Contents

1 Basic Test Results

```
Running...
Opening tar file
TreeAnalyzer.c
OK
Tar extracted O.K.
Checking files...
OK
Making sure files are not empty...
OK
Compilation check...
Compiling...
OK
Compilation seems OK! Check if you got warnings!

=====================
 Public test cases
=====================

=====================
Running test...
OK
Running test...
OK
Test good_root_last__4__3 Succeeded.
Info: valid tree with nodes 4 and 3.
=====================

=====================
Running test...
OK
Running test...
OK
Test invalid_test__4__3 Succeeded.
Info: incorrect number of nodes.
=====================


========================
= Checking coding style =
========================
 ** Total Violated Rules      : 0
 ** Total Errors Occurs       : 0
 ** Total Violated Files Count: 0
```

2 TreeAnalyzer.c

```c
/**
 * @file TreeAnalyzer.c
 * @author  Brahan Wassan <brahan>
 * @version 1.0
 * @date 27 Nov 2019
 *
 * @brief Program that build a tree from a given txt file
 *
 * @section DESCRIPTION
 * The program builds a tree from txt file
 * Input  : txt file with integer that represent the tree nodes
 * Process: checks if the user input is valid, and then print the Tree root, Vertices
 * count, num of Edges in the tree, length of the minimal, and maximal branch in the tree, the tree Diameter, and
 * the shortest path between 2 given nodes
 * Output : print all the above
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>
#include "queue.h"

#define MAX_CLI_ARG 4
#define MAX_LINE 1024
#define FILE_IDX 1
#define FIRST_NODE 2
#define SECOND_NODE 3
#define READ "r"
#define KEY_FACTOR 2
#define NUMBER_BASE 10
#define USAGE_ERR "Usage: TreeAnalyzer <Graph File Path> <First Vertex> <Second Vertex>\n"
#define INPUT_ERR "Invalid input\n"
#define ROOT_MSG "Root Vertex:"
#define NODE_COUNT "Vertices Count:"
#define EDGE_COUNT "Edges Count:"
#define MIN_BRANCH_LEN "Length of Minimal Branch:"
#define MAX_BRANCH_LEN "Length of Maximal Branch:"
#define DIAMETER_LEN "Diameter Length:"
#define SHORTEST_PATH_MSG "Shortest Path Between %d and %d: "
#define UNDEFINED_SIZE -1
#define MIN_TREE_SIZE 1
#define IS_LEAF -2
#define LEAF 1
#define FIRST_LINE 1
#define SEPARATOR " \t"
#define EQUAL 0
#define VISITED 1
#define EMPTY_QUEUE 1
#define SPACE_ASCII 32
#define NEW_LINE '\n'
#define LINE_WIN '\r'
#define NEW_LINE_WIN "\r\n"
#define LEAF_INDICATOR "-"
#define INT_LOW 48
#define INT_HI 57
#define LEAF_WIN "-\r\n"
#define LEAF_LIN "-\n"

```

```c
/**
 * The struct define one Vertex in the Graph
 */
typedef struct Vertex
{
    int parent;
    int *sons;
    int isLeaf;
    int childrenCount;
    int dist;
    int prev;
    int visit;
} Vertex;

/**
 * the function initiate a tree with default values
 * @param tree our tree
 * @param size the size of the tree
 */
void initTree(Vertex **tree, int size)
{
    for (int i = 0; i < size; ++i)
    {
        (*tree)[i].childrenCount = 0;
        (*tree)[i].sons = NULL;
        (*tree)[i].parent = -1;
        (*tree)[i].prev = -1;
        (*tree)[i].dist = -1;
        (*tree)[i].visit = -1;
        (*tree)[i].isLeaf = -1;
    }
}

/**
 * the function free all the allocated memory
 * @param tree the tree we build
 * @param treeSize the tree Size
 */
void freeEverything(Vertex **tree, int treeSize)
{
    int i = 0;
    if (*tree != NULL)
    {
        while (i < treeSize)
        {
            if ((*tree)[i].sons != NULL)
            {
                free((*tree)[i].sons);
            }
            ++i;
        }
        free(*tree);
        *tree = NULL;
    }
}

/**
 * the function prints the needed err msg
 * @param isUsage flag that indicate if its  USAGE err
 */
void errMsg(bool isUsage)
{
    if (isUsage)
    {
        fprintf(stderr, "%s", USAGE_ERR);
    }
