Graphes

Guillaume Rousseau
MP2I Lycée Pierre de Fermat
guillaume.rousseau@ens-lyon.fr

23 mai 2024

1 Introduction

Les graphes sont des structures de données complexes, centrales en informatique. Elles permettent de modéliser des situations où plusieurs objets / éléments sont reliés par des relations. Quelques exemples de situations que l'on pourra modéliser avec des graphes :

— Un réseau social type Facebook, où deux personnes peuvent être amies (relation symétrique)

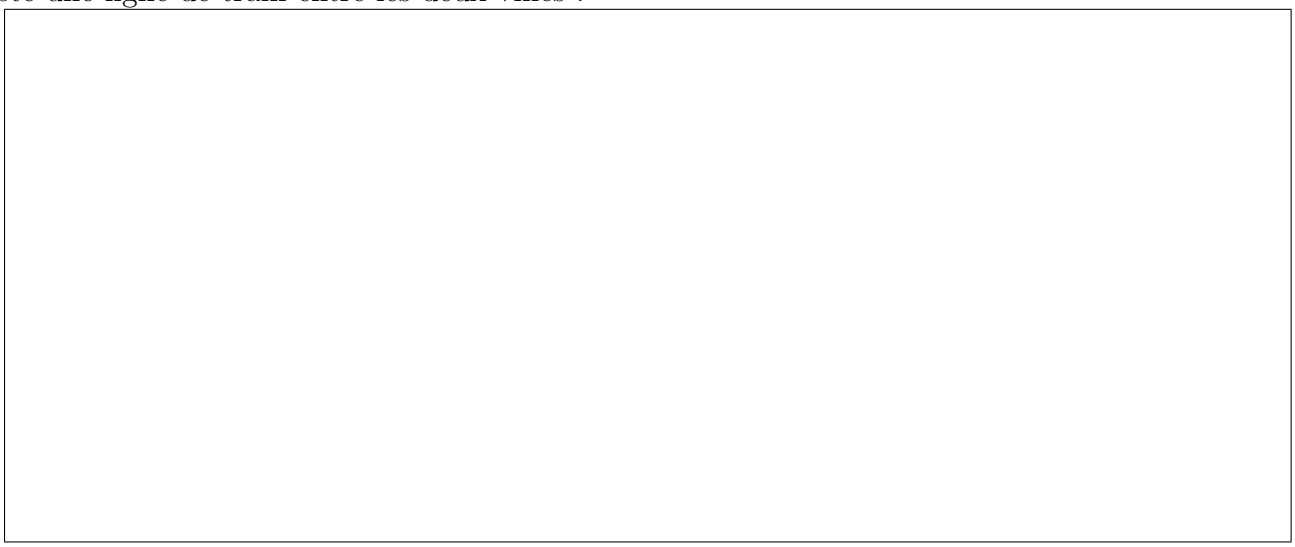
— Un réseau social type Twitter, où une personne peut en suivre une autre (relation asymétrique)

— Le World Wide Web : des pages web pointant les unes vers les autres.

— Une carte du monde : des routes reliant différents points, de longueurs différentes.

A Premières définitions

Schématiquement, un graphe est un ensemble de points reliés par des segments. Les points s'appellent des ***sommets***, et les segments des ***arêtes***. Par exemple, voici un graphe dont les sommets sont des villes de france, et dans lequel deux sommets sont reliés par une arête s'il existe une ligne de train entre les deux villes :



Définition 1. Un graphe non-orienté est un couple $G = (S, A)$ où $A \subseteq \{\{x, y\} \mid x, y \in S\}$. S est appelé l'ensemble des **sommets**, et A l'ensemble des **arêtes**.

Pour $x \in S$, si $\{x\} \in A$ est une arête reliant le sommet x à lui-même, on dit que c'est une **boucle**.

On dit que $x, y \in S$ sont **voisins** s'il existe une arête entre les deux, i.e. si $\{x, y\} \in A$.

Pour $x \in S$, on appelle **voisinage** de x dans G l'ensemble des voisins de x. On le note $\mathcal{V}(x)$:

$$\mathcal{V}(x) = \{y \in S \mid \{x, y\} \in A\}$$

On appelle **degré** de x dans G le nombre de voisins de x, on le note $\mathbf{deg}(x)$:

$$\mathbf{deg}(x) = |\mathcal{V}(x)|$$

Exemple 1. Le graphe donné graphiquement à l'exemple précédent est donc le graphe $G = (S, A)$ non-orienté suivant :

— $S =$ { Toulouse, Strasbourg, Montpellier, Lyon, Marseille, St-Étienne }

— $A =$ {(Toulouse, Montpellier), (Toulouse, Lyon), (Toulouse, Marseille), (Montpellier, Lyon), (Montpellier, Marseille), (Lyon, Lyon), (Lyon, Marseille), (Lyon, St-Étienne)}

Le voisinage de Toulouse est l'ensemble {Montpellier, Lyon, Marseille}, son degré est de 3. Le graphe comporte une boucle, au niveau du sommet Lyon.

Un graphe non-orienté est donc exactement la donnée d'un ensemble S et d'une relation binaire symétrique A sur S. Par opposition, un graphe orienté permettra de représenter des relations pas nécessairement symétriques. Dans un graphe orienté, on relie les sommets non pas par des segments mais par des flèches. Par exemple, le graphe suivant représente quelques types du jeu Pokémon, et l'on ajoute une flèche entre deux types si le premier est efficace contre le deuxième :

Définition 2. Un graphe orienté est un couple $G = (S, A)$ où $A \subseteq S^2$. S est appelé l'ensemble des **sommets**, et A l'ensemble des **arcs**.

Pour $x \in S$, si $(x, x) \in A$ est un arc reliant le sommet x à lui-même, on dit que c'est une **boucle**.

Si $(x, y) \in A$, on dit que x est prédecesseur de y, et que y est successeur de x.

Pour $x \in S$, on notera $\mathcal{V}^-(x)$ l'ensemble de ses prédécesseurs, que l'on appellera **voisinage entrant**, et $\mathcal{V}^+(x)$ l'ensemble de ses successeurs, que l'on appellera **voisinage sortant**. On note $\mathbf{deg}^-(x) = |\mathcal{V}^-(x)|$, on l'appelle **degré entrant** : c'est le nombre de flèches qui entrent dans x. On note de même $\mathbf{deg}^+(x) = |\mathcal{V}^+(x)|$, on l'appelle **degré sortant** : c'est le nombre de flèches qui sortent de x.

