Percolation and the Invasion Process

September 2021

1 Question 1

The chance any node is in an infinite cluster of its descendants is independent of the bonds above it, so must be equal to ϕ_p for all nodes. The node '1' has two child nodes '11' and '12'. For each, the probability it is in an infinite cluster of descendants is ϕ_p so the chance '1' is in an infinite cluster through '11' is $p\phi_p$, and same goes for '12'. Since these probabilities are independent the chance '1' is in an infinite cluster with any of its descendants is $p\phi_p + p\phi_p - p^2\phi_p^2$ by the inclusion-exclusion principle. We can then rearrange to get:

$$\phi_p = 2p\phi_p - p^2\phi_p^2 = 2p\phi_p - 2p^2\phi_p + p^2\phi_p^2 + 2p^2\phi_p - 2p^2\phi_p^2 = 2p(1-p)\phi_p + p^2(\phi_p^2 + 2\phi_p(1-\phi_p))$$

As desired. So ϕ_p satisfies $0 = p^2\phi_p^2 + (1-2p)\phi_p$, which has roots $\phi_p = 0$ and $\phi_p = \frac{2p-1}{p^2}$.

I claim ϕ_p is the maximal solution to the equation $\phi_p = 2p\phi_p - p^2\phi_p^2$, which I showed is a rearrangement of the equation given in the exercise sheet. Let $\pi(n)$ be the probability that a cluster of descendants of '1' reaches n branches down the tree. Thus $\pi(1)$ is the probability at least one of the first branches is 'switched on', which is $2p - p^2$ by inclusion-exclusion. Suppose ψ satisfies $\psi = 2p\psi - p^2\psi^2$, I want to show that $\psi \leq \pi(n) \forall n$, since this implies that $\psi \leq \phi_p$. I will proceed by induction; we can assume $\psi \leq 1$ since ϕ_p is a probability. So $\psi = 2p\psi - p^2\psi^2 \leq 2p - p^2 = \pi(1)$ which gives the base case n=1. Suppose it is true that $\psi \leq \pi(k)$. We note that the chance '1' is in a cluster of length $k+1$ through '11' is $p\pi(k)$, and same goes for through '12'. Since these are independent, by inclusion-exclusion the probability '1' is in any cluster of length $k+1$ is $2p\pi(k) - p^2\pi(k)^2$. So $\pi(k+1) = 2p\pi(k) - p^2\pi(k)^2 \geq 2p\psi - p^2\psi^2 = \psi$ so also true for $\pi(k+1)$. Therefore by induction $\psi \leq \pi(n)$ for all n, which implies that $\psi \leq \phi_p$ and therefore that ϕ_p is the maximal solution to the equation.

Knowing that ϕ_p takes the maximal solution to the equation gives us the closed form of ϕ_p:

$$\phi_p = \begin{cases} 0 & \text{if } p < \frac{1}{2} \\ \frac{2p-1}{p^2} & \text{if } p \geq \frac{1}{2} \end{cases}$$

The probability that Eve is in an infinite open cluster is the probability that at least one of Eve's children is in an infinite open cluster. Since we know the probability '1' is in an open cluster is ϕ_p independent of the probability that '2' or '3' is we can once again use inclusion-exclusion to find that $\theta_p = 3p\phi_p - 3p^2\phi_p^2 + p^3\phi_p^3$. A graph of this formula is shown in figure 1

2 Question 2

If $p \leq \frac{1}{2}$ then $\phi_p = 0$ so by symmetry, given a node the probability it is in an infinite cluster of its descendants is zero. Since there are a countable number of nodes, by countable additivity the probability any node is in an infinite cluster of its descendants is zero. So there are almost surely no infinite clusters.

If $0.5 < p < 1$ let $\{v_i\}$ denote the set of nodes in the graph that do not have a bond with their parent, I will call them orphans. If two orphans are in infinite clusters those clusters must be distinct

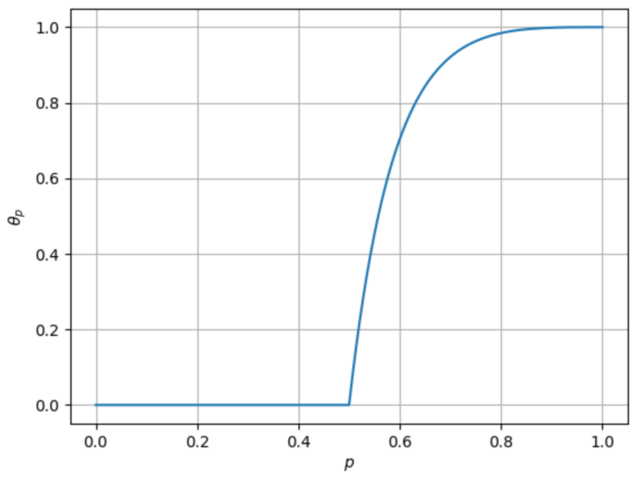

Figure 1: Plot of θ_p against p

since one orphan cannot be a descendant of another orphan, so the number of infinite clusters is greater than the number of orphans in infinite clusters. Also the probability a given orphan is in an infinite cluster of descendants is independent of the nature of any node that is not one of its descendants. So the probability an orphan is in an infinite cluster is ϕ_p independent of all the other orphans. Since $p < 1$ the probability a given node is an orphan is $1 - p > 0$ so $\mathbb{P}(|\{v_i\}| = \infty) = 1$, so there are almost surely infinitely many orphans, and since $p > 0.5$ $\phi_p > 0$ so each orphan independently has a positive probability of being in an infinite cluster, so there are almost surely infinitely many orphans in infinite clusters, which means there are almost surely infinitely many infinite clusters.

3 Question 3

We can instead consider the paths formed by first picking some direction from the origin, and henceforth picking one of the three paths that is not back the way you came. The number of such paths of length n is $4 \times 3^{n-1}$ by simply multiplying the number of choices you make in determining a path. Also any self-avoiding path must be of such a form, although not vice-versa. Therefore we have that $\sigma_n \leq 4 \times 3^{n-1}$ so $\sigma_n^{\frac{1}{n}} \leq (4 \times 3^{n-1})^{\frac{1}{n}} = \frac{4}{3}^{\frac{1}{n}} \times 3$. Therefore

$$\lambda = \limsup_{n\to\infty} \sigma_n^{\frac{1}{n}} \leq \limsup_{n\to\infty} \frac{4}{3}^{\frac{1}{n}} \times 3 = 3$$

Let $\pi(n)$ denote the probability that there exists a self-avoiding walk along open edges starting at (0,0) of length n.

Let ϕ_p denote the probability (0,0) is in an infinite open cluster.

If (0,0) is in an infinite open cluster then for any n we can find a node in the cluster that is at least of distance n away from (0,0). Then the path of open edges connecting such a node to (0,0) will be at least n long, so there will exist a self-avoiding path along open edges of length n. This implies that $\pi(n) \geq \phi_p$ for all n.

