Graphes : Parcours, plus court chemin

Dans ce cours, on va tester l'efficacité d'un même algorithme (l'algorithme du plus court chemin de Dijkstra) selon différentes manières de représenter et d'implémenter un graphe

1. Définitions, notations
2. Parcours
3. Algorithme de Dijkstra
4. Matrices d'adjacence
5. Listes d'adjacence

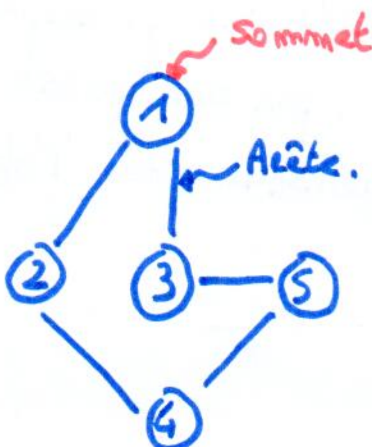
---

1. Définitions, notations

Un graphe est une structure de donnée non linéaire constituée de 2 types d'objets :

- des noeuds, ou sommets, contenus dans un ensemble S
- des arêtes, contenus dans un ensemble A, qui relient des sommets entre eux

Ex :



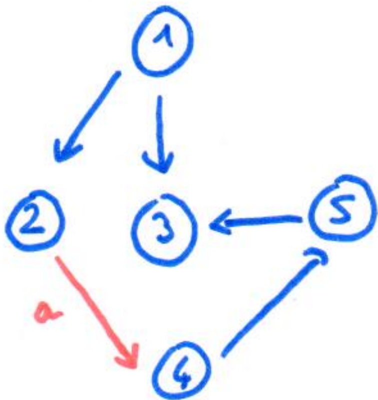
$|S| := n$ nombre de sommets

$|A| := a$ nombre d'arêtes

- Un graphe est dit <u>orienté</u> si toutes les arêtes sont équipées d'une certaine direction, indiquée par une flèche.

Dans un graphe orienté, chaque arête a un sommet source et un sommet but ("target")

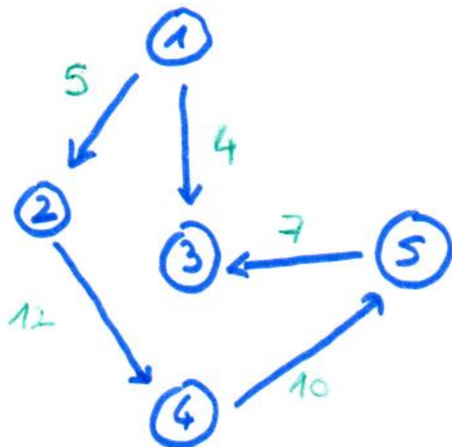
<u>Ex:</u>



src(a) = 2
tgt(a) = 4

- Un graphe est dit <u>pondéré</u> si toutes les arêtes sont étiquetées par un nombre, encodant généralement la distance entre 2 sommets. Ce nombre est appelé poids.

<u>Ex:</u>



<u>Remarque</u>: Un graphe pondéré peut être non orienté...

Dans ce cours, nous allons nous intéresser principalement à des graphes orientés et pondérés, par des poids entiers positifs (la positivité est indispensable pour l'algorithme de Dijkstra).

(2) Parcours

Il existe plusieurs manières de parcourir / d'explorer un graphe, nous allons en voir 2 principales :

(a) Parcours en profondeur

Ex : G =

(1) → (2)
(1) → (4)
(2) → (3)
(2) → (5)
(6) → (5)

- On part d'un sommet src (1) à partir duquel on explore

(1) • On va créer une structure de pile (LIFO, Last In First Out) dans laquelle on va stocker les sommets parcourus

(2) • On cherche un voisin de l'élément en tête de pile (n'importe lequel, le premier qu'on rencontre), qu'on ajoute en tête de pile, et on répète tant que le nouveau sommet en tête de pile a toujours des voisins non utilisés

Pile :

(1)			(1)
(2)	(1)		(2)
(3)	(2)	(1)	(2)

(3) • Lorsque ce n'est plus le cas, on dépile l'élément en tête, et on recherche un autre voisin du sommet précédent, et on l'ajoute en tête si il y en a un ; sinon on dépile aussi, etc.

