A Cellular Automata Implementation of a Wildfire Spread Model

Kristen J. Bains
Adviser: Michael Berry
November 6, 2006

This project's goal was to implement a wildfire spread model with spatial control using a cellular automaton. A literature search did not find any apparent similar approaches. Other cellular automata wildfire spread models exist, but my project makes an attempt at implementing control techniques, specifically, the firebreak. My model differs in the complexity of simulation as well. Many other models aim for accuracy by using complicated mathematical models, such as the Rothermel model. Although, my model uses much simpler calculations, I believe that it behaves realistically.

The project consists of two main components: a graphical user interface (GUI) and the model simulation. I will discuss these components separately.

GUI

The GUI for this project was written in Java. Java was chosen for it's extensive library of GUI controls, Swing. The GUI consists of four main areas. The menu bar is used to control various aspects of model execution. The control panel, located to the left of the screen, allows the user to set various parameters without needing to look in the menu. The image area is located to the right of the screen and is used to display the maps, firebreaks, fire start locations, and output from the model. The palette is located on the far right of the window and gives the user a numerical value for the colors used in the image area.

Menu Bar

File Menu

Figure 1 File Menu

The menu bar contains three main items. The file menu allows the user to load a pre-existing landscape file, load a previous configuration, save a configuration, create a new landscape file, and exit the program. To load a landscape file, the user selects a landscape file from a file chooser dialog. Currently the landscape files are located in the image directory.

There are four pre-existing files in this directory, named `landscape_#.bmp`. The program loads this file into the image area of the screen by overriding paintComponent(). This also writes the image filename and dataset filename to `whichImage.dat` and `whichLandscape.dat` respectively.

The menu item, "File->Load a previous configuration", may be of limited value to the user, and was included mainly as a programming aid. This action loads the image and dataset filenames from `whichImage.dat` and `whichLandscape.dat`, draws the appropriate image to the screen, loads the firebreaks and fire starts from file, and draws these on the image. "Save configuration" simply writes the data required to load a configuration to the needed files.

The create landscape menu item requires the user to set some values in the control panel area "Set values for new landscape". All of these fields have a default value, but the user is encouraged to experiment with the bounding box values especially. These values are used by the `libnoise` module, which is a third-party library used to generate various kinds of noise. A simple explanation of the bounding box values can be found here, http://libnoise.sourceforge.net/tutorials/tutorial3.html. The user can also select the number of rows and columns for the new landscape and name the file something relevant. This filename will be given to both the image file and the data file. When the user selects this option, the Java code writes an initialization file, `landscape/init.dat`, and runs `landscape/generateMap`. When generateMap runs to completion, a halt string is written to stdout and control is given back to the GUI where the new landscape file is loaded. Finally, the "File->Exit" option simply exits the program.

Reset Menu

The reset menu contains three options. "Reset->Reset All" clears the image from the screen and clears the firebreaks and fire start locations. "Reset->Reset breaks" does not clear the landscape image, but clears and resets the fire breaks. Similarly, "Reset->Reset starts" clears the fire start locations.

Run Menu

The run menu allows the user to "Run->Run the model". This option writes the firebreaks and fire starts to their respective data files, writes the init file, and calls the C++ model code. When the code returns the halt string and control is returned to the GUI, the newly burned cells are read from `final.dat` and drawn onto the landscape image.

"Run->Play animation" can be selected after the model has finished running. This routine reads the `iteration.dat` file and creates and animation of the fire. Currently burning cells are drawn in a transparent red color and the burned cells are drawn in a transparent black. The transparency allows the user to visualize the initial firebreaks and fire start locations.

Finally, "Run->Run optimization" requires the user to select an optimization firebreak and a start location. If the firebreak is not selected the user will be prompted. Multiple start locations can be selected but only the first location selected will be used. The user can also set some parameters on the control panel in the "Optimization options" area. "Percent map size to search" limits the search range and "Orientation" allows the user to select from amongst several options. Orientation will refer to the direction that the firebreak will "grow" during the optimization routine. The user is then prompted for a filename for the output. I suggest using the naming convention `*.opt`, but any name is acceptable. These files are stored in the data directory once the optimization routine is finished. The routine writes the initialization file with some added parameters that are used by the optimization routines. The C++ code is invoked, and when control returns to the GUI, the best firebreak will be displayed on the landscape image.

