Simulating Forest Fires with Cellular Automata

Team 27

Hanwen Xu, Will Epperson, Jiayi Ye

https://github.gatech.edu/hxu317/ComputerSimProject1

(The Jupyter Notebooks in the GitHub repo are much cleaner/easier to look at)

Tutorial 1: Simulating the spreading of forest fires using cellular automata

Introduction to Forest Fires

For this tutorial, we will be modeling forest fires. Annually, forest fires consume 4 to 5 million acres of land in the United States rending this land and anything else in the fire's path barren and charred. By attempting to model forest fires, we hope to be able to greater understand the factors that contribute to the spreading of forest fires in order to help people make more educated decisions about stopping them and inform preventative policy measures that can be enacted to curtail these fires.

Introduction to Cellular Automata

Cellular automata models are a type of model where each cell represents a discrete unit. In our case, each unit will be one cell of a forest and is one of 4 states: empty, unburned vegetation, burning vegetation, or extinguished. Cells that border those on fire naturally have a higher likelihood of igniting in the next time steps. In the beginning we will implement a very basic cellular automata, where the likelihood of catching fire is only determined by whether or not the cells adjacent to it are on fire. This model will change as we add more and more components to our function that includes other characteristics of a forest such as wind speed, elevation, and type of tree.

Code

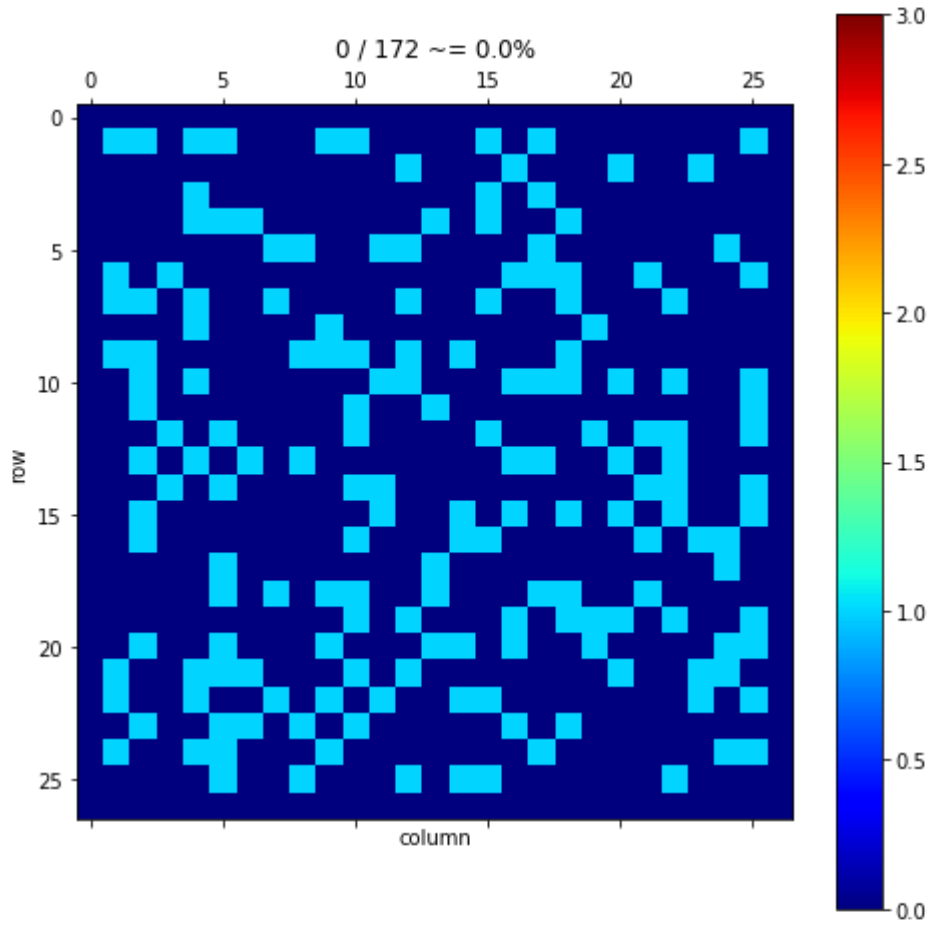
First lets import what we need, and also define some state variables that represent possible initial states for each cell in our forest!

In [1]:
```
import numpy as np
import scipy as sp
import scipy.sparse
import matplotlib as mpl
import matplotlib.pyplot as plt

empty = 0
unburned = 1
burning = 2
burned = 3
```

Then, lets write a few methods that allow us to get the basic parts of the simulation up and running:

- `create_world(n, d)` : Returns a new grid world, where each index will hold a numerical value indicating what is in that cell. The new world will have size `(n+2)` by `(n+2)` due to the one cell padding added on all four size for simpler matrix computations. These boundary cells will all be initialized to be "empty". The probability that any given cell will have vegetation that can catch fire is expressed through `d` .
- `show_world(W)` : Displays the world in a colorful way! Each different possible state a cell can be in is expressed as a different color. The different states are: "Empty" (0), "Unburned" (1), "Burning" (2), "Burned" (3)

In [2]:
```
def create_world(n, q):
    world = np.zeros((n+2, n+2))
    forest = world[1:-1, 1:-1]
    forest[:, :] = np.random.choice([0, 1], p=[1-q, q], size=(n, n))
    return world

def show_world(W, title=None, **args):
    if 'cmap' not in args:
        args['cmap'] = 'jet'
    if 'vmin' not in args and 'vmax' not in args:
        args['vmin'] = 0
        args['vmax'] = 3
    plt.figure(figsize=(8, 8))
    plt.matshow(W, fignum=1, **args)
    plt.xlabel('column')
    plt.ylabel('row')
    plt.colorbar()
    if title is None:
        num_trees = (W > 0).sum()
        if num_trees > 0:
            num_burnt = (W == 3).sum()
            percent = num_burnt / num_trees * 1e2
            title = '{} / {} ~= {:.1f}%'.format(num_burnt, num_trees, pe
rcent)
        else:
            title = ''
    plt.title(title)
    pass

# Demo:
# Uncomment this next line for a 3x3 fully covered unburned world
# World = create_world(3, 1)
World = create_world(25, 0.25)
show_world(World)
```



Now let's define some helper functions that are going to help us in our journey to model a forest fire!

Note that all functions following the `is_<something>` functions will return a matrix of equal size to the given world in variable `W`!

In [3]:
```
def is_empty(W):
    return W == empty

# This tells us if something was vegetation at any point, burned or unbu
rned!
def is_vegetation(W):
    return W > empty

def is_unburned(W):
    return W == 1

def is_burning(W):
    return W == 2

def is_burned(W):
    return W == 3

def count(W, cond_fun):
    return cond_fun(W).sum()

def summarize_world(W):
    def suffix(n):
        return (1, "tree") if n == 1 else (n, "trees")
    m, n = W.shape[0]-2, W.shape[1]-2
    n_trees = count(W, is_vegetation)
    n_unburned = count(W, is_unburned)
    n_burning = count(W, is_burning)
    n_burned = count(W, is_burned)

    print("The world has dimensions: {} x {}".format(m, n))
    print("There are {} cell(s) that have had vegetation in them".format
(n_trees))
    print("There are {} cell(s) of vegetation that are unburned".format(
n_unburned))
    print("There are {} cell(s) of vegetation on fire".format(n_burning
))
    print("There are {} cell(s) of vegetation completely burned".format(
n_burned))

summarize_world(World)
```

```
The world has dimensions: 25 x 25
There are 172 cell(s) that have had vegetation in them
There are 172 cell(s) of vegetation that are unburned
There are 0 cell(s) of vegetation on fire
There are 0 cell(s) of vegetation completely burned
```

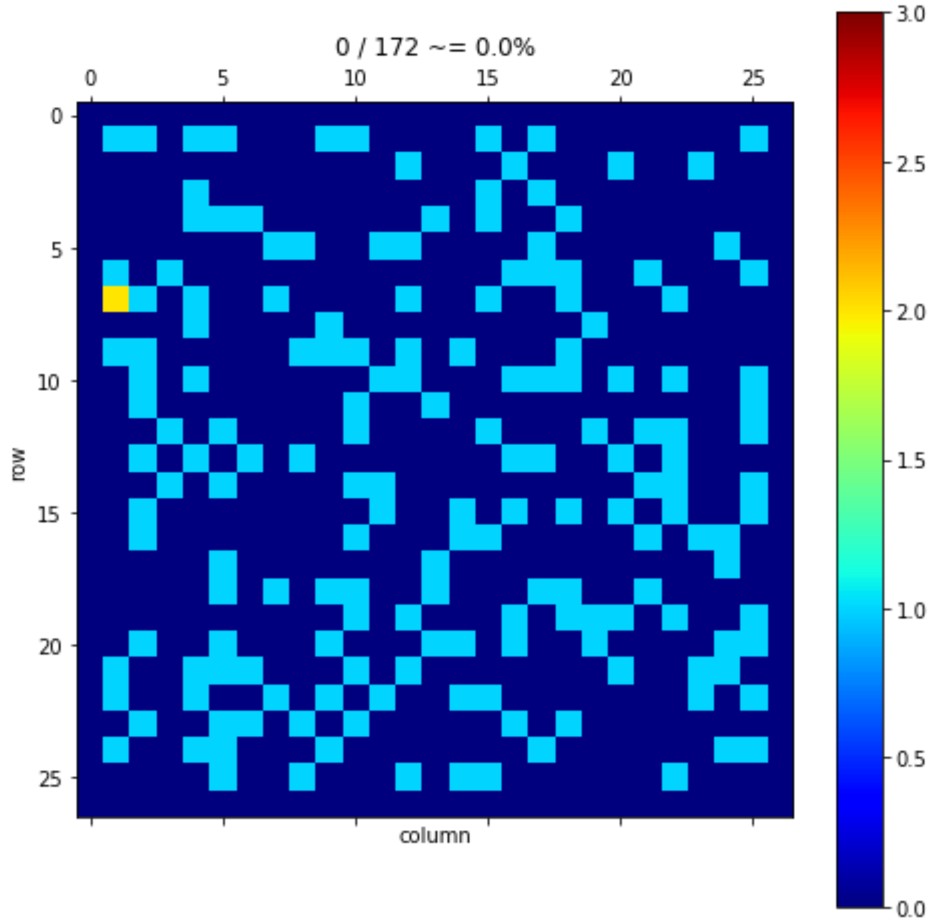
Now we're going to need a way to start our fire! Normally these massive fires start with one or a few spots catching on fire because of human error. To simulate that, let's pick one of the Unburned locations at random and start a fire there!

We'll call our function `start_fire(W)` , and it'll do exactly that. Transition one of our unburned locations to on fire!

The comments also show a way of setting specific cells on fire, if you wanted to start a fire in the same places every time!

In [4]:
```
def start_fire(W, cells=None):

    W_new = W.copy()

    if cells == None:
        F = W[1:-1, 1:-1]
        W_new = W.copy()
        F_new = W_new[1:-1, 1:-1]
        I, J = np.where(is_unburned(F)) # Positions of all trees
        if len(I) > 0:
            k = np.random.choice(range(len(I))) # Index of tree to ignit
e
            i, j = I[k], J[k]
            assert F_new[i, j] == 1, "Attempting to ignite a non-tree?"
            F_new[i, j] += 1
    else:
        W_new = W.copy()

        for x,y in cells:
            W_new[x,y] = 2

    return W_new

# Start a fire somewhere random in the world!
World_next = start_fire(World)
# Uncomment this following line if you want to set custom locations on f
ire!
# World_next = start_fire(World, [(2,2), (2,3), (3,2)])
show_world(World_next)
summarize_world(World_next)
```

```
The world has dimensions: 25 x 25
There are 172 cell(s) that have had vegetation in them
There are 171 cell(s) of vegetation that are unburned
There are 1 cell(s) of vegetation on fire
There are 0 cell(s) of vegetation completely burned
```



Deterministic Spread Fire

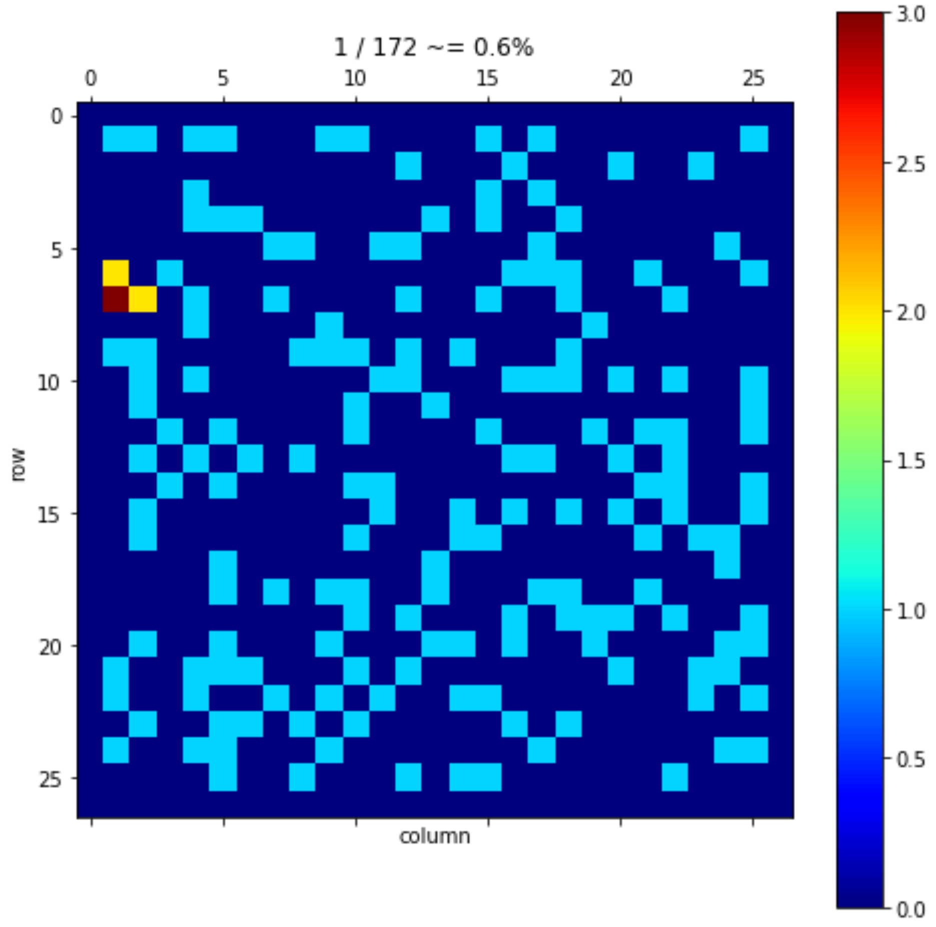
Now that we've successfully started a fire, lets spread it. Here we will implement a function called `spread_fire(W)` that will simulate one time step. How the fire actually spreads in real life will be more complicated then this example, but for now, let's just make it so that whenever the neighbors are on fire, that cell will definitely catch fire! Lets not forget about the tiles that are already on fire, they need to be put out!

Don't worry, we'll implement more complex cases later on.

This function works by taking the locations of all trees (in `Vegetation`) and then checking if any of the 8 cells bordering that tree are on fire. Since this basic model is doing deterministic transitions, if any of the bordering cells are on fire then the center cell will also light on fire.

In [5]:
```
def spread_fire(W):
    W_new = W.copy()
    Vegetation = is_unburned(W)
    Fires = is_burning(W)

    # Spread fire to neighbors
    W_new[1:-1, 1:-1] += Vegetation[1:-1, 1:-1] \
                         & (
                              Fires[:-2, :-2]  | Fires[1:-1, :-2] | Fire
s[2:, :-2]
                            | Fires[:-2, 1:-1] |                    Fire
s[2:, 1:-1]
                            | Fires[:-2, 2:]   | Fires[1:-1, 2:]  | Fire
s[2:, 2:]
                           )

    # Extinguish current fires
    W_new[1:-1, 1:-1] += Fires[1:-1, 1:-1]
    return W_new

# Spread one time-step of fire. Then show and summarize the world!
World_next = spread_fire(World_next)
show_world(World_next)
summarize_world(World_next)
```

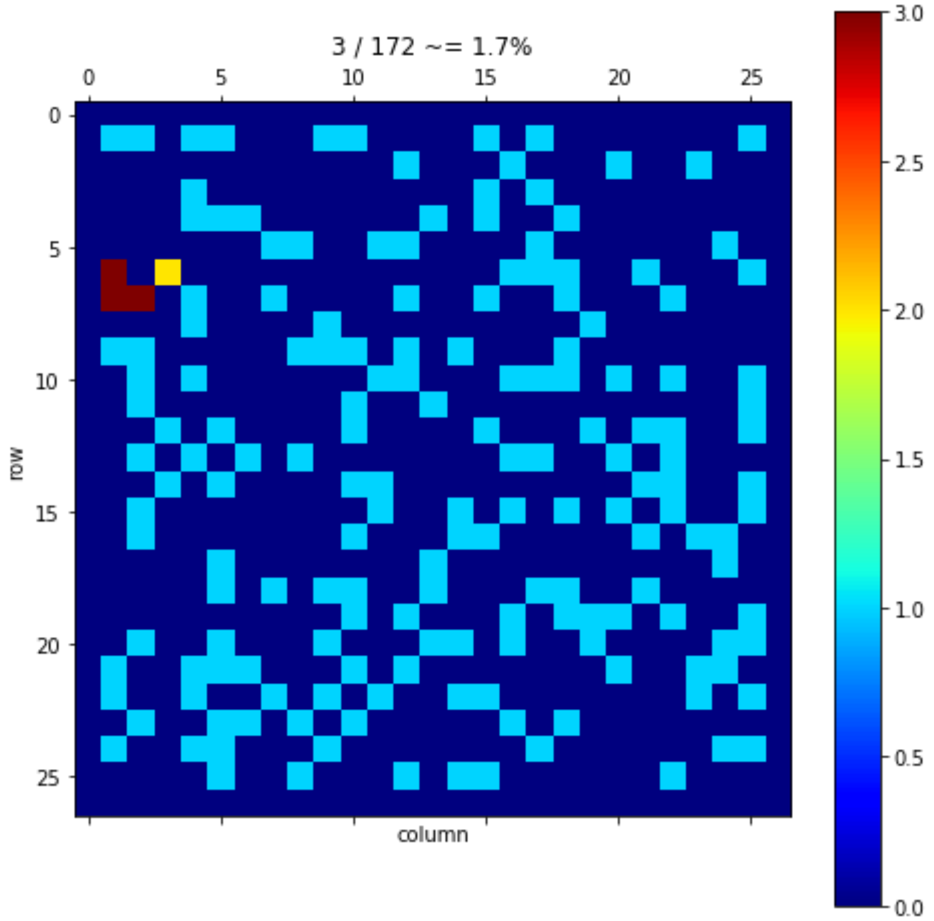
```
The world has dimensions: 25 x 25
There are 172 cell(s) that have had vegetation in them
There are 169 cell(s) of vegetation that are unburned
There are 2 cell(s) of vegetation on fire
There are 1 cell(s) of vegetation completely burned
```



Only doing one timestep is no fun, let's do another!

In [6]:
```
# Second Timestep
World_next = spread_fire(World_next)
show_world(World_next)
summarize_world(World_next)
```

```
The world has dimensions: 25 x 25
There are 172 cell(s) that have had vegetation in them
There are 168 cell(s) of vegetation that are unburned
There are 1 cell(s) of vegetation on fire
There are 3 cell(s) of vegetation completely burned
```



Probabilistic Spreading

Now you may be wondering, how would I make this model behave more like real life fires? The answer is to introduce probabilities! In real life, having a neighbor on fire isn't going to guarantee catching fire! Lets develop a system that can take this probability of catching on fire into account!

This function models the probability that a cell catches on fire as below, where we are considering the probability that cell x is on fire and the probability of fire is p_f, and there are n surrounding neighbors on fire.

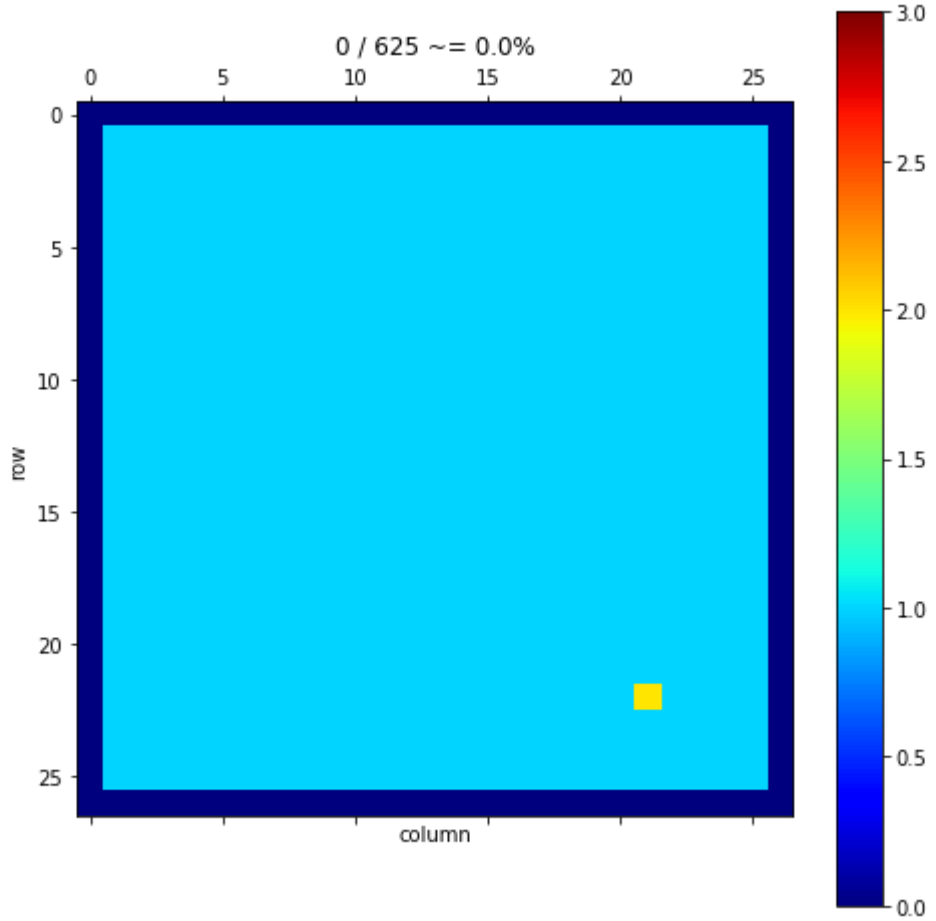
$$P(x) = 1 - (1 - p_f)^n$$

In [7]:
```
def spread_fire_p(W, fire_spread_prob=0.4):
    W_new = W.copy()
    Vegetation = is_unburned(W)
    Fires = is_burning(W)

    # Spread fire to neighbors
    # First count how many neighbor of a cell are on fire!
    num_neighbors_on_fire = (Fires[:-2, :-2].astype(int) + Fires[1:-1, :
-2] + Fires[2:, :-2]
                             + Fires[:-2, 1:-1]
+ Fires[2:, 1:-1]
                             + Fires[:-2, 2:]           + Fires[1:-1, 2
:]  + Fires[2:, 2:])

    # If there's no vegetation at a certain spot, we don't want to calcu
late anything related to it's neighbors,
    # so lets set those locations to zero
    num_neighbors_on_fire = np.multiply(num_neighbors_on_fire, Vegetatio
n[1:-1, 1:-1].astype(int))

    # Now calcualte the chance of each location doesn't catch fire at th
e next time-step.
    # All probabilities start out as 1.0 - fire_spread_prob
    fire_prob_matrix = np.ones(num_neighbors_on_fire.shape) - fire_sprea
d_prob

    # Because each neighbor independently influences a cell, the total p
robability of not spreading is equal to
    # the (probabilty of one neighbor not spreading) ^ (number of neighb
ors). Taking 1.0 - that value transforms the
    # calculated probability to represent the: probability of the cell c
atching on fire due to any of the neighbors.
    fire_prob_matrix = 1.0 - np.power(fire_prob_matrix, num_neighbors_on
_fire)

    # Again we don't care about a specific cell's value if there's no ve
getation there, so we zero out
    # the locations that don't have vegetation there!
    fire_prob_matrix = np.multiply(fire_prob_matrix, Vegetation[1:-1, 1:
-1].astype(int))

    # Now to use our probabiltiy matrix we've generated, we need to gene
rate a matrix of all random
    # values between 0.0 and 1.0, and compare if the random generated va
lue is less than the probability of catching fire!
    randys = np.random.rand(*fire_prob_matrix.shape)
    new_on_fires = randys < fire_prob_matrix

    # Now we "add" one to each location that used to be "unburned" that
 now will be "burning" since the value for
    # "unburned" is (1) and the value for "burning" is (2), this works o
ut to transition our states perfectly!
    W_new[1:-1, 1:-1] += new_on_fires

    # We still want to extinguish current fires after one time-step, so
 let's go do that
```

```
    W_new[1:-1, 1:-1] += Fires[1:-1, 1:-1]

    # Aaaaaand we're done!
    return W_new
```

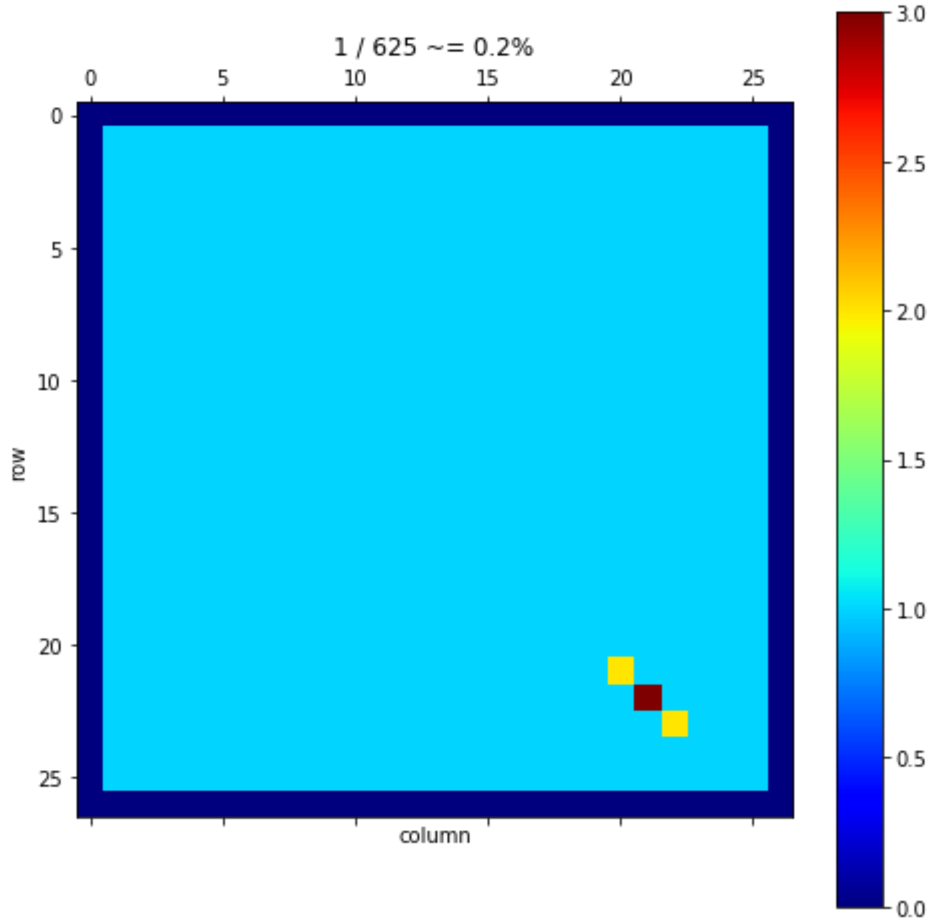
To demonstrate this probabilistic fire spreading, we'll generate a world entirely of vegetation to see how fires do not spread in a uniform circle around the start.