(2)	(1)	(3)

(5) (2) (1) (2)

(2) (1) (3)

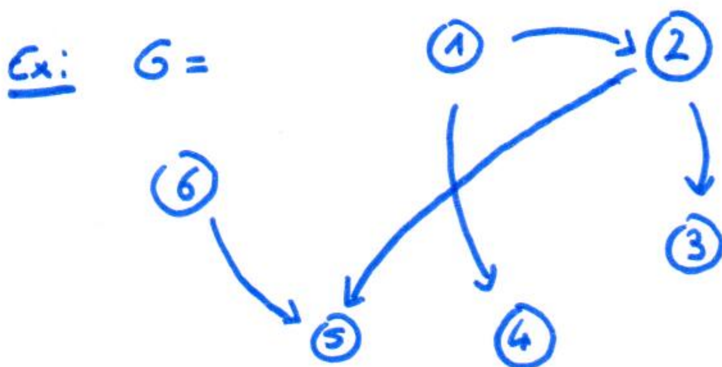
(1) (3)

Empty

On s'arrête lorsque la pile est vide. Au final, tous les sommets qui ont été stockés dans la pile ont été parcours en partant de SRC = (1).

(b) Parcours en largeur

Ex: G =



- On part d'un sommet SRC = (1) à partir duquel on explore

(1) - On va créer une structure de file (LIFO, Last In first out) dans laquelle on place SRC

(2) - On ajoute ensuite tous les voisins de SRC dans la file (idem, dans n'importe quel ordre), puis on supprime l'élément SRC

- On recommence ensuite avec le premier voisin en tête de file, on ajoute ses voisins à la fin, etc

File

(1) (1)

(1) (2) (4) } (2)

(2) (4)

(2) (4) (3) (5) (2)

(3). Si on arrive sur un sommet qui n'a plus de voisin non déjà parcouru, on le supprime de la file

(4) (3) (5)

(3) (5) (3)

(5) (3)

Empty (3)

Le parcours s'arrête lorsque la file est vide. Tous les sommets qui ont été stockés dans la file ont été parcourus en partant de src = (1)

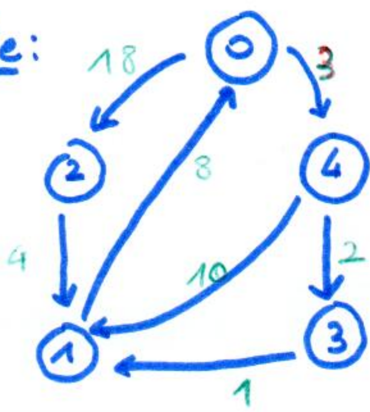
Remarque : • Pour un graphe non orienté, le graphe est connexe <u>ssi</u> tous les sommets sont explorés à partir d'un parcours partant de n'importe quel sommet.

• On peut dessiner l'arbre couvrant dans ce cas en gardant trace du parent à chaque ajout d'un sommet dans la file

(3) Algorithme de Dijkstra

C'est un algorithme permettant de trouver le plus court chemin entre 2 sommets dans un graphe pondéré

Exemple :



Chemins de (0) vers (1) :

(0) → (2) → (1) longueur 24

(0) → (4) → (1) longueur 13

(0) → (4) → (2) → (1) longueur 6

Plus court chemin

Paramètres de l'algorithme :

- un graphe orienté pondéré avec des nombres positifs ;
- un sommet SRC à partir duquel on va renvoyer un tableau sous la forme

Sommets	s_1	s_2	...	s_n
Distance de SRC	d_1	d_2		d_n

d_i := distance de SRC au sommet s_i la plus petite.

<u>Initialisation</u> : On crée un tableau de taille n qu'on remplit avec :
- INF / 9999 pour les sommets différents de SRC
- 0 pour SRC.

<u>Traitement</u> : Commençons par traiter le sommet SRC et tous ses arcs sortants.

