Chapitre 4

Les graphes

4.1 Introduction

Les graphes, de par leur nature abstraite et leur puissante capacité à modéliser des relations complexes, jouent un rôle essentiel dans de nombreux domaines tels que l'informatique, les réseaux sociaux, la logistique et bien d'autres encore. Leur utilisation s'étend également à des problèmes du quotidien, tels que les itinéraires optimaux dans une ville, la planification de projets ou la recommandation de produits en ligne. Dans ce chapitre, nous plongeons au cœur de l'univers des graphes, en nous concentrant sur deux aspects fondamentaux : leur représentation et les méthodes pour les parcourir efficacement. La représentation d'un graphe est un élément clé pour stocker et manipuler les données relatives aux sommets et aux arêtes qui les relient. Elle offre une multitude d'options, allant des structures simples aux plus sophistiquées, chacune ayant ses avantages et ses limitations en termes de temps et d'espace. Nous explorerons en détail ces différentes représentations et analyserons leurs impacts sur les performances des algorithmes.

Outre la représentation, le parcours des graphes est un sujet incontournable pour comprendre et exploiter pleinement leurs propriétés. Que ce soit pour trouver des chemins, détecter des cycles, ou résoudre des problèmes de connectivité, les algorithmes de parcours sont au cœur de nombreuses applications pratiques. Nous découvrirons les algorithmes les plus couramment utilisés, comme la recherche en profondeur (DFS) et la recherche en largeur (BFS), tout en examinant leurs différences, leurs applications spécifiques et leurs implications sur la complexité des calculs.

En acquérant une solide compréhension des représentations de graphes et des techniques de parcours, vous serez équipés pour aborder des problèmes divers et variés dans le monde réel.

4.2 Définitions

Un graphe est un ensemble d'éléments appelés "sommet" (ou "nœud") reliés par des "arêtes". Mathématiquement, un graphe est défini comme un couple ordonné G = (V, E), où :

— V : un ensemble fini d'éléments non ordonnés, appelés sommets (ou nœuds).

— E : un ensemble fini d'éléments non ordonnés de paires de sommets appartenant à V, appelés arêtes. Chaque arête présente une connexion entre deux sommets du graphe.

Les arêtes peuvent être non-directionnelles, ce qui signifie que les connexions entre les sommets n'ont pas de direction spécifique, ou elles peuvent être directionnelles, indiquant ainsi un sens spécifique entre les sommets. Les graphes peuvent être utilisés pour modéliser une grande variété de situations réelles. Ils sont un outil puissant pour résoudre des problèmes d'optimisation, ce qui en fait une structure de données fondamentale en informatique et en mathématiques.

Un graphe peut-être visualisé comme un ensemble de points (les sommets) reliés par des lignes (les arêtes). La figure 4.1 montre un exemple de graphe non-directionnel avec 6 sommets et 7 arêtes.

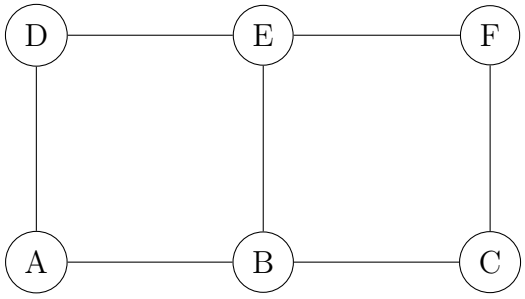

FIGURE 4.1 – Exemple de graphe non-directionnel

Les arêtes d'un graphe peuvent aussi être pondérées, ce qui signifie qu'elles peuvent avoir un poids associé. Le poids d'une arête est une valeur numérique qui peut représenter une distance, un coût, une capacité, etc. Les graphes dont les arêtes sont pondérées sont appelés "graphes pondérés". La figure 4.2 montre un exemple de graphe directionnel pondéré avec 6 sommets et 7 arêtes.

