

Chapitre 4

Notion de graphe

Ce cours traite de la notion de graphe et de la façon de les implémenter et de les parcourir.

1 Notion de graphe

Définition 1.1

Un **graphe (fini)** $\mathcal{G} = (\mathcal{S}, \mathcal{A})$ est la donnée de deux ensembles finis :

- un ensemble fini $\mathcal{S}$ d'éléments, appelés les **sommets** ;
- un sous-ensemble $\mathcal{A}$ de $\mathcal{S} \times \mathcal{S}$, de couples de sommets, appelés les **arêtes** ou les **arcs**.

Le graphe $\mathcal{G} = (\mathcal{S}, \mathcal{A})$ est dit **non orienté** si :

$$\forall (a, b) \in \mathcal{S} \times \mathcal{S}, (a, b) \in \mathcal{A} \implies (b, a) \in \mathcal{A}.$$

Sinon il est dit **orienté**.

Définissons la notion de sous-graphe d'un graphe.

Définition 1.2

Donnés deux graphes $\mathcal{G} = (\mathcal{S}, \mathcal{A})$ et $\mathcal{G}' = (\mathcal{S}', \mathcal{A}')$, on dit que $\mathcal{G}'$ est un **sous-graphe** de $\mathcal{G}$ si $\mathcal{S}' \subset \mathcal{S}$ et $\mathcal{A}' \subset \mathcal{A}$.

On définit aussi les notions de successeurs et de prédécesseurs d'un sommet, et dans un graphe non orienté, la notion de sommets adjacents.

Définition 1.3

Dans un graphe $\mathcal{G} = (\mathcal{S}, \mathcal{A})$:

- si $(a, b) \in \mathcal{A}$ est une arête, les sommets $a \in \mathcal{S}$ et $b \in \mathcal{S}$ sont appelés respectivement origine et l'extrémité de l'arête. Le sommet b est un **successeur** du sommet a , le sommet a est un **prédécesseur** du sommet b ;
- Une arête de la forme $(a, a) \in \mathcal{A}$ est appelée une **boucle** issue de a .
- dans un graphe non orienté, deux sommets $a \in \mathcal{S}$ et $b \in \mathcal{S}$ sont **adjacents** si $(a, b) \in \mathcal{A}$;

On définit aussi pour chaque sommet ses degrés entrants/sortant ; pour un graphe non-orienté les deux notions coïncident.

Définition 1.4

Dans un graphe :

- Le degré sortant d'un sommet $a \in \mathcal{S}$ est le nombre d'arêtes d'origine a ; on le note $d^+(a)$.

$$\forall a \in \mathcal{S}, d^+(a) = \text{Card}\{(a, s) ; s \in \mathcal{S} \text{ et } (a, s) \in \mathcal{A}\}.$$

- Le degré entrant d'un sommet $a \in \mathcal{S}$ est le nombre d'arêtes d'extrémité a ; on le note $d^-(a)$.

$$\forall a \in \mathcal{S}, d^-(a) = \text{Card}\{(s, a) ; s \in \mathcal{S} \text{ et } (s, a) \in \mathcal{A}\}.$$

- dans un graphe non orienté, les degrés entrants et sortants d'un sommet sont égaux ; on parle alors du **degré** d'un sommet $a \in \mathcal{S}$, noté $d(a) = d^+(a) = d^-(a)$.



Dans la suite, pour un graphe non orienté, on ne donnera qu'une arête sur 2 parmi (a, b) et (b, a) . Par exemple, ci-dessus, on s'autorisera à écrire abusivement :

$$\mathcal{A} = \{(A, C); (B, C); (C, D); (A, B); (A, D); (B, D)\}$$

et on dira que le nombre d'arêtes non orientées, noté $\#\mathcal{A}$, est 6.

Représentation graphique

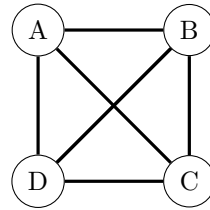
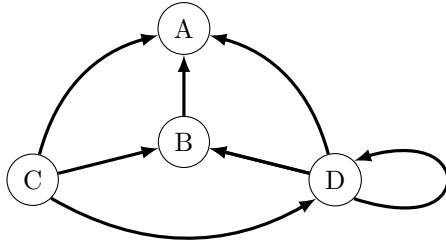
On représente graphiquement un graphe $\mathcal{G} = (\mathcal{S}, \mathcal{A})$ dans le plan, à l'aide d'un motif, point ou cercle, pour chaque sommet (avec pour label le nom du sommet), et d'un arc reliant les sommets a et b pour toute arête $(a, b) \in \mathcal{A}$.

- Pour un graphe orienté, une flèche permet de distinguer l'arête (a, b) de l'arête (b, a) .
- Pour un graphe non orienté, on ne distingue pas l'arête (a, b) de l'arête (b, a) .

Exemple.

- Sur la figure de gauche on a représenté le graphe orienté $\mathcal{G} = (\mathcal{S}, \mathcal{A})$ avec :

$$\mathcal{S} = \{A, B, C, D\} \quad \mathcal{A} = \{(C, A); (C, B); (C, D); (B, A); (D, A); (D, B); (D, D)\}.$$



- Sur la figure de droite, on a représenté le graphe non orienté $\mathcal{G} = (\mathcal{S}, \mathcal{A})$ avec :

$$\mathcal{S} = \{A, B, C, D\}$$

$$\mathcal{A} = \{(A, C); (C, A); (B, C); (C, B); (D, C); (C, D); (B, A);$$

$$(A, B); (D, A); (A, D); (D, B); (B, D)\}$$

Relation binaire associée à un graphe

Un graphe $\mathcal{G} = (\mathcal{S}, \mathcal{A})$ permet de définir une relation binaire $R_{\mathcal{G}}$ sur l'ensemble de ses sommets, c'est-à-dire l'application :

$$R_{\mathcal{G}} : \begin{array}{ccc} \mathcal{S} \times \mathcal{S} & \longrightarrow & \{0, 1\} \\ (a, b) & \longmapsto & aR_{\mathcal{G}}b \end{array} \quad \text{où} \quad aR_{\mathcal{G}}b = \begin{cases} 1 & \text{si } (a, b) \in \mathcal{A} \\ 0 & \text{sinon} \end{cases}$$

Ainsi un graphe $\mathcal{G} = (\mathcal{S}, \mathcal{A})$ est non orienté si et seulement si $R_{\mathcal{G}}$ est symétrique, c'est-à-dire si et seulement si $\forall (a, b) \in \mathcal{S} \times \mathcal{S}, aR_{\mathcal{G}}b = bR_{\mathcal{G}}a$.

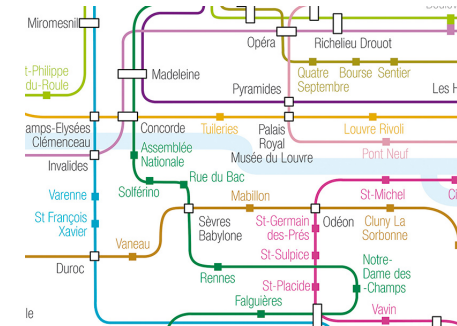
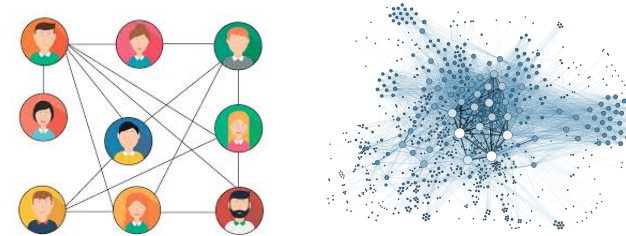


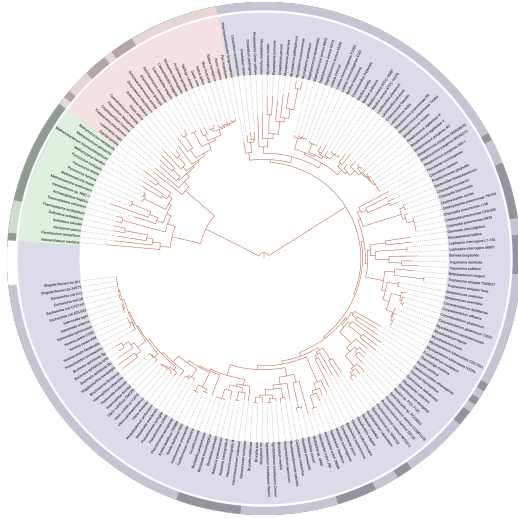
Un graphe n'est qu'une manière graphique de définir une relation binaire (c'est-à-dire une relation ayant deux opérandes) sur un ensemble fini (constitué des sommets), un graphe non orienté définissant une relation symétrique.

C'est la raison pour laquelle l'emploi des graphes est naturellement si prépondérant.

Exemples de situation bien modélisées par un graphe

Énormément de situation peuvent se modéliser à l'aide d'un graphe.

Réseau de transport :**Réseaux sociaux :****Graphe du web :****Arbre Phylogénétique du vivant :**



Graphe non orienté sous-jacent

Un graphe orienté contient plus d'information qu'un graphe non orienté, et à tout graphe on peut lui associer un graphe non orienté, dit sous-jacent, ayant même ensemble de sommets, et obtenu en symétrisant sa relation associée, c'est-à-dire en supprimant toutes les flèches des arcs dans sa représentation graphique.

Définition 1.5

Donné un **graphe (fini)** $\mathcal{G} = (\mathcal{S}, \mathcal{A})$, son **graphe non orienté sous-jacent** est le graphe $\overline{\mathcal{G}} = (\overline{\mathcal{S}}, \overline{\mathcal{A}})$ défini par :

$$\overline{\mathcal{S}} = \mathcal{S} \quad \text{et} \quad \overline{\mathcal{A}} = \left\{ (a, b) \in \mathcal{S} \times \mathcal{S} \mid (a, b) \in \mathcal{A} \text{ ou } (b, a) \in \mathcal{A} \right\}.$$

Évidemment, pour un graphe non orienté $\mathcal{G}$, on a $\overline{\mathcal{G}} = \mathcal{G}$, et cette définition ne présente d'intérêt que pour un graphe orienté.

1.1 Chemin et connexité

Chemin et cycle

La notion de chemin entre sommets est une notion importante dans les graphes ; elle correspond à l'idée intuitive de chemin reliant deux sommets par une succession d'arêtes.

Définition 1.6

Dans un graphe $\mathcal{G} = (\mathcal{S}, \mathcal{A})$, donnés deux sommets $a \in \mathcal{S}$ et $b \in \mathcal{S}$, un **chemin de a à b** est une suite finie de sommets

$$s_0, s_1, \dots, s_i, s_{i+1}, \dots, s_{n-1}, s_n$$

vérifiant : $s_0 = a, s_n = b$ et

$$\forall i \in \llbracket 0, n-1 \rrbracket, (s_i, s_{i+1}) \in \mathcal{A}.$$

L'entier n est la **longueur** du chemin.

La distance entre deux sommets reliés par un chemin est la longueur minimale d'un chemin les reliant.

Un chemin reliant deux mêmes sommets est appelé un cycle.

Définition 1.7

Un **cycle** issu de a est un chemin d'un sommet $a \in \mathcal{S}$ à lui-même, et de longueur ≥ 1 .

Composantes connexes. Graphe connexe

Pour un graphe non orienté, être relié par un chemin est une relation d'équivalence sur les sommets.



Une relation binaire R sur $\mathcal{S}$ est une **relation d'équivalence** si elle est :

- Réflexive. $\forall s \in \mathcal{S}, sRs.$
- Symétrique. $\forall (s, s') \in \mathcal{S}^2, sRs' \implies s'R s.$
- Transitive. $\forall (s, s', s'') \in \mathcal{S}^3, sRs' \text{ et } s'R s'' \implies sRs''.$

Propriété 1.1

Dans un graphe $\mathcal{G}$ non orienté, la relation "être relié par un chemin", est une relation d'équivalence sur l'ensemble $\mathcal{S}$ de ses sommets.

