

mafam /  
HashLife[Code](#) [Issues](#) [Pull requests](#) [Actions](#) [Projects](#) [Security](#) [Ins](#)[HashLife](#) / [README.md](#)**Matthew McDonald** Git repo of Tomas G. Rokicki's beautiful 2006 DDJ article on hash co...

6530357 · 6 years ago



187 lines (144 loc) · 18.4 KB

Preview

Code

Blame

Raw



# HashLife - Tomas G. Rokicki, DDJ, April 01, 2006

An Algorithm for Compressing Space and Time

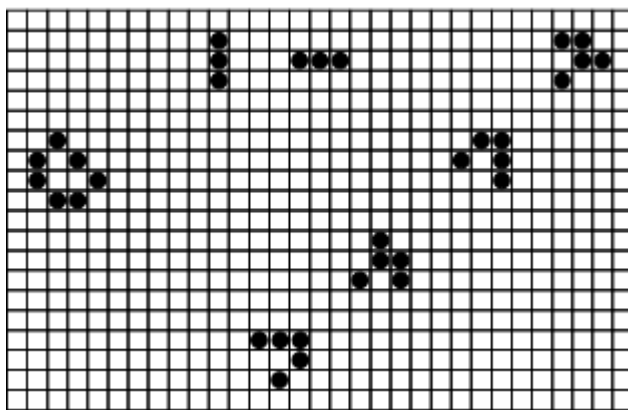
<http://www.drdobbs.com/jvm/an-algorithm-for-compressing-space-and-t/184406478>

By Tomas G. Rokicki, April 01, 2006

Making a slow program fast can lead to both joy and frustration. But sometimes a new approach yields amazing improvements.

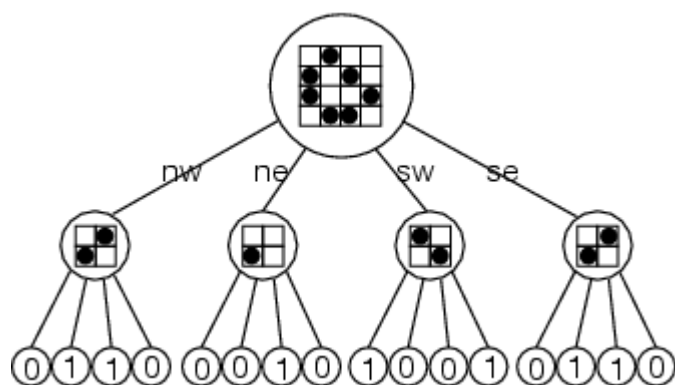
Making a slow program fast can lead to both joy and frustration. Frequently, the best you can do is a low-level trick to double or maybe quadruple the speed of a program; for instance, many readers may have implemented John Conway's "Game of Life" using bit-level operations for a significant speedup. But sometimes a whole new approach, combining just a few ideas, yields amazing improvements. A simple algorithm called "HashLife," invented by William Gosper ("[Exploiting Regularities in Large Cellular Spaces](#)," [Physica 10D, 1984](#)), combines quadrees and memoization to yield astronomical speedup to the Game of Life. In this article, I evolve the simplest Life implementation into this algorithm, explain how it works, and run some universes for trillions of generations as they grow to billions of cells.

Martin Gardner's columns on Conway's Game of Life, published in *Scientific American* from October 1970 onwards, inspired a whole generation of programmers; an amazing world of two-dimensional lifeforms springs from just a few simple rules that are easy to program and display. The Game of Life (Figure 1) is a solitaire game played on an infinite two-dimensional grid. Each cell in the grid is either alive (1, denoted by a black circle) or dead. Every living cell that has two or three living neighbors (of the eight immediately adjacent) remains alive, otherwise it dies. Any dead cell surrounded by exactly three living neighbors is a birth cell and is born into the next generation. All the rules are applied to all the cells at the same instant. These rules lead to life forms that are stable and oscillate—gliders and spaceships that transport themselves across the universe, life forms that grow without bound or completely die away. Seed patterns with as few as nine initial living cells can mutate for many thousands of generations, with constantly moving areas of chaotic activity spewing forth gliders among a large stable menagerie.