Control Panel

The control panel allows the user easy access to all of the parameters that can be set. In this section I will fully describe each option.

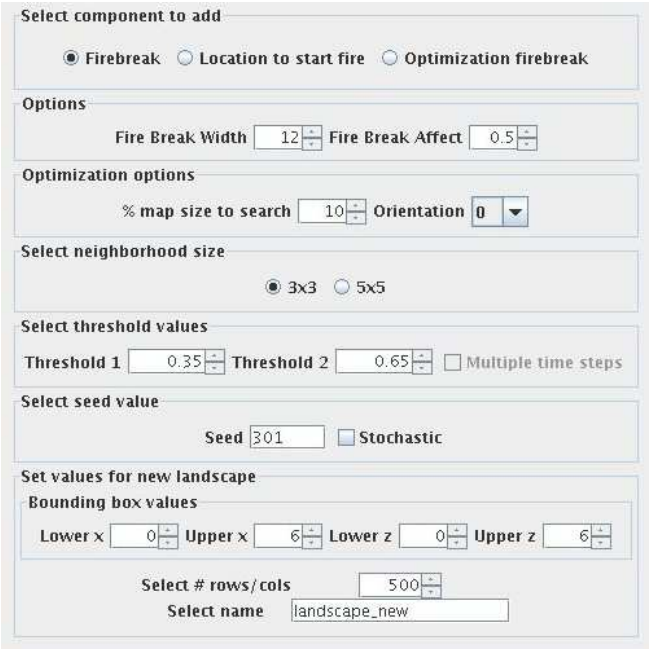


Figure 2 Control Panel

Select component to add

This area allows the user to add components to a loaded landscape image. "Firebreak" lets the user draw a firebreak onto the image, by clicking and dragging the mouse. The program records the (x, y) coordinates from the mouse and stores the values for later use. The firebreaks can be any shape. "Location to start fire" allows the user to select a location to start a fire by clicking on the image. The selected locations are indicated by a small blue bullseye shape. Multiple fire start locations can be selected and are saved and written to a text file. "Optimization firebreak" is similar to selecting a fire start. The user selects a location by clicking on the image and the location is indicated by a small white bullseye shape. The user can only select one location.

Options

This area contains a spinner that lets the user select the width of the firebreak. This width is currently not indicated on the landscape image. The image only displays a width of one. The minimum value is 1 and the maximum value is 100, although these values are easily changed in the GUI code. Another option in this area is currently not enabled. Allowing the computer to randomly assign fire start locations was an idea that I discussed with Jane Comiskey but have not implemented at this time.

Optimization options

"Percent map size to search" allows the user to limit the search range of the optimization routine. Possible values range from 10 to 100%, incrementable by 1%. This value (Formula 1)is used while looping through firebreak lengths in the optimization routine.

$$numRows * percentMapToSearch$$

Formula 1

The "Orientation" option contains four items in a drop down box. "All", "0", "45", "90" and "135" are numerical values representing degrees. "0" degrees indicates a north-south orientation. When a firebreak grows, it will increase by 1 cell in the north direction and 1 cell in the south direction, which means that each growth cycle of the firebreak increases its length by two. When a "0" firebreak increases in width, it grows in an easterly direction. "45" degrees indicates a northeast-southwest orientation. It also grows by two cells each cycle; one cell toward the northeast and one cell toward the southwest. When "45" degrees increases in width, it grows in a southeasterly direction. A "90" degree firebreak is oriented east-west, increases length one cell east and one cell west, and widens in a southerly direction. Finally, "135" degrees indicates a northwest-southeast orientation. See Figure 3 for further clarification. By selecting "All", the optimization routine will loop through these three orientations and return the best firebreak from the three. Other degrees of orientation can be added by simply adding the options to the drop down box in the GUI and adding appropriate routines in the C++ code

and in paintComponent().