Preuve. Notons la relation R ; elle est bien définie, par aRb si il existe un chemin dans $\mathcal{G}$ de a à b . Il s'agit de montrer qu'elle est réflexive, symétrique et transitive.

Réflexivité. Pour un sommet $a \in \mathcal{S}$, la suite a de longueur 0 est un chemin de a à a ; ainsi aRa .

Symétrie. Soit a, b deux sommets. Si aRb , il existe un chemin $(s_k)_{0 \leq k \leq n} = s_0, \dots, s_n$ de a à b . Le graphe étant non orienté : $(s_i, s_{i+1}) \in \mathcal{A} \iff (s_{i+1}, s_i) \in \mathcal{A}$. Ainsi la suite de sommets $(s'_k)_{0 \leq k \leq n}$ définie par $s'_k = s_{n-k}$ est un chemin de b à a . Donc bRa .

Transitivité. Soit a, b, c trois sommets tels que aRb et bRc , et $(s_k)_{0 \leq k \leq n}$, respectivement $(s'_k)_{0 \leq k \leq m}$, un chemin de a à b , respectivement de b à c . Alors nécessairement $s_0 = a, s_n = s'_0 = b$ et $s'_m = c$. Ainsi, la suite de sommets :

$$a = s_0, s_1, \dots, s_n = s'_0, s'_1, \dots, s'_m = c$$

est un chemin de a à c (de longueur $n + m$). Donc aRc .

Donnée une relation d'équivalence sur un ensemble $\mathcal{S}$, l'ensemble se partitionne en classes d'équivalences.



Soit S un ensemble et R une relation d'équivalence; la **classe d'équivalence** de $s \in S$ est $[s] = \{s' \in S \mid sRs'\}$.

Les classes d'équivalence forment une partition de S : il existe une famille $(s_i)_{i \in I}$ d'éléments de S tels que :

$$S = \bigcup_{i \in I} [s_i] \quad \text{et} \quad \forall (i, j) \in I^2, i \neq j \implies [s_i] \cap [s_j] = \emptyset.$$

Donné un graphe $\mathcal{G} = (\mathcal{S}, \mathcal{A})$ et $\mathcal{S}' \subset \mathcal{S}$ un sous-ensemble de sommets, le **sous-graphe** de $\mathcal{G}$ restreint aux sommets $\mathcal{S}'$ est le graphe $(\mathcal{S}', \mathcal{A}')$ avec $\mathcal{A}' = \mathcal{A} \cap (\mathcal{S}' \times \mathcal{S}')$, c'est-à-dire que c'est le graphe obtenu de $\mathcal{G}$ en ne conservant que les sommets de $\mathcal{S}'$ et les arêtes reliant les sommets de $\mathcal{S}'$.

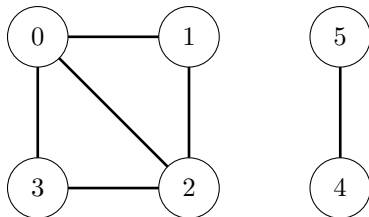
Définition 1.8

• Pour un graphe non orienté $\mathcal{G}$, ses **composantes connexes**, sont les sous-graphes de $\mathcal{G}$ restreints aux classes d'équivalences des sommets pour la relation être relié par un chemin.

• Pour un graphe orienté $\mathcal{G}$, ses **composantes connexes**, sont les sous-graphes de $\mathcal{G}$ restreints aux classes d'équivalences, de son sous-graphe non orienté sous-jacent $\overline{\mathcal{G}}$, pour la relation être relié par un chemin.

Les composantes connexes sont intuitivement " les morceaux " du graphe. Par exemple, le graphe non orienté $\mathcal{G} = (\mathcal{S}, \mathcal{A})$ défini par :

$$\mathcal{S} = \{0, 1, 2, 3, 4, 5\} \quad \mathcal{A} = \{(0, 1); (0, 2); (0, 3); (1, 2); (2, 3); (4, 5)\}$$



□ a deux composantes connexes, $\mathcal{G}_1$ (graphe de gauche) et $\mathcal{G}_2$ (graphe de droite) :

$$\mathcal{G}_1 = (\mathcal{S}_1, \mathcal{A}_1) \quad : \quad \mathcal{S}_1 = \{0, 1, 2, 3\} \quad \mathcal{A}_1 = \{(0, 1); (0, 2); (0, 3); (1, 2); (2, 3)\}$$

$$\mathcal{G}_2 = (\mathcal{S}_2, \mathcal{A}_2) \quad : \quad \mathcal{S}_2 = \{4, 5\} \quad \mathcal{A}_2 = \{(4, 5)\}$$

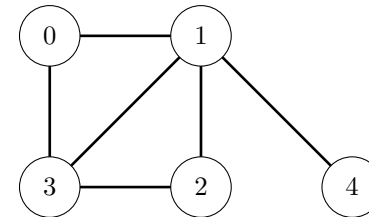
La notion de composantes connexes permet de définir la connexité d'un graphe, intuitivement lorsqu'il est constitué d'un seul morceau.

Définition 1.9

- Un **graphe non orienté** est dit **connexe** lorsqu'il n'a qu'une seule composante connexe.
- Un **graphe orienté** est dit **connexe** lorsque son graphe non orienté sous-jacent est connexe.

Par exemple, le graphe non orienté ci-dessus est non connexe (il a deux composantes connexes), alors que le graphe non orienté suivant est connexe.

$$\mathcal{G} = (\mathcal{S}, \mathcal{A}) \quad : \quad \mathcal{S} = \{0, 1, 2, 3, 4\} \quad \mathcal{A} = \{(0, 1); (0, 3); (1, 2); (1, 3); (2, 3); (1, 4)\}$$



Pour un graphe non orienté, on peut aussi donner la définition alternative.

Propriété 1.2

Un graphe non orienté est connexe lorsque chaque couple de ses sommets est relié par un chemin.

Preuve. Le graphe est connexe si et seulement si la relation " être relié par un chemin " n'a qu'une seule classe d'équivalence, si et seulement si chaque couple de sommet est en relation, c'est-à-dire est relié par un chemin. □

1.2 Arbres

Définition 1.10

Dans un graphe non orienté $\mathcal{G} = (\mathcal{S}, \mathcal{A})$, un chemin (respectivement un cycle) $(s_k)_{0 \leq k \leq n}$ est **réduit** lorsque $\forall k \in \llbracket 1, n-1 \rrbracket, s_{k-1} \neq s_{k+1}$.

Un chemin est réduit lorsqu'il ne contient pas d'aller-retour s_{k-1}, s_k, s_{k-1} . Il est facile de démontrer par récurrence sur la longueur du chemin, que s'il existe un chemin reliant deux sommets a et b , alors il existe un chemin réduit de a à b et empruntant les mêmes arêtes.

Définition 1.11

Un arbre est un graphe non orienté, connexe, n'ayant aucun cycle réduit.

Une définition alternative est donnée par la propriété suivante.

Propriété 1.3

Un graphe non orienté connexe est un arbre si et seulement si pour tout couple de sommet a, b il existe un unique chemin réduit de a à b .

Preuve. Soit $\mathcal{G} = (\mathcal{S}, \mathcal{A})$ un graphe non orienté connexe.

⊖ On montre la contraposée. Si $\mathcal{G}$ n'est pas un arbre, alors $\mathcal{G}$ contient un cycle réduit $\gamma = (s_0, \dots, s_n)$ (avec $s_0 = s_n = a$ et $n > 0$), et γ et (a) sont deux chemins réduits de a à a .

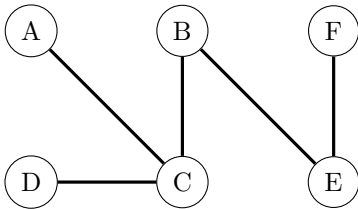
⊕ On montre l'implication directe. Soit deux sommets a et b et deux chemins réduits de a à b , qu'on notera $\alpha = (s_k)_{0 \leq k \leq n}$ et $\beta = (s'_k)_{0 \leq k \leq m}$. Alors le chemin

$$(\underbrace{s_0, s_1, \dots, s_{n-2}, s_{n-1}, s_n}_{\text{réduit}}, \underbrace{s'_{m-1}, s'_{m-2}, \dots, s'_1, s'_0}_{\text{réduit}})$$

est un cycle de longueur $n+m$ issu du sommet a . Donc il n'est pas réduit et nécessairement $s_{n-1} = s'_{m-1}$. On a donc aussi le cycle de longueur $n+m-2$ issu de a :

$$(\underbrace{s_0, s_1, \dots, s_{n-2}, s_{n-1}}_{\text{réduit}}, \underbrace{s'_{m-2}, \dots, s'_1, s'_0}_{\text{réduit}}).$$

Et donc $s_{n-2} = s'_{m-2}$; en poursuivant par le même argument, on obtient finalement $n = m$ et $\forall k \in \llbracket 0, n \rrbracket, s_k = s'_k$. Ainsi $\alpha = \beta$; on a prouvé que $\forall (a, b) \in \mathcal{S}^2$, il existe un unique chemin réduit de a à b . $\square$



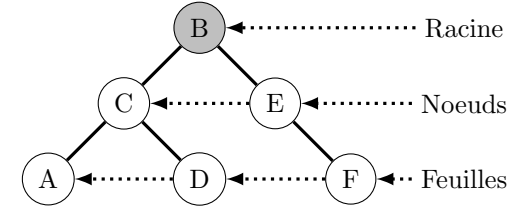
Exemple. Le graphe ci-dessus est un arbre. Par exemple, le seul chemin réduit de A à F est (A,C,B,E,F).

Arbre enraciné. Arbre régulier

Donné un arbre, il est toujours possible d'en choisir un sommet et de représenter graphiquement l'arbre par niveaux : le sommet choisi est appelé la racine et est disposé

au niveau 0. À chaque niveau $k = 1, 2$, etc. on dispose les sommets qui sont reliés à la racine par un chemin réduit de longueur k . Tous les sommets intermédiaires sont alors appelés des noeuds, les sommets terminaux sont appelés des feuilles.

Exemple. En reprenant l'arbre décrit ci-dessus, et en choisissant pour racine le sommet B, on peut représenter l'arbre sous la forme :



Le sommet B est la racine, les sommets C et E sont les noeuds, les sommets A, D, F sont les feuilles.

Définition 1.12

Un arbre enraciné $(\mathcal{G}, s_0)$ est la donnée d'un arbre $\mathcal{G} = (\mathcal{S}, \mathcal{A})$ et d'un sommet $s_0 \in \mathcal{S}$.

- Le sommet s_0 s'appelle la **racine**.
- Les sommets de $\mathcal{S} \setminus \{s_0\}$ ayant pour degré 1 sont appelés les **feuilles**.
- Tous les autres sommets sont appelés des **noeuds**.

Pour un arbre enraciné on utilise souvent la terminologie de père et fils d'un sommet, ou de prédécesseur ou successeur, de la façon suivante : on considère la représentation graphique par niveau de l'arbre enraciné.

- Le successeur (ou fils) d'un sommet est un sommet adjacent situé au niveau inférieur.
- Le prédécesseur (ou père) d'un sommet est un sommet adjacent situé au niveau supérieur.

Ainsi, la racine de l'arbre est l'unique sommet n'ayant pas de père, les feuilles sont les sommets n'ayant pas de fils.

Lorsque la racine et les noeuds ont tous le même nombre de successeurs, l'arbre enraciné est dit régulier.

Définition 1.13

Un arbre enraciné $(\mathcal{G}, s_0)$ est dit **régulier** si pour un entier $n \in \mathbb{N}^*$:

- la racine a pour degré n ;
- tous les noeuds ont pour degré $n+1$.