**Figure 1:** *Some Life examples. On the left is a stable loaf pattern. In the upper left are the two phases of the blinker, an oscillator with period 2. In a diagonal line from the lower left to the upper right are the four phases a glider goes through as it traverses the universe.*

The simplest implementations of Life use a pair of two-dimensional arrays representing the state of a finite portion of the universe. In every generation, the neighbors of a cell in the old array are counted to calculate the state of that cell in the new array, then the arrays are swapped and the screen is updated. This approach has a number of frustrating aspects: The larger the universe, the slower the program runs; yet, a smaller universe bounds the growth of patterns, constraining them artificially. My first improvement to the basic algorithm is to replace this finite universe with an unbounded one; at any given time it is finite, but you can increase its size as necessary. I use a simple tree representation, called a "quadtree" (Figure 2).



**Figure 2:** A quadtree representation of a 4×4 cell containing a Loaf.

Each node of the tree represents a square portion of the universe. The leaves of the tree are single bits, either 1 (alive) or 0 (dead). Each nonleaf node represents a larger square composed of four children, who are named by their direction from center: nw is the name of the northwest smaller square, ne the northeast smaller square, and so on. The level of a node is the distance to the leaves; the leaves are at level 0. A node at level  $n$  thus represents a square of size  $2^n$  on each edge.

The straightforward way to implement the Life algorithm is to write a recursive function that takes a tree at a given level and either updates the tree in place or returns an entirely new tree. I choose to return an entirely new tree. If I could return a tree at the same level, the algorithm would be very simple. Unfortunately, this is not possible without some additional information because the neighbors of that node also influence the calculation of the next generation for the border cells. One approach to solve this would be to pass in the neighbor nodes at the same level; another is to maintain neighbor node pointers. Instead, I choose a simpler approach: The result of the recursive function is a node one level down and centered with respect to the original node. For instance, the function takes a node at level 2, representing a 4×4 square of the universe, and return a node at level 1, representing the 2×2 square you can directly compute from the information in that 4×4 square. Similarly, it takes a node at level 5, representing a 32×32 square of the universe, and return a square at level 4, representing the central 16×16 pixels that are one generation advanced. This decision lets you use a completely functional approach on an immutable data structure, assuming you enjoy such things.

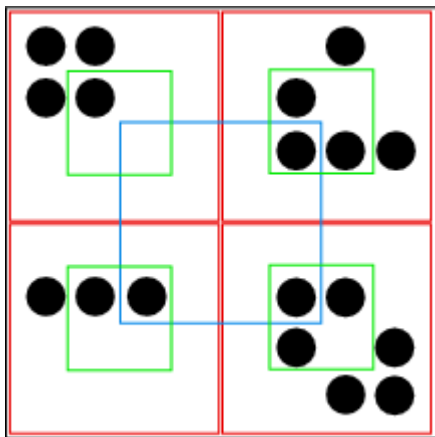
Most recursive algorithms working on trees simply recurse on the children and somehow combine the result. If I do that here, using code like Listing One, there will be gaps between the squares calculated, as in Figure 3. Something more complicated is called for. I cannot use the existing children directly in the recursive call; instead, I must create temporary children that are shifted in place, to make sure I get a result square aligned the way we require.

