

Chapter1

INTRODUCTION

1. Percolation Problem

Given a composite system composed of randomly distributed insulating and metallic materials: what fraction of the materials need to be metallic so that the composite system is an electrical conductor? Given a porous landscape with water on the surface (or oil below), under what conditions will the water be able to drain through to the bottom (or the oil to gush through to the surface)? Scientists have defined an abstract process known as *percolation* to model such situations. ([cs.princeton.edu](https://www.cs.princeton.edu/~bss/) B. S., 2008)

2. Physics with Percolation

Percolation theory applies to many disparate physical phenomena such as polymeric gelation, crystalline impurities, as well as disease propagation through an orchard. One appealing aspect of percolation theories is their universality, in that their behavior depends only upon the spatial dimension close to criticality and thus, different systems display similar behavior. Hence percolation theory is applicable in a wide range of problems in physics as well as other fields, in physics percolation theory is mostly applicable in material science. More practical applications of percolation theory are discussed below. ([mit.edu](https://www.mit.edu/~dave/teaching/8.04/percolation/))

3. Percolation Problem Applications

Percolation is a model for disordered media. Below, we consider some applications of percolation theory that illustrate the modeling aspects of percolation in diverse situations.

1. Forest fires

The propagation of a fire through a forest can be modeled using percolation. In the simplest model, the forest is represented by a lattice, whose squares are either occupied by a tree or not. Each square can be in one of three states: (a) non burnt tree, (b) a burning tree, or (c) no-tree (either already burnt or there was never a tree there). The spread of fire can be simulated as follows: The ignition point can be set or is chosen randomly to model randomness as is the case of fire ignition by a lightning. Once a tree is burning, there is a probability p that fire will spread to a neighboring tree. As soon as the fire spreads to another tree, the state of that tree changes to "burning" and that tree can now propagate the fire to another neighboring tree. The spread of fire is simulated by marking certain sites in the lattice as burning and following an iterative procedure that checks which other trees or sites are going to burn. Varying the conditions that are used to decide whether a tree propagates the fire to its neighbors (probability p) one can account for static attributes, such as fuel type, elevation and slope, as well as dynamic attributes, such as the direction of the wind or the humidity of the air and the temperature. A cluster in this case is a set of trees corresponding to trees that burnt or are burning. One may be interested in determining the probability that a certain point is reached by the burning cluster. Following this simplistic lattice structure, in the next link Forest Fire Modeling we explore a more complex structure, the Voronoi structure, for forest representation. ([D Stauffer, 2018](#))

2. Oil fields

Percolation theory can be used as a simple idealized model for predicting the distribution of the oil or gas inside porous rocks or oil reservoirs¹. The network considered in this case is formed from the pores filled with oil in a rock. The pores can be connected, which would correspond to open bonds in the percolation model, or can be isolated. To the probability with which a site is occupied in the percolation problem it corresponds to the porosity or the average concentration of oil in the rock. In order to obtain a good oil production from a well it is desirable to position it in an area with high porosity. In order to predict the amount of oil that will be produced one needs to estimate the porosity of the rock in the area where the oil reservoir is assumed to be located. To obtain such an estimation, the porosity of rock samples is determined. The difficulty comes from the fact that the samples are usually rock logs with the diameter in the order of centimeters, so in order to obtain the final result one needs to extrapolate the measurement to the scale of the reservoir, which is at least in the order of kilometers. The fundamental question is whether such an extrapolation is legitimate. Percolation theory predicts that this approach is valid when the probability p in the percolation problem is appreciably higher than the percolation threshold p_c . On the other hand, if the probability is close to the threshold, even though there might exist an extended cluster that could be quite ramified containing a lot of 'holes', it is possible that the sample will contain other clusters that unfortunately cannot be reached. In this case the decision to invest would prove to be not profitable. (D Stauffer, 2018)

3. Electrical resistance in a mixture of two media

Consider mixing two materials A, and B, that are in the form of tiny particles, where A is a conducting material while B is a perfect insulator. Make a compact cube of such a mixture and apply a potential differential to two of the opposite faces of the cube. If the mixture is disordered, it may be reasonable to assume that each tiny particle in the mixture is chosen at random to be of either type A or type B. Neighboring type A particles form clusters that conduct electricity. If such a percolating cluster exists that spans the two opposite faces (electrodes), the electrical resistance of the medium is finite, otherwise is infinite. This in turn depends on the ratio of the particles of type A in the mixture, i.e. probability p . There exists a critical probability, p_c , close to .50, such that for $p < p_c$ the resistance is infinite, while for $p > p_c$ the resistance is finite. (D Stauffer, 2018)

4. Gelation and Polymerization

Consider boiling an egg or more precisely the process through which the yolk and the albumen change their consistency. This phenomenon is in fact gelation, the formation of a network of chemical bonds, which span the whole system. The interest in such problems started with the study of polymerization. The initial attempts tried to understand how small branching molecules form larger and larger macromolecules if more and more chemical molecules are formed between the original molecules. In this situation the initial small molecules correspond to the sites of the lattice and the macromolecules to the clusters of the lattice. (D Stauffer, 2018)

4. Experiments with Numerical Simulation

A numerical simulation is a calculation that is run on a computer following a program that implements a mathematical model for a physical system. Numerical simulations are required to study the behavior of systems whose mathematical models are too complex to provide analytical solutions, as in most nonlinear systems.

Computer simulation developed hand-in-hand with the rapid growth of the computer, following its first large-scale deployment during the Manhattan Project in World War II to model the process of nuclear detonation. It was a simulation of 12 hard spheres using a Monte Carlo algorithm. Computer simulation is often used as an adjunct to, or substitute for, modeling systems for which simple closed form analytic solutions are not possible. There are many types of computer simulations; their common feature is the attempt to generate a sample of representative scenarios for a model in which a complete enumeration of all possible states of the model would be prohibitive or impossible. (Bratley, Fox, & Schrage, (2011-06-28))

Computer simulations are used in a wide variety of practical contexts, such as:

- analysis of air pollutant dispersion using atmospheric dispersion modelling
- design of complex systems such as aircraft and logistics systems.
- design of noise barriers to effect roadway noise mitigation
- modelling of application performance^[15]
- flight simulators to train pilots
- weather forecasting
- forecasting of risk
- simulation of electrical circuits
- Power system simulation

4. 1 Numerical simulation approaches for percolation problem

There are a wide variety of algorithms to simulate and model the percolation phenomenon. However it is important for the algorithm to be as efficient as possible because we are dealing with lattices of large dimensions.

Depth First Search

One of the simple algorithms to model percolation is using Depth-first search (DFS). In DFS,

- Try searching from all of the open spots on the top row.
- Search from all legal adjacent spots you have not visited.
- Recurse until you can't search any further or have reached the bottom row.

Although DFS is simple, due to the use of recursion, as the number of occupied sites and dimension of the lattice becomes larger, the number of recursive instances increase exponentially, thus DFS is not suitable for simulation of large lattices.