The probability that a given self-avoiding path of length n consists of open edges is p^n. Let A_i be the event that the ith path consists of open edges, then the probability at least one self-avoiding path consists of open edges is $\mathbb{P}(\cup_i A_i) \leq \sum_i \mathbb{P}(A_i) = p^n \times \sigma_n$. So:

$$p^n \times \sigma_n \geq \pi(n) \geq \phi_p \forall n$$

I claim that if $p < \lambda^{-1}$ then $\phi_p = 0$. Certainly have that $p\lambda < 1$. This then gives:

$$\begin{aligned}
&\limsup_{n\to\infty} p\sigma_n^{\frac{1}{n}} < 1 \\
&\implies \exists c < 1, N \in \mathbf{N} \text{ s.t. } \forall n > N,\, p\sigma_n^{\frac{1}{n}} < c \\
&\implies \forall n > N,\, p^n \times \sigma_n < c^n \\
&\implies \forall n > N,\, \phi_p < c^n \\
&\implies \phi_p = 0 \; \square
\end{aligned}$$

Then since the square lattice has countably many nodes by countable additivity and symmetry the probability that any node is in an infinite open cluster is zero, so if $p < \lambda^{-1}$ there are almost surely no infinite open clusters, which implies that $p_c \geq \lambda^{-1}$.

4 Question 4

Let G_n be the subgraph of G with vertices V_n defined by:

$$V_n = \{(k, l) : 0 \leq k \leq n, 0 \leq l \leq n - 1\}$$

Now let $\bar{G}_n$ be the dual graph of G_n, with vertices V_n' defined as:

$$V_n' = \{(k + \frac{1}{2}, l - \frac{1}{2}) : 0 \leq k \leq n - 1, 0 \leq l \leq n\}$$

With edges between nodes that are distance one apart. A visualisation of these graphs overlaid on each other can be seen in figure 2.

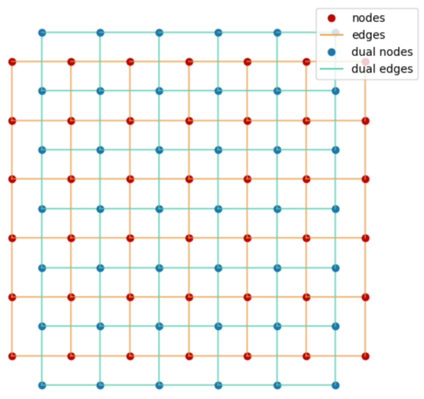

Figure 2: Visualisation of G_6 and $\bar{G}_6$

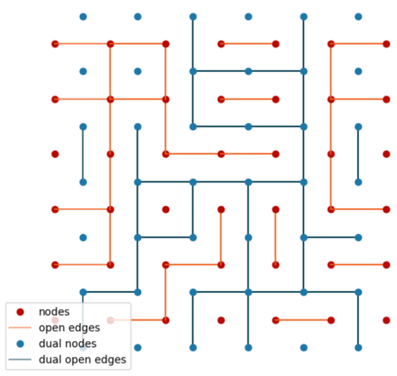

Figure 3: Example of open edges

Now suppose we have an edge $\bar{e}$ in $\bar{G}_n$, we wish to say whether this edge is open or closed. Note that there is a unique edge e in G which intersects $\bar{e}$. We then say that $\bar{e}$ is open if e is closed, and vice versa. An example of this scheme showing the open edges in both G and $\bar{G}$ can be seen in figure 3.

Let A be the event that some node in the left boundary of G_n is connected to the right boundary via a path consisting of open edges, and let A' be the event that some node in the top boundary of $\bar{G}_n$ is connected to the bottom by a path of open edges. I claim that $A' = A^c$.

Certainly $A \cap A' = 0$, if not this says there is a both a left-right path and an up-down path, but then the intermediate value theorem implies that the paths intersect at some point, which contradicts our definition of open edges in $\bar{G}_n$. Therefore it is left to show that $A \cup A' = \Omega$, that is that at least one of these events must occur.

To do this it is enough to show that if there is no path from the left boundary of G_n to the right, then there must be one from the top boundary of $\bar{G}_n$ to the bottom. The other side then follows by symmetry of G_n and $\bar{G}_n$. Consider the following colour scheme on the vertices of G_n: given a vertex v, colour it red if there is a path from the left boundary of G_n to v, and green if there isn't. This is represented visually in figure 4.

We then not that the boundary of the red coloured vertices must be a p-open path in $\bar{G}_n$, since no red vertex in G_n can be connected to a blue vertex by an edge. Therefore the boundary of the red vertices must be a p-open path from the top of $\bar{G}_n$ to the bottom, which completes the proof. Figure 5 shows a picture of how the boundary of the red vertices becomes a p-open path in the dual graph.

To finish the proof that $\mathbf{P}(A) = \frac{1}{2}$, we note that because of the symmetry between G_n and $\bar{G}_n$ and the fact that the probability of an edge being open or closed in G_n is the same:

$$\mathbf{P}(A) = \mathbf{P}(A') = \mathbf{P}(A^c) = 1 - \mathbf{P}(A) \implies \mathbf{P}(A) = \frac{1}{2}$$

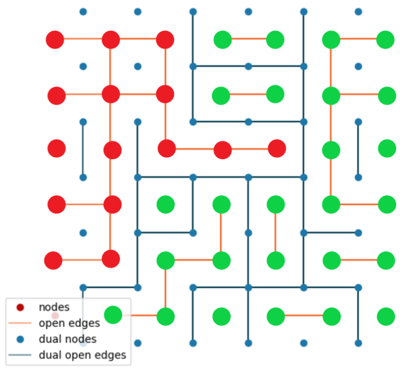

Figure 4: Visualisation of the partition of vertices

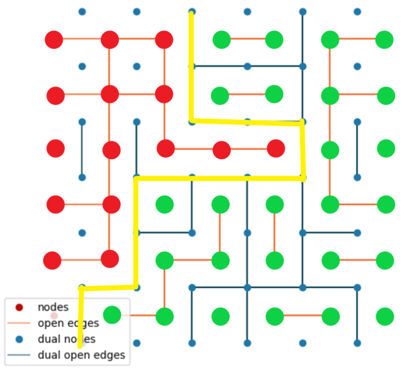

Figure 5: the boundary of the red partition

5 Question 5

Suppose $p_c > \frac{1}{2}$, then by (1) there exists $\psi_{\frac{1}{2}}$ such that $P_{\frac{1}{2}}(0 \leftrightarrow \partial S_n) < e^{-n\psi_{\frac{1}{2}}}$. Let A_i be the event that there is a path in V_n from $(0, i)$ to the right edge. Then $A = \cup A_i$. We define the open sphere about $(0, i)$ as

$$S_n(0, i) = \{(x, y) \in \mathbf{Z}^2 : d((0, i), (x, y)) \leq n\}$$

We also note that if there exists a path from $(0, i)$ to the right edge in V_n then that same path is a path in the box $S_n(0, i)$ to the boundary $\partial S_n(0, i)$, so $P_{\frac{1}{2}}(A_i) \leq P_{\frac{1}{2}}((0, i) \leftrightarrow \partial S_n(0, i))$ and then by symmetry

$$P_{\frac{1}{2}}(A_i) \leq P_{\frac{1}{2}}((0, i) \leftrightarrow \partial S_n(0, i)) = P_{\frac{1}{2}}(0 \leftrightarrow \partial S_n) \leq e^{-n\psi_{\frac{1}{2}}}$$
$$\implies \frac{1}{2} = P_{\frac{1}{2}}(A) = P_{\frac{1}{2}}(\cup A_i) \leq \sum_i P_{\frac{1}{2}}(A_i) \leq ne^{-n\psi_{\frac{1}{2}}}$$

But $\psi_{\frac{1}{2}} > 0$ so $\lim_{n\to\infty} ne^{-n\psi_{\frac{1}{2}}} = 0$, but this contradicts our previous result that $\frac{1}{2} \leq ne^{-n\psi_{\frac{1}{2}}}$ for all n, so must have that $p_c \leq \frac{1}{2}$.