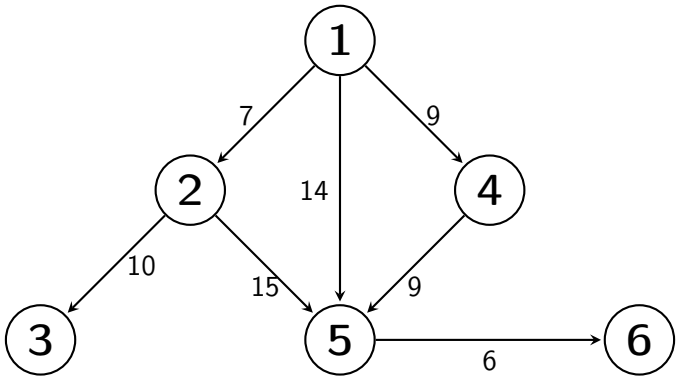

FIGURE 4.2 – Exemple de graphe directionnel pondéré

4.3 Graphes particuliers

Il existe de nombreux types de graphes particuliers, chacun ayant des propriétés, des caractéristiques et des applications spécifiques. En fonction des fonctionnalités requises, certains types de graphes peuvent être plus adaptés que d'autres. Dans cette section, nous présentons les types de graphes particuliers les plus courants.

4.3.1 Graphe simple

Un graphe simple est un graphe non-orienté qui ne comporte pas d'arêtes multiples (ou boucles) entre deux sommets et où chaque paire de sommets est reliée par au plus une arête. Autrement dit, dans un graphe simple, il n'y a pas d'arêtes reliant un sommet à lui-même (boucle) et il n'y a pas de plusieurs arêtes entre deux sommets distincts.

Les graphes simples sont couramment utilisés pour modéliser des relations sans considérer de détails supplémentaires tels que les poids ou les directions des arêtes. Ils sont souvent utilisés pour représenter des situations où une relation binaire entre les éléments est suffisante. Il est important de noter que le terme "simple" est utilisé pour distinguer ce type de graphe des graphes avec des arêtes multiples ou orientées, qui peuvent être utilisés pour des modélisations plus complexes et spécifiques.

4.3.2 Graphe connexe

Un graphe non-orienté est dit "connexe" s'il existe un chemin entre n'importe quel couple de sommets du graphe. Autrement dit, dans un graphe connexe, il est possible de passer d'un sommet à n'importe quel autre sommet en suivant les arêtes du graphe. Si le graphe n'est pas connexe, on peut le diviser en plusieurs sous-graphes connexes, appelés "composantes connexes". Chaque composante connexe est elle-même un graphe connexe.

La propriété de connexité est essentielle dans de nombreuses applications et problèmes liés aux graphes. Par exemple, dans les réseaux de transport, la connexité assure qu'il est possible de voyager d'une ville à une autre en empruntant les bonnes routes ou lignes de transport.

4.3.3 Graphe fortement connexe

Le connexité est une propriété importante pour les graphes non-orientés, mais elle ne s'applique pas aux graphes orientés. En effet, dans un graphe orienté, il est possible qu'il n'existe pas de chemin entre deux sommets, même si le graphe est connexe. Pour cette raison, la propriété de connexité est remplacée par la propriété de "forte connexité" pour les graphes orientés. Un graphe orienté est dit "fortement connexe" s'il existe un chemin "orienté" entre n'importe quel couple de sommets du graphe.

4.3.4 Graphe complet

Un graphe non-orienté est dit "complet" s'il existe une arête entre chaque paire de sommets du graphe. Autrement dit, dans un graphe complet, chaque sommet est relié à tous les autres sommets du graphe. Un graphe complet avec n sommets possède $n(n-1)/2$ arêtes. Nous montrons un exemple de graphe complet avec 4 sommets et 6 arêtes dans la figure 4.3.