Exemple 2. Le graphe dessiné ci-dessus est donc le graphe $G = (S, A)$ orienté suivant :

— $S = \{$ Feu, Eau, Plante, Vol, Glace, Dragon $\}$

— $A = \{$ (Feu, Plante), (Feu, Glace), (Eau, Feu), (Plante, Eau), (Vol, Plante), (Glace, Plante), (Glace, Vol), (Glace, Dragon), (Dragon, Dragon) $\}$

Le type Plante a un degré entrant de 3, et un degré sortant de 1. Le graphe comporte une boucle, car il y a un arc entre le type Dragon et lui-même.

Proposition 1. Soit $G = (S, A)$ un graphe non-orienté, sans boucle. On note $n = |S|, m = |A|$. Alors :

$$\sum_{x \in S} \mathbf{deg}(x) = 2m$$

Proposition 2. Soit $G = (S, A)$ un graphe orienté, sans boucle. On note $n = |S|, m = |A|$. Alors :

$$\sum_{x \in S} \mathbf{deg}^-(x) = \sum_{x \in S} \mathbf{deg}^+(x) = m$$

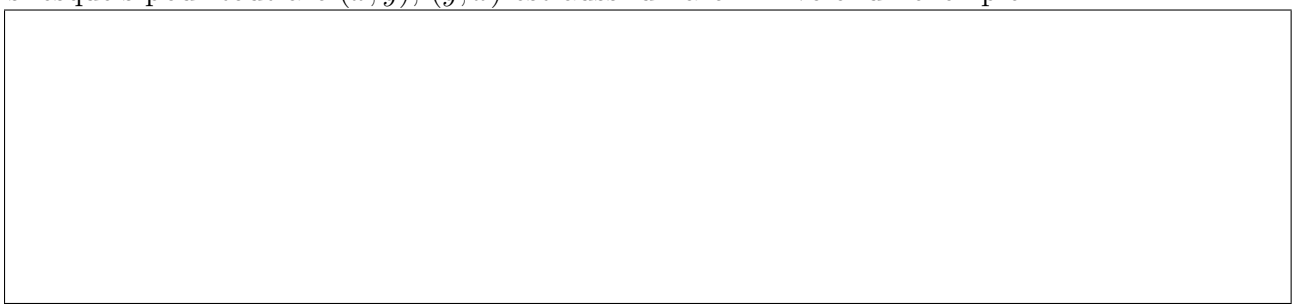
Exercice 1. Un graphe est dit régulier si tous ses sommets sont de même degré.

Question 1. Donner un graphe régulier de degré 2, et un autre de degré 3.

Question 2. Soit G un graphe régulier, de degré d, avec n sommets et m arêtes. Donner un lien entre n, m, d.

Question 3. Décrire à quoi ressemblent les graphes réguliers de degré 2.

Graphe non-orienté Parfois, le terme “graphe non-orienté” désigne les graphes orientés dans lesquels pour tout arc (x, y), (y, x) est aussi un arc. En voici un exemple :



Dans la suite, que l'on parle de graphe orienté ou non-orienté, on écrira (x, y) pour parler des arcs / arêtes, et jamais $\{x, y\}$.

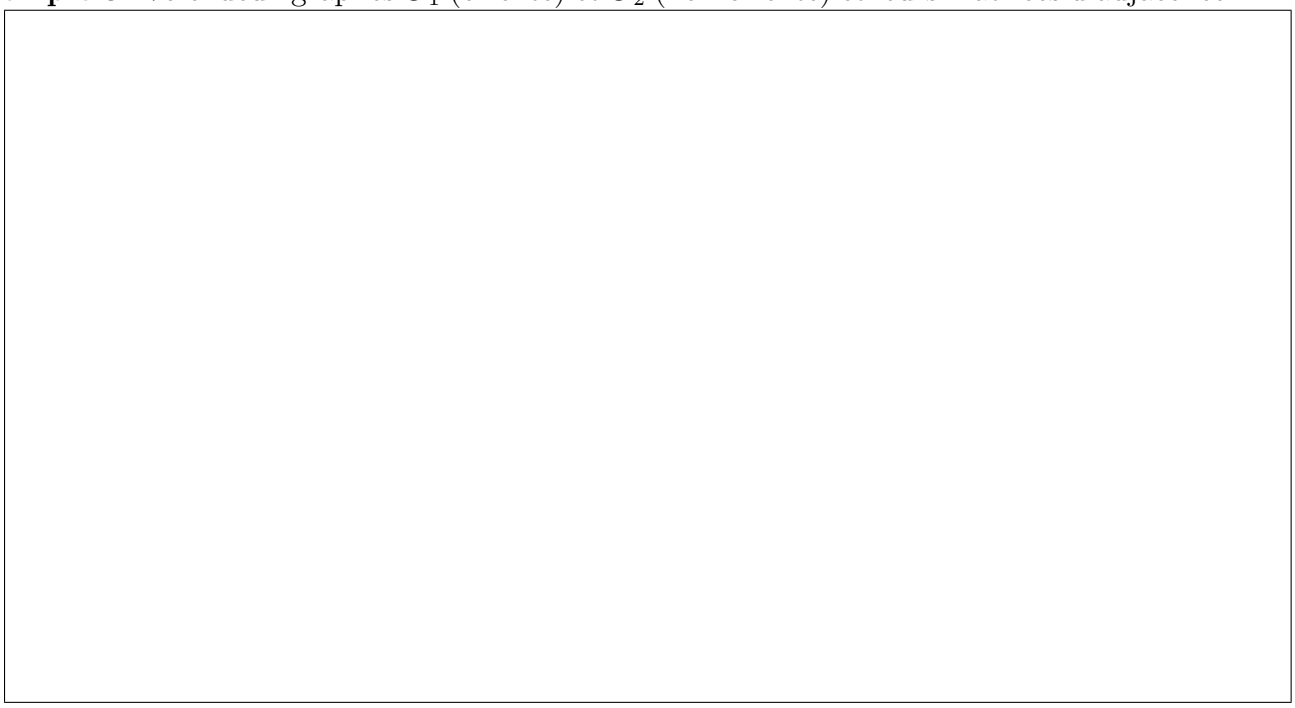
B Représentation en mémoire

Dans cette partie, on considère que nos graphes ont comme ensembles de sommets des intervalles de la forme $[\![1, n]\!]$ avec $n \in \mathbb{N}$. Cela permettra de simplifier les structures pour une première approche, mais on verra en TP comment généraliser les notions vues pour permettre aux graphes d'avoir des ensembles de sommets quelconques.

