Considering Computational Modeling and Complexity Science

Version 0.0.1

Considering Computational Modeling and Complexity Science

Version 0.0.1

Heena Mutha

Preface

Acknowledgements

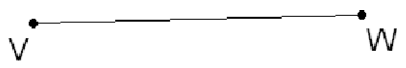
Contents

Chapter 1

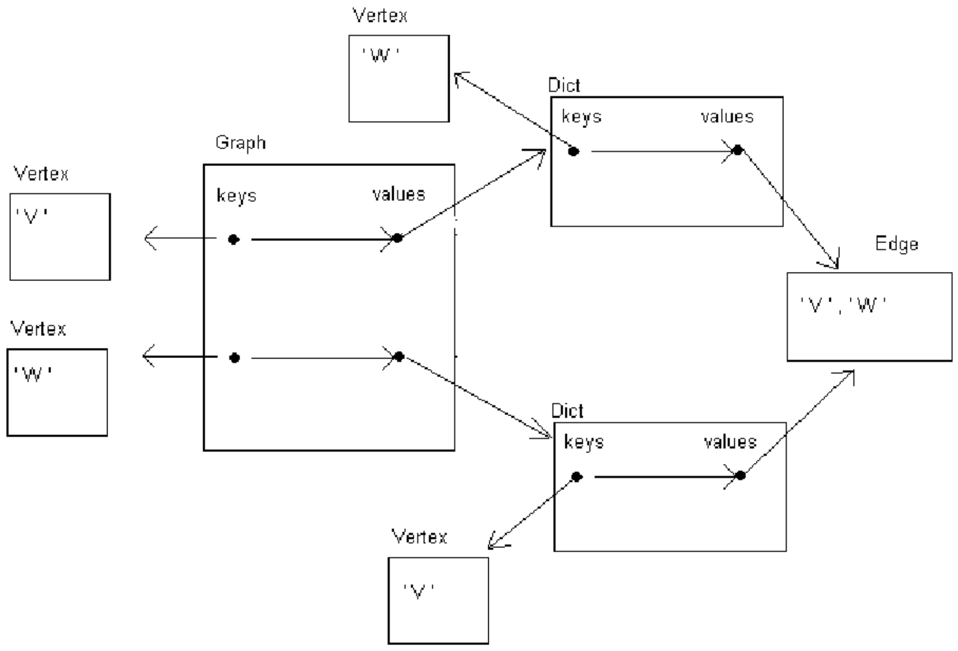
Graphs

1.1 Representing Graphs

Graphs are a mapping system used to describe interconnected discrete elements. These elements are called nodes (or vertices), and the connections to one another, edges. These network models allow us to study a variety of interactions, which we will discuss throughout this chapter and book. In order to create these graphs, we need a data structure that allows us to keep track of the interaction of all nodes and edges. The dictionary data structure available in Python allows us to do just that.We will use Allen Downey's basic graph implementation provided in his text Computational Modeling and Complexity Science. Before proceeding with a discussion of graph manipulation, lets take a moment to interpret the data structure we will be using throughout this chapter. For example, how does our Python program store the graph shown below as data?



Downey's `Graph` is a dictionary of dictionaries. The keys of the outer dictionary are vertices and their values are the inner dictionaries that describe the key's immediate relations (directly connected vertices and the edges that join them). The following object diagram characterizes this relationship:



We see that `Vertex V` is a key in the outer dictionary, and it references an inner dictionary that contains `Vertex W` as a key and the `Edge(V,W)` its value.

1.2 Working with Graphs

When creating a graph, the user should be able to access a variety of information about it. In the simplest cases, a user would likely want to know the number of vertices and edges that make up a graph in order to characterize the network. Because of our use of dictionaries, accessing the keys will give us the vertices, and the values of those values will yield the edges.

```
def vertices(self):
    return self.keys()

def edges(self):
    edges=[]
    vertices=self.values()
    for vertex in vertices:
        es=vertex.values()
        for e in es:
            if e in edges:
                continue
            else:
                edges.append(e)
        return edges
```

We notice that in the method `edges`, we have to check if edges are stored. This is because every vertex stores edge information, and as an edge is defined as the connection between two nodes, we would double-count the edge if we did not check for its listing.

Also, it is useful to check whether two nodes are connected to one another. This proves handy when checking whether a graph is connected. If an edge exists, then the Graph dictionaries will have a reference of that edge. The method `get_edge` allows us to test whether two vertices are connected by an edge:

```
    def get_edge(self, v, w):
       try:
           return self[v][w]
       except:
           return None
```

Along that theme, sometimes we will find it useful to isolate information about a specific node. To check the degree of a graph, we can check what edges are connected to the vertex. In other cases, we can check what vertices are connected to one another by looking at what nodes are connected to our node of interest. The methods `out_vertices` and `out_edges` allow us to achieve this ability:

```
   def out_vertices(self, v):
        attached=self[v]
        out_vertices=attached.keys()
        return out_vertices

    def out_edges(self,v):
        attached=self[v]
        out_edges=attached.values()
        return out_edges
```

Here `attached` is the dictionary of the `Vertex V`, which stores the connected vertices as keys and the edges as values.

In Downey's Graph class, we are given the ability to add edges between nodes. In some cases, we find that it is useful to remove an edge from our graph. This can be achieved by:

```
    def remove_edge(self, e):
        v,w=e[0], e[1]
        del self[v][w]
        del self[w][v]
        del e
```

Because edge is a tuple of the vertices it connects, we can use the edge information to delete the edge, and any references it has in the Graph dicitionary.

With this we achieve basic graphical functionality.The real challenge comes in building graphs.

1.3 Building Graphs

The simplest graph to create would be one in which all nodes are connected directly to every other one.Lets begin an exercise in graph creation by using GraphWorld.py from Downey. Here we define the number of vertices we desire for our Graph and connect them directly to one another. We call this method `add_all_edges` within the Graph class:

```
def add_all_edges(self):
    """starts with an EDGELESS graph add connects every vertex
    with one another"""

    vs=self.keys()
    for i in range(len(vs)):
        v=vs[i]
        for w in vs:
            if v != w:
                e= Edge(v,w)
                self.add_edge(e)
```

By simply creating a list of the vertices and traversing the list, we connect everthing directly.

However, in some cases, we may wish to connect a graph's vertices by a certain degree. Degree refers to the number of direct connections one node has with others. There are many considerations we must make when creating a graph with our own preset degree. By discrete mathematics, we find that not every combination of degrees and number of nodes yield a successful graph. Mathematically there are a few rules that must be checked:

1. A given degree must always be less than the number of nodes in the graph.
2. If the number of nodes is odd, than the degree given must be an even number.

The method `check_creatability` verifies these rules for user inputs:

```
def check_creatability(self, deg, nodes):
    """check whether a network can be built for the
    given number of nodes and the degree desired"""
```

```
        if deg > nodes:
            raise ValueError,'degree greater than # of nodes'

        if self.even_or_odd(nodes)is 'odd':
            if self.even_or_odd(deg)is 'odd':
                raise ValueError, 'even degree given for odd # of nodes'
```

In addition, as the program connects a graph, there are other status checks that must be made: is either node considered already at the user-specified degree, are the nodes already connected, is the computer comparing the same vertex? If all these things are false, then we can connect the two together with an edge. The method we can use to create the graph is given by `add_regular_edges`:

```
def add_regular_edges(self, deg):

        vs=self.keys()
        nodes=len(vs)

        #check whether the degree can exist for given number of nodes

        self.check_creatability(deg, nodes)

        # if exception is not raised than build graph
        for i in range(len(vs)):
            v=vs[i]
            vout_vertices=self.out_vertices(v)
            while len(vout_vertices) < deg:
                 #for random construction we use random to choose a vertex
                w= random.choice(vs)

                if w not in vout_vertices: #check they are not connected
                    if len(self.out_vertices(w))<deg:#check the degree
                       if v != w: #check they're different
                            e=Edge(v,w)
                            self.add_edge(e)
                            vout_vertices.append(w)
```

1.4 Random Graphs

Now that we have a working Graph class, and many capabilities for gathering information and building graphs, we can start to think about stochastic modeling. The study of random graphs is of great fascination to mathematicians. Paul Erdős and Alfréd Rényi developed a random graph model: the existence of edges between nodes depends on a user-defined probability that it can exist. In this system, we say if a number generator yields a value within our threshold, then the edge will be drawn. If not, an edge will not exist between the two nodes. The class utilizes the Erdős-Rényi model to generate these graphs:

```
class RandomGraph(Graph):
    def __init__(self,vs,p):
        """takes vertices and a probability to create a random graph.
        Probability should be between 0 and 1"""

        Graph.__init__(self,vs)
        self.add_random_edges(p)


    def add_random_edges(self, p):
        vs=self.keys()
        for i in range(len(vs)):
            for j in range(i+1,len(vs)):
                n=random.randint(0,100)
                n=n/100.0
                if n<p:
                    e=Edge(vs[i], vs[j])
                    self.add_edge(e)
                else:
                    continue
```

To create the graph, a vertex is chosen. We traverse through the rest of the list of vertices and decide to create an edge between the two vertices by comparing the given probability with a value dictated by a random number generator. If the random number generator returns a number that is beneath the probability threshold, than the edge can be created.In this implementation, we are creating a undirected graph because once two vertices are considered for edge creation, they are never reconsidered as we traverse through the list.

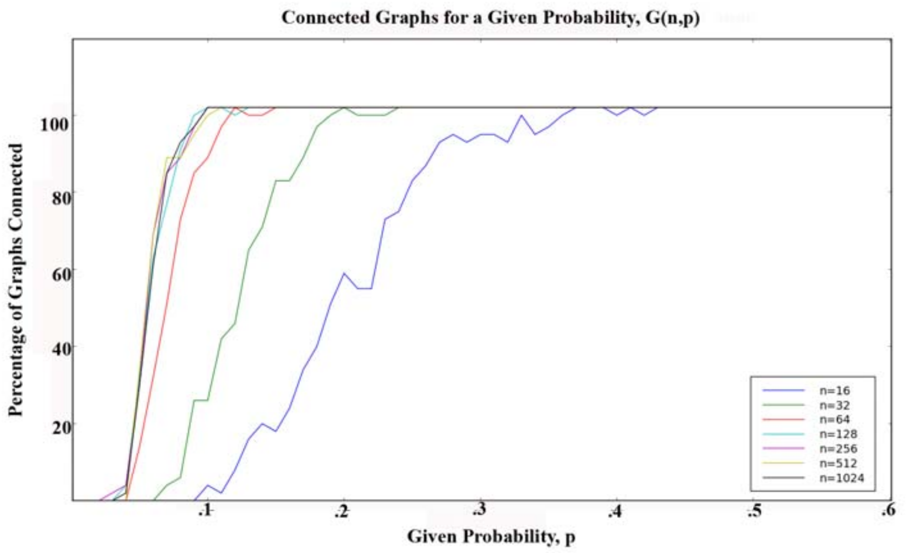
1.5 Connectedness

As we randomly construct graphs, a natural question arises: will the graph be connected? To check whether a graph is connected or not, we can use the breadth-first-search model.This method works by creating a queue. A vertex is selected and added to the queue. Then this vertex is marked, and the vertices directly connected with it are added to the queue if they are unmarked. Another vertex in the queue is marked and the procedure is repeated until the queue is empty.When the queue is empty we are done scanning. If the marked list is the same length as the list of total vertices, than we know the graph is connected. Otherwise, at least one vertex is unmarked, meaning it is not connected with the group.Creating a program with this logic allows us to conduct random graph analysis which will be done in the following section.

1.6 Random Graph Experiment: Finding Critical Probabilities

Erdős and Rényi discovered in their random graph experiments that for a range of probabilities there is a value where the likelihood of a connected graph's existence switches between extremely rare to extremely likely. This critical value, denoted as p*, marks a phase change between these two outcomes.

Using the programs created in the previous sections, we can use our program to find these critical probabilites for a variety of maps.In my experiment, I examined nodes by powers of 2 from 16 up to 1024. 50 trials were conducted for each given probability, and the probabilities tested were from 0 to 1 incremented by .01. The results are given in the figure below:



For n=16 nodes, p*=.22. For n=32 nodes, p*=.125, and for 64 nodes, p* is .07. When the number of nodes is beyond 100, the critical probability p* converges on about .055. As the number of nodes increases, the abruptness becomes steeper and steeper. However, there is a convergence where the number of nodes no longer matters to the critical probability. For large networks, it is very likely that the graph will be connected. This probability testing is useful for applications involving networks: telephones, broadcasting, and the internet rely on this idea. It turns out there's a good chance the network of interest is connected. An interesting exercise would be to see by how many steps are individuals connected with one another in these networks. We will visit this idea in a later chapter.

Chapter 2

Algorithm Analysis

The analysis of algorithms is a branch of computer science focused on considering the performance of algorithms which is useful for optomizing design and processing time. As we begin to create large program investigations, our understanding of algorithm analysis will be very important to our efficiency. After all, time is a valuable commodity, and who wants to waste it?

The relative value of studying algorithm analysis has to do with the size of the problem. In many cases the advantages of a variety of implementations go unnoticed because the processing time is negligible on our human operating scales. However, when we wish to investigate large quantities of items, the algorithm we choose will make the difference between sitting and twiddling our thumbs, or moving through experiments quickly.

2.1 Considering Order of Growth

The time it takes for an algorithm to complete its task is directly dependent on the number of steps the program must run through. When comparing two programs, we chart the run time as a function of problem size and compare the function's asymptotes. Functions have the same order of growth if their asymptotic growth is equivalent. Two functions that grow quadratically will have the same order of growth. A function that grows linearly and another that grows logarithmically will not. Basically, the order of growth of a function is based on the leading terms of the function. As the problem size grows, other terms will become negligible compared to the contributions of the leading term. Big-Oh notation is a mathematical way to classify these types. Two functions with the same leading term will have the same Big-Oh notations: quadratics belong to $O(n^2)$, etc. Mathematically, we can say if f is in $O(g)$ then $a * f + b$ will also be in $O(g)$.If $f1$ is in $O(g)$ and $f2$ is in $O(h)$, $f1+f2$ will be in $O(g+h)$, and $f1 * f2$ will be in $O(g * h)$.We can use the idea of order of growth to study the implementation of various Python methods.

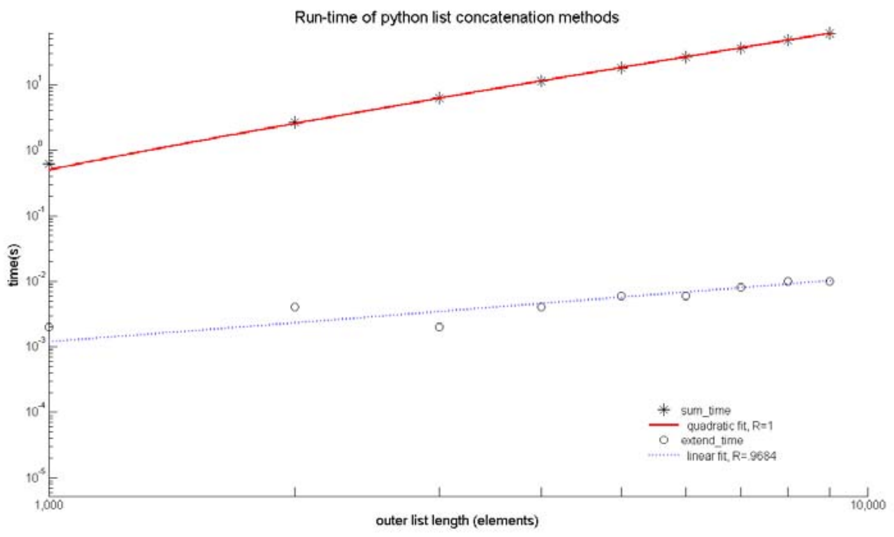
2.2 The Efficiency of Python Functions

One of the most basic operations in programming is the concatenation of lists. This ability allows us to string data together for transportation and ease of use. `sum` and `extend` are two simple functions provided by Python to do just this. Both have linear orders of growth when concatenating a list of single value items such as strings, tuples, integers, etc. However, when concatenating a list of lists, Python's two implementations diverge, and it turns out one is much better than the other. In the following experiment we compare the performance of `sum` and `extend` on a list of lists.

An experiment was set up to compare the sum and extend implementation in Python on a list of lists. To conduct this experiment we use Python functions `sum`, `extend`, and `etime`. The function `etime` initiates a timer which returns system and user time. The following is an excerpt of the code implementation used in the experiment:

```
###the sum algorithm timed:
           start=etime()
           sum(outer_list,[])
           end=etime()

###the extend algorithm timed:
            a=[]
            start=etime()
            for inner in outer_list:
                a.extend(inner)
            end=etime()
```

We define `outer_list` to be a list of a given length where each item within it is a list of length 100. The item values of the internal list is the integer 0. I ran the experiment for a range of lengths between 1000 and 9000. 10 trials were run for each list length for the given function. The results are given in the following graph:



From the figure we can see that the processing time of `sum` is quadratic while `extend` is linear. To note, the `extend` method continues to operate on a 10^{-3} seconds timescale, while the run time for 9000 lists in a list is 60s. I ran the program out of idle curiousity for a list of length 15,000 and found the run time to be 430s. Imagine spending 40 minutes just to run 5 trials on that! As list lengths become large, using the `extend` algorithm will return the same result in a much quicker manner. The extend performance is better. It's best to leave `sum` to its primary job as an adder of integers rather than use it as a concatenator.

Chapter 3

Small World Graphs

3.1 FIFO structures

In the previous chapter, we determined the orders of growth for simple list algorithms. We found that large-scale problems can quickly grow to $O(n^2)$, and terribly inefficient algorithms can grow as badly as $O(\infty)$. Ideal performance would be one of constant time regardless of data size. The graphs that we have created thus far have been relatively small. As we try to move to modeling physical phenomena, the number of vertices and edges of the graphs we use will be much larger.The Breadth-First-Search algorithm discussed in chapter 1 has an order of growth of, $O(|V| + |E|)$, as the search is directly related to marking vertices and comparing edges between vertices. In order to keep this growth small, we try to keep all operations (adding/removing vertices from a queue and marking/checking edges) in constant time. A first-in first-out structure (FIFO) achieves this performance.

In a FIFO, the data structure does two things: it keeps track of the front and rear of the lists. Additions to the list occur at the rear and removals occur from the front. This means no time is wasted in traversing the set to find the item we desire.One such convenient FIFO structure is the doubly-linked list. Each element in the list is an object, where the objects are strung to one-another. Every object has references to the object that comes before and after it. This data structure is useful for adding and removing items anywhere in the list because we never have to re-index the list values, saving us time.

A simple implementation of this FIFO can be achieved in Python. We build a doubly-linked list with nodes that have references to its adjacent nodes, and the list object has a head and tail. The two operations we are capable of is appending and popping (removing) items from the list.The important item to track is the rewiring of nodes during the addition and removal of the objects:

```
def append(self,n):
       '''adds a Node to the list. In FIFO, we default node placement
          to after the last node. Otherwise, we can put in node N
          anywhere following a neighbor node'''

       if self.next_in==None:
           #the list is empty so add the first node
           n.prev=None
           n.next=None
           self.next_in=n
           self.next_out=n
       else:
           n.next=self.next_in.next
           n.prev=self.next_in
           self.next_in.next=n
           self.next_in=n
```

In the case of the empty list, the first node will reference `None` values and the head and tail of the list point at the same node. When adding more nodes, we use the tail to find the end of the list,and then rewire the references so that the new node is in fact inserted into the list with references to the old tail. This code could be modified to be more dynamic, for inserting objects anywhere in the list, but as written only works as a FIFO.

Removing from the list is simpler still. We rewire objects similar to the methods used in append, but the `node.prev` references `None`:

```
    def pop(self):
        '''removes the first node from the list'''
        if self.next_out==self.next_in:
            #there's only one node left in the list
            self.next_out=None
            self.next_in=None

        else:
            n=self.next_out
            self.next_out=n.next
            self.next_out.prev= None
            n.next=None
```

Lastly, we must have some way of knowing that the object we create is truly a doubly-linked list –a good computer scientist validates his/her work. Well, in a doubly linked list, the only way to know if we have achieved success is to traverse through the nodes and check:

1. A node's `next` attribute references a node whose prev attribute is itself.
2. A node's `prev` attribute references a node whose next attribute is itself.

