Directed Percolation in Two Dimensions

by Ken Loo & Randy Doser
1989 REU at OSU

Various percolation models have been studied in depth in recent years, by both mathematicians and physicists. The particular model we looked at was a variation of bond percolation which we will call directed percolation. We will first explain both models since many comparisons will be made between the two models.

I. Models Explained.

A. Bond Percolation.

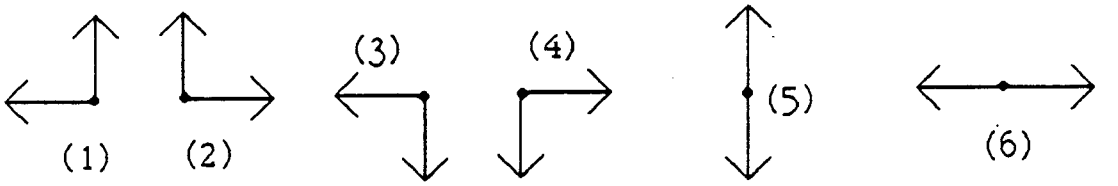
Consider the integer lattice in two dimensions. At each point x in the lattice, define a *neighbor* of x as any point y such that $|x - y| = 1$. Define also a *bond* as the edge of a graph connecting all the neighbors of $\mathbf{Z}^2$. In bond percolation each bond is assigned a probability p of being "open" and 1-p of being "closed", with choices made independently for each bond. We define a cluster containing point x as the set of points y such that $x \to y$ (i.e. y can be reached from x by only open bonds). More precisely, $\exists$ a sequence of neighbors $x=x_1, x_2, x_3, \ldots, x_{n-1}, x_n=y$ in $\mathbf{Z}^2$. such that for every $m \leq n$ $\exists$ an open bond between x_{m-1} and x_m. Percolation is said to occur if $C_0=\{x \in \mathbf{Z}^2: 0 \to x\}$(the zero cluster) then, $P(|C_0|=\infty)>0$. In other words there is a positive probability of the zero cluster containing an infinite number of points. This probability of percolation is a non-decreasing function of p, so we define a critical probability, $p_c = \inf\{p: P(|C_0|=\infty)>0\}$.

It has been shown that p_c is non-trivial and that $p_c=\frac{1}{2}$ (Kesten, 1980).

B. Directed Percolation.

Different from bond percolation in that it has no probability associated with the bonds. At each point in $\mathbf{Z}^2$ choose two of the four neighbors and declare these bonds open in a direction away from the current point. This can be thought of as a system of "one-way streets" on the lattice that allow that allow a first point to be reached from a second point but not the other way around(unless the first point chooses a directed bond lying on the

same edge). Since there are four neighbors, there are $\binom{4}{2}=6$ possible configurations at each point.

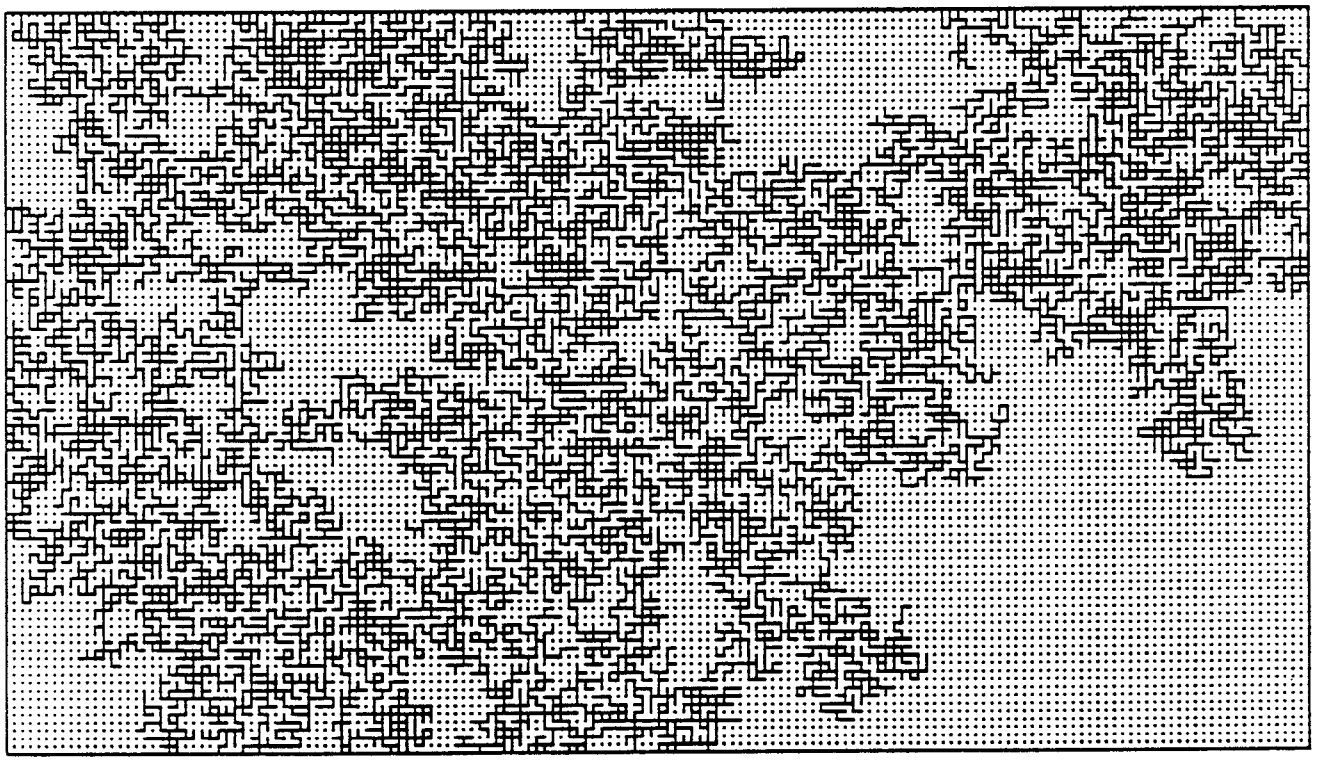


Each configuration is equally likely and so has a probability of $\frac{1}{6}$ of being chosen. The event of percolation is defined in the same way for this model as it was for the bond model; however, the question is not, "at what p_c?", but simply,"Does it percolate?"

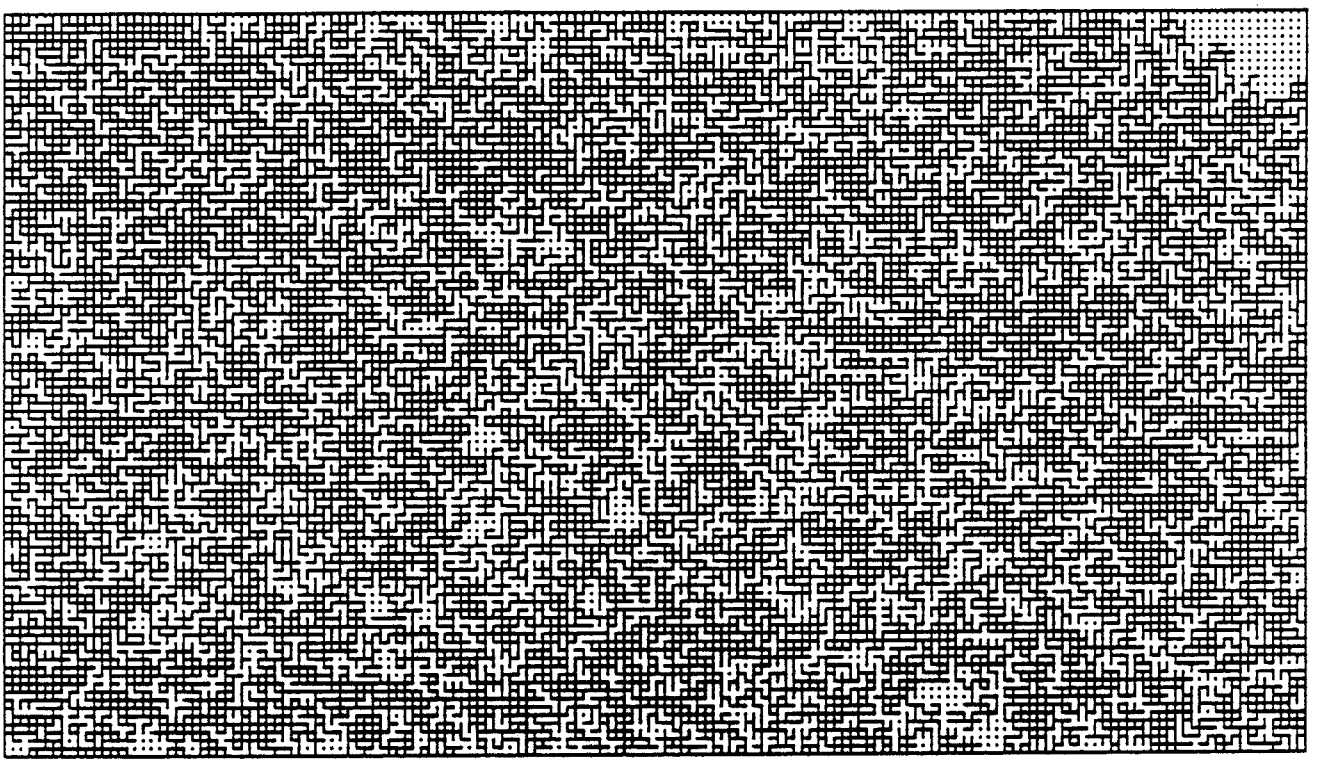
II. Computer Simulations.

A. As A Physical System.

After mucking through some proofs on bond percolation, we wrote a program to simulate both bond and directed percolation. Below is a simulation on the Macintosh of bond at p=.50 and p=.60.

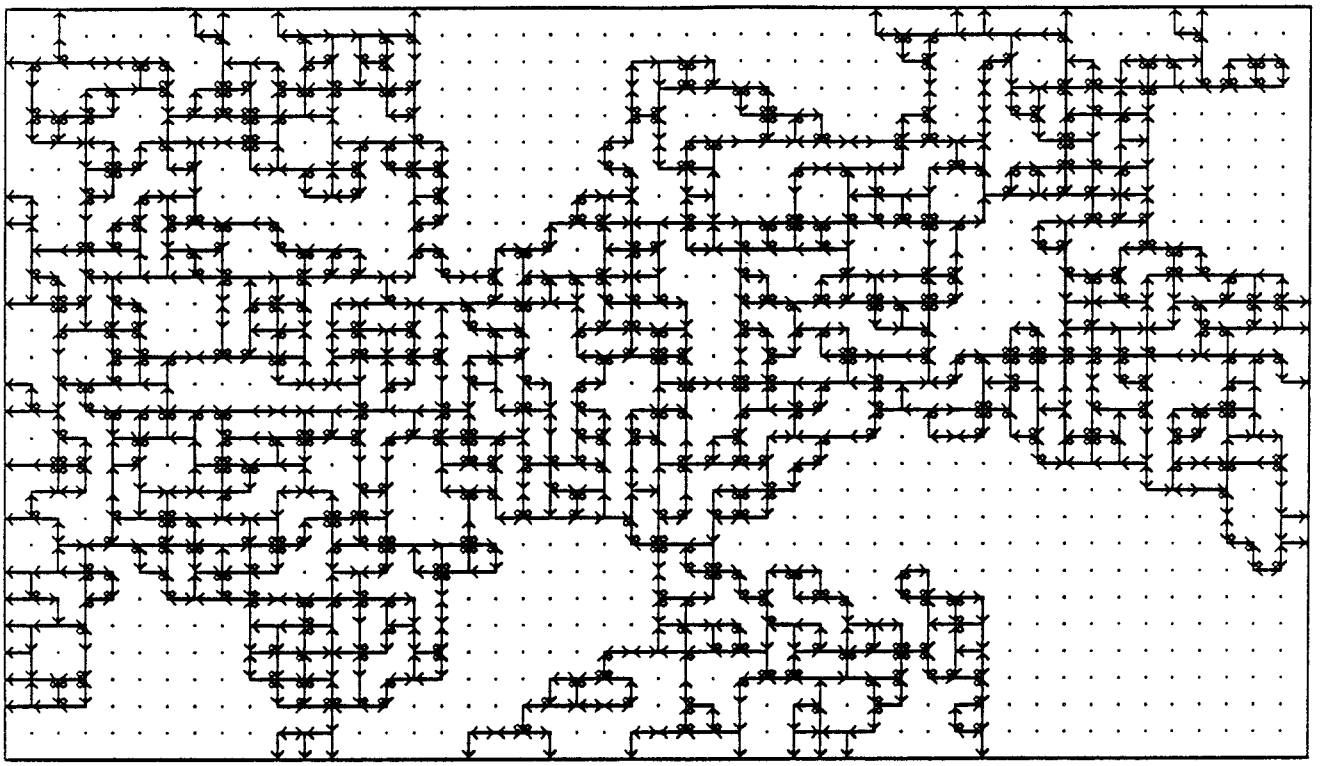


Bond Percolation at p=.50
(origin at center)