In [8]:
```
world_2 = create_world(25, 1)
world_2 = start_fire(world_2)

show_world(world_2)
```



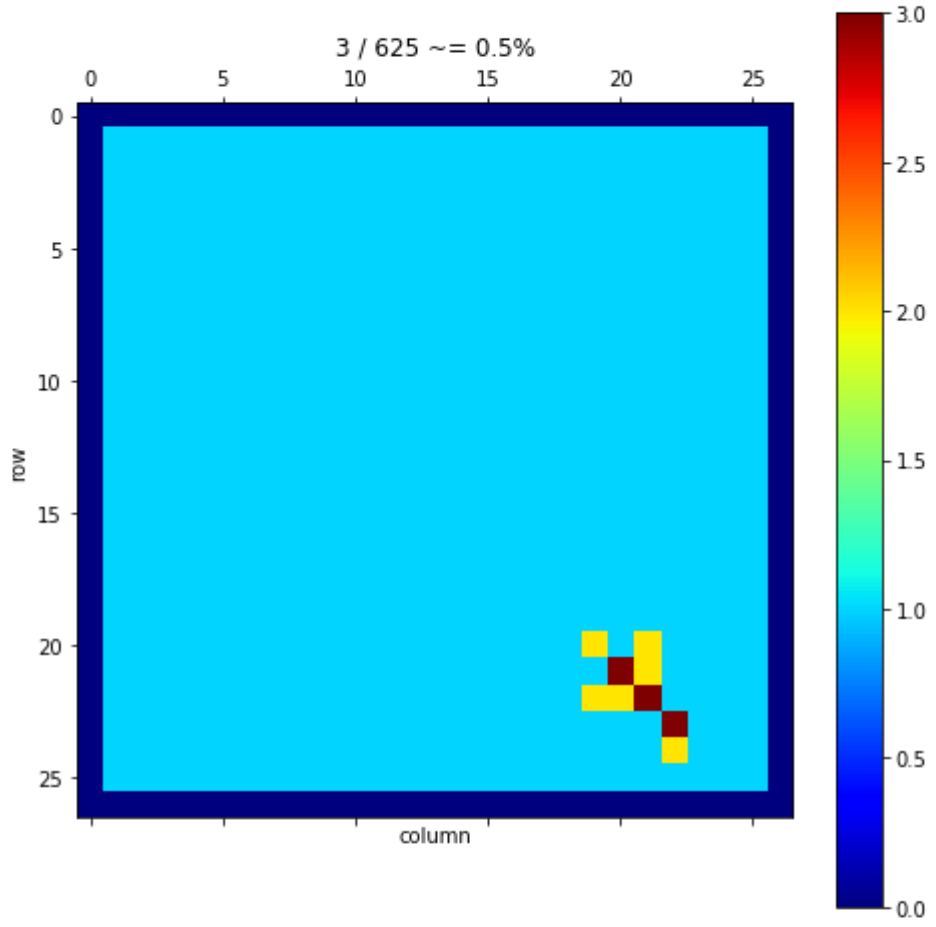
In [9]:
```
world_2 = spread_fire_p(world_2, fire_spread_prob=.5)
show_world(world_2)
summarize_world(world_2)
```

```
The world has dimensions: 25 x 25
There are 625 cell(s) that have had vegetation in them
There are 622 cell(s) of vegetation that are unburned
There are 2 cell(s) of vegetation on fire
There are 1 cell(s) of vegetation completely burned
```



In [10]:
```python
# One more time step

world_2 = spread_fire_p(world_2, fire_spread_prob=.5)
show_world(world_2)
summarize_world(world_2)
```

```
The world has dimensions: 25 x 25
There are 625 cell(s) that have had vegetation in them
There are 616 cell(s) of vegetation that are unburned
There are 6 cell(s) of vegetation on fire
There are 3 cell(s) of vegetation completely burned
```



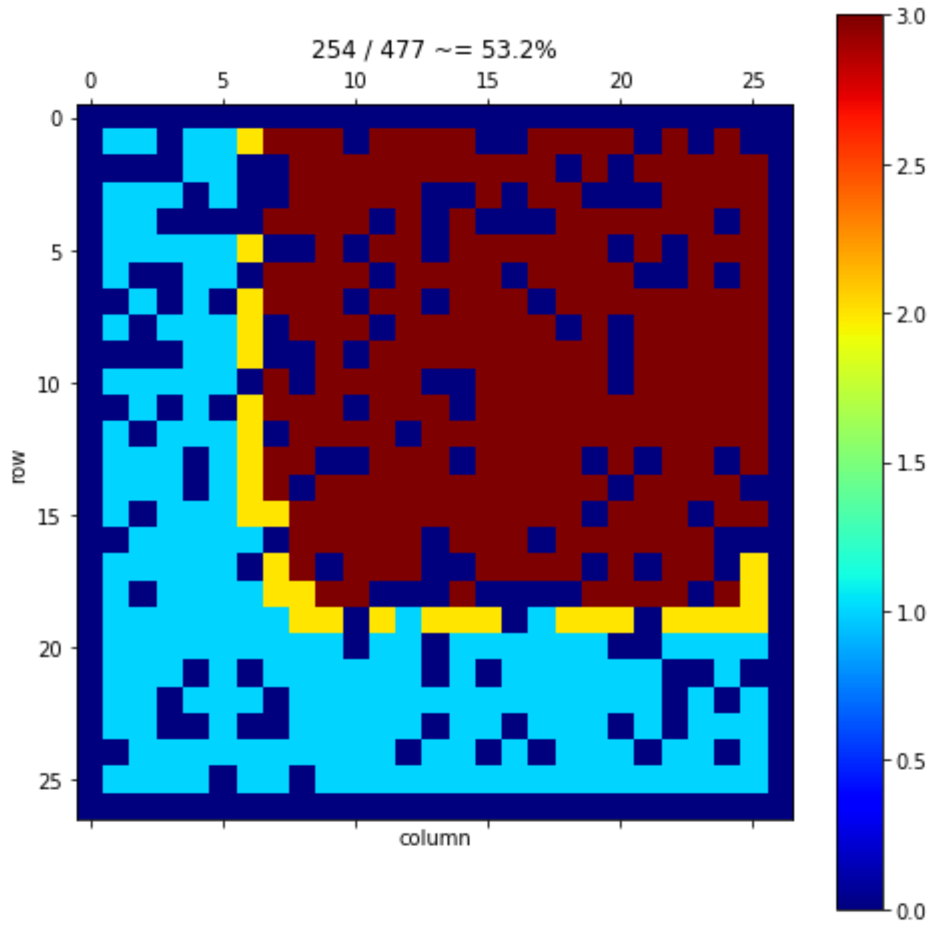
Full Simulation

Now that we've figured out how to do two timesteps of our simulation, let's try and do a full run and see how our fires spread!

We're going to write a function called `simulate(W0, t_max=None, inplace=False)` that by default runs until the world reaches a steady-state configuration!

If `inplace` is set to `False` , then the simulator will also return all the timesteps!

In [11]:
```
def simulate(W0, t_max=None, inplace=False, spread_prob=None):
    if t_max is None:
        n_max = max(W0.shape)
        t_max = n_max * ((2*n_max-1) // 2)
    W = np.zeros((W0.shape[0], W0.shape[1], 2 if inplace else t_max+1))
    t_cur = 0
    W[:, :, t_cur] = W0
    for t in range(t_max):
        t_next = (t_cur+1)%2 if inplace else t+1

        if spread_prob:
            W[:, :, t_next] = spread_fire_p(W[:, :, t_cur], fire_spread_
prob=spread_prob)
        else:
            W[:, :, t_next] = spread_fire(W[:, :, t_cur])

        if (W[:, :, t_cur] == W[:, :, t_next]).all():
            t_cur = t_next
            break
        t_cur = t_next
    return (W[:, :, t_cur], t) if inplace else W[:, :, :t_cur+1]

def viz(W, t=0):
    show_world(W[:, :, t])
    plt.show()
    print("At time {} (max={})...".format(t, W.shape[2]-1))
    summarize_world(W[:, :, t])

def run_simulation(n, q, **args):
    return simulate(start_fire(create_world(n, q)), **args)

W = run_simulation(25, 0.75)
viz(W, W.shape[2] // 2)
```



```
At time 10 (max=20)...
The world has dimensions: 25 x 25
There are 477 cell(s) that have had vegetation in them
There are 194 cell(s) of vegetation that are unburned
There are 29 cell(s) of vegetation on fire
There are 254 cell(s) of vegetation completely burned
```

Now you may be wondering, "I thought setting `inplace` to `False` returned all of the timesteps, not just one of them" and you would be right! Here's a funky little widget that lets you look at every timestep.

In [23]:
```
from ipywidgets import interact

def iviz(t=0):
    viz(W, t);
```

We can also simulate our probabilistic model as below.

In [13]:
```
W = run_simulation(25, 1, spread_prob=.5)
interact(iviz, t=(0, W.shape[2]-1))
```

Out[13]:
```
<function __main__.iviz(t=0)>
```

Monte Carlo Runs

Cool! So we can now simulate a full run-through of a randomly generated forest fire, what if we want to see how one of our parameters changes the outcome? Think back to the beginning, and remember that when we generate a world, we can give it a probability that determines how likely a cell is to contain burnable Vegetation! Let's simulate what would happen if we made that probability higher, or in other words, make our world more dense!

First though, let's write the method `simulate_many(n, q, trials)` that lets us simulate many different iterations with probability `q`

In [14]:
```
def simulate_many(n, q, trials):
    percent_burned = np.zeros(trials)
    time_to_burn = np.zeros(trials)
    for trial in range(trials):
        W_last, t_last = run_simulation(n, q, inplace=True)
        n_trees = count(W_last, is_vegetation)
        n_burnt = count(W_last, is_burned)
        percent_burned[trial] = n_burnt / n_trees if n_trees > 0 else 0.
0
        time_to_burn[trial] = t_last
    return percent_burned, time_to_burn

percentages, times_to_burn = simulate_many(25, 0.25, 100)
avg_percentage, std_percentage = percentages.mean(), percentages.std()
avg_time, std_time = times_to_burn.mean(), times_to_burn.std()
print("Percentage of vegetation that burned: ~ {:.1f}% +/- {:.1f}%".form
at(1e2*avg_percentage,
                                                                      1e
2*std_percentage))
print("Time to burn: ~ {} +/- {:.1f} timesteps".format(avg_time, std_tim
e))
```

```
Percentage of vegetation that burned: ~ 4.7% +/- 4.4%
Time to burn: ~ 4.1 +/- 3.0 timesteps
```

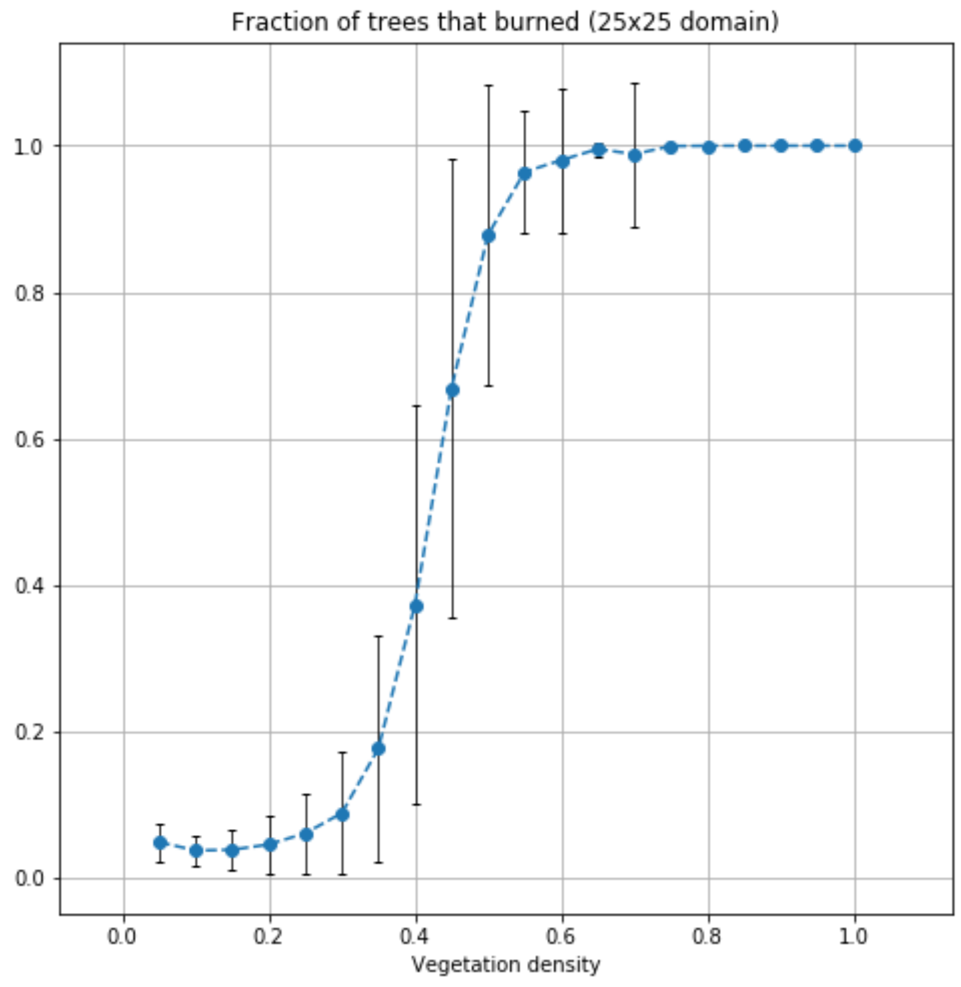
With that method out of the way, now let's simulate world with many different density probability values! Specifically, let's do 5% increments starting from 5% and working our way up to 100%! Oh and also let's make a pretty graph.

In [15]:
```
n_many = 25
Q = np.linspace(0, 1, 21)[1:]
Percentages = np.zeros((len(Q), 2))
Times = np.zeros((len(Q), 2))
for k, q in enumerate(Q):
    print("Simulating vegetation density q={}...".format(q))
    percentages, times = simulate_many(n_many, q, 100)
    Percentages[k, :] = [percentages.mean(), percentages.std()]
    Times[k, :] = [times.mean(), times.std()]
```

```
Simulating vegetation density q=0.05...
Simulating vegetation density q=0.1...
Simulating vegetation density q=0.15000000000000002...
Simulating vegetation density q=0.2...
Simulating vegetation density q=0.25...
Simulating vegetation density q=0.30000000000000004...
Simulating vegetation density q=0.35000000000000003...
Simulating vegetation density q=0.4...
Simulating vegetation density q=0.45...
Simulating vegetation density q=0.5...
Simulating vegetation density q=0.55...
Simulating vegetation density q=0.6000000000000001...
Simulating vegetation density q=0.65...
Simulating vegetation density q=0.7000000000000001...
Simulating vegetation density q=0.75...
Simulating vegetation density q=0.8...
Simulating vegetation density q=0.8500000000000001...
Simulating vegetation density q=0.9...
Simulating vegetation density q=0.9500000000000001...
Simulating vegetation density q=1.0...
```

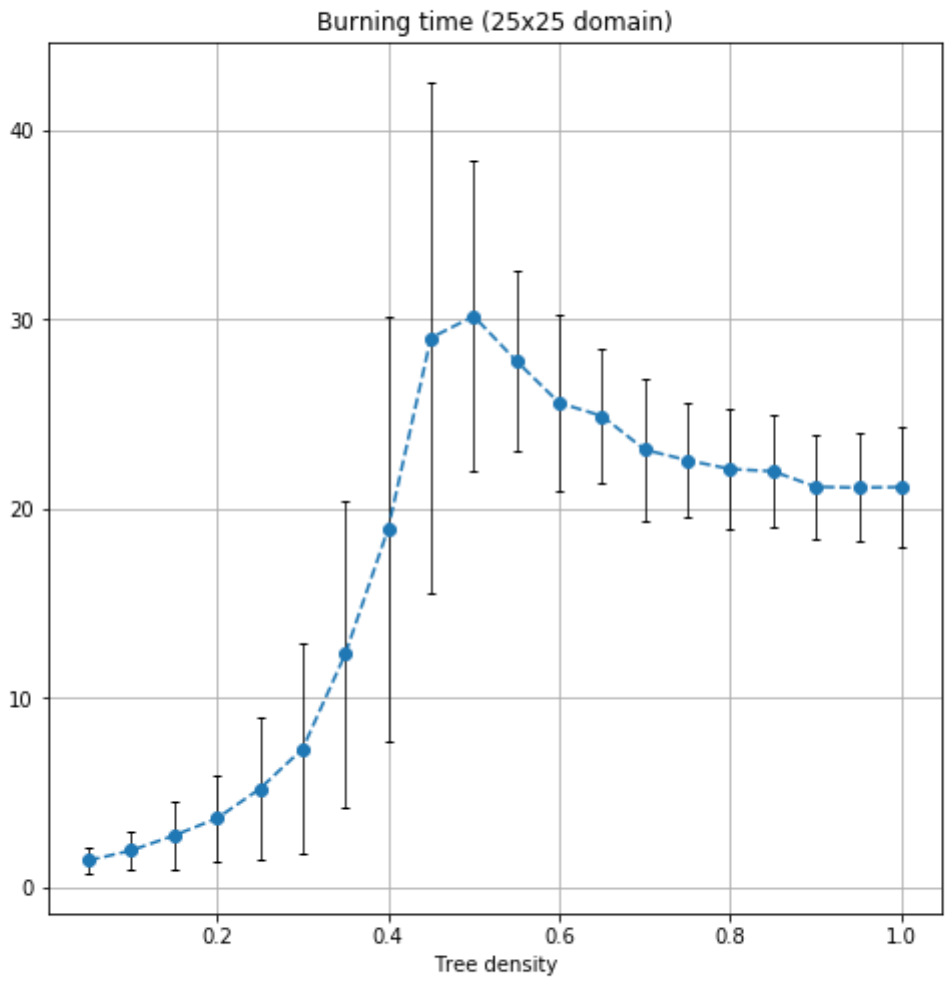
This plot is for the fraction of trees that burned given a certain vegetation density probability.

In [16]:
```
plt.figure(figsize=(8, 8))
plt.errorbar(Q, Percentages[:, 0], yerr=Percentages[:, 1], fmt='o--', ec
olor='black', elinewidth=0.75, capsize=2);
plt.gca().axis('equal')
plt.xlabel('Vegetation density');
plt.title('Fraction of trees that burned ({}x{} domain)'.format(n_many,
n_many));
plt.grid()
```



This second graph will be for time needed to reach the steady-state configuration! In other words, how quickly does our fire either go out/eradicate the entire forest.

In [17]:
```
plt.figure(figsize=(8, 8))
Z = 1 # Q*n_many*n_many # Normalizing factor
plt.errorbar(Q, Times[:, 0]/Z, yerr=Times[:, 1]/Z,
             fmt='o--', ecolor='black', elinewidth=0.75, capsize=2);
#plt.gca().axis('equal')
plt.xlabel('Tree density');
plt.title('Burning time ({}x{} domain)'.format(n_many, n_many));
plt.grid()
```



We can also examine how different spread probabilities effect the fraction of trees burned.

In [18]:
```
def simulate_many_p(n, q, trials, spread_prob):
    percent_burned = np.zeros(trials)
    time_to_burn = np.zeros(trials)
    for trial in range(trials):
        W_last, t_last = run_simulation(n, q, inplace=True, spread_prob=
spread_prob)
        n_trees = count(W_last, is_vegetation)
        n_burnt = count(W_last, is_burned)
        percent_burned[trial] = n_burnt / n_trees if n_trees > 0 else 0.
0
        time_to_burn[trial] = t_last
    return percent_burned, time_to_burn

percentages, times_to_burn = simulate_many_p(25, 0.25, 100, .5)
avg_percentage, std_percentage = percentages.mean(), percentages.std()
avg_time, std_time = times_to_burn.mean(), times_to_burn.std()
print("Percentage of vegetation that burned: ~ {:.1f}% +/- {:.1f}%".form
at(1e2*avg_percentage,
                                                                        1e
2*std_percentage))
print("Time to burn: ~ {} +/- {:.1f} timesteps".format(avg_time, std_tim
e))
```

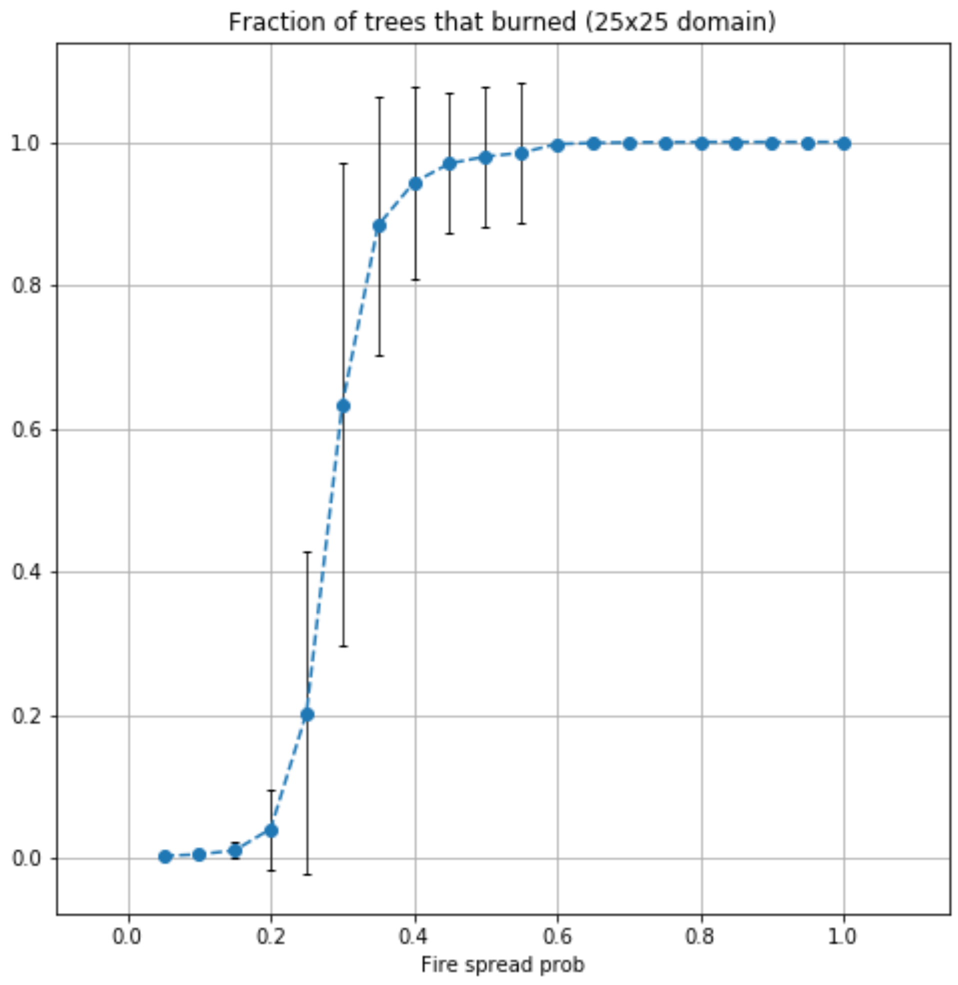
```
Percentage of vegetation that burned: ~ 2.3% +/- 1.8%
Time to burn: ~ 2.75 +/- 1.7 timesteps
```

Here, we'll examine the effect of fire spreading probabilities from 0.5 to 1 for a map of density 1, meaning that it is full of trees.

In [19]:
```
P = np.linspace(0, 1, 21)[1:]
Percentages_2 = np.zeros((len(P), 2))
Times_2 = np.zeros((len(P), 2))
for k, p in enumerate(P):
    print("Simulating fire spread prob p={}...".format(p))
    percentages, times = simulate_many_p(25, 1, 100, p)
    Percentages_2[k, :] = [percentages.mean(), percentages.std()]
    Times_2[k, :] = [times.mean(), times.std()]
```

```
Simulating fire spread prob p=0.05...
Simulating fire spread prob p=0.1...
Simulating fire spread prob p=0.15000000000000002...
Simulating fire spread prob p=0.2...
Simulating fire spread prob p=0.25...
Simulating fire spread prob p=0.30000000000000004...
Simulating fire spread prob p=0.35000000000000003...
Simulating fire spread prob p=0.4...
Simulating fire spread prob p=0.45...
Simulating fire spread prob p=0.5...
Simulating fire spread prob p=0.55...
Simulating fire spread prob p=0.6000000000000001...
Simulating fire spread prob p=0.65...
Simulating fire spread prob p=0.7000000000000001...
Simulating fire spread prob p=0.75...
Simulating fire spread prob p=0.8...
Simulating fire spread prob p=0.8500000000000001...
Simulating fire spread prob p=0.9...
Simulating fire spread prob p=0.9500000000000001...
Simulating fire spread prob p=1.0...
```

In [20]:
```
plt.figure(figsize=(8, 8))
plt.errorbar(P, Percentages_2[:, 0], yerr=Percentages_2[:, 1], fmt='o--'
, ecolor='black', elinewidth=0.75, capsize=2);
plt.gca().axis('equal')
plt.xlabel('Fire spread prob');
plt.title('Fraction of trees that burned ({}x{} domain)'.format(n_many,
n_many));
plt.grid()
```

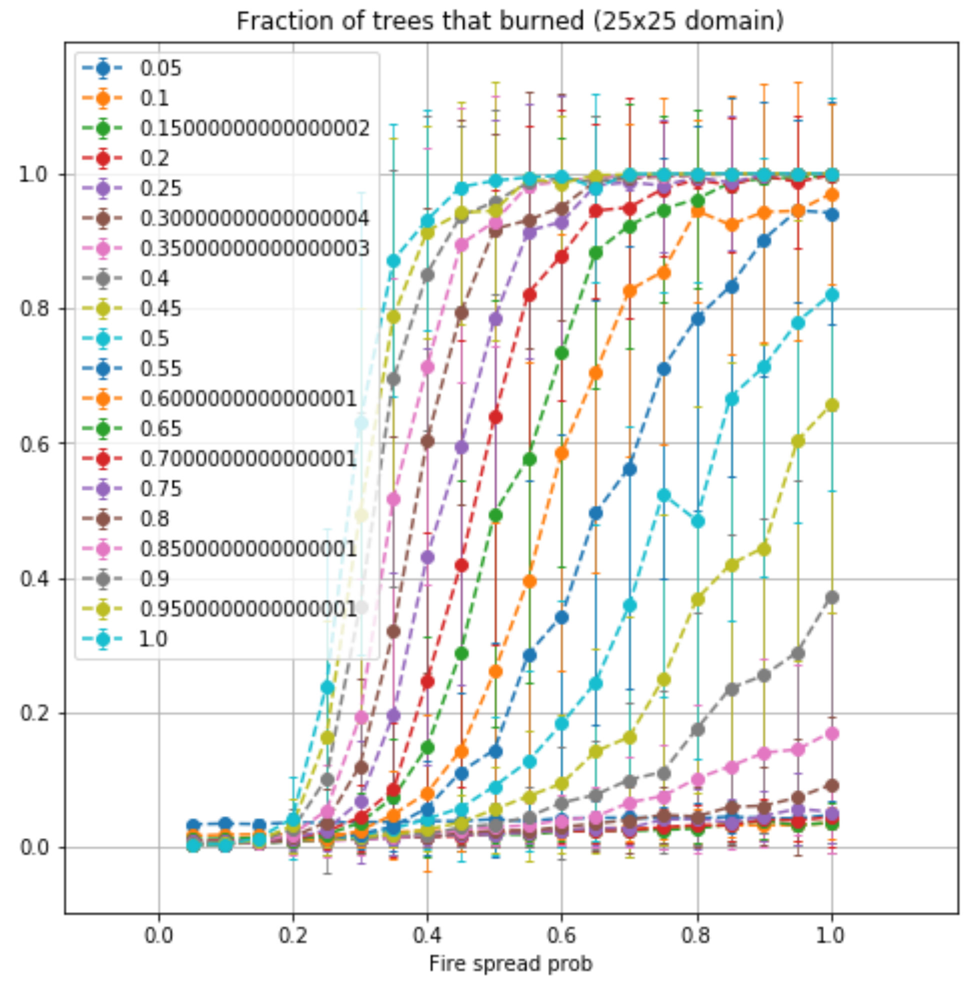


We can run more simulations to see how the combination of vegetation density and fire spread effect the fire's tree consumption

