Simulating Lichtenberg Figures

Simon Chen[1,2], Shan Gao[1,3]

[1] Department of Mathematics and Statistics, McGill University, Montreal, Canada

[2] Department of Physics, McGill University, Montreal, Canada

[3] School of Computer Science, McGill University, Montreal, Canada

June 1, 2022

Abstract

There currently exist many popular search algorithms in the field of graph theory, such as the A* search algorithm, breadth-first search (BFS), depth-first search (DFS), and Dijkstra's algorithm. We present an algorithm analogous to BFS which, although pragmatically inefficient, aims to replicate the path of an electric current travelling through an insulating material.

Contents

0 Acknowledgements

I would like to thank my friend Oscar Chen for getting me started on this project and for providing me with valuable resources along the way. I would also like to thank my friends Andrew Zeng and Sebastian Filner for our conversations over the course of the academic year as well my brother Fernando Chen for his suggested modifications. Finally, I would like to thank my friend Shan Gao for his major contributions to the project and indispensable help in programming, without which I would have been unable to achieve my desired multi-branch algorithm.

Simon

1 Introduction

Dielectric breakdown is a process that occurs when an insulating material suddenly becomes conductive when subjected to a sufficiently strong electric field. The maximum electric field that can be withstood by an insulating material without undergoing electrical breakdown is known as its dielectric strength, and the voltage at which an insulating material becomes conductive is known as its breakdown voltage. During breakdown, an electric current flows through the insulating material, producing a branching, tree-like pattern. These patterns are known as Lichtenberg figures and can appear on the surface or in the interior of insulating materials.

In particular, we were interested in the creation of Lichtenberg figures on wood due to the endless variety of fractal patterns that can be produced. To create real-world Lichtenberg figures, a coat of electrolytic solution is first applied to a piece of wood in order to reduce the resistance on its surface. An electrode is then placed on either end of the wood and a high voltage is passed across them. The current generated from the electrodes will cause the surface of the wood to heat up and burn. Because carbonized wood is mildly conductive, the surface will burn in a pattern travelling outwards from the electrodes. Our algorithm was developed in an attempt to best simulate this behaviour.

2 Single-Branch Algorithm

2.1 Methodology

The single-branch algorithm was first created as a proof of concept for this project. The purpose of this algorithm was to determine the project's feasibility and to examine the algorithm's ability to faithfully replicate the path of a real electric current.

2.1.1 Version 1.0.0

The first step was to ensure that the algorithm could find the entry with the smallest value in its immediate surroundings, in all 8 directions, given an arbitrarily generated matrix:

$$\begin{bmatrix} \ddots & \ddots & \ddots & \ddots & \ddots \\ \ddots & 9 & 12 & 23 & \ddots \\ \ddots & 11 & 7 & 14 & \ddots \\ \ddots & 8 & 25 & 3 & \ddots \\ \ddots & \ddots & \ddots & \ddots & \ddots \end{bmatrix} \tag{2.1}$$

Once the minimum entry is found, the direction of travel and the entry's value are recorded in a dictionary. The algorithm then searches for a new minimum entry in the neighbourhood of the previously determined minimum:

$$\begin{bmatrix} \ddots & \ddots & \ddots & \ddots & \ddots \\ \ddots & 7 & 14 & 12 & \ddots \\ \ddots & 25 & 3 & 8 & \ddots \\ \ddots & 5 & 19 & 10 & \ddots \\ \ddots & \ddots & \ddots & \ddots & \ddots \end{bmatrix} \tag{2.2}$$

In order to prevent infinite loops, the algorithm is prohibited from returning to an entry it had previously been to. The function would call itself recursively until it terminates either by reaching an entry with a value of 0 or by running out of entries to move to. This first version of the algorithm was tested on the following, manually configured, 2-dimensional array:

$$\begin{bmatrix} 15 & 14 & 13 & 12 & 11 & 99 \\ 99 & 99 & 19 & 8 & 10 & 99 \\ 99 & 99 & 99 & 15 & 99 & 99 \\ 99 & 0 & 7 & 99 & 11 & 99 \\ 99 & 99 & 99 & 17 & 12 & 99 \\ 99 & 99 & 99 & 99 & 99 & 99 \end{bmatrix} \tag{2.3}$$

The function terminated once it reached an entry of value 0.

