Ben Campbell

December 4, 2021

Algorithms for Conway's Game of Life

This paper is a presentation of my Computer Science thesis project. I wrote a program to simulate John Conway's Game of Life, and I tried to design algorithms to simulate it in the most efficient way possible. First, I will explain the Game of Life and its rules. Second, I will explain how the simulator I designed works. Third, I will explain each of the algorithms I wrote in detail.

The Game of Life

The Game of Life is a cellular automaton created by John Conway in 1970 (Gardner, 1). It is a game that is played on an infinite grid . The cells on the grid can be either live (on) or dead (off), and the game can advance to the next state or "generation" based off of specific rules. Each state is determined by the state before and what cells are live. The rules are based off of the amount of "neighbors" to a cell that are live. The neighbors of a cell are all the cells that are adjacent, including diagonal to it. It is called the game of life because the rules are based off of birth, death, and survival from underpopulation or overpopulation. Here are the rules:

1. Underpopulation: any live cell with less than two neighbors dies.
2. Survival: any live cell with two or three neighbors lives on to the next generation.
3. Overpopulation: any live cell with more than three neighbors dies.
4. Reproduction: any dead cell with three neighbors becomes live.

These rules can be simplified to make them easier to calculate in an algorithm. If one considers what cells will be live in the next generation, the rules can be expressed in a simple if statement:

If a cell is live and has two or three neighbors

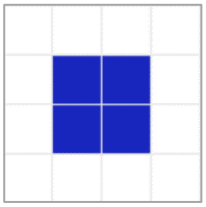
Or it is dead and has three neighbors:

Then it is live in the next generation.

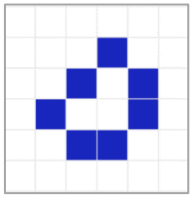
This is the format I use in my algorithms.

By following these rules, patterns can develop in interesting and unexpected ways. Sometimes they disappear in the next generation because all the cells die. Sometimes they grow large and other patterns break off of them. Sometimes they are simply still because no new cell are created and the ones that already exist survive into the next generation. Sometimes they form cycles that end up repeating.

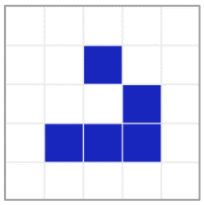
Some interesting or common patterns are given names. Here are some notable ones:



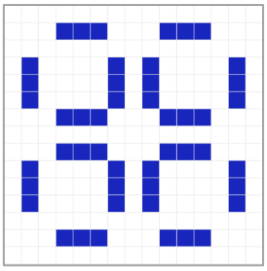
Block



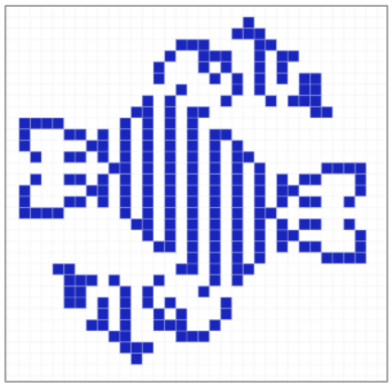
Loaf



Glider



Pulsar



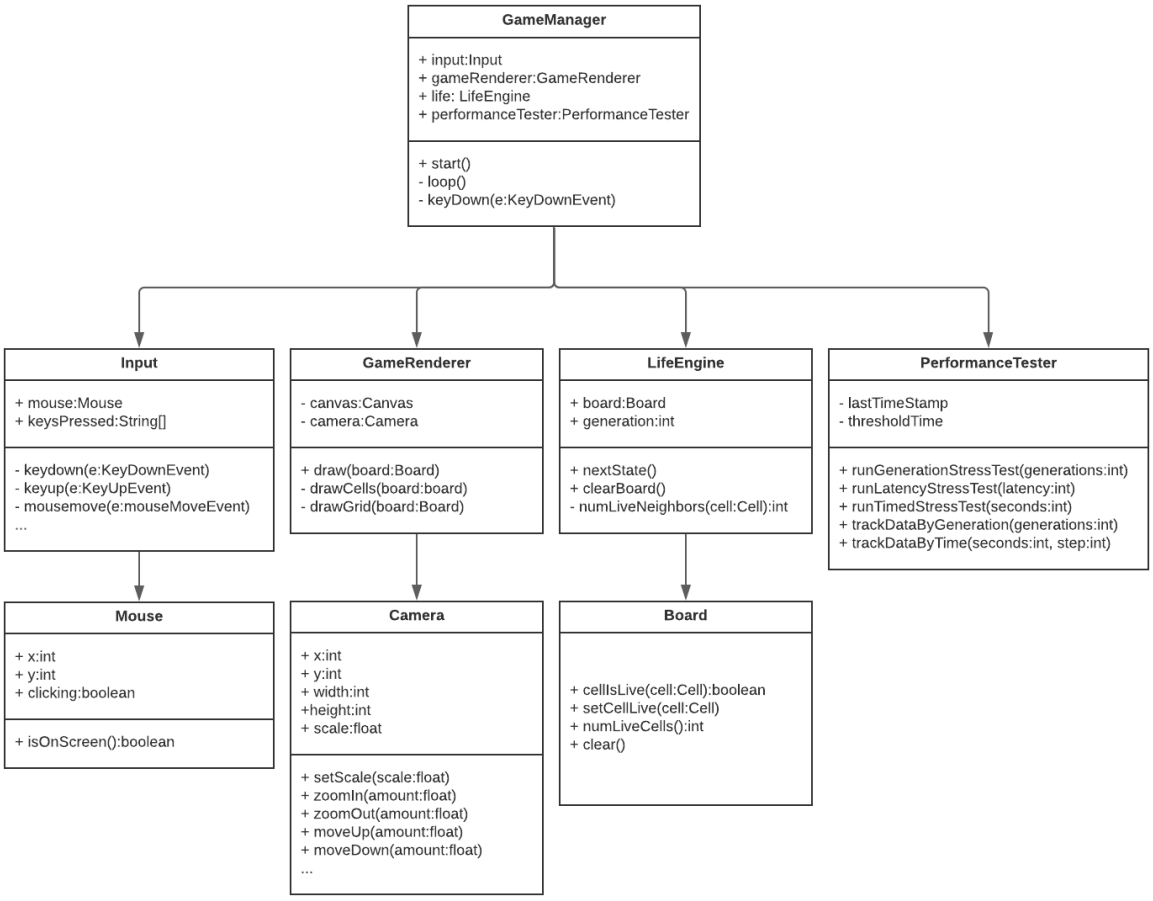
Spacefiller

The block and blinker patterns are still-lifes. They do not change in the next generation. The glider has the interesting property of repeating itself after four generations, but in a location diagonal to where it started. This gives it the ability to travel infinitely across the game space. The pulsar is one of the oscillator patterns. It repeats itself after three generations. One of the especially notable patterns in this project is the spacefiller. It causes a diamond of cells to form, and the number of cells drastically increases over time. I used it in

testing my algorithms so that it would be easy to tell which ones operate faster as the number of cells grows.

The Simulator

I decided to write my simulator in JavaScript because that is one of the languages I am most comfortable in, and it would be easy to put online so that others could try it out if they wish. It ended up having about 23 files and 1600 lines of code. To give a general idea of how it works, here is a diagram of its classes:



The main class is the GameManager. An instance of it is created when the page loads, and it contains the code necessary to initialize the simulator and all of its components and start its main loop. The game manager initializes instances of various helper classes to manage specific aspects of the game. It creates an instance of the Input, GameRenderer, LifeEngine, and PerformanceTester classes.

The Input class is in charge of keeping track of the state of inputs. It has a list of all the keys that are currently being pressed, and it has a reference to a Mouse object that keeps track of the location of the mouse and whether or not it is clicking.

The GameRenderer class manages drawing the game onto the screen. It has a reference to a canvas object on the html page. It has a reference to a Camera object that describes what part of the board the player is able to see. The camera has functions that allow it to move around or zoom in or out so the player can adjust it to see what they wish. When the GameManager tells the GameRenderer to draw the screen, it clears the canvas, draws a grid, and then fills in each of the cells that are live based off of the view that the camera can see. Sometimes it adds other elements based off of what type of algorithm is being used to help give details on how it works.

The LiveEngine class is the main focus of this project. It is the class that manages the Game of Life. It keeps track of what cells are live or dead, and runs the calculations to advance the game to the next generation. It has a reference to a Board class, which is a class used to keep track of what cells are live or dead. The LifeEngine and Board classes are abstract classes, and I created many different versions of them that use different algorithms to calculate the Game of Life.

