The Game of Life and Big-O

Data Structures

© Jeff Parker, 2000

Outline

Introduction to the analysis of time

Introduction to the Game of Life

Sample Algorithm

- Estimates of time required
- Speeding up the algorithm
- Rethinking the algorithm

Complexity

The time it takes to solve a problem often depends upon the "size" of the problem.

To add up 100 numbers takes 99 steps

To add up 200 numbers takes 199 steps

To add up N numbers would take N-1 steps.

The time required to add up the numbers is proportional to the size of the data set.

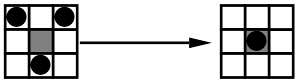
We say this problem is "O(N)" or "Big-Oh of N"

The Game Of Life

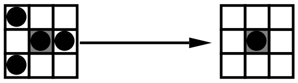
John H. Conway invented a set of rules that can be used to generate a sequence of "pictures"

We use a grid, and define two rules

If an empty cell has 3 neighbors, it will have life in the next generation

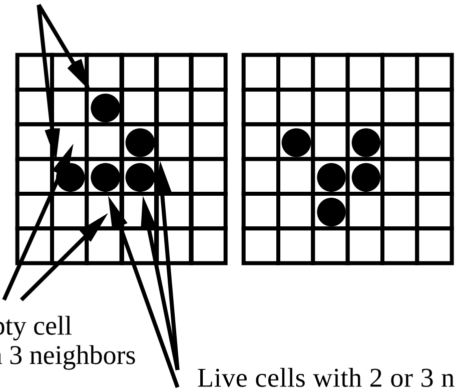


A live cell with 2 or 3 neighbors will continue to live in the next generation



An interesting example

One interesting position is called the "glider"



