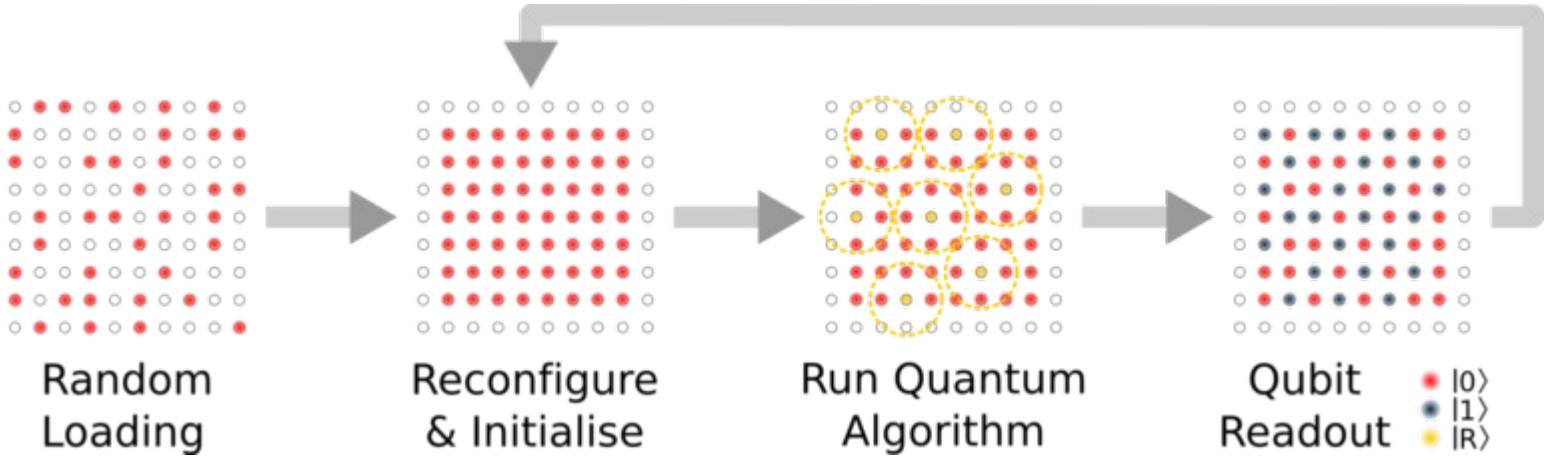
Atom Rearrangement

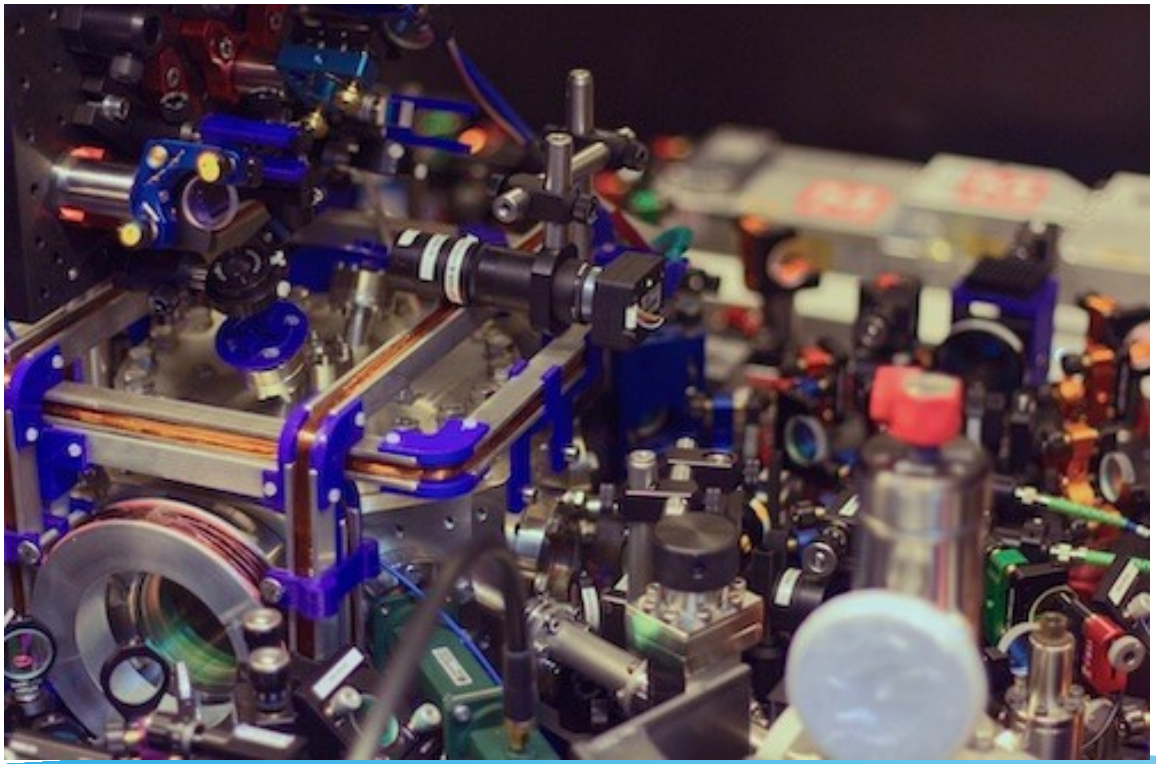


