CSCI165 Computer Science II
Midterm Project: Spring 2024
100 Points, Due Sunday March 24, 2024

Simulating the Spread of Fire

Fighting fires in Southern California or anywhere else is a very risky job, where loss of life is a real possibility. Proper training is essential. In the United States, the National Fire Academy, established in 1974, presents courses and programs that are intended "to enhance the ability of fire and emergency services and allied professionals to deal more effectively with fire and related emergencies" (Studebaker 2003). The Academy has partnered with private contractors and the U.S. Forest Service to develop a three dimensional land fire fighting training simulator. This simulator exposes trainees to a convincing fire propagation model, where instructors can vary fuel types, environmental conditions, and topography. Responding to these variables, trainees may call for appropriate resources and construct fire lines. Instructors may continue to alter the parameters, changing fire behavior. Students can review the results of their decisions, where they can learn from their mistakes in the safety of a computer laboratory (Studebaker 2003).

Modeling is the application of methods to analyze complex, real-world problems in order to make predictions about what might happen with various actions. When it is too difficult, time-consuming, costly, or dangerous to perform experiments, the modeler might employ **computer simulation**, or having a computer program imitate reality, in order to study situations and make decisions. By simulating a process, they can consider various scenarios and test the effect of each.

Cellular Automaton Simulation

One way to look at the world is to study a process as a group of smaller pieces (or cells or sites) that are somehow related.

- Each cell corresponds to an area (or volume) in the world. The cells could represent geographic regions or individual humans.
- Each cell can be associated with one of several possible states at any given time.
- Each cell can transition between the possible states due to circumstances in the surrounding area.

One convenient way to model the world in computer memory is as a rectangular grid of cells (think: 2 dimensional array).

Transition rules specify how a cell changes state over time based on the states of the cells around it. A computer simulation involving such a system is called a **cellular automaton**. Cellular automata are dynamic computational models that are discrete in space, state, and time. We picture space as one-, two-, or three-dimensional (also sometimes called an array, matrix or lattice). A site (or cell . . . array element), of the grid has a ***state***, and the number of states is finite. Rules specifying local relationships and indicating how cells are to change state, regulate the behavior of the system. An advantage of such grid-based models is that we can visualize the progress of events through informative animations. For example, we can view a simulation of the movement of ants toward a food source, the propagation of infectious diseases, heat diffusion, distribution of pollution, or the motion of gas molecules in a container.

Initializing the System

In many simulations, we model a dynamic area under consideration with an n-by-n matrix of numbers (or objects), where a number represents a state (see Figure 1). Each cell in the grid contains a value representing a characteristic of a corresponding location.

For example, in a simulation for the spread of fire, a cell may contain one of three values:

- 0 to indicate an empty cell with no tree or a burnt tree.
- 1 to indicate a cell with a non-burning tree
- 2 to indicate a cell with a burning tree.
- These are just the bare minimum. The potential for detailed states is limitless. There could be different types of trees, trees that are dead or dying (thus increasing their probability of catching fire) or trees that are particularly resistant to fire.

Table 1 lists these values and meanings along with associated names **EMPTY, TREE,** and **BURNING** that have values of 0, 1, and 2, respectively. We initialize these constants at beginning and employ the descriptive names throughout the program. Thus, the code is easier to understand and to change. ***An object containing an enumeration is perfect for this.***

