Controlled Complexity

Carson Syberg
Department of Electrical and Computer Engineering
University of Colorado at Boulder
Boulder, United States
carson.syberg@colorado.edu

***Abstract*—This paper is an exploration of a logic problem involving frogs and waterlogged lily pads. The object of the problem is to stack all frogs onto a single lily pad, however there are rules to how stacks of frogs may move. A frog stack can move as many pads as there are frogs in its stack. A single frog can move once to its left or its right. But the frogs must land on a pad that is not empty. When a frog lands on a pad, the stack size increases. There can also be light frogs, who cannot be jumped onto, and lead frogs, who cannot jump onto others. To solve this problem, trees are generated recursively to map out all possible movement paths of the frogs. The leaf nodes of this tree are end states where the frogs no longer have valid moves, and the leaf nodes that show all frogs on one pad are solutions. By traversing up the tree from the solution nodes, we can view each possible solution path to the given system of frogs and waterlogged pads.**

Keywords—complexity, recursion, algorithms, logic problem

I. Motivation

The main goal of this project is to solve any given system of frogs and lily pads. The user inputs how many pads they would like, then decides if the pad is waterlogged, has a light frog, has a lead frog, or defaults to a regular frog. For someone playing this logic game, this program could be useful to gauge the difficulty of certain frog problem layouts. By running this algorithm on their frog layout first, they could possibly avoid looking for a solution to an impossible problem. The other goal of this project is to analyze how problem complexity changes as different variables are added. With waterlogged pads, light frogs, and lead frogs, the amount of possible movement paths is limited, causing trees to generate shallower and time complexity to be reduced, but at the cost of reduced numbers of solutions.

II. State-of-the-Art

This logic problem is relatively unexplored except for a paper by Christian Woll about turning unsolvable systems of waterlogged pads and frogs into solvable systems and a video exploring the problem by YouTube channel *Numberphile* [1] [2]. Neither of these examples give a clear way to solve any system of frogs, but rather they explore the possible cases and what may be unsolvable. This project does not change cases to make them solvable, it only tells that the problem is unsolvable, but it is also capable of giving every possible solution to any frog problem given to it. This project also adds the complication of light and lead frogs that those sources do not include.

III. General Explanation of Algorithm

A. Node Class

The Node class was created to be able to implement tree data structures. It contains a struct called Node with the following fields: depth, locData, lightLoc, leadLoc, children, parent, original. Depth keeps track of what layer of the tree the node is on. LocData, lightLoc, and leadLoc keep track of the current layout of frogs. Children and parent store information for tree connections. Lastly, original keeps track of which starting tree the node is in for use in printing.

The Node class includes the functions bfPrintTree, printUpPath, generateTree, and getLeafNodes. The function bfPrintTree prints the tree given a root node breadth first. The function printUpPath prints the child then the parent starting from a given node until the parent is null. GenerateTree is used for creating the nodes and linking them to create the tree. Lastly, getLeafNodes returns a list of all the leaf nodes of the tree.

B. Node Class Functions

The generateTree function takes lists padFill, lightFill, and leadFill, as well as the value numPads. These lists are symbolic representations of the frog arrangements: padFill has the number of frogs at each index, light and lead Fill have a 1 or a 0 whether an index contains a light or lead frog or not. Each index, or pad, is then looped through, and if padFill's value at that index is not zero, i.e. there are frogs on the pad, then the valid possible moves of that stack of frogs are calculated and added recursively as children of the current layout of frogs. This continues until all possible valid movement paths have been generated, which happens when a layout of frogs no longer has any valid possible moves and no children are generated, this being the base case of recursion that returns nothing.

The getLeafNodes function traverses the tree recursively from a given node fully until reaching all the leaf nodes. Upon reaching a node with no children, the node is added to a list and returned.

The bfPrintTree function traverses the tree recursively layer by layer. Using a queue to implement breadth first traversal, each level of the tree is printed one after the other.