2.1.2 Version 1.1.0

The next version of this algorithm, when faced with two or more neighbouring entries of the same minimum value, would make a choice based on the direction of travel. Its directions in order of preference are: down-right $(1, 1)$, down $(1, 0)$, right $(0, 1)$, down-left $(1, -1)$, up-right $(-1, 1)$, left $(0, -1)$, up $(-1, 0)$, up-left $(-1, -1)$. This version of the algorithm was likewise tested on a similar, manually configured, 2-dimensional array:

$$\begin{bmatrix} 26 \rightarrow & 24 \rightarrow & 22 \rightarrow & 20 \searrow & 99 & 99 \\ 99 & 99 & 99 & 99 & 18 \downarrow & 99 \\ 99 & 0 & 15 & 99 & 16 \downarrow & 99 \\ 99 & 99 & 99 & 13 & 11 \downarrow & 99 \\ 99 & 0 \nwarrow & 99 & 99 & 13 \swarrow & 99 \\ 99 & 99 & 17 & \leftarrow 15 & 99 & 99 \end{bmatrix} \tag{2.4}$$

The algorithm chose one of the two entries of value 13 based on preferred direction of travel. The function terminated once it reached an entry of value 0.

2.1.3 Version 1.1.1

The algorithm was then modified to accept any 2-dimensional array with an arbitrary starting position, and the up-left $(-1, -1)$ option was removed from the list of possible directions the algorithm could travel in.

2.1.4 Version 1.2.0?

Future modifications could involve the algorithm choosing a path based on the two subsequent minimum entries rather than on the direction of travel. Testing this algorithm on the same, previous, 2-dimensional array:

$$\begin{bmatrix} 26 \rightarrow & 24 \rightarrow & 22 \rightarrow & 20 \searrow & 99 & 99 \\ 99 & 99 & 99 & 99 & 18 \downarrow & 99 \\ 99 & 0 \leftarrow & 12 & 99 & 16 \downarrow & 99 \\ 99 & 99 & 99 & 13 \leftarrow & 11 & 99 \\ 99 & 0 & 99 & 99 & 13 & 99 \\ 99 & 99 & 17 & 15 & 99 & 99 \end{bmatrix} \tag{2.5}$$

Contrary to the path through matrix 2.4, this version of the algorithm would choose the other entry of value 13 because, even though the first minima are equal $(13 = 13)$, the second minima are not $(12 < 15)$. Again, the function would terminate once it reaches an entry of value 0.

2.2 Results

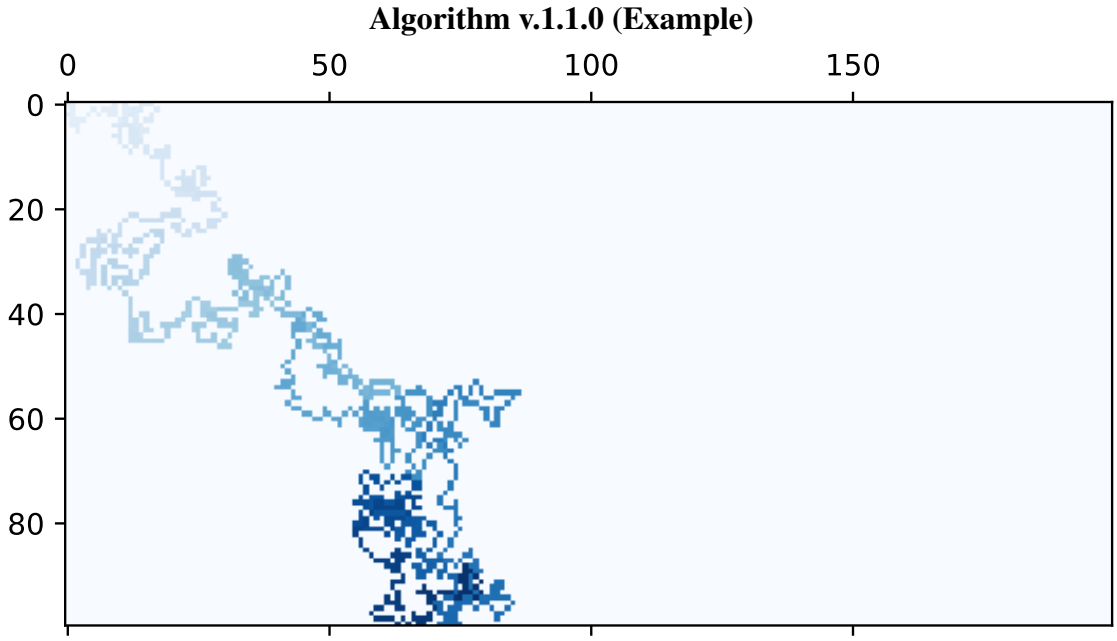

Figure 1: Figure of the single-branch algorithm traversing a randomly generated 100×200 matrix with a starting position of $(0, 0)$.

