Computational Modeling in Python

Version 0.0.6

Computational Modeling in Python

Version 0.0.6

Jayesh Gorasia

The original form of this book is LaTeX source code. Compiling this LaTeX source has the effect of generating a device-independent representation of a textbook, which can be converted to other formats and printed.

Preface

Acknowledgements

Contents

Chapter 1

Graphs

1.1 What is a graph?

If I instruct you to draw a graph, chances are, you would draw a visual representation of a data set, like a bar chart or an EKG. However, the graphs in this chapter would be different.

Think of a graph as a number of points connected together by lines. More formally, the points are called **nodes** and the lines are called **edges**. Graphs can be used to model problems like the Travelling Salesman[1] and the Seven Bridges of Konigsberg[2].

There are many types of graphs, most of which have special names which allude to their properties. Types of graphs are not mutually exclusive, meaning a graph can be classified in a few ways.

Simple graphs are undirected[3], have no self loops[4] and no more than one edge between any two different vertices.In a simple graph with p vertices, the degree[5] of every vertex is less than p.

A regular graph is one where each vertex has the same number of neighbors, i.e., every vertex has the same degree. A regular graph with vertices of degree k is called a k-regular graph or regular graph of degree k.

A complete graph has each pair of its vertices connected together by an edge. Complete graphs are subsets of regular graphs as if each pair of vertices are connected together, then the number of connections each vertex has would be the same.

A path in a graph is a sequence of vertices such that from each of its vertices there is an edge to the next vertex in the sequence. The first vertex is called the start vertex and the last vertex is called the end vertex.

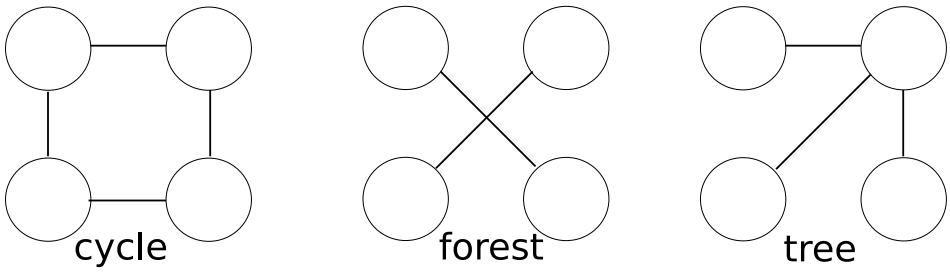
Other frequently used terms are:

Cycle A cycle is a closed path without self-intersections. Every vertex would have degree of at least two.

Forest A forest is a graph with no cycles.

Tree A tree is a connected graph with no cycles.

[1] `http://en.wikipedia.org/wiki/Travelling_salesman_problem`
[2] `http://en.wikipedia.org/wiki/Seven_bridges`
[3] The edges have no specified direction
[4] No edges from a vertex that leads back to the same vertex
[5] The degree is the number of vertices a vertex is connected to



1.2 Representing graphs

Graphs are usually drawn with circles or squares for nodes and lines for edges. In general the layout of a graph is arbitrary, although it can be made representative of the some property, such as the spatial coordinates of the vertices.

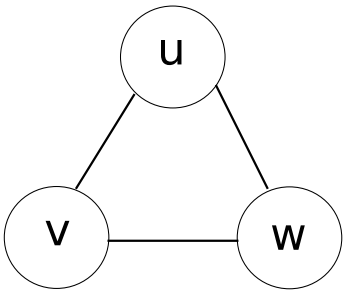
To implement graph algorithms, you will have to figure out how to represent a graph in the form of a data structure. But to choose the best data structure, you have to know which operations the graph should support. To get out of this chicken and egg problem, I am going to use a data structure from Computational Modeling and Complexity Science by Allen Downey. We will revisit it later to evaluate its pros and cons.

This data structure uses a dictionary with dictionaries nested in it, as shown below. The first dictionary has keys which correspond to the different vertices. The nested dictionaries' keys are the vertices to which the vertex is connected to, and the keys give what edge connects them.

Code Listing 1.1: Example of the Graph data structure

```
#(C) Allen Downey
{
  Vertex('w'): {Vertex('v'): Edge(Vertex('v'),Vertex('w'))},
  Vertex('v'): {Vertex('u'): Edge(Vertex('v'),Vertex('u'))},
  Vertex('u'): {Vertex('w'): Edge(Vertex('u'),Vertex('w'))},
}
```

The above code is equivalent to the figure below.



This structure lends itself to easy scalability, where additional functionality can easily scaffold onto.

There are many methods created for the Graph data structure. They are listed below. Note that this is not an exhaustive list, as more functions will be added as needed.

- **Add vertex** Adds a vertex to the graph
- **Add edge** Adds an edge between two edges
- **Get edge** Returns the edge between two vertices if it exists.
- **Remove edge** Removes the edge between two vertices if it exists.
- **Vertices** Returns a list of the vertices in a graph
- **Edges** Returns a list of the edges in a graph
- **Out vertices** Returns a list of the vertices connected to a vertex
- **Out edges** Returns a list of the edges connected to a vertex
- **Add all edges** Connects all vertices to each other. Thus a graph of n vertices will have a degree of $n-1$ for all vertices

When considering regular graphs, it is important to note that not all degrees are possible for a particular number of vertices. An easy rule to follow is to break graphs into those that have an even or an odd degree. The former can create a regular graph as long as the degree is less than the number of vertices, while the latter requires that the graph have an even number of vertices(The degree still needs to be less than the number of vertices). This is summarised in the following figure.

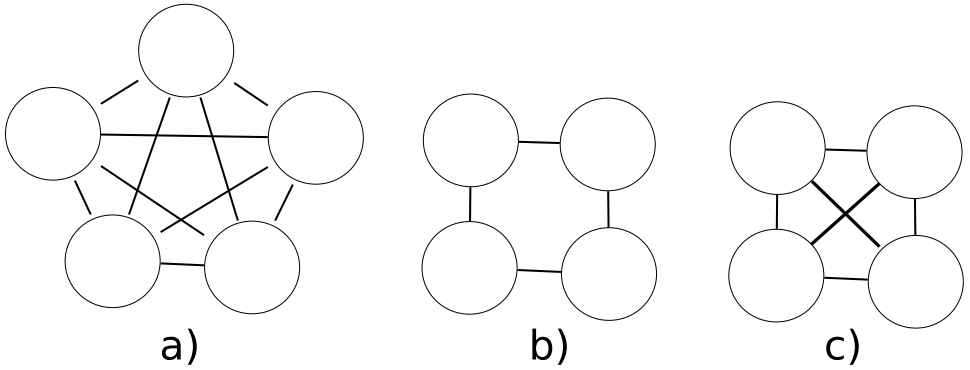


Figure 1.1: The three different cases for regular graphs: a)Odd vertices, even degree b)Even vertices, even degree c)Even vertices, odd degree

Therefore, when creating an algorithm for generating regular graphs, the number of vertices and degree need to be checked. If the degree is even, you could traverse a list of all the vertices and create edges to other edges. At each vertex, the number of edges created should be half the value of the degree[6]. This will prevent the creation of duplicate edges. Similarly for odd degrees, the algorithm should begin by generating a regular graph with a degree one less than the given degree(making the degree even). Then, an additional edge should be added between half the vertices and the vertices "across" the graph from them. An implementation is shown in Figure 1.2.

Another enhancement to the Graph structure would be to give the Graph a string representation(using Python's `__repr__ method`). This would allow multiple graphs with the same structure and labels to be created. For this to work, a new Vertex should not be created if one of the same name already exists. This

[6]E.g. for a degree of 4 you would create 2 edges

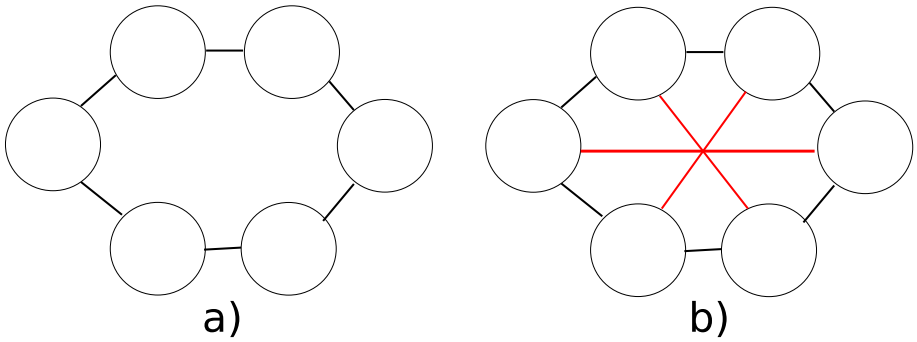


Figure 1.2: The algorithm for generating regular graphs of an odd degree. a)Create a regular graph with a degree one less than desired. b)Connect ”opposite” vertices together.

is a variation of the Singleton pattern[7] where we restrict the instantiation of objects to one per label. My implementation of this is:

Code Listing 1.2: Implementation of the Singleton pattern for the Vertex

```
#(C) Allen Downey

class Vertex(object):
        def __new__(cls, label):
        """if a Vertex with (label) already exists, return
        a reference to it; otherwise create a new one (and store
        a reference in the cache).
        """
        try:
            return Vertex.cache[label]
        except KeyError:
            v = object.__new__(cls, label)
            Vertex.cache[label] = v
            return v
```

As mentioned earlier, we are more concerned with simple graphs which have undirected edges. However, what if we want directed graphs? This would mean we need to tighten our usage of the edges. Currently, we do not distinguish between the order of which vertices appear in the name of an edge. We could define it such that an edge is from the first vertex to the second one. We could override the method `add_edge` to accomplish this.

Code Listing 1.3: Digraph class based on Graph

```
#(C) Allen Downey

class Digraph(Graph):
    def add_edge(self, e):
        """add (e) to the graph.

        If there is already an edge connecting these Vertices,
                the new edge replaces it.
        """
```

[7] http://en.wikipedia.org/wiki/Singleton_pattern

```
        v, w = e
        self[v][w] = e
```

To make use of these directed graphs, we can create a few methods to explore them. We can find out which vertices can be reached from a particular vertex, and which vertices lead to a vertex. An implementation for both is shown below as `in_edges` and `out_edges`.

Code Listing 1.4: Basic methods for digraphs

```
#(C) Allen Downey

class Digraph(Graph):
    def in_vertices(self, v):
        """return the list of vertices that can reach v in one
                hop"""
        return [d[v][0] for d in self.itervalues() if v in d]

    def in_edges(self, v):
        """return the list of edges into v"""
        return [d[v] for d in self.itervalues() if v in d]
```

1.3 Random graphs

A random graph is a graph with edges generated at random. Since there is no particular way to create random graphs, there exists many different models for creating them. One of the more popular kinds is the Erdős - Rényi model, denoted $G(n, p)$, which generates graphs with n nodes, where the probability is p that there is an edge between any two nodes.

An implementation of a random graph is to take an edgeless graph and loop through all vertices. For each vertex, loop through every other vertex and generate a random number between 0 and 1 for each pair. If the random number is above p, then create an edge between the two vertices. To keep the graph simple, there must not be more than one edge between two vertices. The code below is an adaptation of the add all edges code shown previously.

Code Listing 1.5: Random graph generation code

```
import random

class RandomGraph(Graph):
    def add_random_edges(self, p):
        temp = self.vertices()
        edges = []
        for k1 in range(len(temp)):
            for k2 in range(len(temp)):
                if k2 > k1:
                    if random.random() < p:
                        edges.append(Edge(temp[k1],temp[k2]))
        for e in edges:
            self.add_edge(e)
```

1.4 Connected graphs

A graph is **connected** if there is a path from every node to every other node.

There is a simple algorithm to check whether a graph is connected. Start at any vertex and conduct a search, noting every vertex that you can reach. Then check to see whether all vertices are marked. This sort of search is called a breadth-first-search (BFS).

An implementation of a BFS could be as follows:

1. Start with any vertex and add it to the queue.
2. If it is connected to any unmarked vertices, add them to the queue. Remove this from the queue.
3. Iterate on Step 2 until the queue is empty.
4. Check if all the vertices have been visited.

Code Listing 1.6: Algorithm to check if a graph is connected

```
#(C) Allen Downey

   def bfs(self, s, visit=None):
        """breadth first search"""

        # mark all the vertices unvisited
        for v in self.vertices():
            v.visited = False

        # initialize the queue with the start vertex
        queue = [s]

        while queue:

            # get the next vertex
            v = queue.pop(0)

            # skip it if it's already marked
            if v.visited: continue

            # mark it visited, then invoke visit
            v.visited = True
            if visit: visit(v)

            # add its out vertices to the queue
            queue.extend(self.out_vertices(v))

    def isConnected(self):
        """return True if there is a path from any vertex to
        any other vertex in this graph; False otherwise"""
        v = self.random_vertex()
        self.bfs(v)
        return False not in [v.visited for v in self]
```

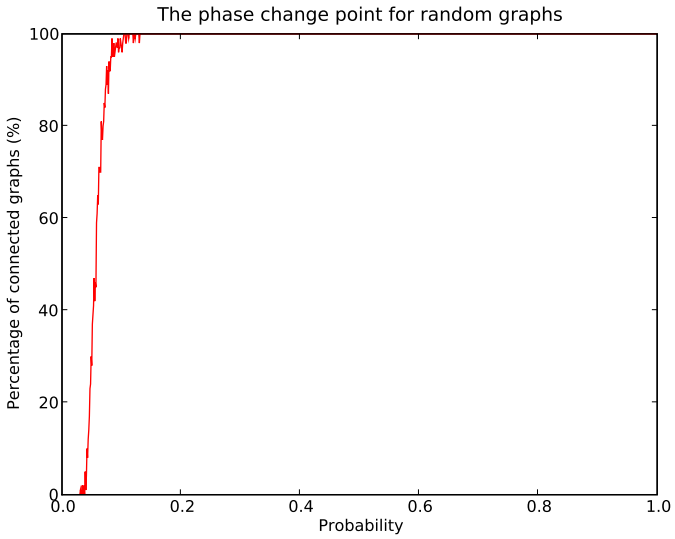


Figure 1.3: The plot shows the "phase change" apparent in large graphs. Notice that the change occurs at a very low probability, 2.8%

1.5 Erdős - Rényi model of random graphs

In the 1960s, Paul Erdős and Afréd Rényi wrote a series of papers introducing their model of random graphs and studying their properties.

One of their most surprising results is the existence of abrupt changes in the characteristics of random graphs as random edges are added. They showed that for a number of graph properties there is a threshold value of the probability p below which the property is rare and above which it is almost certain. This transition is sometimes called a 'phase change' by analogy with physical systems that change state at some critical value of temperature.

To test this, we will create a large number(1 thousand) of random graphs all of which have a large number of vertices(1 million) for a certain value of p. We will repeat this at fixed intervals of p, at a resolution of 0.001. This creates the following figure. Notice the nearly vertical change at around p is 0.03. This means that the phase change point is at $p = 0.028$ (I did a bisector search to get this value). If we use mathematical analysis, we can find that the sharp threshold for the connectivity of the a graph of n vertices with a probability, p, is at $\frac{lnn}{n}$. Therefore, the critical threshold would become smaller for larger graphs.

An interesting finding while conducting the experiment was that when the graphs had small numbers of vertices, the phase change was less abrupt, and was a visible slope for a decent length. For one, the low number of vertices mean that the error from the random number generator is magnified. Also, the importance of a single edge existing between two edges is magnified tremendously. If a vertex has 1000000 different vertices to randomly connect to versus having 10 vertices, the probability of it being connected is much greater.

My implementation of the search algorithm is shown below. It uses the RandomGraph class created earlier as the test piece, and loops through many different samples of random graphs, checking if they are connected and noting that. The code also plots the data with percentage of connected graphs on the vertical axis and the probability of edges being connected on the horizontal axis. (Note: This code is for a small sample and number of vertices.)

Code Listing 1.7: Algorithm to search for the phase change point

```
import pylab
import random
from Graphworld import *

def main(script, n='5', *args):
    # create n Vertices
    n = int(n)
    labels = string.lowercase + string.uppercase + string.punctuation
    vs = [Vertex(c) for c in labels[:n]]
    Result_tally = list()
    for r in range(991):
        Result_tally.append(0)
    def percent(x): return float(x)/1000
    Percent = map(percent, range(10,1001))
    # create a graph and a layout
    vs = [Vertex(c) for c in labels[:60]]
    for c in labels[:80]:
        for p in Percent:
             c = RandomGraph(vs)
             c.add_random_edges(p)
             if c.isConnected() is True:
                 Result_tally[Percent.index(p)] = Result_tally[Percent.index(p)] +1
    dy = list()
    for c in Result_tally:
        dy.append(float(c)/100*100)
    pylab.plot(Percent, dy, '-r')
    pylab.xlabel('Probability')
    pylab.ylabel('Percentage of connected graphs (%)')
    pylab.title('The phase change point for random graphs')
    pylab.show()
```

1.6 Small world theory

We now know the basics to explore small world theory. This will be given a more vigorous treatment later, but for now, consider what the phase change means physically. In Stanley Milgram's famous experiment[8] , the empirical result that a large group of random people could somehow be connected is surprising, as most of the people have no direct links with each other. However, as the previous graph shows, if the number of people being considered is large, there is a good chance that they will be connected as the probability that they are connected need only to be 2.8%.

1.7 Validity of the simulation

This will be a recurring topic throughout the book. When presented proof of a phenomena derived from a computer simulation many people have this skepticism about it. They would be much more inclined to believe an analytical proof which was vigorously done. This skepticism will be discussed in later chapters as we perform more complex experiments.