Finally, we have the PerformanceTester class. This class is used to measure how well the algorithms run. It runs tests on the algorithm based off of how many generations it can calculate in a given time, or how long it takes to calculate a certain number of generations. It only focuses on running the algorithms as quickly as possible, and to prevent interference from the computer calculating other things, it tries to minimize other operations in the program that are being run. This means that when the PerformanceTester runs its tests, the window becomes unresponsive for at time. It does not register input, and it does not display anything on the screen until the tests are complete.

Another feature of the performance tester is recording data and exporting it in spreadsheets. For example, it could run an algorithm for twenty seconds, and record what generation it is on every ten milliseconds. This type of test is more biased than just running the algorithm itself because it must stop to record data often, but it is useful to get detailed information on how an algorithm performs over time.

The Algorithms

In the end, I designed six different algorithms. Since one of the greatest bottlenecks in performance was reading and writing to the state of the board, most of the "algorithms" actually use the same method for calculating the next generation. The main difference is the data structure they use to store the board. In order to explain the differences between my algorithms, first I'm going to explain a common measurement used in mathematics and algorithmic analysis called complexity.

Complexity

Complexity is one of the ways to judge algorithms. It is a way of measuring how fast an algorithm is based off of the number of operations it performs for a given input size. We typically express this as a function f(n) where n is the size of the input and f(n) is the number of operations the algorithm performs.

Since the number of operations performed often varies a lot depending on certain details and what the input is, we generally express complexity with Big O notation. Big O notation is a way of describing how a function ends up acting over time. If we take the function $f(x) = x^2 + 2x - 1$, over time the x^2 term ends up dominating, and the function starts acting a lot like x^2, as compared to x or x^3. Since it acts similar to x^2, we say that f(x) is $O(x^2)$.

Let's look at an example of a specific algorithm to help understand how complexity with Big O notation works. Imagine an algorithm that computes the sum of the numbers in a

list by adding each one to each other. If the list was 2, 5, 8, 2, 8, then the algorithm would compute 2 + 5 + 8 + 2 + 8. As you can see, it performs addition four times. If we make it abstract and assume the list contains n numbers $x_1, x_2, \ldots, x_{n-1}, x_n$, then the algorithm would perform addition n – 1 times. We would say the algorithm is O(n).

The particular complexity an algorithm has ends up mattering significantly as the data size increases. If an algorithm is exponential, such as $O(2^n)$, it is not considered a very good algorithm. On the other hand, if an algorithm is linear (O(n)), it is pretty good, and if it is constant (O(1)), it is incredibly fast. Typically algorithms with constant time are hard to design, but it is possible. Here is a chart of common complexities and how fast they are in terms of computation:

Name	Complexity
Factorial	$O(n!)$
Exponential	$O(2^n)$
Quadratic	$O(n^2)$
Linear	$O(n)$
Logarithmic	$O(\log n)$
Constant	$O(1)$

General Method

Before we go over the algorithms, there are a few details I'd like to explain. The algorithms that I designed actually operate off of the same general method that goes something like this:

```
For each live cell:
  Count the number of live neighbors, set the cell live in the next generation if
  it has two or three.
  For each neighbor:
    Count its live neighbors.
    If it is live, set it live in the next generation if it has two or three.
    If it is dead, set it live in the next generation if it has three.
```

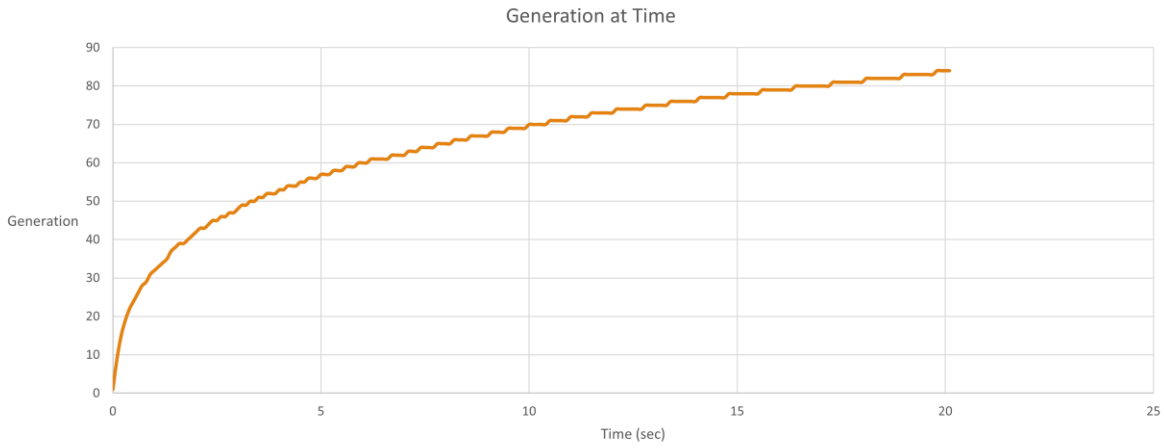
When I describe the complexities of my algorithms, I use the number of live cells for n. One thing to note about this algorithm is that it will never be faster than O(n) because it must perform an operation for each live cell. Even worse, it must count all the live neighbors of each live cell, and all the live neighbors of its neighbors. Since each cell has eight neighbors, this algorithm will never be faster than $O(8 + (8 * 8)n) = O(72n)$. The main differences between my algorithms are mostly their data structures, so the complexity of all of them is roughly O(72n). There are certain shortcuts and efficiencies we can make to do slightly better than this though.