Context

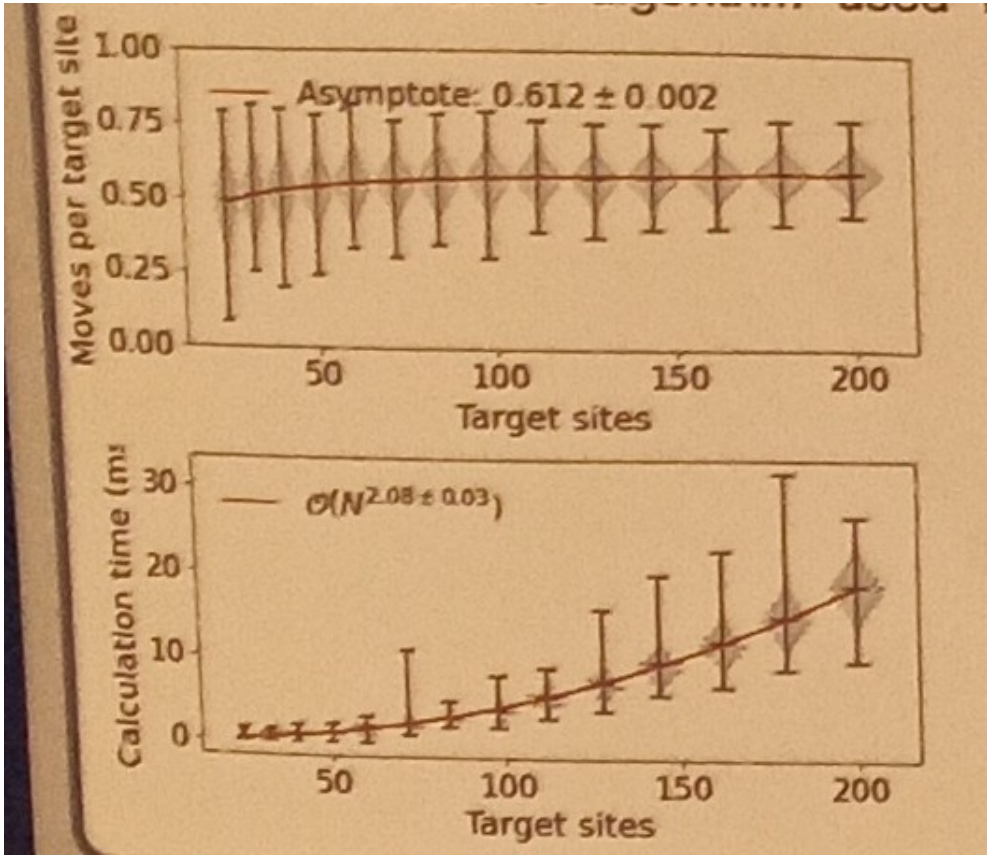
We have a quantum computer on floor 3 of John Anderson.