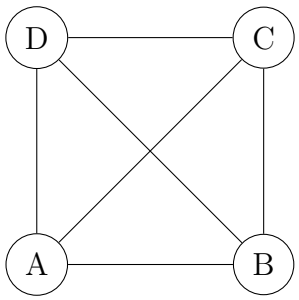

FIGURE 4.3 – Exemple de graphe complet

4.3.5 Graphe biparti

Un graphe biparti est un graphe non-orienté dont l'ensemble des sommets peut être divisé en deux sous-ensembles disjoints V_1 et V_2, tels que chaque arête du graphe relie un sommet de V_1 à un sommet de V_2. Autrement dit, dans un graphe biparti, il n'existe pas d'arête reliant deux sommets du même sous-ensemble. Un exemple de graphe biparti est montré dans la figure 4.4, Dans cet exemple, les sommets peuvent être divisés en deux sous-ensembles disjoints $V_1 = \{A, B, C\}$ et $V_2 = \{D, E, F\}$.

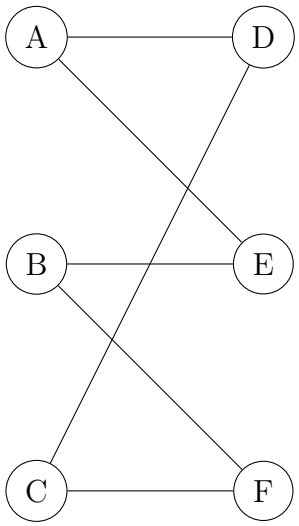

FIGURE 4.4 – Exemple de graphe biparti

4.3.6 Graphe acyclique

Un graphe acyclique est un type de graphe, généralement orienté, qui ne contient aucun cycle, c'est-à-dire qu'il n'existe aucun chemin qui permet de revenir à un sommet en suivant les arcs orientés du graphe. Autrement dit, un graphe acyclique ne possède pas de boucles, où une boucle est une séquence d'arcs qui forme un cycle en revenant au sommet de départ.

Les graphes cycliques sont particulièrement utiles pour modéliser des relations de dépendance ou de hiérarchie, où l'ordre des connexions est important et ne doit pas former de cycles. Les graphes acycliques trouvent de nombreuses applications pratiques, notamment dans la gestion de projets (recherche du chemin critique), l'analyse de réseaux de dépendance (tels que les réseaux de tâches ou les flux de données), l'optimisation de circuits électroniques, la vérification de la cohérence des données, la planification logistique et bien plus encore.

Les algorithmes pour détecter les cycles dans les graphes (comme le parcours en profondeur avec la recherche de backtracking) peuvent être utilisés pour déterminer si un graphe est acyclique. Si aucun cycle n'est détecté, alors le graphe est acyclique. Si un cycle est trouvé, le graphe est dit "cyclique". Les graphes acycliques sont importants car ils possèdent des propriétés mathématiques et algorithmiques spéciales qui permettent de résoudre efficacement des problèmes complexes dans divers domaines.

4.3.7 Graphe probabiliste

Un graphe probabiliste est un type de graphe où chaque arête est associée à une probabilité ou à une valeur numérique représentant une mesure de poids. Contrairement aux graphes classiques où les arêtes sont simplement des connexions binaires entre les sommets, les graphes probabilistes prennent en compte les probabilités pour refléter l'incertitude, la fiabilité ou l'importance des connexions entre les sommets.

Dans un graphe probabiliste, les arêtes peuvent représenter diverses informations probabilistes, telles que :

— Probabilité de transition entre deux états dans un processus stochastique.

— Fiabilité de la connexion entre deux sommets dans un réseau.

— Probabilité de succès d'une action dans un processus décisionnel.

— Coûts ou distance entre deux sommets dans un réseau de transport.

Les graphes probabilistes peuvent être utilisés pour modéliser des systèmes complexes où l'incertitude est présente, et où les probabilités jouent un rôle crucial dans la prise de décision. Ils sont utilisés dans des domaines tels que l'intelligence artificielle, l'apprentissage automatique, la théorie des jeux, la théorie de l'information et bien d'autres. L'analyse des graphes probabilistes implique souvent des algorithmes spécifiques qui tiennent compte des probabilités associées aux arêtes.