Efficient Algorithms – Union Find

- Give unique ID to each cell
- Populate individual cells
- Check for neighbor, if yes allot same ID to both cells
- Repeat until ID of top row cell = ID of bottom row cell

Union find is many times faster than DFS and can handle much larger lattices using the same hardware resources. (cs.princeton.edu, 2020)

4. Random Number

Randomness is important while modeling real world phenomenon, in the case of percolation, sites in the lattice need to be randomly occupied, hence, there is a requirement for generation of a random number each time a cell in the lattice is processed.

In Fortran95 we have the *rand(seed)* function which takes a seed value as a parameter to produce a random real value between 0.00 and 1.00. The seed has to be unique for each random number, so some varying metric such as system date and time can be used as a seed value. This is elaborated in section 2.2.

5. Fortran Programming

Fortran is one of the oldest computer languages. It was developed in the 1950s by IBM for numeric calculations (Fortran is an abbreviation of “Formula Translation”). Despite its age, it is still used for high-performance computing such as weather prediction. However, the language has changed considerably over the years, although mostly maintaining backwards compatibility; well-known versions are FORTRAN 77, Fortran 90, Fortran 95, Fortran 2003, Fortran 2008 and Fortran 2015. ([learnxinyminutes](https://learnxinyminutes.com), 2020)

Fortran is the primary language for some of the most intensive super-computing tasks, such as in astronomy, climate modelling, computational chemistry, computational economics, computational fluid dynamics, computational physics, data analysis, hydrological modelling, numerical linear algebra and numerical libraries (LAPACK, IMSL and NAG), optimization, satellite simulation, structural engineering, and weather prediction.[citation needed] Many of the floating-point benchmarks to gauge the performance of new computer processors, such as the floating-point components of the SPEC benchmarks (e.g., CFP2006, CFP2017) are written in Fortran. Fortran95 is used in this project. (Bratley, Fox, & Schrage, (2011-06-28))

The overview below will discuss some of the essential syntax of Fortran 95 used in this project, Fortran95 was chosen since it is the most widely implemented of the more recent specifications and the later versions are largely similar. *(by comparison FORTRAN 77 is a very different language)*

```

! This is a comment.
program example !declare a program called example.
! Code can only exist inside programs, functions, subroutines or modules.
! Using indentation is not required but it is recommended.
! All declarations must come before statements and expressions.
implicit none !prevents dynamic declaration of variables (recommended!)
! Implicit none must be redeclared in every function/program/module...
! IMPORTANT - Fortran is case insensitive.
real z
REAL Z2
real :: v,x ! WARNING: default initial values are compiler dependent!
real :: a = 3, b=2E12, c = 0.01
integer :: i, j, k=1, m
real, parameter :: PI = 3.1415926535897931 !declare a constant.
real :: array(6) !declare an array of 6 reals.
real, dimension(4) :: arrayb !another way to declare an array.
integer :: arrayc(-10:10) !an array with a custom index.
Z = 1 !assign to variable z declared above (case insensitive).
j = 10 + 2 - 3
a = 11.54 / (2.3 * 3.1)
! Control Flow Statements & Operators
if (z == a) b = 4 !condition always need surrounding parentheses.
if (z /= a) then !z not equal to a
! Other symbolic comparisons are < > <= >= == /=
b = 4
else if (z .GT. a) then !z greater than a
! Text equivalents to symbol operators are .LT. .GT. .LE. .GE. .EQ. .NE.
b = 6
else if (z < a) then !'then' must be on this line.
b = 5 !execution block must be on a new line.
else
b = 10
end if !end statement needs the 'if' (or can use 'endif').
! Arrays
array = (/1,2,3,4,5,6/)
array = [1,2,3,4,5,6] !using Fortran 2003 notation.
arrayb = [10.2,3e3,0.41,4e-5]
! Fortran array indexing starts from 1.
v = array(1) !take first element of array.
print *, array(3:5) !print all elements from 3rd to 5th (inclusive).
print *, array2d(1,:) !print first column of 2d array.
! Input/Output
print *, b !print the variable 'b' to the command line.
print "(I6)", 320 !prints ' 320' ! We can format our printed output
print "(I6.4)", 3 !prints ' 0003'
print "(F6.3)", 4.32 !prints ' 4.320'
read *, v ! We can also read input from the terminal
read "(2F6.2)", v, x !read two numbers
! To read a file.
open(unit=11, file="records.txt", status="old")
! The file is referred to by a 'unit number', an integer that you pick in
! the range 9:99. Status can be one of {'old','replace','new'}.
read(unit=11, fmt="(3F10.2)") a, b, c
close(11)
! To write a file.
open(unit=12, file="records.txt", status="replace")
write(12, "(F10.2,F10.2,F10.2)") c, b, a
close(12)
m = func(3,2,k) !function call.
Print *, func2(3,2,k)
m = func3(3,2,k)
function func2(a,b,c) result(f) !return variable declared to be 'f'.
implicit none
integer, intent(in) :: a,b
integer, intent(inout) :: c
integer :: f !function return type declared inside the function.
integer :: cnt = 0
f = a + b - c
c = 4 !altering the value of an input variable.
cnt = cnt + 1 !count number of function calls.
end function func2
end program example

```

Chapter-2

METHODS

1. The Problem Statement and Model

Here we model percolation using a 2-D matrix and a DFS algorithm. First the entire matrix is populated according to a site occupation probability in the range 0-1. This is done by weighting a random number; a cell is said to be occupied if the random number generated is equal to or above the site occupation probability. A new random number is generated for each cell until the matrix has been traversed. We start off by setting the site occupation probability to 0 and slowly increment with a constant step size until it reaches 1. Once the matrix is populated for a given site occupation probability a DFS algorithm is used to check for a path connecting the top row and the bottom row via occupied sites. Once a path is found, the site occupation probability is recorded to a txt file and the program returns to its initial state. This process is iterated for a certain number of times for consistency and an average of the recorded site occupation probabilities is evaluated to obtain the critical probability of percolation for that matrix size. (cs.usfca.edu, n.d.)

2.Setup

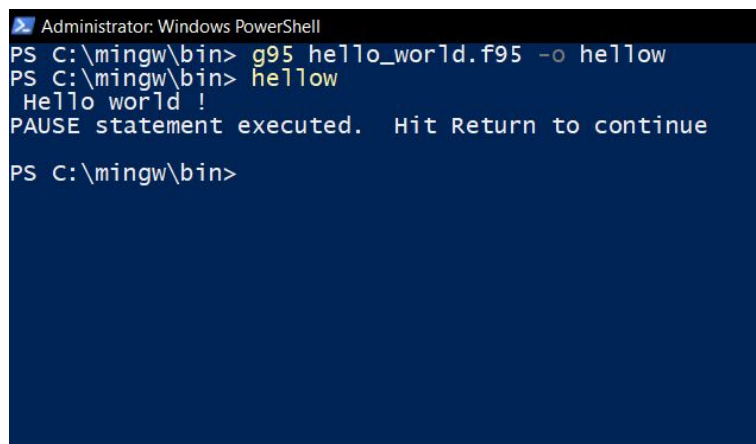
To work with Fortran95, very little overhead is necessary, the G95 compiler on its own is sufficient, although a text editor like [Notepad++](#) or an IDE like [SciTE](#) is recommended for ease of use.

