Cellular Automata Tutorial: Conway’s Game of Life Model

Team 62:

Brian Glowniak and Joel Katz

Github Repository Link:

https://github.gatech.edu/bglowniak3/CX-4230-Project-1

Cellular Automata and Game of Life

This 3-part tutorial examines the use of cellular automata as a simple yet powerful modeling technique for discrete systems. We utilize John Conway's Game of Life, a well-known example of this approach, as a vehicle to first explore the basic concepts, then to investigate methods of analyzing and expanding upon it.

Tutorial Contents

Scripts

Throughout this tutorial, we will occasionally make reference to scripts. All code for this tutorial can be found within our Github repo at https://github.gatech.edu/bglowniak3/CX-4230-Project-1/tree/master/scripts. The interested reader is invited to play around with these scripts to gain further insight into CA and Game of Life.

Informal References

Introduction to the Modeling and Analysis of Complex Systems - Hiroki Sayama, C

A New Kind of Science - Stephen Wolfram https://www.wolframscience.com/nks/

Contributors

This project was developed by Brian Glowniak and Joel Katz, two CS undergraduate students at Georgia Tech, for CX 4230/CSE 6730 Computer Simulation during Spring 2019.

Part 1: Introduction To Cellular Automata

In the first part of this tutorial, we introduce the concept of a cellular automaton (CA), one of the first frameworks formulated to model complex systems, and explore how a seemingly simple schema can powerfully portray behavior in nonlinear dynamic systems. We then introduce our case study, Game of Life, which we will use throughout the extent of this tutorial to reinforce the concepts discussed.

1.1 What are CA?

Originally devised by mathematicians John von Neumann and Stanislaw Ulam in the 1940s, a cellular automaton consists of a regular *n x n* grid of cells, where each cell represents an individual *"automaton."* In computer science and mathematics, the term *automaton* refers to a machine that consists primarily of an internal state that changes as a function of its inputs.

CA are temporally and spatially discrete, which means that the simulation updates in a series of separate timesteps through interactions between distinct spatially distributed cells.

In order to further discuss the framework, we first need to build up a foundation of some of the concepts and definitions behind it.

1.1.a States and Transitions

The core of any cellular automaton is the set of states that belongs to every cell. This finite, pre-defined set includes the states that a cell can take on over the course of a CA, and each state represents some real-world component of the problem that the model is trying to tackle.

States can be categorized as either *quiescent* or *non-quiescent*. A quiescent state remains in the same state if all of its neighbors are also in the same quiescent state - these are sometimes referred to as *vacuums*. Non-quiescent states are also known as *active* states, and these are what drive the behavior of the model.

In a CA, each cell/automaton is essentially a dynamic variable that changes its state over time. The *configuration*, or *generation*, of a CA is the state of every cell at a given point in time, and the model progresses over these configurations via transition rules that occur over discrete time steps.

These transitions are defined by a *state transition function*, which performs an update on a cell's state based on its current state and the states of surrounding cells. This function is defined as:

$$s_{t+1}(x) = F(s_t(x_0), s_t(x_1), s_t(x_2), \dots, s_t(x_n))$$

where $s_t(x)$ returns the state of a given cell x at a given time t. The set $N_x = \{x_0, x_1, x_2, \dots, x_n\}$ represents the neighborhood of x. The exact neighborhood depends on the specific problem, but it is some combination of cells directly adjacent to x. Two common paradigms are the von Neumann neighborhood, which consists of the upper, lower, left, and right cells, and the Moore neighborhood, which is the same as the von Neumann with the addition of the cells along the diagonals.

Figure 1.1: A diagram demonstrating a von Neumann neighborhood (left) and a Moore neighborhood (right). Image from Sayama ch. 11, pg. 187

An essential assumption of a CA is that the same state transition function, state set, and neighborhood rule are applied to all spatial locations.

1.1.b Boundaries

Since CA also have a spatial component, boundary conditions must be defined to explain behavior at the edges of the model. The four categories are:

1. **No Boundaries**: the space is infinite and filled with the quiescent state unless otherwise specified.
2. **Periodic Boundaries**: the space is finite, but the boundaries wrap around, creating a ring in a 1D case, or a torus in a 2D case.
3. **Cut-off Boundaries**: the space has fixed borders with no neighbors beyond those borders.
4. **Fixed Boundaries**: the cells at the edge of the space have a fixed state that will never change.

1.1.c Classes of Cellular Automata

In the 1980s, computer scientist Stephen Wolfram published a book titled *A New Kind of Science* in which he presents and discusses various forms of simple computational models, including CA. Within this book, Wolfram defined four broad classes into which cellular automata can be divided. These definitions relate to the long-term behavior of a cellular automaton as it evolves over time:

1. **Class 1**: Almost all initial patterns evolve into a single uniform state. Behavior is simple.

2. **Class 2**: Initial patterns can evolve into many different stable or oscillating states. These states consist of a set of simple structures.
3. **Class 3**: Initial patterns evolve in a pseudorandom, seemingly chaotic matter. Some small-scale structures may still be seen.
4. **Class 4**: Involves a mixture of order and randomness. Simple local structures are produced (similarly to a Class 2), but these structures interact with each other in increasingly complex ways. This class is thought to be computationally universal.

Figure 1.2: An example of each type of class. The top of each image represents the initial conditions, and timesteps progress downward. Image taken from page 231 of Wolfram's A New Kind of Science.

The interested reader is invited to explore this book further at https://www.wolframscience.com/nks/.

At this point, we have completed a high level overview of CA, and we now have a toolbox of basic concepts and definitions to be able to move forward into our case study. However, the set of concepts and rules defined above are not exhaustive. As it is with any form of modeling, the specifics rely heavily on the description and configuration of the problem itself.

1.2 A Case Study: Game of Life

Proposed by British mathematician John Conway in 1970, Game of Life is one of the most well-known examples of a cellular automaton, and it will serve as our case study for further exploration of this framework. It is a good introduction to CA because not only is it simple and easily understandable, but it is also easily expandable, and the model achieves complex nontrivial behavior despite its simplicity.

Considering 1.1.b and 1.1.c, Game of Life is a Class 4 CA that employs cut-off boundaries.

1.2.a Implementation

In this section, we walk through a basic Python implementation of Game of Life that will serve as a foundation for the rest of this tutorial.

First, let's import our libraries. We utilize NumPy for its powerful array capabilities, Matplotlib for creating colormap plots, the random library for random number generation, and itertools.product for generating Cartesian products.

