Phase Transition in Graph Properties

Brito Rodríguez, Juan Sebastián
juan.sebastian.brito@est.fib.upc.edu
Ramos Domínguez, David
david.ramos.dominguez@est.fib.upc.edu
Torres Cirina, Daniel
daniel.torres@est.fib.upc.edu

G03

Spring 2019

Contents

1. Introduction

During this project we will, on one hand, carry out an experimental study about the phase transition for different graph properties over random graphs, and graphs which its edges and/or nodes may fail. On the other hand, we will analyze and present the ins and outs of different random graph models that are usually used in experimental algorithm studies on graph problems.

Modeling permanent failures

Permanent failures in a graph can be modeled in different ways. Initially, we will assume that it corresponds to a simple random percolation process by nodes (a.k.a. site percolation) or by edges (a.k.a. bond percolation). The percolation model on a graph *G* is defined by a parameter $q \in [0,1]$ that represents the probability of failure and provides us with a new graph G_q. In a percolation process by nodes on a graph *G* the parameter *q* represents the node's probability of failure. This way we will obtain a graph G_q where for each node $u \in V(G)$, independently, we will decide if the node *u* continues in the graph (with probability $1 - q$ of not failing) or if we will erase *u* (with probability *q* of failing). In a percolation process by edges the parameter represents an edge's probability of failing. In this case, for each edge $e \in E(G)$, we will keep *e* with probability $1 - q$ and we will take it out with a probability *q*.

Note: In the percolation process by edges we will have $V(G_q) = V(G)$.

Once we have established a percolation process on a graph *G*, we will look for which values of *q* we may expect that a certain property Π is given on the graph G_q. For many properties we can find a *q*'s threshold value in a way that, with high probability, the graphs G_q with $q > q_{\Pi}$ verify the property Π. But the graphs G_q with $q < q_{\Pi}$ with high probability do not verify Π. When this kind of behaviour is present we will say that the property Π presents a phase transition around q_{Π}.

2. Experimental Analysis

In this project we are asked to experimentally analyze the existence or non-existence of phase transition in percolation processes in graphs and in random graphs for various properties.

To do so we will divide said analysis in multiple sections (as required). Thus making it easier to understand and develop.

2.1. Graphs corresponding to percolation processes by nodes and edges.

We were given the following statement:

> *(a) Donats G i q ∈ [0,1] obtenir els grafs corresponents al procés de percolació de nodes i d'arestes.*

In this section we are going to apply site percolation and bond percolation to a complete undirected graph in order to see the resulting graph with different values of *p*. This graph will have *n* nodes and *m* edges. Since the graph is complete, $m = n(n - 1)/2$.

In order to do the percolation, we have the graph represented as an adjacency matrix. In the case of site percolation, for every node of the graph we will check if it percolates generating a random number and comparing it with *q*, and if it percolates we will set to zero the whole row and column of that node in the adjacency matrix.
In the case of bond percolation, since the graph is undirected, the adjacency matrix is symmetric, so we will only check if the edges stored in the upper diagonal half of the matrix percolate and update the lower half of the matrix at the same time.

Initially, we thought about using adjacency lists in order to use less space to store the graphs, however, we realized that for every edge that we remove from the adjacency list of one node we would have to go through the whole adjacency list of the other node to find it and remove it as well, while with a matrix we would only have to go to two locations in the matrix. Also, if we wanted to remove a node, we would have to go through all the adjacency lists of the nodes in its adjacency list, instead of just visiting a row and a column in the matrix. Therefore, for percolation using an adjacency matrix has a lower time cost and it is also easier to implement.

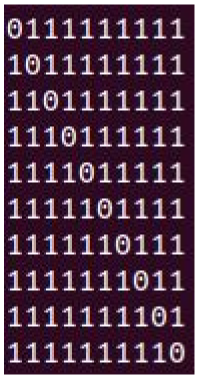
Asymptotic complexity

In both cases, there will be an initial time cost of *$O(n^2)$* to build the original graph adjacency matrix. For site percolation we only check the *n* nodes to see if they percolate, but for every node that percolates we have to set its whole row and column of the matrix to zero. Therefore, in the best case where no node percolates we will have cost *$\Theta(n)$*, but in the worst case where every node percolates we will have cost *$\Theta(n^3)$*, although can be reduced to *$\Theta(n^2)$* by replacing the row with a pre-initialized vector.

For bond percolation we only check the *$n(n - 1)/2$* edges, but for every edge that percolates we have to update two positions of the matrix. Therefore, in the best case where no node percolates we will have a cost of *$\Theta(n(n - 1)/2) = \Theta(n^2/2)$*, but in the worst case where every node percolates we will have a cost of *$\Theta(n(n - 1)/2 \times 2) = \Theta(n^2)$*.

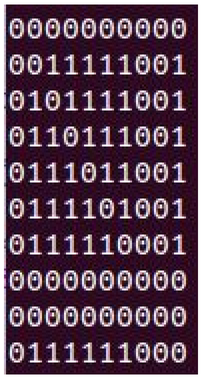
The space cost of both percolations is *$O(n^2)$*, since we will use the same matrix to store the initial adjacency matrix and the one generated after the percolation.

