This article is written by students. It may contain oversights or imperfections, as far as possible reported by our reviewers in the edit notes.

Discrete Water Percolation

2021-2022

Name of the students : Luca-Teodor Apahidean, Stefan Boca, Alex-Robert David, Andrei-Rares Doica, Kevin Kepes, Alexandru-Bogdan Miron, Matei-Ioan Munteanu, Simina-Georgiana Negre, Victor Selegean

Schools : Colegiul National "Emil Racovita" Cluj Napoca, Colegiul National "Mihai Eminescu" Satu-Mare

Teachers : Ariana-Stanca Vacaretu, Daly Marciuc

Researcher : George Turcas, "Babes-Bolyai" University of Cluj-Napoca

Contents

1 Problem statement

The soil is modeled by a grid of 100x50 cells with a certain density of occupied cells (represented by the color black). The water (blue) is on the top layer and it will fill every adjacent cell that is empty (coded with the color white). Let t be the number of iterations required by the water to cross the entire ground from top to bottom. Establish the relation between the time t and the function d, based on the simulations.

Either a cell is black with a probability d of instantiating or the soil (grid) has a density d. The two cases are different to program and compute.

The following image represents an example of such grid based on the problem statement.

Figure 1: Visual representation of the problem statement

2 General meaning and different methods of approaching the problem

2.1 Understanding and introduction to different ways of approaching the problem from different perspectives

Even though it may seem difficult at first, after careful thought we can recognise 2 ways to solve the problem that interpose:

- the mathematical approach which will produce a result based on calculating the probabilities of water going to each cell and finding a general relation using statistics and combinatorics,

- the computer-science driven approach which will compute every single way that the water can spread based on modified basic algorithms.

Here we can divide the problem based on the cases enumerated above: the chance of a black cell to appear and the density of all black cells fluctuating (the water requires $t_{density}$ time to pass through).

For the second approach, we can conclude that this problem is a path finding problem in which we are required to compute the distance between every first-row water cell and the last row that denotes the end of our portion of land.

2.2 Introduction into the computer-science manner of solving and computing a correct percolation model, preliminary notions

An algorithm is a finite sequence of logical, well-defined computer implementable instructions, that has the purpose of solving a problem. It can be expressed within an amount of space and time and in a defined formal language.

This problem classifies as a path-finder, being a representative in computer science for its specific problem-set : given a start position (or positions), compute the minimal time t in which you can access a specific element in the matrix that has the coordinates (x_1, y_1) , where x_1 denotes the row, and y_1 denotes the column of the given end-point(s).

This article will present a multitude of specific algorithms and methods to achieve the desired result, along with the data structures that are required for each one.

A data structure is a data organization, management and storage format that enables efficient access and modification (can be viewed as a collection of data values and the relationships among them, and the functions or operations that can be applied to the data). Some of the most representative structures include queues, stacks, maps, sets and many others, including other variations of them.

Note that not all algorithms are the same. Two algorithms can achieve the same result but one can perform in a much better time than the other. Here is introduced the concept of Time complexity, which stands for the computer time it takes to run an algorithm. The amount of time taken and the number of elementary operations performed by the algorithm are taken to be related by a constant factor.

Time complexity is expressed using big O notation, some examples being $O(n), O(n\log_2 n), O(2^n)$. Time complexities are classified according to the type of of function in the notation. Therefore , an algorithm with time complexity $O(n)$ is a linear time algorithm, whereas $O(n^\alpha)$ is a polynomial time algorithm (α being a constant greater than one).

2.3 Path finding principles and multi-agent theory

In computer science , path-finding is the plotting of the shortest route between two points. A great example that has applicability in real life is solving a maze (the maze being represented by a $n \cdot m$ matrix where 0 stands for a free space that can be accessed by an individual agent, whereas -1 stands for a space that cannot be traversed). Note that the grid can be codified according to the needs of the problem.

Adding another layer of complexity, suppose that there are n agents that have different starting points $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ and need to get to a specified cell in the shortest time. Assuming that all of them start at the same time t, computing the shortest path for each one is not enough, because there will be cases of overlapping.

