

REPUBLIQUE DU CAMEROUN

Paix-travail-patrie

Ministère de l'enseignement  
supérieur

UNIVERSITE DE DSCHANG

-----



REPUBLIC OF CAMEROUN

Peace-work-fatherland

Ministry of higher Education

UNIVERSITY OF DSCHANG

-----

**FACULTE DE SCIENCES**

# DEVOIR DE COMPLEXITE ET ALGORITHME AVANCEE

**Etudiante :**

*DZIKANG VANELLA LIDELLE*

CM-UDS-20SCI2094

**Enseignant :**

**Pr. KENGNE Vianney**

**ANNEE SCOLAIRE**

**2023/2024**

## SOMMAIRE

### Table des matières

<b>I. PLUS COURT CHEMIN DANS UN GRAPHE</b> .....	3
1. Algorithme de Dijkstra .....	3
a. Principe .....	3
b. Implémentation en langage C (Annexe – Dijkstra) .....	3
2. Version parallélisé de cet algorithme avec Open MP .....	4
1) Définitions .....	4
2) Implémentation en C (Annexe – DijkstraOpenMp) .....	4
<b>II. ARBRES ROUGES ET NOIRS</b> .....	5
1) Présentation des concepts des arbres rouges et noirs .....	5
2) Implémentation des opérations de base en langage C (Annexe – ARN) .....	5
a) Rotations : .....	5
b) Recherche : .....	6
c) Insertion : .....	6
d) Suppression : .....	7
<b>III. ARBRES COUVRANTS DE POIDS MINIMUM</b> .....	7
1. Définition .....	7
2. Algorithme de Kruskal (Annexe – Kruskal) .....	8
3. Algorithme de Prim (Annexe – Prim) .....	9
<b>IV. SUITE DE FIBONACCI AVEC PROGRAMMATION DYNAMIQUE</b> .....	10
i. Présentation de la suite de Fibonacci .....	10
ii. Concepts de la programmation dynamique .....	11
iii. Implémentation en langage C .....	11
A. Approche par Tabulation (Annexe – FiboTab) .....	11
B. Approche par Mémorisation (Annexe – FiboMemo) .....	12

## INTRODUCTION

L'objectif de ce devoir est d'explorer divers aspects des algorithmes en informatique, avec un accent particulier sur l'optimisation de l'algorithme de Dijkstra parallèle en utilisant OpenMP, la mise en œuvre des opérations fondamentales des arbres rouges et noirs, la compréhension et l'implémentation des algorithmes de Kruskal et de Prim pour trouver un arbre couvrant de poids minimum, ainsi que l'application de la programmation dynamique pour calculer la suite de Fibonacci. Chacun de ces domaines offre une perspective unique sur la conception et l'efficacité des algorithmes, mettant en évidence l'importance de leur compréhension dans le domaine de l'informatique.

## I. PLUS COURT CHEMIN DANS UN GRAPHE

### 1. Algorithme de Dijkstra

En théorie des graphes, l'algorithme de Dijkstra sert à résoudre le problème du plus court chemin. Il permet, par exemple, de déterminer un plus court chemin pour se rendre d'une ville à une autre connaissant le réseau routier d'une région. Plus précisément, il calcule des plus courts chemins à partir d'une source vers tous les autres sommets dans un graphe orienté pondéré par des réels positifs. On peut aussi l'utiliser pour calculer un plus court chemin entre un sommet de départ et un sommet d'arrivée.

#### a. Principe

L'algorithme prend en entrée un graphe orienté pondéré par des réels positifs et un sommet source. Il s'agit de construire progressivement un sous-graphe dans lequel sont classés les différents sommets par ordre croissant de leur distance minimale au sommet de départ. La distance correspond à la somme des poids des arcs empruntés.

Au départ, on considère que les distances de chaque sommet au sommet de départ sont infinies, sauf pour le sommet de départ pour lequel la distance est nulle. Le sous-graphe de départ est l'ensemble vide.

