Markov Chain Monte Carlo Methods in Percolation
Final Report

Daniel Gnoutcheff '11
Mentor: Prof. Gary Reich

August 21, 2008

Contents

1 Motivation

In general, percolation theory is the study of CLUSTERS. When using percolation theory, one constructs a model that involves a discrete lattice of sites organized in some geometry (which defines which sites are "adjacent" to each other). Each site is assumed to have some probability p of being occupied. Percolation theory can then tell us about the clusters of adjacent occupied sites that are likely to form on the lattice.

Two quantities of interest in percolation theory and in percolation-based models are the CRITICAL EXPONENTS, β and γ. β describes how varying p changes percolation probability (P), or the probability that a randomly selected site is a part of an INFINITE CLUSTER, the cluster that spans the entire lattice. γ describes how varying p changes χ, the sum of the squares of the sizes of all of the clusters excluding the largest one.

Percolation theory has many applications in statistical mechanics, and the critical exponents are often related to experimentally observable properties. Interestingly, experimental data suggests that there are several "groups" or UNIVERSALITY CLASSES of phenomena that show the same critical exponents. It is hoped that with percolation theory, these universality classes can be explained in terms of lattice geometry. Thus, there is great interest in calculating the critical exponents for various geometries (different number of dimensions, etc.) so that we may determine what properties do and do not affect the critical exponents. However, the known methods for estimating these exponents tend to require data that is difficult to obtain. We are interested in finding better ways to obtain this data.

One desired set of data consists of the values of g_{st}/g_s. We define g_s as the number of unique clusters of size s as differentiated by shape and rotation, and we define g_{st} as the number of clusters of size s and perimeter t, where the PERIMETER of a cluster is defined as the number of unoccupied sites adjacent to some site in the cluster. The only known method for calculating g_{st}/g_s exactly is enumeration - that is, we must "count up" all the clusters of size s and determine how many of these clusters have a given perimeter length. Unfortunately, g_s grows *very* quickly as s increases; indeed, at $s = 47$, $g_s \approx 2.7 \times 10^{26}$! Therefore, even with computers, we have only calculated g_{st}/g_s values for $s \leq 22$. This is inadequate for the algorithms with which we want to use this data.

However, it is hoped that *estimates* for g_{st}/g_s values will be sufficient. To make estimates, it is only necessary to take a *random sample* of clusters of size s and count the number of sampled clusters that have perimeter t. As the sample size increases, the experimental probability of encountering a cluster of perimeter t should approach the theoretical probability (i.e. g_{st}/g_s). Our goal is to determine if this sample-based approach is sufficient.

2 Algorithm Overview

Generating a random sample of clusters is not straightforward; it's not as simple as randomly selecting clusters from the list of all clusters. After all, if such a list were available, we would have no need for a mere sample. However, there does exist a method that we believe will produce a random sample. The algorithm is as follows:

1. Start with an arbitrary initial cluster of the desired size. (Our method of generating this cluster is discussed in Section 3.3.)
2. Then, to "randomly select" a new cluster:
   (a) Remove a randomly selected cluster point.
   (b) If this removal "broke" the cluster (i.e. if there now are two clusters), replace the point to its original location.
   (c) Otherwise, put the point back to a randomly selected perimeter point.
3. Add the resulting cluster to our sample.
4. Repeat steps 2 and 3 as many times as desired.

In our case, we "add" clusters to our samples by incrementing the appropriate entry in a perimeter frequency table. These tables are setup to indicate how many clusters of each perimeter were sampled. This way, a single sample will allow us to estimate g_{st}/g_s values for all values of t.

To show that this method will indeed generate a random sample, we call on the mathematics of Markov chains. Consider the general form of our method; it starts at some arbitrary state (i.e. cluster), and in each iteration, it selects a new state via a random process that has a probability distribution determined only by the current state. This meets the definition of a Markov chain, and so the mathematical theorems of Markov chains can be applied.

