Conway's Game of Life

Nikhil B

August 11, 2020

Problem Statement

The universe of the Game of Life is an infinite, two-dimensional orthogonal grid of square cells, each of which is in one of two possible states, live or dead, (or populated and unpopulated, respectively). Every cell interacts with its eight neighbours, which are the cells that are horizontally, vertically, or diagonally adjacent. At each step in time, the following transitions occur:

- Any live cell with fewer than two live neighbours dies, as if by underpopulation.
- Any live cell with two or three live neighbours lives on to the next generation.
- Any live cell with more than three live neighbours dies, as if by overpopulation.
- Any dead cell with exactly three live neighbours becomes a live cell, as if by reproduction.

The rules continue to be applied repeatedly to create further generations. Our program should show evolution based on above rules.

Input File Format

All the standard input files are present inside `Samples/` folder with the extension `.rle`. This format uses `b` for a blank, `o` for a live cell, `$` for a new line, and integer repeat counts before any of this. Ignore any line starting with `#` or `x`

Approach 1: Naive Implementation

Python code for this implementation is in `python/` folder. This is the standard O(`grid size`) algorithm which iterates through the grid to calculate number of neighbors and apply

Conway's rules to determine state in the next generation

Algorithm 1: O(`grid size`) naive algorithm

Result: new grid corresponding to next generation
initialization;
for *each grid cell* **do**
  | calculate number of neighbors and apply conway's rules
end

Drawbacks

- This approach is both space and time intensive. 1000×1000 grid for 1000 generations would take more than billion operations
- Portions with same patterns for which already next generation is calculated are not considered
- Each cell shares a constant bitmap so there is no need to have distinct cells that share the same value.

Tricks

- bit tricks, skip empty regions, skip still lifes, skip oscillators - adds code complexity and in several cases only linear speedup is achieved.

Approach 2: Bill Gosper's Hashlife

Java code for this implementation is in `java/` folder. Hashlife is a quadtree based approach which resolves all the drawbacks of naive algorithm and achieves astronomical improvements in both speed and space. Hashlife combines three simple ideas:

- Compression of space using canonicalized quad trees.
- Caching of result across space and time.
- Astronomical speed by efficient recursion.

0.1 Compression of space using canonicalized quad trees

Quad tree represents $2^n \times 2^n$ grid by a tree of height n. leaves are single cells storing 0 or 1 and each node has 4 children: northeast, northwest, southeast, southwest(Refer Figure 1). In canonical quad trees we identify identical subnodes and combine them into a single one. A hast table is used to store and lookup canonicalized nodes, hence the name Hashlife.(Refer Figure 2 and observe space compression).

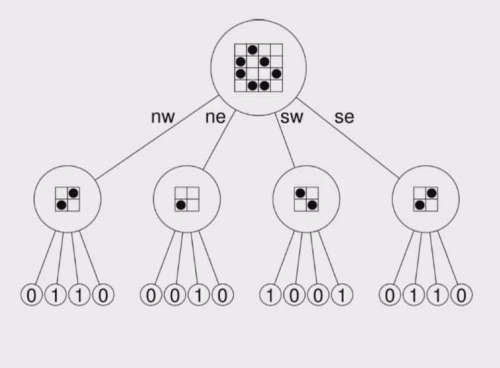

Figure 1: A quad tree representation of the 4×4 grid needs a total of 21 nodes.

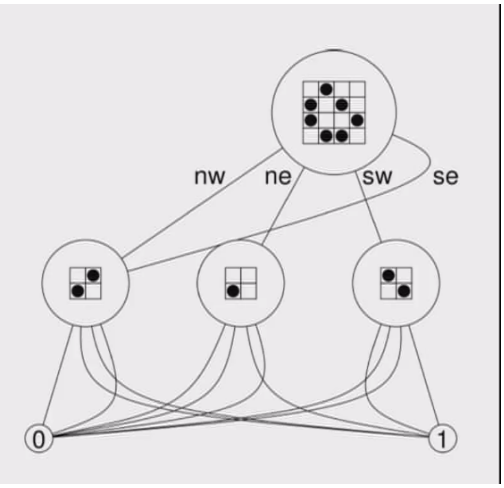

Figure 2: Identical subnodes combined into same node, the grid in previous figure can be represented using 6 nodes only

0.2 Caching of result across space and time

We only want to calculate the result of each node once, so we cache the result of a node directly in the node itself, adding it as a fifth child(Refer Figure 3). Please refer to Appendix for information about calculating result. We already have a hasttable storing each node of our tree for a given generation - share this across generations to reuse calculated results.

