possible. So, in reality, this algorithm may actually be $O(n \cdot \alpha(nx))$ amortized, where α is the inverse Ackermann function.

5.2 Width of Branches

Currently, all branches have the same width, which of course isn't very realistic. In a future version, branches will be given a width and will appear as such in the application. Width will likely follow a similar schematic to length, albeit with different variables.

5.2.1 Da Vinci's Branching Rule

During the Italian Renaissance, Leonardo da Vinci once noted (without much rigor) that "*all the branches of a tree at every stage of its height when put together are equal in thickness to the trunk.*"

Eventually, this was generalized to saying that when a parent branch split into two or more children branches, the sum of the surface areas of the children branches would equal to the surface area of the parent branch. In other words, given a parent branch b with width w that splits into children branches $b_1, b_2, \ldots b_k$ with widths $w_1, w_2, \ldots, w_k$ respectively for some $k \in \mathbb{N}^+$, it follows that

$$w = \sqrt{\sum_{i=1}^{k} w_i^2}$$

Equation 5.1: Da Vinci's Generalized Branching Rule

which will, of course, prove extremely useful when width and splitting are implemented.

5.3 Offshoots v. Splitting: Is there a difference?

When a tree has an offshoot, there is a question to be asked: is the offshoot not just the result of the branch splitting into two, except that one of the resulting split branches simply has a deviation angle of nearly zero?

This question may prove interesting and may greatly impact the future of this project. As of now, however, both will remain integral parts of the project.

5.4 The Leaf

In this document, we have left out a very essential part of many trees, and these are leaves. Leaves are produced on the end of many if not all branches that have either not split after the branch has reached just a few days old. At this time, I have left leaves out of the picture due to them likely being another model that needs to be added to the project.

5.5 Other Types of Trees

All information provided on tree growth has been targeted at very specific tree types: those found in many North American suburban streets, such as breeds of oak and maple. Other trees will have different properties, albeit being produced and structured in a similar manner to other trees (just with different variables).