2.1 Installing Fortran in Windows and running programs.

The G95 compiler binary from 2012 is available on [SourceForge](#).

G95 is a free open source Fortran 95 compiler. It implements the Fortran 95 Standard, some parts of the Fortran 2003 Standard and a few extensions.

Download and run the installer, once finished, go to [C:\mingw\bin](#) to see your program files with [g95](#) extension. To compile and run programs open command prompt in the same directory and type [g95 fortranfile.f95 -o outputfilename](#). Once compile is finished you will see a .exe file with the given name, run it by typing its name and hitting enter in cmd.



```
Administrator: Windows PowerShell
PS C:\mingw\bin> g95 hello_world.f95 -o hellow
PS C:\mingw\bin> hellow
Hello world !
PAUSE statement executed. Hit Return to continue
PS C:\mingw\bin>
```

Figure 2.1: G95 on windows PowerShell

2.2 Hello World in Fortran95.

A hello world example in Fortran95.

```
1. program hello_world
2. print *, "Hello world ! "
3. OPEN(UNIT=1,FILE="data1.txt",position='append',FORM="FORMATTED",STATUS="OLD") ! open
   textfile
4. write(1,*)"Hello world ! "
5. pause
6. end program hello_world
```

2.3 Random Number generation with Fortran

In Fortran95 we have the *rand(seed)* function that takes a ‘seed’ value as argument and returns a random number between 0 and 1 corresponding to that seed. The seed must be unique for each random number to be generated, hence we need some varying metric to be passed as seed to the *rand()* function, one way is to use system time as seed which we will be using and is demonstrated in the code below.

2.3.1 Generating a random number between 0 and 1 using *rand()*

```
1. Program random_example
2. Implicit::none
3. real :: inp1,inp2,rno1,rno2,rno3
4. integer :: key
5. character(8) :: date
6. character(10) :: time
7. character(5) :: zone
8. call date_and_time(date,time,zone)
9. call date_and_time(DATE=date,ZONE=zone)
10. call date_and_time(TIME=time)
11. read( date, '(f10.0)' ) rno1
12. read( time, '(f10.0)' ) rno2
13. read( zone, '(f10.0)' ) rno3
14.
15. !Current date and time are used as seed for the random number generator
16.
17. rno3=rno1/rno3+rno2
18. rno3=rno3+m
19. rno3=rand(int(rno3))
20. !rno3 now contains a random number between 0 and 1
21.
22. print*,"Random number : ",rno3
23.
24. end program random_example
```

The above code generates a random number between 0 and 1, this random number can be weighted according to our site occupation probability to decide whether a cell in the matrix can be considered occupied or not occupied. This is used in the next section to populate an entire matrix according to the site occupation probability

2.4 Fortran program to generate an $n \times n$ array with site occupation

The below program dynamically creates an $n \times n$ array, traverses it cell by cell and each time a new random number is generated to deem the cell as occupied or not occupied according to the site occupation probability given as input. The final populated matrix is then displayed on console and written to a text file for further reference.

```
1. integer function getkey(inp1,inp2,m)
2.     real :: inp1,inp2,rno1,rno2,rno3
3.     integer :: key
4.     character(8) :: date
5.     character(10) :: time
6.     character(5) :: zone
7.
8.     call date_and_time(date,time,zone)
9.     call date_and_time( DATE=date, ZONE=zone)
10.    call date_and_time( TIME=time)
11.    read( date, '(f10.0)' ) rno1
12.    read( time, '(f10.0)' ) rno2
13.    read( zone, '(f10.0)' ) rno3
14.
15.    !Current date and time are used as seed for the random number generator
16.
17.    rno3=rno1/rno3+rno2
18.    rno3=rno3+m
19.    rno3=rand(int(rno3))
20.    !rno3 now contains a random number between 0 and 1
21.    if (rno3 .GE. 0 .AND. rno3 .LE. inp1) then
22.        key = 0 !Cell is empty
23.    end if
24.
25.    if (rno3 .GE. inp1 .AND. rno3 .LT. 1) then
26.        key=1 !Cell is populated
27.    end if
28.    getkey = key !key is returned
29.    key=0
30.end function getkey
31.
32.program perc_occupy
33.implicit none
34.
35.integer, dimension (:,:), allocatable :: darray
36. !dimension is 2, but size is determined at runtime
37.integer :: s1,s2,i,j,key,getkey,m=0
38.real::inp1, inp2
39.print*, "Enter site occupation probability:"
40.read*, inp1
41.inp2=1-inp1
42.print*, "Enter the dimensions of the array:"
43.read*, s1
44.s2=s1
45.allocate ( darray(s1,s1) )
46. ! dynamically allocate the array
47.do i = 1, s1
48.    do j = 1, s2
49.        m=m+50
50.        key=getkey(inp1,inp2,m)
51.
52.        print *, key
53.        if (key .EQ. 0) then
```



```

54.          darray(i,j)=0 !set array elements
55.          end if
56.          if (key .EQ. 1) then
57.              darray(i,j)=1 !set array elements
58.          end if
59.END DO
60.END DO
61.! populated the cells of the array according to probability using getkey()
62.OPEN (UNIT=1,FILE="data1.txt",FORM="FORMATTED",STATUS="OLD") !open textfile
63.do i = 1, s1
64.    write(UNIT=1,FMT="(A)",advance="yes") " " !put linebreak
65.    print*," " !put linebreak in console
66.    do j = 1, s2
67.        if(darray(i,j).EQ.1) then
68.            write(UNIT=1,FMT="(A)",advance="no") "1" !write to txt
69.        end if
70.        if(darray(i,j).EQ.0) then
71.            write(UNIT=1,FMT="(A)",advance="no") "0" !write to txt
72.        end if
73.        write(UNIT=1,FMT="(A)",advance="no") " " !space
74.        write(*,"(I2)",advance="no") darray(i,j) !output matrix to console
75.        write(*,"(A)",advance="no") " " !space
76.END DO
77.END DO
78.print*," " !linebreak
79.close(1)
80.PAUSE
81.END program perc_occupy

```

For array size 10, the matrices generated from the above code for different site occupation probabilities are shown below.