Multi-agent path finding has the purpose of finding paths for multiple agents from their current location to their target locations without colliding with each other, while at the same time computing the shortest way possible. This problem requires a multi-agent approach, each cell that is filled with water being an individual agent that needs to reach the last row of the grid (this method providing a realistic simulation of water percolating through the ground).

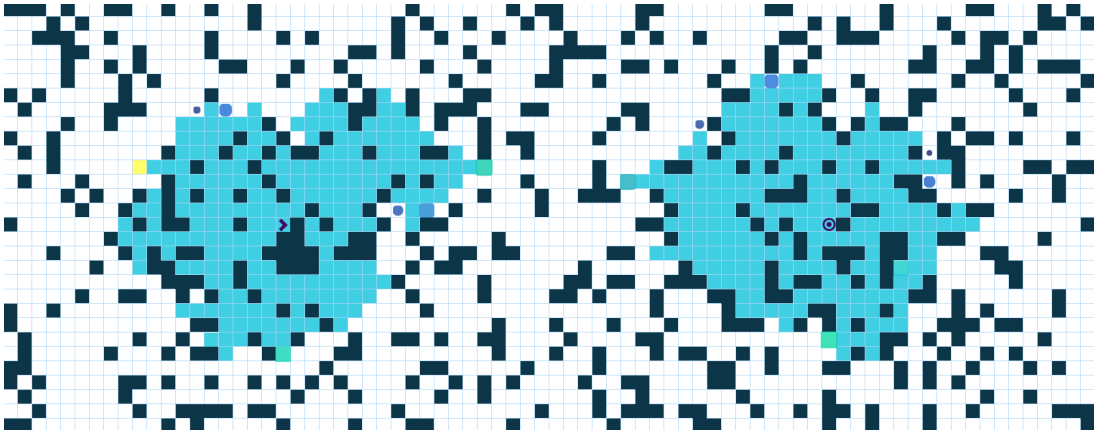

Figure 2: Multi-agent spreading visual representation

3 Describing, implementing and comparing different path-finding models

3.1 Filling the matrix

A naive approach would be filling the the entire matrix to check if there are any positions that will not be filled with water after the last iteration of water dropping through the ground. In order to achieve this result we can consider the following recursive pattern: if the current

element is inside the matrix we mark it with the value v (the value that we want to fill the matrix with, in our case , $v = 1$), we check all the adjacent cells and if one of them is empty the process can be repeated until every traversable cell of the grid has been reached. This algorithm determines if there are spots on the grid that cannot be reached. For solving our problem, this method is enough for simulating the water falling through a surface, but it cannot compute the minimum time it takes to reach the bottom of the ground.

In order to simplify the C++ implementation, direction-arrays can be used. An array is a data structure that allows the storing of multiple variables and accessing them at any time based on the index of the position of an element. This is a technique to encode all directions as arrays, every pair of them representing a distinct direction. A valid recursive C++ implementation of this method is:

```
// direction vectors
const int dx[] = {-1, 0, 1, 0};
const int dy[] = {0, -1, 0, 1}; // N, E, S, V displacement

//function to determine if a given cell is part of the grid,
// i and j being the formal parameters of the function
bool insideGrid(int i, int j){
    return (i >= 0 && i < n && j >=0 && j < m);
}

void Fill(int i, int j){
    grid[i][j] = 1; // filling the current cell
    for(int d = 0; d < 4; d++){ //iterating through the directions
        int iNew = i + dx[d];
        int jNew = j + dy[d];
        if(grid[iNew][jNew] == 0 && insideGrid(iNew, jNew)) //check if the
            adjacent neighbor is empty
            Fill(iNew, jNew);
    }
}
```

Note that this is the general algorithm. For simulating percolation, eliminating the north direction from the directions-array is mandatory because the water cannot fall upwards.

3.2 Breadth-First search applied on a non-directed graph

Breadth-first search (BFS) algorithm is used for computing the path towards an exit point in maze-like problems or similar variations of them. Consider a matrix with n rows and m columns where 0 stands for empty rooms, and -1 stands for occupied cells (positive values will be required for another step therefore we cannot mark such cells with an arbitrary chosen positive value). The task is to compute the minimum time it takes an individual to get to

an end-point knowing that he cannot pass through blocked cells and can only traverse into cells adjacent with the current cell he is placed in either on the same row, or on the same column.

