Connected Components

Lascau Ionut Sebastian
CEN 1.2A
Year 2017 - 2018

Problem statement

Implement two different algorithms to determine the connected components in undirected graphs, e.g., based on DFS or BFS.

Application design

A depth-first search (DFS) is an algorithm for traversing a finite graph.DFS visits the child nodes before visiting the sibling nodes; that is, it traverses the depth of any particular path before exploring its breadth. A stack (often the program's call stack via recursion) is generally used when implementing the algorithm.
A breadth-first search (BFS) is another technique for traversing a finite graph. BFS visits the parent nodes before visiting the child nodes, and a queue is used in the search process.
The main idea of algorithm is to apply a breadth first search or a depth first search for each unvisited node. The number of connected components will be equal to the number of times the function is called because every node from the connected component where the BFS or DFS is applied will be marked with 1.

Pseudocode

Here are the important functions for solving the problem:

```
DFS(*graph, start_node)
1.      stack <- {empty}
2.      is_visited[start_node] <- 1
3.      no_elements_stack = 1;
4.      while no_elements_stack > 0
5.      current = stack[no_elements_stack]
6.      no_elements_stack <- no_elements_stack - 1
7.      for (each node in the graph)
8.              if adj_matrix[current][node] = 1 and is_visited[node] = 0
9.                      no_elements_stack <- no_elements_stack + 1
10.                     stack[no_elements_stack] <- node
11.                     is_visited[node] <- 1
```

```
BFS(*graph, start_node)
1.      queue <- {empty}
2.      is_visited[start_node] <- 1
3.      front <- back <- 0
4.      while front <= back
5.              current <- queue[front]
6.              front <- front + 1
7.              for (each node in the graph)
8.              if adj_matrix[current][node] = 1 and is_visited[node] = 0
9.                      back <- back + 1
10.                     queue[back] <- node
11.                     is_visited[node] <- 1
```

```
no_connected_components(*graph)
1.      no_cc = 0 // a contor for connected components
2.      for (each node in the graph)
3.              if adj_matrix[current][node] = 1 and is_visited[node] = 0
4.                      no_cc <- no_cc + 1
5.                      we apply a BFS() or a DFS()
6.  return no_cc
```