The printUpPath function traverses a single path of the tree from a node to the root. Using the parent pointers of the nodes, each node is printed until the root is reached. This function is

useful for visualizing the solution paths and verifying that solutions found are valid within the rules.

C. *Recursive Definiton of Valid Move Tree*

- A single node is a full valid move tree.
- If r is a node, and t_1, t_2, …, t_n are valid move trees from layout r, then the tree with r as a root is a valid move tree.

Base Case: no valid moves from current node, all children made null

Recursive Case: valid moves from current node, make children all possible valid move trees from current node

D. *Main Functions*

The makeAllMoves function takes numPads, padFill, lightFill, leadFill, and realNumFrogs as input. The frog layout being passed to this function is one that has had no moves made on it yet, but it could be any possible arrangement of frogs, light frogs, lead frogs, and waterlogged pads. It uses the same general form as the generateTree function but creates individual trees for each different possible first move and upon finding a valid move, calls the generateTree function to create all its roots' branches. Each of these roots is then added to a list for later processing.

The getUserInput function takes no inputs. It first asks the user to enter the number of pads, then it creates padFill, lightFill, and leadFill lists of that size. For each index, the user is asked if they would like to make it a waterlogged pad, if they do not, they are then asked if they would like to put a light frog on the pad, if they do not they are then asked if they would like to put a lead frog on the pad, and if they do not want that, then the frog on that pad is considered default. If they answer yes to waterlogging the pad, each of the 3 lists gets a 0 in its index, and if they answer yes to a light or lead frog, then the lightFill or leadFill list gets a 1 in that index and padFill gets a 1 in its spot since a frog is on the pad.

E. *Preconditions and Postconditions*

To ensure the correctness of the program, preconditions and postconditions are included in the code. The precondition is that each spot of padFill has either a 1 or a 0, each spot in lightFill has either a 1 or a 0, and each spot in leadFill has either a 1 or a 0. The post conditions are that padFill must have the same sum of all elements as before, and lightFill and leadFill must have either a 1 or 0 in each of their spots. As the function loops recursively, a loop invariant is the sum of elements in padFill.

IV. Hypothesis

After developing the algorithm to solve these systems of frogs, the next step was analyzing the time complexity of the algorithm. My initial hypothesis was that the frog systems with the worst time complexity for my algorithm would be those that have the most possible moves to make, i.e. the ones with no waterlogged pads, light frogs, or lead frogs, and that the best case time complexity for my algorithm would be the most difficult arrangements that had no possible moves from the start.

A. *Big O Complexity*

With an arrangement of n regular frogs and no waterlogged pads, there are 2n-2 possible first moves to make, each internal frog could move left or right, and each of the two external frogs could move one space inward. After this first move, there is no exact formula for determining the number of children, as the different arrangements produce different numbers of possible moves, but my hypothesis is that it will be some type of logarithmic complexity. The overall complexity for generating the trees for a given frog layout would be O(2n-2 * log(n)) as it would have to generate recursively 2n-2 times. To test the time complexity, multiple different cases of frog problems, some regular with changing amounts of frogs, some with waterlogged pads, and some with light and lead frogs, are timed to determine how long it takes for the movement trees to generate and solutions to be found. The more complex a problem, the less solutions there will be and the less time it will take to find them with brute force.

The best case time complexity is O(2n-2) for arrangements that have no possible first moves, as each of the n pads must still be checked to see if any first moves are possible, but since no moves will be possible, there would be no added logarithmic time complexity for generating trees.

V. Experimentation

A. *Regular Case*

To test the complexity of the regular case, code was written that generates regular arrangements of frogs in varying numbers and calls makeAllMoves on each of the arrangements. Each value of padFill is made 1, and each of the values of lightFill and leadFill are made 0. The time to generate all trees is measured for each arrangement of n frogs. By graphing the time to generate versus the number of frogs, the Big O complexity can be approximated.

