Algorithms for Conway's Game of Life

BEN CAMPBELL'S SENIOR THESIS – FRANCISCAN UNIVERSITY OF STEUBENVILLE

NOVEMBER 22, 2021

Outline

1) The Game of Life
2) The Simulator
3) The Algorithms

1) The Game of Life

1) The Game of Life

A cellular automaton created by John Conway in 1970.

Traditionally infinite grid of cells.

Each cell can be on or off.

The game can advance to the next state (generation) according to certain rules.

1) The Game of Life - Rules

Neighbors: the cells adjacent (including diagonal) to a given cell.

1. Underpopulation: any live cell with less than two neighbors dies
2. Stable: 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

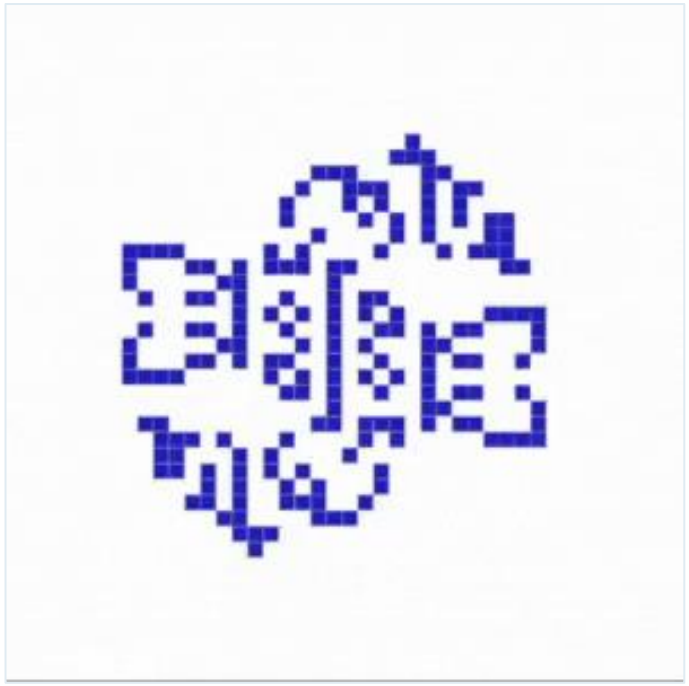
1) The Game of Life – Rules (simplified)

A cell is live in the next generation if:

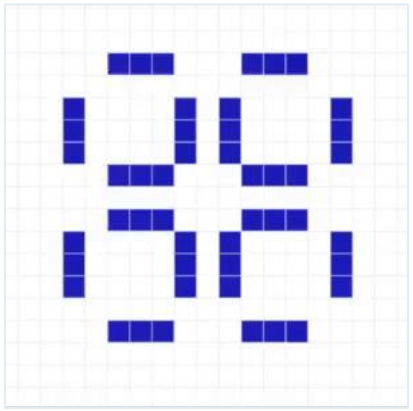
- It is live and has two or three neighbors
- It is dead and has three neighbors

1) The Game of Life – Patterns

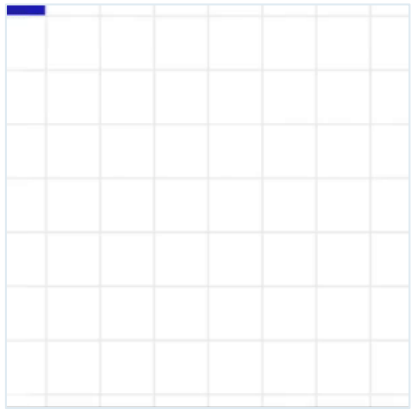
Cool patterns can be made in the game of life.