6 Question 6

My algorithm stored three main pieces of data: which nodes have been invaded, the edges that lead from these nodes to uninvaded nodes, and the values of U_{e_n} travelled down so far. The probabilities U_e along the edges are generated dynamically as the edges are reached. The reason only edges between invaded nodes are NOT stored is to avoid revisiting a vertex. The number of edges stored is certainly less than four times the number of nodes, and the number of data points U_{e_n} is one less than the number of nodes so the total storage used for the first n nodes is less than $6n$, so the storage used is $O(n)$.

Initially the algorithm worked by checking every available edge each iteration of the algorithm. Since the number of edges available is roughly proportional to the number of invaded vertices, each iteration was $O(n)$, which meant the algorithm as a whole was $O(n^2)$. This is a high time complexity and resulted in a slow algorithm that was practically unable to generate results for $n > 40000$ or so, as a result I attempted to make it somewhat more efficient.

I realised that the main inefficiency is the fact that despite a maximum of three new edges being introduced each iteration, every single edge was being checked each time. To reduce this inefficiency I introduced a value c, the algorithm would then store up to c of the lowest values available along the edges in order. Let lowestList be the list of edges with highest invasion potential (lowest probability). The algorithm now check whether lowestList is empty and then proceeds with either process 1 or process 2.

Process 1

lowestList empty
↓
Check every invaded node to fill lowestList with lowest c nodes
↓
Return to top

Process 2

lowestList not empty
↓
Invade first node in lowestList
↓
Check invasion potential of new edges to see if they fit in lowestList
↓
Return to top

Figures 6 through 9 show a visualisation of this process for the first few vertices.

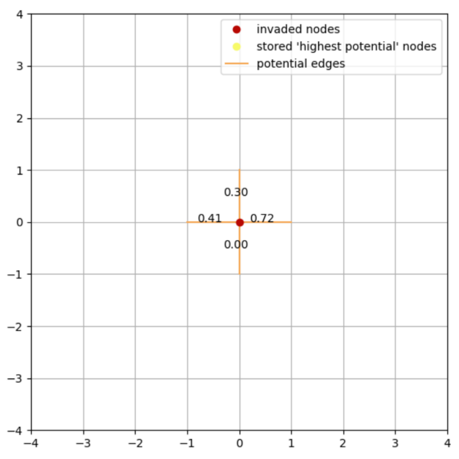

Figure 6: n=1

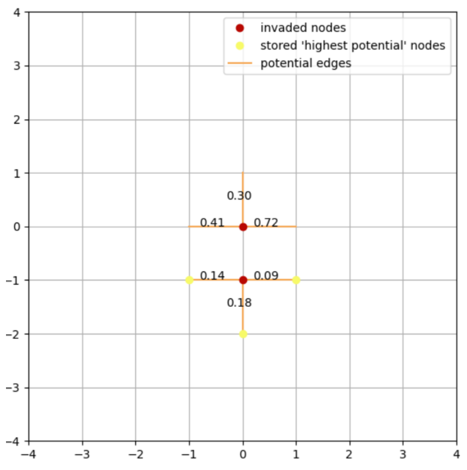

Figure 7: n=2

Since it has to check every invaded node, process 1 still has complexity $O(n)$, however crucially process 2 has complexity $O(1)$. As a result the time complexity of the algorithm as a whole depends on how often the list of lowest values gets depleted. I was not able to work this out theoretically, and I suspect that the probability of the list being depleted (and so having to do an $O(n)$ operation)

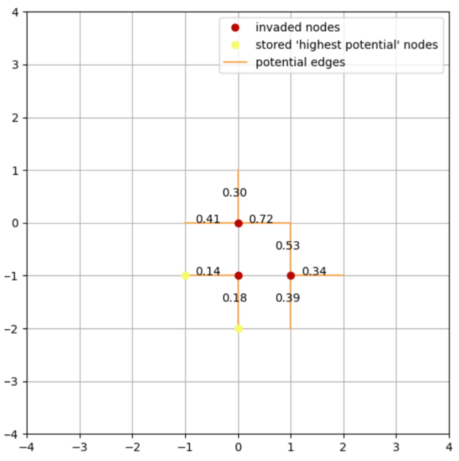

Figure 8: n=3

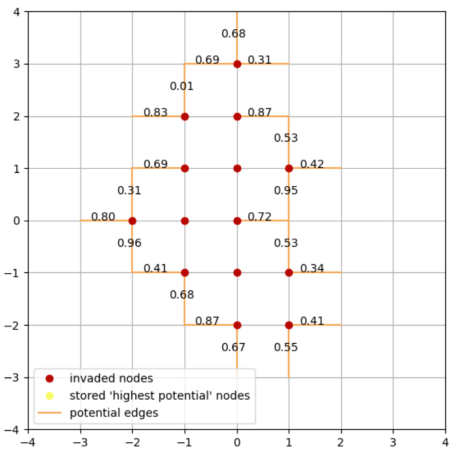

Figure 9: n=14

is constant with n, which would result in the complexity being $O(n^2)$ in the long term, however for values of n in the range of up to 150000 it is roughly O(n), see figure 10.

As such this (admittedly heuristic) enhancement to the algorithm is very useful and significantly improves the speed of the program for the problems that need to be handled in this project. A more complete and mathematically satisfying solution is to use a binary tree to store the edge data, organised such that the lowest value is always at the top of the tree. This was suggested by Yder J. Masson and Steven Pride [1], who also showed that using this method the algorithm had time complexity O(nlogn).