(Paths from top row '1's to bottom row '1's are manually highlighted in red)

```
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
```

P = 0

```
0 0 0 0 0 1 0 0 0 0
1 0 0 0 1 0 0 0 0 0
0 0 1 0 0 0 1 0 0 1
0 1 0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 0 0 1
0 0 1 0 0 0 0 0 1 0
0 1 0 0 0 0 1 1 1 1
0 0 0 0 1 0 0 0 0 0
0 0 0 1 0 0 1 0 0 0
0 0 0 0 0 0 0 0 0 1
```

P = 0.25

```
0 0 1 1 1 1 0 1 1 1
1 1 1 1 1 1 1 1 0 1
1 1 1 1 0 0 1 1 1 1
1 0 1 1 0 1 1 1 1 1
0 1 1 1 0 1 1 1 1 1
1 1 1 1 1 0 1 1 0 1
1 1 1 1 1 1 1 1 1 0
1 0 0 0 0 0 0 0 1 1
1 0 1 1 0 1 1 1 0 1
1 1 1 1 0 0 1 1 1 1
```

P = 0.75

```
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
```

P = 1

```
0 0 1 0 1 0 0 1 0 0
0 1 0 0 1 1 0 0 0 0
0 0 0 0 0 1 0 0 0 0
0 0 1 0 0 1 0 1 0 1
1 0 0 0 0 1 0 0 0 0
0 0 1 0 1 1 0 0 1 0
0 0 0 0 1 0 0 0 0 0
1 0 1 0 1 0 0 1 0 1
0 0 0 1 1 0 0 1 0 0
1 0 0 1 0 0 0 0 0 1
```

P = ??

(Simulated critical probability condition)

2.4 Checking for a path – Depth-First Search

Now, in order to find the critical probability, we need to determine the probability of site occupation at which a single path occurs, i.e., we need to automatically detect whether such path(s) exists in the generated matrix for a given site occupation probability, for this we use the depth-first search algorithm. The DFS algorithm here traverses the matrix, following paths connected by 1's, starting from the first row until the last row is reached and a path is found OR the end of the first row is reached and no path is found. The basic working and control-flow of the recursive DFS algorithm is illustrated in the flowchart below.

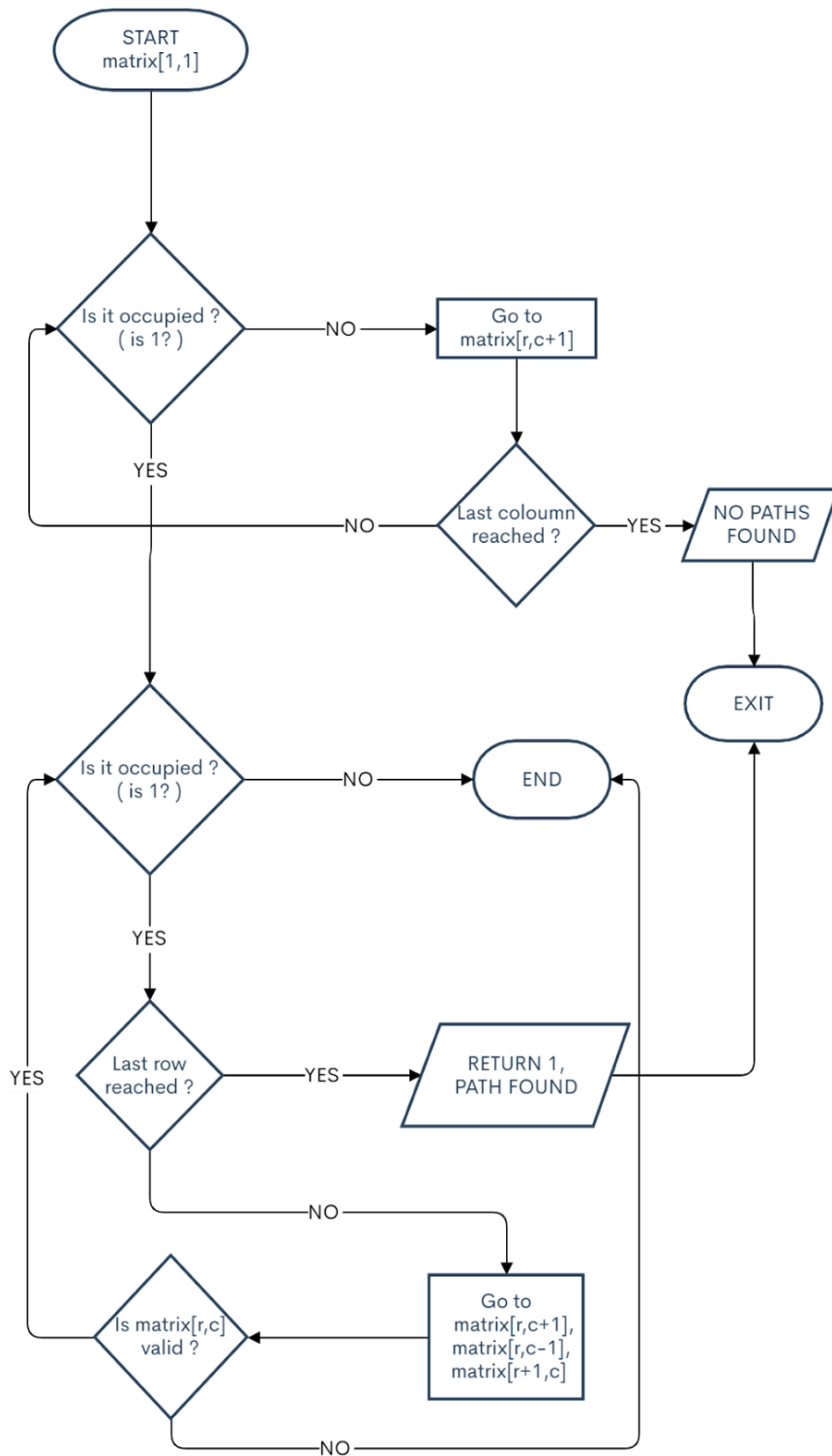


Figure 2.4.1: Flowchart of the DFS algorithm used to find a path

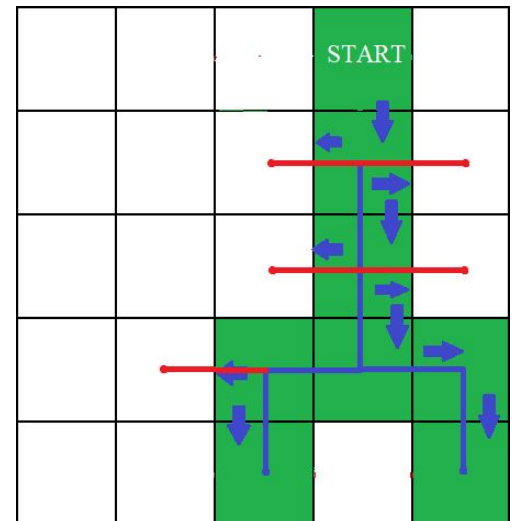


Figure 2.4.2: The depth first traversal of the recursive function is shown below. The direction of traversal is shown by the blue arrows, note how if traversal is towards left, it doesn't go towards right and vice versa, this is to avoid an infinite loop. Programmatically, this is accomplished by the use of a direction variable that keeps track of the direction of traversal. Refer sec. 2.5, lines 21-30.