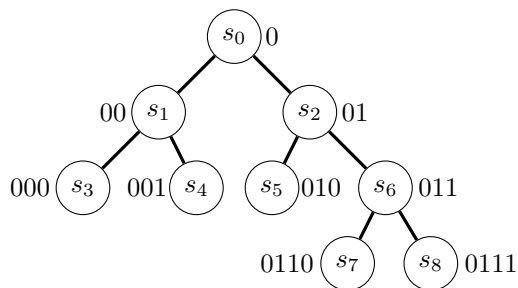
Dans ce cas on dit aussi que l'arbre est n -aire.

Un exemple que nous avons souvent rencontré, fondamental en informatique, est lorsque

$n = 2$. On parle alors d'**arbre binaire** (voir chapitre 6).

Exemple. L'arbre enraciné ci-dessus n'est pas régulier : les noeuds C et E ont des degrés différents, respectivement 3 et 2. L'arbre suivant est un arbre binaire.

Un arbre binaire peut être représenté à l'aide d'un ensemble d'étiquettes écrites en binaire, une étiquette pour chaque sommet, en fixant une convention : par exemple, la racine a l'étiquette 0, et pour chaque sommet l'étiquette est obtenue de celle du père en ajoutant en suffixe 0 pour le fils gauche et 1 pour le fils droit.



L'arbre binaire ci-dessus est entièrement représenté par l'ensemble :

$$\{0, 00, 01, 000, 001, 010, 011, 0110, 0111\}.$$

Arbre couvrant d'un graphe non orienté connexe

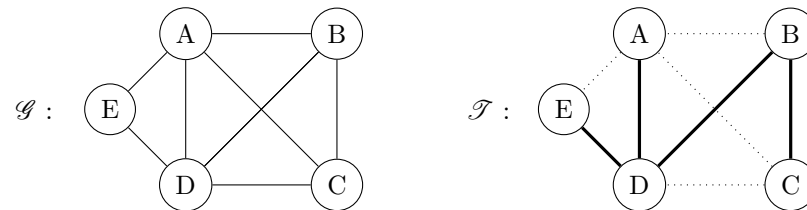
Définition 1.14

Soit $\mathcal{G} = (\mathcal{S}, \mathcal{A})$ un graphe non orienté connexe. Un **arbre couvrant** de $\mathcal{G}$ est un sous-graphe $\mathcal{T} = (\mathcal{S}', \mathcal{A}')$ de $\mathcal{G}$ vérifiant :

- $\mathcal{T}$ est un arbre ;
- $\mathcal{T}$ a mêmes sommets que $\mathcal{G}$: $\mathcal{S}' = \mathcal{S}$;
- toute arête de $\mathcal{T}$ est une arête de $\mathcal{G}$: $\mathcal{A}' \subset \mathcal{A}$.

Si cette notion est importante, c'est parce que tout graphe non orienté connexe contient un arbre couvrant ; l'arbre couvrant n'est pas unique, à moins que $\mathcal{G}$ soit lui-même un arbre (dans quel cas il est égal à son arbre couvrant).

• **Exemple.** Voici un graphe non orienté connexe $\mathcal{G}$ et un arbre couvrant $\mathcal{T}$ de $\mathcal{G}$.



Propriété 1.4

Tout graphe non orienté connexe admet un arbre couvrant.

Preuve. Soit $\mathcal{G} = (\mathcal{S}, \mathcal{A})$ un graphe non orienté connexe. Choisissons arbitrairement un sommet $s_0 \in \mathcal{S}$. Puisque le graphe $\mathcal{G}$ est connexe, pour tout sommet $s \in \mathcal{S}$, il existe un chemin reliant s_0 à s . Donc l'entier $k(s)$, longueur minimale d'un chemin de s_0 à s , est bien défini ; pour $s \neq s_0$, $k(s) \in \mathbb{N}^*$.

Construisons un arbre enraciné de la façon suivante. Chaque sommet $s \in \mathcal{S}$ est disposé au niveau $s(k)$. Tous les sommets au niveau 1 sont donc adjacents avec la racine s_0 : on les relie à s_0 par une arête de $\mathcal{A}$. De la même façon, pour chaque sommet s au niveau $k+1$, choisissons un chemin de longueur minimale de s_0 à s , disons $(s_0, \dots, s', s)$. Les sommets s' et s sont adjacents et nécessairement $k(s') = k$ (autrement on aurait un chemin de longueur $< k+1$ de s_0 à s). On relie alors s' à s par l'arête $(s, s') \in \mathcal{A}$. Notons $\mathcal{T}$ le graphe non orienté sous-jacent au graphe obtenu (définition 1.5) ; alors $\mathcal{T} = (\mathcal{S}, \mathcal{A}')$ a mêmes sommets que $\mathcal{G}$ et puisque $\mathcal{G}$ est non orienté $\mathcal{A}' \subset \mathcal{A}$. Le graphe $\mathcal{T}$ est non orienté, et connexe puisque par construction chaque sommet est relié par un chemin au sommet s_0 .

Chaque sommet de $\mathcal{T}$ situé sur un niveau k a par construction au plus deux sommets adjacents, situés aux niveaux inférieur $k-1$ et supérieur $k+1$. Le graphe $\mathcal{T}$ ne peut pas contenir de cycle réduit ; en effet, supposons qu'il existe un cycle réduit, alors le sommet du cycle ayant le niveau maximal $k_{\max}$ serait adjacent à deux sommets situés sur le même niveau $k_{\max}-1$, ce qui est contradictoire. Ainsi le graphe $\mathcal{T} = (\mathcal{S}, \mathcal{A}')$ est un arbre ayant même ensemble de sommet $\mathcal{S}$ que $\mathcal{G}$, et dont toutes les arêtes sont dans $\mathcal{A}$. L'arbre $\mathcal{T}$ est un arbre couvrant de $\mathcal{G}$. $\square$

Propriété 1.5

Soit $\mathcal{G}$ un graphe non orienté connexe et $\mathcal{T}$ un arbre couvrant de $\mathcal{G}$. Alors $\mathcal{G}$ est un arbre si et seulement si $\mathcal{G} = \mathcal{T}$.

Preuve. Si $\mathcal{G} = \mathcal{T}$ alors $\mathcal{G}$ est un arbre. Réciproquement, montrons la contraposée. Supposons que $\mathcal{G} \neq \mathcal{T}$, alors il existe une arête (s, s') de $\mathcal{G}$ qui n'est pas dans $\mathcal{T}$. Soit un chemin réduit $(s, s_1, \dots, s_{n-1}, s')$ de s à s' dans $\mathcal{T}$ (propriété 1.3). Alors $(s, s_1, \dots, s_{n-1}, s', s)$ est un cycle en $\mathcal{G}$ qui est réduit puisque $(s, s_1, \dots, s_{n-1}, s')$ est réduit et $s_{n-1} \neq s$, et donc $\mathcal{G}$ n'est pas un arbre. $\square$

Caractérisation des arbres

Parmi les graphes non orientés connexes, on a la caractérisation suivante des arbres.

Propriété 1.6

Soit $\mathcal{G} = (\mathcal{S}, \mathcal{A})$ un graphe non orienté connexe. En notant $\#\mathcal{S} = \text{Card}(\mathcal{S})$ le nombre de sommets, $\#\mathcal{A} = \text{Card}(\mathcal{A})/2$ le nombre d'arêtes non orientées. Alors $\#\mathcal{S} \leq \#\mathcal{A} + 1$ et $\mathcal{G}$ est un arbre si et seulement si :

$$\#\mathcal{S} = \#\mathcal{A} + 1.$$

Preuve. Tout d'abord remarquons que pour un arbre enraciné, $\#\mathcal{S} = \#\mathcal{A} + 1$. En effet par une récurrence immédiate sur le nombre d'arête :

– *Initialisation.* La proposition est trivialement vraie pour un arbre ayant un seul sommet et aucune arête.

– *Hérédité.* Si la propriété est vraie au rang n . Si dans un arbre enraciné de $n + 1$ arêtes on supprime une feuille et l'arête qui la relie à son père, on obtient un arbre enraciné avec $\#\mathcal{S}$ et $\#\mathcal{A}$ diminués de 1, ayant n arêtes, et en appliquant l'hypothèse de récurrence, la proposition demeure vraie au rang $n + 1$.

En particulier, on a prouvé que dans un arbre $(\mathcal{S}, \mathcal{A})$, on a la relation $\#\mathcal{S} = \#\mathcal{A} + 1$.

Considérons l'arbre couvrant $\mathcal{T} = (\mathcal{S}, \mathcal{A}')$ de $\mathcal{G}$ (propriété 1.4). Alors $\#\mathcal{S} = \#\mathcal{A}' + 1$ et puisque $\mathcal{A}' \subset \mathcal{A}$, $\#\mathcal{S} \leq \#\mathcal{A} + 1$; cela prouve la première assertion.

Pour prouver la deuxième assertion, utilisons le fait que $\mathcal{G}$ est un arbre si et seulement si il est égal à l'arbre couvrant $\mathcal{T} = (\mathcal{S}, \mathcal{A}')$ (propriété 1.5), si et seulement si $\mathcal{A}' = \mathcal{A}$, si et seulement si $\#\mathcal{S} = \#\mathcal{A} + 1$. $\square$

1.3 Matrice d'adjacence d'un graphe

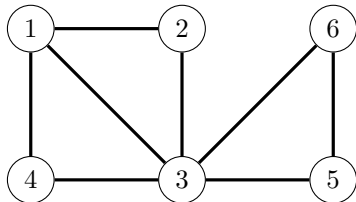
Donné un graphe $\mathcal{G} = (\mathcal{S}, \mathcal{A})$, on peut le représenter à l'aide d'une matrice d'adjacence, après avoir numéroté ses sommets de 1 jusqu'à $\text{Card}(\mathcal{S})$.

Définition 1.15

Soit un graphe $\mathcal{G} = (\mathcal{S}, \mathcal{A})$ ayant n sommets. Donnée une bijection $s : \llbracket 1, n \rrbracket \rightarrow \mathcal{S}$, la **matrice d'adjacence** du graphe est la matrice carrée $A \in \mathcal{M}_n(\mathbb{R})$ constituée de 0 et de 1, et définie par :

$$\forall (i, j) \in \llbracket 1, n \rrbracket^2, \quad A_{i,j} = \begin{cases} 1 & \text{si } (s(i), s(j)) \in \mathcal{A} \\ 0 & \text{sinon} \end{cases}$$

Exemple. Le graphe non orienté :



a pour matrice d'adjacence :

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

Bien sûr la matrice d'adjacence, n'est pas unique, elle dépend de la "numérotation" des sommets. Mais changer cette numérotation ne fait qu'appliquer une suite d'échanges simultanés des lignes et colonnes $L_i \leftrightarrow L_j$ et $C_i \leftrightarrow C_j$.

Propriété 1.7

Une matrice d'adjacence d'un graphe $\mathcal{G}$ est symétrique si et seulement si le graphe $\mathcal{G}$ est non orienté.

Preuve. Immédiate par définition. $\square$

D'autres propriétés plus remarquables relient le graphe à sa matrice d'adjacence.

Propriété 1.8

Soit $\mathcal{G}$ un graphe ayant n sommets et $A \in \mathcal{M}_n(\mathbb{R})$ une matrice d'adjacence de $\mathcal{G}$. Pour tout $k \in \mathbb{N}^*$ et tout $i, j \in \llbracket 1, n \rrbracket$, l'élément $A_{i,j}^k$ ligne i colonne j de la matrice A^k est égal au nombre de chemins de longueur k dans $\mathcal{G}$ allant du sommet numéroté i au sommet numéroté j .

Preuve. On démontre la propriété par récurrence sur k . Pour $i \in \llbracket 1, n \rrbracket$ notons s_k le sommet numéroté k .

Initialisation. Si $k = 0$. Soit $(i, j) \in \llbracket 1, n \rrbracket^2$. La matrice A^0 est l'identité, et il existe un chemin de longueur 0 de s_i à s_j si et seulement si $i = j$. La proposition de récurrence est vraie.

