# Graphes - Composante connexes sur de grands graphes

Bathily Mariame
L3-A
Algorithme avancée

# Sommaire

# 01 Présentation du projet

**1. Objectif Principal:**

- **Analyse de l'efficacité** des structures de données pour le stockage et le parcours de **graphes pondérés de grande taille**.
- Utilisation du **TP3 sur les Graphes** comme base de développement.

**2. Algorithmes Fondamentaux:**

- **Composantes Connexes (CC)** : Identification des sous-ensembles reliés via un **parcours en largeur (BFS)**.
- **Plus court chemin** : Détermination des distances minimales via l'**algorithme de Dijkstra**.

# 02 Analyse des structures de données

**Trois structures implémentées:**

- **Vecteurs de triplets:** Pour générer aléatoirement le graphe (triplets.c)
- **Matrice d'adjacence:** Simple mais occupation mémoire quadratique $O(n^2)$
- **Tableau de Brins:** Utilise trois vecteurs (S, B, W) pour ne stocker que les arêtes réelles, réduisant la complexité spatiale à $O(n + m)$

```c
typedef struct triplet{
    int i;
    int j;
    int poids;
} triplet;
```

```c
typedef struct grapheM{
    int vertices;
    int **adj;
}grapheM;
```

```c
typedef struct {
    int nbs;
    int nba;
    int *S;
    int *B;
    int *W;
} grapheB;
```

# 03 Developpement et implémentation

## A. Gestion et Tableau de Brins (brin.c):

- Fonction convertir_triplets_vers_brin transforme la liste de triplets en format CSR

```c
grapheB* convertir_triplets_vers_brin(grapheT *gt) {
    grapheB *gb = malloc(sizeof(grapheB));
    gb->nbs = gt->nbs;
    gb->nba = gt->nba;
    gb->S = calloc(gt->nbs + 1, sizeof(int));
    gb->B = malloc(gt->nba * sizeof(int));
    gb->W = malloc(gt->nba * sizeof(int));

    // 1. Compter le nombre d'arêtes sortantes par sommet
    for (int k = 0; k < gt->nba; k++) gb->S[gt->aretes[k].i + 1]++;
    // 2. Transformer S en index cumulés
    for (int i = 0; i < gt->nbs; i++) gb->S[i+1] += gb->S[i];
    // 3. Remplir B et W en utilisant un tableau temporaire de positions
    int *pos = malloc(gt->nbs * sizeof(int));

    for(int i=0; i < gt->nbs; i++) pos[i] = gb->S[i];

    for (int k = 0; k < gt->nba; k++) {
        int u = gt->aretes[k].i;
        int p = pos[u]++;
        gb->B[p] = gt->aretes[k].j;
        gb->W[p] = gt->aretes[k].poids;
    }
    free(pos);
    return gb;
}
```

En deux passages seulement, ce code transforme une liste **d'arêtes désordonnée** en un format compact où l'on sait exactement **où trouver les voisins de n'importe quel sommet.**

Passage 1 : **Réserver la place exacte** pour chaque sommet (évite le gaspillage)

Passage 2 : **Ranger les voisins** aux bonnes adresses

# 03 Developpement et implémentation

```c
void CC_mat(grapheM *grph) {
    int n = grph->vertices; // Récupère le nombre de sommets
    int *composante = calloc(n, sizeof(int)); // Tableau pour stocker le numéro de CC de chaque sommet (0 = non visité)
    int num_cc = 0;

    for (int i = 0; i < n; i++) {
        // Si le sommet i n'appartient encore à aucune composante
        if (composante[i] == 0) {
            num_cc++; // On a trouvé une nouvelle composante

            // Initialisation de la file pour le parcours BFS
            fifo *file = new_fifo();
            push(file, i);
            composante[i] = num_cc;

            while (!is_empty_fifo(file)) {
                int u = get_first(file); // Récupère le sommet en tête
                enqueue(file); // Retire le sommet de la file

                // On explore tous les voisins possibles dans la matrice
                for (int v = 0; v < n; v++) {
                    // Si une arête existe (poids > 0) et que le voisin n'est pas visité
                    if (grph->adj[u][v] > 0 && composante[v] == 0) {
                        composante[v] = num_cc;
                        push(file, v); // Ajoute le voisin à la file pour exploration future
                    }
                }
            }
        }
    }

    printf("\nNombre total de composantes connexes (Matrice) : %d\n", num_cc);
    free(composante);
}
```

## B. Composants connexes (CC_mat et CC_brin):

- CC_mat: Chaque sommet scan sa ligne (n colonnes).
- CC_brin: L'algorithme accède directement aux voisins via le tableau d'index S (évite les tests inutiles sur des cases vides)

```c
void CC_brin(grapheB *g) {
    int *marked = calloc(g->nbs, sizeof(int));
    int count = 0;
    fifo *q = new_fifo();
    for (int i = 0; i < g->nbs; i++) {
        if (!marked[i]) {
            count++;
            push(q, i); marked[i] = 1;
            while (!is_empty_fifo(q)) {
                int u = get_first(q); enqueue(q);
                // On ne parcourt QUE les voisins réels, pas toute la matrice
                for (int j = g->S[u]; j < g->S[u+1]; j++) {
                    int v = g->B[j];
                    if (!marked[v]) { marked[v] = 1; push(q, v); }
                }
            }
        }
    }
    printf("Nombre de composantes connexes (Brin) : %d\n", count);
    free(marked);
}
```

**Le principe des deux codes :**

**Point commun** : Les deux utilisent un **parcours en largeur (BFS)** avec une file FIFO pour explorer le graphe étape par étape.