In [21]:
```
n_many = 25

Q = np.linspace(0, 1, 21)[1:]
P = np.linspace(0, 1, 21)[1:]
Percentages_3 = np.zeros((len(Q), len(P), 2))
Times_3 = np.zeros((len(Q), len(P), 2))

for k_1, q in enumerate(Q):
    for k_2, p in enumerate(P):
        print("Simulating p={}, q={}...".format(p, q))
        percentages, times = simulate_many_p(n_many, q, 100, p)
        Percentages_3[k_1, k_2, :] = [percentages.mean(), percentages.st
d()]
        Times_3[k_1, k_2, :] = [times.mean(), times.std()]
```

```
Simulating p=0.05, q=0.05...
Simulating p=0.1, q=0.05...
Simulating p=0.15000000000000002, q=0.05...
Simulating p=0.2, q=0.05...
Simulating p=0.25, q=0.05...
Simulating p=0.30000000000000004, q=0.05...
Simulating p=0.35000000000000003, q=0.05...
Simulating p=0.4, q=0.05...
Simulating p=0.45, q=0.05...
Simulating p=0.5, q=0.05...
Simulating p=0.55, q=0.05...
Simulating p=0.6000000000000001, q=0.05...
Simulating p=0.65, q=0.05...
Simulating p=0.7000000000000001, q=0.05...
Simulating p=0.75, q=0.05...
Simulating p=0.8, q=0.05...
Simulating p=0.8500000000000001, q=0.05...
Simulating p=0.9, q=0.05...
Simulating p=0.9500000000000001, q=0.05...
Simulating p=1.0, q=0.05...
Simulating p=0.05, q=0.1...
Simulating p=0.1, q=0.1...
Simulating p=0.15000000000000002, q=0.1...
Simulating p=0.2, q=0.1...
Simulating p=0.25, q=0.1...
Simulating p=0.30000000000000004, q=0.1...
Simulating p=0.35000000000000003, q=0.1...
Simulating p=0.4, q=0.1...
Simulating p=0.45, q=0.1...
Simulating p=0.5, q=0.1...
Simulating p=0.55, q=0.1...
Simulating p=0.6000000000000001, q=0.1...
Simulating p=0.65, q=0.1...
Simulating p=0.7000000000000001, q=0.1...
Simulating p=0.75, q=0.1...
Simulating p=0.8, q=0.1...
Simulating p=0.8500000000000001, q=0.1...
Simulating p=0.9, q=0.1...
Simulating p=0.9500000000000001, q=0.1...
Simulating p=1.0, q=0.1...
Simulating p=0.05, q=0.15000000000000002...
Simulating p=0.1, q=0.15000000000000002...
Simulating p=0.15000000000000002, q=0.15000000000000002...
Simulating p=0.2, q=0.15000000000000002...
Simulating p=0.25, q=0.15000000000000002...
Simulating p=0.30000000000000004, q=0.15000000000000002...
Simulating p=0.35000000000000003, q=0.15000000000000002...
Simulating p=0.4, q=0.15000000000000002...
Simulating p=0.45, q=0.15000000000000002...
Simulating p=0.5, q=0.15000000000000002...
Simulating p=0.55, q=0.15000000000000002...
Simulating p=0.6000000000000001, q=0.15000000000000002...
Simulating p=0.65, q=0.15000000000000002...
Simulating p=0.7000000000000001, q=0.15000000000000002...
Simulating p=0.75, q=0.15000000000000002...
Simulating p=0.8, q=0.15000000000000002...
Simulating p=0.8500000000000001, q=0.15000000000000002...
Simulating p=0.9, q=0.15000000000000002...
Simulating p=0.9500000000000001, q=0.15000000000000002...
Simulating p=1.0, q=0.15000000000000002...
Simulating p=0.05, q=0.2...
Simulating p=0.1, q=0.2...
Simulating p=0.15000000000000002, q=0.2...
Simulating p=0.2, q=0.2...
Simulating p=0.25, q=0.2...
Simulating p=0.30000000000000004, q=0.2...
Simulating p=0.35000000000000003, q=0.2...
Simulating p=0.4, q=0.2...
Simulating p=0.45, q=0.2...
Simulating p=0.5, q=0.2...
Simulating p=0.55, q=0.2...
Simulating p=0.6000000000000001, q=0.2...
Simulating p=0.65, q=0.2...
Simulating p=0.7000000000000001, q=0.2...
Simulating p=0.75, q=0.2...
Simulating p=0.8, q=0.2...
Simulating p=0.8500000000000001, q=0.2...
Simulating p=0.9, q=0.2...
Simulating p=0.9500000000000001, q=0.2...
Simulating p=1.0, q=0.2...
Simulating p=0.05, q=0.25...
Simulating p=0.1, q=0.25...
Simulating p=0.15000000000000002, q=0.25...
Simulating p=0.2, q=0.25...
Simulating p=0.25, q=0.25...
Simulating p=0.30000000000000004, q=0.25...
Simulating p=0.35000000000000003, q=0.25...
Simulating p=0.4, q=0.25...
Simulating p=0.45, q=0.25...
Simulating p=0.5, q=0.25...
Simulating p=0.55, q=0.25...
Simulating p=0.6000000000000001, q=0.25...
Simulating p=0.65, q=0.25...
Simulating p=0.7000000000000001, q=0.25...
Simulating p=0.75, q=0.25...
Simulating p=0.8, q=0.25...
Simulating p=0.8500000000000001, q=0.25...
Simulating p=0.9, q=0.25...
Simulating p=0.9500000000000001, q=0.25...
Simulating p=1.0, q=0.25...
Simulating p=0.05, q=0.30000000000000004...
Simulating p=0.1, q=0.30000000000000004...
Simulating p=0.15000000000000002, q=0.30000000000000004...
Simulating p=0.2, q=0.30000000000000004...
Simulating p=0.25, q=0.30000000000000004...
Simulating p=0.30000000000000004, q=0.30000000000000004...
Simulating p=0.35000000000000003, q=0.30000000000000004...
Simulating p=0.4, q=0.30000000000000004...
Simulating p=0.45, q=0.30000000000000004...
Simulating p=0.5, q=0.30000000000000004...
Simulating p=0.55, q=0.30000000000000004...
Simulating p=0.6000000000000001, q=0.30000000000000004...
Simulating p=0.65, q=0.30000000000000004...
Simulating p=0.7000000000000001, q=0.30000000000000004...
Simulating p=0.75, q=0.30000000000000004...
Simulating p=0.8, q=0.30000000000000004...
Simulating p=0.8500000000000001, q=0.30000000000000004...
Simulating p=0.9, q=0.30000000000000004...
Simulating p=0.9500000000000001, q=0.30000000000000004...
Simulating p=1.0, q=0.30000000000000004...
Simulating p=0.05, q=0.35000000000000003...
Simulating p=0.1, q=0.35000000000000003...
Simulating p=0.15000000000000002, q=0.35000000000000003...
Simulating p=0.2, q=0.35000000000000003...
Simulating p=0.25, q=0.35000000000000003...
Simulating p=0.30000000000000004, q=0.35000000000000003...
Simulating p=0.35000000000000003, q=0.35000000000000003...
Simulating p=0.4, q=0.35000000000000003...
Simulating p=0.45, q=0.35000000000000003...
Simulating p=0.5, q=0.35000000000000003...
Simulating p=0.55, q=0.35000000000000003...
Simulating p=0.6000000000000001, q=0.35000000000000003...
Simulating p=0.65, q=0.35000000000000003...
Simulating p=0.7000000000000001, q=0.35000000000000003...
Simulating p=0.75, q=0.35000000000000003...
Simulating p=0.8, q=0.35000000000000003...
Simulating p=0.8500000000000001, q=0.35000000000000003...
Simulating p=0.9, q=0.35000000000000003...
Simulating p=0.9500000000000001, q=0.35000000000000003...
Simulating p=1.0, q=0.35000000000000003...
Simulating p=0.05, q=0.4...
Simulating p=0.1, q=0.4...
Simulating p=0.15000000000000002, q=0.4...
Simulating p=0.2, q=0.4...
Simulating p=0.25, q=0.4...
Simulating p=0.30000000000000004, q=0.4...
Simulating p=0.35000000000000003, q=0.4...
Simulating p=0.4, q=0.4...
Simulating p=0.45, q=0.4...
Simulating p=0.5, q=0.4...
Simulating p=0.55, q=0.4...
Simulating p=0.6000000000000001, q=0.4...
Simulating p=0.65, q=0.4...
Simulating p=0.7000000000000001, q=0.4...
Simulating p=0.75, q=0.4...
Simulating p=0.8, q=0.4...
Simulating p=0.8500000000000001, q=0.4...
Simulating p=0.9, q=0.4...
Simulating p=0.9500000000000001, q=0.4...
Simulating p=1.0, q=0.4...
Simulating p=0.05, q=0.45...
Simulating p=0.1, q=0.45...
Simulating p=0.15000000000000002, q=0.45...
Simulating p=0.2, q=0.45...
Simulating p=0.25, q=0.45...
Simulating p=0.30000000000000004, q=0.45...
Simulating p=0.35000000000000003, q=0.45...
Simulating p=0.4, q=0.45...
Simulating p=0.45, q=0.45...
Simulating p=0.5, q=0.45...
Simulating p=0.55, q=0.45...
Simulating p=0.6000000000000001, q=0.45...
Simulating p=0.65, q=0.45...
Simulating p=0.7000000000000001, q=0.45...
Simulating p=0.75, q=0.45...
Simulating p=0.8, q=0.45...
Simulating p=0.8500000000000001, q=0.45...
Simulating p=0.9, q=0.45...
Simulating p=0.9500000000000001, q=0.45...
Simulating p=1.0, q=0.45...
Simulating p=0.05, q=0.5...
Simulating p=0.1, q=0.5...
Simulating p=0.15000000000000002, q=0.5...
Simulating p=0.2, q=0.5...
Simulating p=0.25, q=0.5...
Simulating p=0.30000000000000004, q=0.5...
Simulating p=0.35000000000000003, q=0.5...
Simulating p=0.4, q=0.5...
Simulating p=0.45, q=0.5...
Simulating p=0.5, q=0.5...
Simulating p=0.55, q=0.5...
Simulating p=0.6000000000000001, q=0.5...
Simulating p=0.65, q=0.5...
Simulating p=0.7000000000000001, q=0.5...
Simulating p=0.75, q=0.5...
Simulating p=0.8, q=0.5...
Simulating p=0.8500000000000001, q=0.5...
Simulating p=0.9, q=0.5...
Simulating p=0.9500000000000001, q=0.5...
Simulating p=1.0, q=0.5...
Simulating p=0.05, q=0.55...
Simulating p=0.1, q=0.55...
Simulating p=0.15000000000000002, q=0.55...
Simulating p=0.2, q=0.55...
Simulating p=0.25, q=0.55...
Simulating p=0.30000000000000004, q=0.55...
Simulating p=0.35000000000000003, q=0.55...
Simulating p=0.4, q=0.55...
Simulating p=0.45, q=0.55...
Simulating p=0.5, q=0.55...
Simulating p=0.55, q=0.55...
Simulating p=0.6000000000000001, q=0.55...
Simulating p=0.65, q=0.55...
Simulating p=0.7000000000000001, q=0.55...
Simulating p=0.75, q=0.55...
Simulating p=0.8, q=0.55...
Simulating p=0.8500000000000001, q=0.55...
Simulating p=0.9, q=0.55...
Simulating p=0.9500000000000001, q=0.55...
Simulating p=1.0, q=0.55...
Simulating p=0.05, q=0.6000000000000001...
Simulating p=0.1, q=0.6000000000000001...
Simulating p=0.15000000000000002, q=0.6000000000000001...
Simulating p=0.2, q=0.6000000000000001...
Simulating p=0.25, q=0.6000000000000001...
Simulating p=0.30000000000000004, q=0.6000000000000001...
Simulating p=0.35000000000000003, q=0.6000000000000001...
Simulating p=0.4, q=0.6000000000000001...
Simulating p=0.45, q=0.6000000000000001...
Simulating p=0.5, q=0.6000000000000001...
Simulating p=0.55, q=0.6000000000000001...
Simulating p=0.6000000000000001, q=0.6000000000000001...
Simulating p=0.65, q=0.6000000000000001...
Simulating p=0.7000000000000001, q=0.6000000000000001...
Simulating p=0.75, q=0.6000000000000001...
Simulating p=0.8, q=0.6000000000000001...
Simulating p=0.8500000000000001, q=0.6000000000000001...
Simulating p=0.9, q=0.6000000000000001...
Simulating p=0.9500000000000001, q=0.6000000000000001...
Simulating p=1.0, q=0.6000000000000001...
Simulating p=0.05, q=0.65...
Simulating p=0.1, q=0.65...
Simulating p=0.15000000000000002, q=0.65...
Simulating p=0.2, q=0.65...
Simulating p=0.25, q=0.65...
Simulating p=0.30000000000000004, q=0.65...
Simulating p=0.35000000000000003, q=0.65...
Simulating p=0.4, q=0.65...
Simulating p=0.45, q=0.65...
Simulating p=0.5, q=0.65...
Simulating p=0.55, q=0.65...
Simulating p=0.6000000000000001, q=0.65...
Simulating p=0.65, q=0.65...
Simulating p=0.7000000000000001, q=0.65...
Simulating p=0.75, q=0.65...
Simulating p=0.8, q=0.65...
Simulating p=0.8500000000000001, q=0.65...
Simulating p=0.9, q=0.65...
Simulating p=0.9500000000000001, q=0.65...
Simulating p=1.0, q=0.65...
Simulating p=0.05, q=0.7000000000000001...
Simulating p=0.1, q=0.7000000000000001...
Simulating p=0.15000000000000002, q=0.7000000000000001...
Simulating p=0.2, q=0.7000000000000001...
Simulating p=0.25, q=0.7000000000000001...
Simulating p=0.30000000000000004, q=0.7000000000000001...
Simulating p=0.35000000000000003, q=0.7000000000000001...
Simulating p=0.4, q=0.7000000000000001...
Simulating p=0.45, q=0.7000000000000001...
Simulating p=0.5, q=0.7000000000000001...
Simulating p=0.55, q=0.7000000000000001...
Simulating p=0.6000000000000001, q=0.7000000000000001...
Simulating p=0.65, q=0.7000000000000001...
Simulating p=0.7000000000000001, q=0.7000000000000001...
Simulating p=0.75, q=0.7000000000000001...
Simulating p=0.8, q=0.7000000000000001...
Simulating p=0.8500000000000001, q=0.7000000000000001...
Simulating p=0.9, q=0.7000000000000001...
Simulating p=0.9500000000000001, q=0.7000000000000001...
Simulating p=1.0, q=0.7000000000000001...
Simulating p=0.05, q=0.75...
Simulating p=0.1, q=0.75...
Simulating p=0.15000000000000002, q=0.75...
Simulating p=0.2, q=0.75...
Simulating p=0.25, q=0.75...
Simulating p=0.30000000000000004, q=0.75...
Simulating p=0.35000000000000003, q=0.75...
Simulating p=0.4, q=0.75...
Simulating p=0.45, q=0.75...
Simulating p=0.5, q=0.75...
Simulating p=0.55, q=0.75...
Simulating p=0.6000000000000001, q=0.75...
Simulating p=0.65, q=0.75...
Simulating p=0.7000000000000001, q=0.75...
Simulating p=0.75, q=0.75...
Simulating p=0.8, q=0.75...
Simulating p=0.8500000000000001, q=0.75...
Simulating p=0.9, q=0.75...
Simulating p=0.9500000000000001, q=0.75...
Simulating p=1.0, q=0.75...
Simulating p=0.05, q=0.8...
Simulating p=0.1, q=0.8...
Simulating p=0.15000000000000002, q=0.8...
Simulating p=0.2, q=0.8...
Simulating p=0.25, q=0.8...
Simulating p=0.30000000000000004, q=0.8...
Simulating p=0.35000000000000003, q=0.8...
Simulating p=0.4, q=0.8...
Simulating p=0.45, q=0.8...
Simulating p=0.5, q=0.8...
Simulating p=0.55, q=0.8...
Simulating p=0.6000000000000001, q=0.8...
Simulating p=0.65, q=0.8...
Simulating p=0.7000000000000001, q=0.8...
Simulating p=0.75, q=0.8...
Simulating p=0.8, q=0.8...
Simulating p=0.8500000000000001, q=0.8...
Simulating p=0.9, q=0.8...
Simulating p=0.9500000000000001, q=0.8...
Simulating p=1.0, q=0.8...
Simulating p=0.05, q=0.8500000000000001...
Simulating p=0.1, q=0.8500000000000001...
Simulating p=0.15000000000000002, q=0.8500000000000001...
Simulating p=0.2, q=0.8500000000000001...
Simulating p=0.25, q=0.8500000000000001...
Simulating p=0.30000000000000004, q=0.8500000000000001...
Simulating p=0.35000000000000003, q=0.8500000000000001...
Simulating p=0.4, q=0.8500000000000001...
Simulating p=0.45, q=0.8500000000000001...
Simulating p=0.5, q=0.8500000000000001...
Simulating p=0.55, q=0.8500000000000001...
Simulating p=0.6000000000000001, q=0.8500000000000001...
Simulating p=0.65, q=0.8500000000000001...
Simulating p=0.7000000000000001, q=0.8500000000000001...
Simulating p=0.75, q=0.8500000000000001...
Simulating p=0.8, q=0.8500000000000001...
Simulating p=0.8500000000000001, q=0.8500000000000001...
Simulating p=0.9, q=0.8500000000000001...
Simulating p=0.9500000000000001, q=0.8500000000000001...
Simulating p=1.0, q=0.8500000000000001...
Simulating p=0.05, q=0.9...
Simulating p=0.1, q=0.9...
Simulating p=0.15000000000000002, q=0.9...
Simulating p=0.2, q=0.9...
Simulating p=0.25, q=0.9...
Simulating p=0.30000000000000004, q=0.9...
Simulating p=0.35000000000000003, q=0.9...
Simulating p=0.4, q=0.9...
Simulating p=0.45, q=0.9...
Simulating p=0.5, q=0.9...
Simulating p=0.55, q=0.9...
Simulating p=0.6000000000000001, q=0.9...
Simulating p=0.65, q=0.9...
Simulating p=0.7000000000000001, q=0.9...
Simulating p=0.75, q=0.9...
Simulating p=0.8, q=0.9...
Simulating p=0.8500000000000001, q=0.9...
Simulating p=0.9, q=0.9...
Simulating p=0.9500000000000001, q=0.9...
Simulating p=1.0, q=0.9...
Simulating p=0.05, q=0.9500000000000001...
Simulating p=0.1, q=0.9500000000000001...
Simulating p=0.15000000000000002, q=0.9500000000000001...
Simulating p=0.2, q=0.9500000000000001...
Simulating p=0.25, q=0.9500000000000001...
Simulating p=0.30000000000000004, q=0.9500000000000001...
Simulating p=0.35000000000000003, q=0.9500000000000001...
Simulating p=0.4, q=0.9500000000000001...
Simulating p=0.45, q=0.9500000000000001...
Simulating p=0.5, q=0.9500000000000001...
Simulating p=0.55, q=0.9500000000000001...
Simulating p=0.6000000000000001, q=0.9500000000000001...
Simulating p=0.65, q=0.9500000000000001...
Simulating p=0.7000000000000001, q=0.9500000000000001...
Simulating p=0.75, q=0.9500000000000001...
Simulating p=0.8, q=0.9500000000000001...
Simulating p=0.8500000000000001, q=0.9500000000000001...
Simulating p=0.9, q=0.9500000000000001...
Simulating p=0.9500000000000001, q=0.9500000000000001...
Simulating p=1.0, q=0.9500000000000001...
Simulating p=0.05, q=1.0...
Simulating p=0.1, q=1.0...
Simulating p=0.15000000000000002, q=1.0...
Simulating p=0.2, q=1.0...
Simulating p=0.25, q=1.0...
Simulating p=0.30000000000000004, q=1.0...
Simulating p=0.35000000000000003, q=1.0...
Simulating p=0.4, q=1.0...
Simulating p=0.45, q=1.0...
Simulating p=0.5, q=1.0...
Simulating p=0.55, q=1.0...
Simulating p=0.6000000000000001, q=1.0...
Simulating p=0.65, q=1.0...
Simulating p=0.7000000000000001, q=1.0...
Simulating p=0.75, q=1.0...
Simulating p=0.8, q=1.0...
Simulating p=0.8500000000000001, q=1.0...
Simulating p=0.9, q=1.0...
Simulating p=0.9500000000000001, q=1.0...
Simulating p=1.0, q=1.0...
```

In [24]:
```
plt.figure(figsize=(8, 8))
for i in range(Percentages_3.shape[0]):
    plt.errorbar(P, Percentages_3[i, :, 0], yerr=Percentages_3[i, :, 1],
fmt='o--', elinewidth=0.75, capsize=2);

plt.gca().axis('equal')
plt.xlabel('Fire spread prob');
plt.title('Fraction of trees that burned ({}x{} domain)'.format(n_many,
n_many));
plt.legend(Q, loc='upper left')
plt.grid()
```



Although this graph is very crowded, we can see how the vegetation density and fire spread probability interplay. Higher values of both lead to more higher amounts of burning.

In [ ]:

Tutorial 2: Forest Fires with Terrain and Vegetation Type

In tutorial 1, we explored basic cellular automata models and probabilistic cellular automata models for simulating fire spread. In this tutorial, we will explore the effect of two factors that contribute to how fires spread: vegetation type and terrain.

First, we'll rewrite some of the basics from the first tutorial to set up our model.

In [1]:
```
import numpy as np
import scipy as sp
import scipy.sparse
import matplotlib as mpl
import matplotlib.pyplot as plt

empty = 0
unburned = 1
burning = 2
burned = 3
```

In [2]:
```
def create_world(n, q):
    world = np.zeros((n+2, n+2))
    forest = world[1:-1, 1:-1]
    forest[:, :] = np.random.choice([0, 1], p=[1-q, q], size=(n, n))
    return world

def show_world(W, title=None, **args):
    if 'cmap' not in args:
        args['cmap'] = 'jet'
    if 'vmin' not in args and 'vmax' not in args:
        args['vmin'] = 0
        args['vmax'] = 3
    plt.figure(figsize=(8, 8))
    plt.matshow(W, fignum=1, **args)
    plt.xlabel('column')
    plt.ylabel('row')
    plt.colorbar()
    if title is None:
        num_trees = (W > 0).sum()
        if num_trees > 0:
            num_burnt = (W == 3).sum()
            percent = num_burnt / num_trees * 1e2
            title = '{} / {} ~= {:.1f}%'.format(num_burnt, num_trees, pe
rcent)
        else:
            title = ''
    plt.title(title)
    pass

def is_empty(W):
    return W == empty

def is_vegetation(W):
    return W > empty

def is_unburned(W):
    return W == 1

def is_burning(W):
    return W == 2

def is_burned(W):
    return W == 3

def count(W, cond_fun):
    return cond_fun(W).sum()

def summarize_world(W):
    def suffix(n):
        return (1, "tree") if n == 1 else (n, "trees")
    m, n = W.shape[0]-2, W.shape[1]-2
    n_trees = count(W, is_vegetation)
    n_unburned = count(W, is_unburned)
    n_burning = count(W, is_burning)
    n_burned = count(W, is_burned)
    print("The world has dimensions: {} x {}".format(m, n))
    print("There are {} cell(s) that have had vegetation in them".format
(n_trees))
    print("There are {} cell(s) of vegetation that are unburned".format(
n_unburned))
    print("There are {} cell(s) of vegetation on fire".format(n_burning
))
    print("There are {} cell(s) of vegetation completely burned".format(
n_burned))

def start_fire(W, cells=None):
    W_new = W.copy()

    if cells == None:
        F = W[1:-1, 1:-1]
        W_new = W.copy()
        F_new = W_new[1:-1, 1:-1]
        I, J = np.where(is_unburned(F)) # Positions of all trees
        if len(I) > 0:
            k = np.random.choice(range(len(I))) # Index of tree to ignit
e
            i, j = I[k], J[k]
            assert F_new[i, j] == 1, "Attempting to ignite a non-tree?"
            F_new[i, j] += 1
    else:
        W_new = W.copy()

        for x,y in cells:
            W_new[x,y] = 2

    return W_new
```

Forest Type

Different types of forest have different propensities for fire spreading. According to "A cellular automata model for forest fire spreading simulation" by Xuehua et al (see Citations.txt in this directory for full citation), we can formulate forests as one of three types: coniferous, broadleaf, or mixed.

Coniferous forests have the lowest water content and thus burn the fastest; broadleaf forests have higher water contents and thus burn slower. We can formulate these differences with the following probabilities of spread.

- Coniferous: .9
- Broadleaf: .7

These probabilities could be adjusted with more empircal data from actual forest fires, but were artificially created here to illustrate the differences in forest type.

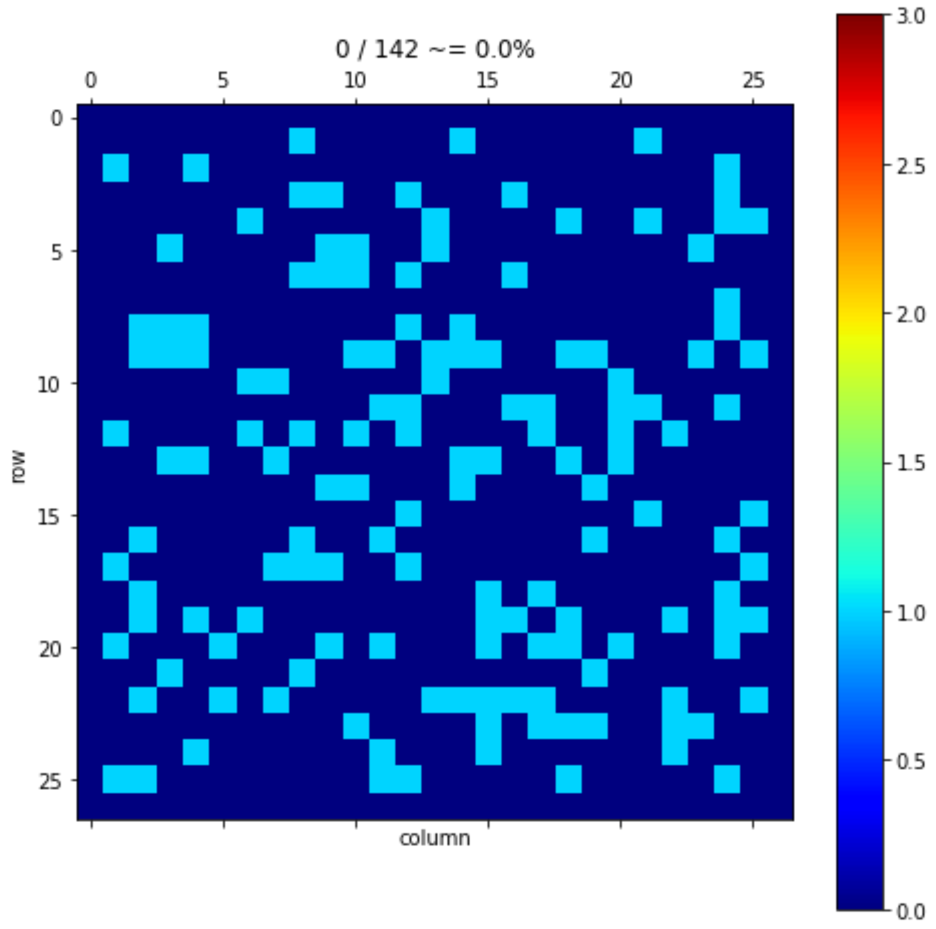
In the funciton below called `create_transition_matrix` we will create a matrix with probabilities for each cell depending on the tree type. If the forest is of mixed type, we randomly put either the Coniferous or Broadleaf probability in cells. This matrix will be cross-referenced with the cells that have vegetation in the spread function later.

```
In [3]: def create_transition_matrix(world, tree_type):
            if tree_type not in ['C', 'B', 'M']:
                raise ValueError('Tree type must be coniferous (\'C\'), Broadlea
        f (\'B\'), or mixed (\'M\')')

            probs = {'C': .9, 'B': .7}

            mat = np.ones(world.shape)

            if tree_type is 'M':
                mat = np.random.rand(*world.shape)

                c_index = mat < .5
                b_index = mat >= .5

                mat[c_index] = probs['C']
                mat[b_index] = probs['B']

            else:
                mat *= probs[tree_type]

            return mat
```

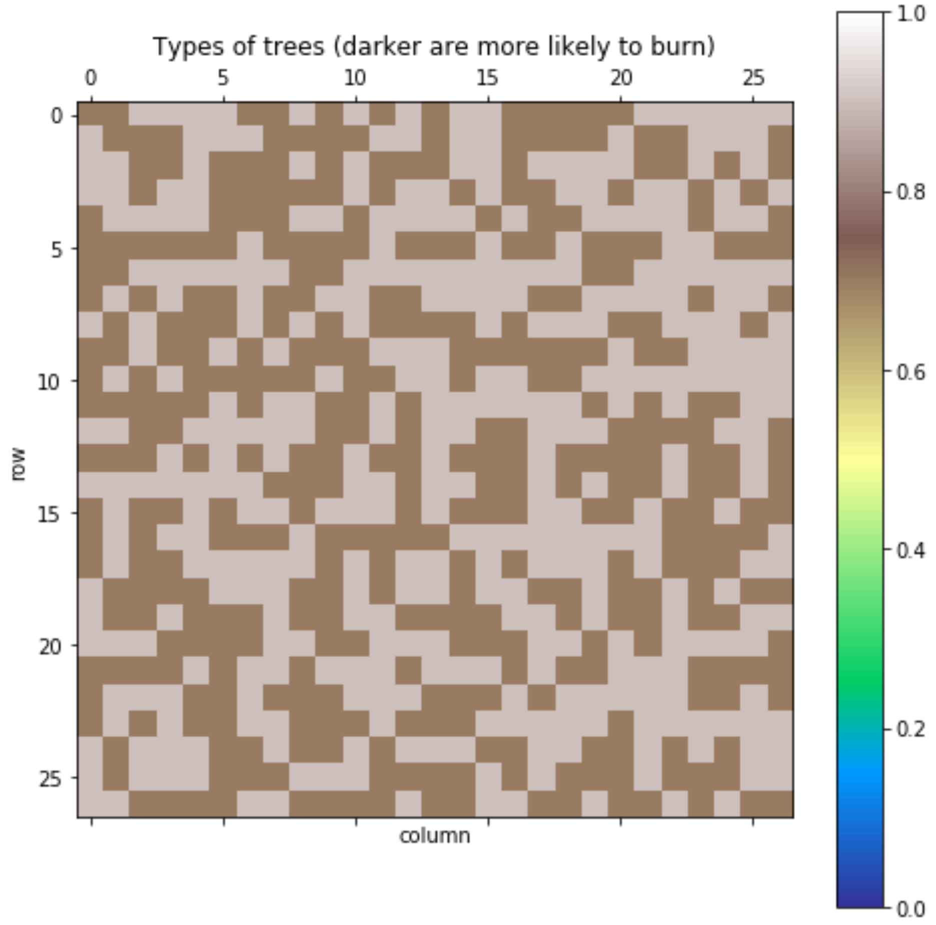
This function lets us visualize our transition matrix.

```
In [4]: def show_tm(W, title=None, **args):
            if 'cmap' not in args:
                args['cmap'] = 'terrain'
            if 'vmin' not in args and 'vmax' not in args:
                args['vmin'] = 0
                args['vmax'] = 1
            plt.figure(figsize=(8, 8))
            plt.matshow(W, fignum=1, **args)
            plt.xlabel('column')
            plt.ylabel('row')
            plt.colorbar()

            title = 'Types of trees (darker are more likely to burn)'

            plt.title(title)
            pass
```

In [5]:
```python
World = create_world(25, 0.25)
show_world(World)
```

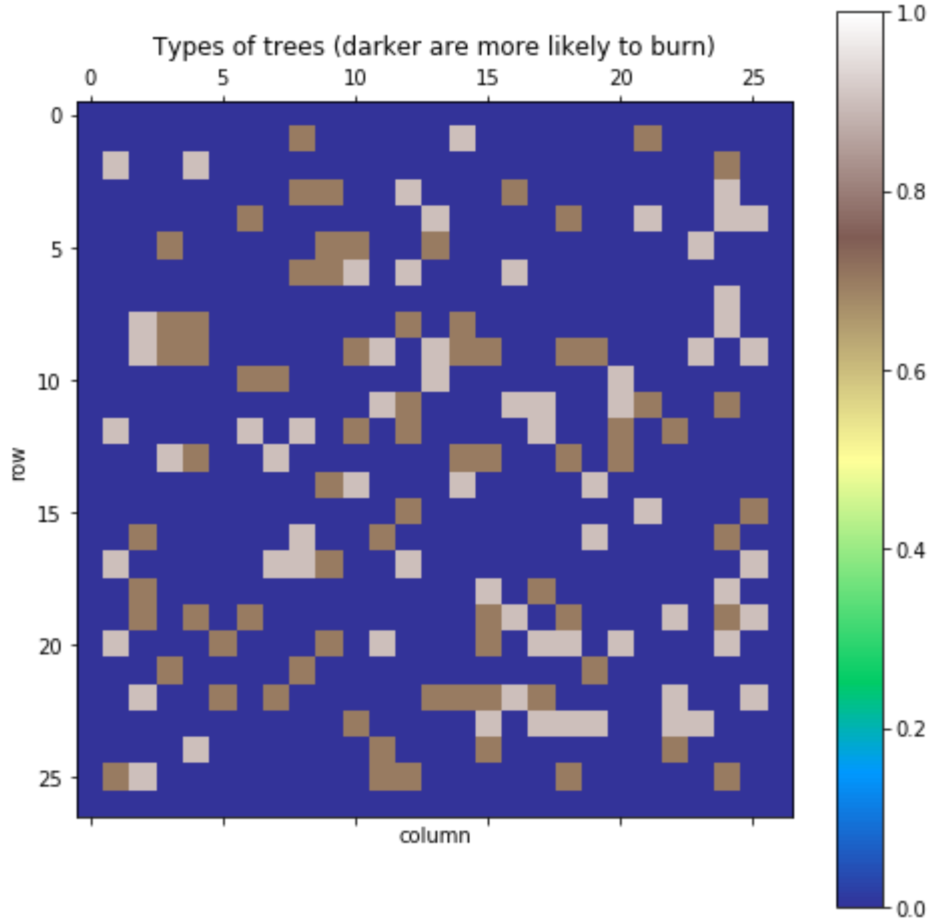


In [6]:
```
tm = create_transition_matrix(World, 'M')
show_tm(tm)
```



We can also visualize the transition matrix only where trees are present as below to get a more accurate picture.