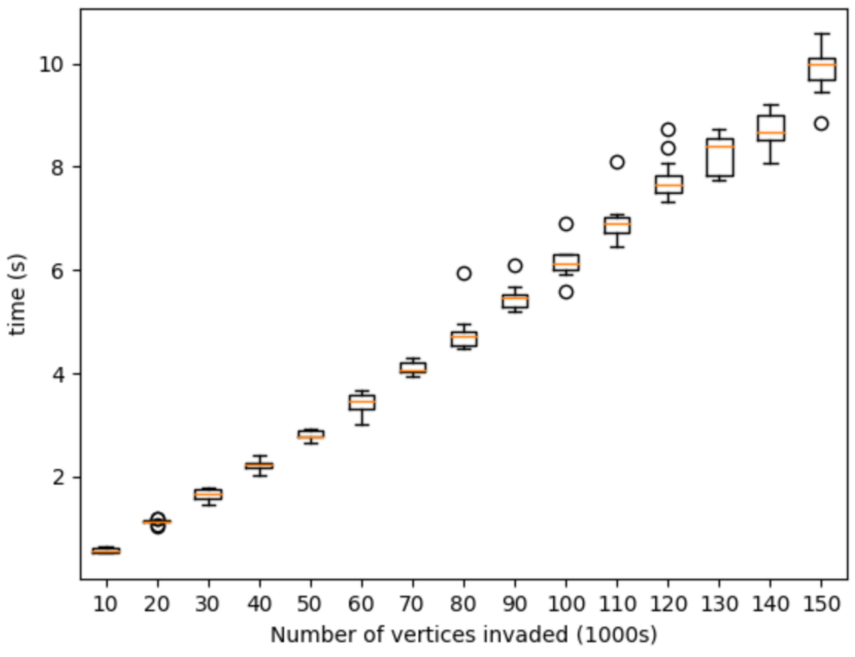

Figure 10: distribution of time taken for 12 runs at various values of n

7 Question 7

The program records all the values of U_{e_n} in order, and we need to estimate $\limsup_{n\to\infty} U_{e_n}$; that is the limit of the suprema as we look deeper and deeper into the data. To represent what we are looking for I ran the program and graphed the values of U_{e_n}, along with a graph of both the averages and the maxima over each segment of 1000 iterations, as can be seen in figure 11.

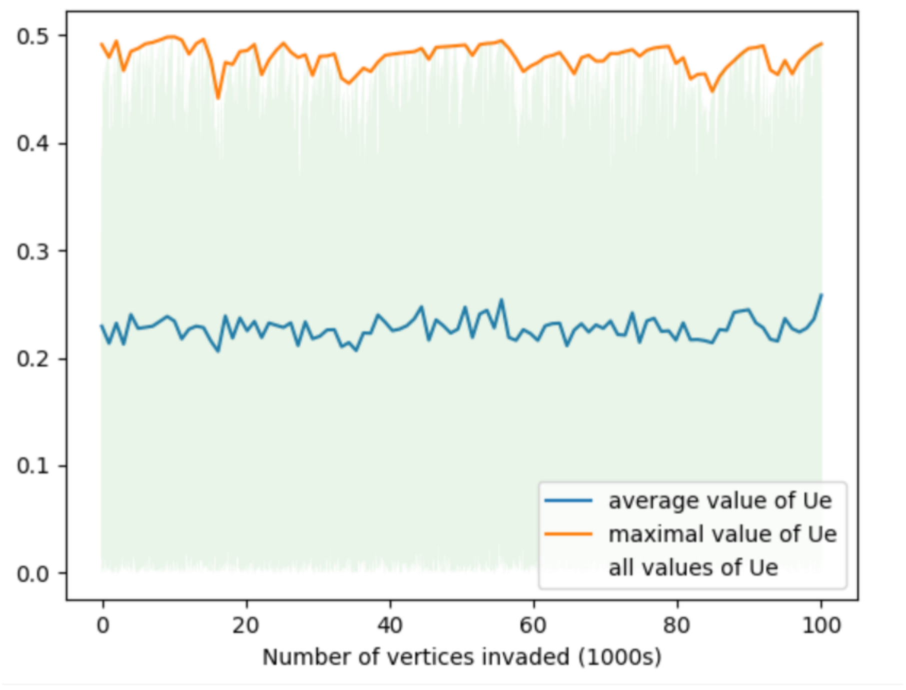

Figure 11: Values of U_{e_n} over one run of the algorithm

This graph shows (as we would expect) that the vast majority of values lie below an upper bound of about 0.5, with the maximal values in the graph approaching this limit. This raises raises an interesting question about how best to approximate this limsup, given that we can only have finitely many values. One must balance the need to exclude earlier values with the desire to take a supremum over as many values as possible. In the end I decided to calculate the first 100,000 values and take a supremum over the last 95000. The results are listed in table 1.

0.4985729514796475
0.4976729463265634
0.4939142517433619
0.4961574030261928
0.5008956272626989
0.4974728562416522
0.4918574624819937
0.5021117856592309
0.4937565493533305
0.4958892239345030

Table 1: p_c estimates for square lattice

These results average to about 0.497, which suggests that p_c may well be 0.5, with the discrepancy explained by the issues with estimating a limsup. However it is worth noticing that this is quite a large error so results in other cases should not be taken to be precise.

To estimate p_c for $\mathbb{L}^3$ I simply change the dimension parameter from two to three and run the program as before. I decided to use 100,000 iterations and take the maximum of the last 95,000. The results for ten runs of this program are listed in table 2, along with the results for $\mathbb{L}^4$ and $\mathbb{L}^5$ for good measure.

	Dimension		
	3	4	5
1	0.2502511902154445	0.15978217462816524	0.11811601445243247
2	0.25114951535866414	0.16032856173553678	0.11744186481231333
3	0.2489738999469402	0.15984965999813772	0.11846831624504117
4	0.2503937601150228	0.16281070747052517	0.11845109056285608
5	0.24951587582753976	0.1636724838679311	0.11902901512141395
6	0.24926625879067665	0.1595165078132269	0.11757343460173209
7	0.24837482089030893	0.16201019975180475	0.1195030152100679
8	0.2460997198449636	0.1615492181084649	0.11814158653200613
9	0.25370249372617326	0.16041935071876856	0.11889433275346528
10	0.24821061016768042	0.16118121191842782	0.11915128579998113
average	0.24959381448834145	0.16111200760109892	0.11847699560913094

Table 2: p_c estimates for lattices of different dimension

8 Question 8

θ_p is the probability that our starting node is in an infinite open cluster. Therefore in order to estimate θ_p it would be useful to estimate, given fixed bond probabilities U_{e_n}, whether (0,0) is in an infinite cluster. We notice that if the invasion process can continue indefinitely without crossing a bond with invasion potential greater than p, then (0,0) must be in an infinite cluster of p-open edges. Equally, if the invasion process *does* cross an edge with potential greater than p, then the cluster before that edge was crossed must have been surrounded by p-closed edges, and thus (0,0) was in a finite open cluster.

This allows us to make a rough estimate: if we calculate U_{e_n} for $1 \leq n \leq N$ then let $M = \max_{1 \leq n \leq N} U_{e_n}$, we can estimate that (0,0) is in an infinite cluster if $p \geq M$, and isn't if $p < M$. Then by averaging over many different trials we can get an estimate for θ_p.

