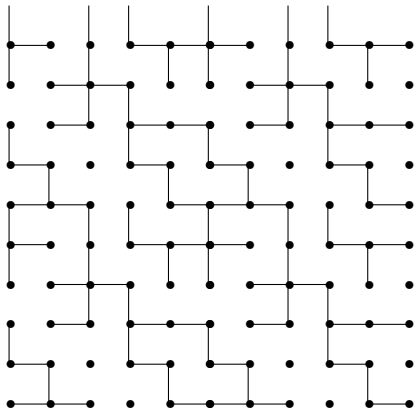
Wesleyan University

Introduction to Percolation and Determinate Percolation

by
Dave Darling



Advisor: Michael S. Keane
Professor of Mathematics

A Dissertation in Mathematics
submitted in partial
fulfillment of the
requirements for the degree of Master of Arts
at Wesleyan University

Middletown, CT **May, 2007**

Acknowledgements

First, I need to thank Professor Michael S. Keane, my advisor and support during the creation of this. A year ago I had no idea for what to write as a thesis, and I came to him for ideas. Without his guidance I don't think the doors to percolation would have opened.

I also must thank my undergraduate advisor, Professor Hanamantagouda Sankappanavar. Without your drive to make me apply to graduate school, I would have never had the opportunity.

Thank you Wesleyan, for allowing me to continue my studies.

Another thanks to Travis Beebe. I wanted a computer simulation for determinate percolation, and didn't have the time or skills to write one. He not only wrote what I wanted, but exceeded my expectations by allowing for dimensions larger than two.

And lastly, thanks to my colleagues, family, friends, and professors. All of you have helped in many ways, from love to learning to keeping me sane.

Contents

Introduction

Take a porous stone and immerse it in water. Does the center of the stone contain water? The answer to this question can be given by understanding mathematical percolation [5].

The percolation process is a model of an event passing through a space. The event can be one such as a disease spreading, or such as an object being permeated by a liquid. The space restricts this. In the pure mathematical sense of the percolation process, we care not about the event, but its ability to span [10]. To be able to handle this problem, we consider our space to be in the form of a random lattice. We mostly work with the square lattice, but will mention some others such as the triangular lattice.

There are two types of mechanics used to describe how our liquid penetrates through our lattice. In *site percolation*, each vertex has a probability (independent of the rest) of being 'open', otherwise it is 'closed'. Our liquid is allowed to pass to an adjacent vertex only if it is open. *Bond percolation* considers edges in our lattice open or closed and events travel from one vertex to an adjacent vertex only if the edge is open [10].

At some probability level, our liquid will succeed in reaching an infinite number of vertices. This leads us to the idea of a *critical probability* (or p_c), where if our vertices/edges have a probability greater than p_c of being open, then with positive probability the liquid will reach an infinite number of vertices. On the other hand, if the probability that a vertex or edge is open is less than our critical probability, then surely the event will not be able to reach infinitely many vertices. There is a *covering lattice* method that will convert a bond problem to site problem, showing $p_c^{bond} \leq p_c^{site}$. The question to what happens at the critical probability remains unanswered for many lattices. For the 'square' lattices with dimension greater than 19, it has been proved that an infinite cluster does not exist p_c [9]. One and two dimensions will be discussed in this thesis, and it remains unknown what happens on the square lattices between 3 and 18 dimensions.

For rigorous calculations (not presented in this thesis), two main inequalities are used. They are the FKG and BK inequality. The FKG tells us that if we know that vertices u and v are connected by an open path, then it is at least as likely that vertices y and z are connected knowing so. The BK inequality states that under the same hypotheses, the chance that v, u, y, and z are not all connected is at most the product of their individual probabilities [5].

In chapter 2 we consider percolation on a finite rectangular sub-lattice. After

determining which edges or vertices are closed in our finite lattice, we 'copy' the outcome and periodically layer the entire plane with these copies. We call this *determinate percolation* as one sub-lattice or *tile* determines the outcome of the entire lattice. This is a new model which I have not been able to find in the literature, and which exhibits apparently some interesting phenomena.

Only what seems to be the very basics of determinate percolation are presented. We only discuss rectangles taken from the 2-dimensional square sub-lattice, and thus in addition to what we managed to unravel, we have more questions than we have answers. For example: when we tile the plane, instead of just translating the tile, what happens if we also allow for reflections and/or rotations? If we expand the size a square tile towards ∞, can we show that percolation occurs with a 50/50 chance when the probability that the vertex/edge is set at p_c? What happens if we expand the tile but at side lengths without a 1:1 ratio? What happens when we use different lattices other than the square (so long as they fit together). What if we tile the plane with different 'determinate' tiles (same shape or of different shapes)?

Chapter 1: Introduction to Percolation

Let V be a countable set and let E be a collection of subsets of V of size two. Then $G = (V, E)$ is a countable *graph*. We call $v \in V$ a *vertex* in G. We call $e \in E$ an *edge* in G, $e = \{u, v\}$ where $u, v \in V$, $u \neq v$. Vertices are also referred to as sites and edges can also be referred to as bonds.

A *subgraph* A of G, denoted $A \subseteq G$, is a graph $A = (V_A, E_A)$ where $V_A \subseteq V$, and $E_A = \{\{v, v'\} : v, v' \in V_A \text{ and } \{v, v\} \in E\}$.

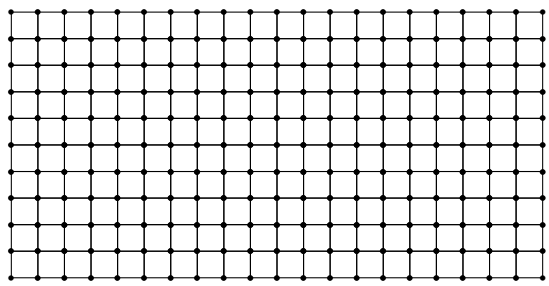


Figure 1.1: $\mathbb{L}^2$ = the square lattice

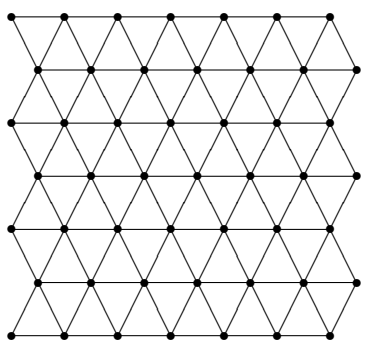


Figure 1.2: The triangular lattice.

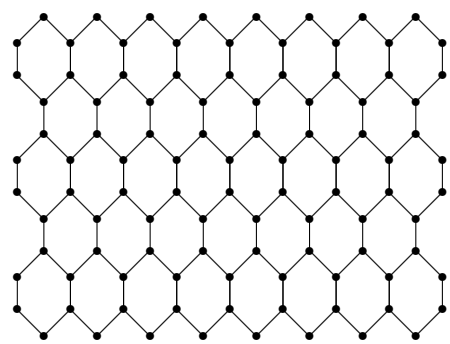


Figure 1.3: The honeycomb lattice.

In this thesis we shall be most interested in subgraphs of the particular graph $G = (V, E)$ such that $V = \mathbb{Z}^d$, the integer points in dimension d (particularly $d \geq 2$), and $E = \mathbb{E}^d := \{\{v, v'\} : v, v' \in V, |v - v'| = 1\}$. This graph is also referred to as the d-dimensional lattice and often denoted by $(\mathbb{L}^d)$.

Two vertices v and v' are *adjacent* if $\exists$ $e \in$E such that $e = \{v, v'\}$. Two edges are adjacent if they share one (and only one) vertex.

A *path*, π, of *length* n from v to v' is a finite sequence of distinct vertices

$\pi = (v = v_0, v_1, ..., v_n = v')$ such that $v_i \in V$, $v_i \neq v_j$ for $i \neq j$, and for each $0 \leq i < n$, $e = \{v_i, v_{i+1}\} \in E$.

A *circuit* is a path plus the edge $e = \{v_n, v_0\}$. If the path has length n, the circuit has length $n + 1$.

A *tree* is a graph which contains no circuits. If $\exists$ a path between v and v', then such a path is unique.

Suppose we have subgraphs A,B of G. We say A and B are *edge-disjoint* if they have no edges in common. A and B are *disjoint* if they have no vertices in common.

Let us give edges the property of being either *open* or *closed*. Let Ω be a configuration by setting every edge as either open or closed, and let $\omega(e)$ signify the state of an edge by $\omega(e) = 1$ if it is open and $\omega(e) = 0$ if it is closed. $\Omega = \Pi_{e \in \mathbb{E}^d}\{0, 1\}$ with elements $\omega = \{\omega(e) : e \in \mathbb{E}^d\}$ [5].

We could have alternatively defined the graph to have open/closed vertices. If we are considering vertices being open or closed, then we are looking at *site percolation*. If we are interested in whether edges are open or closed, we are studying *bond percolation*. Definitions between bond and site percolation to come will differ slightly (and as expected); in this thesis we will be most interested in bond percolation.

Given a configuration, two distinct vertices v, v' are joined by an *open path* $\pi = (v = v_0, v_1, ..., v_n = v')$ if $\omega(\{v_i, v_{i+1}\}) = 1$ for every $0 \leq i < n$. A *closed path* is a path in which every edge is closed.

A vertex v is *isolated* if it has no adjacent vertices, that is, there are no edges $e = \{v, v'\}$ for every $v' \in V - \{v\}$.

Two vertices v, v' belong to the same *cluster* if there is an open path from v to v'. For a vertex $x \in V$, the *cluster of* x, $C(x)$, is the set of vertices which are joined by open paths to x. We denote C as the cluster which includes the origin, and $|C(x)|$ is the cluster size. Note that isolated vertices have a cluster size of 1. A *closed cluster* is likewise a set of vertices which are connected by closed paths.