    else
    {
        fprintf(stderr, "%s", INPUT_ERR);
    }
}

/**
 * closes the file and return fail
 * @param file the file we want to close
 * @return 1
 */
int fileErrorHandling(FILE *file)
{
    fclose(file);
    return EXIT_FAILURE;
}

/**
 * validate the line and parse it into integer arr
 * @param line a line from the given file
 * @return the size of vertex
 */
int checkFirstLine(char line[MAX_LINE])
{
    int i = 0;
    for (; i < (int) strlen(line); ++i)
    {
        if (!(line[i] >= INT_LOW && line[i] <= INT_HI) && line[i] != NEW_LINE && line[i] != LINE_WIN)
        {
            return UNDEFINED_SIZE;
        }
    }
    char *candidate = strtok(line, SEPARATOR);
    int num = (int) strtol(candidate, NULL, NUMBER_BASE);
    if (num < MIN_TREE_SIZE)
    {
        return UNDEFINED_SIZE;
    }
    return num;
}

/**
 * validate single line in the file,
 * check if their no double node (the same node) , no char , the vertices < size
 * @param line a line from the input file
 * @return the num of children if the line is valid, otherwise return -1
 */
int validateLine(char line[MAX_LINE], int curSonsArr[MAX_LINE], int treeSize) //TODO WORKS
{
    int childrens = 0;
    if ((strcmp(line, LEAF_LIN) == EQUAL) || (strcmp(line, LEAF_WIN) == EQUAL) ||
        (strcmp(line, LEAF_INDICATOR) == EQUAL))
    {
        return IS_LEAF;
    }
    if ((strcmp(line, "\n") == EQUAL) || (strcmp(line, NEW_LINE_WIN) == EQUAL))
    {
        return UNDEFINED_SIZE;
    }
    for (int i = 0; i < (int) strlen(line); ++i)
    {
        bool isNum = (!(line[i] >= INT_LOW && line[i] <= INT_HI));
        if (isNum && (line[i] != SPACE_ASCII) && line[i] != NEW_LINE && line[i] != LINE_WIN)
        {
            return UNDEFINED_SIZE;
        }
    }
    char *p = strtok(line, SEPARATOR);
    while (p != NULL)
    {
        if ((*p == NEW_LINE) || (strcmp(line, LEAF_WIN) == EQUAL))
        {
            break;
        }
        int candidate = (int) strtol(p, NULL, NUMBER_BASE);
        if (candidate > treeSize - 1)
        {
            return UNDEFINED_SIZE;
        }
        curSonsArr[childrens] = candidate;
        childrens++;
        p = strtok(NULL, SEPARATOR);
    }
    return childrens;
}

/**
 * parse the file data to a tree
 * @param fileName the given file name
 * @param tree the tree we want to build
 * @param treeSize the tree size
 * @return 1 if failed, 0 otherwise
 */
int parseFile(const char *fileName, Vertex **tree, int *treeSize)
{
    FILE *file;
    char line[MAX_LINE];
    int lineNum = FIRST_LINE;
    file = fopen(fileName, READ);
    if (file == NULL)
    {
        return fileErrorHandling(file);
    }
    int sizeOfChildren = 0;
    int curVertex = 0;
    int curSonsArr[MAX_LINE];
    while ((fgets(line, MAX_LINE, file) != NULL))
    {
        if (lineNum == FIRST_LINE)
        {
            *treeSize = checkFirstLine(line);
            if (*treeSize == UNDEFINED_SIZE)
            {
                return fileErrorHandling(file);
            }
            *tree = (Vertex *) malloc(((*treeSize) * sizeof(Vertex)));
            if (*tree == NULL)
            {
                return fileErrorHandling(file);
            }
            initTree(tree, *treeSize);
            lineNum++;
            continue;
        }
        curVertex = lineNum - KEY_FACTOR;
        sizeOfChildren = validateLine(line, curSonsArr, *treeSize);
        if (sizeOfChildren == UNDEFINED_SIZE)
        {
            return fileErrorHandling(file);
        }
        if (sizeOfChildren == IS_LEAF)
        {
            (*tree)[curVertex].isLeaf = LEAF;
            (*tree)[curVertex].childrenCount = 0;
            ++lineNum;
            continue;
        }
        if (curVertex >= *treeSize)
        {
            return fileErrorHandling(file);