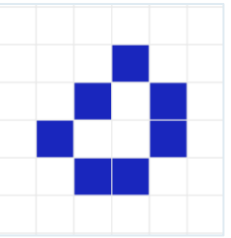
Filler



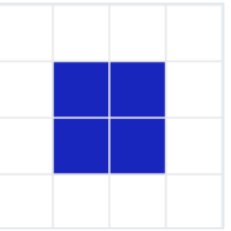
Blinker



Glider



Loaf



Block

2) The Simulator

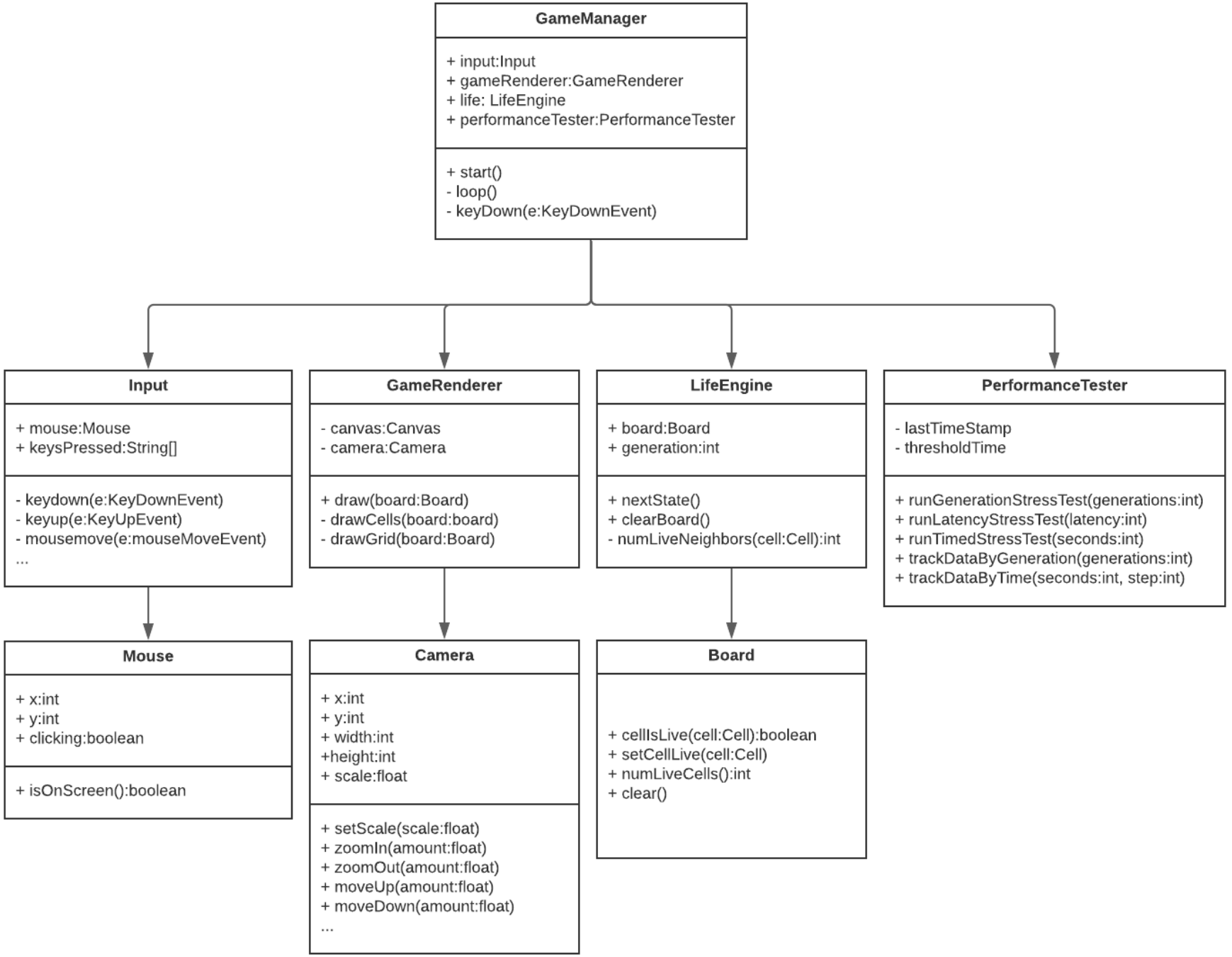
2) The Simulator

Written in HTML, CSS, and JS

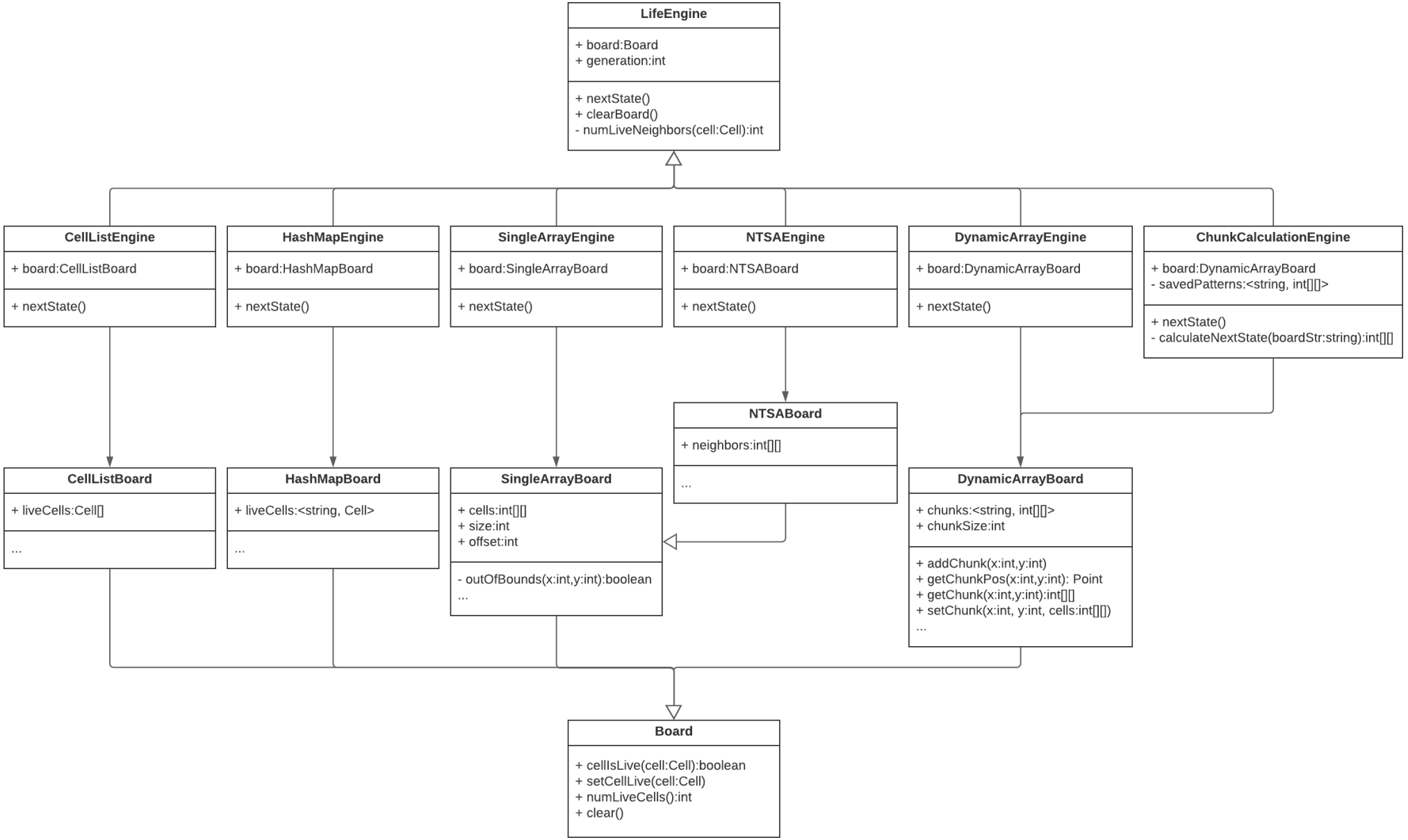
23 files

About 1600 lines of code

2) The Simulator – Class Diagram



2) The Simulator – Engines and Boards



3) The Algorithms

3) The Algorithms - Complexity

Complexity

- A way of measuring how fast an algorithm is.
- How many operations does the algorithm perform for a given input size?
- Define function f(n):
  - n is the size of the input
  - f(n) is number of operations
- Use Big O notation. Algorithm is O(f(n))

3) The Algorithms - Complexity

Example: add up numbers in a list.

- n: the number elements in the list.
- operation: addition

$$[2,5,8,2,8]$$

If the list has 5 numbers

$$2 + 5 + 8 + 2 + 8$$

You perform addition 4 times.

3) The Algorithms - Complexity

Example: add up numbers in a list.

- n: the number elements in the list.
- operation: addition

$$[2,5,8,2,8]$$

If the list has 5 numbers

$$2 + 5 + 8 + 2 + 8$$

You perform addition 4 times.

$$[x_1, x_2, \dots, x_{n-1}, x_n]$$

If the list has n numbers

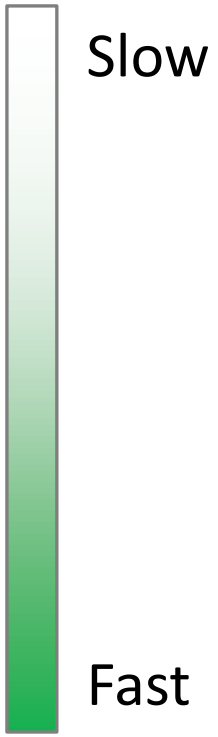
$$x_1 + x_2 + \dots + x_{n-1} + x_n \longrightarrow f(n) = n - 1$$

You perform addition n - 1 times.