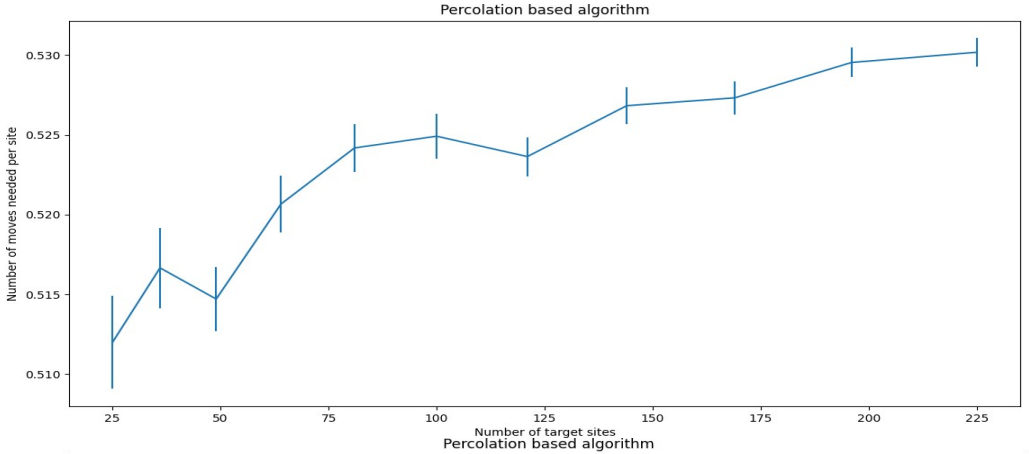
It works by trapping atoms in a lattice, so they can be used as qubits.

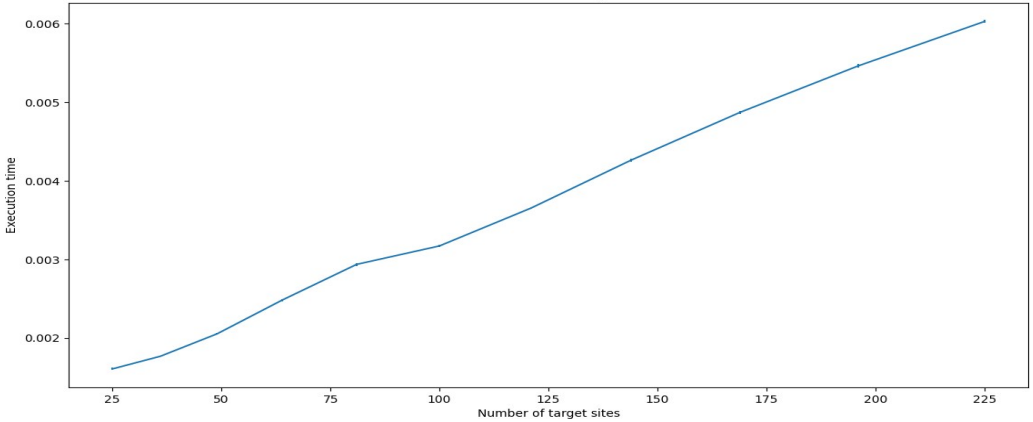
The problem is that each lattice point only has a 50% chance of capturing an atom.