Mathematicians tell us that if we want to produce a random sample with probability distribution π, we just need to use a Markov chain that satisfies two properties:

1. Irreducibility: if the process is run for long enough, every state (i.e. cluster) must have a chance of being reached, regardless of what the starting state is.
2. Stationarity of π: For each state y, $\sum_x \pi_x p_{x\to y} = \pi_y$, where $p_{x\to y}$ is the probability of transitioning to state y when starting from x, and π_x and π_y are the probabilities that we want to be associated with states x and y when making the random sample. (In our case, every cluster should have an equal chance of getting selected, so $\pi_x = \pi_y$ for all x and y).

It has been mathematically proven that our algorithm shows stationary of π. We also believe that our process satisfies irreducibility, though we are not yet aware of a formal proof of this. All told, we are fairly sure that our algorithm is correct.

3 Test implementation

However, mathematics can only tell us that as the number of iterations approaches infinity, our method will *eventually* produce a good random sample. There is no guarantee that a limited number of samples will produce anything useful. Thus, empirical tests are necessary.

We have implemented a program that uses our method to take a random sample of clusters on a two dimensional square lattice. Here, we will discuss some particularly notable properties of our implementation.

3.1 "Trace Cluster"

The most difficult task is determining if the removal of a cluster point has "broken" a cluster. This is difficult because the arrangement of cluster points at locations far from the removed point can change the result of this test.

We used a fairly simple-minded approach to this problem; the "trace cluster" algorithm, implemented in TRACE_CLUSTER.PRO. Ignoring a few details, this procedure accepts a list of coordinates of occupied points and then effectively creates a list of the points that are connected to a given starting point. We consider the cluster to be closed if and only if the number of points in this list is equal to the total number of occupied sites.

The algorithm works by maintaining a list of all locations that are adjacent to points in the "cluster sites" list. In each iteration, we try to find an adjacent point that is occupied. If there exists such a point, we move it to the cluster sites list, and then we update the adjacent points list accordingly. This process continues until there are no more occupied adjacent sites.

This algorithm does work, but it is quite slow. Indeed, the TRACE_CLUSTER procedure is usually the most computationally expensive procedure in our program. Efforts to optimize this procedure or find a faster algorithm would be beneficial.

A side effect of this algorithm is the generation of a list of perimeter sites; when the procedure is complete, the adjacent sites list is effectively a perimeter sites list. In the case where the cluster is not broken, this list is used when randomly selecting a perimeter point and when determining the cluster's perimeter length when updating the frequency table.

3.2 Maps, "uni"s, and "bi"s

Many of our algorithms, including TRACE_CLUSTER, frequently need to determine the "status" of a given location (e.g. determine if it is occupied). While it is possible to do this by searching the list of occupied sites, this is far too slow. Thus, our program makes the use of "maps", large 2-dimensional square arrays that contain one entry for each location that could ever be of interest. To keep the clusters from drifting off these maps, the programs are written to keep clusters "anchored" at the origin point, and we size the map arrays such that even if a cluster were at its maximum extension (i.e. a straight line extending from the origin "anchor"), the cluster and all of its adjacent points will still be on the map. Since we are working with square lattices, this means that the map must have dimensions $(2s+1) \times (2s+1)$ (with the origin at the center).

While these maps are fairly fast, they are memory intensive. For example, to process a cluster of size 8100, one needs about half a gigabyte of memory for the map alone (assuming 2 bytes are needed for each location). This limits the size of the clusters that can be comfortably processed with this program; for larger clusters, hashing methods will be necessary.

While we could use 2-dimensional coordinate pairs (which we call "bi" coordinates) when addressing these maps, here again we can save computer time. Unlike our lattice, computer memory is 1-dimensional and does not support the idea of negative coordinates. Thus, when using 2-dimensional coordinates, a conversion process is necessary to generate the equivalent "uni coordinates", the 1-dimensional indexes with which we reference the "map" array. If our programs used coordinate pairs directly, we would have to perform this conversion every time we used the map. Since many locations are likely to be referenced repeatedly, this is wasteful.

We avoid this inefficiency by writing our programs to use "uni" coordinates internally. Before processing, our programs use the BI2UNI function to convert coordinate pairs in "uni" coordinates, and then use UNI2BI for convert these back into coordinate pairs to return to the user.