In [41]:

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.colors as colors
import random
from itertools import product
```

To begin, we first define our state set. Every cell consists of one of two self-explanatory states:

1. Alive
2. Dead

Per our definitions in 1.1.a, a Dead cell is a quiescent cell, and an Alive cell is an active cell. Let's associate these states with the integers 1 and 0. Although we could use booleans, using integers opens the door to possibly including more states when we expand the model in later tutorials.

In [37]:

```
# States:
ALIVE = 1
DEAD = 0
```

With our states defined, we now need a way of generating an initial configuration of the world.

To start, the n x n world, represented by a NumPy 2D array, is full of dead cells. We generate living cells in the form of clusters that are randomly placed and randomly populated. *cluster_n* and *clusters* determine the size and number of clusters, respectively, while threshold represents the likelihood that a given cell in a cluster is populated during the random generation phase. A threshold of 0 results in only dead cells, and a threshold of 1 results in only living cells.

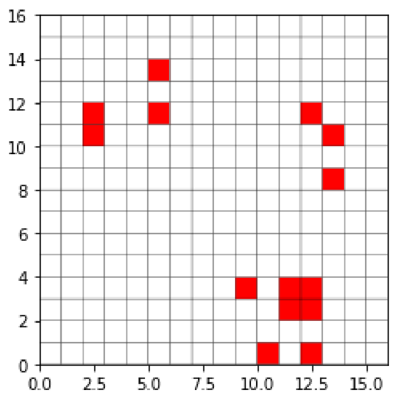
In [38]:

```
def init_world(n = 16, cluster_n = 4, clusters = 3, threshold = 0.25):
    world = np.zeros((n, n), dtype=int)
    for i in range(0, clusters):
        x = random.randint(0, n - cluster_n)
        y = random.randint(0, n - cluster_n)
        for world_x in range(x, x + cluster_n):
            for world_y in range(y, y + cluster_n):
                if random.uniform(0, 1) >= (1 - threshold):
                    world[world_x, world_y] = ALIVE

    return world
```

Now that we have a world created, we need a way to intuitively represent it. To draw our world, we utilize Matplotlib's pseudocolor plot, which represents a 2D array as a rectangular grid of cells of varying colors. Red cells are living, and white cells are dead.

In [40]:

```
def plot_world(world):
    cmap = colors.ListedColormap(['white', 'red'])
    plt.pcolor(world, cmap=cmap, edgecolor="black")
    plt.axis('square')
    plt.show()

world1 = init_world() # initialize world with default conditions
plot_world(world1)
```



Transitions between generations follow the Moore neighborhood paradigm and are as follows:

1. Any living cell with fewer than 2 neighbors dies **(underpopulation)**
2. Any living cell with 2 or 3 neighbors lives on to the next generation
3. Any living cell with more than 3 neighbors dies **(overpopulation)**
4. Any dead cell with exactly 3 neighbors becomes a living cell **(reproduction)**

With these rules in mind, we can define a function that will take in the x and y coordinates of a cell, get its neighbors, and perform an update based on the number of living cells surrounding it. This update will be reflected in our world state variable.

As mentioned previously, our implementation of Game of Life assumes cut-off boundaries, therefore our state transition function must take this fact into account.

In [5]:

```
def update_cell(world, x, y):
    num_alive = 0
    current_state = world[x,y]

    #count neighbors accounting for boundaries
    x_range = range(max(x - 1, 0), min(x + 1, world.shape[0] - 1) + 1)
    y_range = range(max(y - 1, 0), min(y + 1, world.shape[1] - 1) + 1)

    #count the number of living cells in the von Neumann neighborhood of coordinates around our ce
ll
    for cell_x, cell_y in product(x_range, y_range):
        if (not (cell_x, cell_y) == (x, y) and world[cell_x, cell_y] == ALIVE):
            num_alive += 1

    if (current_state == DEAD and not num_alive == 3):
        return DEAD
    elif (current_state == ALIVE and (num_alive < 2 or num_alive > 3)):
        return DEAD
        return DEAD
    else:
        return ALIVE
```

We now have a function that applies an update to an individual cell (represented by an (x,y) coordinate). However, in order to move the current configuration of the world to the next, we need a function that performs an update on each cell of the board.

In [7]:

```
def timestep(world):
    rows = world.shape[0]
    cols = world.shape[1]

    new_state = np.zeros((rows, cols), dtype=int)
    for x in range(0, rows):
        for y in range(0, cols):
            new_state[x,y] = update_cell(world, x, y)

    return new_state
```

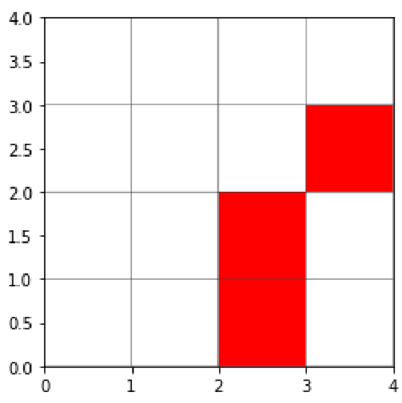
It is important to note that this update is **synchronous**, which means that the entire world is updated simultaneously. If a cell's state changes, that change is not accounted for until the next timestep, and to accomplish this we create a separate world state and apply the changes to that as we iterate through the grid. Depending on the problem, it may be necessary to perform **asynchronous** updates, where a single cell update is immediately taken into account for subsequent updates. We explore this concept in Part 2.

Another important remark is that this current algorithm is inefficient, as it iterates over every cell on the board for an $O(n^2)$ algorithm. This is acceptable for the sake of this tutorial, but for larger scales this could become computationally intractable. A possible angle for optimization would be to recognize that the majority of cells on the board will not change on an update, and that we need only target small areas of activity.

Let's view an example of a timestep. First, let's generate a small enough world so that we can easily view all transitions in a timestep

In [27]:

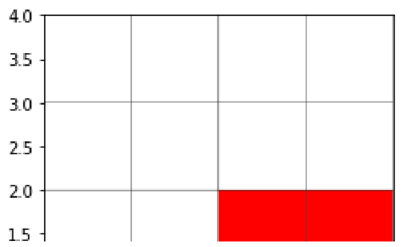
```
world2 = init_world(n = 4, cluster_n = 3, clusters = 2)
plot_world(world2)
```

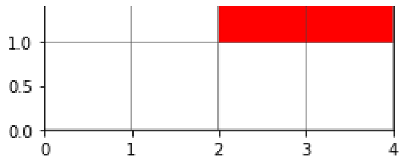


Before running the next block of code, take a second and try to predict what the resultant world will look like based on the description of the rules above. Remember to take into account synchronous updates.

In [39]:

```
world2_next = timestep(world2)
plot_world(world2_next)
```





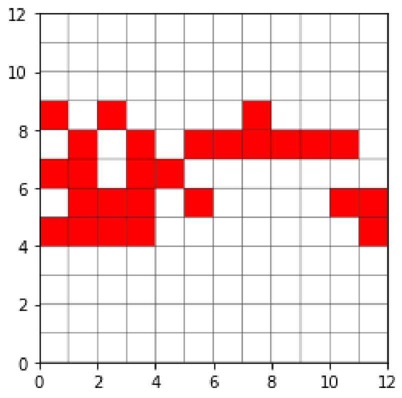
1.2.b Rudimentary Analysis

For the last step of our introduction to Game of Life, we briefly explore a form of analysis for a CA. The idea behind it is simple: if we count the number of living cells in every timestep of a series of configurations, we can plot the density over time.

First, let's create a new world. Let's up the threshold a bit so we start off with a higher density.

In [66]:

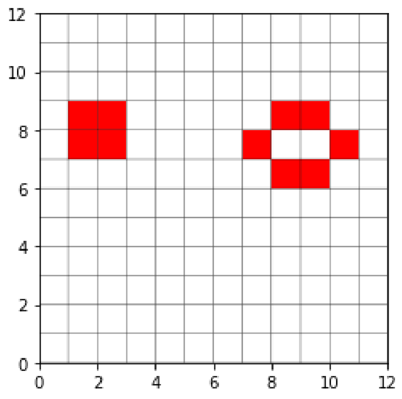
```
world3 = init_world(n = 12, cluster_n = 4, clusters = 4, threshold = 0.4)
plot_world(world3)
```



Using our previous timestep function, we can write a function to generate a timeseries of configurations.

In [70]:

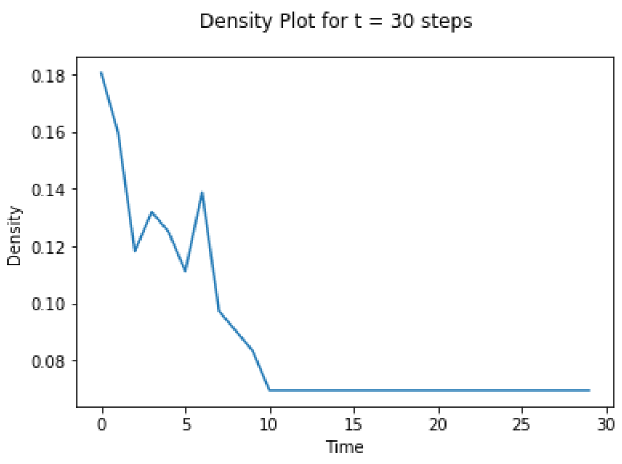
```
def timeseries(world, num_steps):
    simulation_steps = [world]
    for i in range(0, num_steps - 1):
        world = timestep(world)
        simulation_steps.append(world)