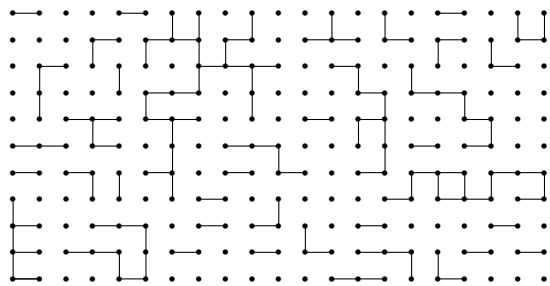


Figure 1.4: A configuration of $\mathbb{L}^2$, $\mathbb{L}^2(\omega)$

Lets look at the graph $G(\omega) = (V, E(\omega)) \subseteq G = (V, E)$, where $E(\omega) = \{e \in E : \text{e open for } \omega\}$ (The only edges shown in the figure 1.4 are open edges). Clusters in $G(\omega)$ are nonempty sets of vertices connected by a path in $G(\omega)$, and form an equivalence relation on V. It is easy to see that every connected component of $G(\omega)$ is a cluster and the set of all clusters partition V.

Let $a, b \in \mathbb{Z}^d, n \in \mathbb{Z}^+$. A *box* is a subset $\mathbb{Z}^d$ denoted by

$$B(a, b) = \{v \in \mathbb{Z}^d : a_i < v_i < b_i \ \forall i\}.$$

We write $B(n)$ for boxes centered at the origin instead of $B(-n, n)$.

Given subsets of vertices $A, B, D \subseteq G$, we write $A \leftrightarrow B$ if $\exists$ an open path joining some vertex in A to some vertex in B, and $A \nleftrightarrow B$ if there does not. $A \leftrightarrow B$ *off* D if A$\leftrightarrow$B exists using no vertex in D.

Let A be a subgraph of G. The *perimeter of* A is the set of vertices not belonging to A but adjacent to at least one vertex in A. This is sometimes referred to as *outer boundary*, denoted $\partial^+ A$.

Let A be a subgraph of G. The *surface of* A, denoted ∂A is the set of vertices in A which are adjacent to at least one vertex in the perimeter of A. That is $\partial A = \{v \in V : v \in A, \{v, v'\} \in E, \text{ and } v' \notin A\}$. The surface can also be referred to as *inner boundary*, or $\partial^- A$.

To determine a configuration of Ω, we generally use the following procedure: Let p be a fixed number such that $0 \leq p \leq 1$. We say an edge is open ($\omega(e) = 1$) with probability p, otherwise it is closed ($\omega(e) = 0$), independent of other vertices. The configuration given in figure 1.4 was randomly generated with $p = 0.3$.

Let $\mathfrak{F}$ be the product σ-algebra on Ω and let $\mathbb{P}_p = \Pi_{e \in \mathbb{E}^d}(1 - p, p)$ be the product *probability measure* on $(\Omega, \mathfrak{F})$ giving each edge a probability p of being open and $1 - p$ of being closed independently of all other edges.

The *percolation probability*, denoted by $\theta(p)$, is the probability that the origin belongs to an infinite cluster. That is,

$$\theta(p) = \mathbb{P}_p(\{\omega \in \Omega : |C(\omega)| = \infty\})$$

If we wish to use a point other than the origin, we write

$$\theta(p) = \mathbb{P}_p(\{\omega \in \Omega : |C(x)(\omega)| = \infty\}) \text{ for any fixed } x \in \mathbb{Z}^d.$$

It is easy to see that $\theta(p)$ is a non-decreasing function of p.

The *critical probability* p_c is defined by the setting

$$p_c = sup\{p : \theta(p) = 0\} = inf\{p : \theta(p) > 0\}$$

Thus, for $p < p_c$ we have $\theta(p) = 0$ and for $p > p_c$ we have $\theta(p) > 0$. If we need to distinguish between the critical probability of bond or site percolation, we write p_c^{bond} or p_c^{site}.

An *embedding* is the placement of a graph upon a surface such that connectivity is preserved. In our case, we are using the infinite plane as our surface.

Given a graph G and an embedding onto a surface S, a *face* is a connected component of $S - G$. In $\mathbb{L}^2$, our faces are the squares lying between its edges.

The *dual* of (planar) G is the graph $G_d = (V_d, E_d)$ such that there exists a unique $v \in V_d$ for every face in G, including the infinite one if it exists. The edge set $E_d = \{v, v'\}$ such that $v, v' \in V_d$ and the corresponding faces of v and v' share a boundary edge in G.

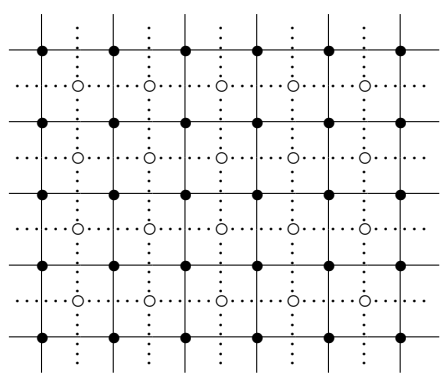


Figure 1.5: $\mathbb{L}^2$ and it dual.

The dual of $\mathbb{L}^2$ is itself shifted right and up both by half a unit. More formally $\mathbb{L}^2_d = (V_d, E_d)$ where $V_d = \{(x+1/2, y+1/2) \forall x, y \in \mathbb{Z}\}$ and $E_d = \{\{v, v'\} : v, v' \in$

V_d, $|v - v'| = 1\}$. Figure 1.5 is the lattice $\mathbb{L}^2$ and it's dual. The vertices of the dual are drawn as open circles, and edges are dotted lines.

For the graph $G = (V, E)$ let us configure the dual $G_d = (V_d, E_d)$ as such: $e \in E$ is open if and only if e_d is open, where $e_d \in E_d$ is the edge which crosses $e \in E$.

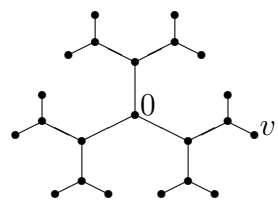


Figure 1.6: A tree graph
(known as a Cayley tree or Bethe lattice).

Lemma 1.7: If $G = (V, E)$ is an infinite tree, then $p_c^{bond} = p_c^{site}$.

Proof: Choose a point as the origin. Short of the the probability that the origin is closed or open in site percolation, for any $v \in V$ there is isomorphism between the connection between the (unique) path from 0 to v using either open sites or open bonds. In figure 1.6, 0 and v belong to the same cluster if either the three bonds or sites between them are open. It does not matter whether the vertex, or the edge immediately before it is determined open or closed. □

Theorem 1.8: For the one-dimensional lattice, percolation (bond or site) occurs only when $p = 1$.

Proof: When $p = 1$ the entire chain is open. Lets assume $0 \leq p < 1$. At any particular vertex v, the probability that there exists an open path to a vertex right (or left) that is of L units away is p^L. Since $p < 1$ as L approaches infinity we have a geometric series.

$$\lim_{L\to\infty}(p^L) = 0$$

So the chance of an infinite cluster has probability zero if $p \neq 1$ [11]. □

Theorem 1.9: If N is an nondecreasing random variable on $(\Omega, \mathbb{F})$, then $E_{p_1}(N) \leq E_{p_2}(N)$ whenever $p_1 \leq p_2$, so long as the mean values exist. If A is an increasing event in $\mathbb{F}$, then $P_{p_1}(A) \leq P_{p_2}(A)$ whenever $p_1 \leq p_2$.

This theorem seems intuitively clear: with an increasing variable, N,or event, A, if you increase the probability of p, we should not expect a lower occurrences of N or less likely chance of A. Grimmett gives a short proof for Theorem 1.9 on page 33 [5]

Theorem (probability) 1.10- Kolmogorov's zero-or-one law [7]: If $F \subseteq \Omega$, $F \in \mathfrak{F}$ is shift invariant, then either $P_p(F) = 0$, or $P_p(F) = 1$.

Theorem 1.11: The probability $\psi(p)$ that there exists an infinite open cluster satisfies: $\psi(p) = 0$, if $\theta(p) = 0$ $\psi(p) = 1$, if $\theta(p) > 0$

Proof: By the zero-one law, ψ can only take the values 0 or 1, since the event F that there exists an infinite open cluster is shift invariant. So if $\theta(p) > 0$, then $\psi(p) \geq P_p(|C| = \infty) > 0$. So $\theta(p) > 0 \Rightarrow \psi(p) = 1$.

Burton and Keane [2] also prove that if there exists an infinite cluster, then it is unique.

Theorem 1.12: $p_c(d+1) < p_c(d)$

Proof: It can be seen that $P_c(d+1) \leq P_c(d)$ since $P_c(d)$ can be embedded in $P_c(d+1)$. Thus, if percolation occurs in a smaller dimension, then it will certainly occur in a larger dimension. Strict inequality is quite hard to show, and thus it is not presented.

Theorem 1.13: $0 < p_c < 1$ for $\mathbb{L}^d$, $d \geq 2$.

Proof: Let's start with $d = 2$ and show that $0 < p_c$. The chance that a vertex is connected to another vertex with path length n can be no more than $4p(3p)^{n-1}$. So for it to belong to an infinite cluster, we let $n \to \infty$. We can see that for any $0 \leq p < 1/3$ the probability goes to 0. For $d > 2$, we can replace the '4' with $2d$ and the '3' with $2d - 1$.

To show $p_c < 1$, assume p is close to one. In the first quadrant, divide up the vertices into 2×2 squares and their edges to the right and above each vertex of these squares. Let us call the square vertex set 'open' if 3 out of 4 vertices have its right and upper edges open (which happens with $p < 1$). An 'open' square vertex set is connected to both the square vertex sets to the right and above.

Next divide up our square vertex sets into 2×2 squares (let's call these blocks). If 3 of the 4 square vertex sets are 'open', then there is an edge connection between the block and the blocks to the right and above. We can keep repeating this

pattern, each step needing only 3 of the 4 units to continue a connection. If this happens, then there is an infinite path from the origin.