The natural way of describing the algorithm is as follows:

- mark the starting position with the value 1 (suppose that it takes one step to get to the initial position from where a single agent can start percolating)
- from the starting point we look at each adjacent neighbours and determine which ones are not occupied
- we give every empty neighbour the value $k+1$, where k is the number of steps that it would take the agent to reach the previous cell
- we remember all of the cells that we initialised with the value $k+1$
- this process is repeated until the grid is fully occupied
- the shortest path would be the value that is stored in the cell with the coordinates $(x_{endpoint}, y_{endpoint})$

From a computer science perspective , the algorithm works as follows: two matrices need to be initialised of size $n \cdot m$, one for storing the coded maze as values of 0 (empty cell) and 1 (occupied cell) and one matrix that has the purpose of storing the number of steps it would take an agent to reach a certain cell that has the coordinates (x_{cell}, y_{cell}). We are also going to need to instantiate direction-arrays, a technique used and explained at section 3, subsection 1. For remembering the correct adjacent neighbors, a queue of pairs is needed. A queue is a linear structure which follows a particular order in which the operations are performed, that order being First In First Out (or commonly known as FIFO). A good real-life example of a queue if any queue of consumers for a resource where the consumer that came in first is served first. Therefore , we define 3 basic operations that we can apply to the queue:

- empty(), which returns a value of true or false by checking if the queue is empty (it has no elements inside)
- push(), which adds an element at the bottom of the queue
- pop(), which removes the first element in the queue

In our case , we can store the adjacent neighbors in a queue as pairs of coordinates $(x_{neighbor}, y_{neighbor})$. The pair container consists of two data elements or objects. The first element is referenced as 'first' and the second element as 'second' and the order is fixed (first, second). In order to access elements, we use variable name followed by dot operator followed by the keyword first or second, depending on which element we want to access, in our case the index of i or j.

The function takes two start parameters, i_{start} and j_{start}. We give the start position the value 1 (suppose that it takes one step to get to the initial position) and we add the start position to the queue as a pair. While the queue is not empty, we iterate through the direction-vector using a for loop and we declare two new positions i_{new} and j_{new} as the first element of the queue summed up with the corresponding values of the displacement on the x axis, respectively on the y axis. We check if the new coordinates are inside the matrix, if the coordinates of the cell on the first matrix A is equal to 0 (the cell is empty) and we also check if we haven't traversed already on this cell by checking if the same cell in the second matrix is empty. If this conditions are met, we can mark the respective position on the second grid B with the start-value of the queue incremented, then we remove the first element in the queue to repeat this process with the other elements.

The method described works only for one starting point (one drop of water on the first row). If we want to realistically simulate and compute the water percolating , the using of the multi-agent principle described in section 2, subsection 3 is necessary.

To achieve the desired result, we are going to store the initial positions of the drops of water in a stack. Stacks are a type of container adaptors with LIFO(Last In First Out) type of working, where a new element is added at one end (top) and an element is removed from that end only. Once the stack of pairs of indexes of the drops of water is created, the queue is filled with the values stored inside the stack.

Lastly, we iterate through the last row of the second matrix and we calculate the medium time for water to reach the ground as

$$\sum_{i=1}^{m} \frac{B_{ni}}{m}$$

.