The code within our DoublyLinked class that verifies this looks like this:

```
def is_doublylinked(self):
     a=self.next_out
     if a==None:
         return 'list is empty'
     else:
         while a.next !=None:
             b=a.next
             if b.prev != a:
                 return False
             a=b
         return True
```

Using this FIFO or another, we reduce list operations to constant time. Also, Python provides a structure that supports FIFO. The deque structure in the collections module can add and remove items in constant time, even when removing from the left. As we move into elaborate graph experiments this structure will save us valuable time.

3.2 Studying Small World Networks

Malcolm Gladwell has immortalized the findings of Stanley Milgram: there are at most six degrees of seperation between you and any other person in the nation. Living within a large nation filled with many different groups, we usually find this to be surprising and contradictory to intuition. However, while we keep most of our friends local, every one of us has just enough ”distant” friends to create a big network that keeps us well-connected. Stanley Milgram conducted a social experiment using real people to find a result.Duncan Watts and Steven Strogatz strove to characterize this type of phenomena using computational modeling.

We define a small world network to be one in which we have high clustering and low average path length. Physically, we can equate clustering to social cliques, or simply: how many close friends does an individual have, and are they friends with one another? The average path length is the equivalent of discussing degrees of seperation; how many neighbors does an individual have to network with to reach another in the network? Middle-schoolers, the greatest social strugglers of our time, will tell you that this data is extrememly important for understanding one’s social standing within the network of pre-adolescents (how many close friends do you have, and how far are you from the popular crowd?).

Watts and Strogatz developed a generative model to create their social networks. In chapter 1, we looked at a variety of graphs: the two significant constructions were the random graph and the regular graph. Most networks are some combination of the two, as nothing is completely random or perfect. Watts and Strogatz determined an algorithm to construct more realistic social networks to study social connectivity. In their method, we start with a regular graph of a certain degree and size. Then, edges are randomly re-wired by a given probability, p. If an edge is selected in the affirmative for re-wiring then the vertex is paired with a new vertex that it does not already have an edge with. Through this method we develop a graph that is a combination of the regular and random graphs.

The process the two scientists used to traverse the network was:

1. Created a regular graph of 1000 nodes and degree 10. The graph was arranged with a ring structure so that it would be clear how to traverse through the graph. Also, this organization has no random development to it.
2. They choose a node with an edge that connects it with its closest neighbor. They then randomly rewire the edge, while forbidding the possibility of creating repeat edges. The second step is repeated for each node using a clockwise traversal of the ring lattice.
3. This process is repeated, except now considering second-closest neighbor.
4. They lap through the lattice until every edge has been considered, which occurs after $k/2$ laps, where k is the degree of the original regular graph.

The Watts and Strogatz system uses a ring-lattice organization for a regular graph. In chapter 1, we developed a random regular graph generator, that does not do the job we want it for. In addition, there is a small chance we could create an unconnected graph should a node not be reached before all others achieve the degree requested for the graph (simple example: a graph with 5 vertices desiring a degree of 2 might not be achieved because a graph with 4 vertices can have degree 2, and can be reached before all 5 are connected). Thus, we borrow a regular graph algorithm developed by Allen Downey available at `http://greenteapress.com/compmod/Graph.py`.

Rather than using precisely Watts and Strogatz's algorithm, we can use something a little simpler to achieve approximately the same result. In chapter 1, we developed methods that could provide us with all the vertices, edges, out-edges, and out-vertices of the graph. Rather than lapping around our ring lattice structure $k/2$ times, we can simply traverse through the list of edges in the network, and randomly re-wire the edge to another vertex, thus checking every edge:

```
def rewire(self,p):
        '''rewire the graph randomly by the given probability'''
        vertices= self.vertices()
        edges= self.edges()
        for e in edges:
            n=random.random()
            if n < p: #if the number is within the probability of change
                v= e[0]
                w= e[1]
                while self.get_edge(v,w)!=None:
                    w=random.choice(vertices) #randomly choose a vertex
                self.remove_edge(e)
                e=Edge(v,w)
                self.add_edge(e)
```

Here we move through the list of edges, randomly choose it for rewiring, and then choose a new vertex that is not connected to the source if we need to rewire. **Challenge**: compare the two rewiring technques and see if they achieve the same results.

In our graphs, we wish to study the clustering coefficient to determine how strongly groups are affiliated with one another. We can do this by traversing through every individual in the network and determining whether its friends are also friends (i.e. checking to see whether all the nodes an individual is directly connected with are also directly connected). To qualify the weight of a clustering value, we must compare it to the possible number of direct connections that exist. Thus, we define $C(p)$ as the number of friendships that exist divided by the total possible friendships for some given probability.The following method achieves this performance:

```
    def cluster_coefficient(self):
        vertices= self.vertices()
        Cp=[]

        for v in vertices:
            neighbors=self.out_vertices(v)

            k_v=len(neighbors)
            m=k_v*(k_v-1.0)/2.0 #maximum connections possible

            if m==0.0: #if the vertex is unconnected
                Cp.append(0.0)
                continue

            else:
                a=0 #initialize actual number of connections
                for i in range(len(neighbors)):
                    for j in range(i+1,len(neighbors)):

                        e=self.get_edge(neighbors[i],neighbors[j])
                        if e != None:
                            a+=1.0
                Cp.append(a/m)

        return sum(Cp)/len(Cp)
```

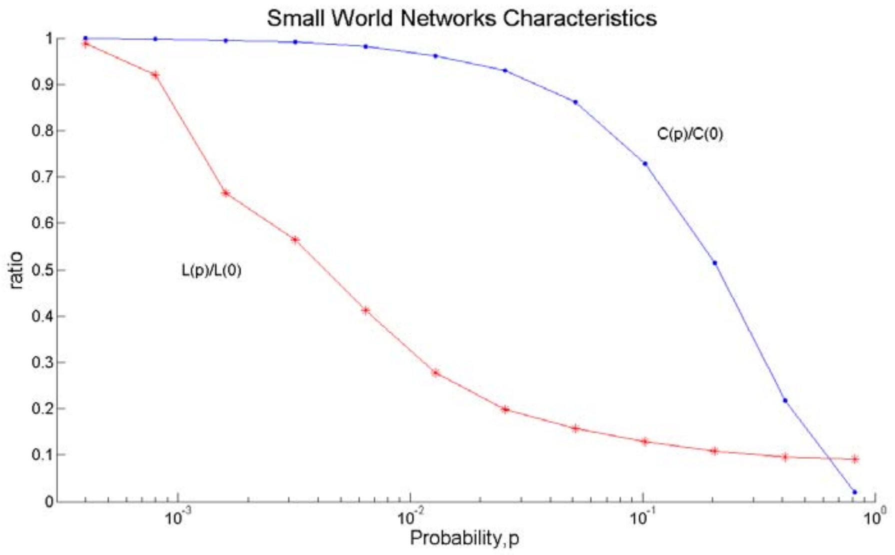
The trickier thing to determine is the average path length, $L(p)$, for a graph with some given probability. Similar to our BFS algorithm used to determine connectedness in chapter 1, we can implement Djikstra's algorithm to find the distances between one node and all other connected members of the graph. In this method, we mark all vertices with an infinite distance from one another. Then we select a source node, mark it with a distance of 0 from itself, then add it to the queue. We remove a node from the queue and add any nodes it is directly connected with that has infinite distance to the queue (this means this is the first time we have reached the node). These child nodes have a distance of 1 greater than its parent. When the queue is empty, we have reached every node connected to the source. The process is illustrated below:

```
def djikstra(self,v):
        '''performs a djikstra search for one vertex and returns a list
        of all the shortest paths to every node'''

        vs=self.vertices()
        for w in vs:
            w.d=-1  #give every vertex a distance attribute of infinity

        v.d=0 #source
        queue=deque([v])#add source to the queue
        dist=[] #keep track of every vertex's distance from source

        while len(queue): #until the queue is empty
            b=queue.popleft()
            d=b.d

            if (d != 0): #if not the source vector
                dist.append(d)#add the distance from source to list

            bs=self.out_vertices(b)
            for vertex in bs:
                if vertex.d == -1:#if unlabeled mark and add to queue
                    vertex.d=d+1.0
                    queue.append(vertex)

        return dist
```

By repeating this process for every node, we can average every shortest path length between nodes to find the average path length for the graph.

3.3 Clustering and Path Lengths for Watts and Strogatz Experiment

Now that we have developed the tools to study small network graphs, we can experiment with the effect of probabilities on the generation of models by comparing the clustering and path length characteristics.I ran an experiment for 1000 nodes with degree 10 and a variety of re-wiring probabilities ranging from 10^{-4} to 10^{0}. Characteristic values for $p > 0$ were normalized by dividing by $C(0)$ and $L(0)$. 5 trials were run on each data point using the graph generation algorithms described in chapter 3. The results are given in the figure below:



We find that as a graph becomes more random both clustering and path length becomes smaller. So, what probabilities achieve a small world? We find that the onset of small-world is quick for small probabilities: we still retain $C(p) \approx C(0)$ and $L(p) << L(0)$. We have high clustering and short path length through a probability of .03, at which point clustering begins to drop off. The trends of our findings are identical to Strogatz and Watts, though more rigorous comparison must be done to determine the robustness of their model.

Chapter 4

Distributions and Scale Free Networks

Throughout this book, we have been concerned with modeling real phenomena and groups. We want to characterize the members of the group, their interactions, and predict behavior. One way to characterize a group is to look at its statistical distribution to determine the likelihood of an occurence, the rate of change within a group, determine an average, and more. Distributions are quick ways to determine what type of model best characterizes a set. In this chapter we use distributions to analyze the population of US cities and compare small world graphs discussed in chapter 3 with a new type of graph we will introduce later in chapter 4.

4.1 Statistical Distributions

Cumulative distributions describe the probability distribution of a set of values. This means that for a given number x, we can describe the likelihood that x or a smaller value exists in the set. This is much more descriptive than the simple use of offering an average value and deviations for a set. The cumulative distribution function (CDF) can give us values such as the percentiles and quantiles of a set. If we were trying to characterize the population distribution in the US, we could use the CDF to describe the number of towns with fewer than 100,000 individuals, greater than several million, etc.

Closed-form distributions are used when we are trying to determine whether a distribution has an exponential trend. The CDF of an exponential distribution is given as $cdf(x) = 1 - e^{-\lambda x}$ and the complementary CDF (CCDF), is given as $CCDF(x) = 1 - cdf(x)$. Therefore if we plot a graph semilog-y, an exponential distribution would visually yield a straight line with slope $-\lambda$. Plotting the

CCDF allows us to determine for what range a given set of values has exponential growth or decay.

We can use the CDF and CCDF to determine whether a set of values follows a Pareto distribution. In a set of values that follow this form, we find the set contains many small values and a few rather large values. This property is sometimes referred to as the long tail distribution. Because the span of a set with a long tail is rather large, it is difficult to say that the range of values is centered around some standard value, or scale. Thus, this type of distribution is scale-free. In line with this long-tail observation, we find that the bulk of the value is distributed only among a few. For example, if were discussing the distribution of wealth, we would find that most people make very little money, and a few have millions and billions, and that most of the money in the world is held among a very elite few.

The CDF of a Pareto distribution is defined as $1-(x/x_m)^{-\alpha}$. Plotting a log-log of the CCDF would therefore yield a line with slope $-\alpha$, the shape parameter of the distribution, and intercept $-\alpha log(x_m)$. Therfore, if were to plot the CCDF of a given set of values we could determine whether the set followed a Pareto distribution.

4.2 Distribution Tools

In order to plot distributions and determine percentiles and quantiles of datasets, we can implement a `Dist` class in python that intakes a histogram and calculates a distribution. We achieved this using the following code:

```
class Dist(list):
    def __init__(self,h):

        self.distribution(h)

    def distribution(self,h):
        '''takes a histogram and converts to a distribution
           that is stored as lists'''
        a=0
        self.qs=[] #quantity
        self.cfs=[]#cumulative frequency
        self.ps=[] #percentile
        for (g,f) in sorted(h.iteritems()):
            self.qs.append(g)
            self.cfs.append(a+f)

            a=a+f
        for i in range(len(self.cfs)):
            self.ps.append(self.percentage(self.cfs[i]))
    def percentage(self,c):
        '''converts cumulative frequency to percentile'''
        length=max(self.cfs)
        return float(c)/length
```

From this class, we know the quantities in the set, and the cumulative frequency of a value's occurrence (i.e. for value x, how many numbers in the set are equal to or less than x). We can use this to calculate percentiles:

```
  def percentile(self,g):
        '''takes a quantity and returns the percentile
           that quantity falls within'''
        if g<self.qs[0]:
            return 0
        if g>self.qs[len(self.qs)-1]:
            return 1.0
        else:
            i=bisect.bisect(self.qs,g)
            return self.ps[i]
```

We can also do the opposite, and find out what quantities fall under a percentage threshold. This is useful for calculating the quantiles, which are frequently used to characterize the distribution of individuals.

```
   def quantile(self,p):
        """takes a percentage as input and returns
           the corresponding quantity"""
        i=bisect.bisect_left(self.ps,p)
        return self.qs[i]
```

We can also build a step-wise CDF of this data by mapping the quantities with the cumulative frequencies:

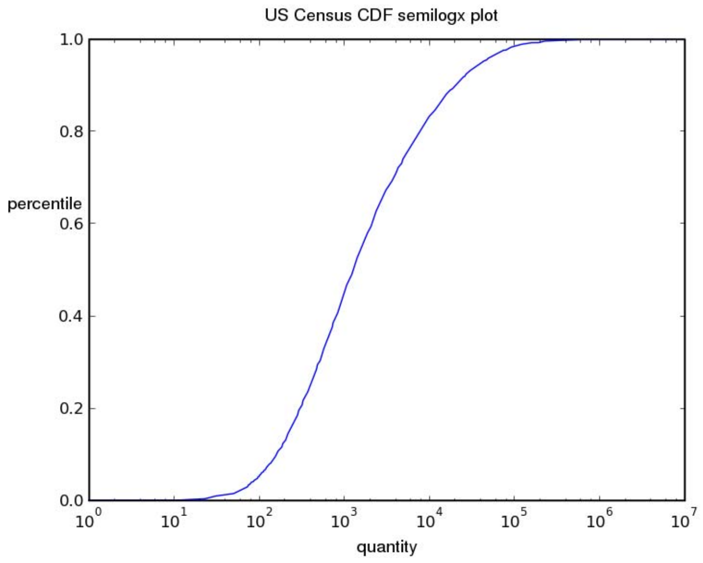
```
    def cdf(self):
       '''takes the distribution and converts it to a CDF'''
        q=[]
        c=[]
        a=0.0
        for i in range(len(self.qs)):
            q.extend([self.qs[i]]*2)
            c.extend([a,self.ps[i]])
            a=self.ps[i]
        return q,c
```

Because the quantity spans a range of percentage occurences we keep track of the endpoints. For example, say we have a data set [1,2,2,4,5]. 10 percent of the data set is below 1, and 30 percent is below 2. The percentage of 2's occurence is 20 percent. In a stepwise, to capture the change, we plot(2,.1) and (2,.3) to continue to make the distinction that 30 percent of values are below 2, and 10 percent of values are below 1.

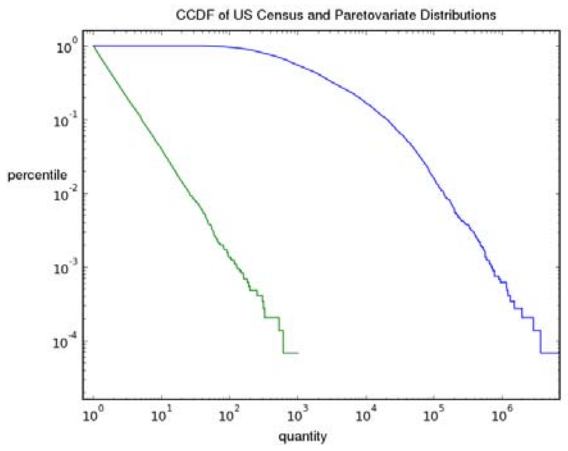
Now that we have these tools, we can characterize different sets of information.

4.3 US Population Distribution

The distribution of city and town populations has been proposed to be Pareto. Using the dataset posted by the U.S. Census Bureau, we determined whether this was true. For the 14,593 cities and towns in the US, we calculated the median to be 1276 individuals in a city, with the 25th percentile at 400, and the 75th percentile as 5335 using the quantile code from the previous section. This means that most cities are actually small towns in the US. Plotting this data as a CDF on a semilog-x scale showed:



The distribution has a long tail as 80 percent of cities have less than 10^5 people. The CDF shows a long tail, characteristic of a Pareto distribution. We can confirm our suspicions by plotting this data as a CCDF on a log-log plot:



The census (blue line) distribution has a straight line for the range roughly from 10^3 to 10^6 quantities with some noise at the ends as this is a finite data set. From the CCDF we calculated the shape parameter to be 1.42 and the `x_m` to be $10^(3.81)$. Using the shape parameter we can create a Pareto distribution using Python's `random.paretovariate` number generator. We plotted the random CCDF (shown in green) against our US Census CCDF. Both lines have similar slopes, indicating that the US census distribution is Pareto. Thus we conclude, that of the 14593 cities, most people live in a small number of major cities, and most cities are really very small towns. This also indicates that the distribution of populations is scale-free –we could see this trend for a wide variety of ranges and country sizes/populations.

4.4 Barabási and Albert Graphs

In chapter 3, we went through an exploration of small world graphs, in which there is high clustering and low average path length. This algorithm developed by Watts and Strogatz, was later challenged by Barabási and Albert. While the small world graph begins to reduce path length and better connect individuals than say a random or regular graph, we are missing one reality: individuals in a network tend to link up with those who are more connected. For example, within a company, you would rather try and be directly connected with the executive board than a middle-management boss. You would rather cite a frequently cited article for a scholarly journal, than an obscure source. You are more likely to want to know people that can connect you quickly with other people.

Thus Barabási and Albert developed a different algorithm for creating a network of nodes and edges: they said that each node has preferential attachment.They were interested in finding how likely a node would have a certain number of edges, characterized by k and $P(k)$, where k is the degree a node has, and $P(k)$ is the probability that a vertex has degree k. When Barabási and Albert did their study of these graphs, they found a power law connectivity, that is as k becomes very large, $P(k)$ becomes asymptotic to $k^{-\gamma}$ where γ represents the rate of decay. Graphs that follow a power law are called scale-free networks (Note: not the same as scale-free distributions).

In order to build their graph, they created a small world graph with `m_o` vertices, and then added additional vertices with initial attachment of degree `m` where $m <= m_o$. The new vertex,v, connects to an existing vertex,w, based on a probability calculated by the number of connections w has divided by the total number of connections in the graph.

Code for implementing this algorithm would look something like this for connecting a vertex:

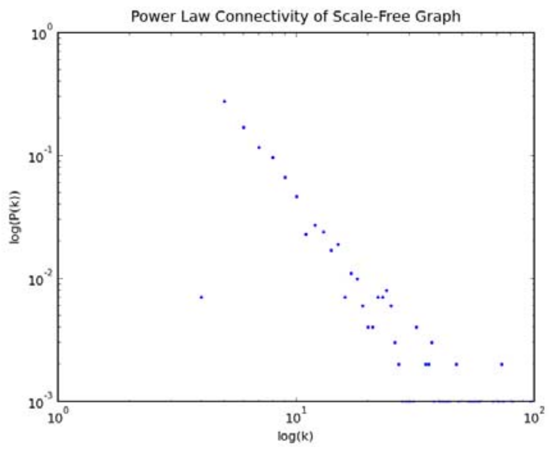
```
def connect_vertex(self, v,m):
    '''takes a vertex and connects it with degree m to
    other vertices in the existing graph'''
    ws=self.vertices()
    self.add_vertex(v)
    c=0
    while c<m:
        w=random.choice(ws)
        if self.get_edge(v,w)== None:
            n=random.random()
            if n<self.probability(w):
                e=Edge(v,w)
                self.add_edge(e)
                c+=1

def probability(self,v):
    '''calculates the probability of connecting to
    vertex based on BA algorithm'''
    kj=len(self.edges())
    ki=len(self.out_edges(v))
    if ki==0: #in the case there are no connections
        return 1.0 #guarantee a connection
    p=float(ki)/kj
    return p
```

The problem with this implementation is that it is slow, and in the worst case has an order of growth of $O(\infty)$. When the program was run, I found it took 10-15 minutes to create a 200 node graph.