Examples of adjacency matrices of graphs before and after percolation:



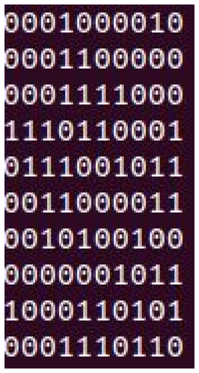
```
0111111111
1011111111
1101111111
1110111111
1111011111
1111101111
1111110111
1111111011
1111111101
1111111110
```

(A)



```
0000000000
0011111001
0101111001
0110111001
0111011001
0111101001
0111110001
0000000000
0000000000
0111111000
```

(B)



```
0001000010
0001100000
0001111000
1110110001
0111001011
0011000011
0010100100
0000001011
1000110101
0001110110
```

(C)

- **Figure 1**. (A) Initial graph with |V| = 10 and fully connected, (B) after applying site percolation with q=0.5 and (C) after applying bond percolation over (A) with q=0.5.

2.2. Phase transition on n ✕ n square grids.

We were given the following statement:

> *(b) Estudiar la transicio de fase a graelles quadrades n x n, sota un proces de percolació de nodes i un d'arestes, de la propietat estudiada a https://algs4. cs.princeton.edu/lectures/15UnionFind-2x2.pdf.*

Implementation

In order to optimize time and memory, our grid implementation will be significantly different than our graph implementation. A grid will be represented as a matrix of nodes, where each node will have 4 class members:

- ➢ ***bool*** *enabled*: True if the node is enabled, otherwise false.
- ➢ ***bool*** *right*: True if the node to the right can be accessed from this node, false otherwise.
- ➢ ***bool*** *bottom*: True if the node below can be accessed from this node, otherwise false.
- ➢ ***pair<int, int>*** *parent*: XY coordinates of the parent. Used in the Union Find algorithm.

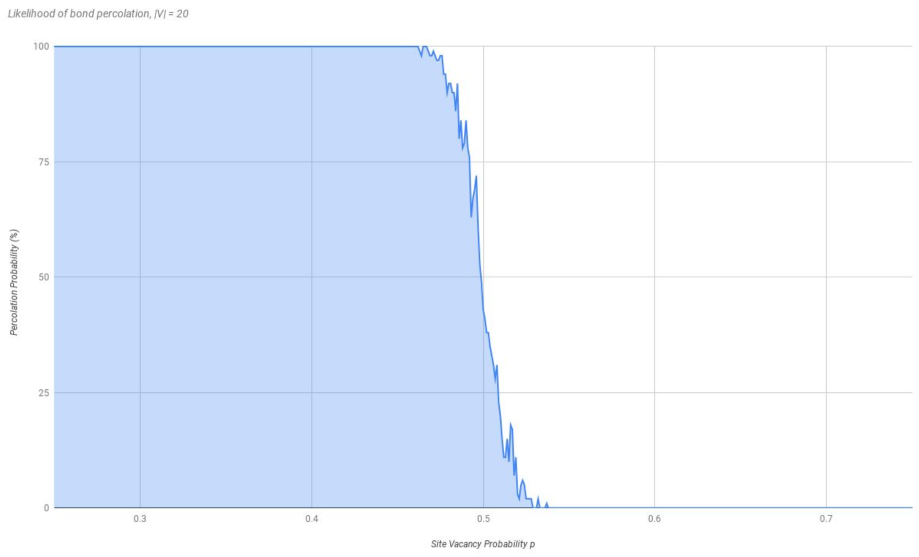
Asymptotic complexity

Percolating a node or an edge only means changing one value in each case, so the time complexity would be $O(n^2)$. But we need to synchronize our Union-Find structure, in the worst case we will have to call $2 \times n^2$ times the *unionUF* function. Each *unionUF* execution will call two times the *findUF* function, with complexity $O(n^2)$, so the total time complexity will be $O(n^2 + 2n^2 \times 2n^2) = O(n^4)$.[1]

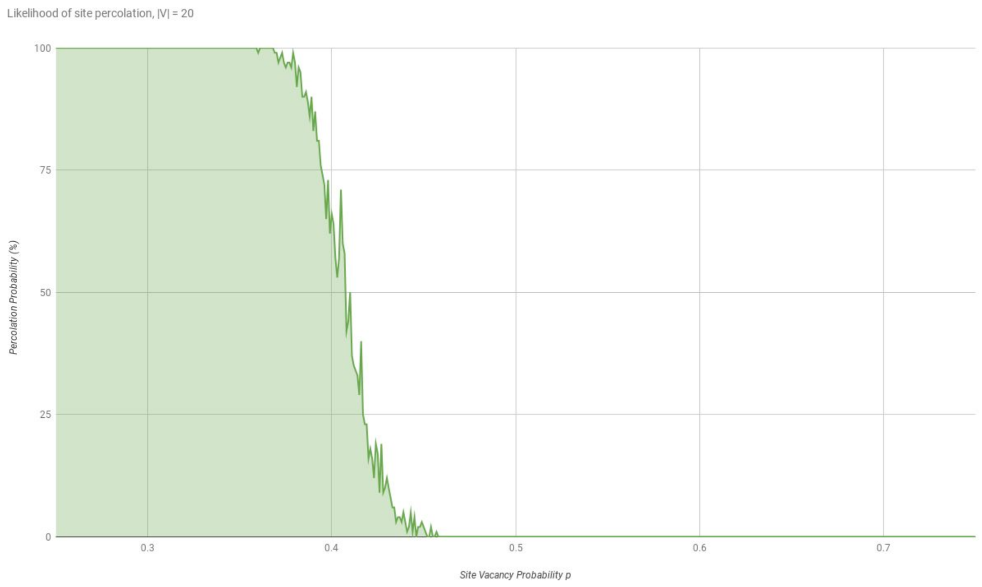
Finally, we need to check if we can get from any of the nodes in the first row to any of the nodes in the last row. In the worst case, we will need to find the parent of the *n* nodes in the first row and the *n* nodes in the last row and check to see if they share any value. Time complexity will be $O(2n^3 + n^2) = O(n^3)$.