A correct C++ implementation (note that for simulating the random cells of the first case the usage of C++ function rand() is mandatory) where the input grid is known is:

```
#include <iostream>
#include <fstream>
#include <queue>
#include <stack>
#define NMAX 1001
using namespace std;

ifstream fin("lee.in");
ofstream fout("lee.out");

// direction vectors for x and y
const int dx[] = {0, 1, 0}; // V S E
const int dy[] = {-1, 0, 1};

int n,m;
```

```
int a[NMAX][NMAX], b[NMAX][NMAX];
stack<pair<int, int>> s;

// function to check if coordinates are inside of the matrix
bool inmat(int i, int j){
    return (i >= 1 && i <= n && j >= 1 && j <= m);
}

void lee(){
    queue<pair<int, int>> q;
    while(!s.empty()){
        q.push(make_pair(s.top().first, s.top().second)); //emptying the queue
        b[s.top().first][s.top().second] = 1;
        s.pop();
    }
    while(!q.empty()){
        for(int d = 0; d < 4; d++){
            int inou = dx[d] + q.front().first;
            int jnou = dy[d] + q.front().second;
            if(inmat(inou, jnou) && a[inou][jnou] == 0 && b[inou][jnou] == 0){
                // checking if the conditions are met
                q.push(make_pair(inou, jnou));
                b[inou][jnou] = b[q.front().first][q.front().second] + 1;
            }
        }
        q.pop(); // removing the first element from the queue
    }
}

int main(){

    fin >> n >> m;
    for(int i = 1; i <= n; i++)
        for(int j = 1; j <= m; j++){
            fin >> a[i][j];
            if(i == 1)
                s.push(make_pair(1, j)); // putting the drops of water in the
                    stack
        }
    lee();
    float ans = 0; // calculating the answer
    for(int j = 1; j <= m; j++)
        ans += b[n][j];
    ans /= m;
```

```
    fout << (float) ans;
    return 0;
}
```

If the positions of the black cells are known from the start and each black cell requires t steps in order to be percolated, the following modifications need to be done to the method of approach described above: a new matrix C is instantiated that has the purpose of storing each unique density for a black cell (x_{cell}, y_{cell}). In that case , in order to achieve the desired result for the second task of the problem, after iterating through the directions off the cells adjacent to a given position and checking if the neighbours respect set conditions, we can conclude that instead of adding the value 1, the value of C at the position iNew and jNew needs to be added (the unique density of each black cell). This slight modification solves for the second set of input that is given and doesn't affect the overall time complexity of the algorithm.

The time complexity of Lee's path finding algorithm is $O(N \cdot M)$, where N and M are the dimensions of the matrix. Using this method , for a grid of 100 by 50 cells , the number of operations that need to be computed is $5 \cdot 10^3$.

4 Interpretation of the generated output, created model of the simulations

Analyzing the output (or the end-result) is required for a better understanding of the path finding mechanism. Suppose that a single agent is placed at the center of an empty grid (no black cells are instantiated). The behavior of the agent (that is placed at the position (3,3)) using a matrix is as follows:

$$\begin{bmatrix} 5 & 4 & 3 & 4 & 5 \\ 4 & 3 & 2 & 3 & 4 \\ 3 & 2 & 1 & 2 & 3 \\ 4 & 3 & 2 & 3 & 4 \\ 5 & 4 & 3 & 4 & 5 \end{bmatrix}$$

Note that the value of a cell in the matrix represent the time t it takes the agent to reach said coordinates from a chosen starting point.

Adding a layer of complexity, the behavior of an agent surrounded by a finite number of walls (represented by the special character '*') that has the possibility of going to every adjacent cell can be described as:

$$\begin{bmatrix} 5 & 4 & 3 & 4 & 5 \\ 6 & * & 2 & * & 4 \\ 7 & * & 1 & 2 & 3 \\ 6 & * & 2 & 3 & 4 \\ 5 & 4 & 3 & 4 & 5 \end{bmatrix}$$

Applying the multi-agent principle for 2 agents that are situated on the first row, the shortest path to a certain point has been decreased due to the number of unique starting positions. It is possible that a position in the grid can't be reached (denoted by its null value).

$$\begin{bmatrix} * & * & 1 & 1 & * & * & * & 1 & 1 \\ 0 & 0 & * & 2 & * & 6 & * & * & 2 \\ * & * & 4 & 3 & 4 & 5 & 5 & 4 & 3 \\ * & 6 & 5 & 4 & * & 6 & 6 & 5 & * \\ * & 7 & * & * & 11 & * & 7 & 6 & 7 \\ * & 8 & 9 & 10 & 10 & 9 & 8 & * & 8 \end{bmatrix}$$

In the case that the water is able to percolate diagonally, the direction vectors can be modified in order to include the offset for the bottom-left and bottom-right diagonals.

