Modelling for Science and Engineering
Optimization
Course 2023–24

Connected graphs

The following code loads the edges of a graph from a given file that must be given as an argument. After loading the graph, it shows at the screen all the information:

```c
#include <stdio.h>
#include <stdlib.h>

typedef struct{
    unsigned nedges;
    unsigned edges[8]; // limit of edges in a node = 8
}node;

int main(int argc,char *argv[]){
    FILE * defgraph;
    node * nodelist;
    unsigned i,j;
    unsigned gsize,gorder,or,dest;

    defgraph=fopen(argv[1],"r");
    if(defgraph==NULL){
        printf("\nERROR: Data file not found.\n");
        return -1;
    }
    fscanf(defgraph,"%u",&gorder);
    fscanf(defgraph,"%u",&gsize);
    if ((nodelist = (node *) malloc(gorder*sizeof(node))) == NULL){
        fprintf(stderr, "\nERROR: no enought memory for allocating the
    nodes\n\n");
        return 2;
    }
    printf("%s defines a graph with %d nodes and %d edges.\n",argv[1],
    gorder,gsize);
    for (i=0; i<gorder; i++) nodelist[i].nedges = 0;
    for (j=0; j<gsize; j++){
        fscanf(defgraph,"%u %u",&or,&dest);
        nodelist[or].edges[nodelist[or].nedges]=dest;
        nodelist[or].nedges++;
        nodelist[dest].edges[nodelist[dest].nedges]=or;
        nodelist[dest].nedges++;
    }
    fclose(defgraph);
    for (i=0; i<gorder; i++){
        printf("Node %u has %u edges:\n",i,nodelist[i].nedges);
        for (j=0; j<nodelist[i].nedges;j++){
            printf("  %u --> %u\n",i,nodelist[i].edges[j]);
        }
    }
    return 0;
}
```

Check the program compiling and executing it. For example, if you execute it with the argument Graph1.txt the program should return:

```
Graph1.txt defines a graph with 4 nodes and 4 edges.
Node 0 has 1 edges:
  0 --> 1
Node 1 has 3 edges:
  1 --> 0
  1 --> 2
  1 --> 3
Node 2 has 2 edges:
  2 --> 1
  2 --> 3
Node 3 has 2 edges:
  3 --> 1
  3 --> 2
```

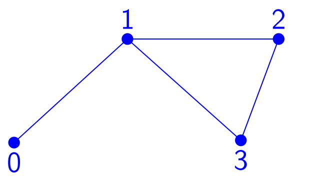
Remark: We will consider all graphs as unoriented graphs (al edges must be considered in both directions). We also consider that a *path* is a sequence of edges such that the root is the origin of the first element and the destination of each edge is the origin of the next one.

You must modify the given code to solve the next exercises. The main objective of this delivery is to program either a BFS or DFS iteration for checking the connectedness of a graph.

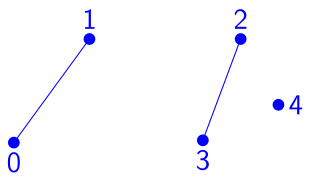
Exercices

To each code, add as a comment at the two first lines your `Name` and `NIA`.

Exercise 1: Here we have two examples of graphs: one connected, and the other one not.



`Graph1`: it is connected.

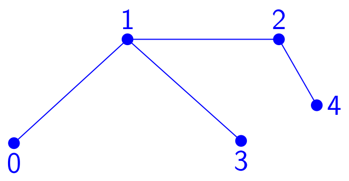


`Graph2`: it is not connected.

Copy the original code as `Ex1.c`. Modify the new code such that the code returns `1` if the graph is connected and `0` if it is not.

Exercise 2: The name of the second code must be `Ex2.c`. This code must return the number of connected components of a given graph. As example, `Graph1` has 1 connected component and `Graph2` has 3 connected components.

Exercise 3: The name of third code must be `Ex3.c`. This code must return `1` if the given graph is a tree (in particular, it must be connected) and `0` otherwise.
For example, `Graph1` and `Graph2` are not trees, while `Graph3` is a tree.



`Graph3`: it is a tree.

Final remarks

Here you have several executions of the programs we have coded, and the corresponding output (the symbol $ correspond to bash). It is important to preserve the format of the output and in the correction we will execute your programs with other graphs.

```
$ ./Ex1 Konigs.txt
1
$ ./Ex1 Graph1.txt
1
$ ./Ex1 Graph2.txt
0
$ ./Ex1 Graph3.txt
1
$ ./Ex2 Konigs.txt
1
$ ./Ex2 Graph1.txt
1
$ ./Ex2 Graph2.txt
3
$ ./Ex2 Graph3.txt
1
$ ./Ex3 Konigs.txt
0
$ ./Ex3 Graph1.txt
0
$ ./Ex3 Graph2.txt
0
$ ./Ex3 Graph3.txt
1
```

The delivery must be uploaded to the *Campus Virtual*. Remember to add your name and NIA at the heading of each file.