Results

After reading the *PDF* provided in the statement (Algorithms by Robert Sedgewick, Kevin Wayne) we decided to apply the percolation process we had programed to a grid of size *n = 20*. For that, we obtained the following results.



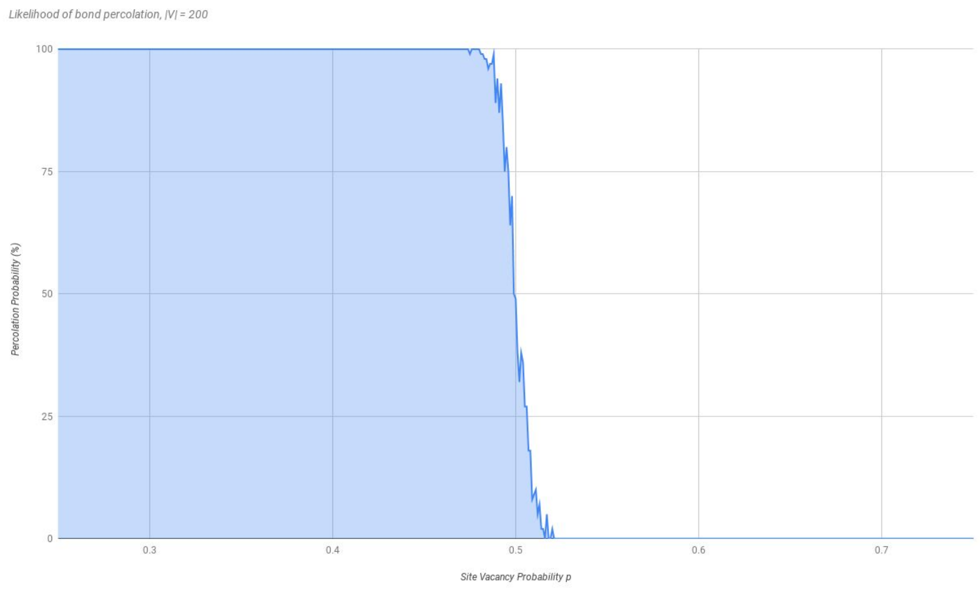
- **Figure 2**. Likelihood of bond percolation, $q \in [0.25,0.75]$, $n = 20$.



- **Figure 3**. Likelihood of site percolation, $q \in [0.25,0.75]$, $n = 20$.

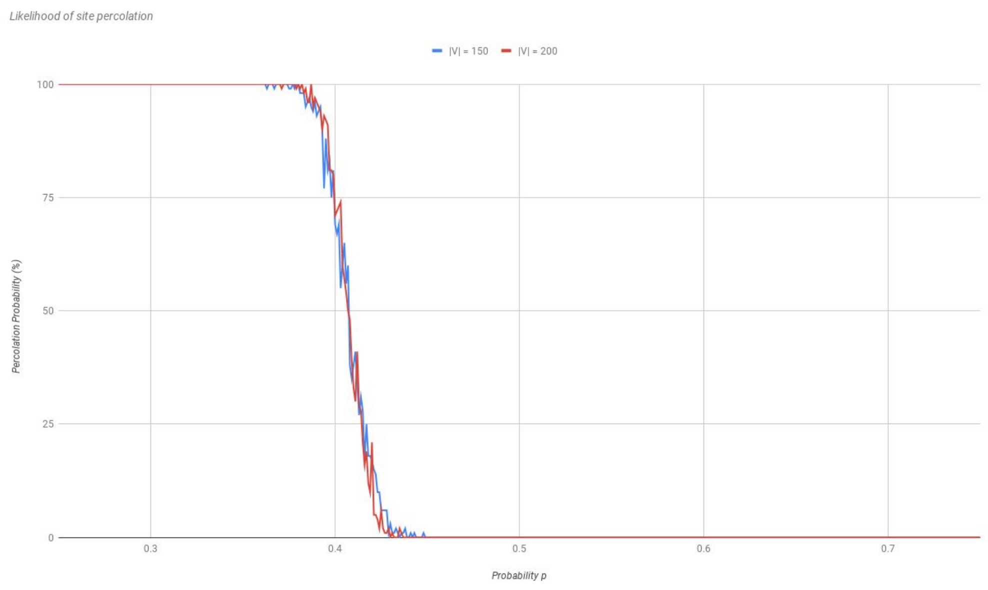
After seeing the resulting curve in the graph and consulting with teacher Maria Jose Serna Iglesias, we learned that the initial choice of the board size was wrong. Said curve shouldn't have a discernible slope.

