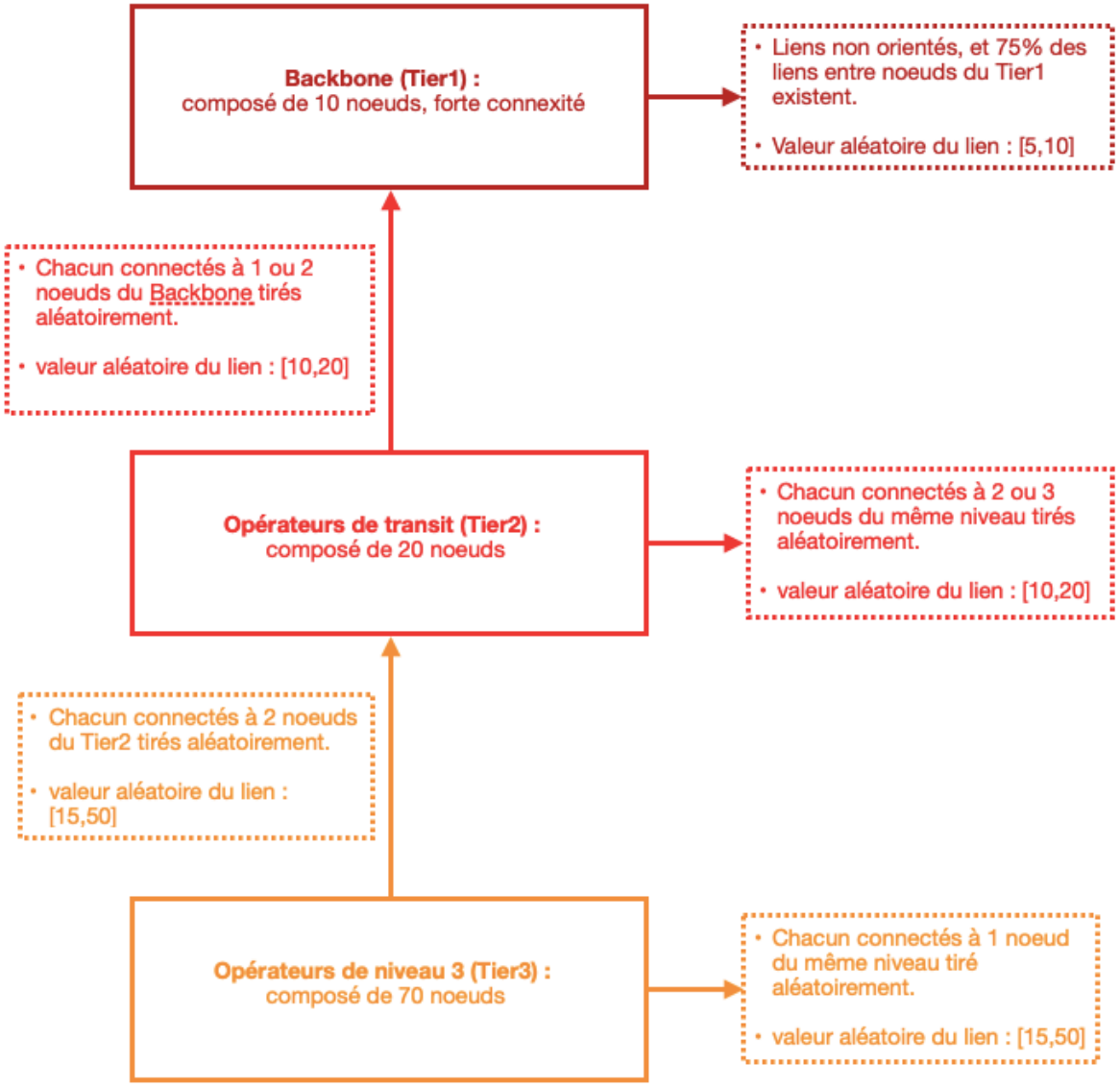
2.1 La création aléatoire d'un réseau réaliste



- **typedef unsigned char uint8_t;**

Ce type sera utilisé pour les entiers.
Un type 'unsigned char' prend en mémoire seulement un octet, ce qui est quatre fois moins que le type pour les entiers natifs 'int'.
De plus, indicateur 'unsigned' est utilisé pour indiquer qu'on ne traitera que des entiers positifs.
Cependant pour calculer les plus courts chemins, on utilisera le type 'short' (2 octets) car un chemin pourrait dépasser 256 ce qui est la limite du type 'unsigned char'.