Let $f(p)$ be the chance that at least 3 of 4 vertices are open. Then

$$f(p) = p^4 + 4p^3(1-p) = 4p^3 - 3p^4$$
$$f(p) - p = 4p^3 - 3p^4 - p \text{ and } f(1) - (1) = 0$$
$$\tfrac{d}{dp}(f(p) - p) = 12p^2 - 12p^3 - 1, \text{ which approaches } -1 \text{ as } p \text{ approaches } 1.$$

Since $f(1) - (1) = 0$, and $\frac{d}{dp}(f(p) - p) < 0$, then $f(p) > p$ or $f(p) - p > 0$ for p close to one.

So then for p strictly less than one, we have $p < f(p) < f(f(p)) < \ldots$. So $\frac{d^n}{dp^n}(f(p) - p)$ approaches 1 as $n \to \infty$ for p only 'close to 1'. So surely we have an infinite cluster for some $p < 1$. Thus $p_c(\mathbb{L}^2) < 1$.

Using Theorem 1.12 to quickly see that for $d > 2$, $p_c(\mathbb{L}^d) < 1$. $\square$

Theorem 1.14: For any two dimensional lattice $\mathfrak{L}$, $p_c(\mathfrak{L}) + p_c(\mathfrak{L}_d) = 1$

The statement is equivalent to $p > p_c(\mathfrak{L})$ if and only if $1 - p < p_c(\mathfrak{L}_d)$. Proof of either statement is quite complicated, so instead we give an intuitive argument. If $p > p_c$, then there is likely an infinite cluster which is unique (proof not given). This cluster extends throughout our lattice, and thus any closed clusters are guaranteed to be finite. The probability of the closed-edge dual (having probability $1 - p$) would not breach the critical probability. See [10] and [6] for proof.

Corollary 1.15: The critical probability of $\mathbb{L}^2 = 1/2$.

Proof: This comes directly from theorem 1.13 and noticing that the dual of $\mathbb{L}^2$ is itself (shifted). $\square$

What happens when $p = 1/2$? In $\mathbb{L}^2$, suppose at p_c we get an infinite cluster. The cluster, being unique, spans throughout the entire lattice. This would mean that clusters of the dual must be finite. The dual (also $\mathbb{L}^2$) at the critical probability then does not have an infinite cluster and contradicts our assumption. Thus it must be the case that $\theta(\frac{1}{2}) = 0$ in $\mathbb{L}^2$.

Theorem 1.16: $p_c(\text{triangle lattice}) < p_c(\mathbb{L}^2) < p_c(\text{hexagonal lattice})$.

Proof: If we rearrange the triangle lattice (Figure 1.16), we can see that the square lattice is a subset of the triangular lattice (missing the diagonal edges). So obviously $p_c(\text{triangle lattice}) \leq p_c(\mathbb{L}^2)$.

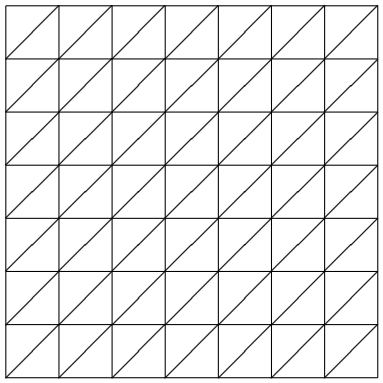


Figure 1.17: rearrangement of the triangle lattice.

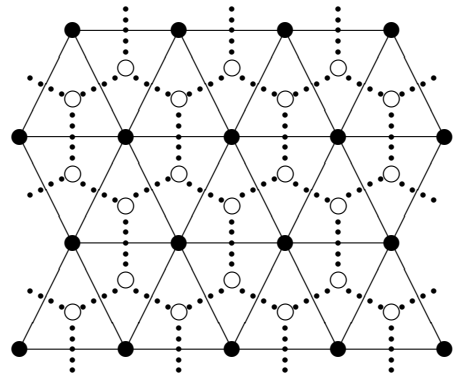


Figure 1.18: The triangle lattice and its dual (the honeycomb).

Strict inequality is harder to show. To do this, we look at the dual of the triangular lattice, the honeycomb (hexagonal) lattice, along with the theorem $p_c(\mathfrak{L}) + p_c(\mathfrak{L}_d) = 1$.

Choose the vertex subset V_2 such that for some $v' \in G = (hex)$, $V_2 = \{v : v = v'$ or the length of the path from v to v' is even$\}$. The triangle lattice 'appears' in the hexagonal lattice using the vertices of V_2. Every two adjacent edges in the honeycomb corresponds to an edge in the triangle lattice (figure 1.18).

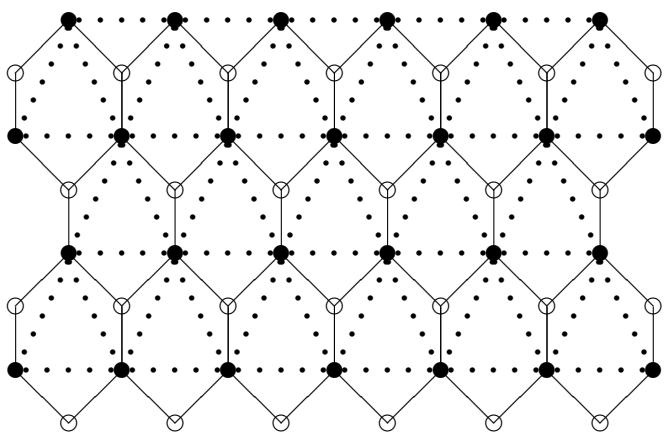


Figure 1.19: The relation between honeycomb and triangle latices.

Of course if the pair of corresponding edges did not overlap with another edge, our problem would be easy. However that is not the case, so we analyze one of our honeycomb 'triangles' by looking at the 'edge' probability (see figure 1.19).

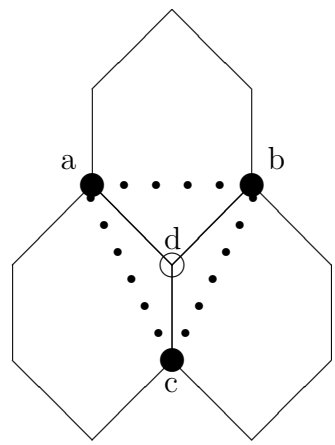


Figure 1.20: A view of one triangle in the honeycomb.

The dotted edges triangle will be open only when the the two edges sharing a vertex and the center are open in the honeycomb. For example, $\{a, b\}$ is open if and only if $\{a, d\}$ and $\{b, d\}$ are open. Furthermore, we know that p_c(triangle lattice) $\leq p_c(\mathbb{L}^2 = 1/2)$, so for $q = 1 - p$, $p \leq q$.

	triangle lattice		triangle lattice in the honeycomb
3 bonds	p^3	$=$	p^3
2 bonds	$3p^2q$	$>$	0
1 bond	$3pq^2$	$>$	$3p^2q$
0 bonds	q^3	$<$	$3pq^2 + q^3 = q^2(3p + q)$

We can clearly see that the honeycomb lattice is strictly going to produce less open triangle bonds than the triangle lattice. Thus the critical probability of the honeycomb lattice is (strictly) higher than the triangular lattice.

Using the inequality above, p_c(triangle lattice) + p_c(triangle lattice dual) = p_c(triangle lattice) + p_c(honeycomb lattice) = 1, and the inequality p_c(triangle lattice) $\leq p_c(\mathbb{L}^2$, we get our desired outcome of

$$p_c(\text{triangle lattice}) < p_c(\mathbb{L}^2) < p_c(\text{hexagonal lattice})\square.$$

Chapter 2: Introduction to Determinate Percolation

Let G be a subset of the graph $\mathbb{L}^2$, and $n_1, n_2 \in \mathbb{Z}^+$ such that the vertices of G, $G(V)$, is a box of vertices with n_1 columns and n_2 rows. Let the lower left vertex be the origin so that $G(V) = B(0, (n_1 - 1, n_2 - 1))$. Let the edge set, $G(E)$, of G be

$$\begin{aligned} E = &\{\{(x,a),(x+1,a)\} \text{ for all } 0 \le x \le n_1 - 1,\ 0 \le a \le n_2,\ a, x \in \mathbb{Z}\} \\ &\bigcup \{\{(b,y),(b,y+1)\} \text{ for all } 0 \le y \le n_2 - 1,\ 0 \le b \le n_1,\ b, y \in \mathbb{Z}\} \end{aligned}$$

Note that G does not contain all the vertices of all it's edges. The top and right sides of the box have *hanging edges*, that is, edges that only have one vertex. A graph with hanging edge components is called a *graph fragment*. A box with a corner on the origin, on the positive axes, with this particular addition of hanging edges we will denote as a *box fragment*, B_{frag}, or $B_{frag}(x)$ where x is the vertex furthest from the origin. Note that B_{frag} still includes the hanging edges, so in $B_{frag}(0,0)$, $n_1 = n_2 = 1$.

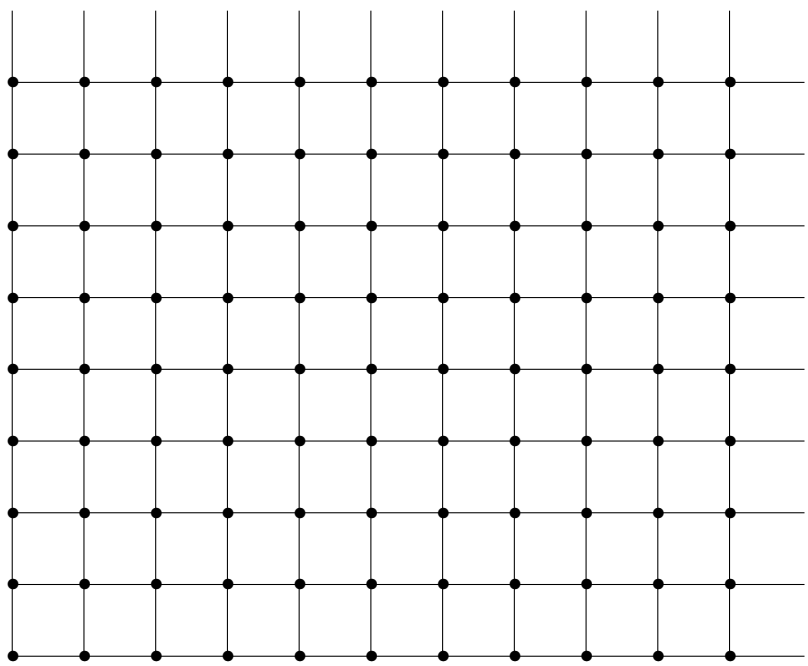


Figure 2.1

The graph fragment in figure 2.1 is a subset of the square lattice ($n_1 = 11, n_2 = 9$). We can extend the box, edges, and hanging edges in any dimension, but for this thesis we will mostly be analyzing the two dimensional box fragment.