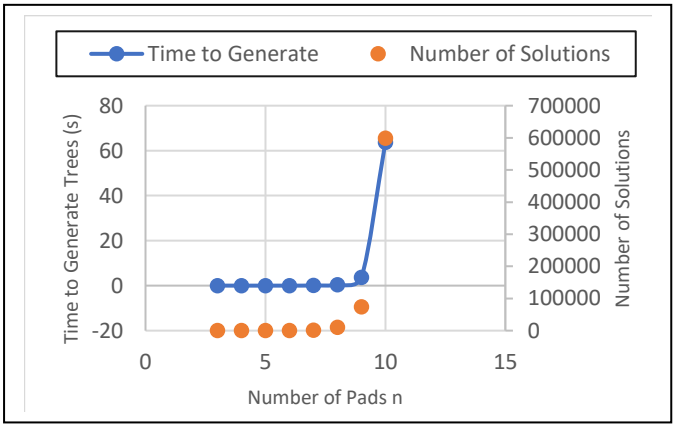

Figure 1: Time to generate trees and number of solutions for n regular frogs on n pads.

For n = 10 pads, the algorithm took 63.7 seconds to generate 598,878 solutions, and based off the graph, the hypothesis for how this algorithm would grow is incorrect. It actually grows exponentially, which means it has poor performance as the arrangements of frogs it is given grows larger without growing more complex.

B. Waterlogged Case

Testing with a waterlogged pad on either end is trivial as this just reduces the problem to a regular case, but a waterlogged pad in an inner spot is more complicated. To test how complexity changes for waterlogged inner pads, arrangements of frogs are created to represent a single waterlogged pad in each inner spot, and trees are generated for each of those arrangements. The time to generate the trees is again measured to see what effect this single waterlogged pad had on the time complexity. This procedure was then repeated with more waterlogged pads, in different combinations of being next to each other and spread out among the inner pads.

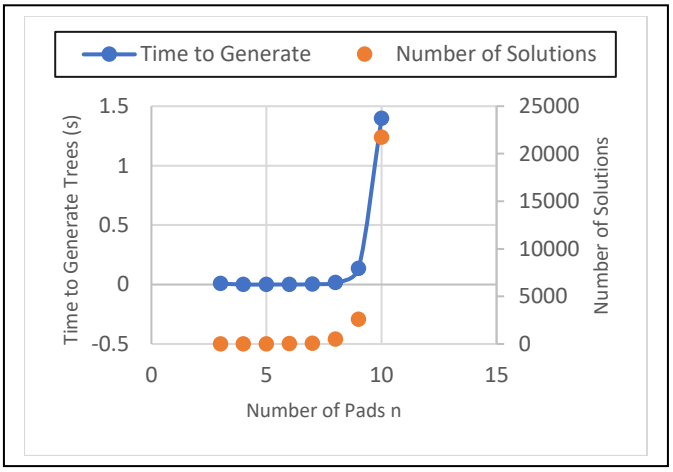

Figure 2: Time to generate trees and number of solutions for n-1 frogs on n pads with one waterlogged pad in the center spot.

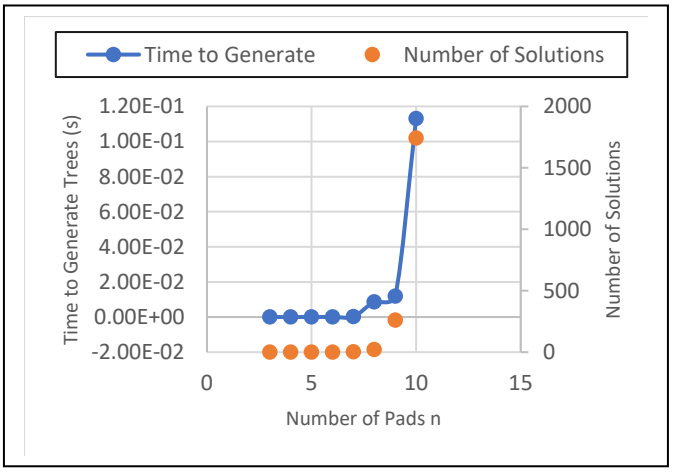

Figure 3: Time to generate trees and number of solutions for n-2 frogs on n pads with two adjacent waterlogged pad in the center spots.

