Water Percolation

Doica Rares-Andrei Miron Alexandru-Bogdan Munteanu Matei-Ioan

Colegiul National "Emil Racovita" Cluj-Napoca

Problem Statement

Figure 1. Visual representation of the 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.

Methods of Approaching the Problem

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

- **Mathematical Approach** computing the number of matrices that don't let the water percolate (based on combinatorics formulas), thus obtaining the percolation probability.
- **Computer-Science Approach** will compute every single way that the water can spread based on modified basic algorithms.

Defining the Methods

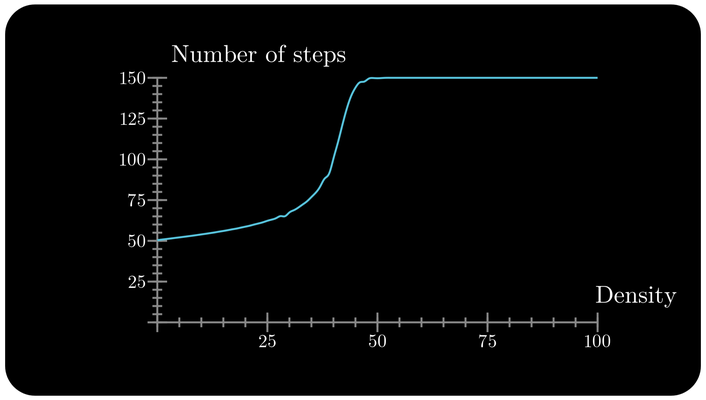
Here we can divide the problem based on the cases enumerated above: the chance of a black cell to appear and the density of all black cells fluctuating (the water requires $t_{density}$ time to pass through).

From a mathematical standpoint, 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 $0 <= p <= 1$ and $0 <= p' <= 1$). Furthermore, if a wall has a length l, than the probability of that wall being instantiated is d^l.

From a computer-science point of view, we can conclude based on these facts 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. In order to achieve such simulation, each individual path for a singular agent(water cell) should be computed at the exact same time as the other agents in order to get a correct result.

Mathematics Curve Visualisation

Using the open-source library Manim, we can generate a mathematical function using the values generated from the 1000 tests we ran for a wide range of densities.



For each density, we generated 100 random matrices of the respective soil density, and for each one we calculated the number of time units necessary for the water to percolate. In case the water cannot percolate, we assign a big value to the number of steps in order to represent the function.

Path-finding Principles

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.

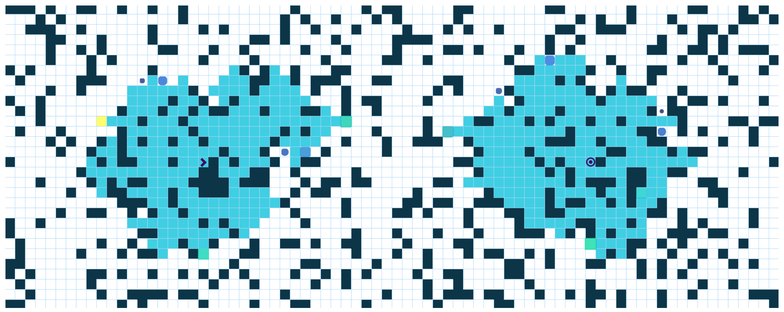


Figure 2. Multi-agent spreading visual representation

Breadth First Search, Lee's Path-finding

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}).

```
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
    }
}
```

Figure 3. A correct implementation of BFS search using multi-agent theory

Analyzing the Generated Output and iOS Implementation

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:

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

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).

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