THÉORIE DES GRAPHES

B. Landelle

Table des matières

I Graphe non orienté **2**
1 Sommet, arête, degré . . . 2
2 Chaînes, cycles, connexité . . . 5
3 Graphe eulérien, hamiltonien . . . 7

II Graphe orienté **11**
1 Sommet, arc, degré . . . 11
2 Chemin, circuit, forte connexité . . . 13
3 Graphe eulérien, hamiltonien . . . 14

III Représentation des graphes **14**
1 Liste d'adjacence . . . 14
2 Matrice d'adjacence . . . 16
3 Implémentation . . . 18

IV Parcours en profondeur **19**
1 Principe . . . 19
2 Implémentation . . . 20
3 Un exemple . . . 23

V Parcours en largeur **27**
1 Principe . . . 27
2 Implémentation . . . 29
3 Un exemple . . . 31

VI Plus court chemin **34**
1 Graphe valué, coût . . . 34
2 Propriétés . . . 35

VII Algorithme de Dijkstra **37**
1 Principe . . . 37
2 Implémentation . . . 37
3 Un exemple . . . 41

VIII Algorithme A$\star$ **45**
1 Principe . . . 45
2 Implémentation . . . 45

I Graphe non orienté

1 Sommet, arête, degré

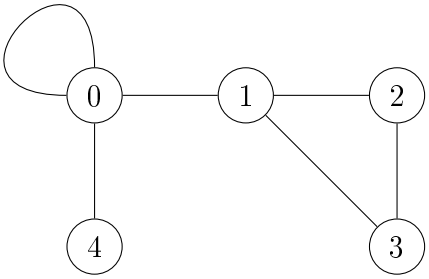
Étant donné X un ensemble, on note $\mathcal{P}_2(\mathrm{X})$ l'ensemble des parties de X à un ou deux éléments :

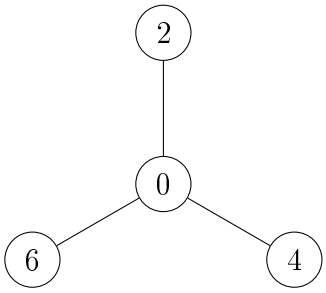
$$\mathcal{P}_2(\mathrm{X}) = \{\{x, y\}, (x, y) \in \mathrm{X}^2\}$$

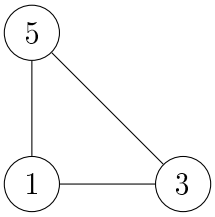
Définition 1. *Un* graphe non orienté G *est un couple* (S, A) *avec* S *un ensemble fini de* sommets *ou* nœuds *et* A *une partie de* $\mathcal{P}_2(\mathrm{S})$ *dont les éléments sont appelés* arêtes. *Le nombre de sommets de* S *est appelé* l'ordre *du graphe.*

Notations : On note $n = \mathrm{Card}\ \mathrm{S}$ et $m = \mathrm{Card}\ \mathrm{A}$.

Exemples :



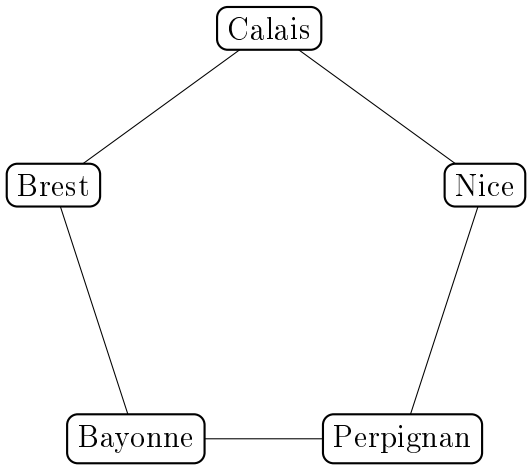


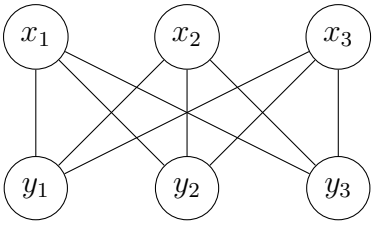


$\mathrm{S}_1 = \{0, 1, 2, 3, 4\}$
$\mathrm{A}_1 = \{\{0\}, \{0, 1\}, \{0, 4\}, \{1, 2\}, \{1, 3\}, \{2, 3\}\}$

$\mathrm{S}_2 = \{0, 1, 2, 3, 4, 5, 6\}$
$\mathrm{A}_2 = \{\{0, 2\}, \{0, 4\}, \{0, 6\}, \{1, 3\}, \{1, 5\}, \{3, 5\}\}$

L'ensemble des sommets peut être un ensemble de stations, de villes, ...





Définition 2. *Soit* $\mathrm{G} = (\mathrm{S}, \mathrm{A})$ *un graphe non orienté. On dit que deux sommets* x *et* y *de* S *sont* adjacents *si* $\{x, y\} \in \mathrm{A}$. *Les sommets* x *et* y *sont les* extrémités *de l'arête* $\{x, y\}$. *Deux arêtes sont dites* adjacentes *si elles ont une extrémité commune.*

Illustration : Les sommets x et y sont adjacents, de même que x et z. Les arêtes $\{x, y\}$ et $\{x, z\}$ sont adjacentes.

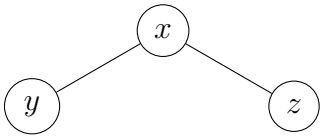


FIGURE 1 – Adjacence entre sommets, entre arêtes

Définition 3. *Soit* $\mathrm{G} = (\mathrm{S}, \mathrm{A})$ *un graphe non orienté. Une arête est* incidente *à un sommet ou un sommet est* incident *à une arête si le sommet est une des extrémités de l'arête. Une arête* $\{x, y\} \in \mathrm{S}$ *est une* boucle *si* $x = y$. *Un graphe sans boucle est dit* simple.

Remarques : (1) La notion de *multi-graphe* avec boucle et/ou multi-arête n'est pas abordée dans ce cours.

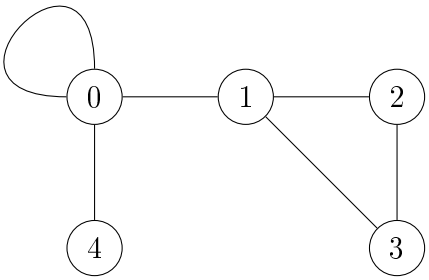


FIGURE 2 – Graphe avec une boucle

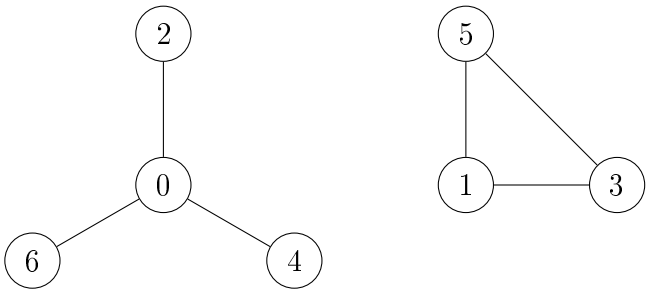


FIGURE 3 – Graphe simple

(2) Si le graphe G est simple, on a $m \leqslant \dfrac{n(n-1)}{2}$ et sinon $m \leqslant \dfrac{n(n-1)}{2} + n = \dfrac{n(n+1)}{2}$.

Définition 4. *Soit* $\mathrm{G} = (\mathrm{S}, \mathrm{A})$ *un graphe non orienté. Le* degré *d'un sommet* $x \in \mathrm{S}$ *noté* $\mathrm{d}(x)$ *est le nombre d'arêtes incidentes à* x. *Un sommet de degré nul est dit* isolé. *Par convention, on considère qu'une boucle compte pour deux arêtes incidentes sur le sommet concerné.*

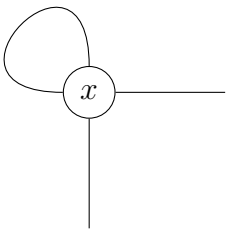


FIGURE 4 – $\mathrm{d}(x) = 4$

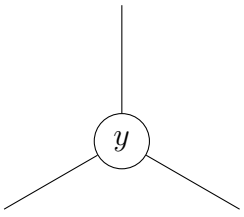
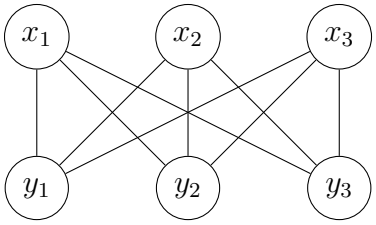


FIGURE 5 – $\mathrm{d}(y) = 3$

Exercice : On considère le graphe



1. Quelles sont les arêtes incidentes au sommet x_2 ?
2. Quelles sont les arêtes adjacentes à l'arête $\{y_1, x_2\}$?
3. Déterminer le degré de chaque sommet.

Corrigé : 1. Les arêtes incidentes au sommet x_2 sont $\{y_1, x_2\}$, $\{y_2, x_2\}$, $\{y_3, x_2\}$.
2. Les arêtes adjacentes à l'arête $\{y_1, x_2\}$ sont $\{y_1, x_1\}$, $\{y_1, x_3\}$, $\{x_2, y_2\}$, $\{x_2, y_3\}$.
3. Tous les sommets sont de degré égal à 3.

Proposition 1. *Soit* $\mathrm{G} = (\mathrm{S}, \mathrm{A})$ *un graphe non orienté. On a*

$$\sum_{x \in \mathrm{S}} \mathrm{d}(x) = 2\,\mathrm{Card}\ \mathrm{A}$$

Démonstration. On a $$\mathrm{Card}\ \mathrm{A} = \sum_{b \in \mathcal{P}_2(\mathrm{S})} \mathrm{Card}\ \{a \in \mathrm{A} \mid a = b\}$$

Considérons l'application surjective $\Phi : \mathrm{S}^2 \to \mathcal{P}_2(\mathrm{S}), (x, y) \mapsto \{x, y\}$. Pour $x \in \mathrm{S}$, l'élément $\{x\} \in \mathcal{P}_2(\mathrm{S})$ admet (x, x) pour unique antécédent et pour $(x, y) \in \mathrm{S}^2$ avec $x \neq y$, l'élément $\{x, y\} \in \mathcal{P}_2(\mathrm{S})$ admet exactement deux antécédents qui sont (x, y) et (y, x). Ainsi, on a par union disjointe

$$\begin{aligned}
2\,\mathrm{Card}\ \mathrm{A} &= 2\sum_{x \in \mathrm{S}} \mathrm{Card}\ \{a \in \mathrm{A} \mid a = \{x\}\} + \sum_{(x,y) \in \mathrm{S}^2, x \neq y} \mathrm{Card}\ \{a \in \mathrm{A} \mid a = \{x, y\}\} \\
&= 2\sum_{x \in \mathrm{S}} \mathrm{Card}\ \{a \in \mathrm{A} \mid a = \{x\}\} + \sum_{x \in \mathrm{S}} \mathrm{Card}\ \{a \in \mathrm{A} \mid \exists y \in \mathrm{S} \setminus \{x\} : a = \{x, y\}\} \\
&= \sum_{x \in \mathrm{S}} \left(2\,\mathrm{Card}\ \{a \in \mathrm{A} \mid a = \{x\}\} + \mathrm{Card}\ \{a \in \mathrm{A} \mid \exists y \in \mathrm{S} \setminus \{x\} : a = \{x, y\}\}\right) \\
2\,\mathrm{Card}\ \mathrm{A} &= \sum_{x \in \mathrm{S}} \mathrm{d}(x)
\end{aligned}$$

□

Exercice : Soit $\mathrm{G} = (\mathrm{S}, \mathrm{A})$ un graphe non orienté. Justifier qu'il y a un nombre pair de sommets de degrés impairs.

Corrigé : On a

$$\sum_{x \in \mathrm{S}} \mathrm{d}(x) \equiv 0\ [2] \quad \text{et} \quad \sum_{x \in \mathrm{S}} \mathrm{d}(x) \equiv \sum_{x \in \mathrm{S} : \mathrm{d}(x)\ \text{impair}} \mathrm{d}(x) \equiv \mathrm{Card}\ \{x \in \mathrm{S} : \mathrm{d}(x)\ \text{impair}\}\ [2]$$

Le résultat suit.

2 Chaînes, cycles, connexité

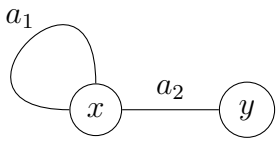
Définition 5. *Soit* $\mathrm{G} = (\mathrm{S}, \mathrm{A})$ *un graphe non orienté et* $(x, y) \in \mathrm{S}^2$*. Une* chaîne *reliant* x *à* y *est une suite finie d'arêtes consécutives reliant* x *à* y*, i.e. une suite d'arêtes du type*

$$\{x, x_1\}, \{x_1, x_2\}, \ldots, \{x_{k-1}, y\}$$

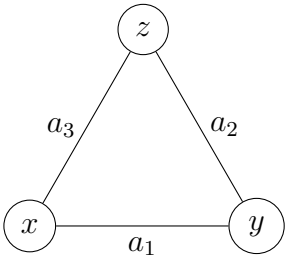
On la note $(x, x_1, x_2, \ldots, x_{k-1}, y)$

Cette chaîne est dite de longueur k *et d'extrémités* x *et* y*. Par convention, tout sommet est relié à lui-même par une chaîne de longueur nulle.*

Définition 6. *Soit* $\mathrm{G} = (\mathrm{S}, \mathrm{A})$ *un graphe non orienté. Une chaîne est dite* élémentaire *si elle ne passe pas deux fois par un même sommet et* simple *si elle ne contient pas deux fois la même arête.*



Chaîne (a_1, a_1) non simple
Chaîne (a_1, a_2) simple non élémentaire



Chaîne (a_1, a_2) élémentaire
Chaîne (a_1, a_2, a_3) simple non élémentaire

Proposition 2. *Soit* $\mathrm{G} = (\mathrm{S}, \mathrm{A})$ *un graphe non orienté. Une chaîne élémentaire est simple.*

Démonstration. Une chaîne élémentaire non simple contient deux fois la même arête et passe donc deux fois par les sommets extrémités de celle-ci ce qui est absurde. □

Remarques : (1) Par contraposée, une chaîne qui n'est pas simple n'est pas élémentaire. En revanche, la réciproque est fausse.
(2) Une chaîne élémentaire est de longueur $\leqslant n - 1$. En effet, c'est une suite d'arêtes

$$\{x_0, x_1\}, \{x_1, x_2\}, \ldots, \{x_{k-1}, x_k\}$$

avec les x_i distincts d'où $k + 1 \leqslant n$.

FIGURE 6 – Chaîne élémentaire

Proposition 3. *Soit* $\mathrm{G} = (\mathrm{S}, \mathrm{A})$ *un graphe non orienté. De toute chaîne reliant deux sommets distincts, on peut extraire une chaîne élémentaire.*

Démonstration. Soient x, y sommets distincts reliés par une chaîne

$$(x_0, x_1, x_2, \ldots, x_{k-1}, x_k)$$

avec $x_0 = x$ et $x_k = y$. Si la chaîne est élémentaire, il n'y a rien à faire. Sinon, il existe des indices i, j dans $[\![0\,;\, k]\!]$ avec $i < j$ tels que $x_i = x_j$. Alors, la chaîne

$$(x_0, x_1, \ldots, x_{i-1}, x_i, x_{j+1}, \ldots, x_{k-1}, x_k)$$

est strictement plus courte. On itère ce procédé tant que la chaîne n'est pas élémentaire. L'algorithme se termine puisque la taille de la chaîne décroît strictement à chaque étape. □

Définition 7. *Soit* $G = (S, A)$ *un graphe non orienté. Une chaîne est dite* fermée *si ses extrémité coïncident. Une chaîne fermée et simple est appelée un* cycle. *Un graphe sans cycle est dit* acyclique.

Remarque : On parle de chaîne fermée pour une chaîne de longueur non nulle.

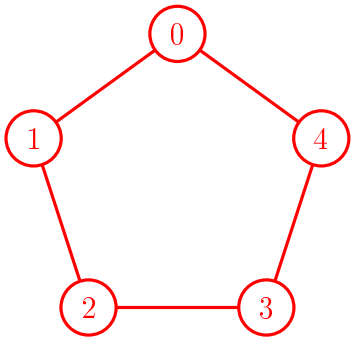


FIGURE 7 – $(0, 1, \ldots, 4, 0)$ cycle

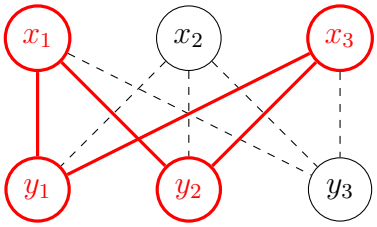


FIGURE 8 – $(x_1, y_1, x_3, y_2, x_1)$ cycle

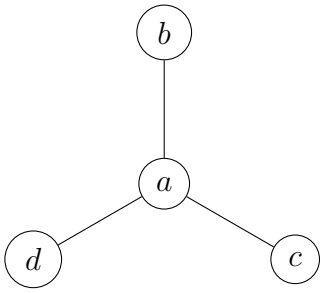


FIGURE 9 – Graphe acyclique

Définition 8. *Soit* $G = (S, A)$ *un graphe non orienté. Les sommets* x *et* y *sont dits* connectés *s'il existe une chaîne reliant* x *à* y.

Proposition 4. *Soit* $G = (S, A)$ *un graphe non orienté. La connexion entre deux sommets du graphe est une relation d'équivalence. Les classes d'équivalence pour cette relation sont appelées* composantes connexes *du graphe.*

Démonstration. Preuve immédiate. □

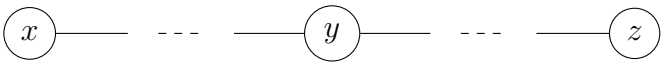


FIGURE 10 – Transitivité de la connexion

Définition 9. *Soit* $G = (S, A)$ *un graphe non orienté. Le graphe* G *est dit* connexe *si toute paire de sommets est constituée de sommets connectés.*

Remarque : Un graphe est connexe si et seulement s'il admet une unique composante connexe.