Source Code

```
//-------------------------"random_generator.h"-------------------------
#ifndef RANDOM_GENERATOR_H_INCLUDED
#define RANDOM_GENERATOR_H_INCLUDED

void random_generate_graph(struct a_graph *graph);

#endif // RANDOM_GENERATOR_H_INCLUDED
//-------------------------"random_generator.c"-------------------------
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "graph_traversals.h"
#include "random_generator.h"

void random_generate_graph(struct a_graph *graph){
    const int max_no_nodes = 5;
    int iterator_rows;
    int iterator_columns;
    int *adj_mat;
    FILE *file_out;

    srand (time(NULL));
    file_out = fopen("C:/Users/Lascau/Desktop/Project/becheru/graf.txt",
        "w");

    graph->no_nodes = rand() % max_no_nodes + 1;
    adj_mat = malloc((graph->no_nodes + 5) * (graph->no_nodes + 5) *
        sizeof(int));
    fprintf(file_out, "%d\n", graph->no_nodes);
    for (iterator_rows = 0; iterator_rows < graph->no_nodes - 1;
        iterator_rows++)
       for (iterator_columns = iterator_rows + 1; iterator_columns <
           graph->no_nodes; iterator_columns++){
           int aux = rand() % 2;
           *(adj_mat + iterator_rows * graph->no_nodes +
               iterator_columns) = aux;
           *(adj_mat + iterator_columns * graph->no_nodes +
               iterator_rows) = aux;
       }

    for (iterator_rows = 0; iterator_rows < graph->no_nodes;
        iterator_rows++){
       for (iterator_columns = 0; iterator_columns < graph->no_nodes;
           iterator_columns++)
           if (iterator_rows == iterator_columns)
               fprintf(file_out, "%d ", 0);
           else
```

```
            fprintf(file_out, "%d ", *(adj_mat + iterator_rows *
                graph->no_nodes + iterator_columns));

      fprintf(file_out, "\n");
    }

    free(adj_mat);
}
```

```
//----------------------------"graph_traversals.h"----------------------
#ifndef GRAPH_TRAVERSALS_H_INCLUDED
#define GRAPH_TRAVERSALS_H_INCLUDED

struct a_graph {
    int no_nodes;
    int *is_visited;
    int *adj_matrix;
};

void DFS(struct a_graph *graph, int start_node);
void BFS(struct a_graph *graph, int start_node);

#endif // GRAPH_TRAVERSALS_H_INCLUDED
//-----------------------------"BFS.c"----------------------------------
#include <stdlib.h>
#include "graph_traversals.h"

void BFS(struct a_graph *graph, int start_node){
    int *queue;
    int front;
    int back;
    int iterator_nodes;
    int temp_node;

    queue = calloc(graph->no_nodes, sizeof(int));
    front = back = 0;
    queue[front] = start_node;
    *(graph->is_visited + start_node) = 1;

    while (front <= back){
        temp_node = queue[front];
        front++;
        for (iterator_nodes = 0; iterator_nodes < graph->no_nodes;
             iterator_nodes++)
            if (*(graph->is_visited + iterator_nodes) == 0 &&
                *(graph->adj_matrix + iterator_nodes * graph->no_nodes +
                temp_node) == 1){
                *(graph->is_visited + iterator_nodes) = 1;
                back++;
```

```
            queue[back] = iterator_nodes;
        }
    }
    free(queue);
}

//-----------------------------"DFS.c"----------------------------------
#include <stdlib.h>
#include "graph_traversals.h"

void DFS(struct a_graph *graph, int start_node){
    int *stack;
    int no_elements_stack;
    int iterator_nodes;
    int temp_node;

    stack = calloc(graph->no_nodes ,sizeof(int));
    no_elements_stack = 1;
    stack[no_elements_stack] = start_node;
    *(graph->is_visited + start_node) = 1;

    while (no_elements_stack > 0){
        temp_node = stack[no_elements_stack];
        no_elements_stack--;
        for (iterator_nodes = 1; iterator_nodes <= graph->no_nodes;
             iterator_nodes++)
            if (*(graph->is_visited + iterator_nodes) == 0 &&
                *(graph->adj_matrix + (iterator_nodes - 1) *
                graph->no_nodes + temp_node - 1) == 1){
                *(graph->is_visited + iterator_nodes) = 1;
                no_elements_stack++;
                stack[no_elements_stack] = iterator_nodes;
            }
    }

    free(stack);
}
```

```
//---------------------------------main.c-------------------------------
#include <stdio.h>
#include <stdlib.h>
#include "graph_traversals.h"
#include "random_generator.h"

void read_adj_matrix(struct a_graph *graph){
    int iterator_rows;
    int iterator_columns;
    FILE *file_in;
```

```
    file_in = fopen("graf.txt", "r");
    fscanf(file_in, "%d", &graph->no_nodes);
    graph->is_visited = calloc(graph->no_nodes, sizeof(int));
    graph->adj_matrix = calloc(graph->no_nodes * graph->no_nodes,
        sizeof(int));
    for (iterator_rows = 0; iterator_rows < graph->no_nodes;
        iterator_rows++)
       for (iterator_columns = 0; iterator_columns < graph->no_nodes;
           iterator_columns++)
          fscanf(file_in, "%d", graph->adj_matrix + iterator_rows *
              graph->no_nodes + iterator_columns);

    fclose(file_in);
}

int no_connected_components(struct a_graph *graph){
    int no_cc = 0;
    int iterator_nodes;

    for (iterator_nodes = 0; iterator_nodes < graph->no_nodes;
        iterator_nodes++)
       if (*(graph->is_visited + iterator_nodes) == 0){
          no_cc++;
          BFS(graph, iterator_nodes);
       }
    return no_cc;
}

void delete_graph(struct a_graph *graph){
    free(graph->adj_matrix);
    free(graph->is_visited);
    free(graph);
}

int main()
{
    struct a_graph *graph;

    graph = malloc(sizeof(struct a_graph));
    random_generate_graph(graph);
    read_adj_matrix(graph);
    printf("The number of connected components is: %d",
        no_connected_components(graph));

    delete_graph(graph);
    return 0;
}
```

Experiments and results

```
Example of undirectd graph without edges:
Input:
Number of nodes is: 5
Adjacency matrix is:
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
Output: 5 connected components

Example with a undirected complet graph:
Input:
Number of nodes is: 6
Adjacency matrix is:
0 1 1 1 1
1 0 1 1 1
1 1 0 1 1
1 1 1 0 1
1 1 1 1 0
Output: 1 connected component

Example with an undirected cyclic graph:
Input:
Number of nodes is: 6
Adjacency matrix is:
0 1 0 0 0 1
1 0 1 0 0 0
0 1 0 1 0 0
0 0 1 0 1 0
0 0 0 1 0 1
1 0 0 0 1 0
Output: 1 connected component

Random example:
Input:
Number of nodes is: 4
Adjacency matrix is:
0 0 1 0
0 0 0 1
1 0 0 0
0 1 0 0
Output: 2 connected component
```

Conclusions

Working on this project, I have gained the ability to work with headers and to divide my project into small pieces with a good meaning which makes it more readable and easier to debug. Also, I learnt two very important graph algorithms.

References

1)https://en.wikipedia.org/wiki/Breadth-first_search

2)https://en.wikipedia.org/wiki/Depth-first_search

3)https://stackoverflow.com

4)http://pt.becheru.net/assignment

5)http://pt.becheru.net/labs

6)http://www.cplusplus.com/reference/cstdlib/rand/

7)http://www.sharelatex.com