On va regarder toutes les arêtes sortant de a, dans un ordre quelconque
- Arc de 0 vers 2, longueur 18, qui est mieux que la valeur précédente INF dans dist[2]
  ⇝ On change la valeur de dist[2] en 18.

- Arc de 0 vers 4, longueur 3
  ⇝ On change la valeur de dist[4] en 3.

Sommets	0	1	2	3	4
Dist. de 0	0	INF	18	INF	3

- On va ensuite choisir un autre sommet.

Peut-on prendre n'importe lequel des deux rencontrés ?

NON !

L'algorithme de Dijkstra impose de choisir celui qui est situé à plus petite distance → ici 4 !

- On traite donc 4 :
  - l'arc 4 → 3 de longueur 2 permet de mettre 5 = dist [4] + 2 dans dist [3].
  - l'arc 4 → 1 de longueur 10 permet de mettre 13 = dist [4] + 10 dans dist [1].

→ A ce stade :

Sommets	0	1	2	3	4
Dist. de 0	0	13	18	5	3

- On recommence avec le sommet non traité qui a la plus petite distance : 3
  - l'arc 3 → 1 de longueur 1 permet de découvrir un chemin de longueur 6 = dist [3] + 1 de 0 vers 1, qui est meilleur que le 13 précédent

    → on change dist [1] en 6.

-

Sommets	0	1	2	3	4
Dist. de 0	0	6	18	5	3

• On traite ensuite 1 :

- l'arc 1 → 0 permet de découvrir un chemin de longueur 14 de à vers a
  ⇝ pas meilleur que le 0 précédent donc on ne change rien.

• On traite ensuite 2 :

- l'arc de 2 vers 1 de longueur 4 permet de découvrir un chemin de longueur 22 de 0 vers 1 ⇝ aucune modification

• On a alors traité tous les sommets, l'algorithme s'arrête et son résultat est

Sommet	0	1	2	3	4
Dist. de 0	0	6	18	5	3

Commentaires : • Pour savoir quels sommets ont déjà été parcourus ou non, on va initialiser un tableau de booléens contenant des 0, et on change la valeur à 1 lorsque le sommet est traité. Avant de traiter un nouveau sommet, il faut donc bien vérifier que il est toujours à 0 dans ce tableau.

• Comment sélectionner le sommet à plus petite distance pour le traiter juste après ?

Il y a plusieurs possibilités :

- on peut écrire une fonction minDistance qui parcourt les sommets adjacents et sélectionne celui de plus petit poids.

Exemple:

```
void dijkstra(graphe *g, int src){
    int *dist = malloc(n * sizeof(int));
    int *acces = malloc(n * sizeof(int));
    int i, j, count;
    for (i=0; i<n; i++){
        dist[i] = INF;
        access[i] = 0;
    }
    dist[src] = 0;
    for (count=0; count<n-1; count++){
        int u = minDistance(dist, access);
        access[u] = 1;
        for (j=0; j<n; j++){
            if ((access[j] == 0) && poids(u->j) != 0
                && dist[u] != INF && dist[u] + poids(u->j)
                                      < dist[j])
                dist[j] = dist[u] + poids(u->j)
    }}}
```

- sinon, il faut stocker les sommets adjacents dans une structure adaptée appelée file de priorité.

C'est une file dans laquelle les tâches de plus petite valeur

enfiler ↪ 16   15   12   5 → défiler

Implémentations possibles:

1) Un tableau
- Enfiler : rajouter à la fin du tableau
- défiler : parcourir toutes les taches jusqu'à trouver le min.