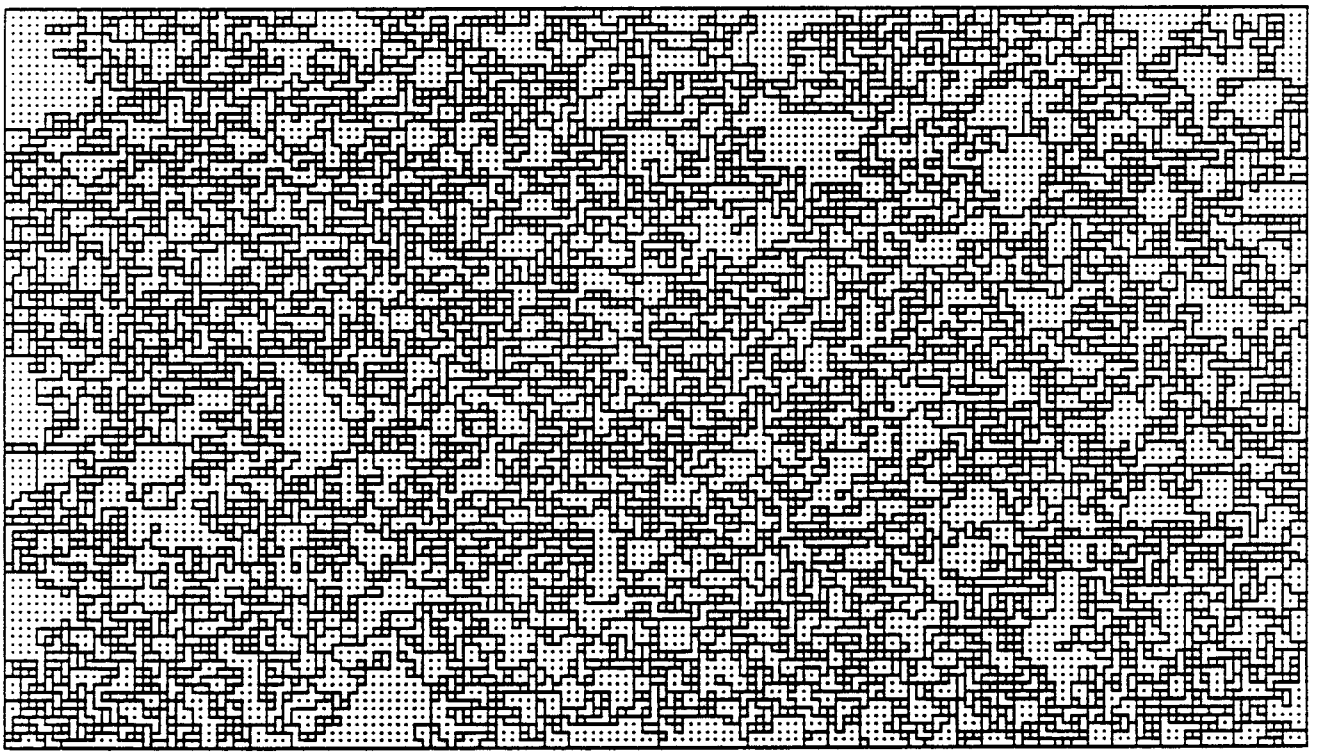


Bond Percolation at p=.60
(origin at center)

We first looked at the models as a physical system with the latticed comparable to a porous substance and C_0 the set of wet sights if a fluid source existed at **0**. With this in mind the comparison dealt only with the "filling" of the screen. A glance at the two pictures above shows what a striking change there is when p differs by only one one-hundredth of a percent. Below is a magnified simulation of directed percolation with arrows to indicate directions and another simulation(without arrows) at the same magnification as the previous bond percolation pictures. A visual comparison would seem to indicate that the extent to which directed percolation fills the plain can be related to bond percolation with a $.50 < p > .60$.



Directed Percolation
(origin at center)



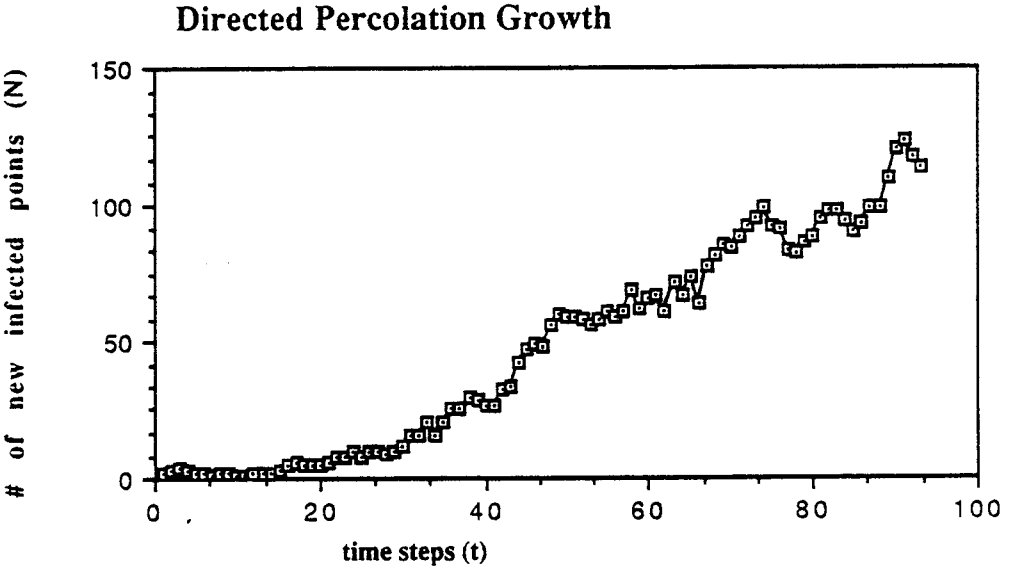
Directed Percolation
(origin at center)

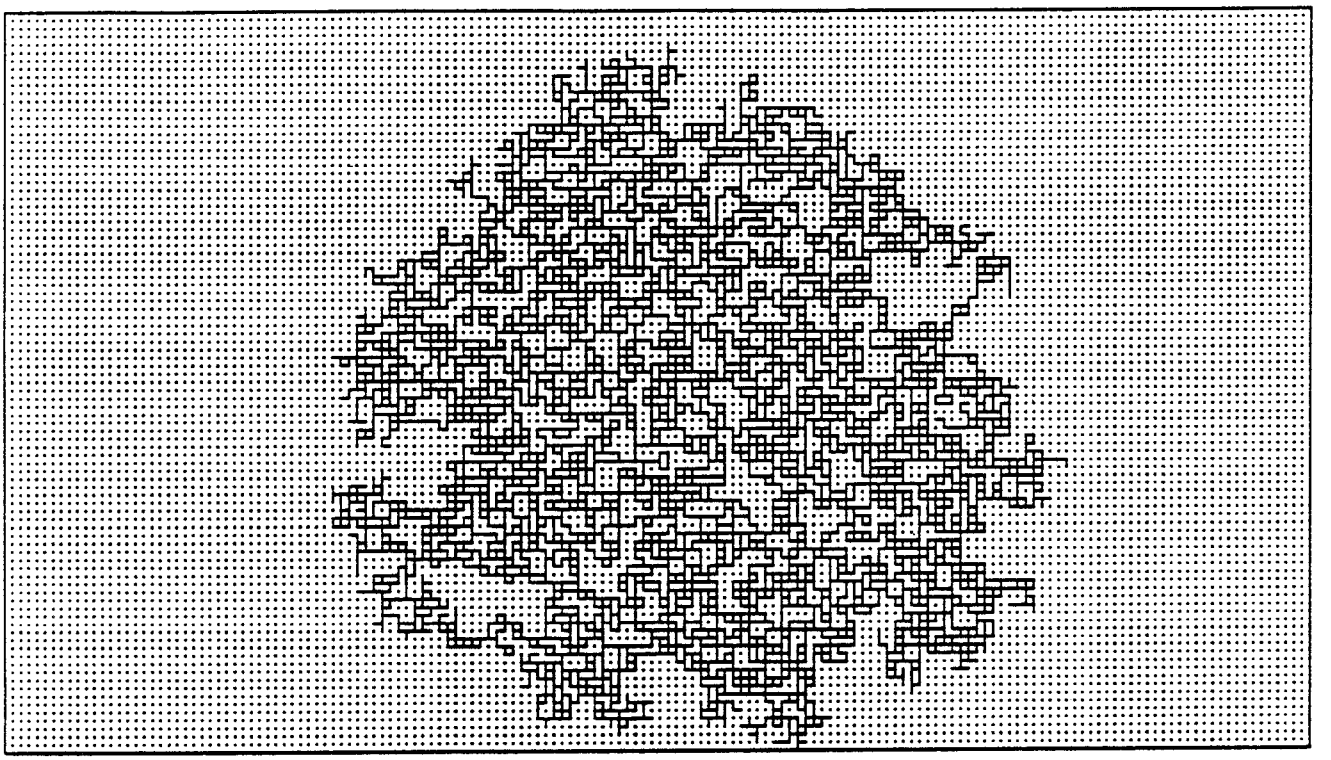
B. As A Growth Model.

After little or no success at relating the two percolation models, we decided to consider them as growth models. Starting with the origin as infected at time t=0, its neighbors are infected (if at all) at time t=1 (the infecting point is now considered dead), and etc. At each time step, the number of points that were infected were saved to a file so that the rate of growth could be analyzed. This was done for both percolation models. Let us set forward some notation. Let $\mathcal{N}_\tau=|C_0|$ at time τ, and let N_τ be the number of newly infected points at time step τ, thus $\mathcal{N}_\tau=\sum_{t=0}^{\tau} N_t$. We can consider N_t to be the sum of a sequence of random variables, $N_t=\sum_{i=0}^{N_{t-1}} X_i$. Thus each N_t is also a random variable.

1. Directed Percolation Growth.

For the directed model it is easy to see that $N_0=1$, $N_1=2$, and N_3 takes on a value between 1 and 4. Below is a graph of the N_t for one run of the directed percolation simulation which is also shown.



