



# Rapport Projet d'algorithmique

2 ème année licence informatique

sur

## Tables de routage

RÉALISÉ PAR :

**FERIEL BOUSSOUAR TD02**  
**MASSILIA MOUNGAD TD02**  
**ILYES BERHIL TD04**

---

ENCADRE PAR :

MR THIERRY MAUTOR



# Table des matières

- INTRODUCTION
- PRÉSENTATION DU PROJET
- LA CRÉATION ALÉATOIRE D'UN RÉSEAU RÉALISTE
- LA VÉRIFICATION DE LA CONNEXITÉ DU RÉSEAU
- LA DÉTERMINATION DE LA TABLE DE ROUTAGE DE CHAQUE NOEUD
- LA RECONSTITUTION DU CHEMIN ENTRE 2 NOEUDS
- PROGRAMME COMPLET DU PROJET
- CONCLUSION

Tout au long du semestre, nous avons eu la chance de découvrir le monde des graphes grâce au module d'algorithmique qui offre des cours intéressants en théorie des graphes.

Nous tenons à remercier Monsieur Thierry Mautor et Monsieur Dominique Barth .

## Introduction

Rien de plus banal que de consulter un plan du métro, utiliser un catalogue de liaisons aériennes pour préparer nos déplacements, envoyer un mail, utiliser un moteur de recherche pour trouver une page web, découvrir quels sont les amis de nos amis sur un réseau social ...

La notion commune est donc un **graphe** .

## Définitions:

**Un graphe** est la donnée d'un ensemble d'éléments, nommés les sommets (qui vont représenter les stations, les pages web, les gens, etc...), et les relations entre ces sommets que l'on appelle les arêtes.

**Routage** est le mécanisme par lequel des chemins sont sélectionnés dans un réseau pour acheminer les données d'un expéditeur jusqu'à un ou plusieurs destinataires.

**Table de routage** est une structure de données utilisée par un routeur ou un ordinateur en réseau et qui associe des préfixes à des moyens d'acheminer les trames vers leur destination.

# Présentation du Projet

**Objet :** Ce projet consiste à réaliser une application qui permette d'établir la table de routage de chaque nœud d'un réseau constitué de 100 nœuds.

**Outils de développement :** nous avons choisi le langage C pour programmer le projet parce que c'est le langage le plus rapide en terme d'exécution, et l'éditeur de texte geany.

**Structure du projet :** Notre projet est structuré en un seul fichier **reseau.c**

Il est divisé en quatre parties, la première partie concerne la création aléatoire d'un réseau connexe et non orienté, ensuite dans la deuxième partie sur la connexité du réseau après la détermination des tables de routage, après la reconstitution du chemin entre deux sommets et pour arriver à la conclusion où nous allons donner le bilan de notre projet .

## **Compilation:**

Le programme se compile via la commande:

**gcc reseau.c**

## **Exécution:**

**./a.out**

La structure de données principale est la struct **reseau**, celle-ci regroupe le nombre de sommets du réseau, la matrice des temps qui sert à stocker les temps de communication de chaque arc, le tableau de couleur pour pouvoir appliquer le parcours en profondeur, un entier de départ et un entier de destination qui servent à stocker les sommets saisis, des matrices adjacentes des prédecesseurs et matrice Floyd Warshall pour pouvoir reconstituer le plus court chemin en appliquant l'algorithme de Floyd Warshall.

```
struct reseau{  
    int sommets;  
    int **temps;  
    int *col;  
    int depart;  
    int destination;  
    int **matrice_adj;  
    int **Pred;  
    int **FW;  
};  
typedef struct reseau Reseau;
```

# La création aléatoire du réseau

Cette partie concerne la création d'un réseau connexe et non orienté caractérisé par 3 types de noeuds (TIER1, TIER2, TIER3), les premiers sont très connectés entre eux, les TIER2 sont presque reliés entre eux et aux sommets TIER1 et pareil pour les TIER3 qui sont connectés entre eux et aux sommets de TIER2. Chaque lien entre deux sommets est pondéré d'une valeur tirée aléatoirement représentant le cout de transit "temps de communication".

Pour réaliser cette partie nous avons utilisé struct `reseau` et les fonctions suivantes:

- `int aleatoire (int min, int max)`: permet de calculer une valeur tirée aléatoirement comprise entre la valeur min et la valeur max.

- **int alea\_temps (int type):** permet de calculer le temps de communication des liens entre les sommets selon les types leurs TIER1, TIER2, TIER 3 ( Ce sont des constantes définis par #define). Nous avons utilisé la fonction aleatoire() afin de tirer une valeur aléatoire .