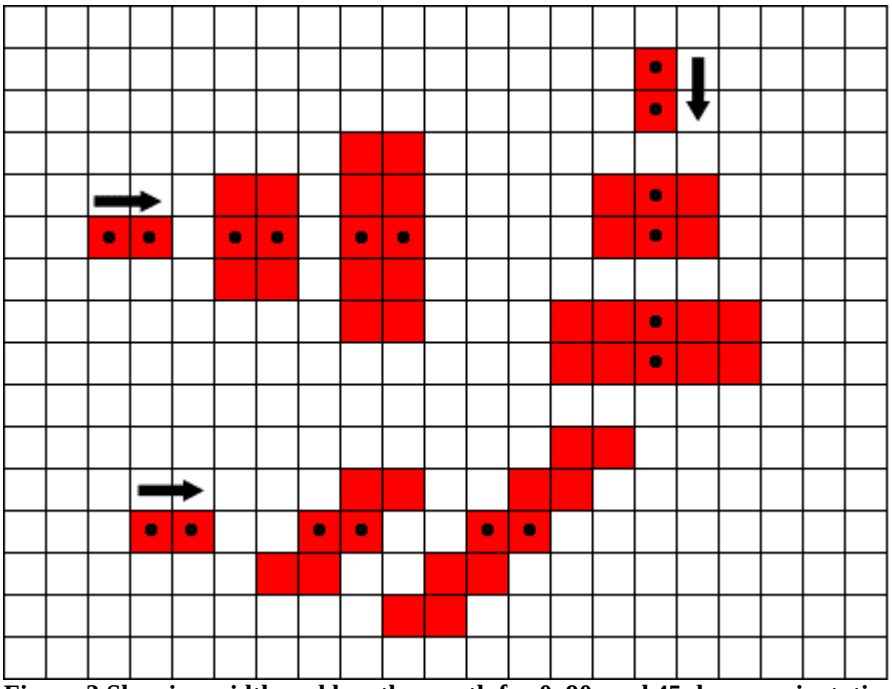


Figure 3 Showing width and length growth for 0, 90, and 45 degree orientations

Select neighborhood size

This area contains a set of radio buttons, of which the user can select one option. "3x3" is the default option and causes the model to use the traditional 8-neighborhood. "5x5" is an interesting option that uses a 24-neighborhood. One method of verifying membership in a neighborhood is by using the distance formula, Formula 2. By taking the floor of the result, you can determine which neighborhood a cell lies in. If the floor is 1, it is in the 8-neighborhood. If the floor is 2, it is in the 24-neighborhood.

$$\left((x_1 - x_2)^2 + (y_1 - y_2)^2\right)^{1/2}$$

Formula 2

Select threshold values

This area contains two spinners and a checkbox. The checkbox allows the user to run the simulation using multiple time steps. The default number of time steps is one, meaning that once a cell begins burning, it will self-extinguish on the next time step. If the user has checked the "Multiple time steps" box, a cell has the potential to burn for two time steps. The values "Threshold 1" and "Threshold 2" control this behavior. "Threshold 1" will be the lower value. If the fuel density of a cell is below this threshold,

the cell will not burn at all. This applies to single time step burns as well as multiple time step burns. "Threshold 2" will be larger than "Threshold 1". If a cell's fuel density is greater than "Threshold 2", the cell will burn for two time steps. For clarification, see Table 1.

Fuel Density (*f*), Threshold 1 (t1), Threshold 2 (t2)	**Action**
$f <$ t1	Will not burn
t1 $< f <$ t2	Will burn 1 time step
$f >$ t1	Can burn 2 time steps

Table 1

Select seed value

This area contains a text field, "Seed", and a checkbox, "Stochastic". The value entered into the text field will be used to seed a pseudo-random number generator. If stochastic is selected, when the simulation is running, if a cell can burn (based on the threshold values described above), a random number will be generated and used to decide if the cell will burn or not.

Set values for new landscape

Finally, this area contains values that are needed when creating a new landscape file. These are explained in detail in the "File Menu" section of this document.

C++ Model Simulation

generateMap

Located in the landscape directory, generateMap is the executable file used to create new landscape maps.

`libnoise` and Perlin noise

generateMap uses a third-party library called `libnoise`, which can be downloaded from http://libnoise.sourceforge.net. I chose this library for its ability to generate Perlin noise. Perlin noise was invented by Ken Perlin to generate textures for the movie *Tron*. Perlin noise can create realistic looking, randomly generated landscape files, which I chose to use for the fuel density maps, instead of obtaining real fuel load maps. The `libnoise` module requires the `noiseutils` files in the landscape directory and the compiled libraries in the lib directory.