2.5 Counting Paths using DFS.

This DFS algorithm has been implemented in the Fortran code below to traverse the array recursively and count the total number of paths in a given matrix. The maximum number of possible paths (at $P=1$) for a matrix of dimension n , according to our definition of a path would be n^n .

```

1. function count_paths(matrix2) result(sum) !sum is being returned
2. implicit none
3.   integer:: i, path_finder,sum,matrix2(4,4)
4.   sum=0
5.   do i = 1,4
6.     sum = sum+ path_finder(1, i,matrix2,0) !initial direction set as down, so no
       restrictions
7.   !path count is incremented by 1 if u=1 is returned by path-finder function
8.   end do
9. end function count_paths
10.
11.recursive function path_finder (r ,c,matrix2,dir) result(u) !u is being returned
12.implicit none
13.   integer:: r,c,u,matrix2(4,4),r1,c1,dir
14.   u=0
15.   !Reset u for the next iteration
16.   if (r.LE.4 .AND. c.LE.4 .AND. matrix2(r,c).EQ.1 .AND. r.GE.1 .AND. c.GE.1 ) then
17. !check if row,coloumn in bounds and whether cell contains '1'
18.   if(r.EQ.4) then
19.     u=1 !u = 1 if we've reached the last row
20.   else
21. !following the path of occupied sites downwards and laterally
22.   if(dir.EQ.-1)Then
23.     u=path_finder(r,c-1,matrix2,-1)+path_finder(r+1,c,matrix2,0)
24.   endif
25.   if(dir.EQ.1)then
26.     u=path_finder(r,c+1,matrix2,1)+path_finder(r+1,c,matrix2,0)
27.   endif
28.   if(dir.EQ.0)then
29.
30.     u=path_finder(r,c-1,matrix2,-1)+path_finder(r+1,c,matrix2,0)+path_finder(r,c+1,matri
       x2,1)
31.   endif
32.   end if
33.end function path_finder
34.
35.program path
36.implicit none
37.integer darray(4,4)
38.integer :: i,j,count_paths,p,element
39.print*,"Enter the 4x4 array "
40.do i = 1, 4
41.  do j = 1, 4
42.    read*,element
43.    darray(i,j)=element !input array
44.END DO
45.END DO
46.p=count_paths(darray) !calling the function to count paths
47.print*,"Number of Paths = ",p
48.close(1)
49.PAUSE
50.end program path

```

Since the DFS algorithm here relies on recursion, as the size of the matrix increases, the number of paths increase exponentially, thus counting all paths for larger matrices is very inefficient due to the large number of recursive instances of the function, requiring a huge amount of processing power.

But these issues can be circumvented as we are dealing with the critical probability, where only one path be counted, and by working with smaller matrix sizes.

2.6 Fortran program to find critical probability.

The below program calculates the critical probability of an NxN matrix by calculating the average probability of site occupation for which a single path is found.

```
1. integer function getkey(inp1,inp2,m)
2.
3.     real :: inp1,inp2,rno1,rno2,rno3
4.     integer :: key
5.     character(8) :: date
6.     character(10) :: time
7.     character(5) :: zone
8.
9.     call date_and_time(date,time,zone)
10.    call date_and_time(DATE=date,ZONE=zone)
11.    call date_and_time(TIME=time)
12.    read( date, '(f10.0)' ) rno1
13.    read( time, '(f10.0)' ) rno2
14.    read( zone, '(f10.0)' ) rno3
15.
16.    !Current date and time are used as seed for the random number generator
17.
18.    rno3=rno1/rno3+rno2
19.    rno3=rno3+m
20.    rno3=rand(int(rno3))
21.
22.
23.    !rno3 now contains a random number between 0 and 1
24.
25.    if (rno3 .GE. 0 .AND. rno3 .LE. inp1) then
26.        key = 0 !Cell is empty
27.    end if
28.
29.    if (rno3 .GE. inp1 .AND. rno3 .LT. 1) then
30.        key=1 !Cell is populated
31.    end if
32.    getkey = key !key is returned
33.    key=0
34.end function getkey
35.
36.
37.function count_paths(matrix2,s1,s2) result(printer) !sum is being returned
38.implicit none
39.    integer:: i, path_finder,sum,matrix2(s1,s2),s1,s2,printer
40.    printer=0
41.    sum=0
42.
43.    do i = 1,s1
44.        sum = sum+ path_finder(1, i,matrix2,0,s1,s2)
45.    !initial direction set as down, so no restrictions
46.
47.    !path count is incremented by 1 if u=1 is returned by path-finder function
48.        if(sum.GT.0) then
49.            printer=1
50.            exit
51.        endif
52.    end do
53.end function count_paths
54.
```

```

55.
56. recursive function path_finder (r ,c,matrix2,dir,s1,s2) result(u) !u is being returned
57. implicit none
58.   integer:: r,c,u,matrix2(s1,s2),r1,c1,dir,s1,s2
59.   u=0
60.
61.   !Reset u for the next iteration
62.   if (r.LE.s1 .AND. c.LE.s2 .AND. matrix2(r,c).EQ.1 .AND. r.GE.1 .AND. c.GE.1 ) then
63. !check if row,coloumn in bounds and whether cell contains '1'
64.   if(r.EQ.s1) then
65.     u=1 !u = 1 if we've reached the last row
66.   else
67. !following the path of occupied sites downwards and laterally
68.   if(dir.EQ.-1)Then
69.     u=path_finder(r,c-1,matrix2,-1,s1,s2)+path_finder(r+1,c,matrix2,0,s1,s2)
70.   endif
71.   if(dir.EQ.1)then
72.     u=path_finder(r,c+1,matrix2,1,s1,s2)+path_finder(r+1,c,matrix2,0,s1,s2)
73.   endif
74.   if(dir.EQ.0)then
75.     u=path_finder(r,c-1,matrix2,-1,s1,s2)+path_finder(r+1,c,matrix2,0,s1,s2)+path_finder(r,
       c+1,matrix2,1,s1,s2)
76.   endif
77.
78.   end if
79.   end if
80. end function path_finder
81.
82.
83. program path
84. integer, dimension (:,:), allocatable :: darray
85. !dimension is 2, but size is determined at runtime
86.
87. integer :: s1,s2,i,j,key,getkey,m=0,count_paths
88. real::inp1, inp2,x
89. inp1=1-inp2
90. print*, "Enter the dimensions of the array:"
91. read*, s1
92. s2=s1
93. allocate ( darray(s1,s1) )
94. ! dynamically allocate the array
95. print*, "Enter the step size (0-1):"
96. read*, x
97. print*, "Enter number of iterations(for averaging) :"
98. read*, av
99. cprob=0
100. OPEN(UNIT=1,FILE="data1.txt",position='append',FORM="FORMATTED",STATUS="OLD") ! open
    textfile
101. write(1,*)"Step size : ",x," Array size : ",s1
102. write(1,*)"-----"
103. close(1)
104.
105. do g=1,av,1
106. do inp2=0.00,1.00,x
107. inp1=1-inp2
108. do i = 1, s1
109.   do j = 1, s2
110.     m=m+50
111.     key=getkey(inp1,inp2,m)
112.
113.     !print *, key
114.     if (key .EQ. 0) then