Therefore I implemented a different idea. Preferential attachment simply means a vertex is more likely to connect with something that is well-connected. Barabási and Albert developed an algorithm that looked at individuals in a network, rather than the connections. Here, I use a different idea: rather than seek out individuals and calculate a probability of attachment based on connectedness, randomly choose existing edges in a graph and then connect randomly to either of the vertices that build the edge. This works because a vertex with more connections will have more edges, and the probability of choosing some vertex by its edge is equivalent to the number of edges the vertex has divided by the total number of edges in the graph. An implementation of this is cleaner and faster:

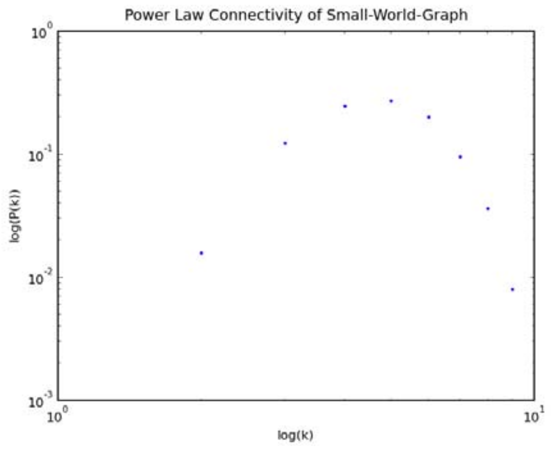
```
def connect_vertex(self,v,m):
    es=self.edges()
    self.add_vertex(v)
    c=0
    while c<m:
        try:
            e=random.choice(es)
            w=random.choice(e)
            if self.get_edge(v,w)==None:
                e=Edge(v,w)
                self.add_edge(e)
                c+=1
        except: #add edges if its an edgeless graph
            for w in self.vertices():
                e=Edge(v,w)
                self.add_edge(e)
            c=m
```

The building of this graph is now reduced to roughly $O(n)$ time, where n represents the number of vertices in the final graph. Running a 1000 node graph took less than 15 minutes! Using this graph-building algorithm, we attempted to build a scale-free network and compare characteristics with Barabási-Albert's research. The plot below graphs $P(k)$ vs. k for a BA graph of 1000 vertices, and m and `m_o` of value 5:



We find that even on this small scale we have the bottom clustering that the Barabási and Albert found in their own work. This graph, produced by our algorithm, has the same characteristics as the 150,000 vertex graph built by Barabási and Albert. Most individuals are only connected to the graph by a small degree, but a few have many connections. There is one blimp to the data, that is attributed to the initial building process where some of the initial data points are not connected to the graph, and so they end up being less and less probable to attach to, and therfore maintain a small number of connections.

How does this compare with the Watts and Strogatz model? We ran a 3000 vertex small world graph as described in chapter 3, giving each vertex degree 5, and rewiring with a 50 percent probability. We found a log-log plot of $P(k)$ vs. k to look like this:



The power law connectivity of a Watts-Strogatz graph is not distributed as a straight line, showing a more parabolic distribution of connections. It is difficult to characterize a rate of decay for this plot, as the graph is not very straight. However compared to the Barabási-Albert graph this ascent and descent is more steep indicating that most individuals are connected equivalently.

In chapter 3 we calculated the clustering coefficient and average path length for small world graphs. For our Barabási-Albert 1000 vertex graph, we calculated a clustering coefficient of .03, and an average path length 3.0. In a small-world graph, regardless of probability, the path length is greater: 4.7 for 10 percent probability of re-wiring, and 6.4 for a 50 percent probability of swapping. The clustering coefficient is also greater in the small world: .04 for 10 percent, and .22 for 50 percent probability. This means that in the Barabási-Albert world we don't need as many friends to be connected with everyone else –we just need to know the few who know everyone else. From this deviation in values, we can ascertain that the scale-free networks have similar path length coefficients but not clustering coefficients to the small-world graphs.

Chapter 5

Cellular Automata

Thus far, we have spent much of our time examining network interactions through graphs. However, computational modeling is much broader than this. We can also model and predict the behavior of individuals. Cellular automaton (CA) is a discrete model of a very simple physical world. Within it there are individuals, labeled as cells, that make simple computations to dictate its state as we move forward in discrete time. One example of an automaton is a cell that increments by 1 for each time-step; it counts.In the CAs we study in this chapter, the cellular automaton will be dictated by rules that allow it to live as a 0 or 1 in space. Most CAs are deterministic; they will always execute in the same manner. The rules and behavior of cellular automata can be used to implement computing algorithms including random number generators!

5.1 Rule 50

Stephen Wolfram studied the dimensionality of CAs. A CA has dimensionality if it is part of some "neighborhood" arranged in some contiguous space. In one-dimensional CAs, a neighborhood includes: the cell itself and the cells directly to the right and left. The evolution of the CA system is decided based on the state of the neighborhood and the rules of evolution given by the programmer. For example, here's a set of rules that would dictate the evolution of individual automata based on their neighborhoods:

prev	111	110	101	100	011	010	001	000
next	0	0	1	1	0	0	1	0

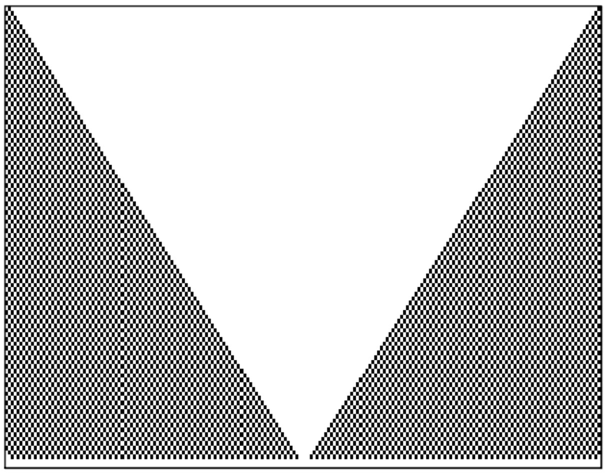
For a neigborhood that can be defined as 3 individuals, each with 2 possible states, we have 8 possible states (which can be written using hexadecimal notation). Each type of neighborhood will yield the cell in question to progress with a certain state. Because the progression of states is some sort of combination of

0s and 1s, we can represent this set using binary notation. 00110010 in binary becomes 50 in decimal, so we call this rule 50.

There are different ways we can configure a CA in space. We can have a finite array, in which every individual has two neighbors, except for the first and last. Alternatively, we can have an infinite sequence where there are an infinite number of cells arranged in a row. As a compromise to these two, we can have a ring structure, where the last cell wraps-around to act as a neighbor to the first and vice-versa. Allen Downey's CA class is designed as a finite sequence. We can modify this to a ring configuration by creating a class called CAwrap that inherits from Downey's CA:

```
class CAwrap(CA):
   '''creates a wrapping cellular automata
    def __init__(self,n=100):
        CA.__init__(self,n+2)

    def step(self):

        i=self.next
        self.array[i-1,0]=self.array[i-1,-2]
        self.array[i-1,-1]=self.array[i-1,1]
        CA.step(self)
```

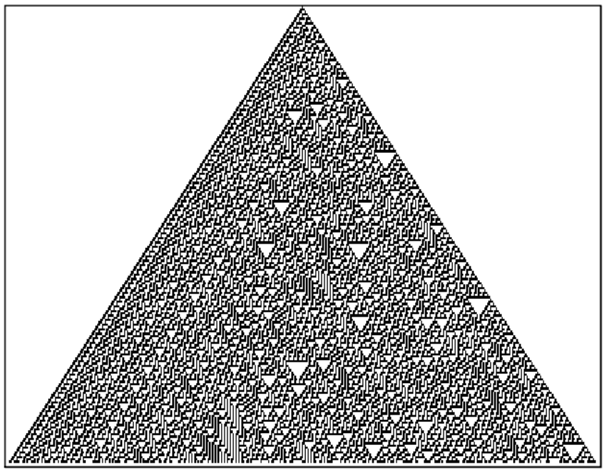
In this class, we use the first and lass elements as "ghost cells" where we reference the ends of the array to the opposite ghost cells. We can create a rule 50 CA that looks like this:



Rule 50 is an elementary CA because it is classified by its uniformity, and will yield the same pattern over time, regardless of most initial conditions. Classified as a type 2 CA, it has a nested structure (that is the pattern contains many smaller pattern versions of itself). Rule 18 is also another example of a Class 2 CA.

5.2 Rule 30

Class 3 CAs are much more complex than class 2. Rule 30,a member of class 3, seems to yield a random spread as we move from left to right:

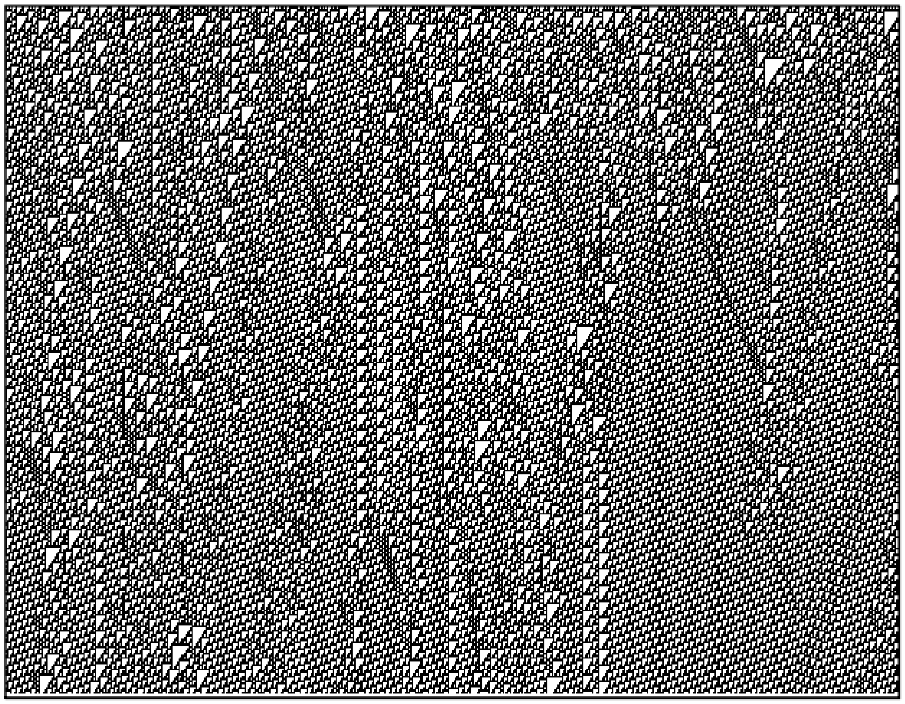


The above class 30 image was creating using CAwrap as described in the previous chapter with an initial black pixel (or a 1) in the center. Examining the center column,if we consider it to be a sequence of bits, the sequence appears to be completely random. Wolfram observed this idea and implemented rule 30 in a pseudo-random number generator (PRNGs) program. PRNGs are statistically similar to truly random sequences, making this algorithm appealing to computational programs. Rule 30 is used in Wolfram's Mathematica.

Challenge: Characterize the sequence given by the center column. Is this PRNG acceptable to pass as a random process?

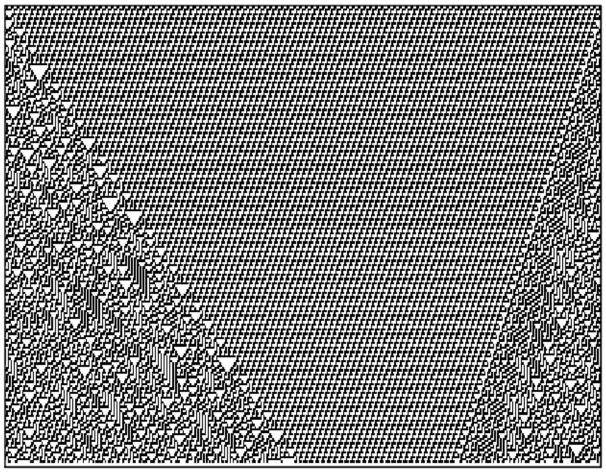
5.3 Rule 110

Perhaps the strangest CA of all is rule 110, a class 4 CA that will blow your mind. The following image was created for 300 timesteps with random initial conditions.

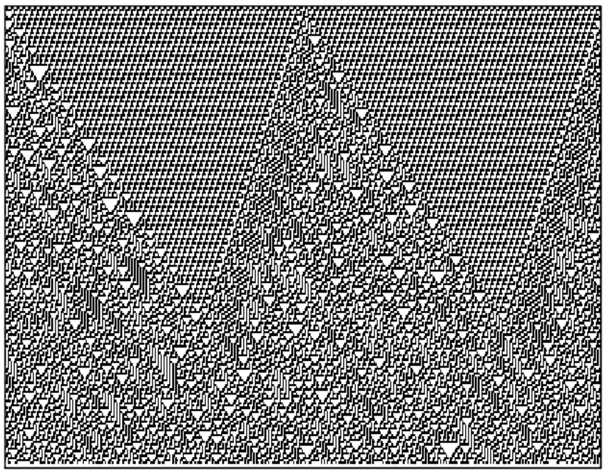


Rule 110 CAs can take any outcome for any set of initial conditions, making it indeterminate. Notice that there are regions that have the same pattern: we call them stable because they do not transform. Then there are diagonal breaks that propogate a non-uniformity: these are referred to as spaceships. When spaceships collide many outcomes occur: the two are destroyed, both continue, or only one continues to propogate. While rule 110 appears to be random and uneasy to characterize, this spaceship behavior makes rule 110 a powerful computational resource. If we consider that spaceships are signals, the collisions could be regarded as logical operators AND or OR. On a computational level, this is what it means to implement logic.

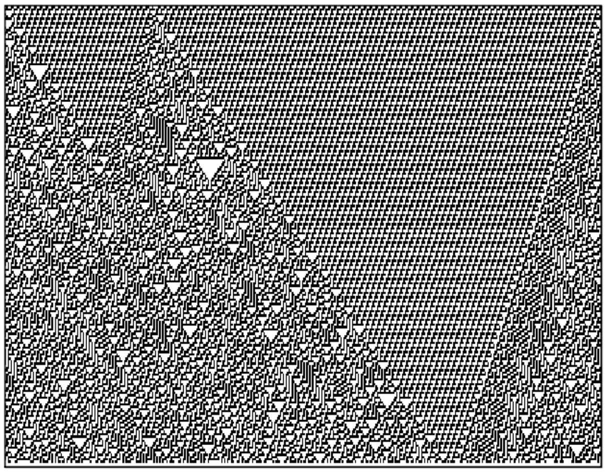
We did some investigation of initial conditions that would yield a stable background. An alternating condition of zeros and ones yields a somewhat stable background. We set up an initial condition of pixels alternating: 1100110011 (two darks followed by two lights). For 200 CA, we found this outcome:



We see that there is some edge behavior that prevents the entire background from stabilizing, but we have a significant stable region. We can use this region to study what initial conditions yield spaceships. If we change a single pixel value, we find that the pattern immediately runs into chaos. We changed the middle pixel from a 0 to a 1, and immediately saw the chaos propogate at a rate of 3 pixels/discrete iteration of time.Interestingly enough, it looks about the same when 3 of the central pixels are set to 1, and break the pattern. This is shown in the figure below:



I decided to see if things change if we shift the location of the "bad" CA. I found that changing the 50th CA to a 0, yielded a similar result:



However, here we see a collision with the edge chaos. The spaceship collides and disappears, while the edge spaceship continues to survive. The edge behavior is dominant.

The study of rule 110 is unlimited. Computer scientists are still studying and characterizing this phenomena. Here we have sampled the possibilities of one-dimensional cellular automata. In the following chapter we will delve into two-dimensional CAs!

Chapter 6

2D Cellular Automata and Game of Life

In the previous chapter, we modeled the evolution of one-dimensional cellular automata for a variety of rules and initial conditions. Here we study the evolution of a grid of cellular automata, where an individual's evolution is dependent on itself and the state of all 8 neighbors. In order to model these two-dimensional CA, we will first discuss abstract classes and Python's imaging capabilities.

6.1 Abstract Classes

Many programmers find that they have a variety of methods for implementing the same method. For example, the CAs displayed in chapter 5 were generated using Pylab by the following method:

```
    def draw(self):
        """draw the CA using pylab.pcolor.  flipud puts the first
        row at the top; negating it makes the ones black."""
        pylab.gray()
        pylab.pcolor(-flipud(self.array))
        pylab.axis([0, self.m, 0, self.n])
        pylab.xticks([])
        pylab.yticks([])
pylab.show()
```

Pylab's methods for drawings CAs are simple; Pylab was designed to convert array data to rectangular representations. However, the implementation is slow for large CAs. Python is equipped with many packages for image display. The Python Imaging Library (PIL) provides imaging processing capabilities to the Python interpreter, and is designed to be efficient in image generation for Python. We can use PIL to do the same job Pylab has provided us. Whenever we find it necessary to develop multiple ways to perform the same task, it is useful to consider developing an abstract type, a class definition that defines an interface. Abstract types are useful in allowing the programmer to define and enforce a certain set of capabilities.

For example, in chapter 5 we used Pylab to draw and show our CA. We could also use PIL or Python's EPS package to draw and show our CA. In addition, we might want to save our CA images using any of these packages. We can create a Drawer abstract class with these capabilities:

```
class Drawer(object):

    def __init__(self):
        """raises error if object is instantiated"""
        raise TypeError

    def draw(self, ca):
        """draw a representation of cellular automaton (ca). This function
        generally has no visible effect"""

    def show(self):
        """display the representation  on the screen, if possible"""

    def save(self, filename):
        """save the representation of the CA in (filename)"""
```

The Drawer class is made up of methods that are docstrings that describe the capabilities of its subclasses. You might notice here that the Drawer object has an instantiation method that raises an error. Typically, abstract types are supported by programming languages, so that the abstract type is naturally forbidden by the language to instantiate. However, Python does not support abstract types, so that an abstract class can be instantiated. While instantiating an abstract class is not very useful, we want the user to understand that he/she is not using this type correctly, so we artifically added an error to the initializing method.

Using the Drawer abstract class, we can create subclass implementations. We can convert our Pylab implementation from chapter 5 into PylabDrawer:

```
class PylabDrawer(Drawer):

    def __init__(self,ca,filename):
        self.draw(ca)
        self.show()
        self.save(filename)
    def draw(self,ca):
        pylab.gray()
        pylab.pcolor(-flipud(ca.array))
        pylab.axis([0,ca.m, 0, ca.n])
        pylab.xticks([])
        pylab.yticks([])

    def show(self):
        pylab.show()

    def save(self, filename):
        pylab.savefig(filename)
```

Using this method, within our own test codes, we only need the line `PylabDrawer(ca,filename)`, rather than all of the code we see here. In addition if we designed a PIL implementation, we would only need to swap out this line of code with `PILDrawer(ca,filename)`, provided we write a PIL drawing subclass. As we move through this chapter, we will implement PIL imaging to parse through our 2D CA.

6.2 Python Imaging Library, Gui, and Tkinter classes

The Python Imaging Library (PIL) provides image processing capabilities for Python. In the simplest implementation we can import the Image and ImageDraw class from the Image module, which gives us the capability to create, draw, and save images:

```
#inititate an image and draw object.
#mode L is black and white, size takes a width and height size in pixels
    image = Image.new(mode='L', size=[w, h], color= 'white')

#ImageDraw takes an image object and draws upon it
    draw = ImageDraw.Draw(image)

#draw a black rectangle of the given size at the x,y location
    x, y, width, height = 100, 50, 100, 100
    draw.rectangle([x, y, x+width, y+height], outline=0, fill='black')

#save the image
    image.save('filename.eps')
```

The ImageDraw class is equipped with the capabilities of drawing basic shapes for a variety of colors and thicknesses and fills. The rectangle method takes two points as its input (the two opposite corners) to draw the shape. In the Image class, coordinates are given with the upper left corner as (0,0) and the numbers become increasingly positive moving to the right and downwards.

Now we have an image object, but we have not displayed it yet. The image class technically provides a show method, so that for this image `image.show()` would display that image on the screen. This implementation of displaying an image is very inefficient, and so if we are trying to display the evolution of a CA over hundreds of evolutions, this would be very slow.

