Alec Brownlie
Juan Martinez

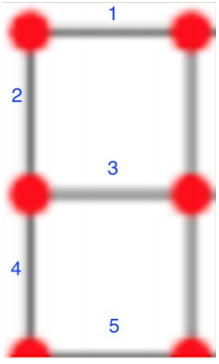
CS 454 Final Project

Problem Statement

A spanning tree is a connected, undirected graph with no cycles and is widely used in various applications including telecommunication networks, piping, and wiring connections. For a given graph G_n on a 3 x n grid, our task is to determine the number of spanning trees in G_n. We have also included two other sub-problems. One is a simple acceptance test of an encoded spanning-tree string, and the other being how many spanning trees are accepted given some specified edges.

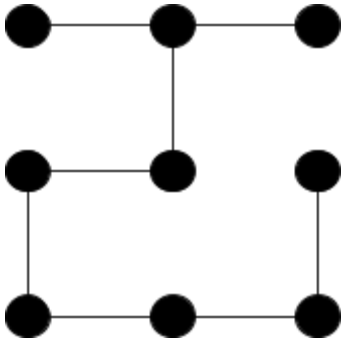
Solution Outline

To find the number of spanning trees in a 3 x n grid, a DFA that accepts strings that encode spanning trees must first be created so we can count the number of strings accepted by the DFA. First, our team needs to determine how to encode a given spanning tree. Consider the following 3 x 2:



We can encode any spanning tree into a binary string where each character is an edge position that follows the order above, where 1 represents the edge is connected, and 0 represents no connection. For example, the encoded string “10101” means in position 1, there is a connection,

and in position 2 there is no connection, and so forth. To signal the ending we use "XX" where X = {0, 1}. This is for the last two vertical edges. The first X for the top one, and second X for the bottom one.

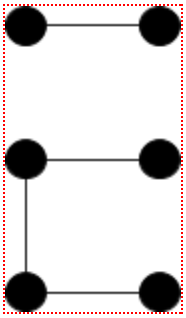


For example in the spanning tree above in a 3 x 3 graph, it would be encoded as:

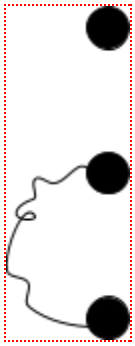
10111 11001 E01

The main problem and subproblems all use the same DFA. Our states in the DFA take the form of (a, b, c, d) where a = {0, 1, 2, 3} representing the components, b = {0, 1} if the top row connects to the middle row, c = {0, 1} if the top row connects to bottom row, and d = {0, 1} if the middle row connects to the bottom row. Each combination of a, b, c, and d is a state. So we have 4 * 2 * 2 * 2 = 32 states + a fail state + an accepting state = **34 states** in total. This is our **Q**. Our **Σ** = {00000, 00001, 00010, ….., 11111} so every possible 3 x 2 tree which is $2^5 = 32$ possible trees. $\mathbf{q_0}$ is the state (0, 0, 0, 0) indicating we have not read any input. Our **F**, or accepting state is ("ACCEPTING"). **δ** is given a state in Q and a symbol in Σ, the resulting state is the given state updated by the symbol and its edges and components. So each of our 34 states has 32 transitions.

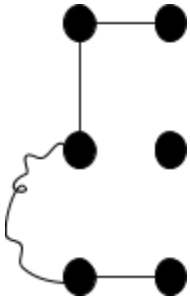
Assume our input is 10111 11001 E01, from the image above. We start at state (0, 0, 0, 0). We