This algorithm is O(n)

3) The Algorithms - Complexity

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



Slow

Fast

3) The Algorithms – General Method

Recall Simplified rule:

A cell is live in the next generation if:

- It is live and has two or three neighbors
- It is dead and has three neighbors

General method

For each live cell:

- Count neighbors, set live on next board if two or three.
- For each neighbor:
  - Count neighbors, set live if two or three.

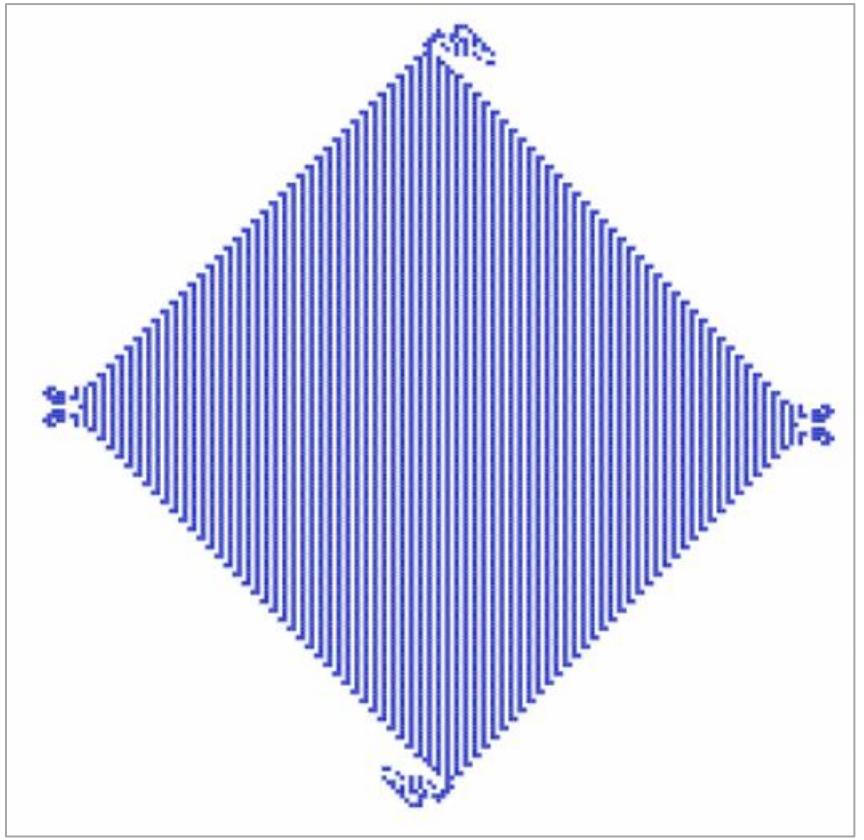
Requires 8 + (8*8) = 72 checks to see if a cell is live.

3) The Algorithms - Testing

Test using filler pattern. (grows quickly)

Ran tests to see how quickly algorithms performed.

- Time-based
- Generation-based
- Minimum processes: only algorithm, recording data, and checking to see when to stop.
  - window became unresponsive while tests were being run



3) The Algorithms

6 algorithms:

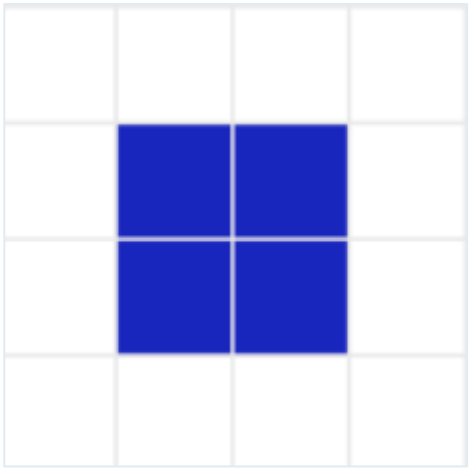
1. Cell List (CL) – $O(72n^2)$
2. Hash Map (HM) – O(72n*?)
3. Single Array (SA) – O(72n) + $O(size^2)$
4. Dynamic Array (DA) - O(72n) + $O(2*n\_chunk*size\_chunk^2)$
5. Neighbor Tracking Single Array (NTSA) – O(~16n) + $O(size^2)$
6. Chunk Calculate (CC) – between O(72n + 24*size_chunk) and O(n_chunk*9*size_chunk)

3.1) Cell List

Stores live cells as an unsorted list of coordinates.

Sometimes has to go through the entire list to figure out if a cell is live or not: O(n) lookup time

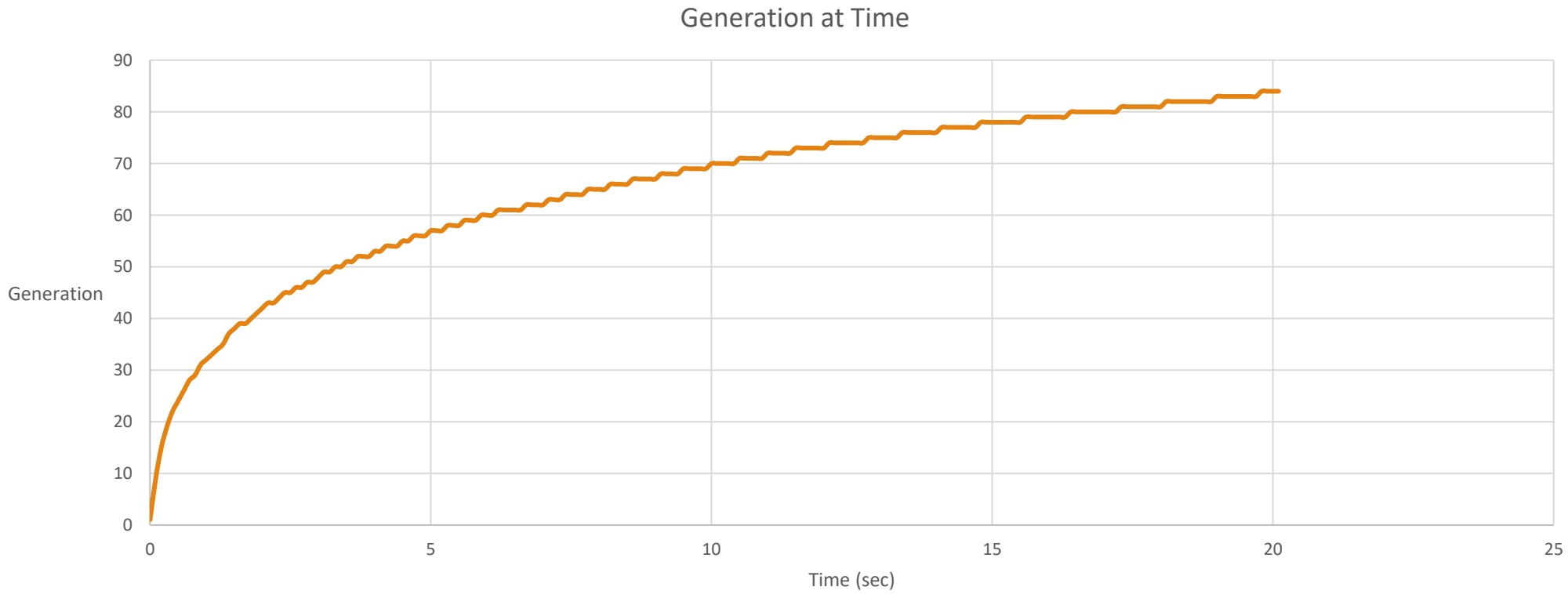
Setting a new cell to live is easy. Just put it on the end of the list.