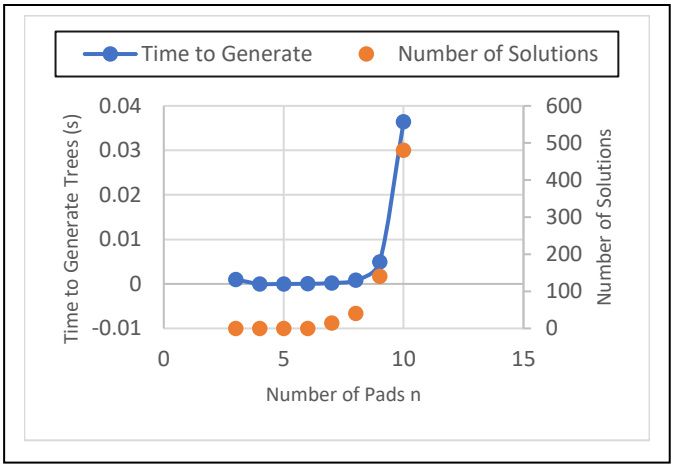

Figure 4: Time to generate trees and number of solutions for n-2 frogs on n pads with two waterlogged pads in the center spots seperated by a frog.

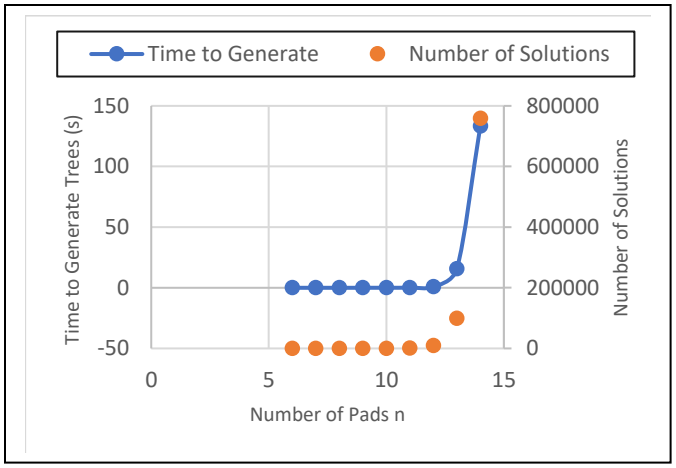

Figure 5: Time to generate trees and number of solutions for n-3 frogs on n pads with three adjacent waterlogged pads in the center spots.

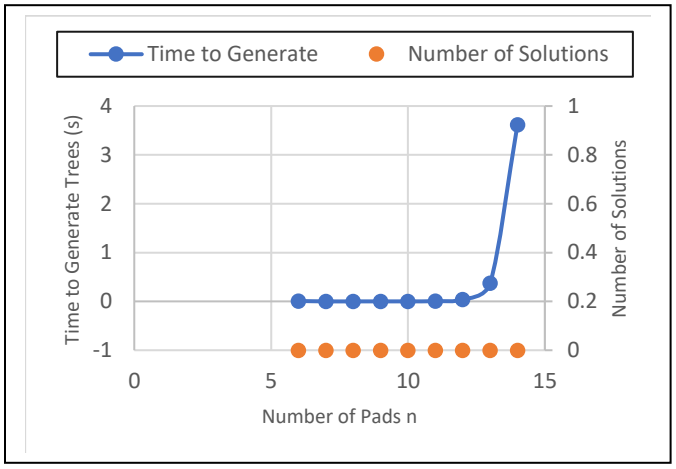

Figure 6: Time to generate trees and number of solutions for n-3 frogs on n pads with three waterlogged pads in the center spots seperated by frogs.