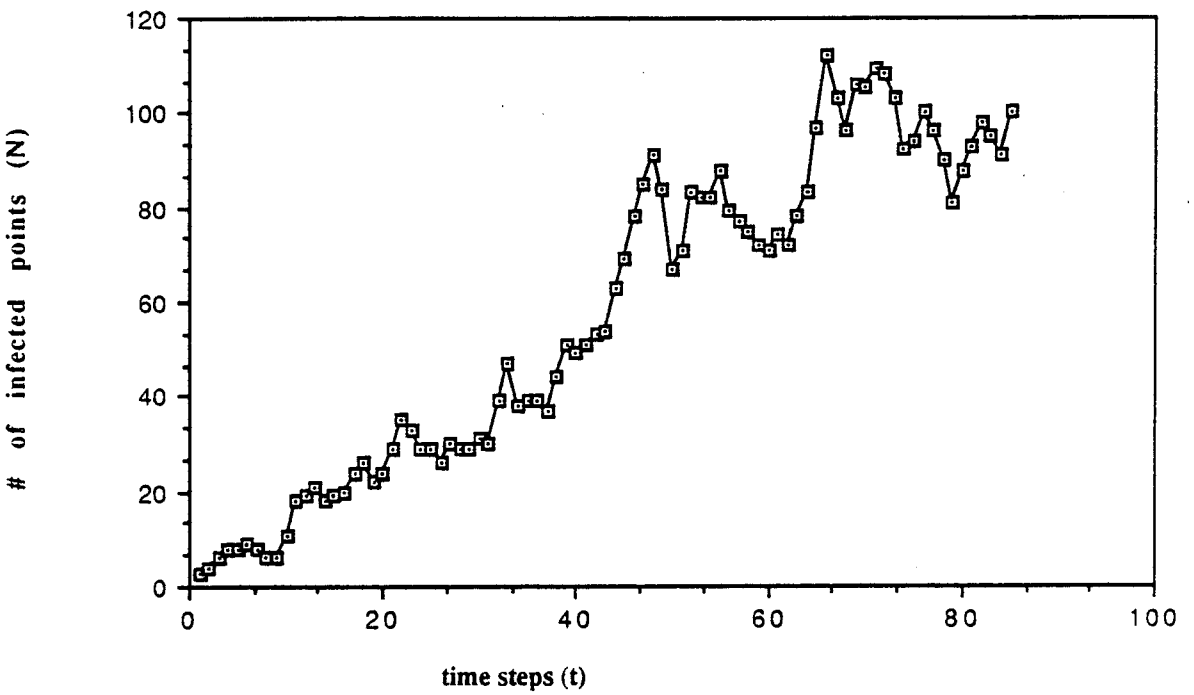


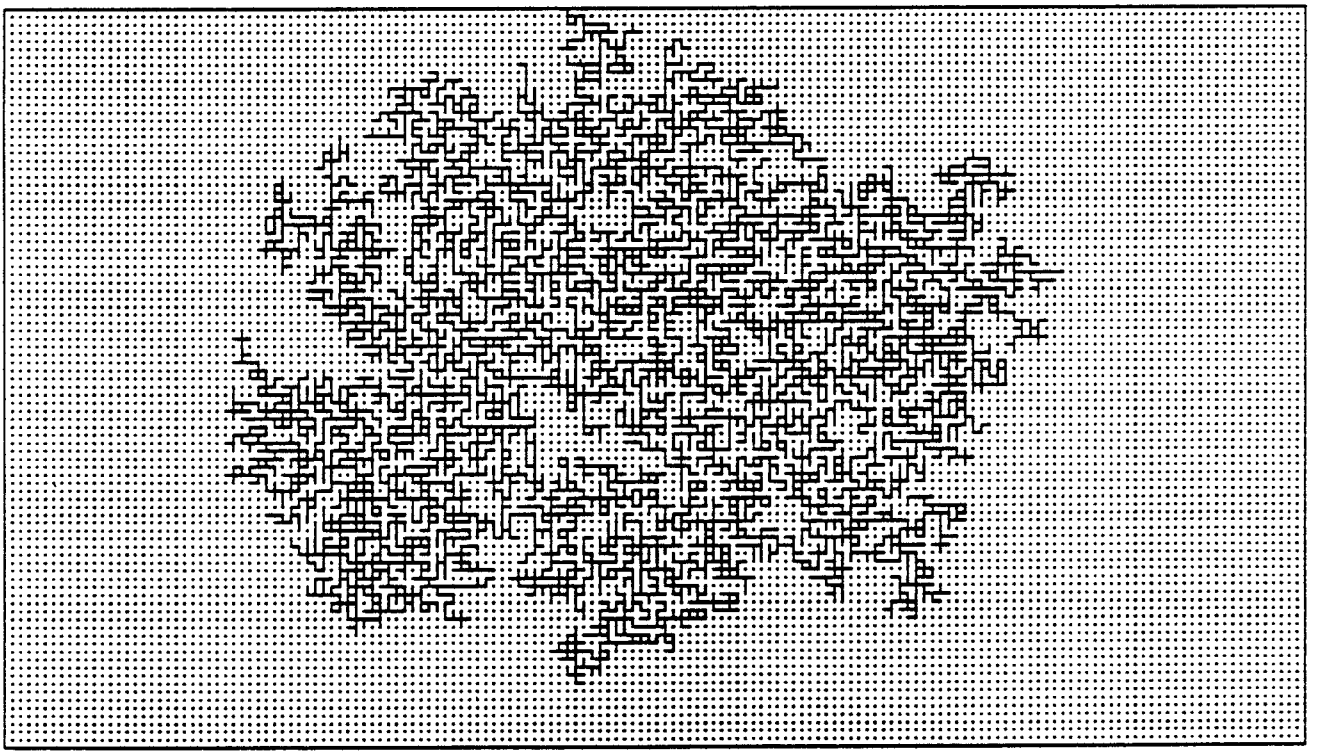
Directed Percolation Growth at time t=93.

This simulation raises questions as to the shape of C_0 for large t. The graph of the N_t shows quite a healthy growth rate and one that suffers very little from the winds of probability. A comparison needs to be made with bond percolation as to the rate of growth that it exhibits. An average of ten runs of the directed percolation growth model yielded an average difference of $|N_t - N_{t-1}| \approx 1.45$. Linear regression yields a similar slope.

2. Bond Percolation Growth.

In contrast to directed percolation, we found that the bond model exhibited a more erratic growth pattern. A very unstable graph of the N_t follows along with the corresponding simulation having a p=.53. One would expect directed percolation to be more stable since bond percolation has a wider range of possible configurations at each point. Similarity is expected also however since near p=.50 the expected number of open bonds at a single point is two, the exact number for directed percolation.





Bond Percolation (p=.53)
at time t=86.

3. Comparing the Models

For the comparison we will define $V(t)=\mathcal{N}_t$.

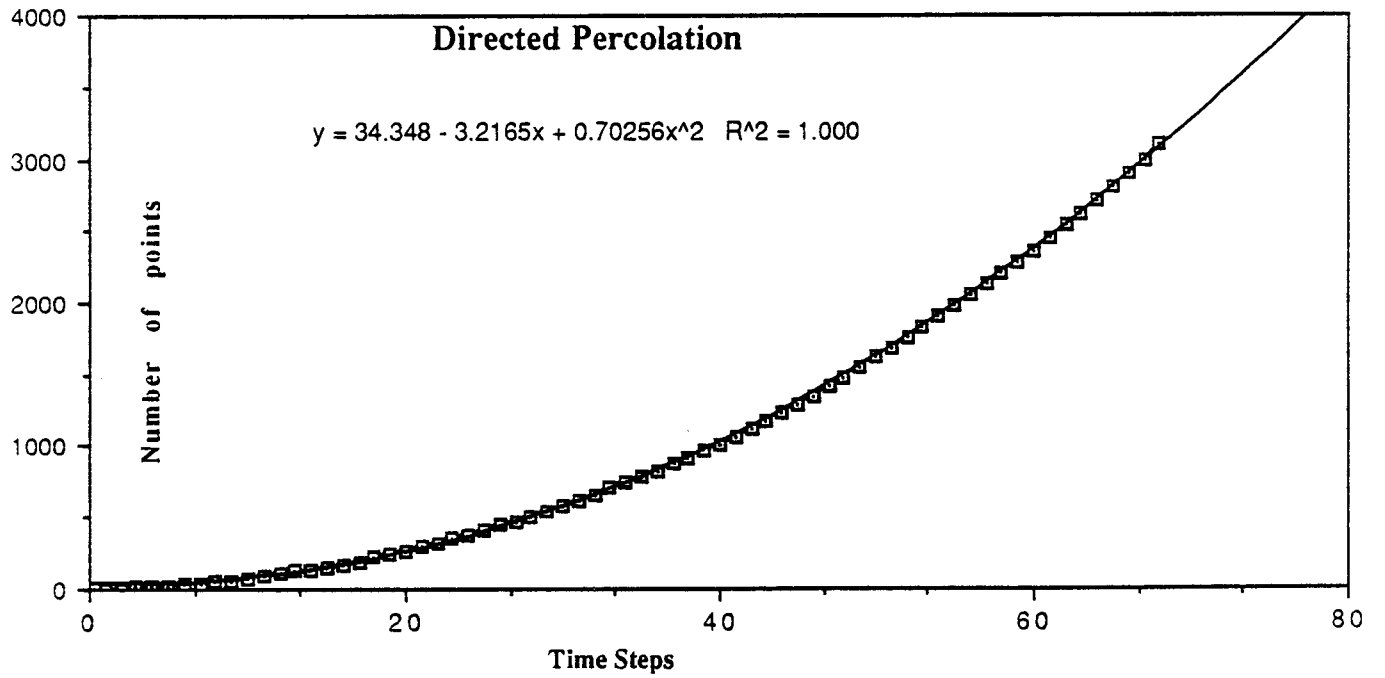
Our program calculated the total number of points our percolation(bond or directed) model reaches at time t. With the assistance of CricketGraph, we found that the total number of points, V, grows quadratically as t.(see graphs) For large t, the square term dominates so $V(t)\rightarrow at^2$:

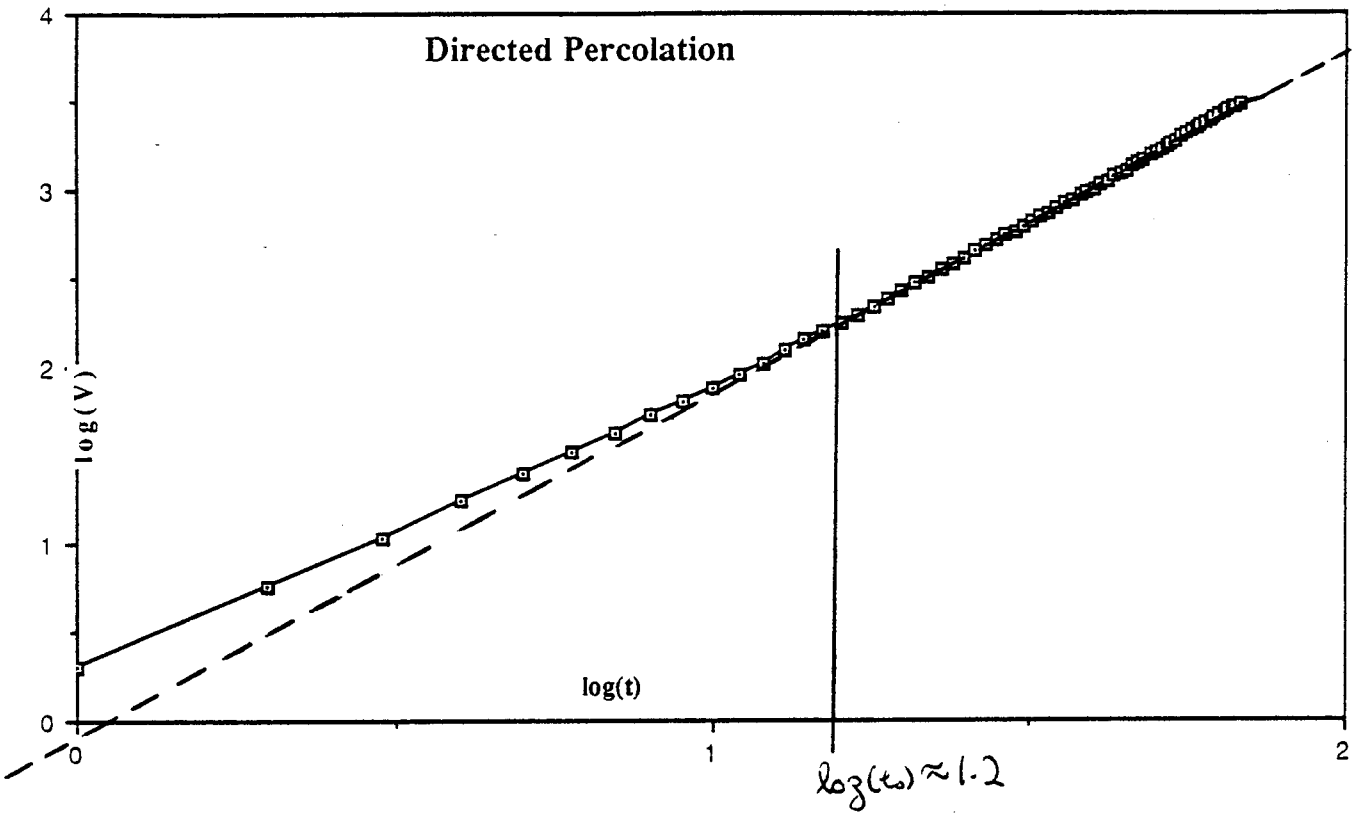
$$V(t_0)=at_0^2 \text{ for } t_0>>0 \text{ where } t_0=\inf\{t:V(t)=at^2\}$$

and

$$\log(V) = \log(a) + 2\log(t)$$

Plotting log(V) vs. log(t) and extrapolating, we found that $t_0\approx 20$ for the graph of the average directed percolation simulations. (reference to graph)