For the Erdős -Rényi model of random graphs, it is not a perfect representation of the real world as even in their analytical proof, they made simplifying assumptions in order to tackle the problem. For one, they

[8] http://en.wikipedia.org/wiki/Small_world_experiment

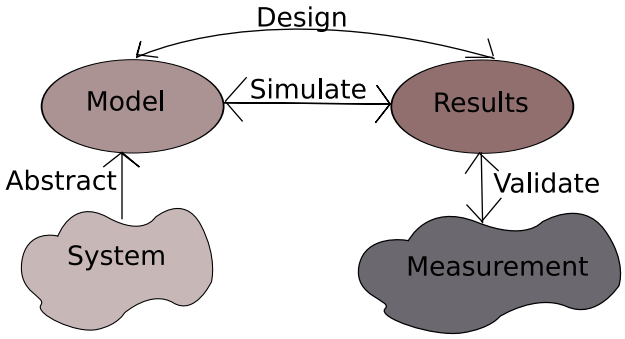


Figure 1.4: The philosophy behind simulations in this book. Diagram adapted from "Physical Modelling in MATLAB" by Allen Downey

make the assumption that the probability of an edge existing two vertices to be random and independent of interaction of other vertices. This is probably not true in reality, but statistically, this assumption can be made.

Look at the following figure. We generally want to understand a physical phenomenon, and will develop a model for this through abstraction. This model will put through simulation(or mathematical analysis) in order to learn more about it. Finally, we will try to validate the simulation results through measurements or some other mean. This cycle allows us to make predictions about systems without getting to hung up with details. These skills will be reiterated and reemphasised throughout the book.

Chapter 2

Analysis of algorithms

Analysis of algorithms is the branch of computer science that studies the performance pf algorithms and is used and to predict the performance of different algorithms in order to guide program design decisions.

If asked to rate the performance of an algorithm, it would be hard to give it a value. Many factors, including the architecture of the computer used to execute the algorithm, influence the result. There is not really any independent metric that can be used.

2.1 Order of growth

We can perform a relative comparison using order of growth analysis. This analysis compares the performance of algorithms at large values by noting the leading term in the algorithm. The algorithm with the larger leading term would take the longest to run.

More formally, an **order of growth** is a set of functions whose asymptotic growth behaviours is considered equivalent. A notation called "Big-Oh notation" is used to quickly summarise the order of growth of a system. It is important to note that the order of growth of a function is dependent only on the leading term, i.e. all others can be discarded. If we wanted to be more thorough, we would not end here, but a more rigorous treatment is best left to its own book.

2.2 Analysis of basic operations

It is most useful for programmers to know the performance of basic operations in programming, thing like sorting, looping and such. They are used frequently within programs, and knowing the right operation to use could be a decision made purely on the performance of the algorithms underlying them.

Later we will use lists for our analysis. Common operations for a list are to index it, sort it, append items and to delete items. Most list operations take a linear time. For instance, using a for loop to loop through a list would be linear if the body of the loop is constant time.

Sorting is a highly studied field where the speed of the algorithm is can be very important. Just think of the information on a database being in a giant list which needs to be sorted through to make the database useful. Many sorts are **comparison sorts** which uses a single comparison parameter to decide how to sort elements. This method has fundamental limits and has a typical growth order of $nlogn$.

At best, a worst case sort algorithm would have a quadratic order of growth and at best it would have a linear order of growth. In this context, we can see why simple bubble sort[1] would not be ideal as it has on average a quadratic order of growth. A radix sort[2] through, is not a comparison sort, and thus can have a typical order of growth of $O(n \cdot \frac{k}{s})$ where n is the number of items in the list, k is the size of each key, and s is the chunk size used by the implementation.

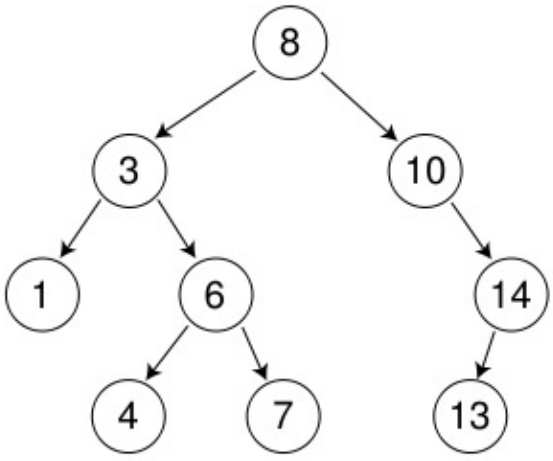
Since we brought up radix sorts, it would be wise to discuss **stability** of sorting algorithms. Most-Significant-Digit Radix sorts are unstable while Least-Significant-Digit Radix sorts are not. Unstable algorithms result in non deterministic sorted results, as they have particular difficulty when the values they are comparing are equal. Therefore, when deciding between sort algorithms, the question of stability may become important.

Lets compare the sorting performance of C and Python. C uses quicksort[3] while Python uses mergesort[4]. Both have *n log n* growth on average, but quicksort can result in quadratic growth. The reason this is brought up, is to point out that different languages have different inherent sorting algorithms, which may make the difference when choosing between a language to program in[5].

2.3 Analysis of search algorithms

We can intuitively tell that different search algorithms would have different orders of growth (That has been the theme thus far). If we have a sorted list, a simple way to search would be to loop through the entire list to find a particular value. This has a linear order of growth which is not ideal. A different data structure would be able to serve us better in searches.

Lets discuss instead a binary search tree[6]. A binary search tree(BST) is useful as a data structure as it can implement searching algorithms very efficiently. Look up the Wikipedia page to understand some of the subsequent syntax. Basically, a BST has a node and a left and right child. The left child is smaller than the node and the right is greater. Now, the children are also nodes, so that means they have their own children. This quickly develops into a tree where every value is sorted.



The previous figure is an example of a BST of size 9 and depth 3, with root 8 and leaves 1, 4, 7 and 13.

[1] http://en.wikipedia.org/wiki/Bubble_sort
[2] http://en.wikipedia.org/wiki/Radix_sort
[3] http://en.wikipedia.org/wiki/Quicksort
[4] http://en.wikipedia.org/wiki/Merge_sort
[5] One of my professors uses MATLAB for numerical simulations, but lapses into FORTRAN when he has a need for speed.
[6] http://en.wikipedia.org/wiki/Binary_search_tree

To implement a BST in Python, we would need to create two new classes, a node class for each node, and a Bst class for the BST. The node would have four attributes, a left child, right child, key and value. The Bst would need to have all the usual functionality, adding new nodes, deleting nodes, and searching through them all.

To add new nodes, we use the following Python code[7]. Basically, it will traverse downward through the tree, checking against all the nodes in its path, until it reaches a suitable position to add the new node.

Code Listing 2.1: Add method for BST

```
class Bst:
        def add(self, key, dt):
        """Add a node in tree"""
        if self.root == None:
            self.root = Node(value = key, data = dt)
            self.l.append(self.root)
            return 0
        else:
            self.p = self.root
            while True:
                if self.p.value > key:
                    if self.p.lchild == None:
                        self.p.lchild = Node(value = key, data = dt)
                        return 0 #success
                    else:
                        self.p = self.p.lchild
                elif self.p.value == key:
                    return -1 # value already in tree
                else:
                    if self.p.rchild == None:
                        self.p.rchild = Node(value = key, data = dt)
                        return 0 # success
                    else:
                        self.p = self.p.rchild
        return -2 #should never happen
```

To delete nodes, again you would traverse the tree, checking the value at each node until reaching the desired node. Just remember, to replace a node with one of its children if it has any.

Code Listing 2.2: Delete method for BST

```
class Bst:
    def deleteNode(self, key):
        """Deletes node with value == key"""
        if self.root.value == key:
            if self.root.rchild == None:
                if self.root.lchild == None:
                    self.root = None
                else: self.root = self.root.lchild
            else:
                self.root.rchild.lchild = self.root.lchild
                self.root = self.root.rchild
            return 1
        self.p = self.root
        while True:
```

[7] http://en.wikipedia.org/wiki/Binary_search_tree

```
            if self.p.value > key:
                if self.p.lchild == None:
                    return 0 #Not found anything to delete
                elif self.p.lchild.value == key:
                    self.p.lchild = self.proceed(self.p,
self.p.lchild)
                    return 1
                else:
                    self.p = self.p.lchild
            # There's no way self.p.value to be equal to key!
            if self.p.value < key:
                if self.p.rchild == None:
                    return 0 #Not found anything to delete
                elif self.p.rchild.value == key:
                    self.p.rchild = self.proceed(self.p,
self.p.rchild)
                    return 1
                else:
                    self.p = self.p.rchild
        return 0
```

Finally, to search through nodes, you would traverse down the tree until the desired value was reached.

Code Listing 2.3: Search method for BST

```
class Bst:
    def search(self, key):
        """Searches Tree for a key and returns data; if not
found returns None"""
        self.p = self.root
        if self.p == None:
            return None

        while True:
#            print self.p.value, self.p.data
            if self.p.value > key:
                if self.p.lchild == None:
                    return None #Not Found
                else:
                    self.p = self.p.lchild
            elif self.p.value == key:
                return self.p.data
            else:
                if self.p.rchild == None:
                    return None #Not Found
                else:
                    self.p = self.p.rchild
        return None #Should never happen
```

Making a common traverse method that all of the previous methods could have used may be a nice exercise for the reader.

This implementation of a BST is not too complex and is not the best way to implement a BST. Other variants of BSTs like Red-Black Trees[8] and Treaps[9] are known to have a better performance.

[8] http://en.wikipedia.org/wiki/Red-black_tree
[9] http://en.wikipedia.org/wiki/Treap

Note though that one of the miracles of computer science, hashtables[10] can implement searches in constant time, without requiring any sorting. Its pretty amazing.

As a final proof of the performance of an algorithm, we can always empirically determine the order of growth for some functions. Consider the summing of lists. If one uses a for loop which iterates through a list, appending it at each cycle, the algorithm would have a quadratic order of growth. However, if one uses the *extend* method for lists, one is just adding terms incrementally to the end of the list. The figure below can show this more clearly. The slope for the left graph is about two and the one on the left is about 1.

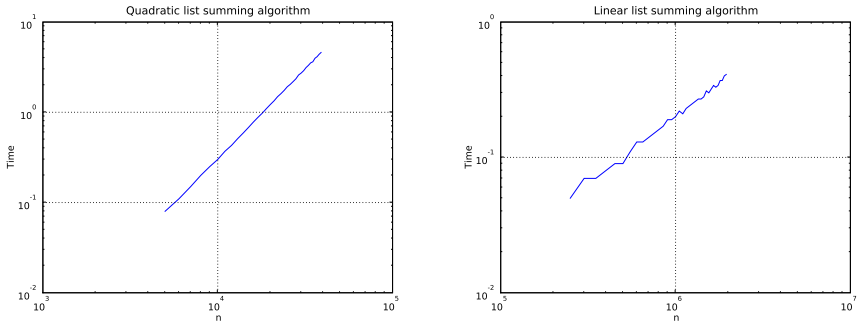


Figure 2.1: Different growths for linear summing algorithms. The left graph has quadratic growth and a slope of 2 while the right graph has linear growth and a slope of 1.

As a small note, it is useful to use list comprehension to loop through a list and perform operations on it. Here is the code for the previous list summing example(the linear algorithm).

Code Listing 2.4: Linear list summing algorithm

```
a = [1,2,3,4]
b = [5,6,7,8]

a.extend([x for x in b]) #Linear order algorithm
```

[10] http://en.wikipedia.org/wiki/Hash_table

Chapter 3

Small world graphs

3.1 Spiral back to graphs

Now that we know about analysing algorithms, lets take a look back at out graph implementation with new eyes. As Allen Downey's book shows, the different methods we have used for the graph methods have different orders or growth. Some of them could be better optimised especially if we expect to use them for larger graphs with complicated topographies. In the following chapters, we will try to review our algorithms and data structures to insure that they have low growth orders.

3.2 FIFO implementation

In that vein, we need to develop a FIFO stack. This datastructure may be useful later in implementing algorithms used for the small graphs.

A FIFO (First In First Out) stack is a datastructure where data can be inserted sequentially, which will be removed in the sequence that they were inserted. Our goal is to insert and remove data from the FIFO in constant time. This is a non-trivial task, as most FIFO implementations have a linear order. Double linked lists can be implemented as a FIFO in Python.

My implementation involves creating two new objects, nodes and the FIFO itself. Think of a double linked list as a series of nodes with two links. One link is to the next member in the list, while the other is to the next member in the list.

Code Listing 3.1: Node object for the double linked list

```
class Node:
    def __init__(self, value=None):
        self.previous = 0
        self.data = value
        self.next = 0
```

The FIFO object collects all these nodes and maintains order within them. As shown in Figure 3.1, the first node will have its previous link going to O, while the last node will have its next link going to zero. By knowing this, we are able to keep the list in order. In *append*, we keep track of the last node in our sequence, and simply add the node and necessary link when needed. In *pop*, we pluck out the first node and rewire the links for the second node, thus making it the new first node. This data structure is now ready to be used as a FIFO stack.

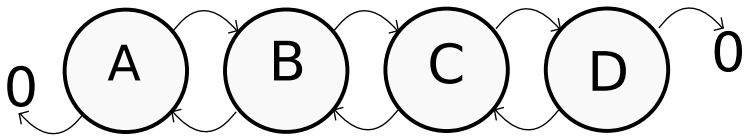


Figure 3.1: Illustration of the double linked list. Note that the first node has its previous pointer at 0 and the last node has its next pointer at 0.

Code Listing 3.2: Double linked list implementation

```
class FIFO:
        """Implementation of a FIFO using doubly-linked list"""
    def __init__(self, value=None):
        frst = Node(value)
        self.firstNode = frst
        self.nextNode = frst

    def append(self, value):
        node1 = Node(value)
        curNode = self.nextNode
        node1.previous = curNode
        node1.next = curNode.next
        curNode.next = node1
        self.nextNode = node1

    def pop(self):
        firstnode = self.firstNode
        try:
            secondnode = firstnode.next
            secondnode.previous = 0
        except:
            None
        try:
            self.firstNode = secondnode
        except:
            None
        return firstnode
```

3.3 Watts and Strogatz

In the 1990s, Watts and Strogatz published a paper which proposed an explanation for the small world phenomenon. They began with the two common known kinds of graphs, regular graphs and random graphs. Their proposal was that small world graphs existed in between these two extremes, offering interesting behaviour that neither end did.

They found that many interesting networks that physically exist are actually small world graphs and not random or regular graphs. Examples from their paper ranged from the power distribution networks in the US, to the neural network of the nematode worm *C. elegans*. This shows a certain robustness in their model as it scales well between very different kinds of networks.

To make a small world graph, they came up with an algorithm which begins with a regular graph. Arrange all the ring, and look at the edges connecting successive vertices in the ring. Given a certain probability, one can rewire these edges. Once all the vertices have been traversed, look at edges that connect vertices that are two apart in the ring. Do the same few steps repeatedly, for vertices that are further apart, until all edges in the graph have been reached.

In order to formally define what a small world graph was (Strogatz was a mathematician after all) they came up with two parameters from which to characterize graphs, clustering coefficient, $C(p)$ and average path length, $L(p)$. These parameters are best described if we imagine our small world graph to be a friendship network.

$C(p)$ describes how close your friends are to each other. A high $C(p)$ would mean that most of your friends are friends with each other, thus their is high clustering. $L(p)$ is average number of connections needed for you to know somebody. It is an average over everyone you could possibly know through your friends and their friends.

The algorithm for determining the clustering coefficient is not too complicated. From a starting vertex, find out all the vertices that are connected to it (Use the out vertices method). In this set of vertices, check every pair of vertices if an edge exists. We know that for n vertices, there can be a maximum of $\frac{(n)(n-1)}{2}$ edges. If we take the ratio of number of edges that exist and the number of edges that can exist, we get the clustering coefficient. Figure 3.2 can help illustrate this.

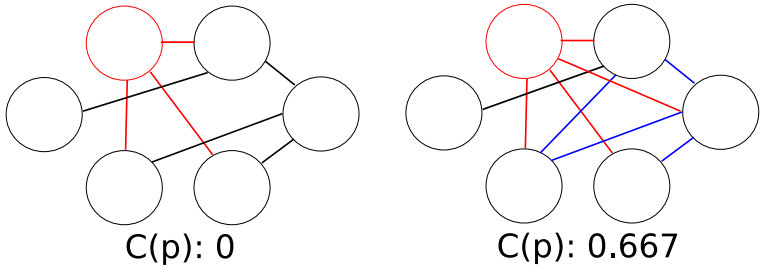


Figure 3.2: Graphs to illustrate the clustering coefficient.

Code Listing 3.3: Clustering coefficient algorithm for SWGs

```
    def clustering_coefficient(self,k):
        vertices = self.vertices()
        cluster = list()
        for v in vertices:
            cluster.append(0)
            out_vertices = self.out_vertices(v)
            length  = len(out_vertices)
            for i in range(len(out_vertices)):
                for j in range(len(out_vertices)):
                    if j>=i:
                        if self.get_edge(out_vertices[i],
out_vertices[j]) is not None:
                            cluster.append(cluster.pop()+1)
            if len(out_vertices)>1:
                cluster.append(float(cluster.pop())/..
(length*(length-1)*0.5))
            else:
                cluster.pop()
        return sum(cluster)/len(cluster)
```

Determining $L(p)$ is a more difficult process. The challenge is more in implementing an algorithm that is efficient, rather than coming up with a solution that might work just fine but break with large graphs. A Dutch computer scientist Edsger Dijkstra, created an algorithm which has a quadratic order of growth.