Now that I have explained everything necessary, we can finally get to the algorithms. I present them in an order that builds upon itself so the former algorithms help explain how the ones after them work.

1. Cell List

The first algorithm I designed is the cell list algorithm. It was my first, rough attempt, and I definitely did not try to make it as efficient as I could. It's main purpose was to be simple. The cell list works by storing the live cells in an unsorted list of coordinates. For example, if the cell at position 1, 1 is set to live, then the algorithm would add the coordinate (1,1) to the end of the list. This is very fast and easy.

The main problem with this algorithm is the amount of time it takes to check whether a given cell is live or not. Since the list is unsorted, it must begin at the beginning and check each coordinate one at a time to see if it is the one it is looking for. It only knows if a cell is dead if it is not in the list, which ends up taking n operations of checking to see if each live cell is the one it's looking for. This algorithm has an O(n) lookup time, and since in my general method a cell must be checked 72n times, the algorithm ends up being $O(n^2)$ to advance the game to the next generation. Here is a graph of how it performs over time when starting out with a Spacefiller pattern:



It's hard to tell how good this is without something to compare it to, so let's look at the next algorithm!

2. Hash Map

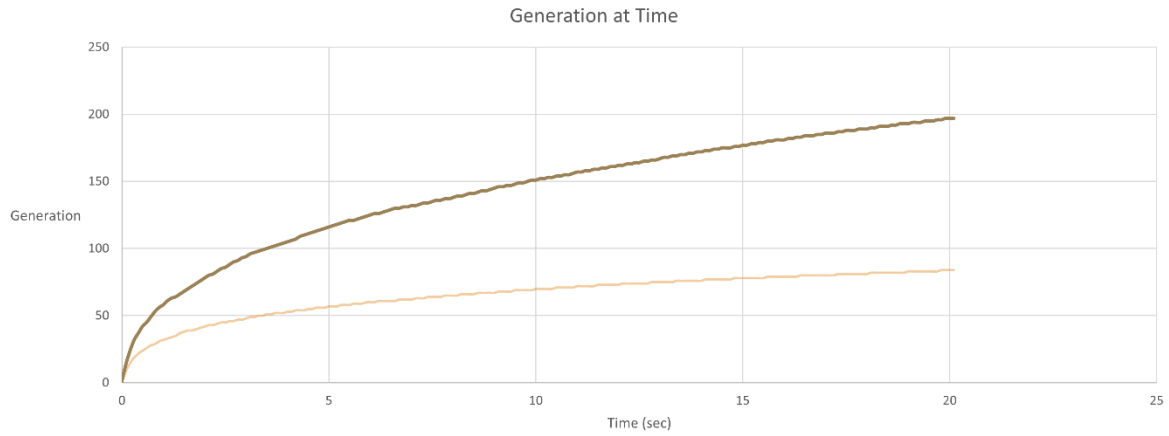
Since the main problem with the cell list algorithm is the lookup time, I thought the best approach would be to design an algorithm that had a faster one. There are many tried-and-true data structures that use sorting to add, remove, and find elements in a list very quickly. Since I wanted to be able to support a potentially infinite number of live cells, I thought a hash map would be a good data structure to try next.

A has map is a data structure that uses key-value pairs to store and look up data. The key is a string of characters, and the value is whatever kind of data you would like to store. The nice thing about hash maps is that you can use whatever string you want for the keys, and it has a potentially very fast lookup time. If you design it right, has maps can have a constant lookup time (O(1)) because they convert the keys into indices in an array, and then use the index to look up the value very quickly.