Comparison against previous algorithm







Typical running times

A typical move takes around 150µs

Let’s take the 14x14 case with 196 sites

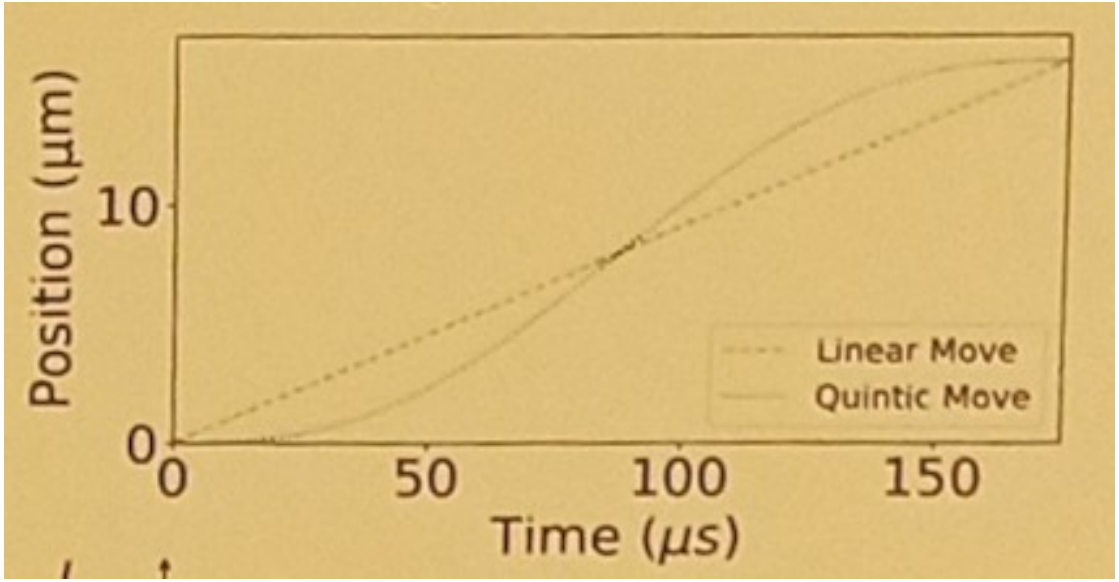
Old algorithm needs

`0.61*196*0.15ms + 20ms = 38.0ms`

New algorithm needs

`0.53*196*0.15ms + 6ms = 21.6ms`

Nearly twice as fast!

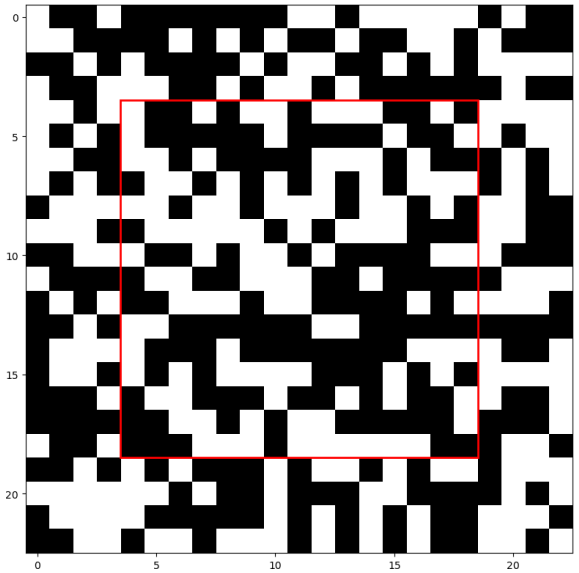


Algorithm Overview

1) Generate random grid
2) Identify atoms that are in the way, and remove them
3) Plot a path from every unfilled space to the perimeter
4) Find an atom outside the grid, and move it into the grid
5) Repeat 4) until filled