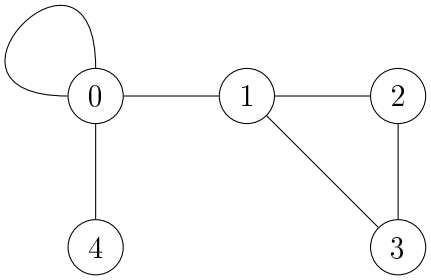


FIGURE 11 – Graphe connexe

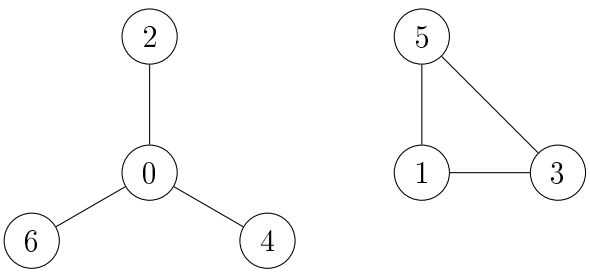


FIGURE 12 – Graphe avec deux composantes connexes

Exercice : Soit $G = (S, A)$ un graphe non orienté. Montrer que si le graphe G est connexe, alors tout sommet est de degré strictement positif.

Corrigé : Un sommet étant relié à tout autre par connexité, il existe une arête incidente à ce sommet.

3 Graphe eulérien, hamiltonien

Définition 10. *Soit* $G = (S, A)$ *un graphe non orienté. Une chaîne est dite* eulérienne *si elle contient une fois et une seule toutes les arêtes du graphe. Un* cycle eulérien *est une chaîne eulérienne qui est un cycle.*

Remarque : Une chaîne eulérienne est une chaîne simple de longueur m.

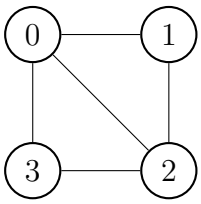
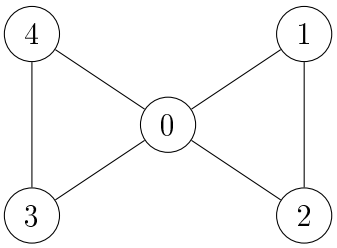


FIGURE 13 – Chaîne eulérienne $(0, 1, 2, 3, 0, 2)$



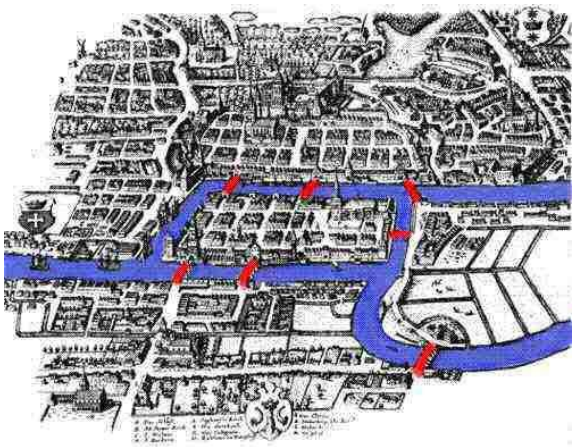
Cycle eulérien $(0, 1, 2, 0, 3, 4, 0)$

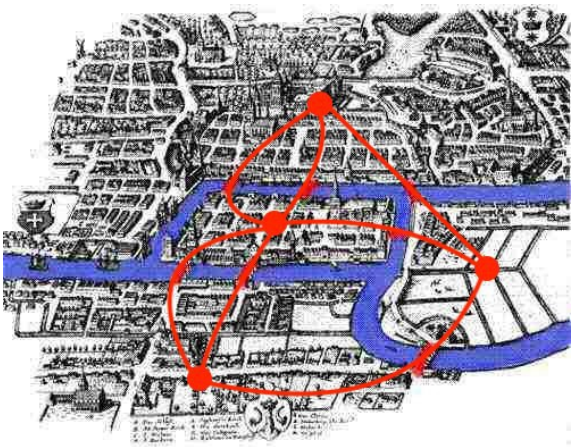
Définition 11. *Soit* $G = (S, A)$ *un graphe non orienté. Si* G *admet un cycle eulérien, alors le graphe* G *est dit* eulérien.

Commentaire : On dit qu'un graphe est eulérien si on peut le tracer, en revenant à l'origine, sans lever le crayon et sans passer deux fois sur la même arête.

Exemple historique : Le problème des sept ponts de la ville de Königsberg

Ce très célèbre problème, résolu négativement par Leonard Euler en 1736, est à l'origine de la théorie des graphes et de la topologie. Le problème consiste à déterminer s'il existe une promenade permettant, à partir d'un point de départ au choix, de passer une fois et seule par chaque pont et de revenir à son point de départ.





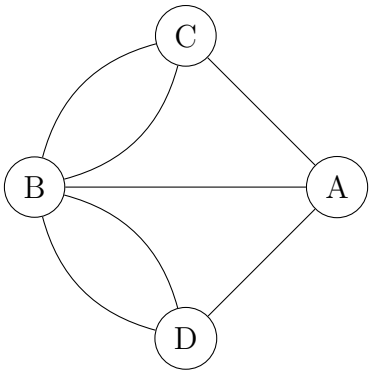


FIGURE 14 – Graphe des ponts de Königsberg

Cet exemple n'est mentionné que pour son caractère historique : le graphe associé est un multigraphe avec deux multi-arêtes entre B et C mais aussi B et D et les multi-graphes ne sont pas abordés dans ce cours.

Théorème 1 (Théorème d'Euler). *Un graphe non orienté connexe est eulérien si et seulement si tous ses sommets sont de degrés pairs.*

Démonstration. Soit G un graphe connexe. Supposons G eulérien. On parcourt un cycle eulérien de G en supprimant, lors de chaque passage sur un sommet (sauf au début et à la fin), les deux arêtes incidentes à ce sommet empruntées (celle qui arrive sur ce sommet et celle qui en sort). Excepté pour le sommet de départ ou d'arrivée, ce procédé conserve la parité des degrés des sommets. En fin de parcours, on a supprimé toutes les arrêtes et aussi celle de départ et d'arrivée d'où la conservation de parité du sommet de départ. Les degrés sont tous nuls et par conservation de parité, on en déduit que tous les sommets du graphe initial sont de degrés pairs.

Réciproquement, supposons tous les sommets de G de degrés pairs. On suppose S non vide et non réduit à un élément (sinon, c'est immédiat). Le graphe étant connexe, les degrés des sommets sont strictement positifs et donc $\geqslant 2$. Par parité des degrés, pour une arête qui arrive sur un sommet, on dispose d'une arête qui en sort. On choisit arbitrairement un sommet de départ x, on choisit arbitrairement une arête puis on réitère ce procédé en choisissant à chaque passage sur un sommet une arête non visitée. Ce choix est possible par parité des degrés : pour une arête nouvellement visitée arrivant sur un sommet, on dispose d'une arête non visitée qui en sort. Comme il y a un nombre fini de sommets, on finit par revenir au sommet de départ et on a donc construit un cycle. Considérons un cycle C de longueur maximale puis supprimons toutes les arêtes de ce cycle. La parité des sommets est inchangée. Supposons qu'il existe une arête non visitée. On dispose alors d'un sommet hors du cycle C et par connexité, on peut relier celui-ci à un sommet du cycle. Par conséquent, on peut trouver un sommet z hors du cycle adjacent à un sommet y du cycle. Considérant la composante connexe de z, on construit un cycle passant par y avec des arêtes non visitées. On peut alors construire un cycle plus grand que C en insérant le cycle passant par y avec $(x, \ldots, y, \underbrace{z, \ldots, y}_{\text{cycle additionnel}}, \ldots)$. Ceci contredit la maximalité du choix de C. On conclut que le cycle C est eulérien.

□

Remarque : L'algorithme de Hierholzer[1] permet une détermination efficace d'un cycle eulérien :

- choisir un sommet initial v_0 ;
- suivre un chemin arbitraire d'arêtes jusqu'à retourner v_0 en ne passant qu'une fois par chaque arête, constituant ainsi un cycle C ;
- tant qu'il y a un sommet v_1 dans le cycle C avec des arêtes non visitées :
  - suivre un chemin arbitraire d'arêtes pas encore choisies en ne passant qu'une fois par chaque arête jusqu'à retourner à v_1, constituant ainsi un cycle C′ ;
  - prolonger le cycle C par C′.

On obtient alors un cycle eulérien en $O(n)$.

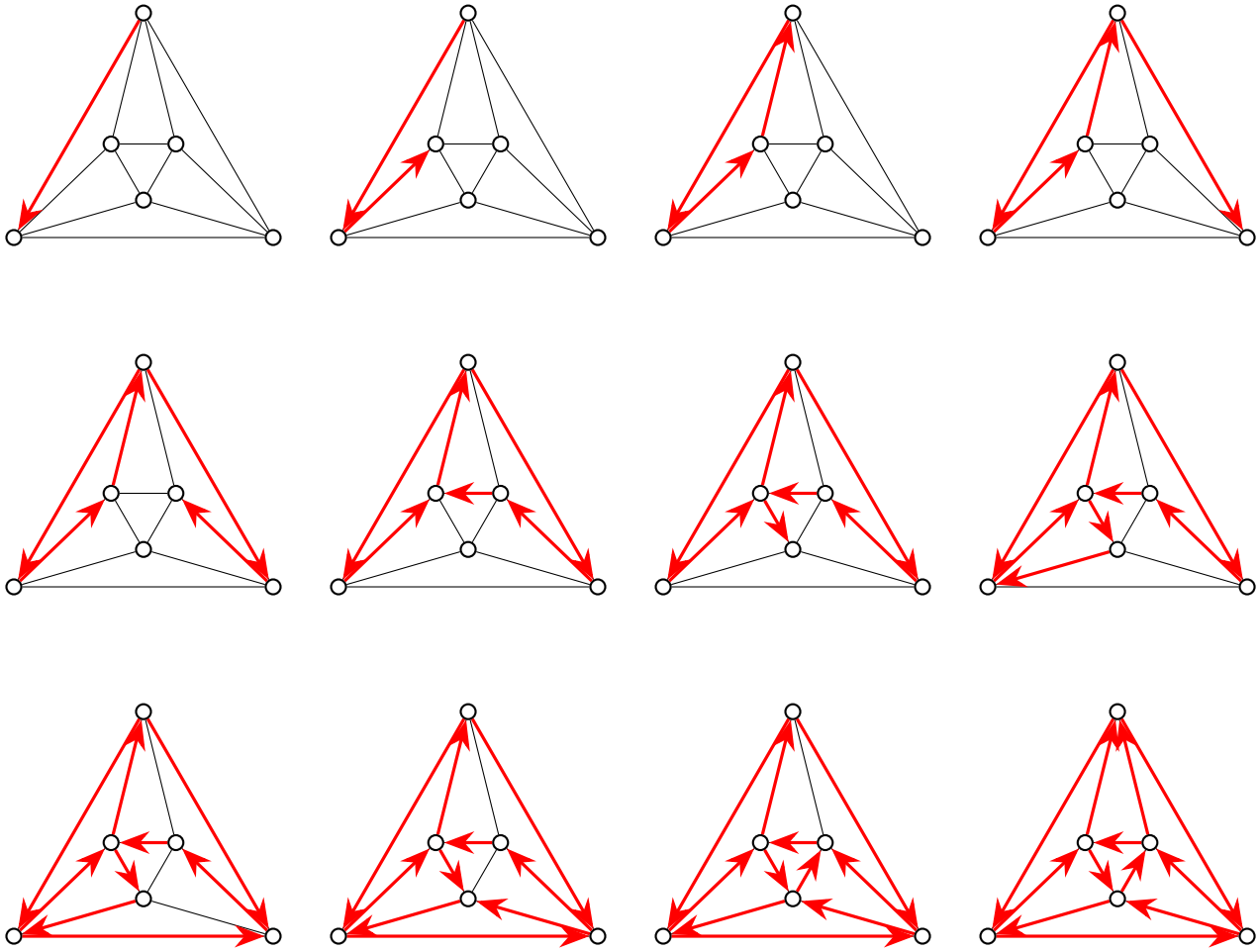


FIGURE 15 – Cycle eulérien

1. Carl Hierholzer, 1840-1871, mathématicien badois - C'est lui qui démontra le caractère suffisant de la caractérisation du théorème d'Euler.

Exercice : Parmi les graphes suivants, déterminer ceux qui sont eulériens.

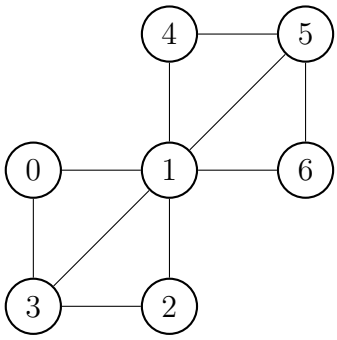


FIGURE 16 – Graphe G_1

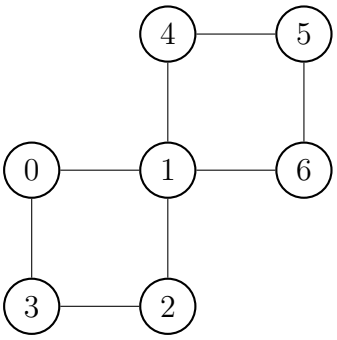


FIGURE 17 – Graphe G_2

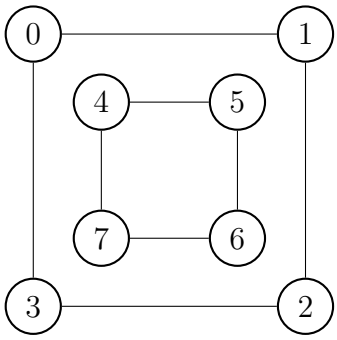


FIGURE 18 – Graphe G_3

Corrigé : Le graphe G_1 n'est pas eulérien : il est connexe mais les sommets 3 et 5 sont de degré égal à 3, impair. Le graphe G_2 est eulérien : il est connexe et tous ses sommets sont de degrés pairs. Le graphe G_3 n'est pas eulérien : tous ses sommets sont de degrés pairs mais il n'est pas connexe.

Définition 12. *Soit* $G = (S, A)$ *un graphe non orienté. Une chaîne est dite* hamiltonienne *si elle passe une fois et une seule par chaque sommet. Un* cycle hamiltonien *est un cycle dont la chaîne privée d'une extrémité est hamiltonienne.*

Remarque : Un graphe qui admet une chaîne hamiltonienne est connexe.

Définition 13. *Soit* $G = (S, A)$ *un graphe non orienté. Si le graphe* G *admet un cycle hamiltonien, alors le graphe* G *est dit* hamiltonien.

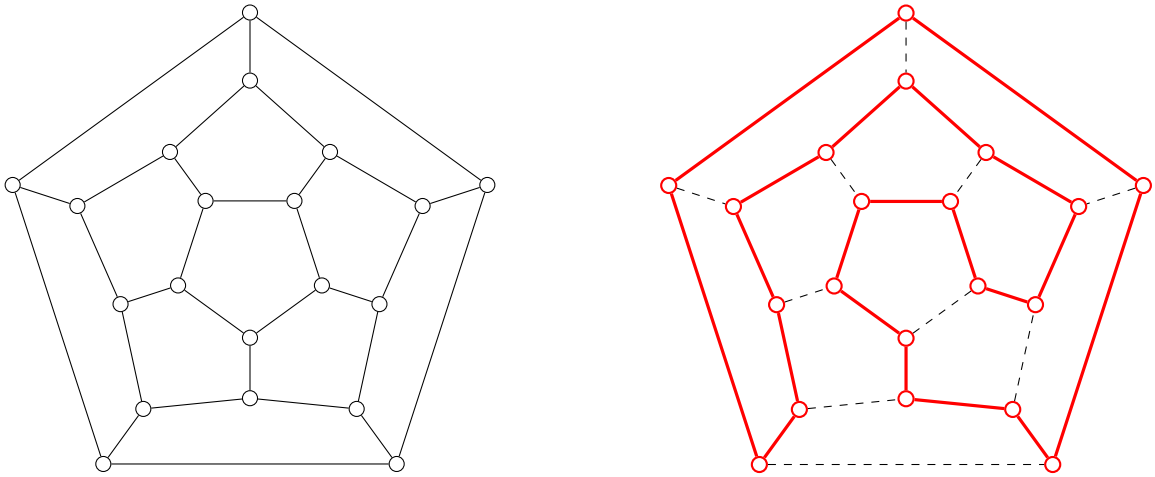


FIGURE 19 – Graphe hamiltonien

Commentaire : La recherche d'un cycle hamiltonien est un problème algorithmiquement très difficile, problème dit *NP-complet*. Le problème du *voyageur de commerce* consiste en la recherche d'un chemin hamiltonien dans un graphe complet (graphe simple tel que tout paire de sommets est reliée par une arête).

II Graphe orienté

1 Sommet, arc, degré

Définition 14. *Un* graphe orienté G *est un couple* (S, A) *avec* S *un ensemble fini de* sommets *ou* nœuds *et* A *une partie de* S^2 *dont les éléments sont appelés* arcs. *Le nombre de sommets de* S *est appelé* l'ordre *du graphe.*

Notations : On note $n = \text{Card S}$ et $m = \text{Card A}$.