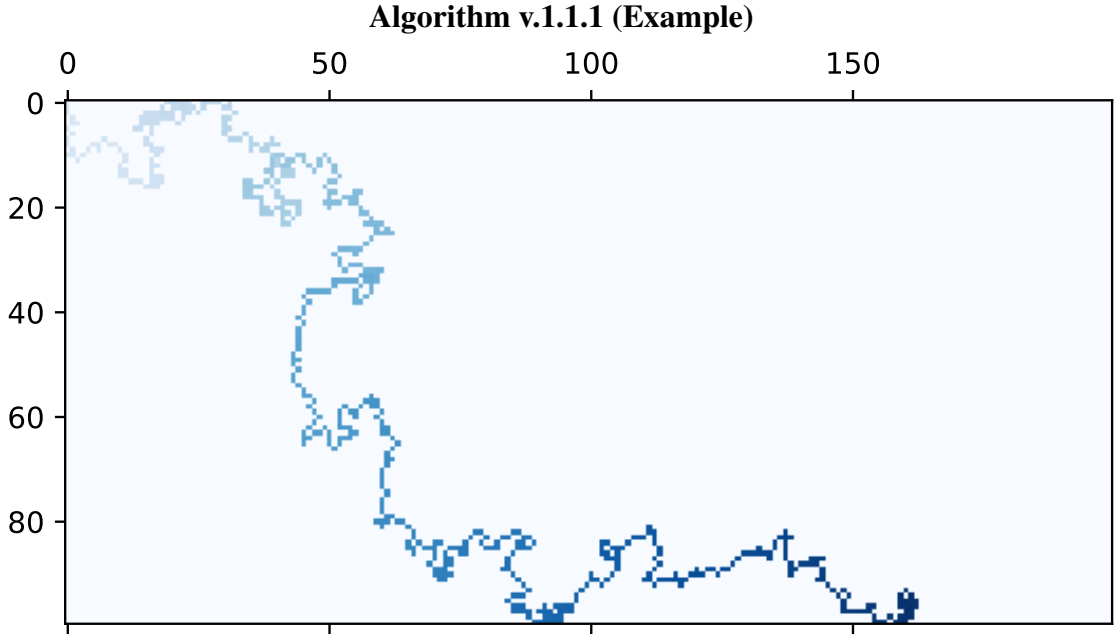

Figure 2: Figure of the single-branch algorithm traversing a randomly generated 100×200 matrix with a starting position of $(0, 0)$.

The path taken by each algorithm is recorded in blue. Given that these are static images, a colour gradient was chosen in order to provide a sense of direction: the earliest traversed entries are lighter while the most recent traversed entries are darker. See GitHub for animations of these runs.

2.3 Discussion

The single-branch algorithm successfully accomplished its purpose but was unsuccessful in faithfully replicating the path of an electric current travelling through an insulator because, unlike a real electric current, the algorithm does not split when trying to find the path of least resistance. Rather, the algorithm travels in a single direction based on a predefined order of preference. This issue was partially resolved in version 2.0.0 and then fully resolved in version 2.1.0.

2.3.1 Version 1.1.1

The algorithm was prohibited from travelling in the up-left $(-1, -1)$ direction in order to reduce the number of times it would loop around itself (note the difference between the path taken in Figure 1 versus the path taken in Figure 2). This signifies that the algorithm completely disregards the up-left direction regardless of whether there exists a minimum in that direction. Although there is nothing physically wrong with the algorithm looping around itself multiple times, it is not a phenomenon often observed in nature.

2.3.2 Version 1.2.0

A future modification to the single-branch algorithm could involve taking into account the subsequent minimum entries rather than having the algorithm choose a direction based on an arbitrary order of preference.

The algorithm would first find the minimum entry in its immediate surroundings. If there exist two minimum entries of equal value, it would analyze the immediate surroundings of both minima to find the next minimum entry. The search process would repeat until a single path with the least amount of resistance is definitively established (see matrix 2.5 for an example).

Although this would imply that an electric current has a priori knowledge of the resistance values at every point in the insulating material it is travelling through, the algorithm should, in theory, be more physically accurate.