To implement this, we create a dictionary to store all the distance values of the vertices from each other, a list of all the vertices we have visited and a queue of all of the vertices to visit. Starting at a vertex, we label its distance as 0, and all the vertices it is connected to as 1. These vertices get added to the queue. Next, we return to the queue and remove the first vertex added to it and carry out the same process with a slight difference. We mark this vertex in the visited list. Now, the distance of the connected vertices will be the sum of the base vertex and the distance between the vertices (one in this case). This process is repeated until all the vertices are marked. This will give us a dictionary with the distances of all the vertices from the starting vertex. We sum this up and average it for the total number of vertices in the graph. Finally, repeat using all the other vertices as the starting vertex. Sum and average again, and the result is $L(p)$.

We could and should have used the FIFO stack developed earlier in the algorithm. However, we would need to make the FIFO iterable. Although implementing this in Python would not be difficult, it is an exercise I leave to the reader. My justification was that the run time of this algorithm was well within my tolerance, beyond which I would crack down on errant algorithms.

Code Listing 3.4: Average path length algorithm for SWGs

```
    def avg_path_length(self):
        cumsum = 0.0
        avgPL = 0.0 #Average path length
        verts = self.vertices()
        numv = len(verts)
        for i in range(numv):
            queue = []
            marked = []
            distance = dict()
            for a in verts:
                if a == verts[i]:
                    distance[a] = 0
                else:
                    distance[a] = 'Inf'
            cumsum += self.one_path_length(verts[i], queue,
marked, distance)
        avgPL = cumsum/(numv*numv)
        return avgPL

    def one_path_length(self, vert, queue, marked, distance):
        verts = self[vert].keys()
        numvs = len(verts)
        curdist = distance[vert]
        # puts all of the vertices that the current vertex
        # is connected to into the queue if they aren't on
                # it yet or aren't on the marked list
        for i in range(numvs):
            if verts[i] not in queue and verts[i] not in marked:
                queue.append(verts[i])
                distance[verts[i]] = curdist + 1
        if len(queue) > 0:
            marked.append(vert)
            popped = queue.pop(0)
        # if the queue has more, recursive call BFS with
next vertex in queue
        if len(queue) > 0:
```

```
            return self.one_path_length(popped, queue,
marked, distance, distnum)
        else:
            return sum(distance.values())
```

Figure 3.3 illustrates the average path length algorithm. Although it does not really help explain my particular implementation, it gives an idea how the algorithm results in the correct distance values for the vertices.

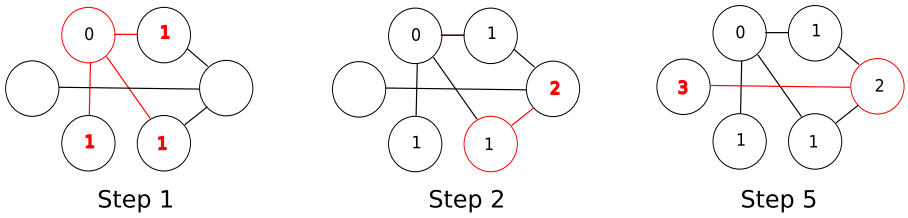


Figure 3.3: The graphs show the different stages in the average path length algorithm. The red circle marks the node that we are currently at. Step 1 is at the initial vertex, Step 2 at the next one, and Step 5 is at the last node. (Step 3 and 4 do not cause changes to the distance values marked in red)

In Watts and Strogatz's 1998 letter to *Nature*, they used a figure which demonstrates how the two parameters vary with different rewiring probabilities. This is reproduced in Figure 3.4 using the implementations described in this chapter. The figure is a very close reproduction of their figure, which validates the implementation above.

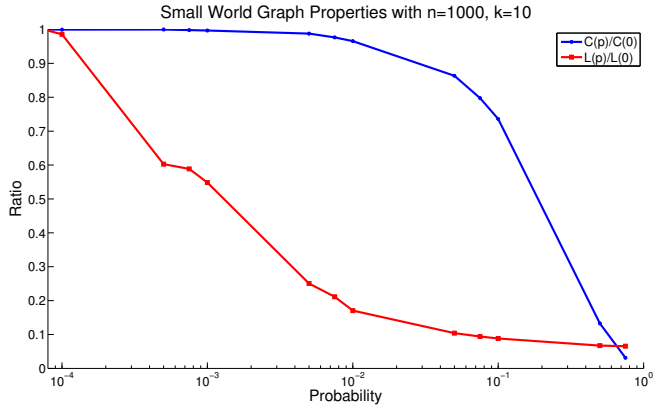


Figure 3.4: Reproduction of Watts and Strogatz's figure which demonstrates that small world graphs have low path lengths and high clustering coefficients

One lingering question is whether the rewiring process had a significant impact on the nature of the graphs. This would cast doubt on the completeness of the small world graph theory, as it would proof that the figure above is the exception, rather than the rule. Using the rewiring algorithm from Stanislaw Antol's *Adventures in Modeling*, I produced Figure 3.5. It shows the same trends and is effectively identical. This seems to suggest that mimicking Watts and Strogatz's algorithm is not essential to demonstrating the behaviour of small world graphs. Of course, this statement comes with the caveat that both algorithms attemped to

replicate Watts and Strogatz's algorithm, and deviations may have been accidental rather than planned out.

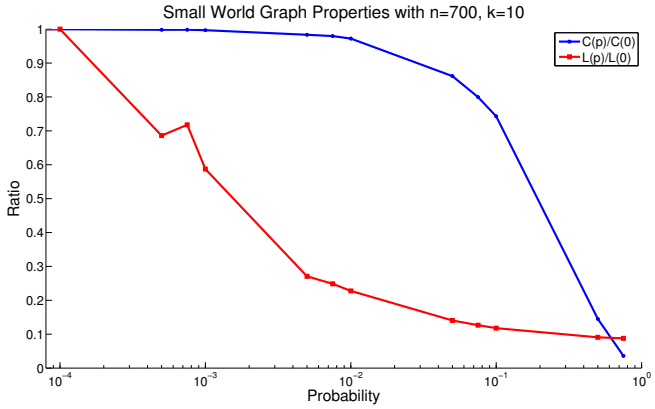


Figure 3.5: A similar figure but using a different rewiring algorithm. The striking similarity of the this figure and Figure 3.4 reflects the robustness of the small world graph phenomenon.

3.4 Why do I care?

As alluded to earlier, small world graphs are good models for many physical networks, like the power distribution network in the US. In 2003 [1], there was a massive power failure in the US. A fault in a single transformer at one node propagated along the network, eventually taking out most of the East Coast.

We could easily model each city (or power station) as a node and the transmission cables as edges[2]. Small would graphs are inherently more robust as they have high clustering and low path lengths. Knowing this, we can modify existing physical networks to become small world networks in order to improve their robustness. Its almost like having a rule of thumb for networks, do x and y and see higher reliabilities instantly[3].

[1] http://en.wikipedia.org/wiki/Northeast_Blackout_of_2003
[2] We would have to extend out implementation of edges to have a length as not all edges would have the same length
[3] Sounds like a cheesy advertisement as well

Chapter 4

Scale Free Networks

4.1 Zipf's Law

Zipf's law describes a relationship between the frequencies and ranks of words in natural languages[1]. More specifically, it predicts that the frequency, f of a word with rank r is:

$$f = c \cdot r^{-s}$$

where s and c are parameters that depend on the language and text. In order to test this, I downloaded a copy of *Christmas Entertainments by Alice M. Kellogg* from `gutenberg.net` and analysed the frequency of words that appeared in it. The words were then ranked in terms of decreasing frequency. In order to test Zipf's law, I took the logarithm on both sides of the equation to get:

$$\log f = \log c - s \log r$$

Therefore, a plot of my empirical results produced Figure 4.1. The result give credence to the validity of Zipf's law. Just to be safe, I carried out the same analysis with two other books. I observed that Zipf's law still held, as their plots produced straight lines on the loglog scale.

4.2 Cumulative distributions

We know that a cumulative distribution(CD) shows the percentage of values less than or equal to x for a range that x that sweeps from the smallest value in the set to the highest. To implement a CD(as a new Python class), I will use the Histogram(Hist) datastructure defined in `greenteapress.com/compmod/Hist.py` as the input.

When initialized, the CD will run through a list of all the values in the histogram, creating a separate list with a running total of the frequency of all the values thus far. A method `print_cdf` prints out the values and their corresponding cumulative frequencies which is useful for debugging. There needs to be two lines per quantity as the values from the histogram are discreet and not continuous.

Another method `percentile` takes a value and returns the corresponding percentile. This was implemented by using the fact that the list of values is sorted. We first search for the index in the values list

[1] `http://wikipedia.org/wiki/Zipf's_Law`

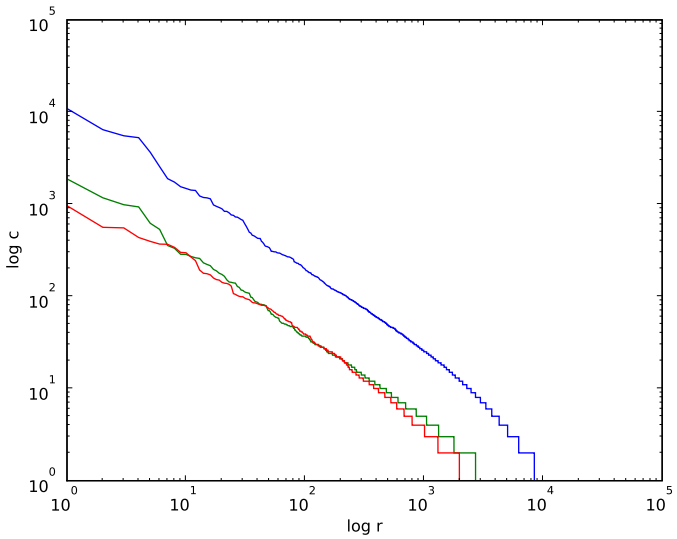


Figure 4.1: The empirical results of frequency analysis of the words in *Christmas Entertainments* (displayed in red). It clearly shows a nice linear slope and a y-intercept. For *History of the United States by Charles A. Beard*(blue) and the *Illustrated History of Furniture by Frederick Litchfield*(green), we observe that they all share a similar slope as they are all in English. Their y-intercepts are different as the styles of the books are different.

where a value would fit in, then use that index to find the corresponding cumulative frequency in the cumulative frequency list. After dividing by the total frequency of the entire set of values, we will get the percentile.

Finally, we will need a `plot_cdf` method which would plot the CD. I should make the dots for each step distinct, one filled and one empty, but I found that to be unnecessary. For the sizes of the datasets I am expecting to use the CD for, it would be impossible to see the individual steps without zooming in a fair amount.

Some attention should be given to the performance of the efficiency of the datastructures used in the CD. In the respect, `percentile` was designed to be *O(logn)* as it used the `bisect` method to search through the values list for the needed index value. Everything else in CD is linear in n.

4.3 Closed-form distributions

We have only seen empirical distributions thus far, distributions that are based on empirical observations. There exists a different kind of distributions called closed-form distributions as they can be expressed by some kind of closed-form function. For example, the length of time a person is waiting in a line can be modelled as an exponential distribution[2].

Speaking of exponential distributions, its CDF is:

$$cdf(x) = 1 - e^{-\lambda x}$$

[2] `http://en.wikipedia.org/wiki/Exponential_distribution`

Now, lets define the complementary distribution (CCDF) to be $1 - cdf(x)$. If we take the logarithm of both sides of the CCDF, we will get:

$$\log y = -\lambda x$$

Therefore, on the log-y scale, we would expect the CCDF to look like a straight line with a slope of $-\lambda$. To prove this, we can use the `expovariate` function in the `random` module to generate random variables from an exponential distribution. For 44 samples from an exponential distribution with a mean of 32.6, we could create the CDF and CCDF plots in Figure 4.2. The roughness of the curves is a result of the discretization of the data, as we only had 44 data points for the entire range of values. If we had a larger sample size, the curves would be smoother

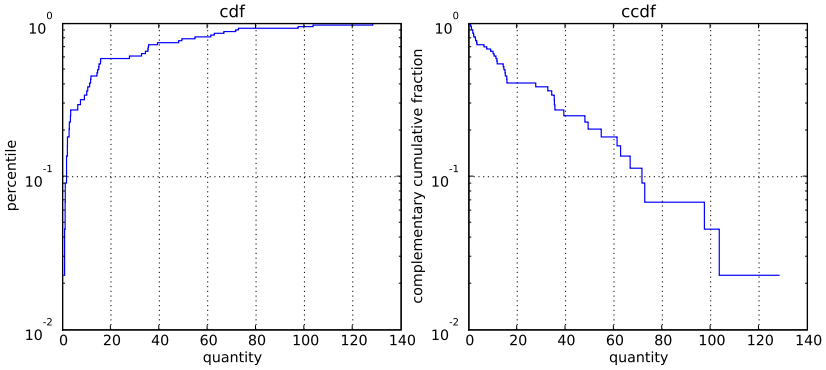


Figure 4.2: The CDF and CCDF for a sample of 44 from an exponential distribution with mean 32.6. Notice the straight line in the CCDF which means that it has a constant gradient.

4.4 Pareto distributions

Another interesting closed form distribution is the Pareto distribution. It has been used to describe a variety of phenomena, from populations of cities, to the distribution of wealth. Its CDF is given by:

$$1 - \left(\frac{x}{x_m}\right)^{-\alpha}$$

The parameters x_m and α determine the scale and shape of the distribution.

Pareto distributions have three interesting properties. Firstly, they have a long, sometimes called **heavy tail**. In contrast, normal distributions have weak tails. This can be interpreted as Pareto distributions have many small values and only a few very large ones. Building on that, another property is their "**scale free**" nature. Typically, bounds of a distribution is about two or three standard deviations away from the mean. Pareto distributions do not have this sort of range, thus get the name "scale-free". Finally, Pareto distributions shows existence of the 80/20 rule, where 80% of the values are contained within a narrow 20% band[3].

The CCDF for a Pareto distribution is given by

$$y = 1 - cdf(x) \sim \left(\frac{x}{x_m}\right)^{-\alpha}$$

Therefore if we take the log of both sides, we will get a straight line with slope $-\alpha$ and intercept at $-\alpha \log x_m$

[3]Just think of how 80% of the words in this book is fluff and how 20% of this book takes up 80% of my time to write.

Now that we know this, lets try to make some use of this knowledge. The distribution of populations for cities and towns could potentially be described with a Pareto distribution. The good folks at the U.S. Census Bureau publish the data on the population of every incorporated city and town in the United States[4]. Using the script from `greenteapress.com/compmod/populations.py`, we can parse the data provided by the Census Bureau and create a list with all the population values.

Before we proceed(I get that this is a jarring break), we need to add a function `quantile` to `Dist` which will take percentile as a parameter and return the corresponding quantity. This will allow us to find out what values are at the different percentile values of the distribution. We first multiply the percentile with the total frequency of the quantities to get the total frequency up to that percentile. Next, use the index of this value to look for the corresponding quantity in the quantity list. Thus, we we will get the quantity corresponding to a percentile, the quantile. For our census data, we learn that the median size is 1276, and the 25th and 75th percentiles are 400 and 5335 respectively.

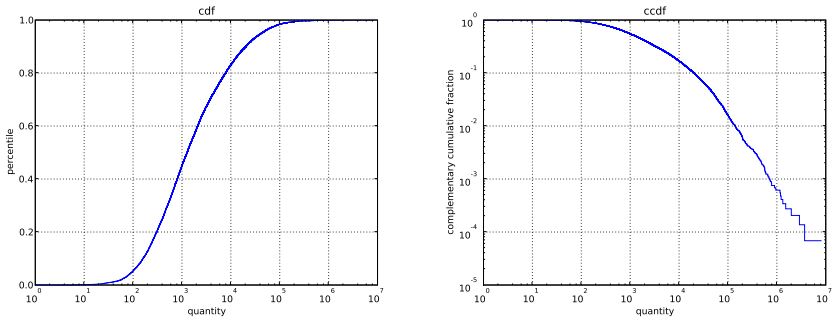


Figure 4.3: The CDF and CCDF for the population in the towns and cities in the US. Notice how this conforms to a power law distribution. Characteristics to look for is the straight line in the loglog plot in the CCDF and the S shape in the semilog-x plot of the cdf.

In Figure 4.3, I made the cdf and the ccdf of the census data. We can conclude from Figure 4.3 that the population of towns and cities in the US follows a power law distribution.

4.5 Barabási and Albert

Previously, we have encountered random graphs of the Erdős Rényi variety, and small world graphs of the Watts and Strogatz. Barabási and Albert (B & A) introduced another kind in their 1999 paper[5].

They claim that graph is a more accurate representation of physical phenomena because of two factors of real networks that it takes into account. Firstly, they allow for the expansion of the network over time, whereas the size of the network is defined at the start for random graphs and SWGs. This is more dynamic and may, intuitively make more sense as physical networks(like the nervous system) are always growing. Secondly, the attachment of new nodes is preferential in B & A graphs, whereas it is random in the others. Their argument for this is that empirical evidence in real networks points towards this. New webpages would tend to be linked to more prominent websites, while more obscure ones would less likely be linked to. Basically, its a situation where the rich get richer.

To prove their ideas, B & A came up with a formal model which they subjected to computational simulation. In their model, they started with a network with a small number of vertices, which they then added vertices to. The new vertices had a probability of $\Pi(k_i) = k_i / \sum_j k_j$ of being connected to a vertex of degree k. In

[4]Even Wasilla!

[5]"Emergence of Scaling in Random Networks" Science

other words, the degree of a vertex over the total degree of the graph, is the probability of a vertex forming an edge with a new vertex. We can postulate that as we add more vertices, the existing vertices with high connectivities would get more new edges, the rich get richer phenomenon.