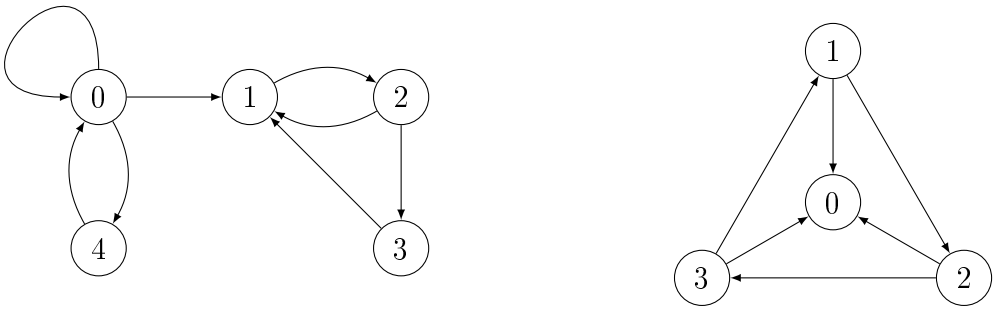


FIGURE 20 – Graphes orientés

Définition 15. *Soit* $G = (S, A)$ *un graphe orienté et* $a \in A$ *avec* $a = (x, y)$. *L'arc* a *est représenté graphiquement par* $x \to y$. *L'arc* (x, y) *est dit* sortant *ou* incident vers l'extérieur *en* x *et* entrant *ou* incident vers l'intérieur *en* y. *Le sommet* x *est l'*origine *ou la* source *de* a *et le sommet* y *est la* cible *ou l'*arrivée *de* a. *On dit que* x *est un* prédécesseur *de* y, *que* y *est un* successeur *de* x *et que* y *est* adjacent *à* x. *Deux arcs sont dits* adjacents *s'ils ont une extrémité commune.*

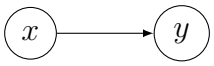


FIGURE 21 – Arc (x, y)

Définition 16. *Soit* $G = (S, A)$ *un graphe orienté. Un arc est* incident *à un sommet ou un sommet est* incident *à un arc si le sommet est une des extrémités de l'arc. Un arc* $(x, y) \in A$ *est une* boucle *si* $x = y$. *Un graphe sans boucle est dit* simple.

Remarques : (1) La notion de *multi-graphe* avec boucle et/ou multi-arc n'est pas abordée dans ce cours.

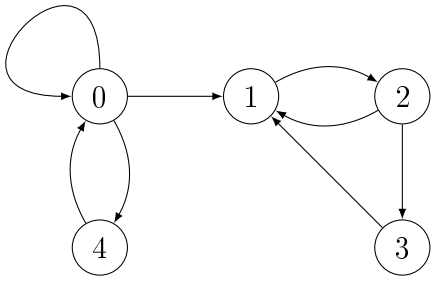


FIGURE 22 – Graphe avec boucle

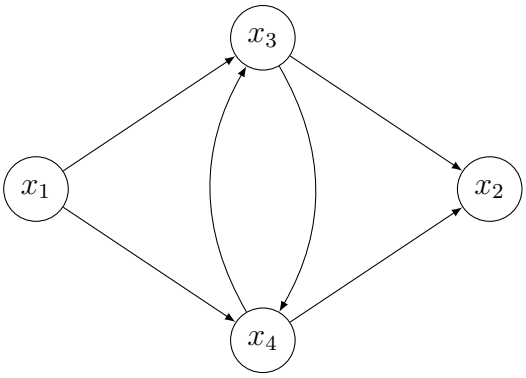


FIGURE 23 – Graphe simple

(2) Si le graphe G est simple, on a $m \leqslant n(n-1)$ et on a $m \leqslant n^2$.

Définition 17. *Soit* $\mathrm{G} = (\mathrm{S}, \mathrm{A})$ *un graphe orienté et* $x \in \mathrm{S}$. *Le* degré sortant *du sommet* x *noté* $\mathrm{d}^+(x)$ *est le nombre d'arcs sortant de* x *et le* degré entrant *du sommet* x *noté* $\mathrm{d}^-(x)$ *est le nombre d'arcs entrant en* x. *Le* degré *du sommet* x *noté* $\mathrm{d}(x)$ *est défini par* $\mathrm{d}(x) = \mathrm{d}^+(x) + \mathrm{d}^-(x)$.

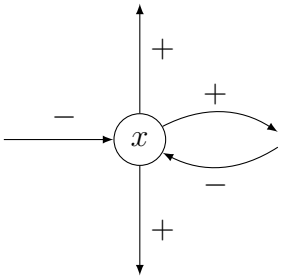


FIGURE 24 – $\mathrm{d}^+(x) = 3,\ \mathrm{d}^-(x) = 2,\ \mathrm{d}(x) = 5$

Proposition 5. *Soit* $\mathrm{G} = (\mathrm{S}, \mathrm{A})$ *un graphe orienté. On a*

$$\sum_{x \in \mathrm{S}} \mathrm{d}^+(x) = \sum_{x \in \mathrm{S}} \mathrm{d}^-(x) = \mathrm{Card}\ \mathrm{A}$$

Démonstration. On a, en observant une union disjointe pour le passage de la première à la deuxième ligne

$$\begin{aligned} \mathrm{Card}\ \mathrm{A} &= \sum_{(x,y) \in \mathrm{S}^2} \mathrm{Card}\ \{a \in \mathrm{A} \mid a = (x,y)\} \\ &= \sum_{x \in \mathrm{X}} \mathrm{Card}\ \{a \in \mathrm{A} \mid \exists y \in \mathrm{S} : a = (x,y)\} \\ \mathrm{Card}\ \mathrm{A} &= \sum_{x \in \mathrm{X}} \mathrm{d}^+(x) \end{aligned}$$

On procède de même pour les degrés entrants. □

2 Chemin, circuit, forte connexité

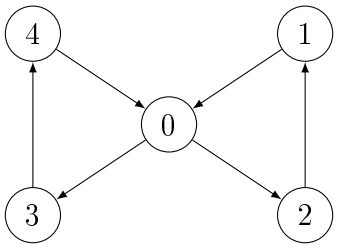
Définition 18. *Soit* $\mathrm{G} = (\mathrm{S}, \mathrm{A})$ *un graphe orienté et* $(x, y) \in \mathrm{S}^2$. *Un* chemin *reliant* x *à* y *est une suite finie d'arcs consécutifs reliant* x *à* y, *i.e. une suite d'arcs du type*

$$(x, x_1), (x_1, x_2), \ldots, (x_{k-1}, y)$$

On le note $(x, x_1, x_2, \ldots, x_{k-1}, y)$

Ce chemin est dit de longueur k *et d'extrémités* x *et* y. *Par convention, tout sommet est relié à lui-même par un chemin de longueur nulle.*

Définition 19. *Soit* $\mathrm{G} = (\mathrm{S}, \mathrm{A})$ *un graphe orienté. Un chemin est dit* élémentaire *s'il ne passe pas deux fois par un même sommet et* simple *s'il ne contient pas deux fois le même arc.*



Chemin $(3, 4, 0, 2, 1)$ élémentaire
Chemin $(4, 0, 2, 1, 0, 3)$ simple non élémentaire

Proposition 6. *Soit* $\mathrm{G} = (\mathrm{S}, \mathrm{A})$ *un graphe orienté. Un chemin élémentaire est simple.*

Démonstration. Identique à celle d'un graphe non orienté. □

Proposition 7. *Soit* $\mathrm{G} = (\mathrm{S}, \mathrm{A})$ *un graphe orienté. De tout chemin reliant deux sommets distincts, on peut extraire un chemin élémentaire.*

Démonstration. Identique à celle d'un graphe non orienté. □

Définition 20. *Soit* $\mathrm{G} = (\mathrm{S}, \mathrm{A})$ *un graphe orienté. Un chemin simple dont les extrémités coïncident est appelé* circuit. *Un graphe sans circuit est dit* acyclique.

Remarque : On parle de circuit pour un chemin de longueur non nulle.

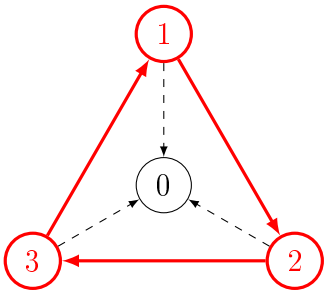


FIGURE 25 – $(1, 2, 3, 1)$ circuit

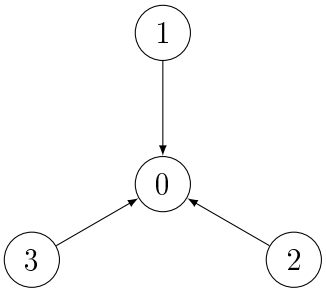


FIGURE 26 – Graphe acyclique

Définition 21. *Soit* $\mathrm{G} = (\mathrm{S}, \mathrm{A})$ *un graphe orienté. Les sommets* x *et* y *sont dits* fortement connectés *s'il existe un chemin reliant* x *à* y *et un autre reliant* y *à* x.

Définition 22. *Soit* $\mathrm{G} = (\mathrm{S}, \mathrm{A})$ *un graphe orienté. Le graphe* G *est dit* fortement connexe *si toute paire de sommets est constituée de sommets fortement connectés.*

3 Graphe eulérien, hamiltonien

Définition 23. *Soit* $G = (S, A)$ *un graphe orienté. Un chemin est dit* eulérien *s'il contient une fois et une seule tous les arcs du graphe. Un* circuit eulérien *est un circuit qui est un chemin eulérien.*

Remarque : Un chemin eulérien est un chemin simple de longueur m.

Définition 24. *Soit* $G = (S, A)$ *un graphe orienté. Si* G *admet un circuit eulérien, alors le graphe* G *est dit* eulérien.

Théorème 2. *Un graphe orienté fortement connexe est eulérien si et seulement si chaque sommet possède un même degré entrant et sortant.*

[Admis]

Définition 25. *Soit* $G = (S, A)$ *un graphe orienté. Un chemin est dit* hamiltonien *s'il passe une fois et une seule par chaque sommet. Un* circuit hamiltonien *est un circuit dont le chemin privé d'une extrémité est hamiltonien.*

Remarque : Un graphe qui admet un circuit hamiltonien est fortement connexe.

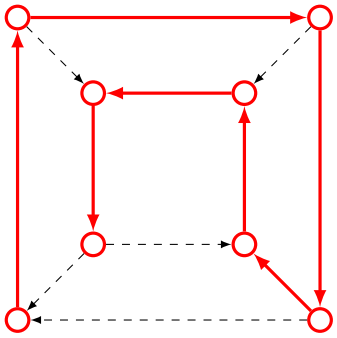


FIGURE 27 – Chemin hamiltonien

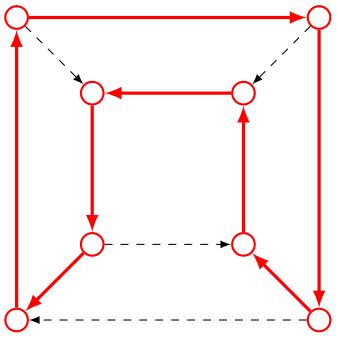


FIGURE 28 – Circuit hamiltonien

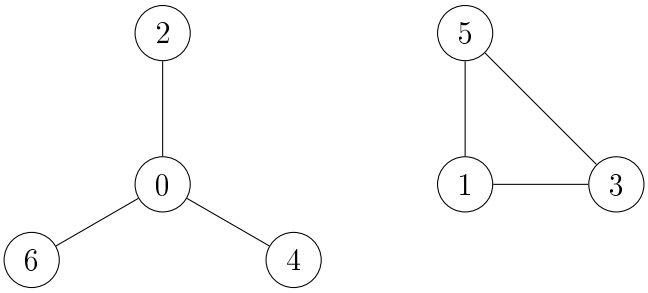
Définition 26. *Soit* $G = (S, A)$ *un graphe orienté. Si le graphe* G *admet un circuit hamiltonien, alors le graphe* G *est dit* hamiltonien.

III Représentation des graphes

1 Liste d'adjacence

Définition 27. *Soit* $G = (S, A)$ *un graphe orienté ou non orienté avec* $S = [s_0, \dots, s_{n-1}]$. *La représentation du graphe* G *par* listes d'adjacences *consiste en une liste de listes* $LA = [LA[i], i \in [\![0\,;\, n-1]\!]]$ *telle que pour* $i \in [\![0\,;\, n-1]\!]$, *la liste* $LA[i]$ *contient les sommets adjacents à* s_i.

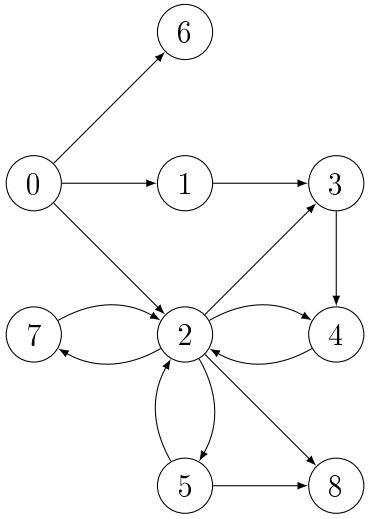
Exemples : 1. On considère le graphe non orienté :



L'ensemble des sommets et la liste d'adjacence sont donnés par

$$\mathrm{S} = [0, 1, \ldots, 6] \qquad \mathrm{LA} = [[2, 4, 6], [3, 5], [0], [1, 5], [0], [1, 3], [0]]$$

2. On considère le graphe orienté :



L'ensemble des sommets et la liste d'adjacence sont donnés par

$$\mathrm{S} = [0, 1, \ldots, 8] \qquad \mathrm{LA} = [[1, 2, 6], [3], [3, 4, 5, 7, 8], [4], [2], [2, 8], [], [2], []]$$

Proposition 8. *Soit* $\mathrm{G} = (\mathrm{S}, \mathrm{A})$ *un graphe non orienté avec* $\mathrm{S} = [s_0, \ldots, s_{n-1}]$ *et* LA *sa représentation par listes d'adjacences. On a*

$$\sum_{i=0}^{n-1} \mathrm{Card\ LA}[i] \leqslant 2\,\mathrm{Card\ A}$$

avec égalité si le graphe G *est simple (sans boucle).*

Démonstration. Pour $i \in [\![0\,;\, n-1]\!]$, on a

$$\mathrm{Card\ LA}\,[i] = \begin{cases} \mathrm{d}(s_i) & \text{si } s_i \text{ sans boucle} \\ \mathrm{d}(s_i) - 1 & \text{sinon} \end{cases}$$

Le résultat suit d'après la proposition 1. □

Exercice Soit $\mathrm{G} = (\mathrm{S}, \mathrm{A})$ un graphe non orienté avec $\mathrm{S} = [s_0, \ldots, s_{n-1}]$ et LA sa représentation par listes d'adjacences. Que représente l'écart $2\,\mathrm{Card\ A} - \sum_{i=0}^{n-1} \mathrm{Card\ LA}\,[i]$?

Corrigé : On a

$$2\,\text{Card A} - \sum_{i=0}^{n-1} \text{Card LA}\,[i] = \sum_{i=0}^{n-1} [\text{d}(s_i) - \text{Card LA}\,(i)] = \sum_{i\in[\![\,0\,;\,n-1\,]\!]:s_i \text{ boucle}} 1$$

L'écart $2\,\text{Card A} - \sum_{i=0}^{n-1} \text{Card LA}\,[i]$ correspond au nombre de boucles dans le graphe.

Proposition 9. *Soit* $\text{G} = (\text{S}, \text{A})$ *un graphe orienté avec* $\text{S} = [s_0, \ldots, s_{n-1}]$ *et* LA *sa représentation par listes d'adjacences. On a*

$$\sum_{i=0}^{n-1} \text{Card LA}[i] = \text{Card A}$$

Démonstration. Pour $i \in [\![\,0\,;\,n-1\,]\!]$, on a $\text{Card LA}[i] = \text{d}^+(s_i)$. Le résultat suit d'après la proposition 5. □

Exercice : Soit $\text{G} = (\text{S}, \text{A})$ un graphe orienté ou non orienté avec $\text{S} = [s_0, \ldots, s_{n-1}]$ et LA sa représentation par listes d'adjacences. Montrer que le coût de stockage de LA est en $\text{O}(n+m)$.

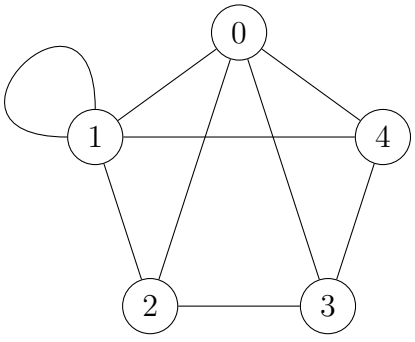
Corrigé : La liste LA contient n sous-listes et la somme des tailles de celles-ci est en $\text{O}(m)$ d'où le résultat.

2 Matrice d'adjacence

Définition 28. *Soit* $\text{G} = (\text{S}, \text{A})$ *un graphe orienté ou non orienté avec* $\text{S} = [s_0, \ldots, s_{n-1}]$ *et* $\text{A} = [a_0, \ldots, a_{m-1}]$. *La* matrice d'adjacence *du graphe* G *est une matrice* $\text{M} = \left(m_{i,j}\right)_{0\leqslant i,j\leqslant n-1} \in \mathscr{M}_n(\mathbb{R})$ *définie par*

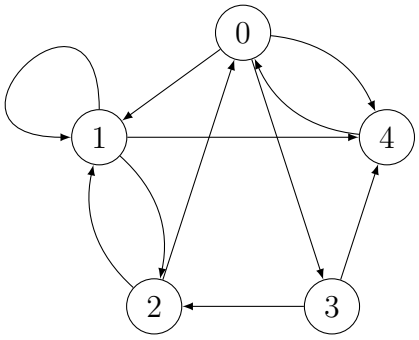
$$\forall (i,j) \in [\![\,0\,;\,n-1\,]\!] \qquad m_{i,j} = \begin{cases} 1 & \text{si } (s_i, s_j) \in \text{A ou } \{s_i, s_j\} \in \text{A} \\ 0 & \text{sinon} \end{cases}$$

Exemples :



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

FIGURE 29 – Graphe non orienté et sa matrice d'adjacence



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

FIGURE 30 – Graphe orienté et sa matrice d'adjacence

Remarques : (1) Dans le cas d'un graphe non orienté, la matrice d'adjacence est symétrique. (2) Dans le graphe non orienté ci-dessus, l'ensemble des sommets et la liste d'adjacence sont donnés par

$$\mathrm{S} = [0,1,2,3,4] \qquad \mathrm{LA} = [[1,2,3,4],[0,1,2,4],[0,1,3],[0,2,4],[0,1,3]]$$

et on observe

$$2\,\mathrm{Card}\ \mathrm{A} = 2 \times 9 \neq 17 = \sum_{i=0}^{4} \mathrm{Card}\ \mathrm{LA}[i]$$

ce qui s'explique par le fait que le graphe n'est pas simple .