Figure 1 Cells to model area with associated numeric values

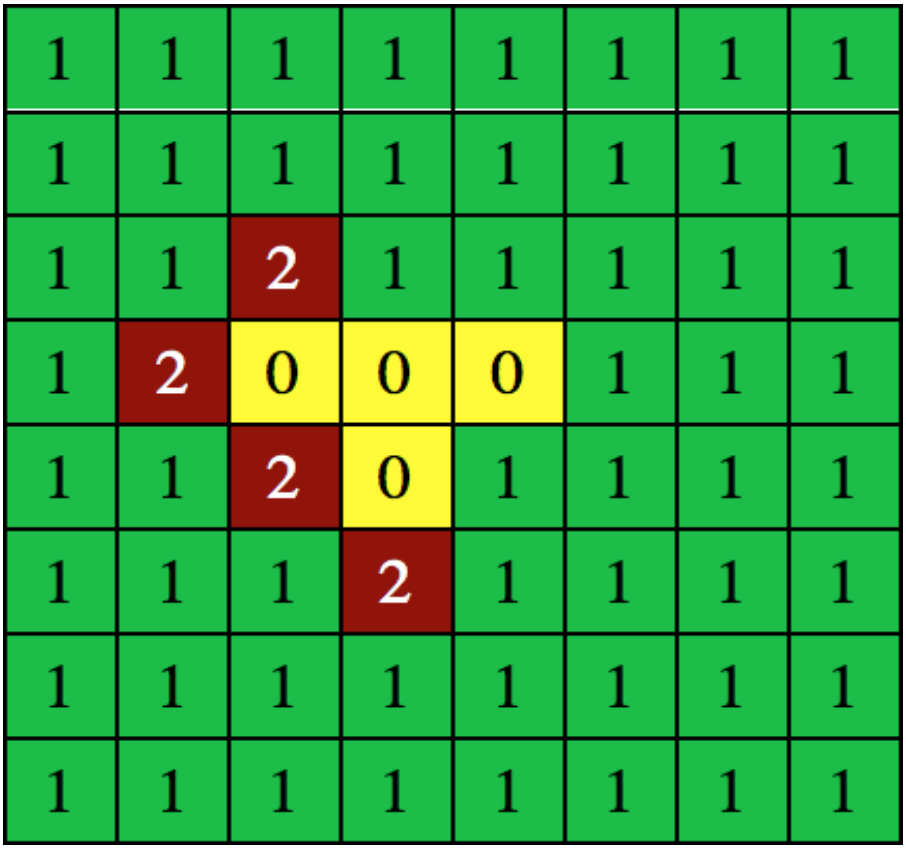
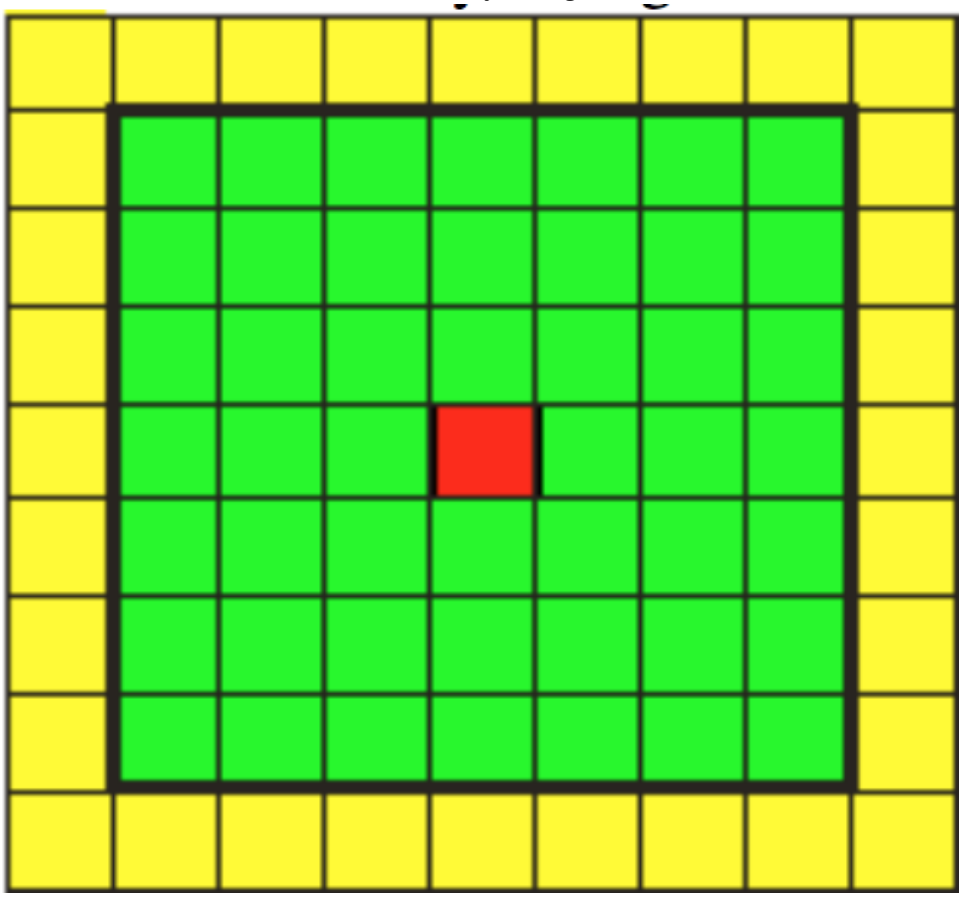