We have that:

$$V(t_0) = at_0^2$$

$$V(t_0 + 1) = at_0^2 + 2at_0 + a = V(t_0) + 2at_0 + a$$

.
.
.

in general,

$$V(t_0 + n) = V(t_0 + n - 1) + 2at_0 + (2n - 1)a$$

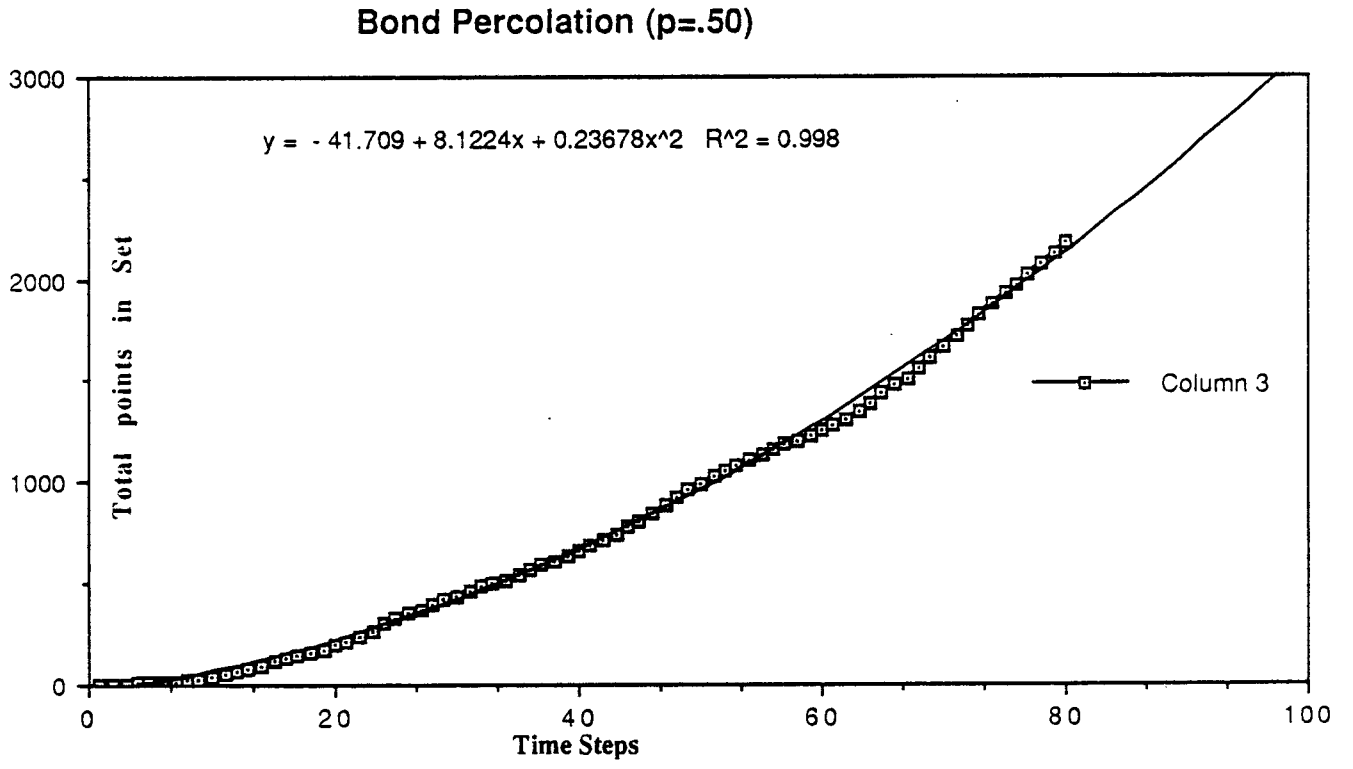
The expression $2at_0 + (2n - 1)a$ is the number of new points created at $t = t_0 + n$. Substituting $t = t_0 + n$ yields :

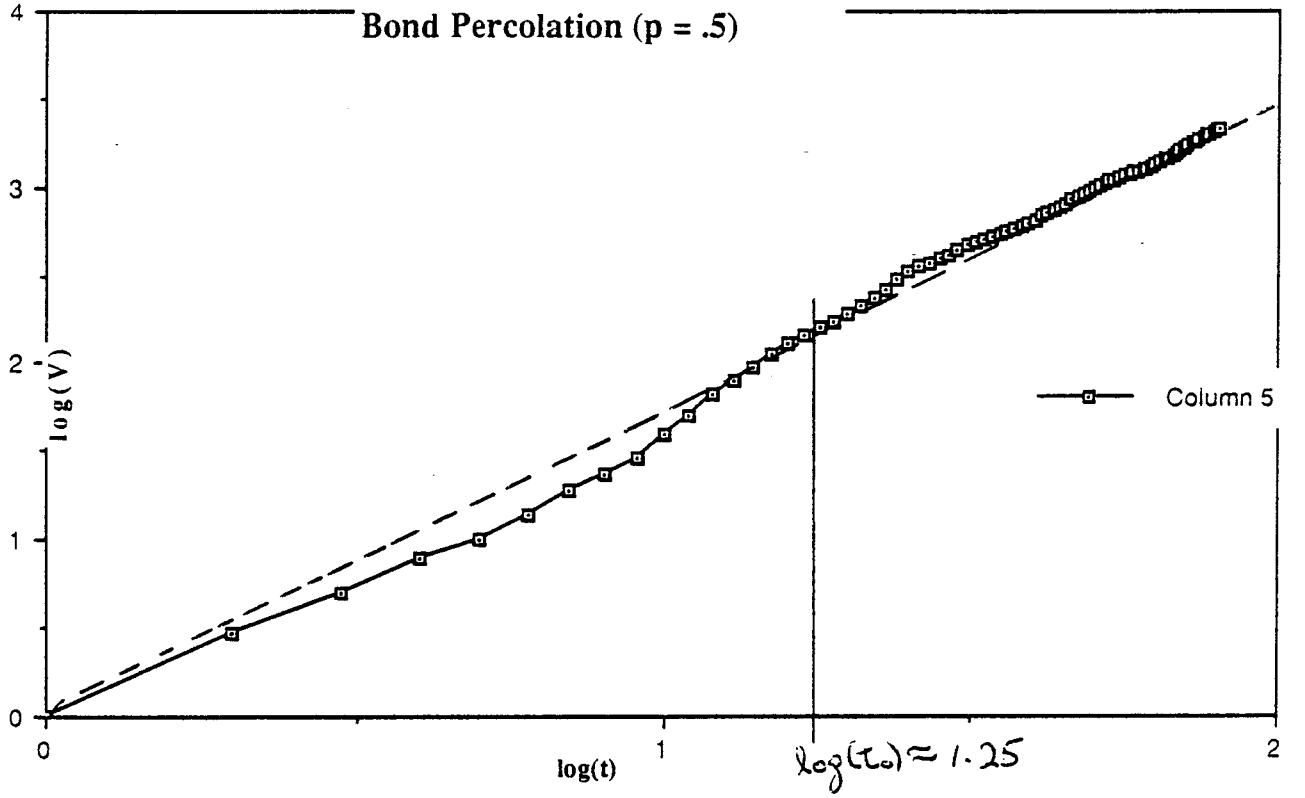
THE NUMBER OF NEW POINTS CREATED AT t IS 2at-a, (t > 20)
THE RATE OF GROWTH OF THE NUMBER OF NEW POINTS IS 2a, (t > 20)

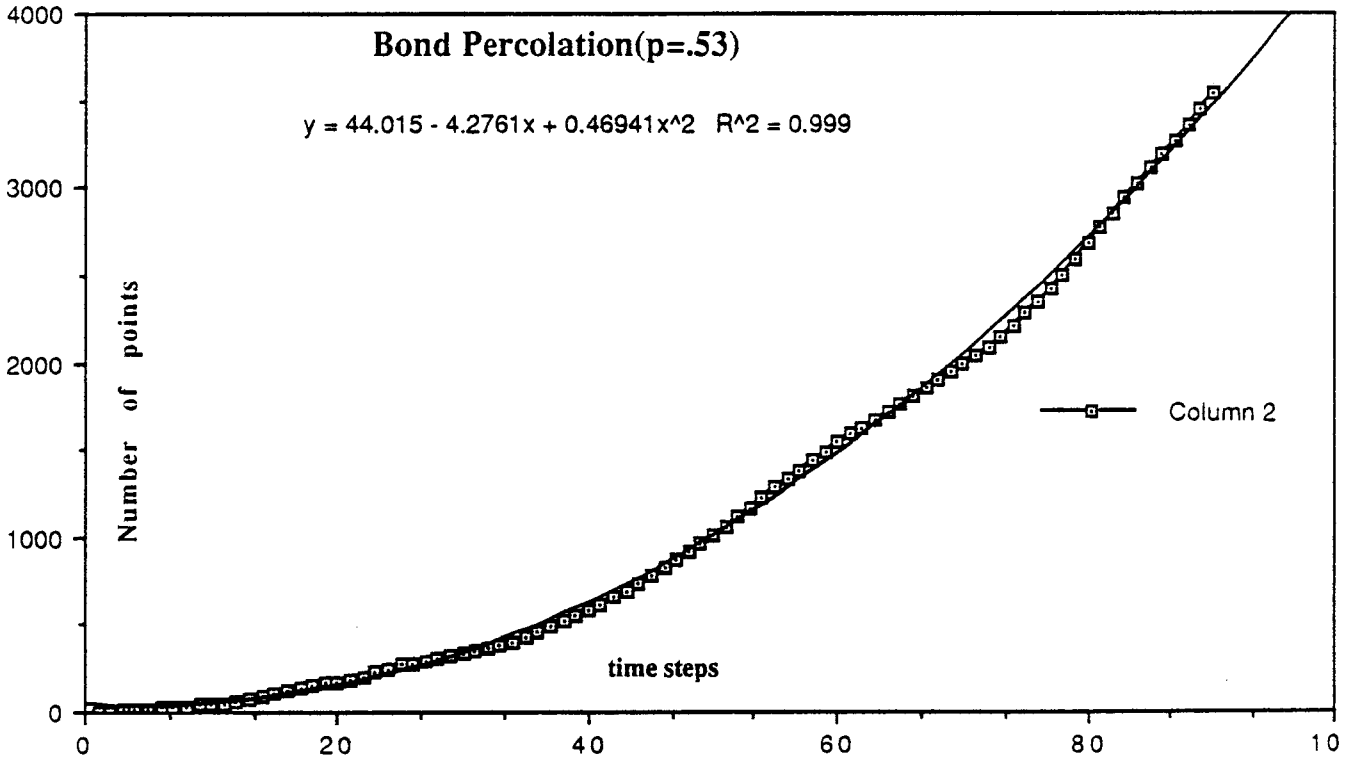
From the equation of the fitted curve(see graphs), a = .7. We have that the growth rate = 1.4.