The time to generate trees for n = 10 pads with a single water logged pad in the center, as shown by Figure 1, was 1.38 seconds and created 21,745 unique solution paths. This single waterlogged pad reduced the time complexity from the normal

case at n = 10 pads by 97.8% and reduced the number of solutions by 96.4%.

Two adjacent waterlogged pads in the center at n = 10 pads reduced the time complexity by 99.8% to 1.38 seconds and the number of solutions by 99.7% to 1742. Two waterlogged pads in the center seperated by a frog reduced the time even more by 99.9% to 0.0364 seconds and the number of solutions by 99.9% to 480.

Three waterlogged pads reduced the time complexity and number of solutions even more, but comparing adjacent waterlogged pads and non-adjacent waterlogged pads showed that non-adjacent pads cause more complex layouts. For n = 14 pads, three adjacent pads in the center had time complexity of 133.5 seconds and generated 758,344 solutions, while three non-adjacent pads in the center had time complexity of 3.62 seconds and no solutions. The more complex the problem, the less solutions and less time it takes to traverse all possible paths, so it follows that non-adjacent pads are more complex elements of the problem.

C. Light Frog Cases

To see the effect of light frogs on time complexity, frog layouts with n pads are created that have the light frog on either end, or in an internal spot. The time to generate movement trees is then measured for these differing arrangements.

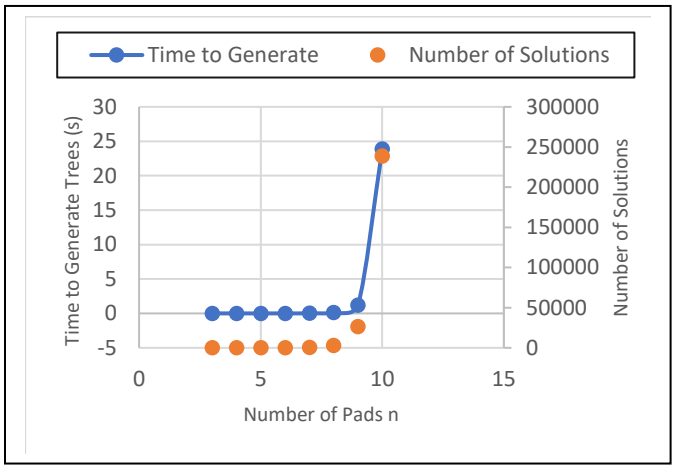

Figure 7: Time to generate trees and number of solutions for n frogs on n pads with one light frog on the first spot.

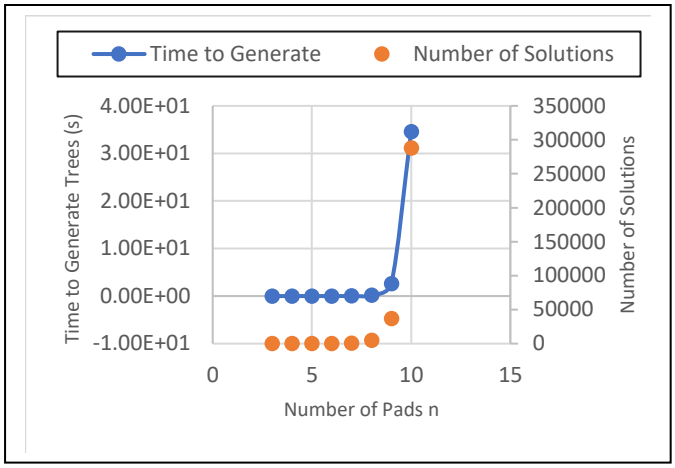

Figure 8: Time to generate trees and number of solutions for n frogs on n pads with one light frog on the middle spot.