4.3.8 Graphe planaire

Un graphe planaire est un type de graphe qui peut être dessiné sur un plan (une surface plane) de telle sorte que ses arêtes ne se croisent pas. Autrement dit, dans un graphe planaire, il est possible de représenter ses sommets et arêtes sur une feuille de papier sans que les lignes représentant les arêtes ne se croisent.

Le concept de graphe planaire est d'une grande importance en théorie des graphes. Les graphes planaires ont des propriétés spéciales et des caractéristiques uniques qui les distinguent des graphes non-planaires et trouvent de nombreuses applications dans la modélisation de réseaux géographiques, de circuits imprimés, de plans d'étages, de cartes, de problèmes de routage et de bien d'autres domaines où la disposition spatiale des sommets et des arêtes est essentielle. Le graphe présenté dans la figure 4.3 est aussi un graphe planaire, car il peut être dessiné sur un plan sans que ses arêtes ne se croisent comme il est montré dans la figure 4.5.

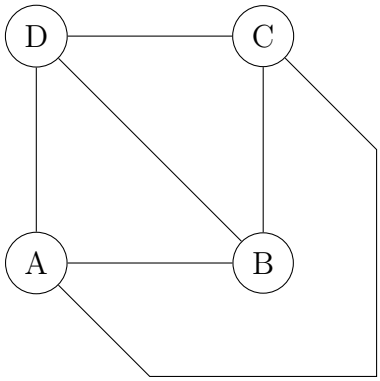

FIGURE 4.5 – Exemple de graphe planaire

4.3.9 Graphe régulier

Un graphe est dit "régulier" lorsque tous ses sommets ont le même degré, c'est-à-dire que chaque sommet est relié au même nombre d'arêtes. Le degré d'un sommet est le nombre d'arêtes qui lui sont incidentes, c'est-à-dire le nombre d'arêtes qui sont directement connectées à ce sommet. On parle souvent de graphes d-réguliers, où d est le degré commun à tous les sommets. Par exemple, un graphe 3-régulier est un graphe où chaque sommet est relié à exactement trois arêtes. Les graphes réguliers peuvent être représentés sous différentes formes et peuvent être non-orientés ou orientés. Ils peuvent être utilisés pour modéliser diverses situations, par exemple dans les réseaux de communication où chaque nœud est relié à un nombre fixe de voisins, ou dans les structures cristallines où chaque atome est lié à un nombre spécifique d'atomes voisins.

Les graphes réguliers sont d'un grand intérêt en théorie des graphes en raison de leurs propriétés spéciales et de leur symétrie. Ils peuvent être utilisés pour résoudre des problèmes spécifiques et ont des applications dans divers domaines tels que les réseaux, la cryptographie, la conception de circuits, et bien d'autres.

4.4 Représentation d'un graphe

Il existe plusieurs manières de représenter un graphe. Chaque représentation a ses avantages et ses inconvénients, et est adaptée à des cas d'utilisation spécifiques. Dans cette section, nous allons présenter les représentations les plus courantes des graphes.

4.4.1 Matrice d'adjacence

La matrice d'adjacence est une matrice carrée qui représente les arêtes d'un graphe. Elle est utilisée pour représenter les graphes orientés et non orientés. La matrice d'adjacence

est symétrique pour les graphes non orientés, ce qui signifie que les arêtes (i, j) et (j, i) sont représentées par la même valeur. Par contre, pour les graphes orientés, la matrice d'adjacence n'est pas forcément symétrique, car les arêtes (i, j) et (j, i) peuvent avoir des valeurs différentes. La matrice d'adjacence est une matrice carrée de taille n x n, où n est le nombre de sommets du graphe. La matrice d'adjacence est une matrice booléenne, où chaque élément de la matrice est soit 0, soit 1. La valeur 1 indique qu'il existe une arête entre les sommets i et j, et la valeur 0 indique qu'il n'y a pas d'arête entre les sommets i et j. Pour modéliser les graphes pondérés, la matrice d'adjacence peut être une matrice de nombres réels, où chaque élément de la matrice représente le poids de l'arête entre les sommets i et j.