Let the edges in B_{frag} be open or closed as normal with probability p and $1-p$. This configuration of B_{frag} is called the *determinate graph fragment*, denoted by G_{det}. We write G^d_{det} when we need to specify which dimension(s) of $\mathbb{L}$ we are considering.

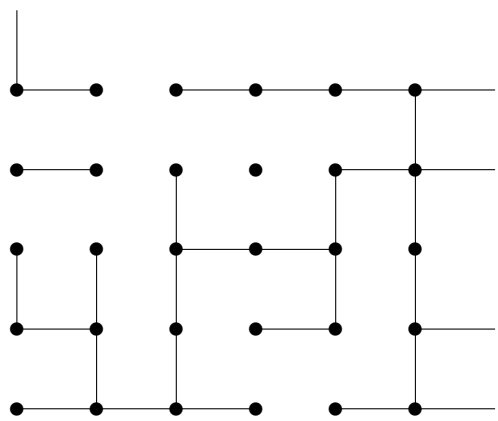


Figure 2.2

Figure 2.2 is the graph $G_{det}(\omega)$, which removes all edges that are closed. In the graph, edges were open with probability $p = 1/2$.

Now let us create isomorphic copies, called *tiles*, of the determinate graph fragment and translate them so that the origin of the determinate graph lies on a vertex in the form of (kn_1, jn_2), $\forall k, j \in \mathbb{Z}$. Therefore, for every $v \in G_{det}$, $v + (kn_1, jn_2)$ is a copy of that vertex. Likewise, if $\{v, v'\}$ is an edge in G_{det}, then $\{v + (kn_1, jn_2), v' + (kn_1, jn_2)\}$ is also an edge in $\mathbb{L}^2$. The configuration of G_{det} is also copied. So if $\{v, v'\}$ is open (closed), then all edges in the form of $\{v + (kn_1, jn_2), v' + (kn_1, jn_2)\}$ are open (closed). The process of covering a infinite space with a set of tiles will be referred to as *tiling*. Lets call G_{tile} the tiling of $\mathbb{L}^2$ by G_{det}. Figure 2.3 is a tiling of $\mathbb{L}^2$ using the G_{det} in Figure 2.2.

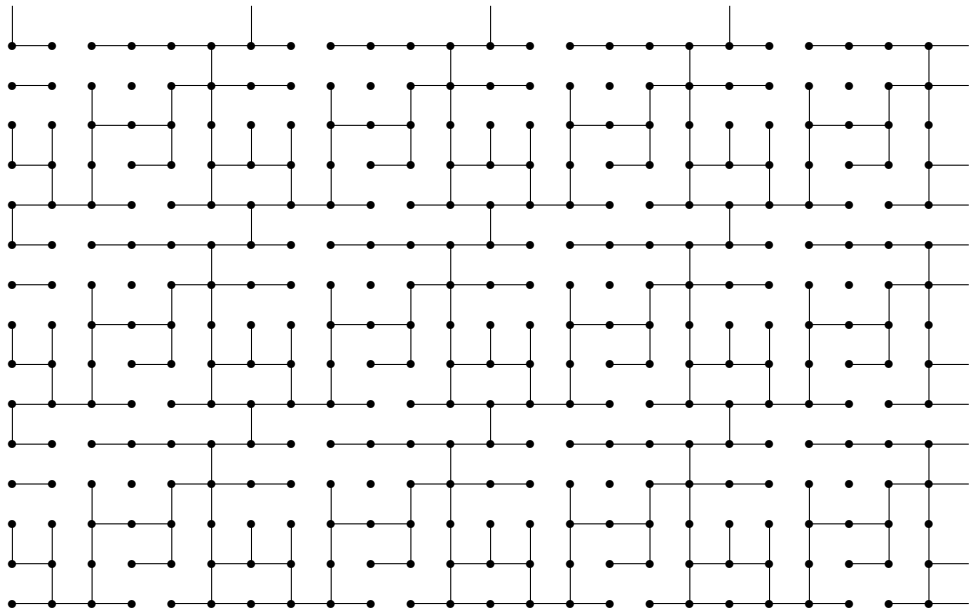


Figure 2.3

Vertices in the form $v + (kn_1, jn_2)$ $\forall k, j \in \mathbb{Z}$ are *similar vertices*. Edges in the form of $e = \{v + (kn_1, jn_2), v' + (kn_1, jn_2)\}$ $\forall k, j \in \mathbb{Z}$ are *similar edges*. And likely, *similar paths* are paths in the form of $P_1 = e_1, e_2, ..., e_n$ and $P_2 = (e_1, e_2, ..., e_n) + (kn_1, jn_2)$. Two clusters are *similar clusters* if all vertices of one cluster are similar vertices of the other.

For any tile T, an *adjacent tile*, U, is a tile that connects, or is connected to, T by one or more hanging edges.

Tile Distance is the shortest number of tiles crossed by a shortest path from one vertex to another. To determine tile distance between two vertices, v_0 and v_1, find their vertices, w_0, w_1 similar to the origin, located on the same tile. w_0 and w_1 should be of the form (k_0n_1, j_0n_2), and (k_1n_1, j_1n_2). Their tile distance is $|k_1 - k_0| + |j_1 - j_0|$.

So when does percolation occur? We know it occurs when for some vertex $v \in \mathbb{L}^d$ there exists an infinite cluster. Due to the repeating pattern, G_{det} *determines* whether or not the lattice, G_{tile}, has an infinite cluster, therefore, we shall call this type of percolation *determinate percolation*. There are numerous properties which are unique to determinate percolation, for example, if $v \in G_{tile}$ is in an infinite cluster, then so is the similar vertex in G_{det}.

Lemma 2.4: Deterministic percolation occurs if and only if for some $v \in G_{det}$, there exists a non-zero open path from v to a similar vertex ($v_1 = v + (kn_1, jn_2)$ for some $k, j \in \mathbb{Z}$, one of which is not zero). Furthermore, if there is such a k, j, then there must be a k, j satisfying $-n_2 \leq k \leq n_2$, and $-n_1 \leq j \leq n_1$.

Proof: Part 1: $\Leftarrow$
Suppose that deterministic percolation occurs. Then in a tile (lets say G_{det}), there exists a vertex v containing an infinite path. Since each tile has only a finite number of vertices, $\forall c \in \mathbb{Z}$, $\exists$ an open path from v to w where w is c tiles away. (If the path length is $\geq cn_1n_2$, then there are not enough vertices for the path to be contained in fewer than c tiles.) Since a hanging edge is used to connect two tiles together, we just need to make $c > n_1 + n_2$ so that a similar hanging edge must have been crossed by the path more than once. Thus there is an non-zero open path between similar vertices.

$\Rightarrow$
Since k, j are not both zero, the path is non-trivial. Even more so, the length of the path must be $\geq kn_1 + jn_2$, since it is kn_1 horizontal units away, and likewise jn_2 vertical units away. If there is a path from v to v_1, by repetition of G_{tile} given G_{det}, there exists a path from v_1 to $v_2 = v_1 + (kn_1, jn_2) = v + (2kn_1, 2jn_2)$. Since paths are transitive, there exists a path from v to v_2, which has length $\geq 2(kn_1 + jn_2)$. Thus we can keep repeating the process, so that for any $c \in \mathbb{Z}^+$ there exists a path from v to v_c with length $\geq c(kn_1 + jn_2)$. Thus there exists a path of infinite length.

Part 2:
There are only n_1 non-similar hanging edges a path can contain to move vertically to a new tile. Thus j is restricted by n_1 by $|j| \leq n_1$. By a similar argument, $|k| \leq n_2$. $\square$

Lemma 2.5: Let C_{det} be a cluster in G_{tile}. For any finite $n_1, n_2, ..., n_d > 0$, if $0 < p < 1$ then $0 < P_p(|C_{det}| = \infty) < 1$.

Proof: Since a sequence of horizontal or vertical open paths across an entire row or column in G_{det} would result in an infinite cluster, there is at least a $p^{min(n_1,n_2,...,n_d)}$ chance that an open path exists between 2 (unique) similar vertices. So for $p > 0$, $p^{min(n_1,n_2,...,n_d)} > 0$.

Likewise, if all hanging edges of G_{det} are closed, then there can be no connection from one tile to another. Since there are only finitely many hanging edges, for $p < 1$, $P_p(|C_{det}| < \infty) > 0$. $\square$

For very small values of n_d, we can calculate for which values of p, there will be

a 50/50 chance of being an infinite cluster in G_{tile}, that is $P_p(|C_{det}| = \infty) = 1/2$. We do this by looking at all the possible configurations of B_{frag} and summing the probabilities of the G_{det} configurations that would (or would not) create an infinite cluster, setting the probability to 1/2, then solving the equation and taking the solution lying between 0 and 1.