### Listing One

```

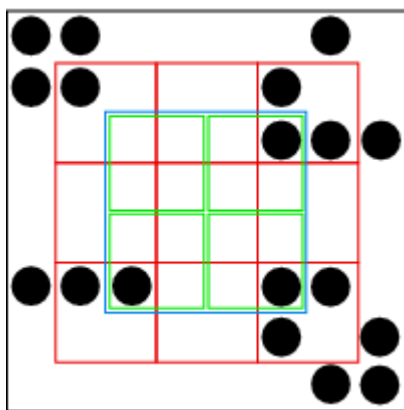
class Node {
    Node nextGeneration() {
        if (level == 2) {
            ... do base case through normal simulation ...
        } else {
            return new Node(nw.nextGeneration(),
                ne.nextGeneration(),
                sw.nextGeneration(), se.nextGeneration())
        }
    }
}

```



**Figure 3:** *Why a simple recursive algorithm will not work. The black outer square represents the area of a node; the four red inner squares are the subnodes. I want to calculate the area of the blue inner square in the next generation, but the inner squares of the subnodes are the green squares, which cannot be combined to form the blue inner square because they do not overlap it.*

I construct nine new nodes two levels down that are shifted appropriately. To construct these nodes two levels down, I actually access nodes that are three levels down to compose the new ones. From these nine new nodes, I create four new nodes that are one level down. It is these four new nodes that I recurse on. This gives me result nodes at the appropriate locations so I can combine them into the required output node at the right level (see Figure 4).



**Figure 4:** *A solution to the gap problem: Construct new subnodes from the components of the existing node to calculate the squares we actually need. Again, blue is the area I wish to calculate; I split it into four green nodes. I compute these four blue nodes by building subnodes out of various combinations of the nine red subnodes that can be constructed from subnodes of the original node.*

I do make a few minor concessions to practicality in Listing Two. Any tree that I know is completely empty, I construct by sharing nodes; see Figure 4. When I calculate the next generation of any node, I first check to see if it is empty and immediately return an empty subtree if so.

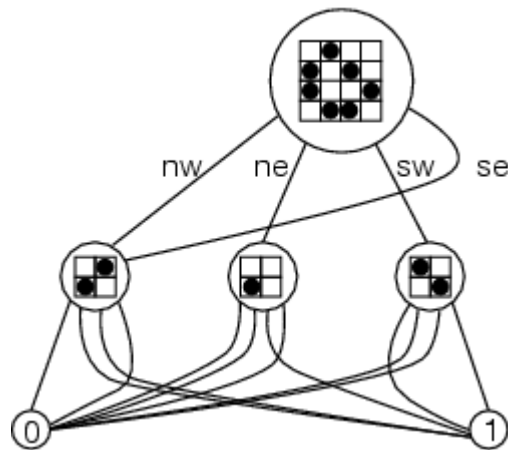
## Listing Two



```
Node centeredSubnode() {
    return new Node(nw.se, ne.sw, sw.ne, se.nw) ;
}
Node centeredHorizontal(Node w, Node e) {
    return new Node(w.ne.se, e.nw.sw, w.se.ne, e.sw.nw)
;
}
Node centeredVertical(Node n, Node s) {
    return new Node(n.sw.se, n.se.sw, s.nw.ne, s.ne.nw);
}
Node centeredSubSubnode() {
    return new Node(nw.se.se, ne.sw.sw, sw.ne.ne,
se.nw.nw) ;
}
Node nextGeneration() {
    if (level == 2) {
        ... do base case through normal simulation ...
    } else {
        Node n00 = nw.centeredSubnode(),
            n01 = centeredHorizontal(nw, ne),
            n02 = ne.centeredSubnode(),
            n10 = centeredVertical(nw, sw),
            n11 = centeredSubSubnode(),
            n12 = centeredVertical(ne, se),
            n20 = sw.centeredSubnode(),
            n21 = centeredHorizontal(sw, se),
            n22 = se.centeredSubnode() ;
        return new Node(
            new Node(n00, n01, n10, n11).nextGeneration(),
            new Node(n01, n02, n11, n12).nextGeneration(),
            new Node(n10, n11, n20, n21).nextGeneration(),
            new Node(n11, n12, n21, n22).nextGeneration());
    }
}
```

So far, my algorithm is not any better. It takes more memory and is slower. It generates a lot of new nodes that need to be managed or garbage collected. It has a somewhat complicated recursion. I fix all of these issues in two steps—canonicalization of the nodes, and "memoization."

