

REPORT LAB2

1.1.2. Report items

a) What tests have you conducted? (Up to what size ($|V|$ and $|E|$) have you considered?)

We have conducted two tests.

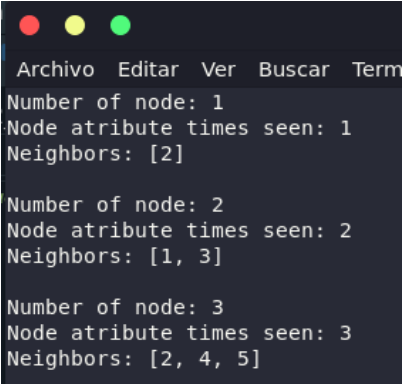
In the first test we made a graph in which the condition was fulfilled. This graph's order is $|V| = 998$ edges and the size is $|E| = 999$ nodes.

On the second test, adjacent nodes had repeated frequencies, so the condition wasn't fulfilled. So, we have one test with an expected result True and another one that is False.

This graph's order is $|V| = 998$ edges and the size is $|E| = 999$ nodes.

b) How many times do you read each node's "frequency" (as a function of $|V|$)?

To know this, we have first calculated how many times we read the attribute of the nodes after the two initial for's. To do this, instead of passing a graph where the attribute of the nodes is 'frequency' we have passed one where the attribute is a counter, all initialized to zero. Every time the node attribute is read, the counter of that node will be increased by one. In this way, we have found out how many times we read it after the for's. As seen in the image, we read it as many times as neighbours the node has.



```
Archivo  Editar  Ver  Buscar  Term
Number of node: 1
Node attribute times seen: 1
Neighbors: [2]

Number of node: 2
Node attribute times seen: 2
Neighbors: [1, 3]

Number of node: 3
Node attribute times seen: 3
Neighbors: [2, 4, 5]
```

We only need to find out how many times it is read in the if, it is easy to know. We will compare the node with all its neighbours. That is, we will read the attribute of the node as many times as it has neighbours.

So, we can say that in total we read the attribute **$2 * \text{neighbours the node has}$** .

For example, if a node has 5 neighbours, its attribute is accessed 10 times. That is, we will read 'frequency' ten times.

1.2.2. Report items

- a) What function(s) in the *itertools* package apply in this example? Have you used any of them?

We could have used four different functions in this example: *product*, *permutations*, *combinations*, *combinations_with_replacement*. All these functions belong to **combinatoric generators**.

Yes, we have to decide which of these functions we need for our exercise. In our exercise we can have repeated books (like AA), so we discard the functions *combinations* and *permutations*. We also take into account that order doesn't matter so we have decided to use the *combinations_with_replacement*, because it's the same if we have a shelf with books AB and another one with books BA, they have the same books.

- b) What tests have you conducted? (Up to what size ($|V|$ and $|E|$) have you considered?)

We have conducted three tests of (4,2), (27,6) and (6,27) books and shelf storage respectively (position of books in the shelf).

- c) Does your execution time change with B or S? Why?

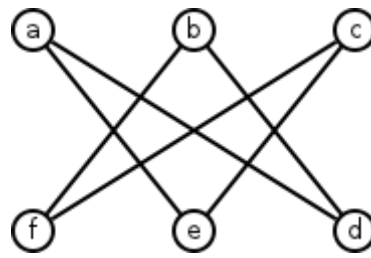
We have compared the execution time of (27,6) and (6,27), (B, S) respectively. After test, we have realized that the execution time increases as we increase B because the number of combinatorics increases faster with higher B than with higher S.

1.3.4. Report items

- a) If you used any new *itertools* or *networkx* library function, explain which, and why.

We have used new *itertools* in **contains_K5_subgraph** and **contains_K33subgraph**. In **contains_K5_subgraph** we used the function *combinations* from library *itertools* to know all the possible combinations between the 5 nodes. We have used this function to know all the possible combinations of the nodes so we can make all possible subgraph combinations.