The solution here is to use Tkinter and a wrapper class developed by Allen Downey, Gui. Tk is Python's Graphical User Interface toolkit, and Tkinter is the interface to it. PIL is equipped with the Tk imaging module. The package python-imaging-tk must be installed in order to use it. ImageTk module provides an interface between the PIL image and Tkinter. We can use the PhotoImage class within ImageTk to create an image that is supported by any Tkinter objects that expect image objects:

```
tkpi = ImageTk.PhotoImage(image)
```

Downey's Gui class is simply a wrapper class for Tkinter, so that we can initiate a Gui object that can handle and display this tkpi. Tkinter has many widgets that support images. The most natural choice for displaying an updating image is the canvas. A canvas is easily updated, and we do not have to delete or destroy it in order to display a new image. Here is how one can implement the display of a CA:

```
import Image, ImageDraw, ImageTk, Gui, Tkinter, time

    gui = Gui.Gui() #instantiate Gui object
    canvas=gui.ca(width=w, height=h,bg='white',relief=Gui.FLAT)

    ## insert code here to draw the Image using Image
    and ImageDraw as described earlier in the chapter##

    tkpi = ImageTk.PhotoImage(image) #convert from PIL image to Tk image
    canvas.image([0,0],image=tkpi)   #show image on canvas
    gui.update() #displays canvas

    time.sleep(.01) #provide a pause
    gui.mainloop() #await user input
```

Between these classes, we have now created something dynamic and usable to display our CAs. We could take these methods and convert them directly into a PILDrawer, clearly breaking up the draw, show, and save methods as we have described earlier. We will use the PIL drawing capabilities as we model John Conway's Game of Life.

6.3 Game of Life

6.3.1 2D CA

In chapter 5, we wrote one dimensional CAs, i.e. the automata were arranged in a single row. In Conway's Game of Life, we deal with two-dimensional CA, i.e. the cells are arranged in a grid. In the 1D CA, our implementation required that each row display a time in the evolution, and the row number corresponded to the timestep of the evolution. Now that we have a grid, we need to use a structure that maps a time to a grid of data. Python's dictionary structure is an excellent choice for this:

```
class GameofLife:
    def __init__(self,n):
        '''creates a CA grid with n rows and m columns'''
        self.m=n+2 #ghost cells
        self.n=n+2

        #initialize dictionary to store array at each time step
        self.array={}
        self.array[0]=zeros((self.m,self.n), dtype=int8)
        self.next=1
```

The dictionary's keys map the time of the evolution to an array of `n` rows and `m` columns. We also created ghost cells that surround the entire grid. In the last chapter, we wrapped the CA to take care of boundary conditions. In the Game of Life, we can either have an infinite background, or wrap the cells in both directions, creating a torus. For the purposes of this investigation, we can use a toroidal grid:

```
def wrap(self):
        '''wraps the CA to take care of boundary conditions'''
        t=self.next

        #wrap all the rows with values (not ghost rows)
        for i in range(1,self.n-1):
            self.array[t-1][i,0]=self.array[t-1][i,-2]
            self.array[t-1][i,-1]=self.array[t-1][i,1]

        #wrap the top and bottom ghost rows (not including corners)
        for j in range(1,self.m-1):
            self.array[t-1][0,j]=self.array[t-1][-2,j]
            self.array[t-1][-1,j]=self.array[t-1][1,j]

        #wrap the corners
        self.array[t-1][0,0]=self.array[t-1][-2,-2]
        self.array[t-1][-1,0]=self.array[t-1][1,-2]
        self.array[t-1][0,-1]=self.array[t-1][-2,1]
        self.array[t-1][-1,-1]=self.array[t-1][1,1]
```

We wrapped the rows in exactly the same manner as in chapter 5, and then simply wrapped in the horizontal direction with similar notation. Finally, we include the corners, so that we have a completely wrapped CA. The main thing to keep track of in Python is the indexing. When we go left to right, we start indexing with 0, but if we travel right to left, we start with -1.

6.3.2 The rules and implementing the Game

In the Game of Life, the cells again have two states, dead or alive, represented in binary as a 0 or 1 and visually as white or black, respectively. A cell lives within a Moore neighborhood: a cell's neighbors are the ones that directly surround it (the east, west, north, south, and 4 diagonal cells). A cell's evolution is based on its state and the state of its neighbors without any consideration of the neighborhood's configuration; the Game of Life is totalistic. John Conway carefully chose the rules for the game of life in 1970 when he began his study and they are summarized in the list below:

1. If the cell is alive, and 2-3 of its neighbors are alive, then the cell will remain alive.
2. If the cell is alive, and 0-1 or 4-8 of its neighbors are alive, then the cell will die.
3. If the cell is dead, and 3 of its neighbors are alive, then the cell will come to life.
4. If the cell is dead, and 0-2 or 4-8 of its neighbors are alive, then the cell will remain dead.

Through these rules we find that there are 2 possible states a cell can be in (alive or dead) whose CA has 9 possible states (0-8 sum of neighbors alive), thus yielding 2^{18} rules for the CA! We can implement Conway's situational rules into our Python code:

```
def rules(neighborhood,ca):
    '''takes a ca's neighborhood to determine what the state of the ca
    will be in the future step (only when considering the 8 surrounding
    neighbors'''
    alive=0
    for x in range(3):
        for y in range(3):
            if (x,y)==(1,1):
                continue
            if neighborhood[x,y]==1:
                alive+=1

    if ca==1:
        if alive==2 or alive==3:
            return 1
        else:
            return 0
    else:
        if alive==3:
            return 1
        else:
            return 0
```

In addition we need some way to step from one time to the next, evolving the CA using these rules. In the 1D CA, we checked the state of each cell's neighborhood and then stepped forward based on the neighborhood rules. We do a similar thing with the 2D CA, careful of our boundary conditions:

```
def step(self):
        """each cell looks at its neighbors in
        current and steps to next state"""
        #take care of boundary by wrapping CA
        self.wrap()

        #iterate
        t=self.next
        self.next +=1
        self.array[t]=zeros((self.m,self.n),dtype=int8)

        for i in range(1,self.n-1):
            for j in range(1,self.m-1):
                ca=self.array[t-1][i,j]
                neighborhood=self.array[t-1][i-1:i+2,j-1:j+2]
                self.array[t][i,j]=rules(neighborhood,ca)
```

We loop through evolutions in the same manner as implemented for the 1D CA, except now traversing in both horizontal and vertical directions.

The drawing of the CA using PIL utilizes the methods described earlier. In order to draw the actual Image object, we have to determine the size of our CA in pixels based on our desired picture width and height:

```
w,h=500,500
duration=max(self.array.keys())#self.next-1

        for t in range(0,duration+1): #create an image for each time
            image = Image.new(mode='L', size=[w, h], color= 'white')
            draw = ImageDraw.Draw(image)
            width, height = w/(self.n-2), w/(self.m-2) #set cell width and height
            array=self.array[t][1:self.n-1,1:self.m-1]
            for i in xrange(0,self.n-2):
                for j in xrange(0,self.m-2):
                    if array[i,j]==1:
                        x=j*width
                        y=i*height
                        draw.rectangle([x, y, x+width, y+height], outline=0,
                                   fill='black')
```

From this image, we can display each image within the large for loop by converting the image to a Tk image and displaying it on a Gui canvas, updating the canvas as we iterate through each timestep of the CA.

Originally, when John Conway set out to study the Game of Life, he used a checkerboard to study the evolution of CA. This human study required a lot of patience and attention. With the power of computers, these studies become much faster and accurate, and therefore we are given the opportunity to explore more aspects of the CA.

6.4 Game of Life Patterns

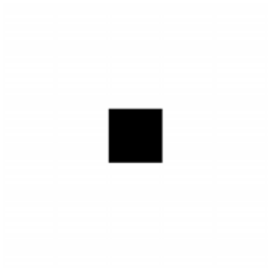
6.4.1 Stable Patterns

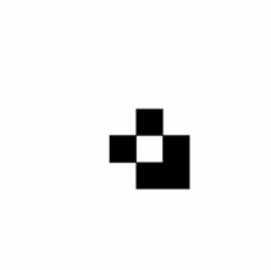
John Conway's main mission with the Game of Life was to determine whether all initial patterns would reach a stable state. We define stability of a cellular automata as when there is no longer new cellular growth. As he explored a variety of shapes, Conway found a number of stable shapes. In addition, many others have spent a great amount of time finding and naming stable patterns (`http://www.ericweisstein.com/encyclopedias/life/`). Some of the most common still lifes include: 4-cell blocks, boats, beehives,and loafs, as shown in the still-lifes figure. Once these automata configurations are achieved each evolution will yield the same shape. Try it for yourself!

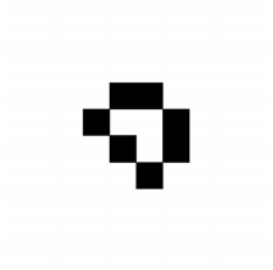
Stable states are not restricted to rigid bodies. Oscillators are considered stable, as there is no new growth, and the pattern cycles through a set of positions. Common two-period oscillators include the blinker, toad, and traffic light, shown in the oscillators figure. These shapes will alternate between 2 configurations until the end of time. The rules restrict them to this shape. In addition to period 2 oscillators, there are many shapes that oscillate for a variety of periods, all the way up to 100+!

6.4.2 Methuselahs

In the Game of Life any initial condition can lead to an interesting outcome. Initial conditions that take a ”long time” to stabilize are known as methuselahs. Through random exploration, we found a beautiful methuselah with a very simple initial condition. For an initial condition of one vertical row and one


(a) Block


(b) Boat


(c) Loaf

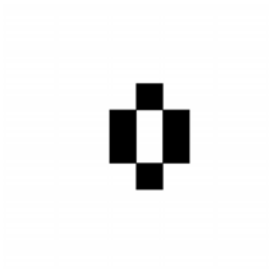
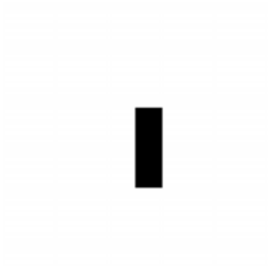
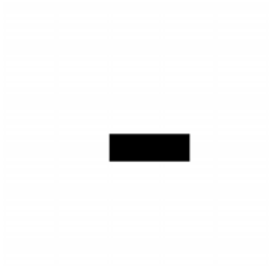
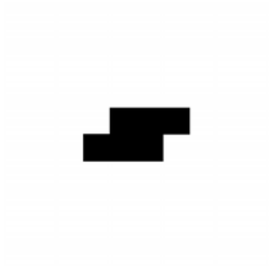
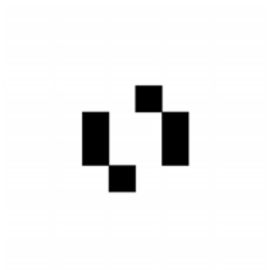

(d) Beehive

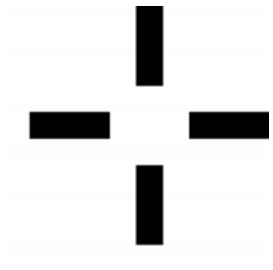
Figure 6.1: Common Still Lifes in Game of Life


(a) Blinker-0


(b) Blinker-1


(c) Toad-0


(d) Toad-1


(e) Traffic Light-0

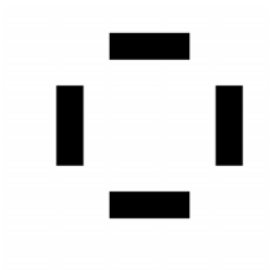
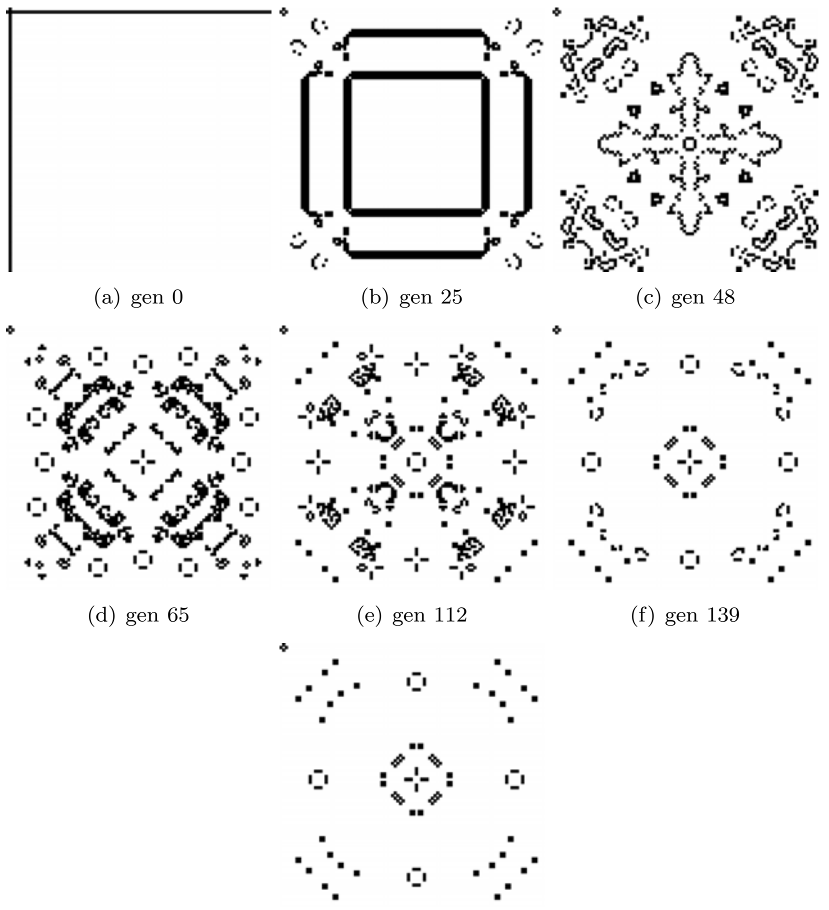

(f) Traffic Light-1

Figure 6.2: Common Period 2 in Game of Life



(a) gen 0 (b) gen 25 (c) gen 48

(d) gen 65 (e) gen 112 (f) gen 139

(g) gen 227

Figure 6.3: Evolution of a Random Methuselah

horizontal live row on a wrapped 100x100 cell grid, we get a beautiful pattern that takes 227 generations to stabilize, shown in the Random Methuselah . In generation 65, we begin to see stable lifes with the emergence of 12 traffic lights. In generation 112, the outer boundary stabilizes with 16 4-cell blocks that do not change as the CA evoloves. In image 139, a lot of the CA's acne (noisy cellular automata that ultimately die out and disappear) has cleared up, and we are left with 5 blinkers. Finally, in generation 227, we completely stabilize, and are left with 40 4-cell blocks, 5 traffic lights, and the 4 ribbons that encase the central traffic light. All that from 2 rows of live cells in the initial generation!

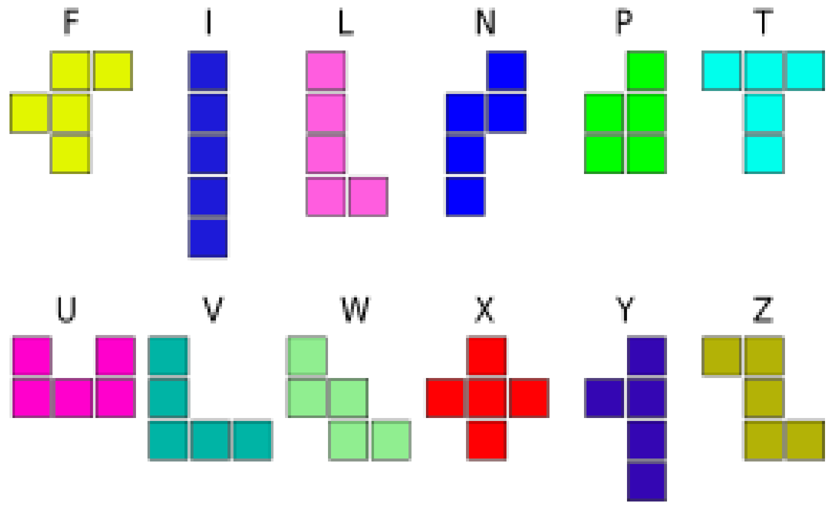


Figure 6.4: 12 Pentominoes using Conway's Nomenclature

6.4.3 Pentominoes

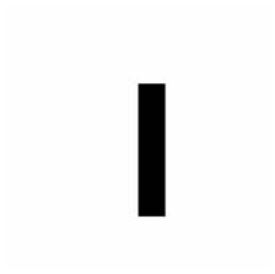
Polyominoes are finite collections of orthogonally connected cells (from http://www.argentum.freeserve.co.uk/lex.htm). Solomon Golomb began the study of polyominoes in 1953. There are 12 pentominoes, 5-cell polyominoes, that can fill a rectangular grid, shown in the pentominoes figure. These shapes once posed the mathematical challenge of finding all possible configurations to organize all 12 shapes within rectangular spaces (sort of like glorified tangrams), .

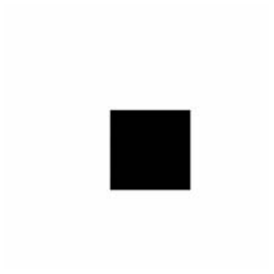
Conway used pentominoes in his first study of the game, as they are easy to track on a checkerboard. In the Scientific American article that popularized Conway's Game of Life, Gardner encouraged readers to explore the life outcomes for the 12 Pentominoes. Using our program, we can take on that challenge. To study the simpler pentominoes, we used a 10x10 grid, run until stabilization.

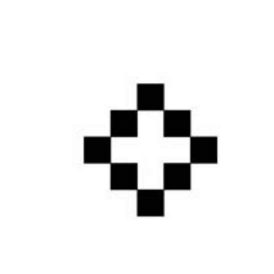
There are 4 pentominoes that form traffic lights through their evolution. The O-pentomino achieves this stable state after 6 generations, Q after 9, T after 10, and X after 6.

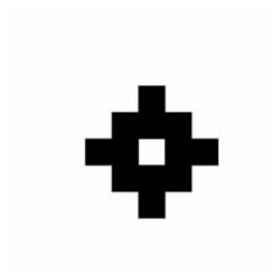
There are 2 pentominoes that become loafs. The V and W pentominoes become loafs after 3 and 2 generations, respectively. The V-pentomino becomes the W-pentomino after one evolution, shown in the loafs figure.

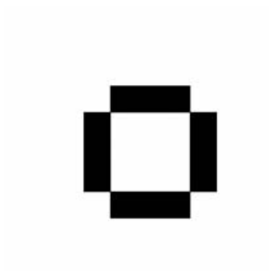
There are 5 pentominoes that simply disappear. The P-pentomino disappears after 4 generations, shown in figure P-pentomino figure. The S-pentomino disappears after 5 generations, shown in the S-pentomino figure.