The quadtree data structure (see Figure 5) takes significantly more space than a simple bitmap (although only by a constant factor); for a tree representing a  $256 \times 256$  universe, the number of 0 leaves plus the number of 1 leaves is 65,536, and there are  $16,384 + 4096 + 1024 + 256 + 64 + 16 + 4 + 1$  or 21,845 nonleaf nodes. Each node represents a constant, immutable bitmap, so there is no need to have distinct nodes that share the same value. All of the 1 leaves can be represented by a single canonical 1 leaf; similarly, all nodes whose northwest quadrant is a 1 leaf and whose other three quadrants are 0 leaves can be represented by a single canonical instance of this node. This canonicalization takes place all the way to the root. Canonicalizing the nodes requires a simple hash set with the usual recurrence on trees; once the nodes are canonicalized, the value of a node is completely represented (including for comparison) by the pointer to that node, so the hash function of a node can be as simple as some mixing of the addresses of the four subnodes. This canonicalization step is similar to what the Java String intern function does and represents one advantage of immutable structures.



**Figure 5:** *Another Methuselah.*

By itself, this canonicalization step is a powerful form of compression of space. All empty areas of the universe compress into a single small set of nodes. Common life forms, such as gliders and blocks, squeeze into commonly shared nodes. Indeed, for some life forms being constructed now, a serialized form of this quadtree provides the only reasonable storage format, as the forms are simply too large to be distributed in the run-length-encoded format in common use.

Further, the hash set providing the canonicalization can be trivially garbage collected, even in languages like C++, when desired (assuming the hash set stores pointers); simply create a new empty hash set, swap it with the current canonicalized hash set, recanonicalize from the current root, then delete all nodes that are in the old hash set but not the new one.

The canonicalization is implemented by the CanonicalTreeNode class, where I provide the necessary infrastructure for hashing and override the factory methods responsible for creating every node instance.

Having solved the space problem, I make a simple change to the nextGeneration function to solve the time problem: I "memoize" the function (that is, I save the computed answer for later reuse, instead of recomputing it). Memoization is nothing more than caching the results of a function in case it is called again with the same arguments. Consider the classic example of memoization, the Fibonacci function. The usual recursive formulation is as in Example 1(a). This function grows exponentially in  $n$ , as do the number of calls required to compute it. But adding a simple array as a cache, as in Example 1(b), makes it run in linear time in  $n$ . While there are even more efficient ways to calculate Fibonacci numbers, the technique itself is widely useful.

### Example 1:

#### (a) *Recursive formulation.*

```
int fib(int n) {  
    if (n < 3)  
        return 1 ;  
    return fib(n-1) + fib(n-2) ;  
}
```



#### (b) *Linear time version using cache.*

```
int cache[] ;  
int fib(int n) {  
    if (n < 3)  
        return 1 ;  
    if (cache[n])  
        return cache[n] ;  
    return cache[n] = fib(n-1) + fib(n-2) ;  
}
```



To memoize nextGeneration, I add a single pointer to the node data structure that stores the result of the nextGeneration function on that node; if this pointer is not null, it contains the result node. The speedup attained through the memoization that was enabled by keeping nextGeneration purely functional more than offsets the inefficiency introduced by returning a node a level down from its argument. The class that implements this extension is MemoizedTreeNode.




At this point, the Life algorithm is not quite HashLife yet. On some large universes with regular patterns, it is faster than conventional algorithms, even highly optimized ones. For instance, on the classic breeder (refer back to Figure 4), the time per generation is constant, even though the population rises with the square of the number of generations. All the infrastructure we have constructed so far lets you make one final huge step towards creating the final HashLife algorithm—compression of time.

Earlier I mentioned I would run some nontrivial patterns for trillions of generations. Even just counting to a trillion takes a fair amount of time for a modern CPU; yet HashLife can run the breeder to one trillion generations, and print its resulting population of 1,302,083,334,180,208,337,404 in less than a second. This requires only a relatively small change to the program I have described so far. (You may want to stop reading here and see if you can figure out what this change is.)

