**Report on Union Find Implementation**

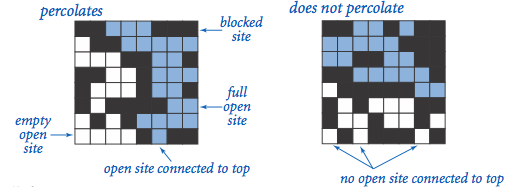
**Abstract.**

This is a report of the analysis of the Union Find Implementation implemented in the project. It also presents asymptotic analysis on the running time of the algorithms used in this program.

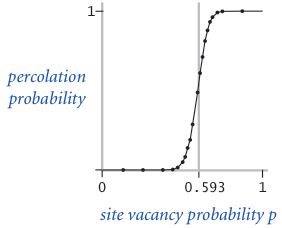
**Introduction.**

**Percolation Threshold:**  
Percolation threshold is a mathematical term related to percolation theory, which is the formation of long-range connectivity in random systems. In [mathematics](http://en.wikipedia.org/wiki/Mathematics), percolation theory describes the behavior of [connected](http://en.wikipedia.org/wiki/Glossary_of_graph_theory) clusters in a [random graph](http://en.wikipedia.org/wiki/Random_graph)[1].

Consider a square lattice with a few open sites; assume that some liquid is poured on top. Percolation threshold describes whether the liquid will make its way from site to site and reach the bottom. A site is "occupied" with probability p or "empty" (in which case its edges are removed) with probability 1-p; the corresponding problem is called **site percolation**.



When S is sufficiently large, there is a threshold value p\* such that when p < p\* a random S-by-S grid almost never percolates, and when p > p\*, a random S-by-S grid almost always percolates. For a given p, the probability that a path exists between top and bottom for a square lattice of size S\*S is 0.592746 [2].

****

**Analysis of Implementation**

**Implementation:**

The implementation consist of two parts:

**1. Percolation rate**

In the first part we input p - percolation probability, n - # of runs and s – board size.  
The program outputs percolation rate. The program creates a board size of s\*s and opens random cells based on percolation probability of each cell. Then it checks whether system percolates for n runs. The program calculates percolation rate as  
 p=(no of times it percolates)/n.

**int**[][] openRandomSites(**float** p):

Initializes a 2D array and generates a random number for each cell; If the cell probability is less than the percolation probability, then it opens the cell by storing the value of two-dimensional array with 1 else closes the cell by storing 0.The time taken to initialize all the elements in the array is O (s2) where s – board size.

A single dimensional array is initialized with values as index and passed to UnionFind () constructor. The **class** UnionFind has two main functions union and find.

**public** **void** union(**int** a, **int** b) {

**int** root\_a = find(a);//finds root of a

**int** root\_b = find(b);//finds root of b

**if** (root\_a != root\_b) //if roots of the two are not equal

array[root\_a] = root\_b;//make one the root of other

}

// find() returns traverses along the path of the tree and returns the root

**int** find(**int** x) {

**while** (x!=array[x])

{ array[x] =array[array[x]];//path compression

x = array[x];

}

**return** x; // At root

}

Path compression used in find reduces the length of the tree and adds the node to the root.

The initialization of single dimension array can be done in time O (s2) where s – board size. The union operator takes O (1) time to change the root, but find operation needs to traverse along the length of the tree to find the root node, so the UnionFind operation takes at most O (n2) time, n-no of elements. The total time complexity for k union find operations is O (k n2).

If path compression is used then after each find the child of a tree is pointed to root node, so now the find operation takes O (n log n) time, n-no of elements. For k union find operations the complexity would be O (k n log n).

**void** converting\_to\_sets(**int**[][] a, **int**[] b, UnionFind uf):

It traverses to the 2D array. For each open cell it checks to see if cell above, below, left and right are open. If they are open the function calls union.

The program then gets the values in the 1D array from unionFind and stores the values in 2D array.