Adding a single light frog in an outer spot reduced the time it took to generate the trees for n = 10 pads by only 45.7% to 34.6 seconds and the number of solutions by 51.9% to 288,138.

Adding a single light frog in an inner spot reduced the time to generate trees for n = 10 pads by a further 62.5% to 23.9 seconds and the number of solutions by 60.1% to 238,826, meaning that a light frog in the center makes a more complicated problem than a light frog in an end spot.

D. Lead Frog Cases

To see the effect of lead frogs on time complexity, frog layouts with n pads are created that have the lead frog on either end, or in an internal spot. The time to generate movement trees is then measured for these differing arrangements.

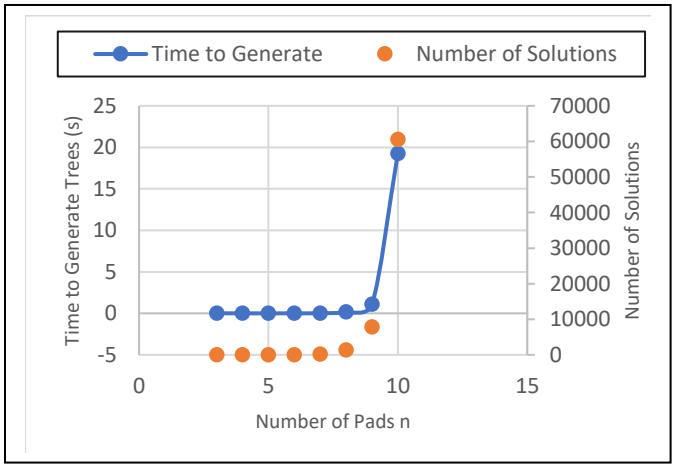

Figure 9: Time to generate trees and number of solutions for n frogs on n pads with one lead frog on the first spot.

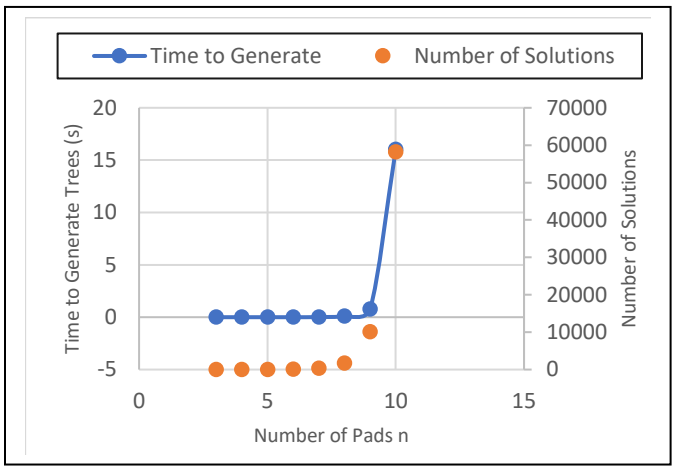

Figure 10: Time to generate trees and number of solutions for n frogs on n pads with one lead frog on the middle spot.

Adding a single lead frog in an outer spot reduced the time it took to generate the trees for n = 10 pads by 69.7% to 19.3 seconds and the number of solutions by 89.9% to 60,507.

Adding a single lead frog in an inner spot reduced the time to generate trees for n = 10 pads by a further 74.9% to 16.0 seconds and the number of solutions by 90.3% to 58,212, meaning that a lead frog in the center makes a more complicated problem than a lead frog in an end spot.

The addition of a lead frog instead of a light frog decreased the number of solutions and time complexity by almost double the amount a light frog decreased it, leading to the outcome that lead frogs create more complicated arrangements than light frogs.

Both lead and light frogs cause less reduction in time complexity than a waterlogged pad. A waterlogged pad added in the center of n = 10 pads only took 1.38 seconds to generate while a light and lead frog took 34.6 and 23.9 seconds, respectively. So lead and light frogs cause less complex problems than waterlogged pads when placed in the center of arrangements.