My has map algorithm works similarly to the cell list, but instead of using an unsorted list to keep track of live cells, it uses a hash map. If the cell 1, -5 is set to live, then it adds an entry to the hash map with a key of "1,-5" and a value of (1, -5). The value itself doesn't matter much – just the fact that the key is in the map. I didn't design my own version of a

hash map. JavaScript already had some that were good enough, and they would probably be faster than anything I could have written because they could use native code that would run faster on the computer than JavaScript. I decided to use the built in Map object.

The results were much better than the cell list algorithm, but they were still not as good as I was expecting. The hash map seemed to be a little over twice as good, but it certainly didn't seem to be operating at O(72n) complexity. Here is how it compares to the Single Array algorithm (once again, starting with a Spacefiller)



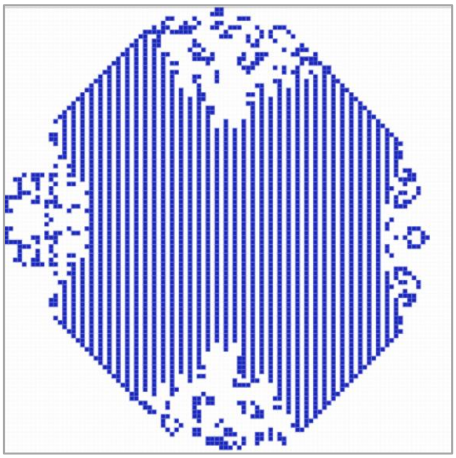
3. Single Array

Since hash maps didn't seem to work as well as I thought, perhaps using normal arrays would be better. Arrays would certainly have an O(1) lookup time. The next algorithm I designed used one giant two-dimensional array to store the state of the board.
This algorithm would have the strength of very fast lookup times, but it would have two big weaknesses. The first is that it would not easily know which cells were live and which ones were not. In order to find out, it would have to start at the beginning of the array and check *every single cell on the board*. This would be an $O(size^2)$ operation, where size is the width or height of the board. I was worried that this would be crippling to the algorithm and it would make it much slower than any benefit having a faster lookup time would give it.

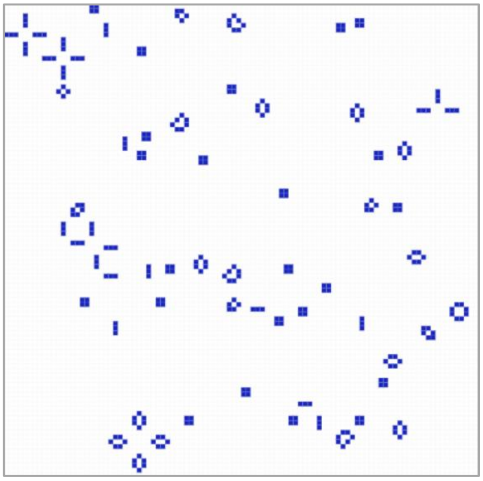
The second weakness is that it would not be an infinite size. In order to make it infinite, the array would have to dynamically increase in size somehow. This is an issue I

attempt to solve later, but for now the algorithm will have to stick to a boundary to its world. This means that the tests I run in it would have to be limited to a certain size, and it would quickly run out of space using the Spacefiller pattern I use for testing. The size of the board would compete with the speed of the algorithm. The smaller the board is, the faster the algorithm would run. The larger the board is, the slower the algorithm would run.

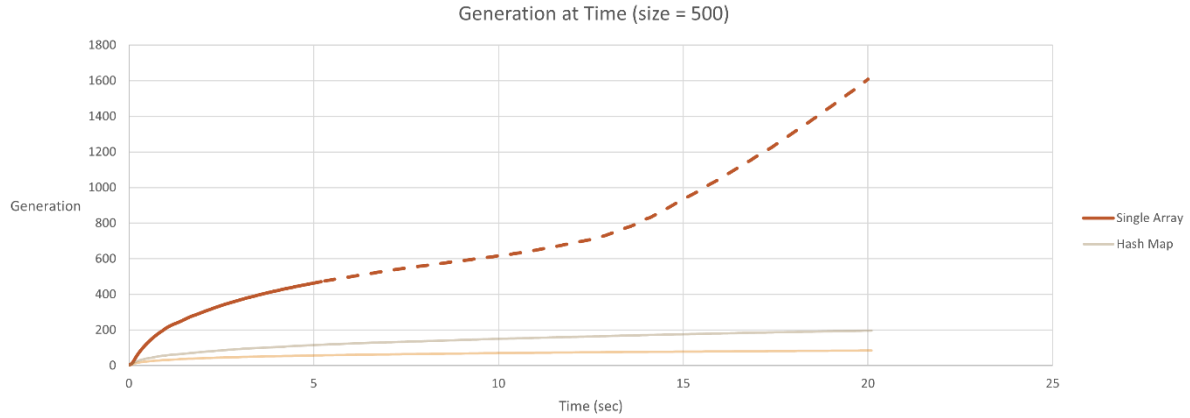
When I run the spacefiller pattern in an algorithm with a non-infinite board size, the test becomes unfair when the pattern hits the boundaries. The pattern ends up deteriorating, which greatly decreases the number of cells and thus makes the algorithm run much faster at a given generation when it really should have more cells to calculate. Here is a picture of what it looks like when the spacefiller pattern hits the boundary:



Eventually the field is full of sparse still-lifes and oscillators, which are much easier to calculate than a spacefiller.



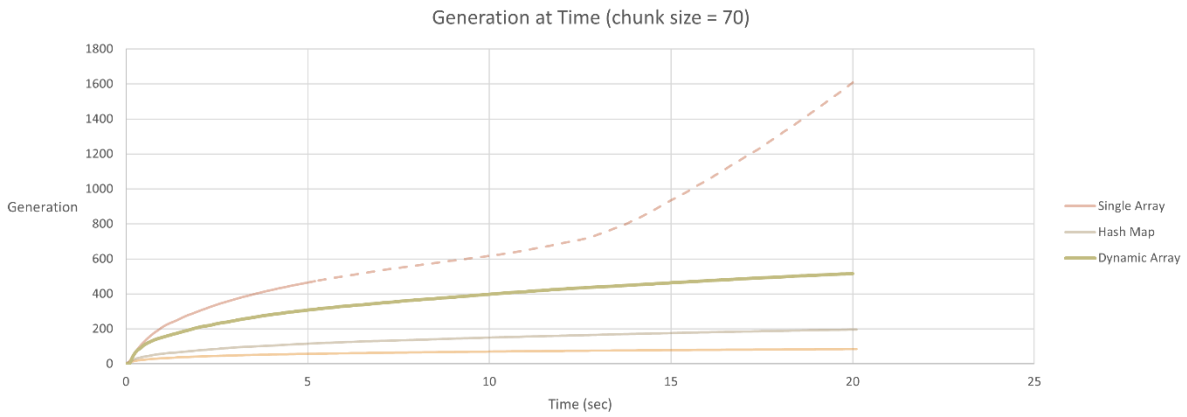
Here is a graph of how it performs when the board is 500x500 cells. As you can see, it performs much better than the Single Array or Hash Map algorithms. The problem is that it quickly runs out of space. It ran the algorithm for twenty seconds as quickly as it could, and it ended up hitting the boundary at about generation 500 in five seconds. I marked the generation it got to after that point in a dotted line to show the progression it made. It actually gets a little faster over time because the number of cells decreases as the pattern deteriorates.



4. Dynamic Array

After the great success of the Single Array algorithm, I wanted to see if I could harness its power while giving it the capacity of an infinite board. I designed an algorithm that uses arrays, but can generate more if necessary to accommodate a growing pattern. It works by splitting up the board in to square groups of cells called chunks. The Dynamic Array Algorithm then takes these chunks and converts them into arrays, and then stores them in a hash map. This way it can have potentially infinite arrays, but still have the benefit of the quick lookup speed. If the pattern would ever end up growing into an area that does not have a chunk stored into the hash map, the algorithm would simply generate one and add it. Another benefit is that this algorithm could potentially significantly decrease the amount of empty space that needs to be checked compared to the Single Array algorithm. One of the problems is that it might take a long time to manage the chunks, and the increased intricacy could slow down the algorithm too much.

If we take a look at the results, the dynamic array performs well, but not as well as the single array. It is still much better than the hash map or cell list algorithms, and it is the best algorithm yet that still supports an infinite board space. Here is how it performs when each chunk is 70x70 cells:



5. Neighbor Tracking Single Array

At this point, I decided to try a different approach. Instead of modifying the algorithm to give it a better data structure, what if I changed the way the base algorithm itself worked? That way I could make it faster than the base O(72n) that has been stopping me before. In taking this new approach, I designed the Neighbor Tracking Single Array algorithm.
This algorithm takes advantage of the fact that counting neighbors is so common in calculating the next generation. It takes 8 operations to count the neighbors of each cell, and often a given cell is checked multiple times because it is the neighbor to many cells. Perhaps I could take a shortcut and keep track of how many live neighbors a cell has, and just check that number whenever I wanted to count the number of live neighbors.