[(0,0), (0,1), (1,0), (1,1)]

3.1) Cell List - Results

Slowest algorithm because it takes so long to tell if a cell is live. $O(72n^2)$



3.1) Idea

What if we had a faster lookup speed?

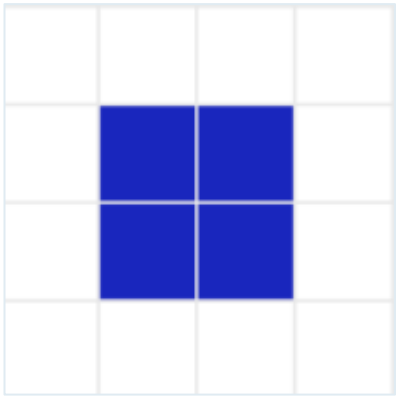
3.2) Hash Map

Hash maps

- Hash maps are a list of key-value pairs.
- {“Bob”: 20, “Joe”: 18, “Sally”: 19, “Sebastian”: 18}
- Lookup time O(1).

Hash map algorithm

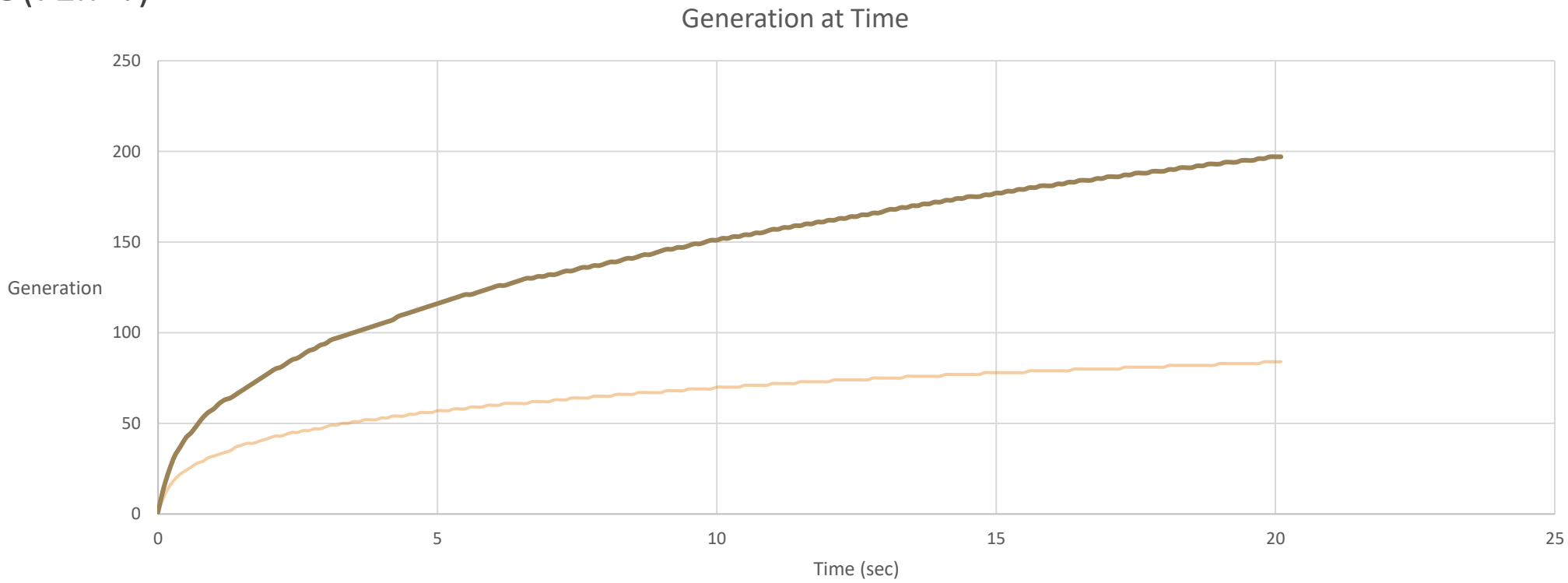
- Uses hash map to store what cells are live.



{“0,0”: (0,0), “0,1”: (0,1),
“1,0”: (1,0), “1,1”: (1,1)}

3.2) Hash Map - Results

Theoretically O(72n). Better than Cell List, but not that great compared to other algorithms. O(72n*?)



3.4) Idea

Maybe Arrays would be better?

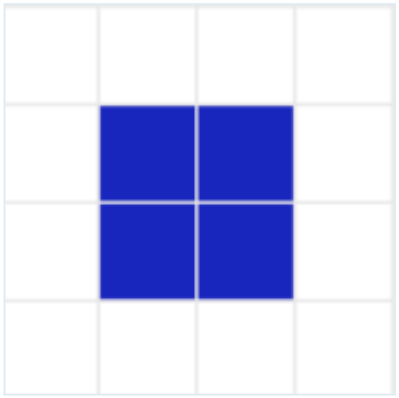
3.3) Single Array

Keeps track of cells in a gigantic two dimensional array.

Lookup speed: O(1) (or maybe O(2))

Not infinite.