Au cours de chaque itération, on choisit en dehors du sous-graphe un sommet de distance minimale et on l'ajoute au sous-graphe. Ensuite, on met à jour les distances des sommets voisins de celui ajouté. La mise à jour s'opère comme suit : la nouvelle distance du sommet voisin est le minimum entre la distance existante et celle obtenue en ajoutant le poids de l'arc entre sommet voisin et sommet ajouté à la distance du sommet ajouté.

On continue ainsi jusqu'à épuisement des sommets (ou jusqu'à sélection du sommet d'arrivée)

#### b. Implémentation en langage C (Annexe – Dijkstra)

Le programme prend comme entrée, le nombre de nœuds du graph, ainsi que le nombre d'arête et pour chaque arête il récupère le nœud initial. A partir de cela on construit la matrice d'adjacence

Il récupère également le sommet source et le sommet destination pour pouvoir retourner le plus court chemin entre les deux nœuds

Voici un cas d'exécution :

```

0. Quitter
Choisissez un algorithme (0-4) : 1
Entrez le nombre de sommets du graphe: 4
Entrez le nombre d'aretes du graphe: 5
Entrez les aretes du graphe avec les poids (debut fin poids):
Arete 1: 0
1
4
Arete 2: 1
3
3
Arete 3: 3
2
2
Arete 4: 1
2
1
Arete 5: 0
2
6
Entrez le sommet source: 0
Entrez le sommet destination: 3
Voici la matrice d'adjacence du graphe construit a partir des aretes que vous avez fourn
is:
0      4      6      0
4      0      1      3
6      1      0      1
0      3      1      0
Sommet  Distance depuis la source  Chemin
0      0
1      4
2      5
3      6
Distance minimale de 0 à 3 : 6

```

## 2. Version parallélisé de cet algorithme avec Open MP

### 1) Définitions

**La parallélisation** est la technique qui consiste à diviser une grande tâche informatique en sous-tâches de petite taille, pouvant être exécutées simultanément sur plusieurs processeurs ou cœurs, dans le but de réduire le temps de calcul global. C'est un concept important en informatique, car il permet un traitement plus rapide et plus efficace de grands volumes de données.

**Open MP (Open Multi-Processing)** est une API (Interface de Programmation d'Application) qui prend en charge la programmation multi-thread sur des architectures parallèles partagées. Elle permet aux développeurs de créer des applications parallèles en utilisant des directives de compilateur spécifiques et des bibliothèques pour exploiter efficacement les ressources des systèmes multicœurs

### 2) Implémentation en C (Annexe – DijkstraOpenMp)

## II. ARBRES ROUGES ET NOIRS

### 1) Présentation des concepts des arbres rouges et noirs

Les *arbres rouges et noirs* sont une variante des arbres binaires de recherche. Leur intérêt principal est qu'ils sont relativement équilibrés. On considère ici des arbres binaires de recherche où les valeurs sont portées par les nœuds internes. Les feuilles ne sont pas prises en compte dans la hauteur de l'arbre.

En plus des restrictions imposées aux arbres binaires de recherche, les règles suivantes sont utilisées :

- a. Un nœud est soit rouge soit noir ;
- b. La Racine est noire;
- c. Les enfants d'un nœud rouge sont noirs ;
- d. Les feuilles **NIL** sont noires
- e. Le chemin de la racine à n'importe quelle feuille (**NIL**) contient le même nombre de nœuds noirs. On peut appeler ce nombre de nœuds noirs la **hauteur noire**.