4.1 Mapping the path in order to understand the behaviour

In order to rebuild the path for a single agent to an end-point based on the already generated output, the simplest way would be a down-top recursive approach. This time, the starting position matches up with the destination. After all the neighbors of the selected cell are checked, we choose the one with the value $t_{neighbor} = t_{cell} - 1$ and display the necessary direction.

Recursively, this algorithm will be applied until the starting cell is reached. For a correct implementation of the solution, we are going to make use of an array of characters that contains the cardinal points E,N,V, opposite to the already created direction arrays.

```
const int dx[] = {0, 1, 0};
const int dy[] = {-1, 0, 1};
const char dir[] = "ENV";

pair<int, int> startPos;
pair<int, int> finalPos;

void PathRebuild(int i, int j){
    if(i != startPos.first || j != startPos.second){ // stopping condition
        for(int d = 0; d < 3; d++){ // iterating through the neighbors
            int iNew = dx[d] + i;
            int jNew = dy[d] + j;
```

```
            if(insideGrid(iNew,jNew) && b[iNew][jNew] == b[i][j] - 1){ //
                checking the conditions
                PathRebuild(iNew, jNew);
                fout << dir[d] << ' ';
                break; // stopping the current iteration of the neighbors
            }
        }
    }
}
```

Note that for rebuilding the path from the starting position to the final position, the function should be called as PathRebuild(finalPos.first, finalPos.second), where the pair data structure finalPos denotes the x and y positions of the final destination.

This reconstruction and behaviour is important in order to study the path of a certain cell. Furthermore, based on the results generated, the prediction for the path of other cells becomes more clear and can be predicted with better accuracy.

5 General soil density approach

5.1 Premise

The general soil (matrix) density can be interpreted in the following manner: all the cells of the matrix have the same probability $\frac{d}{100}$ of being blocked, where d is a number between [0, 100]. This approach is based on simulations: for a density $\frac{d}{100}$ we generate multiple matrixes and run the BFS algorithm on each one. Then we take the mean of the values returned by the program.

We have performed two different time measurements:

- First we looked at the minimum time necessary for the water to reach the last row
- Secondly, we looked at the necessary time to fill up all accessible cells with water

5.2 Generating the matrices for a density d, implementations of the rand() function

We have defined a random function that will be used when generating the matrix and will decide for each cell if it will be accessible or inaccessible. The function takes as parameter a positive integer d, which is the same for all cells. Based on our assumption that d is a number greater or equal to 0 and less than or equal to 100, we can represent the probability that a cell is inaccessible as the probability of picking a number less than or equal to d from the set $\{1, 2, \ldots 100\}$. From the definition of probability, $\frac{\text{favourable cases}}{\text{total cases}}$, we have d favourable cases,

because there are exactly d numbers less than or equal to d and 100 total cases, meaning that the probability is $\frac{d}{100}$.

Below is an implementation of the function in C++.

```
int random(int d) {
    int a = std::rand() % 100 + 1;
    return (a <= d ? -1 : 0);
}
```

There is however, one technicality that needs to be discussed. While the function above produces a number between 1 and 100 (both inclusive), its flaw is in the use of the modulus operator. The *std::rand()* function provided in the C++ library returns a pseudo-randomly chosen integer from the set $\{0,\ 1,\ 2, \ldots RAND\_MAX\}$, where *RAND_MAX* is a compiler defined variable, usually $2^{15}-1$. Since *RAND_MAX* is not a multiple of 100, applying Dirichlet's principle, the remainders of the modulus operator will not be uniformly distributed, skewing our results.

The solution was to move to a language that provides better built-in functions. Since data had to be gathered and plotted, the best option was the Python programming language, that offers many libraries with ease of use. Below is the implementation in python. The approach remained the same.

```
def get_probability(d):
    a = random.randint(1, 101)
    if a <= d:
        return -1
    else:
        return 0
```

5.3 Two different approaches on percolation time

- Firstly we considered the time necessary for the water to reach the last row of the matrix (to percolate to the bottom). This answer is represented by the mean value of the cells in the last row in the B matrix.
- Secondly we considered the minimum time necessary for the water to reach all available cells. This answer is represented by the maximum value in the B matrix.

