Introduction :

Le but de ce TP est de nous permettre d'étudier et de manipuler des graphes non orientés, non valués en représentation "Matrice d'adjacence". Ces arbres sont organisés sous forme de système de branche. Des arbres sont dit connexes quand tous les nœuds du graphes sont reliées donc pouvant être atteint en effectuant un seul parcours.

Appareillage :

Notre dispositif pour effectuer ce TP se compose :

- D'un PC sous Linux
- IDE (Integrated Development Environment) utilisé : Code::Block, Eclipse, XCode ou NetBeans

Documents fournis dans l'archive :

- Le rapport de TP.
- Les fichiers sources : genere_graphe_0.c / header.h / main.c / queue.c
- Graphe généré aléatoirement : graphe_alea.txt
- Un Makefile
- Un exécutable
- Une trace d'exécution : trace.txt

1. Travail à réaliser :

Le but de ce programme consiste à développer deux fonctions permettant de parcourir un graphe. Il y aura la fonction de parcours en profondeur et celle du parcours en largeur. La différence entre chacune d'elle est la méthode utilisée pour parcourir chaque nœud du graphe.

Pour cela quelques indications vont nous permettre d'effectuer les différentes fonctions demandées.

Explication :

Parcours en profondeur :

Pour un parcours en profondeur, on va toujours le plus en profondeur possible, on parcourt une "branche" jusqu'à son extrémité (la feuille).

Le principe est le même que pour le parcours en profondeur d'un arbre binaire. Mais il y a deux différences qui proviennent de la possible non-connexité ou de la présence des cycles :

- A cause de la possible non-connexité : le parcours en profondeur à partir d'un sommet peut ne pas parcourir tout le graphe.
- A cause de l'éventuelle présence des cycles : il faut faire attention de ne pas traiter deux fois le même sommet.

Parcours en largeur :

Le parcours d'un arbre en largeur consiste à parcourir un graphe par niveau. Dans ce type de parcours une file est utilisée. Si le graphe est connexe, tous les nœuds seront visités en une seule fois.

Etape 1 :

Objectif du programme : Implémenter une fonction de parcours en profondeur avec affichage des sommets en ordre préfixe et suffixe et comptage du nombre de noeuds de chaque composante connexe du graphe.

Code de la Structure *parcoursProfondeur* :

```
/*
* Prépare et utilise la fonction visiteProfondeur pour explorer le
* graphe.
*/

bool parcoursProfondeur(GRAPH* graph) {
    int x = 0, nodeNumber = 0;

    if (graph == NULL) {
        return false;
    }

    graph->componentInfo.nbComponents = 0;
    graph->componentInfo.relatedComponents = (int*) calloc(graph-
>nbSummit, sizeof (int));
    graph->nodeStatus = (char*) calloc(graph->nbSummit, sizeof
(char));

    if (graph->nodeStatus == NULL) {
        return false;
    }

    for (x = 0; x < graph->nbSummit; x++) {
        if (graph->nodeStatus[x] == WHITE) {
            graph->componentInfo.nbComponents++;
            nodeNumber += visiteProfondeur(graph, x);
        }
    }

    printf("Nombre de noeuds visites : %d\n", nodeNumber);
    free(graph->nodeStatus);
    graph->nodeStatus = NULL;

    return graph->componentInfo.nbComponents == 1;

}
```

Code de la Structure *visiteProfondeur* :

```
/*
* Parcours du graphe en profondeur.
* Permet l'affichage en préfixe et en suffixe des nœuds parcourus.s
*/

int visiteProfondeur(GRAPH* graph, int nbSummit) {
    int y, nodeNumber = 0;

    if (nbSummit < 0) {
        return 0;
    }

    if (graph->nodeStatus == NULL) {
        return 0;
    }

    graph->nodeStatus[nbSummit] = GREY;
    printf("Noeud prefixe : %d\n", nbSummit);

    for (y = 0; y < graph->nbSummit; y++) {
        if (graph->matrix[nbSummit][y] == 0) {
            continue;
        }

        if (graph->nodeStatus[y] == WHITE) {
            nodeNumber += visiteProfondeur(graph, y);
        }
    }

    graph->nodeStatus[nbSummit] = BLACK;
    nodeNumber++;
    graph->componentInfo.relatedComponents[nbSummit] = graph-
>componentInfo.nbComponents;
    printf("Noeud suffixe : %d\n", nbSummit);

    return nodeNumber;

}
```

Etape 2 :

Objectif du programme : Implémentation une fonction de parcours en largeur avec comptage du nombre de noeuds de chaque composante connexe du graphe. Le fichier du premier TP sur les files a été réutilisé pour effectuer cette fonction.

Code Fonction *ParcoursLargeur* :

```
/*
* Prépare et utilise la fonction visiteLargeur pour explorer le
* graphe.
*/

bool ParcoursLargeur(GRAPH* graph) {
    int nodeNumber = 0, x = 0;

    if (graph == NULL) {
        return false;
    }

    graph->componentInfo.nbComponents = 0;
    graph->componentInfo.relatedComponents = (int*) calloc(graph-
>nbSummit, sizeof (int));
    graph->nodeStatus = (char*) calloc(graph->nbSummit, sizeof
(char));

    if (graph->nodeStatus == NULL) {
        return false;
    }

    for (x = 0; x < graph->nbSummit; x++) {
        if (graph->nodeStatus[x] == WHITE) {
            graph->componentInfo.nbComponents++;
            nodeNumber += visiteLargeur(graph, x);
        }
    }

    printf("Nombre de noeuds visites : %d\n", nodeNumber);
    free(graph->nodeStatus);
    graph->nodeStatus = NULL;

    return graph->componentInfo.nbComponents == 1;
}
```

Code Fonction *visiteLargeur* :

```
/*
* Parcours du graphe en largeur.
* N'affiche que les nœuds visités.
*/

int visiteLargeur(GRAPH* graph, int nbSummit) {
    int y, nodeNumber = 0;
    QUEUE queue;

    if (nbSummit < 0) {
        return 0;
    }

    if (graph->nodeStatus == NULL) {
        return 0;
    }

    graph->nodeStatus[nbSummit] = GREY;
    queue = newEmptyQueue();
    queue = add(nbSummit, queue);

    while (!isEmpty(queue)) {
        int nbSommet = getHead(queue);

        for (y = 0; y < graph->nbSummit; y++) {
            if (graph->matrix[nbSommet][y] == 0) {
                continue;
            }

            if (graph->nodeStatus[y] != WHITE) {
                continue;
            }

            graph->nodeStatus[y] = GREY;
            queue = add(y, queue);
        }

        queue = get(queue);
        graph->nodeStatus[nbSommet] = BLACK;
        printf("Noeud visité : %d\n", nbSommet);
        graph->componentInfo.relatedComponents[nbSommet] = graph-
>componentInfo.nbComponents;
        nodeNumber++;
    }

    return nodeNumber;

}
```

2. Conclusion :

Ce TP aura été pour nous l'occasion de manipuler des graphes à la fois non orientés et non valués. Nous avons de plus pu implémenter le parcours de graphe en profondeur ainsi que le parcours en largeur.