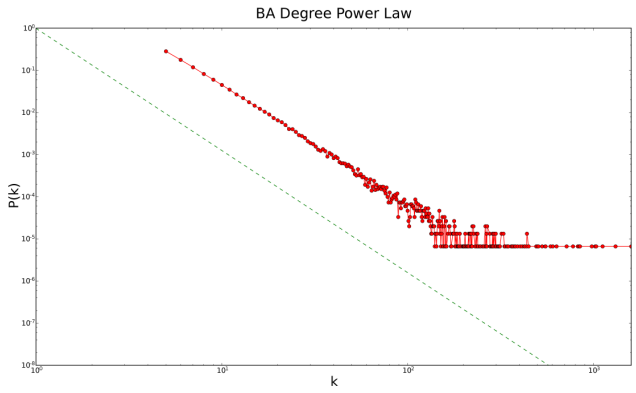


Figure 4.4: The Pk ccdf of a B & A graph against the degree of its vertices. Notice how the number of vertices with a large degree is small, while there is a large number of vertices with a small degree. The line in green is the asymptote of $k^{-\gamma}$. It is slightly translated due to some unresolved technical issues.

Using their algorithm, I implemented a B& A graph. Figure 4.4 shows the trend of the vertices and their degrees. B & A predicted that $P(k)$ is asymptotic to $k^{-\gamma}$, therefore, the ccdf of the distribution would be a straight line, as Figure **??** shows. The algorithm I used is optimised to run faster as the probabilities it looks for it normalised. For example, when the maximum degree of the graph is 6 and the total degree is 30, it will generate a random number between 0 and 6/30, thus making it more likely that a random edge would be created in a cycle.

Code Listing 4.1: Method to add vertices to B & A graphs

```
class ScaleFreeGraph(SmallWorldGraph):
    def __init__(self,mnot,m,t):
                """
                Mnot:initial number of vertices,
                m:number of edges added at each timestep
                t: number of timesteps
                """
        vs = [Vertex(str(c)) for c in range(mnot+t)]
        SmallWorldGraph.__init__(self, vs)
        timesteps = t
        mcount = 0
        maxk = 0.0; #Maximum degree of a vertex
        verts = self.vertices()
        while mcount < m: #initial loop to create one highly connected vertex
            r = random.choice(verts[:mnot])
            e = Edge(r, verts[mnot])
            if self.get_edge(r,verts[mnot]) is None:
                self.add_edge(Edge(r, verts[mnot]))
                mcount +=1
        maxk = m
        for i in range(2,timesteps+1): #loop through all timesteps
            mcount =0
            while mcount <m: #loop to make m edges for vertex i
                r = random.choice(verts[:mnot+i])
```

```
                total = m*(i-1)*2
                r_p = float(len(self.out_vertices(r)))/total
                r_num = random.uniform(0,maxk/total) #normalisation of the
random number generation
                if r_num <r_p:
                    if self.get_edge(r,verts[mnot+i-1]) is None:
                        self.add_edge(Edge(r, verts[mnot+i-1]))
                        mcount += 1
                        maxk = max(maxk, len(self.out_vertices(r)))
```

If we look at the small world characteristics of the B & A model, we find that they do not exist. For a graph following the B & A model and having 1000 vertices, we will end up with an average path length of 17.76 and a clustering coefficient of 0.293. If we scale these numbers, we will get an average path length of 0.280 and a clustering coefficient of 0.976. This corresponds very well to a small world graph.

However if we look at a small world graph from the lens of a B & A graph, we get different results. Looking at Figure 4.5 we see that it only sort of has the P(k) versus k trend that we seek. There is no power law relationship evident in the graph.

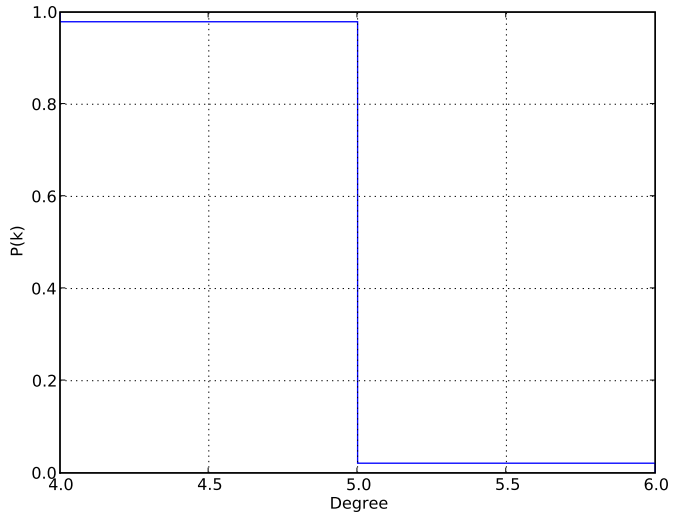


Figure 4.5: The Pk ccdf of a SWG graph against the degree of its vertices. The SWG has 1000 vertices with a p of 0.0075. The trend of P(k) is close to what we expect, as it has the downward slope. However, it not on a loglog scale. Therefore, the distribution of the P(k) does not follow a power series.

Therefore, it is possible to say that B & A graphs have characteristics of SWGs but SWGs do not have the characteristics of B & A graphs.

4.6 Wrapping up

You may have noticed that Zipf's law and the Pareto distribution create very similar plots. This is because Zipf's law is an approximation of the Pareto distribution for discrete intervals. If we went through the derivation for both, you will be able to see their links more readily.

This section brings an end to our study of graphs. It is interesting to note how we had many different models for graphs, all which had their separate roots and assumptions. Knowing where these models were most applicable is all a matter of perspective to me. Depending on the framing of the question, one model can look superior to the others. For example, in the Watts and Strogatz's small world graphs, high clustering happens in a small spatial region, while in B & A graphs, high clustering is evident for vertices that have been around for a while. Whats to say that the high clustering the the W & S graphs is from some old vertices (or popular vertices) that have gained a lot of connections. Its all a matter of perspective to me.

Chapter 5

Cellular Automata

Imagine a line broken up into discrete chunks. Let's call each chunk an automaton. An automaton is a "machine" which is able to perform computations. Using simple rules, automata can be used to perform complex calculations. Since the rules usually do not contain any random elements, they are considered deterministic. This means, we are able to predict the progression of automata in time, given the initial conditions.

5.1 Wolfram's model

Wolfram spent a lot of time exploring one dimensional cellular automata, systematically dividing them up into four classes:

- Class 1 cellular automata produce trivial results. Irregardless of initial state, the system will evolve into a unique homogeneous state.
- Class 2 cellular automata have the feature that the effects of particular site values propagates only a finite distance. Thus a change in the value of a single initial site affects only a finite region of sites around it.
- Class 3 cellular automata have the property that features will propagate at a finite speed forever, and therefore affect more and more distant sites as time goes on. They do have the quality that their features appear random, although they are not really random.
- Class 4 cellular automata have a much greater level of unpredictability, One of its features is that repeating patterns emerge irregardless of the initial state.

One dimentional CA can exist in three configurations: a finite sequence, a ring, and an infinite sequence. We will concern ourselves mainly with the ring configuration as it tightly bound.

As mentioned previously, the evolution of a system is governed by rules. If we consider a cellular automaton to be only affected by its immediate neighbours, we could construct a table to map combinations of the states of the automata to the new state of the automaton. If the cells only have two different states, 0 and 1, we could create a table as follows:

Previous	111	110	101	100	011	010	001	000
Next	0	0	1	1	0	0	1	0

This rule can be summarised as "Rule 50" as 00110010 is 50 in binary. Figure 5.1 shows the evolution of a single cell using Rule 50 over 10 time steps. The triangular shape of the figure is typical for CAs of this type. Rule 50 is an example of a class 3 CA as the triangular pattern will propogate infinitely in time.

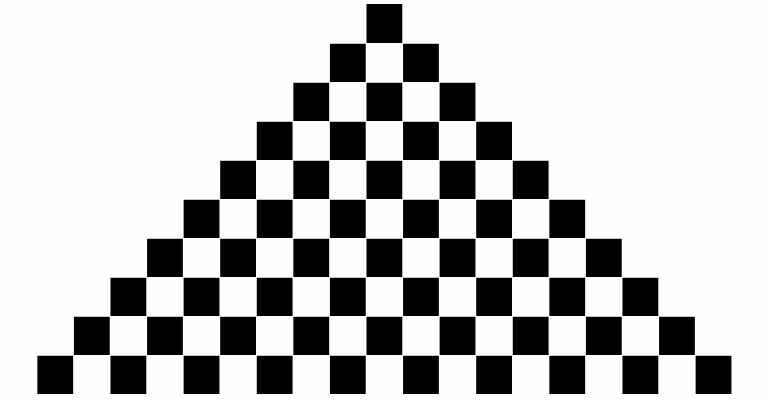


Figure 5.1: A Rule 50 CA running for 10 timesteps.

5.2 Implementing CAs

A simple way to implement a CA is to use a matrix where the columns represent the CA and the rows represent the different timesteps. The Numpy package provided a datastructure called an array which provides precisely that. Using an implemetation created by Downey[1], we have a simple method of manipulating one dimensional CA.

His implementation was made for the finite sequence configuration of CA. This can be easily modified into the ring configuration using ghost cells. Below is the iterative function he uses to cycle through the CA at each timestep. By adding a ghost column to the beginning which mirrors the end and a ghost column to the end which mirrors the beginning, we can continue to use the same step function with a minor change to the indices used.

Code Listing 5.1: Function to create a new row of CA for a ring configuration

```
def step(self):
    """execute one time step by computing the next row of the array"""
    i = self.next
    print i
    self.next += 1
    self.array[i-1,self.m-1]= self.array[i-1,0]
    neighborhood = tuple([self.array[i-1, self.m-2],self.array[i-1, 0],self.array[i-1, 1]])
    self.array[i,0] = self.table[neighborhood]
    for j in xrange(1,self.m-1):
        neighborhood = tuple(self.array[i-1, j-1:j+2])
        self.array[i,j] = self.table[neighborhood]
```

5.3 Randomness

One of the examples of a Class 3 CA is a rule 30 CA. We know from Downey's book that it can be used as a pseudo-random number generator(PRNG). This sort of PRNG is effectively random and and can be used when a random number is needed to be generated.

Another kind of PRNG is a linear congruential generator. It has the form below:

$$X_{n+1} = a(X_n + c)\%m$$

[1] greenteapress.com/compmod/ca.py

If we use appropriate values for a, c and m, we have an effectively random number generator. In order to test this, I plotted the cdf of 100000 successive values from the generator above. If the distribution is uniform, we can make a claim that the generator is random as it produces values that are well spread out. As Figure 5.2 shows, the generator produces a dataset which has a uniform distribution. Barring a more thorough analysis, we can make the claim that it is a good PRNG.

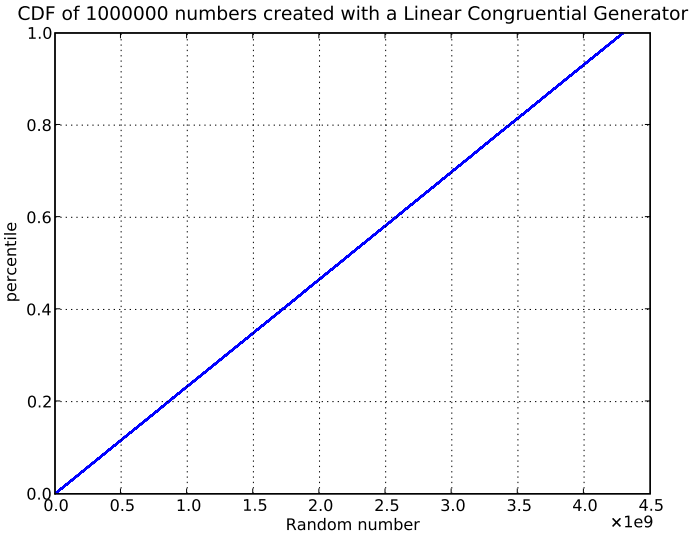


Figure 5.2: The CDF for 100000 values generated by a LCG with $a = 1664525$, $c = 1013904223$, $m = 2 * *32$ and a seed value of 7898979. Since the CDF is a straight line, it is could representing a uniform distribution of numbers.

We can test the ability of a rule 30 CA in generating random numbers by outputing the center column of the CA as the CA evolves. The results for running a rule 30 CA on a ring with 300 cells is shown in Figure 5.3. This time we test the least significant 10 bits of the center column (we will have 290 numbers as before that, the center column is less than 10 bits) for randomness using the STS toolkit from NIST[2]. I found that the numbers generated by the CA is essentially random, with an accuracy of 0.0001, which is acceptable.

5.4 Turing Machines and CA

One outcome of class 4 CA being able to result in patterns means that they are said to be Turing complete, which means that they can compute any computable function. Yes, **any** computational function. Lets show some examples first, in order to make this discussion less abstract, lets examine what the outcome of a class 4 CA, a Rule 110 CA, looks like.

We know from Downey's book that with an initial condition of a single cell, the CA looks like it produces some patterns but nothing truly meaningful. However, if we have a random initial condition, and let the CA evolve for 600 timesteps, we get Figure 5.4, which is a lot more interesting.

Perusing the figure, you will be able to notice some distinct features. The repeating structures, some of which translate diagonally and some of which translate vertically. These structures are sometimes refered to as spaceships.

[2] http://csrc.nist.gov/groups/ST/toolkit/rng/index.html

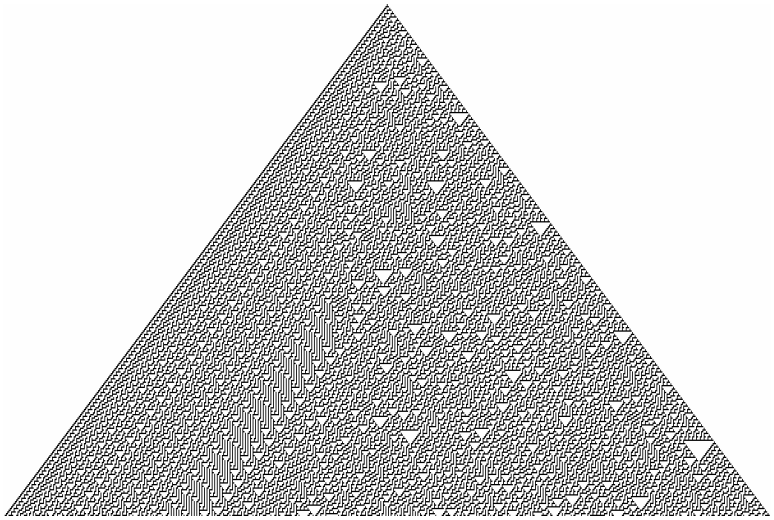


Figure 5.3: A rule 30 CA run for 300 timesteps. Notice that although there are apparent patterns in the CA, at a close enough scale, it could be completely random.

We can notice the ability of the CA to perform computation by observing the collisions between different spaceships. If we imagine the spaceships to be signals and the collisions to be logical operators, we can begin to imagine a system where the collisions of spaceships becomes analogous to the processing of signals. This is a powerful result as we can create spaceships that we want, and preplan collisions using the initial conditions. For instance, using an initial sequence of 0001110111 will result in a spaceship travelling diagonally to the right.

5.5 Falsifiability

When confronted with a theory that seems to be based on pseudoscience, the supernatural or the divine, we are confronted with the need to demark them from what we consider real science. This is the essence of the demarcation problem. Karl Popper thought he had solved the problem when he came up with the idea of falsifiability.

Falsifiability rests on the idea that there exists an experiment, that would contradict the hypothesis if it were false. Therefore, if we come up with a theory, and then proposed an experiment that could be used to contradict the theory, we have come up with a falsifiable theory.

Depending on the goal of your theory, having a unfalsifiable theory might be appealing, as you will never be proved wrong. However, if you want to use your theory to make predictions about the world, then an unfalsifiable theory would be useless.

Unfortunately, philosophers are strongly critical of Popper's philosophy of science, mainly because of Popper's mistrust of inductive reasoning. Inductive reasoning is the process of reasoning in which the premises of an argument are believed to support the conclusion but do not entail the premises. It is like saying since this ice is cold, therefore all ice is cold. Popper's philosophy prevents this from being accepted as science as it says nothing about how, in this case, ice could be proven to be not cold. Therefore, as in most phisophical argument, depending on where you stand on the issue, the argument is either resolved or stuck.

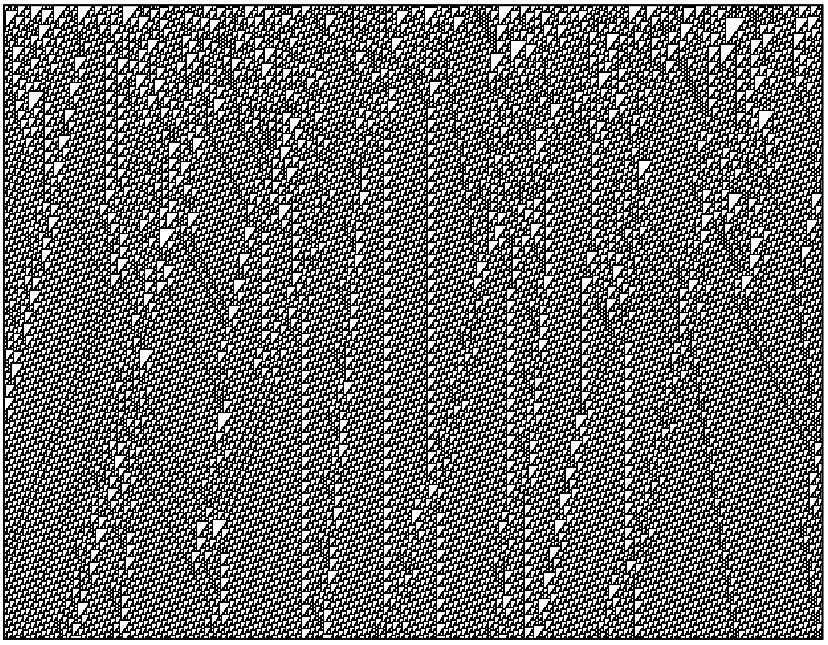


Figure 5.4: This is a rule 110 CA, with a random initial condition, run for 600 timesteps. Notice how the noise at the begining evolved into distinct patterns which display repeatable, predictable patterns.