    return simulation_steps

configurations = timeseries(world3, 30)
plot_world(configurations[len(configurations) - 1])
```



Here we can see the end result of a timeseries of 30 steps. Now that we have our timeseries, we can plot the density of living cells over time.

In [71]:

```python
# define function to count all living entities given a configuration
def count_alive(world):
    return len(np.where((world == ALIVE).astype(int))[0])

def plot_density(timeseries):
    densities = []
    n = len(timeseries[0])
    for i in range(0, len(timeseries)):
        densities.append(count_alive(timeseries[i]) / (n * n))

    plt.plot(densities)
    plt.xlabel("Time")
    plt.ylabel("Density")
    plt.suptitle("Density Plot for t = " + str(len(timeseries)) + " steps")
    plt.show()

plot_density(configurations)
```



If you are running this notebook from the beginning, the plot you see may depend on how your specific world was created. However, in the long term, density typically drops to either 0, oscillates, or sits unchanging at some fixed constant. In Part 4, we explore a few of the patterns that give rise to these trends.

A density plot is an effective way of gaining a rudimentary overview of how a CA evolves over time ("at a glance"), but it doesn't provide much insight into the dynamics that may be occurring below the surface, especially in the case of Game of Life.

Part 2: Extensions to Basic Cellular Automata

In this part of the tutorial, we look at ways to expand upon the cellular automata model, and to make the model more expressive. In the original version, in keeping with Conway's Game of Life, we made many assumptions including deterministic behavior, synchronous updates, and cells that could live indefinitely. In this section we examine each of these assumptions and how to go forego them, in an attempt to more closely mimic characteristics a real population might have.

2.1 Asynchronous Updates

Unlike the original model we developed in section 1, we now explore asynchronous updates to the grid world. This means that instead of updating states as a function of only the previous state, we now update the cells in-place as a function of all prior updates, updating one cell at a time and allowing that to affect subsequent updates. This method of updating the states of cells may better mimic a real population because a whole population doesn't have all births and deaths occur so simultaneously, and instead one death or birth may affect other processes.

To do this, first we keep most of the functions the same from the last version, which is shown below for use in this notebook as well as for your reference.

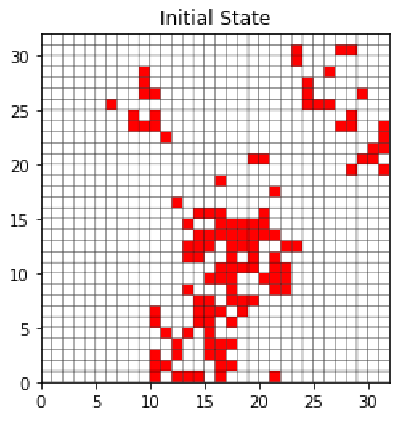
In [1]:

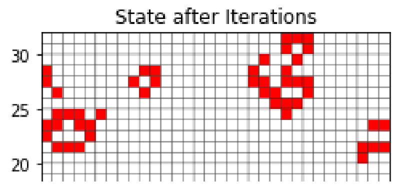
```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.colors as colors
import random
from itertools import product

ALIVE = 1
DEAD = 0

def init_world(n = 16, cluster_n = 4, clusters = 3, threshold = 0.25):
    world = np.zeros((n, n), dtype=int)

    for i in range(0, clusters):
        x = random.randint(0, n - cluster_n)
        y = random.randint(0, n - cluster_n)
        for world_x in range(x, x + cluster_n):
            for world_y in range(y, y + cluster_n):
                if random.uniform(0, 1) >= (1 - threshold):
                    world[world_x, world_y] = ALIVE

    return world

def update_cell(world, x, y):
    num_alive = 0
    current_state = world[x,y]

    #count neighbors
    x_range = range(max(x - 1, 0), min(x + 1, world.shape[0] - 1) + 1)
    y_range = range(max(y - 1, 0), min(y + 1, world.shape[1] - 1) + 1)

    for cell_x, cell_y in product(x_range, y_range):
        if (not (cell_x, cell_y) == (x, y) and world[cell_x, cell_y] == ALIVE):
            num_alive += 1

    if (current_state == DEAD and not num_alive == 3):
        return DEAD
    elif (current_state == ALIVE and (num_alive < 2 or num_alive > 3)):
        return DEAD
    else:
        return ALIVE

def count_alive(world):
    pass

def count_dead(world):
    pass

def timestep(world):
    rows = world.shape[0]
    cols = world.shape[1]

    new_state = np.zeros((rows, cols), dtype=int)
    for x in range(0, rows):
        for y in range(0, cols):
            new_state[x,y] = update_cell(world, x, y)

    return new_state

def plot_world(world, title):
    cmap = colors.ListedColormap(['white', 'red'])
    plt.pcolor(world, cmap=cmap, edgecolor="black")
    plt.axis('square')
    plt.title(title)
    plt.show()

def timeseries(world, num_steps):
    simulation_steps = [world]
    for i in range(0, num_steps):
        world = timestep(world)
        simulation_steps.append(world)

    return simulation_steps

def count_alive(world):
    return len(np.where((world == ALIVE).astype(int))[0])

def count_dead(world):
    return len(np.where((world == DEAD).astype(int))[0])

def plot_density(timeseries):
    densities = []
    n = len(timeseries[0])
    for i in range(0, len(timeseries)):
        densities.append(count_alive(timeseries[i]) / (n * n))

    plt.plot(densities)
    plt.xlabel("Time")
    plt.ylabel("Density")
    plt.suptitle("Density Plot for t = " + str(len(timeseries)) + " steps")
    plt.show()
```

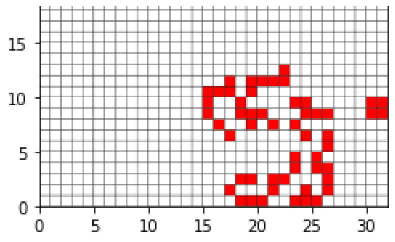
First, we'll run a short simulation, plotting the initial and final state after 20 world update iterations

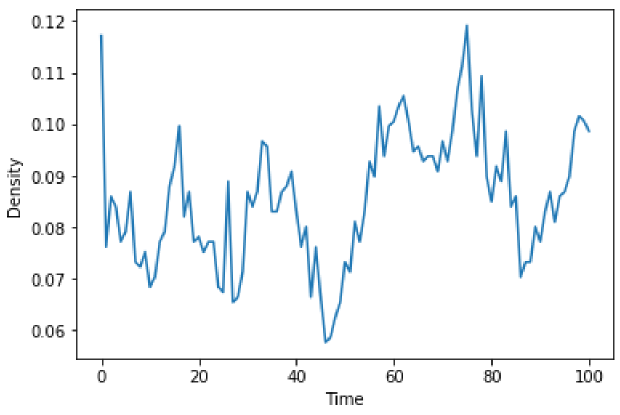
In [2]:

```
world = init_world(n = 32, cluster_n = 7, clusters = 10)
series = timeseries(world, 100)
plot_world(series[0], "Initial State")
plot_world(series[-1], "State after Iterations")
plot_density(series)
```








Density Plot for t = 101 steps

An important thing to note here is that the density of the model over time decreases, or at least doesn't grow explosively. This is a defining feature of the Game of Life in its standard form.

Now we add asynchronous updates by changing the timestep and timeseries functions. First, we change the timestep function by starting the new_state with a copy of the old world, then using the new_state to determing updates while subsequently updating the new_state and repeating for each square in the grid world.

In [3]:

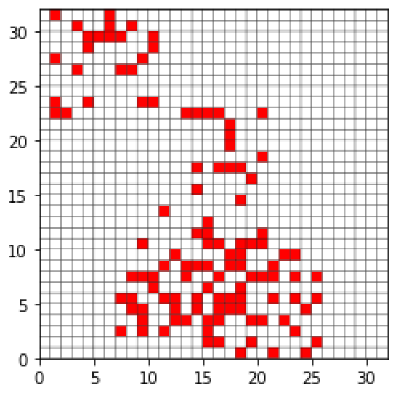
```
#update rectilinearly

def timestep_async(world):
    rows = world.shape[0]
    cols = world.shape[1]

    new_state = world.copy()
    for x in range(0, rows):
        for y in range(0, cols):
            new_state[x,y] = update_cell(new_state, x, y)

    return new_state

def timeseries_async(world, num_steps):
    simulation_steps = [world]
    for i in range(0, num_steps):
        world = timestep_async(world)
        simulation_steps.append(world)

    return simulation_steps
```

Now we see a comparison between the original state, the world after 20 synchrounous updates, and after 20 asynchronous updates to get a good visual comparison of how the world behaves.

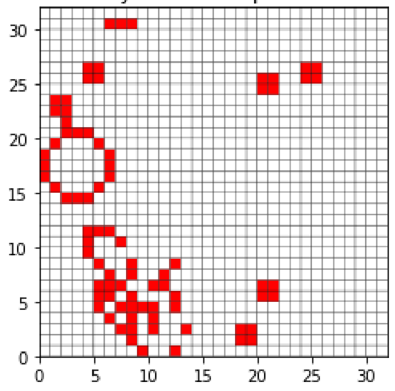
In [4]:

```
world = init_world(n = 32, cluster_n = 10, clusters = 5)
series = timeseries(world, 100)
series_async = timeseries_async(world, 100)
plot_world(world, "Initial State")
plot_world(series[-1], "Synchronous Updates")
plot_density(series)
plot_world(series_async[-1], "Asynchronous Updates")
plot_density(series_async)
```

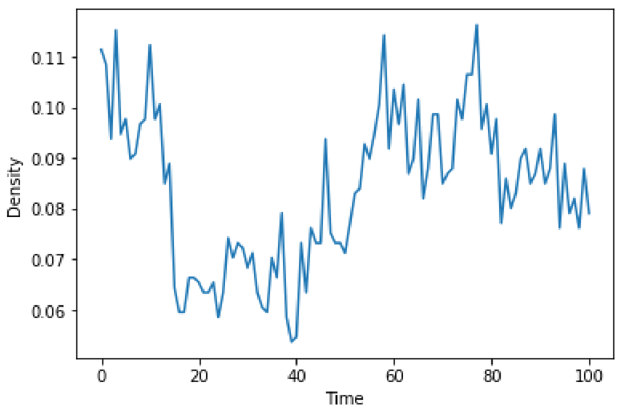
Initial State



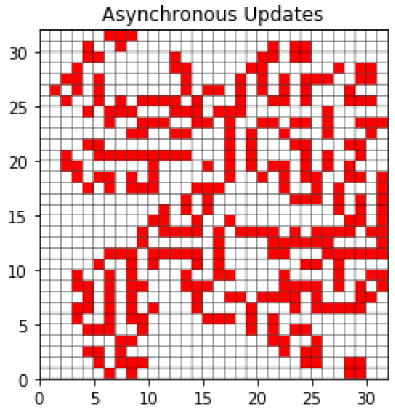
Synchronous Updates



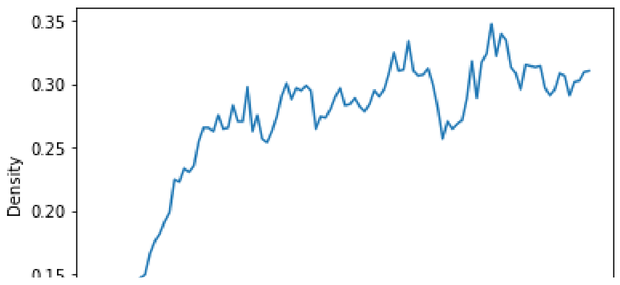
Density Plot for t = 101 steps

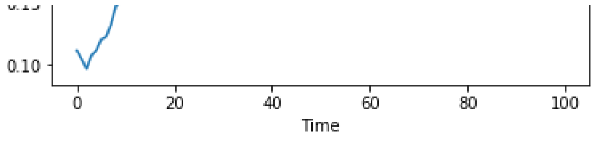


Asynchronous Updates



Density Plot for t = 101 steps





From these graphs, we see that the synchronous updates created a grid with likely fewer live nodes, and more spatially grouped. The asynchronous updates grid however has many more live squares, and a maze-like pattern (lots of horizontal and vertical lines) has formed, showing a distinctly macroscopic organization of the live nodes. This concept relates to the last section of this tutorial, but we get a taste for self-organization here. Also noteworthy is that the density of live cells in the aynchronous grid world grows constantly, defeating the stability of the standard Game of Life.

2.2 Stochasticity

Most cellular automata are perfectly deterministic models, showing the natural progression of a system whose outcome is guaranteed the moment a starting state is chosen. While this model has many interesting properties that create deterministic emergent behaviors like spaceships (a formation that's unlikely with stochasticity), it may be an oversimplification of real populations. In the real world things happen with enough variables that a stochastic model may be a more realistic prediction of behaviors.

There are many ways to add stochasticity to our model, but one that fits well with asynchronous updates from before is to choose cells to update at random, allowing the model to grow and evolve in a much less rigid and organized way, mimicking the randomness a real population might experience.

This new model is a much closer mimic of a real population than the original model, and we can examine its behavior in the exercise below.

To change our model to allow for random asynchronous updates, we take the asynchronous timestep and timeseries functions from before, except this time we change the way timestep operates. In this function, for the total number of squares there are in a grid-world, we choose a cell completely at random using python's random library, creating a random int within the row and column range and updating that cell. We update timeseries to call our new timestep function.

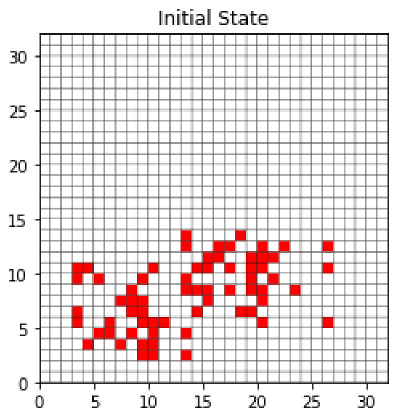
In [5]:

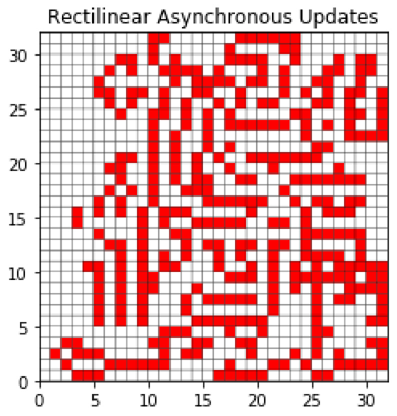
```
#update randomly
import random