E. Combination Cases

The more constraints on where the frogs can move, the less paths will be generated, so these combinations of waterlogged pads, light, and lead frogs should lead to small numbers of solutions and low time complexity.

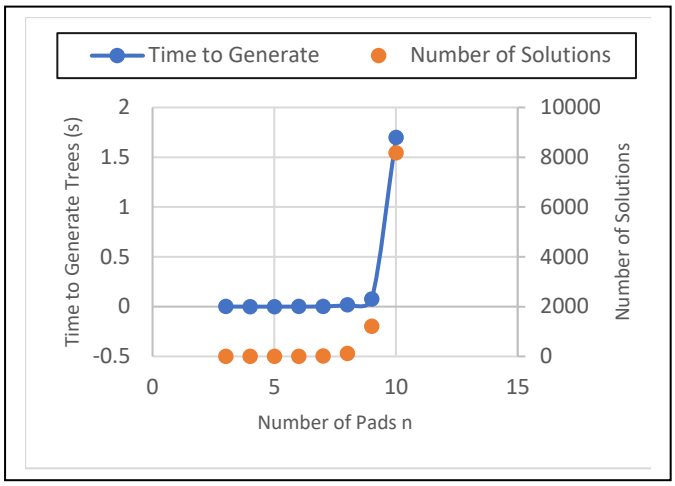

Figure 11: Time to generate trees for n pads with one waterlogged pad in a central spot and one light frog in an adjacent central spot.

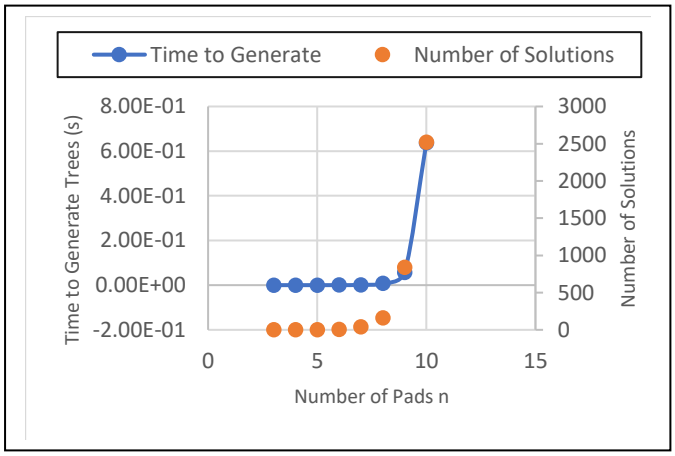

Figure 12: Time to generate trees for n pads with one waterlogged pad in a central spot and one lead frog in an adjacent central spot.

In combinations of waterlogged pads with lead and light frogs, Figures 11 and 12 show that lead frogs cause less numbers of solutions to be found and in less time, meaning that in combination as well, lead frogs cause more complicated arrangmenets than light frogs. More complicated being equated with less possible solutions.

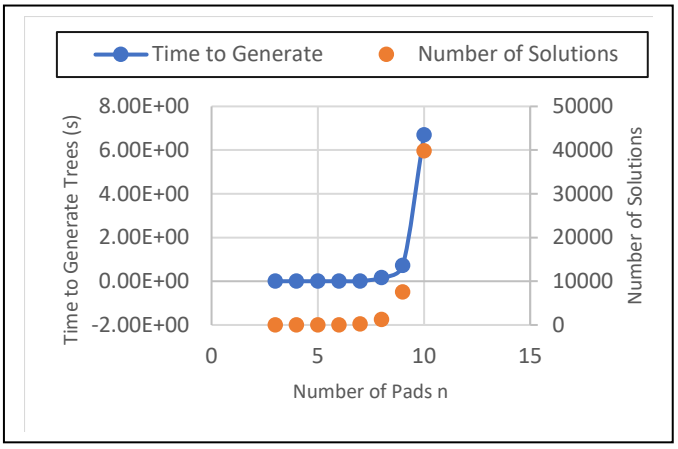

Figure 13: Time to generate trees for n pads with one light frog in a central spot and one lead frog in a central spot adjacent to it.