5.4 Mathematical approach to percolating probability using basic combinatorics patterns

In order to obtain a formula which gives you the probability of the water percolating with respect to the density of the matrix, first we need to observe in which cases the water cannot percolate to the bottom. If we can find a function to express the probability of the water not percolating to the bottom (let's denote it p'), than the probability of the water percolating (let's denote it p) is $p = 1 - p'$ (assuming that $0 \leq p \leq 1$ and $0 \leq p' \leq 1$).

Let a wall be a continuous group of adjacent black cells, and d the density of the matrix (the probability that every cell will be instantiated black). If the water cannot percolate through the bottom, it means that in the matrix exists a continuous wall starting from the first column of the matrix and ending in the last column, that "traps" the water above it. Thus the probability of the water not percolating equals to the number of walls that "trap" the water multiplied by the probability of every wall being instantiated. Furthermore, if a wall has a length l, than the probability of that wall being instantiated is d^l.

We managed to find a formula for a specific type of walls. We list now the properties of the walls that we are able to count. Let W be the set of all possible walls having the following properties:

1. They start from a cell in the first column of the matrix and end in a cell in the last column of the matrix;

2. Every cell of the wall either has the left side or the upper side adjacent to the previous cell (in other words, the wall can only "go" right, or down);

3. Every cell of the wall must be inside the matrix

Suppose we take two cells in the matrix: the first one will be on the first column (with coordinates (y_1, 0)) which will be the starting cell of the wall, and the second one will be on the last column (with coordinates (y_2, 0)) which will be the ending cell of the wall (assume $y_2 \leq y_1$). Than we know that the walls that start and end in those coordinates have exactly (y_2 - y_1) "down moves" (number of cells added under the previous cell) and exactly (width - 1 = 99) "right moves" (number of cells added to the right of the previous cell) thus all these walls having the same length. If we interpret a "down move" as a "1" digit and a "right move" as a "0" digit, than our wall can be represented by a number with (99 + y_2 - y_1) digits, from which (y_2 - y_1) digits are "1". Thus, the number of our walls will be equal to the number of ways to form a (99 + y_2 - y_1) digit number with (y_2 - y_1) digits of "1", which is equal to:

$$\binom{99 + y_2 - y_1}{y_2 - y_1}$$

In the case that $y_2 \leq y_1$, than the "down moves" will be switched with "up moves" and y_2 will switch places with y_1 in the above formula. A formula that includes both cases would look

something like this:

$$\binom{99+|y_2-y_1|}{|y_2-y_1|}$$

The probability that one off these walls instantiate is equal to (the number of walls) times (the probability of one wall instantiating) which is

$$\binom{99+|y_2-y_1|}{|y_2-y_1|} \cdot d^{100+|y_2-y_1|}$$

Now to obtain the general formula that gives us the probability of water percolating we have to look at every possible (y_1, y_2) pairs (with $1 \leq y_1 \leq 50$, and $1 \leq y_1 \leq 50$) and sum up all these probabilities.

$$1 - \sum_{y_1=1}^{50} \sum_{y_2=1}^{50} \binom{99+|y_2-y_1|}{|y_2-y_1|} \cdot d^{100+|y_2-y_1|}$$

The formula above is an upper bound for the probability of percolation. In order to get an exact formula for the latter probability, one could try to count all the possible walls arising. This could be an interesting future research direction.

6 Specific cell densities approach

6.1 Premise

A different approach is viewing the matrix as a collection of cells, each with its own individual probability of being filled. When doing so, we define d_{ij} as the probability that the cell m_{ij} is blocked. Each d_{ij} is a number in the interval $[0,1]$ where 0 denotes no possible obstacle and 1 is a decisively blocked cell.