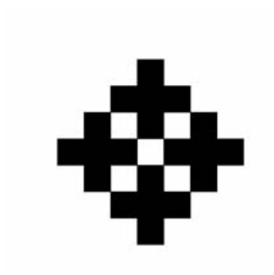

(a) O 0


(b) O 1


(c) O 2


(d) O 3


(e) O 4


(f) O 5

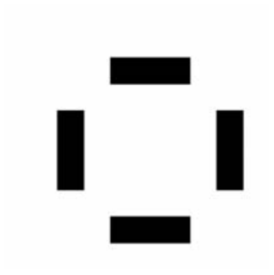
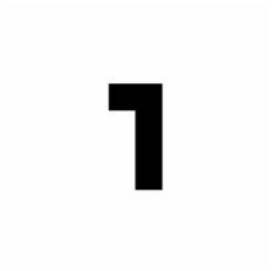
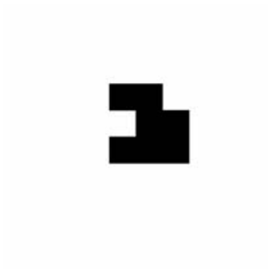
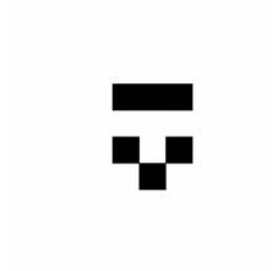
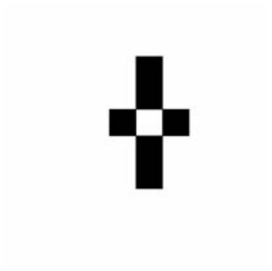

(g) O 6

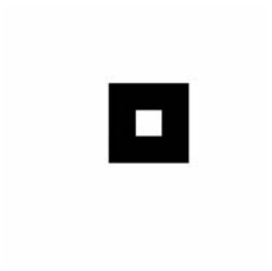
Figure 6.5: O-Pentomino evolution

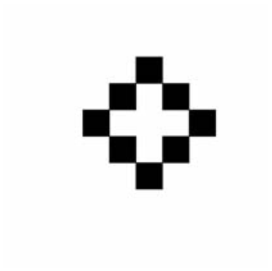

(a) Q 0

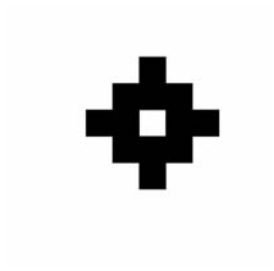

(b) Q 1

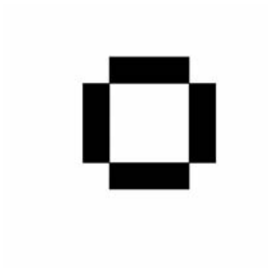

(c) Q 2

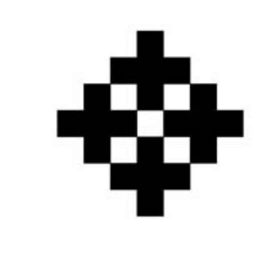

(d) Q 3


(e) Q 4


(f) Q 5


(g) Q 6


(h) Q 7


(i) Q 8

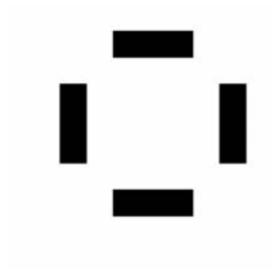
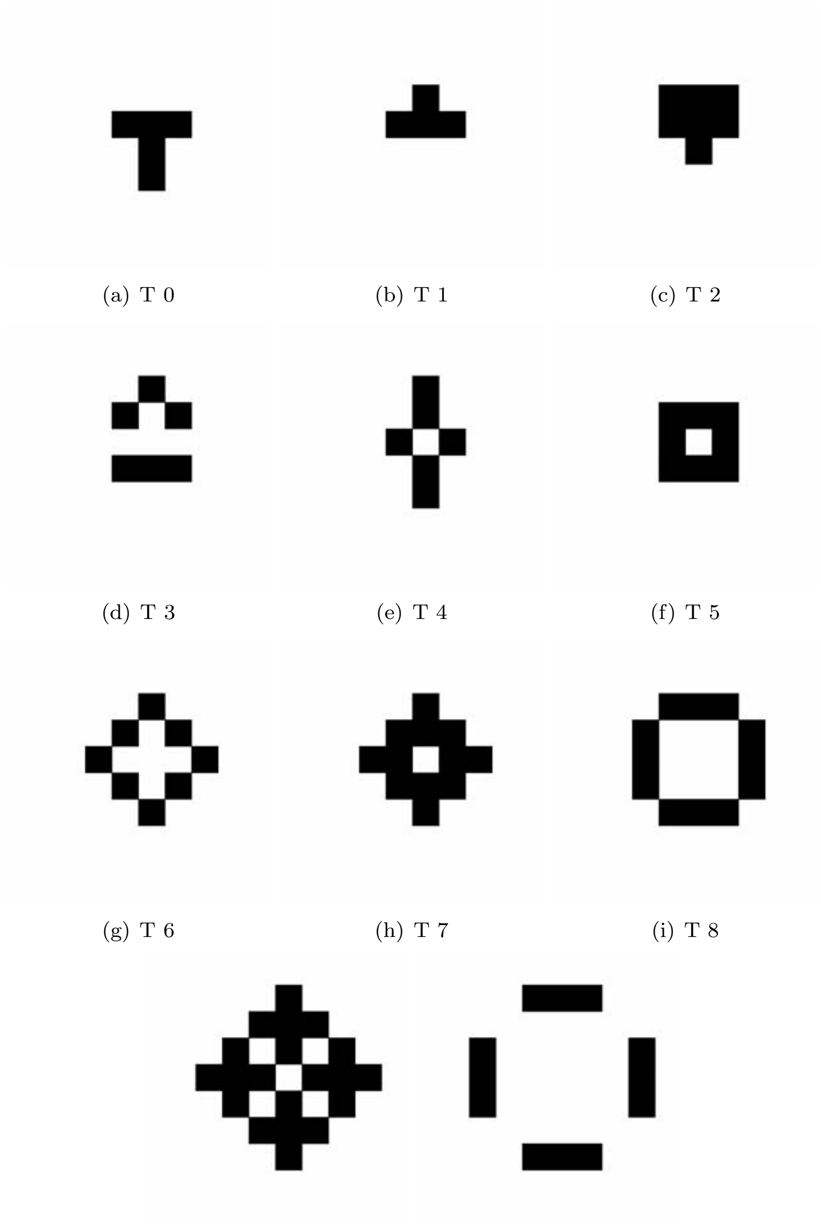

(j) Q 9

Figure 6.6: Q-Pentomino evolution



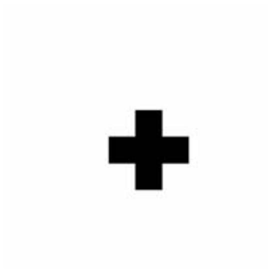
(a) T 0 (b) T 1 (c) T 2

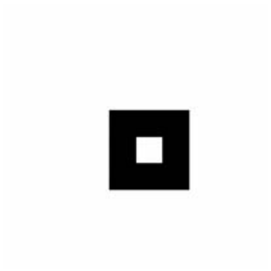
(d) T 3 (e) T 4 (f) T 5

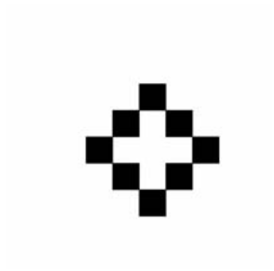
(g) T 6 (h) T 7 (i) T 8

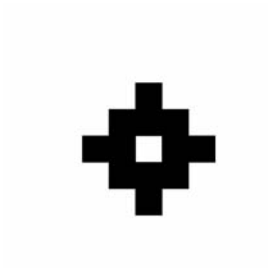
(j) T 9 (k) T 10

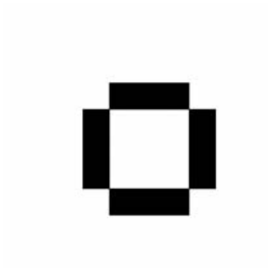
Figure 6.7: T-Pentomino evolution

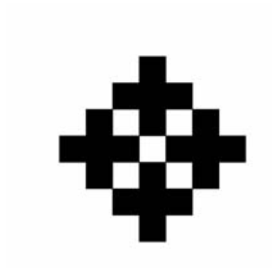

(a) X 0


(b) X 1


(c) X 2


(d) X 3


(e) X 4


(f) X 5

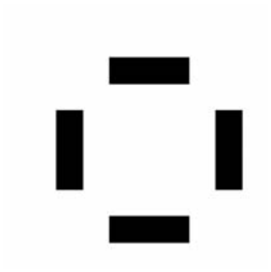
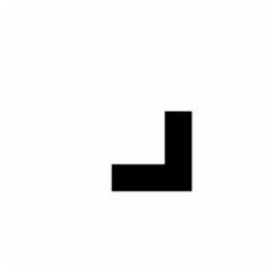
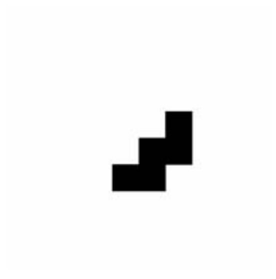
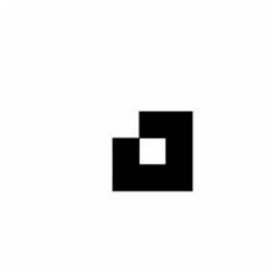

(g) X 6

Figure 6.8: X-Pentomino evolution


(a) V 0


(b) V 1 (W 0)


(c) V 2 (W 1)

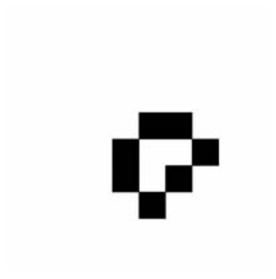
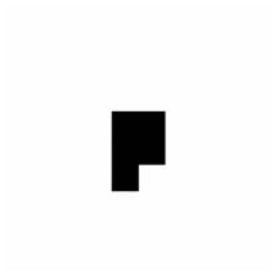
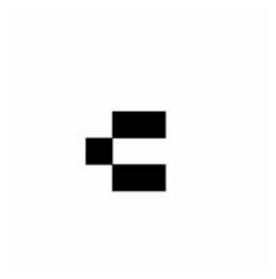
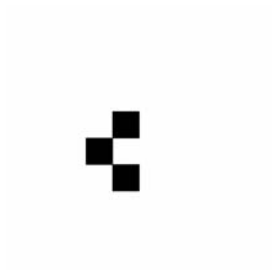
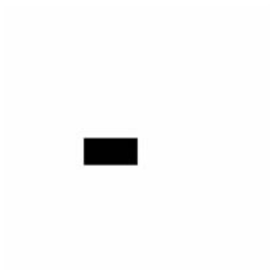

(d) V 3 (W 2)

Figure 6.9: Pentominoes that yield loafs


(a) P 0

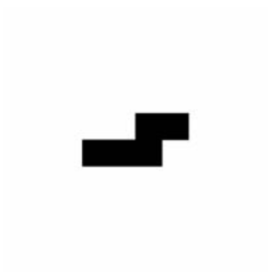

(b) P 1

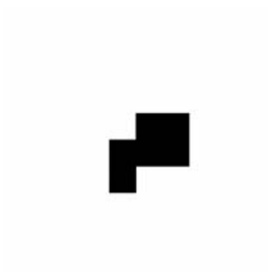

(c) P 2

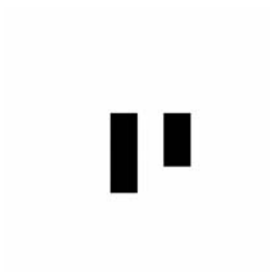

(d) P 3

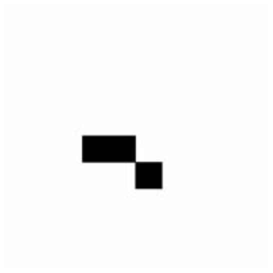
(e) P 4

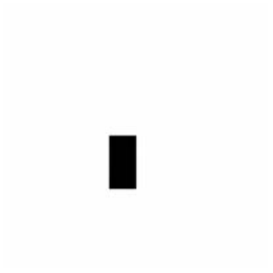
Figure 6.10: P-pentomino


(a) S 0


(b) S 1


(c) S 2


(d) S 3


(e) S 4

(f) S 5

Figure 6.11: S-pentomino

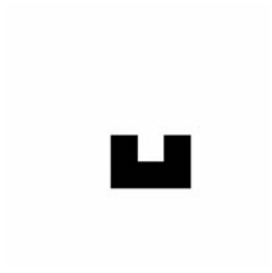
The U-pentomino disappears after 4 generations, shown in the U-pentomino figure. Notice that the U-pentomino evolves exactly as the P-pentomino does, but rotated 90 degrees counterclockwise.

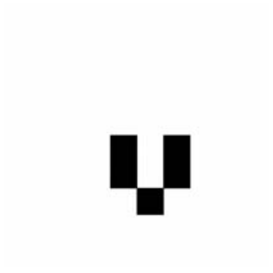
The Y-pentomino also disappears after 3 generations, but reaches its end through a different evolution than P and U, shown in the Y-pentomino figure. Lastly, the Z-pentomino disappears very quickly, within 3 generations, shown in the Z-pentomino figure.

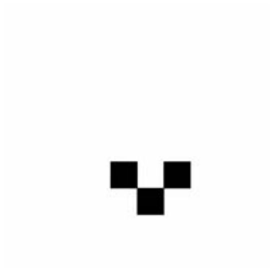
The achievement of these steady states accounts for 11 of the 12 pentominoes. The R-Pentomino turns out to be a methuselah.

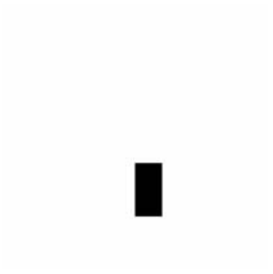
6.4.4 R-Pentomino

By far, Conway's most interesting finding while studying the Game of Life was his investigation of the R-pentomino evolution. This pentomino turned out to be difficult to study by hand. After 480 generations, Conway left its study because it was simply too complicated to do by hand. With the power of computing, we can track the pentomino for a much longer period of time. We started with our 100x100 cell grid with the r-pentomino in the middle, shown in the early-generation R-pentomino figure. Within 100 generations, it becomes quickly obvious that this cellular automata is not about to stabilize anytime soon, as shown in the early-generation R-pentomino figure. At generation 20 we have


(a) U 0

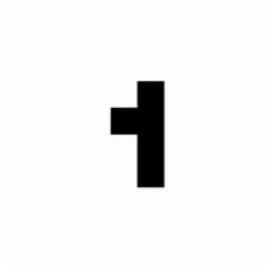

(b) U 1

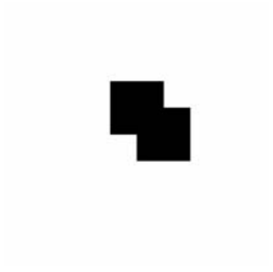

(c) U 2

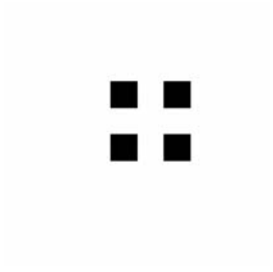

(d) U 3

(e) U 4

Figure 6.12: U-pentomino

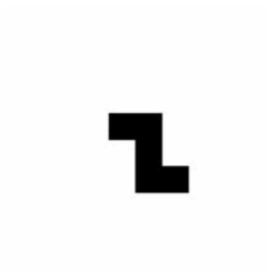

(a) Y 0

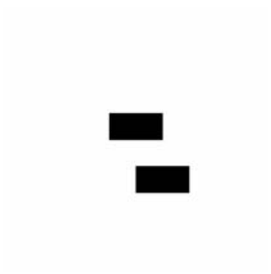

(b) Y 1

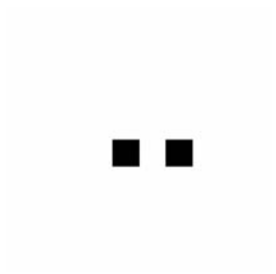

(c) Y 2

(d) Y 3

Figure 6.13: Y-pentomino


(a) S 0


(b) S 1


(c) S 2

(d) S 3

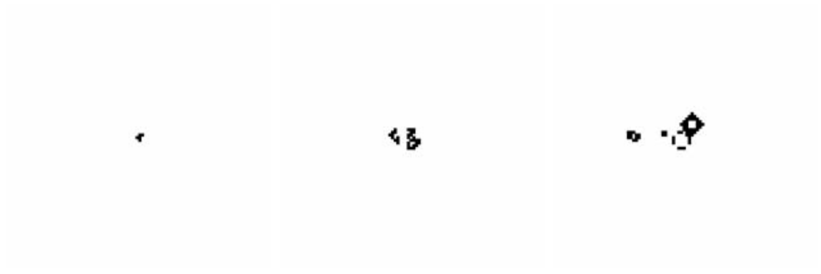
Figure 6.14: S-pentomino

much more complicated behavior than a 5 cell CA. By the 70th generation we have the emergence of a glider (a 5-cell propogating shape in the center-left area of the grid). At generation 100 the glider has shot northeast, and we have 6 4-cell blocks and a beehive, as well as a leftward propogating mass.

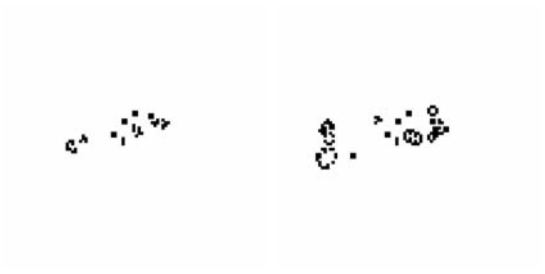
The behavior becomes more complicated as time moves forward. At generation 250 we have 5 gliders, 2 beehives, 2 blocks, 1 boat, 2 blinkers, and a chaotic mass (figure **??**). At generation 500, even the 100x100 cell grid proves to be a bit too small (**??**). The wrapped behavior generates a chaotic mass containing a boat and the propogating mass, that some call ships. In the center we have 3 beehives, 3 blocks, 5 blinkers, and at least one glider.

Eventually, this behavior stabilizes at 1138 steps, shown in the generation 1138 figure. Others have found the steady state at 1103, but due to our wrap-around grid, we get slightly different outcomes. At stable state, the R-pentomino on a 100x100 wrap-around CA yields 4 traffic lights (2 full, 4 halves), 5 blinkers, 7 blocks, 2 beehives, 2 wider beehives, 1 boat, and 1 glider.

The r-pentomino, while complicated, does eventually stabilize. Conway conjectured that there is no initial condition which yields infinite growth. He offered 50 dollars to anyone who proved him wrong. Not too long after this offer was made, Conway found his wallet to be a little lighter.

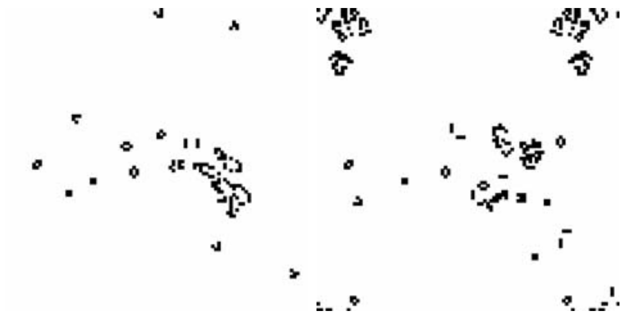


(a) gen 0 (b) gen 20 (c) gen 50



(d) gen 70 (e) gen 100

Figure 6.15: R-pentomino on a wrapped 100x100 cell grid, early generations.