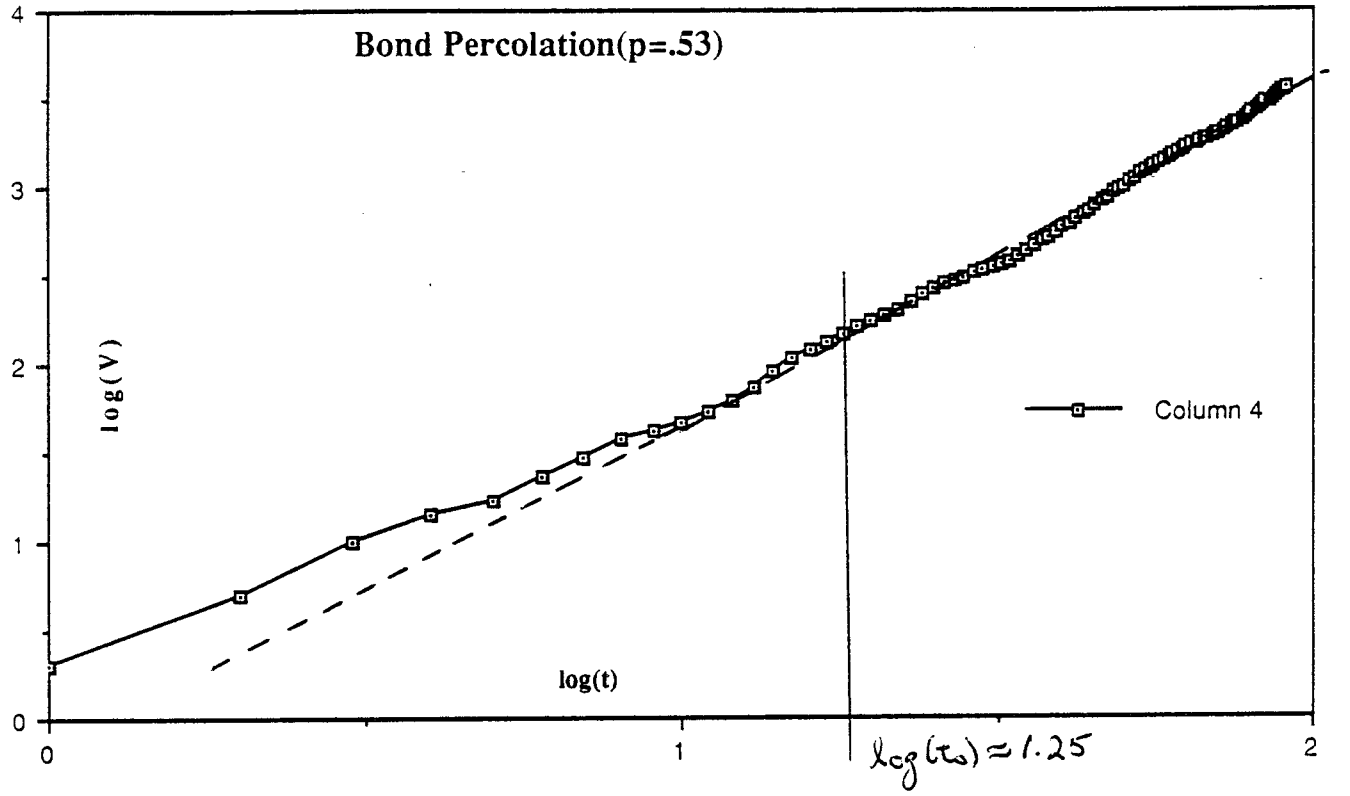
Plotting the total number of points at time t for bond percolation, and using the same argument as above, we have the following data:

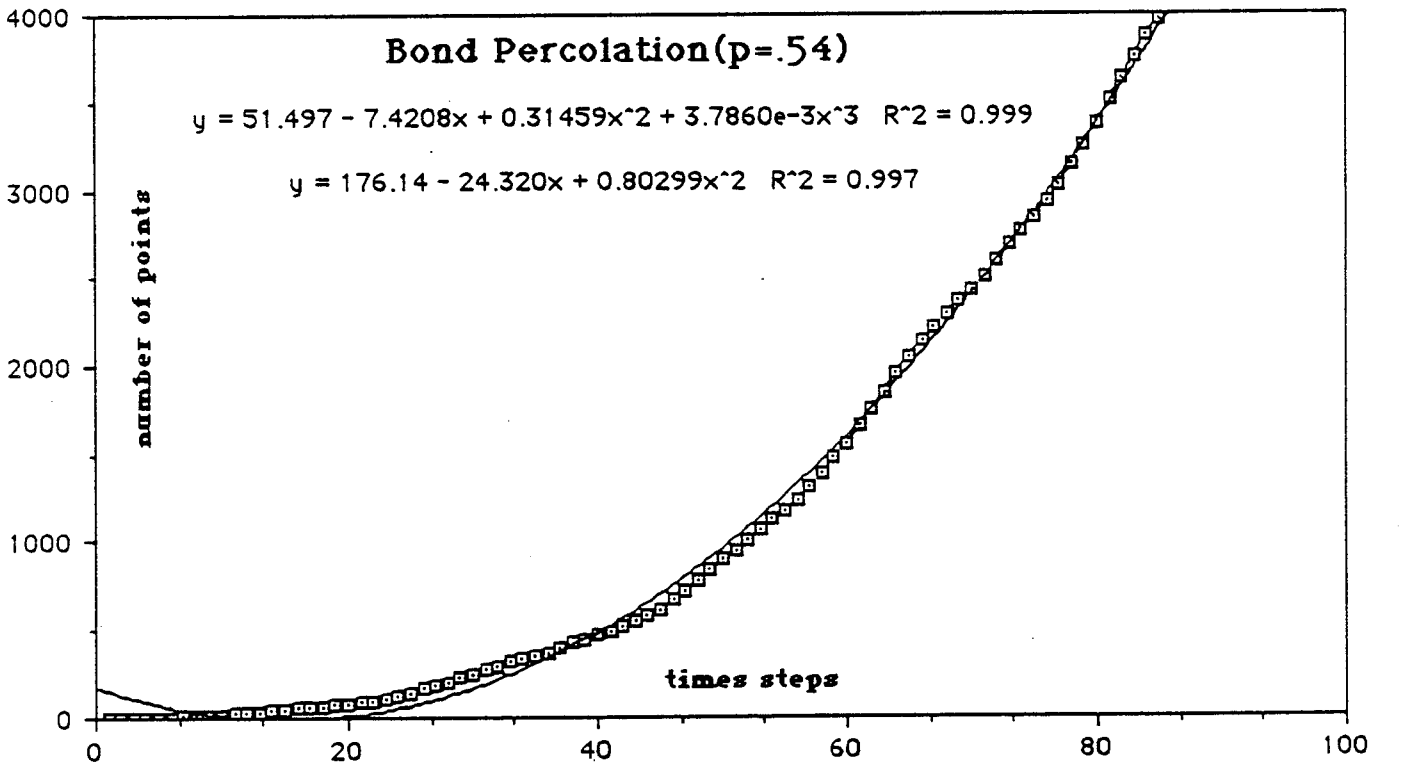
p	growth rate	t_0 (time at which $V \rightarrow at^2$)
.50	.47	20
.53	.94	20
.54	1.60	20
.55	1.88	15

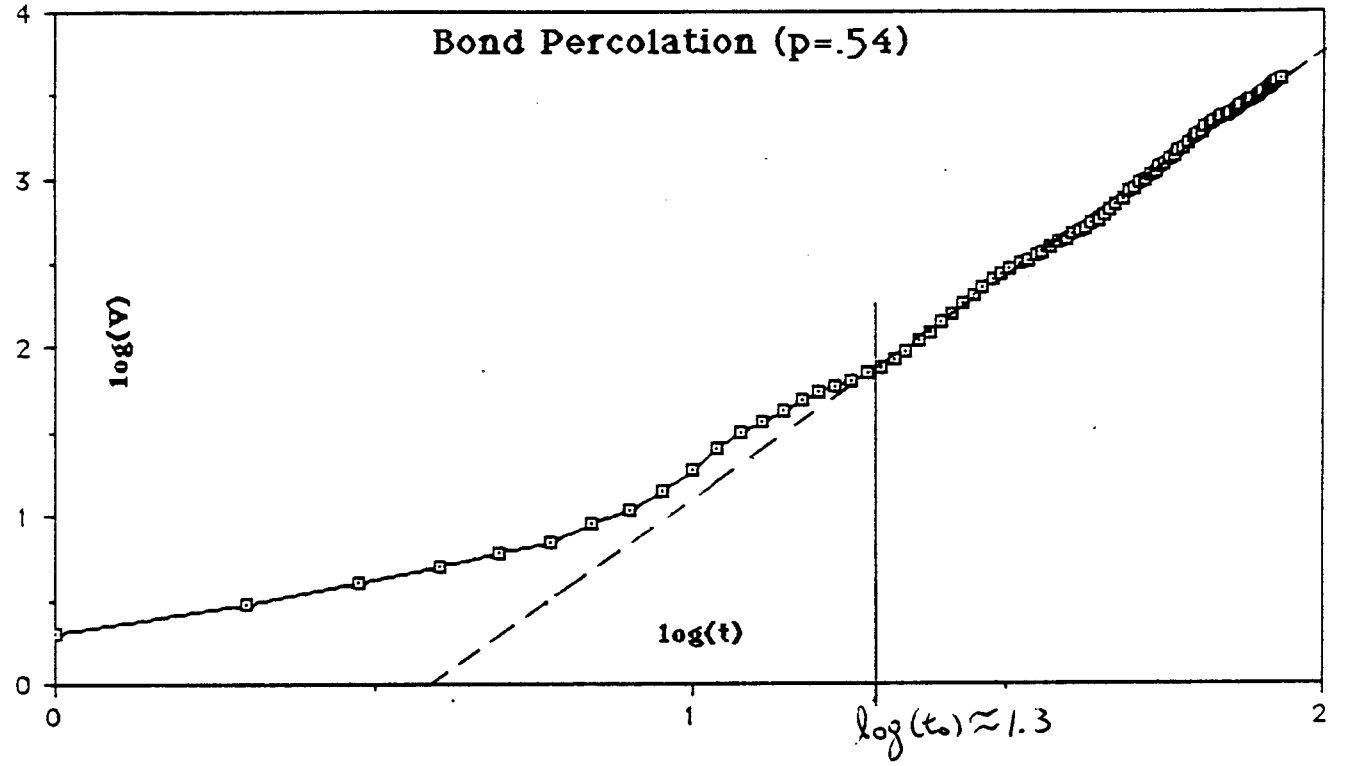


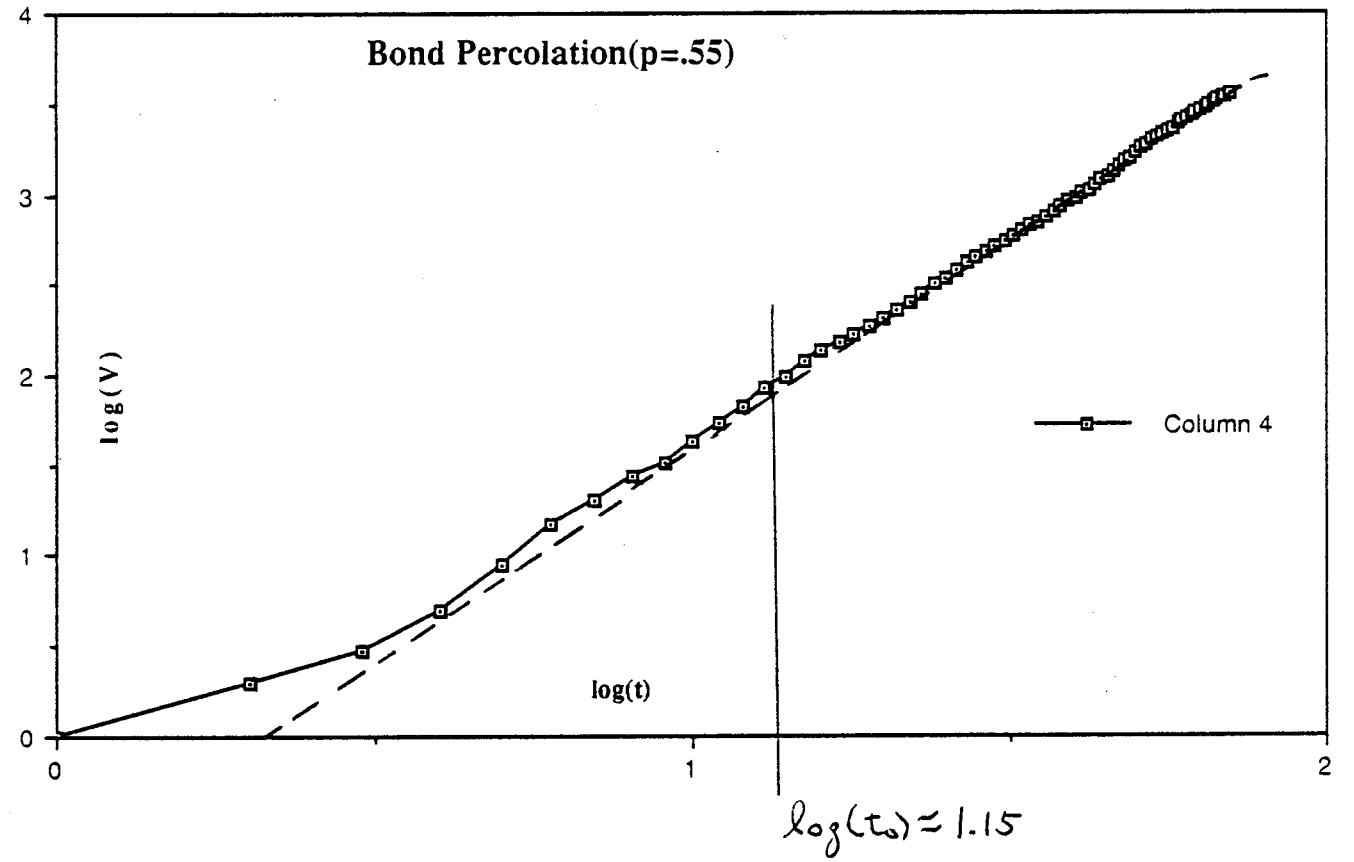

Bond Percolation (p = .5)
log(V)
log(t)
Column 5
log(τ₀) ≈ 1.25
0
1
2
3
4


Bond Percolation(p=.53)
y = 44.015 - 4.2761x + 0.46941x^2 R^2 = 0.999
Number of points
time steps
Column 2
0
1000
2000
3000
4000
0
20
40
60
80
10