The main potential inaccuracy with this technique stems from the fact that we can only simulate the invasion process for a finite amount of time; this means we must be wary of cases where $\max_{1 \leq n \leq N} U_{e_n} < \max_{1 \leq n \leq \infty} U_{e_n}$, however given $\limsup_{n \to \infty} U_{e_n} = p_c$, these errors should be localised around p_c, which in this case is 0.5. In particular, for values of p that are slightly less than 0.5 we may estimate that θ_p is slightly greater than zero, when it in fact should be zero by definition of p_c.

Performing this estimation is algorithmically fairly simple, we perform the invasion process 2000 times, each time with n=10000, and record that θ_p is 1 for $p \geq M$ and 0 for $p < M$. Then take an average over all these estimates to get an estimate for θ_p. Figure 12 shows the results of this, the inaccuracy around $p = 0.5$ can be seen from the fact that it is non-zero in a small region of $p < 0.5$. This error could be reduced by using a larger value of n, however time constraints would start to cause an issue, as this example took about 15 minutes.

References

[1] Yder J. Masson, Steven Pride. A Fast Algorithm for Invasion Percolation. Transport in Porous Media, Springer Verlag, 2014, 102 (2), pp.301 - 312. ff10.1007/s11242-014-0277-8ff. ffhal-01653926f

9 Code Listing

9.1 Question 1

```
import numpy as np
import matplotlib.pyplot as plt
xaxis = np.linspace(0,1,1000)
```

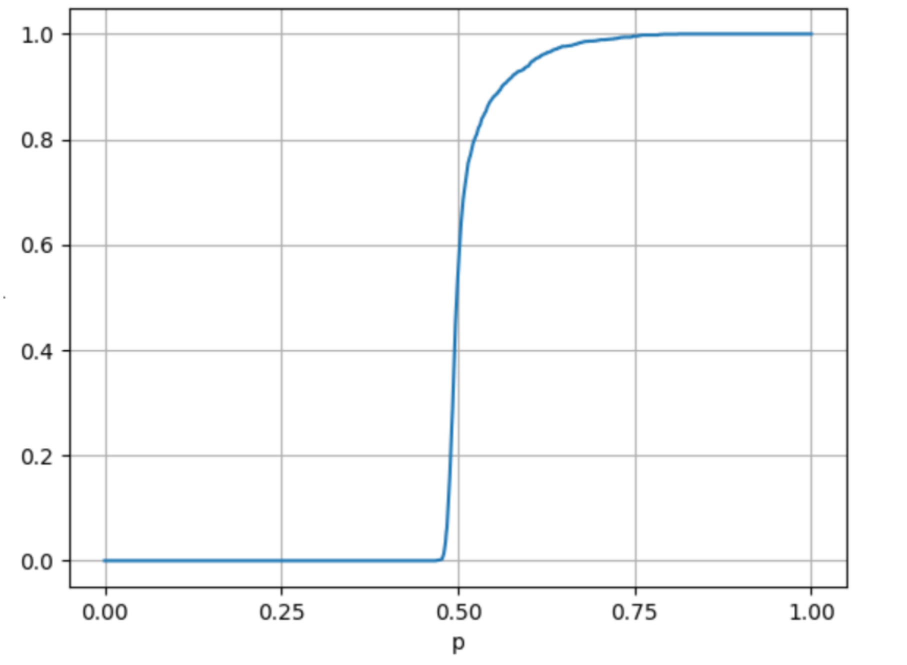

Figure 12: Estimate of θ_p against p over 2000 tests with n=10000

```
yaxis = np.zeros((1000))
for i in range(1000):
    p = xaxis[i]
    if p<=0.5:
        phi=0
    else:
        phi=(2*p-1)/p**2
    yaxis[i]=3*p*phi-3*p**2*phi**2+p**3*phi**3
plt.plot(xaxis,yaxis)
plt.xlabel(r'$p$')
plt.ylabel(r'$\theta_p$')
plt.grid()
plt.show()
```

9.2 Question 4

9.2.1 First Graph

```
import numpy as np
import matplotlib.pyplot as plt
class graph:
    def __init__(self,nodes=None,dimension = 2,lowestEdges=[],c=20):
        if nodes == None:
            nodes = {
        }
        self.nodes = nodes
        self.onedge = set()
        self.dimension = dimension
        self.c=c
    def addEdge(self,node1,node2):
        node1=tuple(node1)
        node2=tuple(node2)
        rand = np.random.random()

        if node1 not in self.nodes[node2]:

            self.nodes[node2].append([node1,rand])
            self.nodes[node1].append([node2,rand])

    def addNode(self,node):
        node = tuple(node)


        self.nodes[node]=[]
        for i in range(self.dimension):
            for j in [-1,1]:
                neighbour = list(node)
                neighbour[i]+=j
                neighbour = tuple(neighbour)
                if neighbour in self.nodes:
                    self.addEdge(neighbour,node)
n = 6
grid1 = graph()
grid2 = graph()
for i in range(n+1):
    for j in range(n):
        grid1.addNode(np.array([i,j]))
        grid2.addNode(np.array([j+0.5,i-0.5]))
fig = plt.figure(figsize = (10,10))
axis = fig.add_subplot(1,1,1)
plt.plot([],[],'o',color='#b80802',label='nodes')
plt.plot([],[],color = '#f5a958',label='edges')
plt.plot([],[],'o',color='#1d77a8',label='dual nodes')
plt.plot([],[],color = '#63d4b8',label='dual edges')
for i in grid1.nodes:
    for j in grid1.nodes[i]:
        plt.plot(np.array([i[0],j[0][0]]),np.array([i[1],j[0][1]]),color =
        '#f5a958',alpha=0.5)
    plt.plot(i[0],i[1],'o',color='#b80802')
for i in grid2.nodes:

    for j in grid2.nodes[i]:
        plt.plot(np.array([i[0],j[0][0]]),np.array([i[1],j[0][1]]),color =
        '#63d4b8',alpha =0.5)
    plt.plot(i[0],i[1],'o',color='#1d77a8')
plt.xlim(-1,n+1)
plt.ylim(-1,n)
plt.gca().set_aspect('equal', adjustable='box')
plt.axis('off')
plt.legend()
plt.show()
```