Let P be the probability function that distributes the variable:

P: F $\rightarrow [0,1]$, where F is the set of possible outcomes

$x_1, x_2, ..., x_n \in$ F

$p_1, p_2, ..., p_n \in [0,1]$

The sum of the probabilities of all the distinct events will naturally be equal to 1

$$\sum_{i=1}^{n} p_i = 1$$

6.2 Implications and a first implementation

In order to implement such an approach, we would have to make some changes to the previous code in order to allow cells to have rational, as opposed to integer, values inside them.

As a convention, we considered a value of 0 to mean a definitive clear passage and a value of 1 a definitive block. All values in between 0 and 1 of the form 0.n denote an n%probability of a block being there.

To reach an answer, the program would have to render all possible matrices, find out the shortest path to the end in each of them and associate a probability to each before making a weighted arithmetic mean of all the results for the given input.

This approach fails, however, when taking into consideration the fact that the program would have to generate 2^n matrices, where n is the number of all cells that have $d_{ij} \in (0,1)$, so a probability that is not integer. In order to better understand why that is, we can imagine every rational density as a path branching in two, one with the weight of d_{ij} and one with the weight $1 - d_{ij}$. Naturally, when d_{ij} is either 0 or 1, one of the paths will have a weight equal to 0. For our intents and purposes, almost all cells will have rational densities, so the number of matrices the program would have to generate would be somewhere around to 2^{5000}.

Obviously this method is very inefficient and requires an absurd amount of time to study only a very specific input matrix. As such a new method had to be developed.

6.3 The method of simulated chance

To test this assumption, we created a smaller matrix which we would be able to compute with both of the programs. As the number of tests we required the second program to run got bigger, from 100 to 1 000 and even 100 000, so too the results got more accurate while the time of execution remained no bigger than a few minutes at most.

In order to accurately simulate random chance, when encountering a new cell of rational probability, the program chooses a random rational value in the [0,1] interval which we will call n. If $n \geq d_{ij}$, the cell will be considered from thereon out as a definitive clear passage. If the contrary is true and $n < d_{ij}$ then the cell will be considered a definitive block for the rest of the simulation.

6.4 Interpretation of results

Having solved the efficiency issue we must now consider how we interpret the results. A simple average of all the values output when the percolation was successful would give a general idea, but we would have no certainty that our answer will actually fall anywhere near that average.

In order to create a more general result, a commonly used method in statistics is the use of samples. By calculating the average of multiple tests and plotting the result over and over again we would come up with a graph resembling a Normal (also referred to as Gaussian) Distribution.

The Central Limit Theorem tells us as the amount of tests required for an average approaches infinity, the value of the average calculated approaches the real average of the population we study.

At this point, we have to define a number of concepts. These are either relating to the overall population or to the finite number of tests we compile to study a sample.

6.4.1 Absolute Values

μ (mu) denotes the population mean. This is the value that we look for but are unable to calculate due to the limitations of the program previously described in section 5.2.

$$\mu = \sum_{i=1}^{n} (x_i * p_i),$$

where n is the amount of tests in the sample, x_i is the time required for the water to percolate and p_i is the associated probability of that output time.

σ (sigma) denotes the standard deviation. The value of σ tells us how far away from the population mean our results are likely to fall. The smaller σ is, the more close our output will be to μ. However, just as μ is incalculable in our context, so too is σ.

$$\sigma = \sqrt{\frac{\sum_{i=1}^{n} (x_i - \mu)}{n}}$$

According to the Empirical Rule (also called the Three Sigma Rule) almost all values observed for different samples fall in the interval $[\mu-3\sigma;\mu+3\sigma]$. Precisely, 68% of data observed will fall in the range $[\mu-\sigma;\mu+\sigma]$, 95% in the range $[\mu-2\sigma;\mu+2\sigma]$ and 99.7% in $[\mu-3\sigma;\mu+3\sigma]$. As such, by determining these two values, we will have a very precise, although not exact, answer to any given input matrix.