3 Multi-Branch Algorithm

3.1 Methodology

The creation of a functioning multi-branch algorithm was the following step in attempting to achieve this project's objective. The single-branch algorithm did not faithfully replicate the patterns seen in Lichtenberg figures and hence had to be modified. The multi-branch algorithm was first inadvertently developed as a DFS algorithm and then later modified into a BFS algorithm.

3.1.1 Version 2.0.0

The first version of the multi-branch algorithm was recursive and analogous to a DFS algorithm; if two or more neighbouring entries have the same minimum value, the algorithm chooses to travel in the direction of the first recorded minimum, prioritizing by preferred direction of travel. This version of the algorithm was tested on the following, manually configured, 2-dimensional array:

$$\begin{bmatrix} 15 \rightarrow & 14 & 13 & 99 & 99 & 99 \\ 19 & 13 & 99 & 12 & 99 & 99 \\ 12 & 10 & 99 & 99 & 11 & 99 \\ 6 & 6 & 99 & 99 & 10 & 99 \\ 99 & 0 & 99 & 99 & 5 & 5 \\ 99 & 99 & 99 & 0 & 99 & 99 \end{bmatrix} \tag{3.1}$$

The algorithm chose one of the two entries of value 6 based on preferred direction of travel. The function terminated once it reached an entry of value 0.

Once the first branch terminates, the algorithm backtracks to the most recent entry where more than one neighbouring minimum was found, and travels down the branch of the second recorded minimum:

$$\begin{bmatrix} 15 \rightarrow & 14 & 13 & 99 & 99 & 99 \\ 19 & 13 & 99 & 12 & 99 & 99 \\ 12 & 10 & 99 & 99 & 11 & 99 \\ 6 & 6 & 99 & 99 & 10 & 99 \\ 99 & 0 & 99 & 99 & 5 & 5 \\ 99 & 99 & 99 & 0 & 99 & 99 \end{bmatrix} \tag{3.2}$$

The algorithm then followed the branch created by the other entry of value 6 since it was the first most recently recorded splitting point. This time, the function terminated because it ran out of entries to move to.

The algorithm then continues to work its way backwards, branching off in order of the most recently recorded minima:

$$\begin{bmatrix} 15 & 14 & 13 & 99 & 99 & 99 \\ 19 & 13 & 99 & 12 & 99 & 99 \\ 12 & 10 & 99 & 99 & 11 & 99 \\ 6 & 6 & 99 & 99 & 10 & 99 \\ 99 & 0 & 99 & 99 & 5 & 5 \\ 99 & 99 & 99 & 0 & 99 & 99 \end{bmatrix} \tag{3.3}$$

The algorithm then travelled in the direction of the other entry of value 13 since it was the second most recently recorded splitting point. Once again, the function terminated once it reached an entry of value 0.

3.1.2 Version 2.0.1

The algorithm was then modified to accept any 2-dimensional array with an arbitrary starting position, and the up-left (-1, -1) option was removed from the list of possible directions the algorithm could travel in.

3.1.3 Version 2.1.0

For the following version, the multi-branch algorithm was modified into a BFS algorithm. This version of the algorithm was tested on a similar, manually configured, 2-dimensional array:

$$\begin{bmatrix} 15 & 14 & 13 & 99 & 99 & 99 \\ 19 & 13 & 99 & 12 & 99 & 99 \\ 12 & 10 & 99 & 99 & 11 & 99 \\ 6 & 6 & 99 & 99 & 10 & 99 \\ 99 & 0 & 99 & 99 & 5 & 5 \\ 99 & 99 & 99 & 0 & 99 & 0 \end{bmatrix} \tag{3.4}$$

Rather than travelling down a single branch completely before backtracking and travelling down another one, the algorithm travels down both branches in an alternating manner.