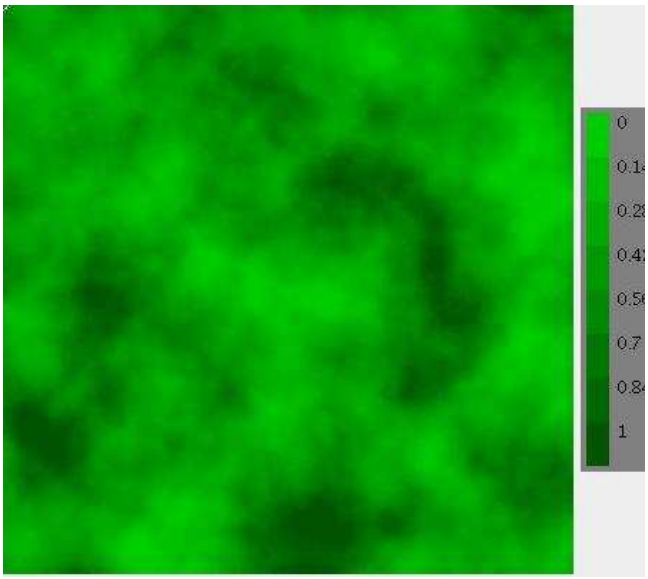


Figure 4 Perlin Noise Generated Image

landscape.cc, main.cc, landscape.h

These files call functions and library routines from `noiseutils` and `libnoise`. The initialization file is read and parsed to set the parameters that are needed to generate a map. After the map has been created, an image file is saved and the raw data is saved to the data/`landscape_#.dat` file. Since Perlin noise generates numbers in the range -1 to 1, these raw numbers are converted to the range 0 to 1. All fuel densities are in this range. Some outliers in the raw data exist and are treated as special cases. The landscape data file is written in the following format: the first row contains the number of rows and the number of columns in the dataset. Subsequent rows of the file contain rows from the raw data array.

Fire

The executable for the simulation is `fire`. There are two main classes defined for the simulation: *CACell* and *CASpace*.

cell.cc, cell.h

The *CACell* class is defined and implemented in `cell.h` and `cell.cc`. This class keeps information about each cell (see Table 2). The methods in this class

int burnState;	BURNING, BURNED, FIREBREAK
int iterationFireStarts;	the iteration number that the fire started
float slope;	Not used
float fuelMoisture;	Not used
float fuelDensity;	The fuel density value of this cell
int Row;	The row number of this cell
int Col;	The column number of this cell
int timeStepsToBurn;	The number of time steps this cell can potentially burn
int timeStepsBurned;	The number of time steps this cell actually burned
ToList toList;	List of type std::list<Pair>, keeps a list of the cells this cell spread fire to

Table 2 CACell Instance Variables

only perform the action of setting/changing instance variable values or retrieving these values. It would be extremely easy to add additional functionality to this model by simply adding new instance variables. For instance, slope and fuelMoisture are currently unused, but "layers" can be added to the model with variables here and functionality in the *CASpace* class.

space.cc, `space.h`

While the *CACell* class keeps information about individual cells, the *CASpace* class, contained within `space.cc` and `space.h`, keeps information about the cellular automata itself. In practical terms, this means that *CASpace* runs the simulation. There are three main methods used to run the simulation. initializeSpace() creates the space, represented by a vector of vector of *CACell*'s. It also reads the fuel densities and firebreaks from file. startFire() reads the start locations from file and adds these locations to a list of currently burning cells. go() begins the burn cycle.

The idea behind the burn cycle is to spread the fire. Each cell that is currently burning is kept on a burn list. I iterate through this list. For each cell on this list, I try to spread the fire to it's neighbor cells. If the neighbor cell can burn, it gets added to the end of the burn list. After I've checked all of the cells neighbors, I remove him from the burn

list. When the burn list is empty, I have spread the fire to every possible cell and I can exit. A cell can burn if it's fuel density is greater than Threshold 1. If the stochastic flag is selected, a random number will be generated and checked against to determine if the cell will burn. A cell will burn for two time steps if the multiple flag is selected and it's