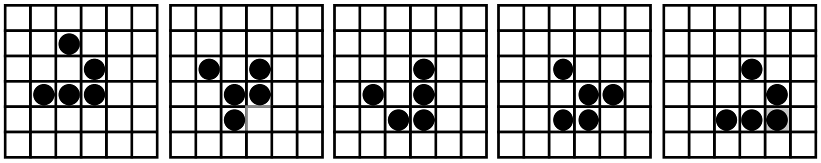
Live cells with less than 2 neighbors

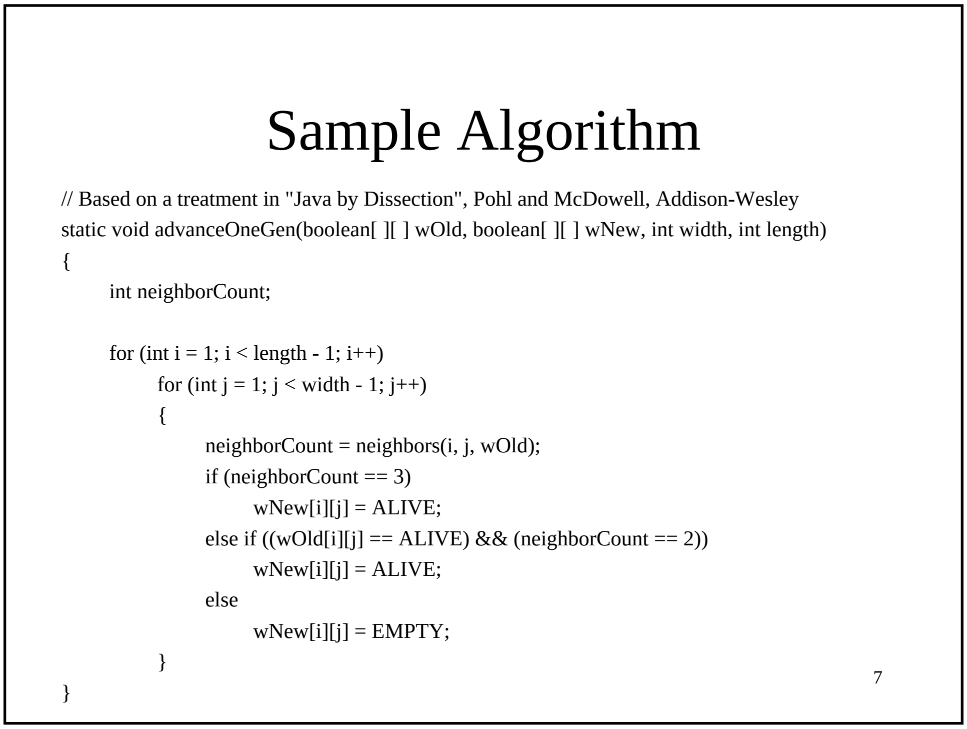
Empty cell
with 3 neighbors

Live cells with 2 or 3 neighbors

5 generations of the glider

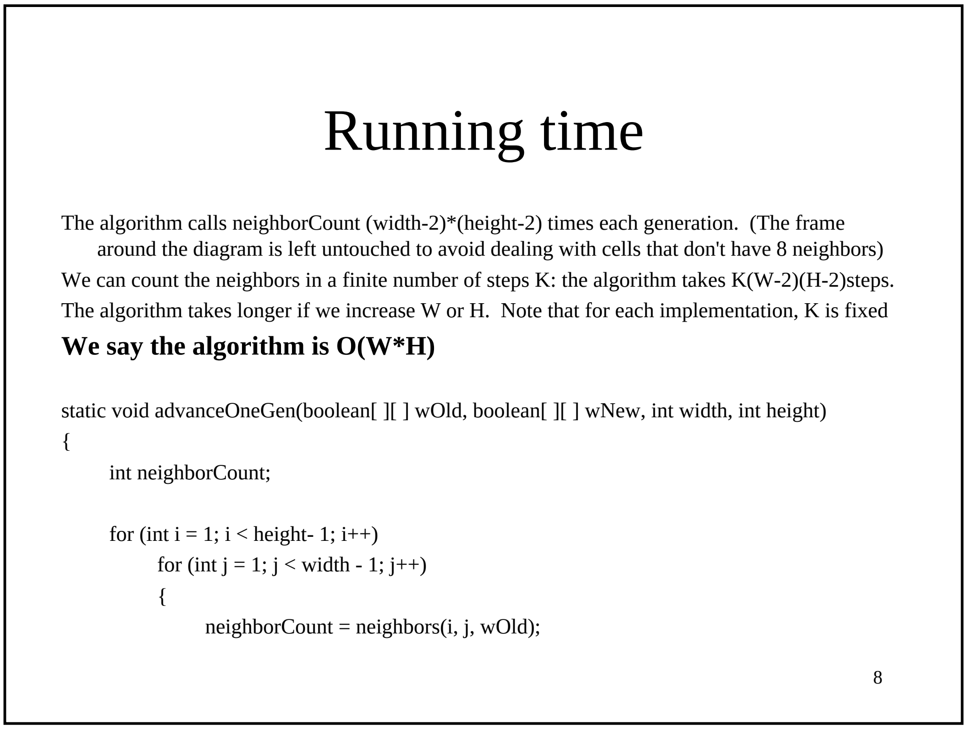
The glider shows that the game of Life has rules that are complex enough to allow movement





Sample Algorithm

```
// Based on a treatment in "Java by Dissection", Pohl and McDowell, Addison-Wesley
static void advanceOneGen(boolean[ ][ ] wOld, boolean[ ][ ] wNew, int width, int length)
{
    int neighborCount;

    for (int i = 1; i < length - 1; i++)
        for (int j = 1; j < width - 1; j++)
        {
            neighborCount = neighbors(i, j, wOld);
            if (neighborCount == 3)
                wNew[i][j] = ALIVE;
            else if ((wOld[i][j] == ALIVE) && (neighborCount == 2))
                wNew[i][j] = ALIVE;
            else
                wNew[i][j] = EMPTY;
        }
}
```



Running time

The algorithm calls neighborCount (width-2)*(height-2) times each generation. (The frame around the diagram is left untouched to avoid dealing with cells that don't have 8 neighbors)

We can count the neighbors in a finite number of steps K: the algorithm takes K(W-2)(H-2)steps.

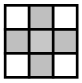
The algorithm takes longer if we increase W or H. Note that for each implementation, K is fixed

We say the algorithm is O(W*H)

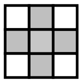
```
static void advanceOneGen(boolean[ ][ ] wOld, boolean[ ][ ] wNew, int width, int height)
{
    int neighborCount;

    for (int i = 1; i < height- 1; i++)
        for (int j = 1; j < width - 1; j++)
        {
            neighborCount = neighbors(i, j, wOld);
```

Count the neighbors

```
// Count the number of alive neighbors of a cell
// Don't count the cell itself
static int neighbors(int row, int col, boolean[ ][ ] w) {
    int neighborCount = 0;

    for (int i = -1; i <= 1; i++)
        for (int j = -1; j <= 1; j++)
            if ((i != 0) || (j != 0))    // Then this isn't the cell itself
                if (w[row + i][col + j] == ALIVE)
                    neighborCount++;

    return neighborCount;
}
```



Steps used



```
for (int i = -1; i <= 1; i++)
    for (int j = -1; j <= 1; j++)
        if ((i != 0) || (j != 0))
            if (w[row + i][col + j] == ALIVE)
                neighborCount++;
```

In C, C++, or Java, an expression like

```
if ((i != 0) || (j != 0))
```

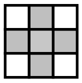
uses "short circuit" evaluation. If (i != 0), then we do not need to test j

Thus the code above performs 6*1 + 3*2 comparisons, and tests the contents of w 8 times, for a total of 20 comparisons

But the contents of w are more likely to be dead than alive...

Improvement

Assume that no more than a third of the cells are alive (this is very generous)

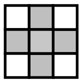


```
for (int i = -1; i <= 1; i++)
    for (int j = -1; j <= 1; j++)
        if (w[row + i][col + j] == ALIVE)
            if ((i != 0) || (j != 0))
                neighborCount++;
```

6 times out of 9 we get off with one test (is this cell alive). Even if we require testing both i and j the three times w is alive, that is still 9 + 3*2 = 15 comparisons.