The algorithm's path through a matrix becomes unclear with a BFS algorithm, so its traversal order through matrix 3.4 will be represented using a tree:

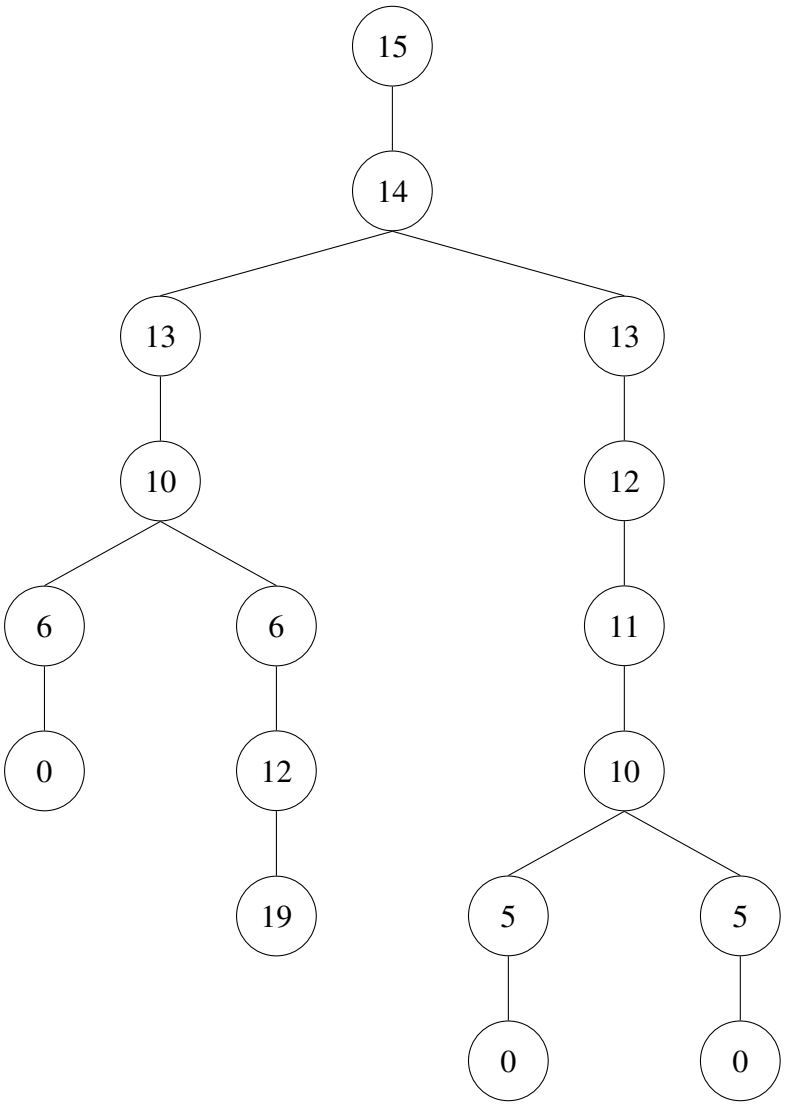

Figure 3: Tree representation of the algorithm's traversal order through matrix 3.4

3.1.4 Version 2.1.1

The algorithm was then modified to accept any 2-dimensional array with an arbitrary starting position, and the left (0, -1) and up-left (-1, -1) options were removed from the list of possible directions the algorithm could travel in.

3.1.5 Version 2.2.0?

Future modifications could involve creating an algorithm with two starting positions. Each starting position would be the origin of an independent tree, with both taking the path of least resistance towards each other.

3.2 Breadth-First Search Implementation

The BFS method of traversing through a 2-dimensional array was implemented with the use of queues.

If two or more neighbouring entries have the same minimum value, the function records those minima into a queue in order of preferred direction of travel. The algorithm then pops the first element from the queue and searches for all the minimum entries in the neighbourhood of that first element. If two or more minima are found, they are appended to the queue, again, in order of preferred direction of travel. The order in which each minimum is popped becomes the traversal order of the algorithm through the 2-dimensional array. The algorithm terminates once there are no more minima in the queue.

The particular portion of the algorithm that generates the aforementioned queue can be described using the following pseudocode:

Algorithm 1: Pseudocode of a portion of algorithm v2.1.0

```
Data: a 2-dimensional array
Result: the BFS traversal sequence through the matrix
initialize empty queue q;
initialize empty list of visited nodes ℓ;
q.enqueue(root) /* the root node is at (0, 0) */;
while q is not empty do
    cur ← q.dequeue;
    append cur to list ℓ;
    for each minimum of cur do
        q.enqueue(minimum);
    end
end
```

The algorithm's traversal through a 2-dimensional array can also be viewed as a sequence of queues q after having visited each minimum entry. The front element of the queue is popped in the upcoming queue and any new enqueued minima are appended to the rear.