def timestep_rasync(world):
    rows = world.shape[0]
    cols = world.shape[1]

    new_state = world.copy()
    for x in range(0, rows*cols):
        x = random.randint(0,rows-1)
        y = random.randint(0,cols-1)
        new_state[x,y] = update_cell(new_state, x, y)

    return new_state

def timeseries_rasync(world, num_steps):
    simulation_steps = [world]
    for i in range(0, num_steps):
        world = timestep_rasync(world)
        simulation_steps.append(world)

    return simulation_steps
```

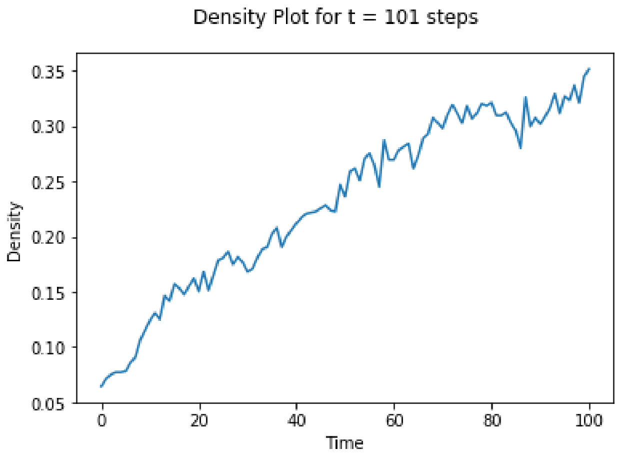
Now we run a simulation using both asynchronous models we have developed so far. First we examine the initial state, then the world after rectilinear asynchronous updates, then random asynchronous updates.

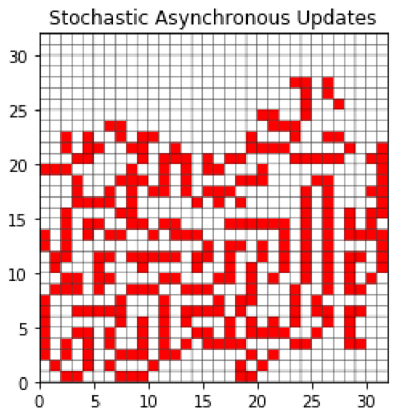
In [6]:

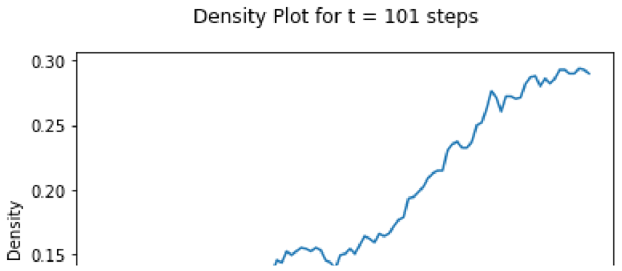
```
world = init_world(n = 32, cluster_n = 8, clusters = 5)
series_async = timeseries_async(world, 100)
series_rasync = timeseries_rasync(world, 100)
plot_world(world, "Initial State")
plot_world(series_async[-1], "Rectilinear Asynchronous Updates")
plot_density(series_async)
plot_world(series_rasync[-1], "Stochastic Asynchronous Updates")
plot_density(series_rasync)
```

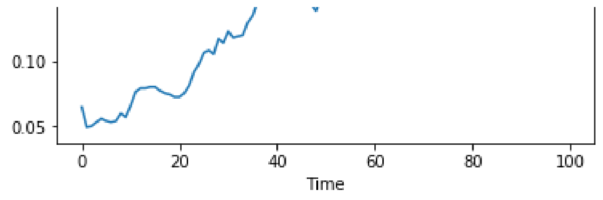

Initial State
0
5
10
15
20
25
30


Rectilinear Asynchronous Updates


Density Plot for t = 101 steps
Density
Time


Stochastic Asynchronous Updates


Density Plot for t = 101 steps
Density



Here we notice that the random updates tend to form a strikingly similar model to the asynchronous updates. However, the asynchronous model creates a visually smaller region of living cells, indicating that random updates may disrupt a pattern within the rectilinear updates that spreads more evenly across the grid space. Also, we again see that the densities of the asynchronous update grid worlds grow significantly over time, which if this continues forever, it could be a poor model of many real world populations that all have some carrying capacity.

2.3 Time To Live (TTL)

The cells in our first model in section 1 lived forever, which could be seen as somewhat unnatural. In reality all members of a population eventually die, so we can use a time-to-live component where if a cell is alive for too long, it dies. Here, we'll consider a cell going from DEAD to ALIVE resets the time to live, while decrementing a counter for each update a cell stays ALIVE.

To do this, we can simply replace the world array with the time to live values, making 0 be still DEAD, while any number greater than 0 indicates the ALIVE state. We need to decide on an initial time to live value, which we'll call START_TTL. This value can be adjusted to whichever value seems to produce the most interesting results. We have left this start value as 6 from our own experiments, but the interested reader is encouraged to play around with this value to examine how it affects the behavior of the model.

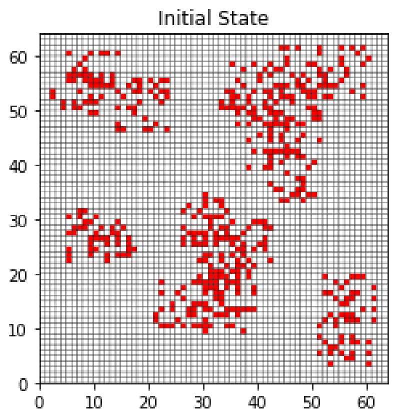
In [7]:

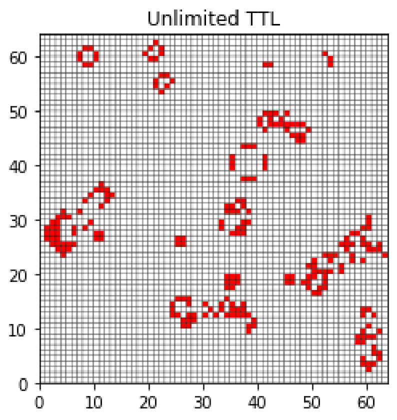
```
#time to live component
DEAD = 0
START_TTL = 6

def init_world_ttl(n = 16, cluster_n = 4, clusters = 3, threshold = 0.25):
    world = np.zeros((n, n), dtype=int)

    for i in range(0, clusters):
        x = random.randint(0, n - cluster_n)
        y = random.randint(0, n - cluster_n)
        for world_x in range(x, x + cluster_n):
            for world_y in range(y, y + cluster_n):
                if random.uniform(0, 1) >= (1 - threshold):
                    world[world_x, world_y] = START_TTL #change
    return world

def update_cell_ttl(world, x, y):
    num_alive = 0
    current_state = world[x,y]