Exercice : Quel est le coût de stockage d'une matrice d'adjacence ? Dans le cas d'un graphe simple non orienté, quelle est la perte de stockage (espace superflu) ?

Corrigé : Une matrice d'adjacence contient n^2 valeurs d'où un coût de stockage en $\mathrm{O}(n^2)$. Dans le cas d'un graphe simple non orienté, la matrice est symétrique avec des termes diagonaux nuls. Seuls les termes au-dessus de la diagonale sont pertinents d'où $\dfrac{n(n-1)}{2}$ termes. La perte de stockage porte sur les $n^2 - \dfrac{n(n-1)}{2} = \dfrac{n(n+1)}{2}$ termes inutiles.

Proposition 10. *Soit* $\mathrm{G} = (\mathrm{S}, \mathrm{A})$ *un graphe orienté ou non de matrice d'adjacence* $\mathrm{M} = \left(m_{i,j}\right)_{0 \leqslant i,j \leqslant n-1} \in \mathscr{M}_n(\mathbb{R})$. *Pour* ℓ *entier et* $(i,j) \in [\![0\,;n-1]\!]^2$, *on note* $m_{i,j}^{(\ell)}$ *le nombre de chemins ou chaînes de longueur* ℓ *reliant les sommets* i *à* j. *On a* $m_{i,j}^{(\ell)} = \left(\mathrm{M}^\ell\right)_{i,j}$.

Démonstration. On procède par récurrence sur ℓ. La propriété est immédiate pour $\ell = 0$. Supposons la vraie pour un rang ℓ entier fixé. Pour $(a,b) \in [\![0\,;n-1]\!]^2$ et p entier, on note $\mathrm{C}_{a,b}(p)$ l'ensemble des chemins de longueur p reliant le sommet a à b. En considérant l'avant-dernier sommet atteint dans les chemins de longueur $\ell+1$, on trouve

$$\mathrm{C}_{i,j}(\ell+1) = \bigsqcup_{k=0}^{n-1} \mathrm{C}_{i,k}(\ell) \times \mathrm{C}_{k,j}(1)$$

Et considérant les cardinaux, il vient

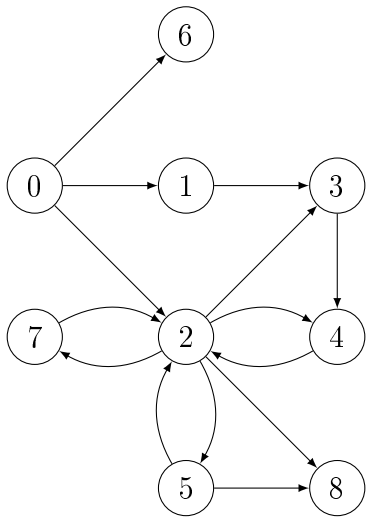
$$m_{i,j}^{(\ell+1)} = \sum_{k=0}^{n-1} m_{i,k}^{(\ell)} m_{k,j} = \left(\mathrm{M}^{\ell+1}\right)_{i,j}$$

ce qui clôt la récurrence. □

3 Implémentation

Une liste d'adjacence peut être implémentée en tant que liste si les sommets sont naturellement numérotés. Sinon, on peut utiliser un dictionnaire (situation typique quand les sommets sont des villes, des stations, ...)

Exemple : Pour le graphe orienté



on peut utiliser une liste :

```
S=list(range(9))
A=[[1,2,6],[3],[3,4,5,7,8],[4],[2],[2,8],[],[2],[]]
```

ou un dictionnaire :

```
S=list(range(9))
A={0:[1,2,6],1:[3],2:[3,4,5,7,8],3:[4],4:[2],5:[2,8],6:[],7:[2],8:[]}
```

Les sommets peuvent ne pas être des nombres :

```
S=[str(k) for k in range(9)]
A={"0":["1","2","6"],"1":["3"],"2":["3","4","5","7","8"],
"3":["4"],"4":["2"],"5":["2","8"],"6":[],"7":["2"],"8":[]}
```

Une matrice d'adjacence peut être implémentée en tant que liste de listes ou en tant que tableau `numpy` (type `ndarray`).

Le coût de stockage d'une liste d'adjacence est en $O(n+m)$ tandis que celui d'une matrice d'adjacence est en $O(n^2)$. Dans le cas d'un graphe peu dense avec m très inférieur à n^2, la liste d'adjacence fournit une représentation plus compacte. Avec une liste d'adjacence, pour tester l'existence d'un arc ou d'une arête (s_i, s_j), il n'existe pas de moyen plus efficace que de parcourir la liste $\text{LA}[s_i]$ dont le coût est en $O(n)$ dans le pire des cas. Avec une matrice d'adjacence, déterminer l'existence d'un arc ou d'une arête est en $O(1)$. Ces considérations détermineront des choix de représentation en situation.

IV Parcours en profondeur

1 Principe

On propose une version récursive du *parcours en profondeur* (DFS en anglais, Depth First Search). Celui-ci s'applique indifféremment aux graphes orientés ou non orientés. Le principe est le suivant :

- on part d'un sommet donné que l'on considère comme « en cours de visite » ;
- s'il admet des sommets adjacents « non visités » (des successeurs), on appelle récursivement le parcours en profondeur depuis chacun d'eux ;
- s'il n'y a pas ou plus de successeur, le sommet est considéré comme « visité ».

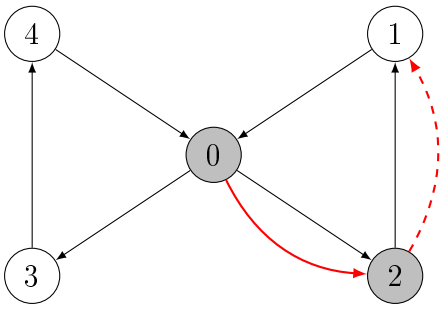


FIGURE 31 – Début de parcours en profondeur depuis le sommet 0

Une implémentation non récursive est possible mais demande plus d'effort et requiert une structure de pile.

On considère le graphe

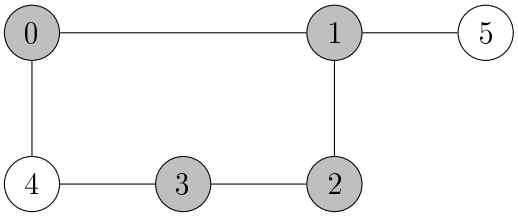


FIGURE 32 – Début du parcours en profondeur depuis le sommet 0

On suppose qu'on démarre depuis le sommet 0 et qu'on cherche à atteindre le sommet 3 et qu'on procède à un parcours en profondeur avec, à chaque étape, un balayage des sommets adjacents par numéro croissant. On visite dans l'ordre les sommets 0, 1, 2 et on atteint 3. Ce parcours nous indique donc qu'un tel chemin existe. On peut observer que le nombre d'arêtes traversées n'est pas minimal. Si on poursuit le parcours en profondeur depuis le sommet 3, on visite le sommet 4 puis on rencontre le sommet 0 qui est « en cours de visite ». On en déduit la présence d'un cycle.

On peut d'ores et déjà envisager des applications au parcours en profondeur comme :

- balayage d'une composante connexe ;
- détection de cycles (problématique importante en informatique : cycles des générateurs de nombres pseudo-aléatoires, cycles des fonctions de hachage, ...).

2 Implémentation

Les graphes seront représentés par listes d'adjacence implémentées soit par des listes, soit par des dictionnaires.

Les états d'un sommet seront décrits par les chaînes de caractères :
- `"blanc"` pour non visité ;
- `"gris"` pour en cours de visite ;
- `"noir"` pour visité.

On utilisera les dictionnaires suivants :
- `couleur` : pour `s` un sommet, `couleur[s]` est sa couleur ;
- `deb` : pour `s` un sommet, `deb[s]` est le temps de début de visite ;
- `fin` : pour `s` un sommet, `fin[s]` est le temps de fin de visite ;
- `predec` : pour `s` un sommet, `predec[s]` est son prédécesseur dans le parcours en profondeur.

On utilisera une liste `tps` initialisée à `[0]` pour mesurer le temps écoulé lors du parcours en profondeur.

Les listes et les dictionnaires sont mutables et sont adaptés pour la transmission d'arguments dans l'écriture récursive du parcours en profondeur.

```
def dfs_rec(S,A,sorg,tps,couleur,deb,fin,predec):
    deb[sorg]=tps[0]
    couleur[sorg]="gris"
    tps[0]+=1
    for s in A[sorg]:
        if couleur[s]=="blanc":
            predec[s]=sorg
            dfs_rec(S,A,s,tps,couleur,deb,fin,predec)
    couleur[sorg]="noir"
    fin[sorg]=tps[0]
    tps[0]+=1

def dfs_init(S,A,s0):
    tps,couleur,deb,fin,predec=[0],{},{},{},{}
    for s in S:
        couleur[s]="blanc"
    dfs_rec(S,A,s0,tps,couleur,deb,fin,predec)
    return deb,fin,predec,couleur
```

```
def dfs_rec(S,A,sorg,tps,couleur,deb,fin,predec):
                                # parcours en profondeur depuis sorg
                                # sorg : sommet d'origine
    deb[sorg]=tps[0]            # debut de visite de sorg
    couleur[sorg]="gris"        # on grise le sommet
    tps[0]+=1                   # on incrémente le compteur de temps
    for s in A[sorg]:           # on parcourt les sommets adjacents à sorg
        if couleur[s]=="blanc": # si un sommet adjacent s n'est pas visité
            predec[s]=sorg      # sorg est son prédécesseur
```

```
                                # et on lance récursivement
                                # le parcours en profondeur depuis s
            dfs_rec(S,A,s,tps,couleur,deb,fin,predec)
    couleur[sorg]="noir"        # quand il n'y a plus d'adjacents non visités
                                # le sommet est noirci
    fin[sorg]=tps[0]            # fin de visite de sorg
    tps[0]+=1                   # on incrémente le compteur de temps


def dfs_init(S,A,s0):           # amorce le parcours en profondeur depuis s0
    tps,couleur,deb,fin,predec=[0],{},{},{},{}
                                # tps est dans une liste, objet mutable
                                # les dictionnaires sont aussi mutables
    for s in S:                 # initialisation :
        couleur[s]="blanc"      # les sommets sont non visités
                                # parcours en profondeur récursif depuis s0
    dfs_rec(S,A,s0,tps,couleur,deb,fin,predec)
    return deb,fin,predec,couleur
```

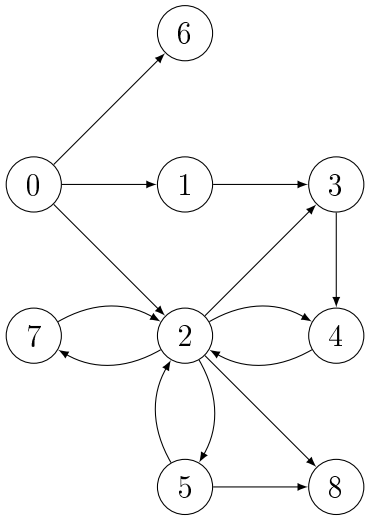
Remarque : La couleur grise n'est pas réellement utilisée dans l'algorithme précédent. L'état « en cours de visite » prend son sens par exemple lors de la recherche de circuits dans un graphe.

Exercice : Déterminer la complexité spatiale de `dfs_init`. On rappelle que $n = \text{Card S}$ et $m = \text{Card A}$.

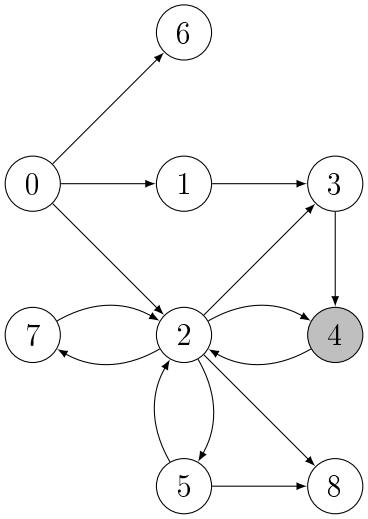
Corrigé : La variable `tps` est une liste avec une seule variable. Les dictionnaires `couleur, deb, fin, predec` sont de taille $\leqslant n$. Dans l'arbre de récursions, la plus longue branche d'appels correspond à une succession de sommets adjacents donc au plus n. Par conséquent, la complexité spatiale est en $\mathrm{O}(n)$.

3 Un exemple

On parcourt en profondeur depuis le sommet 4 le graphe orienté :

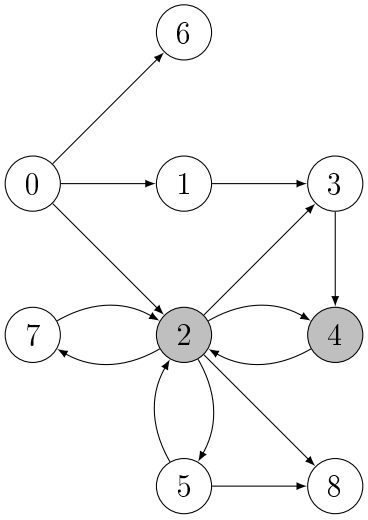


• On visite le sommet initial 4.



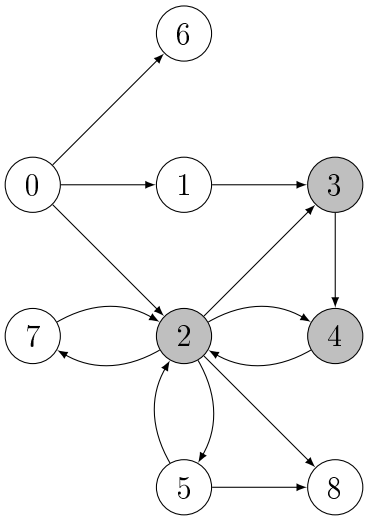

```
sommet= 4
tps= [0]
couleur=
{0: 'blanc', 1: 'blanc', 2: 'blanc',
 3: 'blanc', 4: 'gris', 5: 'blanc',
 6: 'blanc', 7: 'blanc', 8: 'blanc'}
deb= {4: 0}
fin= {}
predec= {}
```

• Le sommet 2 est adjacent au sommet 4 : on le visite.




```
sommet= 2
tps= [1]
couleur=
{0: 'blanc', 1: 'blanc', 2: 'gris',
 3: 'blanc', 4: 'gris', 5: 'blanc',
 6: 'blanc', 7: 'blanc', 8: 'blanc'}
deb= {2: 1, 4: 0}
fin= {}
predec= {2: 4}
```

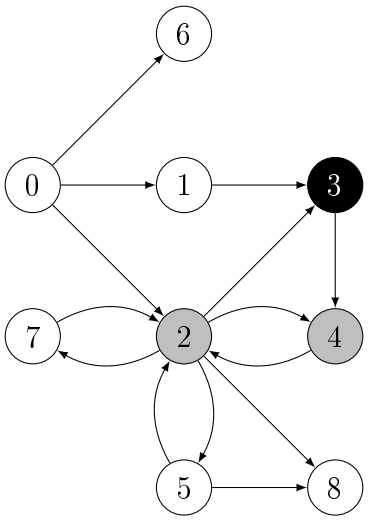
• Le sommet 3 est adjacent au sommet 2 : on le visite.




```
sommet= 3
tps= [2]
couleur=
{0: 'blanc', 1: 'blanc', 2: 'gris',
 3: 'gris', 4: 'gris', 5: 'blanc',
 6: 'blanc', 7: 'blanc', 8: 'blanc'}
deb= {2: 1, 3: 2, 4: 0}
fin= {}
predec= {2: 4, 3: 2}
```

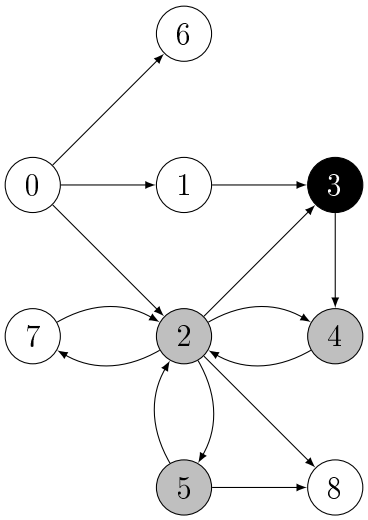
FIGURE 33 – Graphe orienté

• Le sommet 3 n'a pas de sommet adjacent non visité : on le noircit.



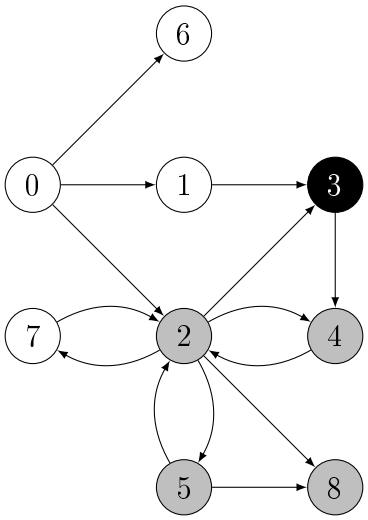

```
sommet= 3
tps= [3]
couleur=
{0: 'blanc', 1: 'blanc', 2: 'gris',
 3: 'noir', 4: 'gris', 5: 'blanc',
 6: 'blanc', 7: 'blanc', 8: 'blanc'}
deb= {2: 1, 3: 2, 4: 0}
fin= {3: 3}
predec= {2: 4, 3: 2}
```

• Le sommet 5 est adjacent au sommet 2 : on le visite.