Chapter 6

Games of Life

Conway's Game of Life(GoL), is a two dimensional cellular automaton created by the British mathematician John Conway. It is known to result in many interesting patterns, sparking discussion about realism and instrumentalism.

6.1 Abstract classes

Before we delve into the GoL, lets touch on an important construct in software design, **abstract classes**. In the previous chapter, we used pylab to generate figures for us. Since the figures were vector graphics, they took a long time to render and manipulate. Therefore, it would be nice to have another way of producing figures. In addition, it would be nicer if the other way had an identical interface to previous way, making it easy for the user to flip between using the different methods.

Abstract classes provide a means for this, by providing an interface to a class, but leaving the implementation up in the air. Thus, in our case, we would create a class, `Drawer` that provided our three main needs:

- **draw** : Draw a representation of cellular automaton. This function would have no visible effects.
- **show** : Display the representation on the screen, if possible.
- **save** : Save the representation of the CA.

We would then create separate classes which inherit from `Drawer`, and provide the implementations for these methods. Python does not enforce the creation of abstract classes, thus it is entirely possible for the subclasses to modify their interface. Therefore, care must be taken to adhere to the abstract class, as Python will not call you out if you violate it.

Since we already have an implementation for `Drawer` using Pylab, we just have to modify it to have the appropriate interface. Another method to generate images is using the Python Imaging Library (PIL). The steps for generating images using the PIL is simple:

1. Import Image and ImageDraw
2. Create a new image and specify its size in pixels. It is useful to set a constant size and scale the size of the individual cells appropriately.

3. Loop through the array containing the cellular automaton and draw rectangles at prescribed intervals using the `ImageDraw.Image.Rectangle` method. All that is needed is to specify the the coordinates of the top left and bottom right corners. Crucially, the base image is black in color, thus, draw rectangles in white when encountering 0s in the array.

Some exceptions need to be made to make sure that the cells created are at least a 2x2 pixel square, in order to not to break the PIL and create strange images. Saving and showing images is trivial, as the PIL has built in methods that accomplish this in single lines. Most amazing is that the PIL can save to different filetypes merely by specifying the appropriate suffix for the filename. Below is my implementation of `PILDrawer`:

Code Listing 6.1: Implementation of `PILDrawer`

```
class PILDrawer(Drawer):
    def draw(self,ca):
        """Draw the CA using the PIL"""
        l = len(ca.array); w = 2*l+1;
        m=0; n=0; boxwidth = 1000/w; boxheight = 500/l;
        if boxwidth < 3:
            boxwidth = 3
        if boxheight < 4:
            boxheight = 4
        self.image = Image.new("L", (w*boxwidth,l*boxheight))
        draw_square = ImageDraw.Draw(self.image).rectangle
        for j in ca.array:
            m=0
            for i in j:
                if i == 0:
                    draw_square(((boxwidth*m,boxheight*n),\
                                 (boxwidth*m+boxwidth,\
                                  boxheight*n+boxheight)),\
                                  fill='white')
                m = m+1
            n= n+1

    def show(self):
                """display the representation on the screen"""
        self.image.show()

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

6.2 Conway's GoL

Conway's GoL exists on a torus, a two dimensional grid which is wrapped in both directions. Each cell has two states, alive or dead, and eighth neighbours in the cardinal and intercardinal directions. This is occasionally called a Moore neighbourhood of range 1.

The rules of the GoL are totalistic, and dependant on the state of a cell and its number of live neighbours. Here are the rules, with a brief logical description for them.

- If a cell is alive, it will die of loneliness if it has fewer than 2 neighbours. Conversely, it will die from overcrowding if it has more than 3 neighbours. If it has 2 or 3 neighbours, it remains unchanged.

- If a cell is dead, it will generally lie dead. However, if it has exactly 3 neighbours, it will spring to life.

With this rules, the GoL turned out to be very popular as:

- A small perturbation to the initial conditions yielded surprisingly different behaviour. This hallmark of chaotic systems will be explored in later sections.
- There are a number of patterns that are stable, either by remaining static, oscillating or translating. These can be thought of as attractor points in a chaotic system.
- Most significantly, the GoL is Turing complete. This characteristic will be explored further in later sections.
- Conway's initial conjecture, that no pattern can grow infinitely, was a challenge taken up by many. This challenge was met by a team from MIT led by Bill Gosper with his "Gosper gun".
- It is now possible to create animations for the GoL using computers, which is much more engaging than Conway's original implementation. Animations help reinforce patterns that change in time, allowing greater exploration of them.

In order to implement a GoL CA, we can essentially use the same datastructure of a one dimensional CA. The main difference is that the we take the vertical axis to mean another spatial dimension instead of time. To create a simple method of scrolling through all the cells in the CA, I considered all the peripheral cells to be ghost cells. Therefore a 20 by 20 array would represent an 18 by 18 CA.

In order to store all this information, I created a new array (labeled `newarray`) to store the value of the CA at a timestep. After creating the ghost cells and populating them with the appropriate values, loop through the CA, and find out the total number of neighbours for each cell. Knowing this, and the state of the cell, we can use the rules of the GoL to set if the cells are dead or alive. This algorithm can be represented by Figure 6.1

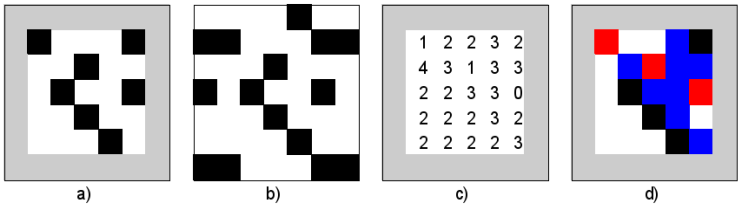


Figure 6.1: Algorithm for implementing the Game of Life. a)Have an array with a layer of ghost cells around it, with the array having the size of the desired CA. b)Populate the appropriate values into the ghost cells. c)Count the number of neighbours that each cell in the CA has. d)Using the rules of the GoL, decide if a cell lives or dies. I labeled the cells that remained alive to be black, the ones which died in red, and the ones which were resurrected in blue.

Using the `show` method of whichever subclass of `Drawer` we use, we can display the CA as it evolves in time. Now we are all set to explore patterns in the GoL.

6.3 Patterns

Starting from some random initial state, if we leave the game to run, it will generally settle down and display a few distinct patterns. A lot of time and effort has been put into discovering this patterns and characterising them. This source[1] has a huge lexicon cataloging the different patterns that people have

[1] `http://www.argentum.freeserve.co.uk/lex.htm`

found over the years.

As mentioned previously, most patterns either are still, oscillate or translate. However, there are some simple patterns aptly called "Methuselahs[2]", which have very long lifespans. Some of them, like the r-pentomino, can result in 25 different patterns in its lifespan of 1103 steps.

It is pretty tedious to manually enter specific patterns in order to test their evolution. It would be best to create methods that would create specific patterns given a set of coordinates. This is easy enough to implement.

To see how a random CA would evolve in time, I created a one, and let it run for 2000 timesteps. The results, shown in Figure 6.3, is that nearly all of the patterns created were either oscillatory or static. It leads to the question of if more complex patterns are wanted, whether more specific initial conditions are needed. I would say that this would be the case, as this would be consistent with other chaotic phenomena. Static and oscillatory patterns can be strong attractors that are stable, whereas translating and evolving patterns are unstable. However, it is these unstable patterns that are the most interesting, as they lead to more new patterns, "making new life".

Since Bill Gosper first disproved Conway's conjecture, we know that it is possible for patterns to exist that never stabilize. Conway did not design his game in a way that would make his conjecture obviously true or false. By design, patterns are not easy to find. Different rules for a 2-D CA would yield either trivial or uninteresting results. Conway thus avoided Wolfram's Class 1 and Class 2 behaviour (and probably Class 3), by setting the rules of the GoL the way he did. By setting the rules to cause Class 4 type behaviour, Conway created much more interesting CAs.

6.4 Realism and Instrumentalism

People have noticed many patterns in the GoL, giving them names and characterizing them. It seems obvious then that they are real. What do we mean by real? By real we mean they exist as persistent patterns that we can observe, predict and manipulate. If this explanation seems unsatisfying, this may be because of differing philosophical stances that you can take to this issue.

You could be a realist. A realist who believes that entities in the world exist independent of human perception and conception. In our context, it would mean that the patterns in the GoL exist, and are not constructs of the human mind.

Conversely, you could be an instrumentalist. An instrumentalist believes that we can't really say that a theory is true or false, because we can't know whether a theory corresponds to reality. We use instruments like our senses in order to determine whether something is real or otherwise, thus there is nothing to say that our senses give a correct representation of reality. An instrumentalist would be more concerned if a theory has any purpose. If a theory is fit for its purpose, then it is useful. In the context of the GoL, if patterns can be used to explain propagation of signals, they exist.

Note that both realism and instrumentalism exist on a gradient, with strong and weak versions of both. A strong statement of realism is that:

> A theory is true if it describes reality correctly, and false otherwise. The entities postulated by true theories are real; others are not.

Similarly, a strong instrumentalist statement would be:

[2]Named after the oldest person whose age is mentioned in the Hebrew Bible

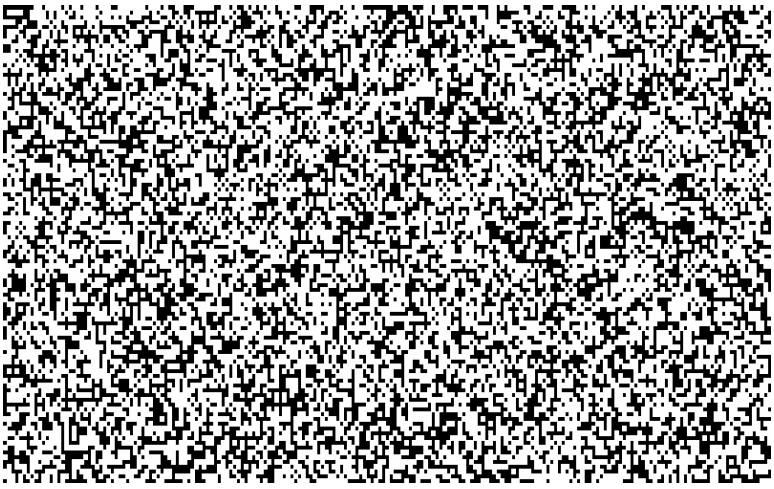


Figure 6.2: A random starting condition for the CA

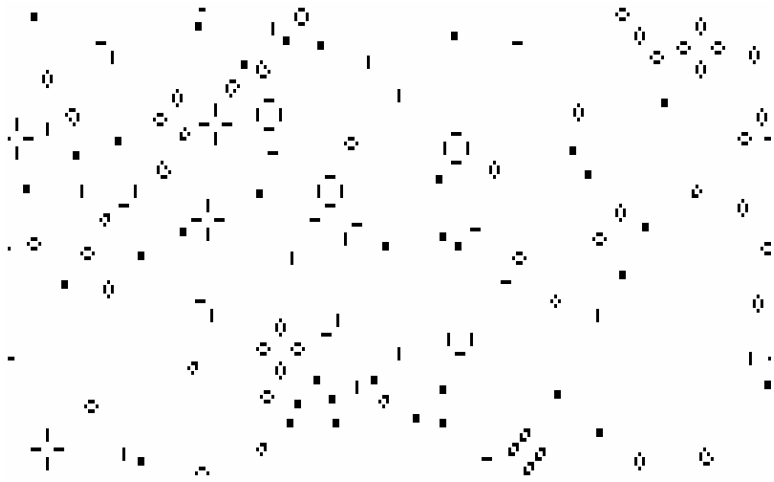


Figure 6.3: The CA from Figure 6.2 after 2000 timesteps. All the patterns in the CA are either static or oscillatory. This would suggest that patterns which have more interesting behaviour need very specific initial conditions.

> It is meaningless to declare a theory true or false. Theories are an end to a mean and as long as they serve a purpose they are useful.

Both statements are not too useful and usually weaker statements are adopted for both. As the next section will show, there is some purpose to GoL CAs. As long as there is some use to the patterns, the instrumentalists should be happy.

I slant towards the instrumentalist viewpoint for the patterns in CA, as I have a very liberal interpretation of usefulness. From an academic viewpoint, the existence of different patterns is interesting enough to warrant their investigation. Furthermore, as GoL CA are Turing complete, it is useful, as they could hypothetically be used as a computing architecture. For instance, Figure 6.4 shows a complete Turing Machine implemented using the GoL[3]. It consists of adders, comparators, stacks and all the other required components. I find this to be one of the strongest arguments for the existence of patterns in CA. Even if we do not

[3] http://rendell-attic.org/gol/tm.htm

perceive the patterns, or understand them, we can still appreciate the fact they are a well enough studied phenomena to be considered real and true.

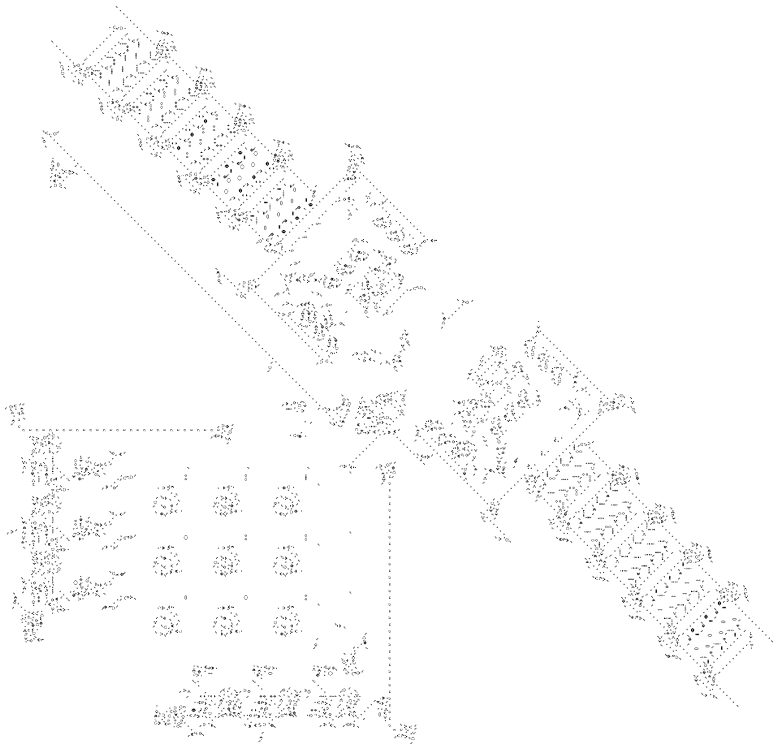


Figure 6.4: A Turing machine implemented using Gol CA. Although it is not apparent from the picture, each part of CA consists of a subsystem which allows the greater machine to function. For a detailed explanation of how everything works, visit Paul Rendall's website.

6.5 Turmites

A turmite is a generalised Turing machine in two dimensions, named in homage to Turing. A famous example of a Turmite is Langton's Ant, discovered by Chris Langton. The ant is the read write head of the Turing machine, with four states, canonically labelled as facing the cardinal directions. The cells in the CA have two states, black and white. The ant's behaviour is governed by two simple rules. If it is on a black cell, it turns right, makes the cell white, and moves forward one cell. When on a white cell, it turns left, makes the cell black, and then moves forward.

When a turmite is set loose, it is observed that it seems to have a random behaviour for a long time, then settles on a cycle with a period of 104 steps. After each cycle, the ant would have translated diagonally, creating a trail which is called a highway. Multiple turmites will interact in complex ways, making for interesting viewing.

In 2000, Gajardo proved that Langton's Ant could be used to calculate any Boolean circuit using the trajectory of an Ant. It is fascinating that this is possible, as the rules for Langton's Ant seem almost trivial. Figure 6.5 shows two of the gates. Its is clearer now, that it is nontrivial to create patterns in CA that mean anything. It brings me back to the idea that a million monkeys banging on typewriters would result in the next great American novel. The Internet has proved that this is not the case, as has the GoL which results in "boring" patterns. If we change our frame to life then, we see how the intricacies of life, how cells make up

tissues, tissues make up organs and organs make up organisms (a very simplistic viewpoint), we see how organisation of small elements, acting over a short range, can create effects that propagate far beyond their size. Maybe then, the name ”Game of Life” becomes more appropriate, as it is a random roll of the dice for the initial conditions which results in emergence (or absense) of complex phenomena.

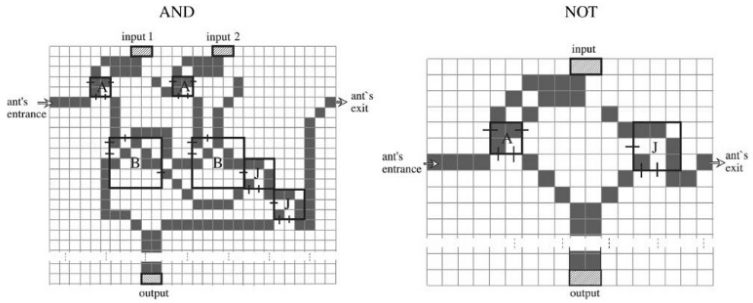


Figure 6.5: An example of two different gates that can be created by a turmite from Gajardo. In this case, the signal is the ”head of the Turing Machine”. A and B are types of crossings, while J is a junction.

6.6 Modeling using CAs

So far, the CAs we have examined have not reflected real life, and were treated as constructs with their own meaning. In later chapters, we will explore CAs that model physical phenomena like percolation, forest fires and avalanches. Notice how all involve things where things propagate from sources and small disturbances can often result in dramatic effects.

Chapter 7

Self-organized criticality

7.1 Tkinter interface

Before we get into the chapter proper, I would like to describe the Tkinter platform I used to visually display the simulations. This interface was created later than expected, I did not have an interest in creating an animated interface before, as it was not as important to observe transient effects. Below is the generic body for my Tkinter simulation.

Code Listing 7.1: Discrete Fourier Transform