The sequence of queues for the algorithm's traversal through matrix 3.4 is given by:

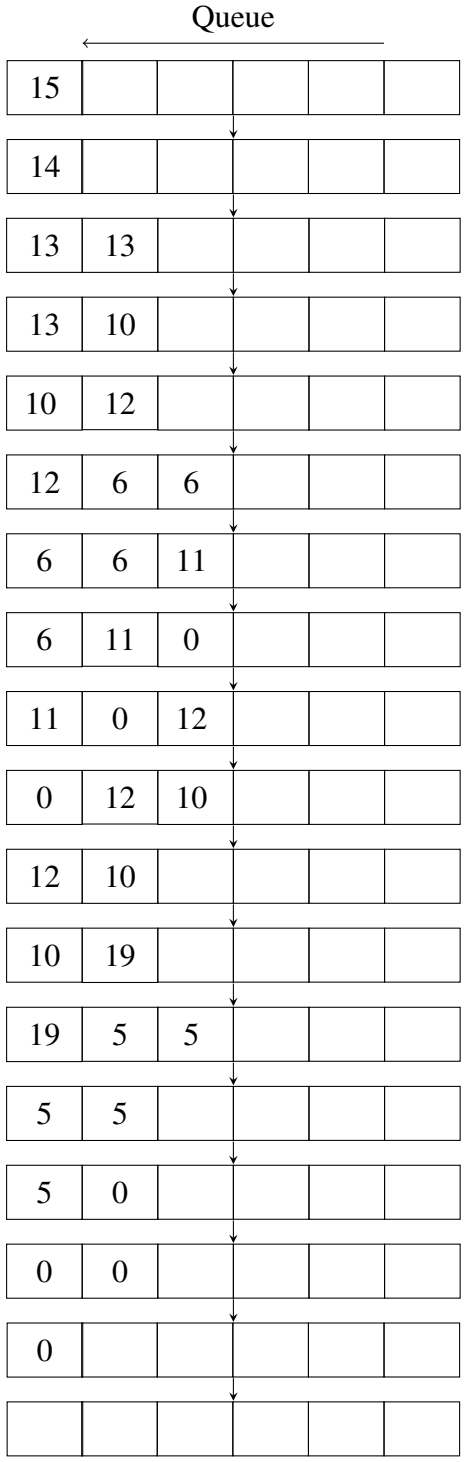

Figure 4: Queue q in the process of enqueuing and dequeuing after visiting each minimum entry in matrix 3.4

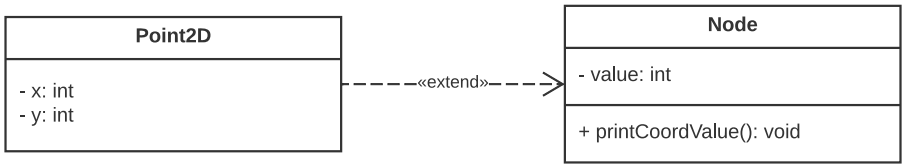

Figure 5: UML diagram of the variables and methods of the Point2D and Node classes

A `Point2D` object represents a single point in a 2-dimensional array, with class variables `x` and `y` representing the row and column coordinates respectively. `Node` extends `Point2D` and has an additional variable `value` of type float. This is the value of the node at the (x, y) coordinates of the matrix and is used to determine the algorithm's direction of travel.

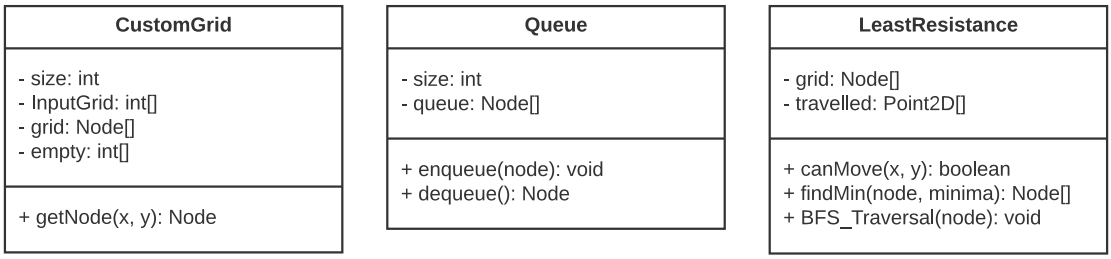

Figure 6: UML diagram of the instances and methods of the CustomGrid, Queue, and LeastResistance classes

`CustomGrid` takes in a 2-dimensional array of floats as an argument and generates an array with the same dimensions as the argument. `CustomGrid` is necessary because the grid it generates consists of `Node` objects rather than regular (x, y) coordinates.