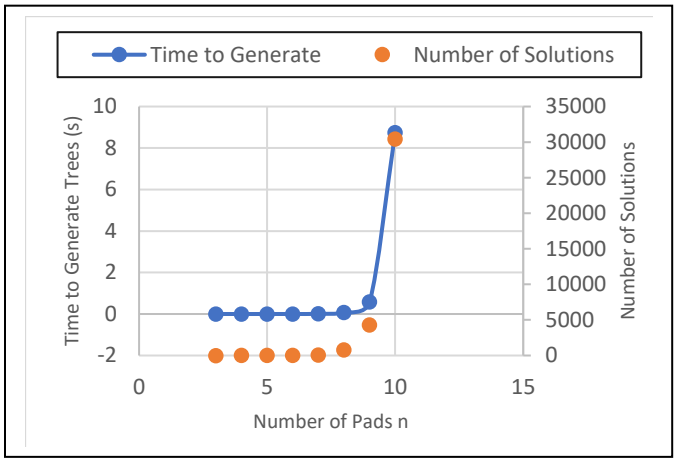

Figure 14: Time to generate trees for n pads with one light frog on an outer spot and one lead frog on the other outer spot.

As shown in Figures 13 and 14, the arrangments with a light and lead frog seperated on the outer spots rather than adjacent on the central spots cause less numbers of solutions to be generated but in more time. This has not been the trend with the rest of the data, as all the other data has had the same relationship that when less solutions are found, it takes less time. What could have happened to cause this, is more final arrangments being generated that were not valid solutions, as no matter what, the size of the tree or the number of leaf nodes it has would be directly related to the time to generate the tree. But still, complication of the problem in this paper is equated to lower number of solutions, so the arrangmenet with the frogs on the outside spots would be more complicated with 30,426 solutions to the other arrangment's 39,816.

VI. Conclusion

As shown in all the graphs, the Big O complexity hypothesized before is not correct. The actual complexity of the algorithm used for generating movement trees is exponential. Depending on the complications added such as waterlogged pads, lead, and light frogs, the exponential graph shrinks and grows, but it remains growing exponentially all the same. This means that this algorithm will not work well for large arrangements, as the time it takes to generate all those solutions grows too large to be worth it. However, this algorithm does work well for smaller more complicated arrangements with low numbers of solutions and is feasible to use for these types of arrangements. In general, this will be efficient enough, as these frog problems are not generally done in incredibly large numbers.

The added complications that best shortened the time taken to generate trees were waterlogged pads. A single waterlogged pad in the middle for 10 pads created only 21,745 solutions while the same arrangement with a lead and light frog in the center created 60,507 and 238,826 solutions, respectively. The next best single complication to add is the lead frog, as it generated less solutions than a light frog. In combination cases, adjacent complications caused more solutions to be generated than complications that were spread out. For two waterlogged pads in the center of 10 pads, if they were adjacent, there were 1742 solutions, while if there was a single frog between them, there were only 480 solutions. For a lead and light frog adjacent in the middle of 10 pads, 39,816 solutions were generated, but for the same 10 pads when they were on the start and end positions, only 30,426 solutions were found.

So long as complications are added as the arrangements grow larger, this algorithm will keep up with solving them. But incredibly simple and large arrangements will not work well.

References

[1] Christian Woll, "Jumping Frogs – Challenge #2," 2017, mathpickle.com/project/jumping-frogs/

[2] *Numberphile*, "Frog Jumping – Numberphile," 2017 https://youtu.be/X3HDnrehyDM