READ ME

In order to solve the first problem, we created a class called "Graph", that is containing the characteristic functions which are contributing to the creation of a graph with:

-function setNodeInfo:

Which gives and sets in the certain node identity and specific information.(In our case the information consists in a letter);

-function getNodeInfo:

This shows the information that is found inside the node;

-function addEdge:

Which is a specific function that adds an edge to the graph;

-function removeEdge:

This is used in order to remove an unwanted edge or it is used in to obtain a new shape/form for the desired graph;

~ First, we read from the file “date.txt” the first row that contains the letters that are stored in the nodes. Then, two by two, we read the edges using the function ‘add Edge’.

You can use the function dfs (or bfs) in order to see the display of the nodes.

~For the second part of the exercise, we use the function ‘password’ in order to see if each word introduced in the main block can be the password.

-function ‘password’:

It receives as a parameter the word that we want to check (cuv[]). To be sure that the word can be used as a parameter, we also declared a char array in which we stored the word that we formed when we checked each letter (also, x represents the number of letters of the word stored in s).

First we check if the first letter of the word cuv[] is found in one of the nodes information. If it is, then we store the index of the node in the path[] vector and we copy the letter in the backup vector s.

If the number of the elements of the backup vector s[] is still the same as in the beginning, it means that the word cuv[] can't be a password.

Secondly, we check if the neighbors of the first letter link to the next letter that we need. If we find such node, we copy the letter in the backup vector s[] and we also store in the path vector the index of the letter. We do this until the variable lit (which represents the index of the searched word that we need to find, if it exists) is equal to the number of the letters in the cuv[] word.

Finally, if the backup vector s[] is equal to the original word cuv[], it means that the word can be a password and we print the message yes, and the path vector. If not, we print the word no.

~For the third part of the exercise, we got to the conclusion that we need three colors for the nodes based on the next theory:

-if we have a complete graph, then the numbers of colors must be equal with the number of the nodes;

-if we compute the minimum and the maximum numbers of the friends of a node, the numbers of colors needed can be either nr_maxim / nr_ minim if they are equal (our case) or nr_maxim – nr_minim if they are different;

We created the functions 'complet_check', 'kinda_lista_prieteni' and 'demonstration' in order to solve the problem.

-function 'complet_check':

It returns 0 if we find a value in the adjacent matrix that is not 1 (except the principal diagonal). If there is no such thing, it return 1, meaning that we have a complete graph.

-function 'kinda_lista_prieteni':

It computes the numbers of friends each node has in a vector called 'prieteni[]'.

We use the adjacent matrix in order to see for each node how many edges with other nodes it has.

-function ‘demonstration’:

First of all, we compute the minimum and maximum numbers of friends of a node. Then we check if we have a complete graph, then, if not, we check if the maximum and minimum numbers are equal or not.

Finally, for the last part of the exercise, we drew the graph.