### 2) Implémentation des opérations de base en langage C (Annexe – ARN)

#### a) Rotations :

Les rotations sont des modifications locales d'un arbre binaire. Elles consistent à échanger un nœud avec l'un de ses fils. Dans la rotation droite, un nœud devient le fils droit du nœud qui était son fils gauche. Dans la rotation gauche, un nœud devient le fils gauche du nœud qui était son fils droit. Les rotations gauche et droite sont inverses l'une de l'autre. Elles sont utilisées pour maintenir les propriétés de l'arbre après une insertion ou une suppression d'un nœud.

```
Arbre après insertion :
Niveau 2 : 1.NOIR Niveau 1 : 2.NOIR Niveau 3 : 3.NOIR Niveau 2 : 4.ROUGE Niveau 4 : 5.ROUGE Niveau 3 : 6.NOIR
Menu :
1. Insertion
2. Suppression
3. Recherche
4. Rotation à gauche
5. Rotation à droite
6. Quitter
Choix : 4
Entrez la clé du nœud sur lequel vous voulez roter : 2
nœud objet de recherche
2, 0
Affichage de l'arbre après rotation à gauche sur le nœud de clé 2
Niveau 3 : 1.NOIR Niveau 2 : 2.NOIR Niveau 3 : 3.NOIR Niveau 1 : 4.ROUGE Niveau 3 : 5.ROUGE Niveau 2 : 6.NOIR
Menu :
1. Insertion
2. Suppression
3. Recherche
4. Rotation à gauche
5. Rotation à droite
6. Quitter
Choix : 5
Entrez la clé du nœud sur lequel vous voulez roter : 4
nœud objet de recherche
4, 1
Affichage de l'arbre après rotation à droite sur le nœud de clé 4
Niveau 2 : 1.NOIR Niveau 1 : 2.NOIR Niveau 3 : 3.NOIR Niveau 2 : 4.ROUGE Niveau 4 : 5.ROUGE Niveau 3 : 6.NOIR
Menu :
```

## b) Recherche :

La recherche d'un élément se déroule de la même façon que pour un arbre binaire de recherche : en partant de la racine, on compare la valeur recherchée à celle du nœud courant de l'arbre. Si ces valeurs sont égales, la recherche est terminée et on renvoie le nœud courant. Sinon, on choisit de descendre vers le nœud enfant gauche ou droit selon que la valeur recherchée est inférieure ou supérieure. Si une feuille est atteinte, la valeur recherchée ne se trouve pas dans l'arbre.

```
Arbre après insertion :
Niveau 2 : 2.NOIR Niveau 1 : 8.NOIR Niveau 3 : 10.ROUGE Niveau 2 : 15.NOIR Niveau 3 : 20.ROUGE
Menu :
1. Insertion
2. Suppression
3. Recherche
4. Rotation à gauche
5. Rotation à droite
6. Quitter
Choix : 3
Entrez la clé à rechercher : 10
La clé 10 a été trouvée dans l'arbre.
Voici les informations du noeuds en question
Cle: 10,
Couleur: 1,
Parent: 15,
Fils Droit: (null)
Fils Gauche: (null)
```

## c) Insertion :

Pour insérer une nouvelle valeur dans un arbre, on commence par effectuer une insertion aux feuilles, le nouveau nœud inséré étant de couleur rouge pour respecter la règle (e). Cependant, la règle (c) peut, elle, ne plus être vérifiée, et nous allons corriger cela en faisant « remonter » l'éventuel conflit jusqu'à la racine, pour le faire finalement disparaître

Illustrons avec des cas :

```
Menu :
1. Dijkstra - Plus court chemin dans un graphe
2. Fibonacci (Programmation dynamique)
3. Kruskal - Arbre couvrant de poids minimum
4. ARN - Arbre rouge noir
5. Prime - Arbre couvrant de poids minimum
0. Quitter
Choisissez un algorithme (0-4) : 4
Menu :
1. Insertion
2. Suppression
3. Recherche
4. Rotation à gauche
5. Rotation à droite
6. Quitter
Choix : 1
Entrez le Nombre de noeuds de votre arbre : 1
Entrez la clé du nœud 1 : 5
Arbre après insertion :
Niveau 1 : 5.NOIR
Menu :
```

Insertion de plusieurs nœuds



```

Menu :
1. Dijkstra - Plus court chemin dans un graphe
2. Fibonacci (Programmation dynamique)
3. Kruskal - Arbre couvrant de poids minimum
4. ARN - Arbre rouge noir
5. Prime - Arbre couvrant de poids minimum
0. Quitter
Choisissez un algorithme (0-4) : 4

Menu :
1. Insertion
2. Suppression
3. Recherche
4. Rotation à gauche
5. Rotation à droite
6. Quitter
Choix : 1
Entrez le Nombre de noeuds de votre arbre : 5
Entrez la clé du nœud 1 : 2
Entrez la clé du nœud 2 : 8
Entrez la clé du nœud 3 : 15
Entrez la clé du nœud 4 : 10
Entrez la clé du nœud 5 : 20
Arbre après insertion :
Niveau 2 : 2.NOIR Niveau 1 : 8.NOIR Niveau 3 : 10.ROUGE Niveau 2 : 15.NOIR Niveau 3 : 20.ROUGE