Table 1 Constants and their associated values

Value	Constant	Meaning
0	EMPTY	The cell is empty ground or a burnt tree containing no tree.
1	TREE	The cell contains a tree that is not burning.
2	BURNING	The cell contains a tree that is burning.

How do we address the problem of the edge of the world at interest? At minimum, for this simulation, we assume our world is surrounded by empty cells—that is, cells that contain no trees or anything else that can burn. The boundary of empty cells is similar to a firebreak or an area with no trees; proximity to such a boundary cell cannot cause an internal tree to catch fire. This insulating boundary is called an absorbing boundary condition. Figure 2 depicts an initial system with green indicating a non-burning tree, red a burning tree, and yellow an empty cell.

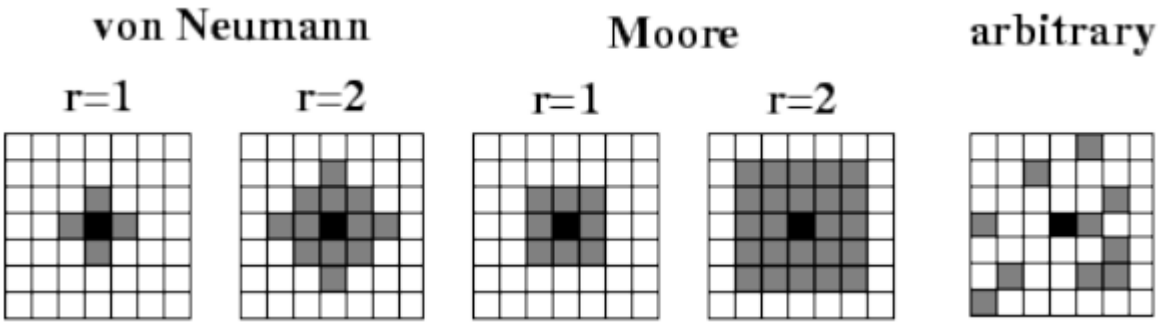
Figure 2 Grid of trees with one burning tree in the middle and with each cell on an extended boundary having a value of EMPTY



Updating Rules

How do we model the spread of fire? If a small area of a forest catches fire, then that fire spreads to a neighboring cell with some probability determined by factors such as wind, precipitation falling, or dryness of the leaves and timber. We model transition rules by a function ***spread***. At each simulation iteration, we apply *spread* to each cell to determine its value —*EMPTY*, *TREE*, or *BURNING*—***at the next time step***. A cell's value at the next instant depends on the cell's current value (***site*)** and the values of its neighbors to the north (***N***), east (***E***), south (***S***), west (***W***), north east (**NE**), south east (**SE**), north west (**NW**) and south west (**SW**). Thus, we use numeric parameters—***site, N, E, S, W, NW, NE, SW and SE***— for *spread*. In a call to this function, each argument is one of the values *EMPTY*, *TREE*, or *BURNING*. **Figure 3** pictures the cells that determine a site's next value along with their indices, where the indices for *site* are **(*i, j*)**. Thus, the term **neighbor** refers to one of the cells directly to the north, east, south, west, north west, north east, south west and south east of a site's cell.

For each cell, a set of cells called its neighborhood is defined relative to the specified cell. The image below shows the ***von Neumann*** and ***Moore*** neighborhoods with sample radius. Any type of neighborhood can be defined as long as it is uniform and finite. At minimum you will be dealing with a ***r = 1 Moore neighborhood,*** but feel free to experiment.