The final adjustment is to change the number of generations that the recursive call computes. Currently, it computes only a single generation at every level. This can provide a speedup, but the runtime will always be at least linear in the number of generations, even for an empty universe. Instead, we rewrite the `nextGeneration` function to compute a number of generations that increases for each level. That is, at level two, and a  $4 \times 4$  node, it will compute one generation forward, as it stands; at level three, representing an  $8 \times 8$  node, it computes two generations; at level eight, with a  $256 \times 256$  node, it computes 64 generations in the future.

It turns out this change simplifies the recursive function somewhat, because the computation of the next generation will take care of some of the node shifting that necessitated so many additional functions earlier. Listing Three shows what the code becomes. The full implementation is in the class `HashLifeTreeNode` (see attached zip). The code provided there always takes steps according to the current level of the root; this can be modified by using the original `nextGeneration` function we gave for `TreeNode` for levels higher than a fixed value, and the new function for those below that level. If a larger step size is desired, the level of the root node can be easily increased by calls to the `expandUniverse` method.

### Listing Three



```
Node horizontalForward(Node w, Node e) {
    return node(w.ne, e.nw, w.se, e.sw).nextGeneration();
}
Node verticalForward(Node n, Node s) {
    return node(n.sw, n.se, s.nw, s.ne).nextGeneration();
}
Node centeredForward() {
    return node(nw.se, ne.sw, sw.ne, se.nw).nextGeneration() ;
}
Node nextGeneration() {
    if (level == 2) {
        ... do base case through normal simulation ...
    } else {
        Node n00 = nw.nextGeneration(),
            n01 = horizontalForward(nw, ne),
            n02 = ne.nextGeneration(),
            n10 = verticalForward(nw, sw),
            n11 = centeredForward(),
            n12 = verticalForward(ne, se),
            n20 = sw.nextGeneration(),
            n21 = horizontalForward(sw, se),
            n22 = se.nextGeneration() ;
        return new Node(
            new Node(n00, n01, n10, n11).nextGeneration(),
            new Node(n01, n02, n11, n12).nextGeneration(),
            new Node(n10, n11, n20, n21).nextGeneration(),
            new Node(n11, n12, n21, n22).nextGeneration());
    }
}
```

This is HashLife. As presented, this code obtains amazing speedups on most common patterns; for most patterns, after an initial warmup period during which it runs slower than a conventional algorithm due to all the hashing, it then "runs away," computing exponentially increasing numbers of generations per unit time.

Let me return to the beginning. In December 2000, on a private mailing list, Nick Gotts posted a new and wonderful 52-cell pattern called "metacatacryst." He was confident it exhibited quadratic growth, but was not absolutely certain. He asked if anyone could run a HashLife against it. Having just finished a fast conventional Life program, and having seen enough hints and rumors about how such a program might work, and since Bill Gosper's original implementation only worked on Lisp machines of which there were few, I thought it was time to try to write one. With my embryonic incorporation of the ideas I have presented so far, I was able to run metacatacryst to many trillions of generations and show that it exhibited quadratic growth throughout that range. This pattern is amazingly beautiful with fractal properties that cannot be appreciated without running it for billions of generations and having the ability to display the resulting universe in some sufficiently scaled, zoomable form.

I have also used HashLife to find another Methuselah, which is a small initial life form that lasts a long time before setting down, with 12 cells in an initial population that lasts over 12,000 generations. Other people are using it for investigations of large constructed patterns that simulate Turing machines, register machines, and spaceships of previously unattained speeds.

The real joy is the algorithm, and that is entirely due to Bill Gosper, to whom I accord all credit and honor. For those who want to experiment, the open-source program Golly is available at <http://sf.net/projects/golly/>.

HashLife is a unique algorithm consisting of memoization of the Life next-generation function applied to a quadtree representation of the universe. The same technique can be applied to other domains as well.

[Download the sourcecode that accompanies this article.](#)

Tomas is Director of Technology at Instantis. He can be reached at [rokicki@gmail.com](mailto:rokicki@gmail.com).