It is indeed stated that in the process to check whether an N-by-N system percolates it's dependent on two parameters; the site vacancy probability *q* (which we already vary the range) and the board size. So now we will augment the size of the board until a better result is obtained.



- **Figure 4**. Likelihood of bond percolation, $q \in [0.25,0.75]$, $n = 200$.

Now we can observe that the amplitude of the curve has decreased and the percolation threshold q_{Π} is 0.499 for a percolation probability (%) of 50. We have stopped at $n = 200$ because the running time for it already exceeded the 30 minutes.
With a percolation process established and a property Π defined as "*System percolates if top and bottom are connected by open sites.*" on a graph G_q. We have found the threshold value in a way that, with high probability, the graphs G_q with $q > q_{\Pi}$ verify the property Π. But the graphs G_q with $q < q_{\Pi}$ with high probability do not verify Π. This kind of behaviour confirms that the property Π presents a phase transition around q_{Π}.



- **Figure 5**. Likelihood of site percolation, *q ∈ [0.25,0.75], n = 150 & 200*.

Here we can see that although there isn't much difference between *n = 150* and *n = 200* the amplitude of the curve has decreased compared to the other site percolation shown in **Figure 3**. Thus, with more accuracy we can establish that the percolation threshold q_{π} is 0.407 for a percolation probability (%) of 50 for |V| = 200.

Execution time

We have measured the execution time and CPU ticks of the method we followed to calculate the phase transition: Incrementing from 0 to 1 by 0.001 the probability *q* and checking 100 times per value if it percolates or not for a grid of 100x100. This makes a total of 10.000 scenarios.

	CPU ticks	Time (in seconds)
Bond Percolation	215.369	215,4
Site Percolation	169.826	169,8

Execution time may differ from one system to another.

2.3 Phase transition on random graphs.

We were given the following statement:

> *(c) Estudiar la transició de fase a grafs aleatoris (segons diferents models) d'una versio adient de la propietat estudiada a (b).*

To do so, first we needed to obtain said random graphs. One choice we had was to use the already implemented generators given in NetworkX. Multiple choices were presented and we decided to proceed with the Barabási–Albert model [2]. Later on, after consulting with professor Josep Diaz Cort, we were told to avoid said model and focus on a simpler one. Given professor Diaz's CV, we decided to follow his advice and choose to generate random graphs $G_{n,p}$ also known as an Erdős-Rényi model or binomial graphs.

This model builds a random graph by generating *n* disconnected nodes, and then for every possible edge between these nodes it uses the probability p to see if it's added to the graph. It is similar to the bond percolation, but instead of starting with a graph with edges and then removing them with probability *q*, it starts with a graph with no edges and adds them with probability p.

In this model each edge has a fixed probability of being in the graph, so each edge is equally likely to appear in the graph and they all are independent from each other. This makes it easier to analyze the graph, however, in most real networks the edges influence each other (for example: in a social network, if two people have the same friend it's likely that they are also friends). Therefore, this model is mostly used to study properties in graphs. [3]

Since there are no 'rows', 'top nor 'bottom' 'in a random graph, we can't use the same property as in the previous section, but we can adapt. We will see if the graph is connected. Meaning, $\forall\, u \in V\ \exists\, w \in V \mid$ *there is a path between (u,w)*.

Asymptotic complexity

In order to know if the graph is connected we use a BFS, starting from the first node in the adjacency matrix and then checking which nodes it is connected to and storing them in a queue. Then, for every node in the queue we will also go through its adjacent nodes, and if they haven't been visited we add them to the queue.

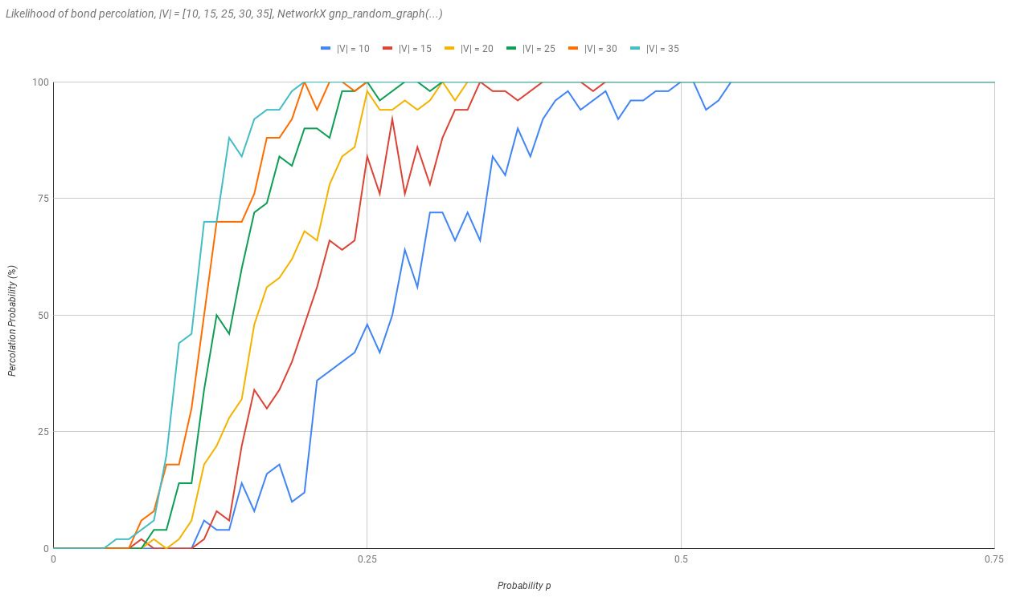
In order to select the initial node we check every position of the adjacency matrix until we find an adjacency, so in the worst case we will visit the whole matrix, with complexity $O(n^2)$.