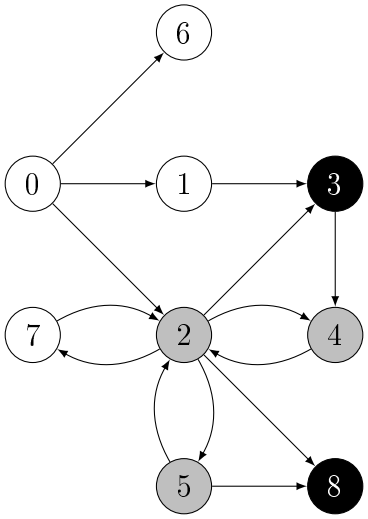

```
sommet= 5
tps= [4]
couleur=
{0: 'blanc', 1: 'blanc', 2: 'gris',
 3: 'noir', 4: 'gris', 5: 'gris',
 6: 'blanc', 7: 'blanc', 8: 'blanc'}
deb= {2: 1, 3: 2, 4: 0, 5: 4}
fin= {3: 3}
predec= {2: 4, 3: 2, 5: 2}
```

• Le sommet 8 est adjacent au sommet 5 : on le visite.



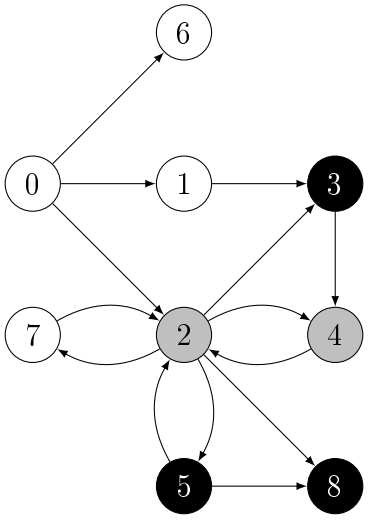

```
sommet= 8
tps= [5]
couleur=
{0: 'blanc', 1: 'blanc', 2: 'gris',
 3: 'noir', 4: 'gris', 5: 'gris',
 6: 'blanc', 7: 'blanc', 8: 'gris'}
deb= {8: 5, 2: 1, 3: 2, 4: 0, 5: 4}
fin= {3: 3}
predec= {8: 5, 2: 4, 3: 2, 5: 2}
```

• Le sommet 8 n'a pas de sommet adjacent non visité : on le noircit.



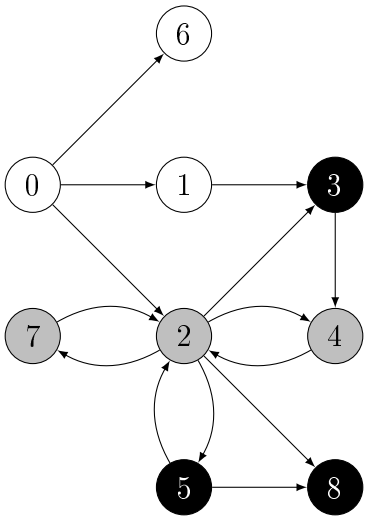

```
sommet= 8
tps= [6]
couleur=
{0: 'blanc', 1: 'blanc', 2: 'gris',
 3: 'noir', 4: 'gris', 5: 'gris',
 6: 'blanc', 7: 'blanc', 8: 'noir'}
deb= {8: 5, 2: 1, 3: 2, 4: 0, 5: 4}
fin= {8: 6, 3: 3}
predec= {8: 5, 2: 4, 3: 2, 5: 2}
```

• Le sommet 5 n'a plus de sommet adjacent non visité : on le noircit.



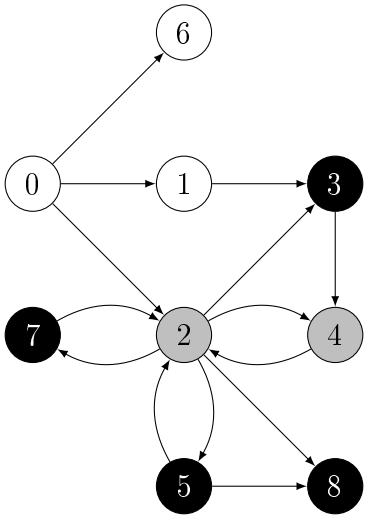

```
sommet= 5
tps= [7]
couleur=
{0: 'blanc', 1: 'blanc', 2: 'gris',
 3: 'noir', 4: 'gris', 5: 'noir',
 6: 'blanc', 7: 'blanc', 8: 'noir'}
deb= {8: 5, 2: 1, 3: 2, 4: 0, 5: 4}
fin= {8: 6, 3: 3, 5: 7}
predec= {8: 5, 2: 4, 3: 2, 5: 2}
```

• Le sommet 7 est adjacent au sommet 2 : on le visite.



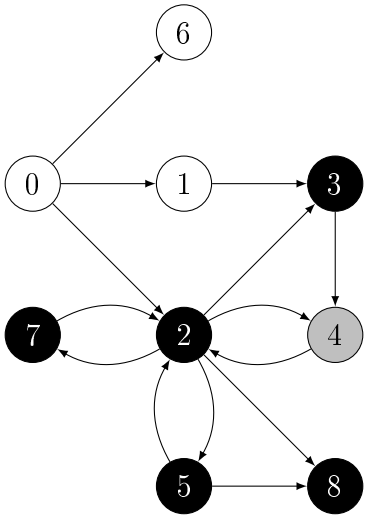

```
sommet= 7
tps= [8]
couleur=
{0: 'blanc', 1: 'blanc', 2: 'gris',
 3: 'noir', 4: 'gris', 5: 'noir',
 6: 'blanc', 7: 'gris', 8: 'noir'}
deb= {2: 1, 3: 2, 4: 0, 5: 4, 7: 8, 8: 5}
fin= {8: 6, 3: 3, 5: 7}
predec= {8: 5, 2: 4, 3: 2, 5: 2, 7: 2}
```

• Le sommet 7 n'a pas de sommet adjacent non visité : on le noircit.



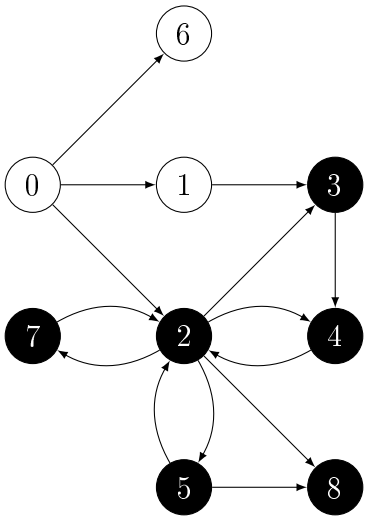

```
sommet= 7
tps= [9]
couleur=
{0: 'blanc', 1: 'blanc', 2: 'gris',
 3: 'noir', 4: 'gris', 5: 'noir',
 6: 'blanc', 7: 'noir', 8: 'noir'}
deb= {2: 1, 3: 2, 4: 0, 5: 4, 7: 8, 8: 5}
fin= {8: 6, 3: 3, 5: 7, 7: 9}
predec= {8: 5, 2: 4, 3: 2, 5: 2, 7: 2}
```

• Le sommet 2 n'a plus de sommet adjacent non visité : on le noircit.




```
sommet= 2
tps= [10]
couleur=
{0: 'blanc', 1: 'blanc', 2: 'noir',
 3: 'noir', 4: 'gris', 5: 'noir',
 6: 'blanc', 7: 'noir', 8: 'noir'}
deb= {2: 1, 3: 2, 4: 0, 5: 4, 7: 8, 8: 5}
fin= {8: 6, 2: 10, 3: 3, 5: 7, 7: 9}
predec= {8: 5, 2: 4, 3: 2, 5: 2, 7: 2}
```

• Le sommet 4 n'a plus de sommet adjacent non visité : on le noircit.




```
sommet= 4
tps= [11]
couleur=
{0: 'blanc', 1: 'blanc', 2: 'noir',
 3: 'noir', 4: 'noir', 5: 'noir',
 6: 'blanc', 7: 'noir', 8: 'noir'}
deb= {2: 1, 3: 2, 4: 0, 5: 4, 7: 8, 8: 5}
fin= {2: 10, 3: 3, 4: 11, 5: 7, 7: 9, 8: 6}
predec= {8: 5, 2: 4, 3: 2, 5: 2, 7: 2}
```

V Parcours en largeur

1 Principe

Pour effectuer le parcours en largeur, on utilise une structure de *file*.

Définition 29. *Une* file *aussi appelée* file d'attente *(queue en anglais) est une structure de données basée sur le principe « premier entré, premier sorti » (FIFO en anglais, First In First Out).*

FIGURE 34 – File

En python, on dispose du module `deque` de la bibliothèque `collections` qui propose une implémentation de cette structure

```
from collections import deque
```

Sur une variable de type `deque`, on utilise les méthodes :
- `append` : ajoute un élément à droite ;
- `appendleft` : ajoute un élément à gauche ;
- `pop` : renvoie et supprime l'élément de droite ;
- `popleft` : renvoie et supprime l'élément de gauche.

```
>>> D=deque()
>>> D.append(1)
>>> D.appendleft(2)
>>> D.appendleft(3)
>>> D
deque([3, 2, 1])
```

```
>>> D.pop()
1
>>> D
deque([3, 2])
>>> D.popleft()
3
>>> D
deque([2])
```

L'implémentation proposée est optimisée : les opérations `append`, `appendleft`, `pop`, `popleft` sont de complexité temporelle en O(1) (complexité amortie).

Le *parcours en largeur* (BFS en anglais, Breadth First Search) s'applique indifféremment aux graphes orientés ou non orientés. Son principe est le suivant :

- on part d'un sommet qu'on place dans une file et que l'on considère comme « en cours de visite » ;
- tant que la file n'est pas vide, on sort le sommet premier entré, on fait entrer dans la file tous ses sommets adjacents « non visités » puis on considère le sommet d'origine comme « visité ».

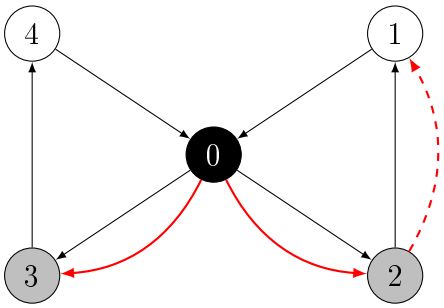


FIGURE 35 – Début de parcours en largeur depuis le sommet 0

On considère le graphe

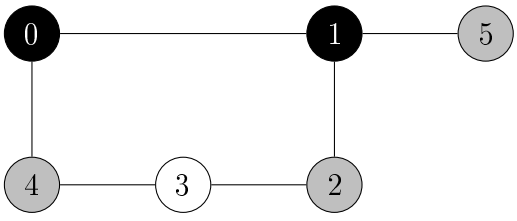


FIGURE 36 – Début du parcours en largeur depuis le sommet 0

On suppose qu'on démarre depuis le sommet 0 et qu'on cherche à atteindre le sommet 3 et qu'on procède à un parcours en largeur avec, à chaque étape, un balayage des sommets adjacents par numéro croissant. On sort le sommet 0 de la file et on le visite, on place dans la file les sommets 1 et 4 et on marque le sommet 0 comme visité, on sort le sommet 1 de la file et on le visite, on place les sommets 2 et 5 dans la file et on marque le sommet 1 comme visité, on sort le sommet 4 et enfin, on place dans la file le sommet 3. On a visité tous les sommets avant que le sommet 3 entre dans la file donc cela peut sembler moins bien que le déroulé du parcours en profondeur. Mais en réalité, on peut, pour chaque sommet placé dans la file, garder la mémoire

de son prédécesseur et on observe que pour atteindre le sommet 3 depuis le sommet 0, on suit la chaîne $(0, 4, 3)$ qui est de longueur minimale.

2 Implémentation

Les graphes seront représentés par listes d'adjacence implémentées soit par des listes, soit par des dictionnaires.

Les états d'un sommet seront décrits par les chaînes de caractères :
- `"blanc"` pour non visité ;
- `"gris"` pour en cours de visite ;
- `"noir"` pour visité.

On utilisera les dictionnaires suivants :
- `couleur` : pour `s` un sommet, `couleur[s]` est sa couleur ;
- `dist` : pour `s` un sommet, `dist[s]` est la distance au sommet de départ ;
- `predec` : pour `s` un sommet, `predec[s]` est son prédécesseur dans le parcours en largeur.

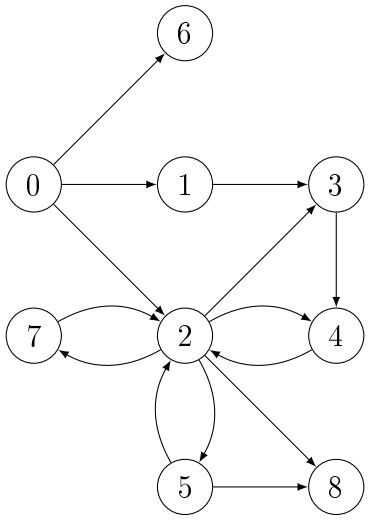
```
def bfs(S,A,s0):
    F=deque()
    couleur,dist,predec={},{},{}
    for s in S:
        dist[s]=float('inf')
        couleur[s]="blanc"
    F.append(s0)
    dist[s0]=0
    couleur[s0]="gris"
    while len(F)!=0:
        sorg=F.popleft()
        for s in A[sorg]:
            if couleur[s]=="blanc":
                F.append(s)
                dist[s]=dist[sorg]+1
                couleur[s]="gris"
                predec[s]=sorg
        couleur[sorg]="noir"
    return dist,predec,couleur
```

```
def bfs(S,A,s0):                    # parcours en largeur depuis s0
    F=deque()                       # création d'une file vide F
    couleur,dist,predec={},{},{}    # dist : dictionnaire des distances à s0
    for s in S:                     # initialisation :
        dist[s]=float('inf')              # les sommets à distance infinie de s0
        couleur[s]="blanc"          # et non visités
    F.append(s0)                    # s0 entre dans F
    dist[s0]=0
    couleur[s0]="gris"              # on grise s0
    while len(F)!=0:                # tant que F non vide
        sorg=F.popleft()            # sortie du sommet sorg entré en premier dans F
                                    # sorg : sommet d'origine
        for s in A[sorg]:           # on parcourt les sommets adjacents
            if couleur[s]=="blanc": # si un sommet adjacent s n'est pas visité
                F.append(s)         # il entre dans F
                dist[s]=dist[sorg]+1
                                    # distance à s0 augmentée de 1
                couleur[s]="gris"   # s est grisé
                predec[s]=sorg      # sorg est son prédécesseur
                                    # après entrée dans F des adjacents de sorg
        couleur[sorg]="noir"        # celui-ci est noirci
    return dist,predec,couleur
```

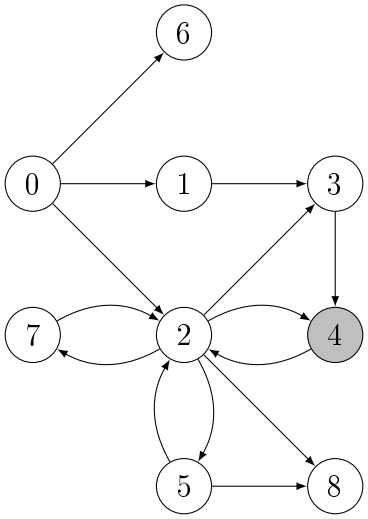
On peut démontrer (voir [3]) que les distances stockées dans le dictionnaire `dist` sont minimales en le nombre d'arêtes reliant un sommet au sommet initial.

3 Un exemple

On parcourt en largeur depuis le sommet 4 le graphe orienté :

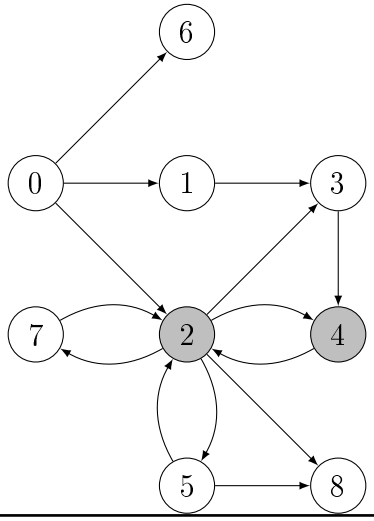


• On visite le sommet initial 4 : il entre en file.



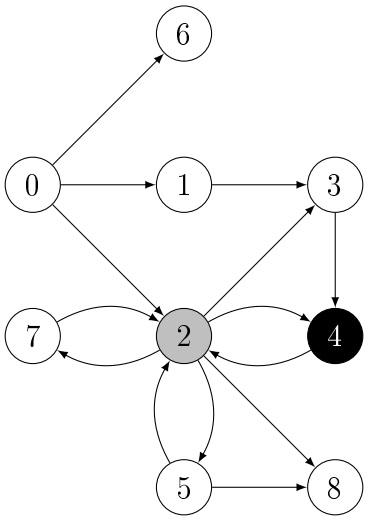

```
F= deque([4])
couleur=
{0: 'blanc', 1: 'blanc', 2: 'blanc',
 3: 'blanc', 4: 'gris', 5: 'blanc',
 6: 'blanc', 7: 'blanc', 8: 'blanc'}
dist={0: inf, 1: inf, 2: inf,
 3: inf, 4: 0, 5: inf,
 6: inf, 7: inf, 8: inf}
predec= {}
```

• On détermine les sommets adjacents au sommet 4 non visités : le sommet 2 entre en file et le sommet 4 en sort.



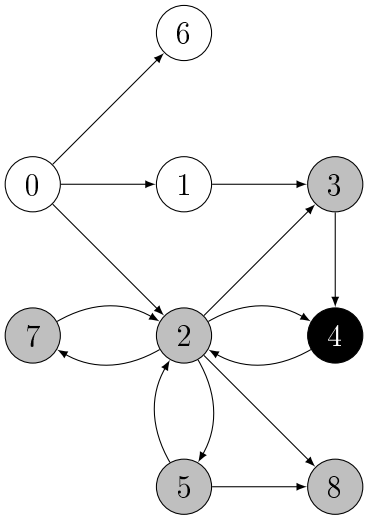

```
F= deque([2])
couleur=
{0: 'blanc', 1: 'blanc', 2: 'gris',
 3: 'blanc', 4: 'gris', 5: 'blanc',
 6: 'blanc', 7: 'blanc', 8: 'blanc'}
dist=
{0: inf, 1: inf, 2: 1,
 3: inf, 4: 0, 5: inf,
 6: inf, 7: inf, 8: inf}
predec= {2: 4}
```

• On a fini la visite du sommet 4 : on le noircit.