In [7]:
```
veg = is_vegetation(World)

tm_tree = np.multiply(tm, veg)

show_tm(tm_tree)
```



Probabilistic Spreading with forest type

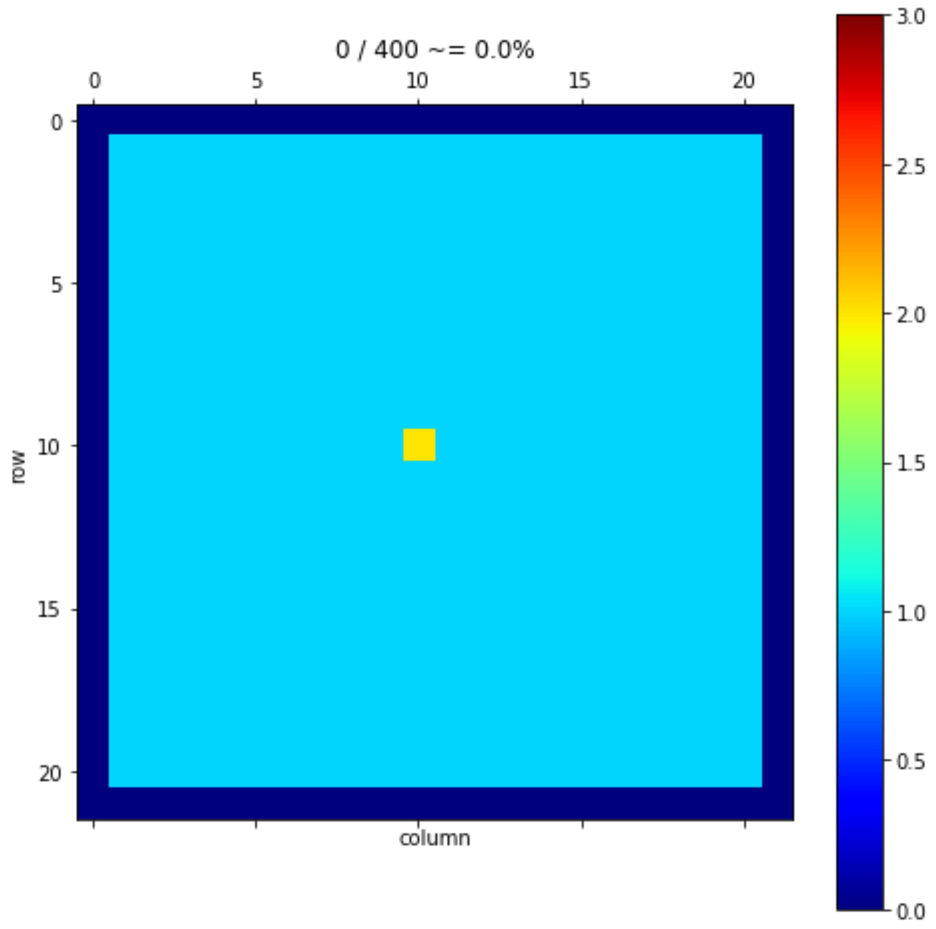
Building off our earlier probabilistic model, we can edit our spread fire funciton to take the type of forest into account

In [8]:
```
def spread_fire_forest_type(W, transition_matrix):
    W_new = W.copy()
    Vegetation = is_unburned(W)
    Fires = is_burning(W)

    num_neighbors_on_fire = (Fires[:-2, :-2].astype(int) + Fires[1:-1, :
-2] + Fires[2:, :-2]
                             + Fires[:-2, 1:-1]
+ Fires[2:, 1:-1]
                             + Fires[:-2, 2:]            + Fires[1:-1, 2
:]  + Fires[2:, 2:])

    num_neighbors_on_fire = np.multiply(num_neighbors_on_fire, Vegetatio
n[1:-1, 1:-1].astype(int))

    # Here, we take into account the transition probabilities at each ce
ll
    fire_prob_matrix = np.ones(num_neighbors_on_fire.shape) - transition
_matrix[1:-1, 1:-1]

    fire_prob_matrix = 1.0 - np.power(fire_prob_matrix, num_neighbors_on
_fire)

    # As before, we only look at the cells where trees actually exist
    fire_prob_matrix = np.multiply(fire_prob_matrix, Vegetation[1:-1, 1:
-1].astype(int))

    randys = np.random.rand(*fire_prob_matrix.shape)
    new_on_fires = randys < fire_prob_matrix

    W_new[1:-1, 1:-1] += new_on_fires

    W_new[1:-1, 1:-1] += Fires[1:-1, 1:-1]

    return W_new
```

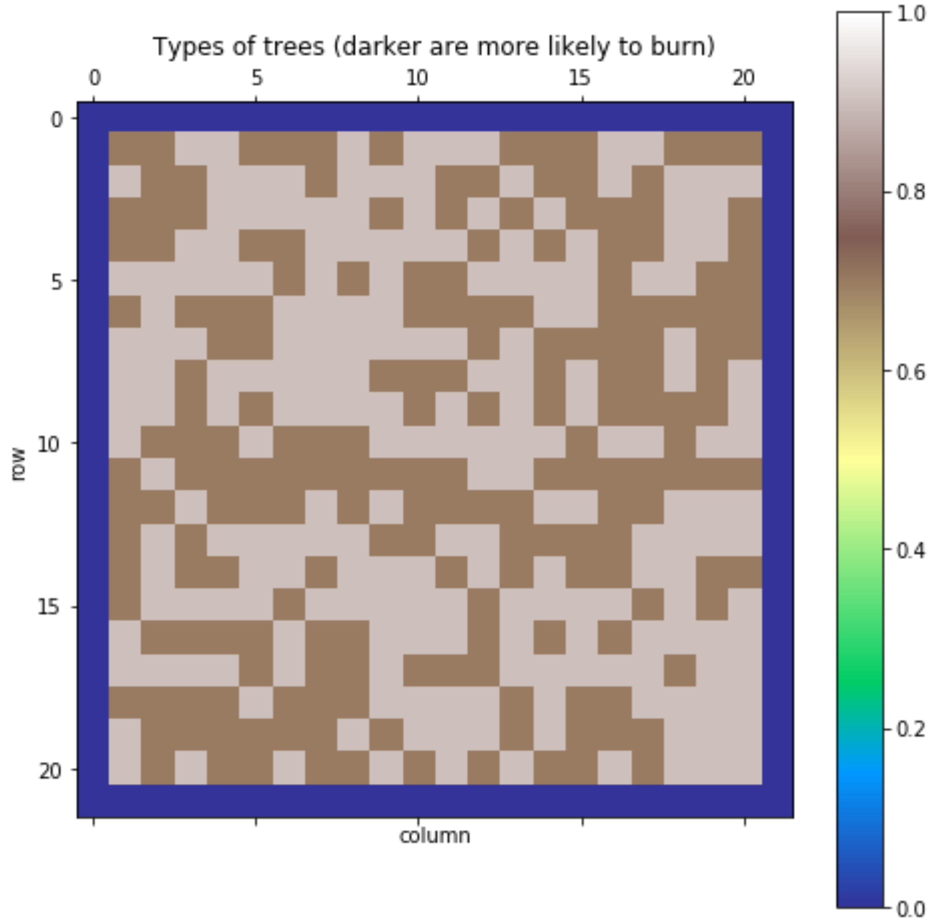
Below, an example using our new spread funciton.

In [12]:
```
World = create_world(20, 1)
tm = create_transition_matrix(World, 'M')
World_next = start_fire(World, [(10,10)])
show_world(World_next)
summarize_world(World_next)
```

```
The world has dimensions: 20 x 20
There are 400 cell(s) that have had vegetation in them
There are 399 cell(s) of vegetation that are unburned
There are 1 cell(s) of vegetation on fire
There are 0 cell(s) of vegetation completely burned
```

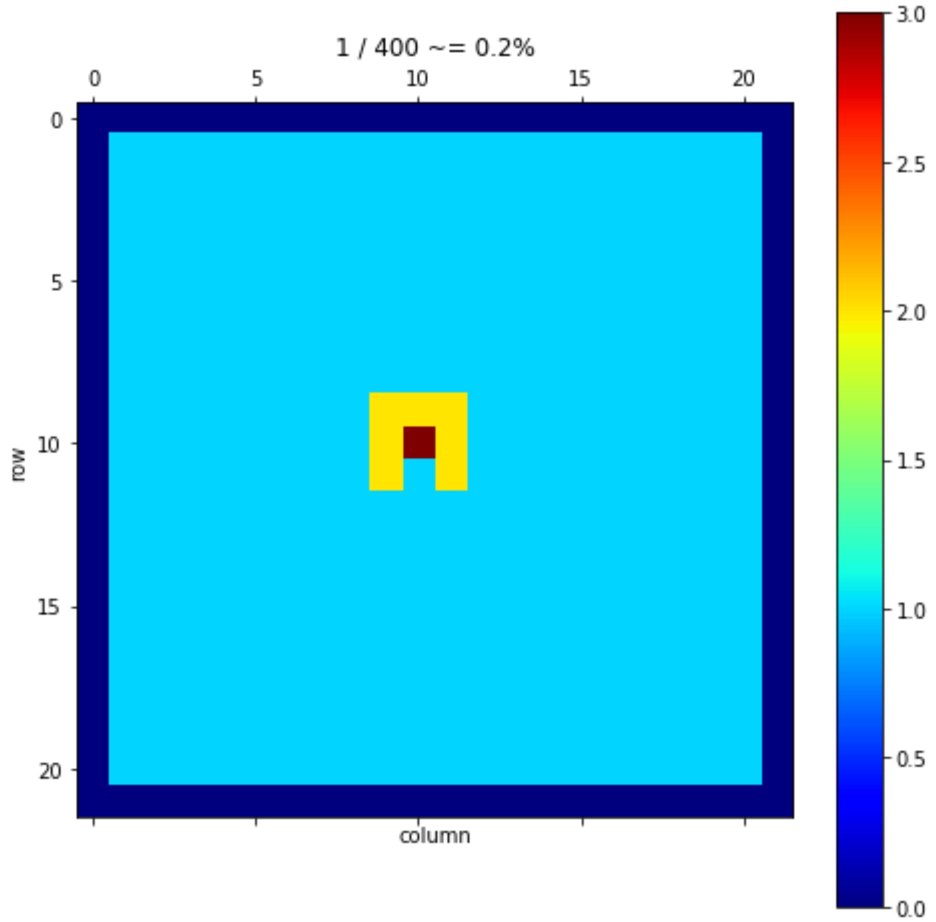


In [13]:
```
show_tm(np.multiply(tm, is_vegetation(World)))
```



In [14]:
```
# Second Timestep
World_next = spread_fire_forest_type(World_next, tm)
show_world(World_next)
summarize_world(World_next)
```

```
The world has dimensions: 20 x 20
There are 400 cell(s) that have had vegetation in them
There are 392 cell(s) of vegetation that are unburned
There are 7 cell(s) of vegetation on fire
There are 1 cell(s) of vegetation completely burned
```



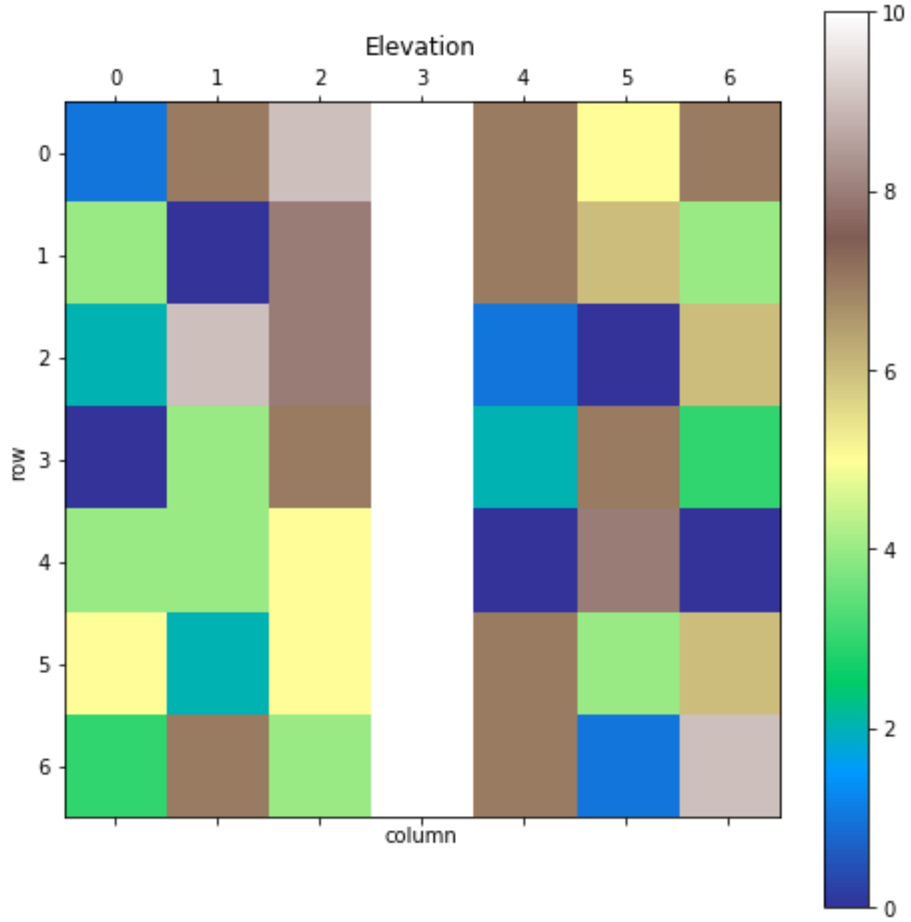
Terrain

In addition to forest types, the elevation in a forest can impact the spread of fire. If a central cell is higher than surrounding cells, then the angle between the flames and the fuel will cause the fire to spead faster. We can model this phenomenon as below.

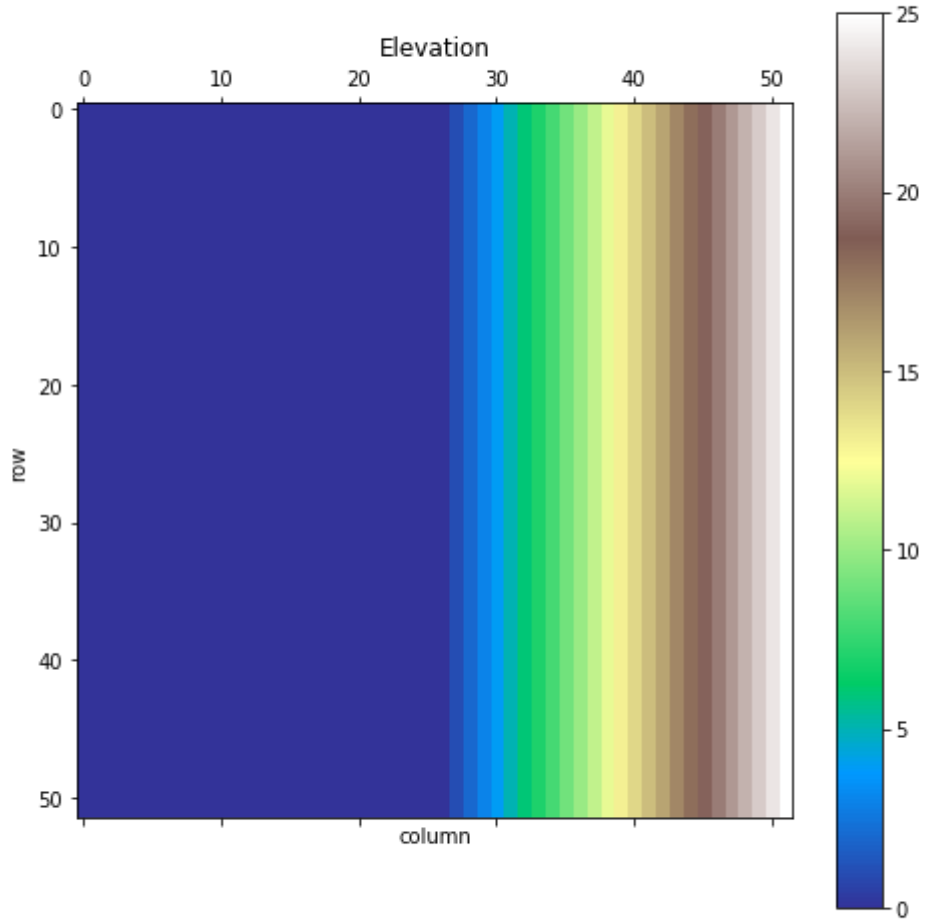
First, we'll create some functions to add elevations to a world.

In [15]:
```
# create random elevations between 0 and 10
def create_random_elevations(world):
    elevs = np.random.randint(0, 10, size=world.shape)

    return elevs

# add ridge of elevation 10 down the middle of the map
def add_middle_ridge(elevs):

    elevs[:, elevs.shape[1]//2] = 10

    return elevs

# create a map with 0 elevation on one half and increasing on the other
def create_increasing_elevations(world):
    elevs = np.zeros(world.shape)
    num_tiers = elevs.shape[1]//2

    tiers = np.linspace(1, num_tiers, num_tiers)

    r = num_tiers

    if elevs.shape[1] % 2:
        r += 1

    for i in range(1, r):
        elevs[:, num_tiers + i] = np.full((1, elevs.shape[0]), i)

    return elevs

def show_elevs(elevs, title=None, **args):
    if 'cmap' not in args:
        args['cmap'] = 'terrain'
    if 'vmin' not in args and 'vmax' not in args:
        args['vmin'] = 0
        args['vmax'] = np.max(elevs)
    plt.figure(figsize=(8, 8))
    plt.matshow(elevs, fignum=1, **args)
    plt.xlabel('column')
    plt.ylabel('row')
    plt.colorbar()

    title = 'Elevation'

    plt.title(title)
    pass
```

In [16]:
```
w = create_world(5, 1)
e = create_random_elevations(w)
e = add_middle_ridge(e)

show_elevs(e)
```



In [17]:
```
w = create_world(50, 1)
e = create_increasing_elevations(w)

show_elevs(e)
```



To actually see how a cell compares to its neighbors, we'll use the gradient to approximate how a cell compares to its neighbors. We can look at the gradient in both the row-wise and column-wise directions to determine how a cell compares to its N, E, S, W neighbors and then shift this gradient to center the values on the cell rather than its neighbors.

In [18]:
```
def elev_transition_prob(elevs_mat):
    vert, row = np.gradient(elevs_mat)

    c0 = np.roll(row, 1, axis=1)
    c0[:, 0] = 0

    c1 = np.roll(vert, 1, axis=0)
    c1[0, :] = 0

    c = np.maximum(c0, c1)

    diff = c.max() - c.min()
    if not diff: diff = 1

    c = (c - c.min()) / diff

    return c
```

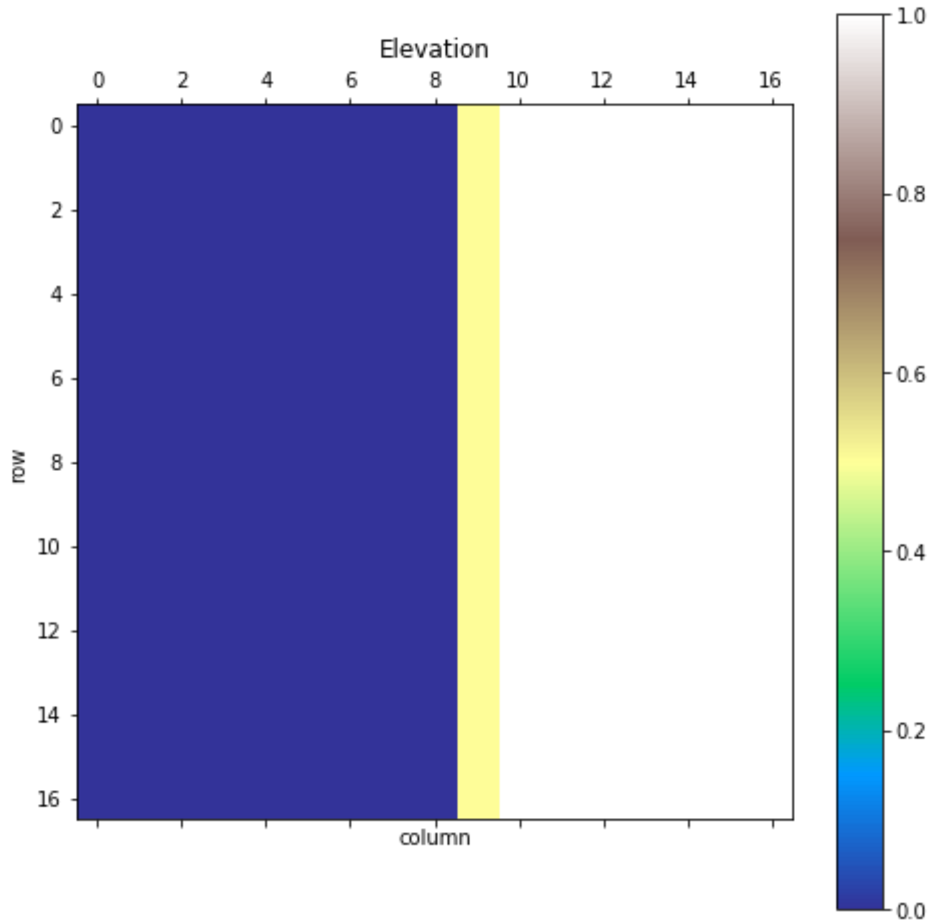
In [19]:
```
w = create_world(15, 1)
e = create_increasing_elevations(w)

# e = np.matrix([[0,0,0,0,0],
#               [0,0,0,0,0],
#               [0,0,0,0,0],
#               [0,0,0,0,0],
#               [0,0,0,0,0]])

t = elev_transition_prob(e)
```

```
In [20]: t
```

```
Out[20]: array([[0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0.5, 1. , 1. , 1.
         ,
                 1. , 1. , 1. , 1. ],
                [0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0.5, 1. , 1. , 1.
         ,
                 1. , 1. , 1. , 1. ],
                [0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0.5, 1. , 1. , 1.
         ,
                 1. , 1. , 1. , 1. ],
                [0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0.5, 1. , 1. , 1.
         ,
                 1. , 1. , 1. , 1. ],
                [0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0.5, 1. , 1. , 1.
         ,
                 1. , 1. , 1. , 1. ],
                [0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0.5, 1. , 1. , 1.
         ,
                 1. , 1. , 1. , 1. ],
                [0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0.5, 1. , 1. , 1.
         ,
                 1. , 1. , 1. , 1. ],
                [0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0.5, 1. , 1. , 1.
         ,
                 1. , 1. , 1. , 1. ],
                [0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0.5, 1. , 1. , 1.
         ,
                 1. , 1. , 1. , 1. ],
                [0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0.5, 1. , 1. , 1.
         ,
                 1. , 1. , 1. , 1. ],
                [0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0.5, 1. , 1. , 1.
         ,
                 1. , 1. , 1. , 1. ],
                [0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0.5, 1. , 1. , 1.
         ,
                 1. , 1. , 1. , 1. ],
                [0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0.5, 1. , 1. , 1.
         ,
                 1. , 1. , 1. , 1. ],
                [0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0.5, 1. , 1. , 1.
         ,
                 1. , 1. , 1. , 1. ],
                [0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0.5, 1. , 1. , 1.
         ,
                 1. , 1. , 1. , 1. ],
                [0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0.5, 1. , 1. , 1.
         ,
                 1. , 1. , 1. , 1. ]])
```

We can also use our `show_elevs` function to visualize the gradient on our map.

In [22]:
```
show_elevs(t)
```



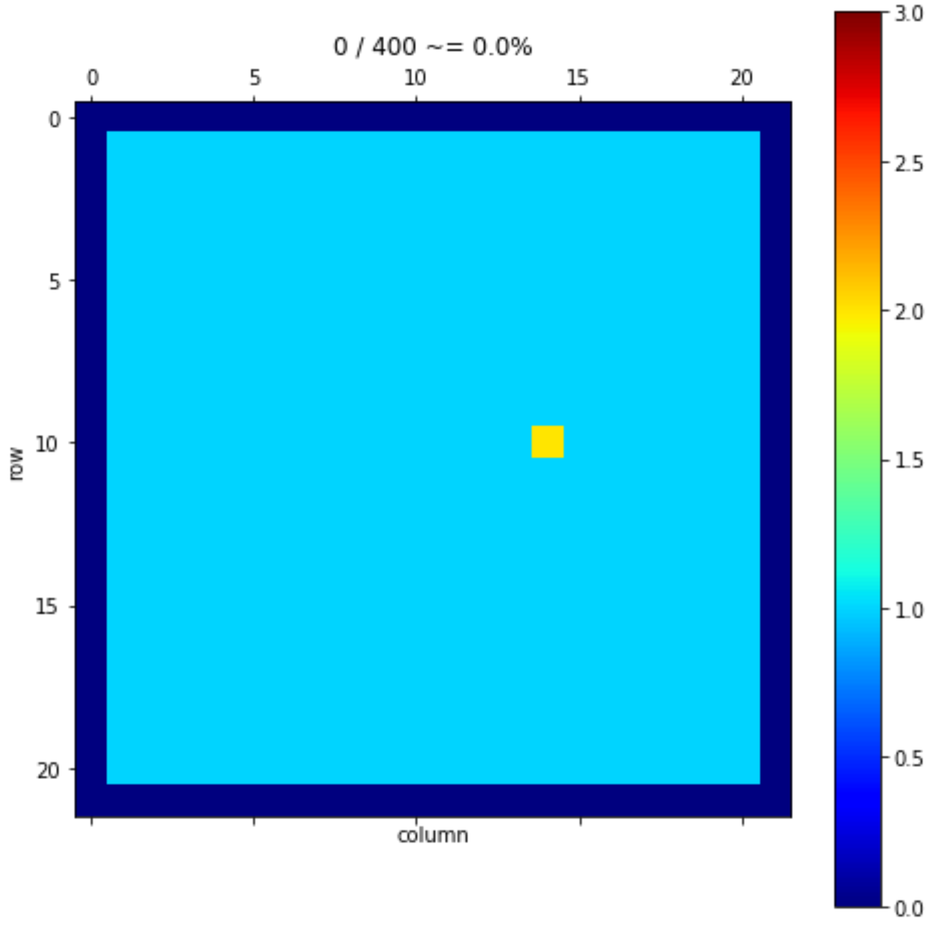
To test the effectiveness of this elevation method, we can create a fire spread where fires only spread according to the outputs of our elevation funciton

```
In [23]: def spread_fire_elevation(W, elevs):
             W_new = W.copy()
             Vegetation = is_unburned(W)
             Fires = is_burning(W)

             '''
             Add elevation effect here

             1) get the elevation matrix
             2) only look at the elements that border the areas on fire
             3) add this element to the exisiting transition probabilities
             '''
             # testing isolated effects of terrain
             fire_prob_matrix = np.zeros((W.shape[0] - 2, W.shape[1] - 2 ))

             elevation_effect = 1
             elev_p = elev_transition_prob(elevs)
             elev_p = elev_p[1:-1, 1:-1]
             elev_i = Vegetation[1:-1, 1:-1] \
                                 & (
                                       Fires[:-2, :-2]  | Fires[1:-1, :-2] | Fire
         s[2:, :-2]
                                     | Fires[:-2, 1:-1] |                   Fire
         s[2:, 1:-1]
                                     | Fires[:-2, 2:]   | Fires[1:-1, 2:]  | Fire
         s[2:, 2:]
                                    )

             e = np.multiply(elev_p, elev_i)
             fire_prob_matrix += e * elevation_effect

             # normal part
             fire_prob_matrix = np.multiply(fire_prob_matrix, Vegetation[1:-1, 1:
         -1].astype(int))

             randys = np.random.rand(*fire_prob_matrix.shape)
             new_on_fires = randys < fire_prob_matrix

             W_new[1:-1, 1:-1] += new_on_fires

             W_new[1:-1, 1:-1] += Fires[1:-1, 1:-1]

             return W_new
```

Here, we'll create a mock world of increasing elevations and start our fire on the elevated part. As you can see, within a few time steps the fire is only spreading on the elevated part. Since the model is still probabilistic, it doesnt spread entirely evenly, however it does not spread on the flat part of the forest.