For example, for the case $n_1 = n_2 = 1$, if any edge is open then there is an infinite cluster. So we set the probability that both edges are closed, $(1-p)^2$, equal to 1/2 and taking the unique root between 0 and 1. Doing this we get our solution of $1 - \sqrt{2}/2 \approx .293$. Below are more calculations which are left for the reader to verify for other values of n_1 and n_2 for when $P_p(|C_{det}| = \infty) = 1/2$. Note that p for $n_1, n_2 = p$ for n_2, n_1. There are $2^{2n_1n_2}$ (or $2^{2n_1n_2\ldots n_d}$)configurations of a graph, so for any remotely large n_1 and n_2 (up to n_d for larger dimensions), the actual calculation seems impossible; the $2x3$ graph had 4096 configurations! (and no, I didn't draw them all out.)

n_1	n_2	p
1	1	$1 - \sqrt{2}/2 \approx .293$
1	2	$\approx .246$
1	3	$\approx .204$
2	2	$\approx .386$
2	3	$\approx .388$

A computer simulation [1] was written specifically for finding this 'critical probability' for larger lattices. The program assigns a random number between 0 and 1 to each edge, and finds the least value path which causes an infinite cluster and reports that number. After taking a large number of samples, it calculates the median and mean. Below are estimates given by the computer program ($n_x = 0$ if the dimension $d < x$).

n_1	n_2	n_3	n_4	p
2	2	0	0	.3861
2	3	0	0	.3854
2	4	0	0	.3641
2	20	0	0	.1083
2	100	0	0	.08253
5	5	0	0	.453
5	10	0	0	.4386
10	10	0	0	.4724
25	25	0	0	.4861
50	50	0	0	.4913
100	100	0	0	.4975
2	2	2	0	.2213
5	5	5	0	.2483
10	10	10	0	.2491
50	50	50	0	.2499
2	2	2	2	.1381
5	5	5	5	.1624
10	10	10	10	.1631

Conjecture 2.6: Let $x = (n, n, ..., n)$ for $n \in \mathbb{N}$. As n approaches infinity, $P_p(|C_{det}| = \infty) = 1/2$ when $p = p_c$.

It appears to be the case for $d = 2$ where $p_c = 1/2$ and $d = 3$ where $p_c \approx .25$[5]. As G_{det} gets large, our tiling has less to do with its repeating nature and models more closely to that of the lattice. If this holds true for higher dimensions, then it seems we may have a way to generate good estimates for $\mathbb{L}^d$ by taking an n^d box and running the program for large values of n.

Corollary 2.7: Given $\mathbb{L}^d$ and $B_{frag}(x)$ for some $x \in \mathbb{Z}^d$, If $0 < p < p_c$, then $P_p(|C| = \infty)(\mathbb{L}^d) < P_p(|C| = \infty)(G_{tile})$. For $p_c < p < 1$, $P_p(|C| = \infty)(G_{tile}) < P_p(|C| = \infty)(\mathbb{L}^d)$.

Proof: This comes trivially from the previous lemma and by using the theorem $\psi(p) = 0$ if $\theta(p) = 0$, and $\psi(p) = 1$ if $\theta(p) > 0$. $\square$

Let the *cluster order* be the number of infinite clusters in G_{tile} given G_{det}. Let $||C| = \infty|$ denote the cluster order. We may use $||C_{n_1,...,n_d}| = \infty|$ to denote the cluster order for G^d_{det} with box of size $n_1 - 1 \times n_2 - 1 \times ... \times n_d - 1$.

Lemma 2.8: Given any G^2_{det}, $||C| = \infty| = 0, 1$, or ∞.

Proof: By Lemma 2.4, if there does not exist an open path to a similar vertex then the cluster order is zero. Likewise, if all edges are open then $G_{tile} = \mathbb{L}^2$ and the cluster order is one. Lastly if all horizontal lines are open and all vertical lines are closed in G_{det}, G_{tile} is a graph of infinite horizontal lines. Clearly all three cluster orders exist.

To get another finite sized cluster order, we need first establish one infinite cluster in G_{tile}. However, in the two-dimensional plane, by creating our first unique infinite cluster, the remaining spaces remaining are finite. Thus it is impossible to create another unique infinite cluster (see figure 2.9 as an example). □

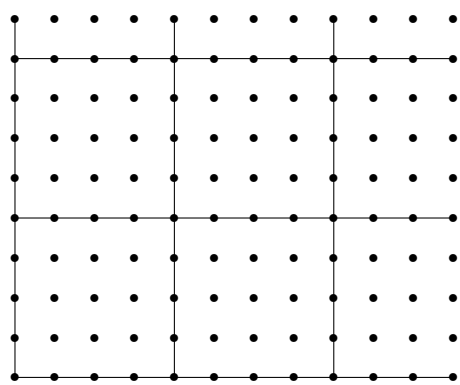


Figure 2.9

Lemma 2.10: Take G_{det} and project it onto the torus. There exists an open non-trivial loop if and only if there is an infinite cluster in G_{tile}.

Proof: $\Rightarrow$
Since on all sides of the tile we are connecting ourselves to another isomorphic tile, lets instead connect ourselves back on the other side of the tile. We can see that the top and bottom vertices with the same x coordinate, say $(kx, j(n_2) - 1)$ and $(kx, (j-1)n_2)$ for some $k, j \in \mathbb{Z}$, are connected to another tile by similar edges, and thus when projected on the torus connected by the same edge. Likewise the left and right edges are also connected in a similar manner.

$\Leftarrow$
By the Lemma 2.4, an infinite cluster exists if and only if there is an open path between two (distinct) similar vertices. This path on the torus would create our non-trivial loop. The loop would be in the form of Z $\times$ Y where Z is the number of horizontal tiles and Y is the number vertical tiles the path between the similar vertices in G_{tile} would be. □

The dual of B_{frag}, $B_{frag-dual}$ is the vertex set of B_{frag} shifted $(1/2, 1/2, ..., 1/2)$ along with the nearest neighbor edges plus hanging edges which cross the planes

determined by the axes. $B^2_{frag-dual}$ is shown in figure 2.11.

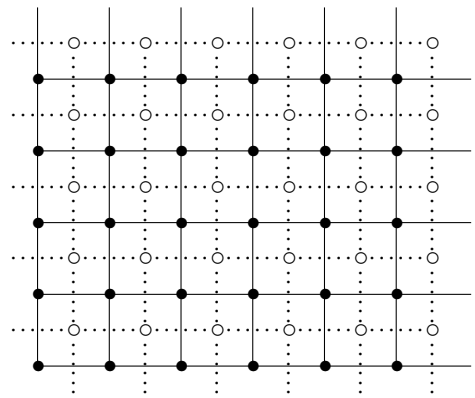


Figure 2.11

Corollary 2.12: Take G^2_{det} projected onto the torus. If there are no open loops then $||C| = \infty| = 0$, if there is an open loop in G_{det} and no closed loops in $G_{det-dual}$ then $||C| = \infty| = 1$, and if there is an open loop in G_{det} and a closed loop in $G_{det-dual}$ then $||C| = \infty| = \infty$.

Proof: It takes a moment's thought to see that by the Lemma 2.10, if there are no closed loops in the dual, then all the closed clusters in the dual are finite and thus there is one open cluster that spans throughout the lattice. If both the open and closed cluster exist, then each open infinite cluster is neighbored by a closed infinite cluster in the dual on each side, and vice versa. □

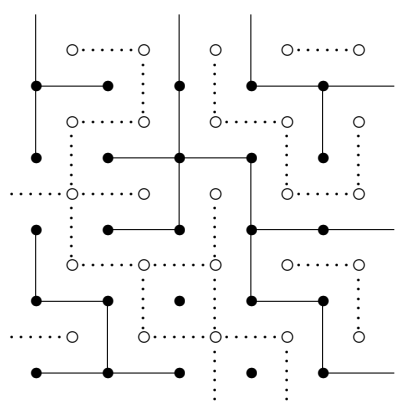


Figure 2.13

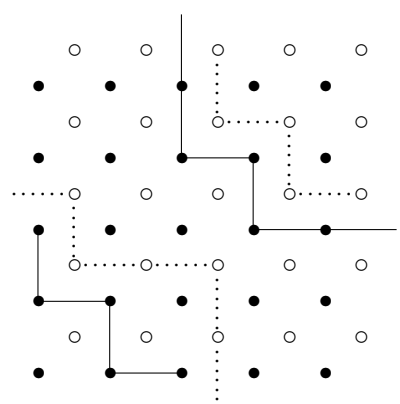


Figure 2.14

Figure 2.13 is a configuration for $n_1 = n_2 = 5$ and the closed edges of its dual. Figure 2.14 shows the open loop in G_{det} and the closed loop in $G_{det-dual}$. Figure 2.15 shows the tiling of G_{det} along with the closed loop in the dual that separate the infinite clusters in G_{tile}.

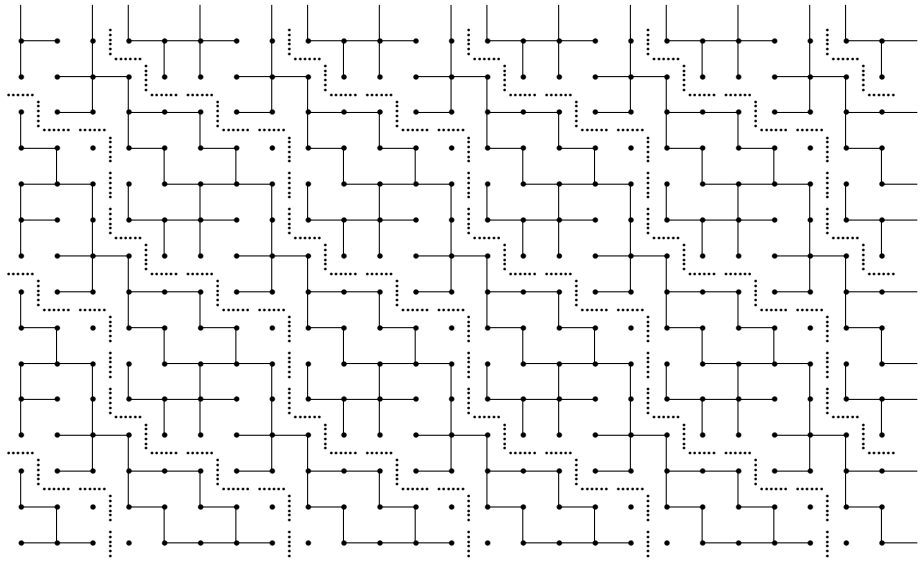


Figure 2.15

Let $P_p(||C| = \infty| = n)$ be the probability that $||C| = \infty| = n$. We could also use inequalities such as $P_p(0 < ||C| = \infty| < \infty)$, which asks for the probability that there will be any finite number of infinite clusters, but at least 1.