- **Reseau initialiser\_reseau**

**(Reseau R):** permet de créer le réseau R, donc allouer et initialiser une zone mémoire à l'aide de la fonction malloc pour tous les champs de la structure réseau et on retourne reseau R à la fin , par exemple pour la tableau de couleur col, nous avons fait :

**Reseau reseau\_aleat(Reseau R) ;**  
permet de créer le reseau donc de créer les 3 TIERS, nous avons initialisé R en utilisant la fonction précédente

```

if ( type == TIER1 ) {
    temps = aleatoire(5,10);
    return temps;
}
else if ( type == TIER2 ) {
    temps = aleatoire(10, 20);
    return temps;
}
else if ( type == TIER3 ) {
    temps = aleatoire(15, 50);
    return temps ;
}

```

R.col = malloc (R.nb\_sommets \* sizeof (int));

R = **initialiser\_reseau(R)**

La création des TIERS:

**TIER1:** nous avons fait une boucle FOR qui permet de connecter 10 nœuds de TIER1 entre eux et de remplir la matrice des temps en utilisant la fonction `alea_temps()` afin de générer une valeur tirée aléatoirement comprise entre 5 et 10

- `Rand () % 100 > 25` : permet de donner une chance de 75%.

**TIER2:** nous avons fait des boucles imbriquées :

La première permet de connecter 20 nœuds de TIER2 à 1 ou 2 nœuds de TIER1, les nœuds du tier 1 sont choisis aléatoirement en utilisant la fonction `aleatoire` et remplir la matrice de temps en utilisant la fonction `alea_temp` afin de générer une valeur tirée aléatoirement comprise entre 10 et 20.

`nb = aleatoire(1, 2)` : 1 ou 2 sommets  
`x = aleatoire(0, 9)` ; les sommets de TIER 1

```
if ((rand () % 100) > 25 && i != j)
R.matrice_adj[i][j] = R.matrice_adj[j][i] = 1;
R.temps[j][i] = R.temps[i][j] =
alea_temp(TIER1)
}
```

```
int nb, nb2, x;
nb = aleatoire (1, 2);
for (j = 0; j < nb; j++){
x = aleatoire (0, 9);
R.matrice_adj[i][x] = R.matrice_adj[x][i]=1;
R.temps[i][x] = R.temps[x][i] =
alea_temp (TIER2) }
```

- La deuxième sert à connecter chaque nœud de TIER2 à 2 ou 3 nœud de TIER2 en utilisant la fonction **aleatoire**, et remplir la matrice de temps en utilisant la fonction **alea\_temp** afin de générer une valeur tirée aléatoirement comprise entre 10 et 20.

**nb** = **aleatoire** (1, 2) : 1 ou 2

sommets

**x** = **aleatoire**(0, 9) ; les sommets de TIER2

```

nb2 = aleatoire (2, 3);
for (j = 0; j < nb2; j++){
  x = aleatoire (10, 29);
  R.matrice_adj[i][x] = R.matrice_adj[x][i] = 1;
  R.temps[i][x] = R.temps[x][i] =
    alea_temp (TIER2)
}

```

**TIER3:** nous avons fait aussi des boucles imbriquées :

La première boucle permet de connecter 70 nœuds du TIER3 à 2 nœuds de TIER2, les nœuds du TIER2 sont choisis aléatoirement en utilisant la fonction **aleatoire**, et remplir la matrice de temps en utilisant la fonction **alea\_temp** afin de générer une valeur tirée aléatoirement comprise entre 15 et 50.

**x** = **aleatoire**(10, 29) ; les sommets de TIER 2

```

for (j = 0; j < 2; j++){
  x = aleatoire (10, 29);
  R.matrice_adj[i][x] = R.matrice_adj[x][i] = 1;
  R.temps[i][x] = R.temps[x][i] =
    alea_temp (TIER3)
}

```

La deuxième boucle sert à connecter chaque nœud de TIER3 à 2 ou 3 nœud de TIER3 en appelant la fonction **aleatoire**. Remplir la matrice de temps en appelant **alea\_temp** afin de générer une valeur tirée aléatoirement comprise entre 15 et 50 .