(a) gen 250 (b) gen 500

Figure 6.16: R-pentomino on wrapped 100x100 cell grid, generation 250 and 500

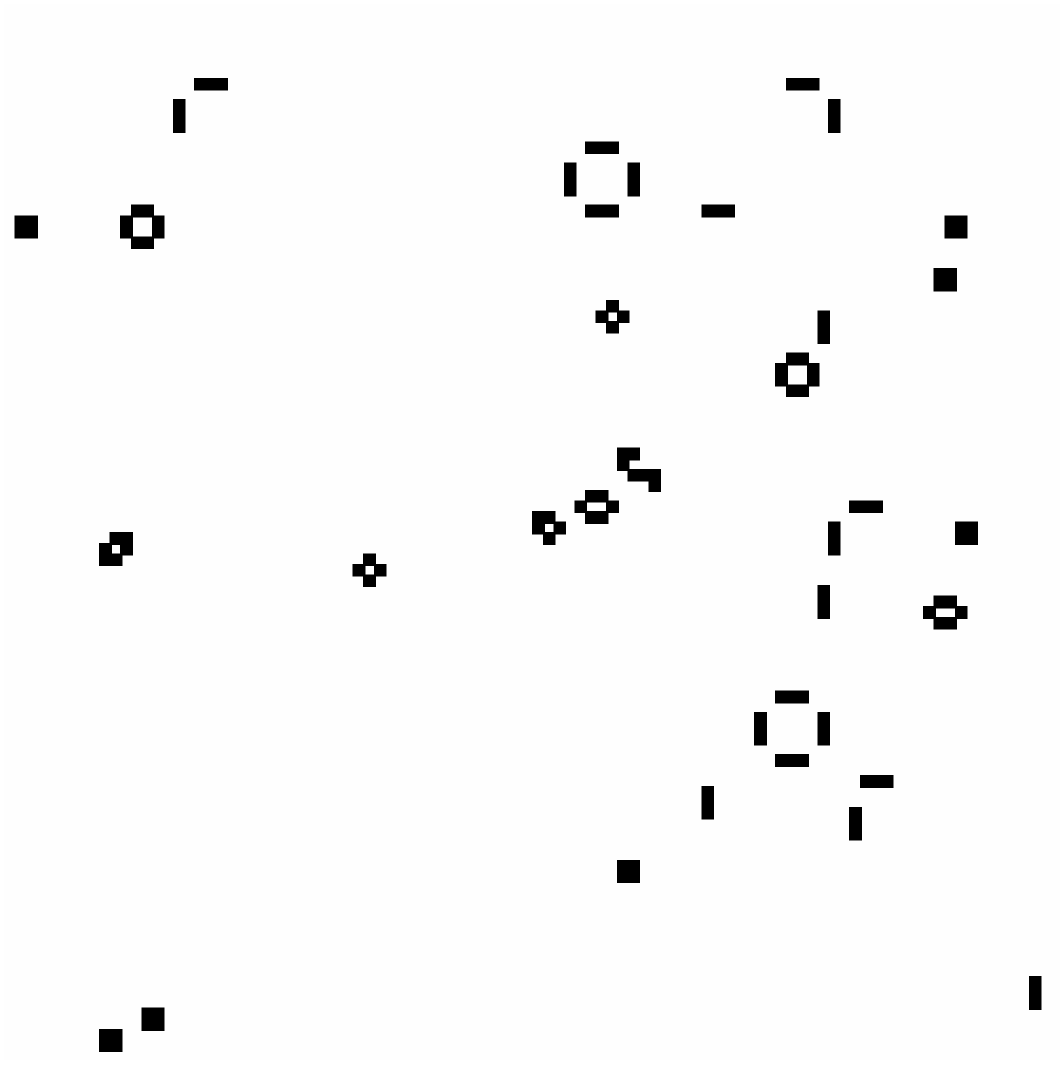


Figure 6.17: r-pentomino generation 1138

6.5 Game of Life Applications and other 2D CA

Bill Gosper developed Gosper's gun in response to Conway's challenge and claimed a 50 dollar prize. This Game of Life pattern grows unbounded, and is also a period 30 oscillator; it shoots a glider every 30 generations (http://www.ericweisstein.com/encyclopedias/life/GosperGun.html). Another pattern that grown unbounded is a puffer train, a body that has a translating pattern that leaves live cells in its wake (Downey's Computational Modeling).

The glider gun can be used to implement computer logic [1]. Gates are methods in which two inputs are compared and then either a true or false value is returned. An AND gate is a digital logic gate that returns 1 when both inputs are high (1), and 0 otherwise. Two guns representing the two inputs can shoot gliders positioned to destroy an object. We run the CA for a period of time and then check whether the object is still present or destroyed by grabbing the values of the cells at the object's location (live=1, dead=0). If an input is 0, the glider gun will be positioned to shoot the object and destroy it, if the input is 1, then the glider gun will miss the object. When both inputs are low, the object is destroyed, and we return false(0). If one input is high, and the other low, the object is still destroyed, and we return a false (0). Only in the situation where both inputs are high, will we return true (1), because the object will remain intact. OR gate logic works similarly. Game of Life was proved to be Turing Complete in 1983.

Another popular 2D CA is Langton's Ant. In 1986, Chris Langton developed a cellular automaton that travels by checking its state, rotating position and changing colors. These turmites appear to behave in complex manners. The ant's trajectory is always unbounded for all initial configurations. It has also been observed that there seems to be some sort of highway interaction, where an ant will travel in a single path for a long period of time, traveling away from the epicenter. It has not been proved whether this is true for all initial conditions. The ant has a period of motion of 104 steps, at which point it translates diagonally leaving these highways behind (`http://en.wikipedia.org/wiki/Langton%27s_ant`)

In addition 2D cellular automata are handy for modeling real-life situations. Scientists have implemented this type of logic in studying animal behavior, computer virues, and human diseases. In addition, it can be useful to study and potentially find the solution to unjamming traffic jams. The Game of Life is filled with emergent complexity that can be utilized to uncover and model the natural world's behaviors.

[1]Sapin, E., Bull, L. The Emergence of Glider Guns in Cellular Automata found by Evolutionary Algorithms. Available at: uncomp.uwe.ac.uk/sapin/cv/ijuc2.pdf

Chapter 7

Self-Organized Criticality

Thus far, we have studied networks and basic cellular automata. We can also use the principles of 2D cellular automata to study the criticality of systems. Systems are described to be critical when they are in transition between two phases. A common example of criticality is water at its freezing/melting point; any change will cause the entire system to tend towards either the solid or liquid state.

7.1 The Sand-Pile Model

In 1987, three physicists, Per Bak, Chao Tang, and Kurt Wisenfeld in Upton, New York became interested in self-organized criticality [1]. They were concerned with studying spatially extended dynamical systems, systems that have freedom in time and space. A self-organized system is one in which the group naturally tends towards a critical state.

Typically, critical systems are unstable. Small deviations will push the system to tend towards a particular phase. In a self-orgaanized critical system, a perturbance will push around the system for some time, but will eventually reach some critical state again. The three physicists proposed a sand pile model that exhibited self-organized criticality, and studied some interesting characteristics regarding these systems.

The sand pile is a two-dimensional cellular automata, in which each cell represents the slope of the sand pile at that location. When a sand pile becomes too steep, some of the sand will slide, decreasing the slope at the origin of the slide, and increasing the slopes of its neighbors. Specifically, the 2D CA is driven by this rule: if the slope, z, of a cell is greater than the critical slope value, K, then

[1] Bak, Tang, Wiesenfeld.1987. Self-organized criticality. *Physical Review A*, 38(364-373)

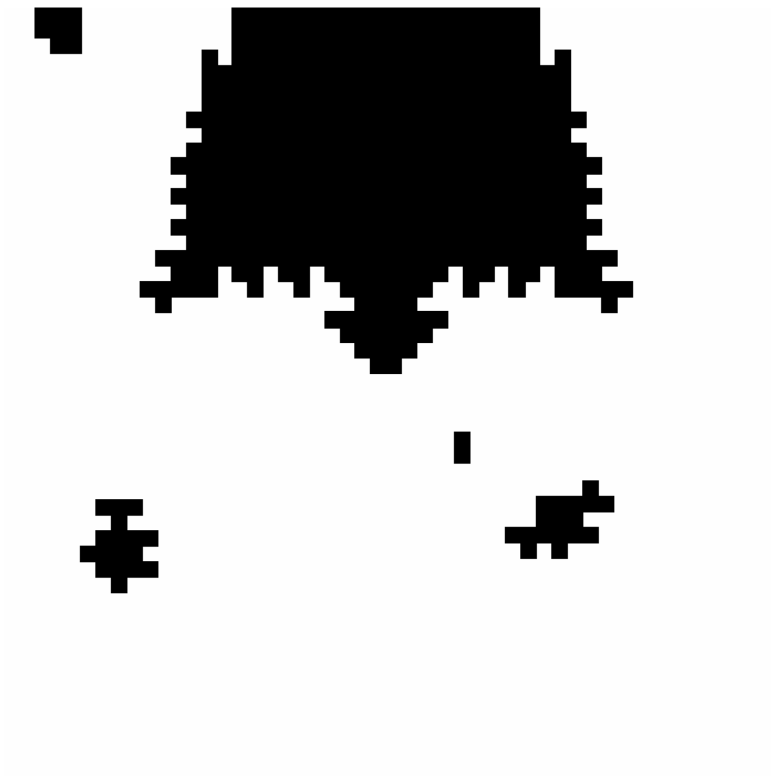
decrease the cell's slope by 4, and increase the four adjacent neighbors by 1. While this is not a very realistic model of a sand pile, it is a popular study of self-organized criticality.

Using our 2DCA created in chapter 6 (named GameofLife previously), we can modify the rules to follow the evolution based on the sand-pile:

```
    def step(self):
        """each cell looks at its neighbors in
        current and steps to next state"""

        t=self.next
        self.next +=1

self.array[t]=zeros((self.m,self.n),dtype=int8)
        self.array[t]=copy(self.array[t-1])

        self.k=5 #critical slope value
        for i in range(1,self.n-1):
            for j in range(1,self.m-1):
                if self.array[t-1][i,j]>self.k:
                    self.array[t][i,j]+= -4
                    self.array[t][i,j-1]+= 1
                    self.array[t][i,j+1]+=1
                    self.array[t][i-1,j]+=1
                    self.array[t][i+1,j]+=1

        #borders should be clear of sand
        self.array[t][:,0]=0
        self.array[t][0,:]=0
        self.array[t][:,-1]=0
        self.array[t][-1,:]=0
```

In the GameofLife, we had started a new dictionary array in discrete time by initializing a zero array. Here, because we are adding and subtracting values, we copied the array from the previous time-step before incrementing it. In addition, the border is set to zero; our sand pile investigation makes use of open boundary conditions. The sand is allowed to spill off the table.

Bak et al. studied the sand pile criticality by studying the size of sand-slides when a single point perturbance is induced. We ran our sand-pile CA with a critical value, K=5, begininning initially with all cells at Z=6. The system was allowed to run to stabilization, and then a critical cell was incremented by 1 to have a slope of K+1. Following a single point perturbance, the system was run again until stabilization, while tracking the span of the distribution. The selection of the critical cell was not random; like Bak et al., every critical cell was tested to find the span of the slide. Results of varying slide structures for a 50x50 array are given in the following figure:



Some perturbances tend to have very small landslides, while a few have large slides, sometimes spanning the entire grid. Bak et al. found this distribution to be long-tailed, a characteristic of critical systems.

In addition, they found that the sand pile model exhibits pink, or $1/f$, noise. In this characteristic, low-frequency components have more power than high-frequency parts. This flicker noise has been observed in a variety of extended dynamical systems including the flow of rivers, stock price indices, the current flowing through a resistor, and the intensity of sun-spots [2]. The flow of sand falling off the pile in the Bak et al. model exhibits this $1/f$ behavior. In order to determine the frequencies of the system, we must first discuss spectral density and calculating frequencies for a given set of data.

7.2 Spectral Density and the Fast Fourier Transform

Any signal varying in time can be described by its power spectral density. The power spectral density, $P(f)$, maps a frequency, f to the power of the signal

[2]ibid

at that frequency. Sounds for example, contain many components at varying frequencies. A piano, for example, gives off a complex tone. Striking a middle A will yield a signal that contains both the dominant frequency for "A" (440 hz), but also contain other sounds with higher frequencies, giving us the harmonic sound we here when the piano is played. A pure sinusoidal wave will only have a single frequency. A signal's spectral density refers to the different frequencies that make up the signal.

In order to study the frequencies of a signal,$h(t)$, we must first transform the signal from the time domain to the frequency domain, using a Fourier transform:

$$H(\omega) = \int_{-\infty}^{\infty} h(t)e^{i\omega t}dt$$

where $\omega = 2\pi f$ is the angular frequency in radians per second. The transformation from the time domain to the frequency domain yields a complex number. The power spectral density, $P(f)$, is simply the magnitude of $H(2\pi f)$ squared:

$$P(f) = |H(2\pi f)|^2$$

If we do not care about whether we have f or $-f$, we can calculate the one-sided power spectral density by summing the magnitude of $H(2\pi f)$ squared and $H(-2\pi f)$ squared.

However, the above equations assume our signal is continuous. When studying the sand pile model, we are doing everything in discrete time, so we need to modify these continuous functions:

$$H_n = \sum_{k=0}^{N-1} h_k e^{2\pi ikn/N} \tag{7.1}$$

where N is the number of values we have. This is known as the discrete Fourier transform (DFT). The frequency, f_n that corresponds to H_n is given by:

$$f_n = \frac{n}{Nd}$$

The one-sided power-spectral density can be calculated from this by simply summing the magnitude of H_n squared and H_-n squared. Since sinusoidal waves are periodic, we can avoid negative indices by computing H_n from 0 to N-1, and use the relationship that $H_{-n} = H_{N-n}$ to convert.

We can code something in Python that calculates the DFT for a given signal:

```
def dft(h,d):
    '''takes h, a sequence of N values, and the time step
    and calculates the discrete fourier transform with n in the range of 0
    to N-1, and the frequency indices for the signal'''

    N= len(h)
    f=[] #frequency index
    H=[] #DFT

    for n in range(0, N): # range(N) returns list to N-1 values
        fn=float(n)/(N*d)
        f.append(fn)

        Hn=[]#dft for given frequency index
        for k in range (0, N):
            hk=h[k]
            p=cmath.exp(2*pi*i*k*n/N)
            Hn.append(hk*p)

        H.append(sum(Hn))

    return f,H
```

Using the DFT, we can then calculate the power spectral density of a given signal using the PSD equation:

```
def psd(h):
    '''calculates the one-sided power spectral density P
    for a given sequence h'''
    N=len(h)
    H=my_fft(h) #H is periodic (i.e. H(n)=H(n+N))

    P=[]
    for n in range(0,N):#range gives indices from 0 to N-1
        print n,N
        Hn1=H[n]
        Hn2=H[(N-1)-n]

        mag1=cmath.sqrt(Hn1.real**2+Hn1.imag**2)
        mag2=cmath.sqrt(Hn2.real**2+Hn2.imag**2)
        Pn=mag1**2+mag2**2
        P.append(Pn)

    return P
```

While we have successfully calculated the PSD, as good programmers, we are concerned about the algorithm speed. The order of growth for the DFT is $O(N^2)$, as we have to move through this double for loop of length N, making this implementation very slow for large data sets.

A much more efficient way to calculate the DFT is to use the fast Fourier transform (the FFT). The FFT makes this distinct substitution, shortening the calculation time: $W = e^{2\pi i/N}$. The new Fourier calculation is written as:

$$H_n = \sum_{k=0}^{N-1} h_k W^{nk} \tag{7.2}$$

Using the Danielson-Lanczos Lemma, we can calculate H_n as the sum of the DFT of even-indexed elements and the DFT of odd-indexed elements:

$$H_k = H_k^e + W^k H_k^o$$

Implementing this algorithm, allows us to develop a much more efficient program:

```
def my_fft(h):
    '''calculates the dft using fast fourier transform for a list, h'''
    N=len(h)
    if N==1:
        return h
    if N%2==1:
        raise ValueError, 'h is not divisible 2'

    he=h[0::2]
    ho=h[1::2]

    He=my_fft(he) #calculate even DFT
    Ho=my_fft(ho) #calculate odd DFT

    He=He+He #bring He up to N elements
    Ho=Ho+Ho #bring Ho up to N elements

    H=[]
    W=cmath.exp(2*cmath.pi*i/N)
    for k in range(N):
        Hk=He[k]+(W**k)*Ho[k]
        H.append(Hk)
    return H
```

This implementation calculates the DFT by bisecting the list, computing the even and odd DFT through recursion, and then combining the DFTs by the Danielson-Lanczos lemma. Because the FFT relies on bisection, it only accepts data sets that are powers of 2. The order of growth for this algorithm is $O(NlogN)$ which is much improved from the original DFT algorithm.

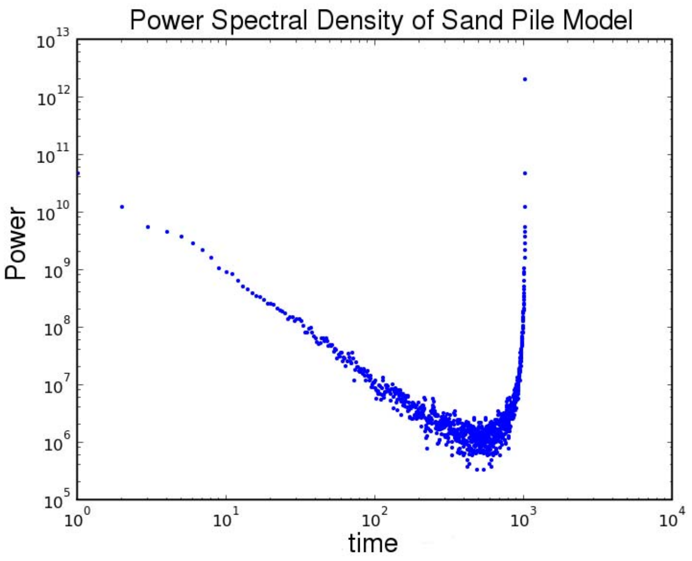
7.3 Sand Piles and Pink Noise

In their work, Bak et al. found that their sand pile model exhibited pink noise. Using our PSD on our sand pile implementation, we can determine whether we have pink noise. Pink noise is called $1/f$ noise because that is the power spectral density of the system. The frequency index, f_n, was defined earlier as $\frac{Nd}{n}$, so that the PSD of a system exhibiting pink noise is simply:

$P_n = \frac{Nd}{n}$

If we plot the PSD on a log-log scale, a $1/f$ system will yield a line with slope $-d$. Since our units of time for this model are arbitrary, we can set d=1.

A new experiment to determine pink-noisiness was run for our open boundary sand pile model. At any given time, we collected the number of cells that were above the critical slope value. We seeded avalanches by perturbing random cells at random times. A cell was pulled into an avalanche with probability .01 (i.e. as we iterate across all the cells in each time step, there was 1.0 percent chance that the cell would be incremented).



The log-log plot of the PSD is approximately -1 from the beginning to 10^3(though closer to -1.5, which we note is what Bak et al. found for their 50x50 array). Our end behavior is a bit noisy, but this is likely a result of running the program longer than needed (Bak et al. ran their program only for 100 discrete time steps). The sand pile model of self-organized criticality exhibits pink noise behavior. This model demonstrates typical extended dynamics behavior, and could be used to characterize other extended dynamics models including turbulence, current through a resistor, glass quenching, and much more.

7.4 SOC, causation and prediction

The investigation of the sand pile system, which exhibits $1/f$ behavior and has a long-tailed distributions, suggests one thing: it is difficult to predict the causes that push the system from its equilibrium. We can determine the proximate cause of large avalanches (i.e. know which cell was the source of the sand slide), but it is difficult to know what the ultimate cause is. Perturbing some cells had nearly no effect, while others cause the entire system to be effected, and there's no clear reason to indicate why this discrepancy exists.

Many real-life situations are characterized by long-tail distributions. There are many small wars, but a few global ones. There are many inventions, but only a few that stick and are used universally/timelessly. If these systems behave like the Bak et al.'s model, then these tipping points are unpredictable. The idea that history's most pivotal moments are unpredictable and inexplicable does not sit well with those who believe that "we can learn from those before us, and prevent their mistakes, and imitate their successes". For example, Great Man theorists would likely be irritated with the ideas presented by SOC. The Great Man theory, attributed to Thomas Carlyle and Abdul Ali, is the belief that "The history of the world is but the biography of great men" [3]. These followers believe that great men, with their charisma and power, have had significant impact on the world. They also believe that by studying the lives of great heroes they can uncover truths about themselves. Criticizers of the Great Man theory claim that there the social environment of these heroes has a great effect on the magnitude of their impacts on the global community. Critics are backed by the SOC studies, which indicate that widespread impacts are dependent on both the individual and the state of the entire system.

The Great Man theory is fairly outdated, as contemporary historians attribute events to factors involving not only the individual, but social, economic, and technological influences as well.

[3] `http://en.wikipedia.org/wiki/Great_man_theory`

Chapter 8

Agent-Based models

The two-dimensional cellular automata we have modeled thus far have been generated under simple rules. The Game of Life was a way to model the change in behavior of a grid of cellular automata. Each generation was created based on the neighborhood rules of the previous generation. However, these individual cells were not changing with any specific purpose. In agent-based modeling, we study intelligent behavior. Each individual cell, governed by a simple set of rules, will move and change based on a simple set of rules. Using agent-based modeling we can study and begin to predict the behavior of dynamic systems using a set of governing local rules. In this chapter, we study a few agent-based models.

8.1 Schelling's Segregation Model

In the 1960s, Thomas Schelling developed a simple model of racial segregation. He proposed that people are interested in living with others who are similar to them, but that they do not require everyone to be the same. That is, most individuals don't mind living in a diverse community, as long as a few individuals are like them. These individuals are not considered to be "racist".