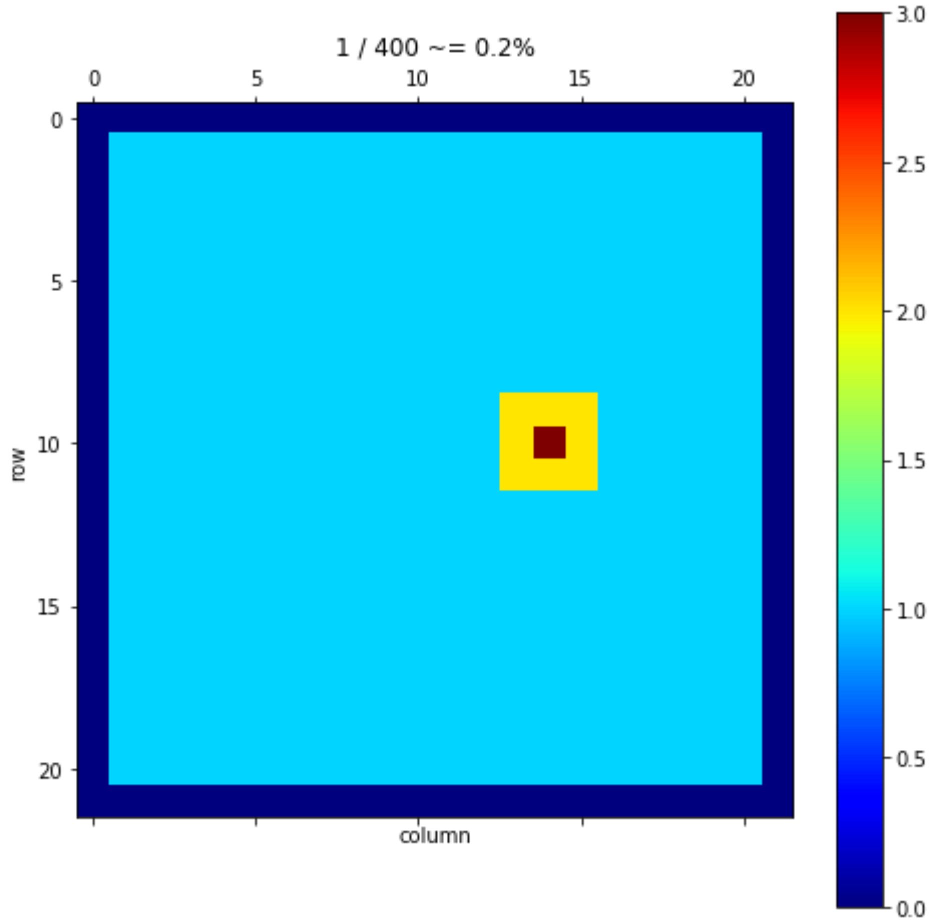
In [24]:
```
w = create_world(20, 1)
w = start_fire(w, [[10,14]])
e = create_increasing_elevations(w)

show_world(w)
summarize_world(w)
```

```
The world has dimensions: 20 x 20
There are 400 cell(s) that have had vegetation in them
There are 399 cell(s) of vegetation that are unburned
There are 1 cell(s) of vegetation on fire
There are 0 cell(s) of vegetation completely burned
```



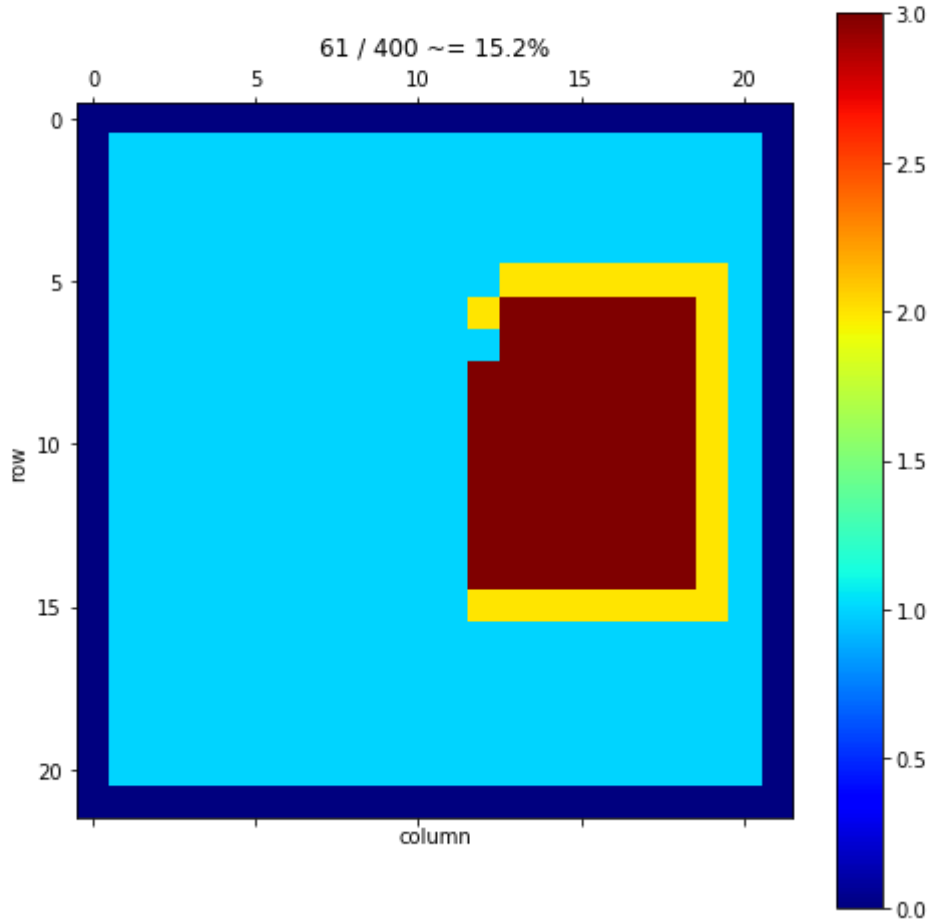
In [25]:
```
# step once
w = spread_fire_elevation(w, e)
show_world(w)
summarize_world(w)
```

```
The world has dimensions: 20 x 20
There are 400 cell(s) that have had vegetation in them
There are 391 cell(s) of vegetation that are unburned
There are 8 cell(s) of vegetation on fire
There are 1 cell(s) of vegetation completely burned
```



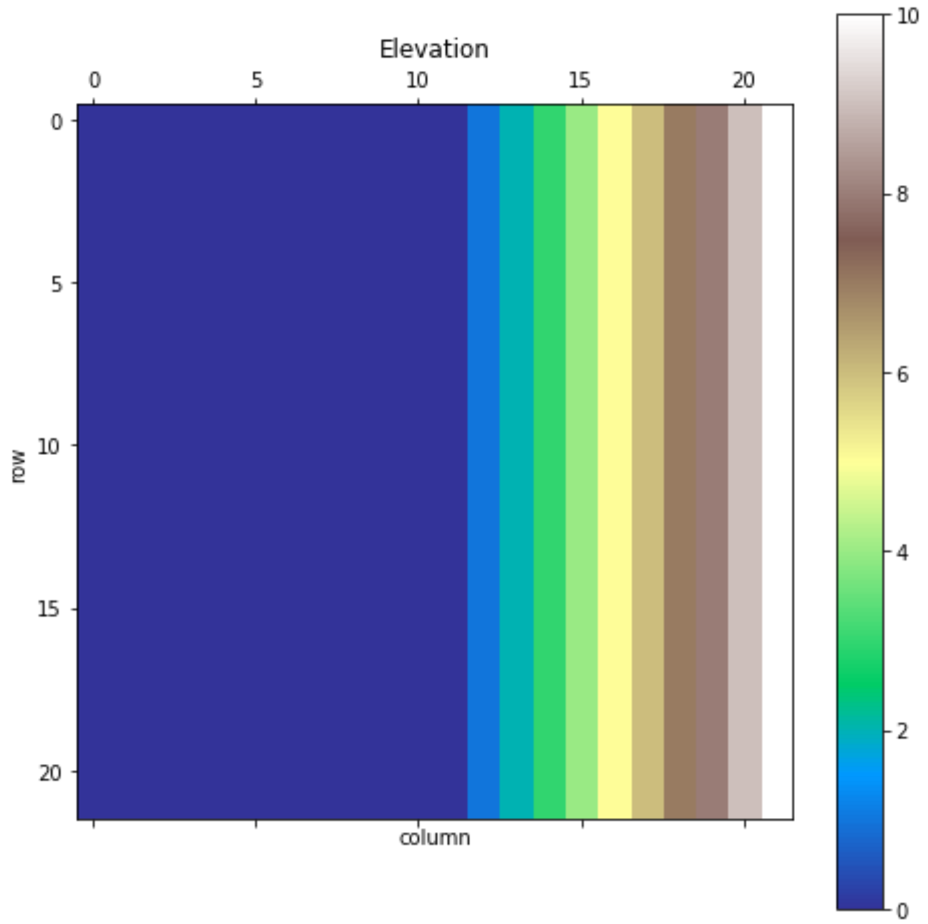
In [26]:
```
# step several more times to demonstrate effect
w = spread_fire_elevation(w, e)
w = spread_fire_elevation(w, e)
w = spread_fire_elevation(w, e)
w = spread_fire_elevation(w, e)

show_world(w)
summarize_world(w)
```

```
The world has dimensions: 20 x 20
There are 400 cell(s) that have had vegetation in them
There are 314 cell(s) of vegetation that are unburned
There are 25 cell(s) of vegetation on fire
There are 61 cell(s) of vegetation completely burned
```



As we can see, the fire is spreading along the gradient as reflected below.

In [27]:
```
show_elevs(e)
```



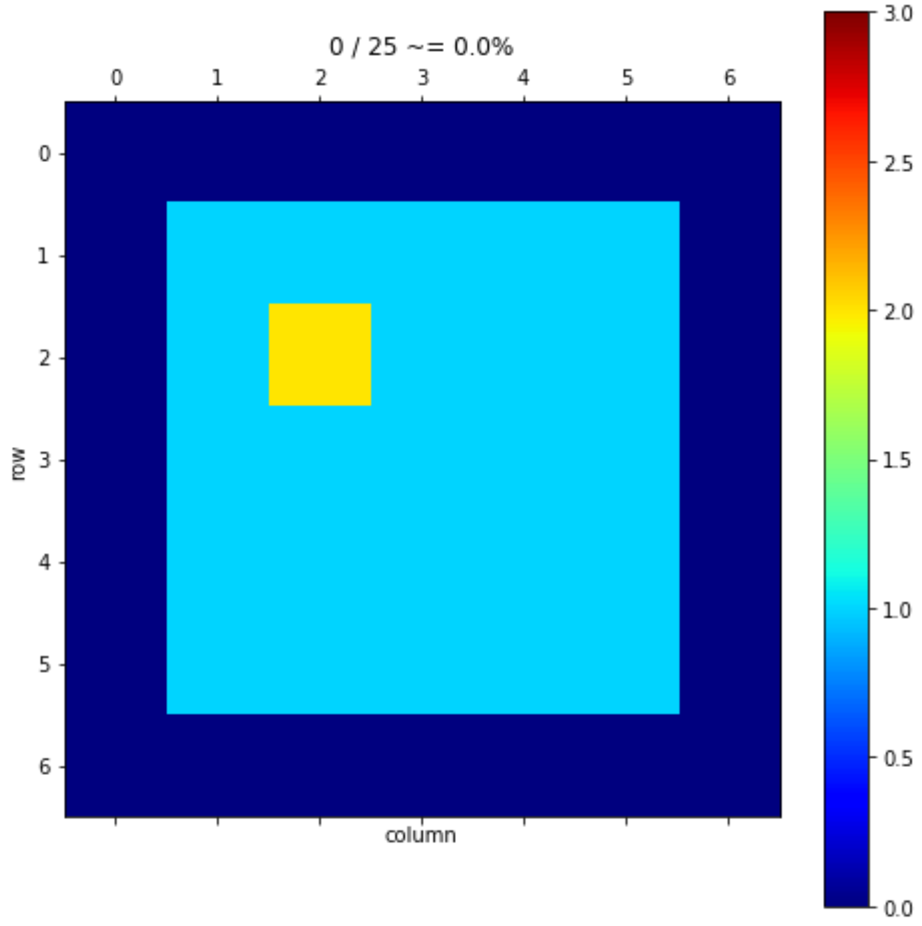
Below, we combine the elevation and terrain types.

In [29]:
```
def spread_fire_both(W, transition_matrix, elevs):
    W_new = W.copy()
    Vegetation = is_unburned(W)
    Fires = is_burning(W)


    num_neighbors_on_fire = (Fires[:-2, :-2].astype(int) + Fires[1:-1, :
-2] + Fires[2:, :-2]
                             + Fires[:-2, 1:-1]
+ Fires[2:, 1:-1]
                             + Fires[:-2, 2:]           + Fires[1:-1, 2
:]  + Fires[2:, 2:])

    num_neighbors_on_fire = np.multiply(num_neighbors_on_fire, Vegetatio
n[1:-1, 1:-1].astype(int))

    fire_prob_matrix = np.ones(num_neighbors_on_fire.shape) - transition
_matrix[1:-1, 1:-1]

    fire_prob_matrix = 1.0 - np.power(fire_prob_matrix, num_neighbors_on
_fire)

    elevation_effect = 1
    elev_p = elev_transition_prob(elevs)
    elev_p = elev_p[1:-1, 1:-1]
    elev_i = Vegetation[1:-1, 1:-1] \
                        & (
                              Fires[:-2, :-2]  | Fires[1:-1, :-2] | Fire
s[2:, :-2]
                            | Fires[:-2, 1:-1] |                    Fire
s[2:, 1:-1]
                            | Fires[:-2, 2:]   | Fires[1:-1, 2:]  | Fire
s[2:, 2:]
                           )

    e = np.multiply(elev_p, elev_i)
    fire_prob_matrix = .5 * fire_prob_matrix +  .5 * e * elevation_effec
t

    # normal part
    fire_prob_matrix = np.multiply(fire_prob_matrix, Vegetation[1:-1, 1:
-1].astype(int))

    randys = np.random.rand(*fire_prob_matrix.shape)
    new_on_fires = randys < fire_prob_matrix

    W_new[1:-1, 1:-1] += new_on_fires

    W_new[1:-1, 1:-1] += Fires[1:-1, 1:-1]

    return W_new
```

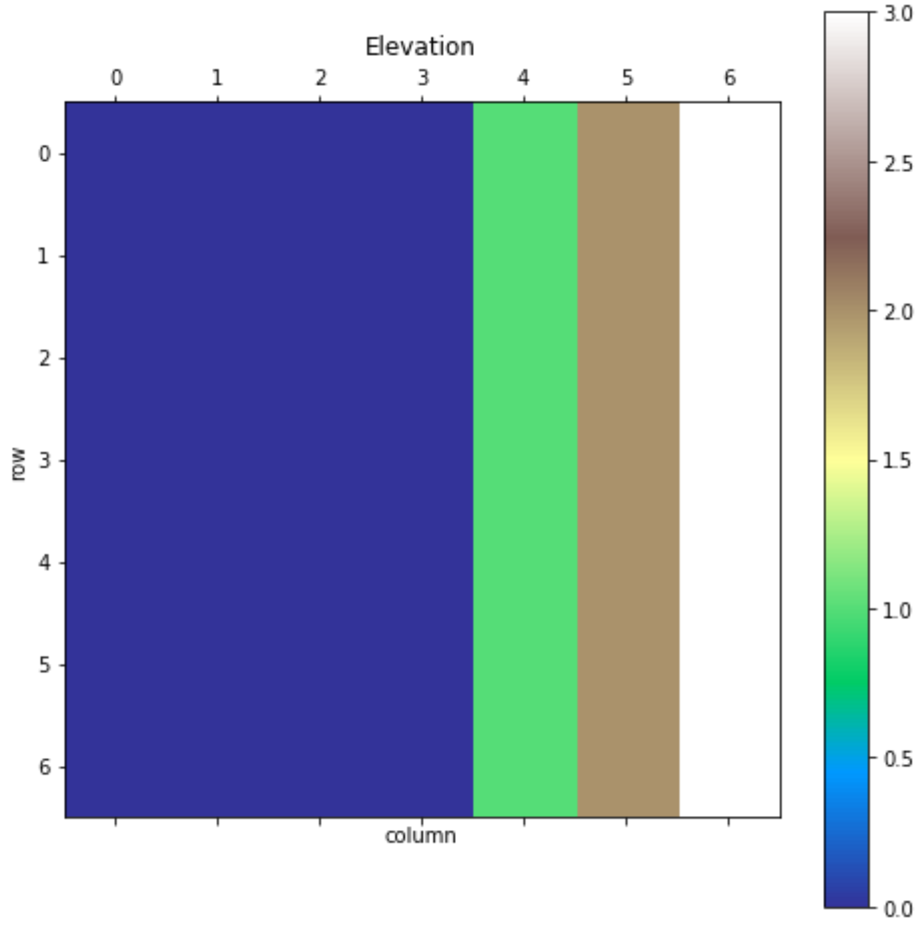
Testing

In [30]:
```
w = create_world(5, 1)
w = start_fire(w, [[2,2]])
tm = create_transition_matrix(w, 'M')
e = create_increasing_elevations(w)

show_world(w)
summarize_world(w)
```

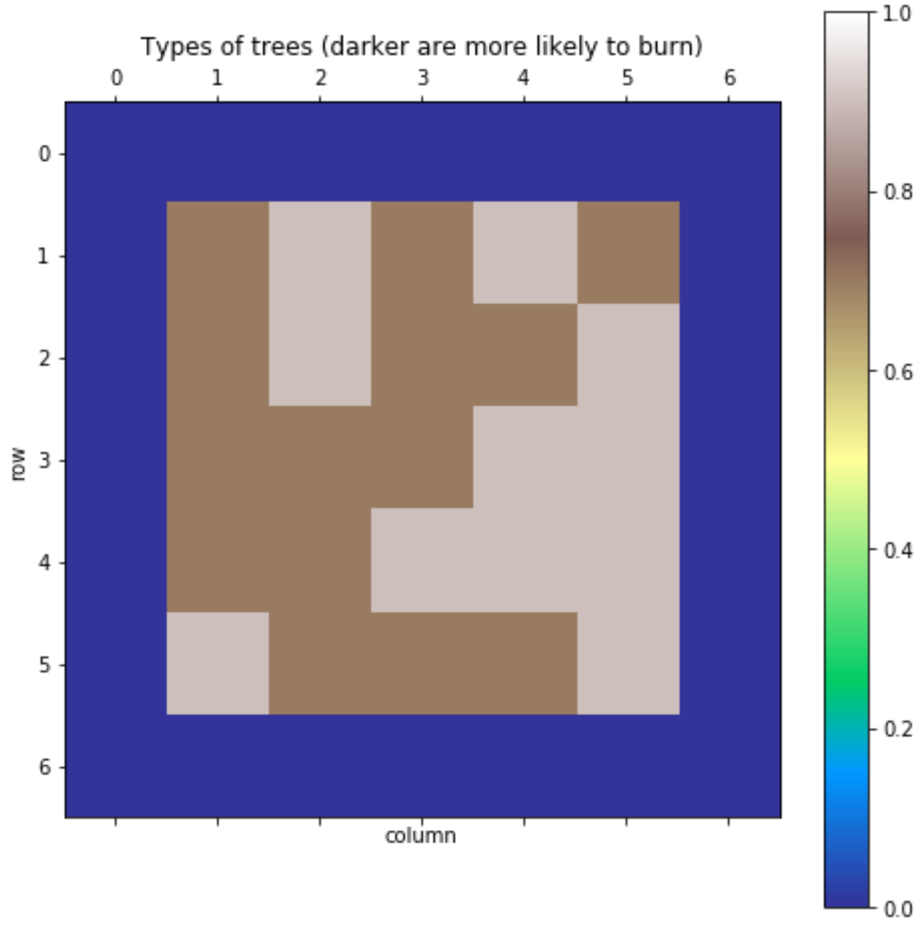
```
The world has dimensions: 5 x 5
There are 25 cell(s) that have had vegetation in them
There are 24 cell(s) of vegetation that are unburned
There are 1 cell(s) of vegetation on fire
There are 0 cell(s) of vegetation completely burned
```



In [31]:
```
show_elevs(e)
```

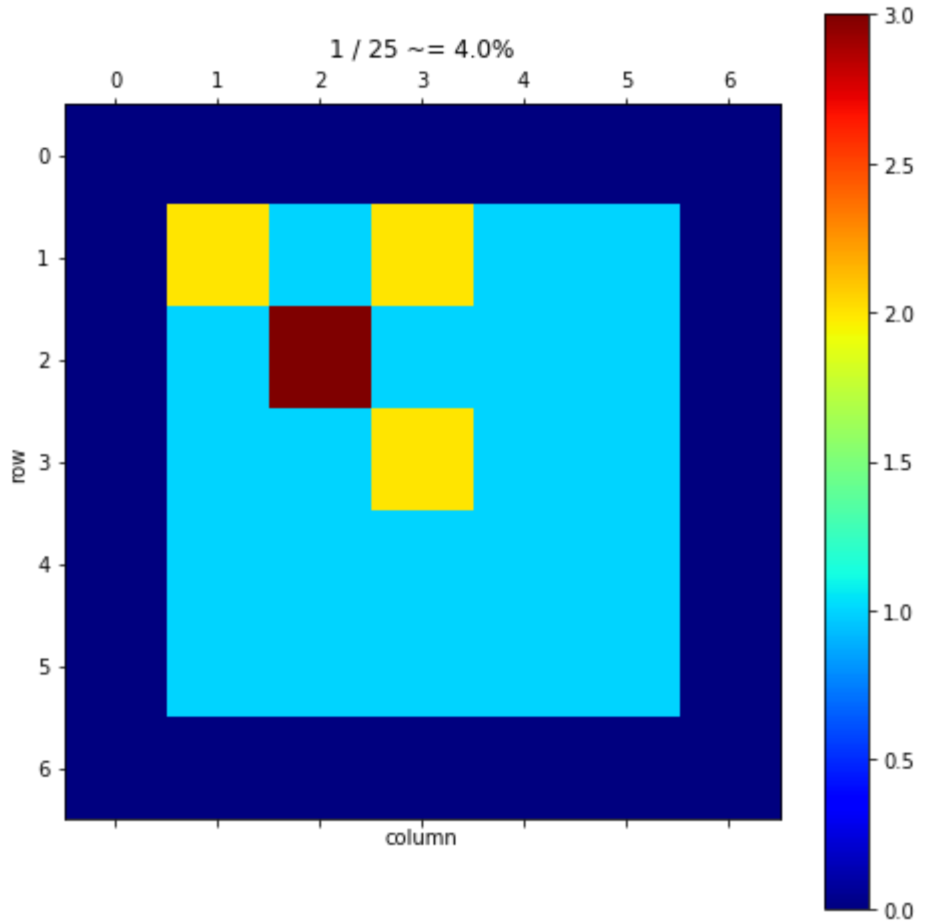


In [32]:
```
show_tm(np.multiply(tm, is_vegetation(w)))
```



In [33]:
```
w = spread_fire_both(w, tm, e)
show_world(w)
summarize_world(w)
```

```
The world has dimensions: 5 x 5
There are 25 cell(s) that have had vegetation in them
There are 21 cell(s) of vegetation that are unburned
There are 3 cell(s) of vegetation on fire
There are 1 cell(s) of vegetation completely burned
```



Simulations

In [34]:
```
def simulate(W0, tree_type, elevs, t_max=None, inplace=False):
    if t_max is None:
        n_max = max(W0.shape)
        t_max = n_max * ((2*n_max-1) // 2)
    W = np.zeros((W0.shape[0], W0.shape[1], 2 if inplace else t_max+1))

    tm = create_transition_matrix(W0, tree_type)

    t_cur = 0
    W[:, :, t_cur] = W0
    for t in range(t_max):
        t_next = (t_cur+1)%2 if inplace else t+1
        W[:, :, t_next] = spread_fire_both(W[:, :, t_cur], tm, elevs)
        if (W[:, :, t_cur] == W[:, :, t_next]).all():
            t_cur = t_next
            break
        t_cur = t_next
    return (W[:, :, t_cur], t) if inplace else W[:, :, :t_cur+1]

def viz(W, t=0):
    show_world(W[:, :, t])
    plt.show()
    print("At time {} (max={})...".format(t, W.shape[2]-1))
    summarize_world(W[:, :, t])

def run_simulation(n, q, tree_type, e, **args):

    w = create_world(n, q)
    w = start_fire(w)

    return simulate(w, tree_type, e, **args)
```

In [35]:
```
def simulate_many(n, q, trials, tree_type, e, **args):
    percent_burned = np.zeros(trials)
    time_to_burn = np.zeros(trials)

    for trial in range(trials):
        W_last, t_last = run_simulation(n, q, tree_type, e, inplace=True
, **args)
        n_trees = count(W_last, is_vegetation)
        n_burnt = count(W_last, is_burned)
        percent_burned[trial] = n_burnt / n_trees if n_trees > 0 else 0.
0
        time_to_burn[trial] = t_last
    return percent_burned, time_to_burn
```

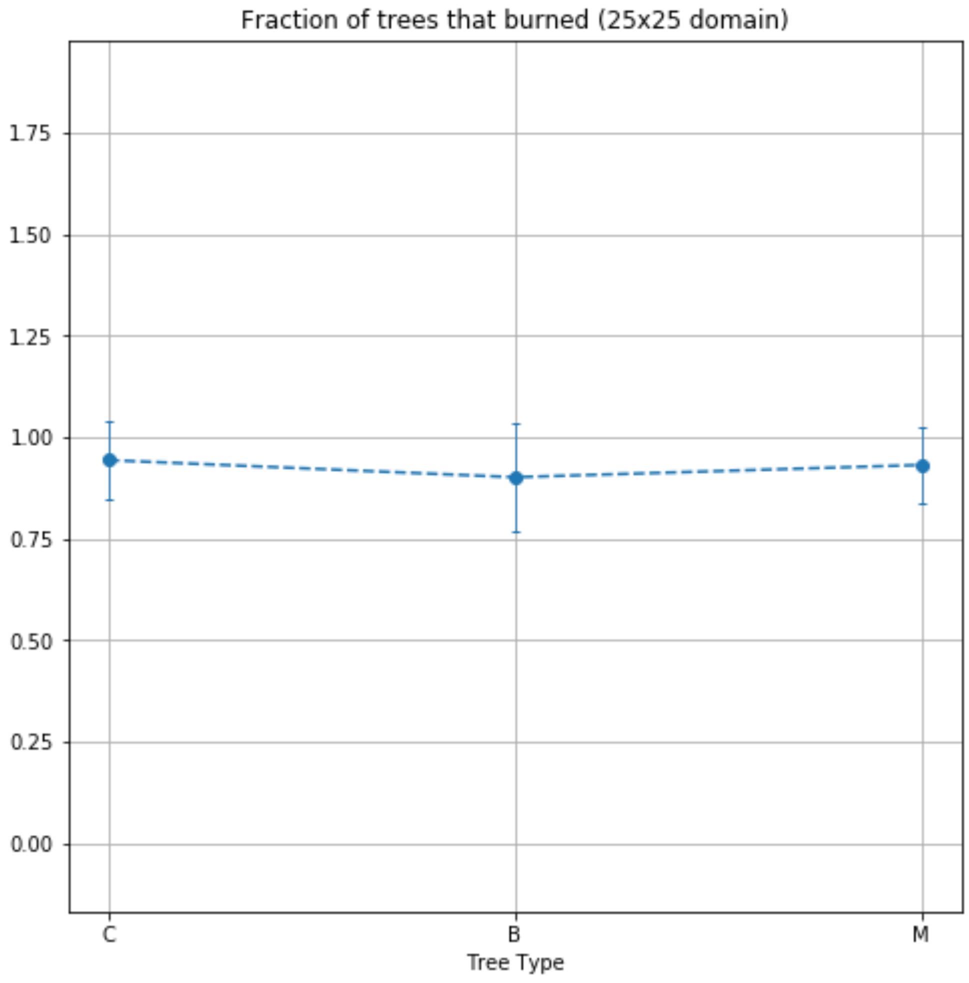
Analyzing effect of tree type

In [36]:
```
n_many = 25
tree_types = ['C', 'B', 'M']

Percentages = np.zeros((len(tree_types), 2))
Times = np.zeros((len(tree_types), 2))

e = create_increasing_elevations(np.zeros((n_many + 2, n_many + 2)))

for k, t in enumerate(tree_types):
    print("Simulating tree type:{}...".format(t))
    percentages, times = simulate_many(n_many, 1, 100, t, e)
    Percentages[k, :] = [percentages.mean(), percentages.std()]
    Times[k, :] = [times.mean(), times.std()]
```

```
Simulating tree type:C...
Simulating tree type:B...
Simulating tree type:M...
```

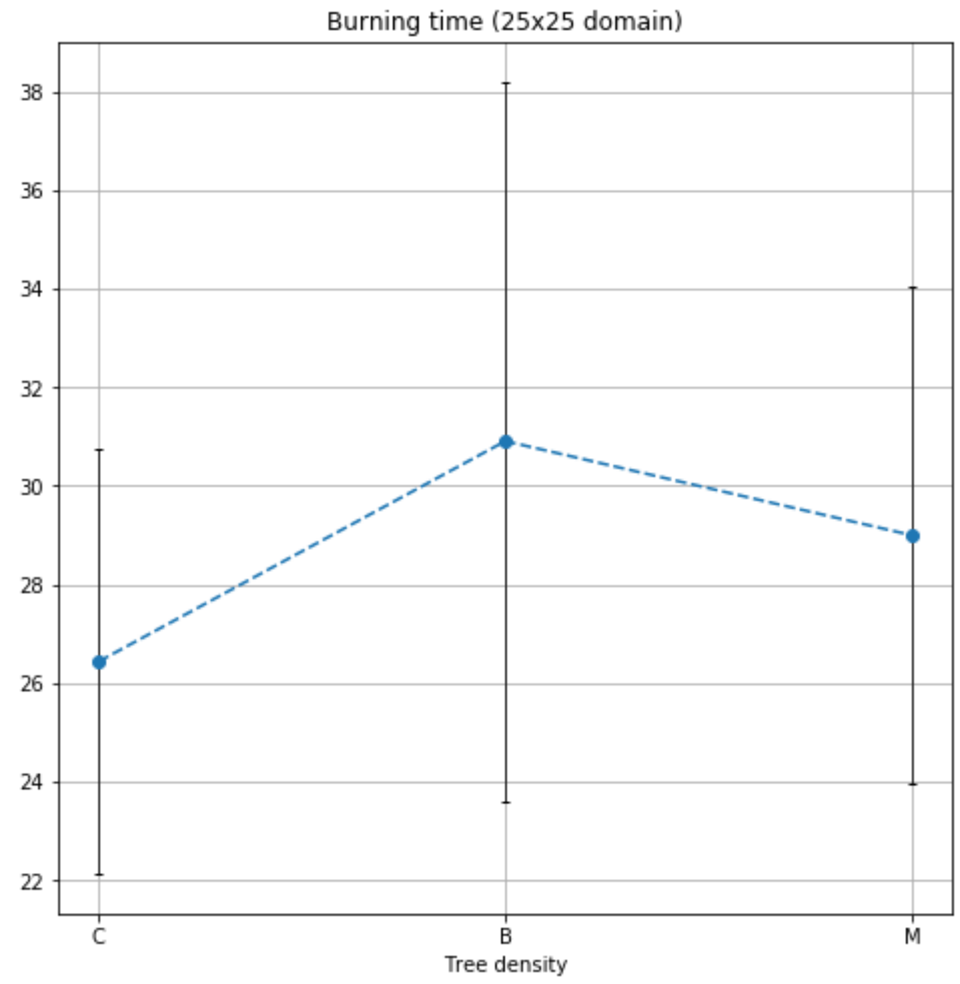
The plot below shows the fraction of trees that burn for each type, however since our forest was of density 1, each tree types is burned completley and thus this graph is not very interesting.