`x = aleatoire(30, 99);` les sommets de TIER1

- Nous avons initialisées la matrice adjacence 1 pour tous les sommets

**void afficher\_reseau(Reseau R):**  
elle permet d'afficher les liens entre les sommets du reseau R et les temps de communication entre eux en faisant deux boucles imbriquées for .

```
for (j = 0; j < 1 ; j++){
    x = aleatoire (30, 99);
    R.matrice_adj[i][x] = R.matrice_adj[x][i]=1;
    R.temps[i][x] =R.temps[x][i] =
        alea_temps(TIER3)
```

```
for (int i = 1; i < R.sommets; i++) {
    for (int j = 1; j < R.sommets; j++) {
        if (R.temps[i][j] != 0) {
            printf("A %d <== temps : %d ==> %d B\n",
                i, R.temps[i][j], j);
        }
    }
}
```

# Affichage dans le terminal

```
A 5 <== TEMPS : 6 ==> 1      B
A 5 <== TEMPS : 3 ==> 2      B
.....
A 5 <== TEMPS : 8 ==> 9      B
A 18 <== TEMPS : 13 ==> 6   B
A 18 <== TEMPS : 17 ==> 27 B
A 34 <== TEMPS : 21 ==> 11 B
A 34 <== TEMPS : 44 ==> 91 B
.....
A 99 <== TEMPS : 39 ==> 53 B
```

## Analyse et vérification:

- A 5 est de TIER1, B 6 est de TIER1 et le temps de communication est 6 (5 < 6 < 10)
- A 18 est de TIER2, B 6 est de TIER1 et le temps de communication est 13 (10 < 13 < 20)
- A 18 est de TIER2, B 27 est de TIER2 et le temps de communication est 17 (10 < 17 < 20)
- A 34 est de TIER3, B 11 est de TIER2 et le temps de communication est 21 (15 < 21 < 50)

- **A** 34 est de TIER3, **B** 91 est de TIER3 et le temps de communication est **44** ( $15 < 44 < 50$ )
- **A** 99 est de TIER3, **B** 53 est de TIER3 et le temps de communication est **39** ( $15 < 39 < 50$ )

Nous constatons que notre réseau a été bien créé avec les trois types différents de tiers et avec des valeurs respectives aléatoires de temps de communication .

# La vérification de la connexité du réseau

Le but de cette partie est de faire un parcours en profondeur pour vérifier la connexité du réseau en appliquant l'algorithme de coloriage des composantes connexes que nous avons vu en cours et en TD .

## Algorithme de coloriage:

Colorier chaque nœud du graphe d'une même couleur si elle fait partie de la composante connexe et si la même couleur est utilisée cela signifie que le graphe est bien connexe sinon il n'est pas connexe et on relance la création de réseau .

**Colorier(Graphe G, Sommet s,Couleur c)**

couleur[s] = c

Pour chaque sommet v voisin de s faire

Si (couleur[v] ==0)alors

**Colorier(G,v,c)**

**Composantes connexes(Graphe G)**

couleur = 1

Pour chaque sommet de x de v faire couleur[x] = 0

Pour chaque sommet de x de v faire

Si couleur[x] == 0

**Colorier(G,x,couleur)**

couleur = couleur+1

# Le principe de parcours en profondeur

Il trouve l'ensemble des sommets accessibles depuis un sommet donné s.C'est-à-dire ceux vers lesquels il existe un chemin partant de s, ce sont les sommets marqués par l'algorithme, (On peut utiliser ce parcours pour le calcul les composantes connexes).

Autrement dit, l'exploration progresse à partir d'un sommet s en s'appelant récursivement pour chaque sommet voisin de s.

## En C :

- **Reseau colorier\_sommet(Reseau R, int sommet, int couleur):** c'est une fonction récursive, elle permet de marquer les sommets ayant été parcourus à l'aide de l'appel récursif **R = colorier\_sommet(R, i, couleur)**, et à chaque itération elle affiche "le sommet est colorié"
- **Reseau colorier\_sommet(Reseau R, int sommet, int couleur){**  
    int i;  
    R.col[sommet] = couleur;  
    printf("le sommet %d est colorié \n", sommet);  
    **for** (i = 0; i < R.sommets; i++) {  
        **if** (R.col[i] == 0 **&&** R.col[i] != couleur)  
        }  
        R = **colorier\_sommet(R, i, couleur);**  
    **return** R;  
}

- **Reseau parcours\_profondeur(Reseau R):** le but de cette fonction est d'explorer tous les sommets et les marquer en utilisant la fonction **colorier\_sommet** afin d'atteindre chaque sommet et de vérifier la connexité du réseau R

```
int compte = 0, int couleur ;
```