        }
        else
        {
            (*tree)[curVertex].childrenCount = sizeOfChildren;
            (*tree)[curVertex].sons = (int *) malloc((sizeOfChildren) * sizeof(int));
            for (int i = 0; i < sizeOfChildren; ++i)
            {
                if ((*tree)[curSonsArr[i]].visit != VISITED)
                {
                    (*tree)[curVertex].sons[i] = curSonsArr[i];
                    int i1 = curSonsArr[i];
                    (*tree)[i1].visit = VISITED;
                }
                else
                {
                    return fileErrorHandling(file);
                }
            }
            ++lineNum;
        }
    }
    if ((*treeSize != (lineNum - KEY_FACTOR)))
    {
        fclose(file);
        return EXIT_FAILURE;
    }
    fclose(file);
    return EXIT_SUCCESS;
}

/**
 * iterate thru the tree nodes and connect between a node and its parent
 * @param tree the tree
 * @param treeSize the tree size
 */
void setParent(Vertex **tree, int treeSize)
{
    for (int k = 0; k < treeSize; ++k)
    {
        if ((*tree)[k].isLeaf != LEAF)
        {
            for (int l = 0; l < (*tree)[k].childrenCount; ++l)
            {
                int sonIdx = (*tree)[k].sons[l];
                (*tree)[sonIdx].parent = k;
            }
        }
    }
}

/**
 *  the function find the tree root
 * @param tree the tree
 * @param treeSize the tree size
 * @return the tree root, -1 if not found
 */
int getRoot(Vertex *tree, int treeSize)
{
    int i = 0;
    for (i = 0; i < treeSize; i++)
    {
        if (tree[i].parent == UNDEFINED_SIZE)
        {
            return i;
        }
    }
    return UNDEFINED_SIZE;
}

/**
 * bfs according to the given psudo code
 * @param tree the tree
 * @param treeSize  the tree size
 * @param vertex the vertex we start from
 */
void bfs(Vertex **tree, int treeSize, int vertex)
{
    for (int i = 0; i < treeSize; ++i)
    {
        (*tree)[i].dist = UNDEFINED_SIZE;
    }
    (*tree)[vertex].dist = EQUAL;
    (*tree)[vertex].prev = UNDEFINED_SIZE;
    Queue *queue = allocQueue();
    enqueue(queue, (vertex));
    while (queueIsEmpty(queue) != EMPTY_QUEUE)
    {
        int curKey = (int) dequeue(queue);
        int keyParent = (*tree)[curKey].parent;
        if (keyParent != UNDEFINED_SIZE)
        {
            if ((*tree)[keyParent].dist == UNDEFINED_SIZE)
            {
                (*tree)[keyParent].prev = (int) curKey;
                (*tree)[keyParent].dist = (*tree)[curKey].dist + 1;
                enqueue(queue, keyParent);
            }
        }
        for (int i = 0; i < (*tree)[curKey].childrenCount; ++i)
        {
            int curSonIdx = (*tree)[curKey].sons[i];
            if ((*tree)[curSonIdx].dist == UNDEFINED_SIZE)
            {
                (*tree)[curSonIdx].prev = (int) curKey;
                (*tree)[curSonIdx].dist = (*tree)[curKey].dist + 1;
                enqueue(queue, curSonIdx);
            }
        }
    }
    freeQueue(&queue);
}

/**
 * finds the minimum and maximum branches in the tree
 * @param tree the tree
 * @param treeSize  the tree size
 * @param root the root of the tree
 * @param minVal the shortest branch
 * @param maxVal the longest branch
 * @return the maxVal idx
 */
int findMinMaxBranch(Vertex *tree, int treeSize, int root, int *minVal, int *maxVal)
{
    int curMin = treeSize + 1;
    int curMax = 0, maxIdx = 0;
    bfs(&tree, treeSize, root);
    for (int i = 0; i < treeSize; ++i)
    {
        if (tree[i].dist > curMax)
        {
            curMax = tree[i].dist;
            maxIdx = i;
        }
        if ((tree[i].dist != EQUAL) && (tree[i].dist < curMin) && (tree[i].isLeaf == LEAF))
        {
            curMin = tree[i].dist;
        }
    }
