Lecture P4: Cellular Automata

Array Review

Arrays allow manipulation of potentially huge amounts of data.

- All elements of the same type.
  - double, int
- N-element array has elements indexed 0 through N-1.
- Fast access to arbitrary element.
  - a[i]
- Waste of space if array is "sparse."

Cellular Automata

Cellular automata. (singular = cellular automaton)

- Computer simulations that try to emulate laws of nature.
- Simple rules can generate complex patterns.

John von Neumann. (Princeton IAS, 1950s)

- Wanted to create and simulate artificial life on a machine.
- Self-replication.
- "As simple as possible, but no simpler."

Applications of Cellular Automata

Modern applications.

- Simulations of biology, chemistry, physics.
  - ferromagnetism according to Ising model
  - forest fire propagation
  - nonlinear chemical reaction-diffusion systems
  - turbulent flow
- biological pigmentation patterns
- breaking of materials
- growth of crystals
- growth of plants and animals

- Image processing.
- Computer graphics.
- Design of massively parallel hardware.
- Art.
How Did the Zebra Get Its Stripes?

1-D cellular automata.
  - Sequence of cells.
  - Each cell is either black (alive) or white (dead).
  - In each time step, update status of each cell, depending on color of nearby cells from previous time step.

Example rule. Make cell black at time $t$ if at least one of its proper neighbors was black at time $t-1$.

Cellular Automata: The Code

```c
#include <stdio.h>
#define STEPS 128           // # of iterations to simulate
#define CELLS 256           // # of cells
#define PS 512.0            // size of canvas
int main(void) {
  int i, t;
  int cells[CELLS] = {0}; // cell contents at time t
  int old[CELLS];         // cell contents at time t-1
  cells[CELLS/2] = 1;     // init one cell to black
  // simulate cellular automata
  // INSERT CODE on next slide here
  return 0;
}
```

Cellular Automata: Designing the Code

- How to store the row of cells.
  - An array.

- How to store the history of cells.
  - A multidimensional array.
  - wastes a lot of space
  - An array that stores only previous time step.
  - wastes a little time updating history array

- How to output results.
  - Turtle graphics.
Cellular Automata: The Code

```c
// simulate cellular automata
for (t = 1; t < STEPS; t++) {
    // output current row in turtle graphics
    for (i = 0; i < CELLS; i++) {
        if(cells[i] == 1) {
            printf("F %f %f\n", PS*i/N, PS - PS*j/CELLS);
            printf("S %f\n", PS / CELLS);
        }
    }
    // copy old values
    for (i = 0; i < CELLS; i++) old[i] = cells[i];
    // update new cells according to rule 250
    for (i = 1; i < CELLS - 1; i++) {
        if (old[i-1] + old[i+1] > 0) cells[i] = 1;
        else cells[i] = 0;
    }
}
```

Cellular Automata Rules

Rule 250 (repetition).

Cellular Automata: Change the Rules

What happens if we change the update rule?

**New rule.** Make cell black at time t if exactly one proper neighbor was black at time t-1.

```
// update new cells according to rule 90
for (i = 1; i < CELLS - 1; i++) {
    if (old[i-1] + old[i+1] == 1) cells[i] = 1;
    else cells[i] = 0;
}
```
Binary Numbers

Binary and decimal representation of integers.

- Binary is base 2.
- Decimal is base 10.

Example.

- $250_{10} = 1111010_2$.

<table>
<thead>
<tr>
<th>Dec</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
</tr>
<tr>
<td>4</td>
<td>0100</td>
</tr>
<tr>
<td>5</td>
<td>0101</td>
</tr>
<tr>
<td>6</td>
<td>0110</td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dec</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>1000</td>
</tr>
<tr>
<td>9</td>
<td>1001</td>
</tr>
<tr>
<td>10</td>
<td>1010</td>
</tr>
<tr>
<td>11</td>
<td>1011</td>
</tr>
<tr>
<td>12</td>
<td>1100</td>
</tr>
<tr>
<td>13</td>
<td>1101</td>
</tr>
<tr>
<td>14</td>
<td>1110</td>
</tr>
<tr>
<td>15</td>
<td>1111</td>
</tr>
</tbody>
</table>

Cellular Automata Rules

- Rule 250 (repetition).
- Rule 90 (nesting).
- Rule 30 (randomness).
- Rule 110 (localized structure).

Example:

- $1111010_2 = 250_{10}$
- $01011010_2 = 90_{10}$
- $00011110_2 = 30_{10}$
- $01101110_2 = 110_{10}$
Cellular Automata Rules

Rule 250 (repetition).

Rule 90 (nesting).

Rule 30 (randomness).

Rule 110 (localized structure).

Source: A New Kind of Science by Steve Wolfram.
Pigmentation Patterns on Mollusk Shells

Source: A New Kind of Science by Steve Wolfram.

Cellular Automata: Designing the Code

Rule 30.

old[i-1]  old[i]  old[i+1]
0 1 0

possible code for rule 30

// update new cells according to rule 30
for (i = 1; i < CELLS - 1; i++) {
    if ((old[i-1] == 1 && old[i] == 0 && old[i+1] == 0) ||
        (old[i-1] == 0 && old[i] == 1 && old[i+1] == 1) ||
        (old[i-1] == 0 && old[i] == 1 && old[i+1] == 0) ||
        (old[i-1] == 0 && old[i] == 0 && old[i+1] == 1))
        cells[i] = 1;
    else cells[i] = 0;
}

Cellular Automata: Designing the Code

Rule 30.