- Boucle for pour initialiser le tableau de couleur des sommets à 0
- On initialise couleur à 1 , couleur = 1
- Boucle for pour tester si la couleur des sommets égale à 0, on utilise la fonction **colorier\_sommet**, **R = colorier\_sommet(R, j, couleur)** et on incrémente couleur , couleur ++

le sommet 0 est colorié  
 le sommet 2 est colorié  
 le sommet 3 est colorié  
 le sommet 4 est colorié  
 .....  
 le sommet 99 est colorié

- Pour la vérification de la connexité du réseau, on fait une boucle for qui teste si un sommet est de couleur 1 , si c'est le cas on incrémente la variable compteur
- Si tous les sommets ont une couleur 1 alors, ils sont d'une même composante connexe
- Si compteur == le nombre de sommets donc le réseau est connexe, on affiche "le réseau est bien connexe",sinon "le réseau n'est pas connexe" et on relance la création du réseau en utilisant la fonction **resau\_aleat , R = resau\_aleat(R)**

```

Reseau parcours_profondeur(Reseau R) {

    int i, j, couleur, compteur = 0;

    for (i = 0; i < R.sommets; i++) {
        R.col[i] = 0;
    }
    couleur = 1;
    for (j = 0; j < R.sommets; j++) {
        if (R.col[j] == 0) {
            R = colorier_sommet(R, j, couleur);
            couleur++;
        }
    }
    printf("\n");
    printf("La vérification de la connexité du réseau\n");
    for (i = 0; i < R.sommets; i++) {
        printf("%d ", R.col[i]);
        if (R.col[i] == 1) {
            compteur++;
        }
    }
    printf("\n");

    if (compteur == R.sommets) {
        printf("Le réseau est bien connexe.\n");
    } else {
        printf("Le réseau n'est pas connexe \n");
        R = reseau_aleat(R);
    }
    return R;
}

```

Dans notre projet, le réseau est bien connexe puisque compteur == le nombre de sommets, c'est à dire que tous les sommets sont connectés entre eux

# La détermination de la table de routage de chaque nœud

Le but de cette partie est de déterminer la table de routage de chaque nœud.

- Calculer le plus court chemin dans un graphe intervient dans les protocoles de communication (pour le routage par exemple, avec des protocoles comme RIP ou BGP), pour calculer des trajets optimaux (dans un GPS par exemple),... De nombreux algorithmes ont été inventés dans les années 50 et utilisent chacun des mécanismes différents. On peut citer notamment l'algorithme de Floyd-Warshall (1959), l'algorithme de Dijkstra (1959) ou l'algorithme de Ford-Bellman. Tous ces algorithmes sont quadratiques ou cubiques en le nombre de sommets du graphe, ce qui est souvent trop en pratique.

Nous avons utilisé l'algorithme de **Floyd-Warshall** car il permet de calculer les distances des plus courts chemins entre toutes paires de sommets dans un graphe pondéré.

Autrement dit, pour tout couple de sommets(i, j), il détermine s'il existe un chemin de i à j ,il calcule sa longueur et il construit une table de routage .

Il prend en entrée une matrice d'adjacence du graphe de taille n \*n

ou n : le nombre de sommets

---

#### Igorithme 11: Algorithme de Floyd

---

Données : Un graphe orienté pondéré  $G = (X, A, W)$

Résultat : Le plus court chemin entre toute paire de sommets de G

// M : matrice des plus courts chemins  
// P : matrice des prédecesseurs pour les plus courts chemins  
Initialiser M à  $+\infty$   
Initialiser P à 0

pour i allant de 1 à N faire

$M_{i,i} \leftarrow 0$   
 $P_{i,i} \leftarrow i$   
pour tout successeur j de i faire  
└  $M_{i,j} \leftarrow W[i, j]$

// Calcul des matrices successives

pour k allant de 1 à N faire

pour i allant de 1 à N faire  
  pour j allant de 1 à N faire  
    si  $M_{i,k} + M_{k,j} < M_{i,j}$  alors  
      └  $M_{i,j} = M_{i,k} + M_{k,j}$   
      └  $P_{i,j} = P_{k,j}$

si  $\exists i | M_{i,i} < 0$  alors

  └ retourner Il existe un circuit de longueur négative passant par i

sinon

  └ retourner M

---

Pour pouvoir remplir la matrice FW avec les temps, nous avons parcourus la liste des sommets de R et initialisé la matrice FW d'un sommet à 0, ensuite dans la deuxième boucle nous avons rempli la matrice FW des sommets avec les temps entre sommet i et sommet j .