Since the queue will have at most $(n - 1)$ nodes (if the initial node is adjacent to all others) and for every node we have to check its whole row of the adjacency matrix, the cost of using the queue is $O(n^2)$. Checking every node to see if it has been visited has cost $O(n)$. Therefore, our BFS has cost $O(n^2+n^2+n) = O(n^2)$.

Once we have defined the proceedings to apply, we generate the graphs. We do it through the function *gnp_random_graph(n, p, seed=None, directed=False).* This algorithm runs in $O(n^2)$ time. [4]

At this point we were developing this section and the section ***2.4. Other graphs properties' phase transition on random graphs*** at the same time. We had encounter some setbacks due to the exponential cost of execution the code from said section in graphs with more than 15 nodes. For that reason we decided to begin the experiments with "small" graphs.

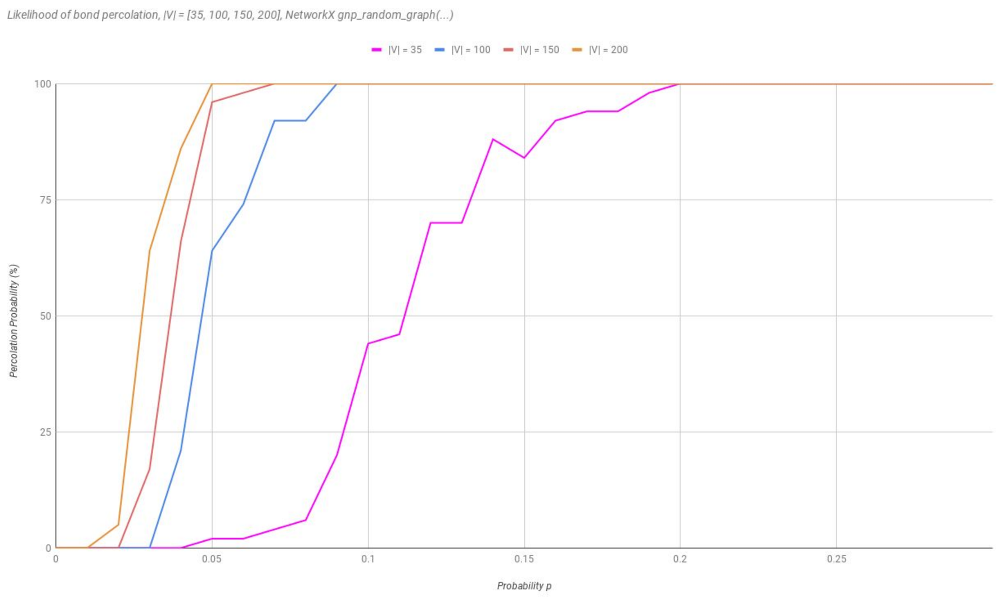
Results



- **Figure 6**. Likelihood of bond percolation on a random graph, $p \in [0.00,0.75]$.

Seeing the results obtained and after consulting with professor Maria Jose Serna Iglesias we again had to reduce the amplitude of the curve. Otherwise, we could not correctly analyze the phase transition. Instead of increasing by 5 nodes each time, we started from 100 and increased by 50 instead.

Note: For this executions, to get the *Percolation Probability (%)*, from the beginning we decided to generate 100 graphs with the same value of *p* and see if it percolates to later on make the median. This parameter is described as *repetition* inside the code in Python. But for this part we first started by only doing 1 repetition for each *p* just in case it could take a very long time to execute. After some testing, we decided on limiting the value of |*V*| to a maximum of 200. Otherwise, if anybody else were to test it, it would take up to half an hour to generate and verify the percolation process for the biggest graph size.