9.2.2 Second Graph

```
import numpy as np
import matplotlib.pyplot as plt
np.random.seed(9)
class graph:
    def __init__(self,nodes=None,dimension = 2,lowestEdges=[],c=20):
        if nodes == None:
            nodes = {
        }
        self.nodes = nodes
        self.onedge = set()
        self.dimension = dimension
        self.c=c
    def addEdge(self,node1,node2):
        node1=tuple(node1)
        node2=tuple(node2)
        rand = np.random.random()

        if node1 not in self.nodes[node2]:

            self.nodes[node2].append([node1,rand])
            self.nodes[node1].append([node2,rand])

    def addNode(self,node):
        node = tuple(node)


        self.nodes[node]=[]
        for i in range(self.dimension):
            for j in [-1,1]:
                neighbour = list(node)
                neighbour[i]+=j
                neighbour = tuple(neighbour)
                if neighbour in self.nodes:
                    self.addEdge(neighbour,node)
n = 6
grid1 = graph()
grid2 = graph()
for i in range(n+1):
    for j in range(n):
        grid1.addNode(np.array([i,j]))
        grid2.addNode(np.array([j+0.5,i-0.5]))
fig = plt.figure(figsize = (10,10))
axis = fig.add_subplot(1,1,1)
plt.plot([],[],'o',color='#b80802',label='nodes')
plt.plot([],[],color = '#f06424',alpha=0.5,label='open edges')
plt.plot([],[],'o',color='#1d77a8',label='dual nodes')
plt.plot([],[],color = '#023c4f',alpha=0.5,label='dual open edges')
for i in grid1.nodes:
    for j in grid1.nodes[i]:
        if (i[0]==0&j[0][0]==0) or (i[0]==6&j[0][0]==6):
            pass
        elif j[1]>0.5:
            plt.plot(np.array([i[0],j[0][0]]),np.array([i[1],j[0][1]]),co
            lor = '#f06424',alpha=0.8)
        else:
            coord1 = ((-i[1]+j[0][1])/2+(i[0]+j[0][0])/2,(i[0]-j[0][0])/2+(i[1]+j[
            0][1])/2)
            coord2 =  ((i[1]-j[0][1])/2+(i[0]+j[0][0])/2,( -i[0]+j[0][0])/2+(i[1]+j
            0][1])/2)
            plt.plot(np.array([coord1[0],coord2[0]]),np.array([coord1[1],
            coord2[1]]),color = '#023c4f',alpha=0.8)
    plt.plot(i[0],i[1],'o',color='#b80802')
for i in grid2.nodes:
    plt.plot(i[0],i[1],'o',color='#1d77a8')
plt.xlim(-1,n+1)
plt.ylim(-1,n)
plt.gca().set_aspect('equal', adjustable='box')
plt.axis('off')
plt.legend()
plt.show()
```

9.3 Question 6

9.3.1 Diagrams

```
import numpy as np
import time
import matplotlib.pyplot as plt
np.random.seed(1)

class graph:
    def __init__(self,nodes=None,dimension = 2,lowestEdges=[],c=20):
        if nodes == None:
            nodes = {
        }
        self.nodes = nodes
        self.onedge = set()
        self.dimension = dimension
        self.lowestEdges=lowestEdges
        self.c=c
    def addLowest(self,newNode,filling = False):
        if self.lowestEdges == []:
            if filling:
                self.lowestEdges = [newNode]
                return None
            else:
                return None
        newRand = newNode[1]
        length = len(self.lowestEdges)
        if length ==self.c:
            filling = False
        if newRand > self.lowestEdges[-1][1]:
            if not filling:
                pass
            else:
                self.lowestEdges= self.lowestEdges+[newNode]
                return None
        else:
            for i in range(length-2,-1,-1):
                if newRand >self.lowestEdges[i][1]:
                    self.lowestEdges = self.lowestEdges[0:i+1]+[newNode]+self.lowes
                    return None
            self.lowestEdges=[newNode]+self.lowestEdges[0:self.c-1]
            return None
    def addEdge(self,node1,node2):
        node1=tuple(node1)
        node2=tuple(node2)
        rand = np.random.random()
        if node1 not in self.nodes:
            if node2 in self.nodes:
                self.nodes[node2].append([node1,rand])
                self.addLowest([node1,rand],c)

            else:
                print('tried to make edge between non-existant nodes')
        else:
            if node2 in self.nodes:
                pass
            else:
                self.nodes[node1].append([node2,rand])
                self.addLowest([node2,rand])
    def removeEdge(self,node1,node2):
        node1=tuple(node1)
        node2=tuple(node2)
        for i in range(len(self.nodes[node1])):
            if self.nodes[node1][i][0]==node2:

                del self.nodes[node1][i]
                return None
            else:
                pass
    def addNode(self,node):
        node = tuple(node)
        if self.nodes.get(node,False):
            print('hm')
        else:
            unbound = False
            self.nodes[node]=[]
            for j in range(len(self.lowestEdges)-1,-1,-1):
                if self.lowestEdges[j][0]==node:
                    del self.lowestEdges[j]
            for i in range(self.dimension):
                for j in [-1,1]:
                    neighbour = list(node)
                    neighbour[i]+=j
                    neighbour = tuple(neighbour)
                    if neighbour in self.nodes:
                        self.removeEdge(neighbour,node)
                        if self.nodes[neighbour]==[]:
                            self.onedge.remove(neighbour)
                    else:
                        self.addEdge(node,neighbour)
                        unbound = True
            if unbound:
                self.onedge.add(node)



main = graph(c=3)
main.addNode([0,0])
rands = []
count = 1
while count < 14:

    if main.lowestEdges != []:
        count+=1
        newNode = list(main.lowestEdges)[0]
        rands.append(newNode[1])
        main.addNode(newNode[0])
    else:
        for node in main.onedge:
            for edge in main.nodes[node]:
                main.addLowest(edge, filling=True)
fig = plt.figure(figsize = (10,10))
axis = fig.add_subplot(1,1,1)
plt.plot([],[], 'o', color='#b80802', label='invaded nodes')
plt.plot([],[], 'o', color='#f7fa69 ', label='stored \'highest potential\' nodes')
plt.plot([],[], color = '#f5a958 ', label='potential edges')
for i in main.nodes:

    for j in main.nodes[i]:
        plt.plot(np.array([i[0], j[0][0]]), np.array([i[1], j[0][1]]), color =
        '#f5a958 ')
        plt.text((i[0]+j[0][0])/2 -0.3,(i[1]+j[0][1])/2, str(j[1])[:4])
    plt.plot(i[0], i[1], 'o', color='#b80802')
for i in main.lowestEdges:
    plt.plot(i[0][0], i[0][1], 'o', color='#f7fa69 ')
plt.xlim(-4,4)
plt.ylim(-4,4)
plt.gca().set_aspect('equal', adjustable='box')
ticks = np.linspace(-3,3,1)
plt.grid()
plt.legend()
plt.show()
```