Bond Percolation(p=.53)
log(V)
log(t)
Column 4
0
1
2
3
4
0
1
2
log(τ) ≈ 1.25


Bond Percolation(p=.54)
y = 51.497 - 7.4208x + 0.31459x^2 + 3.7860e-3x^3 R^2 = 0.999
y = 176.14 - 24.320x + 0.80299x^2 R^2 = 0.997
number of points
times steps
0
1000
2000
3000
4000
0
20
40
60
80
100


Bond Percolation (p=.54)
log(V)
log(t)
0
1
2
3
4
$\log(t_0) \approx 1.3$


Bond Percolation(p=.55)
log(V)
log(t)
Column 4
0
1
2
3
4
$\log(t_0) \approx 1.15$

Comparing the growth rate of directed percolation with bond percolation, we speculate that the directed model might be correlated with the bond model with $.53 < p < .54$. Imperically, it seems that if $V(t)=at^2$ holds for $t \rightarrow \infty$, directed percolation percolates.

III. Conclusions.

Although we were unable to prove that the directed model percolates, the similarities to the bond percolation model would suggest that it does. The extent to which the model "fills" the plain and the rate at which it grows when looked at as a growth model both give strong indications that the model can be proved to percolate. Directed percolation is deceptively simple in that there is no inherent probability in the model other than the choice of direction. The dependencies introduced by this aspect give it a difficulty that is not encountered in other bond percolation models. We hope that others will be encouraged, by the evidence we have presented here, to pursue a precise proof of its percolation.

Bibliography

R. Durrett (1988). *Lecture Notes on Particle Systems and Percolation.* Wadsworth & Brooks/Cole, Belmont.

H. Kesten (1980). The critical probability of bond percolation on the square lattice equals 1/2. *Comm. Math. Phys.* 74, 41-59.

Appendix

to

Directed Percolation in Two Dimensions:

Programs and Documentation used in the analysis of.

READ ME

Documentation for the Pascal Programs
Growth and D2dPerc
written by
Randy Doser

These two programs were written to study the problem of directed percolation in the integer lattice in two dimensions. They are computer simulations of the problem which was proposed in a lecture by Bob Burton during the summer of 1989 at Oregon State University. If you are not familiar with LightSpeed™ Pascal please consult the nearest manual. *DirPercolation* is a compiled version of *D2dPerc* under version 1.11 and *GrowComp* is a compiled version of *Bnd Grow Comp* under version 2.0.

I. THE MODEL

A. Directed Percolation

The model is a percolation model on the two dimensional integer lattice. As in other percolation models it deals with the sights or points of the lattice as well as what are called "bonds" between these points. Distance on the lattice is defined in a "taxi cab" metric fashion: for example the distance between the origin and the point (x,y) is merely x + y. With this notion of distance we say that bonds exist between points that are distance one apart (called neighbors), giving us four possible bonds at each point. In directed percolation(sometimes called oriented) each point has two bonds that allow the passage of a "fluid" from itself to a neighbor. These two bonds allow passage of the fluid in only one direction: away from the point. The question for this model is simply, "does percolation occurr?" Percolation is said to occurr if there is a positive probability of the fluid flowing to infinitly many points from a particular starting point. In other words if C_0 is the set of all points such that these points can be reach by a path of bonds from the origin, Percolation occurrs if the probability of the number of elements of C_0 being infinity is greater than zero:

$$C_0=\{x\in Z^2: 0 \to x (x \text{ can be reached from } 0)\}.$$

and,

$$P(|C_0| \to \infty)>0.$$

B. Bond Percolation

The model for bond percolation is slightly different(and much more studied) from directed percolation. Each bond is assigned a probability p of being open and 1-p of being closed independent of all other bonds. The question is similar to directed, however, with the added parameter of p. The question thus becomes, "At what probability p does percolation occurr?" This probability is referred to as P_c (read as "p critical").

For a good treatment of this percolation model refer to Richard Durrett's book, . P_c is known to be 0.5 for bond percolation.

II. THE PROGRAMS

A. D2dPerc.

This program was written with the view of directed percolation as a system of random one way streets. Thus it is a recurrsive algorithm that generates one "street" at a time for each point and following that street until it dead ends at the edge of the screen, the restricting radius, or into another previously generated street. When the particular path dead ends the recurrsion returns to the previous points on the path (stack) until it finds a point that has not yet chosen two directions and then starts a new path at that point, repeating this until all points have chosen two directions. The program offers several options which will be explained below.

1."Bond or Directed Percolation?(B/D):"

This allows you to choose between the two different models explained above. Bond percolation has been thoroughly studied in two dimensions as was included in the program for comparison with directed percolation.

2."Arrows?(y/n):"

This allows the user to include arrows in the graphics of the directed percolation model in order to better visualize what is occurring. The arrows can be very helpful at higher magnifications (8-10) but usually just clutter up the screen at the lower magnifications.

3."What probability?:"

This is for Bond percolation. The user should enter a real number on the open interval (0,1). This number will be used as the probability that a bond is "open".

4."What magnification?(3-10):"

This determines the number of lattice points that will be visible on the screen and the size of the bonds. The number

represents the number of pixels from one lattice point to another. We could have gone down as low as 2, however, because of memory limitations this would only make the picture smaller and give us no more lattice points than magnification 3.

5. **"Restricting Radius?(y/n):"**
   The resticting radius was added to aid the study of the model. Its utility is questionable at best. The idea is that the radius (which in the chosen metric form a closed set of points equal distances from the origin) of size n might aid in analysis since every minimum path is of length n to the outside of the radius. The problem here lies in ridding the union of all probabilities of minimum length paths of the intersection of these paths. Running the program several times reveals that rarely if ever is a minimum length path the one that crosses the radius.
6. **CLICK THE MOUSE TO TERMINATE AFTER CURRENT PERCOLATION.**