**Différence dans CC_mat** : À chaque sommet, l'ordinateur fait une boucle de 0 à n (le nombre total de sommets). S'il y a 20000 sommets, il fait 20000 tests, même si le sommet n'a qu'un seul voisin.

**Différence dans CC_brin** : L'ordinateur utilise l'index **S** pour aller directement aux voisins. S'il y a 3 voisins, il ne fait que 3 itérations.

# 03 Developpement et implémentation

**C. Plus court chemin avec Tas binaire (heap.c):**

- Couplage de Dijkstra +Tas Binaire (Min-Heap): permet d'extraire le sommet à distance minimale en O(log n) au lieu de O(n) (baisse la complexité totale à O(m log n)

```c
void dijkstra_brin(grapheB *g, int src) {
    int n = g->nbs;
    int *dist = malloc(n * sizeof(int));
    Tas *tas = creer_tas(n);

    for (int i = 0; i < n; i++) {
        dist[i] = (i == src) ? 0 : INF;
        tas->array[i] = new_noeudTas(i, dist[i]);
        tas->pos[i] = i;
    }
    tas->taille = n;
    decrease_value(tas, src, 0);

    while (!is_empty_tas(tas)) {
        noeudTas *minNode = extract_min(tas);
        int u = minNode->v;
        for (int j = g->S[u]; j < g->S[u+1]; j++) {
            int v = g->B[j];
            int w = g->W[j];
            if (is_in_tas(tas, v) && dist[u] != INF && dist[u] + w < dist[v]) {
                dist[v] = dist[u] + w;
                decrease_value(tas, v, dist[v]);
            }
        }
    }
    free(dist);
}
```

**Le principe du code:**

1. **Initialisation** : On met toutes les distances à l'infini (INF), sauf pour le sommet de départ qui est à 0. On remplit le **Tas Binaire** avec tous les sommets.
2. **La Boucle while** : Tant que le Tas n'est pas vide, on demande au Tas : "*le sommet qui a la plus petite distance actuelle*" via la fonction extract_min.
3. **La mise à jour** : Pour chaque voisin du sommet extrait, on regarde si passer par lui raccourcit le chemin. Si oui, on met à jour sa distance et on prévient le Tas avec decrease_value pour qu'il se réorganise.

# 04 Tests et benchmarks

**main.c (Section 4):** exécute les tests sur des graphes de **10 000**, **15000** et **20 000 nœuds** avec une densité d'arêtes de 1%.

**On mesurera:**

1. L'**occupation mémoire** calculée en Mo.

2. Le **temps CPU réel** via la fonction clock().

# 05 Analyse des résultats

```
>>> TEST POUR N = 10000 SOMMETS (Densité: 1.0%)
------------------------------------------------------------
[Mémoire] Matrice: 381.47 Mo | Brin: 7.67 Mo
[CC] Calcul en cours...

Nombre total de composantes connexes (Matrice) : 1
  > Temps CC Matrice : 0.1378 s
Nombre de composantes connexes (Brin) : 1
  > Temps CC Brin    : 0.0026 s

[Dijkstra] Calcul depuis le sommet 0...
  > Temps Dijkstra Matrice : 0.6292 s
  > Temps Dijkstra Brin    : 0.0102 s
```

```
>>> TEST POUR N = 15000 SOMMETS (Densité: 1.0%)
------------------------------------------------------------
[Mémoire] Matrice: 858.31 Mo | Brin: 17.20 Mo
[CC] Calcul en cours...

Nombre total de composantes connexes (Matrice) : 1
  > Temps CC Matrice : 0.3249 s
Nombre de composantes connexes (Brin) : 1
  > Temps CC Brin    : 0.0054 s

[Dijkstra] Calcul depuis le sommet 0...
  > Temps Dijkstra Matrice : 1.5514 s
  > Temps Dijkstra Brin    : 0.0211 s
```

```
>>> TEST POUR N = 20000 SOMMETS (Densité: 1.0%)
------------------------------------------------------------
[Mémoire] Matrice: 1525.88 Mo | Brin: 30.59 Mo
[CC] Calcul en cours...

Nombre total de composantes connexes (Matrice) : 1
  > Temps CC Matrice : 0.5556 s
Nombre de composantes connexes (Brin) : 1
  > Temps CC Brin    : 0.0090 s

[Dijkstra] Calcul depuis le sommet 0...
  > Temps Dijkstra Matrice : 2.7986 s
  > Temps Dijkstra Brin    : 0.0355 s
```

# MERCI !