```

#### d) Suppression :

La suppression commence par une recherche du nœud à supprimer, comme dans un arbre binaire classique. Après suppression du nœud, une opération de réparation de l'arbre est effectuée pour s'assurer que les propriétés des arbres rouge-noirs soient respectés grâce à des fonctions tel que la rotation gauche et droite.

```

6. Quitter
Choix : 1
Entrez le Nombre de noeuds de votre arbre : 3
Entrez la clé du nœud 1 : 8
Entrez la clé du nœud 2 : 5
Entrez la clé du nœud 3 : 3
Arbre après insertion :
Niveau 2 : 3.ROUGE Niveau 1 : 5.NOIR Niveau 2 : 8.ROUGE
Menu :
1. Insertion
2. Suppression
3. Recherche
4. Rotation à gauche
5. Rotation à droite
6. Quitter
Choix : 2
Entrez la clé à supprimer : 3
Arbre après suppression :
Niveau 2 : -1.NOIR Niveau 1 : 5.NOIR Niveau 2 : 8.ROUGE
Menu :

```

### III. ARBRES COUVRANTS DE POIDS MINIMUM

#### 1. Définition

En théorie des graphes, étant donné un graphe non orienté connexe dont les arêtes sont pondérées, un arbre couvrant de poids minimal (ACM), arbre couvrant minimum ou arbre sous-tendant minimum de ce graphe est un arbre couvrant (sous-ensemble qui est un arbre et qui connecte tous les sommets ensemble) dont la somme des poids des

arêtes est minimale (c'est-à-dire de poids inférieur ou égal à celui de tous les autres arbres couvrants du graphe).

Plus formellement, soit  $G$  un graphe pondéré avec  $V$  sommets et  $E$  arêtes, chaque arête ayant un poids associé. Un arbre couvrant de poids minimum  $T$  pour  $G$  est un arbre qui satisfait les conditions suivantes :

1. Arbre Couvrant :  $T$  est un arbre, ce qui signifie qu'il est connexe et ne contient pas de cycles.
2. Couverture des Sommets : Tous les sommets du graphe  $G$  sont inclus dans  $T$ .
3. Poids Minimum : La somme des poids des arêtes de  $T$  est minimale parmi tous les arbres couvrants de  $G$ .

Les algorithmes classiques pour trouver un arbre couvrant de poids minimum comprennent l'algorithme de Kruskal et l'algorithme de Prim. Ces algorithmes garantissent de trouver un arbre couvrant de poids minimum pour un graphe pondéré donné.

## **2. Algorithme de Kruskal (Annexe – Kruskal)**

L'algorithme de Kruskal est un algorithme de recherche d'arbre recouvrant de poids minimum (ARPM) ou arbre couvrant minimum (ACM) dans un graphe connexe non-orienté et pondéré. L'algorithme construit un arbre couvrant minimum en sélectionnant des arêtes par poids croissant. Plus précisément, l'algorithme considère toutes les arêtes du graphe par poids croissant (en pratique, on trie d'abord les arêtes du graphe par poids croissant) et pour chacune d'elles, il la sélectionne si elle ne crée pas un cycle.

### **Principe :**

Pour trouver l'arbre recouvrant de poids minimum, l'algorithme de Kruskal prévoit les étapes ci-dessous :

1. Trier les arêtes ( $a$ ) du graphique ( $G$ ) par poids croissant. Ce faisant, les arêtes les plus légères sont examinées en premières.
2. Partir d'un arbre ( $T$ ) vide. C'est-à-dire un arbre qui ne contient aucune arête ; son poids est égal à 0. Sa construction est progressive en suivant les étapes suivantes.
3. Sélectionner les arêtes par poids croissant. Il faut donc commencer par les arêtes les plus légères. Si l'ajout d'une arête n'implique pas la création d'un cycle dans le graphe, il convient de l'ajouter à l'arbre.
4. Répétez l'opération jusqu'à ce que tous les sommets ( $S$ ) soient reliés. Et ce, sans créer de cycle dans l'arbre couvrant de poids minimal.