The class `Queue` simulates a queue. Adding an element to the queue appends it to the back of the queue, and removing an element from the queue pops the first element of the queue and returns it. Both these operations are performed using `enqueue(node)` and `dequeue()` respectively. `enqueue(node)` was implemented using the function `append(node)` and `dequeue()` was implemented using the function `pop(0)`. Additionally, the queue implementation runs in $O(n_q)$ time because `pop(0)` runs in $O(n_q)$ time where n_q is the length of the queue. Indeed, the operation is analogous to removing and returning the first element of a list and then shifting the other $n_q - 1$ elements forward by one slot, which runs in $O(n_q)$ time.

`LeastResistance` is the main class in which the BFS traversal occurs. Its constructor takes in an object of type `CustomGrid` as an argument. If n and m are respectively the number of rows and columns of `CustomGrid`, then `LeastResistance` runs in $O(n \times m)$ time. This is because every node $N_i = (x_i, y_i)$ in the 2-dimensional array is visited at most once. Since N_i enters and is removed from the queue q at most once, each node is scanned at most once.

3.3 Results

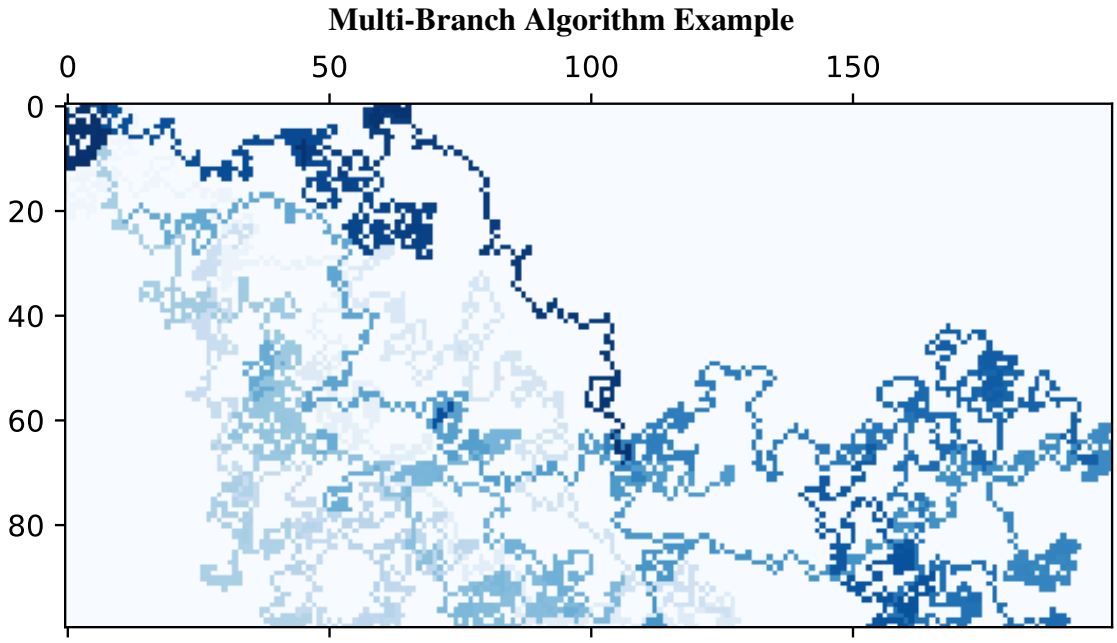

Figure 7: Figure of the multi-branch algorithm v2.0.1 traversing a randomly generated 100×200 matrix with a starting position of $(0, 0)$.