Et pour remplir la matrice Pred avec les prédecesseurs, nous avons fait trois boucle imbriquées qui parcourent la liste des sommets de R

Pour afficher les matrices FW et Pred

```
for (int i = 0; i < R.sommets; i++) {
    R.FW[i][i] = 0 ;
    for( int j = 0; j < R.sommets; i++) {
        if (R.matrice_adj[i][j] == 1){
            R.FW[i][j] = R.temps[i][j];
        }
    }
}
```

```
for (k = 0; k < R.sommets; k++) {
    for (i = 0; i < R.sommets; i++) {
        for (j = 0; j < R.sommets; j++) {
            if (R.FW[i][k] + R.FW[k][j] < R.FW[i][j]) {
                R.FW[i][j] = R.FW[i][k] + R.FW[k][j];
                R.Pred[i][j] = R.Pred[k][j];
            }
        }
    }
}
```

```
printf ("MATRICE DES PREDECESSEURS:\n ");
for (i = 0; i < R.nb_sommets; i++)
    for (j = 0; j < R.nb_sommets; j++)
        printf (" %3d ", R.Pred[i][j])

printf ("MATRICE DES TEMPS:\n ");

for (i = 0; i < R.nb_sommets; i++)
    for (j = 0; j < R.nb_sommets; j++)
        printf (" %3d ", R.FW[i][j])
```

Cet algorithme permet de déterminer le plus court chemin d'un noeud à tous les autres (all to all) à l'aide des matrices (matrice Pred des prédecesseurs et la matrice FW pour Floyd Warshall)

Nous avons stocké dans la matrice FW les temps minimaux et dans la matrice Pred les prédecesseurs d'un sommet, chaque sommet a une matrice des temps minimaux par rapport aux autres sommets ( 99 sommets ) et chaque sommet a une matrice de prédecesseur. Donc en total nous avons 100 matrices FW et 100 matrices Pred.

En c :

- **void Floyd\_Warshall (Reseau R):** nous avons fait cette fonction pour implémenter l'algorithme de Floyd Warshall, Nous avons fait des boucles for imbriquées pour pouvoir remplir les matrices FW et Pred du reseau R pour chaque sommet.

# Affichage dans le terminal

100 matrices FW et 100 matrices Pred sont affichées pour les 100 noeuds, nous avons pris l'exemple de premier sommet :

## Matrice des temps

```
0 6 7 6 5 11 8 6 5 8 19 23 16 21 22 20  
27 24 15 17 19 22 23 17 19 23 10 23 17 30 52 35  
51 36 63 44 34 44 53 45 48 60 52 43 45 62 53 51  
49 38 47 39 34 43 57 67 57 42 46 35 48 58 60 50  
42 48 54 48 46 50 41 44 45 57 40 45 45 46 31 47  
48 51 47 47 53 58 38 52 27 60 53 56 45 56 43 31  
46 53 43 45
```

## Matrice des prédecesseur

```
0 0 0 0 0 3 0 0 0 0 4 2 3 7 1 9  
9 6 0 3 6 8 2 8 6 26 0 1 0 5 18 20  
31 24 11 15 19 28 24 14 21 26 20 15 13 15 10 10  
23 12 19 15 19 21 20 23 13 19 26 15 27 22 23 16  
13 19 16 27 12 88 25 12 18 24 20 28 18 15 26 19  
25 24 95 16 23 25 24 19 26 14 13 10 23 10 27 26  
5 22 16 19
```

# La reconstitution du chemin entre 2 noeuds

Le but de cette partie est de reconstituer un chemin entre un sommet émetteur de message et sommet destinataire à l'aide des tables de routage .

Nous avons utilisé **les structures de données suivantes “ pile” et “ Élément”** pour empiler tous les noeuds qui sont dans la matrice des prédecesseurs Pred afin de reconstituer le chemin dans le bon sens entre les deux sommets .

```
typedef struct Element Element;
struct Element{
    int nombre;
    Element *suivant;
};
```

```
typedef struct Pile Pile;
struct Pile{
    Element *premier;
};
```

Les principales fonctions pour implémenter une pile sont les suivantes :