It must be noted that "uni" coordinates depend on the size of the map that they are indexes for. If the map size is changed, the "uni" coordinates will change, even if their positions relative to the origin are not changed. "Uni" coordinates cannot represent points that are outside their corresponding map, and "bi" coordinates are much easier for users to interpret. Therefore, for storing lists of locations and for manipulating clusters manually, "bi" coordinates are preferred.

3.3 Initial cluster selection

It is also important to discuss the selection of the initial cluster with which to start the sampling. In theory, this initial cluster doesn't matter, but in practice, the limited number of samples require that we get the sampling process on a "good start".

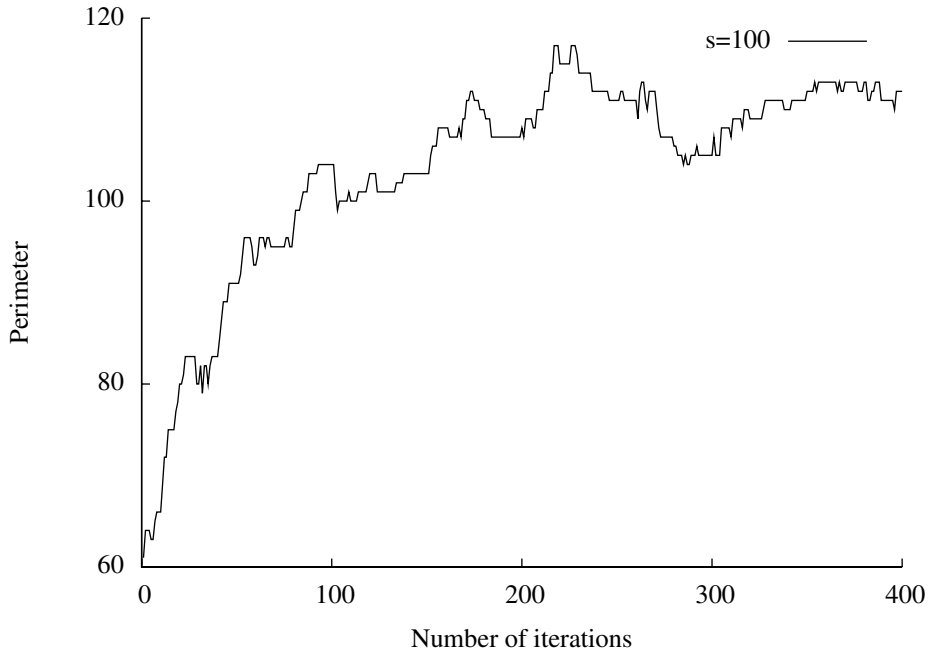

Figure 1: A graph generated during a typical "equilibrium watch" run. We started with a cluster of size 100 as generated by Eden growth, and we performed number of iterations of our "sampling" process, recording the perimeter of the cluster at each iteration. Note that the process is started with a cluster of a relatively small perimeter, and after about 300 iterations, perimeter length reaches a plateau. This plateau indicates that we have reached a "typical" perimeter length.

In our case, we are interested in cluster perimeters, so it is desirable to start with a cluster that has a "typical" perimeter length. If we start with a cluster with an "unusual" perimeter, it will take some time before the sampling method will reach clusters with more typical perimeters. As a result, these "unusual" perimeters are almost guaranteed to be selected for the sample, and unless the sample is very large, these perimeters will be overrepresented in the sample. Starting with a "typical" perimeter avoids this problem.

We use a two step method for generating an initial cluster. We call the first step "Eden growth" and the second "equilibrium watch".

We use Eden growth to just generate *some* cluster of the desired size[1]. The Eden growth process starts with some "cluster" (which could be as simple as a single occupied point), and then for each iteration, we add a cluster point to a randomly selected perimeter site. We continue this until the cluster is large enough.