```

```

115.          darray(i,j)=0 !set array elements
116.          end if
117.          if (key .EQ. 1) then
118.              darray(i,j)=1 !set array elements
119.          end if
120.      END DO
121.  END DO
122.
123.  p=count_paths(darray,s1,s2) !calling the function to count paths
124.  !print*, p
125.
126.  if(p.EQ.1) then
127.      cprob=cprob+inp2
128.      OPEN(UNIT=1,FILE="data1.txt",position='append',FORM="FORMATTED",STATUS="OLD") !open
        textfile
129.      write(1,*)". ",inp2
130.      !print*, "p is 1"
131.      exit
132.  endif
133.  end do
134.  end do
135.  cprob=cprob/av
136.  OPEN(UNIT=1,FILE="data1.txt",position='append',FORM="FORMATTED",STATUS="OLD") !open
        textfile
137.  write(1,*)"Critical probability : ",cprob
138.  close(1)
139.
140.  print*, "Data sucessfully written to data1.txt. Critical probability : ",cprob
141.
142.  end program path

```

USAGE

1. *INPUTS: Dimensions of the array, step size(precision), number of iterations (for averaging P_c)*
2. *Use extra argument `-freal-loops` during compile time to account for real loop variables in Fortran95.*
3. *Results are stored in 'data1.txt', make sure file exists in directory. (C:\minigw\bin\)*

Chapter-3

RESULTS AND DISCUSSIONS

1. Critical probability vs Array size

Using the code discussed at sec. 2.6, the critical probabilities of arrays of sizes 2x2 through 25x25 were determined, with a step size of 0.01 and iteration count of 100 for averaging the results. The below graph in Fig.1 was plotted using this data taking array size on X and critical probability on Y axes respectively.

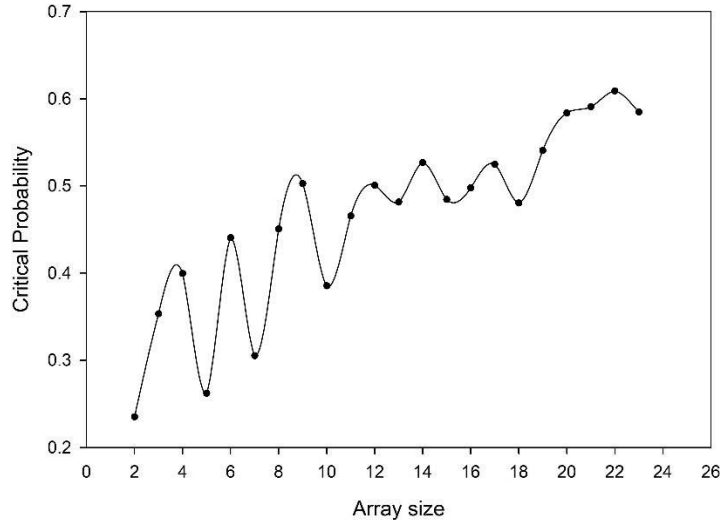


Figure 1: Critical probability vs array size for 100 iterations.

Here, the value of critical probability initially shows large variation when the size of the array is small, but as the array size is increased, the variation is reduced. This can be better observed in Fig 2 which shows the variation of the value for critical probability through these 100 iterations, for convenience only array size increments of 5 are shown.

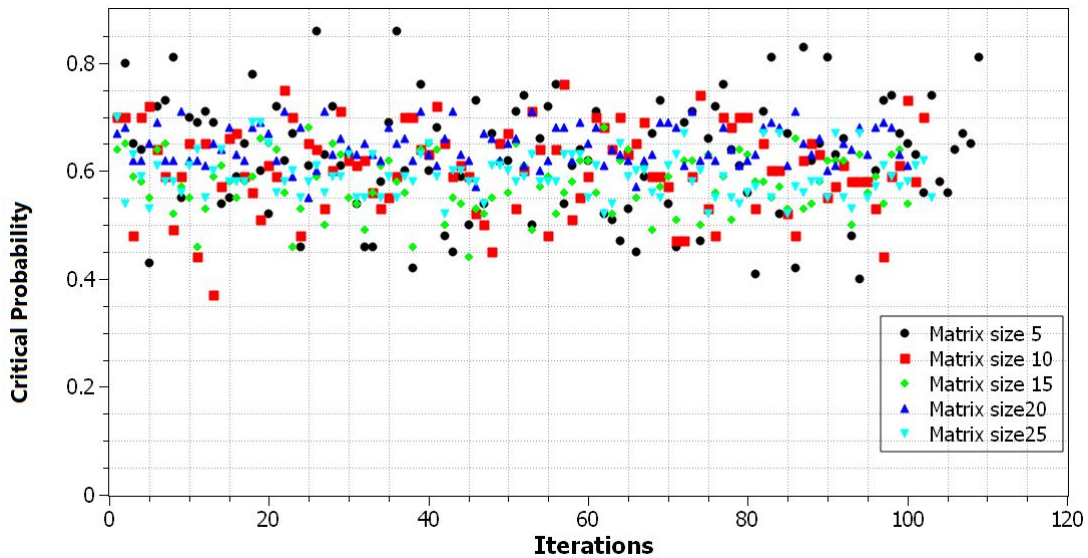
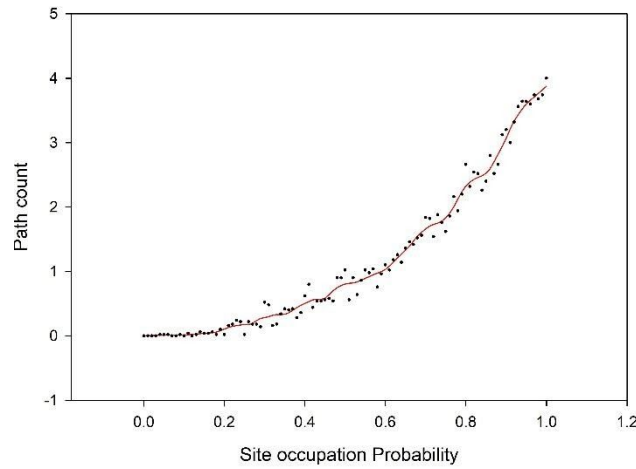


Figure 2: Variation of critical probability between each iteration.