To model his town, Schelling created a grid model, where each cell represents a house. Each house is occupied with either red or blue agents. About 10 percent of the cells are empty. In this model, an agent is content with where it lives if at least 2 individuals in its neighborhood are the same; the agent is unhappy if only 0 or 1 individuals are the same. A neighborhood is defined in the same manner as the Game of Life; the neighborhood is the 8 surrounding cells of the automata. Modeled in discrete time, the simulator randomly chooses individuals to check their happiness, and if they are unhappy they move to an empty cell.

Using the two-dimensional CA capabilities from the previous chapter, implementing this model is simple. We implement the rules of the Schelling model for a CA to check its happiness:

```
def racist_rules(neighborhood,happiness=2):
    '''take's an individual's neighborhood and calculates the percentage of
    individuals that are the same. if it meets the happiness quota return happy
    else return unhappy'''

    same=0
    n=0
    for x in range(3):
        for y in range(3):
            if (x,y)==(1,1):
                continue
            if neighborhood[x,y]==neighborhood[1,1]:
                same+=1

    if same >= happiness:
        return 'happy'
    else:
        return 'unhappy'
```

A grid of 9 cells (the neighborhood +the individual) is inputted, and the number of surrounding cells that are the same is checked. If there are at least enough same cells for the CA to be happy, we return that the CA is happy. Otherwise, they are unhappy. This implementation does not check whether the cell is occupied or not. That check must be done elsewhere.

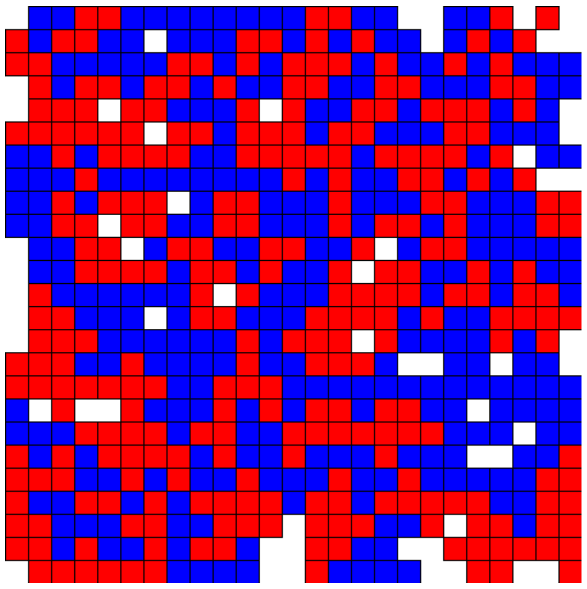
We modified the 2D CA `step` method to accomodate the new algorithm. For each cell, we check whether it is empty, and then with some random probability we check its happiness and move it, if it is unhappy.

```
    def step(self):
        t=self.next
        self.next +=1
        self.array[t]=zeros((self.m,self.n),dtype=int8)

        for i in range(1,self.n-1):
            for j in range(1,self.m-1):
                if self.array[t-1][i,j]!=0: #if occupied
                    if random.random()<.2: #check happiness randomly
                        neighborhood=self.array[t-1][i-1:i+2,j-1:j+2]
                        if racist_rules(neighborhood)=='happy':
                            self.array[t][i,j]=self.array[t-1][i,j]
                        else:
                            self.move(t,(i,j))
                    else:
                        self.array[t][i,j]=self.array[t-1][i,j]
```

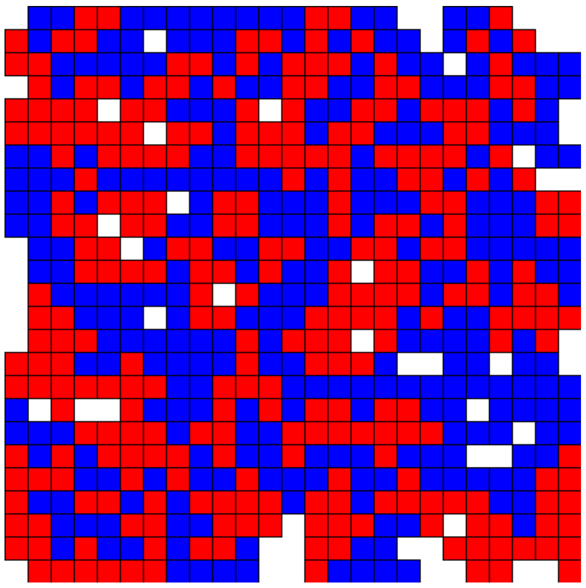
To move cells when they are unhappy, we developed a method `move`. The grid object stores a list of all the cells that are empty at any given time. If a cell is unhappy, and wishes to move, we randomly choose a new cell for it to occupy, and add its old address to the empty cells list.

```
    def move(self,t,(i,j)):
        '''move an individual at a discrete time to an empty cell'''
        self.restless +=1
        l=len(self.empty)
        if l==0:#if there's no where to move, don't
            self.array[t][i,j]=self.array[t-1][i,j]
            return
        n=random.choice(range(0,l))
        (x,y)=self.empty.pop(n)
        self.array[t][x,y]=self.array[t-1][i,j]
        self.empty.append((i,j))
```

Using Schelling's initial conditions, we modeled the town.



Fairly quickly, the town begins to segregate. Over time, we see large clusters of segregation, where very few individuals live in mixed neighborhoods. This is the city, stable, after 100 generations:

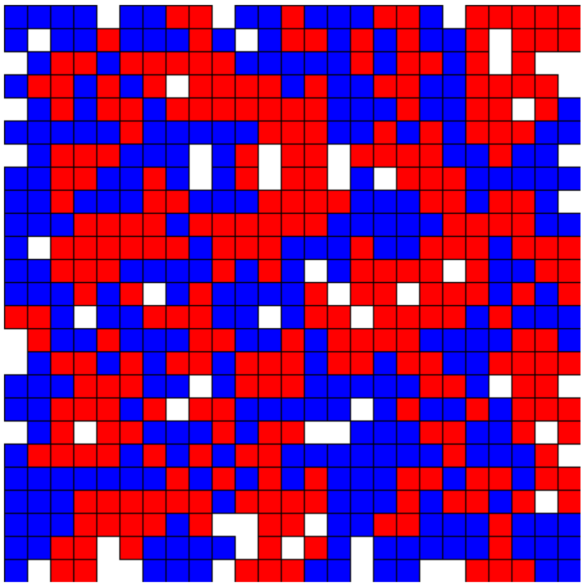


While this model may be too simple to assert that cities with segregated neighborhoods are not necessarily racist, it does help show that we can not automatically assume that cities with segregated neighborhoods are racist.

A way to make this model a little more intelligent is to have individuals move to neighborhoods more like them. Bill Bishop, in a recently published book, ıthe Big Sort, cliamis that American society is increasingly segregated by political opinion. People choose to live with those who are like-minded. We modified our `move` method to be a little more intelligent; if a CA is unhappy, it will move to a neighborhood with people like it, and not completely randomly:

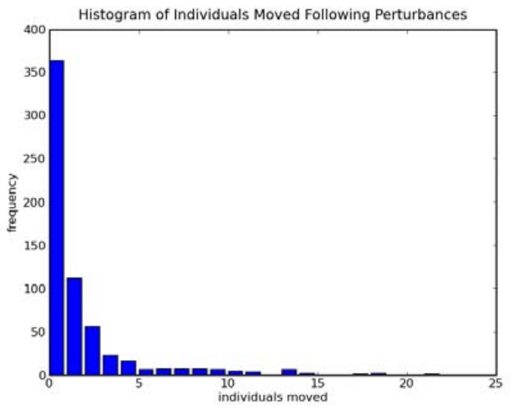
```
    def move(self,t,(i,j)):
        '''move an individual at a discrete time to an empty cell'''
        self.restless +=1
        l=len(self.empty)
        if l==0:#if there's no where to move, don't
            self.array[t][i,j]=self.array[t-1][i,j]
            return

ca='unhappy'
        if ca=='unhappy':#preferential moving
            n=random.choice(range(0,l))
            (x,y)=self.empty.pop(n)
            neighborhood=self.array[t-1][i-1:i+2,j-1:j+2]
            neighborhood[1,1]=self.array[t-1][i,j]
            ca=racist_rules(neighborhood,5)
        self.array[t][x,y]=self.array[t-1][i,j]
        self.empty.append((i,j))
```

Again, we find segregation, but a bit more of it. At steady-state the model is more segregated than the previous:

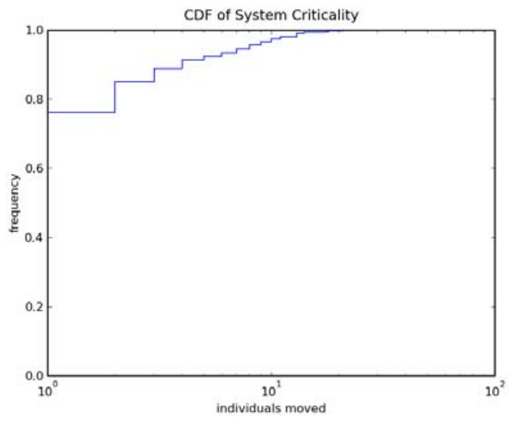


In the other model, there were still some individuals that lived in mixed neighborhoods, but here, there is much larger clustering. We decided to examine the criticality of this system. That is, if a single cell were flipped, would many individuals have to move to find happiness again?

For a 50x50 grid at steady state, we went through and flipped each cell, and counted how many moves would occur over the next century (100 generations) due to the perturbance. We found that most perturbances did not cause any moves, and the largest movement stirred was 21 moves, which is not a lot when we consider the grid to be made up 2500 cells.



We plotted the CDF of this set, and found that over 60 percent of perturbances don't induce any motion. This distribution is not long-tailed; 21 moves is not an incredibly large number.This system is not critical.



This lack of criticality, indicates that most individuals are not irritated by the presence of someone different in their community. That being said, this model is not incredibly realistic if it turns out individuals are racist. An interesting study would be to modify this model and create mixed cities where some individuals are virulently racist and others are just like these individuals.

8.2 Traffic Jams

We can also use agent-based modeling to study the dynamics of traffic jams. Studying traffic jam models yields a couple interesting results. The wave of a traffic jam propogates backwards: as cars pull away from the jam at the front, and others join the jam from the back, we find that the location of the traffic jam moves back. In addition,contrary to the belief of most drivers, traffic jams don't have to be the result of a collision or some external cause which forces local traffic to almost completely stop. The modeling of traffic jams has shown that traffic jams can occur spontaneously, without external cause. So what types of parameters create traffic jams? We used Downey's Highway class in order to model traffic jams. The cars (shaped as turtles) in Downey's program exist on a one-lane infinite highway, but are mapped onto a 400x400 pixel canvas to study.

Drivers naturally speed up when there are large open spaces in front of them, and slow down when they come upon a car that is going slower, if they cannot overtake them. It turns out that this type of driving is enough to create traffic jams. The number of drivers on the road, the rate at which a driver speeds up and slows down, and the following distance that cars keep will effect whether driver behavior will yield a jam. We developed a `BadDriver`, which is a driver that slows down when cars are in front of it, and speeds up when there is a large open space in front of it:

```
class Bad_Driver(Driver):
    def choose_speed(self, dist):
        if dist<self.d:
            self.speed=self.speed-self.rate #slow down

        if dist>self.d:
            self.speed=self.speed+self.rate

        if self.speed<0:
            self.speed=0

        if self.speed>10:
            self.speed=10
```

The rate at which the driver speeds up and slows down, `self.rate`, is chosen when the drivers are made, as is the following distance `self.d`.

Commuters can contest that when there are more cars on the road, there is more traffic, and more jamming. We studied how many jams appeared for a variety of cars on the road. We put a varying number of `BadDriver`s on the road with a following distance of 50 pixels and a slowing down/speeding up rate of +/- 1. We found that until about 30 cars, there were no jams. At 30 cars on the highway, we found one jam that only included a couple cars and dissipated quickly. However at 50 cars we found 3 jams. By 100 cars we found 10 jams on the screen that were rather large.

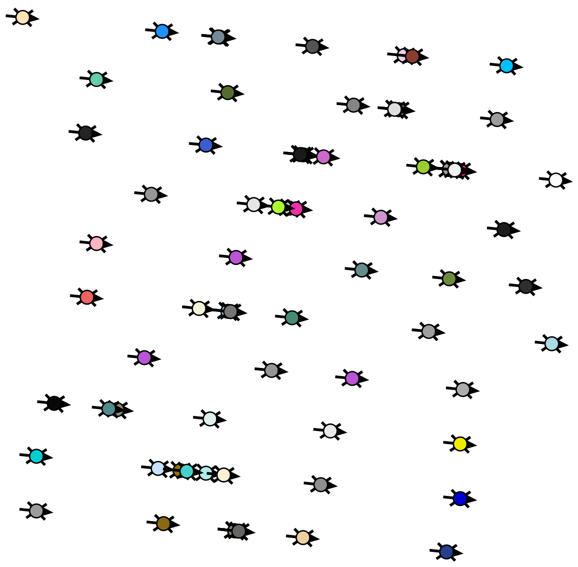


Figure. 50 cars

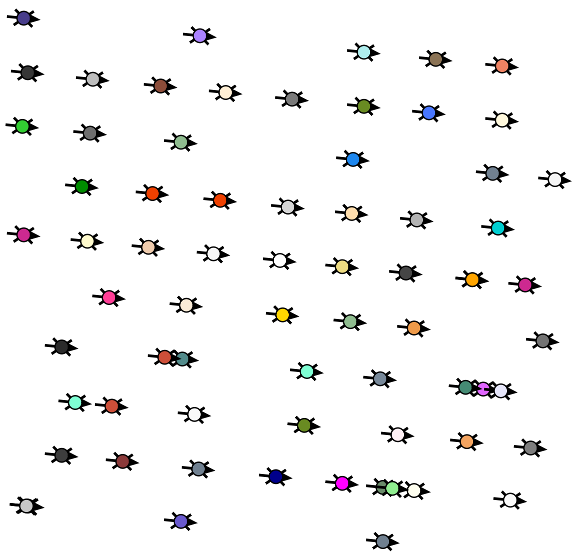


Figure. 100 cars

We decided to see if varying the following distance would change the traffic conditions. Varying the following distance for a Highway containing 70 cars, and a varying speed rate of +/- 1 depending on how far away the car ahead is, we studied what would yield a traffic jam. For a following distance smaller than 20 pixels, we have no traffic jams. When the following distance increases to 30 pixels we have a few small jams. At 40 cars we have a few larger jams. By 100 pixels, traffic is almost at a stand-still because the following distance is so large that jams become quite large.

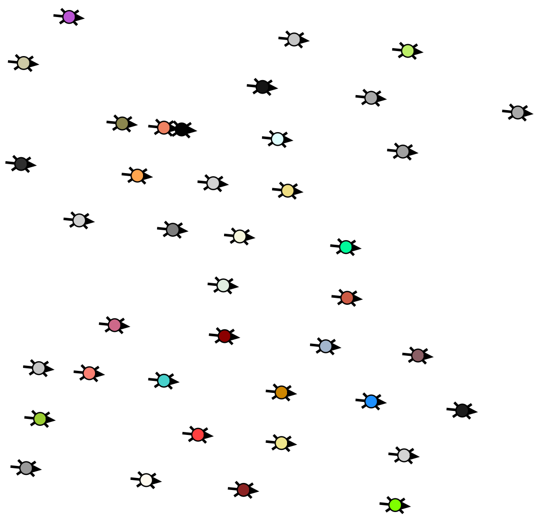


Figure. Following Distance of 40 pixels

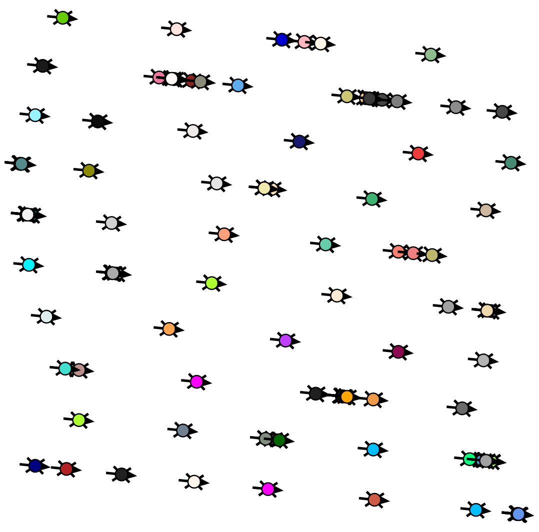


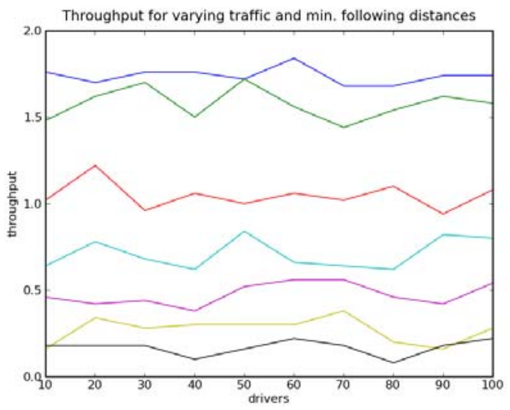
Figure. Following Distance of 100 pixels

We can optomize the highway, by determing what type of driver conditions would yield the highest throughput. A throughput is the rate at which cars pass through a particular point. We studied the throughput for this highway by counting how many cars at any given step passed through a point on the canvas. Since the cars (`Animals` in `World`) only store a single position value that gets mapped onto the canvas we say a car has passed through the through point if the mod of its position is greater than the through point and was not previously.

```
def throughput(self):
    n=math.sqrt(self.ca_width**2+self.ca_height**2)
    counter=0
    c=300
    for animal in self.animals:
        if animal.position_tracker[-2]%n <c:
            if animal.position_tracker[-1]%n >=c:
                counter+=1
    try:
        self.threshold.append(counter)
    except: #initialize throughput counters
        self.threshold=[counter]
```

The `Highway step` method was modified such that animals are incremented and then the `throughput` method is called. The `Highway` object keeps track of how many cars pass through the point at every time step, and the `Animals` keep track of their positions over time.

Using this method, we studied the optimal conditions for the throughput.



What we see here is that the throughput stays fairly constant as cars increase (likely because while the number of cars slow down, more cars means more to pass through a point). In addition, an increase in minimum following distances yields a higher throughput, which suggests the flow of traffic moves smoother when there is some space, though across cars. A better investigation would be to study the average speed of cars on the road as a measure of traffic jamminess, as throughput varies for both speed of cars traveling and quantity.

Chapter 9

Stochastic Modeling

Stochastic quite literally means random. Thus far, we have studied models dictated by rules and preferences. In a slightly different direction, here we study the modeling of random processes, to understand the probable outcomes of a given situation, given a certain set of circumstances or information. Stochastic modeling is used on non-deterministic models, where the state of a system is dependent on the probable outcomes that an event will occur. This type of computer simulation, can be useful to quickly understand the likelihood of an event occurring, without doing very much hand-analysis, which can sometimes be tricky or overwhelming. In this section, we will explore a variety of stochastic models.

9.1 A Girl Named Florida

9.1.1 the problem

Leonard Mlodinow published a number of brainteasers in ıthe Drunkard's Walk,in which he poses the "A Girl Named Florida" problem. We study a family with two children and pose the question: with is the probability that the family has two girls? If the probability of having a child that's female is 50%, we would guess that the chance of the two kids being female is: $\frac{1}{2} * \frac{1}{2}$ or 25%. However, as Mlodinow provides more information about the children, there exists a higher probability that the two children are female. What are the probabilities that the two children are girls if:

1. one of them is a girl?
2. the older one is a girl?
3. we know that at least one of them is a girl named Florida?

At first glance, we might be confused why this information would change the probabilities. After all, it seems as if the children combinations possible are still: boy-boy, boy-girl, girl-boy, and girl-girl. More importantly, why would the probability between numbers 1 and 3 be any different? Here is a case where our intuition may inhibit us from determining the answers to these seemingly simple questions. Simulations of stochastic events are frequently used and taken as proofs to the situation. Let's use simulation to answer these questions!

9.1.2 the model

In order to randomly model this situation, we generate a lengthy list of families where the kid born can either be a boy or a girl. Then based on the knowledge we have about a family, we remove any children combinations that are impossible. This creates a randomly generated list of all the possible outcomes given the information we have. Let us first simulate the situation where we don't have any information, except that the family has two children.