Eden growth does not select clusters fairly; it strongly favors clusters with small perimeters. The "equilibrium watch"[2] procedure is used to counteract this. Here, we perform a number of iterations of our sampling method. Instead of recording perimeter frequencies, we record perimeter length over time. Figure 1 on page 4 is an an example of the data collected from this process. Generally, the perimeter will approach the "typical" perimeter length and will stabilize that that perimeter (i.e. reach equilibrium). We visually inspect graphs such as Figure 1 to determine if equilibrium has been reached, and we continue to run the process until that happens.

Note that once a suitable initial cluster is obtained, that cluster can be used to help obtain larger initial clusters. Instead of using Eden growth to create a cluster "from scratch" (i.e. by starting from a cluster that consists of a single point), we can use Eden growth to "grow" the smaller initial cluster to a larger size. Not only does this shorten the Eden growth process, but the resulting cluster's perimeter will be closer to the "typical" perimeter, which will shorten the "equilibrium watch" step. When working with large clusters, this shortcut is recommended.

[1]This process is implemented in the program file EDEN_GROWTH.PRO. The function in CLUSTER_INIT.PRO is a shortcut for generating an Eden growth cluster from scratch.

[2]This is implemented in EQUILIBRIUM_WATCH.PRO.

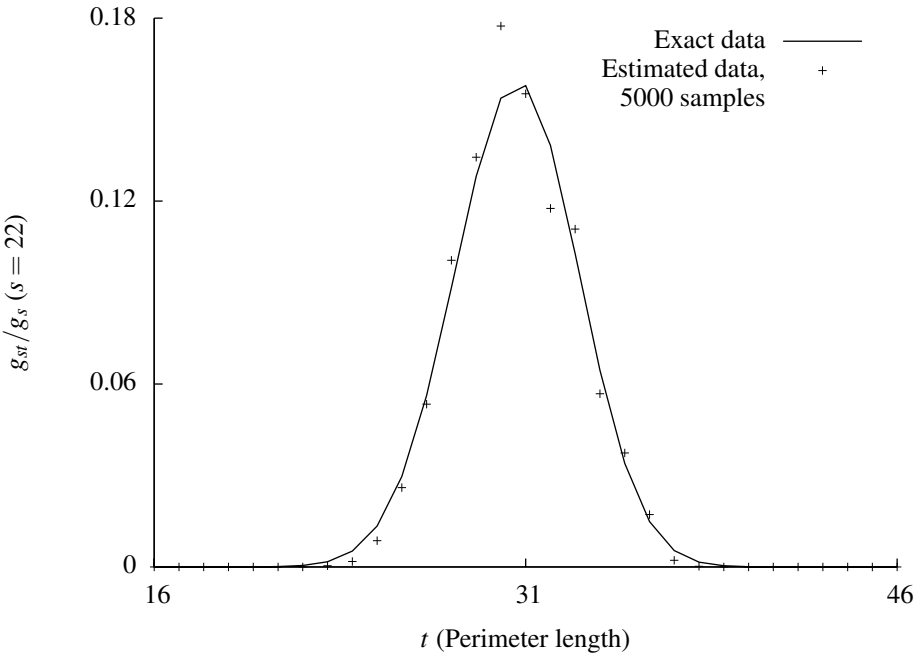

Figure 2: Comparing exact g_{st}/g_s values (for $s = 22$) with estimates based on a sample of 5000 clusters.

4 Basic tests and results

Once we implemented the sampling method, we applied a few empirical tests to check the feasibility of our method and the correctness of our implementation.

4.1 "Histogram" test

For our first test, we compared the available exact values of g_{st}/g_s with estimates based on samples generated with our sampling method. We generated g_{st}/g_s estimates for $2 \leq s \leq 22$, and for each s, we simply plotted the exact and estimated data on the same graph and did a visual comparison.[3]

Figures 2 and 3 are typical examples of the graphs generated in this process. In all of the comparisons made, the estimates where at least as good as that found in Figure 2. Furthermore, the estimates in Figure 3 are particularly impressive when we consider that they are based on a sample of 500,000 clusters out of about 3.5×10^{11} clusters of size 22.