- **struct noeud**

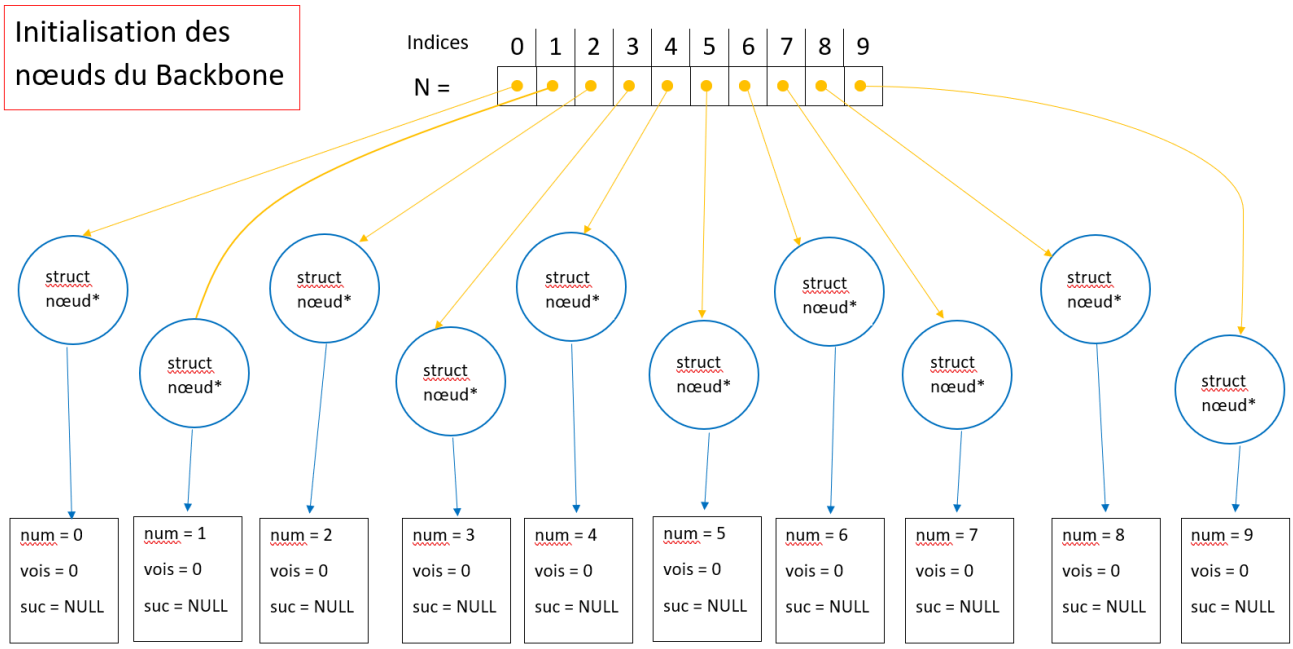
Voici la structure noeud, elle contient :
- Un pointeur vers un tableau de ses successeurs.
- Son nombre de voisins, de type uint8_t car il ne pourra pas dépasser 255
- Son numéro de noeud, qui varie entre 0 et 99
- Une chaîne de caractère qui sert à transporter une chaîne de caractère entre les noeuds

PROJET ALGORITHMIQUE : TABLES DE ROUTAGE

Lirzin Léo, Armour-Lazzari Raphaël, Pouliquen Chloé

- **typedef struct noeud** NOEUD;**

On définit ensuite le type NOEUD comme un double pointeur vers un struct noeud.
Le premier pointeur nous servira à faire un tableau de pointeur.
Chacun de ces (deuxièmes) pointeurs pointe vers un struct noeud où sera stocké le n-ième noeud si le pointeur est le n-ième dans le tableau de pointeur.



- **macros**

#define SIZE 100 -> La macro SIZE définit le nombre de noeuds que l'on souhaite créer
#define T1 10 -> La macro T1 définit le nombre de noeuds présents dans le tier 1.
#define T2 30 -> La macro T2 définit le nombre de noeuds présents dans le tier 2.

Le nombre de noeuds présents dans le tier 3 est obtenu par SIZE auquel on soustrait T1 et T2.