This algorithm is based off of the single array algorithm, but it uses another array of the same size to keep track of how many neighbors each cell has. It is much faster at counting the neighbors of each cell, but it is slower at setting a cell live. Instead of being an O(1)

operation, it is now an O(9) because it must increment the live neighbor counts of each of the neighbors whenever it sets a cell live. This may be too big of a price to pay, or it may not. It depends on how many cells are live and how close they are to each other.

This algorithm ends up performing very similarly to the single array algorithm. This is to be expected. In the particular test I ran, it did slightly worse than Single Array at the beginning, but it ended up passing it later on. The exact behavior depends on what particular pattern is being used, but I think the Neighbor Tracking Single Array algorithm would generally be better than the Single Array algorithm.

6. Chunk Calculation

At this point, I had almost run out of ideas. I thought the Neighbor Tracking algorithm was about the best I could do, but something was bothering me. Because the general method I was using checked every single live cell in order to calculate the state of the next generation, it could never get faster than O(n). Was there some way to design an algorithm that was faster than O(n)? It would have to calculate large chunks of the board at once. This led me to the Chunk Calculation algorithm.

The Chunk Calculation algorithm is based off of the Dynamic Array algorithm, but it works a little different. It calculates the state of the next generation the same way for each chunk, but then it stores the solution it calculates in a has map so that it can just use the same solution later instead of having to re-calculate it. For the key, it converts the chunk into a string that has the values of each of the cells, and newline characters to represent new rows. The value is an array of the next state of the chunk.

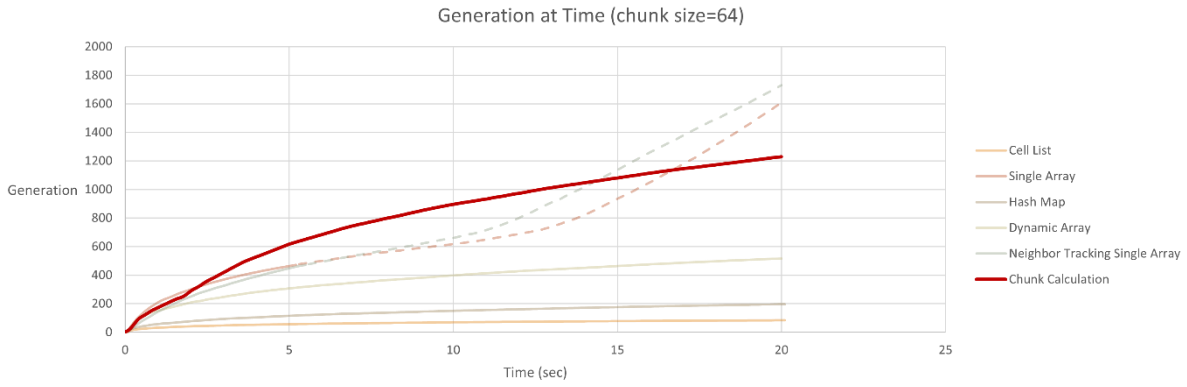
This algorithm has various strengths and weaknesses. It is based off of Dynamic Array, so in it's worst case scenario it would probably be very similar. The added complexity of calculating keys could slow it down, but it shouldn't be significant. At it's best, it would be

O(c) where c is the number of chunks. This would be much less than n because each chunk would contain multiple live cells depending on how big they are.

The algorithm would naturally be better at patterns that have large repeating sequences, like the spacefiller pattern, and it would do worse in patterns that do not have much repeats.

There were extra difficulties in programming this algorithm that I did not anticipate. I ran into a lot of issues in dealing with the edges of chunks. I had to make the keys for each solution slightly bigger than the solution because the answer would depend on the values of the neighbors on the edges of the neighboring chunks.

In the end, the work paid off. This was the best performing algorithm yet, even better than the array based algorithms. It also had the potential for an infinite board space, so it was even better in that aspect as well.