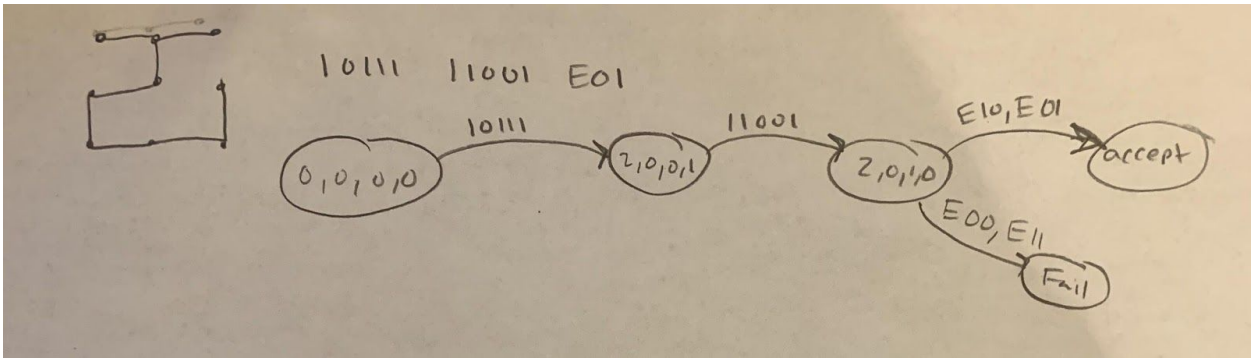
then take the first block of 5 characters, 10111, represented on the left. We then figure out the number of components, in this case, 2. Then we check if the top row connects to the middle and it does not. Only the middle row connects to the bottom one. So the resulting state is (2, 0, 0, 1), two components and only the middle and bottom row connect. We can imagine (2, 0, 0, 1) as the bottom left



image, a squiggly line meaning its connected through potentially various stages. The red borders just distinguish each image. So all these states represent what we have seen, and aid us in determining a cycle and if it is accepted or not by our DFA. If the next input was XXX1X, we would have a cycle and (2, 0, 0, 1) on any input of form XXX1X, would go to the fail state since the bottom left vertical edge is a 1. Continuing with our example, (2, 0, 0, 1) on



11001 goes to (2, 0, 1, 0) since, if you look at the next image to the left, there are 2 components, the single point, and the other connected part. The top row does not connect to all of the second row, and neither does the middle with the bottom. However, the top **does** connect all the way to the bottom row. So now the only way to fail is E11 creating a cycle. The middle point still needs to be touched so we can not have E00 because that does not connect. Only E10 and E01 connect the graph, making it be accepted by our DFA.



Data Structures and Algorithms

For the DFA, we went with a more object-oriented approach. We have a DFA class, a State class, and a Transition class. The DFA class has an array of States, an array of accepting states, even though only one state is accepting, and a start state. Building the DFA requires various functions. A generate function generates all the states needed, in total 34. Every State has a label [a, b, c, d],

and ID, and an array of Transitions. Transitions have State p, symbol e, and resulting State q. Generate then calls link_states which links states and sets transitions. The set_transition function was probably the most tedious, since based on what is passed, which is a symbol, you have to determine what State q should be based on what p is. We had to this somewhat manually thus is the longest part of the code. We have to check whether some edges are 1 or not, check whether 2 edges are together or, or even check if 3 edges are 1 but a certain edge is not. We had to work these out by hand before coding it. It is pretty quick since there are not many states, and we can even reduce the number of states! Some states we do not need like (0, 1, 1, 1) which means 0 components but everything is together.

Moving on to the problem and subproblems. The original problem states that for a given graph G_n on a 3 x n grid, determine the number of spanning trees in G_n. We used the same dynamic programming algorithm we used in Project 1. To solve, for example, n = 5, we first do the base case, where there is only one accepting state. Next, we determine how many states have a transition to the accepting state, and count those transitions. We basically count the number of

accepting spanning trees when n = 2, then use that to figure out n = 3, up to n. It works very fast, as opposed to a brute force approach.