Hérédité. Supposons la proposition vraie au rang $k \in \mathbb{N}^*$. Soit $(i, j) \in \llbracket 1, n \rrbracket^2$, on a :

$$A_{i,j}^{k+1} = \sum_{t=1}^n A_{i,t}^k \times A_{t,j}$$

où $A_{t,j}$ vaut 1 si et seulement si (s_t, s_j) est une arête, et 0 sinon. Par hypothèse de récurrence $A_{i,t}^k$ est le nombre de chemins de longueur k de s_i à s_t .

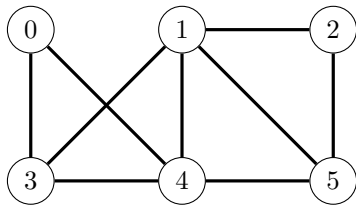
Or pour chaque sommet $s_t \in \mathcal{S}$ tel que (s_t, s_j) soit une arête, chaque chemin de s_i à s_t de longueur k donne un chemin différent de s_i à s_j de longueur $k + 1$.

Ainsi $A_{i,j}^{k+1}$ est égal au nombre de chemins de longueur $k + 1$ allant de s_i à s_j . La proposition de récurrence demeure vraie au rang $k + 1$. $\square$

2 Implémentation des graphes

2.1 Implémentation d'un graphe par une liste d'adjacence

On stocke dans une liste (ou un dictionnaire) le nombre de sommets et la liste des sommets adjacents (respectivement successeurs) pour chaque sommet d'un graphe non orienté (respectivement orienté) ; si la liste est L , $L[i]$ contiendra la liste des sommets adjacents à i . Par exemple, pour le graphe non orienté :



on peut déclarer :

```
Graphe = [ 6, [ [3, 4],
                [2, 3, 4, 5],
                [1, 5],
                [0, 1, 4],
                [0, 1, 3, 5],
                [1, 2, 4]] ]
```

Liste des sommets adjacents (successeurs) s'obtient en $O(1)$; c'est l'avantage de cette implantation.

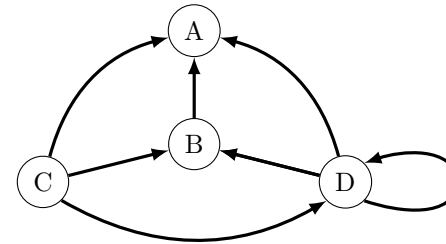
Cette approche est efficace pour des graphes ayant peu d'arêtes.

2.2 Implémentation d'un graphe par une matrice d'adjacence

Pour l'implémentation d'un graphe, un usage répandu, notamment lorsque le graphe comporte beaucoup d'arêtes, est d'utiliser une matrice d'adjacence. Il faut alors identifier ses sommets avec les entiers 0, 1, 2, etc. qui donneront les indices des lignes et colonnes dans la matrice. Une liste pourra permettre de stocker aux mêmes indices les identifiants des sommets, si nécessaire. Un dictionnaire permet d'obtenir les indices correspondant aux sommets.

• **Exemple.** Pour le graphe orienté : $\mathcal{G} = (\mathcal{S}, \mathcal{A})$ avec :

$$\mathcal{S} = \{A, B, C, D\} \quad \mathcal{A} = \{(C, A); (C, B); (C, D); (B, A); (D, A); (D, B); (D, D)\}$$



	A	B	C	D
A	0	0	0	0
B	1	0	0	0
C	1	1	0	1
D	1	1	0	1

• Implantation du graphe.

```
from numpy import array
```

```
A = array([[0,0,0,0], # Matrice d'adjacence
           [1,0,0,0],
           [1,1,0,1],
           [1,1,0,1]])
```

```
S = ['A', 'B', 'C', 'D'] # Sommets
```

• On peut alors obtenir la liste des arêtes à l'aide de la fonction :

```
def aretes(matrice, sommets):
    aretes = []
    n = len(matrice)
    for i in range(n):
        for j in range(n):
            if matrice[i,j]:
                aretes.append((sommets[i],sommets[j]))
    return aretes
```

Comme on le voit :

```
In [2]: aretes(A,S)
```

```
Out[2]:
```

```
[('B', 'A'), ('C', 'A'), ('C', 'B'), ('C', 'D'), ('D', 'A'),
 ('D', 'B'), ('D', 'D')]
```

• Les successeurs et prédécesseurs d'un sommet s :

```
def successeurs(matrice, sommets, s):
    i = sommets.index(s)
    succ = []
```



```

for j in range(len(matrice)):
    if matrice[i,j]:
        succ.append(sommets[j])
return succ

def predecesseurs(matrice,sommets,s):
    j = sommets.index(s)
    pred = []
    for i in range(len(matrice)):
        if matrice[i,j]:
            pred.append(sommets[i])
    return pred

```

Par exemple :

```

In [3]: successeurs(A,S,'C')
Out[3]: ['A', 'B', 'D']

```

```

In [4]: predecesseurs(A,S,'A')
Out[4]: ['B', 'C', 'D']

```

- Ou encore déterminer si le graphe est non orienté.

```

def nonOriente(A):
    n = len(A)
    for i in range(n):
        for j in range(i):
            if A[i,j] != A[j,i]:
                return False
    return True

```

```

In [5]: nonOriente(A)
Out[5]: False

```

- Constituer le graphe non orienté sous-jacent.

```

def grapheNonOriente(A):
    n = len(A)
    B = array([[0]*n for k in range(n)])
    for i in range(n):
        for j in range(i+1):
            if A[i,j]==1 or A[j,i]==1:

```

```

B[i,j] = B[j,i] = 1

```

```

return B

```

```

In [6]: grapheNonOriente(A)
Out[6]:
array([[0, 1, 1, 1],
       [1, 0, 1, 1],
       [1, 1, 0, 1],
       [1, 1, 1, 1]])

```

- Pour un graphe non orienté obtenir la liste des sommets adjacents à un sommet.

```

def adjacents(A,sommets,s):
    i = sommets.index(s)
    L = []
    for j in range(len(A)):
        if A[i,j]:
            L.append(sommets[j])
    return L

```

```

In [7]: B = grapheNonOriente(A)
In [8]: adjacents(B,S,'D')
Out[8]: ['A', 'B', 'C', 'D']

```

- Calculer la distance entre deux sommets dans un graphe. On utilise ici la propriété que $A_{i,j}^d$ est le nombre de chemins de longueur d de s_i à s_j (propriété 1.6), et que s'il existe un chemin de s_i à s_j , alors il en existe un de longueur $\leq \# \mathcal{S}$.

```

from numpy import dot # Produit matriciel

```

```

def distance(A,sommets,s1,s2):
    i = sommets.index(s1)
    j = sommets.index(s2)
    if i==j:
        return 0
    B = A[:,:]
    for d in range(len(A)):
        if B[i,j]:
            return d+1
    B = dot(B,A)

```

```
In [7]: distance(A,S,'D','A')
Out[7]: 1
```

Cet algorithme bien qu'élégant n'est pas efficace. En utilisant un algorithme de multiplication matriciel naïf (de complexité cubique sur la dimension de l'espace), il est en $O(n^4)$ dans le pire des cas, où $n = \#\mathcal{S}$; avec des améliorations on peut atteindre $O(n^3)$ et même théoriquement $O(n^{2.5})$ (Coppersmith et Winograd); mais cela reste peu efficace. De plus l'algorithme ne fournit pas de plus court chemin.

• On peut déjà améliorer l'algorithme en remarquant qu'il suffit de calculer la seule ligne correspondant au sommet de départ dans chaque puissance de A , pour une complexité $O(n^3)$ dans le pire des cas.

```
def distance(A,sommets,s1,s2): # Calcul distance en  $O(n^3)$ 
    i = sommets.index(s1)
    j = sommets.index(s2)
    if i==j:
        return 0
    L = A[i,:] # Ligne i de A
    n = len(A)
    for d in range(n):
        if L[j]:
            return d+1
        L1 = array([0]*n)
        for k in range(n):
            for l in range(n):
                L1[k] += L[l]*A[l,k]
        L = L1 # Ligne i de  $A^{(d+2)}$ 
```

3 Parcours d'un graphe, en profondeur, en largeur

Un parcours de graphe consiste en un algorithme qui visite les sommets d'un graphe connexe selon un ordre établi.

Dans un parcours en profondeur on suit une branche, et dès qu'on est bloqué on revient au noeud précédent.

Dans un parcours en largeur on visite d'abord tous les successeurs d'un sommet, puis leur successeurs, et ainsi de suite.

Un parcours de graphe a beaucoup d'applications dans l'implémentation des graphes.

3.1 Parcours en profondeur d'un graphe

Dans cette partie, pour simplifier les graphes seront supposés non orientés. Mais le principe reste le même dans un graphe orienté.

Principe

Dans un parcours en profondeur, on dit aussi parcours en profondeur avec retour arrière, ou en anglais *backtracking*, le principe est le suivant :

- on part d'un sommet quelconque du graphe;
- on tient à jour un marquage des sommets ayant été déjà visités;
- on se déplace sur un sommet adjacent qui n'ait pas encore été visité. On le marque comme visité;
- si à partir d'un sommet tous les sommets adjacents ont déjà été visités, on reprend au sommet précédent, si c'est possible; c'est le retour arrière;
- lorsqu'on est sur le sommet initial, et que celui-ci n'a plus aucun sommet adjacent non encore visité, l'algorithme s'arrête.

Intuitivement, on suit les sommets non encore visités du graphe le long d'un chemin, et dès qu'on est bloqué on revient au sommet précédent pour poursuivre sur une autre branche. Lorsqu'on est bloqué sur le sommet initial l'algorithme s'arrête. Ce faisant on est assuré de visiter tous les sommets d'un graphe, lorsqu'il est connexe, et tous ceux d'une composante connexe, sinon.

Un parcours en profondeur peut aussi s'exécuter dans un graphe orienté, en passant d'un sommet à un sommet successeur (changer dans le code "adjacent" en "successeur"). Mais dans ce cas on ne pourra parcourir que les sommets reliés par un chemin au sommet initial.

En théorie des graphes, le parcours en profondeur a des applications importantes, comme par exemple déterminer un arbre couvrant ou déterminer la connexité, les composantes connexes d'un graphe.

Parcours à l'aide d'une pile

Un parcours en profondeur nécessite l'usage d'une pile pour permettre le retour arrière.

On empile dans la pile les sommets successivement visités, tout en tenant à jour une liste du marquage des sommets visités. Lors du retour arrière on dépile le sommet de la pile pour poursuivre au sommet précédent. L'algorithme s'arrête dès que la pile est vide.

L'architecture générale de l'algorithme de parcours en profondeur à l'aide d'une pile est la suivante. On aura besoin d'une fonction renvoyant les sommets adjacents à un sommet.

```

P ← pile vide          # Pile de la position dans le graphe
empiler le sommet initial
M ← liste de marquage    # Liste des sommets visités
marquer dans L le sommet initial
TANT QUE P est non vide :
    s ← sommet en haut de la pile
    SI s a un sommet adjacent s' non marqué dans M ALORS :
        empiler s'
        marquer s' dans la liste M
    SINON:
        dépiler s
    FIN SI
FIN TANT QUE

```

Parcours par récursivité

Un algorithme récursif utilisant une pile d'exécution dans la mémoire centrale, un parcours en profondeur peut aussi s'effectuer naturellement par récursivité. L'usage de la pile est remplacé par les appels récursifs. Il faudra encore déclarer la liste des sommets visités, et disposer d'une fonction renvoyant les sommets adjacents à un sommet, ainsi que d'une fonction pour décider si un sommet figure dans la liste des sommets visités.

L'architecture générale de l'algorithme de parcours en profondeur par récursivité est la suivante.

```

Fonction parcours(sommet s, liste M des sommets visités):
    Marquer s dans M
    POUR TOUT s' sommet adjacent à s qui n'est pas marqué dans M:
        Appeler recursive(s',M)
    FIN POUR
FIN Fonction parcours

M ← liste de marquage
appeler parcours(sommet initial, M)

```

Exemple : arbre couvrant d'un graphe

En guise d'exemple, écrivons deux algorithmes effectuant une recherche en profondeur pour déterminer un arbre couvrant à un graphe non orienté connexe passé en argument. Le premier algorithme sera itératif et utilisera une pile, le deuxième utilisera une implantation récursive.