// rule 30
int rule[8] = {0, 1, 1, 1, 1, 0, 0, 0};

// update new cells according to arbitrary rule
for (i = 1; i < CELLS - 1; i++) {
    val = 4*old[i-1] + 2* old[i] + old[i+1];
    cells[i] = (RULE >> val) & 1;
}

Ex.

rule = 3010 = 000111102
val = 210 = 0102

000111102 >> 210 = 0001112

001112 & 0000012

0000012

# define RULE 30

// update new cells according to arbitrary rule
for (i = 1; i < CELLS - 1; i++) {
    val = 4*old[i-1] + 2* old[i] + old[i+1];
    cells[i] = (RULE >> val) & 1;
}
Two Dimensional Cellular Automata

2-D cellular automata.
- $N \times N$ grid of cells.
- Each cell is either black (1) or white (0).
- Update status of each cell, depending on neighbors.
- Repeat.

Example.
- Update cell $(i, j)$ by considering all cells within Manhattan distance 3 of $(i, j)$.
- Color cell $(i, j)$ black if following SUM is greater than 1.4:
  - add together twice the color values of each cell that is distance 0 or 1 away
  - subtract 120% of color values of cells that are distance 2 or 3 away

```
#include <stdio.h>
#define N 512
#define STEPS 10
#define GRIDSIZE 512.0
#define MAXDIST 3

int randomInteger(int n) { . . . }
int cells[N][N];   // two dimensional array

int main(void) {
  int i, j, k, ii, jj, dist, color;
  double sum, threshhold = 1.4;
  double weight[MAXDIST + 1] = {2.0, 2.0, -1.2, -1.2};

  // initialize with random pattern
  for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
      cells[i][j] = randomInteger(2);

  for (k = 0; k < STEPS; k++) {
    for (i = 0; i < N; i++) {
      for (j = 0; j < N; j++) {
        // update cell (i, j)
        sum = 0.0;
        // consider only cells within distance 3
        for (ii = 0; ii < N; ii++) {
          for (jj = 0; jj < N; jj++) {
            dist = abs(ii-i) + abs(jj-j);
            color = cells[ii][jj];
            if (dist <= MAXDIST) sum += weight[dist] * color;
          }
        }
        if (sum > threshhold) cells[i][j] = 1;
        else cells[i][j] = 0;
      }
    }
  }

  for (a = 0; a < 10; a++) {
    for (b = 0; b < N; b++) {
      for (c = 0; c < N; c++) {
        for (d = 0; d < N; d++) {
          for (e = 0; e < N; e++) {
            // code here in innermost loop is executed 10N^4 times.
          }
        }
      }
    }
  }
```

Why Are You So Slow?

Why is the program so slow?
- 221 seconds for $N = 128$.
  - $10N^4 > 2.7$ billion
- 15.7 hours for $N = 512$.
  - $10N^4 > 687$ billion.

A quintuply nested loop

```
for (a = 0; a < 10; a++) {
  for (b = 0; b < N; b++) {
    for (c = 0; c < N; c++) {
      for (d = 0; d < N; d++) {
        for (e = 0; e < N; e++) {
          // code here in innermost loop is executed 10N^4 times.
        }
      }
    }
  }
```
Two Dimensional Cellular Automata

2-d cellular automata

for (k = 0; k < STEPS; k++) {
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            // update cell (i, j)
            sum = 0.0;

            // consider only cells within distance 3
            for (ii = i - MAXDIST; ii <= i + MAXDIST; ii++) {
                for (jj = j - MAXDIST; jj <= j + MAXDIST; jj++) {
                    dist = abs(ii-i) + abs(jj-j);
                    color = cells[ii % N][jj % N];
                    if (dist <= MAXDIST) sum += weight[dist] * color;
                }
            }

            if (sum > 0) cells[i][j] = 1;
            else cells[i][j] = -1;
        }
    }
}

Algorithmic Speedup

Original program. (code in innermost loop executed $10N^4$ times)

- 221 seconds for $N = 128$.
  - $10N^4 > 2.7$ billion
- 15.7 hours for $N = 512$.
  - $10N^4 > 687$ billion.

Improved program. (code in innermost loop executed $490N^2$ times)

- 1.1 seconds for $N = 128$.
  - 200x speedup
- 17.5 seconds for $N = 512$.
  - 3000x speedup

Stay tuned: analysis of algorithms lecture.

Tweaking the parameters.

Source: A New Kind of Science by Steve Wolfram.
Conway's Game of Life

- Based on theories of von Neumann.
- 2-D cellular automaton to simulate synthetic universe.
- Can also simulate general purpose computer.

Critters live and die in depending on 8 neighboring cells:
- too few? (0 or 1)
  - die of loneliness
- too many? (4-8)
  - die of overcrowding
- just right? (2 or 3)
  - survive to next generation
- exactly 3 parents?
  - critter born in empty square

Glider gun.
- Produces a new glider every 30 iterations.

Glider gun synthesizer.
- 8 gliders collide to form a glider gun.

John Conway