**boolean** ispercolates(**int** d[][]):

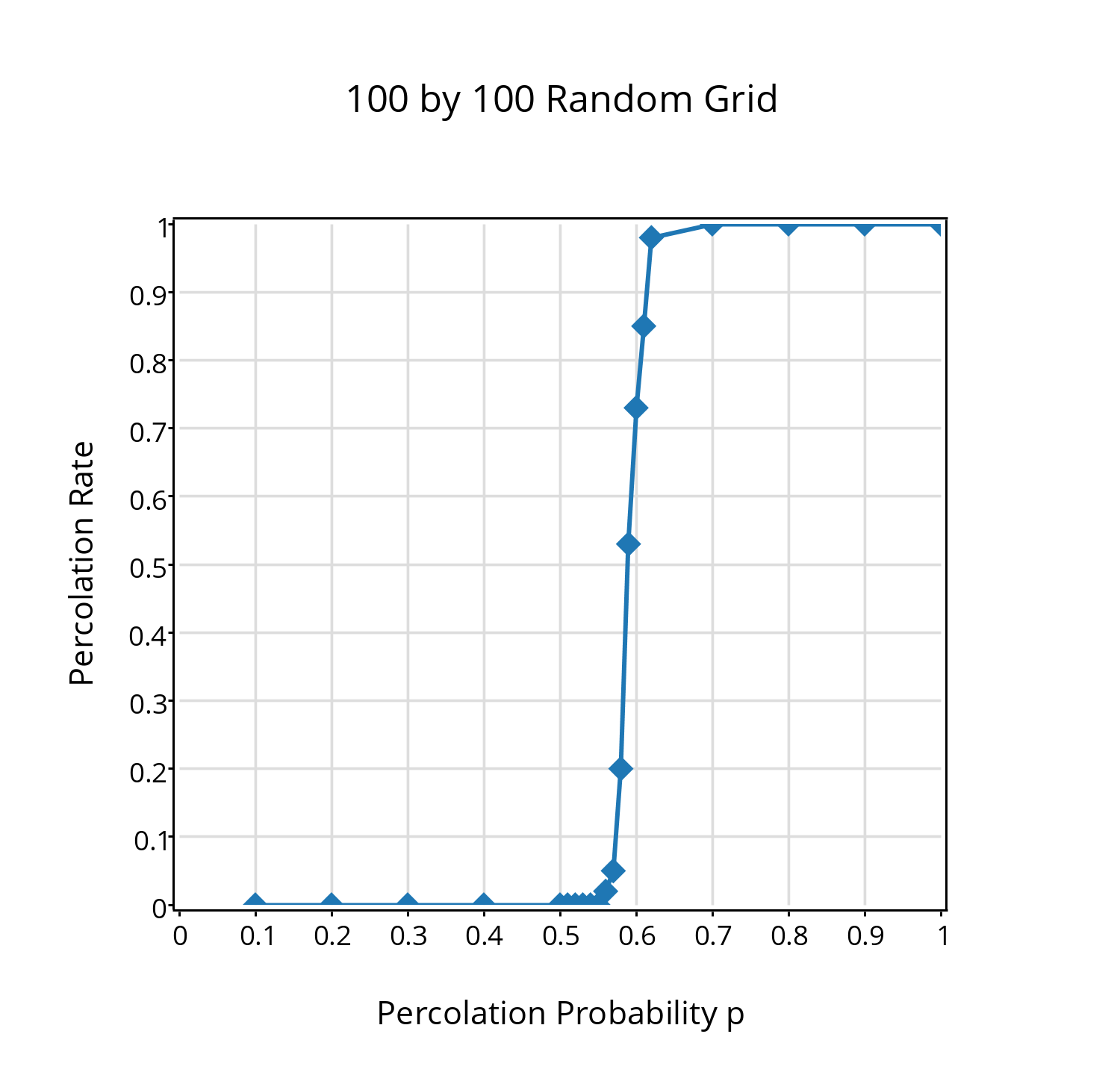
Compares each element in first row and with each element in last row. If the elements are not zero and equal then there is a definite path from first row to last row. So the function returns true that system percolates. This function takes O (S2) time where S is size of first row/last row (size of the board).

For n runs the program, randomly selects the sites to open and then maintains the count if the square lattice percolates. The percolation rate is calculated as:

Percolation rate=(no of times lattice percolates for n runs)/n (no of runs).

Table showing sample execution output:

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| P-percolation probability | 0.0 | 0.5 | 0.55 | 0.56 | 0.57 | 0.58 | 0.59 | 0.6 | 0.61 | 0.62 | 0.7 | 1.0 |
| S-Board size | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 |
| N- no of runs | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 |
| Percolation Rate | 0.0 | 0.0 | 0.02 | 0.02 | 0.05 | 0.20 | 0.53 | 0.73 | 0.85 | 0.98 | 1.0 | 1.0 |



**2.Clusters**

In the second part, the input to the program is a game board setting saved in a text file. We input the file name from command line the program outputs number of clusters in the file. It also outputs two files:  
'Colored\_Image.ppm' - Image file that consists of all the clusters with different colors where white color represents the closed cells.  
 'Simple\_Image.ppm' - Image file that consists of two colors where red represents closed cells and green represents open cells.

The program reads the file using FileReader and BufferedReader. The program initially counts number of lines in the text file by traversing each line until it has no line left. The number of lines can be read as no of rows in the input file. This takes O (S) time s-size of the board.  
After getting the size of the input, a 2D array is initialized .The each line of the input is read until no line is left in the file. For each line, split is performed so that each integer is stored into strings by considering the space between them. Then each integer is stored in a 2D array. The time taken is O (S2), s-size of the board.  
  
The program then uses union find to get the output array. The final array consists of similar values for cells that are connected and zeroes for the closed cells.

For example if the initial input from text file is

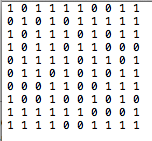
|  |  |  |
| --- | --- | --- |
| 1 | 0 | 1 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

Then out put will be like:

|  |  |  |
| --- | --- | --- |
| 12 | 0 | 7 |
| 12 | 0 | 0 |
| 12 | 12 | 12 |

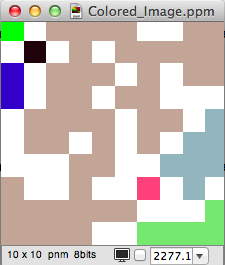
The 1D array is then sorted and removed for duplicate values. The number of unique values is equal to the number of clusters. Using hash map, a unique color (i.e. random color with random RGB value) is assigned for a unique value in the sorted array.  
**void** createColoredPPM(**int**[][] d, **int** s):  
While traversing array the hash map uses the key value and gets a random unique color and writes the content to a file with PPM format ASCII encoding.

For the sample input file:

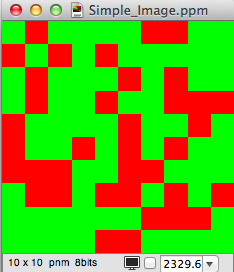


The output is two Image files as shown below:

The output in the file “Colored\_Image.ppm” shows 7 different clusters with 7 different colors. The white colors show blocked cells:



The output in the file “Colored\_Image.ppm” shows 7 different clusters with 7 different colors. The red color shows blocked cells and green shows open cells:



**References**

[1]. <http://en.wikipedia.org/wiki/Percolation_theory>

[2]. <https://www.cs.princeton.edu/courses/archive/fall10/cos226/assignments/percolation.html>

[3]. <http://www.cs.uakron.edu/~zduan/class/435/projects/project2/project2.pdf>