L'algorithme (dans les deux cas) prendra en paramètre une matrice d'adjacence du

graphe, et renverra la matrice d'adjacence d'un arbre couvrant sous la forme de tableaux **numpy** bi-dimensionnels. Il faut d'abord écrire une fonction qui renvoie la liste des sommets adjacents à un sommet (autres que le sommet lui-même) sous la forme d'une liste d'entiers.

```

from numpy import array

def adjacent(Mat,s):
    L = []
    for j in range(len(Mat)):
        if Mat[s,j] and s!=j:
            L.append(j)
    return L

```

La matrice d'adjacence de l'arbre couvrant sera constituée durant le parcours en profondeur. Initialement la matrice à retourner est remplie de zéros et de mêmes dimensions que la matrice du graphe. Pour chaque nouveau sommet visité, on change deux valeurs en 1 dans la matrice, à des positions symétriques, pour ajouter à l'arbre l'arête venant d'être empruntée.

- Premier algorithme : implantation à l'aide d'une pile.

La pile est implantée à l'aide d'une liste à laquelle on n'applique que des primitives des piles (créer une pile vide, empiler, dépiler, accéder au sommet de la pile, tester si la pile est vide).

Pour une implantation plus efficace, on a stocké dans la pile un couple constitué d'un sommet et de la liste de ses sommets adjacents. À chaque déplacement sur un sommet adjacent, ce dernier est retiré de la liste d'adjacence. Cela évitera de le considérer à nouveau après retours arrières.

```

def arbreCouvrant(Mat):
    n = len(Mat)
    T = array([[0]*n for k in range(n)])
    pile = []
    pile.append((0, adjacent(Mat,0)))
    visites = [0] * n
    while pile != []:
        sommet, voisins = pile[-1]
        rechercheSuivant = True
        while rechercheSuivant and voisins != []:
            sommetSuivant = voisins.pop()
            if visite[sommetSuivant] == 0:

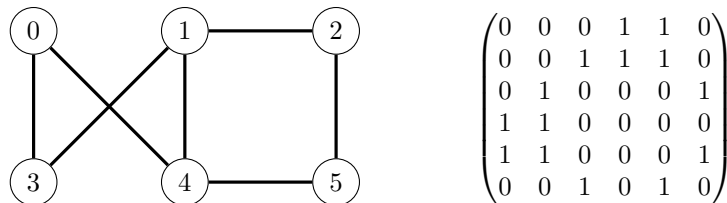
```

```

visites[sommetSuivant]= 1 # marquage
pile.append((sommetSuivant,\
              adjacent(Mat,sommetSuivant)))
T[sommet,sommetSuivant] = 1
T[sommetSuivant,sommet] = 1
rechercheSuivant = False
if rechercheSuivant:
    pile.pop()
return T

```

• **Exemple.** Appliquons l'algorithme au graphe non orienté connexe suivant, dont on a donné aussi la matrice d'adjacence :



```

In [2]: A = array([[0, 0, 0, 1, 1, 0],
                  [0, 0, 1, 1, 1, 0],
                  [0, 1, 0, 0, 0, 1],
                  [1, 1, 0, 0, 0, 0],
                  [1, 1, 0, 0, 0, 1],
                  [0, 0, 1, 0, 1, 0]])

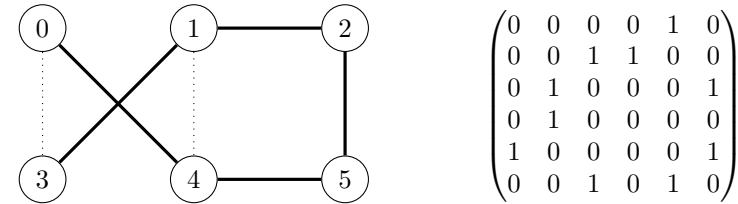
```

```

In [3]: arbreCouvrant(A)
Out[3]:
array([[0, 0, 0, 0, 1, 0],
       [0, 0, 1, 1, 0, 0],
       [0, 1, 0, 0, 0, 1],
       [0, 1, 0, 0, 0, 0],
       [1, 0, 0, 0, 0, 1],
       [0, 0, 1, 0, 1, 0]])

```

Ainsi on obtient l'arbre couvrant :



• Deuxième algorithme : implantation récursive.

```

def arbreCouvrantRec(Mat,T,sommet,visites):
    voisins = adjacent(Mat,sommet)
    for sommetSuivant in voisins:
        if visites[sommetSuivant] == 0:
            visites[sommetSuivant] = 1
            T[sommet,sommetSuivant] = 1
            T[sommetSuivant,sommet] = 1
            arbreCouvrantRec(Mat,T,sommetSuivant,visites)

```

```

def arbreCouvrant(Mat):
    n = len(M)
    T = array([[0]*n for k in range(n)])
    visites = [0] * n
    arbreCouvrantRec(Mat,T,0,visites)
    return T

```

• **Exemple.** Appliquons l'algorithme au même graphe que précédemment.

```

In [2]: A = array([[0, 0, 0, 1, 1, 0],
                  [0, 0, 1, 1, 1, 0],
                  [0, 1, 0, 0, 0, 1],
                  [1, 1, 0, 0, 0, 0],
                  [1, 1, 0, 0, 0, 1],
                  [0, 0, 1, 0, 1, 0]])

```

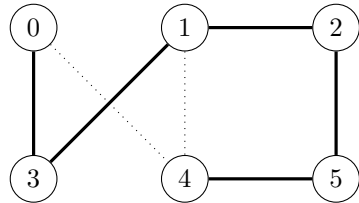
```

In [3]: arbreCouvrant(A)
Out[3]:
array([[0, 0, 0, 1, 0, 0],
       [0, 0, 1, 1, 0, 0],
       [0, 1, 0, 0, 0, 1],
       [1, 1, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 1],
       [0, 0, 0, 0, 0, 1]])

```

$[0, 0, 1, 0, 1, 0]]]$

On obtient l'arbre couvrant :



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

3.2 Parcours en largeur d'un graphe

Dans un parcours en largeur, les sommets sont parcourus de proches en proches ; à partir d'un sommet on parcourt tous ses sommets adjacents, puis leur sommets adjacents non déjà parcourus, etc... le principe est le suivant :

On considère une liste et une file d'attente :

- la liste M du marquage des sommets déjà parcourus ; elle contient autant d'élément qu'il n'y a de sommets, et permet de marquer les sommets déjà parcourus. Initialement elle ne contient que des 0 ; sauf pour le sommet s_0 , qu'on marque en 1.

- La liste F file d'attente des sommets à parcourir ; initialement elle contient le sommet initial s_0 .

On retire de la file d'attente son premier sommet s ; on ajoute à la fin de F tous les sommets successeurs de s non marqués, et on les marque dans M .

On poursuit ainsi tant que F est non vide.

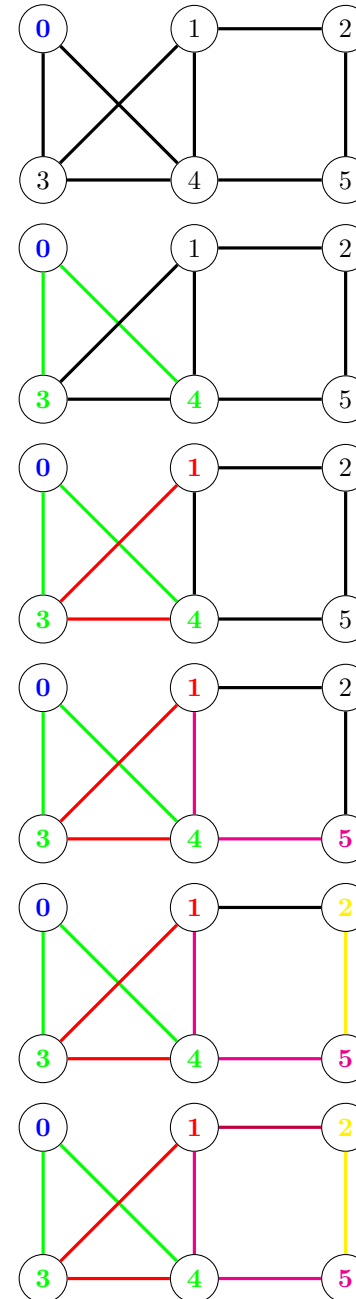
À la fin la liste M permettra de récupérer tous les sommets marqués, c'est à dire parcourus.

Dans un graphe non orienté, on obtiendra tous les sommets dans la même composante connexe que s_0 .

Dans un graphe orienté, on obtiendra tous les sommets pour lesquels il existe un chemin au départ de s_0 les reliant.

Exemple :

Au départ du sommet 0 :



$M = [1, 0, 0, 0, 0, 0]$; $F = [0]$

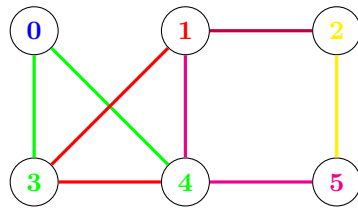
$M = [1, 0, 0, 1, 1, 0]$; $F = [3, 4]$

$M = [1, 1, 0, 1, 1, 0]$; $F = [4, 1]$

$M = [1, 1, 0, 1, 1, 1]$; $F = [1, 5]$

$M = [1, 1, 1, 1, 1, 1]$; $F = [5, 2]$

$M = [1, 1, 1, 1, 1, 1]$; $F = [2]$



$M = [1,1,1,1,1,1]$; $F = []$

L'architecture générale de l'algorithme de parcours en largeur à l'aide d'une file est la suivante. On aura besoin d'une fonction renvoyant les sommets adjacents à un sommet.

```

F ← file vide          # File de la position dans le graphe
enfiler le sommet initial
L ← liste vide         # Liste des sommets visités
insérer dans L le sommet initial et le marquer
TANT QUE F est non vide :
    retirer le premier sommet de la file et l'affecter à s
    POUR tous sommet s' successeur de s non marqué :
        enfiler s'
        et le marquer
FIN TANT QUE
Les sommets visités sont tous ceux marqués.

```

Implémentation à partir d'une liste d'adjacence

```

def parcours_largeur(Liste, s0):
    n = Liste[0]          # Nbre sommets
    Ad = Liste[1]         # Ad[i] contient les successeurs de i
    M = [0] * n           # tableau Marquage
    file = [s0] # File
    M[s0] = 1 # Marquage sommet initial
    while file != []:
        s = file.pop(0) # Retrait 1er élt file
        for v in Ad[s]: # Pour tous ses successeurs
            if M[v] == 0: # si non marqué
                file.append(v) # ajout en fin de file
                M[v] = 1 # et marquage
    parcourus = [] # création liste sommets parcourus
    for i in range(n):
        if M[i] == 1:
            parcourus.append(i)
    return parcourus

```

Cette implémentation est améliorable : utiliser une liste pour la file rend coûteux le retrait du premier élément de la file. Pour cela on peut utiliser le module `collections` pour utiliser des objets `deque` : ajout et retrait en début et en fin se font en $O(1)$:

```

>>> from collections import deque
>>> d = deque()
>>> d.append(1)
>>> d.append(2)
>>> d
deque([1, 2])
>>> d.popleft()
1

```

Un objet `deque` généralise les piles et les files ; il admet les méthodes (primitives en $O(1)$) `append`, `appendleft`, `pop`, `popleft`. La fonction `len` renvoie le nombre d'éléments.