In [37]:
```
plt.figure(figsize=(8, 8))
plt.errorbar(tree_types, Percentages[:, 0], yerr=Percentages[:, 1], fmt=
'o--', elinewidth=0.75, capsize=2);
plt.gca().axis('equal')
plt.xlabel('Tree Type');
plt.title('Fraction of trees that burned ({}x{} domain)'.format(n_many,
n_many));
plt.grid()
```



However, below we can see how the times to burn differ between the tree types. As we would expect, coniferous forests burn the fastest, broadleaf the slowest and mixed somewhere in between

In [38]:
```
plt.figure(figsize=(8, 8))
Z = 1
plt.errorbar(tree_types, Times[:, 0]/Z, yerr=Times[:, 1]/Z, fmt='o--', e
color='black', elinewidth=0.75, capsize=2);
plt.xlabel('Tree density');
plt.title('Burning time ({}x{} domain)'.format(n_many, n_many));
plt.grid()
```



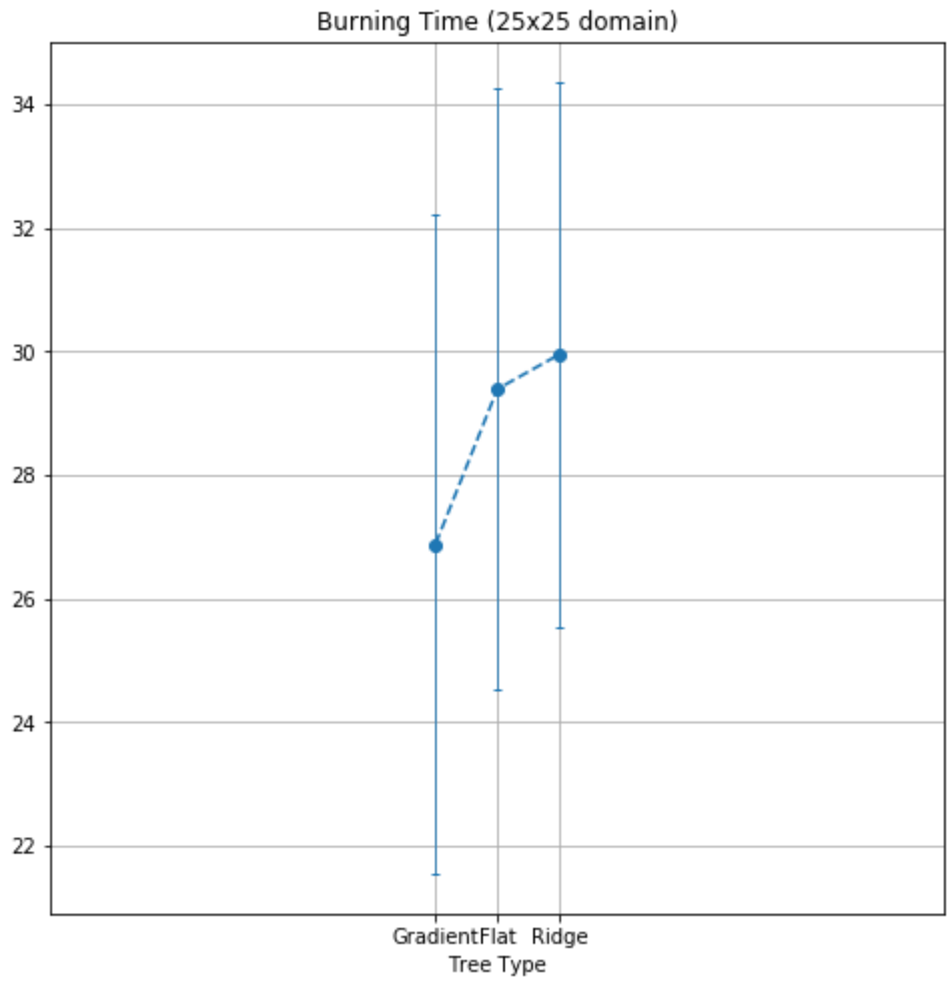
Analyzing effect of elevation

In [39]:
```
n_many = 25

w_base = np.zeros((n_many + 2, n_many + 2))

e_raised = create_increasing_elevations(w_base)
e_flat = w_base
e_ridge = add_middle_ridge(w_base)

elevations = {'Gradient': e_raised, 'Flat': e_flat, 'Ridge': e_ridge}

e_keys = ['Gradient', 'Flat', 'Ridge']

Percentages_2 = np.zeros((len(elevations), 2))
Times_2 = np.zeros((len(elevations), 2))


for index, key in enumerate(e_keys):
    print("Simulating elevation type:{}...".format(key))
    percentages, times = simulate_many(n_many, 1, 100, 'C', elevations[k
ey])
    Percentages_2[index, :] = [percentages.mean(), percentages.std()]
    Times_2[index, :] = [times.mean(), times.std()]
```

```
Simulating elevation type:Gradient...
Simulating elevation type:Flat...
Simulating elevation type:Ridge...
```

As the plot below demonstrates, the gradient elevation map where half of the map is increasing has a lower number of time steps to burn to completion than the flat. The ridge where a middle ridge of elevated height exists also has a lower number of steps than the pure flat one. Since this model is probabilistic, theses differences are subtle and more pronounced in some runs than others.

In [40]:
```
plt.figure(figsize=(8, 8))
plt.errorbar(e_keys, Times_2[:, 0], yerr=Times_2[:, 1], fmt='o--', eline
width=0.75, capsize=2);
plt.gca().axis('equal')
plt.xlabel('Tree Type');
plt.title('Burning Time ({}x{} domain)'.format(n_many, n_many));
plt.grid()
```



In [ ]:

In [ ]:

Wind effect on the fire spreading

In this tutorial, we'll explore the effect of wind (both wind power and direction) on fire spreading.

Probabilistic Spreading considering the wind factor

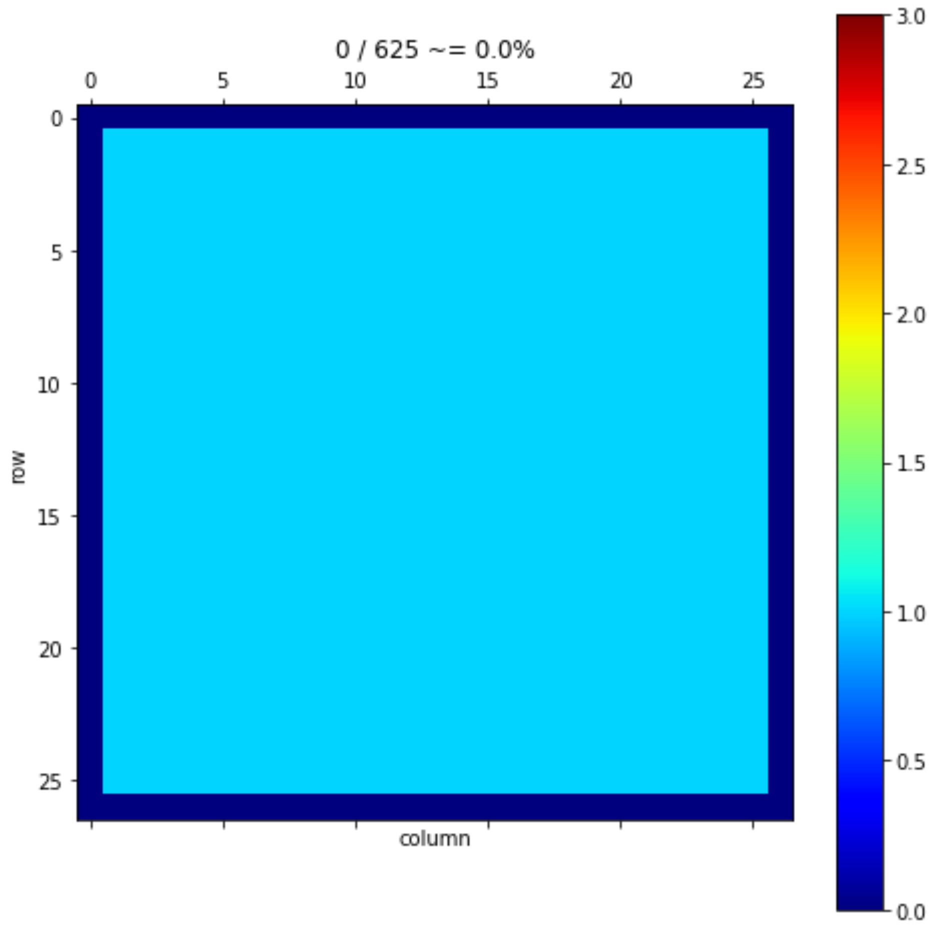
Building on our previous probabilistic model, we take wind into account in this model. To get a perfect forest fire spreading model, both wind power and wind direction and their interaction cannot be ignored. To consider the change of central cell (i, j) after a time period, we need to study the eight neighborhood cells' impact on the central cell.

In real fires, wind helps fuel the flames and cause the fire to grow, rarely having a dampening effect on the fire's growth. Below, we attempt to translate this phenomenon into our model.

We start off by creating a 25 by 25 world and assume the whole forest is burnable.

In [1]:
```
import numpy as np
import scipy as sp
import scipy.sparse
import matplotlib as mpl
import matplotlib.pyplot as plt

empty = 0
unburned = 1
burning = 2
burned = 3
```

In [2]:
```
def create_world(n, q):
    world = np.zeros((n+2, n+2))
    forest = world[1:-1, 1:-1]
    forest[:, :] = np.random.choice([0, 1], p=[1-q, q], size=(n, n))
    return world

def show_world(W, title=None, **args):
    if 'cmap' not in args:
        args['cmap'] = 'jet'
    if 'vmin' not in args and 'vmax' not in args:
        args['vmin'] = 0
        args['vmax'] = 3
    plt.figure(figsize=(8, 8))
    plt.matshow(W, fignum=1, **args)
    plt.xlabel('column')
    plt.ylabel('row')
    plt.colorbar()
    if title is None:
        num_trees = (W > 0).sum()
        if num_trees > 0:
            num_burnt = (W == 3).sum()
            percent = num_burnt / num_trees * 1e2
            title = '{} / {} ~= {:.1f}%'.format(num_burnt, num_trees, pe
rcent)
        else:
            title = ''
    plt.title(title)
    pass

World = create_world(25, 1)
show_world(World)
```

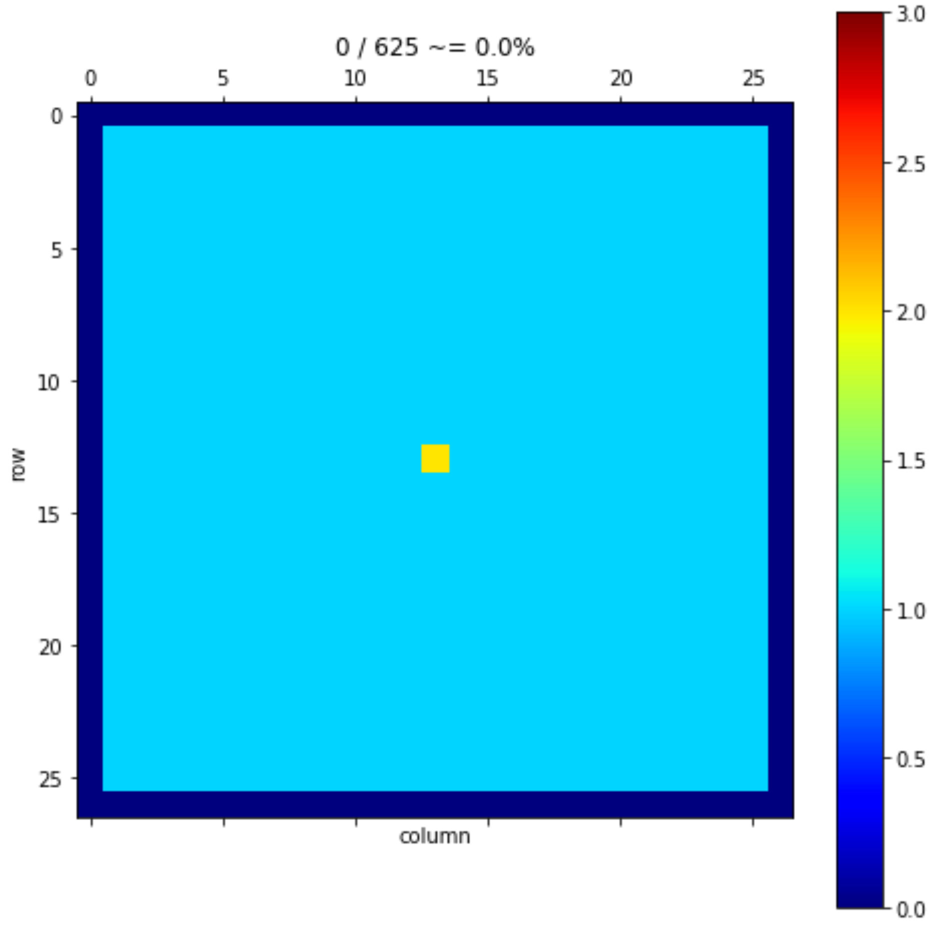

0 / 625 ~= 0.0%
0
5
10
15
20
25
row
column
3.0
2.5
2.0
1.5
1.0
0.5
0.0

In [3]:
```
def is_empty(W):
    return W == empty

def is_vegetation(W):
    return W > empty

def is_unburned(W):
    return W == 1

def is_burning(W):
    return W == 2

def is_burned(W):
    return W == 3

def count(W, cond_fun):
    return cond_fun(W).sum()

def summarize_world(W):
    def suffix(n):
        return (1, "tree") if n == 1 else (n, "trees")
    m, n = W.shape[0]-2, W.shape[1]-2
    n_trees = count(W, is_vegetation)
    n_unburned = count(W, is_unburned)
    n_burning = count(W, is_burning)
    n_burned = count(W, is_burned)

    print("The world has dimensions: {} x {}".format(m, n))
    print("There are {} cell(s) that have had vegetation in them".format
(n_trees))
    print("There are {} cell(s) of vegetation that are unburned".format(
n_unburned))
    print("There are {} cell(s) of vegetation on fire".format(n_burning
))
    print("There are {} cell(s) of vegetation completely burned".format(
n_burned))

summarize_world(World)
```

```
The world has dimensions: 25 x 25
There are 625 cell(s) that have had vegetation in them
There are 625 cell(s) of vegetation that are unburned
There are 0 cell(s) of vegetation on fire
There are 0 cell(s) of vegetation completely burned
```

In [4]:
```
# In order to show the effect of the wind direction on the fire spreadin
g, we set the fire starting point as the middle point of the world

def start_fire(W, m, cells=None):
    # m is true for the middle point, false for the random point
    W_new = W.copy()
    if m == True:
        middle = int(len(World)/2)
        W_new[middle,middle]=2
        return W_new
    else:
        if cells == None:
            F = W[1:-1, 1:-1]
            W_new = W.copy()
            F_new = W_new[1:-1, 1:-1]
            I, J = np.where(is_unburned(F)) # Positions of all trees
            if len(I) > 0:
                k = np.random.choice(range(len(I))) # Index of tree to i
gnite
                i, j = I[k], J[k]
                assert F_new[i, j] == 1, "Attempting to ignite a non-tre
e?"
                F_new[i, j] += 1
        else:
            W_new = W.copy()

            for x,y in cells:
                W_new[x,y] = 2

    return W_new
# Start a fire in the world, middle for True and random for False
World_next = start_fire(World,True)
show_world(World_next)
summarize_world(World_next)
```

```
The world has dimensions: 25 x 25
There are 625 cell(s) that have had vegetation in them
There are 624 cell(s) of vegetation that are unburned
There are 1 cell(s) of vegetation on fire
There are 0 cell(s) of vegetation completely burned
```



Wind factor

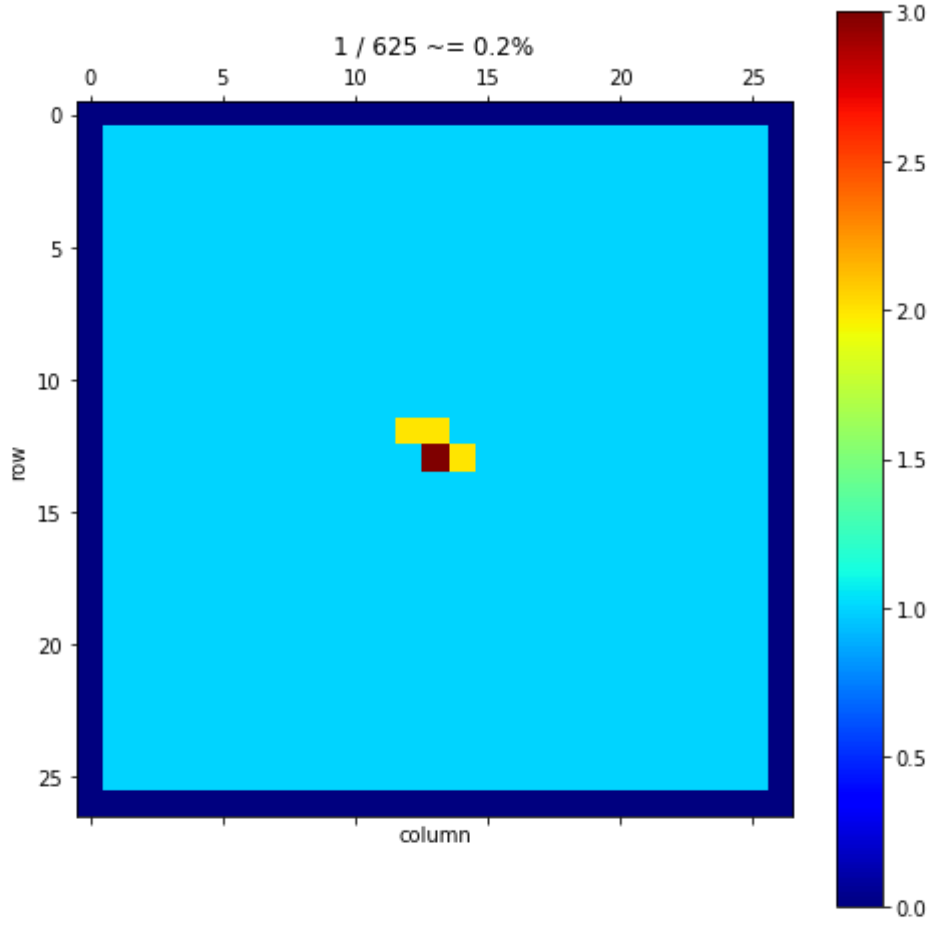
Our wind factor can be decomposed into 2 parts: wind direction and wind power. We combine these two to create a new wind weight that translates to higher probabilities of ignition for cells downwind from a fire.

Here, we have 8 directions for wind, and we assume the wind power is scaled greater than 1. Additionally, we assume the effect of the wind power on the fire spreading is linear.

Note: Wind direction in our model is the direction where wind blows to. e.g. N wind will affect the cell above the burning cell.

In [5]:
```
# Probability of fire spread
fire_spread_prob = 0.3

def spread_fire(W, wind_dir, wind_power):
    W_new = W.copy()
    Vegetation = is_unburned(W)
    Fires = is_burning(W)

    #identify the 8 neighbors of the burning cell
    F1 = Fires[:-2, :-2]
    F2 = Fires[1:-1, :-2]
    F3 = Fires[2:, :-2]
    F4 = Fires[:-2, 1:-1]
    F5 = Fires[2:, 1:-1]
    F6 = Fires[:-2, 2:]
    F7 = Fires[1:-1, 2:]
    F8 = Fires[2:, 2:]

    # Add the wind direction and wind power effect on the probabilibty
    # There are 8 neighbors 1-8, and we consider 8 wind directions N S W
E WN EN WS ES.
    # Then assign different weights to the probability for all 8 neighbo
rs of the cell

    if wind_dir =='S':
        F4 = F4 *wind_power
        F5 = F5 *0
    elif wind_dir =='N':
        F5 = F5 *wind_power
        F4 = F4 *0

    elif wind_dir =='W':
        F7 = F7 *wind_power
        F2 = F2 * 0

    elif wind_dir =='E':
        F2 = F2 *wind_power
        F7 = F7 * 0

    elif wind_dir =='SW':
        F6 = F6 *wind_power
        F3 = F3 *0
    elif wind_dir =='SE':
        F1 = F1 *wind_power
        F8 = F8 *0

    elif wind_dir =='NW':
        F8 = F8 *wind_power
        F1 = F1 *0
    elif wind_dir =='NE':
        F3 = F3 *wind_power
        F6 = F6 *0

    # Spread fire to neighbors
    # First count how many neighbors of a cell are on fire!
    num_neighbors_on_fire = (F1 + F2+F3+ F4+ F5+ F6 + F7+ F8)
    # If there's no vegetation at a certain spot, we don't want to calcu
late anything related to it's neighbors,
    # so lets set those locations to zero
    num_neighbors_on_fire = np.multiply(num_neighbors_on_fire, Vegetatio
n[1:-1, 1:-1].astype(int))

    # Now calcualte the chance of each location doesn't catch fire at th
e next time-step.
    # All probabilities start out as 1.0 - fire_spread_prob
    fire_prob_matrix = np.ones(num_neighbors_on_fire.shape) - fire_sprea
d_prob

    # Because each neighbor independently influences a cell, the total p
robability of not spreading is equal to
    # the (probabilty of one neighbor not spreading) ^ (number of neighb
ors). Taking 1.0 - that value transforms the
    # calculated probability to represent the: probability of the cell c
atching on fire due to any of the neighbors.
    fire_prob_matrix = 1.0 - np.power(fire_prob_matrix, num_neighbors_on
_fire)

    # Again we don't care about a specific cell's value if there's no ve
getation there, so we zero out
    # the locations that don't have vegetation there!
    fire_prob_matrix = np.multiply(fire_prob_matrix, Vegetation[1:-1, 1:
-1].astype(int))

    # Now to use our probabiltiy matrix we've generated, we need to gene
rate a matrix of all random
    # values between 0.0 and 1.0, and compare if the random generated va
lue is less than the probability of catching fire!
    randys = np.random.rand(*fire_prob_matrix.shape)
    new_on_fires = randys < fire_prob_matrix

    # Now we "add" one to each location that used to be "unburned" that
 now will be "burning" since the value for
    # "unburned" is (1) and the value for "burning" is (2), this works o
ut to transition our states perfectly!
    W_new[1:-1, 1:-1] += new_on_fires

    # We still want to extinguish current fires after one time-step, so
 let's go do that
    W_new[1:-1, 1:-1] += Fires[1:-1, 1:-1]

    # Aaaaand we're done!
    return W_new

# Spread one time-step of fire. Then show and summarize the world!
# We can assign the wind direction'NW'and power 3 here.
World_next = spread_fire(World_next,'NW',3)
show_world(World_next)
summarize_world(World_next)
```

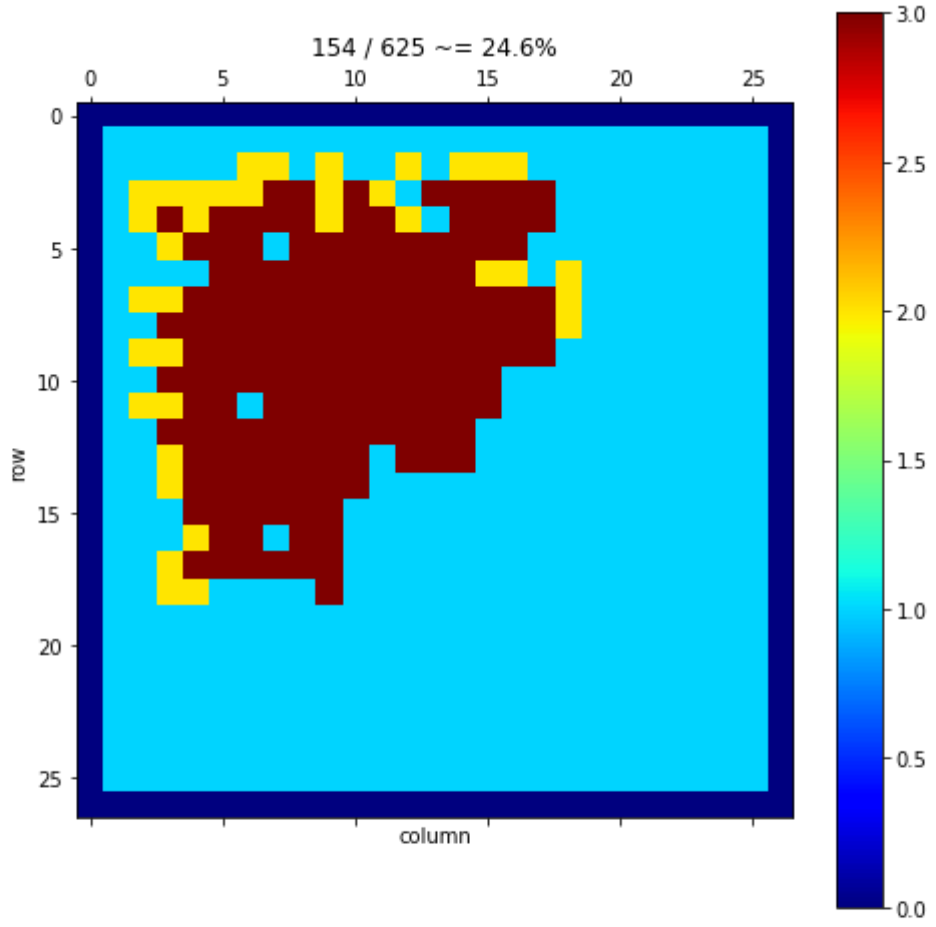
```
The world has dimensions: 25 x 25
There are 625 cell(s) that have had vegetation in them
There are 621 cell(s) of vegetation that are unburned
There are 3 cell(s) of vegetation on fire
There are 1 cell(s) of vegetation completely burned
```



Then we pick a direction 'NW' and a wind power of 3 to confirm that the wind has large impact to the fire spreading. When running simulations, we can see that the left upper corner of the forest is almost on fire.

In [6]:
```
for _ in range(10):
    World_next = spread_fire(World_next,'NW',3)

show_world(World_next)
summarize_world(World_next)
```

```
The world has dimensions: 25 x 25
There are 625 cell(s) that have had vegetation in them
There are 435 cell(s) of vegetation that are unburned
There are 36 cell(s) of vegetation on fire
There are 154 cell(s) of vegetation completely burned
```



Simulation

Below, we run some Monte Carlo simulations as in the first couple of tutorials to see the effect of wind direction and wind speed on our fire's spread.