Lorsque l'on représente des graphes dans un programme, on utilise généralement une des deux implémentations concrètes suivantes : les matrices d'adjacence et les listes d'adjacence.

Matrice d'adjacence On considère un graphe $G = (S, A)$, avec $S = \{1, \dots, n\}$. La matrice d'adjacence de G est $M = (m_{ij})_{1 \leq i,j \leq n}$ avec $m_{ij} = 1$ si (i, j) est un arc, 0 sinon.

Exemple 3. Voici deux graphes G_1 (orienté) et G_2 (non-orienté) et leurs matrices d'adjacence :



Remarque 1. La matrice d'adjacence d'un graphe non-orienté est symétrique.

On peut donc représenter un graphe $G = (S, A)$ avec $|S| = n$ par un tableau 2D de dimensions $n \times n$ stockant les coefficients de la matrice d'adjacence. La taille nécessaire est $\Theta(n^2)$. Intéressons nous à la complexité d'opérations simples :

- Étant donné $s, t \in S$, déterminer si (s, t) est une arête : $\mathcal{O}(1)$. Il suffit de regarder la case correspondante de la matrice d'adjacence.
- Étant donné $s \in S$, déterminer la liste des voisins de s : $\mathcal{O}(n)$. On parcourt la liste des sommets de G, en regardant pour chaque sommet $t \neq s$ si (s, t) est une arête.

Exercice 2. Implémenter les deux fonctions précédentes en python, et les tester sur un petit graphe :

```
1 def est_arete(M, i, j):
2   """ Renvoie un booléen indiquant s'il y a une arête entre les sommets
3   i et j dans le graphe dont M est la matrice d'adjacence """
4
5 def voisins(M, i):
6   """ Renvoie la liste des voisins du sommet i dans le graphe dont M
7   est la matrice d'adjacence """
```

Listes d'adjacence On considère un graphe orienté $G = (S, A)$, avec $S = \{1, \ldots, n\}$. La représentation de G par listes d'adjacence de G est la donnée, pour chaque sommet $i \in S$, de la liste des successeurs de i, ce que l'on appelle sa liste d'adjacence. La représentation de G par listes d'adjacence est donc un tableau de taille n, contenant les listes des voisins de chaque sommet.

Exemple 4. Voici deux graphes G_1 (orienté) et G_2 (non-orienté) et leurs représentations par listes d'adjacence :

On remarque que pour un graphe non-orienté, si deux sommets (s, t) forment une arête, alors s est dans la liste d'adjacence de t, et t est dans la liste d'adjacence de s.

Cette représentation demande un espace proportionnel à $\sum_{i=1}^{n}(1+\mathbf{deg}(s_i)) = n+m$, car il faut stocker la liste d'adjacence (éventuellement vide) de chaque sommet. Étudions les complexités des deux opérations simples étudiées pour les matrices d'adjacences.

- Étant donné $s, t \in S$, déterminer si (s, t) est une arête : $\mathcal{O}(\mathbf{deg}(s))$. On parcourt la liste d'adjacence de s pour y chercher t. Si le graphe est non-orienté, on peut chercher également dans la liste d'adjacence de t pour y chercher s, et donc en choisissant la liste la plus courte, on obtient une complexité en $\mathcal{O}(1 + \min(\mathbf{deg}(s), \mathbf{deg}(t)))$.
- Étant donné $s \in S$, déterminer la liste des voisins de s : $\mathcal{O}(\mathbf{deg}(s))$. On copie la liste d'adjacence de s.

Exercice 3. Implémenter les deux fonctions précédentes en python, et les tester sur un petit graphe :

```
1 def est_arete(M, i, j):
2   """ Renvoie un booléen indiquant s'il y a une arête entre les sommets
3   i et j dans le graphe dont L est le tableau des listes d'adjacence
4   """
5
6 def voisins(M, i):
7   """ Renvoie la liste des voisins du sommet i dans le graphe dont M
8   est le tableau des listes d'adjacence """
```

Exercice 4. Quel est le nombre maximal d'arêtes dans un graphe à n sommets ?

Pour déterminer quand utiliser quelle représentation, il faut donc réfléchir aux opérations que l'on veut effectuer sur le graphe, ainsi qu'à la structure du graphe. S'il est très dense, c'est à dire s'il contient beaucoup d'arêtes, alors m est de l'ordre de n^2, donc les deux représentations prennent la même place en mémoire. Cependant, si le graphe est creux, c'est à dire si m est très faible devant n^2, alors la représentation par listes d'adjacence est bien plus légère.

2 Accessibilité

Les questions d'accessibilité dans un graphe portent sur la manière dont les différentes parties du graphe sont connectées : est-il possible de passer d'un sommet à un autre en suivant des arêtes / arcs, est-il possible de passer de tout sommet à tout autre sommet, etc...

Définition 3. Soit $G = (S, A)$ un graphe orienté, et $s, t \in S$. Un **chemin** entre s et t dans G est une suite de sommets $s_0 s_1 \dots s_k$ tels que :

- $s_0 = s$
- $s_k = t$
- $\forall i \in [\![1, k]\!], (s_{i-1}, s_i) \in A$

La longueur d'un chemin est le nombre d'**arcs** qu'il comporte, avec les notations introduites, c'est k.

Pour $0 \leq i \leq j \leq k$, le chemin $s_i \dots s_j$ est un **sous-chemin** de $s_0 \dots s_k$.

On étend ces définitions de manière analogue aux graphes non-orientés. Parfois, pour les graphes non-orientés, on utilise le terme "**chaîne**" plutôt que "chemin".

Définition 4. Soit $G = (S, A)$ un graphe, et $s \in S$. Un chemin entre s et s est appelé un **circuit**. Parfois, pour les graphes non-orientés, on utilise le terme "**cycle**".

On dit qu'un chemin est **élémentaire** s'il ne contient aucun circuit, autrement dit si aucun de ses sous-chemuns n'est un circuit.