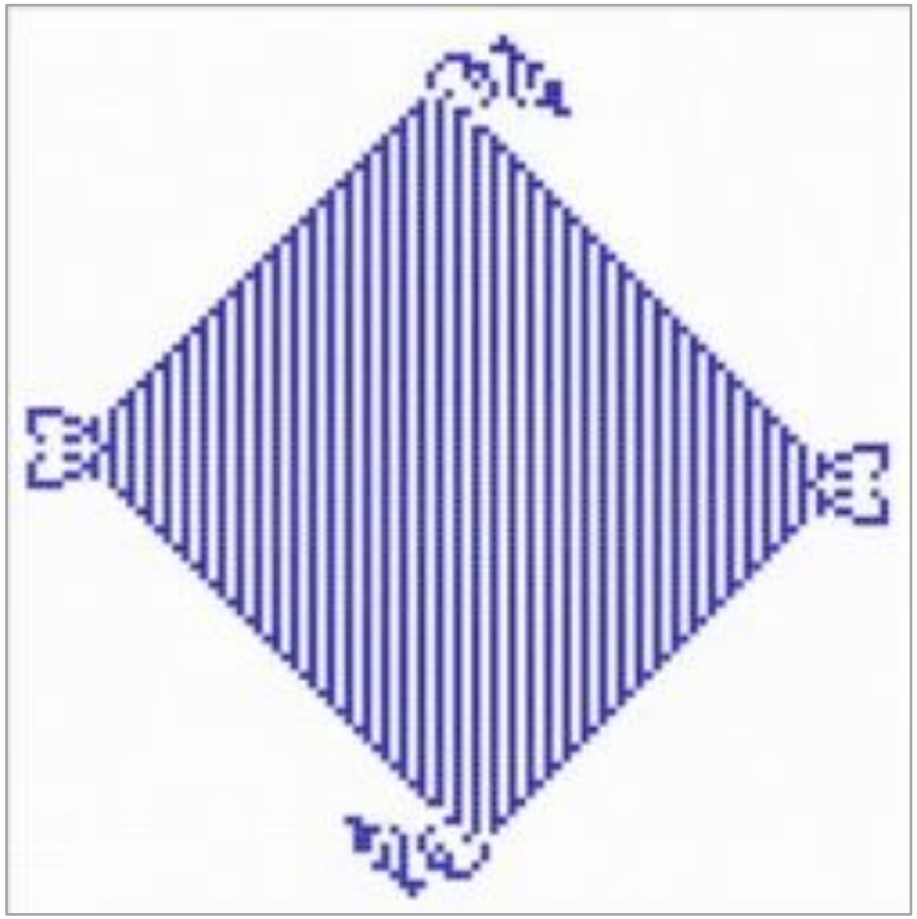
Needs to check every single cell to find the live ones: O(size^2)



	0	1	2	3
0	0	0	0	0
1	0	1	1	0
2	0	1	1	0
3	0	0	0	0

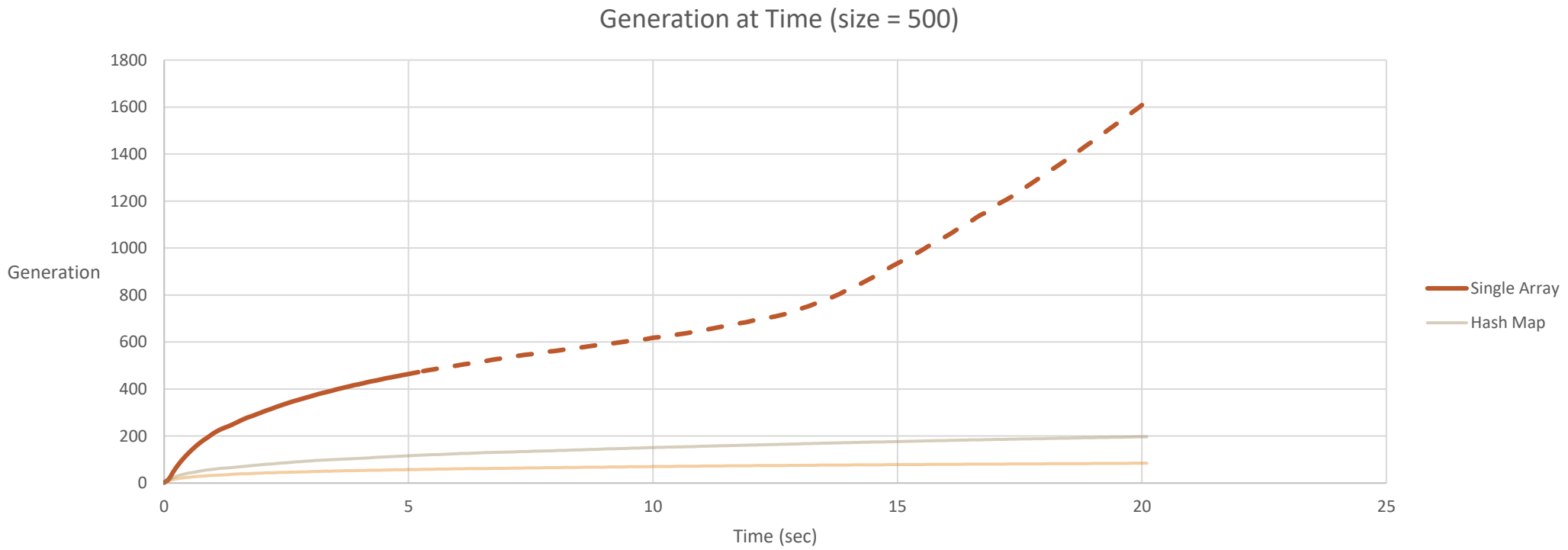
3.3) Single Array

Because the board size is not infinite, the pattern deteriorates once it hits the borders.