In **contains_K33_subgraph** we also used the function *combinations* from library *itertools* to have all the possible combinations using the 6 nodes. We have used this function to know all the possible combinations of the nodes and subgraph and look if there is any K33 subgraph on all possible subgraphs. We also used the algorithms *is_bipartite* and *bipartite.sets* from the *networkx* library. First, we used *is_bipartite* because if the graph is not bipartite it can't be K33. So, we decided to put this algorithm in the first line so that if the graph given is not a bipartite graph it won't continue the code. Then, we used the *bipartite.sets* algorithm to distinguish the two sets (for example in this picture the nodes a,b,c are in the first set and d,e,f in the second one). This has been done for each subgraph. We also used it to know if there were 3 nodes in each set. If this is not fulfilled it means the subgraph is not a K33. This algorithm also allows us to see if every node is connected to all the nodes from the other set.



b) How can one combine these three functions to make a planary test? Write your pseudocode.

We can make a planar graph testing using *Kuratowski's* theorem.

Firstly, we will use the `remove_subdivisions` function to decrease the execution time of the following steps and make it easier to find K5 and K33 subgraphs.

In graph theory, Kuratowski's theorem states that a finite graph is planar if and only if it does not contain a subgraph that is a subdivision of K5 (we could use our function `contains_K5_subgraph`) or of K33 (complete bipartite graph on six vertices, three of which connect to each of the other three, we could use our function `contains_K33_subgraph`). We could use the `remove_subdivisions` function to transform a k33 or k5 subgraph into a non k33 or k5, respectively.

So we have that Kuratowski's theorem states that a finite graph G is planar, if it is not possible to subdivide the edges of K5 or K33, and then possibly add additional edges and vertices, to form a graph isomorphic to G. Equivalently, a finite graph is planar if and only if it does not contain a subgraph that is homeomorphic to K5 or K33.

Notice that if G is a graph that contains a subgraph 'H' that is a subdivision of K5 or K33, then H is known as a Kuratowski subgraph of G. With this notation, Kuratowski's theorem can be expressed as --> a graph (G) is planar if and only if it does not have a Kuratowski subgraph (H).

A Kuratowski subgraph of a nonplanar graph can be found in linear time, as measured by the size of the input graph. So, $O(n)$.

We also must consider that if the graph G is planar, then every subgraph of G is planar.

We will use the `remove_subdivisions` function to save time in executing the k3,3 and k5 functions. In these, the nodes that have degree 2 (two neighbors) are not relevant, so if we delete them before passing them, we will save time in the iterations of both functions.

Pseudocode

```
def is_planar(G):  
    remove_subdivisions (G)  
    if G is not k33 & G is not k,5,5  
        G is plannar  
  
    G is not plannar
```

c) What tests have you conducted? (Up to what size ($|V|$ and $|E|$) have you considered?)

For the `remove_subdivisions` function we made two graphs. The first graph's order is $|V| = 7$ edges and the size is $|E| = 5$ nodes and the second one's order is $|V| = 4.952$ edges and the size is $|E| = 102$ nodes

For the `contains_k5_subgraph` function we made three graphs. The first graph's order is $|V| = 10$ edges and the size is $|E| = 5$ nodes, the second one's order is $|V| = 16$ edges and the size is $|E| = 7$ nodes and for the last graph of the function, the order is $|V| = 10$ edges and the size is $|E| = 5$ nodes.

For the `contains_k33_subgraph` function we made four graphs. The first one's order is $|V| = 9$ edges and the size is $|E| = 6$ nodes, the second one's order is $|V| = 10$ edges and the size is $|E| = 7$ nodes, the third one's order is $|V| = 10$ edges and the size is $|E| = 5$ nodes and the last one's order is $|V| = 15$ edges and the size is $|E| = 6$ nodes

d) Provide a justified estimation of the complexity of your implementation of these three functions.

To estimate the complexity $O()$ we have decided to increase the input size (which is the parameter we pass to the function) and calculate the time it takes to execute it every time.

We've made a graphic where the X is the parameter and the Y is the time it took the machine to execute our programs.

Analysing the curve of the graph we can estimate the complexity of each function.

At first sight we can see that both functions are exponential, but if we take a closer look, we can see that when both functions have the same input size(x) `remove_subdivisions` take more time on the execution. So, we estimate that the `remove_subdivisions` function is more complex than the `contains_k5_subgraph` function, or at least is slower.

We couldn't make a graph of the `contains_k33_subgraph` function but we think it is also exponential and it is more complex than the other two functions because we make more comparisons and it has more FORs than the other functions.