Exemple 5. Dans le graphe G suivant, il existe un circuit de longueur 4 : 2-4-3-5-2. Il existe donc une infinité de chemins entre 1 et 6 : 1-2-6 qui est de longueur 2, 1-2-4-3-5-2-6 qui est de longueur 6, etc...

A Composantes connexes

On considère des graphes non-orientés.

Remarque 2. Soit $G = (S, A)$ un graphe. Pour $s, t \in S$, on note $s \leftrightarrow t$ s'il existe un chemin entre s et t.

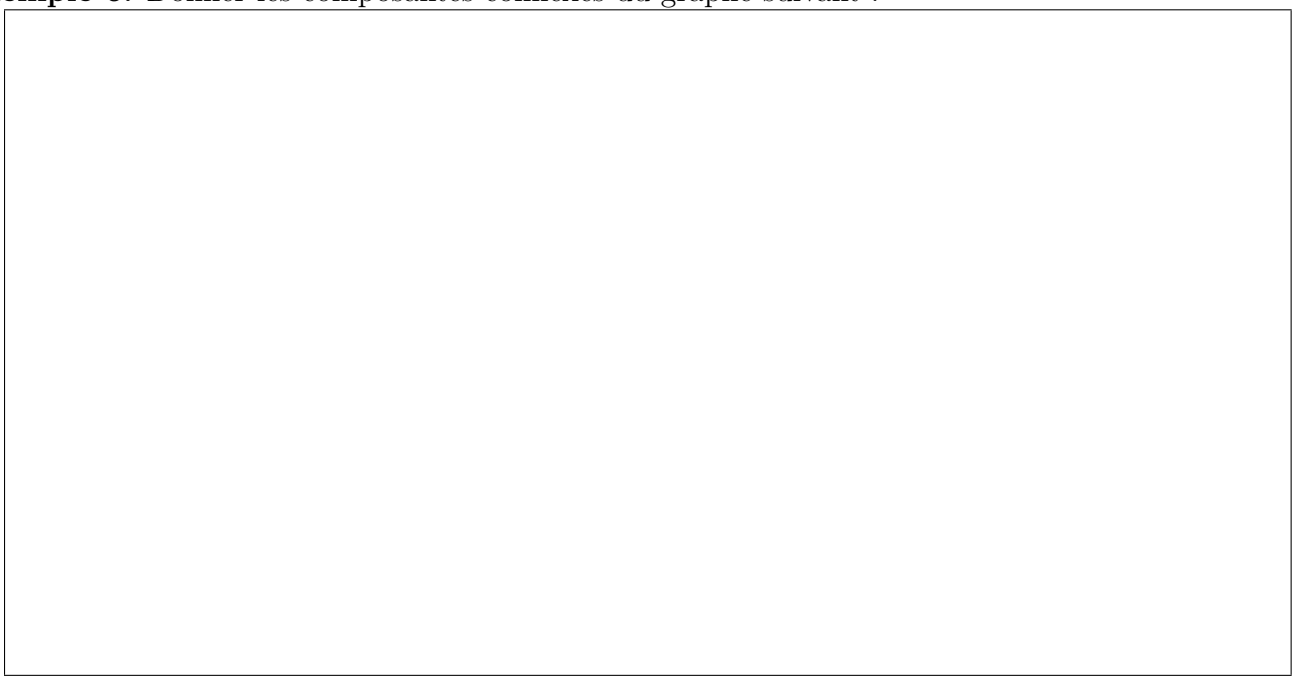
Définition 5. Une composante connexe d'un graphe $G = (S, A)$ est un ensemble $C \subseteq S$ tel que :

— $\forall x, y \in C$, il existe un chemin entre x et y ;

— $\forall x \in C, \forall y \notin C$, il n'existe pas de chemin entre x et y.

Les composantes connexes de G sont donc les ensembles maximaux de sommets étant reliés entre eux. On peut les voir comme les classes d'équivalence de la relation $\leftrightarrow$.

Exemple 6. Donner les composantes connexes du graphe suivant :



Remarque 3. En tant que classes d'équivalences d'une relation d'équivalence, les composantes connexes d'un graphe $G = (S, A)$ forment une partition de l'ensemble des sommets S.

Définition 6. On dit qu'un graphe $G = (S, A)$ est connexe s'il ne possède qu'une seule composante connexe, i.e. si pour tout couple de sommets $(x, y) \in S^2$, il existe un chemin entre x et y.

B Parcours de graphe non orienté

Un parcours de graphe est une procédure permettant de visiter chaque sommet d'un graphe une unique fois. Ils sont très utiles, et permettent par exemple de calculer les composantes connexes d'un graphe.

Étudions deux algorithmes classique : le parcours en profondeur, et le parcours en largeur. On suppose que l'on a implémenté les graphes en python et que l'on dispose des trois fonctions suivantes :

- `taille(g)` renvoie le nombre de sommets du graphe g
- `liste_sommets(g)` renvoie la liste des sommets de g
- `est_arete(g, s, t)` détermine si (s, t) est une arête de g
- `voisins(g, s)` renvoie la liste des voisins de s dans le graphe g

Préliminaire : ensembles en Python En Python, on peut créer des `set`, qui permettent de stocker des ensembles d'objets. Ce type a de nombreuses opérations, mais les suivantes nous suffisent pour le moment :