Overall, the results where encouraging; at small cluster sizes, no glaring errors appear. However, to test our estimates at cluster sizes beyond those that have been enumerated, another test is necessary.

4.2 $f(s)$ test

Consider the function $n_s(p) = \sum_t g_{st} p^s (1-p)^t$; this gives the probability of obtaining at least one cluster of size s when the probability of any given lattice point being occupied is p. Now consider $f(s)$, which gives the value of p that maximizes $n_s(p)$. It has been shown that as s increases, $f(s)$ should converge on p_c (the "critical probability", the minimum probability needed for an infinite cluster to appear).

$f(s)$ can be calculated by finding the zero of the derivative,

$$\frac{dn_s}{dp} = \sum_t g_{st} \left(s p^{s-1} (1-p)^t - t p^s (1-p)^{t-1} \right) = s p^{s-1} \sum_t g_{st} (1-p)^t - p^s \sum_t g_{st} t (1-p)^{t-1} \tag{1}$$

[3]The programs in SIM_COMPARE.PRO and COMPARE.PRO were used for this.

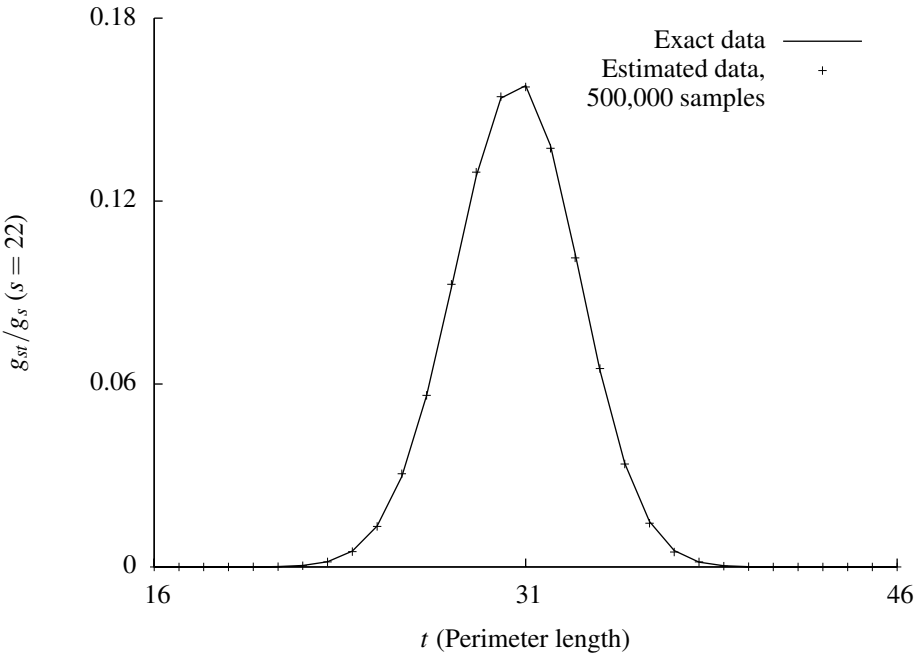

Figure 3: A repeat of Figure 2 on page 5, this time with 500,000 samples. Note that the estimates are a great deal more accurate.

Multiplying k/p^{s-1} (where k is an arbitrary constant), we determine that finding the zero of (1) is equivalent to finding the zero of the expression

$$s\sum_t kg_{st}(1-p)^t - p\sum_t kg_{st}t(1-p)^{t-1} \tag{2}$$

This is true regardless of what k is. So, if we let $k = 1/g_s$, we can use our g_{st}/g_s values to generate $f(s)$ estimates[4]. Thus, we can check our g_{st}/g_s values by using them to make a graph of $f(s)$ and ensuring that the graph has the expected shape.

Figure 4 on page 7 is just such a graph. For those cluster sizes where exact g_{st}/g_s values are available, we have calculated and plotted $f(s)$ exactly. Then, we calculated and plotted $f(s)$ values using g_{st}/g_s estimates for $1 \leq s \leq 47$. A visual inspection of the graph shows that the $f(s)$ estimates are quite consistent with the exact values, and $f(s)$ is shown to be monotonically increasing, just as expected. It does appear, however, that we will have to look at much larger cluster sizes in order to see a convergence on p_c.