NW (i – 1, j – 1)	**N** (i – 1, j)	**NE** (i – 1, j + 1)
W (i, j – 1)	**SITE** (i, j)	**E** (i, j + 1)
SW (i + 1, j – 1)	**S** (i + 1, j)	**SE** (i + 1, j + 1)

Figure 3 Cells that determine a site's next value in a

r = 1 Moore Neighborhood

Updating rules apply to different situations:

- If a site is empty (cell value *EMPTY*), it remains empty at the next time step.
- If a tree grows at a site (cell value *TREE*), at the next instant the tree may or may not catch fire (value *BURNING* or *TREE*, respectively) due to fire at a neighboring site. We make the simplifying assumption that a burning tree (cell value *BURNING*) always burns down in one time step, leaving an empty site (value *EMPTY*) for the next time step. Obviously this can be tweaked to oblivion
- We also assume that a site with empty ground and a site with a burnt tree are in the same state and have the value *EMPTY*.
- We consider each situation separately.

When a tree is burning, the first argument, which is the site's value, is *BURNING*. Regardless of its neighbors' situations, the tree burns down, so that at the next iteration of the simulation the site's value becomes *EMPTY*. Thus, the relevant rule for the ***spread function*** has a first argument of *BURNING*; each of the other four arguments are immaterial; and the function returns value of *EMPTY*.

To continue development of this dynamic, discrete, stochastic system, we employ the following additional probability:

probCatch – The probability of a tree in a cell catching fire if a tree in a neighboring cell is on fire.

Thus, if a site contains a tree (site value of *TREE*) and fire threatens the tree due to proximity, *probCatch* is the probability that the tree will catch fire at the next time step.

Determine Probability

To illustrate how a probability can be determined during a simulation, suppose ***probCatch is 0.15 = 15%***. For each cell with the value *TREE* that has a burning neighbor, we generate a uniformly distributed random floating point number from 0.0 up to 1.0. On the average, 15% of the time this random number is less than 0.15, while 85% of the time the number is greater than or equal to 0.15.

Thus, we employ the following logic:

if *site* is *TREE* and (*N* | *E* | *S* | *W* | *NW* | *NE* | *SW* | *SE* is *BURNING*)

    if a random number between 0.0 and 1.0 is less than *probCatch*
        transition to *BURNING*
    else
        remain *TREE*

Even if a tree has the potential to burn because of a neighboring burning tree, it may not because of conditions, such as wet weather. Such a tree has a probability of *probCatch* of burning.

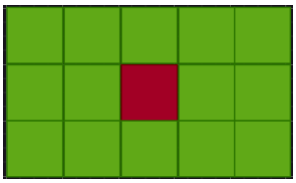
Applying Spread to Each Cell

To run a simulation, we have to represent the ***passing of time and the state of the world at each time step***. Time is represented in discrete units, such as one minute, one hour, one day, and so on. At each discrete time step, the simulation program determines the state of each cell at the ***next time step*** based on the state of each cell at the current time and the states of the cells in the neighborhood. Thus, the program runs a simulation by counting time steps and at each one producing a ***new grid*** for the next time step.

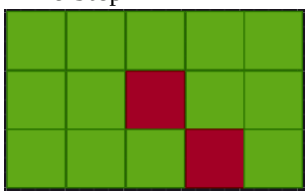
It is important to understand that state transition of a cell is simply recorded for the next time step, and does not have any effect in the ***current time step.*** A burning cell can only affect one of it's direct neighbors in the current time step. A time step consists of an iteration over the matrix from **[0][0]** to **[matrix.length – 1][matrix[row].length -1]**

For instance: Given the following sample matrix and assuming a left-to-right row major path

Time Step 1



Time Step 2



Notice that when cell [2][3] is encountered it transitions from TREE to BURNING. That transition should have no effect on any of cell[2][3] neighbors in the current time step, so when the iteration steps to cell [2][4], the state of cell[2][3] should still be TREE. The transition is not active until the next time step.