```
class Generic():
    def __init__(self,size):
        self.master = Tkinter.Tk()
        self.board = Tkinter.Canvas(self.master, width=size, height=size, background='white')
        self.board.pack()

    def updatecanvas(self):
        self.board.create_rectangle(0,0,1100,40,fill='white',outline='')
        self.board.create_line(0, 16, 1100, 16,fill='black')

    def step(self):
        self.updatecanvas()
        self.master.update_idletasks()

    def loop(self,steps=100):
        for i in xrange(steps):
            print 'Step %d' %i
            self.step()
```

The `init` function initializes a canvas where everything is drawn on. I use a canvas object, which allows me to draw shapes onto it easily, which is perfect for the sort of animation I hope to carry out. `Updatecanvas` is what is used to modify the canvas. In an actual implementation, there would be a more sophiticated pattern in its body, drawing things based on data. `Step` is a large function that is meant to encapsulate all actions that need to be done in a single timestep. Finally, `loop` is what decides how many steps to execute, making it easy to run the simulation for known periods.

I plan to use this basic framework for all my subsequent animations. It is an easy to implement option for simulations which is non computational intensive if used correctly. To learn more about Tkinter, go to `http://www.pythonware.com/library/tkinter/introduction/`.

7.2 Bak, Tang and Wiesenfeld

This trio came up with the idea of self organized criticality, and published their findings in 1987. A critical system is one that is in transition between states, like water at its boiling point. The self organized part claims that critical system emerge naturally, and do not have to be artificially created.

Common behaviors of critical systems are:

- Long tailed distributions of some physical quantities.
- Fractal geometries.
- Variations in time that exhibit pink noise.

What we perceive as noise is actually a collection of many different frequencies which do not add up to sound like anything pleasant. White noise is when all the frequencies have equal power. Pink noise is when the lower frequencies have more power than higher frequency noise. If we use the visible light spectrum, if all frequencies are of equal power, it is white, while if the lower ones dominate, it will appear pink.

BT& W proposed a model of a sandpile where the grains of sand self organize into a critical state. The model is not particularly realistic, but is an interesting model nonetheless. Allen Downey's book gives a good explanation of how to set up the model.

There is some ambiguity about how to initially set up the system. BT& W's paper just states that every cell should be above some critical value, but this is not enough when someone has to implement the system. Suppose my critical value is 4. I found experimentally that if the initial conditions was between 5 and 10 or 5 and 20 made no difference to the way the system behaved. My explorations were not conclusive, as I found it difficult to determine what data to use to try and compare the systems. One explanation was that the self organized criticality behaviour was so robust that it did not matter what the exact initial conditions were. After buying into that explanation, I chose an initial condition of 5 to 10, in order to get the system to stabilize faster.

An important consideration in this model is the sophistication needed to track the changing cells. There is two instances when the cell tracking differs: when the system is stabilizing after the initial conditions, and when it is stabilizing after being perturbed in a single cell. In the first case, we can track it using a simple variable. At the beginning of the timestep, we reset the variable. If at any point of the timestep a cell changes, set the variable. If at the end of the timestep the variable is set, that means the system is still changing. The second case requires more sophistication. In this case, we want to know the number of cells changing, and thier identities. It is possible to concieve scenarios where a cell changes, only to return to its initial state. This makes it important to track transient behaviour of the cells.

An easy way it to use a python data type called sets. They are like lists, but can only have one object of a certain value in it at a time. Therefore, everytime a cell changes, append the x and y indices of the cell into the set. If it already exists, nothing happens. We still use the variable mentioned previously in order to check if the system is changing as a whole. Once the system stabilizes, the change variable would not be set, thus signalling that the simulation can end.

The result from my simulation is shown below. It was ambigious which distribution to plot, therefore I made the logical choice to choose to plot the ccdf of the function. As shown in Figure 7.1, the figure does not really resemble what is shown in the paper. However, it is known the graphs with a ccdf like in Figure 7.1 have a straight pdf on a log log scale. Therefore, if the graph in the paper is of a pdf, then this result has been corroborated.

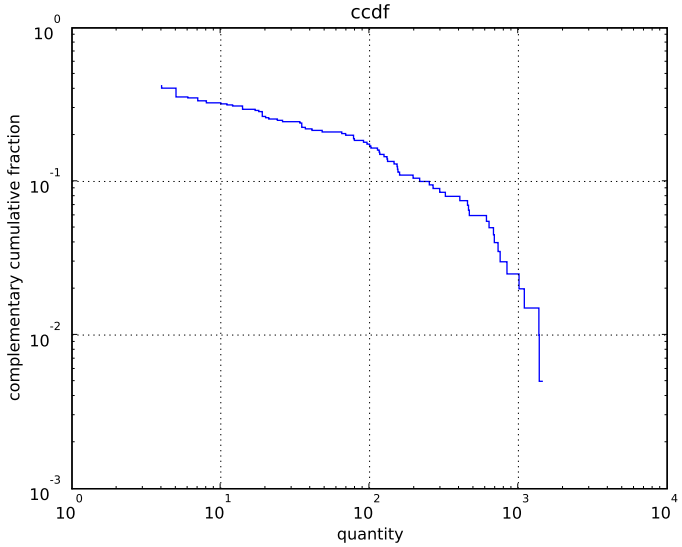


Figure 7.1: The frequency of the number of cells that changed after being perturbed. Since the shape is consistent for the ccdf of a system with pink noise, we have corroborated the results from the paper.

7.3 Spectral Density

We need a tool in order to analyze the frquency of phenomena in the sandpile and other 1/f phenomenon. The tool of choice is the Fourier Transform. The Fourier transform takes a input that is varying in time, and returns the magnitude and phase of the periodic functions that it is composed of. The canonical example is of the square wave. It can be proven to be the sum of a few sine waves of different frequencies. Figure 98 shows the square wave that results from summing up 4 different sine waves. With more sine waves, the irregularities in the resulting square wave would be ironed out.

From the other direction, decomposing an input signal involves taking its Fourier transform

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

It is useful to use the relationship $\omega = 2\pi f$ as it decreases the number of times 2π appears.

Since we are dealing with discrete datasets, we have to discretize this function into:

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

Using the above expression, we can tell the magnitude and phase of various frequencies in the signal. If we want to measure the power of the various frequencies, we need to take the square of the corresponding frequecies' magnitude.In effect, we are calculating the power signal density, $P(f)$, which is a function that maps from a frequency , f, to the power associated with that frequnecy in the signal.

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

In our application, we do not really care to distinguish between f and $-f$. For real signals, the PSD is symmetric about 0Hz, and thus taking the values from 0 to N/2 is sufficient to completely characterize the

PSD. However, to obtain the average power over the entire interval, it is necessary to introduce the concept of the one-sided PSD:

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

Creating a discrete fourier transform in Python is aided by the `cmath` module which provides support for complex numbers. Below is my implementation of a DFT. It is simply applying the DFT algorithm mentioned above.

Code Listing 7.2: Discrete Fourier Transform

```
def dft(x):
    from cmath import pi, exp
        sign =-1
    N, W = len(x), []
    for i in range(N): # exp(-j...) is default
        W.append(exp(sign * 2j * pi * i / N))
    X = []
    for n in range(N):
        sum = 0
        for k in range(N):
            sum = sum + W[n * k \mod N] * x[k]
        X.append(sum)
    return X
```

An easy way to verify if a fourier transform is functioning is to tranform a known signal, and verify if the expected results occur.

Although DFTs are pretty simple conceptually, they are not very fast. My implementation has a quadratic order of growth, which is not ideal. Others have noticed this issue, which led to the discovery of the Fast Fourier Transform(FFT). FFT is an efficient algorithm for computing the DFT, commonly having a $O(n \log n)$.

The first step is to substitute $W = e^{2\pi i n/N}$ into Equation (7.1)

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

The second step is the Danielson-Lanczos Lemmas which states

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

where H_k^e is the DFT of the even indexed terms, and H_k^o is the DFT of the odd indexed terms. This suggests a recursive algorithm could be used to quickly evaluate the DFT sequence. First we recursively split h into h^e and h^o. Then we compute H^e and H^o by making recursive calls. Then we use the lemma to combine the terms to form H

There is an interesting implementation that uses the reversal of bits to make the splitting of the series faster. If we reverse the bits in 8 (0b100), we get 1 (0b001). If we had a signal which consisted of 16 segments, and we split it by recursively dividing it into odd and even, we would get a sequence of:

$$0814122106141951331 1715$$

If we had performed bit reversal on the index values, we would have acheived the same result. Note how 8 became 1, therefore it is the second term in the new series. Below is a method that converts a list of values into their bit-reversed equivalents.

Code Listing 7.3: Discrete Fourier Transform

```
def bitrev(x):
    N, x = len(x), x[:]
    if N != nextpow2(N): raise ValueError, 'N is not power of 2'
    for i in range(N):
        k, b, a = 0, N>>1, 1
        while b >= a:
            if b & i: k = k | a
            if a & i: k = k | b
            b, a = b>>1, a<<1
        if i < k: # important not to swap back
            x[i], x[k] = x[k], x[i]
    return x
```

It uses a helper function `nextpow2` which returns the next power of 2 greater than the number. We now can implement the FFT algorithm mentioned above. The decomposition of the signal takes $log(n)$ steps and the recombination for a complete FFT is linear for the number of discrete chunks, giving the algorithm an order of $n \log n$

Code Listing 7.4: Discrete Fourier Transform

```
def fft(x, sign=-1):
    from cmath import pi, exp
    N, W = len(x), []
    for i in range(N): # exp(-j...) is default
        W.append(exp(sign * 2j * pi * i / N))
    x = bitrev(x)
    m = 2
    while m <= N:
        for s in range(0, N, m):
            for i in range(m/2):
                n = i * N / m
                a, b = s + i, s + i + m/2
                x[a], x[b] = x[a] + W[n % N] * x[b], x[a] - W[n % N] * x[b]
        m = m * 2
    return x
```

Using the FFT function we have created, we can create a function to calculate the PSD of a signal. `Pylab` has a function to compute the PSD of a signal. Therefore, I used it whenever I needed to calculate the PSD of a signal. For future reference, the one sided power spectral density is defined as

$$P_n = |H_n|^2 + |H_{-n}|^2$$

7.4 Pink Noise

If a signal consists of pink noise, its PSD is defined as

$$P_n \sim 1/f_n = \frac{Nd}{n}$$

Taking the log of both sides, we get:

$$\log P_n \sim \log N - \log n$$

Therefore on a log-log scale, the PSD of pink noise is a straight line with a slope of -1.

7.5 Forest Fire models

Another model that demonstrates 1/f noise is the forest fire model by Drossel and Schwabl. It is a CA with three states: empty, on fire, or occupied by a tree.

The rules are simple.

1. An empty cell can grow a tree with probability p_g
2. A cell with a tree will catch fire if any of its neighbours is on fire.
3. A cell will spontaneously burst into flames with probability p_f
4. A burning cell becomes empty in the next timestep.

Drossel and Schwabl mentioned in their paper that if $p_f/p_g = 0.1$ the system is in a critical state. It is easy enough to modify our sandpile implementation in order to become the forest fire model. The exact details are left to the reader, as it is pretty trivial.

The difficult part was trying to to get a cell to catch fire. A cell has to be aware of the state of its neighbouring cells, and change to the on fire state if any of its neighbours are on fire. The easiest way was to create a method with created a list of tuples of coordinates if the coordinates of the center cell was given. By looping through the neighbouring cells, it is possible to check if any of them are on fire. Other than this, all other rules of the model are pretty easy to implement.

Once the model has been implemented, it is fun to play around with the values of p_g and p_f. If the rate of growth is too high, the forest gets too dense, and fires spread easily across the map. On the other hand, if the rate of fire is too great, the forests are constantly bursting into flames, preventing the creation of any clusters of trees.

I did not manage to find any data in the model that produced 1/f noise. It was not too clear what data to use. Due to time constraints, I left it at that.

7.6 Reductionism and Holism

Reductionism and holism is given a very good treatment in Downey's book. I have no comment.

It was interesting to do the exercise where data was taken from different distributions, where the data eventually collapsed into a normal distribution. I thought that was pretty cool.

7.7 Self organized criticality

The premise of the 'Great Man' theory does not sit well with me, as it gives too much credit to the efforts of few, over collective forces that I feel are more important. As a society, we are prone to exalting the achievements of a few, maybe as a form of motivation for us. I prefer a more nuanced approach where

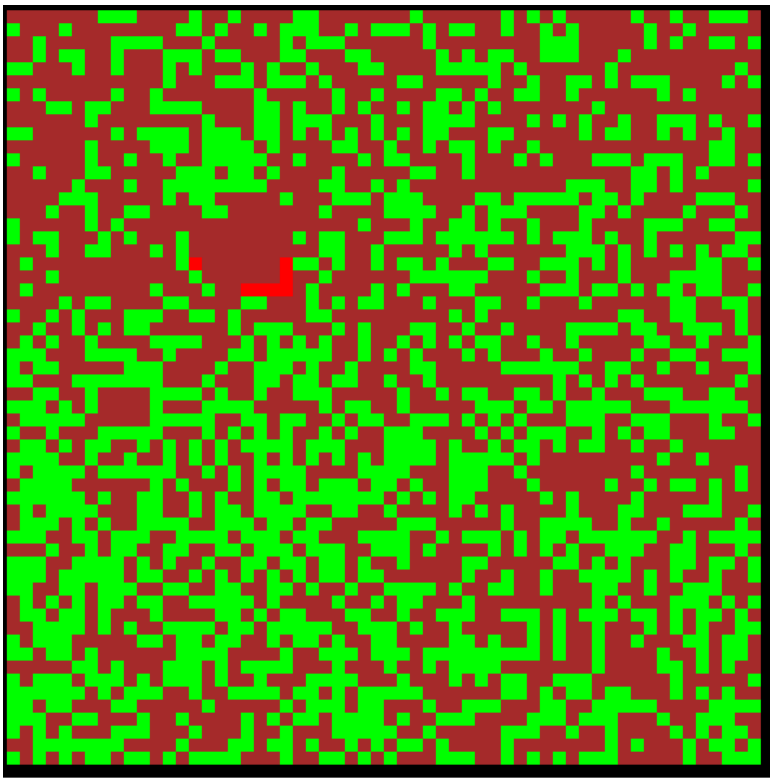


Figure 7.2: The red cells indicate trees on fire, the green is trees and brown is the ground.

collective forces, like movements in society like the French Revolution, result in change, and those great people are merely the leaders that emerge from those movements. It is the movement that empowers the great person, and without it they will fail. This is not a very generaliable statement, and more work must be put to disprove this theory.

Chapter 8

Agent-Based models

Agent-based models are simulations based on the global consequences of local interactions of members of a population. These individuals might represent plants and animals in ecosystems, vehicles in traffic, people in societies, or autonomous characters in animation and games. These models typically consist of an environment or framework in which the interactions occur and some number of individuals defined in terms of their behaviors and characteristic parameters. In an individual-based model, the characteristics of each individual are tracked through time. This stands in contrast to modeling techniques where the characteristics of the population are averaged together and the model attempts to simulate changes in these averaged characteristics for the whole population.

8.1 Characteristics

There are a few key characteristics to agent based models:

- Agents that model intelligent behaviour, usually with a simple set of rules
- The agents are usually situated in space and interact with each other locally
- They usually have imperfect, local information.
- Often there is variability between agents
- Often there are random elements, either among the agents or in the world.

These are not hard and fast rules, but are more of a guide to classify models. Interestingly, agent-based models are able to model systems that are not in equilibrium although they can also be used to study those at equilibrium.

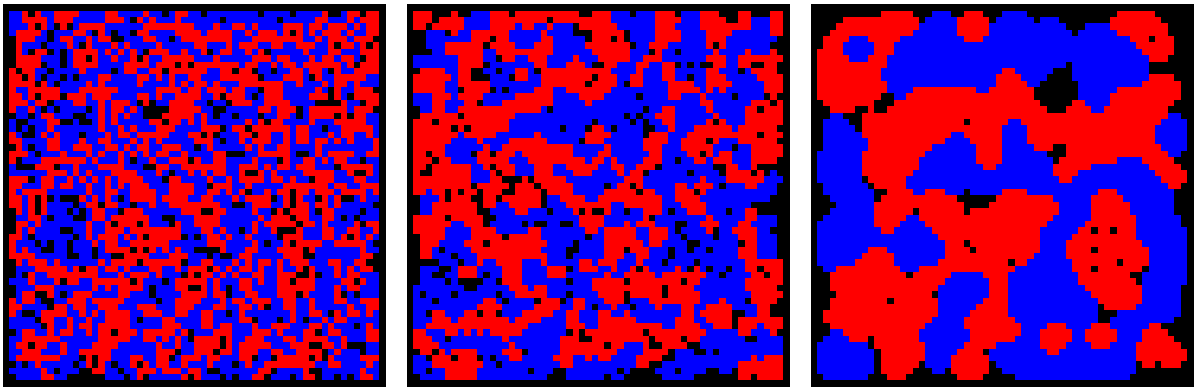
8.2 Schelling's Segregation

Racism is a complex phenomena which may prove daunting to simulate. However, Thomas Schelling's 1971 paper, "Dynamic Models of Segregation" showed that a supposed manifestation of racism, segregation, could be easily simulated. He discerns in his paper how segregation can be organised, result from

economic processes or result from discriminatory individual behaviour. In choosing to focus on the last reason, he allowed it to be modeled using an agent based model.

First, create a two dimensional grid of a neighbourhood where the cells represents available houses. Next, populate the houses randomly with two kinds of occupants, nominally the Reds and Blues. The rules the occupants will follow is simple. Each occupant will determine the occupants of the 8 surrounding houses. If there are less than two neighbours of the same kinds, they will be unhapppy and move randomly to an empty house. This process will continue until everyone does not want to move anymore as they are all happy.

You would expect that with a threshold of two neighbours of the same kind that segregation would not be significant. However, the results are striking. Figure 8.1 shows the result of segregation with different levels of 'tolerance'. With a tolerance of 2, the neighbourhood looks diverse if the percent of empty houses is 10%. However, if you observe the patterns closer it becomes clearer that there is a significant amount of segregation occurring. A high density of occupants may obfuscate the segregation which exists in a neighborhood.



(a) Minimum of 2 (b) Minimum of 3 (c) Minimum of 4

Figure 8.1: The Schelling model with two different kinds of occupants and 10% empty houses. The different subfigures have different minimum limits of similarity before an agent is unhappy and chooses to move.

However, if we make the occupants more racist, the effects of segregation are a lot more apparent. Figure 8.1 shows the neighborhood with an intolerance(Minimum number of same types. This is a loose term) of 3 and 4. There is a much clearer delineation between the part of the neighborhood for Blues and Reds. Increasing the intolerance will result in unstable systems where nobody is happy. In a cluster of a Blue or Red houses, it would be improbable that the cells on the edge would have say 6 neighbours of the same kind. (If they did, they probably would not be on the edge of the cluster).

Another interesting variation is to set an upper limit to neighbours of the same kind. This would mean cells would not want to be in a society that is too homogeneous, and would want diversity. Comparing the neighborhood with(Figure 8.2) and without this factor, it is apparent that this factor only helps marginally in creating more diverse neighborhoods.

Another perspective to segregation is Bill Bishop's, where people are moving randomly, but they choose explicitly where they move to. He theorizes[1] that because people are more likely to choose to move into areas where there are more people like them, they end up causing segregation. I did not implement this, but it is still interesting to discuss it. You could guess that this algorithm would create a much more dynamic system, as people will always move. Maybe at a higher level, where we do not distinguish particular agents, the system would be in equilibrium, I would wager that the system would not stabilizes quickly.

[1] http://www.thebigsort.com

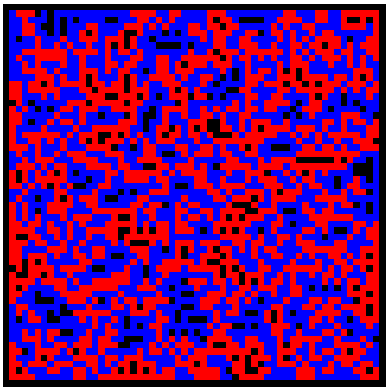


Figure 8.2: Segregation model with a minimum of 2 and maximum of 6 for happiness. Notice how it is not too different from the case when there is no maximum.

Instead, I would wager that the system would continue to evolve until both types of inhabitants would be perfectly separated. There seems to be nothing that pulls the system away from becoming segregated, no contentment nor phobia of too much similarity. I leave it as an exercise for the reader to prove or disprove my assertion.

8.3 Traffic jams

The dynamics of highway traffic is ideal for simulating using an agent-based model. The agents, drivers on a highway, follow simple distance maintaining rules. If they are too close to the next vehicle, slow down. If the distance exceeds some limit, speed up. Their speed has a speed limit, and they cannot have negative speed(go backwards).

There are two distinct results of these rules. First, is that traffic jams can occur spontaneously, without any apparent cause. Second, is that the traffic jam will appear to propagate backwards. One of the main reasons for this is because the agents are able to brake much faster than they can accelerate. Therefore, vehicles leave the traffic jam slower than they enter it.

My implementation of the highway was the simplest case of a highway that is a loop. More complicated implementations would have a rate of vehicles entering the highway, and even multiple lanes to allow overtaking[2]. All the vehicles in my highway are identical, following the same set of rules. Figure 8.3 is a screenshot of the highway.

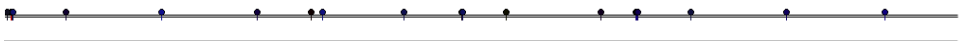


Figure 8.3: A screen shot of my highway implementation in Tkinter. The vertical lines show the exact position of the vehicles, as they are 1 pixel wide, and the circles aid in making the different vehicles more obvious. By default, the vehicles all have different dark colors, which enables me to selectively color vehicles brightly in order to visually track them. Although the highway is displayed as a line, it represents a loop as both ends are connected.

Since overtaking is not allowed in my highway, the vehicles are always in the same order. Therefore, each vehicle can have an attribute with the identity of the next vehicle on the highway. This makes it simple to find out the other attributes for each vehicle. Each vehicle has a position, speed and distance to next vehicle(following gap). To get the following gap, we just need to subtract the position of the two vehicles.

[2] http://www.traffic-simulation.de/

It is important to decrement this gap value by one in order to prevent vehicles from choosing to be in the same position in space. The logic system of the vehicles would use this gap value to decide whether to speed up or slow down. Below is the speed choosing algorithm.

Code Listing 8.1: Algorithm used by the vehicle to choose its speed