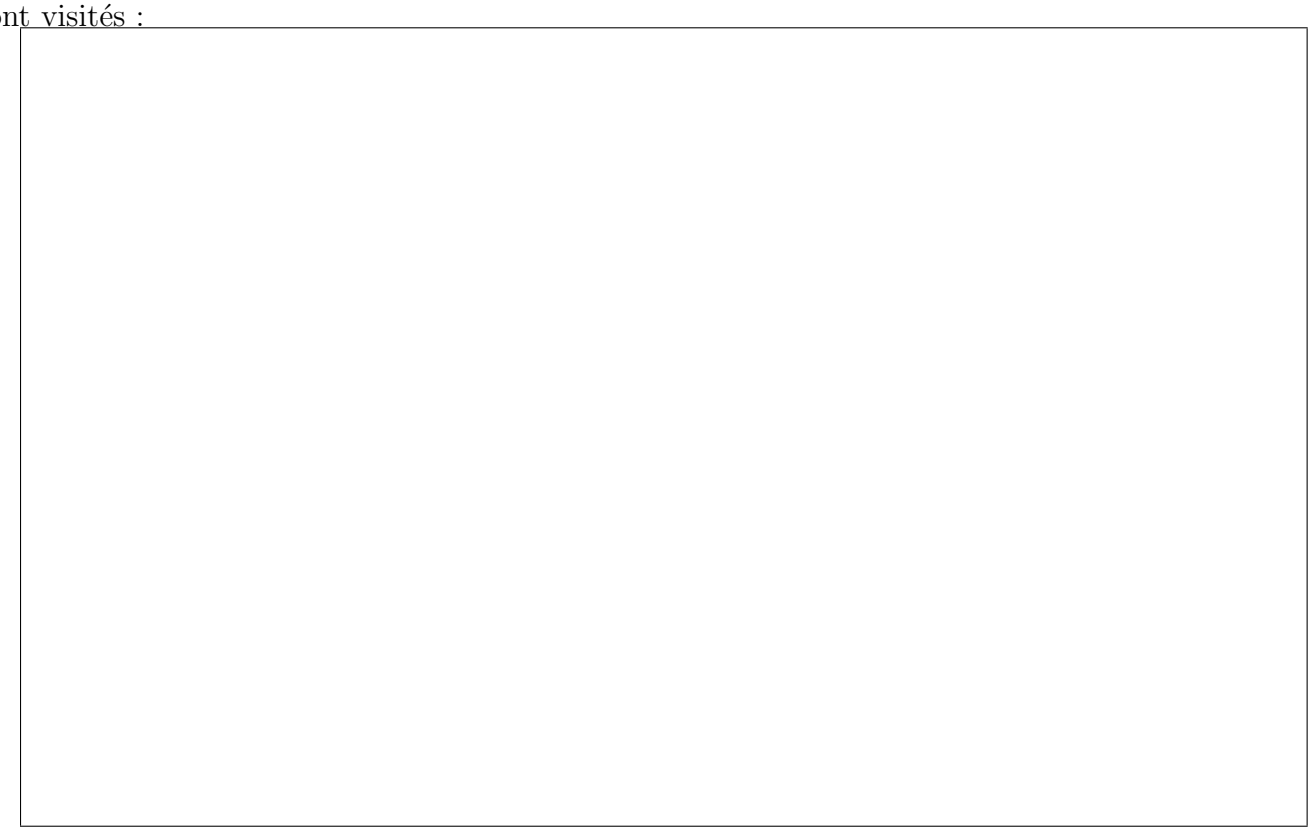
- `set()` crée un ensemble vide
- `e.add(s)` ajoute s à l'ensemble e
- `e1.update(e2))` ajoute tous les éléments de l'ensemble e_2 à l'ensemble e_1
- `list(e)` renvoie la liste des éléments de e dans un ordre arbitraire.

Les ensembles sont implémentés par tables de hachage, on pourra donc considérer que toutes les opérations sont de complexité $\mathcal{O}(1)$ (en réalité, les opérations sont **en moyenne** en $\mathcal{O}(1)$.

Parcours en profondeur Cet algorithme utilise une **pile**, i.e. une structure First In Last Out. Il consiste à partir d'un sommet, et utiliser la pile pour se souvenir des sommets que l'on n'a pas encore visité. Tant que la pile n'est pas vide, on dépile le sommet au sommet de la pile, et en empile tous ses voisins. Afin de ne pas se bloquer dans une boucle infinie, on utilise un ensemble pour se rappeler des sommets que l'on a déjà vu.

```
1  def parcours_profondeur_source(g, s):
2    """
3    Parcourt le graphe g en profondeur, à partir du sommet s. Affiche les
4    sommets visités dans l'ordre, et renvoie l'ensemble des sommets visités.
5    """
6    P = [] # pile
7    vu = set() # ensemble des sommets vus
8    P.append(s)
9    vu.add(s)
10
11   while len(P) > 0:
12     u = P.pop()
13     print(f"Sommet {u} visité")
14     for v in voisins(g, u):
15       if v not in vu:
16         # nouveau sommet à rajouter à visiter
17         P.append(v)
18         vu.add(v)
19   return vu
```

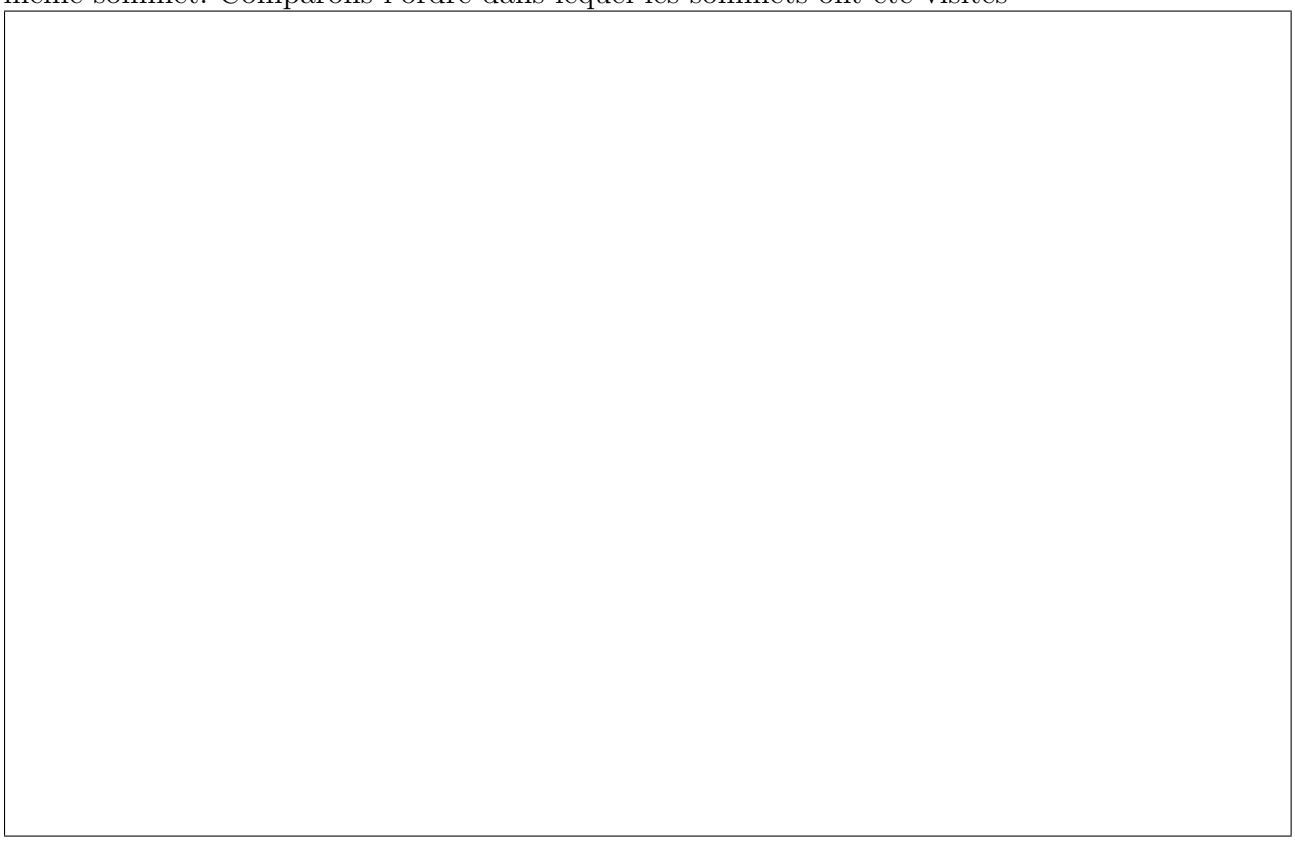
Appliquons cet algorithme sur le graphe suivant, et notons l'ordre dans lequel les sommets sont visités :