Step 1

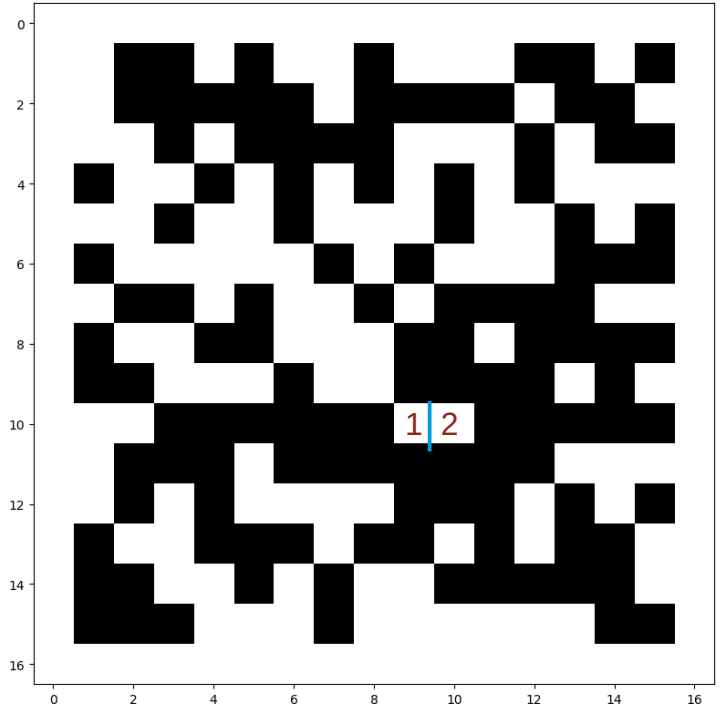
Generate random grid



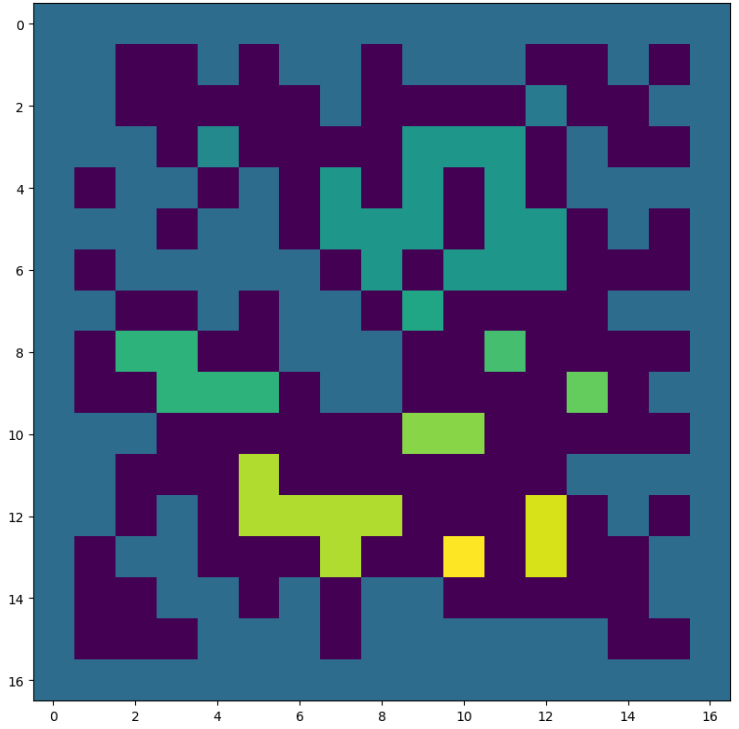

```
arr = np.random.randint(0,2,(N,N))
```

Step 2

If there's a path to hole 1 from the perimeter, there's a path to hole 2.



So we care about clusters of holes



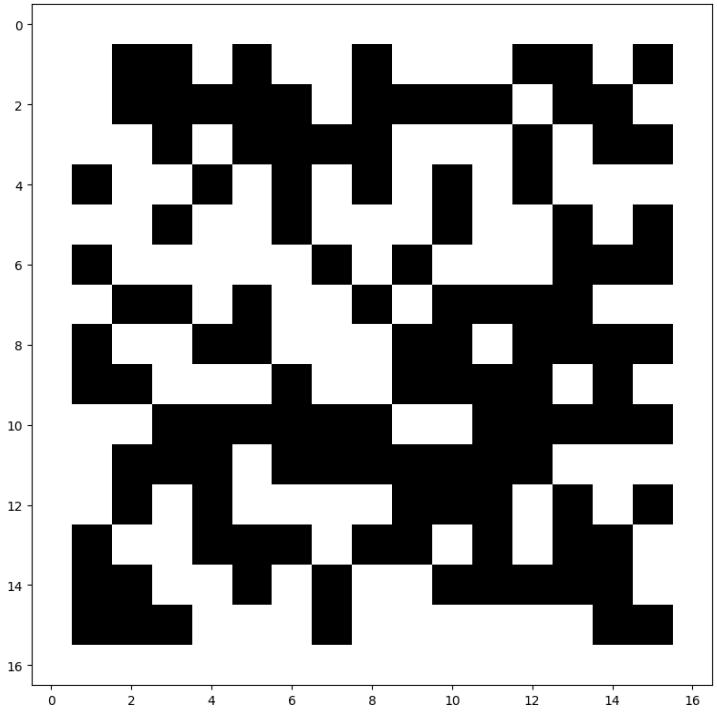
Step 2

We want to remove the minimal number of atoms so that we're left with only one cluster

At every atom position, we can count the number of unique clusters touching it. Let's call this `cluster_num`