2) Un tableau trié

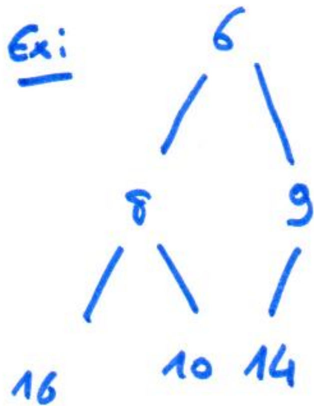
- Défiler : Retirer l'élément en début de tableau qui est le minimum
- Enfiler : il faut trouver où insérer l'élément à la bonne place

Mieux:

3) Un tas binaire C'est un arbre binaire presque complet (chaque niveau est rempli complètement sauf éventuellement le dernier) dans lequel chaque tâche est plus prioritaire que ses enfants

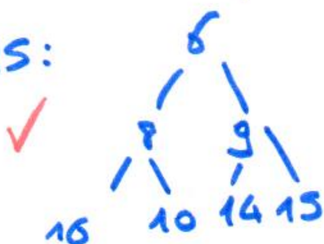
⇝ La tâche la plus prioritaire (le minimum) est la racine

Ex:

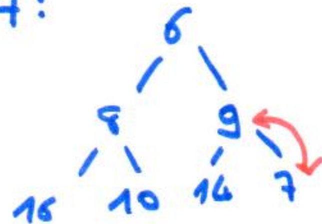


- Enfiler: On insère où il y a une place de libre, en veillant à garder la propriété de tas binaire
  - * si la tâche est moins prioritaire, on laisse
  - * sinon, on échange avec le parent et on répète tant qu'il faut

Enfiler 15:



Enfiler 7:

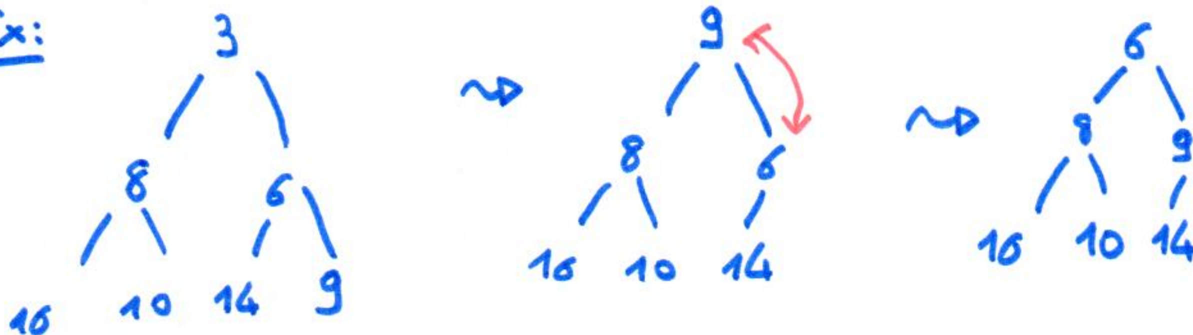


- <u>Défiler</u> : * on enlève l'événement le plus prioritaire qui est à la racine

  * On place le dernier élément en bas à droite à la racine et on compare
    - racine
    - racine (fils-g)
    - racine (fils-d)

  et on met le plus petit en haut

<u>Ex :</u>



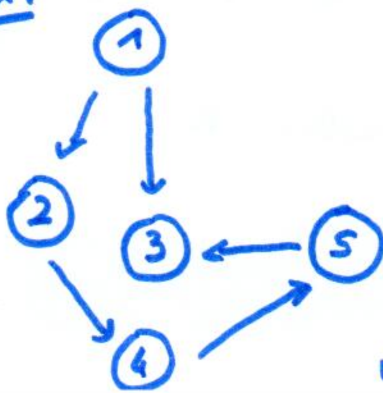
Ces 2 opérations sont en $O(\log n)$: elles se font sur la hauteur d'une branche.

④ Matrices d'adjacence

On peut encoder tout un graphe (orienté ou non, pondéré ou non) par une matrice de taille $n \times n$ appelée matrice d'adjacence.

$$M[i][j] := \begin{cases} 1 & \text{si il y a une arête de } i \text{ vers } j \\ 0 & \text{sinon} \end{cases}$$

<u>Ex :</u>



$$M = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{pmatrix} 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \end{pmatrix} \end{array}$$

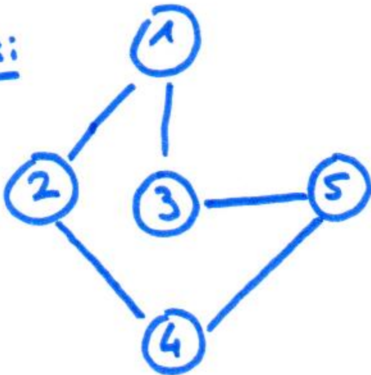
Nombre de 1 dans M = a.