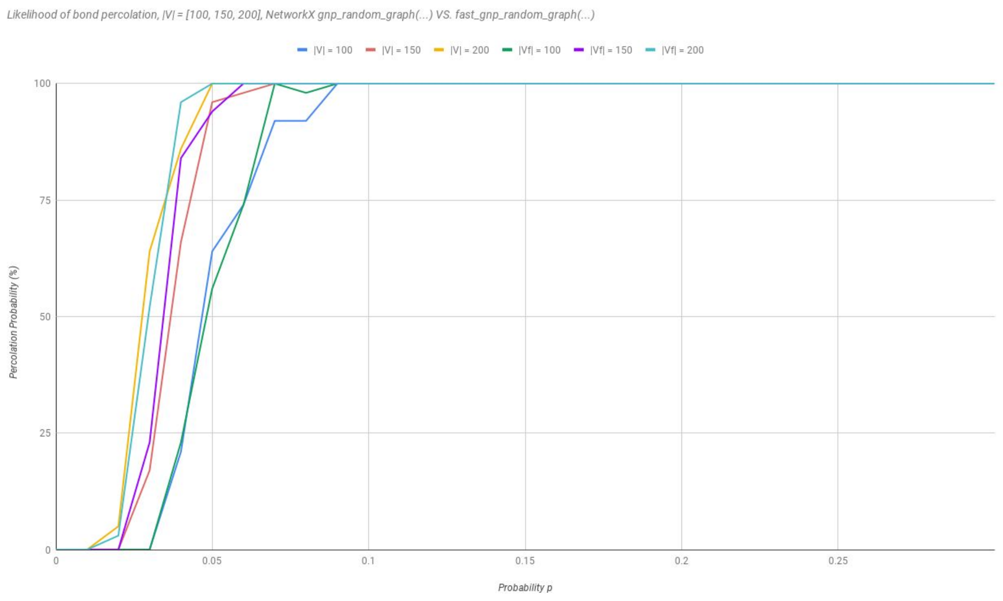


- **Figure 7**. Likelihood of bond percolation on a random graph, $p \in [0.00, 0.30]$.

Here we can see a comparison among the new curves and the one made with the biggest graph from **Figure 6** with |*V*| = 35. It was quite surprising that given a graph with 200 nodes it would percolate with such a small value of *p*. The percolation threshold q_{Π} sits around *0.02* and *0.03* for a percolation probability (%) of 50. And tops it at *0.05 with a* percolation probability (%) of *100*.

While we were looking at the functions provided on NetworkX we noticed that for the same model there was another function available which would give us a random graph. Said function is called *fast_gnp_random_graph(n, p, seed=None, directed=False)* [5].

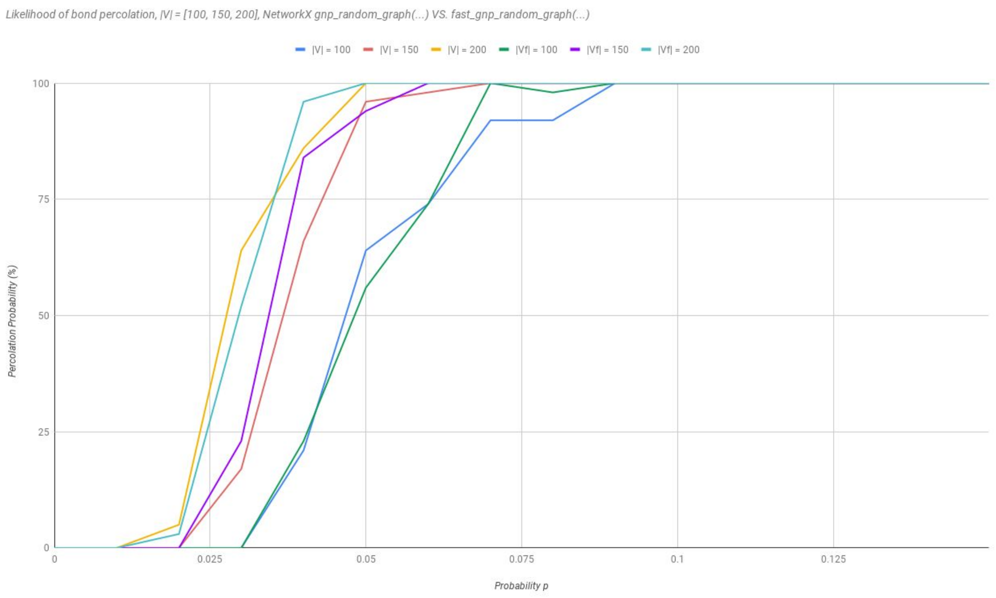
And in the description for the other function we had already used, in its description we saw an advice. “*For sparse graphs (that is, for small values of p), fast_gnp_random_graph() is a faster algorithm.*” We understood that. But, is it always valid? Even for “small” and “big” graphs? To prove so we decided to do another battery of tests. We will execute both functions, for a range of values of *p* and different number of nodes.



- **Figure 8**. Likelihood of bond percolation, *p ∈ [0.00,0.30]*.

Here we can see the result of the execution and compare both functions. |Vf| means the number of nodes on the graphs that have been executed with the fast function. In all three cases we can see that it tops the y axis (the Percolation Probability (%) with a value of 100) with a smaller value of *p* than the equivalent executed with the other function. We thought that this wouldn’t be the case since we only expected better performance of the execution of the program not better chance of percolation. But as seen in the data, it does happen.

Let’s take a closer look at the graphic to understand what happens on small values of *p*.



- **Figure 9**. Likelihood of bond percolation, *p ∈ [0.00,0.15]*.

This is the same data as in **Figure 8** but with a narrower range for *p*. The formal definition for the fast functions is described as follow:

> *"The $G_{n,p}$ graph algorithm chooses each of the [n(n−1)]/2 (undirected) or n(n−1) (directed) possible edges with probability p.*
>
> *This algorithm runs in O(n+m) time, where m is the expected number of edges, which equals pn(n−1)/2. This should be faster than gnp_random_graph() when p is small and the expected number of edges is small (that is, the graph is sparse)."*

We can see how for small values of *p* in two cases it is true that for the same graph size we can obtain a better mark for the percolation probability although this doesn't necessarily mean a better performance. For the performance we will compare it in the next section.