We generate a number of families to see how likely it is we will have a family with two girls. Note the formatting of the function `family` was created so that more information could be added:

```
def make_children(number=1):
    '''returns a two-element list of the two children'''
    type=[0,10] #genders given by number, boy=0, girl=10
    children=[random.choice(type),random.choice(type)]
    return children


def family(number=1):
    '''creates a set of families, and then invokes our knowledge
    which corresponds to the number problem given in exercise 9.2'''

    families=[make_children(number) for i in range (10000)] #family has 2 kids
    f=[]

    if number==1:  #no other info
        f=families
    two_girls(f)

def two_girls(families):
    '''takes the list of all family possibilities and returns the percentage
    that have 2 girls'''
    c=0
    for children in families:
        if sum(children)==20:
            c+=1
    t=float(c)/len(families)
    p=t*100
    print '%d percent chance' %p
```

The method `'make_children'`generates a child pair of boy and girl with a 50 % probability any given child will be female. A number of families are created and then we check what percentage of those have two girls, which we confirm is 25%.

Now, we can add more information to the problem. If we know one is a girl, what is the chance they have two? In the `family` function, we add an if statement to remove outcomes where there are no boys:

```
    if number==2: #we know one is a girl
         for children in families:
            if sum(children)!=0:
                f.append(children)
```

This is why we chose a number assignment for the kids. Its much easier to check the sum of lists then iterate through items. It turns out that this outcome yields a 33% probability of a family having two girls! This might seem counterintuitive, but essentially, we have removed the boy-boy possibilities, so that the girl-girl possibility instead of having a $\frac{1}{4}$ chance of occuring we have $\frac{1}{3}$ chance of occurrence.

What if we knew that the older child was a girl? We remove outcomes where the family doesn't have an older girl:

```
#within family
  if number==3: #we know there's the older sibling is a girl
        for i in range(len(families)):
            if families[i][0] == 10:
                f.append(families[i])
```

It turns out that then the probability of a family having two girls is 50%! A little analysis shows that the only two possible families would be: girl-girl, and girl-boy, thus yielding the outcome our simulation provides.

This leads us to the last problem: what if we knew that one of them was a girl named Florida. In our simulation, we assume a family only has one child named Florida. In this study, now instead of just having a girl or boy, we also have a girl-named-florida. We must modify our `'make_children'`function so that we can either have a male, female, or florida-female:

```
#within make_children
    if number==4: #girl named florida problem
        type.append(21) #girl named florida=21
```

We then modify our `family` function to observe the Florida problem:

```
    if number==4: #make this version work
        for i in range(len(families)):
            if sum(families[i])>=21:
                if sum(families[i])!=42:
                    f.append(families[i])

        for i in range(len(f)):
            if f[i][0]==21: #girl named florida, also a girl
                f[i][0]=10

            if f[i][1]==21:
                f[i][1]=10
```

We kept any family combinations where there was one (but not two) girls named Florida, and then changed the Florida type to a girl so that the `'two_girls'`function can determine how many families left have two girls. We find this number to be roughly 50%! So, how is this problem different than knowing that one of them is a girl?

In the Girl-Named-Florida problem, we have modified how much knowledge we have of the children. Not only can someone have a girl (g) or a boy (b), but they can also have a girl named Florida (gf) giving these possible families given our information: g-gf, gf-g, gf-b, b-gf. Thus the possibility of a family having 2 girls is 50%! If we had assumed that a family could have two girls named florida with equal probability, then we would have a 60% chance.

This problem gives the idea that having more information will increase our probabilities of getting a certain outcome.

9.2 Bayesian Statistics

The Girl Named Florida problem sheds light on an interesting aspect of probability: the more knowledge we have, the more likely a certain outcome is. This is the guiding principle behind Bayesian statistics. Bayes' theorem links the relationship between conditional probabilities of 2 events. It is often interpreted as a statement of how evidence, E affects the probability of a hypothesis, H and is given by:

$$P(H|E) = P(H)\frac{P(E|H)}{P(E)}$$

What this equation is saying is that the probability of the hypothesis occuring based on evidence (the posterior), is equal to the old probability (the prior) multiplied by the probability some event occurs based on the hypothesis, divided by the probability the event occurs ever (the likelihood ratio).

For example, let's say a boy named Fred had two cookie bowls in front of him: bowl 1 contains 30 plain and 10 chocolate chip, the bowl 2 contains 20 of each. Fred picks a bowl at random and then a cookie. The cookie he takes turns out to be plain. What is the likelihood he chose it from bowl 1? The hypothesis here, or the question we're asking is what bowl did he choose? The evidence, or the knowledge we have gained about the situation, is that the cookie is plain.

The prior probability of choosing a bowl is $\frac{1}{2}$. $P(E|H)$ is $\frac{30}{40}$. That is, if Fred chooses bowl one, the likelihood of choosing a plain cookie is 3 out of 4.

Calculating the probability of the event, or $P(E)$ is a little trickier. We must consider all probabilities of the event occuring which is a weighted probability of choosing the bowl and choosing a cookie, given by:

$$P(E) = P(H_1)P(E|H_1) + P(H_2)P(E|H_2)$$

This value comes out to be 5/8. When we plug these numbers into Baye's Theorem, we find that Fred's likelihood of choosing bowl 1 is 60% (from an original 50%). By gaining more evidence, our probabilities of a certain situation happening does increase! We can use this type of understanding to study whether drug tests are positive, and also use it to prove a problem that has baffled people for years: the Monty Hall problem.

9.3 Monty Hall

Another counterintuitve problem is the Monty Hall one, which has been debated for years. Monty Hall was the host of a show ıLet's make a Deal. In the show, Monty had a game where there were three doors, behind one of which was a car. The contestant on the show was asked to guess which door the car was behind. Monty, in order to drive suspense, would open the door that the contestant had not chosen, but also one that did not contain the car. Then he would test the contestant's decision: ”would you like to switch?”, he would ask. And this is where the Monty Hall problem comes in. You might think that since one door has been removed from the list of possibilities, that the probability of the car being behind your door or the remaining is 50/50. But it turns out that the contestant has 2:1 odds of winning if he/she switches doors!

This might not sit well with you. I have spent most of my grown up life struggling with it, but in the last two weeks, I figured it out. Today, I hope to convince you of the truth as well. We will attack this problem from a logic, computational model, and mathematical approach. Hopefully, one of these perspectives will help you see the importance of switching doors so that you too can win a car on ıthe Price is Right or some other show someday!

9.3.1 the logical approach

We will first try to verbally explain the Monty Hall problem. Say you're on the show and there are three doors: A, B, and C. If you choose door A, there are a couple possibilities of what Monty can do:

1. If the car is behind door A, Monty will choose B or C at random, because he can display an empty door.
2. If the car is behind door B, Monty will have to open door C.
3. If the car is behind door C, Monty will have to open door B.

Let's study each of these situation outcomes. For number 1, if you switch doors then you lose. But in number 2 and 3, if you switch to the other door, you win. So switching give you a $\frac{2}{3}$ chance of winning, instead of $\frac{1}{3}$!

But perhaps I have not convinced you yet? It's okay. Even the great mathematician, Paul Erdős, took a long time to be convinced. He was assuaged when a computer simulation proved him wrong. So let's try a computer simulation.

9.3.2 the computer model

We design the Monty Hall simulation exactly as if we were playing the game and making decisions based on the knowledge we are gaining. The advantage to a simulation is that we can run it a number of times, quickly, to determine something about the stochastic situation. Here we wrote the Monty Hall problem as:

```
def monty_hall(u='switch'):
    '''simulate the monty hall problem. makes decisions based on whether
    you would switch, and if monty has knowledge of where the car is'''
    doors=[0,1,2]
    car=random.choice(doors) #put the car somewhere
    you=random.choice(doors)#competitor chooses a door
    monty=choose_other(doors,car,you)#monty shows the other door

    if monty == car:
        return 0
    if u=='switch':
        you=choose_other(doors,you,monty) #you switch

    if you==car: #did you win?
        return 1
    else:
        return 0
```

```
def choose_other(l,a,b):
    '''in a list containing elements a and b choose c'''
    for n in l:
        if n!=a:
            if n!=b:
                return n

def rand_choose(l,a):
    '''from a list randomly choose any value that's not a'''
    n=random.choice(l)
    if n!=a:
        return n
```

The `'choose_other'`function allows us to choose the element in the list that isn't one that is taken already, and `'rand_choose'`is a function will choose anything that's not the taken value. We ran this program for 10,000 iterations and found that switching doors yields a success rate of 66%. We also decided to simulate the importance of Monty having knowledge. Instead of monty choosing the other door with some knowledge of where the car is, we had Monty choose a door that was not the contestants (by modifying the Monty line from `'choose_other'`to `'rand_choose'.`In the case that you switch doors when Monty has no knowledge, the success rate is only 33%, or the original probability of winning given no new knowledge. The simulation shows the influence of Monty's knowledge on the probabilities.

But perhaps, you are someone who does not accept simulation as proof. Perhaps a more pure mathematical approach can convince you.

9.3.3 Bayesian Statisitcs

We can also prove the switching is better case, by using Baye's Theorem. We'll say we have doors A, B, and C, and the probability the prize is behind any of them initially is:$P(A) = P(B) = P(C) = \frac{1}{3}$, which is our prior. Let's say you're playing the game, and you choose door A. Let's say the event/evidence, is that Monty chooses door B. If Monty has no knowledge of where the car is, there's a $\frac{1}{2}$ chance he chooses door B, given that he won't open yours. We'll call this the event, or $P(M) = \frac{1}{2}$. Now we need to fill in the likelihood that Monty would choose door B, given that he knows where the car is:

- If the car is behind door A, then Monty is free to choose B or C. He has a $\frac{1}{2}$ chance of choosing door B.
- If the car is behind door B, then Monty must choose C. He has 0 chance of choosing B.
- If the car is behind door C, then Monty must choose B. There is a 1.0 chance of choosing B.

Now if plug these values into Baye's Theorem, we can determine how likely the car is behind your door, as opposed to the others: $P(A|M) = \frac{1}{3}$, $P(B|M) = 0$, and $P(C|M) = \frac{2}{3}$. You're more likely to win if you switch! But maybe you already knew that.

9.4 Poincaré

Henri Poincaré was a French mathematician in the nineteenth century, immortalized in an apocryphal story on how he used mathematics to go after a baker that was cheating the town.

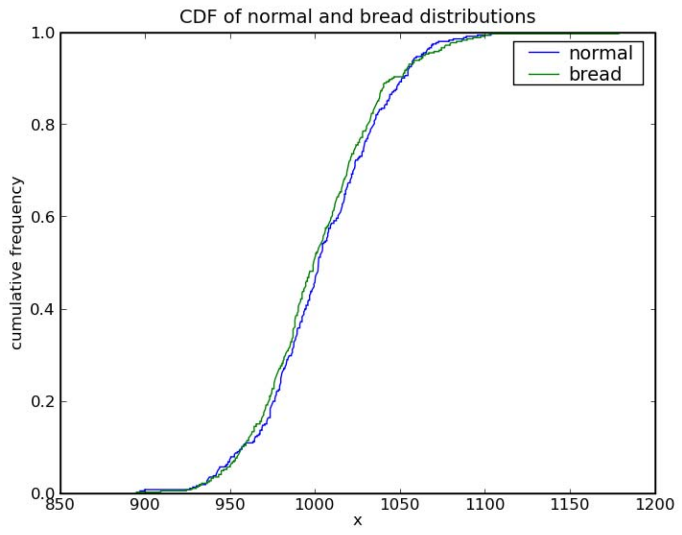
Poincaré, used to buy bread daily from a man who claimed to sell 1kg loaves (with a standard distribution of 50g). Suspicious of being cheated, or perhaps a wary customer, Poincaré used to bring the bread home daily and weigh it. At the end of the year, he plotted the distribution and found that the mean was 950g with a standard deviation of 50g –significantly lower than what the baker had advertised! Poincaré complained to the police, who then issued a warning to the baker. The clever baker, not interested in changing his ways, began to sell Poincaré his heavier loaves. Poincaré continued to monitor the weight of his bread. At the end of the year, the mathematician plotted the distribution and found that while the mean was 1 kg, the distribution was assymetric!The police fined the baker.

How can this be? How did the baker successfully give Poincaré bread that would give a 1kg mean at the end of the year? How did Poincaré catch the trick?

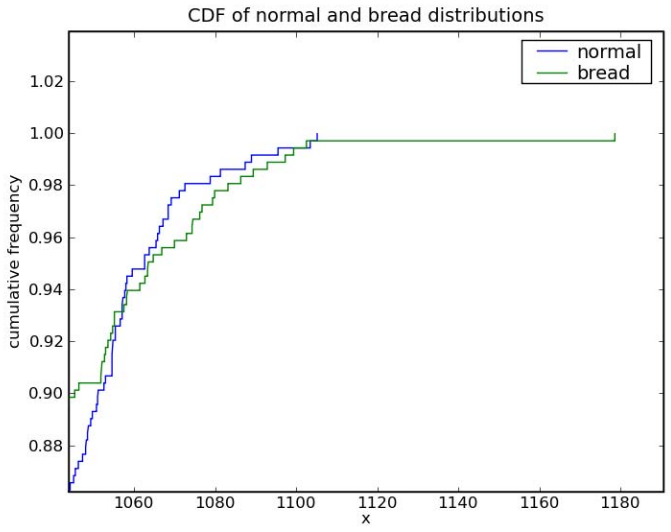
We can simulate the interactions between the baker and Poincaré to determine what decision-making would have made this possible. In order to give Poincaré the loaves he did, the baker must have chosen the heaviest loaf from a sampling of his bread. Depending on how many loaves he used to choose the one for his favorite mathematician, the baker could create a distribution where the mean is what he advertised: 1kg. We simulated it this way:

```
class Poincare:
    def __init__(self,n):
        self.breads=[]
        self.day=1
        self.n=n #number of loaves baker picks up

    def loop(self,n):
        step=[self.visit_baker() for i in range(n)]

    def visit_baker(self):
        '''the interaction with the baker on a single day'''
        bread=[random.normalvariate(mu=950,sigma=50) for i in range(self.n)]
        bread.sort()
        self.breads.append(bread.pop(-1)) #give poincare the heaviest breads
        self.day+=1
```

In our first investigation, we looped through the 365 days in a year, and let the baker give Poincaré the heaviest loaf from n-sample loaves, and determined that in order to achieve a mean of 1000g, the baker would need to choose from 4 loaves. An n-value of 4 loaves yields a distribution with mean 1002g and standard deviation of 34.8 g. Using these values, we compared the bread's distribution to a normal distribution about these values to find:



At first glance, it seems like both of these distributions are the same. But then, if we zoom in at the top we actually find that there is an assymetry to the bread distribution:



Poincaré's bread spans all the way 1180g where the normal distribution ends at 1100! The baker was trying to hide his cheating ways!

9.5 Dances

Another item we can use stochastic modeling for is to hypothesize the possibility of viewing certain events in our population. A professor of mine once wondered out loud, if we were to go to a random dance, how many heterogenous couples would have male partners taller than females. We decided to model the likelihood using Python's random module, and specifically the `normalvariate` method, which takes a mean and standard deviation to generate a random number from the distribution set. Having done some digging aroung, we found the average American male height to be 69" with a standard deviation of 3" [1], and the average American female height to be 65.5", with a standard deviation of 2.5" [2].

We created random pairings using these values, as given in the program below:

```
def dates(n):
    #create n pairs of men and women
    pairs=[]
    for i in range(n):
        man,woman = random.normalvariate(69,3), random.normalvariate(65.5,2.5)
        pairs.append((man,woman))
    return pairs
```

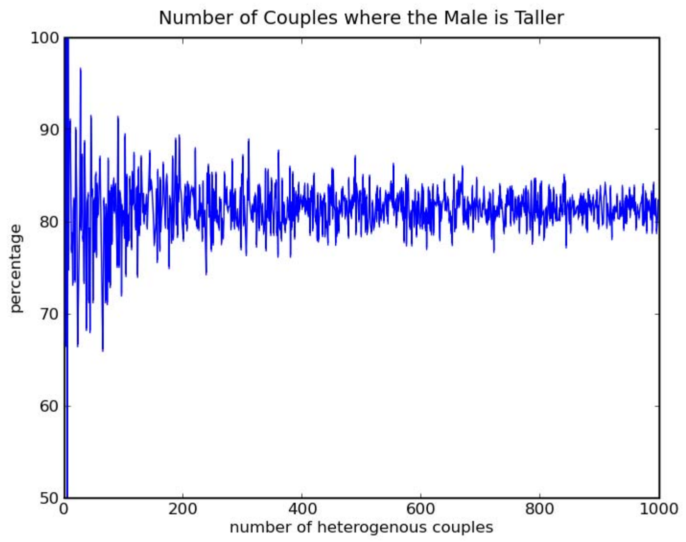
[1] http://investing.calsci.com/statistics.html

[2] http://www-slat.stanford.edu/~naras/jsm/NormalDensity/NormalDensity.html

We ran these pairs through a program that determined who was taller, to reveal what percentage of couples exist where the male is taller:

```
def taller_guy(p):
    '''determines the percentage of guys taller for a given set of couples'''
    c=0
    for i in range(len(p)):
        if p[i][0]> p[i][1]:
            c+=1
    percentage= float(c)/len(p)*100
    return percentage
```

Running this program for a number of pairings at a dance we found the following result:



What's interesting is that the system stabilizes to a value of 82%, which is much lower than I had expected. If I think about my life, I only know a few couples where the female is taller than the male, but in a more pure world where we randomly assign partners, and have no attraction preferences on height, this value seems reasonable.

It's exciting to think that we can use simple data to model trends and possibilities!

9.5.1 Streaks

Other probability situations include studying sports. People are good at ignoring truly random situations and noting things as trends: in sports, we love to find trends for our favorite players in terms of successes. Certainly there have been some serious streaks in the careers of some of sports bests: no one can forget Joe DiMaggio's 56 game streaks where he had at least one hit per game. However, in most cases we imagine streaks where there are none. Monte Carlo simulations, where we continuously and randomly sample a set, can be a good way to compute whether something we observe is meaningful.

We investigated the probability of observing a streak during a basketball season (made up of 82 games). Assuming a game has 10 players, and that each one will take 15 shots during the game with a 50% chance of scoring, what is the likelihood of observing a streak? A streak is defined as 10 consecutive wins or misses. We created a program that modeled a basketball game with this situation:

```
class Season:
    def __init__(self):
        self.streak=0
        self.games=[Game(n) for n in range(82)]
        self.count_streaks()

    def count_streaks(self):
        for game in self.games:
            self.streak+=game.streak

class Game:
    def __init__(self,n):
        self.game=n
        self.streak=0
        self.players=[Player(n) for n in range(10)]
        self.check_streak()

    def check_streak(self):
        for player in self.players:
            self.streak+=player.streak()

class Player:
    def __init__(self,number):
        self.shots=[self.shoot() for i in range(15)]
        self.number=number

    def shoot(self):
        '''makes a shot(1) with a 50% probability of making it'''
        return random.choice([0,1])

    def streak(self):
        '''sees if a player successfully had 10 consecutive shots or misses'''
        shot=self.shots[0]
        s=1
        for i in range(1,len(self.shots)):
            if self.shots[i]==shot:
                s+=1
            else:
                shot=self.shots[i]
                s=1 #restart counter

        if s>=10:
            return 1
        else:
            return 0
```

A season is made up of games, that are made up of players. A player keeps track of his/her streaks. For a simulation run for 100 seasons, we found the probability of observing a streak to be 1.6 streaks/season.

Though not painful, this simulation takes much longer to run by computer than by doing some simple hand analysis. It turns out the calculation for this program is relatively simple. There's a 50% chance of making a given shot, so that to make 10 consecutive shots/misses the probability is:

$$(\tfrac{1}{2})^{10}$$

Since a player of a game will take 15 shots, there are 5 opportunities for a player to have a streak. There are 10 players in a game. There are 82 games in a season. So if we multiply these values together we find:

$$(\tfrac{1}{2})^{10} * 5 * 10 * 82 = 4 \text{ streaks/season}$$

Our simulation would likely need to run longer to better approximate the analytical result.

9.6 Remarks

We can use computational modeling for a variety of situations. We can use it to predict situations and prove ideas to others. By reading this book, I hope you too will consider modeling the world around you!