To generate this graph, we evaluated $f(s)$ by numerically finding the zero of Equation (2)[5]. It should be noted that this method does not work well; as s increases, the slope of (2) near the zero becomes *very* small. As a result, when s is large, our program demands unreasonably high precision from zero-finding procedures. Indeed, early implementations of this function broke down at about $s = 40$, and even the current implementation is known to be useless by the time s reaches 100. Therefore, any further investigation of $f(s)$ will require the creation of a better algorithm.

5 Usage tutorial

To begin, start up IDL (or an equivalent program, such as the free and open-source GDL) and set the current working directory to the directory where the program files are stored. Then, run the `mz0` command to setup the random number generator.

[4]We can simplify this further by setting $k = n_s/g_s$, where n_s represents the number of samples we have taken of clusters of size s. Then, we will need to supply $g_{st}(n_s/g_s)$ values - which correspond exactly to the numbers in the frequency tables. This way, we don't even need to convert our frequency tables into g_{st}/g_s estimates. Indeed, this is what our $f(s)$ implementation does.

[5]This is implemented in DNDP_ZERO.PRO.

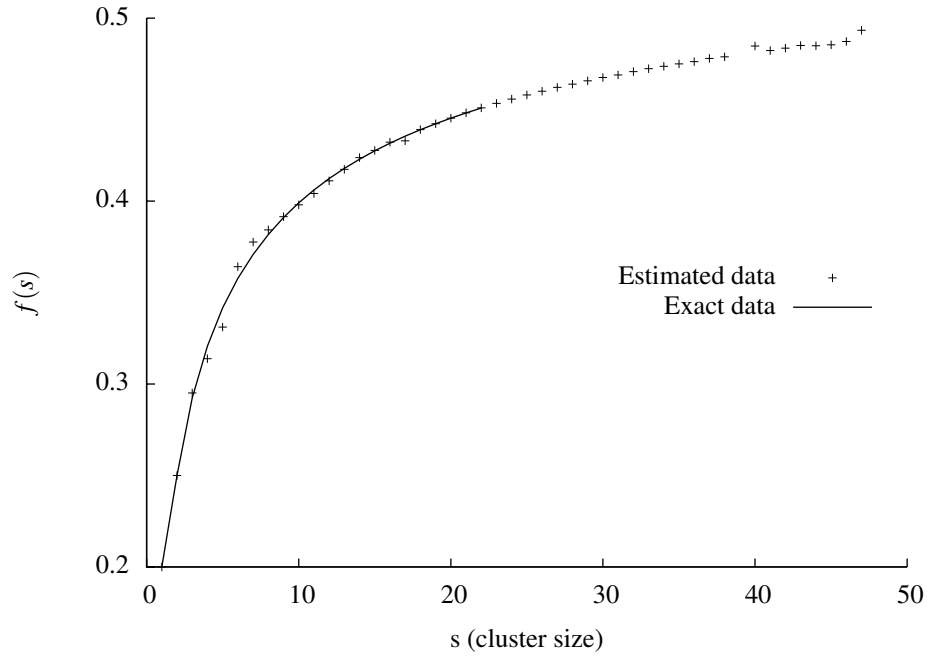

Figure 4: Graph of $f(s)$ as calculated using both exact and estimated g_{st}/g_s values.

```
gnoutchd@dan-lap:~$ gdl
GDL - GNU Data Language, Version 0.9
For basic information type HELP,/INFO
'GDL_STARTUP'/'IDL_STARTUP' environment variables both not set.
No startup file read.
GDL> cd, '~/College/08cSU/Research'[6]
GDL> mz0
% Compiled module: MZ0.
GDL>
```