We can create a few obvious equalities about cluster orders, such as:

$$P_p(0 \leq ||C| = \infty| \leq \infty) = P_p(||C| = \infty| = \infty) + \Sigma_{n=0}^{\infty} P_p(||C| = \infty| = n) = 1$$
$$P_{p=1}(||Cn_1, ..., n_d| = \infty| = 1) = 1$$
$$P_{p=0}(||Cn_1, ..., n_d| = \infty| = 0) = 1$$

Lemma 2.16: The cluster order created by a configuration of $B_{frag}(x)^d$ for $d > 2$ can be any non-negative finite number, f, or ∞.

We have seen cases in which the cluster order is $0, 1$ or ∞. For any natural number n, there is a configuration in which we can create a G_{tile} with n infinite clusters: In $B_{frag}(x)$ where $x = n^d$ for $d > 2$ let the open edge set be:

$$\{(y, 0, 0, ..., 0), (y + 1, 0, 0, ..., 0)\}$$
$$\{(0, y, 0, ..., 0), (0, y + 1, 0, ..., 0)\}$$
$$\vdots$$
$$\{(0, 0, ..., 0, y), (0, 0, ..., 0, y + 1)\}$$

$$\{(y, 1, 1, ..., 1), (y + 1, 1, 1, ..., 1)\}$$
$$\{(1, y, 1, ..., 1), (1, y + 1, 1, ..., 1)\}$$
$$\vdots$$

$$\{(1,1,...,1,y),(1,1,...,1,y+1)\}$$

$$\vdots$$

$$\{(y,n,n,...,n),(y+1,n,n,...,n)\}$$
$$\{(n,y,n,...,1),(n,y+1,n,...,n)\}$$
$$\vdots$$
$$\{(n,n,...,n,y),(n,n,...,n,y+1)\}$$

$\forall y \in \mathbb{Z}$ such that $0 \leq y < n$.

This graph creates n unique interlocking (infinite) lattices. So $\forall n \in \mathbb{N}$, we can create a G_{tile} with $||C| = \infty| = n$. An example for $d = 3n = 2$ is drawn in figure 2.17. $\square$

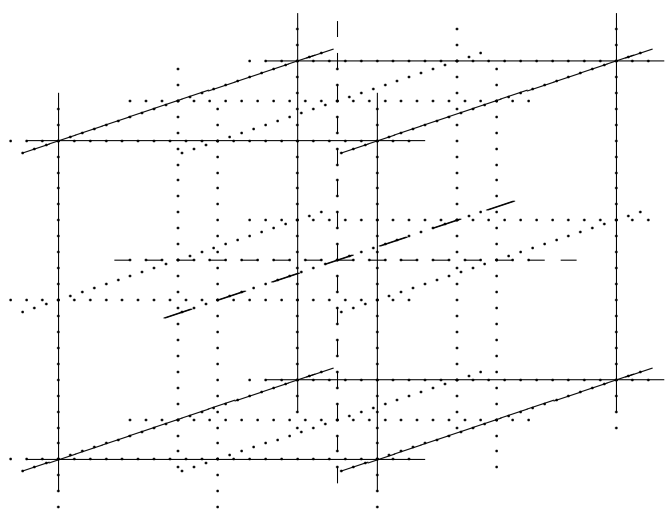


Figure 2.17: The solid lines are open edges belonging to one infinite cluster, the dashed lines are open edges belonging to the other infinite cluster, and the dotted lines are closed edges.

Conjecture 2.18: If the number of infinite clusters is a finite number, f, then $0 \leq f \leq \min\{n_1, n_2, ..., n_d\}$.

The conjecture states that the setup above (or a similar one) is the most efficient way to place n unique infinite clusters in a lattice. If we shrink the above box size by one in any dimension, then my estimate is that we will only be able to create a G_{det} with a cluster order of $n-1$.

Lemma 2.19: Take G^2_{det} and project it onto the torus. If there is a point $v \in G_{det}$, such that there are two non-trivial, simple loops at v in the form of $Z \times Y$ and $X \times W$ such that (Z, Y) and (X, W) are relatively prime, $Z, Y, X, W \in \mathbb{Z}$, and $Z \times Y \neq X \times W$, then there there is only one infinite cluster; otherwise there are an infinite number of them.

Proof: Lets start with supposing that for every point, there is at most one loop (or set of loops with the same homotopy class). Then a vertex v with a $Z \times Y$ loop is connected to it's similar vertices in the form of $v + a(Z \cdot n_1, Y \cdot n_2) \forall a \in \mathbb{Z}$. Since there are no other loops, it cannot however connect to any other similar vertices, and thus the cluster order in G_{tile} will be ∞.

Suppose now that v has two (non homotopic) such simple loops. v is now connected to a similar vertex not in the form of $v + a(Z \cdot n_1, Y \cdot n_2)$. Therefore the infinite cluster containing v is now connected to it's neighboring infinite similar cluster, and thus all infinite clusters are now connected. $\square$

Conjecture 2.20: Given $B_{frag}(x)$ there exists a p_d where for all $p < p_d$, $P_p(||C| = \infty| = \infty) > P_p(||C| = \infty| = n)$ for some $n \in \mathbb{Z}^+$, and for all $p > p_d$, $P_p(||C| = \infty| = n) > P_p(||C| = \infty| = \infty)$ for some $n \in \mathbb{Z}^+$.

It takes a moments thought to realize that for lower values of p, if there are any infinite clusters, then there would be an infinite amount, since it is likely there will only be one similar path between two similar vertices. For large p, there are only going to be a few edges missing, so it is very likely that G_{tile} will be connected as one graph. The hypothesis is that there is a critical deterministic probability, p_d, that divides the finite cluster orders from an infinite cluster order. It should be at least the case for two dimensions, where there are only 3 cases of cluster order. It might not exist for higher dimensions, since it is possible to have cluster order one, then by opening more edges switch the cluster order back to infinite.

References

[1] Beebe, Travis S. April 14, 2007. Computer program source code. "Determinate Percolation Simulator." Lattice.h, main.cpp.

[2] Burton, Robert; Keane, Michael. Density and uniqueness in Percolation, Comm. Math. Phys. 121, no. 3, 501–505 (1989).

[3] Gandolfi, A.; Keane, M.; Newman, C. M. Uniqueness of the infinite component in a random graph with applications to percolation and spin glasses. Probability Theory Related Fields 92, no. 4, 511–527 (1992).

[4] Gandolfi, A.; Keane, M.; Russo, L. On the uniqueness of the infinite occupied cluster in dependent two-dimensional site percolation. Annals of Probability 16, no. 3, 1147–1157 (1988).

[5] Grimmett, Geoffrey. Percolation. Second edition. Grundlehren der Mathematischen Wissenschaften [Fundamental Principles of Mathematical Sciences], 321. Springer-Verlag, Berlin, 1999.

[6] Kesten, Harry. The critical probability of bond percolation on the square lattice equals 1/2. Comm. Math. Phys. 74, no. 1, 41–59 (1980).

[7] Kolmogorov, A. N. Foundations of the theory of probability. Translation edited by Nathan Morrison, with an added bibliography by A. T. Bharucha-Reid. Chelsea Publishing Co., New York, 1956.

[8] Seymour, P.D., Welsh, D.J.A. Percolation probabilities on the square lattice. Ann. Discrete Math. 3, 227–245 (1978).

[9] G. Slade. Probabilistic Models of Critical Phenomena. This is an essay intended for a general mathematical audience, from The Princeton Companion to Mathematics, edited by Timothy Gowers. Scheduled for publication in 2007. Reprinted by permission of Princeton University Press. Posted November 22, 2004.

[10] Smythe, R. T.; Wierman, John C. First-passage percolation on the square lattice. Lecture Notes in Mathematics, 671. Springer, Berlin, 1978.

[11] Stauffer, Dietrich. Introduction to percolation theory. Taylor & Francis, Ltd., London, 1985.