Execution time

We were asked to quantify the amount of work that our program does. Initially we considered adding the number of clock ticks that the processor made. But in this case it was not possible since part of the operations are done through Python and piping the results onto the C++ program. Also, we don't have the proper environment to execute and obtain objective results. Every computer has its own kernel, task scheduler… Plus other programs running in the background that cannot be quantified. That's why we conclude that the best option is to focus on the asymptotic complexity instead of the different performance on different computers with different processors. Nonetheless, we include the following measurements to comply with what has been asked. We will execute the programs for different graph sizes and varying the the value of *p*.

Graph generator	|V|	repetitions	time
gnp_random_graph	100	50	169.83
gnp_random_graph	150	50	312.78
gnp_random_graph	200	50	560.15

Graph generator	|V|	repetitions	time
fast_gnp_random_graph	100	50	165.38
fast_gnp_random_graph	150	50	304.09
fast_gnp_random_graph	200	50	594.25

- **Figure 10**. Execution time for the random graph generation and percolation process.

As we can observe on the table when the expected number of edges is small, that is, the graph is sparse we can observe that the execution time is shorter for fast_gnp_random_graph. Although not by much. But a quantifiable amount nonetheless. This means that it is true what was prefaced about performance on the function documentation.

2.4. Other graphs properties' phase transition on random graphs

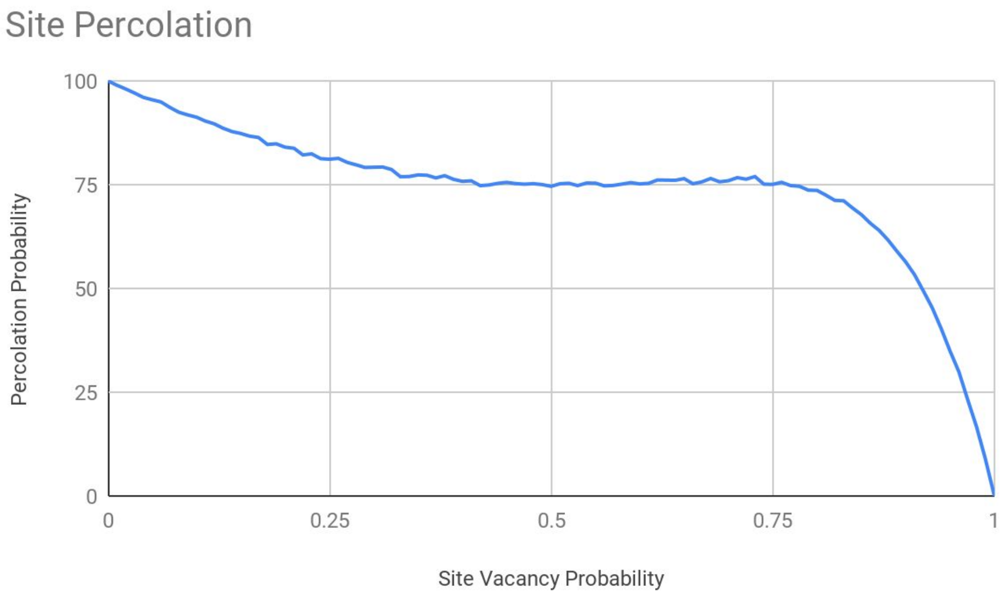
We were given the following statement:

> *(d) Estudiar la transició de fase d'altres propietats de grafs (diferents de la propietat estudiada a (b)) a grafs amb percolació i/o a grafs aleatoris.*

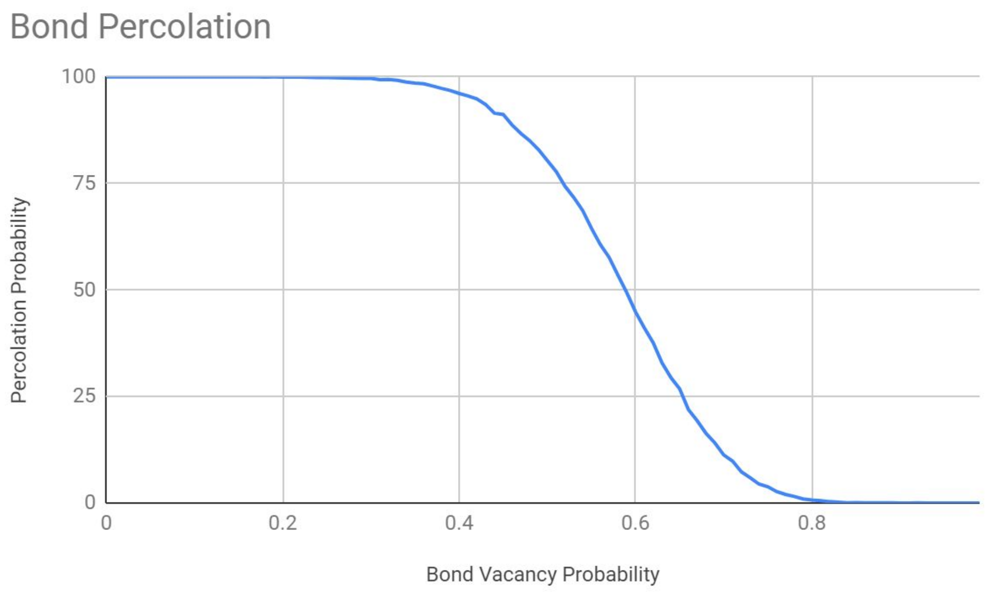
The property we chose is the existence of Hamiltonian cycles. A Hamiltonian cycle is a graph cycle that visits each node exactly once.