```
    def set_speed(self):
        for v in self.vehicles:
            if v.distance > self.limit:
                v.speed += 1
            elif v.distance < self.limit:
                v.speed -= self.brakingspeed
            if v.speed > self.speedlimit:
                v.speed = self.speedlimit
            elif v.speed < 0:
                v.speed = 0
```

However, the vehicle does not necessarily move at its speed. Another function determines actually how the vehicle moves.

Code Listing 8.2: Algorithm to chose how much to actually move

```
    def move(self):
        for v in self.vehicles:
            v.speed = min(v.speed, v.distance)
            v.position = v.position + v.speed
            if v.position >self.size:
                v.position = v.position - self.size
```

If its speed is greater than the distance between the vehicles, its speed would be the gap value instead. This is how the vehicles are able to decelerate very quickly. If the gap value was not less than the actual distance by one, the vehicles would end up occupying the same position. In real life, that means they have crashed. Unfortunately, I did not implement explosions when vehicles collided, which would have been interesting.

Varying the different parameters yields some interesting results. My highway consisted of a loop with 1000 discrete positions for the vehicles to travel in. My metric to measure the throughput of the highway is the average speed of all the vehicles on the highway, normalised for the speed limit for the vehicles. The first parameter I varied was the following distance. As Figure 8.4 shows, the average speed decreases as the following distance increases. This makes sense, as the total amount of space is finite. If the following distance is too large, the vehicles will all be forced to slow down, and even stop, just to maintain the distance.

Now, if we vary the number of vehicles, with a following distance of 70, we find, that the speed decreases as the number of vehicles increases(Figure 8.5). We would expect this, as the more vehicles there are, the more the finite space has to be distributed among them to maintain their distance. After a point, the speed drops precipitously as the vehicles jostle with each other for space.

The final parameter I varied was the top speed of the vehicles. Interestingly enough, increasing the top speed creates a sudden peak that gradually decreases(Figure 8.6). This peak shifts when the other parameters like following distance and number of vehicles are varied.

I felt it would be interesting to view the kinematics of a vehicle on this highway. Setting the highway in a condition which would yield moderate traffic jams, I determined the speed of the vehicle in time(Figure 8.7). It followed a trend that I expected. The vehicle would get stuck in a traffic jam, wait for a while, then accelerate quickly away from it, only to rapidly hit the brakes and get caught in the back end of the same

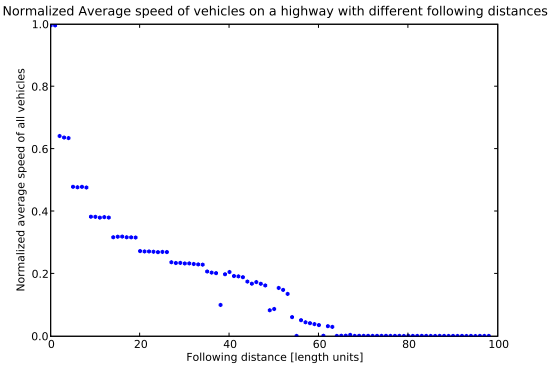


Figure 8.4: As the desired following distance of the vehicles increase, their average speed decreases rapidly. This is mainly because of the limited space of the highway. 50 vehicles with a top speed of 10

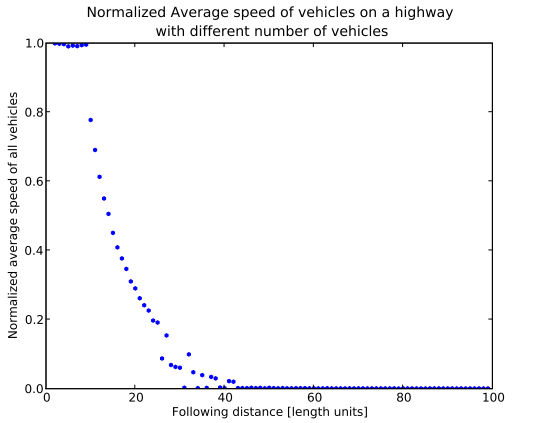


Figure 8.5: Increasing the number of vehicles has the expected decrease in the average vehicle speed. All vehicles used a following distance of 70 and had a top speed of 10

jam. In real life, this sort of driving would probably wear out your brakes pretty fast and result in poor mileage. In the Python universe, these concerns are moot.

A possible extension is to have the vehicles come up with a following distance based on their top speed and braking rate. When you get a new car, you would use these values to determine your following distance. The goal (pretty reckless one) is to stop just behind the next vehicle. Therefore, if you are travelling at a speed of 10 units, and could decelerate 1 unit per time step, you would need a following distance of $10 + 9 + 8 + ... + 1 = 55$. If all vehicles used this algorithm to determine their following distance, it will partially solve the problem of traffic jams. This is assuming that there is enough space such that traffic jams do not have to exist.

8.4 Boids

It has always been fascinating how birds and fish are able to move together without a central conductor. They are able to execute pretty elaborate patterns by following simple rules. Each agent has three behaviours:

1. Collision avoidance: Avoid obstacles

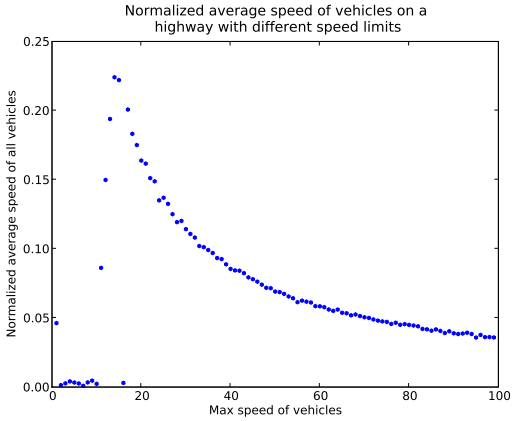


Figure 8.6: Increasing the top speed of the vehicles has the expected decrease as the speeds get too high, as traffic becomes very start stop. However, the increase at the beginning is unexpected, and is due to the vehicles all being very close to each other, and no vehicle ever speeding up that far away from the pack. This claim was made after observing the vehicles all coalesce into packs that crawl along if their speed is too low. 20 vehicles with a following distance of 100 were studied

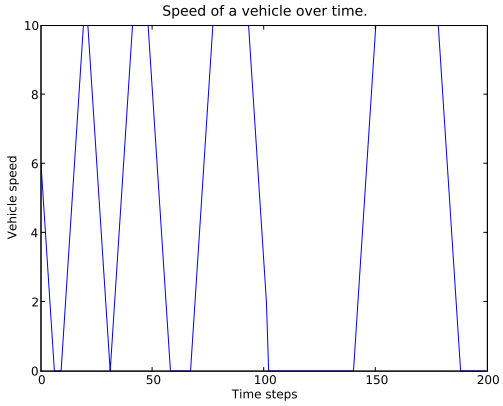


Figure 8.7: The crazy speed versus time movement a vehicle. There were 20 vehicles on the highway, with a following distance of 50 and a top speed of 10. Notice the rapid increase and decrease in the vehicle's speed, as it leaves a traffic jam, only to enter into another one.

2. Flock centering: Move to the center of the flock
3. Velocity matching: Align velocity with neighbouring boids
4. Line of sight: If there is a boid in front, move laterally away

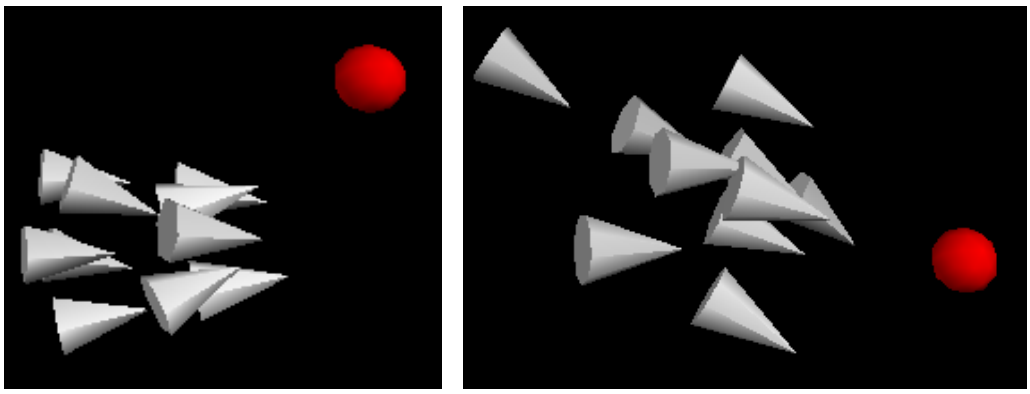
Boids are not gifted with omniscient information. Rather, they only have a limited field of vision and range from which to make decisions with. However, this is actually enough for the flocking behaviour of the boids to emerge.

An implementation of boids that we could build on is Allen Downey's[3], which represents boids as cones in three dimensional space using Visual Python. This implementation uses a sphere as a target, and the sphere

[3] `http://www.greenteapress.com/compmod/Boids.py`

can be made to follow the mouse cursor. In doing so, Downey has created an implementation which has a simple interface while maintaining good visual feedback. A fifth behaviour is needed for the boids to be attracted to the sphere, which Downey implemented using vector subtraction.

One behaviour that it leaves out is the line of sight behaviour. Using knowledge of vectors, this is easy to implement. First, find the average heading of the boids within the (very narrow) field of vision of a boid. We would want a narrow field for this, as this behaviour is meant for moving away from boids directly in a boid's way. Take the cross product of the average heading and a random vector to get a random vector perpendicular to average heading of the boids in front of it.



(a) With the line of sight behaviour (b) Without the line of sight behaviour

Figure 8.8: The boids with a behaviour, line of sight, turned off and on. It is still pretty difficult to perceive the difference that the behaviour causes, even with the total absense of a behaviour. I find that only interacting with the computer simulation allowed the behaviours of the boids to manifest clearly.

The effects of each behaviour is proportional to their weighting. Personally, I found it difficult to perceive the effects of varying the weight of an effect marginally. The most effective way to explore the space of the weights was to turn of an effect, or make it very large. Most of the results observed were pretty much as expected. For example, decreasing the weight of flock centering resulted in much larger flocks, as the boids had less need to clump together.

There is a lot of room for extensions to this boids simulation. One track could be to determine the set of weights which would create specific shapes and sizes for the flocks. Another is to introduce obstacles into the environment and try to get the flocks to effectively navigate through them without dispersing. Here is something to inspire readers to extend the model furthur: `BlenderBoids-http://www.youtube.com/watch?v=nSHycabSexo`

8.5 Ants

8.5.1 Introduction and setup

Go outside and try to find an ant colony. If you are lucky enough to find one, you will be in the presence of one of the most organized societies in nature. Ants exist in very structured communities that make it able to almost act as a single organism[4].

How do they achieve this? While ant societies do have queens, the queen is not the central conductor of ant society. Rather, ants uses pheromones to communicate and coordinate their activity. A common agent based model is that of an ant colony searching for food.

[4]Oster GF, Wilson EO (1978). Caste and ecology in the social insects, Princeton University Press, Princeton. p. 2122.

The ant agent will be able two different kinds of pheromones. One is a long lasting pheromone that increases in concentration nearer to the ant nest. This gives the ant an ability to home in on its nest. The other pheromone, which is transient, indicates the presence of food. By following this pheromone ants are able to find food.

There are two states for the ants: foraging and returning home. If the ants is carrying food, they will return food, otherwise they will be foraging for food. While they are returning to the nest with food, an ant will drop a quantity of the food pheromone with every step. This is how the trail of pheromones to the food source is created. The ant will 'sniff' for the nest pheromone and tend towards where the concentration is highest.

If the ant is foraging, it is trying to sniff for the food pheromone and tend towards the direction of the pheromone. If there is no food pheromone scent, the ant will move randomly. This is simple enough for an ant to use the pheromone trail to find a source of food. Following the pheromone trail will get the ants close enough to the food source, as the food pheromone trail will decay in time. Once close, the random movement of the ant will allow (logically and from later simulations) it to find the food, and then reestablish the food pheromone trail.

It follows then that if many ants reach a certain food source, the pheromone trail to it will gradually increase to the point that it increasing fast enough not to diminish with time. Therefore, a path will be established from the food source to the nest for the ants to follow. It will become possible to observe almost a highway of ants moving up and and down the path, busy squirreling away food into their nest.

Using Python, we can implement this model on a 2-D landscape. It is important to try and make as efficient as possible code in this simulation, as we will be dealing with a lot of dynamic agents. Slow simulations will obfuscate patterns that are emerging from the CA, preventing us from recognizing them.

The terrain would have three 2-D arrays to represent it; one to note the quantity of nest pheromone, another for the food pheromone and a third for the food. The indices of the elements in the array would correspond to spatial coordinates in the world, making it simple to determine the amount of pheromone or food at a point on the map. As said above, the nest pheromone array would contain

Next, we need to design an ant agent. The agent must be able to sniff the terrain and choose the appropriate direction to go towards. When it is following a pheromone trail, it will compare the quantity of pheromone ahead of it and to its left and right, then head towards the direction where the quantity is greatest. If no pheromones are found, the ant will randomly choose to turn left or right, or just go forward.

The sniffing process outlined above depends on the ants ability to know its heading, and the heading to its left and right. I used a tuple to store the headings in the format shown in Figure 8.9. An easy way to indicate the direction to the left or the right of a particular heading is using lists which are offsetted appropriately. Therefore, to find the direction to the left of (1,0), you would find the index of (1,0) in a headings list, then determine the value that corresponds to that index in the turnleft list. This lookup is efficient, and is very intuitive for the programmer. An issue with Tkinter is that is uses the top left corner as its origin, with its axis pointing right and down instead of the right and up which we are used to. Therefore, by abstracting this complication away from the programmer, it becomes easier to control the agents.

One large issue in this simulation is the number of variables that need to be tracked and then drawn onto the canvas. My implementation used a Tkinter canvas object to display the terrain and agents. It would be unreasonable to draw new rectangles (everything is represented by rectangles on a grid) at every timestep as you will soon end up with large numbers of layers on your canvas, which will drain your memory and be very slow.

A better way would be to store the ids of all the rectangles that make up the terrain. If you create it in a way such that the ids follow like the way a typewriter moves, it would be easy to back out the id of a rectangle. It would be the sum of the horizontal coordinate and the product of the vertical coordinate and width of

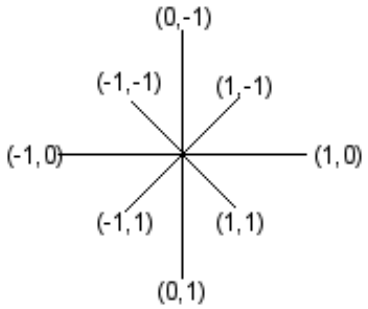


Figure 8.9: The direction system used by the ants. The reason why up is negative, is because of the Tkinter coordinate system which points right and down. This simple direction scheme allows easy turning algorithms to be devised using list lookups.

the terrain. Using the function `itemconfig` on the canvas, you can modify any object on the canvas by specifying its id. In my implementation, I just modify the color of points on the grid. I could have used a specific rectangle to represent an agent and kept on modifying its position. However, I decided to use the first way as it was the way I implemented it first. Figure 8.10 is a screenshot of the ant simulation.

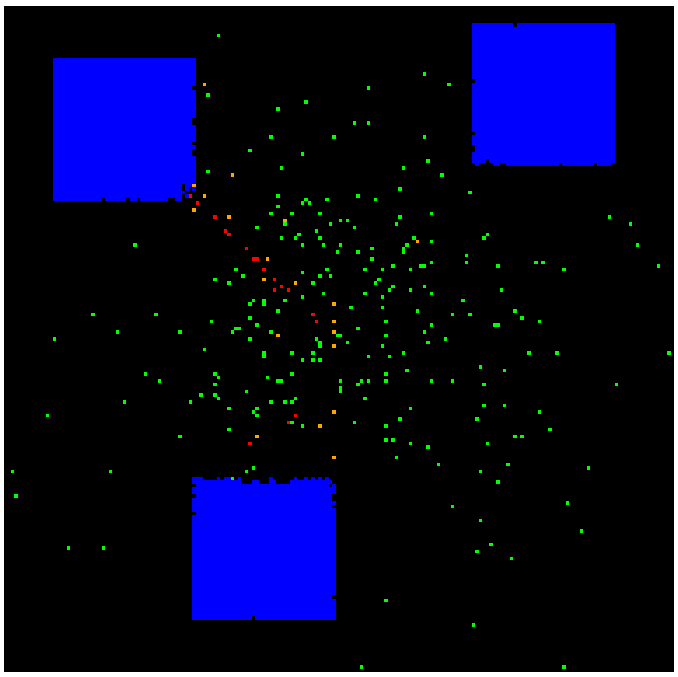


Figure 8.10: This is a screenshot of the ants in the terrain. The blue represents the pieces of food, but the nest is not shown. Ants can exist in four colors, indicating their behaviour. Green is a foraging ant, red is an ant following a trail, and orange is an ant returning to the nest with food. The trail of food pheromone is not shown as animating that would use up too much computing resources. Instead, we rely on the ant changing color to reveal traces of the pheromone. For scale, the ant can move a single step in every timestep, and the terrain is a 200 by 200 grid.

8.5.2 Analysis

Once the terrain and agents have been set up, you could begin to let the simulation run and observe the results. You should be able to observe the 'highway' of ants travelling from the nest to the food sources(Figure 8.10). If you do not observe this, you might have set some constants badly. The key constants are:

1. Amount of food pheromone to drop at each time step
2. Rate of decay of the food pheromone
3. Maximum distance of the food sources from the nest
4. When foraging, the probability of the ant changing direction
5. the number of ants