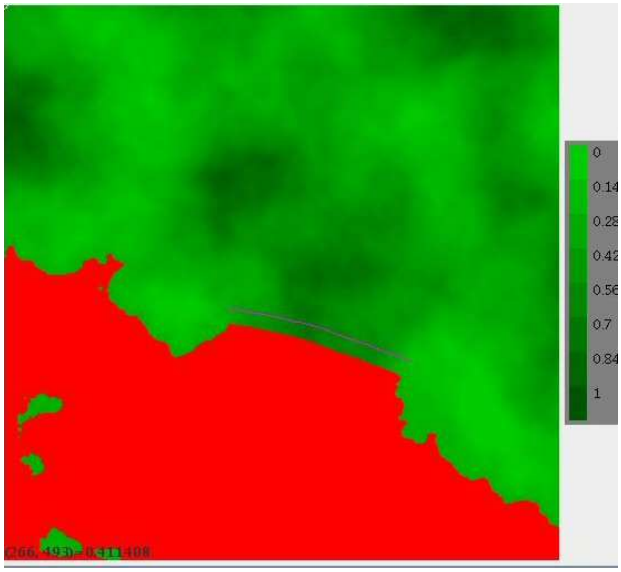


Figure 5 Landscape with burned cells

fuel density is greater than Threshold 2. In this instance, the cell will stay on the burning list until after time step 2. At the end of the burn cycle, the total number of burned cells is calculated and the final burned cells are written to a file to be used by the GUI for displaying the image.

CASpace also contains some methods to run the optimization routine, but to explain how the optimization routine works, we need to take a step back and look at main.cc, the driver for the simulation.

main.cc

`main.cc` controls the simulation. It first reads and parses the `init.dat` file (again, thanks to Dr. Banks for the parsing routine), sets the global variables and begins the simulation by calling initializeSpace() and startFire(). If the optimization flag is not set, it proceeds by calling go(). However if the optimization flag is set, it takes a different path. main.cc contains a routine called doOpt(), whose first step is to run the simulation to completion with a firebreak of length=1 and width=1, with origin at the location the user selected. This can be accomplished by calling the go() routine and retaining the total number of cells burned. Once this is accomplished, we loop through all of the user selected orientations. Inside this loop, we loop through the various lengths and widths. To avoid having to run the simulation to completion each time, we can take advantage of the fact that by starting with the smallest possible firebreak, and by increasing it

incrementally, we can essentially reverse the fire. We can do this by keeping track of each cell that a fire spread to. For instance, if cell[42] was able to spread to cell[41] and cell[43] but to no other cells, cell[42] would keep a list: {41, 43}. We also need to keep track of which cells are affected by a firebreak. For this implementation, the firebreak will affect it's immediate neighbors, or those that are at a distance of 1 (see Formula 1). For each lengthening cycle, we recalculate the affected cells. If one of the newly affected cells had burned previously, it is no longer burned because the lengthened firebreak has affected it, so that cell is put on the affected list and the total number of cells burned is decreased by 1. We also have to check all of the newly affected cells. We do this by iterating through their toList. If one of the cells on their toList was burning, it is newly affected, no longer burned, and the total number of cells burned decreases by 1. We continue in this manner until we have reversed the fire and we store the total number of burned cells, the length and width of the firebreak, and the orientation. See Figure 6for an example. We can now determine what the most effective firebreak would be. This data is written to a text file that is read by the GUI and the best break is displayed on the image.