The same technique used in the first exercise, was applied in this one. The main difference is that the property to be satisfied has changed.

Since checking the existence of Hamiltonian cycles has a time complexity O(n!), the graph size needs to be reduced. For a graph with |V| = 10, we can achieve an acceptable execution time.



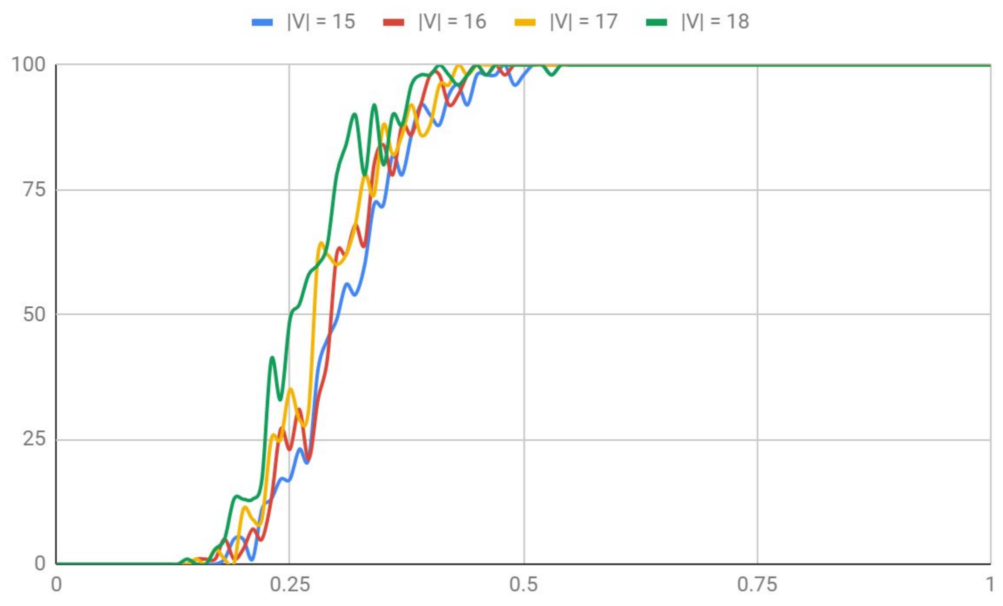
- **Figure 11**. Likelihood of site percolation, *p ∈ [0,1], n = 10*.



- **Figure 12**. Likelihood of bond percolation, *p ∈ [0,1], n = 10*.

We can observe that for the site percolation, the percolation probability is barely affected for p ∈ [0.4, 0.75], but it rapidly decreases for p > 0.8.

However, in the bond percolation chart we can see a similar fashion to the charts obtained in the exercise 1, but with a wider curve.



- **Figure 13**. Likelihood of percolation on random graphs

Execution time

We have measured the execution time and CPU ticks of a total of 100.000 escenarios: 1000 per each *p*, incrementing it from 0 to 1 by 0.001.

	CPU ticks	Time (in seconds)
Bond Percolation	4.140	4,1
Site Percolation	1.410	1,41

- **Figure 14**. Table with CPU ticks and Time from different executions.

3. Conclusions

In this project we learned about the concept of percolation and the modifications it can make to a regular graph. After that, we had to study the same process but in a grid, and in order to do that we figured how to transform the grid into a graph with the same information, and then, to check if the first row of the grid was connected to the last. We learned how to implement and use Union-Find in our graph.
We also learned some different ways of generating a random graph in order to choose one model to use, and, although we used a simple one in the project, we also researched some complex models like the Barabási–Albert model that we mentioned.

Once we decided on a model and had implemented all the necessary code we were able to obtain the necessary data to make assert properties that can be applied to all random graphs of the same model. We now know how the number of edges and its possible failure may affect the connectivity of the graph, what are the fastest ways to generate and analyze said graphs and how to properly communicate different processes between different languages. Especially noticeable is how we were wrong about what the correct graph size should be and that using a faster function also affects the percolation probability.

When we changed the property to check the existence of Hamiltonian cycles, we learned that we would have to use a different range of graph sizes to be able to study the property, because with the same sizes as before it would have taken days to get the results.

In conclusion, we learnt how percolating nodes or edges on a graph can have a different impact for different graph properties and how the graph size plays a critical role in execution time.

4. References

[1]
https://jariasf.wordpress.com/2012/04/02/disjoint-set-union-find/

[2]
http://barabasi.com/f/622.pdf

[3]
https://sarcasticresonance.wordpress.com/2017/07/09/random-graphs-the-erdos-renyi-gnp-model/

[4]
E. N. Gilbert, Random Graphs, Ann. Math. Stat., 30, 1141 (1959).

[5]
https://networkx.github.io/documentation/networkx-2.0/reference/generated/networkx.generators.random_graphs.fast_gnp_random_graph.html#networkx.generators.random_graphs.fast_gnp_random_graph