Euler Circle Paper: Hashlife

Jerry Guo

December 18, 2022

Abstract

This paper seeks to provide an overview of the concept of hashlife, an algorithm used for simulations of cellular automata. It explores the origin of the algorithm as well as the mechanics behind its function.

1 The Game of Life

The game of life, created by John Horton Conway in 1970, was a cellular automaton on a two-dimensional grid that progressed based on a fixed set of rules, with each cell taking into account its current state, along with those of its neighbors, in order to produce its state one step into its future. Therefore, it is commonly termed to be a "zero-player game", requiring no input from a user after the initial setup.

However, in the 1970s, there was limited computational power available. As such, the basic method of taking every cell on the grid and checking its borders was unsatisfactory. As such, optimizations were needed to shorten the simulation time for large-scale automata. Some types of optimization methods include detecting and skipping empty space, still lifes, oscillators, et cetera. However, there will be a certain point where attempts to detect more objects will slow down the speed of the code.

2 Hashlife

In 1984, Ralph William Gosper Jr. created an algorithm that greatly increased the speed at which the game of life can be computed. The algorithm, which he termed "Hashlife", relied on two main principles:

- Hash tables, a way to store situations that have already been processed in a way that enables fast access.
- Macrocells, a way of dividing the grid in a way that allows computation to proceed several steps at a time.

We will go over each of these ideas in order.

3 Hashlife Principles

3.1 Hash Tables

Hash tables are a type of data structure that is composed of two main elements: a hash function and a bucket array. The hash function is used to generate a number from the data. This number, the hash, is used to determine where the data should be stored in the bucket array. A common way of doing this is taking the hash modulo the number of buckets. The bucket array is essentially a list of receptacles that can hold the data. When data is inserted into the hash table, the hash function is used to generate a number, which is used to determine which bucket the data should be stored in.

However, there is a problem associated with hashing via modulo, which is one of the collisions: different data might be mapped to the same bucket. A way to solve this would be to create a list in the bucket containing both pieces of data.

When data is requested from the hash table, the hash function is used again to generate the hash of the data that was requested. This hash is then used to quickly look up the data associated with it in the bucket array. This makes hash tables a very efficient way to store and look up data, often having $O(1)$ in data lookups. In some cases involving collisions, the lookup process involves searching through the list inside the bucket. However, the speed increase through data lookups is still substantial.

3.2 Macrocells

A macrocell is a way to represent any section in the grid that is 2^n by 2^n, where n is an integer such that $n > 1$. It utilizes a structure called a quadtree, where the macrocell is split into four quadrants, and each quadrant is stored as a node in a tree under the macrocell.

For example, a 4 by 4 macrocell contains four nodes of its quadrants, which, in turn, have four nodes of its quadrants, which, in this case, would either be a 1 or a 0.

Immediately, we can see some points of optimization. one of them is that if there is a repeat of macrocells, instead of storing the same information twice, we can use two pointers to point toward one instance of the macrocell. As shown below, this offers a significant decrease in the number of nodes. This effect becomes extremely pronounced when we have a large number of stored macrocells, as we can share its children across the same level with different nodes.

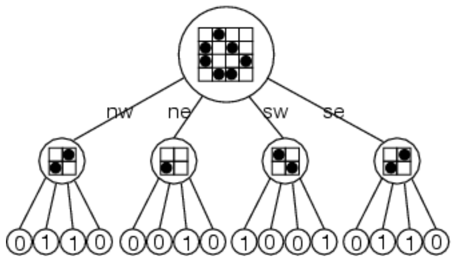

Figure 1: Our original tree, with 21 nodes

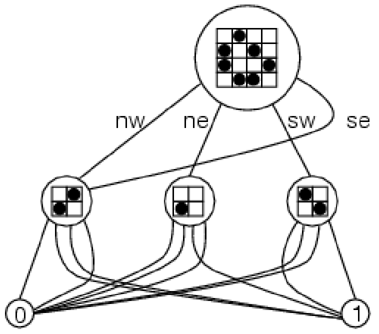

Figure 2: Our new tree, with six nodes

As we can see, this greatly simplifies the space needed to store one macrocell.

Now, we can establish the "speed of light" in the game of life, the fastest possible speed of information transfer, to be one cell per step. Therefore. Every time we simulate one macrocell 1 step forward, we have to remove the outer ring of cells, as those are not necessarily accurate anymore.

A neat and useful quirk of the macrocell is that, after simulating 2^{n-2} steps, we get another, smaller macrocell of dimensions 2^{n-1}, which we shall call the result of the macrocell and store with the macrocell in the hash table.