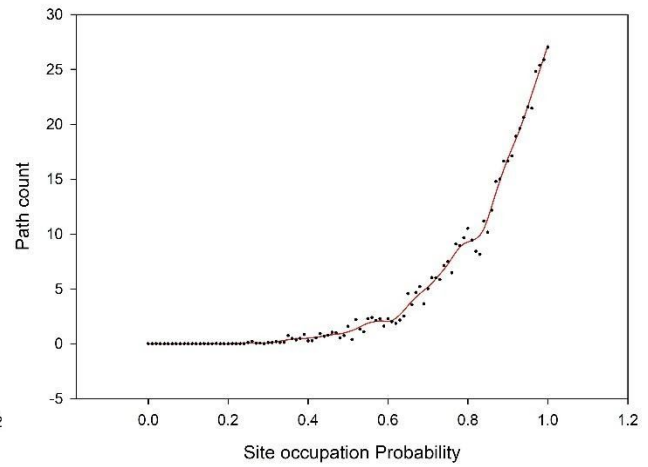
It can be observed that the calculated critical probability values for a larger array size such as 25 are more closely grouped than for a smaller array size like 5.

2. Number of paths vs critical probability

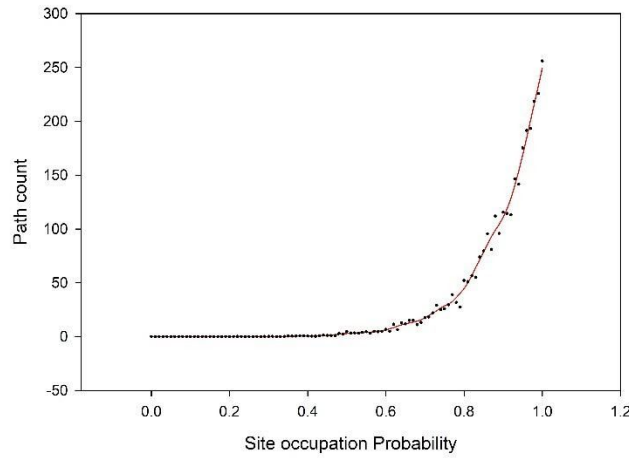
By removing the *exit* statement at line 121 in sec.2.6 we can obtain the corresponding number of paths formed for each site occupation probability. And the increase in number of paths as site occupation probability exceeds the critical probability can be observed for different array sizes.