### **Implémentation en C :**

Le programme prend comme entrée, le nombre de nœuds du graph, ainsi que le nombre d'arête et pour chaque arête il récupère le nœud initial. A partir de cela on construit la matrice d'adjacence. Et retourne en sortie l'arbre couvrant de poids minimum

Illustration pour cas :



```

1. Dijkstra - Plus court chemin dans un graphe
2. Fibonacci (Programmation dynamique)
3. Kruskal - Arbre couvrant de poids minimum
4. ARN - Arbre rouge noir
5. Prime - Arbre couvrant de poids minimum
0. Quitter
Choisissez un algorithme (0-4) : 3
Entrer le nombre d'arêtes: 5

Entrer les arêtes (start end weight):
Arêtes 1: 0
1
4
Arêtes 2: 1
3
3 CB Volume
Arêtes 3: 3
2
1
Arêtes 4: 1
2
1
Arêtes 5: 0
2
6
Voici les arêtes dans l'ACM construit
Arête (3, 2) de poids 1
Arête (1, 2) de poids 1
Arête (0, 1) de poids 4
Cout Minimum de l'Arbre Couvrant : 6

```

### 3. Algorithme de Prim (Annexe – Prim)

L'algorithme de Prim est un algorithme glouton qui calcule un arbre couvrant minimal dans un graphe connexe pondéré et non orienté. En d'autres termes, cet algorithme trouve un sous-ensemble d'arêtes formant un arbre sur l'ensemble des sommets du graphe initial et tel que la somme des poids de ces arêtes soit minimale. Si le graphe n'est pas connexe, alors l'algorithme détermine un arbre couvrant minimal d'une composante connexe du graphe.

#### Principe :

L'algorithme consiste à faire croître un arbre depuis un sommet. On part d'un sous-ensemble contenant un sommet unique. À chaque itération, on agrandit ce sous-ensemble en prenant l'arête incidente à ce sous-ensemble de coût minimum. En effet, si l'on prend une arête dont les deux extrémités appartiennent déjà à l'arbre, l'ajout de cette arête créerait un deuxième chemin entre les deux sommets dans l'arbre en cours de construction et le résultat contiendrait un cycle.

L'étape d'initialisation consiste à choisir au hasard un sommet. Au bout de la première étape, on se retrouve ainsi avec un arbre contenant 1 sommet et 0 arête. Ensuite, on construit récursivement l'arbre minimal de la façon suivante : à l'étape  $n$ , ayant déjà construit un arbre contenant  $n$  sommets et  $n-1$  arêtes, on établit la liste de toutes les arêtes liant un sommet de l'arbre à un sommet qui n'est pas sur l'arbre. On choisit alors une arête de poids minimal, que l'on rajoute à l'arbre ; l'arbre contient à présent  $n+1$  sommets et  $n$  arêtes. L'algorithme se termine lorsque tous les sommets du graphe sont contenus dans l'arbre.

## Implémentation en C :

Le programme prend comme entrée, le nombre de nœuds du graph, ainsi que le nombre d'arête et pour chaque arête il récupère le nœud initial. A partir de cela on construit la matrice d'adjacence. Et retourne en sortie l'arbre couvrant de poids minimum

## Illustration :



```
3. Kruskal - Arbre couvrant de poids minimum
4. ARN - Arbre rouge noir
5. Prime - Arbre couvrant de poids minimum
0. Quitter
Choisissez un algorithme (0-4) : 5
Entrez le nombre de sommets du graphe: 4
Entrez le nombre d'aretes du graphe: 5
Entrez les aretes du graphe avec les poids (debut fin poids):
Arete 1: 0
1
4
Arete 2: 1
3
Arete 3: 3
2
Arete 4: 1
2
Arete 5: 0
2
6
Voici la matrice d'adjacence du graphe construit a partir des aretes que vous avez fourn
is:
0      4      6      0
4      0      1      3
6      1      0      1
0      3      1      0
Voici les aretes contenues dans l'ACM
Arêtes  Poids
(0 , 1)      4
(1 , 2)      1
(2 , 3)      1
Au revoir !
```