To that end, we define a function ***applySpread*** that takes a grid (matrix) and a probability of the fire spreading to a cell (*probCatch*) and returns another grid with *spread* applied to each internal cell of *matrix*. The function *applySpread* applies ***spread(site, N, E, S, W, NE, NW, SE, SW)*** to each internal cell in grid—that is, *applySpread* does not process the cells on the boundary. Thus, for *i* going through the indices for each internal row (not including the first and last rows) and for *j* going through the indices for each internal column (not including the first and last columns), *applySpread* obtains a cell value for a new (n + 2)-by-(n + 2) grid as the application of *spread* to each site with coordinates *i* and *j* and its neighbors with corresponding coordinates as in **Figure 3**

Simulation Program

To perform the simulation of spreading fire, we define a function *fire* with parameters for *n*, the internal grid size, or number of internal grid rows or columns; and *probCatch*, the probability of catching fire. Pseudocode for this version of *fire* is as follows:

***fire*(*n, probCatch*)**

Method to return a list of grids in a simulation of the spread of fire in a forest, where a cell value of *EMPTY* indicates the cell is empty; *TREE*, the cell contains a non-burning tree; and *BURNING*, a burning tree

Pre:

- *n* is the size (number of rows or columns) of the internal grid and is positive.
- *probCatch* is the probability (a number between 0.0 and 1.0) of a tree catching fire.
- *spread* is the function for the updating rules of each cell.

Post: A list of the initial grid and the grid at each time step of the simulation was returned.

Algorithm:

- initialize *forest* to be an (n + 2) x (n + 2) grid of values, as follows:
  - *EMPTY* (no tree) cells are on the boundary and
  - *TREE* (non-burning tree) cells are in the interior except for one *BURNING* (burning tree) cell in the middle
  - *grids* ← a list containing *forest*
- do as long as a burning tree is in *forest*
  - *forest* ← the result of call to *applySpread* with arguments *forest* and *probCatch*, which returns an (n + 2) x (n + 2) grid with *spread* applied to each internal cell of *forest*
  - *grids* ← the list with *forest* appended onto the end of *grids*

    return *grids*

EMPTY	EMPTY	EMPTY	EMPTY	EMPTY	EMPTY	EMPTY	EMPTY	EMPTY
EMPTY	TREE	TREE	TREE	TREE	TREE	TREE	TREE	EMPTY
EMPTY	TREE	TREE	TREE	TREE	TREE	TREE	TREE	EMPTY
EMPTY	TREE	TREE	TREE	TREE	TREE	TREE	TREE	EMPTY
EMPTY	TREE	TREE	TREE	BURN-ING	TREE	TREE	TREE	EMPTY
EMPTY	TREE	TREE	TREE	TREE	TREE	TREE	TREE	EMPTY
EMPTY	TREE	TREE	TREE	TREE	TREE	TREE	TREE	EMPTY
EMPTY	TREE	TREE	TREE	TREE	TREE	TREE	TREE	EMPTY
EMPTY	EMPTY	EMPTY	EMPTY	EMPTY	EMPTY	EMPTY	EMPTY	EMPTY