If we remove an atom, the number of clusters reduces by `(cluster_num - 1)`

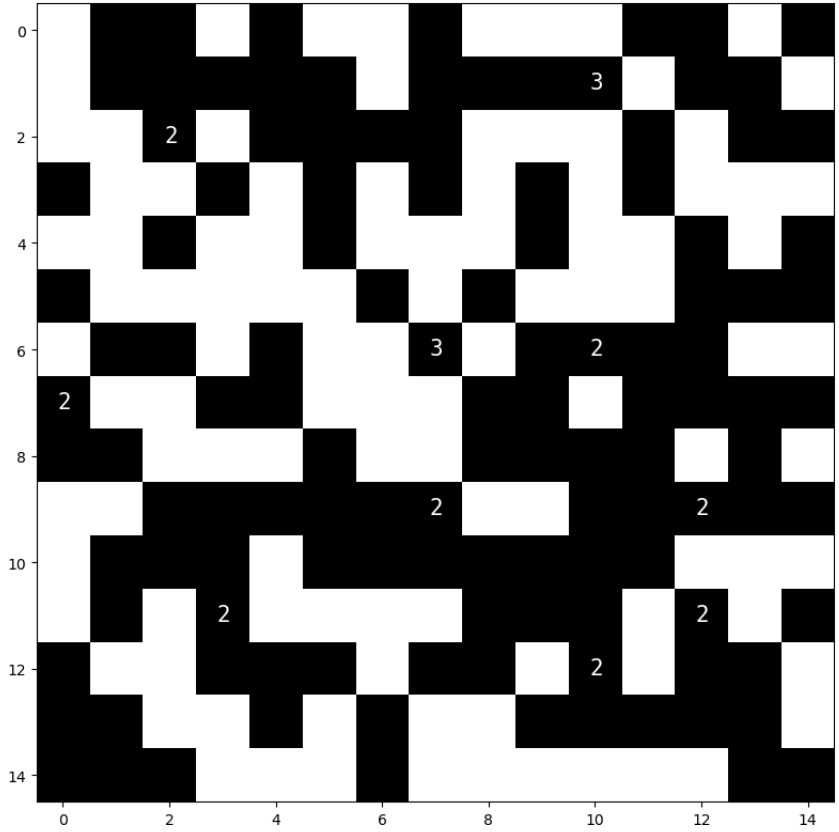
We sort the atoms by their cluster_num, and then remove them in that order until one cluster remains.



Step 2

We have now labelled the atoms that we will remove with their cluster numbers.

For the next step, we pretend these atoms no longer exist



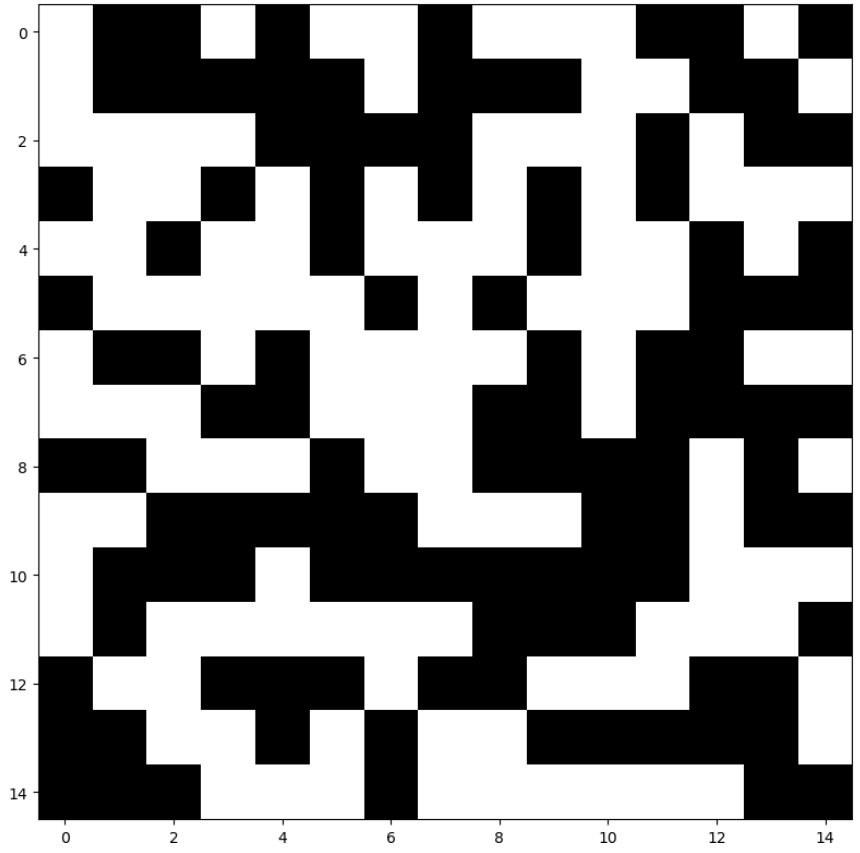
Step 3

Now we want to create paths to every point, using a technique called breadth first search.

First, we create a list, and we add the coordinates of holes on the perimeter to the list and mark them as searched.

Now, for every hole in the list, we check for any adjacent unsearched holes. If any are found, add them to queue and mark them as searched.