To use Eden growth to generate a cluster of some size, use the `cluster_init` function. You can use the `plot_cluster` function to view the result. (See Figure 5 on page 8 for the graph generated by `plot_cluster`).

```
GDL> test_cluster = cluster_init(40)
% Compiled module: CLUSTER_INIT.
% Compiled module: EDEN_GROWTH.
% Compiled module: N_BI_POINTS.
% Compiled module: BI2UNI.
% Compiled module: FIND_ADJACENT.
% Compiled module: UNI2BI.
% Compiled module: ARRAY_INDICES.
GDL> plot_cluster, test_cluster
% Compiled module: PLOT_CLUSTER.
GDL>
```

We will then want to perform the "equilibrium watch" step, as described in Section 3.3. First, we need to generate the structure that contains temporary memory for the `markov_chain` and `trace_cluster` programs. (Be aware that if we are working with a large cluster, this structure can be very large. Avoid making copies of it!)

[6] ~/College/08cSU/Research is the path to the directory where these programs are stored on my system. Replace this with the path to the directory where they are stored on your system.

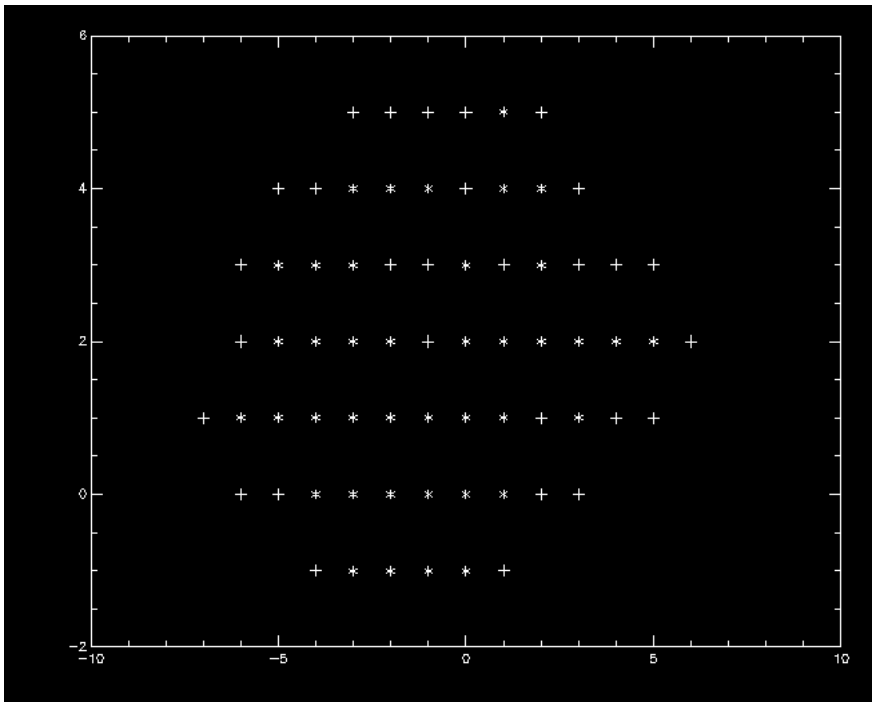

Figure 5: Typical graph from `plot_cluster`

```
GDL> test_markov = bi2markov(test_cluster)
% Compiled module: BI2MARKOV.
GDL>
```

We can then use the equilibrium watch function to run the sampling process and watch the perimeter length. These commands, for example, will run 20 iterations and make a the graph of perimeter length over time. (The resulting graph is shown in Figure 6 on page 9.)

```
GDL> watch = equilibrium_watch(test_markov, 20)
% Compiled module: EQUILIBRIUM_WATCH.
% Compiled module: MARKOV_CHAIN.
% Compiled module: MZRAND.
% Compiled module: TRACE_CLUSTER.
GDL> plot, watch
GDL>
```

If we are not sure that we have reached equilibrium, we can perform more iterations and append the data to the `watch` array, like so:

```
GDL> watch = [watch, equilibrium_watch(test_markov, 40)]
GDL>
```

Once we are satisfied that we have reached equilibrium, we can start the sampling process in earnest. It is recommended that one use the `markov_interruptible` function; this is a wrapper for `markov_chain` that implements a workaround for a bug in the Microsoft Windows version of IDL. The workaround ensures that the sampling process can be aborted with `Ctrl+Break`[7]. Here, we run the sampling process for 5000 iterations:

```
GDL> test_hist = markov_interruptible(test_markov, 5000)
% Compiled module: MARKOV_INTERRUPTIBLE.
GDL>
```

[7] See `http://www.dfanning.com/misc_tips/endlessloop.html` for information about this bug and the available workarounds.