3.3) Single Array - Results

O(72n) + O(size^2). Much faster than Cell List or Hash Map. Gets slower as array size increases

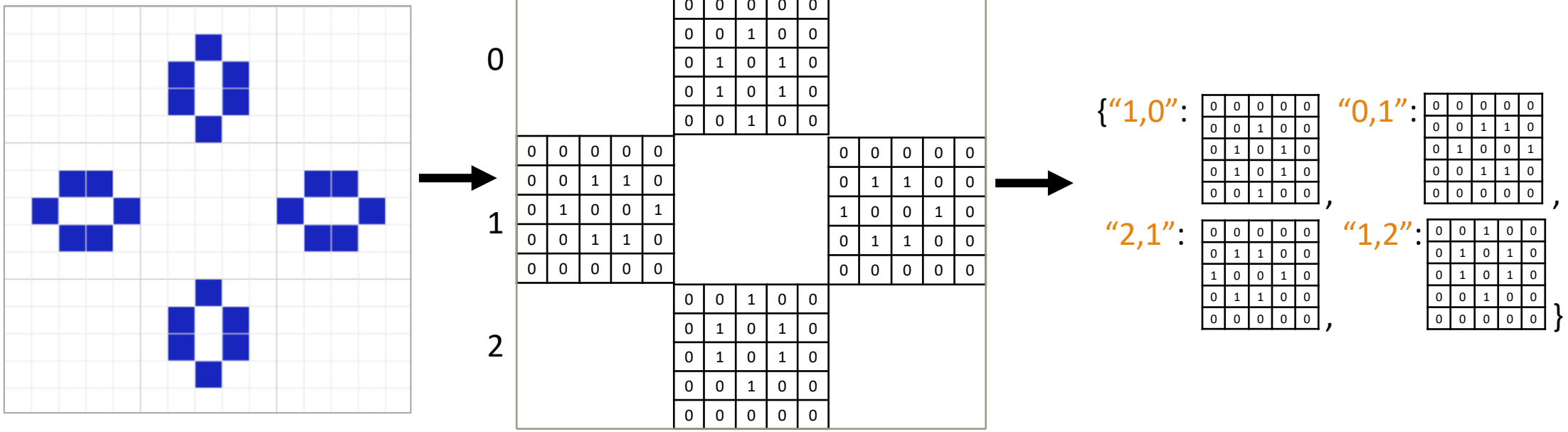


3.3) Idea

Can we do arrays, but also infinite?

3.4) Dynamic Array

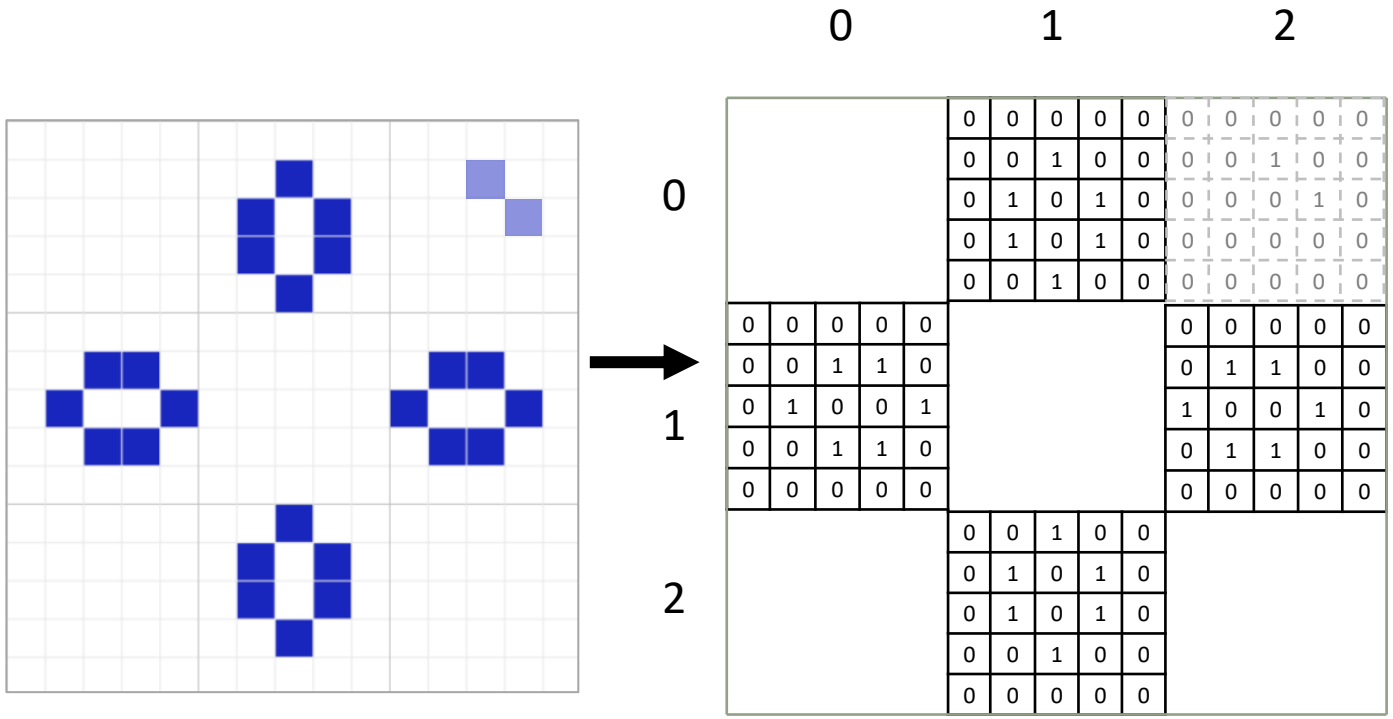
Method using arrays, but also infinite. Less empty space to check.



Board is split up into small arrays.

Arrays are stored in a hash map.

3.4) Dynamic Array



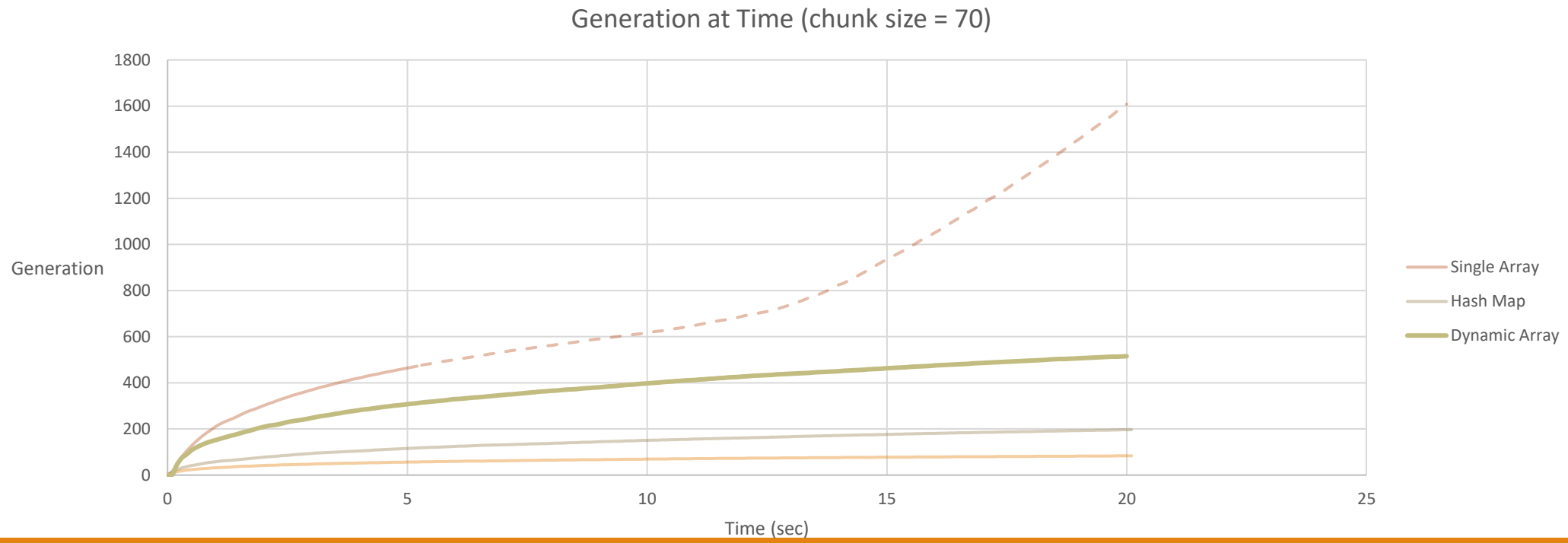
When a cell needs to be set into a space that doesn't have an array, a new one is made.