Étudions la complexité. Un sommet ne peut être empilé sur P que s'il n'a pas été visité, auquel cas on le marque comme visité. Ainsi, chaque sommet n'est empilé, et donc dépilé, qu'une seule fois au plus. De plus, pour chaque sommet, lorsqu'on le dépile, on doit parcourir la liste de ses voisins. En utilisant une matrice d'adjacence, cela prendrait $\mathcal{O}(n)$ par sommet, soit un total de $\mathcal{O}(n^2)$. En utilisant une liste d'adjacence, lorsqu'on dépile un sommet $x \in S$, parcourir la liste des voisins de x prend un temps $\mathcal{O}(\mathbf{deg}(x))$. Donc, la complexité totale est $\mathcal{O}(\sum_{x \in S}(1 + \mathbf{deg}(x)))$, i.e. $\mathcal{O}(n + m)$

Parcours en largeur En remplaçant la pile par une file, on obtient un autre type de parcours, appelé **parcours en largeur**.

Reprenons le graphe précédent, et appliquons l'algorithme de parcours en largeur, en partant du même sommet. Comparons l'ordre dans lequel les sommets ont été visités



Remarque 4. Le parcours en largeur s'appelle ainsi car il visite d'abord le sommet de départ, puis tous les sommets adjacents au sommet de départ, puis tous les sommets à distance 2 du sommet de départ, puis ceux à distance 3, et ainsi de suite... Il effectue donc des balayages en largeur successifs. Au contraire, le parcours en profondeur s'éloigne immédiatement le plus possible du sommet de départ, vers les profondeurs du graphe.

Les parcours sont fortement liés aux composantes connexes. En effet, les deux parcours vus ne peuvent pas sauter d'une composante connexe à une autre. Pour obtenir un parcours de graphe complet, on va parcourir le graphe depuis chaque sommet. Afin de ne pas parcourir une composante connexe plusieurs fois, on met en commun l'ensemble des sommets vus :

```
1  def parcours_profondeur(g):
2    """
3    Parcourt le graphe g en profondeur.
4    Affiche les sommets visités dans l'ordre
5    """
6    vu = set() # ensemble de tous les sommets visités
7    for s in liste_sommets(g):
8      if s not in vu:
9        c = parcours_profondeur_source(g, s) # nouveaux sommets visités
10       vu.update(c)
```

Appliquons l'algorithme sur le graphe suivant :

Exercice 5. Modifier la fonction `parcours_profondeur` précédente pour obtenir une fonction calculant les composantes connexes d'un graphe G :

```
1 def composantes_connexes(g):
2   """
3   Renvoie une liste des composantes connexes de g. Chaque composante
4   connexe est donnée par une liste de sommets (dans un ordre quelconque).
5   """
```

3 Plus court chemin

A Calcul de chemin

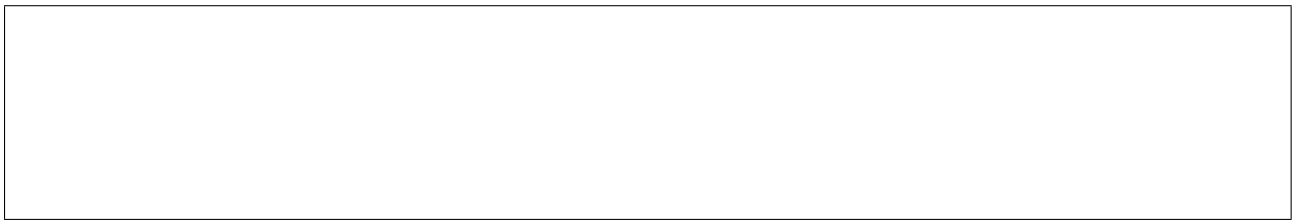
Effectuons un parcours en profondeur du graphe suivant, et notons les arêtes utilisées lors du parcours, i.e. les arêtes utilisées pour ajouter un sommet sur la pile :

On a déjà constaté qu'étant donné un graphe G, parcourir G à partir d'un sommet s permet de trouver tous les sommets t accessibles à partir de s. Mais en plus, le parcours construit toutes les informations nécessaires pour donner un **chemin** de s vers n'importe quel sommet accessible.