## IV. SUITE DE FIBONACCI AVEC PROGRAMMATION DYNAMIQUE

### i. Présentation de la suite de Fibonacci

La suite de Fibonacci est une suite de nombres entiers dans laquelle chaque nombre est la somme des deux nombres qui le précèdent. Elle commence par les nombres 0 et 1 puis se poursuit avec 1 (comme somme de 0 et 1), 2 (comme somme de 1 et 1), 3 (comme somme de 1 et 2), 5 (comme somme de 2 et 3), 8 (comme somme de 3 et 5), etc. Les termes de cette suite, i.e. les nombres apparaissant dans cette suite, sont appelés nombres de Fibonacci.

De façon plus formelle,  $(F_n)_{n \in \mathbb{N}}$  est définie par  $F_0 = 0$ ,  $F_1 = 1$ , et la relation de récurrence  $F_n = F_{n-1} + F_{n-2}$  pour  $n \geq 2$

## ii. Concepts de la programmation dynamique

La programmation dynamique est une méthode algorithmique utilisée pour résoudre des problèmes d'optimisation, en particulier ceux qui présentent une structure de sous-problèmes imbriqués ou une nature récursive. Elle s'appuie sur la décomposition d'un problème complexe en sous-problèmes plus petits, en stockant les solutions intermédiaires pour éviter de les recalculer à chaque étape. La programmation dynamique suit deux approches principales :

**Top-down (ou "Mémorisation") :** Cette approche commence par résoudre le problème initial et s'appuie sur la résolution récursive des sous-problèmes. Elle utilise une structure de données (comme un tableau ou une carte) pour stocker les résultats des sous-problèmes déjà résolus, évitant ainsi les calculs redondants.

**Bottom-up (ou "Tabulation") :** Cette approche consiste à résoudre d'abord les sous-problèmes les plus simples et à progresser vers la résolution du problème initial. Elle construit une table de solutions aux sous-problèmes en ordre croissant de complexité, en utilisant les résultats précédents pour résoudre les problèmes suivants.

Le problème de la suite de Fibonacci. Lors de l'utilisation de la récursion simple, le même calcul est effectué plusieurs fois pour des valeurs répétées. En utilisant la mémorisation, on peut stocker les résultats de la suite de Fibonacci déjà calculés dans un dictionnaire ou un tableau, de sorte que les appels récursifs suivants puissent utiliser ces valeurs sans avoir à les recalculer.

## iii. Implémentation en langage C

Explorons maintenant l'implémentation concrète de la suite de Fibonacci en langage C, en mettant en lumière les avantages de la programmation dynamique pour résoudre ce problème classique.

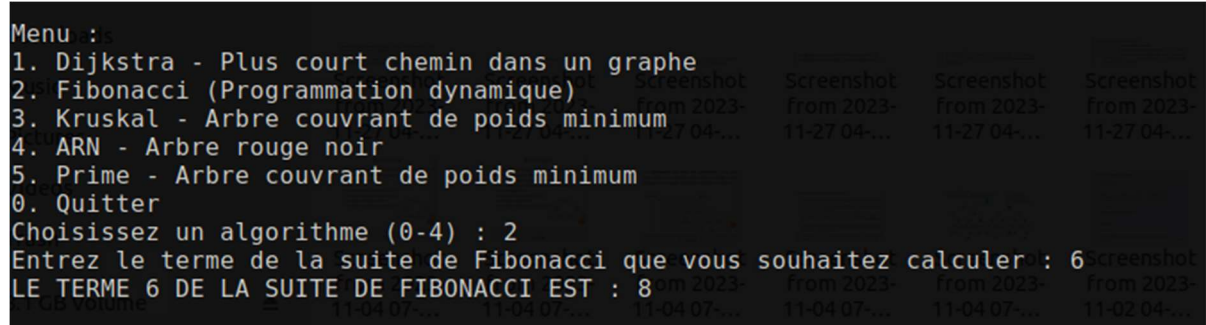
### A. Approche par Tabulation (Annexe – FiboTab)