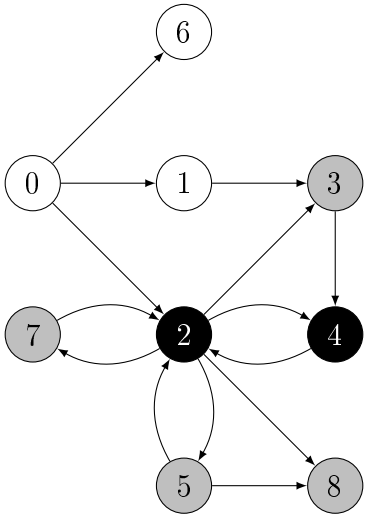

```
F= deque([2])
couleur=
{0: 'blanc', 1: 'blanc', 2: 'gris',
 3: 'blanc', 4: 'noir', 5: 'blanc',
 6: 'blanc', 7: 'blanc', 8: 'blanc'}
dist=
{0: inf, 1: inf, 2: 1,
 3: inf, 4: 0, 5: inf,
 6: inf, 7: inf, 8: inf}
predec= {2: 4}
```

• On détermine les sommets adjacents au sommet 2 non visités : les sommets 3, 5, 7 et 8 entrent en file et le sommet 2 en sort.



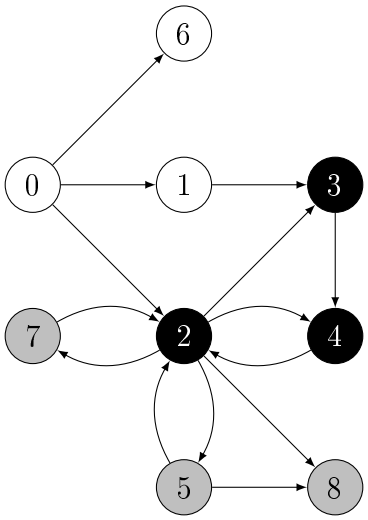

```
F= deque([3, 5, 7, 8])
couleur=
{0: 'blanc', 1: 'blanc', 2: 'gris',
 3: 'gris', 4: 'noir', 5: 'gris',
 6: 'blanc', 7: 'gris', 8: 'gris'}
dist=
{0: inf, 1: inf, 2: 1,
 3: 2, 4: 0, 5: 2,
 6: inf, 7: 2, 8: 2}
predec= {8: 2, 2: 4, 3: 2, 5: 2, 7: 2}
```

• On a fini la visite du sommet 2 : on le noircit.



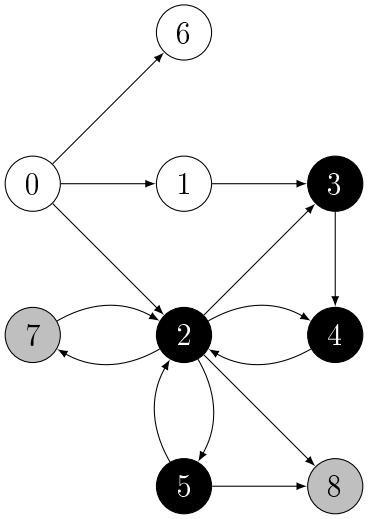

```
F= deque([3, 5, 7, 8])
couleur=
{0: 'blanc', 1: 'blanc', 2: 'noir',
 3: 'gris', 4: 'noir', 5: 'gris',
 6: 'blanc', 7: 'gris', 8: 'gris'}
dist=
{0: inf, 1: inf, 2: 1,
 3: 2, 4: 0, 5: 2,
 6: inf, 7: 2, 8: 2}
predec= {8: 2, 2: 4, 3: 2, 5: 2, 7: 2}
```

• Le sommet 3 n'a pas de sommet adjacent non visité : on a fini sa visite, on le noircit.



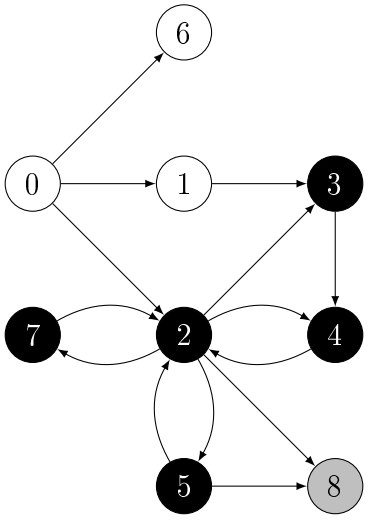

```
F= deque([5, 7, 8])
couleur=
{0: 'blanc', 1: 'blanc', 2: 'noir',
 3: 'noir', 4: 'noir', 5: 'gris',
 6: 'blanc', 7: 'gris', 8: 'gris'}
dist=
{0: inf, 1: inf, 2: 1,
 3: 2, 4: 0, 5: 2,
 6: inf, 7: 2, 8: 2}
predec= {8: 2, 2: 4, 3: 2, 5: 2, 7: 2}
```

• Le sommet 5 n'a pas de sommet adjacent non visité : on a fini sa visite, on le noircit.



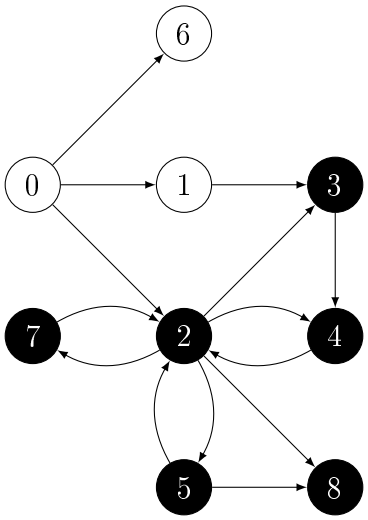

```
F= deque([7, 8])
couleur=
{0: 'blanc', 1: 'blanc', 2: 'noir',
 3: 'noir', 4: 'noir', 5: 'noir',
 6: 'blanc', 7: 'gris', 8: 'gris'}
dist=
{0: inf, 1: inf, 2: 1,
 3: 2, 4: 0, 5: 2,
 6: inf, 7: 2, 8: 2}
predec= {8: 2, 2: 4, 3: 2, 5: 2, 7: 2}
```

• Le sommet 7 n'a pas de sommet adjacent non visité : on a fini sa visite, on le noircit.




```
F= deque([8])
couleur=
{0: 'blanc', 1: 'blanc', 2: 'noir',
 3: 'noir', 4: 'noir', 5: 'noir',
 6: 'blanc', 7: 'noir', 8: 'gris'}
dist=
{0: inf, 1: inf, 2: 1,
 3: 2, 4: 0, 5: 2,
 6: inf, 7: 2, 8: 2}
predec= {8: 2, 2: 4, 3: 2, 5: 2, 7: 2}
```

• Le sommet 8 n'a pas de sommet adjacent non visité : on a fini sa visite, on le noircit.




```
F= deque([])
couleur=
{0: 'blanc', 1: 'blanc', 2: 'noir',
 3: 'noir', 4: 'noir', 5: 'noir',
 6: 'blanc', 7: 'noir', 8: 'noir'}
dist=
{0: inf, 1: inf, 2: 1,
 3: 2, 4: 0, 5: 2,
 6: inf, 7: 2, 8: 2}
predec= {8: 2, 2: 4, 3: 2, 5: 2, 7: 2}
```

VI Plus court chemin

1 Graphe valué, coût

Définition 30. *Un* graphe valué *ou* pondéré G *est un triplet* $(\mathrm{S}, \mathrm{A}, \nu)$ *avec* (S, A) *un graphe orienté ou non muni d'une application* $\nu : \mathrm{A} \to \mathbb{R}$ *appelée* valuation *ou* pondération *du graphe.*

Exemple : Typiquement, dans un graphe dont les sommets sont des villes ou des gares, la valuation peut quantifier une distance ou un temps de trajet.

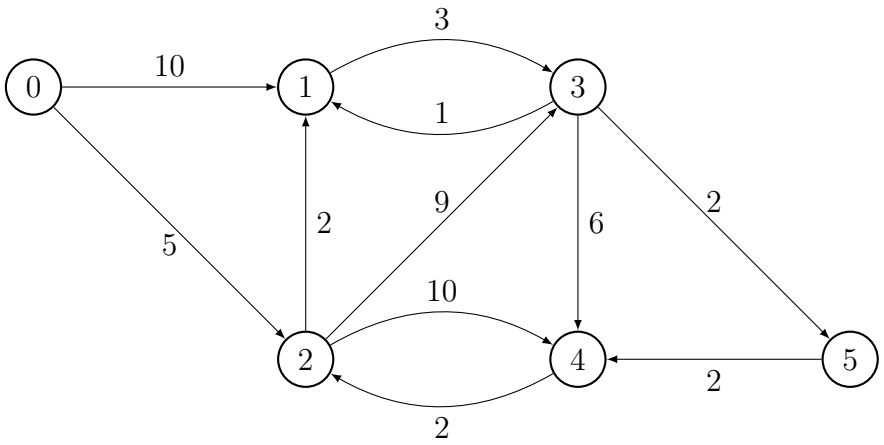
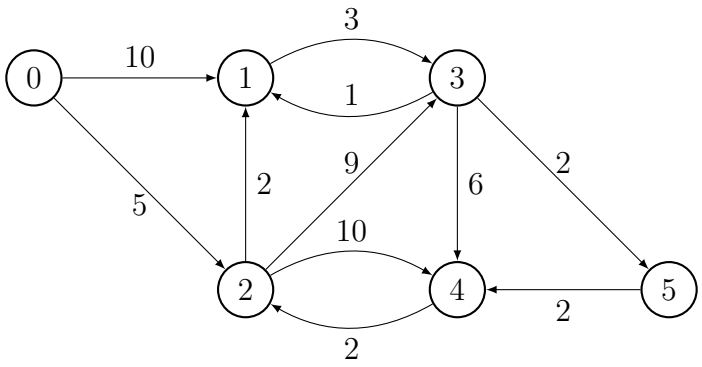


FIGURE 37 – Graphe valué

Définition 31. *Soit* $\mathrm{G} = (\mathrm{S}, \mathrm{A}, \nu)$ *un graphe valué orienté ou non orienté avec* $\mathrm{S} = [s_0, \ldots, s_{n-1}]$ *et* $\mathrm{A} = [a_0, \ldots, a_{m-1}]$. *La* matrice d'adjacence *du graphe* G *est une matrice* $\mathrm{M} = \left(m_{i,j}\right)_{0 \leqslant i,j \leqslant n-1} \in \mathscr{M}_n(\mathbb{R})$ *définie par*

$$\forall (i,j) \in [\![\, 0\,;\, n-1 \,]\!] \qquad m_{i,j} = \begin{cases} \nu(s_i, s_j) & \text{si } (s_i, s_j) \in \mathrm{A} \text{ ou } \{s_i, s_j\} \in \mathrm{A} \\ +\infty & \text{sinon} \end{cases}$$



$$M = \begin{pmatrix} +\infty & 10 & 5 & +\infty & +\infty & +\infty \\ +\infty & +\infty & +\infty & 3 & +\infty & +\infty \\ +\infty & 2 & +\infty & 9 & 10 & +\infty \\ +\infty & 1 & +\infty & +\infty & 6 & 2 \\ +\infty & +\infty & 2 & +\infty & +\infty & +\infty \\ +\infty & +\infty & +\infty & +\infty & 2 & +\infty \end{pmatrix}$$

FIGURE 38 – Graphe valué et sa matrice d'adjacence

Définition 32. *Soit* $G = (S, A, \nu)$ *un graphe valué. Étant donnée* $c = (x_0, \ldots, x_k)$ *une chaîne ou un chemin de* G*, on définit le* poids *ou* coût *de la chaîne ou du chemin* c *par*

$$L(c) = \sum_{i=1}^{k} \nu(x_{i-1}, x_i)$$

2 Propriétés

Définition 33. *Soit* $G = (S, A, \nu)$ *un graphe valué. On définit le* coût d'un plus court chemin *entre les sommets* x *et* y *noté* $\delta(x, y)$ *par*

$$\delta(x, y) = \begin{cases} \inf \{L(c), c : \textit{chemin de } x \textit{ à } y\} & \textit{s'il existe un chemin de } x \textit{ à } y \\ +\infty & \textit{sinon} \end{cases}$$

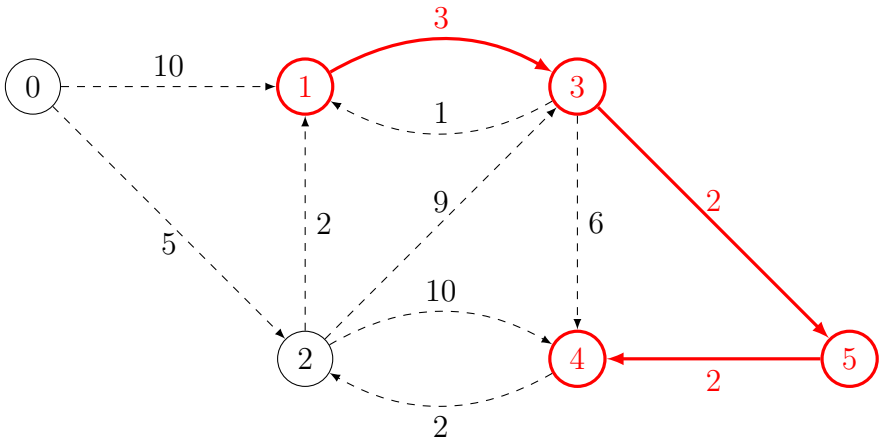


FIGURE 39 – Plus court chemin du sommet 1 au sommet 4

Dans ce qui suit, on considérera le cas d'un graphe orienté valué et on utilisera exclusivement le mot *chemin*. Ce n'est pas restrictif puisqu'un graphe non orienté peut être interprété comme un graphe orienté en remplaçant chaque arête par une paire d'arcs reliant les sommets dans les deux sens.

Dans la recherche d'un chemin de x à y, plusieurs cas se présentent :

— il n'existe pas de chemin de x à y,
— il existe un ou plusieurs chemins de x à y mais pas de plus courts (configuration où $\delta(x,y) = -\infty$) ;
— il existe un ou plusieurs plus courts chemins de x à y.

Définition 34. *Soit* $G = (S, A, \nu)$ *un graphe orienté valué. Un circuit de coût strictement négatif est dit* absorbant.

Remarques : (1) Un graphe orienté valué positivement ne contient pas de circuit absorbant. (2) Si on autorise des valuations négatives dans un graphe non orienté ayant au moins deux sommets, alors celui-ci peut contenir un circuit absorbant.

Proposition 11. *Soit* $G = (S, A, \nu)$ *un graphe orienté valué et deux sommets* x *et* y *reliés par un chemin. Il existe un plus court chemin entre* x *et* y *si et seulement aucun chemin de* x *à* y *ne contient de circuit absorbant.*

Démonstration. S'il existe un chemin de x à y contenant un circuit absorbant, alors on peut, en bouclant dans le circuit absorbant, rendre le poids du chemin arbitrairement petit d'où $\delta(x,y) = -\infty$ et il n'existe donc pas de plus court chemin de x à y. Supposons qu'aucun chemin de x à y ne contient de circuit absorbant. Si un chemin de x à y contient un cycle, alors celui-ci est de coût nul ou positif et on peut donc ne considérer que les chemins élémentaires de x à y qui sont en nombre fini ($\leqslant \sum_{k=0}^{n-2} k!\binom{n}{k}$, choix du nombre de k sommets autres que x et y puis des k sommets avec ordre). Par suite, l'ensemble des coûts de ces chemins élémentaires est un ensemble fini qui admet donc un plus petit élément d'où l'existence d'un plus court chemin de x à y. □

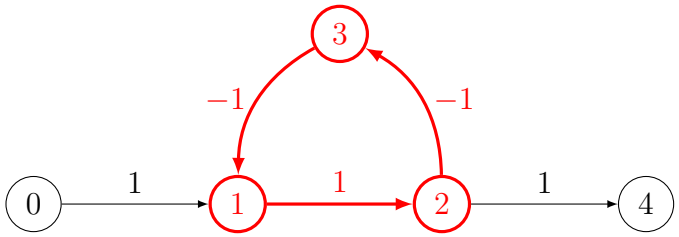


FIGURE 40 – Circuit $(1, 2, 3, 1)$ absorbant

En bouclant dans le circuit $(1, 2, 3, 1)$, on rend le coût du chemin du sommet 0 au sommet 4 arbitrairement petit.

Remarque : Cette situation peut arriver sur les marchés financiers et s'appelle une *opportunité d'arbitrage*. C'est le principe d'un repas gratuit : on a tout intérêt à en profiter autant qu'il est possible ...

Théorème 3. *Soit* $G = (S, A, \nu)$ *un graphe orienté valué fortement connexe. Il existe un plus court chemin entre tout couple de sommets si et seulement s'il n'existe pas de circuit absorbant dans* G.

Démonstration. S'il n'existe pas de circuit absorbant, alors on applique la proposition précédente à un couple quelconque de sommets qui sont fortement connectés pour lesquels aucun chemin ne contient de circuit absorbant. S'il existe un circuit absorbant dans G, alors en considérant deux sommets confondus sur ce circuit, il n'existe pas de plus court chemin entre ces sommets d'après le résultat de la proposition précédente. □

Proposition 12. *Soit* $\mathrm{G} = (\mathrm{S}, \mathrm{A}, \nu)$ *un graphe orienté valué. Pour* x, y z *des sommets de* G, *on a*

$$\delta(x, z) \leqslant \delta(x, y) + \nu(y, z)$$

Démonstration. Soit c un chemin de x à y. Alors $\mathrm{L}(c) + \nu(y, z)$ est le coût d'un chemin de x à z qui est donc supérieur à $\delta(x, z)$ d'où $\mathrm{L}(c) \geqslant \delta(x, z) - \nu(y, z)$. Ceci vaut pour tout chemin c de x à y et la borne inférieure étant le plus grand des minorants, le résultat suit. □

Proposition 13. *Tout sous-chemin d'un plus court chemin est un plus court chemin.*

Démonstration. Sinon, on pourrait substituer le sous-chemin en question par un chemin strictement plus court et le chemin initial ne serait plus le plus court. □

VII Algorithme de Dijkstra

1 Principe

L'algorithme de Dijkstra [2] est un algorithme *glouton*, c'est-à-dire qu'il réalise, étape par étape, un optimum local, en vue d'obtenir un optimum global. Un algorithme glouton ne fournit pas systématiquement la solution optimale ; ce n'est pas le cas de l'algorithme de Dijkstra qui est correct pour la recherche de plus court chemin dans un graphe orienté valué positivement, c'est-à-dire si tous les arcs ont un coût positif.