    #count neighbors
    x_range = range(max(x - 1, 0), min(x + 1, world.shape[0] - 1) + 1)
    y_range = range(max(y - 1, 0), min(y + 1, world.shape[1] - 1) + 1)

    for cell_x, cell_y in product(x_range, y_range):
        if (not (cell_x, cell_y) == (x, y) and world[cell_x, cell_y] > DEAD):
            num_alive += 1

    if current_state == DEAD:
        if not num_alive == 3:
            return DEAD
        else:
            return START_TTL
    else:
        if num_alive != 2 and num_alive != 3:
            return DEAD
        else:
            return current_state - 1

def timestep_ttl(world):
    rows = world.shape[0]
    cols = world.shape[1]

    new_state = np.zeros((rows, cols), dtype=int)
    for x in range(0, rows):
        for y in range(0, cols):
            new_state[x,y] = update_cell_ttl(world, x, y)

    return new_state

def timeseries_ttl(world, num_steps):
    simulation_steps = [world]
    for i in range(0, num_steps):
        world = timestep_ttl(world)
        simulation_steps.append(world)

    return simulation_steps
```

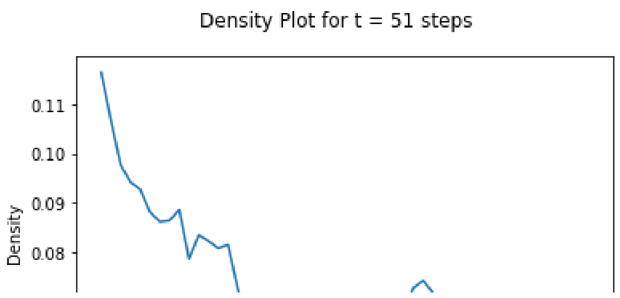
Below, we test the model first showing the initial state, then the world after being updated according to the standard Game of Life, and finally after being updated according to the functions using a time to live value. Again, we encourage the reader to adjust the START_TTL value and rerun the cell below to examine different behaviors.

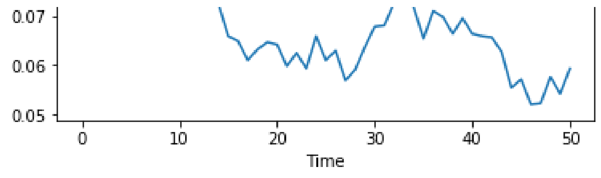
In [8]:

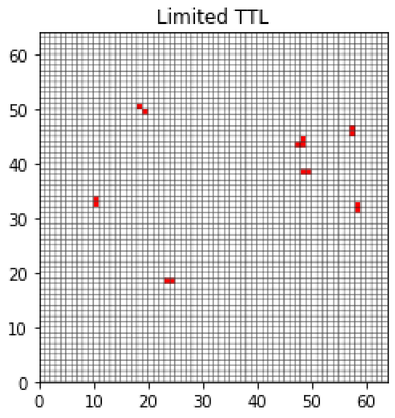
```
START_TTL = 4

world = init_world(n = 64, cluster_n = 10, clusters = 20)
world_ttl = init_world_ttl(n = 64, cluster_n = 10, clusters = 20)
series = timeseries(world, 50)
series_ttl = timeseries_ttl(world_ttl, 50)
plot_world(world, "Initial State")
plot_world(series[-1], "Unlimited TTL")
plot_density(series)
plot_world(series_ttl[-1], "Limited TTL")
plot_density(series_ttl)
```

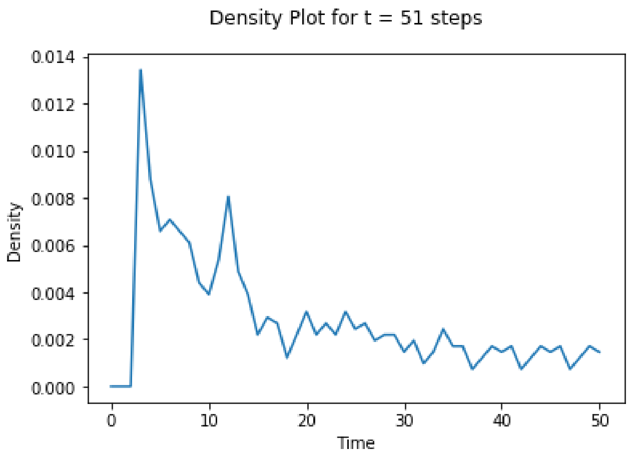












Examining the output of the above graphs, we notice that with a large START_TTL, the output will be remarkably similar to the standard game of life. This is because if the time to live for a node is very high, it will take a long time for it to die any other way than Game of Life's normal lonliness or overcrowding. However, as the START_TTL is reduced towards a small number (4 or smaller), the grid world tends to lose all life. This is in contrast to the asynchronous update models which tend to grow explosively. Therefore, in the next section we combine these two in order to make an attempt at achieving a stable growth rate.

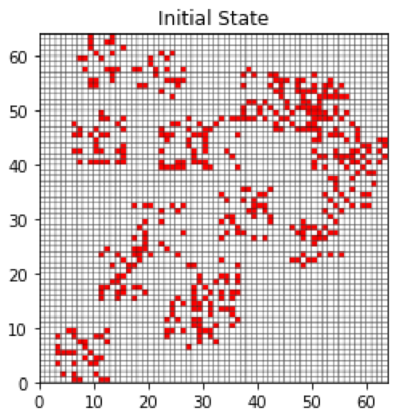
In [9]:

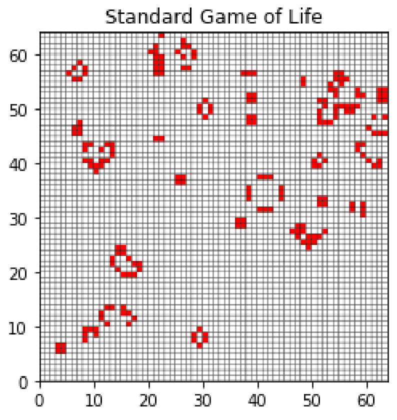
```
def timestep_async_ttl(world):
    rows = world.shape[0]
    cols = world.shape[1]

    new_state = world.copy()
    for x in range(0, rows):
        for y in range(0, cols):
            new_state[x,y] = update_cell_ttl(new_state, x, y)

    return new_state

def timeseries_async_ttl(world, num_steps):
    simulation_steps = [world]
    for i in range(0, num_steps):
        world = timestep_async_ttl(world)
        simulation_steps.append(world)

    return simulation_steps
```

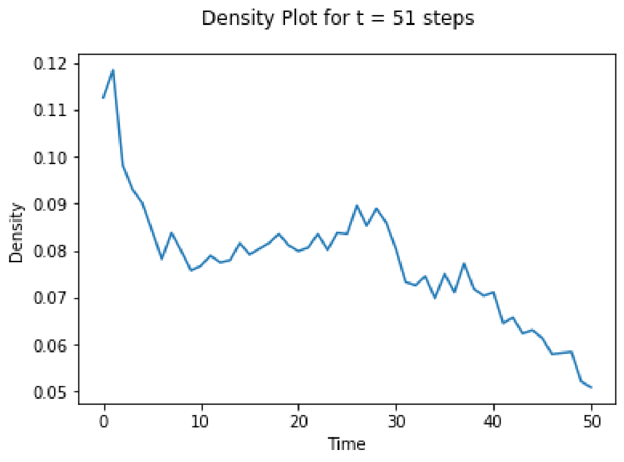
In [10]:

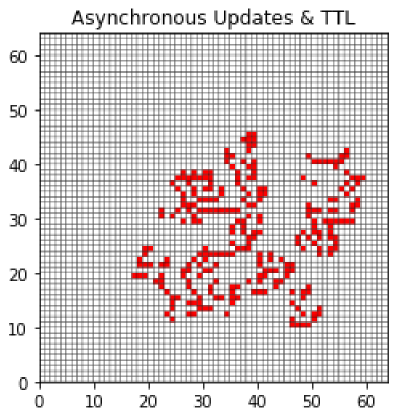
```
START_TTL = 4

world = init_world(n = 64, cluster_n = 10, clusters = 20)
world_ttl = init_world_ttl(n = 64, cluster_n = 10, clusters = 20)
```