3.4) Dynamic Array - Results

Better than Cell List and Hash Map, but not as good as Single Array.

Performs well in first 150 generations, but then starts tapering off.

O(72n) + O(2*n_chunk*size_chunk^2)



3.4) Idea

We count neighbors much more than we set a cell live.

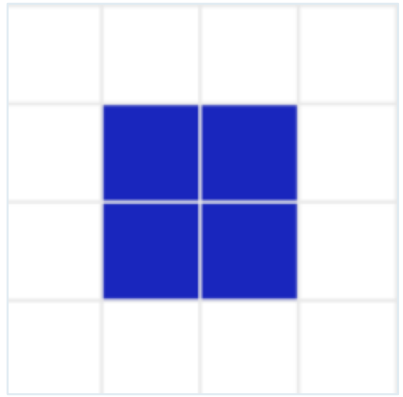
What if we keep track of how many neighbors a cell has so we don’t have to keep re-counting?

3.5) Neighbor Tracking Single Array

Same as Single Array, but use another array to keep track of the number of neighbors each cell has.

O(9) to set a cell live, but only O(1) to count neighbors

Not infinite



Which cells are live

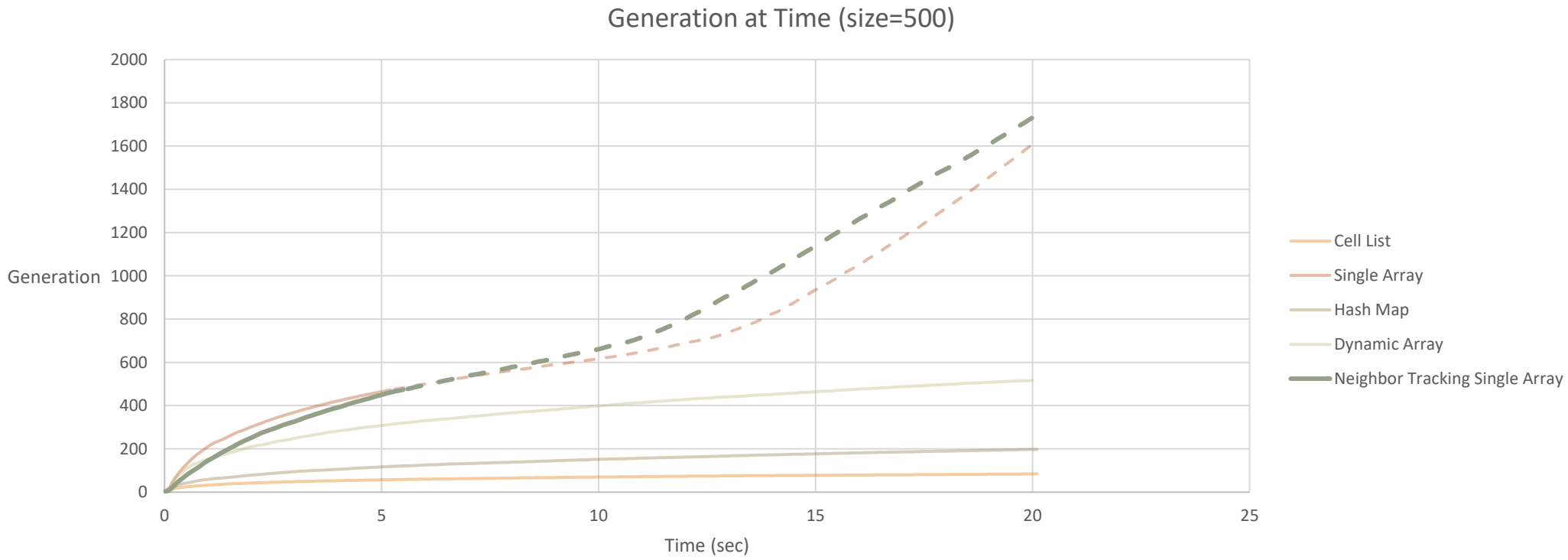
0	0	0	0
0	1	1	0
0	1	1	0
0	0	0	0

How many neighbors a cell has

1	2	2	1
2	3	3	2
2	3	3	2
1	2	2	1

3.5) Neighbor Tracking Single Array - Results

Roughly O(16n) + O(size^2). Ends up being almost the same as Single Array, but beats it at gen 520.



3.5) Idea

Recap of General method:

For each live cell:

- Count neighbors, set live on next board if two or three.
- For each neighbor:
  - Count neighbor, set live if two or three.

If we check every cell to determine the next state, we will never get faster than O(n).

Is there a way to get faster than O(n)?

We would need to calculate several cells at once.

3.6) Chunk Calculation

Based of of Dynamic Array. When it calculates the next state of a chunk, it stores the solution in a hashmap, and uses this so it doesn't have to re-calculate it.