- `pile * initialiser()`, elle permet d'initialiser une pile .
- `void empiler(Pile * pile , int Nomb)`, elle permet d'empiler dans la pile .
- `int depiler (Pile * pile)`, elle permet de dépiler la pile .
- `Pile * vider Pile(Pile * pile)`, elle permet de vider la pile .
- `void afficherPile(Pile * pile, Réseau R )`, elle permet d'afficher la pile .

```
Pile * initialiser() {  
  
    Pile * pile = malloc(sizeof( *  
    pile));  
    pile -> premier = NULL;  
    return pile; }
```

```
void empiler(Pile * pile, int Nomb) {  
  
    Element * nouveau = malloc(sizeof( *  
    nouveau));  
    if (pile == NULL || nouveau == NULL) {  
        exit(EXIT_FAILURE);  
    }  
    nouveau -> nombre = Nomb;  
    nouveau -> suivant = pile -> premier;  
    pile -> premier = nouveau; }
```

```
int depiler(Pile * pile) {  
  
    if (pile == NULL) {  
        exit(EXIT_FAILURE);  
    }  
    int nombreDepile = 0;  
    Element * elementDepile = pile ->  
    premier;  
  
    if (pile != NULL && pile -> premier !=  
    NULL) {  
        nombreDepile = elementDepile ->  
        nombre;  
        pile -> premier = elementDepile ->  
        suivant;  
        free(elementDepile);  
    }  
    return nombreDepile; }
```

```
Pile * viderPile(Pile * pile) {  
  
    while (pile -> premier != NULL) {  
        depiler(pile);  
    }  
    return pile; }
```

```

void afficherPile(Pile * pile, Reseau R) {
    if (pile == NULL) {
        exit(EXIT_FAILURE);
    }
    Element * noeud = pile -> premier;
    do {
        printf("Le message prend
le sommet %d ==>", noeud -> nombre);
        noeud = noeud -> suivant;
    }
    while (noeud != NULL); }
```

La fonction principale qui permet de reconstituer le chemin entre deux noeuds:

- **Void reconstitution\_chemin\_sommets(Reseau R)**, dans cette fonction on a utilisé la fonction **scanf** pour saisir un sommet émetteur et un destinataire.

```

printf("Noeud émetteur:");
scanf("%d", &(R.depart));
printf("\n");
printf("Noeud destinataire:");
scanf("%d", &(R.destination));
```

Ensuite, on a initialisé une pile p en utilisant la fonction **initialiser()** afin de déterminer le destinataire d'un sommet et l'empiler dans la pile, on a utilisé la fonction **afficherpile()** pour afficher le sommet qui suit le sommet de départ ( le successeur) et on affiche le sommet d'arrivée.

```
Pile * p = initialiser();
printf("Le message part de sommet %d ==>", R.depart);
do {
    R.destination = R.Pred[R.depart][R.destination];
    empiler(p, R.destination);
}
while (R.Pred[R.depart][R.destination] != R.depart);
afficherPile(p, R);
printf("Le message arrive au sommet %d\n", recevoir);
```

On affiche le temps pour recevoir le message de sommet de départ au sommet destinataire à l'aide de la matrice FW

```
printf("Le temps pour recevoir le message de %d à %d est :%d\n",
R.depart, recevoir, R.FW[R.depart][recevoir]);
```

## Affichage dans le terminal:

NOEUD ÉMETTEUR : ( À SAISIR AU CLAVIER ) PAR  
EXEMPLE 6

NOEUD DESTINATAIRE ( À SAISIR AU CLAVIER) PAR  
EXEMPLE 4

LE MESSAGE PART DE SOMMET 6 ==> PREND LE SOMMET  
6 ==> LE MESSAGE ARRIVE AU SOMMET 4

LE TEMPS POUR RECEVOIR LE MESSAGE DE 6 À 4 EST 10

# Programme Complet:

```
#define TIER1 - 10
#define TIER2 - 20
#define TIER3 - 30
#define MATRICE 1000

struct reseau {
    int sommets; //nombre de sommets
    int depart; //depart du routage
    int destination; //destination du routage
    int ** temps; // matrice des temps
    int * col; //tableau pour parcours profondeur
    int ** matrice_adj; //matrice adjacence
    int ** Pred; //matrice des predecesseurs
    int ** FW; //matrice modifiée pour Floyd Warshall
};

typedef struct reseau Reseau;

// la création du réseau

int aleatoire(int min, int max) {
    if (max < min) {
        return 0;
    }
    int alea = (rand() % (max - min + 1)) + min;
    return alea;
}

int alea_temps (int type) {
    int temps;
    if (type == TIER1) {
        temps = aleatoire(5, 10);
        return temps;
    }
    else if (type == TIER2) {
        temps = aleatoire(10, 20);
        return temps;
    }
    else if (type == TIER3) {
        temps = aleatoire(15, 50);
        return temps;
    }
    return 0;
}
```

```

Reseau initialiser_reseau(int n) {
    Reseau R;
    R.sommets = n;
    R.col = malloc(R.sommets * sizeof(int));
    int i;
    R.matrice_adj = malloc(R.sommets * sizeof(int *));
    for (i = 0; i < R.sommets; i++) {
        R.matrice_adj[i] = calloc(R.sommets, sizeof(int));
    }
    R.temps = malloc(R.sommets * sizeof(int *));
    for (i = 0; i < R.sommets; i++) {
        R.temps[i] = malloc(R.sommets * sizeof(int));
    }
    R.Pred = malloc(R.sommets * sizeof(int *));
    for (i = 0; i < R.sommets; i++) {
        R.Pred[i] = malloc(R.sommets * sizeof(int));
    }

    R.FW = malloc(R.sommets * sizeof(int *));
    for (i = 0; i < R.sommets; i++) {
        R.FW[i] = malloc(R.sommets * sizeof(int));
    }
    return R;
}