In [7]:
```
def simulate(W0, wind_dir,wind_power, t_max=None, inplace=False):
    if t_max is None:
        n_max = max(W0.shape)
        t_max = n_max * ((2*n_max-1) // 2)
    W = np.zeros((W0.shape[0], W0.shape[1], 2 if inplace else t_max+1))
    t_cur = 0
    W[:, :, t_cur] = W0
    for t in range(t_max):
        t_next = (t_cur+1)%2 if inplace else t+1
        W[:, :, t_next] = spread_fire(W[:, :, t_cur],wind_dir,wind_power
)
        if (W[:, :, t_cur] == W[:, :, t_next]).all():
            t_cur = t_next
            break
        t_cur = t_next
    return (W[:, :, t_cur], t) if inplace else W[:, :, :t_cur+1]

def viz(W, t=0):
    show_world(W[:, :, t])
    plt.show()
    print("At time {} (max={})...".format(t, W.shape[2]-1))
    summarize_world(W[:, :, t])

def run_simulation(n, q, wind_dir, wind_power, **args):
    w = create_world(n, q)
    # start the fire randomly in the forest
    w = start_fire(w,False)

    return simulate(w, wind_dir,wind_power, **args)
```

In [8]:
```
def simulate_many(n, q,trials, wind_dir, wind_power):
    percent_burned = np.zeros(trials)
    time_to_burn = np.zeros(trials)
    for trial in range(trials):
        W_last, t_last = run_simulation(n, q, wind_dir,wind_power,inplac
e=True)
        n_trees = count(W_last, is_vegetation)
        n_burnt = count(W_last, is_burned)
        percent_burned[trial] = n_burnt / n_trees if n_trees > 0 else 0.
0
        time_to_burn[trial] = t_last
    return percent_burned, time_to_burn
```

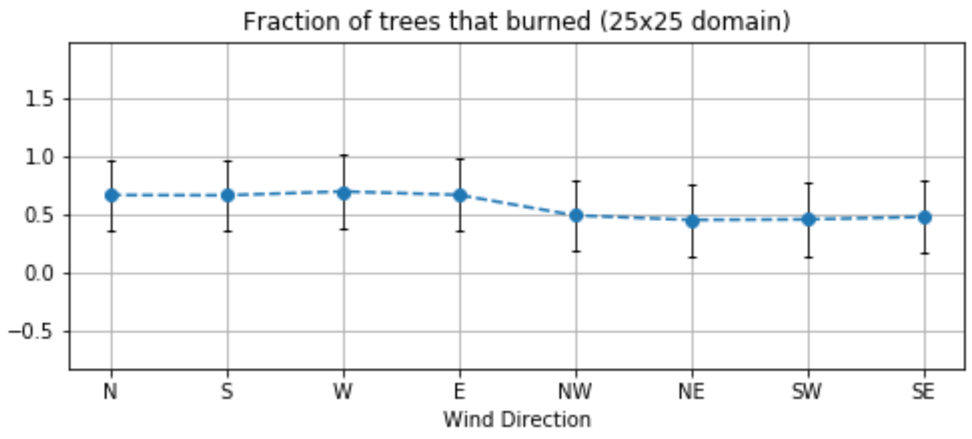
Wind direction analysis

In [9]:
```python
n_many = 25
wind_dir = ['N','S', 'W', 'E', 'NW','NE','SW','SE']
Percentages = np.zeros((len(wind_dir), 2))
Times = np.zeros((len(wind_dir), 2))

for k, w in enumerate(wind_dir):
    print("Simulating wind direction:{}...".format(w))
    # iterate through wind_dir and set the wind power as constant 5
    percentages, times = simulate_many(n_many, 1, 100, w, 5)
    Percentages[k, :] = [percentages.mean(), percentages.std()]
    Times[k, :] = [times.mean(), times.std()]
```

```
Simulating wind direction:N...
Simulating wind direction:S...
Simulating wind direction:W...
Simulating wind direction:E...
Simulating wind direction:NW...
Simulating wind direction:NE...
Simulating wind direction:SW...
Simulating wind direction:SE...
```

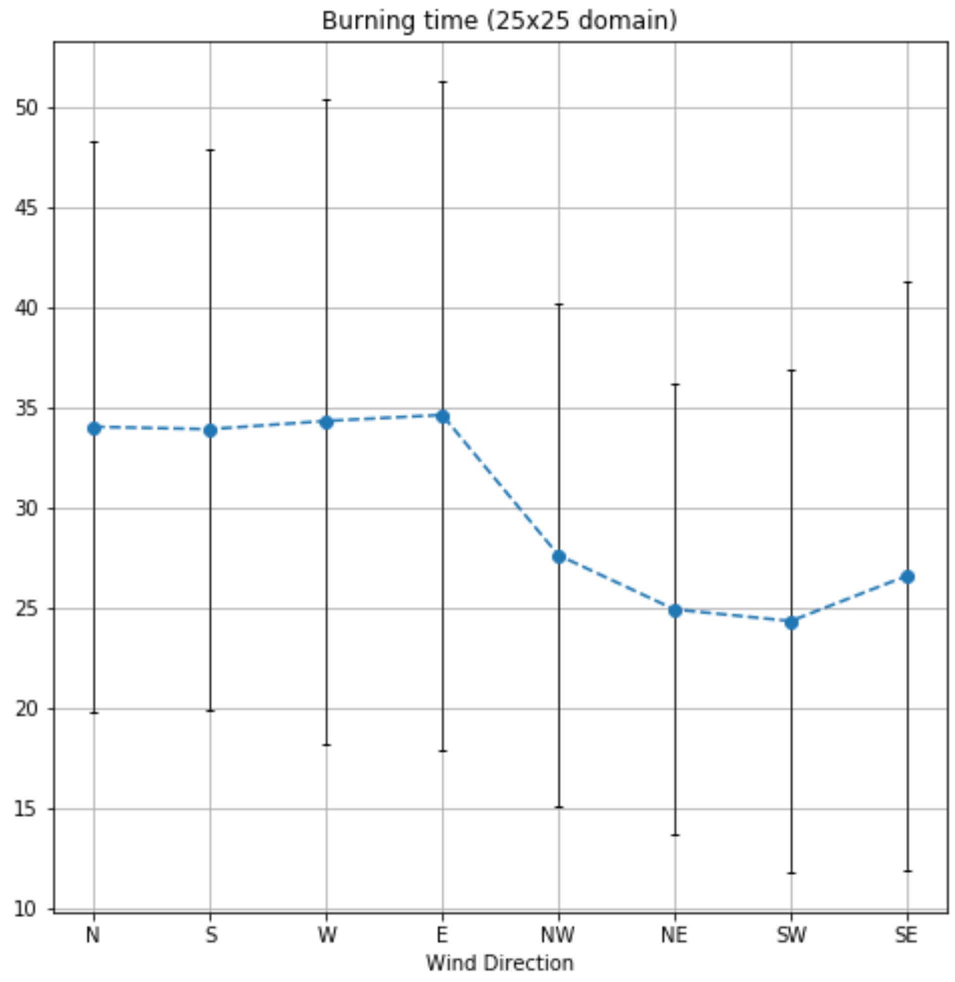
From the plot below, we observe that the fraction of trees burned varies among 8 wind directions. The directions of *NW, NE, SW, SE* wind are less than *N, S, W, E*. This could be attributed to the format of our CA. For instance, in the row or column order directions of *N, S, W, E*, the wind is blowing along one of the axis of our 2d square world and thus has higher exposure to cells than going diagonally to one of the corners of our grid world.

In [10]:
```python
plt.figure(figsize=(8, 3))
plt.errorbar(wind_dir, Percentages[:, 0], yerr=Percentages[:, 1], fmt='o
--', ecolor='black', elinewidth=0.75, capsize=2);
plt.gca().axis('equal')
plt.xlabel('Wind Direction');
plt.title('Fraction of trees that burned ({}x{} domain)'.format(n_many,
n_many));
plt.grid()
```



Below we can see the burning time of the different worlds. Although the diagonal directions appear to burn through the forest faster, they were also consistently burning less forest before going out as our plot above revealed and thus it make sense for these fires to take less time.

In [11]:
```
plt.figure(figsize=(8, 8))
Z = 1
plt.errorbar(wind_dir, Times[:, 0]/Z, yerr=Times[:, 1]/Z,
             fmt='o--', ecolor='black', elinewidth=0.75, capsize=2);
plt.xlabel('Wind Direction');
plt.title('Burning time ({}x{} domain)'.format(n_many, n_many));
plt.grid()
```



Wind Power Analysis

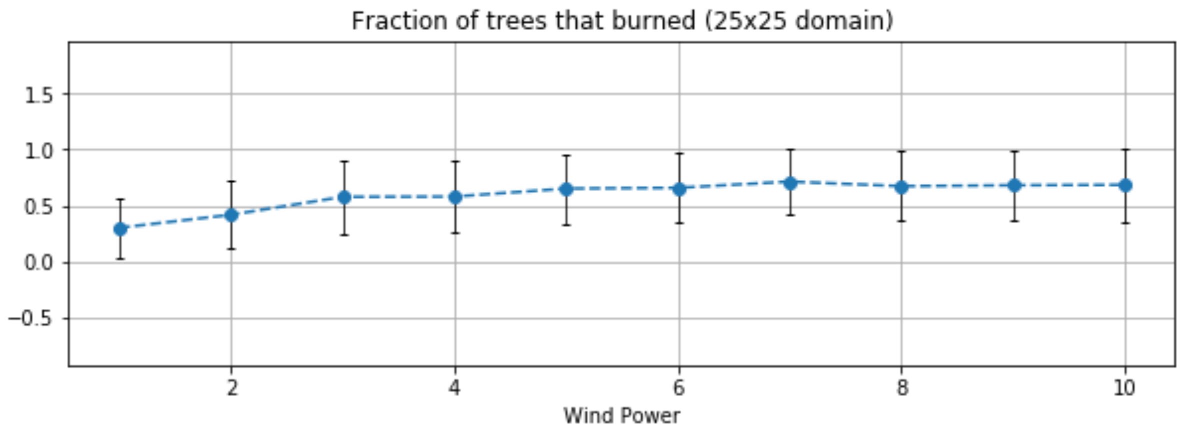
Here we want to see how the wind power will affect the fire spreading. We set the wind direction as 'N' for testing.

In [12]:
```
n_many = 25
wind_power = np.linspace(1, 10, 10)
Percentages = np.zeros((len(wind_power), 2))
Times = np.zeros((len(wind_power), 2))

for k, w in enumerate(wind_power):
    print("Simulating wind direction:{}...".format(w))
    # iterate through wind power and set the wind direction as 'N'
    percentages, times = simulate_many(n_many, 1, 100, 'N', w)
    Percentages[k, :] = [percentages.mean(), percentages.std()]
    Times[k, :] = [times.mean(), times.std()]
```

```
Simulating wind direction:1.0...
Simulating wind direction:2.0...
Simulating wind direction:3.0...
Simulating wind direction:4.0...
Simulating wind direction:5.0...
Simulating wind direction:6.0...
Simulating wind direction:7.0...
Simulating wind direction:8.0...
Simulating wind direction:9.0...
Simulating wind direction:10.0...
```

From the plot, we can see that higher wind power will contribute to a higher fraction of the forest being burned.

In [13]:
```
plt.figure(figsize=(10, 3))
plt.errorbar(wind_power, Percentages[:, 0], yerr=Percentages[:, 1], fmt=
'o--', ecolor='black', elinewidth=0.75, capsize=2);
plt.gca().axis('equal')
plt.xlabel('Wind Power');
plt.title('Fraction of trees that burned ({}x{} domain)'.format(n_many,
n_many));
plt.grid()
```



In [ ]:

Tutorial 4: Forest Fires with All Factors

So far, we have discussed the effect of terrain, vegetation type, and wind on fire spread. We will now combine all of these methods in order to get a robust model of fire movement and then explore the effects a common anti-fire tactic of firelines to see how the fire's behaviour changes.

First, we'll rewrite some of the basics from the first tutorial to set up our model.

In [1]:
```
import numpy as np
import scipy as sp
import scipy.sparse
import matplotlib as mpl
import matplotlib.pyplot as plt
import itertools

empty = 0
unburned = 1
burning = 2
burned = 3
```

In [2]:
```
def create_world(n, q):
    world = np.zeros((n+2, n+2))
    forest = world[1:-1, 1:-1]
    forest[:, :] = np.random.choice([0, 1], p=[1-q, q], size=(n, n))
    return world

def show_world(W, title=None, **args):
    if 'cmap' not in args:
        args['cmap'] = 'jet'
    if 'vmin' not in args and 'vmax' not in args:
        args['vmin'] = 0
        args['vmax'] = 3
    plt.figure(figsize=(8, 8))
    plt.matshow(W, fignum=1, **args)
    plt.xlabel('column')
    plt.ylabel('row')
    plt.colorbar()
    if title is None:
        num_trees = (W > 0).sum()
        if num_trees > 0:
            num_burnt = (W == 3).sum()
            percent = num_burnt / num_trees * 1e2
            title = '{} / {} ~= {:.1f}%'.format(num_burnt, num_trees, pe
rcent)
        else:
            title = ''
    plt.title(title)
    pass

def is_empty(W):
    return W == empty

def is_vegetation(W):
    return W > empty

def is_unburned(W):
    return W == 1

def is_burning(W):
    return W == 2

def is_burned(W):
    return W == 3

def count(W, cond_fun):
    return cond_fun(W).sum()

def summarize_world(W):
    def suffix(n):
        return (1, "tree") if n == 1 else (n, "trees")
    m, n = W.shape[0]-2, W.shape[1]-2
    n_trees = count(W, is_vegetation)
    n_unburned = count(W, is_unburned)
    n_burning = count(W, is_burning)
    n_burned = count(W, is_burned)
    print("The world has dimensions: {} x {}".format(m, n))
    print("There are {} cell(s) that have had vegetation in them".format
(n_trees))
    print("There are {} cell(s) of vegetation that are unburned".format(
n_unburned))
    print("There are {} cell(s) of vegetation on fire".format(n_burning
))
    print("There are {} cell(s) of vegetation completely burned".format(
n_burned))

def start_fire(W, cells=None):
    W_new = W.copy()

    if cells == None:
        F = W[1:-1, 1:-1]
        W_new = W.copy()
        F_new = W_new[1:-1, 1:-1]
        I, J = np.where(is_unburned(F)) # Positions of all trees
        if len(I) > 0:
            k = np.random.choice(range(len(I))) # Index of tree to ignit
e
            i, j = I[k], J[k]
            assert F_new[i, j] == 1, "Attempting to ignite a non-tree?"
            F_new[i, j] += 1
    else:
        W_new = W.copy()

        for x,y in cells:
            W_new[x,y] = 2

    return W_new
```

Forest Type

Our Forest Type Functions are below.

In [3]:
```
def create_transition_matrix(world, tree_type):
    if tree_type not in ['C', 'B', 'M']:
        raise ValueError('Tree type must be coniferous (\'C\'), Broadlea
f (\'B\'), or mixed (\'M\')')

    probs = {'C': .9, 'B': .7}

    mat = np.ones(world.shape)

    if tree_type is 'M':
        mat = np.random.rand(*world.shape)

        c_index = mat < .5
        b_index = mat >= .5

        mat[c_index] = probs['C']
        mat[b_index] = probs['B']

    else:
        mat *= probs[tree_type]

    return mat

def show_tm(W, title=None, **args):
    if 'cmap' not in args:
        args['cmap'] = 'terrain'
    if 'vmin' not in args and 'vmax' not in args:
        args['vmin'] = 0
        args['vmax'] = 1
    plt.figure(figsize=(8, 8))
    plt.matshow(W, fignum=1, **args)
    plt.xlabel('column')
    plt.ylabel('row')
    plt.colorbar()

    title = 'Types of trees (darker are more likely to burn)'

    plt.title(title)
    pass
```

Terrain

Our terrain type functions copied over as well.

In [4]:
```python
# create random elevations between 0 and 10
def create_random_elevations(world):
    elevs = np.random.randint(0, 10, size=world.shape)

    return elevs

# add ridge of elevation 10 down the middle of the map
def add_middle_ridge(elevs):

    elevs[:, elevs.shape[1]//2] = 10

    return elevs

# create a map with 0 elevation on one half and increasing on the other
def create_increasing_elevations(world):
    elevs = np.zeros(world.shape)
    num_tiers = elevs.shape[1]//2

    tiers = np.linspace(1, num_tiers, num_tiers)

    r = num_tiers

    if elevs.shape[1] % 2:
        r += 1

    for i in range(1, r):
        elevs[:, num_tiers + i] = np.full((1, elevs.shape[0]), i)

    return elevs

def show_elevs(elevs, title=None, **args):
    if 'cmap' not in args:
        args['cmap'] = 'terrain'
    if 'vmin' not in args and 'vmax' not in args:
        args['vmin'] = 0
        args['vmax'] = np.max(elevs)
    plt.figure(figsize=(8, 8))
    plt.matshow(elevs, fignum=1, **args)
    plt.xlabel('column')
    plt.ylabel('row')
    plt.colorbar()

    title = 'Elevation'

    plt.title(title)
    pass

def elev_transition_prob(elevs_mat):
    kernel = np.matrix([[-1,-2,-1], [0,0,0], [1,2,1]])

    vert, row = np.gradient(elevs_mat)

    c0 = np.roll(row, 1, axis=1)
    c0[:, 0] = 0

    c1 = np.roll(vert, 1, axis=0)
    c1[0, :] = 0

    c = np.maximum(c0, c1)

    diff = c.max() - c.min()
    if not diff: diff = 1

    c = (c - c.min()) / diff

    return c
```

Combined Fire Spread

Below, we combine the elevation, terrain types, and wind.

1. First we take in wind as in tutorial 3 and factor it into our `num_neighbors_on_fire` and `fire_prob_matrix`
2. Next, we combine the effects of tree type with our `transition_matrix`
3. Finally, we take elevation into account with the elevation transition probability matrix (`elev_p`). Since this matrix also represents probabilities of fire spreading, we combine this and our earlier `fire_prob_matrix` by weighing each by half and adding them. More advanced models could look at different ratios between these factor and their effect on fire spread.

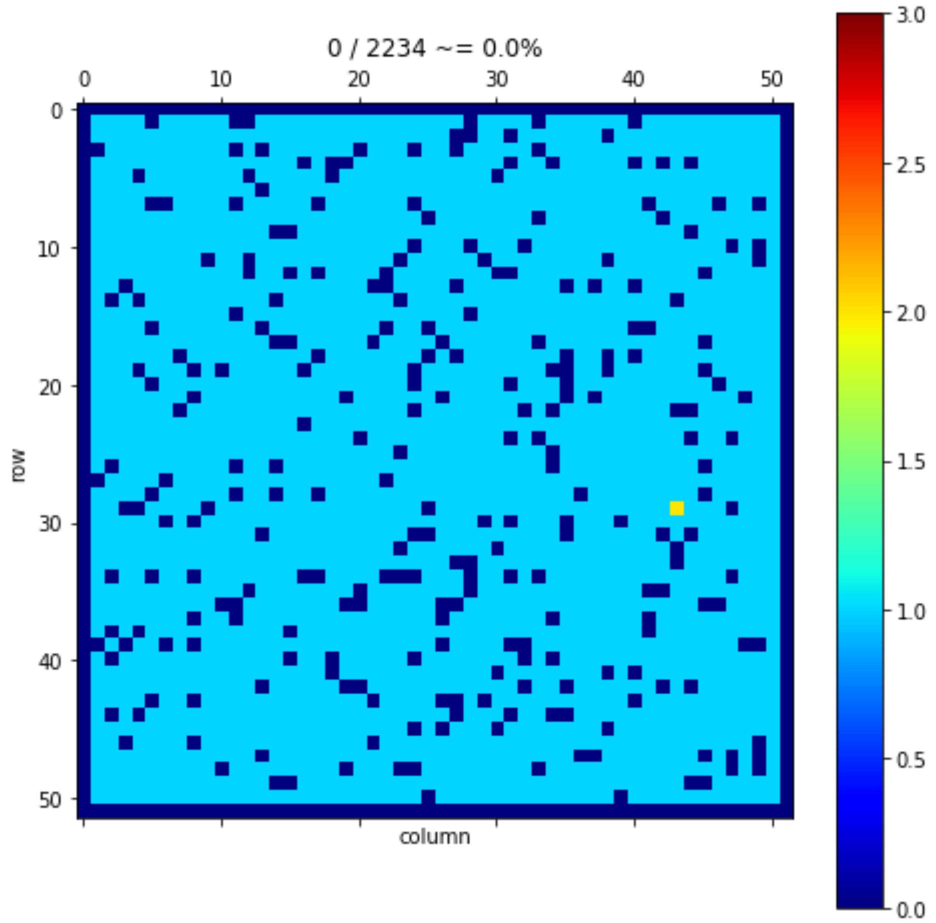
In [5]:
```
def spread_fire_all(W, transition_matrix, elevs, wind_args):
    wind_dir, wind_power = wind_args
    W_new = W.copy()
    Vegetation = is_unburned(W)
    Fires = is_burning(W)

    # 1) WIND
    F1 = Fires[:-2, :-2]
    F2 = Fires[1:-1, :-2]
    F3 = Fires[2:, :-2]
    F4 = Fires[:-2, 1:-1]
    F5 = Fires[2:, 1:-1]
    F6 = Fires[:-2, 2:]
    F7 = Fires[1:-1, 2:]
    F8 = Fires[2:, 2:]

    if wind_dir =='S':
        F4 = F4 *wind_power
        F5 = F5 *0
    elif wind_dir =='N':
        F5 = F5 *wind_power
        F4 = F4 *0

    elif wind_dir =='W':
        F7 = F7 *wind_power
        F2 = F2 * 0

    elif wind_dir =='E':
        F2 = F2 *wind_power
        F7 = F7 * 0

    elif wind_dir =='SW':
        F6 = F6 *wind_power
        F3 = F3 *0
    elif wind_dir =='SE':
        F1 = F1 *wind_power
        F8 = F8 *0

    elif wind_dir =='NW':
        F8 = F8 *wind_power
        F1 = F1 *0
    elif wind_dir =='NE':
        F3 = F3 *wind_power
        F6 = F6 *0

    num_neighbors_on_fire = (F1 + F2 + F3 + F4 + F5 + F6 + F7 + F8)

    num_neighbors_on_fire = np.multiply(num_neighbors_on_fire, Vegetatio
n[1:-1, 1:-1].astype(int))

    # 2) TREE TYPE
    fire_prob_matrix = np.ones(num_neighbors_on_fire.shape) - transition
_matrix[1:-1, 1:-1]
    fire_prob_matrix = 1.0 - np.power(fire_prob_matrix, num_neighbors_on
_fire)
```

```
    # 3) ELEVATION
    elev_p = elev_transition_prob(elevs)
    elev_p = elev_p[1:-1, 1:-1]
    elev_i = Vegetation[1:-1, 1:-1] \
                        & (
                              Fires[:-2, :-2]  | Fires[1:-1, :-2] | Fire
s[2:, :-2]
                            | Fires[:-2, 1:-1] |                    Fire
s[2:, 1:-1]
                            | Fires[:-2, 2:]   | Fires[1:-1, 2:]  | Fire
s[2:, 2:]
                           )

    e = np.multiply(elev_p, elev_i)
    fire_prob_matrix = .5 * fire_prob_matrix + .5 * e

    # probabilistic transition
    fire_prob_matrix = np.multiply(fire_prob_matrix, Vegetation[1:-1, 1:
-1].astype(int))

    # print('Final prob matrix', fire_prob_matrix)
    # print('Final prob max', np.max(fire_prob_matrix))

    randys = np.random.rand(*fire_prob_matrix.shape)
    new_on_fires = randys < fire_prob_matrix

    W_new[1:-1, 1:-1] += new_on_fires

    W_new[1:-1, 1:-1] += Fires[1:-1, 1:-1]

    return W_new
```

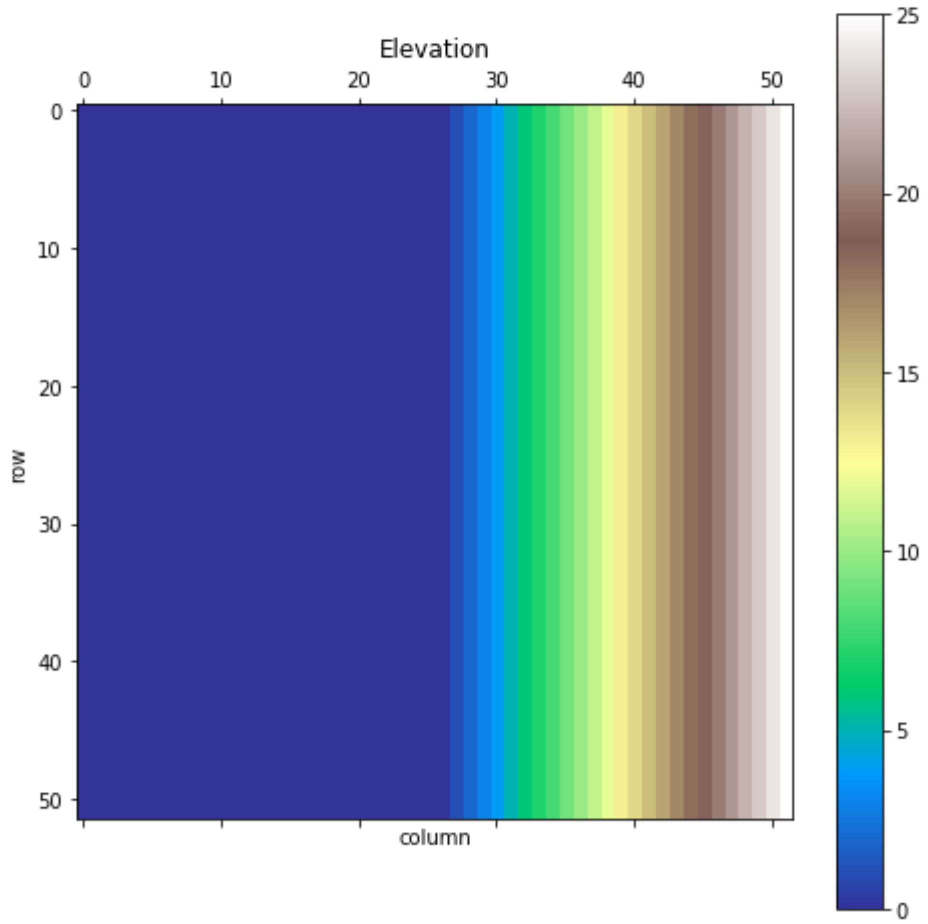
Testing

In [6]:
```
w = create_world(50, .9)
w = start_fire(w)
tm = create_transition_matrix(w, 'M')
e = create_increasing_elevations(w)

show_world(w)
summarize_world(w)
```

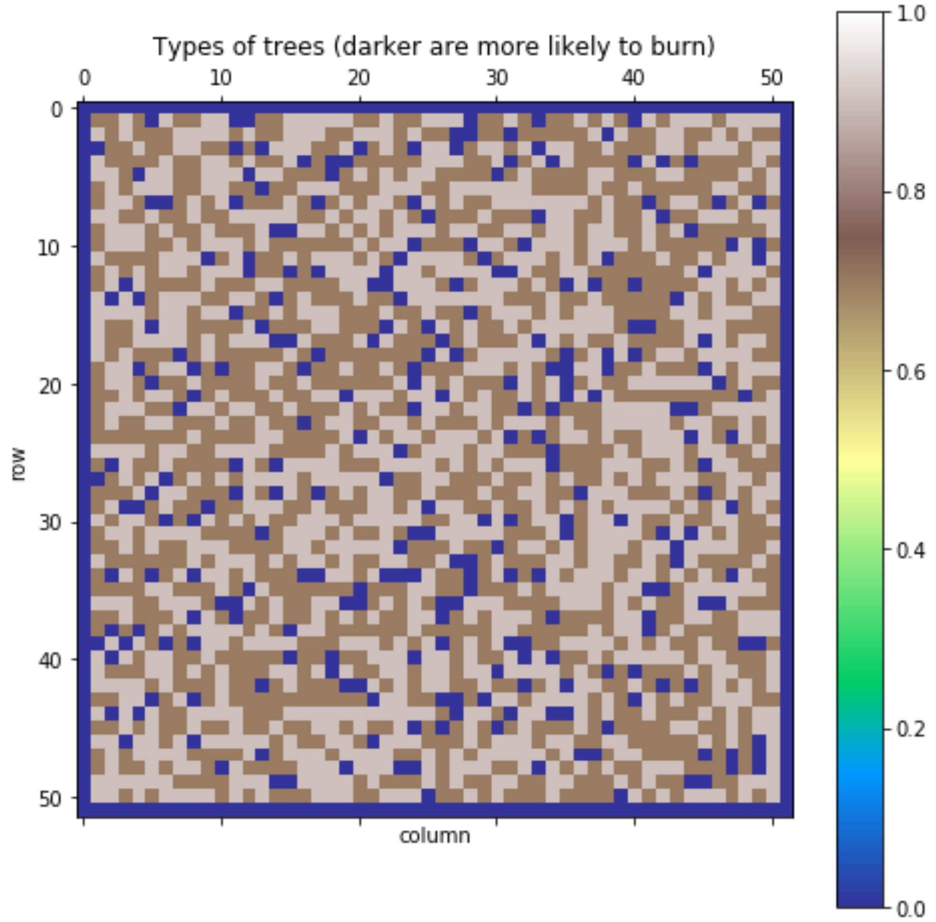
```
The world has dimensions: 50 x 50
There are 2234 cell(s) that have had vegetation in them
There are 2233 cell(s) of vegetation that are unburned
There are 1 cell(s) of vegetation on fire
There are 0 cell(s) of vegetation completely burned
```