Previously calculated solutions

{"0, 0, 0, 0, 0:
0, 0, 0, 0, 0
0, 1, 1, 1, 0
0, 0, 0, 0, 0
0, 0, 0, 0, 0"

0	0	0	0	0
0	0	1	0	0
0	0	1	0	0
0	0	1	0	0
0	0	0	0	0

,

"0, 0, 0, 0, 0:
0, 0, 1, 0, 0
0, 0, 1, 0, 0
0, 0, 1, 0, 0
0, 0, 0, 0, 0"

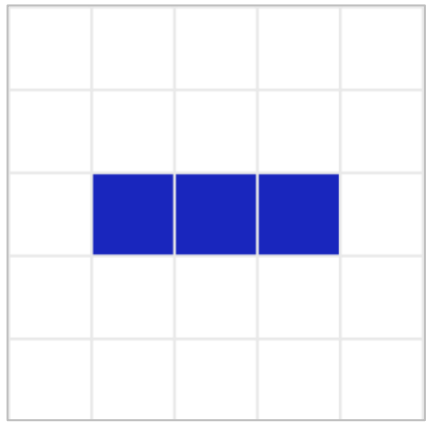
0	0	0	0	0
0	0	0	0	0
0	1	1	1	0
0	0	0	0	0
0	0	0	0	0

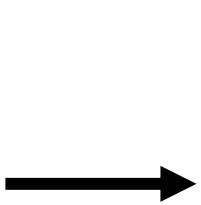
, … }

Key is the chunk expressed as a string.

Value is an array of the calculated solution.

3.6) Chunk Calculation - Process





"0, 0, 0, 0, 0
0, 0, 0, 0, 0
0, 1, 1, 1, 0
0, 0, 0, 0, 0
0, 0, 0, 0, 0"

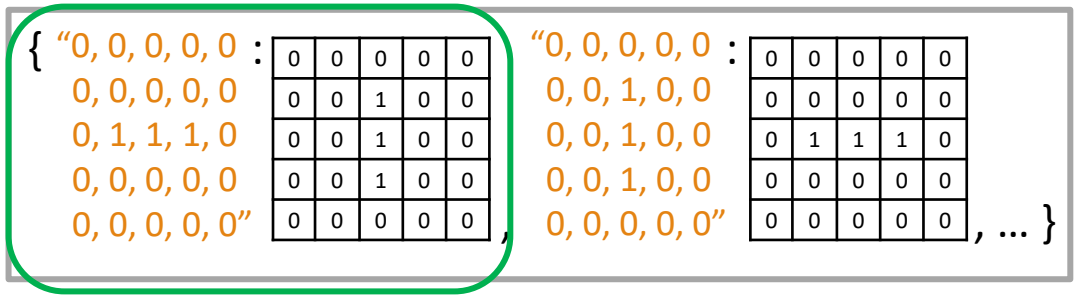
Previously calculated solutions

Convert each chunk to a string.

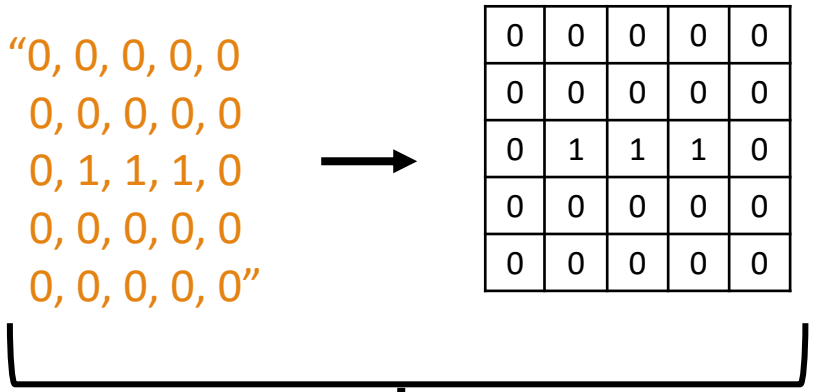
Check to see if the solution was previously calculated.

3.6) Chunk Calculation - Process

Previously calculated solutions



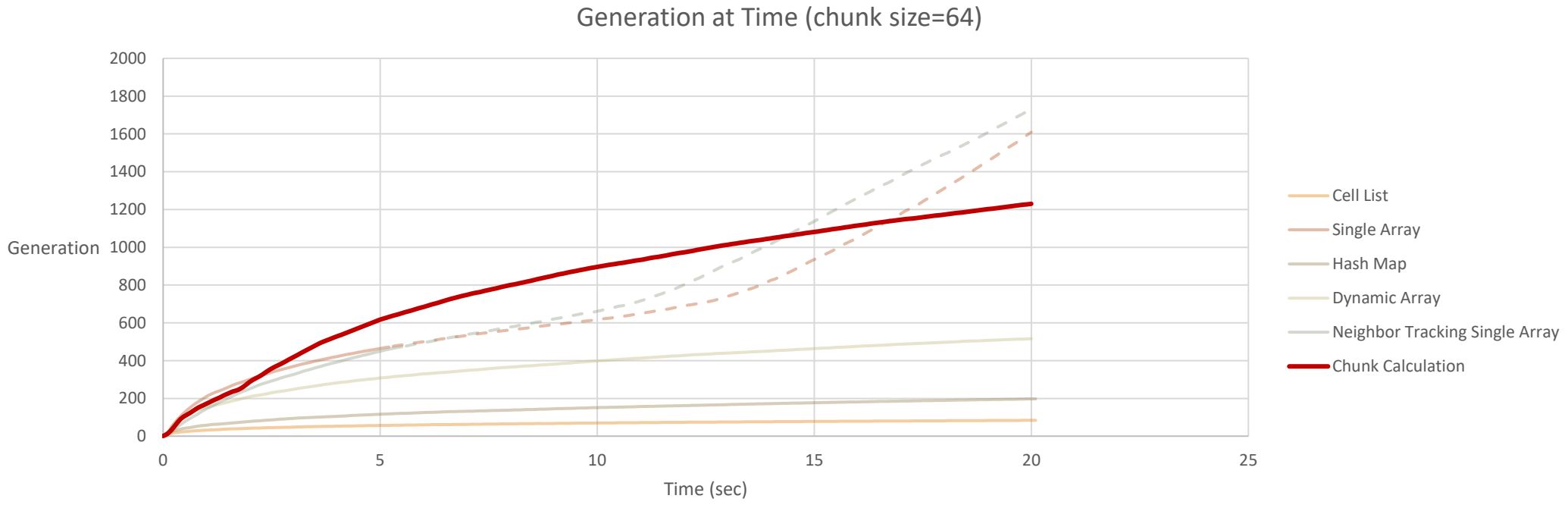
If it has, use the solution.



Otherwise, calculate it and store it in the hashmap.

3.6) Chunk Calculation - Results

Best overall. May be biased to be good at the filler. If the board size is small, the array algorithms are better.



3.6) Chunk Calculation – Interesting Bugs

Because it uses the same array for multiple chunks, interfering with it after the initial state causes the change to happen wherever the array is used.

Since you're changing the solution it calculated, it remembers the interferences you made later.