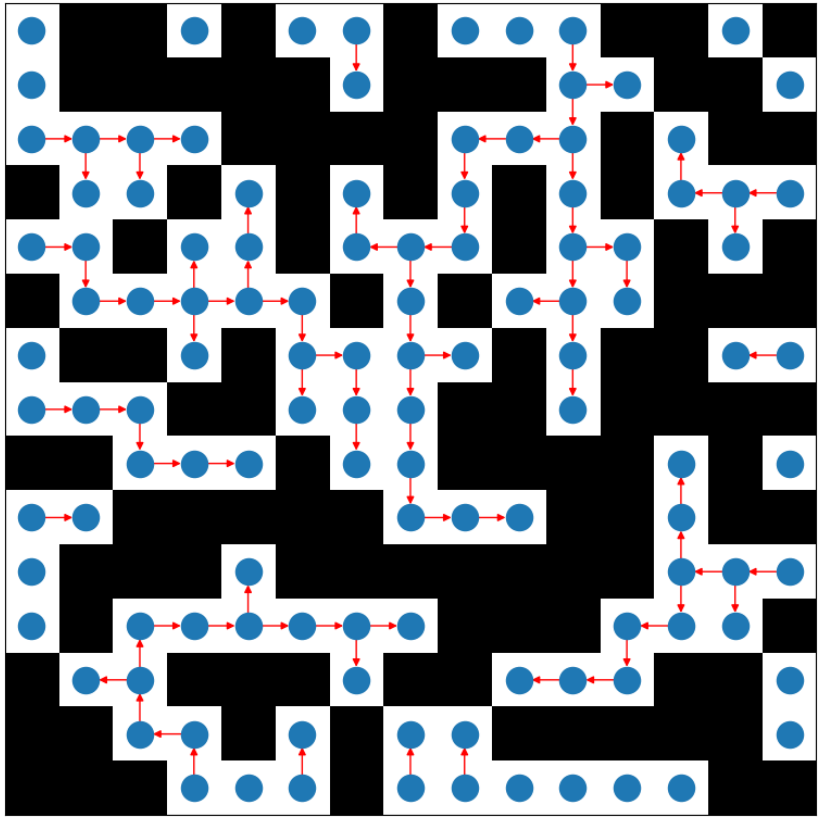
For every hole, we keep track of its ‘children’ by drawing an arrow between them.



Step 3

We can now properly deal with the atoms we pretended don’t exist – pick an arrow at that position and push it that direction, repeating until there isn’t an arrow to follow.

This is the push_to_bottom function, it pushes an atom to the bottom of our graph structure, filling in a hole and removing the arrow pointing to it.



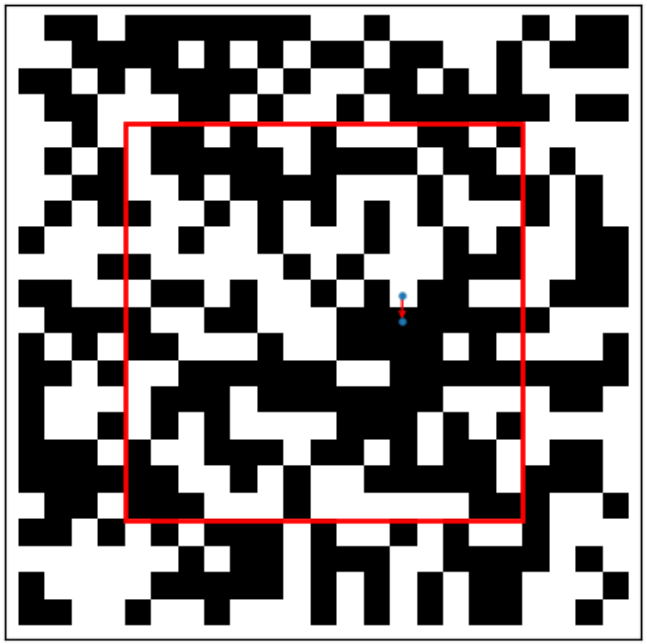
Step 4

1) Pick a hole on the perimeter
2) Use breadth first search to find the nearest atoms

    When an atom is found, move it to the hole, then call `push_to_bottom`

3) Repeat 2) until the hole is filled
4) Repeat 1) – 3) until every hole is filled