Appendix I

main.cpp:

```
/*****************************************************************************

Determinate Percolation Simulator Created by Travis S. Beebe
tbeebe(AT)alumni.unh.edu Updated April 14, 2007

The above email address can be used if you have any questions
regarding this code or the program it represents. I will try to
respond in a timely manner, but this is not guaranteed!

This program is freeware. This source can be used or modified freely
for non-commercial purpose, as long as proper credit is given. Other
use requires permission from the creator.

*****************************************************************************/
// This program was originally created for David Darling, to be used for
// simulation, in his thesis on Deterministic Percolation.
//
// This program accept input from the user to genereate a lattice of vectors
// and edges. Dimensions data is accepted by the user. After user input,
// several samples of lattices are generated, with pseudo-random values
// assigned to each edge. For each sample, a bond percolation point p is found
// where the lattice contains an infinite path for each edge value >= p. Sample
// results are averaged to predict a critical probability for the deterministic
// percolation of lattices witl the given dimensions.
//
// Results are saved to percOut-stat.txt. Details for generated lattices can
// optionally be saved to percOutput*.txt. Output contains enough information to
// describe the complete lattice: the p-value found and number of dimensions,
// and descriptions of each vertex and edge. Vertices are listed by arbitrary id
// and the edges they connect to. Each edge is identified as a pair of the
// vertices it connects, along with its p-value and some flags, "b" identifies
// a border edge, "-" is an enabled edge for the p found, and "#" is an edge in
// the path found. Optionally, a picture of the lattice tile follows (only for
// two-dimensional lattices).
//
// The program has a hard-coded soft cap of 500000 unique vertices (parameters
// that would generate more vertices are not accepted).

// not optimized for multi-processor systems


#include <cstdlib> #include <iostream> #include <fstream> #include
<ctime> #include <cmath>

#include "Lattice.h"

using namespace std;
```

```
// maximum vertex limit for lattice tile - can be changed at programmer's
//  discretion
const int VERTICES_MAX = 500000 ;

void saveToFile( Lattice * l, int p, int x ) {
    ofstream out ;
    char filename[32] = "percOut " ;
    char i[6] ;
    itoa ( x, i, 10 );
    strcat( filename, i ) ;
    strcat( filename, ".txt" ) ;
    out.open( filename, ios_base::binary) ;
    if(out){
        out << *l << "\r\n" ;
        if ( p != 0 )
        l -> print2dPath( out ) ;
        out.close() ;
    }else{
        out.close() ;
        cout << "*" << endl ;
    }
}

int main(int argc, char *argv[])
// contains all interactive code, fires off generator and simulator found
//  in Latttice class
{
    // initialize pseudo-random number generator
    srand(time(0));

    cout << "*** Percolation Simulation ***" << endl ;
    cout << "Created by Travis Beebe" << endl ;
    cout << endl << endl ;

    cout << "  Enter number of samples (>0): " ;
    unsigned long int s ;
    cin >> s ;
    // valid range
    if ( s < 0 ) s = 1 ;

    cout << "  Enter number of dimensions for lattice (>1): d = " ;
    short int d ;
    cin >> d ;
    // valid range
    if ( d > 255 ) d = 255 ;
    if ( d < 2 ) d = 2 ;

    short int p = 0 ;
    cout << "  Output samples to text files (y/n)? " ;
    char save ;
    cin >> save ;
```

```
if ( save != 'y' && save != 'Y' ) save = 'n' ;
else
{
    if ( d == 2 )
    {
        cout << "  Enter size of picture output (1-10, 0 for no pic): " ;
        cin >> p ;
        if ( p > 10 ) p = 10 ;
        if ( p < 1 && p > 0 ) p = 1 ;
        if ( p < 0 ) p = 0 ;
    }
}
cout << endl ;

long vertices = 1 ;
int * dim = new int[d] ;
for(short int n=0;n<d;n++)
{
     cout << "  Enter size of dimension " << n+1 << " (>1): " ;
     cin >> dim[n] ;
     // valid range
     if ( dim[n] > 999999 ) dim[n] = 999999 ;
     if ( dim[n] < 2 ) dim[n] = 2 ;
     vertices *= dim[n] ;
     cin.ignore() ;
}

double pSum = 0.0000 ;
double * samp = new double[s] ;
for(int i=0; i<s; i++) samp[i] = 2.0000 ;
if ( vertices > VERTICES_MAX )
{
    cout << endl << "*** Warning: Too many vertices (" << vertices ;
    cout << "), aborting..." << endl ;
} else {

    Lattice * l ;
    cout << endl << "Sampling lattices " ;
    for(int x=0; x<s; x++)
    {
        if ( s > 100 && x % ( s / 10 ) == 0 ) cout << x / ( s / 10 ) ;
        else cout << "." ;
        cout.flush() ;
        // generate lattice
        l = new Lattice( int(d), dim, p ) ;
        // find p-value
        double pVal = l -> findPath() ;
        // store sum of p-values to use for average calc
        pSum += pVal ;
        // insertion sort p-value to use for median calc
        int i = x ;
        while( i > 0 && samp[i-1] >= pVal )
```

```
            {
                samp[i] = samp[i-1] ;
                i-- ;
            }
            samp[i] = pVal ;
            // save lattice to file?
            if ( save != 'n' ) saveToFile( l, p, x ) ;
            delete l ;
        }
        cout << "done." << endl ;
    }
    cout << endl ;
    cout.precision( 4 ) ;
    double median = samp[s/2] ;
    if ( s % 2 == 0 ) median = ( samp[s/2] + samp[s/2-1] ) / 2 ;
    cout << endl << "##### AVERAGE p = " << pSum / s << endl ;
    cout << "##### MEDIAN p = " << median << endl << endl ;

    // save stats to file
    ofstream out ;
    out.open( "percOut-stat.txt", ios_base::binary) ;
    if(out){
        out << "##### AVERAGE p = " << pSum / s << "\r\n" ;
        out << "##### MEDIAN p = " << median << "\r\n\r\n" ;
        int i = 0 ;
        while( i < s )
        {
            out << samp[i] << "\r\n" ;
            i++ ;
        }
        out.close() ;
    }else{
        out.close() ;
        cout << "*" << endl ;
    }

    system("PAUSE");
    return EXIT_SUCCESS;
}
```