L'algorithme optimisé grâce à l'usage d'une file devient :

```

from collections import deque

def parcours_largeur(Liste, s0):
    n = Liste[0]          # Nbre sommets
    Ad = Liste[1]         # Ad[i] contient les successeurs de i
    M = [0] * n           # tableau Marquage
    file = deque([s0]) # File
    M[s0] = 1 # Marquage sommet initial
    while len(file) != 0:
        s = file.popleft() # Retrait 1er élt file
        for v in Ad[s]: # Pour tous ses successeurs
            if M[v] == 0: # si non marqué
                file.append(v) # ajout en fin de file
                M[v] = 1 # et marquage
    parcourus = [] # création liste sommets parcourus
    for i in range(n):
        if M[i] == 1:
            parcourus.append(i)
    return parcourus

```

4 Graphe valué

Définition

Un graphe est valué lorsque chaque arête est munie d'une valeur, appelée son poids.

Définition 4.1

Un **graphe valué** est la donnée d'un graphe $\mathcal{G} = (\mathcal{S}, \mathcal{A})$ et d'une application $p : \mathcal{A} \rightarrow \mathbb{R}$ appelée valuation.

Pour chaque arête $(s, s') \in \mathcal{A}$, le nombre $p(s, s')$ est appelé son poids.

Une valuation p est dite **strictement positive** si $\forall (s, s') \in \mathcal{A}, p(s, s') > 0$.

La valuation d'un graphe s'étend en une application : $\mathcal{S} \times \mathcal{S} \rightarrow \overline{\mathbb{R}}$ en posant $p(s, s') = +\infty$ lorsque $(s, s') \notin \mathcal{A}$.

Matrice de valuation

On peut décrire un graphe valué à l'aide de sa matrice de valuation.

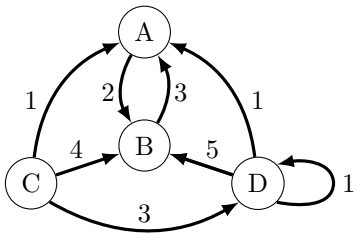
Définition 4.2

Donnés un graphe $\mathcal{G} = (\mathcal{S}, \mathcal{A})$ ayant $n = \#\mathcal{S}$ sommets, une valuation p du graphe et une bijection $s : \llbracket 1, n \rrbracket \rightarrow \mathcal{S}$, la **matrice de valuation** est la matrice $M \in \mathcal{M}_n(\mathbb{R})$ définie par :

$$\forall (i, j) \in \llbracket 1, n \rrbracket^2, M_{i,j} = \begin{cases} p(s(i), s(j)) & \text{si } (s(i), s(j)) \in \mathcal{A} \\ \infty & \text{sinon} \end{cases}$$

C'est une adaptation de la notion de matrice d'adjacence aux graphes valués, en changeant 1 par le poids d'une arête et 0 par $+\infty$.

• **Exemple.** Un graphe valué et sa matrice de valuation :



	A	B	C	D
A	∞	2	∞	∞
B	3	∞	∞	∞
C	1	4	∞	3
D	1	5	∞	1

Longueur d'un chemin. Chemin géodésique

Définition 4.3

Dans un graphe $\mathcal{G} = (\mathcal{S}, \mathcal{A})$ à valuation strictement positive, la **longueur d'un chemin** $\gamma = (s_0, s_1, \dots, s_n)$ est le réel :

$$\ell(\gamma) = \sum_{k=0}^{n-1} p(s_k, s_{k+1}) .$$

Un chemin de s à s' de longueur minimale est un **chemin géodésique** ou **plus court chemin** de s à s' .

Si de plus le graphe est non orienté et sa valuation symétrique (c'est-à-dire $p(s, s') = p(s', s)$), la longueur minimale d'un chemin (géodésique) de s à s' , si elle existe, est la **distance** $d(s, s')$ entre les sommets s, s' .

Si le graphe est de plus connexe, l'application d ainsi définie est une distance sur $\mathcal{S}$, c'est-à-dire :

- Séparation. $\forall s, s' \in \mathcal{S}, d(s, s') = 0 \iff s = s'$.
- Symétrie. $\forall s, s' \in \mathcal{S}, d(s, s') = d(s', s)$.
- Inégalité triangulaire. $\forall s, s', s'' \in \mathcal{S}, d(s, s'') \leq d(s, s') + d(s', s'')$.

4.1 Algorithme de Dijkstra

Dans un graphe valué à valuation strictement positive, l'algorithme de Dijkstra est un algorithme efficace pour déterminer la longueur d'un plus court chemin d'un sommet s à un sommet s' quelconques, ainsi qu'une géodésique de s à s' . Il ne s'applique pas lorsque la valuation n'est pas strictement positive.

Cet algorithme est largement employé. Il a pour application directe tous les moteurs de recherche des logiciels de navigation (mappy, google maps, waze, logiciels embarqués dans les GPS). Dans ces systèmes, le réseau routier est implanté sous forme d'un graphe, et selon les valuations employées, l'algorithme de recherche de géodésique permet de déterminer :

- le plus court trajet pour se rendre par la route d'un lieu A à un lieu B ;
- le trajet le plus rapide pour se rendre par la route d'un lieu A à un lieu B ;
- le trajet le plus économe (carburant et péages) pour se rendre par la route d'un lieu A à un lieu B , etc.

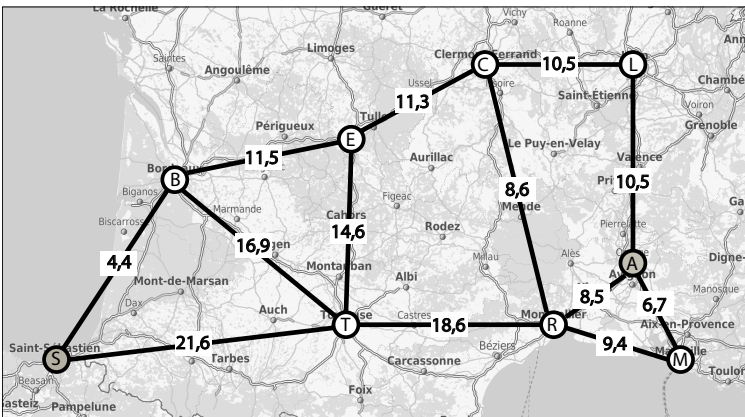
L'algorithme de Dijkstra s'utilise notamment dans certains protocoles efficaces de routage internet.

Illustration sur un exemple

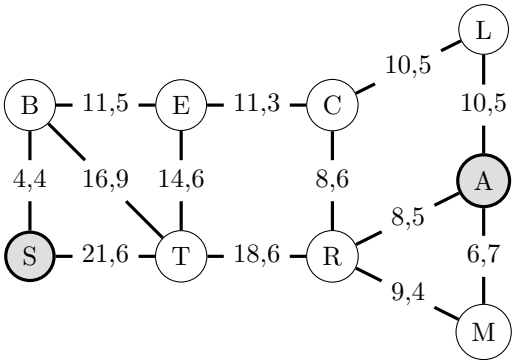
Décrivons le fonctionnement de l’algorithme sur un exemple. On souhaite effectuer un trajet d’Avignon à la ville de Saint-Sébastien (à la frontière espagnole) par le réseau autoroutier du sud de la France.



Connaissant le coût (en péage et carburant) pour chaque tronçon autoroutier, l’algorithme permet de déterminer le parcours autoroutier dont le tarif est le plus économique pour relier Avignon à Saint-Sébastien.



On code ces informations dans un graphe valué non orienté : ses sommets représentent les échangeurs (aux grandes villes Avignon, Marseille, montpellierR, Lyon, Clermont-ferrand, tulleE, Bordeaux, Toulouse, Saint-sébastien), et ses arêtes les tronçons autoroutiers les reliant. Les valuations des arêtes sont les coûts des différents tronçons exprimés en €, pour le véhicule utilisé et pour une consommation moyenne.



On consigne toutes ces informations dans la **matrice de valuation du graphe** :

	M	A	L	R	C	E	T	B	S
M	∞	6,7	∞	9,4	∞	∞	∞	∞	∞
A	6,7	∞	10,5	8,5	∞	∞	∞	∞	∞
L	∞	10,5	∞	∞	10,5	∞	∞	∞	∞
R	9,4	8,5	∞	∞	8,6	∞	18,6	∞	∞
C	∞	∞	10,5	8,6	∞	11,3	∞	∞	∞
E	∞	∞	∞	∞	11,3	∞	14,6	11,5	∞
T	∞	∞	∞	18,6	∞	14,6	∞	16,9	21,6
B	∞	∞	∞	∞	∞	11,5	16,9	∞	4,4
S	∞	∞	∞	∞	∞	∞	21,6	4,4	∞

C’est une matrice symétrique ; on a compté les mêmes coûts sur chaque tronçon dans les deux directions. Les arêtes absentes sont représentées par un poids ∞ .

Principe de l’algorithme de Dijkstra

Dès que tous les poids sont strictement positifs, l’algorithme permet de déterminer un chemin géodésique d’un sommet à un autre sommet.

Algorithme de Dijkstra.

- Données :
 - le graphe valué, donné par exemple par sa matrice de valuation, $p(sa, sb)$ représente le poids entre les sommets sa et sb ;
 - le sommet de départ $s0$;
 - (éventuellement) le sommet de destination sd .

- Retourne :
 - les plus courtes distances dans le graphe du sommet **s0** à chacun des autres sommets ;
 - éventuellement, des chemins géodésiques de **s0** à chacun des autres sommets ;
 - éventuellement, on peut arrêter la recherche dès que la plus courte distance de **s0** à **sd** a été trouvée.
- Implantation :
 - un tableau **D** contient les distances **Dist(s0,sk)** de **s0** à chaque autre sommet **sk**. Initialement toutes les valeurs dans **D** sont ∞ ;
 - l'ensemble des sommets est partitionné en deux sous-ensembles : l'ensemble des sommets marqués et l'ensemble des sommets non marqués. Initialement tous les sommets sont non marqués.

• Algorithme :

```
Dist(s0,s0) = 0.  
Tant qu'il reste des sommets non marqués :  
    Choisir un sommet s non marqué avec Dist(s0,s) minimal  
    Marquer s  
    Pour tout sommet t non marqué :  
        Si Dist(s0,s) + p(s,t) < Dist(s0,t):  
            Dist(s0,t) = Dist(s0,s) + p(s,t)
```

- À la fin de l'algorithme, le tableau **Dist** contient les longueurs de chemins géodésiques de **s0** à chaque autre sommet. En fait dès que le sommet **sk** est marqué, **Dist(s0,sk)** ne change plus et contient la longueur géodésique de **s0** à **sk**.
- Pour déterminer les chemins géodésiques il suffit de noter pour chaque sommet **s**, sa provenance : c'est le sommet marqué **t** qui a permis la dernière mise à jour de **Dist(s0,s)** à partir de **Dist(s0,t)+p(t,s)**. Au départ d'un sommet destination **sk**, en remontant ainsi jusqu'à **s0** " en marche arrière " on construit un chemin géodésique de **s0** à **sk**. Ce procédé peut s'appliquer au départ d'un sommet destination **sk** dès que le sommet **sk** est marqué ; il donnera un chemin géodésique de **s0** à **sk**.

Résolution

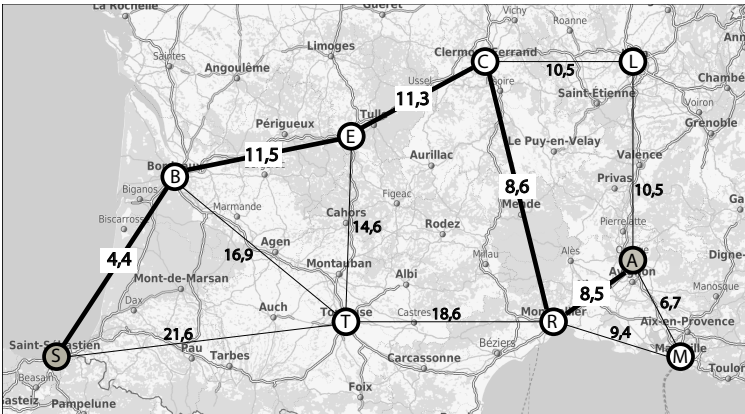
L'exécution de l'algorithme peut s'effectuer sur un tableau. Appliquons-le à la recherche du trajet d'Avignon à Saint-Sébastien.