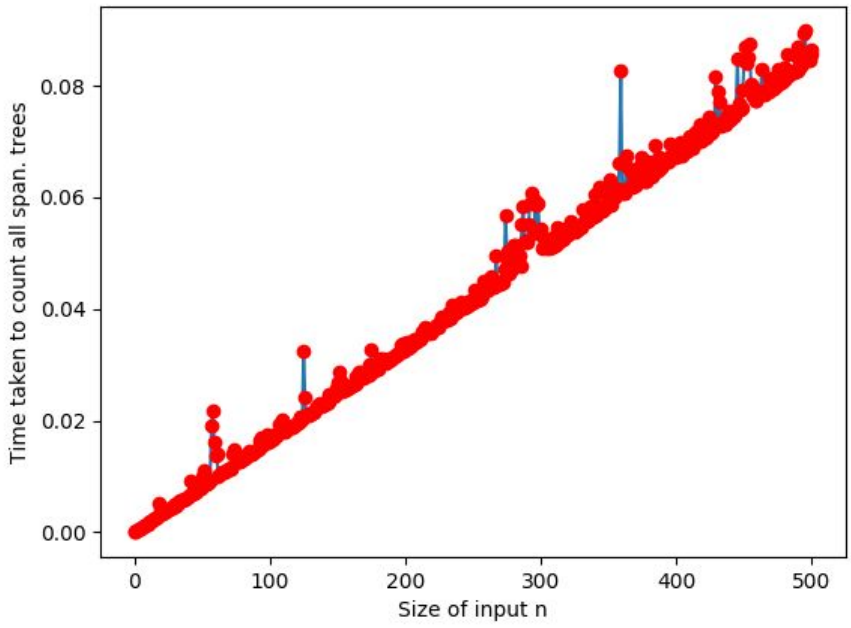
The first sub-problem, determining if an encoded tree string is a spanning tree is fairly simple. We first remove spaces, then loop over every 5 character blocks, for example, [10101] [01101], but not every substring of 5 characters. We then determine the current state by converting that block to decimal since it is only 1s and 0s, and returning the transition associated with that decimal value. For example, if we have a state p and symbol is 11111 = 15, then we return transition[15].get_q(). If the label of the new state is the fail label, return false, otherwise, continue looping. Once we reach an E character we append the last two edges, EXX, and return the appropriate state, either fail or accepting.

The second and last sub-problem states how many spanning trees are accepted given some specified edges, and was not solved efficiently due to limited time. We first input the encoding with specific edges being a 1, the rest are Xs. EXAMPLE: 1XXX1 XX1XX X1X1X 1XXX1 E1X. We then determine the n by removing the spaces, subtracting 3, dividing by 5, and adding 1. We then create a 2d array with n sections. We then take the first block, ex: 1XXX1 and append to sections[0], the first section, any symbols of the form 1XXX1 so 10001, 10011, …., 11111. We do this by keeping track of the indexes where the character = 1, then compare every symbol to it to see if it matches the 1s in the appropriate place. Then we move on to the next section on so on including the ending. Once we have our sections, we create every possible combination from each section. For example [[1,2,3],[4,5,6],[7,8,9,10]] ->

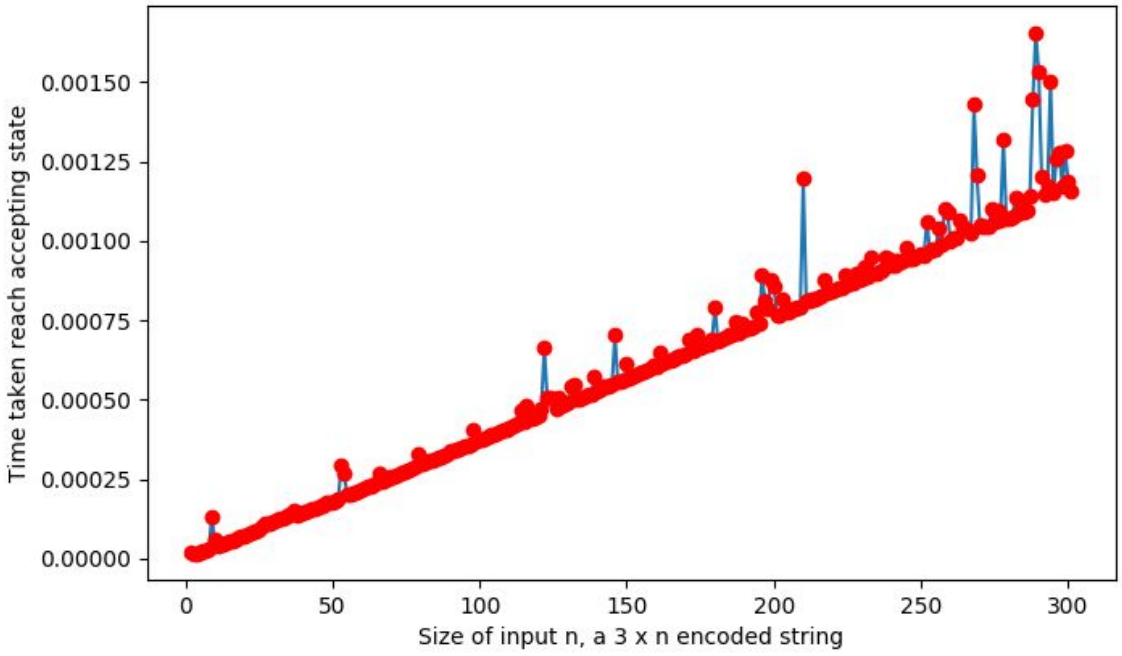
[[1,4,7],[1,4,8],...,[3,6,10]] but using the symbols. Once we created a list of all possible combinations, we check if the combination is in the DFA and keep track of the count. The problem arises if a user enters all Xs, this will test ALL combinations which are really slow since it's practically brute force. It is exponential in the worst case. When a user enters only Xs, the user should not do that since having only Xs is the same goal of the first problem, which the first option solves much faster. For example, having XXXXX XXXXX EXX is the same as solving the number of acceptable spanning trees for a 3 X 3 graph. Obviously the more 1s present the better. It does not exclude edges, for example, 0X011.