```
series = timeseries(world, 50)
series_async_ttl = timeseries_async_ttl(world_ttl, 50)
plot_world(world, "Initial State")
plot_world(series[-1], "Standard Game of Life")
plot_density(series)
plot_world(series_async_ttl[-1], "Asynchronous Updates & TTL")
plot_density(series_async_ttl)
```

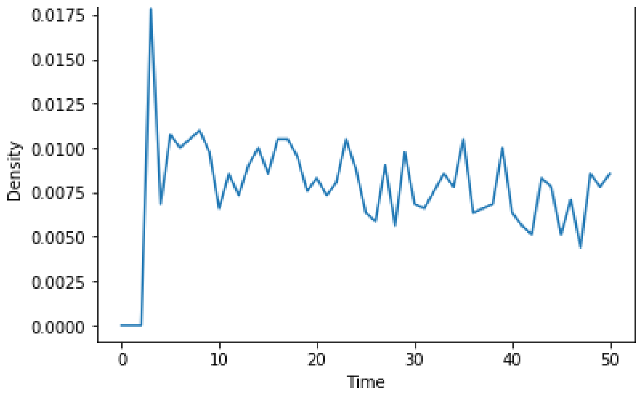








Density Plot for t = 51 steps



With the initial TTL value of 4, we achieve a mostly stable growth rate, similar to that of the standard Game of life. However, our resulting model has more properties similar to that of a real population, namely that cells don't live forever, and updates to cells influence all future updates asynchronously.

Part 2 Conclusion

These are only a few of the ways we can update the cellular automata model; there are many other ways to work with cellular automata that make them an exceptionally expressive model. For our purposes, it creates some interesting emergent behaviors as with the asynchronous updates and the resulting maze-like pattern, and the reader is encouraged to consider other ways the model could be changed to further explore its applications.

Part 4: Exploring Patterns in Game of Life

In the final part of this tutorial, we investigate how interesting patterns and behaviors can emerge over time from even a simple CA. Using Game of Life, we start at the most basic level, still lifes, then slowly build our way up to more complex examples. We conclude with a look at transmitting information signals and creating logic gates using only the rules of Game of Life.

Before we start, we need to set up our boilerplate code by pulling from our Part 1 implementation (conway_basic.py). We also define a new function called *plot_series*, which allows us to plot a few subsequent timesteps in a row.

In [1]:

```
import os
import sys
import matplotlib.pyplot as plt
import matplotlib.colors as colors

# this code allows us to import functions from our full scripts
module_path = os.path.abspath(os.path.join('../scripts'))
if module_path not in sys.path:
    sys.path.append(module_path)

# import functions from basic implementation as helper functions
from conway_basic import plot_world, timeseries

# generic timeseries plotting
def plot_series(world, steps):
    series = timeseries(world, steps)

    fig, ax = plt.subplots(1, steps)

    for i in range(0, steps):
        ax[i].pcolor(series[i], cmap=cmap, edgecolor="black")
        ax[i].axis('square')
        if not i == 0:
            ax[i].yaxis.set_visible(False)
        ax[i].set_title("t = " + str(i))

    plt.show()
```

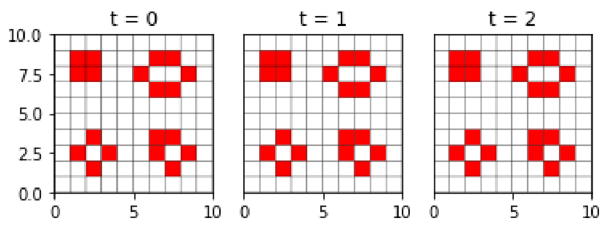
4.1 Still Lifes

The simplest patterns that emerge within Game of Life are those known as *still lifes*. These are fixed patterns that do not change from generation to generation.

As a note before we jump into the code, all of these worlds are created using hardcoded initialization functions in order to effectively demonstrate the patterns. We do not include these functions as part of the tutorial, as they are simply setting values in the initial NumPy array, but if one is interested, they can be found in pattern_worlds.py.

In [14]:

```
# import world init function
from pattern_worlds import init_fixed_world

world = init_fixed_world()
plot_series(world, 3)
```



Here we see four different still lifes. In clockwise order around the grid, these are termed *block*, *beehive*, *boat*, and *tub*.

Although still lifes on their own are trivial, they are still important to understand because they are often integral components to constructing more complicated patterns, which we will investigate further on. For example, certain still lifes are known as *eaters*, which means that they have the ability to interact with and destroy certain patterns without suffering any permanent damage. These help stabilize and control debris that may arise in complex reactions. Similarly, another example is the concept of a *reflector*, which is a still life capable of reflecting certain patterns without receiving damage.

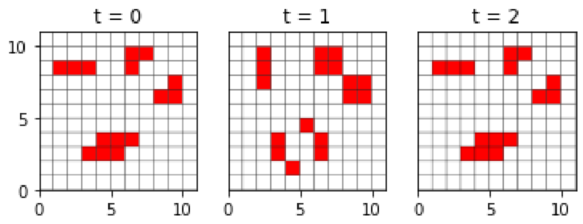
As we will see later on, a block is able to serve as both, and is therefore one of the most important still lifes.

4.2 Oscillators

Moving up in complexity from still lifes, we come next to oscillators. These are patterns who return to their initial state after a finite number of generations (this number is known as the *period* of the oscillator).

In [11]:

```
# import world init function
from pattern_worlds import init_oscillator_world

world = init_oscillator_world()
plot_series(world, 3)
```



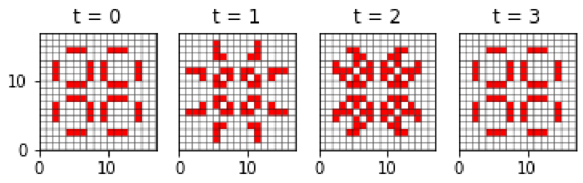
Here we see three examples of simple oscillators. In clockwise order around the grid, these are termed *blinker*, *beacon*, and *toad*. As you can see, after 2 timesteps they have returned to their initial state, giving each of them a period of 2. These are considered to be the most common oscillators.

Similarly to still lifes, oscillators can sometimes also act as eaters and reflectors, but less commonly so.

Another more interesting example is a *pulsar*, which, despite its apparent complexity, returns to its initial state after only 3 generations!

In [12]:

```
# import world init function
from pattern_worlds import init_pulsar_world

world = init_pulsar_world()
plot_series(world, 4)
```



4.3 Spaceships

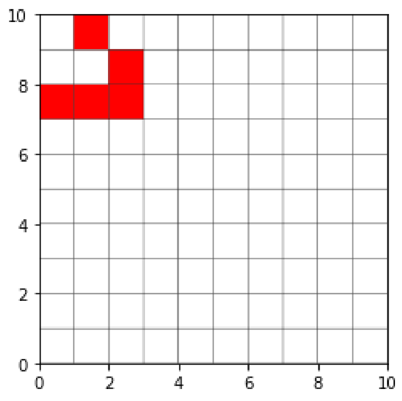
So far, the patterns we've investigated don't really mean much. They may be interesting to look at (especially pulsars!), but on their own, their behavior is trivial.

Stepping forward in complexity, we now investigate behaviors that *move*, which as we will see, have some fairly profound implications for what you can do with Game of Life. These behaviors are affectionately known as *spaceships*. Let's look at an example.

In [6]:

```
# import world init function
```

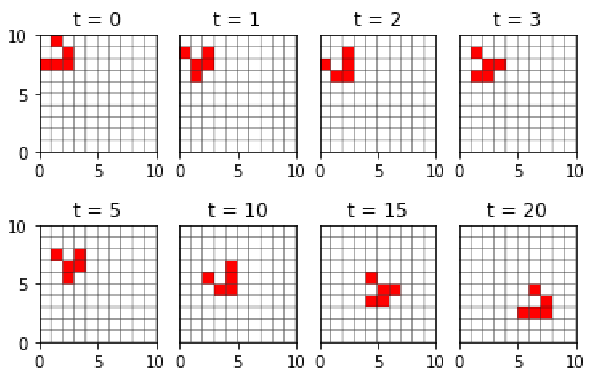
```python
from pattern_worlds import init_glider_world

world = init_glider_world()
plot_world(world)
```



This is the initial configuration of a *glider*, which was the first spaceship discovered as well as the smallest and most common. Let's now view how it evolves over time.

In [27]:

```python
def plot_glider_series():
    world = init_glider_world()
    series = timeseries(world, 25)

    fig, ax = plt.subplots(2, 4)

    for i in range(0, 4):
        ax[0, i].pcolor(series[i], cmap=cmap, edgecolor="black")
        ax[0, i].axis('square')
        ax[0, i].set_title("t = " + str(i))

        ax[1, i].pcolor(series[(i + 1) * 5], cmap=cmap, edgecolor="black")
        ax[1, i].axis('square')
        ax[1, i].set_title("t = " + str((i + 1) * 5))

        if not i == 0:
            ax[0, i].yaxis.set_visible(False)
            ax[1, i].yaxis.set_visible(False)

    plt.show()

plot_glider_series()
```



In the above graph, we show timesteps $t = 0$ through $t = 3$ to demonstrate the period of a glider ($t = 5$ will return to the initial configuration, therefore the period is 5 generations).

Then, timesteps $t = 5$ to $t = 20$ are plotted in steps of 5 to demonstrate how a glider evolves over time. Every 5 steps, a glider translates one cell diagonally (to the right and downward). Different directions can be achieved by rotating the initial configuration. This continues indefinitely until it hits a boundary or other active cells.

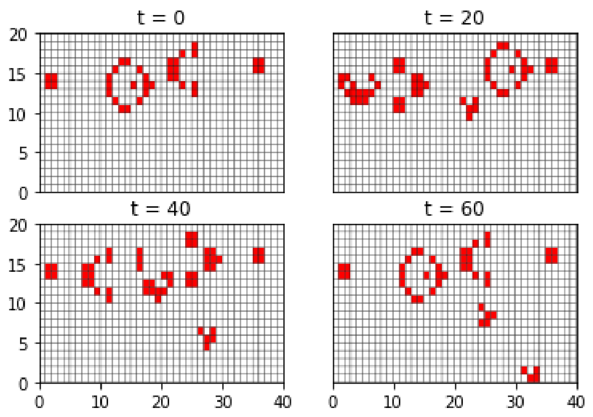
Generating Gliders

Generating Gliders

Now that we have introduced gliders, we can take it a step further by exploring a construction that can spontaneously and indefinitely produce these gliders. These types of structures are known as *guns*, and the subject of our study today is *Gosper's glider gun* (named after mathematician Bill Gosper, who discovered it in 1970). Below is a demonstration of it.

In [12]:

```python
# import world init function
from pattern_worlds import init_glider_gun

def plot_gun_series():
    world = init_glider_gun()
    series = timeseries(world, 100)

    fig, ax = plt.subplots(2, 2)

    for i in range(0, 2):
        for j in range(0, 2):
            ax[i, j].pcolor(series[(i * 2 + j) * 20], cmap=cmap, edgecolor="black")
            ax[i, j].set_title("t = " + str((i * 2 + j) * 20))

            if not j == 0:
                ax[i, j].yaxis.set_visible(False)

            if i == 0:
                ax[i, j].xaxis.set_visible(False)

    plt.show()

plot_gun_series()
```



$t = 0$ demonstrates the initial structure of the gun. The two structures in the middle oscillate back and forth, and the two blocks on the edges serve as stabilizers, reflecting the structures.

$t = 20$ demonstrates how, after a collision in the center of the gun, a glider has been created. $t = 40$ demonstrates an in-progress collision, and $t = 60$ shows how it has returned to an initial configuration with two gliders having been spawned. Based on this, the period of the glider gun is 30 generations, with one glider created per period.

Lastly, for those viewing this tutorial in Jupyter notebook rather than PDF format, below is a GIF demonstrating an animation of Gosper's glider gun for a clearer view of what is happening.

Source: http://www.conwaylife.com/w/images/b/b6/Gosperglidergun.gif

4.4 Information Signals and Logic Gates

As the culminating topic of this tutorial, we now look into how we can transmit information and create logic gates using Game of Life. Having covered gliders and glider guns, we have the tools necessary to do so.

In this case, we can consider a stream of gliders as an information signal. Thinking of binary, gliders represent 1 (True), and no gliders represents 0 (False). Two streams of gliders that collide in a certain way will annihilate each other.

With these glider guns in hand and little bit of cleverness, we can direct streams of gliders in specific ways to create NOT, OR and AND gates!

Since the behaviors and construction of these can be quite complicated, we utilize an open source program called Golly to generate and view the behavior of these logic gates.

4.4.a NOT Gate

A NOT gate takes in one input and outputs the reverse.

To build our gate, we use a glider

4.4.b AND Gate

An AND gate only returns True if both inputs are True. It return False in every other case.

To build our gate, we use two glider guns directed to the right as inputs, and a glider directed to the left as our gate. Our output is any stream of gliders to the right.

In the first case, A = True and B = True. B annihilates the stream going down the screen to the left, allowing A to move to the output, returning True.

In the second case, A = True and B = False. Since B is not transmitting, the signal from the gate collides with A, annihilating it and resulting in no output (False).

In the third case, A = False and B = True. Although B and the gate collide, A is not transmitting, so there is no output (False).

In the final case (which is not shown), A = False and B = False. Although the gate is transmitting to the left, there is no output to the right (False).

4.4.c OR Gate

An OR gate returns True if at least one of the inputs are True. It only returns False if both are False.

We leave this gate as an exercise to the reader! Download Golly, build your glider guns, and play around with the inputs.

Since we can create NOT, OR, and AND gates in Game of Life, this means it is Turing Complete, or computationally universal! This shows that even in a simple CA, an impressive diversity of behaviors can emerge. This concludes our tutorial on emergent patterns in Game of Life.

```
In [ ]:
```

Team Collaboration:
Brian Glowniak produced much of the organizational structure and direction for the project, as well as the base Game of Life model to work on, and part 3 of the tutorial. Joel Katz produced part 2 of the tutorial, and took care of much of the paperwork, including for the checkpoint and for the project 1 submission.