6.4.2 Relative Values

These values refer to whatever we can observe during our finite number of tries. $\overline{x}$ refers to the sample mean. Just as μ denotes the apex point of the "absolute" curve, $\overline{x}$ is the apex of our observable curve . We can find it using the formula:

$$\overline{x} = \frac{\sum_{i=1}^{n} x_i}{n}$$

s is the standard sample deviation. The formula for finding s out is the following:

$$s = \frac{\sum_{i=1}^{n} (x_i - \overline{x})^2}{n}$$

6.5 Practical implementation of statistics

Four functions are defined in order to achieve the desired result:

```
void gen(int n, int m){ //generating binary matrices according to the
    probabilities
  int x, y;
  cout << "Probability interval (x, y): ";
  cin >> x >> y;
  fo << fixed << setprecision(2);
  srand(time(NULL));
  for (int i = 1; i <= n; i++, fo << '\n')
    for (int j = 1; j <= m; j++)
      fo << ((rand() % (y + 1 - x) * 1.0) + x) / 100 << ' ';
  fo.close();
}
void input(int n, int m, float d[][101]){ //inserting data (input)
  for (int i = 1; i <= n; i++)
    for (int j = 1; j <= m; j++)
      fi >> d[i][j];
  fi.close();
}

void clean(int n, int m, int A[][101]){ //clearing the matrix to original state
  for (int i = 1; i <= n; i++)
    for (int j = 1; j <= m; j++)
      A[i][j] = 0;
}

int PBFS(int n, int m, float d[][101]){ //An adaptation of the Breadth-First
    Search covered in Section 3.2
  clean(n, m, A);
  clean(n, m, V);
  queue < pair < int, int >> Q;
  for (int i = 1; i <= m; i++)
    Q.push(make_pair(0, i));
  while (!Q.empty()) {
    int i = Q.front().first, j = Q.front().second;
    Q.pop();
    for (int k = 0; k < 4; k++) {
      int in = i + di[k], jn = j + dj[k];
      float random = rand() % 100;
      random /= 100;
      if ( in >= 1 && in <= n && jn >= 1 && jn <= m && V[ in ][jn] == 0 &&
```

```
        random < (1 - d[ in ][jn])) //1-d[in][jn] is the probability that the
        water can reach cell [in][jn] from [i][j].
      {
        Q.push(make_pair( in , jn));
        V[ in ][jn] = 1;
        A[ in ][jn] = A[i][j] + 1;
        if ( in == n)
          return A[i][j] + 2;
      }
    }
  }
  return -1;
}
```

Lastly, we integrated the functions described above and implemented the formulas defined at the subsections 6.4.1 and 6.4.2:

```
#include <iomanip>
#include <vector>
#include <cmath>
#include "functions.h"
using namespace std;

int n,m,tests,res;
float t_value;//the t_value corresponding to the desired confidence.
float NULL_p;
int NULL_cases,S_cases;//the number of cases in which the water did not
    successfully cross the matrix; the number of cases in which it crossed the
    matrix.
float d[51][101];
double sd,sm;//sample standard deviation and sample mean respectively.
vector<unsigned short> X;

int main()
{
    n=50;
    m=100;
    bool ok;
    cout<<"Do you want to generate the matrix? :D\n";
    cin>>ok;
    if(ok)
        gen(n,m);
    cout<<"Number of tests: ";
```

```
    cin>>tests;
    cout<<"t_value = ";//(percentage)
    cin>>t_value;
    input(n,m,d);
    cout<<fixed<<setprecision(5);
    srand(time(NULL));//Initializing the random number generator.
    for(int i=1;i<=tests;i++)
    {
        res=PBFS(n,m,d);
        if(res==-1)
            NULL_cases++;
        else
        {
            sm+=res;
            X.push_back(res);
            cout<<res<<'\n';
        }
    }
    NULL_p=100-NULL_cases*1.0/tests*100;
    S_cases=tests-NULL_cases;
    sm/=S_cases;
    for(auto it:X)
        sd+=(it-sm)*(it-sm);
    sd/=(S_cases-1)*S_cases*S_cases;
    sd=sqrt(sd);
    cout<<"For the corresponding t_value, the expected value belongs to the
        interval: ["<<sm-t_value*sd<<", "<<sm+t_value*sd<<"]\n";
    return 0;
}
```