	A	L	M	R	C	E	T	B	S	
<u>0</u>	∞	∞	∞	∞	∞	∞	∞	∞	∞	Tableau initial
<u>0</u>	10,5	6,7	8,5	∞	∞	∞	∞	∞	∞	Successeurs de A
<u>0</u>	10,5	<u>6,7</u>	8,5	∞	∞	∞	∞	∞	∞	Marquer M
<u>0</u>	10,5	<u>6,7</u>	8,5	∞	∞	∞	∞	∞	∞	Successeurs de M
<u>0</u>	10,5	<u>6,7</u>	8,5	∞	∞	∞	∞	∞	∞	Marquer R
<u>0</u>	10,5	<u>6,7</u>	8,5	17,1	∞	27,1	∞	∞	∞	Successeurs de R
<u>0</u>	<u>10,5</u>	<u>6,7</u>	<u>8,5</u>	17,1	∞	27,1	∞	∞	∞	Marquer L
<u>0</u>	<u>10,5</u>	<u>6,7</u>	<u>8,5</u>	17,1	∞	27,1	∞	∞	∞	Successeurs de L
<u>0</u>	<u>10,5</u>	<u>6,7</u>	<u>8,5</u>	17,1	∞	27,1	∞	∞	∞	Marquer C
<u>0</u>	<u>10,5</u>	<u>6,7</u>	<u>8,5</u>	17,1	28,4	27,1	∞	∞	∞	Successeurs de C
<u>0</u>	<u>10,5</u>	<u>6,7</u>	<u>8,5</u>	<u>17,1</u>	28,4	27,1	∞	∞	∞	Marquer T
<u>0</u>	<u>10,5</u>	<u>6,7</u>	<u>8,5</u>	<u>17,1</u>	28,4	<u>27,1</u>	43,9	48,7	48,7	Successeurs de T
<u>0</u>	<u>10,5</u>	<u>6,7</u>	<u>8,5</u>	<u>17,1</u>	28,4	<u>27,1</u>	43,9	48,7	48,7	Marquer E
<u>0</u>	<u>10,5</u>	<u>6,7</u>	<u>8,5</u>	<u>17,1</u>	28,4	<u>27,1</u>	39,9	48,7	48,7	Successeurs de E
<u>0</u>	<u>10,5</u>	<u>6,7</u>	<u>8,5</u>	<u>17,1</u>	<u>28,4</u>	<u>27,1</u>	39,9	48,7	48,7	Marquer B
<u>0</u>	<u>10,5</u>	<u>6,7</u>	<u>8,5</u>	<u>17,1</u>	<u>28,4</u>	<u>27,1</u>	39,9	44,3	48,7	Successeurs de B
<u>0</u>	<u>10,5</u>	<u>6,7</u>	<u>8,5</u>	<u>17,1</u>	<u>28,4</u>	<u>27,1</u>	<u>39,9</u>	44,3	44,3	Marquer S. Fin

Chemin géodésique obtenu, d'Avignon à Saint-Sébastien :

A - R - C - E - B - S coût = 44,3

Avignon - Montpellier - Clermont-Ferrand - Tulle - Bordeaux - Saint-Sébastien



4.2 Code Python

Commençons par décrire l'implantation pour déterminer toutes les longueurs géodésiques d'un sommet s_0 aux autres sommets. Le programme affichera en sortie toutes ces distances, ainsi que, si un paramètre `geodesique` vaut `True`, une géodésique.

Pour l'implantation, le graphe valué est donné par sa matrice de valuation, à l'aide d'un tableau numpy bidimensionnel.

- Dans la matrice de valuation on utilise la constante `inf` de numpy (+INF de la norme IEEE-754 sur l'écriture flottante) pour représenter ∞ .
- Les n sommets sont représentés par leur indice dans la matrice de valuation, c'est-à-dire par un entier dans $\llbracket 0, n-1 \rrbracket$. On pourra passer en paramètre une liste du noms des sommets.
- Un tableau `Dist` contient initialement n valeurs égales à `numpy.inf`. À la fin de l'algorithme il contient à chaque indice i la longueur géodésique de s_0 au sommet s_i .
- On constitue deux listes `Marques` et `NonMarques` contenant les indices des sommets marqués et non marqués. Initialement `Marques` est vide, `NonMarques` est plein.
- Un tableau `Provenance` mémorise pour chaque sommet s_i , le dernier sommet marqué ayant permis la mise à jour de `Dist[i]`. Il n'est utilisé que pour déterminer le chemin géodésique.
- L'algorithme s'exécute au sein d'une boucle `while` tant que tous les sommets ne sont pas marqués. Chaque passage dans la boucle modifie les tableaux `Dist` et les listes `Marques` et `NonMarques`.
- Une fonction `plusCourt(D,I)` pour un tableau de nombres `D`, et un tableau `I` d'indices du tableau `D`, qui renvoie l'indice de l'élément minimal parmi les éléments de `D` à indice dans `I`.
- **Implantation.** Pour déterminer toutes longueurs géodésiques issues d'un sommet s_0 .

```
import numpy as np

oo = np.inf      #L'infini. Pour un réel a : a<oo et a+oo = oo

def plusCourt(D,I): # Renvoie l'indice de min D(i), i dans I
    minimum = oo
    indice = 0
    for i in I:
        if D[i] < minimum:
            indice = i
            minimum = D[i]
    return indice
```

Algorithme de Dijkstra

```
def Dijkstra(Adj,s0,sommets=None):
```

```
    n = len(Adj)
```

```
    if sommets == None :
```

```
        sommets = [k for k in range(n)]
```

```
    NonMarques = [k for k in range(n)]
```

```
    Marques = [ ]
```

```
    Dist = oo * np.ones(n)
```

```
    Dist[s0] = 0
```

```
    Provenance = [None for k in range(n)]
```

```
    while len(Marques) < n:
```

```
        i = plusCourt(Dist,NonMarques)
```

```
        Marques.append(i)
```

```
        NonMarques.remove(i)
```

```
        for k in NonMarques:
```

```
            if Dist[i] + Adj[i,k] < Dist[k]:
```

```
                Dist[k] = Dist[i] + Adj[i,k]
```

```
                Provenance[k] = i
```

Affichage des géodésiques :

```
    for k in range(n):
```

```
        print("Distance de",sommets[s0],"à",sommets[k],":",\
              Dist[k],end='')

```

```
        chemin = sommets[k]
```

```
        j = k
```

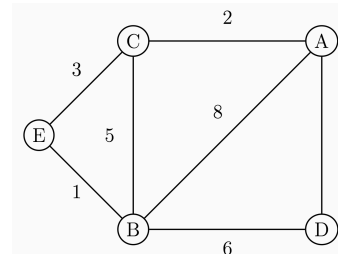
```
        while j != s0:
```

```
            j = Provenance[j]
```

```
            chemin = sommets[j] + '-' + chemin
```

```
        print(":",chemin)
```

- **Exemple.** Exécutons l'algorithme sur l'exemple suivant :



	A	B	C	D	E
A	0	8	2	1	∞
B	8	0	5	6	1
C	2	5	0	∞	3
D	1	6	∞	0	∞
E	∞	1	3	∞	0

```
sommets = ['A','B','C','D','E'] # Nom des sommets
```

```
# Matrice d'adjacence du graphe :
```

```
Adj = np.array([[oo, 8, 2, 1, oo],
                [8, oo, 5, 6, 1],
                [2, 5, oo, oo, 3],
                [1, 6, oo, oo, oo],
                [oo,1, 3, oo, oo]])
```

- Appel avec pour sommet de départ, le sommet *A* (indice 0), et sans affichage des géodésiques.

```
In [2]: Dijkstra(Adj,0,False,sommets)
```

```
Distance de A a A : 0.0
Distance de A a B : 6.0
Distance de A a C : 2.0
Distance de A a D : 1.0
Distance de A a E : 5.0
```

- Avec affichage des géodésiques.

```
In [3]: Dijkstra(Adj,0,True,sommets)
```

```
Distance de A a A : 0.0: A
Distance de A a B : 6.0: A-C-E-B
Distance de A a C : 2.0: A-C
Distance de A a D : 1.0: A-D
Distance de A a E : 5.0: A-C-E
```

- Implantation pour déterminer une géodésique entre deux sommets passés en paramètre. La fonction renvoie le couple de la longueur géodésique et d'un chemin géodésique.

```
# Algorithme de Dijkstra : géodésique entre 2 sommets
```

```
def Dijkstra(Adj,s0,s1):
    n = len(Adj)
    NonMarques = [k for k in range(n)]
    Marques = [ ]
    Dist = oo * np.ones(n)
    Dist[s0] = 0
    Provenance = [None for k in range(n)]
    i = -1
    while s1 != i:
```

```
        i = plusCourt(Dist,NonMarques)
        Marques.append(i)
        NonMarques.remove(i)
        for k in NonMarques:
            if Dist[i] + Adj[i,k] < Dist[k]:
                Dist[k] = Dist[i] + Adj[i,k]
                Provenance[k] = i
    # Détermination des géodésiques
    G = [s1]
    j = s1
    while j != s0:
        j = Provenance[j]
        G.append(j)
    return Dist, G[::-1]
```

- Exemple.

```
s0, s1 = 0, 1
dist, chemin = Dijkstra(Adj,s0,s1)
print("Distance de",sommets[s0],"a",sommets[s1],':',dist[s1])
print("Geodesique :",[sommets[s] for s in chemin])
```

Ce qui produit.

```
Distance de A a B : 6.0
Geodesique : ['A', 'C', 'E', 'B']
```

4.3 Terminaison, correction et complexité

- **Terminaison.** Le nombre de sommets non marqués est un variant de la boucle **while** du programme. D'où la terminaison de l'algorithme.

- **Correction de l'algorithme :** on la prouve en montrant qu'on a l'invariant de boucle :

"Pour tout sommet s marqué, $Dist(s_0, s)$ contient la longueur géodésique de s_0 à s ."

Pour tout sommet $s \in \mathcal{S}$, notons $d(s_0, s)$ la longueur géodésique de s_0 à s . On commence par renuméroter les sommets pour les ordonner par valeur de $d(s_0, s)$ croissante :

$$\mathcal{S} = \{s_0, s_1, \dots, s_n\} \quad \text{avec} \quad d(s_0, s_0) \leq d(s_0, s_1) \leq \dots \leq d(s_0, s_n).$$

On note p la valuation du graphe. Durant le déroulement de l'algorithme $M \subset \mathcal{S}$ est l'ensemble des sommets marqués, $Dist(s_0, s)$ est la valeur dans le tableau pour le sommet s . On montre que :

– $M = \{s_0, s_1, \dots, s_{i-1}\}$;

– pour tout sommet marqué s_j , $Dist(s_0, s_j) = d(s_0, s_j)$.

Par récurrence sur le nombre d'itérations i de la boucle.

Initialisation. Si $i = 1$, le sommet marqué est le sommet de départ s_0 . Or $Dist(s_0, s_0)$ contient 0, qui est égal à la longueur géodésique $d(s_0, s_0)$.

Hérédité. Supposons la proposition vraie après la i -ème itération de boucle, et montrons qu'elle le reste après l'itération suivante $i + 1$.

Considérons une géodésique de s_0 à s_i , et notons s_k le prédécesseur de s_i pour cette géodésique. Alors $d(s_0, s_i) = d(s_0, s_k) + p(s_k, s_i)$; en particulier, puisque $p(s_k, s_i) > 0$, on a $d(s_0, s_k) < d(s_0, s_i)$ et donc par hypothèse de récurrence $k < i$: autrement dit $s_k \in M$ a été marqué lors d'une étape précédente.