B. GrowthRate.

This program was written to study directed percolation as a growth model. The growth model starts with one individual(the origin) at time t=1 which "infects" to two neighboring points at random. These points in turn "infects" to two neighboring points at time t=2, and etc. Once a point has infected two neighbors it is considered "dead" and can no longer infect or be infected. The aspect that the model was created to look at was the rate of growth or spread of the "disease" and so the number of new infections at each time step is recorded in a file for later analysis (on a spread sheet for instance). OPTIONS INCLUDE ITEMS 2, 4, 5, & 6 FROM THE PREVIOUS PROGRAM.

```
program GrowthComp;

{This program was written to model Directed Percolation(see READ ME). It treats the problem as}
{a lattice with two one way streets chosen at random from the four directions available at each point.}
{The streets always head away from the point at which they are chosen. Since it concentrates on the }
{paths and not on a growth type model (see READ ME) it uses a recurrsive routine that follows each }
{to its termination and then returns to the last point on the stack and follows it until all points have }
{chosen two directions. Too compare the model to Bond percolation a bond percolation model is also }
{available.}

 const
   WIDTH = 480;
   HEIGHT = 280;


 type
   pointPTR = ^elmnt;                    {Define the linked list used to store old & new points.}
   elmnt = record
      y, x: integer;
      next: pointPTR;
    end;

   four = 1..4;
   point = record
      paths: set of four;
      dead: boolean;
    end;
   graph = array[-80..80, -47..47] of point;

 var
   streets: graph;
   oldlist, newlist: pointPTR;
   afile: text;
   x_origin, y_origin, i: integer;
   arrows, radius: boolean;
   respns, perc_choice: char;
   yes, bond: set of char;
   mag: 3..10;
   n_box: 2..46;
   openprob: real;
   Event: EventRecord;

 procedure WindowSetup;
{=============================================================================}
{= PURPOSE: Resizes the drawing window for the graphics display.             =}
{= SUBROUTINES CALLED:  SetRect, SetDrawingRect, ShowDrawing(system tools).  =}
{= INPUT: none.                                                              =}
{= OUTPUT: none.                                                             =}
{= GLOBALS: none.                                                            =}
{=============================================================================}
   var
     text_window, drawing_window: rect;
 begin
```

```
    HideAll;
    SetRect(drawing_window, 0, 37, 510, 339);
    SetDrawingRect(drawing_window);
    ShowDrawing;
  end;

{*************************************************************************}

  procedure DrawLattice (var dead: graph);
{=========================================================================}
{= PURPOSE: Draws the integer lattice according to the restricting radius and the magnification  =}
{=          selected by the user.                                                                =}
{= SUBROUTINES CALLED:  PenNormal, MoveTo, LineTo (Quickdraw).                                   =}
{= INPUT: dead= array of lattice points boolean.                                                 =}
{= OUTPUT:  dead= initialized.                                                                   =}
{=          (draws the lattice).                                                                 =}
{= GLOBALS: radius= bolean, tells whether user wants to restrict the growth to a specific radius =}
{=                from the origin on the lattice.                                                =}
{=          n_box= integer value of the restricting radius.                                      =}
{=          mag= integer value of the magnification. Represents the # of pixels                  =}
{=========================================================================}
    var
      x, y, i, j: integer;
  begin
    PenNormal;
    if not (radius) then
{If the user chose not to use a restricting radius then draw a lattice on the full window.}
      for i := (WIDTH div mag) downto 0 do
        for j := (HEIGHT div mag) downto 0 do
          begin
            x := i - (WIDTH div (2 * mag));
            y := j - (HEIGHT div (2 * mag));
            streets[x, y].dead := false;
            streets[x, y].paths := [];
            MoveTo(i * mag, j * mag);
            LineTo(i * mag, j * mag);
          end
    else
{If the user wants a restricting radius then draw only those lattice points that are within "n_box"}
{units of the origin.}
      begin
        for i := n_box downto -n_box do
          for j := (n_box - abs(i)) downto -(n_box - abs(i)) do
            begin
              streets[i, j].dead := false;
              streets[i, j].paths := [];
              MoveTo(x_origin + i * mag, y_origin + j * mag);
              LineTo(x_origin + i * mag, y_origin + j * mag);
            end;
{Draw the radius itself.}
        MoveTo(n_box * mag + x_origin, y_origin);
        LineTo(x_origin, n_box * mag + y_origin);
        LineTo(x_origin - n_box * mag, y_origin);
```

```
      LineTo(x_origin, y_origin - n_box * mag);
      LineTo(n_box * mag + x_origin, y_origin);
    end;
  end;{DrawLattice}

{*****************************************************************************}

  procedure FloodIt (x, y, dir: integer; var u, v: integer);
{=============================================================================}
{= PURPOSE: Figures the coordinates of the new point from the given dir(ection) and draws an  =}
{=          arrow or a line to that point.                                                    =}
{= SUBROUTINES CALLED:  LineTo, MoveTo, Move, Line (QuickDraw).                               =}
{= INPUT: x,y: integer coordinates of the current point.                                      =}
{=        dir: the direction (1,2,3,or 4) to move from current point.                         =}
{=        u,v: the coordinates to be calculated.                                              =}
{= OUTPUT: u,v: the coordinates of the point that is one unit in dir(ection) from x,y.        =}
{= GLOBALS: arrows: boolean telling whether the user wants arrows drawn or line segments.     =}
{=============================================================================}
  begin
    u := x - (dir - 3) * ((dir mod 2) - 1);          {Compute the new point's coordinates.}
    v := y + (dir - 2) * (dir mod 2);
    MoveTo(x_origin + x * mag, y_origin + y * mag);
    LineTo(x_origin + u * mag, y_origin + v * mag);
    if arrows then
      begin
        Move(x - u, y - v);
        Line(((x - u) * 2 + (y - v) * (-2)), ((x - u) * (-2) + (y - v) * 2));
        Move(((x - u) * (-2) + (y - v) * 2), ((x - u) * 2 + (y - v) * (-2)));
        Line(((x - u) * 2 + (y - v) * (2)), ((x - u) * (2) + (y - v) * 2));
      end;
  end;{FloodIt}

{*****************************************************************************}

  function OnTheEdge (x, y: integer): boolean;
{=============================================================================}
{= PURPOSE: Determines if a point is on the edge of the radius or the screen.                 =}
{= SUBROUTINES CALLED:  abs.                                                                  =}
{= INPUT: x,y: the coordinates of the point in question.                                      =}
{= OUTPUT: OnTheEdge: boolean value.                                                          =}
{= GLOBALS: none.                                                                             =}
{=============================================================================}
  begin
    if ((abs(x) = (WIDTH div (2 * mag))) or (abs(y) > (HEIGHT div (2 * mag) - 1))) then
      OnTheEdge := true
    else if radius and ((abs(x) + abs(y)) = n_box) then
      OnTheEdge := true
    else
      OnTheEdge := false;
  end;


{*****************************************************************************}
```

```
  procedure AddElmnt (var n, x, y: integer; var n_list: pointPTR; var dead: graph);
{=============================================================================}
{= PURPOSE: Adds an element to the linked list of new points if it has not already been visited.  =}
{= SUBROUTINES CALLED:  none.                                                  =}
{= INPUT: n: the number of points in the new point list.                       =}
{=        x,y: coordinates of the new point.                                   =}
{=        n_list: pointer to the current element (last element) of the new list.=}
{=        dead: 2D array of boolean values telling whether a point has been visited.=}
{= OUTPUT: n: new number of points in list (if element added).                 =}
{= GLOBALS: newlist: pointer to the root of the new point list.                =}
{=============================================================================}
    var
      new_elmnt: pointPTR;
  begin
    If not (streets[x, y].dead) then  {If the point has not yet been visited...}
      begin
        streets[x, y].dead := true;       {It's as good as dead now.}
        n := n + 1;                       {Increment list length.}
        If newlist = nil then             {If the list is empty...}
          begin
            new(newlist);                     {Initialize the root pointer.}
            n_list := newlist;                {Initialize the list pointer.}
          end
        else                              {If the list is not empty...}
          begin
            new(n_list^.next);                {Add a pointer to the end of the list.}
            n_list := n_list^.next;           {Move the list pointer to the new element.}
          end;
        n_list^.x := x;                   {Add the values to the new linked list element.}
        n_list^.y := y;
        n_list^.next := nil;
      end;
  end;

{*****************************************************************************}

  procedure Grow (var dead: graph; z, w: integer);
{=============================================================================}
{= PURPOSE:  This is the major routine of the program.  This procedure goes over the list of old  =}
{=           points and chooses two directions for each point.  It calls AddElmnt for each new   =}
{=           point and FloodIt to create the graphics display.  After the old list has been exhausted=}
{=           the new list becomes the next old list. This process repeats until OnTheEdge returns =}
{=           a value of true.                                                  =}
{= SUBROUTINES CALLED:  OnTheEdge, FloodIt, AddElmnt.                          =}
{= INPUT: dead: Array of boolean values representing the lattice and telling whether each point  =}
{=              has been visited or not.                                       =}
{=        z,w: Starting point for the Grow routine (the origin unless changed).=}
{= OUTPUT: (Graphics of directed percolation and growth data to a file.)       =}
{= GLOBALS: newlist: root pointer to the list of new points.                   =}
{=          oldlist: root pointer to the list of old points.                   =}
{=============================================================================}
    var
```

```
    far_enough: boolean;          {far_enough tells if the generation goes off screen or radius.}
    n_list, o_list: pointPTR;     {list pointers.}
    x, y, dir1, dir2, u, v, n, i: integer;
  begin
   far_enough := false;
   newlist := nil;
   new(oldlist);                  {Set the first oldlist equal to the starting point provided.}
   oldlist^.x := z;
   oldlist^.y := w;
   oldlist^.next := nil;
   repeat{until far_enough}
    n := 0;
    while (oldlist <> nil) and not (far_enough) do
{while there are still points in the oldlist and the generation hasn't violated its radius.}
      begin
       dir1 := 0;                          {Initialize the primary direction.}
       x := oldlist^.x;                    {Get the coordinates of the next point.}
       y := oldlist^.y;
       if OnTheEdge(x, y) then
        far_enough := OnTheEdge(x, y)
       else                                {If point is not off screen or over radius...}
        begin
         dir1 := (abs(random) mod 4) + 1;      {Choose the first direction.}
         FloodIt(x, y, dir1, u, v);            {Update the picture.}
         AddElmnt(n, u, v, n_list, dead);      {Add new point to the new list(if not dead).}

         repeat                                {Choose a direction different from the first.}
          dir2 := (abs(random) mod 4) + 1;
         until dir1 <> dir2;
         FloodIt(x, y, dir2, u, v);            {Update the picture.}
         AddElmnt(n, u, v, n_list, dead);      {Add new point to the new list(if not dead).}

         o_list := oldlist;                    {Save pointer to previous point.}
         oldlist := oldlist^.next;             {Move root to next list element.}
         dispose(o_list);                      {dispose of previous point.}
        end;
      end;
    writeln(afile, n);                   {Write the generation size to the data file.}
    oldlist := newlist;                  {Swap lists.}
    newlist := nil;
   until far_enough;
  end;{Grow}

{*************************************************************************}

  procedure BondGrow (var streets: graph; z, w: integer);
{=========================================================================}
{= PURPOSE:                                                               =}
{= SUBROUTINES CALLED:   OnTheEdge, FloodIt                               =}
{= INPUT: x,y: the coordinates of the point needing directions.           =}
{=        streets: an array of lattice point paths and whether the point has chosen 2 directions. =}
{= OUTPUT: (Pretty pictures).                                             =}
{= GLOBALS: none.                                                         =}
```

```
{=============================================================================}
   var
    far_enough: boolean;          {far_enough tells if the generation goes off screen or radius.}
    n_list, o_list: pointPTR;     {list pointers.}
    x, y, u, v, n, i: integer;
  begin
   far_enough := false;
   newlist := nil;
   new(oldlist);                  {Set the first oldlist equal to the starting point provided.}
   oldlist^.x := z;
   oldlist^.y := w;
   oldlist^.next := nil;

   repeat {until far_enough}
    n := 0;
    while (oldlist <> nil) and not (far_enough) do
     begin
      x := oldlist^.x;
      y := oldlist^.y;
      if (OnTheEdge(x, y)) then
       far_enough := true
      else
       for i := 1 to 4 do                          {Do it for all four directions.}
        if not (i in streets[x, y].paths) then     {If bond hasn't been closed/opened...}
         if abs(Random) <= (MAXINT * openprob) then   {If bond should be open...}
          begin
           FloodIt(x, y, i, u, v);                      {Draw path to new point.}
      {Add the reverse path to the new point .}
           streets[u, v].paths := streets[u, v].paths + [(i - (i div 3) * (4) + 2)];
           AddElmnt(n, u, v, n_list, streets);
          end
         else                                       {If bond should be closed...}
          begin
           u := x - (i - 3) * ((i mod 2) - 1);          {Calculate opposite point.}
           v := y + (i - 2) * (i mod 2);
           streets[u, v].paths := streets[u, v].paths + [(i - (i div 3) * (4) + 2)];{Add reverse path.}
          end;
      o_list := oldlist;
      oldlist := oldlist^.next;
      dispose(o_list);
     end;
    writeln(afile, n);
    oldlist := newlist;
    newlist := nil;
   until far_enough;
  end;{BondGrow}

{*****************************************************************************}

begin
 bond := ['b', 'B'];
 yes := ['y', 'Y'];
 RandSeed := TickCount div 2;
```

```
  showtext;
  write('Bond or Directed Percolation?(B/D):');
  readln(Perc_choice);
  if not (Perc_choice in bond) then
    begin
      write('Arrows?(y/n):');
      readln(respns);
      arrows := respns in yes;
    end
  else
    begin
      write('What probability?:');
      readln(openprob);
    end;

  write('What magnification?(3-10):');
  readln(mag);
  x_origin := (WIDTH div (2 * mag)) * mag;
  y_origin := (HEIGHT div (2 * mag)) * mag;

  write('Restricting Radius?(y/n):');
  readln(respns);
  radius := respns in yes;
  if radius then
    begin
      write('Radius =(2-46)');
      readln(n_box);
    end;
  write('Prepare to create a data file.');
  for i := 1 to 400 do{Pause}
    ;
  rewrite(afile, NewFileName(''));


  while not (GetNextEvent(2, Event)) do        {Repeat until mouse is clicked.}
    if perc_choice in bond then
      begin
        arrows := false;
        WindowSetup;
        DrawLattice(streets);
        BondGrow(streets, 0, 0);
      end
    else
      begin
        WindowSetup;
        DrawLattice(streets);
        Grow(streets, 0, 0);
      end;

end.{GrowthComp}
```

```
program D2dperc;

{This program was written to model Directed Percolation(see READ ME). It treats the problem as}
{a lattice with two one way streets chosen at random from the four directions available at each point.}
{The streets always head away from the point at which they are chosen. Since it concentrates on the }
{paths and not on a growth type model (see READ ME) it uses a recurrsive routine that follows each }
{to its termination and then returns to the last point on the stack and follows it until all points have }
{chosen two directions. Too compare the model to Bond percolation a bond percolation model is also }
{available.}

  const
    WIDTH = 480;
    HEIGHT = 280;


  type
    four = 1..4;
    point = record
        paths : set of four;
        dead : boolean;
      end;
    graph = array[-80..80, -47..47] of point;

  var
    streets : graph;
    x_origin, y_origin : integer;
    arrows, radius : boolean;
    respns, perc_choice : char;
    yes, bond : set of char;
    mag : 3..10;
    n_box : 2..46;
    openprob : real;
    Event : EventRecord;

  procedure WindowSetup;
{=============================================================================}
{= PURPOSE: Resizes the drawing window for the graphics display.             =}
{= SUBROUTINES CALLED:  SetRect, SetDrawingRect, ShowDrawing(system tools).  =}
{= INPUT: none.                                                              =}
{= OUTPUT: none.                                                             =}
{= GLOBALS: none.                                                            =}
{=============================================================================}
    var
      text_window, drawing_window : rect;
  begin
    HideAll;
    SetRect(drawing_window, 0, 37, 510, 339);
    SetDrawingRect(drawing_window);
    ShowDrawing;
  end;

{*****************************************************************************}
```