La matrice d'adjacence suivante correspond au graphe dans la figure 4.4 :

	A	B	C	D	E	F
A	0	0	0	1	1	0
B	0	0	0	0	1	1
C	0	0	0	1	0	1
D	1	0	1	0	0	0
E	1	1	0	0	0	0
F	0	1	1	0	0	0

L'avantage de la matrice d'adjacence est qu'elle est simple et facile à implémenter. Cependant, elle est inefficace en termes d'espace mémoire, car elle nécessite un espace de stockage de $O(n^2)$, même si le graphe est très creux. De plus, la matrice d'adjacence ne peut pas être utilisée pour représenter les graphes avec un nombre de sommets très élevé à cause de sa complexité mémoire.

4.4.2 Matrice d'incidence

La matrice d'incidence est une matrice rectangulaire qui représente les arêtes d'un graphe. Elle est utilisée pour représenter les graphes orientés et non orientés. La matrice d'incidence est une matrice de taille n x m, où n est le nombre de sommets du graphe et m est le nombre d'arêtes du graphe. Dans le cas d'un graphe non orienté, la matrice d'incidence est une matrice booléenne, où chaque élément de la matrice est soit 0, soit 1. La valeur 1 indique que le sommet i est incident à l'arête j, et la valeur 0 indique que le sommet i n'est pas incident à l'arête j. Dans le cas d'un graphe orienté, la matrice d'incidence est une matrice ou chaque élément de la matrice est soit 0, soit 1, soit -1. La valeur 1 indique que le sommet i est l'origine de l'arête j, la valeur -1 indique que le sommet i est la

destination de l'arête j, et la valeur 0 indique que le sommet i n'est pas incident à l'arête j.

La matrice d'incidence suivante correspond au graphe non-orienté dans la figure 4.4 :

	1	2	3	4	5	6
A	1	1	0	0	0	0
B	0	0	1	1	0	0
C	0	0	0	0	1	1
D	1	0	0	0	1	0
E	0	1	1	0	0	0
F	0	0	0	1	0	1

Comme pour la matrice d'adjacence, l'avantage de la matrice d'incidence est qu'elle est simple et facile à implémenter. En plus, elle peut être plus efficace dans le cas où le graphe est très creux, Mais moins efficace dans le cas où le graphe est très dense.

4.4.3 Liste d'adjacence

La liste d'adjacence est une structure de données qui représente les arêtes d'un graphe. Elle est utilisée pour représenter les graphes orientés et non orientés. La liste d'adjacence est une liste de taille n, où n est le nombre de sommets du graphe. Chaque élément de la liste est une liste qui contient les sommets adjacents au sommet correspondant. Dans le cas d'un graphe orienté, la liste d'adjacenc peut contenir soit les sommets qui sont l'origine des arêtes qui arrivent du sommet correspondant, ou les sommets qui sont la destination des arêtes qui partent au sommet correspondant.

La liste d'adjacence suivante correspond au graphe dans la figure 4.4 :

- A : D, E
- B : E, F
- C : D, F
- D : A, C
- E : A, B
- F : B, C

4.5 Parcours de graphes

Le parcours de graphes est un algorithme qui permet de visiter tous les sommets d'un graphe. Il existe deux types de parcours de graphes, le parcours en profondeur (DFS) et le

parcours en largeur (BFS). Ces deux algorithmes sont basés sur le principe de marquage des sommets d'un graphe, une technique qui permet de marquer les sommets visités pour les distinguer des sommets non visités.