On note s_0 le sommet source et $\mathrm{d}[u]$ la longueur d'un plus court chemin de s_0 à un sommet u découvert à un certain stade de l'algorithme.

Le principe de l'algorithme est le suivant :
- on initialise $\mathrm{d}[s_0] \leftarrow 0$ et $\mathrm{d}[u] \leftarrow \infty$ pour tout $u \in \mathrm{S} \setminus \{s_0\}$;
- tant qu'il y a des sommets « non visités » :
  - on détermine un sommet u non visité tel que $\mathrm{d}[u]$ est minimal ;
  - le sommet u est considéré comme « visité » ;
  - pour chaque sommet s non visité et successeur de u, on effectue l'opération de *relâchement* :

    $$\text{si } \mathrm{d}[s] > \mathrm{d}[u] + \nu(u, s), \text{ alors } \mathrm{d}[s] \leftarrow \mathrm{d}[u] + \nu(u, s)$$

    autrement dit : si pour aller à `s` depuis `s0`, il est plus court de passer par `u`, alors on le fait.

2 Implémentation

On utilisera les dictionnaires suivants :
- `cout` : pour `s` un sommet, `cout[s]` est la valeur de $\mathrm{d}[s]$;
- `predec` : pour `s` un sommet, `predec[s]` est son prédécesseur dans le plus court chemin depuis s_0 ;
- `djvu` : pour `s` un sommet, `djvu[s]` est un booléen qui dit si le sommet a été déjà vu ou pas.

2. Edsger Dijkstra, 1930-2002, mathématicien et informaticien néerlandais.

```
def dijkstra(S,A,s0):
    cout,predec,djvu={},{},{}
    for s in S:
        cout[s]=float('inf')
        djvu[s]=False
    cout[s0]=0
    for _ in S:
        smin,cmin=s0,float('inf')
        for s in S:
            if not djvu[s] and cout[s]<cmin:
                smin,cmin=s,cout[s]
        djvu[smin]=True
        for s,nu in A[smin]:
            if not djvu[s] and cout[s]>cmin+nu:
                cout[s]=cmin+nu
                predec[s]=smin
    return cout,predec
```

```
def dijkstra(S,A,s0):              # recherche de plus courts chemins depuis s0
    cout,predec,djvu={},{},{}
    for s in S:                    # initialisation :
        cout[s]=float('inf')             # les sommets à cout infini de s0
        djvu[s]=False              # et non visités
    cout[s0]=0
    for _ in S:
        smin,cmin=s0,float('inf')        # recherche d'un sommet pas déjà vu
                                   # et à cout minimal
        for s in S:
            if not djvu[s] and cout[s]<cmin:
                smin,cmin=s,cout[s]
        djvu[smin]=True            # le sommet est considéré visité
        for s,nu in A[smin]:       # pour les successeurs non visités
                                   # on relache
            if not djvu[s] and cout[s]>cmin+nu:
                cout[s]=cmin+nu
                predec[s]=smin
    return cout,predec
```

Remarque : L'implémentation ci-avant détermine la totalité des plus courts chemins existants depuis `s0`. On peut aussi effectuer une recherche plus ciblée d'un plus court chemin entre un sommet de départ et un sommet d'arrivée (à fournir en arguments).

Théorème 4. *Soit* $\mathrm{G} = (\mathrm{S}, \mathrm{A}, \nu)$ *un graphe orienté valué positivement. Étant donné un sommet* $s_0 \in \mathrm{S}$*, l'algorithme de Dijkstra se termine et est correct pour la détermination des prédécesseurs de chaque sommet dans un plus court chemin depuis* s_0 *et le coût d'un plus court chemin depuis* s_0 *à chaque sommet qui lui est relié.*

Démonstration. La terminaison de l'algorithme est immédiate puisqu'on réalisée simplement deux boucles `for` imbriquées. On note V l'ensemble des sommets visités au cours de l'algorithme et on considère l'invariant

$$\forall x \in \mathrm{V} \qquad \mathrm{d}[x] = \delta(s_0, x) \quad \text{et} \quad \forall y \notin \mathrm{V} \qquad \mathrm{d}[y] = \operatorname*{Min}_{x \in \mathrm{V} \cap \Gamma(y)} \mathrm{d}[x] + \nu(x, y)$$

avec la notation $\Gamma(x)$ qui désigne les successeurs du sommet x. Initialement, on a $\mathrm{V} = \varnothing$ et l'invariant est vrai. Considérons l'étape où un sommet $x \notin \mathrm{V}$ passe dans V. On a donc

$$\mathrm{d}[x] = \operatorname*{Min}_{z \notin \mathrm{V}} \mathrm{d}[z]$$

L'étape de relaxation met à jour les valeurs de $\mathrm{d}[y]$ pour tout $y \notin \mathrm{V}$ et tel que $x \in \Gamma(y)$. Supposons $\delta(s_0, x) < \infty$. Il existe donc un plus court chemin de s_0 à x. On note u le dernier sommet visité et v son successeur dans ce plus court chemin (on sait que x n'est pas visité donc il fait sens de considérer le dernier sommet visité et son successeur dans ce chemin). Par positivité de la valuation ν, on a

$$\delta(s_0, x) = \mathrm{L}(s_0, \ldots, u, v, \ldots, x) \geqslant \mathrm{L}(s_0, \ldots, u, v) = \mathrm{L}(s_0, \ldots, u) + \nu(u, v)$$

D'après la proposition 13, un sous-chemin d'un plus court chemin étant lui-même plus court chemin, il vient $\mathrm{L}(s_0, \ldots, u) = \delta(s_0, u)$ et comme le sommet u a été visité, on sait par hypothèse de récurrence $\delta(s_0, u) = \mathrm{d}[u]$. Ainsi, on a

$$\delta(s_0, x) \geqslant \mathrm{d}[u] + \nu(u, v)$$

Or, on a
$$\mathrm{d}[v] = \operatorname*{Min}_{z \in \mathrm{V} \cap \Gamma(v)} \mathrm{d}[z] + \nu(z, v)$$

d'où
$$\mathrm{d}[u] + \nu(u, v) \geqslant \mathrm{d}[v] \geqslant \mathrm{d}[x]$$

la dernière inégalité résultant du choix de x. Comme $\delta(s_0, x) < \infty$, le prédécesseur de x est l'argument du minimum ($\arg\min$) de $z \in \mathrm{V} \cap \Gamma(x) \mapsto \mathrm{d}[z] + \nu(z, x)$ que l'on note $x^\star$. On a

$$\delta(s_0, x) \geqslant \mathrm{d}[x] = \delta(s_0, x^\star) + \nu(x^\star, x)$$

Par inégalité triangulaire, on conclut que l'inégalité est une égalité. Si $\delta(x_0, s) = \infty$, alors l'ensemble $\mathrm{V} \cap \Gamma(x)$ est vide sans quoi, il y aurait un chemin de s_0 à x et on a donc $\mathrm{d}[x] = +\infty$. □

Exercice : Déterminer la complexité temporelle et spatiale de `dijkstra`. On note $n = \mathrm{Card}\ \mathrm{S}$ et $m = \mathrm{Card}\ \mathrm{A}$. Les accès en lecture en écriture dans une liste ou un dictionnaire sont supposés être en $\mathrm{O}(1)$ (complexité amortie pour une liste et complexité moyenne pour un dictionnaire).

Corrigé : On effectue une première boucle de parcours de `S` puis, à l'intérieur de celle-ci, on parcourt tous les sommets de `S` avec un test et éventuellement une affectation puis on parcourt tous les sommets adjacents au sommet `smin` avec un test et éventuellement deux affectations. Le nombre de sommets adjacents à `smin` est majoré par n et on a donc un coût temporel en

$$\sum_{i=1}^{n} \sum_{j=1}^{n} \mathrm{O}(1) = \mathrm{O}(n^2)$$

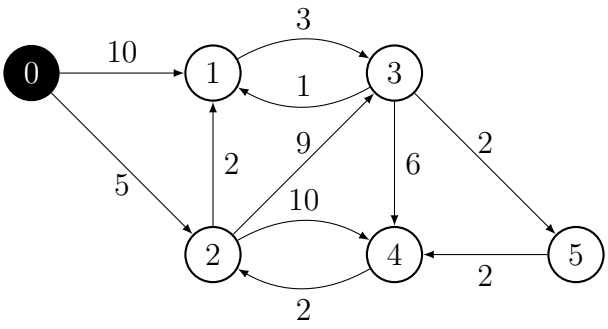
Les dictionnaires `cout` et `djvu` sont initialisés avec autant de clés que de sommets. Le dictionnaire `predec` contient les sommets pour lesquels un relâchement a été effectué donc au plus n. Les autres variables sont en nombre fixe et de taille fixée. Par conséquent, la complexité spatiale est en $\mathrm{O}(n)$.

Remarques : (1) Si le sommet à atteindre est connu, on peut provoquer la sortie de la boucle dès que l'objectif a été visité.

(2) Dans le cas d'un graphe peu dense avec m très inférieur à n^2, on peut améliorer les performances de l'algorithme en utilisant un type de données adapté pour une détermination efficace du sommet non visité à distance minimale : une *file de priorité*. Pour implémenter une telle file de priorité, on peut utiliser une structure de *tas binaire* auquel cas on parvient à une complexité temporelle en $\mathrm{O}((m+n)\log(n))$ ou aussi une structure de *tas de Fibonacci* qui permet d'atteindre un coût en $\mathrm{O}(m+n\log(n))$.

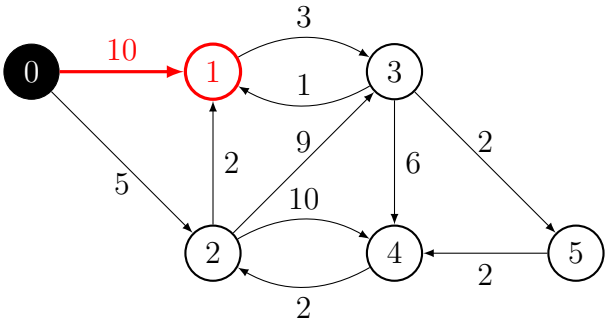
3 Un exemple

• On visite le sommet 0 et on affecte d[0] = 0.



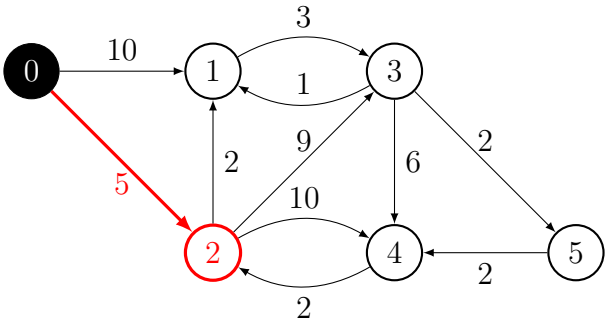

```
sommet= 0
djvu=
{0: True, 1: False, 2: False,
3: False, 4: False, 5: False}
cout=
{0: 0, 1: inf, 2: inf,
3: inf, 4: inf, 5: inf}
predec= {}
```

• Le sommet 1 est non visité et successeur du sommet 0 : on relâche d[1] ← d[0] + $\nu(0, 1)$.



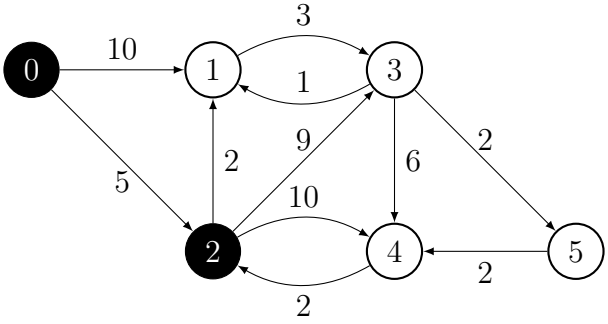

```
sommet= 0
djvu=
{0: True, 1: False, 2: False,
3: False, 4: False, 5: False}
cout=
{0: 0, 1: 10, 2: inf,
3: inf, 4: inf, 5: inf}
predec= {1: 0}
```

• Le sommet 2 est non visité et successeur du sommet 0 : on relâche d[2] ← d[0] + $\nu(0, 2)$.



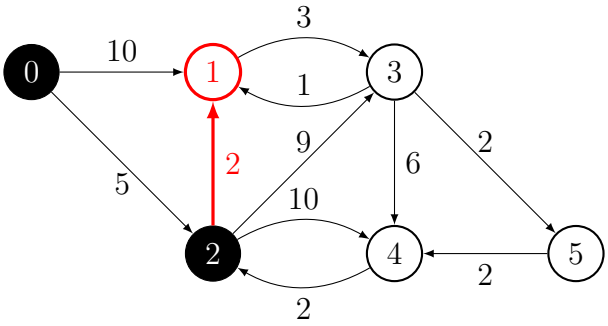

```
sommet= 0
djvu=
{0: True, 1: False, 2: False,
3: False, 4: False, 5: False}
cout=
{0: 0, 1: 10, 2: 5,
3: inf, 4: inf, 5: inf}
predec= {1: 0, 2: 0}
```

• On visite le sommet 2 qui est non visité et à distance minimale du sommet 0.



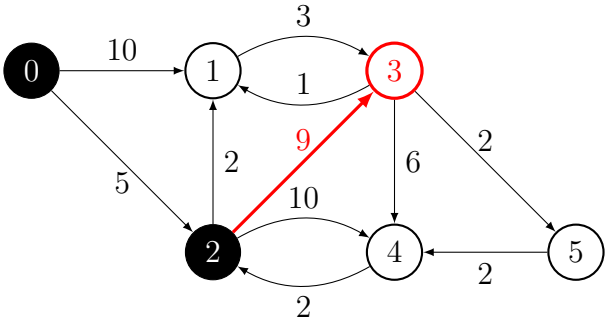

```
sommet= 2
djvu=
{0: True, 1: False, 2: True,
3: False, 4: False, 5: False}
cout=
{0: 0, 1: 10, 2: 5,
3: inf, 4: inf, 5: inf}
predec= {1: 0, 2: 0}
```

• Le sommet 1 est non visité et successeur du sommet 2 : on relâche $d[1] \leftarrow d[2] + \nu(2,1)$.



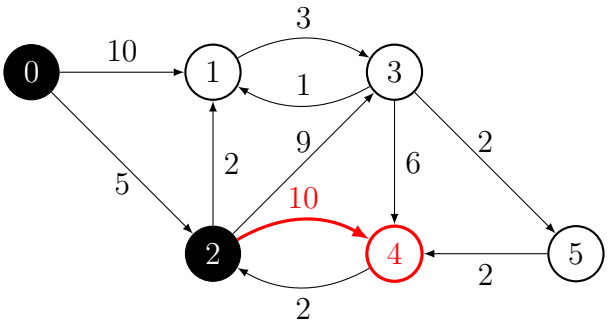

```
sommet= 2
djvu=
{0: True, 1: False, 2: True,
3: False, 4: False, 5: False}
cout=
{0: 0, 1: 7, 2: 5,
3: inf, 4: inf, 5: inf}
predec= {1: 2, 2: 0}
```

• Le sommet 3 est non visité et successeur du sommet 2 : on relâche $d[3] \leftarrow d[2] + \nu(2,3)$.



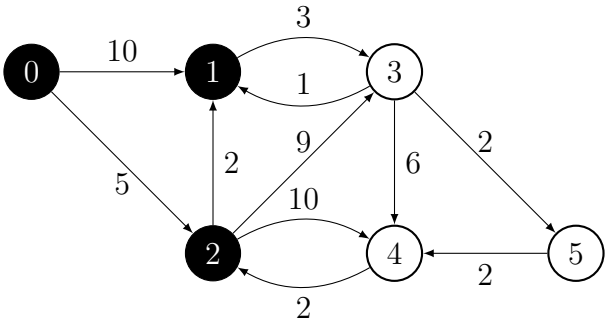

```
sommet= 2
djvu=
{0: True, 1: False, 2: True,
3: False, 4: False, 5: False}
cout=
{0: 0, 1: 7, 2: 5,
3: 14, 4: inf, 5: inf}
predec= {1: 2, 2: 0, 3: 2}
```

• Le sommet 4 est non visité et successeur du sommet 2 : on relâche $d[4] \leftarrow d[2] + \nu(2,4)$.



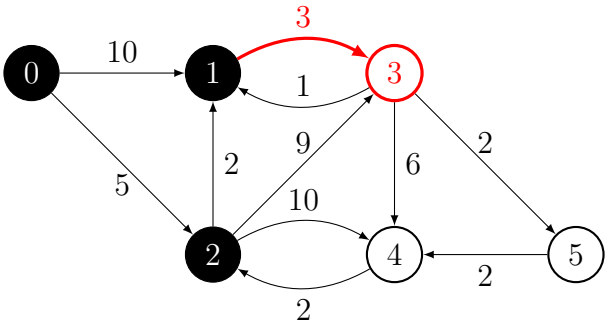

```
sommet= 2
djvu=
{0: True, 1: False, 2: True,
3: False, 4: False, 5: False}
cout=
{0: 0, 1: 7, 2: 5,
3: 14, 4: 15, 5: inf}
predec= {1: 2, 2: 0, 3: 2, 4: 2}
```

• On visite le sommet 1 qui est non visité et à distance minimale du sommet 0.