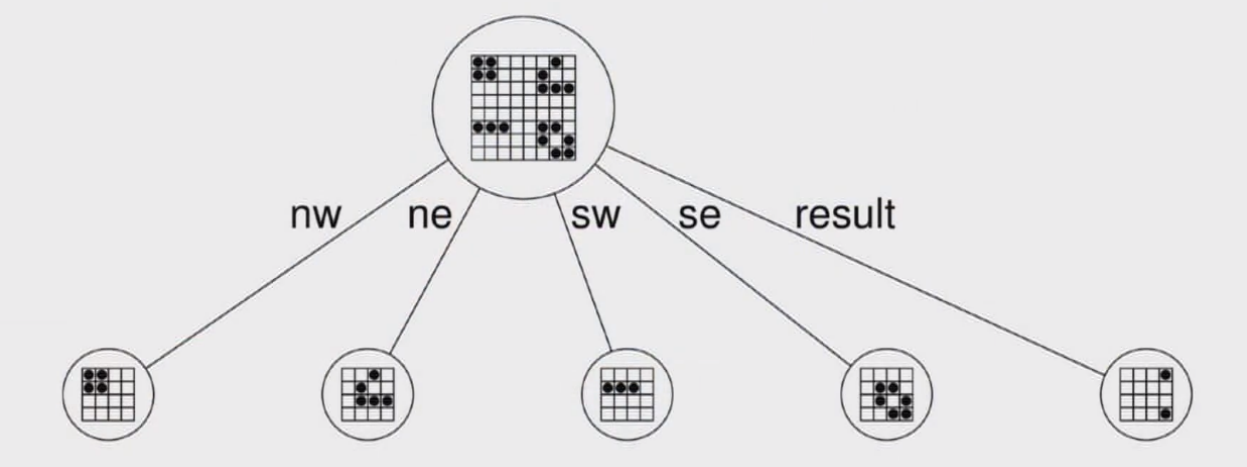

Figure 3: Cache the result of the node directly as a child to node. if result in not null it will be storing the result

0.3 Astronomical speed by efficient recursion

Make result of a node of size $2^n \times 2^n$ to advance $2^{(n-2)}$ generations. So a 256×256 node will advance 64 generations at once. The recursive calculation steps the child subnodes twice, not just once; so the 256×256 node will advance 128×128 subnodes twice, 32 generations each. More details on how this can achieve, read Appendix.

Appendix: Calculation of Result of a node [1]

Result of a node at level n is a node of level n-1 centered w.r.t original node(Refer Figure4).

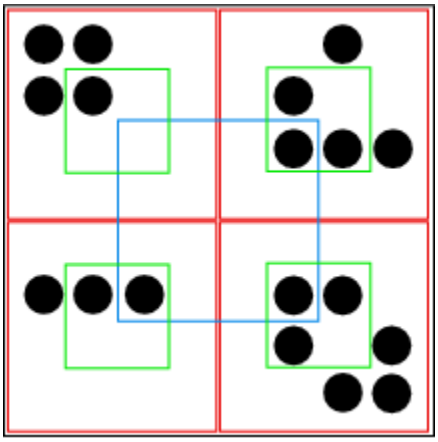

Figure 4: Black outer box is the node for which result(blue box) is to be calculated, 4 red inner nodes are the subnodes. We can see that the inner squares of red subnodes(green boxes) are not sufficient to calculate the result(blue box)

This result cannot be calculated using usual tree recursion. Now in order to calculate result we construct 9 new nodes two levels down(using nodes that are three level down) represented as red boxes in Figure 5. We use appropriate combinations of these 9 new nodes to calculate result.
Now for astronomical speed up, instead of taking this generation red nodes of Figure 5 we take next generation red nodes to calculate green node as next generation of appropriate combination of red nodes. This would give the astronomical speed up as discussed in subsection 0.3.

[1] reference https://www.drdobbs.com/jvm/an-algorithm-for-compressing-space-and-t/184406478

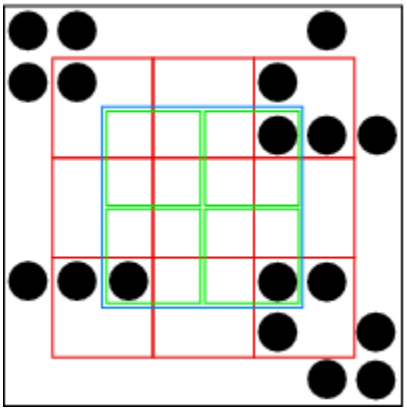

Figure 5: We construct 9 new nodes(red boxes) and splitting blue box as 4 green boxes. Each of these green box is a result of appropriate combinations of red boxes. Combining these green boxes we get result of the parent node.