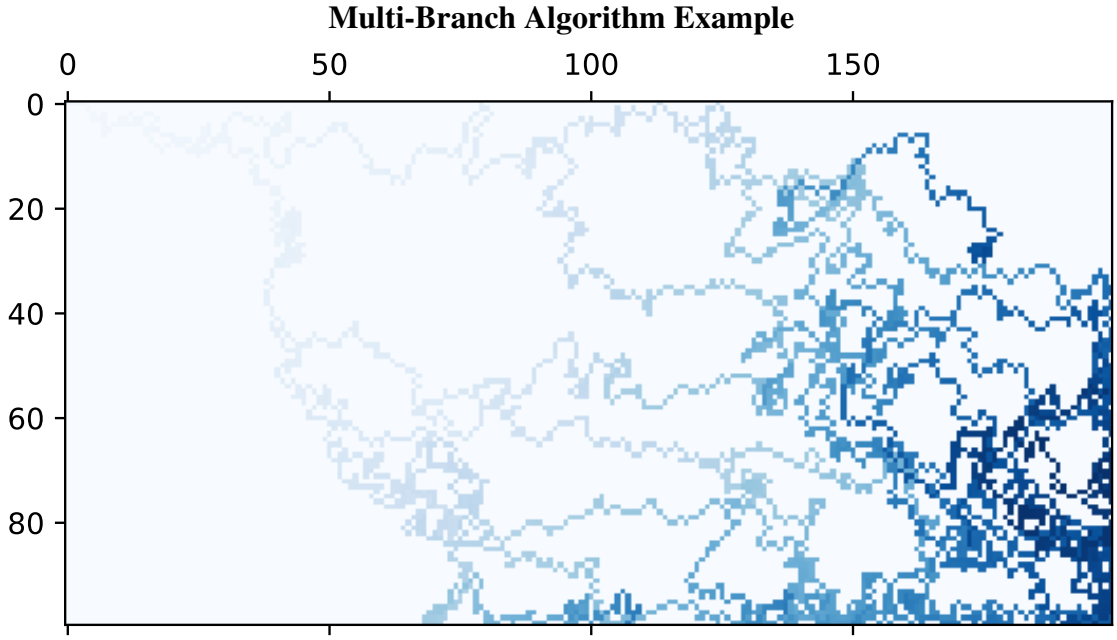

Figure 8: Figure of the multi-branch algorithm v2.1.1 traversing a randomly generated 100×200 matrix with a starting position of $(0, 0)$.

The path taken by each algorithm is recorded in blue. See GitHub for animations of these runs.

3.4 Discussion

The multi-branch algorithm was successful, to a certain extent, in faithfully replicating the path of an electric current travelling through an insulator.

Nonetheless, a quick observation of real-world Lichtenberg figures reveals that the main branch of a passing electric current is supposed to be wider than the branches that split from it. This issue is more aesthetic in nature and will not be the main focus of future modifications due to the fact that implementing the formation of wider branches will create additional clutter in the final figures.

Additionally, the branches that split are supposed to terminate quickly after splitting and should therefore be much shorter than the main branch. The latest version of the algorithm does not take this distinctive characteristic into account, therefore preventing the algorithm from fully achieving the objective of this project. This issue will be addressed in upcoming modifications.

3.4.1 Version 2.0.0

The downside of a DFS algorithm is that the branches are generated in a very unphysical manner. Rather than splitting into different branches simultaneously, this algorithm travels down a single branch completely before moving on to the next one. In order to better replicate the path of an electric current travelling through an insulator, the DFS method was abandoned in favour of BFS.

3.4.2 Version 2.1.0

This version of the algorithm achieved the objective of this project albeit with some remaining flaws, such as the absence of analogous substitutes for a cathode and an anode from which a real-world Lichtenberg figure is produced.

Additionally, the algorithm travels down multiple branches in an alternating manner rather than simultaneously. This issue is likewise more aesthetic in nature and as such, due to programming limitations, can be disregarded for the time being.

Also, it is worth mentioning that the queue implementation is inefficient because `pop(0)` runs in $O(n_q)$ time where n_q is the length of the queue. A faster way would be use a linked list so that `dequeue()` could be run in constant time. However, this operation is not what is bottlenecking the algorithm's runtime since $O(n_{\text{queue}})$ is faster than $O(m \times n)$, the runtime of `LeastResistance`, so this queue implementation is acceptable.

3.4.3 Version 2.2.0

A future modification to the multi-branch algorithm could involve inputting two different starting positions, with one acting as a cathode and another as an anode. Each would be the origin of an independent BFS tree similar to version 2.1.0 with the added condition that the branches of one tree try to travel towards the branches of the other. This would be the final major modification necessary to properly replicating Lichtenberg figures in a 2-dimensional array.

4 Conclusion

The objective of this project was to create an algorithm that could replicate the path of an electric current travelling through an insulator. The algorithm presented in this paper has achieved, to our present satisfaction, this objective. We find that version 2.1.1 is, to date, the algorithm that most closely mimics the creation of Lichtenberg figures and are currently trying to implement the modifications previously discussed in this paper. In the meantime, we invite the reader to experiment with theses algorithms and share with us their discoveries.