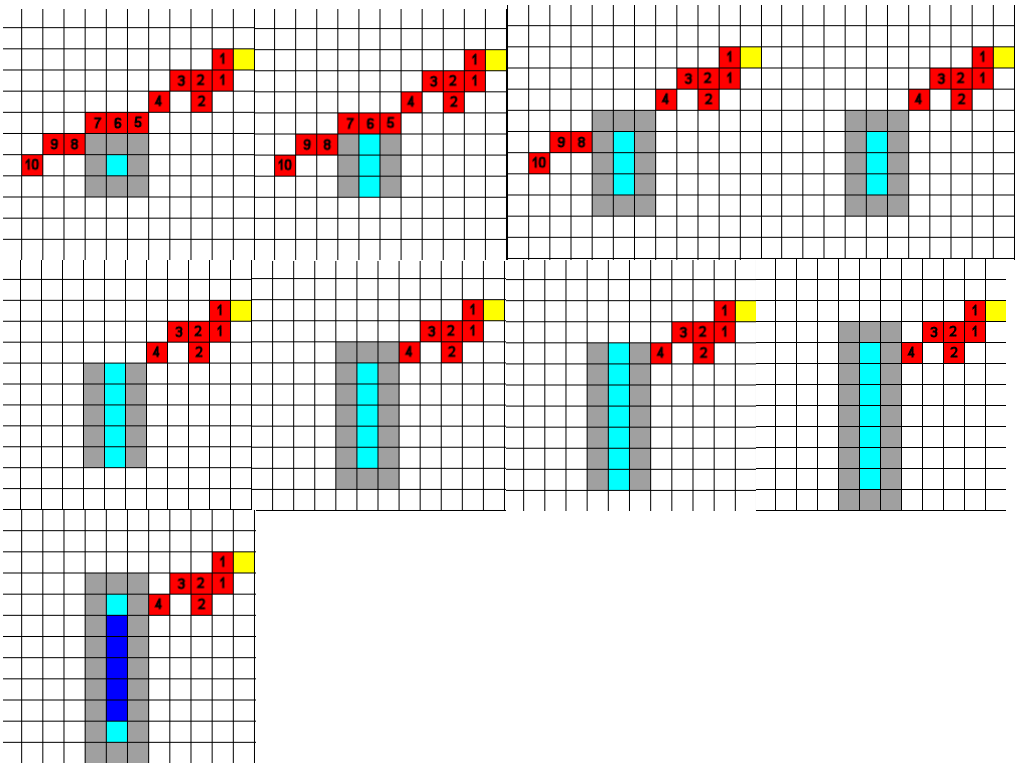


Figure 6 Sequence demonstrating fire reversal algorithm

Makefile and Performance

The Makefile used is fairly simple. It uses some ideas from Dr. Banks to automatically create pdf documents from the source code. When compiled without any optimization flags, the code runs fairly slowly. For a 500 x 500 grid, it takes approximately two minutes for a complete burn, depending, of course, which machine is running the code and what your connection speed is. When compiled with the -O3 flag and while running on a Cetus machine, execution time is less than 30 seconds. On

Franken, it takes a little longer. The speed of the GUI is also dependent on the machine and connection. While running on my laptop over a cable modem, the GUI can be annoyingly slow, not registering all of the mouse locations when selecting a firebreak, but when running on a lab machine, it runs extremely smoothly.

All source code has been tested, compiled and run on Linux boxes. Parts of this code have been converted to run under Cygwin on a Windows machine, so it is possible, however, due to the many problems I encountered doing this I don't recommend it. The source code is located in a CVS repository at /research/bains/fire. A README file is included to help with compilation and running.

Future Work

There are many things that could be done to improve and embellish this code. I designed this code in C++ mainly so it would be easily extensible. It is a fairly simple task to add components to the Java GUI, and as I discussed earlier it is not difficult at all to add "layers" to the CA. For instance, the next layer I would consider adding would be fuel moisture. To do this, you would add a instance variable to the *CACell*, fuelMoisture. You would add two methods: one to set the variable and one to get the variable. Then you would need to consider how the fuel moisture would affect how the cell burns and add the corresponding code to *CASpace*. Examples of other "layers" that could be added include wind, slope, fire temperature, and fuel moisture.