Le même principe s'appliquant pour tout sommet s_j avec $j < k$, nécessairement s_k a été marqué à la $k + 1$ -ème itération. À l'issue de cette $k + 1$ -ème itération, on a

$$Dist(s_0, s_i) = Dist(s_0, s_k) + p(s_k, s_i) = d(s_0, s_k) + p(s_k, s_i) = d(s_0, s_i) .$$

En début de la $i + 1$ -ème itération $Dist(s_0, s_i) = d(s_0, s_i)$ vérifie pour tous sommet non-marqué s_j , avec $i < j$,

$$Dist(s_0, s_i) \leq d(s_0, s_i) \leq d(s_0, s_j) \leq Dist(s_0, s_j) .$$

Le sommet s_i est donc celui qui est marqué à la $i + 1$ -ème itération. Il vérifie $Dist(s_0, s_i) = d(s_0, s_i)$. La proposition demeure vraie au rang $i + 1$. Ceci conclut la récurrence et la preuve de l'invariant de boucle.

En particulier en fin d'algorithme le tableau $Dist(s_0, \cdot)$ contient les longueurs géodésiques de s_0 à chaque sommet (marqué).

De plus, pour le sommet marqué s_i , notons $\bar{s}_{i-1}$ le sommet marqué qui a permis la dernière mise à jour $Dist(s_0, s_i) = Dist(s_0, \bar{s}_{i-1}) + p(\bar{s}_{i-1}, s_i)$. On a aussi $d(s_0, s_i) = d(s_0, \bar{s}_{i-1}) + p(\bar{s}_{i-1}, s_i)$ d'après ce qui précède. Ainsi le chemin $s_0 = \bar{s}_p, \bar{s}_{p+1}, \dots, \bar{s}_{i-1}, s_i$ est un chemin géodésique de s_0 à s_i . Or c'est le chemin retourné par l'algorithme.

• **Complexité** : Chaque passage dans la boucle nécessite $O(\#\mathcal{S})$ opérations élémentaires, et il y a au plus $\#\mathcal{S}$ passages dans la boucle. En notant $S = \#\mathcal{S}$ le nombre de sommets, la complexité est donc dans le pire des cas de $O(\#\mathcal{S}^2)$ pour une implantation utilisant la matrice de valuation.

• En se donnant une liste d'adjacence (en plus de la matrice d'adjacence) l'algorithme est plus efficace.

Algorithme de Dijkstra

def Dijkstra(L_Adj, s0, sommets=None):

if sommets == None:

 sommets = [k **for** k **in** range(n)]

 n = L_Adj[0] # Nbre sommets

 V = L_Adj[1] # V[i] est la liste des successeurs de i

 Marques = [0] * n

 Dist = oo * np.ones(n)

 Dist[s0] = 0

 Marques[s0] = 1

 Provenance = [None **for** k **in** range(n)]

while len(Marques) < n:

 i = plusCourt(Dist, NonMarques)

 Marques[i] = 1

for k **in** V[i]:

if Marques[k] == 0:

if Dist[i] + Adj[i,k] < Dist[k]:

 Dist[k] = Dist[i] + Adj[i,k]

 Provenance[k] = i

On peut alors améliorer la complexité : en améliorant la complexité de la recherche de l'élément non marqués à plus petite distance ; en $O(\#\mathcal{S} + \#\mathcal{A} \times \log(\#\mathcal{S}))$. (Nous ne le ferons pas) : l'algorithme de Dijkstra effectue une recherche en largeur avec une "file de priorité". En améliorant la file de priorité on améliore sa complexité.

4.4 L'algorithme A^*

Supposons que l'on cherche à déterminer le plus court chemin entre deux sommets d'un graphe, représentant des points d'un espace métrique, disons du plan euclidien.

L'algorithme de Dijkstra a alors pour défaut d'effectuer un parcours en largeur du graphe, et de rechercher une géodésique dans toutes les directions, alors que si le sommet à atteindre se trouve au nord, il ne sera pas utile de chercher trop longtemps vers le sud.

L'algorithme A^* l'améliore sur ce point en utilisant une heuristique : une estimation grossière de la distance du sommet à la destination. Comparons les deux algorithmes :

Algorithme de Dijkstra
Entrée : Un graphe $(\mathcal{S}, \mathcal{A})$ valué de poids ≥ 0 .
deux sommets s et d

Sortie : géodésique de s à d .
Pour u de 1 à $\#\mathcal{S}$:
 $D(u) \leftarrow +\infty$
 $D(s) = 0$
Tant que d est non marqué:
 $u \leftarrow$ sommet non marqué avec $D(u)$ minimal
Marquer u
Pour tout successeur v de u :
Si $D(u) + d(u,v) < D(v)$:
 $D(v) \leftarrow D(u) + d(u,v)$

Algorithme A^*
Entrée : Un graphe $(\mathcal{S}, \mathcal{A})$ valué de poids ≥ 0 .
deux sommets s et d
une heuristique H admissible

Sortie : géodésique de s à d .
Pour u de 1 à $\#\mathcal{S}$:
 $D(u) \leftarrow +\infty$
 $D(s) = 0$
Tant que d est non marqué:
 $u \leftarrow$ sommet non marqué avec $D(u) + H(u)$ minimal
Marquer u
Pour tout successeur v de u :
Si $D(u) + d(u,v) < D(v)$:
 $D(v) \leftarrow D(u) + d(u,v)$

– Une heuristique est dite admissible si elle ne surestime jamais la distance qu'il reste à parcourir : $H(u) \leq d(u,d)$.

– En choisissant $H(u)=0$ c'est l'algorithme de Dijkstra.

– Dans le cas de la distance euclidienne, on peut prendre :

$$H(u) = \sqrt{(x_u - x_d)^2 + (y_u - y_d)^2}$$

C'est une heuristique admissible.

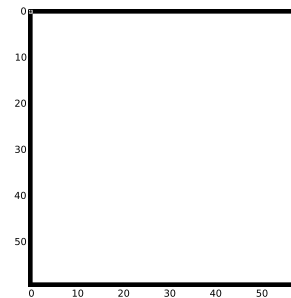
Propriété 4.1

- Pour une heuristique admissible, l'algorithme A^* renvoie bien une géodésique.
- Si H_1 et H_2 sont deux heuristiques admissibles, et si $H_1 \leq H_2$, alors avec l'heuristique H_2 on aboutira plus rapidement qu'avec H_1 .
- Si H n'est pas admissible, la solution renvoyée ne sera pas géodésique. Si M est le surcoût maximal de H ($\forall s \in \mathcal{S}, H(s) \leq d(s,d) + M$) le surcoût de la solution renvoyée sera au plus M .

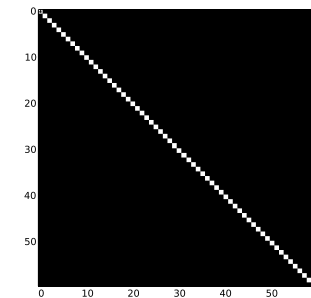
Preuve. Elle procède de la même façon que celle de Dijkstra. Nous ne la détaillerons pas. $\square$

L'efficacité de l'algorithme dépend de l'heuristique. Pour un choix convenable il s'avère bien plus rapide que Dijkstra ; au pire ($H = 0$) on retrouve Dijkstra ; attention avec H non admissible la solution renvoyée n'est pas exacte. Il est très utilisé en IA.

Exemple. On compare les deux algorithmes : on considère le graphe dont les sommets sont tous les points du plan de coordonnées dans $\llbracket 0, 60 \rrbracket^2$, les arêtes toutes celles entre sommets voisins, horizontales $(i, j) \rightarrow (i \pm 1, j)$, verticales $(i, j) \rightarrow (i, j \pm 1)$, et diagonales $(i, j) \rightarrow (i \pm 1, j \pm 1)$ valuées par leur distance euclidienne. On confronte les deux algorithmes pour chercher une géodésique entre les sommets $(0, 0)$ et $(58, 58)$: on a pris pour heuristique $H(u)$: distance euclidienne de u à la destination. On a tracé en blanc tous les sommets qui ont été marqués durant la recherche, et on a mesuré le temps d'exécution :



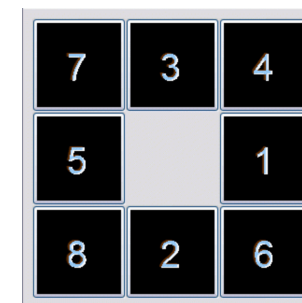
Dijkstra
Durée 9,5s



Algorithme A^*
Durée 0,7s

On donne ci-après le code utilisé.

Exemple d'utilisation : dans le jeu du taquin ; déplacer les cases jusqu'à les mettre dans l'ordre croissant.



Il s'agit de trouver un chemin dans un graphe dont les sommets sont les différentes configurations, de la configuration donnée, à la configuration souhaitée. Les arêtes relient les sommets obtenus par un déplacement. Chercher une géodésique c'est chercher la résolution la meilleure.

On peut appliquer l'algorithme A^* avec pour heuristique :

$$H(s) = \sum_{i=1}^8 \text{distance du } i \text{ à sa position finale}$$

C'est bien une heuristique admissible. L'implémentation sera réalisée en TP.

```
from numpy import array, inf

N = 60
p = 58

# Sommets du graphe
S = array([[i,j) for i in range(N)] for j in range(N)])

# Renvoie les sommets adjacents
def voisins(i,j):
    V = []
    for dx in (-1,0,1):
        if 0<i+dx<N:
            for dy in (-1,0,1):
                if 0<j+dy<N:
                    V.append((i+dx,j+dy))

    return V

# Distance entre 2 sommets = distance euclidienne
def dist(s1,s2):
    return ((s1[0]-s2[0])**2+(s1[1]-s2[1])**2)**.5

# Algorithme A*
# Heuristique
def H(s1,s2):
    return dist(s1,s2)

def minimalA(D,M,d):
    m = inf
    im, jm = -1, -1
    for i in range(N):
        for j in range(N):
            if M[i,j] == 0 :
                if D[i,j] + H((i,j),d) < m:
```

```
                m = D[i,j] + H((i,j),d)
                im, jm = i, j
    return im, jm

def algorithmeA(s,d):
    M = array([[0 for i in range(N)] for j in range(N)])
    D = array([[inf for i in range(N)] for j in range(N)])
    D[s[0],s[1]] = 0
    while M[d[0],d[1]] == 0:
        u = minimalA(D,M,d)
        M[u[0],u[1]] = 1
        V = voisins(u[0],u[1])
        for v in V:
            if D[u[0],u[1]] + dist(u,v) < D[v[0],v[1]]:
                D[v[0],v[1]] = D[u[0],u[1]] + 1
    return M

# Algorithme Dijkstra
def minimal(D,M):
    m = inf
    im, jm = -1, -1
    for i in range(N):
        for j in range(N):
            if M[i,j] == 0 :
                if D[i,j] < m:
                    m = D[i,j]
                    im, jm = i, j
    return im, jm

def dijkstra(s,d):
    M = array([[0 for i in range(N)] for j in range(N)])
    D = array([[inf for i in range(N)] for j in range(N)])
    D[s[0],s[1]] = 0
    while M[d[0],d[1]] == 0:
        u = minimal(D,M)
        M[u[0],u[1]] = 1
        V = voisins(u[0],u[1])
        for v in V:
            if D[u[0],u[1]] + dist(u,v) < D[v[0],v[1]]:
                D[v[0],v[1]] = D[u[0],u[1]] + 1
```

```
    return M

# Tracé et mesure
from matplotlib.pyplot import imshow, show

from time import time
t0 = time()
M1 = algorithmeA((0,0),(p,p))
t1 = time()
M2 = dijkstra((0,0),(p,p))
t2 = time()
print('Algorithme A* ; duree : ', t1-t0, 'sec')
print('Dijkstra ; duree : ', t2-t1, 'sec')
imshow(M1,cmap='gray', vmin=0, vmax=1, interpolation='nearest')
imshow(M2,cmap='gray', vmin=0, vmax=1, interpolation='nearest')
show()
```