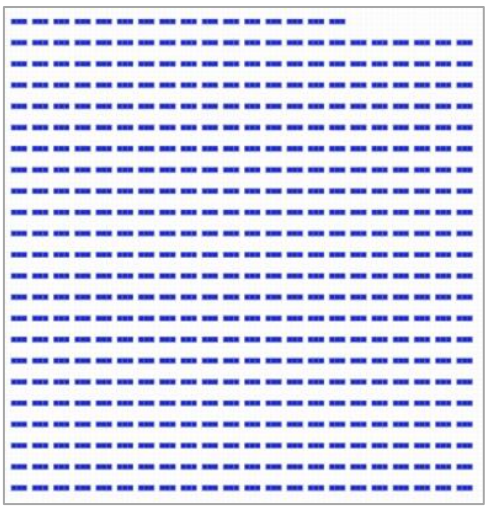


Running a Different Test

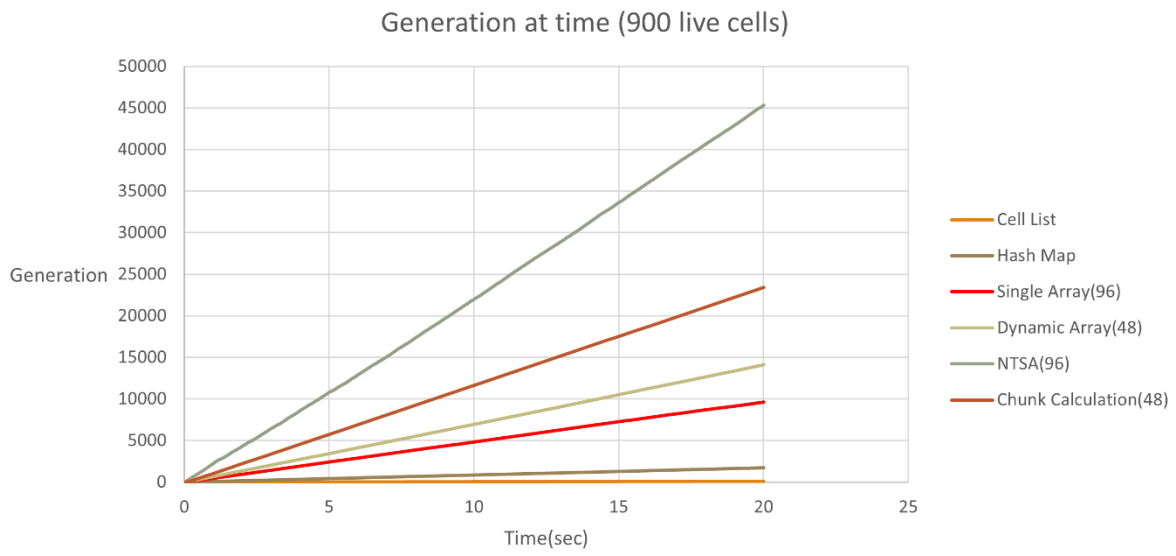
One thing to note about these algorithms is that they perform differently in different situations. The test with the spacefiller that I ran tested how they performed with a high amount of live cells, but this situation doesn't necessarily occur in general us of the Game of Life. Often people only experiment with a small amount of live cells. I decided to run a different test that had a constant amount of cells to see how the different algorithms performed. I set up a grid of blinkers, a simple oscillator that repeats itself every other

generation. This would cause there to be a constant number of cells on the board while still having each generation be different from the previous one.

The result was very different from the tests I ran with the spacefiller previously. In this new test, the Neighbor Tracking Single Array ended up dominating, followed by Chunk Calculation, Dynamic Array, Single Array, Hash Map, and Cell List. I was surprised to see that Dynamic Array beat Single array. I wasn't sure why, but it shows that these algorithms don't always act predictably, and although there are some that are generally better or worse, a lot of it depends on the situation and what your needs are.



The grid of blinkers I used in the second test



Conclusion

The game of life is a very interesting and unpredictable game, yet it is also deterministic. There are many fun things that can be done in it. I decided to design some algorithms to simulate it as quickly and efficiently as I could. The main problem ended up being the data structure that I used. It was difficult to find one that could support an infinite board while still being able to get and set the state of cells quickly. All of my algorithms were O(n) except for one of them, and that one was generally the best, but it depended on the particular pattern that you needed to calculate. Each of the algorithms had different strengths and weaknesses, and different algorithms would do better in different situations.

Works Cited

Gardner, Martin. “The fantastic combinations of John Conway’s new solitaire game ‘life’”. Mathematical Games. *Scientific American*. Vol 223 no. 4. pp 120-123. doi:10.1038/scientificamarian1070-120