In [7]:
```
show_elevs(e)
```

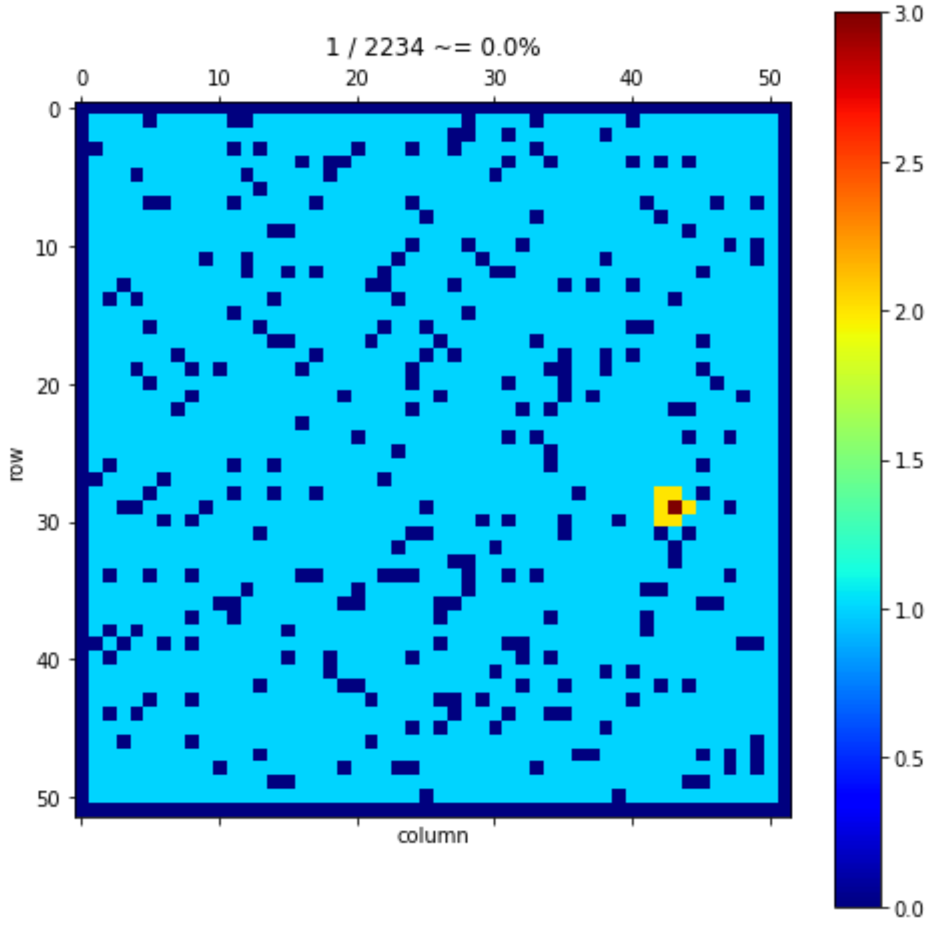


In [8]:
```
show_tm(np.multiply(tm, is_vegetation(w)))
```



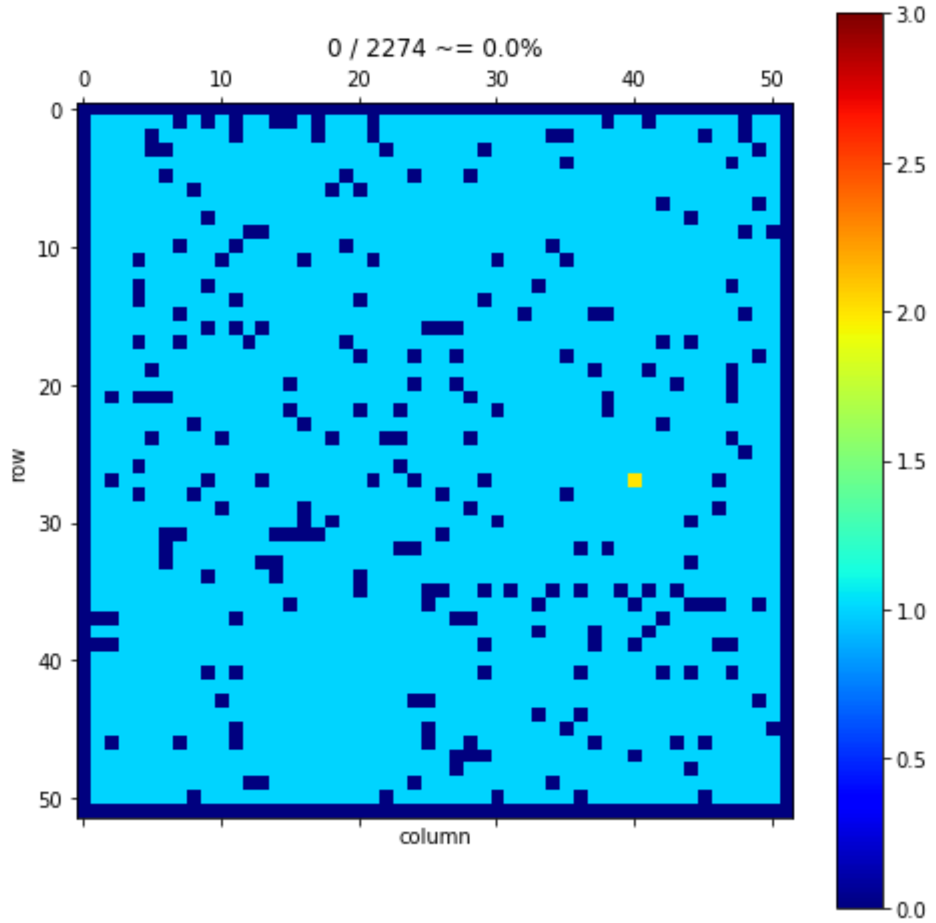
In [9]:
```
w = spread_fire_all(w, tm, e, ('E', 10))
show_world(w)
summarize_world(w)
```

```
The world has dimensions: 50 x 50
There are 2234 cell(s) that have had vegetation in them
There are 2227 cell(s) of vegetation that are unburned
There are 6 cell(s) of vegetation on fire
There are 1 cell(s) of vegetation completely burned
```



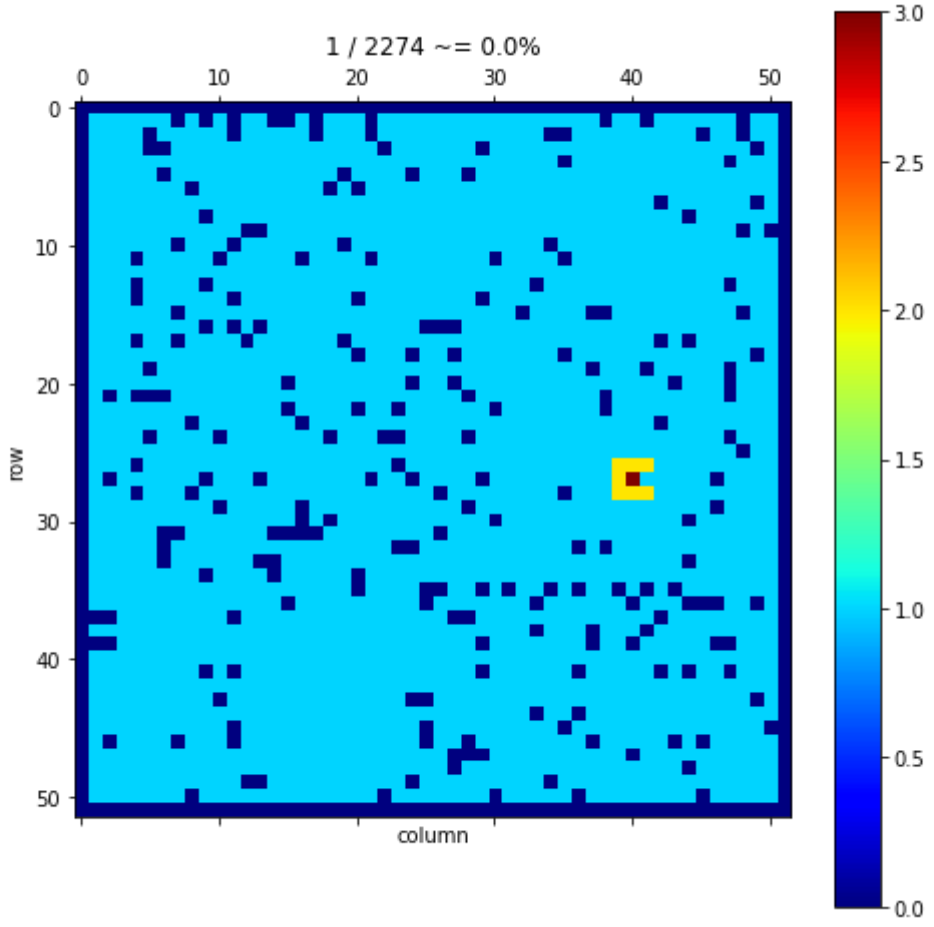
In [10]:
```
w = create_world(50, .9)
w = start_fire(w)
tm = create_transition_matrix(w, 'M')
e = create_increasing_elevations(w)

show_world(w)
summarize_world(w)
```

```
The world has dimensions: 50 x 50
There are 2274 cell(s) that have had vegetation in them
There are 2273 cell(s) of vegetation that are unburned
There are 1 cell(s) of vegetation on fire
There are 0 cell(s) of vegetation completely burned
```



```
In [11]: w = spread_fire_all(w, tm, e, ('W', 10))
         show_world(w)
         summarize_world(w)
```

```
The world has dimensions: 50 x 50
There are 2274 cell(s) that have had vegetation in them
There are 2266 cell(s) of vegetation that are unburned
There are 7 cell(s) of vegetation on fire
There are 1 cell(s) of vegetation completely burned
```



Simulations for All Factors

In [12]:
```
def simulate(W0, tree_type, elevs, wind_args, t_max=None, inplace=False
):
    if t_max is None:
        n_max = max(W0.shape)
        t_max = n_max * ((2*n_max-1) // 2)
    W = np.zeros((W0.shape[0], W0.shape[1], 2 if inplace else t_max+1))

    tm = create_transition_matrix(W0, tree_type)

    t_cur = 0
    W[:, :, t_cur] = W0
    for t in range(t_max):
        t_next = (t_cur+1)%2 if inplace else t+1
        W[:, :, t_next] = spread_fire_all(W[:, :, t_cur], tm, elevs, win
d_args)
        if (W[:, :, t_cur] == W[:, :, t_next]).all():
            t_cur = t_next
            break
        t_cur = t_next
    return (W[:, :, t_cur], t) if inplace else W[:, :, :t_cur+1]

def viz(W, t=0):
    show_world(W[:, :, t])
    plt.show()
    print("At time {} (max={})...".format(t, W.shape[2]-1))
    summarize_world(W[:, :, t])

def run_simulation(n, q, tree_type, e, wind_args, **args):

    w = create_world(n, q)
    w = start_fire(w)

    return simulate(w, tree_type, e, wind_args, **args)
```

In [13]:
```
def simulate_many(n, q, trials, tree_type, e, wind_args, **args):
    percent_burned = np.zeros(trials)
    time_to_burn = np.zeros(trials)

    for trial in range(trials):
        W_last, t_last = run_simulation(n, q, tree_type, e, wind_args, i
nplace=True, **args)
        n_trees = count(W_last, is_vegetation)
        n_burnt = count(W_last, is_burned)
        percent_burned[trial] = n_burnt / n_trees if n_trees > 0 else 0.
0
        time_to_burn[trial] = t_last
    return percent_burned, time_to_burn
```

Analyzing effect of different parameter combos

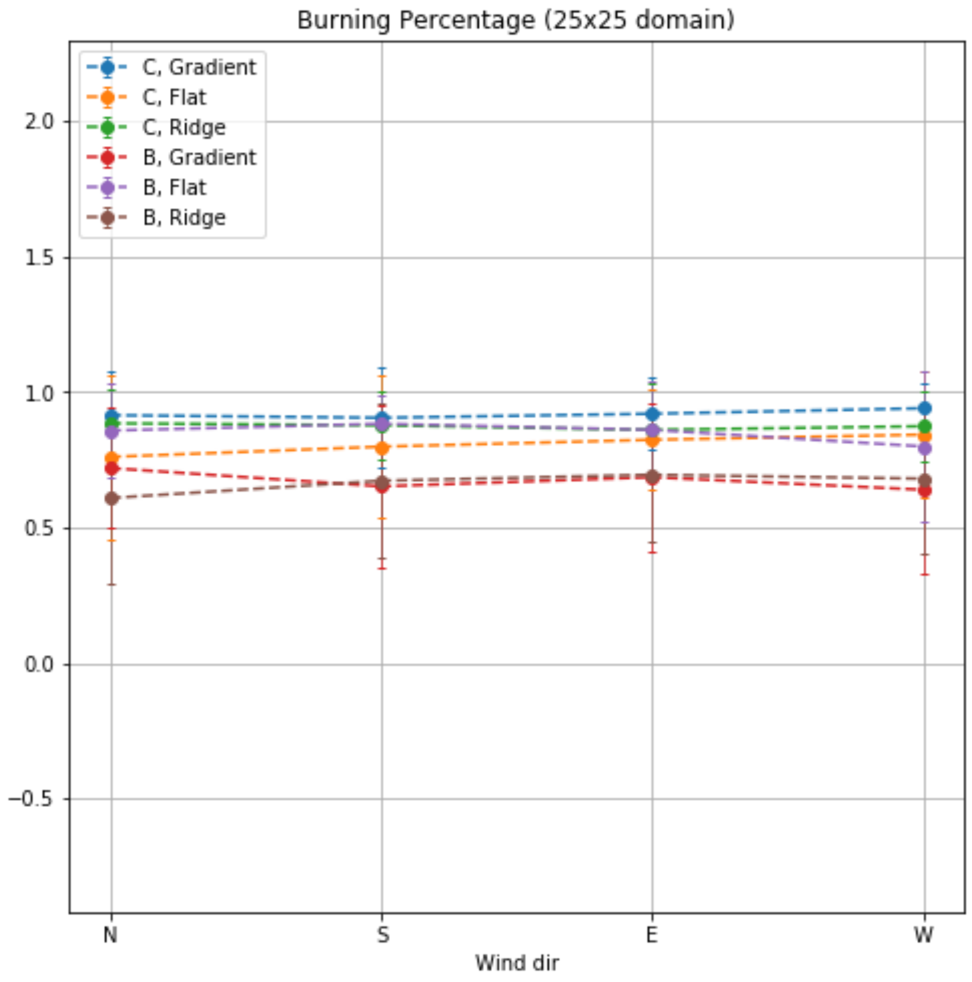
In [14]:
```
n_many = 25

w_base = np.zeros((n_many + 2, n_many + 2))

tree_types = ['C', 'B']

e_raised = create_increasing_elevations(w_base)
e_flat = w_base
e_ridge = add_middle_ridge(w_base)

elevations = {'Gradient': e_raised, 'Flat': e_flat, 'Ridge': e_ridge}
e_keys = ['Gradient', 'Flat', 'Ridge']

wind_dir = ['N','S', 'E', 'W']

Percentages = np.zeros((len(wind_dir), len(e_keys), len(tree_types), 2))
Times = np.zeros((len(wind_dir), len(e_keys), len(tree_types), 2))

for i_w, w in enumerate(wind_dir):
    for i_e, e_key in enumerate(e_keys):
        for i_t, t in enumerate(tree_types):

            print("Simulating args: w={}, e={}, t={}...".format(w, e_key
, t))

            percentages, times = simulate_many(n_many, 1, 100, t, elevat
ions[e_key], (w, 10))
            Percentages[i_w, i_e, i_t, :] = [percentages.mean(), percent
ages.std()]
            Times[i_w, i_e, i_t, :] = [times.mean(), times.std()]
```

```
Simulating args: w=N, e=Gradient, t=C...
Simulating args: w=N, e=Gradient, t=B...
Simulating args: w=N, e=Flat, t=C...
Simulating args: w=N, e=Flat, t=B...
Simulating args: w=N, e=Ridge, t=C...
Simulating args: w=N, e=Ridge, t=B...
Simulating args: w=S, e=Gradient, t=C...
Simulating args: w=S, e=Gradient, t=B...
Simulating args: w=S, e=Flat, t=C...
Simulating args: w=S, e=Flat, t=B...
Simulating args: w=S, e=Ridge, t=C...
Simulating args: w=S, e=Ridge, t=B...
Simulating args: w=E, e=Gradient, t=C...
Simulating args: w=E, e=Gradient, t=B...
Simulating args: w=E, e=Flat, t=C...
Simulating args: w=E, e=Flat, t=B...
Simulating args: w=E, e=Ridge, t=C...
Simulating args: w=E, e=Ridge, t=B...
Simulating args: w=W, e=Gradient, t=C...
Simulating args: w=W, e=Gradient, t=B...
Simulating args: w=W, e=Flat, t=C...
Simulating args: w=W, e=Flat, t=B...
Simulating args: w=W, e=Ridge, t=C...
Simulating args: w=W, e=Ridge, t=B...
```

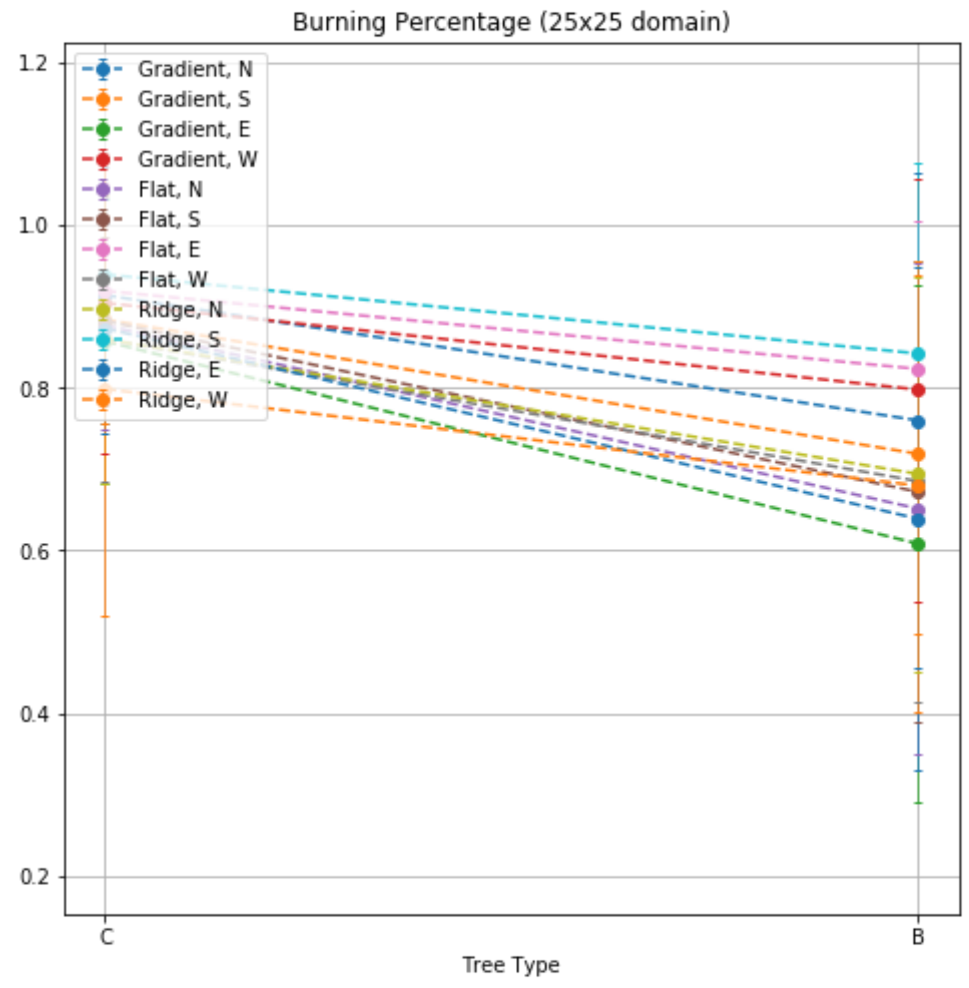
Graphs

The following three charts let us examine these different combinations of Wind direction, tree type, and elevation. Interestingly, we see the same trends from our earlier analysis still hold true when isolated from the other factors. For instance, coniferous forests burn at a higher rate than broadleaf (graph 2) and forests with a gradient elevation burn at a higher rate than flat forests (graph 3). Similar graphs could be created for time analysis but is left to the user for exploration.

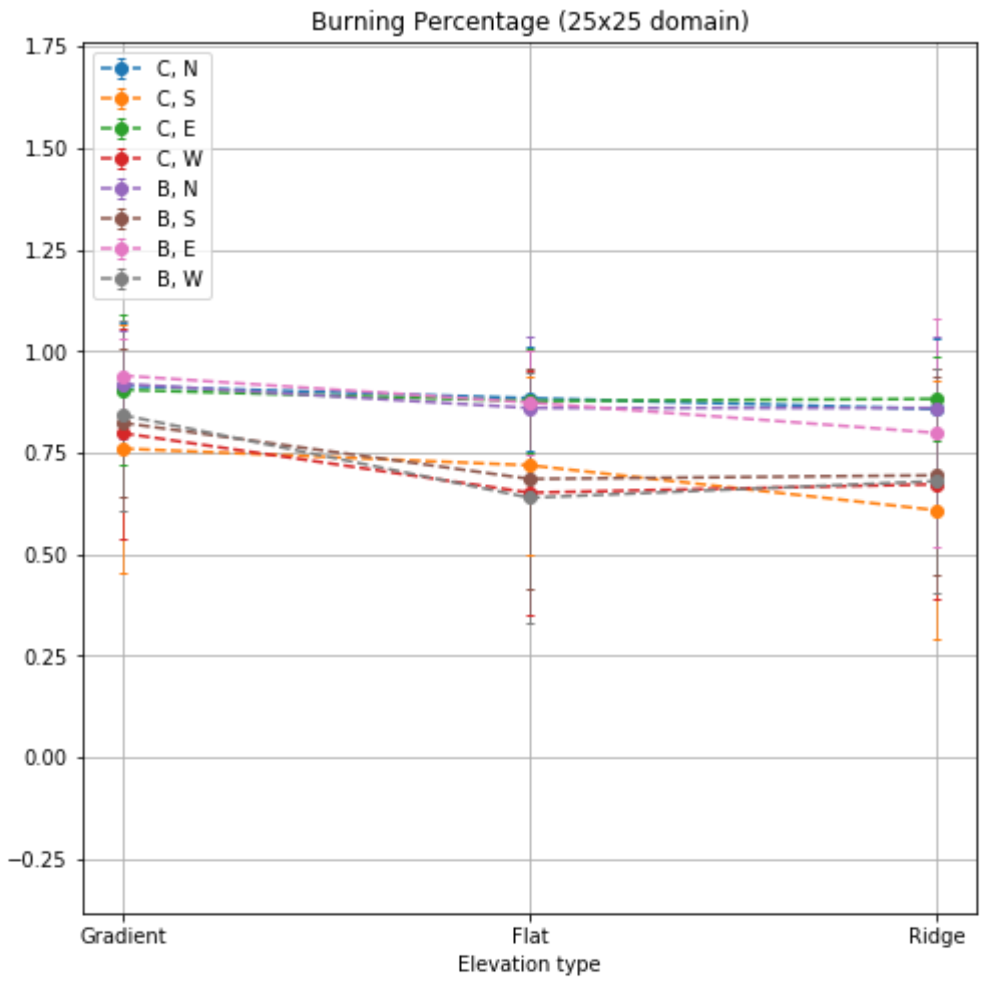
In [15]:
```
plt.figure(figsize=(8, 8))

for i in range(Percentages.shape[1]):
    for j in range(Percentages.shape[2]):
        plt.errorbar(wind_dir, Percentages[:, i, j, 0], yerr=Percentages
[:, i, j, 1], fmt='o--', elinewidth=0.75, capsize=2);


plt.gca().axis('equal')
plt.xlabel('Wind dir');
plt.title('Burning Percentage ({}x{} domain)'.format(n_many, n_many));
plt.legend(map(', '.join, itertools.chain(itertools.product(e_keys, tree
_types), itertools.product(tree_types, e_keys))), loc='upper left')
plt.grid()
```



```
In [16]: plt.figure(figsize=(8, 8))

         for i in range(Percentages.shape[0]):
             for j in range(Percentages.shape[1]):
                 plt.errorbar(tree_types, Percentages[i, j, :, 0], yerr=Percentag
         es[i, j, :, 1], fmt='o--', elinewidth=0.75, capsize=2);


         plt.gca().axis('equal')
         plt.xlabel('Tree Type');
         plt.title('Burning Percentage ({}x{} domain)'.format(n_many, n_many));
         plt.legend(map(', '.join, itertools.chain(itertools.product(wind_dir, e_
         keys), itertools.product(e_keys, wind_dir))), loc='upper left')
         plt.grid()
```



In [17]:
```
plt.figure(figsize=(8, 8))

for i in range(Percentages.shape[0]):
    for j in range(Percentages.shape[2]):
        plt.errorbar(e_keys, Percentages[i, :, j, 0], yerr=Percentages[i
, :, j, 1], fmt='o--', elinewidth=0.75, capsize=2);


plt.gca().axis('equal')
plt.xlabel('Elevation type');
plt.title('Burning Percentage ({}x{} domain)'.format(n_many, n_many));
plt.legend(map(', '.join, itertools.chain(itertools.product(wind_dir, tr
ee_types), itertools.product(tree_types, wind_dir))), loc='upper left')
plt.grid()
```



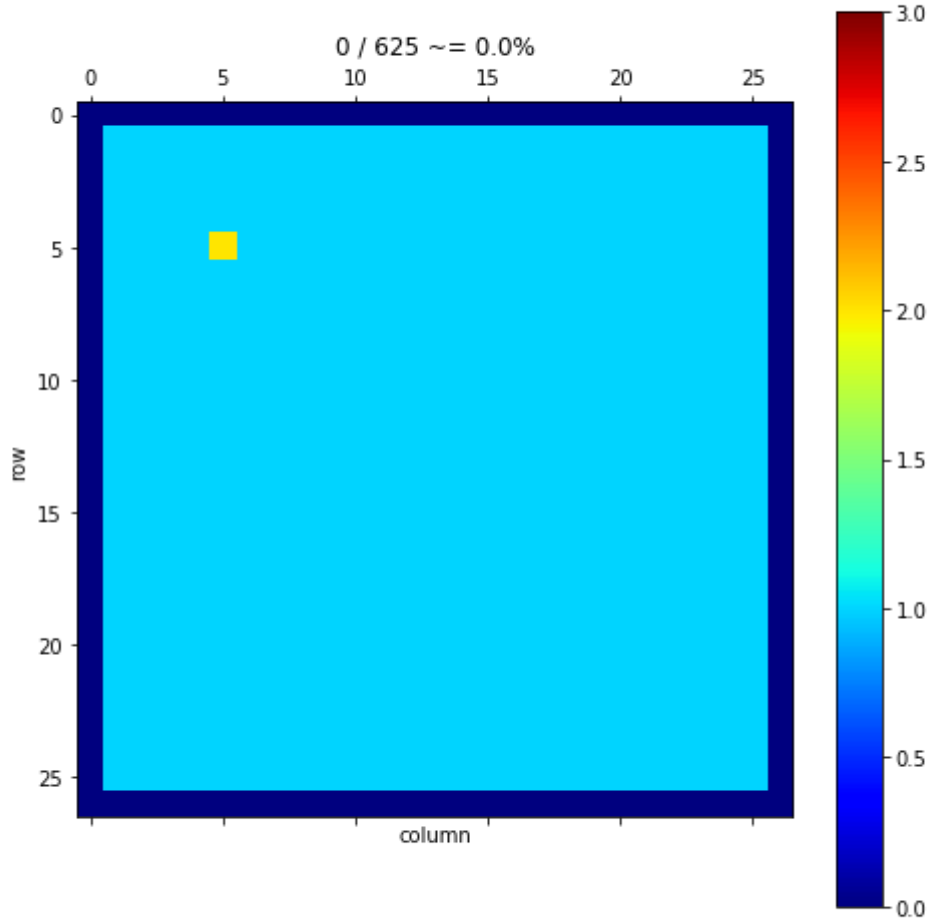
Fire Line

Let's have some fun with this simulator we've built shall we?

In real life, forest fires tend to grow out of hand very quickly. There are several techniques that are able to help combat the spread. One of these is creating what is known as a Fire Line. Fire Lines are areas that have been cleared of vegetation, whether through manual labor or controlled burns, put in strategic spots so that the fire would have a hard time bridging that gap. It is a useful thing to simulate because this could help fire fighters see where to put a fire line to best slow down fire spreading. Below we'll walk through how to simulate a fire line.

First, let's make our world. In our case we're going to make our world have size `25x25` , and start a fire at `(5,5)` For the sake of this demonstration, our fire lines will only be straight, and have a maximum line length of `15` . Let's start by importing the `interact` package from `ipywidgets` and also set our initial parameters. Let's also show our world just so we can visualize it better

In [18]:
```python
from ipywidgets import interact

world_size = 25
fire_start = (5,5)
max_line_length = 15
rand_seed = 42

# Coniferous Trees, Gradient elevation, East wind with strength 10
tree_type = 'C'
e_raised = create_increasing_elevations(w_base)
wind_dir = 'E'
wind_str = 10

np.random.seed(rand_seed)
W = create_world(world_size, 1.0)
W_next = start_fire(W, [fire_start])
show_world(W_next)
```



Now let's write the code that will help us visualize a vertical fire line. Setting the seed is necessary because every time a variable is changed interactively, `fireLineivizVert` is called again. Without a seed, the world generated would constantly change. The `fLine...` sliders decide the location of the fire line, and the `t` slider determines which time step frame to show! Go ahead, play around with the sliders and see what happens as you change t!

```
In [19]: def fireLineivizVert(t=0, fLineStartX=0, fLineStartY=0, fLineLength=0):
             np.random.seed(rand_seed)
             W = create_world(world_size, 1.0)

             W[fLineStartY:fLineStartY+fLineLength, fLineStartX] = empty

             W_result = simulate(start_fire(W, [fire_start]), tree_type, e_raised
         , (wind_dir, wind_str))
             if t < W_result.shape[2]:
                 viz(W_result, t);
             else:
                 viz(W_result, W_result.shape[2] - 1)

         interact(fireLineivizVert, t=(0, 40), fLineStartX=(1,world_size), fLineS
         tartY=(1,world_size+1-max_line_length), fLineLength=(0,max_line_length))

Out[19]: <function __main__.fireLineivizVert(t=0, fLineStartX=0, fLineStartY=0,
         fLineLength=0)>
```

Why stop at vertical fire lines, lets also make a horizontal fire line simulator!

```
In [20]: def fireLineivisHoriz(t=0, fLineStartX=0, fLineStartY=0, fLineLength=0):
             np.random.seed(rand_seed)
             W = create_world(world_size, 1.0)

             W[fLineStartY, fLineStartX:fLineStartX+fLineLength] = empty

             W_result = simulate(start_fire(W, [fire_start]), tree_type, e_raised
         , (wind_dir, wind_str))
             if t < W_result.shape[2]:
                 viz(W_result, t);
             else:
                 viz(W_result, W_result.shape[2] - 1)

         interact(fireLineivisHoriz, t=(0, 40), fLineStartX=(1,world_size+1-max_l
         ine_length), fLineStartY=(1,world_size), fLineLength=(0,max_line_length
         ))

Out[20]: <function __main__.fireLineivisHoriz(t=0, fLineStartX=0, fLineStartY=0,
         fLineLength=0)>
```

Conclusion

Through these tutorials, we have been able to explore a variety of cellular automata models ranging from very basic to rather complex. We hope you have been able to see how different factors play into the spreading of fires in forests and how some basic things like fire lines can be used to shape the spread of a fire and curtail its effects.

Those interested in simulating fires based on real forests could hopefully utilize our model with empirical data on tree types and elevations to get a sense of how fires could spread in real forests in the world.

In [ ]:
```

```

How we split our work:

Everyone helped setup the basic cellular automata structure, as well as all the research done.

Michael: Finished notebook 1, polishing up basic cellular automata and added probabilistic fire spreading. Also implemented Fire Lines in notebook 4, and helped a tiny bit with combining all the spreading types into notebook 4.

Will Epperson: Worked on notebook 2, detailing how fire spread is affected by vegetation and terrain elevation. Combined all the different types of fire spreading into the final notebook along with Jiayi.

Jiayi Ye: Worked on notebook 3, dealing with how wind strength and direction affects fire spread. Also as listed above, helped combine all types of spreading into notebook 4.