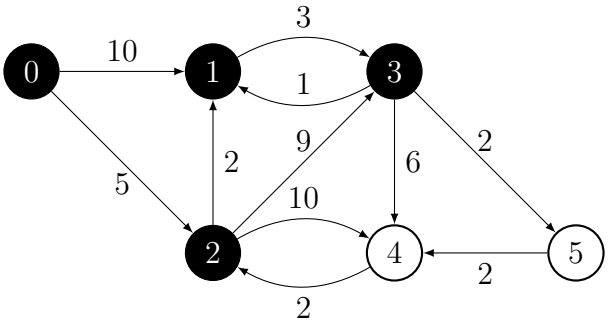

```
sommet= 1
djvu=
{0: True, 1: True, 2: True,
3: False, 4: False, 5: False}
cout=
{0: 0, 1: 7, 2: 5,
3: 14, 4: 15, 5: inf}
predec= {1: 2, 2: 0, 3: 2, 4: 2}
```

• Le sommet 3 est non visité et successeur du sommet 1 : on relâche $d[3] \leftarrow d[1] + \nu(1,3)$.



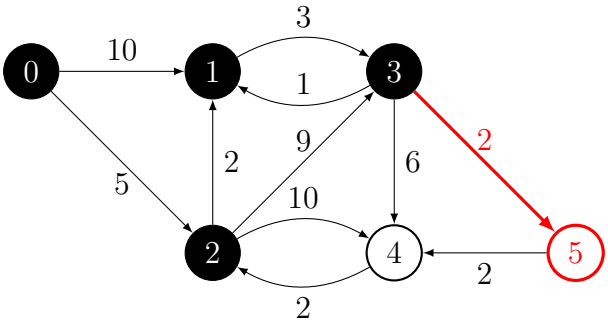

```
sommet= 1
djvu=
{0: True, 1: True, 2: True,
3: False, 4: False, 5: False}
cout=
{0: 0, 1: 7, 2: 5,
3: 10, 4: 15, 5: inf}
predec= {1: 2, 2: 0, 3: 1, 4: 2}
```

• On visite le sommet 3 qui est non visité et à distance minimale du sommet 0.



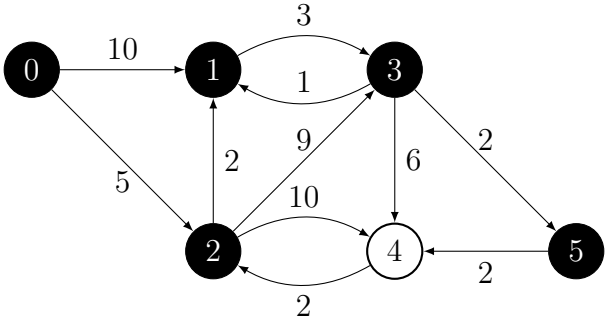

```
sommet= 3
djvu=
{0: True, 1: True, 2: True,
3: True, 4: False, 5: False}
cout=
{0: 0, 1: 7, 2: 5,
3: 10, 4: 15, 5: inf}
predec= {1: 2, 2: 0, 3: 1, 4: 2}
```

• Le sommet 5 est non visité et successeur du sommet 3 : on relâche $d[5] \leftarrow d[3] + \nu(3,5)$. On ne relâche pas d[4] car transiter par le sommet 3 ne raccourcit pas le chemin.



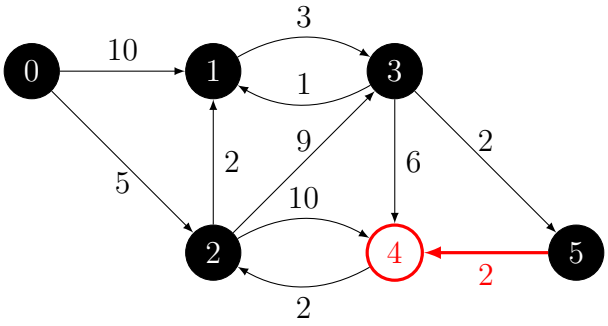

```
sommet= 3
djvu=
{0: True, 1: True, 2: True,
3: True, 4: False, 5: False}
cout=
{0: 0, 1: 7, 2: 5,
3: 10, 4: 15, 5: 12}
predec= {1: 2, 2: 0, 3: 1, 4: 2, 5: 3}
```

• On visite le sommet 5 qui est non visité et à distance minimale du sommet 0.



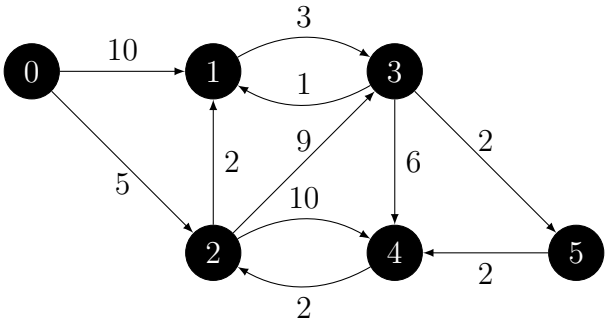

```
sommet= 5
djvu=
{0: True, 1: True, 2: True,
3: True, 4: False, 5: True}
cout=
{0: 0, 1: 7, 2: 5,
3: 10, 4: 15, 5: 12}
predec= {1: 2, 2: 0, 3: 1, 4: 2, 5: 3}
```

• Le sommet 4 est non visité et successeur du sommet 5 : on relâche d[4] ← d[5] + $\nu(5,4)$.




```
sommet= 5
djvu=
{0: True, 1: True, 2: True,
3: True, 4: False, 5: True}
cout=
{0: 0, 1: 7, 2: 5,
3: 10, 4: 14, 5: 12}
predec= {1: 2, 2: 0, 3: 1, 4: 5, 5: 3}
```

• On visite le sommet 4 qui est non visité et à distance minimale du sommet 0.




```
sommet= 4
djvu=
{0: True, 1: True, 2: True,
3: True, 4: True, 5: True}
cout=
{0: 0, 1: 7, 2: 5,
3: 10, 4: 14, 5: 12}
predec= {1: 2, 2: 0, 3: 1, 4: 5, 5: 3}
```

Exercice : On considère les graphes suivants :

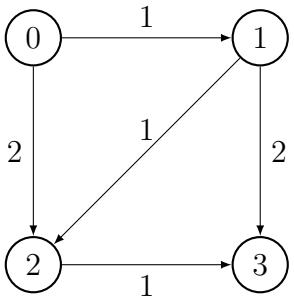


FIGURE 41 – Graphe G_1

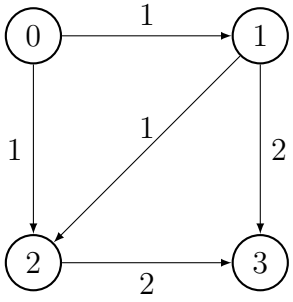


FIGURE 42 – Graphe G_2

1. Identifier tous les plus courts chemins du sommet 0 au sommet 3 pour chacun des graphes.
2. Dans l'algorithme de Dijkstra, pourrait-on, en mémorisant tous les sommets qui réalisent la minimisation dans la boucle, obtenir tous les plus courts chemines de 0 à 3 ?

Corrigé : 1. Pour le graphe G_1, les chemins $(0,1,3)$, $(0,1,2,3)$ et $(0,2,3)$ sont des plus courts chemins. Pour le graphe G_2, les chemins $(0,1,3)$ et $(0,2,3)$ sont des plus courts chemins.

2. Pour le graphe G_1, après visite du sommet 0, le sommet 1 est le seul à distance minimale du sommet 0. Comme on trouve un plus court chemin depuis le sommet 1, on ne va donc pas retenir le comportement depuis le sommet 2 puisque le chemin $(0,2,3)$ n'est pas de coût strictement meilleur. Pour le graphe G_2, après visite du sommet 0, les sommets 1 et 2 sont à distance minimale et si on les mémorise, on peut donc, avec l'algorithme de Dijkstra, trouver tous les plus courts chemins depuis ces sommets.

VIII Algorithme A$\star$

1 Principe

On reprend les notations de la partie VII. L'algorithme A$\star$ est un algorithme de recherche de plus court chemin dans un graphe orienté valué positivement. Il s'agit d'une variante de l'algorithme de Dijkstra. Cet algorithme utilise une *heuristique*, règle reposant sur une expertise orientant la recherche de chemins vers les bons candidats.

Si on suppose que les sommets dans graphe sont repérés par leurs coordonnées dans le plan euclidien $\mathbb{R}^2$, il est raisonnable de privilégier la recherche de chemin d'un sommet source s à un sommet cible c en s'efforçant d'aller dans la direction de c et donc de réduire la distance euclidienne à c. C'est précisément le rôle joué par la fonction heuristique.

Définition 35. *Soit* $\mathrm{G} = (\mathrm{S}, \mathrm{A}, \nu)$ *un graphe orienté valué positivement et* $c \in \mathrm{S}$. *Une* heuristique h *est une fonction* $h : \mathrm{S} \to \mathbb{R}_+$. *L'heuristique* h *est dite* admissible *pour la recherche de plus court chemin au sommet* c *(la cible) si*

$$\forall s \in \mathrm{S} \qquad h(s) \leqslant \delta(s, c)$$

Une heuristique admissible ne surestime pas le coût réel pour atteindre une cible donnée.

Définition 36. *Soit* $\mathrm{G} = (\mathrm{S}, \mathrm{A}, \nu)$ *un graphe orienté valué positivement,* $c \in \mathrm{S}$ *et* h *une heuristique pour la recherche de plus court chemin au sommet* c. *L'heuristique* h *est dite* consistante *si*

$$\forall (s, u) \in \mathrm{A}^2 \qquad h(s) \leqslant \nu(s, u) + h(u)$$

On peut démontrer qu'une heuristique consistante est admissible (voir [7]).

Le principe de l'algorithme A$\star$ consiste à considérer, dans l'algorithme de Dijsktra, pour $u \in \mathrm{S}$, non plus la quantité $\mathrm{d}[u]$ mais $\mathrm{d}[u] + h(u)$, quantité qu'on note $c[u]$ comme *coût* depuis u. On effectue :

- on initialise $\mathrm{d}[deb] \leftarrow 0$, $c[deb] \leftarrow h(deb)$, $\mathrm{d}[u] \leftarrow \infty$ et $c[u] \leftarrow \infty$ pour tout $u \in \mathrm{S} \setminus \{deb\}$;
- tant qu'on n'a pas visité le sommet fin :
  - on détermine un sommet u non visité tel que $c[u]$ est minimal ;
  - le sommet u est considéré comme « visité » ;
  - pour chaque sommet s non visité et successeur de u, on effectue l'opération de *relâchement* :

    si $\mathrm{d}[s] > \mathrm{d}[u] + \nu(u, s)$, alors $\mathrm{d}[s] \leftarrow \mathrm{d}[u] + \nu(u, s)$ et $c[s] \leftarrow \mathrm{d}[s] + h(s)$

    autrement dit : si pour aller à s depuis *deb*, il coûte « moins cher » de passer par `u`, alors on le fait.

Remarques : (1) On peut démontrer qu'avec une heuristique admissible, l'algorithme A$\star$ résout le problème de plus court chemin et qu'avec une heuristique consistante, l'algorithme A$\star$ résout le problème de plus court chemin en complexité optimale (voir [7]).
(2) Dans le cas d'une heuristique nulle, c'est exactement l'algorithme de Dijkstra.

2 Implémentation

On utilisera les dictionnaires suivants :

— `dist` : pour `s` un sommet, `dist[s]` est la valeur de $\mathrm{d}[s]$;
— `cout` : pour `s` un sommet, `cout[s]` est la valeur de $\mathrm{d}[s] + h(s)$;
— `predec` : pour `s` un sommet, `predec[s]` est son prédécesseur dans le plus court chemin depuis s_0 ;
— `djvu` : pour `s` un sommet, `sommet[s]` est un booléen qui dit si le sommet a été déjà vu ou pas.

```
def d2(A,B):
    xA,yA=A
    xB,yB=B
    return np.sqrt((xA-xB)**2+(yA-yB)**2)


def Astar_deb_fin(S,A,deb,fin):
    def h(s):
        return d2(s,fin)
    cout,dist,predec,djvu={},{},{},{}
    for s in S:
        cout[s]=float('inf')
        dist[s]=float('inf')
        djvu[s]=False
    cout[deb]=h(deb)
    dist[deb]=0
    while not djvu[fin]:
        # recherche elt à cout minimal
        smin,cmin=deb,float('inf')
        for s in S:
            if not djvu[s] and cout[s]<cmin:
                smin,cmin=s,cout[s]
        djvu[smin]=True
        # relachement chez les successeurs
        for s,nu in A[smin]:
            if not djvu[s] and dist[s]>dist[smin]+nu:
                dist[s]=dist[smin]+nu
                cout[s]=dist[s]+h(s)
                predec[s]=smin
    return cout,predec
```

```
def d2(A,B):                    # distance euclidienne dans R^2
    xA,yA=A
    xB,yB=B
    return np.sqrt((xA-xB)**2+(yA-yB)**2)

def Astar_deb_fin(S,A,deb,fin):   # recherche d'un plus court chemin de deb à fin
    def h(s):                   # h est l'heuristique
        return d2(s,fin)        # ici : distance entre s et fin
    cout,dist,predec,djvu={},{},{},{}
    for s in S:                 # initialisation :
        cout[s]=float('inf')          # les sommets à cout infini de deb
```

```
        dist[s]=float('inf')            # les sommets à distance infini de deb
        djvu[s]=False             # et non visités
    cout[deb]=h(deb)
    dist[deb]=0
    while not djvu[fin]:
        # recherche elt à cout minimal
        smin,cmin=deb,float('inf')
        for s in S:
            if not djvu[s] and cout[s]<cmin:
                smin,cmin=s,cout[s]
        djvu[smin]=True           # le sommet est considéré visité
        for s,nu in A[smin]:      # pour les successeurs non visités
                                  # on relache
            if not djvu[s] and dist[s]>dist[smin]+nu:
                                  # si passer par smin pour aller en s
                                  # est mieux, alors on actualise
                                  # et le cout prend en compte l'heuristique
                dist[s]=dist[smin]+nu
                cout[s]=dist[s]+h(s)
                predec[s]=smin
    return cout,predec
```

Remarque : Le programme est correct si les sommets `deb` et `fin` sont reliés. Dans le cas contraire, il faut prévoir une sortie de boucle.

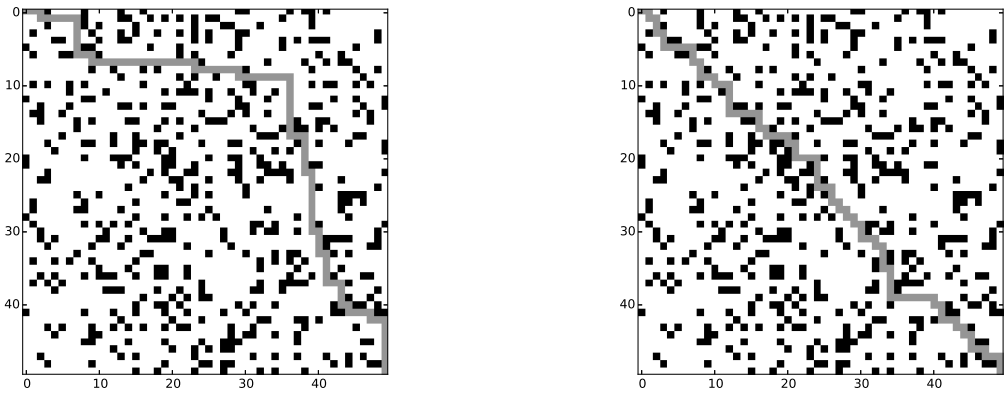


FIGURE 43 – Plus court chemin avec Dijkstra et A$\star$

Pour une configuration en cul-de-sac, les deux algorithmes parcourent toutes les configurations possibles, en vain, avec toutefois plus d'effort pour l'algorithme A$\star$ que l'algorithme de Dijkstra puisqu'il y a le coût lié au calcul d'heuristique.

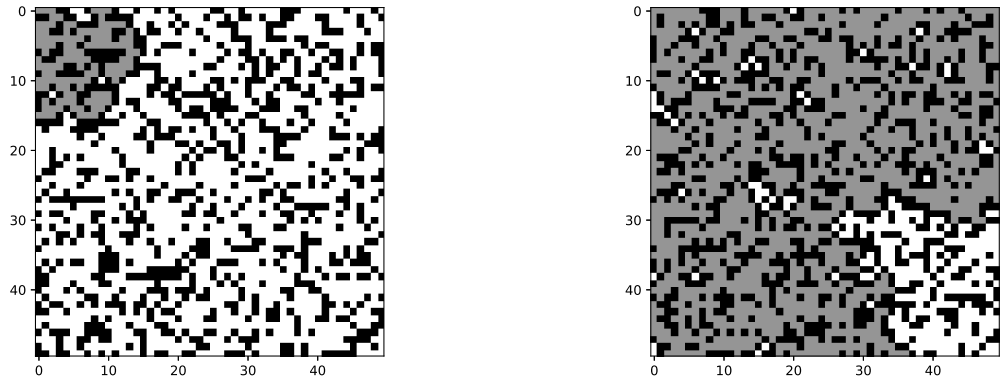


FIGURE 44 – Configurations sans plus court chemin

Le lecteur curieux pourra compléter cette étude en consultant les ouvrages référencés dans la bibliographie.

Références

[1] Alain Bretto, Alain Faisant, François Hennecart, *Éléments de théorie des graphes*, Springer, 2012

[2] Adrian Bondy, U.S.R. Murty, *Graph Theory*, Springer, 2008

[3] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, *Introduction to Algorithms*, Third Edition MIT Press and McGraw-Hill, 2009

[4] Reinhard Diestel, *Graph Theory*, Fifth Edition, Springer, 2017

[5] Robert Sedgewick, Kevin Wayne, *Algorithms*, Fourth Edition, Addison-Wesley, 2011

[6] Dieter Jungnickel, *Graphs, Networks and Algorithms*, Fourth Edition, Springer, 2013

[7] Stuart Russell, Peter Norvig, *Artificial Intelligence, A Modern Approach*, Fourth Edition, Pearson, 2021