4.5.1 Parcours en largeur

Le parcours en largeur (BFS - Breadth-First Search) est un algorithme pour parcourir ou explorer tous les sommets d'un graphe de manière itérative, en commençant par un sommet de départ donné, puis en visitant tous ses voisins avant de passer aux sommets voisins de ses voisins, et ainsi de suite. Cela signifie que l'algorithme explore d'abord tous les sommets à une distance de 1 du sommet de départ, puis tous les sommets à une distance de 2, et ainsi de suite. Les étapes de l'algorithme de parcours en largeur sont les suivantes :

1. Choisir un sommet de départ.
2. Marquer le sommet de départ comme visité.
3. Ajouter le sommet de départ à la file d'attente.
4. Tant que la file d'attente n'est pas vide, faire :
   (a) Retirer le sommet en tête de la file d'attente.
   (b) Pour chaque sommet adjacent au sommet retiré, faire :
      i. Si le sommet adjacent n'est pas marqué, faire :
         A. Marquer le sommet adjacent comme visité.
         B. Ajouter le sommet adjacent à la file d'attente.

L'algorithme garantit que tous les sommets accessibles depuis le sommet de départ sont visités, et qu'aucun sommet n'est visité plus d'une fois. Le parcours en largeur permet également de déterminer le niveau de chaque sommet, c'est-à-dire la distance entre le sommet de départ et chaque sommet visité. L'algorithme de parcours en largeur est largement utilisé pour résoudre divers problèmes de graphe, tels que la recherche de chemins les plus courts, la connectivité, la recherche de composantes connexes, la détection de cycles, et bien plus encore.

Dans l'algorithme ci-dessous, nous utilisons le parcours en largeur pour afficher tous les sommets accessibles depuis le sommet de départ. Nous supposons que le graphe est représenté par une matrice d'adjacence, et que la structure de données *queue* avec les fonctions de base *push*(), *pop*() et *empty*() est déjà implémentée.

```
1 void bfs(int **graph, int n, int start) {
```

```
2   int visited[n] = {0};
3   queue q;
4   push(&q, start);
5   visited[start] = 1;
6   while (!empty(&q)) {
7     int current = pop(&q);
8     printf("%d ", current);
9     for (int i = 0; i < n; i++) {
10      if (graph[current][i] == 1 && visited[i] == 0) {
11        push(&q, i);
12        visited[i] = 1;
13      }
14    }
15  }
16 }
```

Algorithm 4.1 – Parcours en largeur

4.5.2 Parcours en profondeur

Le parcours en profondeur (DFS - Depth-First Search) est un algorithme pour parcourir ou explorer tous les sommets d'un graphe de manière récursive, en remontant aussi loin que possible le long de chaque branche avant de revenir en arrière pour explorer d'autres branches. Cela signifie que l'algorithme explore autant que possible en profondeur avant de revenir en arrière.

Les étapes de l'algorithme de parcours en profondeur sont les suivantes :

1. Choisir un sommet de départ.
2. Marquer le sommet de départ comme visité.
3. Pour chaque sommet adjacent au sommet de départ, faire :
   (a) Si le sommet adjacent n'est pas marqué, faire :
      i. Appeler récursivement la fonction DFS avec le sommet adjacent comme sommet de départ.

L'algorithme de parcours en profondeur explore d'abord en profondeur le plus loin possible le long d'une branche avant de revenir en arrière et d'explorer d'autres branches. Cela signifie qu'une fois qu'un sommet est visité, l'algorithme va d'abord explorer tous ses voisins avant de passer aux autres sommets du graphe. L'algorithme de parcours en profondeur est largement utilisé pour résoudre divers problèmes de graphe, la détection de cycles, la recherche de composantes connexes, et bien plus encore. Il est également utilisé dans des problèmes d'exploration de labyrinthes et de résolution de problèmes liés aux

arbres et aux graphes.