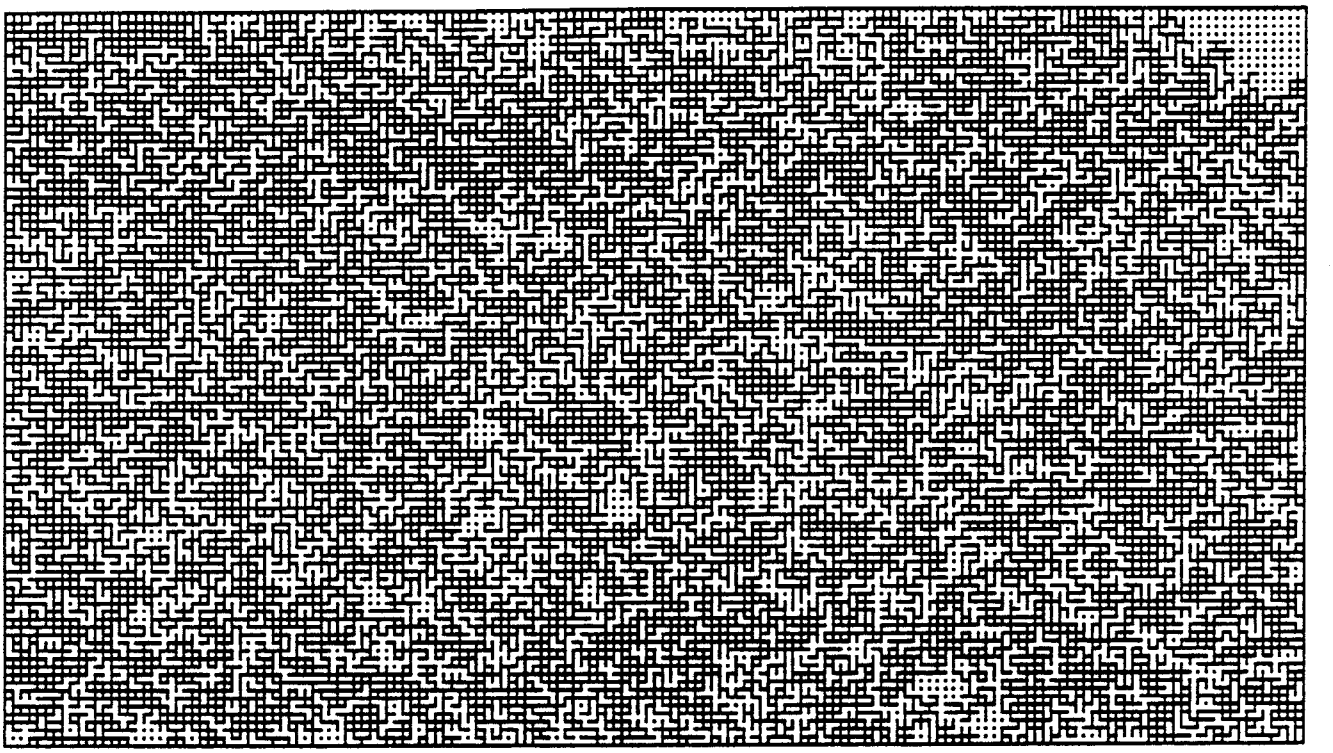
```
  procedure DrawLattice (var dead : graph);
{=============================================================================}
{= PURPOSE: Draws the integer lattice according to the restricting radius and the magnification  =}
{=          selected by the user.                                                               =}
{= SUBROUTINES CALLED:  PenNormal, MoveTo, LineTo (Quickdraw).                                  =}
{= INPUT: dead= array of lattice points boolean.                                                =}
{= OUTPUT:  dead= initialized.                                                                  =}
{=          (draws the lattice).                                                                =}
{= GLOBALS: radius= bolean, tells whether user wants to restrict the growth to a specific radius =}
{=                from the origin on the lattice.                                               =}
{=          n_box= integer value of the restricting radius.                                     =}
{=          mag= integer value of the magnification. Represents the # of pixels                 =}
{=============================================================================}
    var
      x, y, i, j : integer;
  begin
    PenNormal;
    If not (radius) then
{If the user chose not to use a restricting radius then draw a lattice on the full window.}
      for i := (WIDTH div mag) downto 0 do
        for j := (HEIGHT div mag) downto 0 do
          begin
            x := i - (WIDTH div (2 * mag));
            y := j - (HEIGHT div (2 * mag));
            streets[x, y].dead := false;
            streets[x, y].paths := [];
            MoveTo(i * mag, j * mag);
            LineTo(i * mag, j * mag);
          end
    else
{If the user wants a restricting radius then draw only those lattice points that are within "n_box"}
{units of the origin.}
      begin
        for i := n_box downto -n_box do
          for j := (n_box - abs(i)) downto -(n_box - abs(i)) do
            begin
              streets[i, j].dead := false;
              streets[i, j].paths := [];
              MoveTo(x_origin + i * mag, y_origin + j * mag);
              LineTo(x_origin + i * mag, y_origin + j * mag);
            end;
{Draw the radius itself.}
        MoveTo(n_box * mag + x_origin, y_origin);
        LineTo(x_origin, n_box * mag + y_origin);
        LineTo(x_origin - n_box * mag, y_origin);
        LineTo(x_origin, y_origin - n_box * mag);
        LineTo(n_box * mag + x_origin, y_origin);
      end;
  end;{DrawLattice}

{*****************************************************************************}

  procedure FloodIt (x, y, dir : integer;
```

```
                var u, v : integer);
{=============================================================================}
{= PURPOSE: Figures the coordinates of the new point from the given dir(ection) and draws an  =}
{=          arrow or a line to that point.                                    =}
{= SUBROUTINES CALLED:  LineTo, MoveTo, Move, Line (QuickDraw).               =}
{= INPUT: x,y: integer coordinates of the current point.                      =}
{=        dir: the direction (1,2,3,or 4) to move from current point.         =}
{=        u,v: the coordinates to be calculated.                              =}
{= OUTPUT: u,v: the coordinates of the point that is one unit in dir(ection) from x,y.  =}
{= GLOBALS: arrows: boolean telling whether the user wants arrows drawn or line segments.  =}
{=============================================================================}
  begin
    u := x - (dir - 3) * ((dir mod 2) - 1);        {Compute the new point's coordinates.}
    v := y + (dir - 2) * (dir mod 2);
    MoveTo(x_origin + x * mag, y_origin + y * mag);
    LineTo(x_origin + u * mag, y_origin + v * mag);
    if arrows then
      begin
        Move(x - u, y - v);
        Line(((x - u) * 2 + (y - v) * (-2)), ((x - u) * (-2) + (y - v) * 2));
        Move(((x - u) * (-2) + (y - v) * 2), ((x - u) * 2 + (y - v) * (-2)));
        Line(((x - u) * 2 + (y - v) * (2)), ((x - u) * (2) + (y - v) * 2));
      end;
  end;{FloodIt}

{*****************************************************************************}

  function OnTheEdge (x, y : integer) : boolean;
{=============================================================================}
{= PURPOSE: Determines if a point is on the edge of the radius or the screen.  =}
{= SUBROUTINES CALLED:  abs.                                                   =}
{= INPUT: x,y: the coordinates of the point in question.                       =}
{= OUTPUT: OnTheEdge: boolean value.                                           =}
{= GLOBALS: none.                                                              =}
{=============================================================================}
  begin
    if ((abs(x) = (WIDTH div (2 * mag))) or (abs(y) > (HEIGHT div (2 * mag) - 1))) then
      OnTheEdge := true
    else if radius and ((abs(x) + abs(y)) = n_box) then
      OnTheEdge := true
    else
      OnTheEdge := false;
  end;

{*****************************************************************************}

  procedure Percolate (var streets : graph;
                       x, y : integer);
{=============================================================================}
{= PURPOSE: This is a recurrsive routine that models directed percolation.  Each point chooses one =}
{=          direction at random calling Percolate on the new point until the path dead ends on the =}
{=          edge of the screen or radius or into another path.  The recurrsion then returns to the =}
{=          last point before the dead end and chooses the second direction, repeating this       =}
```

```
{=          process until all points have chosen two directions at random.                  =}
{= SUBROUTINES CALLED:  OnTheEdge, FloodIt, Percolate.                                      =}
{= INPUT: x,y: the coordinates of the point needing directions.                             =}
{=        streets: an array of lattice point paths and whether the point has chosen 2 directions.  =}
{= OUTPUT: (Pretty pictures).                                                               =}
{= GLOBALS: none.                                                                           =}
{============================================================================}

  var
    u, v, dir : integer;
begin
  if not (OnTheEdge(x, y)) then
    while not (streets[x, y].dead) do      {while the point hasn't chosen both directions...}
      begin
        if (streets[x, y].paths = []) then      {If the point has chosen no directions...}
          begin
            dir := abs(Random) mod 4 + 1;          {Choose the primary direction.}
            streets[x, y].paths := [dir];          {Record it in the points record.}
            FloodIt(x, y, dir, u, v);              {Draw the path.}
          end
        else                                   {If the point has chosen one direction..}
          begin
            repeat                                 {Choose a different direction.}
              dir := abs(Random) mod 4 + 1;
            until not (dir in streets[x, y].paths);
            streets[x, y].paths := streets[x, y].paths + [dir];  {Add it to the point's record.}
            streets[x, y].dead := true;            {The point is now "dead".}
            FloodIt(x, y, dir, u, v)               {Draw the path.}
          end;
        Percolate(streets, u, v);              {Call Percolate on the new point.}
      end
end;{Percolate}

{**************************************************************************}

  procedure BondPercolation (var streets : graph;
                             x, y : integer);
{============================================================================}
{= PURPOSE: Simulates bond percolation on the integer lattice using a recurrsive algarithm similar=}
{=          to the above.                                                   =}
{= SUBROUTINES CALLED:  OnTheEdge, FloodIt, Percolate.                      =}
{= INPUT: x,y: the coordinates of the point needing directions.             =}
{=        streets: an array of lattice point paths and whether the point has chosen 2 directions.  =}
{= OUTPUT: (Pretty pictures).                                               =}
{= GLOBALS: none.                                                           =}
{============================================================================}
  var
    i, u, v : integer;
begin
  if not (OnTheEdge(x, y)) then
    if not (streets[x, y].dead) then
      for i := 1 to 4 do                       {Do it for all four directions.}
        if not (i in streets[x, y].paths) then     {If bond hasn't been closed/opened...}
```

```
        if abs(Random) <= (MAXINT * openprob) then    {If bond should be open...}
          begin
            FloodIt(x, y, i, u, v);                      {Draw path to new point.}
        {Add the reverse path to the new point .}
            streets[u, v].paths := streets[u, v].paths + [(i - (i div 3) * (4) + 2)];
            if i = 4 then                                {If all paths are computed...}
              streets[x, y].dead := true;                   {Point is dead.}
            BondPercolation(streets, u, v);              {Call recurrsion.}
          end
        else                                          {If bond should be closed...}
          begin
            u := x - (i - 3) * ((i mod 2) - 1);          {Calculate opposite point.}
            v := y + (i - 2) * (i mod 2);
            streets[u, v].paths := streets[u, v].paths + [(i - (i div 3) * (4) + 2)];{Add reverse path.}
          end;
  end;{BondPercolation}

{*****************************************************************}

begin
  bond := ['b', 'B'];
  yes := ['y', 'Y'];
  RandSeed := TickCount div 2;

  showtext;
  write('Bond or Directed Percolation?(B/D):');
  readln(Perc_choice);
  If not (Perc_choice in bond) then
    begin
      write('Arrows?(y/n):');
      readln(respns);
      arrows := respns in yes;
    end
  else
    begin
      write('What probability?:');
      readln(openprob);
    end;

  write('What magnification?(3-10):');
  readln(mag);
  x_origin := (WIDTH div (2 * mag)) * mag;
  y_origin := (HEIGHT div (2 * mag)) * mag;

  write('Restricting Radius?(y/n):');
  readln(respns);
  radius := respns in yes;
  If radius then
    begin
      write('Radius =(2-46)');
      readln(n_box);
    end;
```

```
  while not (GetNextEvent(2, Event)) do      {Repeat until mouse is clicked.}
    If perc_choice in bond then
      begin
        arrows := false;
        WindowSetup;
        DrawLattice(streets);
        BondPercolation(streets, 0, 0);
      end
    else
      begin
        WindowSetup;
        DrawLattice(streets);
        Percolate(streets, 0, 0);
      end;

end.{D2dPerc}
```



Bond Percolation at p=.60
(origin at center)

We first looked at the models as a physical system with the latticed comparable to a porous substance and C_0 the set of wet sights if a fluid source existed at 0. With this in mind the comparison dealt only with the "filling" of the screen. A glance at the two pictures above shows what a striking change there is when p differs by only one one-hundredth of a percent. Below is a magnified simulation of directed percolation with arrows to indicate directions and another simulation(without arrows) at the same magnification as the previous bond percolation pictures. A visual comparison would seem to indicate that the extent to which directed percolation fills the plain can be related to bond percolation with a $.50 < p < .60$.