```

```

Reseau reseau_aleat(Reseau R) {
    R = initialiser_reseau(nb_sommets);
    int i, j;
    /*TIER1*/
    for (i = 0; i <= 9; i++) {
        for (j = 0; j <= 9; j++) {
            if ((rand() % 100) > 25 && i != j) {
                R.matrice_adj[i][j] = R.matrice_adj[j][i] = 1;
                R.temps[j][i] = R.temps[i][j] = alea_temps(TIER1);
                // Sommet A //Sommet B
            }
        }
    }
}

```

```

/*TIER2 */

int nb, nb2, x;
for (i = 10; i <= 29; i++) { //pour chaque noeud du TIER2
    nb = aleatoire(1, 2); // 1 ou 2 noeuds de TIER1

    for (j = 0; j < nb; j++) {
        x = aleatoire(0, 9); //choix du noeud du TIER1
        R.matrice_adj[i][x] = R.matrice_adj[x][i] = 1;
        R.temps[i][x] = R.temps[x][i] = alea_temps(TIER2);
    }
}

for (i = 10; i <= 29; i++) { //pour chaque noeud du TIER2
    nb2 = aleatoire(2, 3); // 2 ou 3 noeuds de TIER2
    for (j = 0; j < nb2; j++) {
        x = aleatoire(10, 29); //choix du noeud de TIER2
        R.matrice_adj[i][x] = R.matrice_adj[x][i] = 1;
        R.temps[i][x] = R.temps[x][i] = alea_temps(TIER2);
    }
}
/*TIER3 */

for (i = 29; i <= 99; i++) {
    for (j = 0; j < 2; j++) {
        x = aleatoire(10, 29); // 2 noeuds de TIER2
        R.matrice_adj[i][x] = R.matrice_adj[x][i] = 1;
        R.temps[i][x] = R.temps[x][i] = alea_temps(TIER3);
    }
}

for (i = 29; i <= 99; i++) {
    for (j = 0; j < 1; j++) {
        x = aleatoire(29, 99); //1 noeud du TIER3
        R.matrice_adj[i][x] = R.matrice_adj[x][i] = 1;
        R.temps[i][x] = R.temps[x][i] = alea_temps(TIER3);
    }
}

return R;

void afficher_reseau(Reseau R) {
    printf("La creation du reseau \n");
    int i, j;
    for (i = 1; i < R.sommets; i++) {
        for (j = 1; j < R.sommets; j++) {
            if (R.temps[i][j] != 0){
                printf("A %d <= temps : %d ==> %d B\n", i, R.temps[i][j], j);
            }
        }
    }
}

```

```

void desalloue_reseau(Reseau R) {
    int i;
    for (i = 0; i < R.sommets; i++) {
        free(R.matrice_adj[i]);
    }
    free(R.matrice_adj);

    for (i = 0; i < R.sommets; i++) {
        free(R.temps[i]);
    }
    free(R.temps);

    for (i = 0; i < R.sommets; i++) {
        free(R.Pred[i]);
    }
    free(R.Pred);

    for (i = 0; i < R.sommets; i++) {
        free(R.FW[i]);
    }
    free(R.FW);

    free(R.col);
}

```

### **//La verification de la connexité du reseau**

```

Reseau colorier_sommet(Reseau R, int sommet, int couleur) {
    int i;
    R.col[sommet] = couleur;
    printf("le sommet %d est colorié \n", sommet);
    for (i = 0; i < R.sommets; i++) {
        if (R.col[i] == 0 && R.col[i] != couleur) {
            R = colorier_sommet(R, i, couleur);
        }
    }
    return R;
}