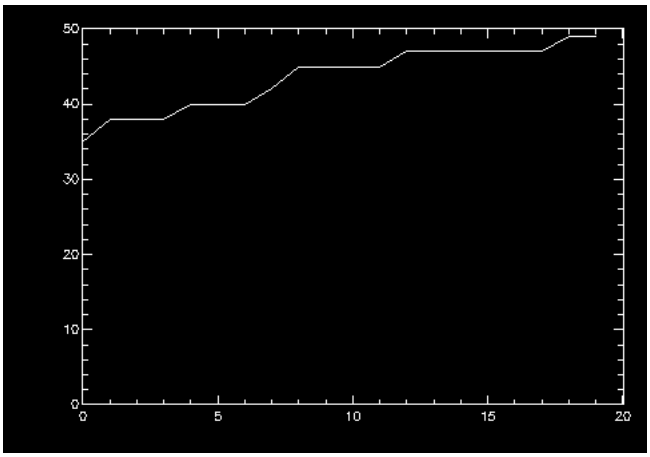

Figure 6: Graphing the output of the `equilibrium_watch` function.

`test_hist` now contains a frequency table; the number of times a cluster of perimeter t was sampled is stored in the element `test_hist[t-1]`.

If we now wish to take a sample of some larger clusters, we can save some time in the equilibrium watch step by starting eden growth with a smaller cluster. These commands will grow our cluster to size 80:

```
GDL> test_cluster = markov2bi(test_markov)
GDL> test_cluster_2 = eden_growth(test_cluster, 80)
GDL>
```

We can then apply the equilibrium watch procedure, just as before.

5.1 Parallelization

If multiple computers are available, it is possible to increase the number of samples taken by running multiple instances of the sampling process in parallel. Once we have generated an acceptable starting cluster, we can save it and use it to start several sampling runs. To save the starting cluster:

```
GDL> initial_cluster = markov2bi(test_markov)
% Compiled module: MARKOV2BI.
GDL> save, initial_cluster, filename='initial_cluster.sav'
GDL>
```

Then, copy the resulting file (`initial_cluster.sav`) on to several systems. On each machine:

```
GDL> mz0
GDL> restore, 'initial_cluster.sav'
GDL> test_hist_n = markov_interruptible(bi2markov(initial_cluster), 5000)
GDL> save, test_hist_2, filename='test_hist_2.sav'
GDL>
```

(n should be a number unique to each machine).

Then, we can copy all of the histogram save files to one system and add them together:

```
GDL> restore, 'test_hist_1.sav'
GDL> restore, 'test_hist_2.sav'
GDL> test_hist = test_hist_1 + test_hist_2
GDL>
```

The histogram table is ready for analysis.

6 Todo

This research is very preliminary in nature; there is much that can be done from here. Here are some possible tasks:

1. Attempt to find the critical exponents. These numbers are already known for square lattices, but this would be the ultimate test of our method.
2. Perform some formal statistical analysis. The tests so far have relied upon visual inspection, a technique that is notoriously unreliable. Statistical analysis would give us a more rigorous understanding of the properties of our technique. In particular, it would be good to know exactly how large our samples should be.
3. Extend the existing programs. These programs are currently limited to relatively small cluster sizes (due to memory limitations), are quite slow, and don't easily permit changes in lattice geometry. Solving these problems will require major modifications to these programs.

7 Acknowledgements

- My research mentor, Prof. Gary Reich, for the majority of the ideas discussed here.
- The Union College CT Scholars program, for funding.
- The National Science Foundation, for supporting the CT Scholars program.
- Brandon Bartell '10, who worked on a related project and with whom I made a joint presentation of our reserach.
- Prof. Valerie Barr, for Bartell's funding.