Une implémentation dynamique de la suite de Fibonacci consiste à utiliser une table de taille  $n+1$  pour stocker les premiers  $n$  termes de la suite. La valeur du  $n$ -ième terme de la suite peut alors être calculée en un temps constant, en consultant la table.

```
sunshine@sunshine-Latitude-E6520:~/Documents/Vanella/projects/Cprojects/Algo$ ./main
Menu :
1. Dijkstra - Plus court chemin dans un graphe
2. Fibonacci (Programmation dynamique)
3. Kruskal - Arbre couvrant de poids minimum
4. ARN - Arbre rouge noir
5. Prime - Arbre couvrant de poids minimum
0. Quitter
Choisissez un algorithme (0-4) : 2
Entrez le terme de la suite de Fibonacci que vous souhaitez calculer : 19
LE TERME 19 DE LA SUITE DE FIBONACCI EST : 4181
```

## B. Approche par Mémorisation (Annexe – FiboMemo)

Cette implémentation fonctionne de la même manière que l'implémentation dynamique précédente, sauf qu'elle vérifie d'abord si la valeur du n-ième terme est déjà stockée dans la table. Si c'est le cas, la valeur est simplement renvoyée. Sinon, le terme est calculé et stocké dans la table de cache avant d'être renvoyé.



```
Menu :
1. Dijkstra - Plus court chemin dans un graphe
2. Fibonacci (Programmation dynamique)
3. Kruskal - Arbre couvrant de poids minimum
4. ARN - Arbre rouge noir
5. Prime - Arbre couvrant de poids minimum
0. Quitter
Choisissez un algorithme (0-4) : 2
Entrez le terme de la suite de Fibonacci que vous souhaitez calculer : 6
LE TERME 6 DE LA SUITE DE FIBONACCI EST : 8
```

**NB :** pour tous les tests à faire sur les graphes il faut numéroté les sommets à partir de 0

L'exécutable se trouve en annexe : Annexe - Exécutable

## CONCLUSION

En conclusion, ce devoir a permis d'explorer un large éventail d'aspects des algorithmes, démontrant la diversité des approches nécessaires pour résoudre différents types de problèmes. L'implémentation de l'algorithme de Dijkstra en parallèle avec OpenMP souligne l'importance de tirer parti des ressources disponibles pour accélérer les calculs, tandis que la compréhension des arbres rouges et noirs offre un aperçu des structures de données complexes. Les algorithmes de Kruskal et de Prim pour les arbres couvrants de poids minimum illustrent les défis liés à la recherche d'optimalité dans des ensembles de données variés. Enfin, l'application de la programmation dynamique pour résoudre la suite de Fibonacci souligne l'efficacité des approches récursives avec mémorisation. En somme, ce devoir souligne l'importance cruciale de la conception d'algorithmes dans le domaine de l'informatique, en mettant en lumière les différentes stratégies nécessaires pour résoudre des problèmes algorithmiques complexes.

## I. BIBLIOGRAPHIE

- ❖ [https://github.com/jfmartinez/course\\_project\\_dijkstras\\_parallel/blob/master/Dijkstra-OpenMP.c](https://github.com/jfmartinez/course_project_dijkstras_parallel/blob/master/Dijkstra-OpenMP.c)
- ❖ [https://fr.wikipedia.org/wiki/Arbre\\_bicolore](https://fr.wikipedia.org/wiki/Arbre_bicolore)
- ❖ [https://fr.wikipedia.org/wiki/Algorithme\\_de\\_Kruskal](https://fr.wikipedia.org/wiki/Algorithme_de_Kruskal)

- ❖ <https://www.geeksforgeeks.org/kruskals-minimum-spanning-tree-algorithm-greedy-algo-2/>
- ❖ [https://fr.wikipedia.org/wiki/Algorithme\\_de\\_Prim](https://fr.wikipedia.org/wiki/Algorithme_de_Prim)
- ❖ <https://www.geeksforgeeks.org/prims-minimum-spanning-tree-mst-greedy-algo-5/>