```

```

Reseau parcours_profondeur(Reseau R) {
    int i, j, couleur, compteur = 0;
    for (i = 0; i < R.sommets; i++) {
        R.col[i] = 0;
    }
    couleur = 1;
    for (j = 0; j < R.sommets; j++) {
        if (R.col[j] == 0) {
            R = colorier_sommet(R, j, couleur);
            couleur++;
        }
    }
}

```

```

printf("\n");
printf("La vérification de la connexité du réseau\n");
for (i = 0; i < R.sommets; i++) {
    printf("%d ", R.col[i]);
    if (R.col[i] == 1) {
        compteur++;
    }
}

```

```

printf("\n");
if (compteur == R.sommets) {
    printf("Le réseau est bien connexe.\n");
} else {
    printf("Le réseau n'est pas connexe \n");
    R = reseau_aleat(R)
    return R;
}

```

## // la détermination de la table de routage de chaque noeud

```

void Floyd_Warshall(Reseau R) {
    int i, j, k;
    for (i = 0; i < R.sommets; i++) {
        for (j = 0; j < R.sommets; j++) {
            R.FW[i][j] = MATRICE;
            R.Pred[i][j] = 0;
            if (R.matrice_adj[i][j] == 1) {
                R.Pred[i][j] = i;
            }
        }
    }
    for (i = 0; i < R.sommets; i++) {
        R.Pred[i][i] = i;
        R.FW[i][i] = 0;
        for (j = 0; j < R.sommets; j++) {
            if (R.matrice_adj[i][j] == 1) {
                R.FW[i][j] = R.temps[i][j];
            }
        }
    }
    for (k = 0; k < R.sommets; k++) {
        for (i = 0; i < R.sommets; i++) {
            for (j = 0; j < R.sommets; j++) {
                if (R.FW[i][k] + R.FW[k][j] < R.FW[i][j]) {
                    R.FW[i][j] = R.FW[i][k] + R.FW[k][j];
                    R.Pred[i][j] = R.Pred[k][j];
                }
            }
        }
    }
}

```

## // afficher la matrice des temps

```

printf("Matrice des temps minimaux :\n");
for (i = 0; i < R.sommets; i++) {
    for (j = 0; j < R.sommets; j++) {
        printf(" %3d ", R.FW[i][j]);
    }
    printf("\n");
}
printf("\n");

```

## // la matrices des prédecceuseur

```

printf("Matrice des prédecesseur:\n");
for (i = 0; i < R.sommets; i++) {

```

```

for (j = 0; j < R.sommets; j++) {
    printf("%3d ", R.Pred[i][j]);
}
printf("\n");
}

```

## La reconstitution de chemin entre deux noeuds

```

void reconstitution_chemin_sommets(Reseau R) {
    printf("Noeud émetteur:");
    scanf("%d", &(R.depart));
    printf("\n");
    printf("Noeud destinataire:");
    scanf("%d", &(R.destination));
    printf("\n");
    int recevoir = R.destination;
    Pile * p = initialiser();
    printf("Le message part de sommet %d ==>, R.depart);
    do {
        R.destination = R.Pred[R.depart][R.destination];
        empiler(p, R.destination);
    }
    while (R.Pred[R.depart][R.destination] != R.depart);
    afficherPile(p, R);

    printf("Le message arrive au sommet %d\n", recevoir);
    printf("\n");
    printf("Le temps pour recevoir le message de %d à %d est :%d\n",
    R.depart, recevoir, R.FW[R.depart][recevoir]);
    p = viderPile(p);
    free(p);
}

int main(int argc, char * argv[]) {
    srand(time(NULL));
    Reseau R;
    R = reseau_aleat(R);
    afficher_reseau(R);
    R = parcours_profondeur(R);
    Floyd_Warshall(R);
    reconstitution_chemin_sommets(R);
    desalloue_reseau(R);
    return 0;
}

```

# Conclusion

Tout au long de la préparation de notre projet, nous avons essayé de mettre en pratique nos connaissances acquises durant ce semestre en théorie des graphes et cela dans le but de réaliser une application sur les tables de routage d'un réseau de 100 nœuds.

L'expérience de ce projet a été pour nous très enrichissante, il était une véritable expérience de travailler en groupe et il nous a permis de bien gérer la répartition des tâches et de renforcer l'esprit de partage des connaissance ainsi que la synchronisation de notre travail

Cependant nous pouvons encore améliorer notre application en ajoutant une interface graphique afin de bien montrer le fruit de notre travail .