- **uint8_t** init_MA()**

Pour notre projet, on a choisi de représenter les arcs et leur poids à l'aide d'une matrice adjacente. En C, cela se représente par un tableau à deux dimensions.

Dans notre allocation, on a choisi d'utiliser une matrice triangulaire, en effet, on a pris avantage du fait que la graphe soit non-orienté, c'est à dire qu'il existe une liaison de poids w du noeud i vers le noeud j si et seulement si il existe une liaison de même poids w du noeud j vers le noeud i.
De plus, il existe aucun arc entre un noeud et lui-même.
Cette réflexion nous permet d'économiser plusieurs octets, en effet :

Tout allouer :
SIZE*SIZE = SIZE² (dans le cas de SIZE 100 -> 10 000 octets)

Notre allocation :
Somme de 0 à SIZE-1 = SIZE*(SIZE-1) / 2 (dans le cas de SIZE 100 -> 4 950 octets)

Nous avons donc économisé plus de la moitié des octets.

PROJET ALGORITHMIQUE : TABLES DE ROUTAGE

Lirzin Léo, Armour-Lazzari Raphaël, Pouliquen Chloé

- **M[i][j] = -1;**

Sachant que notre type uint8_t est non-signé, donné la valeur -1 aux 'cases' de la matrice adjacente revient à leur donner la plus grande valeur possible, c'est-à-dire 255.
Dans un cas idéal, cette valeur serait infinie mais les limitations matérielles d'un ordinateur nous en empêche. Malgré tout, 255 suffit à imiter l'infini dans notre cas car la valeur la plus grande possible d'un poids d'un arc est 50.

- **NOEUD init_N ()**

On alloue SIZE fois un pointeur sur un struct noeud, ensuite pour chacun de ses pointeurs, on alloue l'espace pour un struct noeud

Au début, aucun n'est connecté.
Ils n'ont donc pas de voisins (N[i]->vois = 0; et N[i]->suc = NULL;)
Cependant, on peut initialiser le numéro des sommets.
Donner i en tant que numéro de sommets assure que le numéro est différent à chaque itération et les numéros se suivent, ce qui se trouve utile dans les fonctions suivantes.

- **void free_MA(uint8_t** M), void free_N (NOEUD N) et void free_Table_Routage()**

Tout espace alloué est ensuite libérer par l'appel de ces fonctions.

- **int suc_in (struct noeud* S, struct noeud* P)**

Cette fonction vérifie si S a dans ses successeurs le sommet P

if ((S->suc[i]) == P) :

Cette condition fonctionne car on compare des adresses. En effet, on stocke les successeurs d'un sommet en copiant l'adresse dans le tableau de successeurs et non un struct noeud indépendant du successeur lui-même dans lequel toutes les valeurs du successeur sont copiées.

- **int suc_tier (struct noeud* N, int Tier)**

Cette fonction renvoie le nombre de voisins d'un noeud N appartenant au Tier T.
Bien sûr, T doit être égal soit à 1, à 2 ou à 3.

- **void assign (uint8_t** M, int i, int j, int val)**

Cette fonction écrit dans la matrice adjacente un arc entre le noeud i et le noeud j de poids val.
La matrice d'adjacence est triangulaire donc il faut savoir dans quelle moitié il faut écrire.

- **void new_vois (NOEUD N, int n, int v)**

Cette fonction assigne deux noeuds en tant que voisins. Ces deux noeuds sont situés à l'indice n et v dans le tableau de pointeurs vers un struct noeud N.

Pour chacun des deux noeuds, on augmente son nombre de voisins. Ensuite, on alloue de la mémoire pour un pointeur vers un struct noeud en plus pour accueillir le nouveau voisin. A la fin, l'un des noeuds a l'autre dans son tableau de voisins.

PROJET ALGORITHMIQUE : TABLES DE ROUTAGE

Lirzin Léo, Armour-Lazzari Raphaël, Pouliquen Chloé

- **void Tier1 (uint8_t** M, NOEUD N)**

Algorithme de Tier 1 :
Entrées : Matrice d'adjacence M et l'ensemble de noeuds du graphe N
Variables locales : i, j, k, l : entiers;

DEBUT
Pour i de 0 à T1-1 faire

Pour j de 0 à T1-1 faire

k <- nombre aléatoire entre 0 et 3;