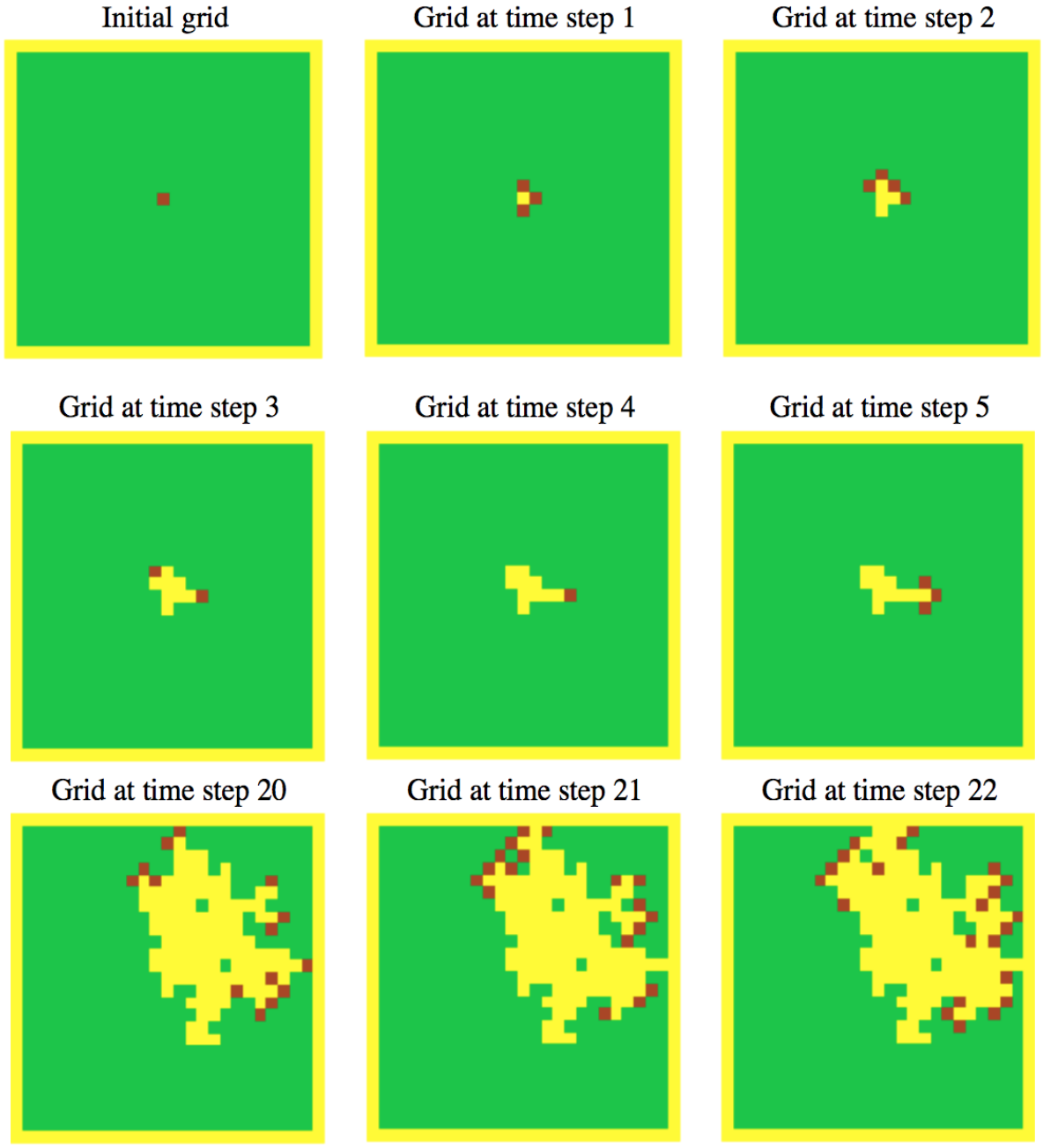
Display Simulation

Visualization helps modelers understand the meaning of the simulation. For each grid in the list returned by fire, we generate a graphic for a rectangular grid with yellow representing an empty site; green, a tree; and red, a burning tree. We animate the sequence of graphics to view the changing forest scene.

Figure 4 displays several frames (time steps 0-5, 20-22) of a fire with

n = 25 and probCatch = 0.55.

Figure 4 Several frames in an animation of the spreading of fire



Assignment

Write a Java implementation of the spreading fire simulation. Implement a rudimentary visualization by animating the grid using ***Java 2D Graphics***. Draw the grid after each time step.

The grid must be flexible. Implement a command line argument to specify the size of the matrix using a single number. For example, if **50** is entered the matrix will be **50 x 50. T**he pixel size of the rectangle may need to be tweaked based on this input.

Object Orientation

This application can benefit from decomposition and object orientation. With that in mind define ***at least*** the following classes. This can be extended at your discretion. Maybe you want to have a Tree class that encapsulates various property of the tree, that could affect the probability of it catching fire.