Lattice.h:

```
#include <iostream> using namespace ::std; #include <ctime>

#ifndef LATTICE_H #define LATTICE_H

////////////////////////////////////////////////////////////////////////////////
// declarations
```

```
struct Edge ;

struct Vertex
// this structure represents vertex "v"
{
        // I/O
        //friend istream& operator >>(istream&, Vertex&);
        friend ostream& operator <<(ostream&, const Vertex&);

        long id ; // unique id number
        long numEdges ; // number of adjacent edges
        Edge ** adjEdges ; // array of pointers to adjacent edges
} ;

struct Edge
// this structure represents edge "e"
{
        // I/O
        //friend istream& operator >>(istream&, Edge&);
        friend ostream& operator <<(ostream&, const Edge&);

        double value ; // "p" value
        Vertex * headVertex ; // adjacent vertices
        Vertex * tailVertex ;

        bool enabled ; // true if edge is "on"
        bool searched ; // marker to track pathfinding
        bool curPath ; // marker to track current path in pathfinding
        bool border ; // true if this edge leads to a new "tile"
        int borderDim ; // dimension the border edge occupies
        Edge * next ; // linked list for traversal of all edges (destructor)
} ;

class Lattice
// this class represents a lattice of vertices and edges with d dimentions for
// each unique tile
{
        // I/O
        friend istream& operator >>(istream&, Lattice&);
        friend ostream& operator <<(ostream&, const Lattice&);

    public:

        // constructor - pass in the size of each dimension (array of sizes)
        Lattice( int d, int * dArray, int pp ) ;

        // destructor - will take care of vertices and edges
        ~Lattice() ;

        // sets the lattice to a certain state for a given p value
        void setState( double p ) ;
```

```
        // seraches for a valid "infinte path" in the lattice
        // uses recursive depth-first search on enabled edges
        // lowest valued disabled edge is toggled on bettwen each search
        double findPath() ;

        // prints picture of a 2d lattice tile
        // pre: lattice must be 2-dimensional
        void Lattice::print2dPath( ostream & out ) ;

    private:

        int dimensions ; // number of dimensions
        int * dSize ; // array of dimension sizes - used for pretty print (d=2)
        int ppSize ; // char size of edges in pretty print
        long totalVertices ;
        Vertex * v ; // G(V)
        Edge * e ; // G(E)
        Edge * pEdge ; // enabled edge with highest p-value
        bool pathFound ; // enabled if dfs finds path

        // depth-first search helper function
        bool dfs( Vertex * v, Vertex * dest, long count ) ;
        int * tile ; // use in helper function to exclude non-infinite cycles
} ;

////////////////////////////////////////////////////////////////////////////
// I/O

ostream & operator <<(ostream & out, const Vertex & v ) {
    out << " " ;
    out << v.id ;
    out << " (" ;
    // output adjacent edges
    int a = 0 ;
    while( a < v.numEdges && v.adjEdges[a] != 0 )
    {
        if ( a != 0 ) out << "," ;
        Edge e = *(v.adjEdges[a++]) ;
        out << "{" << e.headVertex -> id << "," << e.tailVertex -> id << "}" ;
    }
     out << ")" ;
    return out ;
}

ostream & operator <<(ostream & out, const Edge & e ) {
    out.precision( 4 ) ;
    if ( e.curPath ) out << "#"; else out << " " ;
    if ( e.enabled ) out << "-"; else out << " " ;
    if ( e.border ) out << "b" ; else out << " " ;
    out << "{" << e.headVertex -> id << "," << e.tailVertex -> id << "} = " ;
    out << e.value ;
    return out ;
```

```
}

ostream & operator <<(ostream & out, const Lattice & l ) {
    out << "d = " << l.dimensions << "\r\n" ;
    if ( l.pEdge == 0 )
        out << "p = nil" << "\r\n" ;
    else
        out << "p = " << *(l.pEdge) << "\r\n" ;
    out << "\r\n" << "Vertices:" << "\r\n" ;
    for(int vert=0;vert<l.totalVertices;vert++)
        out << l.v[vert] << "\r\n" ;
    out << "\r\n" << "Edges:" << "\r\n" ;
    // traverse edges
    Edge * ep = l.e ;
    while( ep )
    {
        out << *ep ;
        if ( l.pEdge == ep ) out << " p" ;
        out << "\r\n" ;
        ep = ep -> next ;
    }
    // out << l.e[edg] << endl ;
    return out ;
}

// prints picture of a 2d lattice
// pre: lattice must be 2-dimensional
void Lattice::print2dPath( ostream & out ) {
    if( dimensions != 2 )
    {
        out << "*** Picture only displays for d = 2 ***\r\n" ;
        return ;
    }
    Edge * ex = e ;
    Edge * ey = e ;
    for(int ei=0;ei<totalVertices;ei++)
        ey = ey -> next ;
    out << "p = " << *pEdge << "\r\n\r\n" ;
    for(int y=0;y<dSize[1];y++)
    {
        for(int t=0;t<ppSize+1;t++)
        {
            Edge * ex2 = ex ;
            Edge * ey2 = ey ;
            for(int x=0;x<dSize[0];x++)
            {
                int i = y * dSize[0] + x ;
                if ( t == 0 )
                {
                    out << "O" ;
                    char cEdge = ' ' ;
                    if ( ex2 == pEdge ) cEdge = 'p' ;
```

```
                    else if ( ex2 -> curPath ) cEdge = '#' ;
                    else if ( ex2 -> enabled ) cEdge = '-' ;
                    else cEdge = ' ' ;
                    for(int ct=0;ct<ppSize;ct++) out << cEdge ;
                }
                else if ( t == 1 )
                {
                    if ( ey2 == pEdge ) out << 'p' ;
                    else if ( ey2 -> curPath ) out << '#' ;
                    else if ( ey2 -> enabled ) out << '|' ;
                    else out << ' ' ;
                    out.setf( ios_base::left,ios_base::adjustfield ) ;
                    out.width( ppSize );
                    if (ppSize >= int( log( totalVertices - 1 )/log( 10 ) + 1 ))
                      out << v[i].id ;
                    else out << " " ;
                }
                else
                {
                    if ( ey2 == pEdge ) out << 'p' ;
                    else if ( ey2 -> curPath ) out << '#' ;
                    else if ( ey2 -> enabled ) out << '|' ;
                    else out << ' ' ;
                    out.width( ppSize ) ;
                    out << " " ;
                }
                ex2 = ex2 -> next ;
                ey2 = ey2 -> next ;
            }
            out << "\r\n" ;
        }
        for(int ct=0;ct<dSize[0];ct++)
        {
            ex = ex -> next ;
            ey = ey -> next ;
        }
    }
    return ;
}

////////////////////////////////////////////////////////////////////////////
// Lattice methods

// constructor - pass in the size of each dimension (array of sizes)
Lattice::Lattice( int d, int * dArray, int pp ) {
    ppSize = pp ;
    dSize = new int[d] ;
    pathFound = false ;
    totalVertices = 1 ;
    for(int dim=0;dim<d;dim++)
    {
        totalVertices *= dArray[ dim ] ;
```

```
    dSize[dim] = dArray[dim] ; // used for pretty print only
}
dimensions = d ;

// create array of vertices
v = new Vertex[ totalVertices ] ;
for(long vert=0;vert<totalVertices;vert++)
{
    v[vert].id = vert ;
    v[vert].numEdges = 2 * d ;
    v[vert].adjEdges = new Edge *[ 2 * d ] ;
    for(int a=0;a<2*d;a++)
            v[vert].adjEdges[a] = 0 ;
    //v[vert].curPath = false ;
}

// create edges
Edge * ep, * eq ;
long m = 1 ; // used for assigning vertices
long m2 = 1 ; // used for assigning border edges
long dLast = 1 ; // used for assigning border edges
for(int dim=0;dim<d;dim++)
{
    m2 *= dArray[ dim ] ;
    for(long x=0;x<totalVertices;x++)
    {
        ep = new Edge() ;

        // assign random p-trigger value
        ep -> value = ((double)rand() / ((double)(RAND_MAX)+(double)(1))) ;

        // assign head vertex
        ep -> headVertex = &v[x] ;
        // add edge to head vertex
        int xv = 0 ;
        while( v[x].adjEdges[xv] != 0 ) xv++ ;
        v[x].adjEdges[xv] = ep ;
        // assign tail vertex
        long y ;
        if ( x % m2 >= m2 - m ) y = x-m2+m ;
        else y = x+m ;
        ep -> tailVertex = &v[y] ;
        // add edge to tail vertex
        int yv = 0 ;
        while( v[y].adjEdges[yv] != 0 ) yv++ ;
        v[y].adjEdges[yv] = ep ;
        // set flags
        ep -> enabled = false ;
        ep -> curPath = false ;
        if ( x % m2 >= m2 - m )
        {
            ep -> border = true ;
```

```
                ep -> borderDim = dim ;
            }
            else ep -> border = false ;

            if ( dim == 0 && x == 0 )
                e = ep ;
            else eq -> next = ep ;
            eq = ep ;
        }
        m *= dArray[ dim ] ;
        dLast = dArray[ dim ] ;
    }
    ep -> next = 0 ;

    // set pEdge pointer
    pEdge = 0 ;
}

// destructor - will take care of vertices and edges
Lattice::~Lattice() {
    // destroy vertices
    for(long vert=0;vert<totalVertices;vert++)
        delete[] v[vert].adjEdges ;
    delete[] v ;
    // destroy edges
    // traverse linked list
    Edge * ep = e ;
    while( ep )
    {
        ep = ep -> next ;
        delete e ;
        e = ep ;
    }
    // destroy dSize
    delete[] dSize ;
}

// sets the lattice to a certain state for a given p value
// was not used much in searching algorithm, mostly for debugging
void Lattice::setState( double p ) {
    Edge * ep = e ;
    pEdge = 0 ;
    while( ep )
    {
        if ( p >= ( ep -> value ))
        {
            ep -> enabled = true ;
            if ( !pEdge || (( pEdge -> value ) < ( ep -> value )))
            pEdge = ep ;
        } else
            ep -> enabled = false ;
```

```
        ep = ep -> next ;
    }
}

// seraches for a valid "infinte path" in the lattice
// uses recursive depth-first search on enabled edges
// greedy algorithm accepts first path found, which is acceptable for this
//  problem...path may not be "optimal", and may contain loops (user can see
//  this if picture is in output)
// lowest valued disabled edge is toggled on bettwen each search
// some heuristics used to attempt reduce number of searches
double Lattice::findPath() {
    //int nextTile[ dimensions ] ;
    pathFound = false ;
    double pathFoundP = -1.0 ;
    setState( 0.0 ) ;

    // this loop doesn't need an escape clause because eventually a path
    //   should be found

    // while we haven't found the path
    long count = 0 ;
    while( pathFoundP < 0.0 && count < totalVertices * dimensions )
    {
        // enable lowest value edge that is off
        Edge * ep = e ;
        Edge * elow = e ;
        while( ep )
        {
            if (( ep -> enabled == false ) &&
                (( elow -> value ) > ( ep -> value )) ||
                (( elow -> enabled ) == true ))
                elow = ep ;
            ep = ep -> next ;
        }
        pEdge = elow ;
        pEdge -> enabled = true ;

        // clear paths traversed
        ep = e ;
        while( ep )
        {
            ep -> curPath = false ;
            ep -> searched = false ;
            ep = ep -> next ;
        }

        bool headE = false ;
        bool tailE = false ;
        // head vertex connected to another enabled edge?
        Vertex * vp = pEdge -> headVertex ;
```

```
        int veindex = 0 ;

        while((veindex < vp -> numEdges)&&((vp -> adjEdges[veindex] == pEdge) ||
              ( vp -> adjEdges[veindex] -> enabled == false )))
            veindex++ ;
        if ((veindex < vp -> numEdges)&&(vp -> adjEdges[veindex] != pEdge) &&
            ( vp -> adjEdges[veindex] -> enabled == true ))
            headE = true ;
        // tail vertex connected to another enabled edge?
        vp = pEdge -> tailVertex ;
        veindex = 0 ;

        while((veindex < vp -> numEdges)&&((vp -> adjEdges[veindex] == pEdge) ||
              ( vp -> adjEdges[veindex] -> enabled == false )))
            veindex++ ;
        if ((veindex < vp -> numEdges)&&( vp -> adjEdges[veindex] != pEdge ) &&
            ( vp -> adjEdges[veindex] -> enabled == true ))
            tailE = true ;

        // if both vertices of enabled edge connect to another enabled edge...
        if ( headE && tailE )
            // ...then perform depth-first search on head vertex
        {
            tile = new int[dimensions] ;
            for( int t=0; t<dimensions; t++)
                tile[t] = 0 ;
            if ( dfs( pEdge -> headVertex, pEdge -> headVertex, count ))
                pathFoundP = pEdge -> value ;
            delete[] tile ;
        }
        count++ ;
    }

    return pathFoundP ;
}

// search helper function
bool Lattice::dfs( Vertex * v, Vertex * dest, long count ) {
    // starting from this vertex...
    // for each adjacent edge that is enabled and not searched
    //   and while pathFound is false
    int ae = 0 ;
    bool found = false ;
    while( !found && ae < dimensions * 2 )
    {
        if ( v -> adjEdges[ae] -> enabled == true &&
             v -> adjEdges[ae] -> curPath == false &&
             v -> adjEdges[ae] -> searched == false )
        {
            Vertex * vh = v -> adjEdges[ae] -> headVertex ;
            Vertex * vt = v -> adjEdges[ae] -> tailVertex ;
            // follow that edge
```

```
            Vertex * ve = vh ;
            if ( v == vh )
                ve = vt ;
            // flag that edge
            int t = 0 ;
            v -> adjEdges[ae] -> curPath = true ;
            v -> adjEdges[ae] -> searched = true ;
            if( v -> adjEdges[ae] -> border == true )
            {
                t = v -> adjEdges[ae] -> borderDim ;
                if (( v -> id ) > ( ve -> id ))
                    tile[t] += 1 ;
                else
                    tile[t] += -1 ;
            }
            // check for tiling
            // cycle that is not infinite doesn't count
            // must cross borders a certain way
            bool tiling = false ;
            for(t=0; t<dimensions; t++)
                if ( tile[t] != 0 ) tiling = true ;
            // if the next vertex is the destination
            if ( ve == dest && tiling )
            {
                // flag path found
                return true ; }
            // else
            else
            {
                // search from next vertex
                found = dfs( ve, dest, count ) ;
                if ( !found )
                {
                    v -> adjEdges[ae] -> curPath = false ;
                    if( v -> adjEdges[ae] -> border == true )
                    {
                        t = v -> adjEdges[ae] -> borderDim ;
                        if (( v -> id ) > ( ve -> id ))
                            tile[t] -= 1 ;
                        else
                            tile[t] -= -1 ;
                    }
                }
            }
        }
        ae++ ;
    }
    return found ;
}
#endif
```