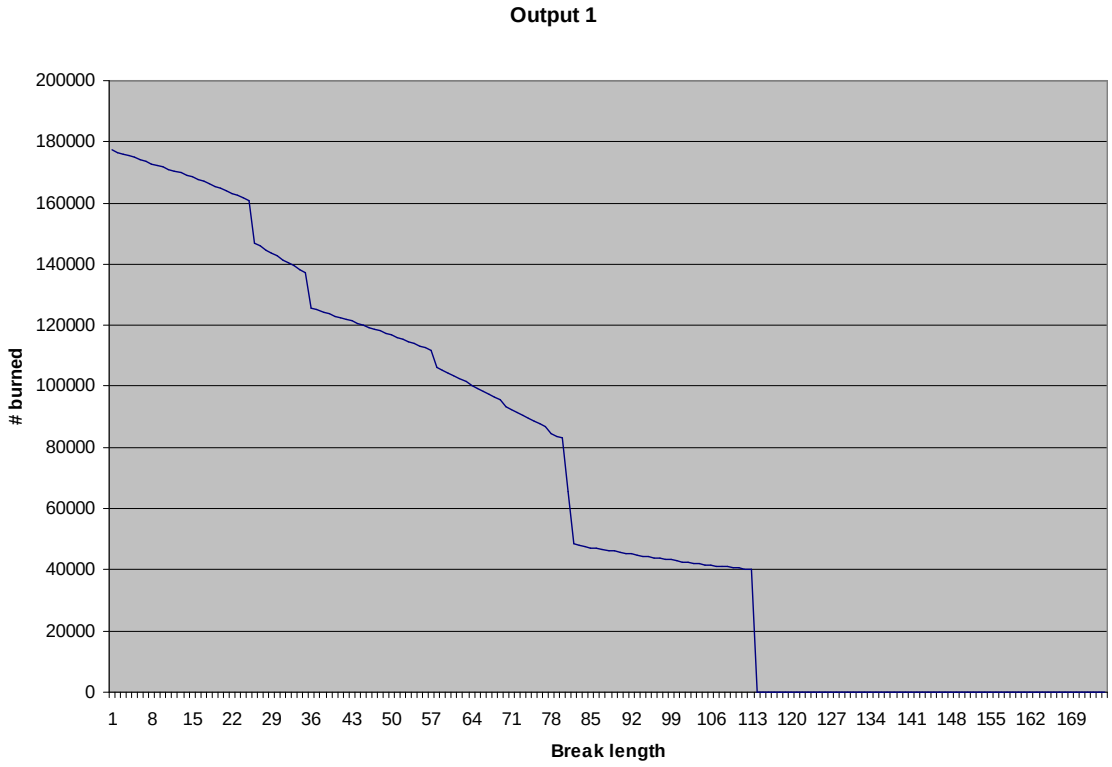
Some issues that I was unable to resolve include seeing the firebreak as you draw it, showing the actual width of the firebreak on the image, adding other control methods, adding more orientations, and adding "real" optimization. I think it would also be nice to load an actual landscape or fuel density file from an ESRI product, burn the fire, and then be able to view the output in the GIS. This can easily be accomplished with some sample code furnished by Eric Carr.

- Extend control techniques
    - Chemical control
    - Helicopter/dumping water
- Extend model parameters
    - Wind
    - Slope
    - Fuel moisture
    - Fire temperature
- GA or Linear programming based optimization

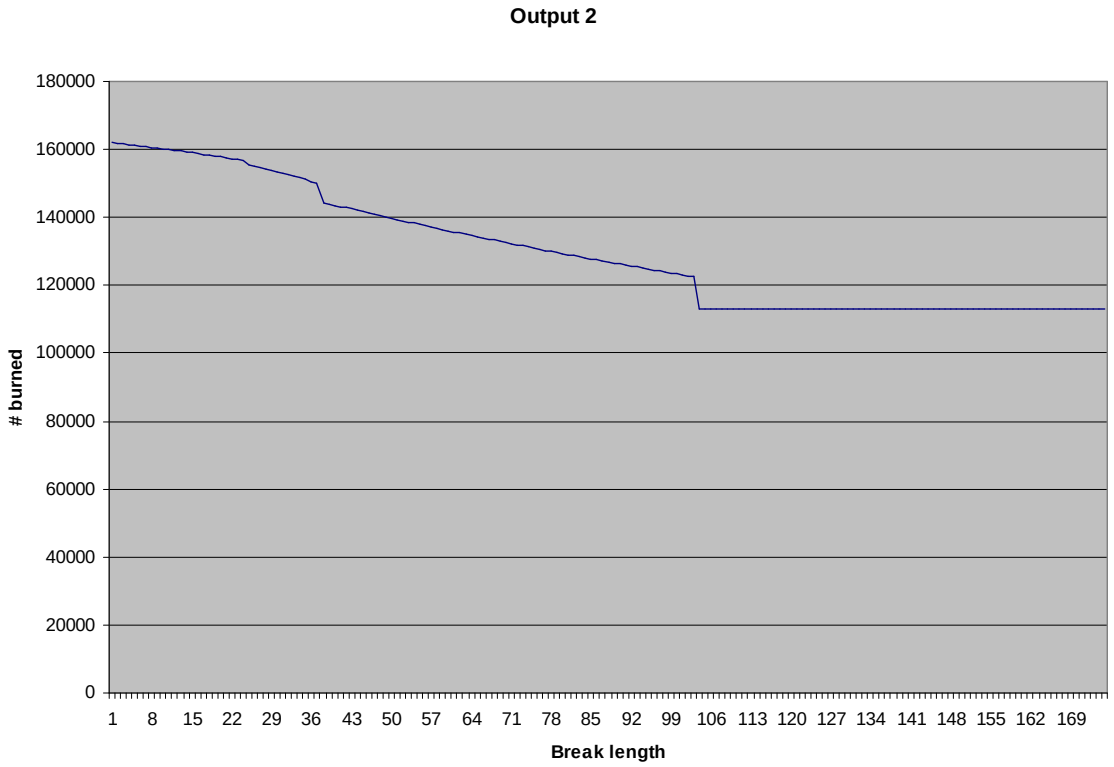
- Implement model with OpenInventor
- Implement reading ASCII grid files