    if (treeSize == MIN_TREE_SIZE)
    {
        curMin = 0;
    }
    *minVal = curMin;
    *maxVal = curMax;
    return maxIdx;
}

/**
 * finds the diameter of the tree
 * @param tree the tree
 * @param treeSize the tree size
 * @param maxIdx the vertex  in the end of the longest branch
 * @return the tree diameter
 */
int findDiameter(Vertex *tree, int treeSize, int maxIdx)
{
    int diameter = 0;
    bfs(&tree, treeSize, maxIdx);
    for (int i = 0; i < treeSize; ++i)
    {
        if (tree[i].dist > diameter)
        {
            diameter = tree[i].dist;
        }
    }
    return diameter;
}

/**
 * finds the path between two nodes
 * @param tree the tree
 * @param treeSize the tree size
 * @param u the first node
 * @param v the second node
 */
void findPath(Vertex *tree, int treeSize, int u, int v)
{
    fprintf(stdout, SHORTEST_PATH_MSG, u, v);
    bfs(&tree, treeSize, v);
    int curNode = u;
    if (u == v)
    {
        fprintf(stdout, "%d\n", v);
    }
    else
    {
        fprintf(stdout, "%d ", u);
        while (tree[curNode].prev != v && tree[curNode].prev != UNDEFINED_SIZE)
        {
            fprintf(stdout, "%d ", tree[curNode].prev);
            curNode = tree[curNode].prev;
        }
        fprintf(stdout, "%d\n", v);
    }
}

/**
 * prints the output of the program
 * @param tree the tree
 * @param treeSize the tree size
 * @param u the first node
 * @param v the second node
 */
void printOutput(Vertex *tree, int treeSize, int u, int v)
{
    int root = getRoot(tree, treeSize);
    fprintf(stdout, "%s %d\n", ROOT_MSG, root);
    fprintf(stdout, "%s %d\n", NODE_COUNT, treeSize);
    int edges = (int) (treeSize) - 1;
    fprintf(stdout, "%s %d\n", EDGE_COUNT, edges);
    int minVal, maxVal;
    int maxIdx = findMinMaxBranch(tree, treeSize, root, &minVal, &maxVal);
    fprintf(stdout, "%s %d\n", MIN_BRANCH_LEN, minVal);
    fprintf(stdout, "%s %d\n", MAX_BRANCH_LEN, maxVal);
    int diameter = findDiameter(tree, treeSize, maxIdx);
    fprintf(stdout, "%s %d\n", DIAMETER_LEN, diameter);
    findPath(tree, treeSize, u, v);
}

/**
 * parse the two CLI given nodes into integers
 * @param node the given cli node
 * @param treeSize the tree size
 * @return -1 if not valid,node value otherwise
 */
int parseNodes(char *node, int treeSize)
{
    for (int i = 0; i < (int) strlen(node); ++i)
    {
        bool isNum = (!(node[i] >= INT_LOW && node[i] <= INT_HI));
        if (isNum)
        {
            return UNDEFINED_SIZE;
        }
    }
    char *parsed = strtok(node, SEPARATOR);
    int nodeValue = (int) strtol(parsed, NULL, NUMBER_BASE);
    if (nodeValue == 0 && (strcmp(node, "0") != EQUAL))
    {
        return UNDEFINED_SIZE;
    }
    if (nodeValue > treeSize - 1 || nodeValue < 0)
    {
        return UNDEFINED_SIZE;
    }
    return nodeValue;
}

/**
 * program main
 * @param argc cli args
 * @param argv cli args
 * @return 0 if ok,1 otherwise
 */
int main(int argc, char *argv[])
{
    Vertex *tree = NULL;
    int treeSize = 0, flag = 0;
    if (argc != MAX_CLI_ARG)
    {
        errMsg(true);
        return EXIT_FAILURE;
    }
    flag = parseFile(argv[FILE_IDX], &tree, &treeSize);
    if ((flag == EXIT_FAILURE) || tree == NULL)
    {
        errMsg(false);
        freeEverything(&tree, treeSize);
        return EXIT_FAILURE;
    }
    setParent(&tree, treeSize);
    int firstNode = parseNodes(argv[FIRST_NODE], treeSize);
    int secondNode = parseNodes(argv[SECOND_NODE], treeSize);
    if (firstNode == UNDEFINED_SIZE || secondNode == UNDEFINED_SIZE)
    {
        errMsg(false);
        freeEverything(&tree, treeSize);
        return EXIT_FAILURE;
    }
    printOutput(tree, treeSize, firstNode, secondNode);
    freeEverything(&tree, treeSize);
    return EXIT_SUCCESS;
}
```