Dans l'algorithme ci-dessous, nous utilisons le parcours en profondeur pour afficher tous les sommets accessibles depuis le sommet de départ. Nous supposons que le graphe est représenté par une matrice d'adjacence. L'algorithme utilise le principe de récursivité pour explorer tous les sommets accessibles depuis le sommet de départ.

```
1 void dfs_rec(int **graph, int n, int start, int *visited) {
2   printf("%d ", start);
3   visited[start] = 1;
4   for (int i = 0; i < n; i++) {
5     if (graph[start][i] == 1 && visited[i] == 0) {
6       dfs_rec(graph, n, i, visited);
7     }
8   }
9 }
```

Algorithm 4.2 – Parcours en profondeur

La fonction $dfs_rec()$ est appelée avec le sommet de départ et le tableau *visited* initialisé à 0 comme paramètres pour le premier appel. La fonction $dfs_rec()$ affiche le sommet de départ et le marque comme visité. Ensuite, elle appelle récursivement la fonction $dfs()$ pour tous les sommets adjacents au sommet de départ qui ne sont pas encore visités. La fonction $dfs()$ affiche tous les sommets accessibles depuis le sommet de départ. Autrement, la fonction $dfs_rec()$ ne fait rien, pourrait être utilisée comme fonction utilitaire pour cacher les paramètres de récursivité.

4.5.3 L'algorithme de Dijkstra

L'algorithme de Dijkstra est un algorithme de recherche de plus court chemin, qui permet de trouver le plus court chemin entre un sommet de départ et tous les autres sommets d'un graphe pondéré orienté. L'algorithme de Dijkstra est basé sur le principe de relaxation, qui consiste à mettre à jour la distance minimale d'un sommet à partir du sommet de départ, en comparant la distance actuelle avec la distance calculée en passant par un autre sommet. L'algorithme de Dijkstra est largement utilisé dans les systèmes de routage, les systèmes de navigation, les réseaux de télécommunications, etc. L'algorithme de Dijkstra peut être utilisé sur un graphe pondéré non-orienté ou orienté avec des poids non-négatifs. Les étapes à suivre pour trouver le plus court chemin entre un sommet de départ et tous les autres sommets sont les suivantes :

1. Initialiser la distance de tous les sommets à l'infini, sauf le sommet de départ dont la distance est 0.

2. Ajouter le sommet de départ à l'ensemble des sommets à traiter.
3. Tant que l'ensemble des sommets à traiter n'est pas vide, faire :
   (a) Retirer le sommet avec la distance minimale de l'ensemble des sommets à traiter.
   (b) Pour chaque sommet adjacent au sommet retiré, faire :
      i. Si la distance du sommet adjacent est supérieure à la distance du sommet retiré plus le poids de l'arête qui relie le sommet retiré au sommet adjacent, faire :
         A. Mettre à jour la distance du sommet adjacent.
         B. Ajouter le sommet adjacent à l'ensemble des sommets à traiter.

L'implémentation de l'algorithme de Dijkstra pour un graphe pondéré modélisé par une matrice d'adjacence est donnée ci-dessous :

```
1 int min_distance(int *distance, int *visited, int n) {
2   int min = INT_MAX;
3   int min_index = -1;
4   for (int i = 0; i < n; i++) {
5     if (visited[i] == 0 && distance[i] <= min) {
6       min = distance[i];
7       min_index = i;
8     }
9   }
10  return min_index;
11 }
12
13 void dijkstra(int **graph, int n, int start) {
14  int *distance = malloc(n * sizeof(int));
15  int *visited = malloc(n * sizeof(int));
16  for (int i = 0; i < n; i++) {
17    distance[i] = INT_MAX;
18    visited[i] = 0;
19  }
20  distance[start] = 0;
21  for (int i = 0; i < n; i++) {
22    int u = min_distance(distance, visited, n);
23    visited[u] = 1;
24    for (int v = 0; v < n; v++) {
25      if (graph[u][v] != 0 && visited[v] == 0 && distance[u] != INT_MAX
26        && distance[u] + graph[u][v] < distance[v]) {
27        distance[v] = distance[u] + graph[u][v];
28      }
29    }
```

```
30   }
31   print_distances(distance, n);
32 }
```