Si k est supérieur à 0,
le noeud i et le noeud j ne sont pas les mêmes,
le noeud j n'est pas un successeur du noeud i
Alors
l <- nombre aléatoire entre 5 et 10;
Ecrire dans la matrice d'adjacence qu'il existe un arc
entre le noeud i et le noeud j de poids l;
Assigner les noeuds i et j comme voisins de l'un et de l'autre;
Fin Si

Fin Pour
Fin Pour
FIN

Algorithme de Tier 1 : (de manière plus littéraire)
Entrées : Matrice d'adjacence M et l'ensemble de noeuds du graphe N
Variables locales : i, j, k, l : entiers;

DEBUT
Pour tous les noeuds i appartenant au tier 1

Pour tous les noeuds j appartenant au tier 1 tel que i différent de j

k <- nombre aléatoire entre 0 et 3

Si k n'est pas tombé sur 0 (75% de chance) et
si les noeuds i et j ne sont pas déjà connectés
Créer un arc entre ces deux noeuds de valeur aléatoire entre 5 et 10
Fin Si
Fin Pour
Fin Pour
FIN

PROJET ALGORITHMIQUE : TABLES DE ROUTAGE

Lirzin Léo, Armour-Lazzari Raphaël, Pouliquen Chloé

- **void Tier2 (uint8_t** M, NOEUD N)**

Algorithme de Tier 2 :

Entrées : Matrice d'adjacence M et l'ensemble de noeuds du graphe N

Variables locales : i, j, k, l, r1, r2 : entiers;

DEBUT

Pour i de T1 à T2-1 faire

r1 <- nombre aléatoire entre 1 et 2;

Pour j de 0 à r1 faire

k <- nombre aléatoire entre 0 et T1-1;

Si le noeud j n'est pas un successeur du noeud k

Alors

l <- nombre aléatoire entre 10 et 20;

Ecrire dans la matrice d'adjacence qu'il existe un arc

entre le noeud j et le noeud k de poids l;

Assigner les noeuds j et k comme voisins de l'un et de l'autre;

Fin Si

Fin Pour

Tant que le nombre de successeur appartenant au Tier 2 du noeud i

est strictement inférieur à 2

k <- nombre aléatoire entre T1 et T2-1;

Si La noeud i n'est pas le même que le noeud k,

le noeud i n'est pas un successeur du noeud k,

le nombre de successeur du noeud k appartenant au Tier 2 est

strictement inférieur à 3

Alors

l <- nombre aléatoire entre 10 et 20;

Ecrire dans la matrice d'adjacence qu'il existe un arc

entre le noeud i et le noeud k de poids l;

Assigner les noeuds i et k comme voisins de l'un et de l'autre;

Fin Si

Fin Tant que

r2 <- nombre aléatoire entre 0 et 3

Si r2 est égal à 0

k <- nombre aléatoire entre T1 et T2-1;

i La noeud i n'est pas le même que le noeud k,

le noeud i n'est pas un successeur du noeud k,

le nombre de successeurs du noeud k appartenant au Tier 2 est

strictement inférieur à 3,

le nombre de successeurs du noeud i appartenant au Tier 2 est

strictement inférieur à 3

Alors

l <- nombre aléatoire entre 10 et 20;

Ecrire dans la matrice d'adjacence qu'il existe un arc

entre le noeud i et le noeud k de poids l;

Assigner les noeuds i et k comme voisins de l'un et de l'autre;

Fin Si

Fin Pour

FIN

PROJET ALGORITHMIQUE : TABLES DE ROUTAGE

Lirzin Léo, Armour-Lazzari Raphaël, Pouliquen Chloé

Algorithme de Tier 2 : (de manière plus littéraire)
Entrées : Matrice d'adjacence M et l'ensemble de noeuds du graphe N
Variables locales : i, j, k, l, r1, r2 : entiers;

DEBUT
Pour tous les noeuds i appartenant au Tier 2

Créer un ou deux arcs de valeur aléatoire entre 10 et 20 entre le noeud i et un noeud du Tier 1

Créer deux arcs de valeurs aléatoire entre 10 et 20 entre le noeud i et un autre noeud du Tier 2 tel que à tout moment, le nombre de successeurs appartenant au Tier 2 d'un noeud appartenant lui aussi au Tier 2 est au moins égal à 2.