Time Complexity Analysis

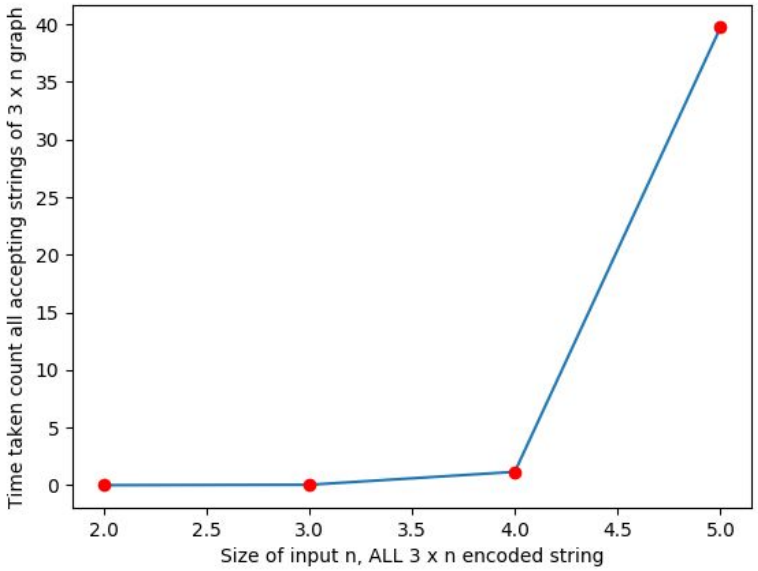
The original problem, using dynamic programming, appears to be solved in O(n). We have a double for loop, the outside executing n times, and the inner one executing 34 times, the size of the DFA. O(n*34) = O(n). We can view the chart below to see that it is linear, with some spikes. Y-axis is time taken to determine the amount of all accepting spanning trees of a 3 x n graph.



The check string algorithm to determine if an encoded spanning tree is in the DFA is also O(n) since we just loop through all the characters in the string.



The last algorithm, using specific edges, is $O(32^n)$. Worst case we make every combination possible in a 3 x n graph, so 32 options x 32 options ….. n times. When n = 5, it took 40 seconds



Contributions of Each Member

Alec hand-crafted the first DFA and wrote the program to generate the DFA on the fly and contributed many of the helper functions. Juan helped with the overall design and implementation of the main functions, as well as determining how the strings should be encoded.

Summary

We tried our best to understand the problem and spent countless hours determining the inner workings of the DFA. Building the DFA both by hand and programmatically took the majority of our time. Overall, if we had more time and mentoring in solving this problem, I'm sure our team could have found a solution that is better than exponential in the specific edge problem. Also, we can reduce the number of states for future work.