9.3.2 Boxplot

```
import numpy as np
import time
import matplotlib.pyplot as plt
np.random.seed(1)
```

```
class graph:
    def __init__(self,nodes=None,dimension = 2,lowestEdges=[],c=20):
        if nodes == None:
            nodes = {
        }
        self.nodes = nodes
        self.onedge = set()
        self.dimension = dimension
        self.lowestEdges=lowestEdges
        self.c=c
    def addLowest(self,newNode,filling = False):
        if self.lowestEdges == []:
            if filling:
                self.lowestEdges = [newNode]
                return None
            else:
                return None
        newRand = newNode[1]
        length = len(self.lowestEdges)
        if length ==self.c:
            filling = False
        if newRand > self.lowestEdges[-1][1]:
            if not filling:
                pass
            else:
                self.lowestEdges= self.lowestEdges+[newNode]
                return None
        else:
            for i in range(length-2,-1,-1):
                if newRand >self.lowestEdges[i][1]:
                    self.lowestEdges = self.lowestEdges[0:i+1]+[newNode]+self.lowes
                    1:self.c-1]
                    return None
            self.lowestEdges=[newNode]+self.lowestEdges[0:self.c-1]
            return None
    def addEdge(self,node1,node2):
        node1=tuple(node1)
        node2=tuple(node2)
        rand = np.random.random()
        if node1 not in self.nodes:
            if node2 in self.nodes:
                self.nodes[node2].append([node1,rand])
                self.addLowest([node1,rand],c)

            else:
                print('tried to make edge between non-existant nodes')
        else:
            if node2 in self.nodes:
                pass
            else:
                self.nodes[node1].append([node2,rand])
                self.addLowest([node2,rand])
    def removeEdge(self,node1,node2):
        node1=tuple(node1)
        node2=tuple(node2)
        for i in range(len(self.nodes[node1])):
            if self.nodes[node1][i][0]==node2:

                del self.nodes[node1][i]
                return None
            else:
                pass
    def addNode(self,node):
        node = tuple(node)
        if self.nodes.get(node,False):
            print('hm')
        else:
            unbound = False
            self.nodes[node]=[]
            for j in range(len(self.lowestEdges)-1,-1,-1):
                if self.lowestEdges[j][0]==node:
                    del self.lowestEdges[j]
            for i in range(self.dimension):
                for j in [-1,1]:
                    neighbour = list(node)
                    neighbour[i]+=j
                    neighbour = tuple(neighbour)
                    if neighbour in self.nodes:
                        self.removeEdge(neighbour,node)
                        if self.nodes[neighbour]==[]:
                            self.onedge.remove(neighbour)
                    else:
                        self.addEdge(node,neighbour)
                        unbound = True
            if unbound:
                self.onedge.add(node)




def runTest(nodes):
    main = graph(c=100)
    main.addNode([0,0])
    rands = []
    count = 1
    checkcount=0
    while count < nodes:
        if main.lowestEdges != []:
            count+=1
            newNode = list(main.lowestEdges)[0]
            rands.append(newNode[1])
            main.addNode(newNode[0])
        else:
            checkcount+=1
            for node in main.onedge:
                for edge in main.nodes[node]:
                    main.addLowest(edge, filling=True)
    return checkcount
xaxis=[]
ylist=[]
for i in range(1,16):
    xaxis.append(i*10000)
    yaxis=[]
    for j in range(12):
        start = time.time()
        print(runTest(i*10000))
        end = time.time()
        yaxis.append(end-start)
    ylist.append(list(yaxis))
fig, ax = plt.subplots()
ax.boxplot(ylist)
ax.set_xticklabels(xaxis)
ax.set_xlabel('Number of vertices invaded (1000s)')
ax.set_ylabel('Time (s)')
plt.show()
```

9.4 Question 7

9.4.1 Table values

```
import numpy as np
import time
start = time.time()
class graph:
    def __init__(self,nodes=None,dimension = 2,lowestEdges=[],c=20):
        if nodes == None:
            nodes = {
        }
        self.nodes = nodes
        self.onedge = set()
        self.dimension = dimension
        self.lowestEdges=lowestEdges
        self.c=c
    def addLowest(self,newNode,filling = False):
        if self.lowestEdges == []:
            if filling:
                self.lowestEdges = [newNode]
                return None
            else:
                return None
        newRand = newNode[1]
        length = len(self.lowestEdges)
        if length ==self.c:
            filling = False
        if newRand > self.lowestEdges[-1][1]:
            if not filling:
                pass
            else:
                self.lowestEdges= self.lowestEdges+[newNode]
                return None
        else:
            for i in range(length-2,-1,-1):
                if newRand >self.lowestEdges[i][1]:
                    self.lowestEdges = self.lowestEdges[0:i+1]+[newNode]+self.lowes
                    return None
            self.lowestEdges=[newNode]+self.lowestEdges[0:self.c-1]
            return None
    def addEdge(self,node1,node2):
        node1=tuple(node1)
        node2=tuple(node2)
        rand = np.random.random()
        if node1 not in self.nodes:
            if node2 in self.nodes:
                self.nodes[node2].append([node1,rand])
                self.addLowest([node1,rand],c)

            else:
                print('tried to make edge between non-existant nodes')
        else:
            if node2 in self.nodes:
                pass
            else:
                self.nodes[node1].append([node2,rand])
                self.addLowest([node2,rand])
    def removeEdge(self,node1,node2):
        node1=tuple(node1)
        node2=tuple(node2)
        for i in range(len(self.nodes[node1])):
            if self.nodes[node1][i][0]==node2:

                del self.nodes[node1][i]
                return None
            else:
                pass
    def addNode(self,node):
        node = tuple(node)
        if self.nodes.get(node,False):
            print('hm')
        else:
            unbound = False
            self.nodes[node]=[]
            for j in range(len(self.lowestEdges)-1,-1,-1):
                if self.lowestEdges[j][0]==node:
                    del self.lowestEdges[j]
            for i in range(self.dimension):
                for j in [-1,1]:
                    neighbour = list(node)
                    neighbour[i]+=j
                    neighbour = tuple(neighbour)
                    if neighbour in self.nodes:
                        self.removeEdge(neighbour,node)
                        if self.nodes[neighbour]==[]:
                            self.onedge.remove(neighbour)
                    else:
                        self.addEdge(node,neighbour)
                        unbound = True
            if unbound:
                self.onedge.add(node)




results = [[],[],[]]
for k in range(3):
    for j in range(10):
        main = graph(c=100,dimension =k+3)
        main.addNode([0]*(k+3))
        rands = []
        for i in range(100000):
            if main.lowestEdges != []:
                newNode = list(main.lowestEdges)[0]
                rands.append(newNode[1])
                main.addNode(newNode[0])


            else:


                for node in main.onedge:
                    for edge in main.nodes[node]:
                        main.addLowest(edge,filling=True)
        results[k].append(max(rands[5000:]))
for i in range(10):
    line = str(i+1)+'&'
    for j in range(3):
        if j==2:
            line = line+str(results[j][i])+'\\'
        else:
            line = line+str(results[j][i])+'&'
    print(line)
for i in range(3):
    print(sum(results[i])/10)
```