On peut aussi définir une matrice d'adjacence pour un graphe non orienté :

dans ce cas, si il y a une arête de i vers j, il y en a aussi une de j vers i

$\rightsquigarrow M[i][j] = M[j][i]$ (la matrice d'adjacence est symétrique)

Ex :



$$M = \begin{pmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 \end{pmatrix}$$

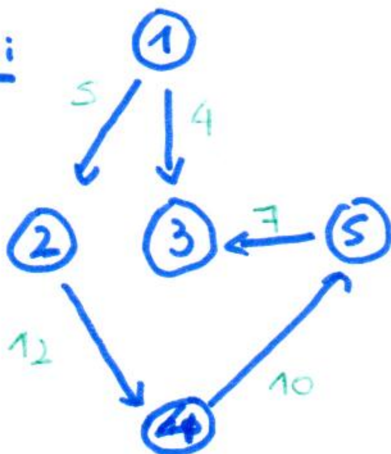
(lignes et colonnes numérotées 1 à 5)

Nombre de 1 = $2 * a$

Idem pour un graphe pondéré :

$$M[i][j] = \begin{cases} \text{poids}(i \to j) & \text{si il y a une arête de } i \text{ vers } j \\ 0 & \text{sinon} \end{cases}$$

Ex :



$$M = \begin{pmatrix} 0 & 5 & 4 & 0 & 0 \\ 0 & 0 & 0 & 12 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 10 \\ 0 & 0 & 7 & 0 & 0 \end{pmatrix}$$

(lignes et colonnes numérotées 1 à 5)

Nombre de coeffs non nuls : a.

```
typedef struct graphe{
    int vertices;
    int adj[MAX_NOEUD][MAX_NOEUD];
}graphe
```

Fonctions: newgraph pour créer un graphe à n sommets, sans arêtes

display pour afficher le nb de sommets et la matrice d'adjacence

add_edge(_weight) pour ajouter une arête d'un sommet i vers un sommet j (avec poids ou non)

Remove_edge pour supprimer une arête (avec poids ou non) de i vers j.

Une propriété: Si G est un graphe (non) orienté de matrice d'adjacence A. Le nombre de chemins de longueur m d'un sommet i vers un sommet j est égal au coeff (i, j) de la matrice A^m.

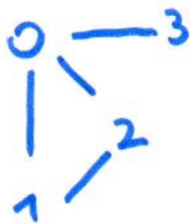
Ceci fournit un autre algo de parcours / plus court chemin...
On calcule toutes les puissances successives A^k $(k \geq 1)$ de A par exponentiation rapide, et on s'arrête si on trouve un entier m tel que
$$\begin{cases} A^m(i,j) \neq 0 \\ A^k(i,j) = 0 \quad \forall\, k < m \end{cases}$$

5 Listes d'adjacence

Un graphe à n sommets peut aussi être représenté par un tableau de n listes contenant chacune les successeurs du sommet en question

En C, on va représenter ces listes par des listes chaînées

Ex:



Sachant qu'on veut aussi accéder au poids des arêtes, on va encoder un noeud d'une liste d'adjacence par

```
typedef struct nodeAdjList{
        int target;
        int weight;
        struct nodeAdjList* next;
    }nodeAdjList;

typedef struct AdjList{
        struct nodeAdjList *head;
    }AdjList
```

On peut ensuite représenter un graphe comme suit:

```
typedef struct graphe{
        int vertices;
        struct AdjList *array;
}graphe
```

encodant le nombre de sommets et un tableau de listes d'adjacence.

<u>Fonctions</u>:

- newgraph pour créer un graphe à n sommets, sans arêtes
- add_edge(graph, src, tgt, weight) pour ajouter une arête avec un poids
- print_adjlist pour afficher les listes d'adj. d'un graphe
- Remplirgraphe(n) qui permet de générer des arêtes pondérées (avec une borne) aléatoirement dans un graphe à n sommets.