r2 <- nombre aléatoire entre 0 et 3
Si r2 est tombé sur 0 (25% de chance)
On rajoute un arc de valeur aléatoire entre 10 et 20 entre le noeud i et un autre noeud k du Tier 2 tel que les noeuds i et k peuvent accueillir un troisième arc et que cet arc n'existe pas encore.
Fin Si
Fin Pour
FIN

PROJET ALGORITHMIQUE : TABLES DE ROUTAGE

Lirzin Léo, Armour-Lazzari Raphaël, Pouliquen Chloé

- **void Tier3 (uint8_t** M, NOEUD N)**

Algorithme de Tier 3 :
Entrées : Matrice d'adjacence M et l'ensemble de noeuds du graphe N
Variable locales : i, j, k, l : entier;

DEBUT
Pour i de T2 à SIZE-1 faire

Pour j de 0 à 2 faire
k <- nombre aléatoire entre T1 et T2-1;

Si le noeud i n'est pas un successeur du noeud k
Alors
l <- nombre aléatoire entre 15 et 50;
Ecrire dans la matrice d'adjacence qu'il existe un arc entre le noeud i et le noeud k de poids l;
Assigner les noeuds i et k comme voisins de l'un et de l'autre;
Sinon j--;
Fin Si
Fin Pour

Tant que que le noeud i n'a pas de voisins appartenant au tier 3
k <- nombre aléatoire entre T2 et SIZE-1;

Si le noeud i n'est pas le noeud k, et le noeud k n'a pas de voisins dans le tier 3
Alors
l <- nombre aléatoire entre 15 et 50;
Ecrire dans la matrice d'adjacence qu'il existe un arc entre le noeud i et le noeud k de poids l;
Assigner les noeuds i et k comme voisins de l'un et de l'autre;
Fin Si
Fin Tant que
Fin Pour
FIN

Algorithme de Tier 3 : (de manière plus littéraire)
Entrées : Matrice d'adjacence M et l'ensemble de noeuds du graphe N
Variable locales : i, j, k, l : entier;

DEBUT
Pour tous les noeuds i appartenant au tier 3

Créer deux arcs de valeur aléatoire entre 15 et 50 entre le noeud i et un autre noeud du Tier 2

Si le noeud i n'a pas encore de successeur dans le Tier 3
Créer un arc de valeur aléatoire entre 15 et 50 entre le noeud i et un autre noeud k du Tier 3 tel que cet arc n'existe pas encore et que k n'ait pas de successeur dans le Tier 3
Fin Si

Fin Pour
FIN

2.2 La vérification de la connexité du réseau

Un réseau est connexe si et seulement tous les sommets peuvent être atteints à partir d'un sommet quelconque.

- **Méthode utilisée dans cette partie :**

Pour cette partie nous avons décidé de programmer un algorithme de marquage en utilisant un tableau. Chaque noeud sera représenté par une case du tableau.

- **void explorer (struct noeud *N, uint8_t *b)**

Cette fonction sera appelée dans la fonction de parcours en profondeur pour explorer les voisins de chaque noeud.
A chaque appel de cette fonction, la case du tableau, correspondant au numéro du noeud pris en paramètre, prendra la valeur 1.

- **int ParcoursProfondeur(NOEUD N)**

On alloue un tableau de taille SIZE en mémoire pour représenter le passage dans chaque noeud, en effet si l'on a visité le i-ème noeud, la valeur de la i-ème case sera de 1, sinon la valeur de cette case sera égale à 0.

On appelle la fonction explorer, qui associe à la case du tableau, correspondant au numéro du noeud pris en paramètre, la valeur 1, pour montrer que ce noeud à été exploré.
Si ce noeud à des voisins non marqués la fonction explorer sera de nouveau appelée pour modifier la valeur du tableau associée à ses voisins.

Pour finir, on somme chaque valeur de chaque case du tableau et on retourne la valeur de cette somme.

- **int Connexite(NOEUD N)**

Dans cette fonction on récupère la valeur de la somme de la fonction ParcoursProfondeur, et on la compare avec SIZE (le nombre de noeuds).
La fonction retourne le résultat de cette comparaison.
Ainsi, si les deux valeurs sont égales, cela signifie que chaque noeud a été visité, et donc que le réseau est connexe, par conséquent la valeur retournée sera égale à 0.
Sinon, on retourne 1, résultat de la comparaison, et on relance la création d'un nouveau réseau.