End



Theoretical Lower Bound

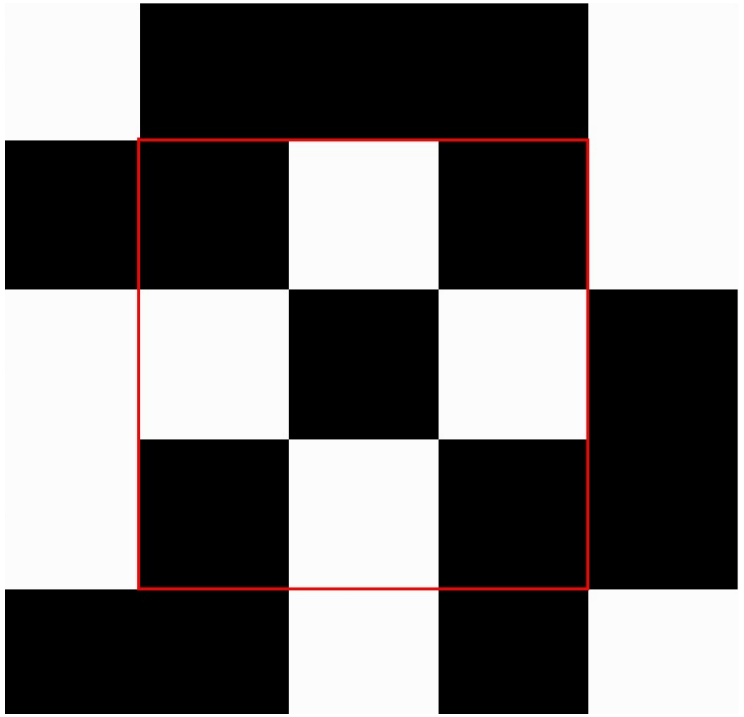
- Previous best known lower bound was 0.5 moves per site:
  - The chance of a target being unfilled is 50%
  - All unfilled targets require a move to be filled

We can do better!

Lower Bound

The best case scenario is that four clusters can be connected by removing a single atom

The pattern shown in red is a necessary condition for this to be possible, so we can just search the image for where this pattern appears.



The chance of this pattern appearing at a given position is

```
(1/2)**9 = 1/512
```

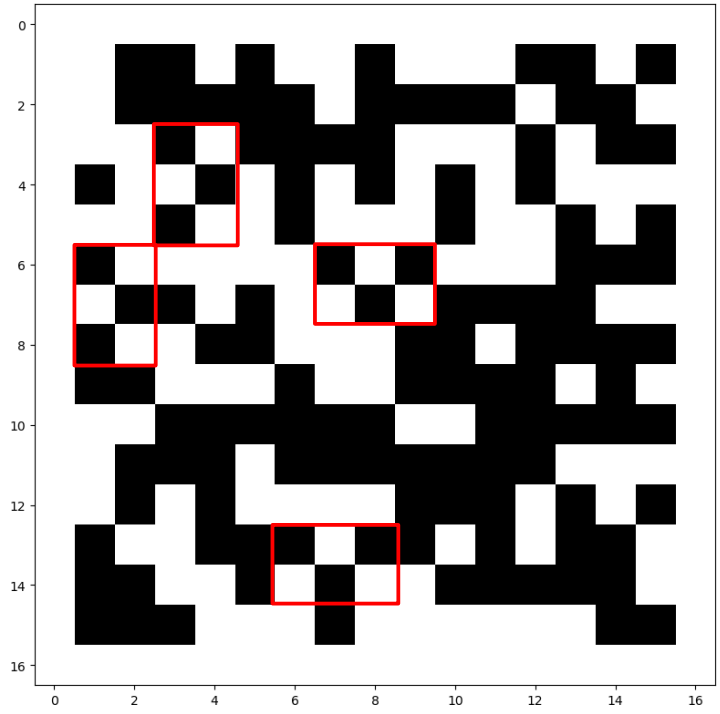
Lower Bound

Next best case is connecting three clusters with one move

This also has a pattern we can look for, but we have to check for all four possible rotations too.

The chance of one of these patterns appearing at a given position is

```
4*(1/2)**6 = 1/16
```



Lower Bound

Finally, how many clusters are there?

By generating a 10,000x10,000 grid and counting the number of clusters, we find that there are 0.0663 clusters per site.

- To connect these, we connect clusters four at a time with the `1/512 available moves per site, leaving 0.0604 clusters per site`
- `There are enough moves to connect the rest three at a time, requiring 0.0604/2 = 0.0302 more moves.`
- `This gives a total of 0.5322 moves required`

Upper Bound

By running our new algorithm on some huge examples, we get an upper bound of around 0.545

This gives a true optimum somewhere between 0.545 and 0.532