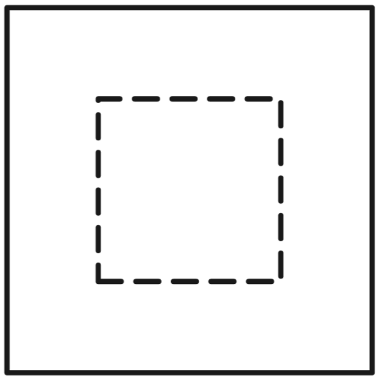
Figure 3: A macrocell, with its result in dashed lines.

However, as fancy as this seems, this offers a minimal computational advantage compared to the algorithm before simulating every grid. The key insight is to have an inductive function that calculates macrocell progressions based on smaller macrocells.

For the base case, we can use a 4 by 4 macrocell, brute-forcing all possible results after one step and storing them inside the hash table.

Now, let us proceed with the induction step. Given macrocells of size 2^n and their results after 2^{n-2} steps, we aim to compute a macrocell of size 2^{n+1} after 2^{n-1} steps.

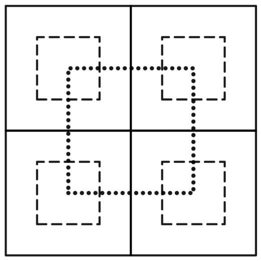

Figure 4: Our larger macrocell, with its desired result and the 4 macrocells inside it.

Immediately, we can see two problems with this configuration.

1. We only have the smaller macrocell's result after 2^{n-2} steps. This means that we are going to have two separate steps in order to progress 2^{n-1} steps.

2. As we can see above, the smaller macrocells do not have results that completely fill up the larger macrocell's result.

We can do this with the following algorithm:

First, we use nine overlapping 2^n macrocells such that combining their results make a $3 \cdot 2^{n-1}$ cell group.

Since the five added macrocells are "irregular", that is, they are not aligned with the larger macrocell, we may not know their result. Fortunately, suppose we scan across the hash table and don't encounter a macrocell. In that case, we may construct those irregular macrocells from the quadrants of the 2^n macrocells and use these to calculate the result of the irregular macrocells.

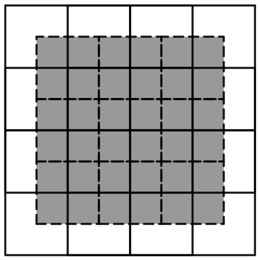

Figure 5: The 9 macrocells making a $3 \cdot 2^{n-1}$ cell group.

Now, we have the $3 \cdot 2^{n-1}$ cell group after 2^{n-2} steps.

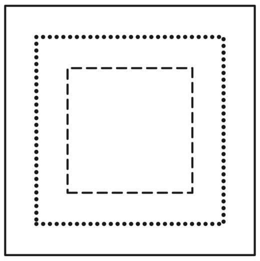

Figure 6: Solid line denotes our original macrocell, dotted line denotes our current process, and dashed line denotes our desired result.

We can create another four overlapping 2^n macrocells such that their results match our expected result in our 2^{n+2} macrocell. Again, if we have not encountered an irregular macrocell, we may construct it from the quadrants of the 2^n macrocells and calculate its result from there.

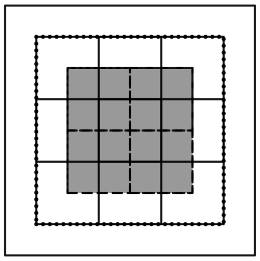

Figure 7: The 4 overlapping 2^n macrocells combining to form our final result.

Now, through 2 steps each of length 2^{n-2} steps, we have obtained the result of the bigger macrocell after 2^{n-1} steps, as desired.

The recursive nature of this algorithm means that, provided the memory, the board can be calculated in logarithmic time, which is an enormous improvement over the polynomial time of simulating every cell.

4 Bibliography

References

[1] Gosper, R. Wm. "Exploiting regularities in large cellular spaces." *Physica D: Nonlinear Phenomena* 10.1-2 (1984): 75-80.

[2] Rokicki, Tomas G. "An Algorithm for Compressing Space and Time." *Dr. Dobb's*, 1 Apr. 2006, www.drdobbs.com/jvm/an-algorithm-for-compressing-space-and-t/184406478. Accessed 18 Dec. 2022.

[3] "Hashlife." *Wikipedia*, Wikimedia Foundation, 21 Jul. 2021, en.wikipedia.org/wiki/Hashlife. Accessed 18 Dec. 2022.