2.3 La détermination de la table de routage de chaque noeud

Pour cette partie, nous avons eu besoin des fonctions ci-dessous:

- int Tous_Marque(struct tab* T)
- uint8_t poids(uint8_t** M,uint8_t i,uint8_t j)
- uint8_t renvoie_minimum (struct tab* T, struct Dij* D)
- void ajouter_tableau (struct tab* T, uint8_t new)
- uint8_t** Dijkstra(uint8_t** M,NOEUD N,int source)
- uint8_t*** Tables_Routages(uint8_t** M,NOEUD N)

- **Structure de la pile et fonctions associées**

Nous avons également eu besoin de l'utilisation d'une pile (LIFO) représentée par la structure suivante:

```
struct pile{
        uint8_t Contenu[SIZE];
        uint8_t Taille;
};
```

Ainsi que les fonctions suivantes qui permettent d'empiler un élément et de dépiler un élément:

- void empiler(struct pile* P,uint8_t new)
- uint8_t depiler(struct pile* P)

- **uint8_t*** Tables_Routages(uint8_t** M,NOEUD N)**

Cette fonction est notre fonction principale qui va nous permettre la construction des tables de routages.

La fonction commence par une initialisation de la longueur du tableau à trois dimensions, puis est poursuivit par une boucle 'for' qui va faire un appel à notre fonction **uint8_t** Dijkstra(uint8_t** M,NOEUD N,int source)** qui va remplir la table de routage de chaque noeud et ainsi contenir le plus court chemin du noeud à l'indice i vers chaque autre noeud du graphe.
Notre fonction aura donc une complexité de O(n) fois la complexité de la fonction uint8_t**Dijkstra(uint8_t** M,NOEUD N,int source).

- **struct Dij**

```
struct Dij{
        uint8_t pere;       // Père du i-ième sommet
        short distance;       // distance du i-ième sommet à la source
};
```

- **struct tab**

```
struct tab{
        uint8_t mark[SIZE];         //tableau des sommets marqués
        uint8_t t[SIZE];           //tableau des sommets à traiter
        uint8_t top;              //le nombre de sommets à traiter
};
```

- **void ajouter_tableau (struct tab* T, uint8_t new)**

Cette fonction ajoute un sommet au tableau de sommets à traiter et incrémente le nombre de sommets présents dans le tableau de une unité.

PROJET ALGORITHMIQUE : TABLES DE ROUTAGE

Lirzin Léo, Armour-Lazzari Raphaël, Pouliquen Chloé

- **uint8_t** Dijkstra(uint8_t** M,NOEUD N,int source)**

Cette fonction prend en argument la matrice d'adjacence, le tableau des noeuds de notre réseau ainsi que le numéro du noeud duquel nous souhaitons partir.
Notre fonction va ainsi commencer par une initialisation de la mémoire de notre structure Dij qui va être de la taille du nombre de noeuds de notre graphe. Par exemple dans notre variable struct Dij* D, à l'indice D[0] nous aurons le père du noeud 0 ainsi que sa distance jusqu'à la source.
On procède ensuite par une initialisation de la mémoire de la struct tab T qui contient le tableau des sommets traités, le tableau des sommets à traiter ainsi que le nombre de sommets présents dans le tableau de sommets à traiter:

Nous procédons par la suite, à une initialisation de valeurs de chaque case des tableaux :

-> les cases du tableau de pères sont initialisées à -1 pour signifier qu'aucun noeuds a de pères
-> la distance entre chaque noeud et la source est initialisée à 9999 (pour représenter l'infini)
-> les cases du tableau de sommets traités sont initialisées à 0 pour signifier que tous les sommets sont non traités
-> les cases du tableau des sommets à traiter prennent la valeur de -1

Ensuite on commence par ajouter la valeur de notre sommet source au tableau de sommets à traiter, car évidemment le sommet source va être le premier traité dans notre fonction. L'ajout de la valeur au tableau des sommets à traiter va être fait par la fonction void ajouter_tableau (struct tab* T, uint8_t new)

La boucle principale de cette fonction est la boucle 'while' qui va s'arrêter uniquement lorsque l'ensemble des sommets du graphe sera marqué. En début de boucle 'while' le sommet ayant la plus petite distance est renvoyé et est ainsi marqué à travers la fonction uint8_t renvoie_minimum (struct tab* T, struct Dij* D).

La boucle 'while' va ainsi prendre la valeur du sommet ayant la plus petite distance parmi les sommets présents dans le tableau de sommets à traiter.