Plus précisément, lorsque l'on parcourt un graphe, on peut stocker, pour chaque sommet v visité, **depuis quel sommet** il a été visité. On construira donc un dictionnaire `par`, tel que `par[v]` est le sommet qui a permis de visiter v, i.e. son parent :

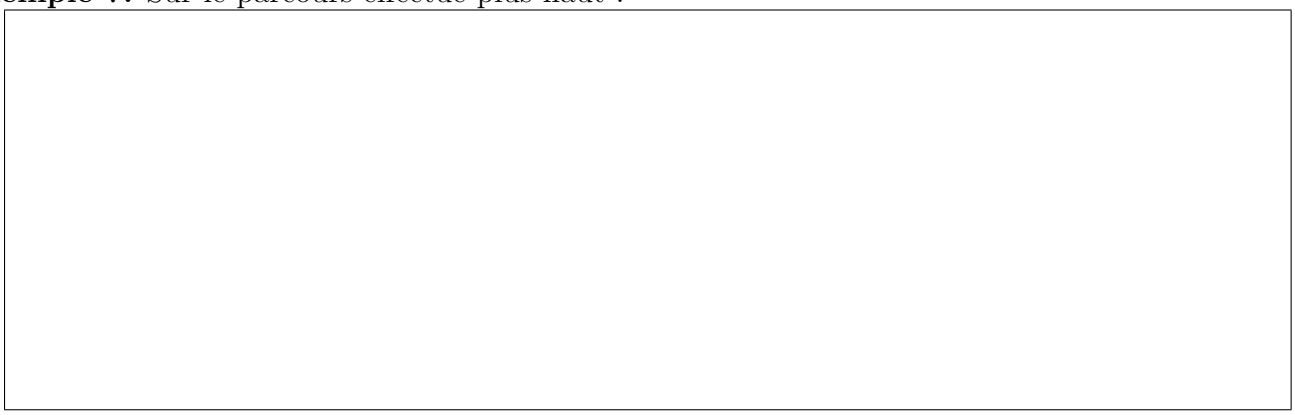
```
1  def parents(g, s):
2    """ Parcourt le graphe g depuis le sommet s, et renvoie un
3    couple (vu, par) avec vu l'ensemble des sommets visités, et
4    par un dictionnaire associant à chaque sommet visité son parent """
5    par = dict()
6    vu = set()
7    P = []
8    vu.add(s)
9    P.append(s)
10   while len(P) > 0:
11     u = P.pop()
12     for v in liste_voisins(u):
13       if u not in vu:
14         P.append(v)
15         vu.add(v)
16         par[v] = u
17   return vu, par
```

Par exemple, sur le graphe précédent, le dictionnaire des parents construit sera :



Étant donné un sommet source s, et *par* le dictionnaire des parents ainsi construit par un parcours de graphe depuis s, on peut reconstituer un chemin depuis s vers n'importe quel sommet cible t. Pour cela, il suffit d'utiliser *par* et de remonter la piste à l'envers : on part de t, on regarde quel est son parent t', puis quel est le parent de t', etc... jusqu'à avoir atteint la source s.

Exemple 7. Sur le parcours effectué plus haut :



Algorithme de reconstruction : en TP.

Une propriété fondamentale des parcours en **largeur** (avec une file) est qu'ils permettent de calculer des **plus courts chemins**. Ainsi, si l'on calcule le dictionnaire des parents depuis un sommet source s en utilisant un parcours en largeur, pour tout sommet t accessible depuis s, le chemin reconstruit sera un chemin de longueur minimal entre s et t.

B Graphes pondérés

Nous avons vu dans la partie précédente comment calculer des plus courts chemins dans des graphes. Cependant, les graphes étudiés jusqu'à présent sont trop simples, et ne permettent pas de représenter fidèlement certaines situations. Par exemple, si l'on considère un graphe dont les sommets sont des villes, et dont les arêtes représentent des lignes de train, le parcours en largeur permettra de calculer des chemins entre les villes passant par le moins de villes possible, alors que ce qui nous intéresse est plutôt de calculer des chemins dont le temps de trajet est minimal. Nous allons étendre notre définition des graphes pour permettre d'ajouter des informations sur les arêtes : une longueur, un coût, une quantité de ressources, etc...

Définition 7. Un graphe pondéré est un triplet $G = (S, A, w)$ avec (S, A) un graphe et $w : A \to \mathbb{R}$ que l'on appelle pondération, ou fonction de poids.

Ce nouveau type de graphe, qui peut être orienté ou non, permet de représenter certaines situations plus fidèlement que les graphes, car chaque liaison entre deux sommets est pondérée.

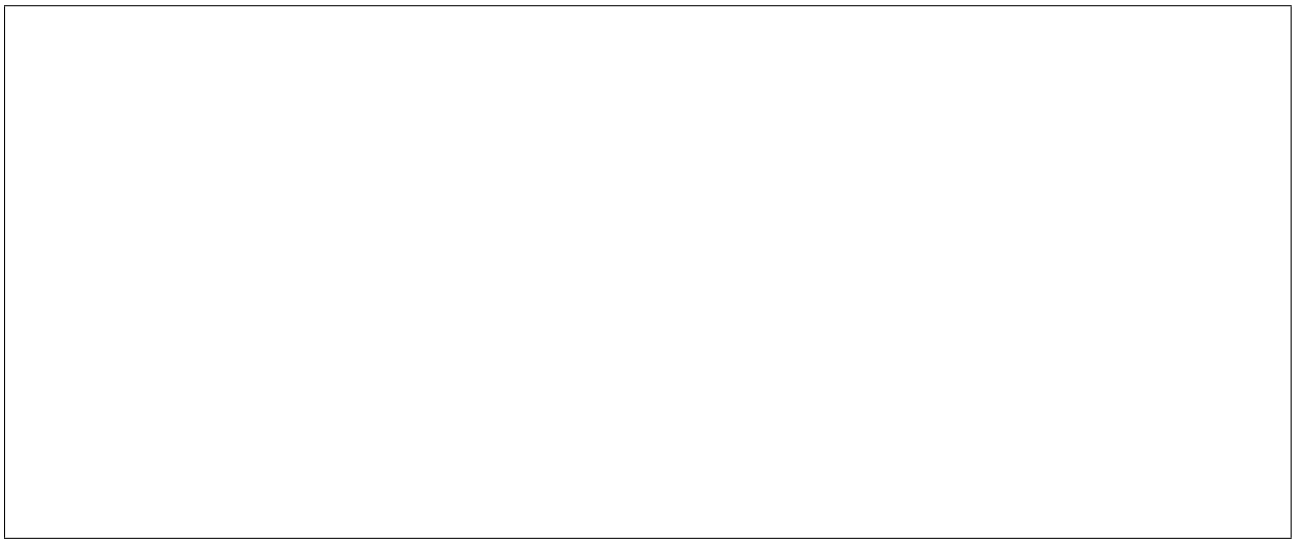
Exemple 8. Un exemple de graphe pondéré :

Concrètement, une pondération peut correspondre à un coût : $w(u, v)$ est le coût de passer de u à v, ou bien une capacité : $w(u, v)$ représente le débit maximal entre u et v, ou bien tout autre type de donnée imaginable.