The five constants are all related. If the amount of food pheromone dropped is say 100, and the decay rate is 1, then after 100 timesteps, the trail would disappear. Assuming the maximum distance of the food source to the nest was 50. Then, things would work out as the ant would be able to retrace its trail back to the food source before the trail decayed. However, if the maximum distance was 100, then when the ant set out again to find the food source, it will only be able to go 50% the way back as the rest of the trail would have decayed. Then, the foraging probability constant takes over.

In my implementation, it appeared that the ants almost had a GPS like ability to home in on the nest and return to it. Therefore, the trails that they made were generally straight lines with maybe one bend. This implies that if an ant started following a rapidly decaying trail and mostly just kept on heading in the direction in which it last found pheromones, it should reach the food source. There is a fallacy here where a sharp bend could lead a few ants to wander off into the abyss. In conclusion, it might not be ideal to set the probability of the ant changing directions to be very low.

Finally, the number of ants can significantly change the dynamics of the system. A lot of the problems mentioned above can be mitigated with more ants. For instance, if two ants at about the same time go from a food source to the nest, they will create a stronger trail from the food source to the nest. (The ants would generally travel to the nest in almost the exact same path if they came from close food sources.) If the time difference was say 50 time steps (to match the constants we used above), the first ant would actually be able to smell the pheromone 75% the way back to the food source. This would significantly increase its chance of returning to the food source.

The effect of more ants scales amazingly. I set up the decay to be one unit of food pheromone at each timestep. Therefore if there was 10 units of pheromone on a patch of terrain, it would take 10 timesteps to decay, but if there was 100 units, it would take 100 timesteps. The pheromone then takes much longer to decay, increasing the probability of an ant stumbling upon it and following it.

A crucial nuance in the model is that the ants have a heading, and can turn left or right. This is important, as there will be situation where the greatest amount of food pheromone is directly behind an ant. Everytime an ant returns to follow a trail that has been laid out by another ant, the path to the nest would be stronger, as it would have been made later. If the ants could do an about turn and follow the strongest smell, it would be heading back to the nest, instead of the food. This would result in the trail dying off, as no new ants would help reinforce it before it decayed.

It would have been excellent to provide more quantitative measures of how these constants affected the system. However, they are all very closely related, and the trends of the results would not be too surprising. In addition, the constants are so strongly coupled that if a constant was defined such that an interesting result occured, it might be giving a false indication of general behaviour of the system.

In closing, I would like to tell a story of one of my debugging adventures. I implemented the model in steps, first creating the terrain, adding behaviour to the ant, then adding the pheromones. My penultimate step was implementing the decay of the food pheromone. Before I implemented it, the ants would do this strange thing of making these loops of going to the food source and going back to the nest, but not collecting any food. It was very puzzling. This ended once I implemented the decay. In my final step, I implemented a visual system to see the pheromone trail. Playing around with turning on and off the decay factor, I realized what had happened. The ants would follow trails made by other ants so well, that they would never leave it. Thus they were stuck following the trail made by the initial 'trailblazers' and kept on grazing the food source, getting close, but not close enough. This anecdote highlights for me the challenges of implementing an agent based model; it is difficult to pin point where the error is. Rephrasing that, it is difficult to know if the half finished model is working, or if it contains significant design flaws.

8.6 Emergence

The word emerge has been thrown around a bit in this chapter, but what does it really mean? To get there, we need to explore the concept of emergence more. Professor Jeffrey Goldstein defines it that a system has emergent behaviour if has the following characteristics:

- Radical novelty
- Coherence or correlation
- A global or macro level
- It is a product of a dynamical process
- It is ostensive (It can be perceived)
- Supervenience (Downward causation)

All these characteristics do not have to exist for a system to be considered emergent, as that would preclude many systems. Radical novelty also has a ambiguous connotation. Once you have observed something, would it still be radically novel? It might be more apt to consider it radically novel if resulted in behaviour that was unexpected if only the agent's behaviour was known. From the ants model:

- An ant is able to somehow find food and bring it back to the nest efficiently. The highways of ants is also pretty radical.
- It is possible to make sense of the global ant behaviour by considering the behaviour of an ant.
- There is the global behaviour of the ants working in concert to bring food to the nest.
- The motion of the ants is a product of them moving around sniffing for pheromones.
- It is easy to observe the highways of the ants moving from the nest to the food sources.
- The movement of the ants, is dictated by the greater behaviour of the ants creating trails of pheromones.

The concept of supervenience could be more easily explained if we think of it as feedback. If we think of a system as a heirachy of more complex systems on the top and simple agents on the bottom, we can divide behaviour and characteristics of the system throughout the heirachy. We would expect all systems to have upward causation, where the more elementary components determine the behaviour of the system as a

whole. In contrast, downward causation implies that the more complex systems provide feedback to the more elementary units.

Emergence may be divided into two different perspectives: Weak emergence and strong emergence. Weak emergence describes new properties arising in systems as a result of the interactions at an elemental level. In this case, emergence is used as a model in order to describe a system's behaviour. By comparison, strong emergence implies that a system is irreducible into its components. Behaviour that observed in the system is a result of the interaction of its components, thus cannot be determined by determining the behaviour of the components.

It may seem that all the models described in this chapter are weakly emergent, as I have traced system behaviour down to the behaviour of individual agent. However, this ignores that I have been vague about the exact effect of an agent's behaviour on the greater system. I have pointed out trends, and made rational deductions about why such behaviour emerges. Emergence is often cited as a strong claim about the etiology of a system. Etiology is a word primarily used in philosophy to describe the study of why things work. My attempts at describing the nature of emergent systems is far from complete, as I do not purport to be able to predict accurately the behaviour of a system based on the choice of parameters used. In that sense, emergence could be cited as the reason for the system's behaviour. This would provide a framework to define the system, as we have already established that emergent system are difficult to reduce into components. It is interesting how by defining system to be strongly emergent, we can mostly get away from trying to provide closed solutions to the behaviour of emergent systems.

As a footnote, I want to point out the difficulty in designing emergent systems. The very characteristics that define it make it excruciatingly difficult to design. How would you design radical novelty? It is most likely a natural selection process enabled the optimization of such systems in the natural world. One certainty is that there are many poor combinations for parameters in an emergent system, and only a few optimal solutions.

Chapter 9

Stochastic modeling

Stochastic modeling is where we explore how non-deterministic systems act, aided by the mighty tools from probability and statistics.

9.1 Monty Hall

An very non intuitive problem is the Monty Hall problem. Wikipedia[1] has a great page describing the gameshow Let's make a deal, where the problem originates from. With that, we are comfronted with the decision, to switch or not to switch.

Knowing probability theory would help you determine the answer. The first choice is either the right or wrong door, with a $\frac{1}{3}$ chance of being right. Therefore if you switch, you have no chance of winning. However if you are wrong the first time, you now have a perfect chance of winning, as Monty has removed the wrong door, if you switch. Since the probability of being right the first time is $\frac{1}{3}$ the probability of winning if you switch is:

$$P = \frac{1}{3} \cdot 0 + \frac{2}{3} \cdot 1 = \frac{2}{3}$$

This takes some thinking in order to believe. The main piece that alludes to the benefit of switching is that Monty provides you with information when he opens one door. If he randomly opened doors, then the math above would be false.

The idea of conditional probability is powerful, as it can exaggerate the effects of rare events. Imagine a family with two children, each of which could be a boy or girl (assume the probability of either is equal). We know that there are three possible cases for the two children, both boys, both girls, and mixed in either order. Therefore, the probability of having two girls is one fourth.

However, if we constrict it to say we know that one child is a girl, then the probability of having two girls is 0.5. Its essentially saying ignore that there is two children, and focus on the probability of the second. Mathematically, conditional probability can be expressed as:

$$P(A|B) = \frac{P(A \bigcap B)}{P(B)}$$

where the probability of event A given event B, is the ratio of the event that both occur over the probability of B occurring.

[1] http://en.wikipedia.org/wiki/Monty_Hall_problem

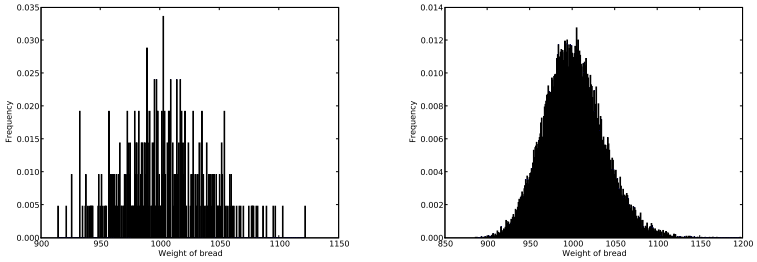
To illustrate the power of this, assume you have a needle in a 100 by 100 grid haystack. The probability of it being anywhere in the grid is $\frac{1}{10000}$. However, if we have searched through say half the haystack and have not found it, we know that the needle has to be in the other half of the haystack. In other words, the event of finding the needle is only in the space of finding it in half the hay stack, thus $\frac{1}{5000}$.

We can easily model these events in python as it contains a random number generator which is fairly good. They should all give the same results as the analytic solutions. You may need to simulate a large data set in order to get data that matches the analytical solution as there will be deviation due to random behaviour.

9.2 Poincare

Poincare supposedly had an issue with his baker: he thought that the baker was selling loaves which were lighter than advertised.

We can simulate a normal distribution in Python using the normalvariate function, which can generate a random sample from a normal distribution of a given mean and standard deviation. To prove Poincare's problem, we sample from a normal distribution of mean 950 and standard deviation 50. We choose n loaves from this distribution, and take the heaviest loaf. This is our choice. Doing this choice 365 times (to simulate a year), allows us to determine the distribution of bread that Poincare got. By varying the number of loaves looked at before choosing the heaviest, we can tune the resulting distribution to have the right mean. I found that to get a mean of around 1000, you would need to choose from four loaves. Figure 092 shows the distribution of the bread. Although it has a mean of 1000, it is a more asymmetric than one would expect. The difference is not very apparent, but it is clearly asymmetric.



(a) Bread for 365 days

(b) Bread for 36500 days, in order to make the distribution clearer

Figure 9.1: The distribution of bread given to Poincare by the baker. The graph on the right is an exagerration in order to show the shape more clearly. As the figures show, the distributions are asymmetric, thus Poincare has good reason to believe that his baker is shortchanging other customers.

9.3 Streaks

In sports, there is a tendenecy to glamorize streaks. Commentators will exclaim how amazing it was for some record to be set, and how it would go down in histiry. A simple simulation could illustrate that the sheer number of games played in sports would result in statistical rare events eventually.

The simulation is of a basketball team over a 82 game season. There are 10 players in each game, each of which takes 15 shots a game. Each shot has a 50% chance of going in. A streak would be considered having either 10 or more hits or misses in a game in a row. My simulation proved that, on average, there would be 1.5 such streaks every season.

Determining the length of streaks was something I had to ponder for a while. The best solution I found was the algorithm below:

Code Listing 9.1: Count Streaks

```
for i in xrange(15):
    if random.random()<0.5:
        hits += 1
        misses = 0
    else:
        hits = 0
        misses += 1
```

The for loop goes through every shot a player makes, and increments a counter if he hits or misses. By reseting the counter when the player changes what he does (missing when he has been scoring or vice versa), the counters effectively become counters for the length of streaks.

9.4 Bayes Theorem

Allen Downey's book gives a good introduction of Bayes theorem. An example of use of the Bayes theorem is in doing a Bayesian search. In such a search, you would use your accumulated knowledge in order to refine your prediction of the system. In the book, Ship of Gold in the Deep Blue sea, it was explained that some sort of Bayesian search was used to locate the ship wreckage based on historical knowledge, sensor data and other sources. In that sense, the ability of Bayesian inferences to use different kinds of information in order to determine probabilities is useful.

You can use also Bayes theorem in order to solve so called trivial problems as well. Suppose that you have two jars of cookies. In one, half is chocolate, and the other is plain. In the other jar, $\frac{3}{4}$ is chocolate, while the rest is plain. If you close your eyes, and randomly pick a cookie from one of the jars and get a chocolate cookie, what is the probability that you chose from the first jar?

If we define C to be the event a chocolate cookie was chosen, and O to the be event that the jar was the first one, we can see:

$$P(O|C) = \frac{P(C|O)}{P(C)} \cdot P(O)$$

which simplifies to:

$$P(O|C) = \frac{0.5}{3/8} \cdot 0.5 = \frac{3}{16}$$

Therefore, you could guess the identity of a jar pretty well just by sampling one cookie from it. More important uses of such sampling is in quality control, where a limited sample is used to try and predict the performance of a larger number of objects.