Output

The first sample output was run on landscape_003 with a 90 degree orientation and searching 70% of the map space. Notice that there is a sharp drop off at approximately length 120 and by length 225 there are no more cells burning.



The second sample was run on landscape_004 with an orientation of 45 degrees and searching 75% of the map space. The number of burned cells decreases linearly until it levels out at length 207.



Directory Structure

It may be helpful to have a description of the directory structure of this project. The main directory is named `fire`. There are several subdirectories which are important. All of the image files are located in the `image` directory. Source code documentation is located in `doc`. The `data` directory contains initialization files for the C++ code and raw data files. The `include` directory contains all of the C++ header files and the `lib` directory contains library files for `libnoise`, third-party software that is used to create new landscapes. The `tools` directory contains a file to create automatic source code documentation. Lastly, the `landscape` directory contains the C++ source code that is used to create new landscape files. The `generateMap` executable uses an initialization file, and it is also located in the `landscape` directory. Both the `landscape` directory and the `fire` directory have separate makefiles. One other file to note is `setup.csh`, located in the `fire` directory. This file, when sourced, sets environment variables that are needed for the project to compile and run. The main `fire` directory also houses all of the java source and class files and the C++ source code files for the model.

The files located in the data directory may require some clarification. The GUI portion of this project is written in Java, but the model backbone is written in C++. There does not exist a clean method of passing parameters from a Java executable to a C++ executable, therefore I use several data/text files to allow the passing of parameters. `init.dat` is an initialization file that contains several parameters that are entered by the

user. The main C++ routine contains a function that parses this file and sets global variables for use by the model. (This parsing routine was borrowed from Dr. David Banks.) `whichImage.dat` and `whichLandscape.dat` are files that are used by the GUI code to keep track of the landscape data and image filenames. `start.dat` and `firebreak.dat` contain the (x, y) coordinates of the user selected fire starts and firebreaks respectively. These are used in both the Java and the C++ code. `final.dat` is written by the C++ code after the model has finished and contains the (x, y) coordinates of the burned cells. This is used by the GUI to draw the burned cells on the image. Similarly, `iteration.dat` contains an array of time steps which are used by the GUI to display an animation of the fire. The data directory also houses the landscape data files, usually named `landscape_#.dat`, but can be called anything. Each landscape data file has a matching file in the image directory. The data files contain an array of fuel densities. Finally, the `landscape` directory contains `*.opt` files, which are the output files from the optimization routine. They list the total number of cells burned, the length and width, and the orientation for each run of the optimization routine. These files are written by the C++ code and used by the GUI code to output the best firebreak on the landscape image.

Acknowledgements

I would like to thank the National Science Foundation for funding under NSF Award No. IIS-0427471. I would like to thank my committee members, Dr. Michael Berry, Dr. Louis Gross and Dr. David Banks. I would also like to thank TIEM members, Dr. Shih-Lung Shaw, Dr. Dali Wang, Eric Carr, Jane Comiskey, Nick Buchanan and Ling Yin.

References

Rothermel, R.C., 1972. A mathematical model for predicting fire spread in wildland fuels. USDA For. Serv., Intermt For. and Range Exp. Stn, Ogden, UT. Res. Pap. INT-1 15: pp. 40.

Packages

libnoise

http://libnoise.sourceforge.net

Java Advanced Imaging

http://java.sun.com/products/java-media/jai/

Web References

Perlin Noise
http://www.cs.cmu.edu/~mzucker/code/perlin-noise-math-faq.html

Java API Specification, 1.5.0
http://java.sun.com/j2se/1.5.0/docs/api/