On procède par la suite avec une boucle 'for' qui va parcourir tous les sommets voisins du noeud dont le numéro a été précédemment renvoyé. On effectue ensuite une comparaison qui permet de tester si la distance est améliorante ou pas. Si c'est le cas on inclut le sommet current dans notre tableau des sommets pères qui correspondent au parcours du plus court chemin en partant du sommet source.

On termine la boucle 'while' par une condition de vérification qui teste si le sommet voisin du sommet traité a déjà été marqué ou non. Si il est non marqué, on l'ajoute dans le tableau des sommets à traiter.

Le calcul du plus court chemin et de sa distance entre le point source et chaque autres points du graphe sont ainsi calculé dans notre struct Dij* D.

Ensuite, la deuxième partie de notre fonction va d'abord allouer la mémoire dynamique nécessaire. Elle va se poursuivre par une boucle 'for' qui à partir de notre strut Dij* D va remplir la table de routage correspondante à notre 'i'.

On parcourt ainsi chaque sommets représentés par la variable 'i'. Puis dans la boucle 'while (j != source)', on part d'un sommet autre que la source et on remonte par les pères tout en les empilant dans la pile. On alloue par la suite la taille de la table de routage qui a la taille du nombre d'éléments qui va être dépilé.

Dans notre deuxième boucle 'while (T->top != 0)' on dépile tous les sommets que l'on avait empilés dans la pile jusqu'à ce que la pile soit vide.
On libère par la suite nos struct D et struct T, puisque nous n'en avons plus besoin.

algorithme de Dijkstra :

Dijkstra (Matrice adjacence M,graphe N) avec le sommet de départ Sdeb

Initialiser tous les sommets comme étant « non marqués » à 0.
Affecter la valeur infini (9999) à tout D

D(sdeb) = 0 // *La distance entre le sommet de départ et lui même est 0*

Tant qu'il existe un sommet non marqué
Choisir un sommet a non marqué de plus petite valeur D
Marquer a

Pour chacun des sommets b non marqués voisins de a
Si la valeur D du sommet b est plus grande que la somme de la valeur D du sommet a avec la valeur de la distance entre a et b
Alors
La valeur D du sommet b devient la somme de la valeur D du sommet a avec la valeur de la distance entre a et b
Le père du noeud b est le noeud a

Fin Si
Fin Pour
Fin Tant Que

Retourne D tableau des distances minimales de chaque sommet depuis Sdeb, un tableau P qui contient pour chaque sommet le sommet qui le précède dans un plus court chemin de la source à Sdeb.

- **Complexité de notre algorithme de Dijkstra :**

L'algorithme de Dijkstra est donc effectué avec une complexité de $O(n^2)$ car on va parcourir la boucle principale 'while' pour chaque noeud puis on parcourt l'ensemble de ses voisins, on peut grossièrement dire que cette boucle a un coût de n opérations.

On est donc à une complexité total de $O(n^3)$ pour la création des tables de routage car la fonction Dijkstra va être appelé n fois pour déterminer la table de routage de chaque noeud.

Une petite attention à notre fonction poids qui renvoie la valeur de l'arête entre deux sommets. Notre matrice étant représentée sous forme triangulaire car on est dans un graphe non orienté, la valeur de l'arête M[4][3] est la même que celle de M[3][4], la fonction permet donc de renvoyer la valeur correspondante en fonction du cas de figure dans lequel nous sommes.

L'algorithme de Floyd-Warshall possède la même complexité mais il calcule seulement la valeur du plus court chemin et ne permet pas après de lister les sommets à traverser pour obtenir le plus court chemin.

2.4 La reconstitution du chemin entre 2 noeuds

- **Méthode utilisée dans cette partie :**

Pour chaque noeud nous avons déterminé une table de routage avec les fonctions de la partie 2.3, ainsi, dans cette partie nous avons appelé les fonctions de la partie précédente en indiquant le noeud émetteur de message et le noeud destinataire, pour ensuite pouvoir afficher le plus court chemin emprunté par un message, du noeud émetteur jusqu'au noeud destinataire. Aucun calcul n'a été refait, nous avons seulement fait une fonction d'affichage du plus court chemin en question à l'aide de printf.

- **void print_parcours(uint8_t* TR, uint8_t source, uint8_t target)**

Cette fonction affiche le parcours du plus court chemin entre les deux sommets entrés par l'utilisateur.