Cette année, on considère le cas simple où la pondération correspond à un coût ou bien à une distance. Par convention, lorsque u et v sont des sommets mais (u, v) n'est pas un arc, on posera $w(u, v) = +\infty$.

Représentation mémoire Pour représenter en mémoire un graphe pondéré, on peut adapter les matrices et les listes d'adjacences vues plus tôt. Pour les matrices d'adjacence, on peut considérer w comme une matrice, et stocker ses coefficients dans un tableau 2D. Pour les listes d'adjacence, plutôt que de stocker, pour un sommet u donné, la liste de ses voisins / successeurs, on stocke des couples (v, d), où v est un voisin de u et d le poids de l'arête (u, v).

Exemple 9. Voici le graphe précédent sous forme de matrice d'adjacence, puis de liste d'adjacence :



Définition 8. Soit $G = (S, A, w)$ un graphe pondéré et C un chemin dans G. Le poids de C est la somme des poids de ses arêtes.

Un plus court chemin entre deux sommets u et v de G est un sommet de poids minimal.

Il n'existe pas toujours de plus court chemin entre deux sommets. En effet, si le graphe contient des arêtes de poids négatifs, alors il se peut que le graphe contienne un cycle $u_0u_1 \dots u_{k-1}u_0$ de poids négatif, qui peut alors être emprunté autant de fois que l'on veut, pour diminuer la longueur totale d'un chemin passant par un sommet de cycle.

C Algorithme de Dijkstra

L'algorithme de Dijkstra permet de calculer des plus courts chemins dans un graphe, mais nécessite qu'aucune arête du graphe ne soit négative. Il existe d'autres algorithmes qui fonctionnent quand même sans cette contrainte, et qui permettent même de détecter la présence de cycles de poids négatifs dans un graphe Par exemple, l'algorithme de Floyd-Warshall ou l'algorithme de Bellman-Ford. Ils ont néanmoins de moins bonnes complexités.

L'algorithme de Dijkstra est une modification du parcours en largeur, permettant de prendre en compte que les arêtes sont pondérées. Reprenons le principe du parcours en largeur. On considère un graphe $G = (S, A)$ et $s \in S$ un sommet de départ du parcours. On considère s et ses voisins. Un voisin u de s est à distance exactement 1 de s. Une fois que l'on a visité tous les voisins de s, les prochains sommets à explorer sont les voisins des voisins de s, qui sont exactement à distance 2 de s, et ainsi de suite.

Considérons maintenant un graphe $G = (S, A, w)$ pondéré, avec w à valeurs dans R^+, et $s \in S$ un sommet de G. Considérons les voisins de s. Parmis eux, on considère le sommet u tel que $w(s, u)$ est minimal. Alors, on est certain que le chemin $s \to u$ est un plus court chemin. En effet, tout autre chemin doit emprunter un autre voisin de s, et est donc au moins plus long. De plus, u est le sommet le plus proche de s du graphe, à part s lui-même.

Le principe de l'algorithme de Dijkstra est d'identifier un à un les sommets les plus proches de s, comme dans un parcours en largeur. Regardons le pseudo-code de l'algorithme.

Algorithme 1 : PCC : Dijkstra

Entrée(s) : $G = (S, A, w)$ graphe pondéré à n sommets, $s \in S$
Sortie(s) : **d** tableau des distances depuis s, et **Pred** tableau des prédecesseurs

```
1  d ← dictionnaire avec S comme clés, et ∞ pour toutes les valeurs;
2  Pred ← dictionnaire vide;
3  d[s] = 0;
4  Q ← ensemble contenant chaque élément de S;
5  tant que Q non vide faire
6      u ← extraire sommet de Q avec d[u] minimal;
7      pour v voisin de u faire
8          si d[u] + w(u, v) < d[v] alors
9              d[v] ← d[u] + w(u, v);
10             Pred[v] ← u;
11 retourner d, Pred
```

Appliquons l'algorithme sur un exemple pour mieux comprendre comment il fonctionne :

Passons maintenant à l'étude de cet algorithme.

Terminaison La boucle tant que de l'algorithme s'exécute au plus n fois, car à chaque passage on extrait un élément de Q, qui contient chaque sommet au début de l'exécution. Autrement dit, $|Q|$ est un variant de boucle assurant la terminaison.

Complexité Q est en réalité une file de priorité, où la priorité d'un élément u est $\mathbf{d}[u]$ (plus cette quantité est faible, plus l'élément est prioritaire). On utilise trois opérations pour cette file de priorité : ajouter, extraire, et modifier la priorité. notons respectivement $A(p), E(p)$ et

$M(p)$ le coût de ces opérations sur une file à p éléments.

Alors, le coût total est $\mathcal{O}(nA(n)+nE(n)+mM(n))$. En effet, comme pour les algorithmes de parcours, chaque arête est utilisée un nombre borné de fois : une fois pour les graphes orientés, deux fois pour les graphes non-orientés. Donc, la ligne 9, qui consiste à modifier la priorité d'un élément, ne s'exécute que m fois.

Si l'on utilise une file de priorité naïve, stockée dans une liste chaînée, où l'on parcourt la liste linéairement pour extraire l'élément le plus prioritaire, alors $A(n)$ et $M(n)$ sont en $\mathcal{O}(n)$, mais $E(n)$ est en $\mathcal{O}(1)$. On obtient donc une complexité $\mathcal{O}(n^2 + mn)$.

Si l'on utilise un tas binaire (Cf. chapitre sur les arbres), alors $A(n)$, $M(n)$ et $E(n)$ sont en $\mathcal{O}(\log n)$. On obtient donc une complexité $\mathcal{O}(n \log n + m \log n) = \mathcal{O}((n + m) \log n)$.

L'implémentation des files de priorité par tas de Fibonacci est assez complexe, mais permet les complexités amorties suivantes : $E(n) = \mathcal{O}(\log n)$, $A(n) = \mathcal{O}(1) = M(n)$. On trouve donc une complexité $\mathcal{O}(n \log n + m)$.

En pratique, les tas binaires sont largement suffisant, car la constante du $\mathcal{O}$ de l'implémentation par tas de Fibonacci est assez large, et ne compense pas le facteur $\log n$ gagné pour des valeurs raisonables de n.

Correction