Better yet

Don't test: just rectify



```
for (int i = -1; i <= 1; i++)
    for (int j = -1; j <= 1; j++)
        if (w[row + i][col + j] == ALIVE)
            neighborCount++;

    if (w[row][col] == ALIVE)
        neighborCount--;
```

This may require an additional addition and subtraction, but takes only 10 comparisons in best, worst, and average case.

Is it worth it?

This simple change more than halved the number of comparisons needed.

We will be able to do much better than this, but it does show some important lessons

1) It is possible to quantify the number of operations required. (We could use more accurate accounting)

2) What we can measure, we can improve

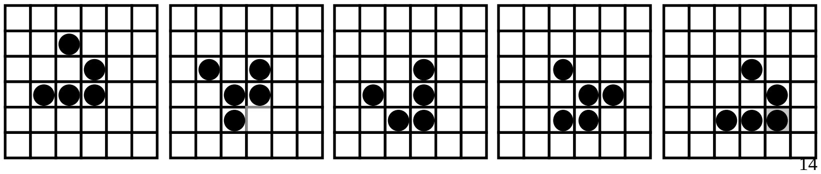
Truth in Learning: The two types of comparisons are not equivalent: it is less work to compare i to 0 than to decide if (w[row + i][col + j] == ALIVE): you need to lookup the value in the array.

Is it worth it?

If you are simulating a glider on a 20 by 20 grid, the old way requires 18*18 * 20 comparisons per generation, while the new way requires less than 18 * 18 * 10 = 3240

Note that we only have 5 live cells.

Given L live cells on an W by H board, can we find an algorithm that requires time proportional to L, rather than W * H?



How can we do better?

There are two ways to survey the neighbors

- Have every cell count every neighbor. (Peering over the back yard fence)
- Have the live neighbors announce themselves (Welcome Wagon approach)

In the second scheme, each live neighbor pitches a welcome basket into the yards of each neighbors.

By counting the baskets, you know how many neighbors you have.

This takes less work, as so few cells have life

New Algorithm

```
// Inform the neighbors
for (int i = 1; i < height - 1; i++)
        for (int j = 1; j < width - 1; j++)
                if (wOld[i][j])
                        informNeighbors(i, j, wOld, temp);

// Set the values of the new array
for (int i = 1; i < height - 1; i++)
        for (int j = 1; j < width - 1; j++) {
                if (temp[i][j] == 3)
                        wNew[i][j] = ALIVE;
                else if ((wOld[i][j] == ALIVE) && (temp[i][j] == 2))
                        wNew[i][j] = ALIVE;
                else
                        wNew[i][j] = EMPTY;
        }
```

Complexity of new algorithm

```
for (int i = 1; i < height - 1; i++)
    for (int j = 1; j < width - 1; j++)
        if (wOld[i][j])
            informNeighbors(i, j, wOld, temp);
```

Still requires W * H steps - but we only run one comparison, rather than 10.

For each live neighbor, we increment 8 numbers

Then we make another pass over the array, looking for live cells

```
// Set the values of the new array
for (int i = 1; i < length - 1; i++)
    for (int j = 1; j < width - 1; j++) {
        if (temp[i][j] == 3)
            wNew[i][j] = ALIVE;
        else if ((wOld[i][j] == ALIVE) && (temp[i][j] == 2))...
```

Improvements on Algorithm

```
// Count the neighbors
for (int i = 1; i < height - 1; i++)
    for (int j = 1; j < width - 1; j++)
        if (wOld[i][j])
            informNeighbors(i, j, wOld, temp);
```

The first phase is really of the form

```
for (all live cells)
    inform the neighbors
```

If we kept a list of the L live cells, we could do this step in O(L) steps

New Algorithm

```
// Set the values of the new array
for (int i = 1; i < height - 1; i++)
    for (int j = 1; j < width - 1; j++) {
        if (temp[i][j] == 3)
            wNew[i][j] = ALIVE;
        else if ((wOld[i][j] == ALIVE) && (temp[i][j] == 2))
```

```
This step is really of the form
    for (all cells with a neighbor)
        check for liveness
```

If we had a list of the cells with neighbors, this could be done in at most 8*L steps

New Algorithm

For each cell that was live in the last generation

- Tell all neighbors
- If neighbor's previous count was 0, and neighbor was not alive in the last generation, add to new list "potential"

For each cell that is alive or potential, compute the new status and clear the counts

Since a potential element must have a live neighbor, there are no more than 8*L potential elements

This algorithm takes O(L) steps, and L <= W*H. This algorithm is much better than O(W * H)

Suggested Reading

The class text, Standish, Data Structures in Java, discusses Big-O notation in section B, starting on page 451

Any good book on Data Structures will introduce these ideas. See the Recommend Readings page on the web site for suggested Data Structures texts.

The Game of Life has many interesting figures: some are included on the web site.

http://www.people.fas.harvard.edu/~adm119/homework/hw1/life-patterns.pdf