Algorithm 4.3 – Algorithme de Dijkstra

La fonction *dijkstra*() prend en paramètres la matrice d'adjacence du graphe, le nombre de sommets du graphe et le sommet de départ. Le tableau de distances *distance* est initialisé à l'infini pour tous les sommets, sauf le sommet de départ dont la distance est 0. La fonction *dijkstra*() appelle la fonction *min_distance*() pour retourner l'indice du sommet non visité avec la distance minimale. Ce dernier est marqué comme visité et la distance de tous ses sommets adjacents est mise à jour si nécessaire. Une fois que tous les sommets sont visités, les distance minimales de tous les sommets sont affichées avec la fonction *print_distances*() que nous n'allons pas détailler ici.

4.6 Conclusion

Les graphes, sont des structures mathématiques puissantes pour modéliser et représenter des relations complexes entre des objets. Au cours de ce chapitre, nous nous sommes concentrés sur deux aspects fondamentaux des graphes : leur représentation et leur parcours. La représentation des graphes est essentielle pour stocker et manipuler les informations relatives aux sommets et aux arêtes. Nous avons étudié différentes méthodes de représentation, dont la matrice d'adjacence, la liste d'adjacence et la matrice d'incidence. Chaque méthode présente ses avantages et ses limites, et le choix de la représentation dépend du type de graphe et des opérations que l'on souhaite effectuer. En ce qui concerne le parcours des graphes, nous avons examiné deux algorithmes classiques : le parcours en largeur (BFS) et le parcours en profondeur (DFS). Ces deux approches offrent des stratégies différentes pour explorer les sommets. Le parcours en largeur est utile pour trouver les plus courts chemins dans un graphe pondéré, la connectivité, et bien d'autres problèmes. Quant au parcours en profondeur, il est adapté à la recherche de cycles, à la détermination des composantes connexes et à la résolution de problèmes liés aux arbres.

Les graphes sont omniprésents dans de nombreux domaines, tels que les réseaux sociaux, les systèmes de transport, la gestion de projets, la logistique et bien d'autres. Leur utilité dans la résolution de problèmes complexes est indéniable. Nous avons seulement effleuré la surface des possibilités offertes par cette discipline passionnante qu'est la théorie des graphes. En guise de conclusion, nous pouvons affirmer que la compréhension des graphes et des algorithmes qui y sont associés est un atout précieux pour tout informaticien, mathématicien ou décideur qui souhaite modéliser et résoudre des problèmes du monde réel. Les graphes offrent des outils puissants pour l'analyse, la planification et la prise de

décision, et continuent d'être au cœur de nombreuses applications pratiques et théoriques. Ainsi, en maîtrisant les concepts de représentation et de parcours des graphes, nous nous donnons la capacité d'explorer et de comprendre le fonctionnement complexe de nombreux systèmes et réseaux qui nous entourent, ouvrant ainsi la voie à de nouvelles possibilités et découvertes.

Bibliographie

[1] D Beauquier, J Berstel et Ph Chrétienne. *Eléments d'algorithmique*. 2005.

[2] Gilles Brassard et Paul Bratley. *Fundamentals of algorithmics*. Prentice-Hall, Inc., 1996.

[3] Thomas H Cormen, Charles Eric Leiserson, Ronald L Rivest, Philippe Chrétienne et Xavier Cazin. *Introduction à l'algorithmique*. Dunod, 1994.

[4] Sampath Kannan, Moni Naor et Steven Rudich. « Implicit representation of graphs ». In : *Proceedings of the twentieth annual ACM symposium on Theory of computing*. 1988, p. 334-343.

[5] Aditya Dev Mishra et Deepak Garg. « Selection of best sorting algorithm ». In : *International Journal of intelligent information Processing* 2.2 (2008), p. 363-368.

[6] Robert Sedgewick et Philippe Flajolet. « Introduction à l'analyse des algorithmes ». In : *(No Title)* (1996).