Class Cell that encapsulates the various properties of a single discrete cell in this automation.

Static (Class) Features

- An enumeration representing the finite collection of states: ***EMPTY, TREE, BURNING*** (add more states if you like. Include comments as to their meaning)
- An array representing the associated colors of these states: ***Yellow, Green, Red.*** The ***java.awt.Color*** class has some nice public static constants representing the RGB values of these colors. If you end up adding additional states you must also define colors for these states.
- In comments in your code include a description of what it means for these features to be static

Instance Features

- An enum variable representing the cell's state.
- These are just suggestions, I will allow you freedom to flex your burgeoning OOP design skills.
- Maybe the a field to track the probability, the type of tree, a moisture factor. Possibilities are endless

Class World that encapsulates the world we are modeling.

Instance Features

- A matrix (2D array) of ***Cell objects***
- The ability to iterate through the matrix and spread fire

I will leave the remaining architecture of the program up to you. I really want to grant you some freedom of design. The functions mentioned in this document are ***only suggestions,*** but you do need to show some ability to decompose tasks and appropriately design a ***very basic*** object oriented system. I would very much enjoy having architecture discussions around this.

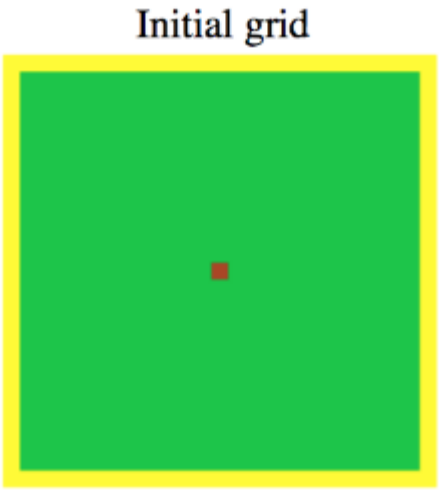
Let's discuss this

Unit Testing: Hopefully the lesson of ***planning*** has been learned at this point. **DO NOT** try to write this application ***all at once***. I cannot stress the importance of this!!! You have two weeks to complete this project, spanning multiple lab days. The required code is surprisingly minimal, but logically dense. Do not try to fire this off without significant planning and testing. This is a great opportunity to develop, test and verify small bits of logic. You can work with small, manageable 2D arrays in isolation, working out the kinks in the state transitions. **This is required!** I want to see a set of unit tests that prove that your state transition logic is on point.

Let's discuss this

What to expect in the next two weeks?

You will spend the next two weeks working on this project (which is more than ample time). We will not be covering any new topics. I will cover the basic concepts of dealing with Java 2D graphics and 2 dimensional arrays. Examples will be provided for you to use as a reference.



The lack of cell borders makes this difficult to imagine but this is a collection of colored 10 x 10 pixel rectangles. Each rectangle corresponds to a single Cell object that exists in a 25x25 2D array. The color of the rectangle is determined by the cell's state when looping over the matrix.

This is the essence of the good ole ***bitmap*** image format: https://en.wikipedia.org/wiki/Bitmap

Class time will be utilized to specifically discuss this project. You will be given assistance if you ask for it.

Please ask questions.

I know that there are some folks who are going to eat this up and ask for more. I do have another project that I like to rotate with this one. I will be glad to share it with anyone who wants to do more of this type of programming.

Rubric on next page.

RUBRIC:

1. **50 Points:** Working application that adheres to the ***minimum*** criteria laid out in this document.
2. **30 Points:** Object Orientation at a minimum based on what is described in this document
3. **10 Points:** A collection of unit tests that demonstrate that you ***spread*** logic functions correctly.
4. **10 Points:** Appropriate use of JavaDoc and regular style commenting and documentation

RESTRICTIONS:

1. If you use any code generated from an AI tool or copied from the internet without appropriate attribution your work will be egregiously penalized. Trust me when I say that this is obvious.

2. You are not allowed to simply search the internet and copy or design this system however you wish. In order to receive full credit, your work must meet the minimum guidelines as laid out in this document. If you have any questions about this please let me know.