9.4.2 Graph

```
import numpy as np
import matplotlib.pyplot as plt
import time
np.random.seed(1)
start = time.time()
class graph:
    def __init__(self,nodes=None,dimension = 2,lowestEdges=[],c=20):
        if nodes == None:
            nodes = {
        }
        self.nodes = nodes
        self.onedge = set()
        self.dimension = dimension
        self.lowestEdges=lowestEdges
        self.c=c
    def addLowest(self,newNode,filling = False):
        if self.lowestEdges == []:
            if filling:
                self.lowestEdges = [newNode]
                return None
            else:
                return None
        newRand = newNode[1]
        length = len(self.lowestEdges)
        if length ==self.c:
            filling = False
        if newRand > self.lowestEdges[-1][1]:
            if not filling:
                pass
            else:
                self.lowestEdges= self.lowestEdges+[newNode]
                return None
        else:
            for i in range(length-2,-1,-1):
                if newRand >self.lowestEdges[i][1]:
                    self.lowestEdges = self.lowestEdges[0:i+1]+[newNode]+self.lowes
                    return None
            self.lowestEdges=[newNode]+self.lowestEdges[0:self.c-1]
            return None
    def addEdge(self,node1,node2):
        node1=tuple(node1)
        node2=tuple(node2)
        rand = np.random.random()
        if node1 not in self.nodes:
            if node2 in self.nodes:
                self.nodes[node2].append([node1,rand])
                self.addLowest([node1,rand],c)

            else:
                print('tried to make edge between non-existant nodes')
        else:
            if node2 in self.nodes:
                pass
            else:
                self.nodes[node1].append([node2,rand])
                self.addLowest([node2,rand])
    def removeEdge(self,node1,node2):
        node1=tuple(node1)
        node2=tuple(node2)
        for i in range(len(self.nodes[node1])):
            if self.nodes[node1][i][0]==node2:

                del self.nodes[node1][i]
                return None
            else:
                pass
    def addNode(self,node):
        node = tuple(node)
        if self.nodes.get(node,False):
            print('hm')
        else:
            unbound = False
            self.nodes[node]=[]
            for j in range(len(self.lowestEdges)-1,-1,-1):
                if self.lowestEdges[j][0]==node:
                    del self.lowestEdges[j]
            for i in range(self.dimension):
                for j in [-1,1]:
                    neighbour = list(node)
                    neighbour[i]+=j
                    neighbour = tuple(neighbour)
                    if neighbour in self.nodes:
                        self.removeEdge(neighbour,node)
                        if self.nodes[neighbour]==[]:
                            self.onedge.remove(neighbour)
                    else:
                        self.addEdge(node,neighbour)
                        unbound = True
            if unbound:
                self.onedge.add(node)



main = graph(c=100)
main.addNode([0,0])
rands = []

checkcount=0

time1=time.time()
i=0
while i < 100000:
    if main.lowestEdges != []:
        i+=1
        newNode = list(main.lowestEdges)[0]
        rands.append(newNode[1])
        main.addNode(newNode[0])



    else:

        checkcount+=1
        for node in main.onedge:
            for edge in main.nodes[node]:
                main.addLowest(edge,filling=True)
end=time.time()
xaxis1= np.linspace(0,100,100000)
```

```
xaxis2=np.linspace(0,100,1000)
xaxis3=np.linspace(0,100,100)
avgrand = np.zeros(100)
suprand= np.zeros(100)
for i in range(100):
    avgrand[i]=np.average(rands[i*1000:(i+1)*1000-1])
    suprand[i]=np.max(rands[i*1000:(i+1)*1000-1])
plt.plot(xaxis3,avgrand, label = 'average value of Ue')
plt.plot(xaxis3,suprand, label = 'maximal value of Ue')
print(checkcount)
plt.plot(xaxis1,rands,linewidth=0.1,alpha=0.1,label='all values of Ue')
plt.xlabel('Number of vertices invaded (1000s)')
plt.legend(loc='lower right')
plt.show()
print(end-start)
```

9.5 Question 8

```
import numpy as np
import matplotlib.pyplot as plt
import time
np.random.seed(1)
start = time.time()
class graph:
    def __init__(self,nodes=None,dimension = 2,lowestEdges=[],c=20):
        if nodes == None:
            nodes = {
        }
        self.nodes = nodes
        self.onedge = set()
        self.dimension = dimension
        self.lowestEdges=lowestEdges
        self.c=c
    def addLowest(self,newNode,filling = False):
        if self.lowestEdges == []:
            if filling:
                self.lowestEdges = [newNode]
                return None
            else:
                return None
        newRand = newNode[1]
        length = len(self.lowestEdges)
        if length ==self.c:
            filling = False
        if newRand > self.lowestEdges[-1][1]:
            if not filling:
                pass
            else:
                self.lowestEdges= self.lowestEdges+[newNode]
                return None
        else:
            for i in range(length-2,-1,-1):
                if newRand >self.lowestEdges[i][1]:
```

```
                self.lowestEdges = self.lowestEdges[0:i+1]+[newNode]+self.lowes
                return None
        self.lowestEdges=[newNode]+self.lowestEdges[0:self.c-1]
        return None
    def addEdge(self,node1,node2):
        node1=tuple(node1)
        node2=tuple(node2)
        rand = np.random.random()
        if node1 not in self.nodes:
            if node2 in self.nodes:
                self.nodes[node2].append([node1,rand])
                self.addLowest([node1,rand],c)

            else:
                print('tried to make edge between non-existant nodes')
        else:
            if node2 in self.nodes:
                pass
            else:
                self.nodes[node1].append([node2,rand])
                self.addLowest([node2,rand])
    def removeEdge(self,node1,node2):
        node1=tuple(node1)
        node2=tuple(node2)
        for i in range(len(self.nodes[node1])):
            if self.nodes[node1][i][0]==node2:

                del self.nodes[node1][i]
                return None
            else:
                pass
    def addNode(self,node):
        node = tuple(node)
        if self.nodes.get(node,False):
            print('hm')
        else:
            unbound = False
            self.nodes[node]=[]
            for j in range(len(self.lowestEdges)-1,-1,-1):
                if self.lowestEdges[j][0]==node:
                    del self.lowestEdges[j]
            for i in range(self.dimension):
                for j in [-1,1]:
                    neighbour = list(node)
                    neighbour[i]+=j
                    neighbour = tuple(neighbour)
                    if neighbour in self.nodes:
                        self.removeEdge(neighbour,node)
                        if self.nodes[neighbour]==[]:
                            self.onedge.remove(neighbour)
                    else:
                        self.addEdge(node,neighbour)
                        unbound = True
                if unbound:
                    self.onedge.add(node)




def invade(n):
    main = graph(c=100)
    main.addNode([0,0])
    rands = []

    checkcount=0
    i=0
    while i < n:
        if main.lowestEdges != []:
            i+=1
            newNode = list(main.lowestEdges)[0]
            rands.append(newNode[1])
            main.addNode(newNode[0])


        else:

            checkcount+=1
            for node in main.onedge:
                for edge in main.nodes[node]:
                    main.addLowest(edge, filling=True)
    return rands
results = []
hd = 1000
for i in range(2000):
    print(i)
    rands = invade(10000)
    sup = np.amax(rands)
    template = np.zeros(hd)
    for j in range(hd):
        if j/hd<sup:
            pass
        else:
            template[j]=1
    results.append(np.array(template))
results = np.array(results)
yaxis = np.average(results, axis=0)
xaxis = np.linspace(0,1,len(yaxis))
plt.xticks(ticks = np.array([0,0.25,0.5,0.75,1]))
plt.xlabel('p')
plt.ylabel(r'$\theta_p$')
plt.grid()
plt.plot(xaxis,yaxis)
plt.show()
```