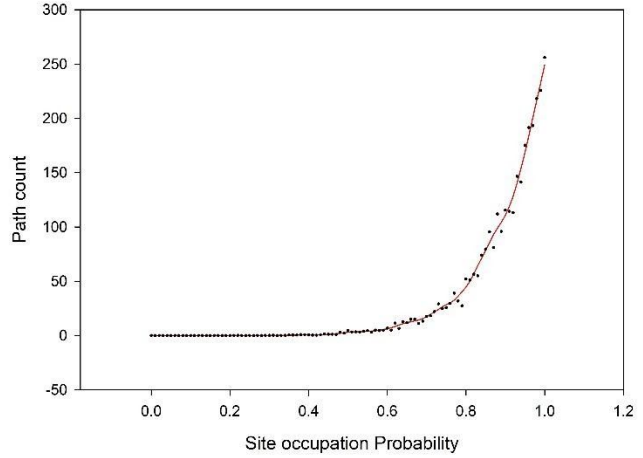
2x2



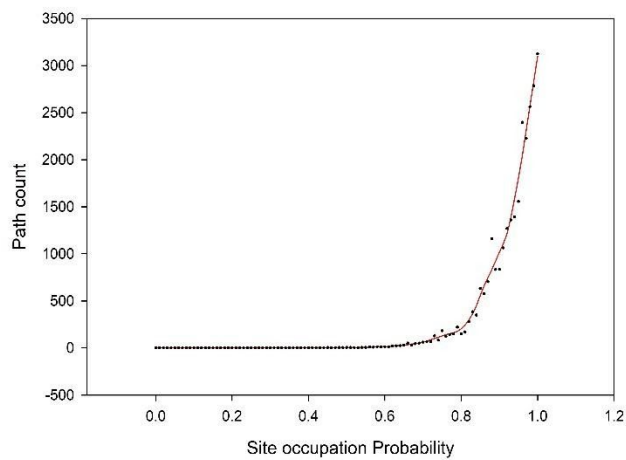
3x3



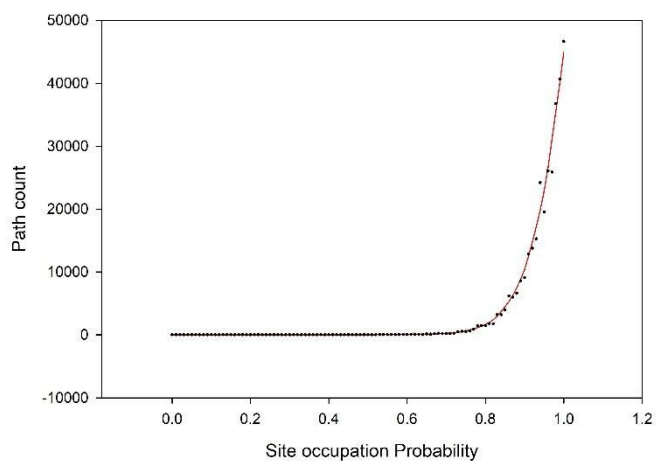
4x4



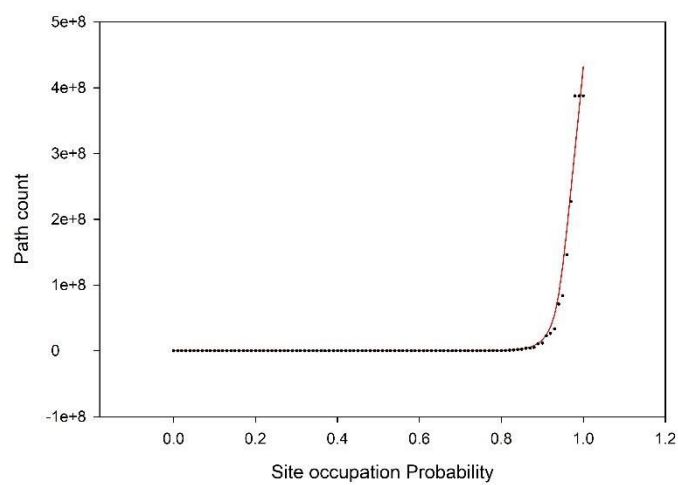
5x5



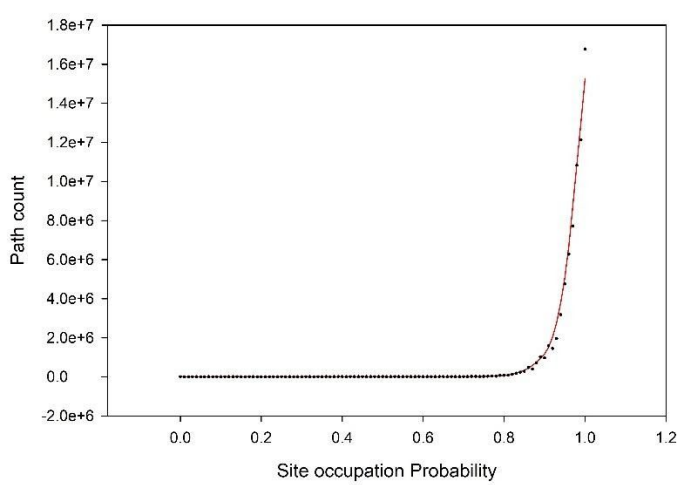
6x6



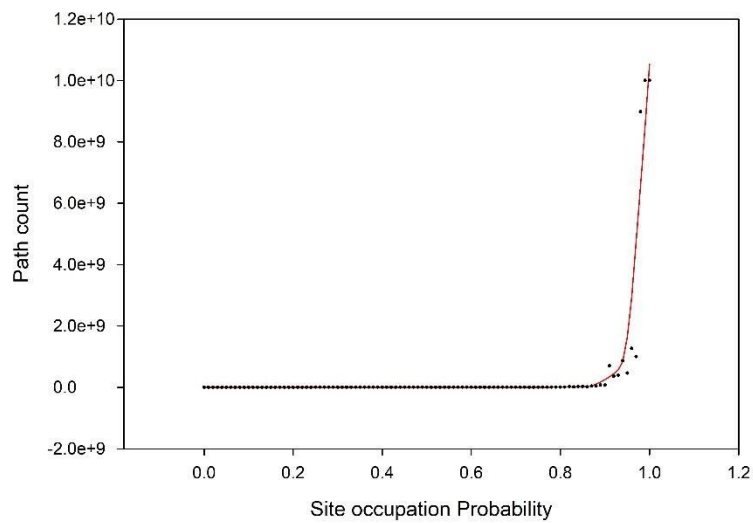
7x7



8x8



9x9



10x10

The sharp increase in the number of paths for probability of site occupation beyond the critical probability in these graphs can be much clearly observed for larger array sizes (see 10x10). The number of paths ideally, should be maximum for values of site occupation probability even just beyond the critical probability, but this is not the case here because we are dealing with small arrays, In general, as array size approaches infinity, the above graph of path count vs site occupation probability should resemble that of a continuous smooth step function, where path count is 0 for values of site occupation probability less than critical probability and (here), approaching n^n for values of site occupation probability greater than the critical probability.

3. Distribution of critical probability

Using the calculated critical probability values for the 100 iterations, the frequency distribution of the values was studied, the results for an 8x8 array with $P_c = 0.465$ are shown below.

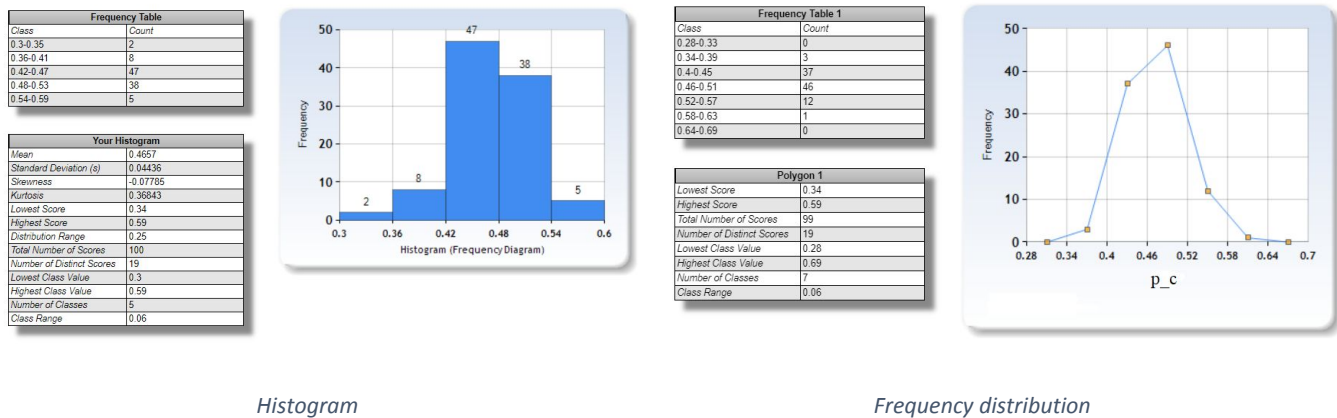


Fig.3.3 shows the frequency distribution for an 8x8 array represented as a simple spline curve, the average value of critical probability corresponds to the peak of the curve, $P_c = 0.465$.

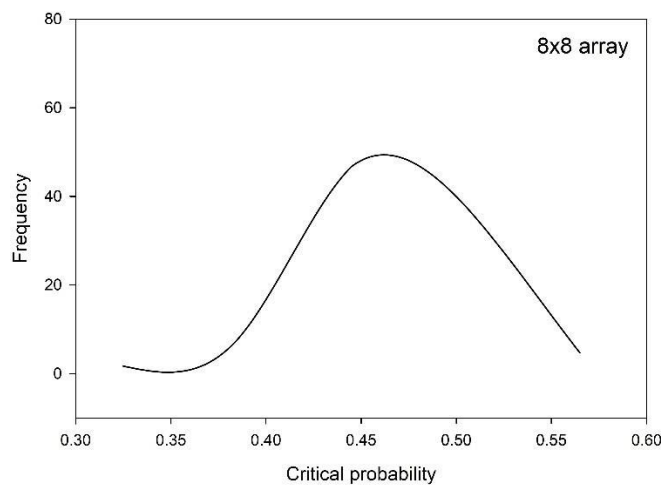


Figure 3.3

References

Bratley, P., Fox, B. L., & Schrage, L. E. ((2011-06-28)). *A Guide to Simulation*. Springer Science & Business Media.

cs.princeton.edu. (2020, 5 10). *cs.princeton.edu*. Retrieved from cs.princeton.edu:

<https://www.cs.princeton.edu/courses/archive/spring15/cos226/lectures/15UnionFind-2x2.pdf>

cs.princeton.edu, B. S. (2008). *cs.princeton.edu*. Retrieved 5 2020, from cs.princeton.edu:

<https://coursera.cs.princeton.edu/algs4/assignments/percolation/specification.php>

cs.usfca.edu. (n.d.). *cs.usfca.edu*. Retrieved from cs.usfca.edu: <https://www.cs.usfca.edu/~galles/visualization/DFS.html>

D Stauffer, A. A. (2018). *Introduction to percolation theory*. Taylor & Francis. Retrieved from

<https://books.google.co.in/books?id=E0ZZDwAAQBAJ&lpg=PP1&ots=Kt6YxMwvAx&dq=forest%20fires%20percolation&lr&pg=PR3#v=onepage&q=forest%20fires%20percolation&f=false>

learnxinyminutes. (2020, 2 12). *fortran95*. Retrieved from learnxinyminutes:

<https://learnxinyminutes.com/docs/fortran95/>

mit.edu. (n.d.). *mit.edu*. Retrieved from mit.edu:

<http://web.mit.edu/8.334/www/grades/projects/projects15/BacanuAlex.pdf>