Chapitre 3 : Les graphes

Thomas MEGARBANE

PCSI

Plan

Plan

Pré-requis ensemblistes

Définition

Étant donné un ensemble E et $k \in \mathbb{N}^*$, on définit :

- $\mathcal{P}(E)$ *: l'ensemble des parties de* E *:* $\mathcal{P}(E) = \{F \mid F \subset E\}$
- $\mathcal{P}_k(E)$ *: l'ensemble des* k*-parties de* E *:* $\mathcal{P}_k(E) = \{F \in \mathcal{P}(E) \mid \text{card}(F) = k\}$
- E^k *: l'ensemble des* k*-uplets d'éléments de* E *:* $E^k = \{(a_1, \ldots, a_k) \mid a_1, \ldots, a_k \in E\}$ *De plus, si* $i \in \{1, \ldots, k\}$ *et* $a = (a_1, \ldots, a_k) \in E^k$*, on dira que* a_i *est la* i*-ème* ***coordonnée*** *de* a*.*

Proposition

Si E *est un ensemble fini de cardinal* n *et* $k \in \mathbb{N}^*$*, alors les ensembles* $\mathcal{P}(E)$*,* $\mathcal{P}_k(E)$ *et* E^k *sont finis, avec :*

$$\text{card}(\mathcal{P}(E)) = 2^n, \ \text{card}(\text{P}_k(E)) = \binom{n}{k} \text{ et } \text{card}(E^k) = n^k.$$

Pré-requis ensemblistes

Remarque

*Si on se donne $a_1, \ldots, a_k \in E$ **deux-à-deux distincts** alors on peut leur associer :*

- *la k-partie : $\{a_1, \ldots, a_k\}$*
- *le k-uplet : $(a_1, \ldots, a_k)$.*

La différence est que l'ordre a une importance pour les k-uplets, mais pas pour les k-parties. Ainsi :

- *avec les mêmes éléments : on ne peut définir qu'une seule k-partie, mais $k!$ k-uplets (en permutant les coordonnées)*
- *si certains des a_i sont égaux : on ne peut plus définir de k-partie, mais on peut toujours définir des k-uplets*

*En particulier, il ne faut pas confondre la **paire** $\{a, b\}$ et le **couple** (a, b).*

Exemple

Si $E = \{a, b, c\}$ alors :

- *il y a 3 paires d'éléments de E : $\mathcal{P}_2(E) = \{\{a, b\}, \{a, c\}, \{b, c\}\}$*
- *il y a 9 couples d'éléments de E :*
  $E^2 = \{(a, b), (b, a), (a, c), (c, a), (b, c), (c, b), (a, a), (b, b), (c, c)\}$.

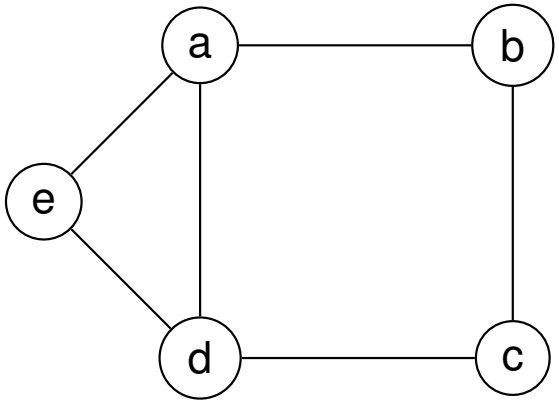
Graphes non-orientés

Définition (Graphe non-orienté)

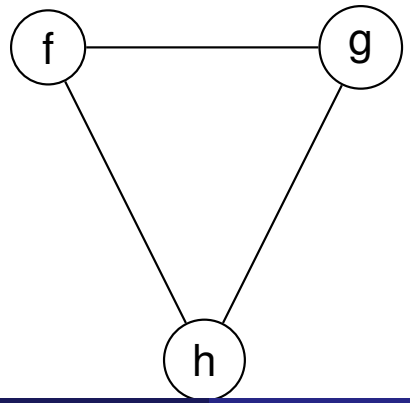
*Un **graphe non-orienté** est la donnée d'un couple de la forme $G = (S, A)$, où :*

- *S est un ensemble non vide, dont les éléments sont les **sommets** (ou **nœuds**) ;*
- *A est un ensemble de **paires** ou de **singletons** d'éléments de S, dont les éléments sont les **arêtes***

*Les éléments d'une arêtes sont ses **extrémités** et on dira qu'une arête ne possédant qu'une extrémité est une **boucle**.*



$$S = \{a, b, c, d, e, f, g, h\}$$



$$A = \big\{\{a, b\}, \{a, d\}, \{a, e\}, \{b, c\}, \{c, d\}, \{d, e\}, \{f, g\}, \{f, h\}, \{g, h\}\big\}$$

Graphes non-orientés

Exemple

Dénombrons tous les graphes non-orientés sans boucles à 3 sommets. Si on note $S = \{a, b, c\}$ *les sommets d'un tel graphe, il y a autant de graphes possibles que de choix d'arêtes. Les arêtes forment un sous-ensemble de* $\mathcal{P}_2(S) = \{\{a, b\}, \{a, c\}, \{b, c\}\}$*, c'est-à-dire que* $A \in \mathcal{P}(\mathcal{P}_2(S))$*, ce qui laisse* $2^3 = 8$ *choix. Il y a donc, une fois les sommets fixés, 8 graphes possibles.*

Plus généralement : si on fixe $S = \{a_1, \ldots, a_n\}$ *les sommets d'un graphe, les arêtes forment un élément de* $\mathcal{P}(\mathcal{P}_2(S))$*, et donc :*

- *comme* $\text{card}(S) = n$*, alors* $\text{card}(\mathcal{P}_2(S)) = \binom{n}{2} = \dfrac{n(n-1)}{2}$ *;*
- *et donc :* $\text{card}\left(\mathcal{P}\left(\mathcal{P}_2(S)\right)\right) = 2^{\text{card}\mathcal{P}_2(S)} = 2^{\frac{n(n-1)}{2}}$

ce qui représente $2^{\frac{n(n-1)}{2}}$ *graphes possibles.*

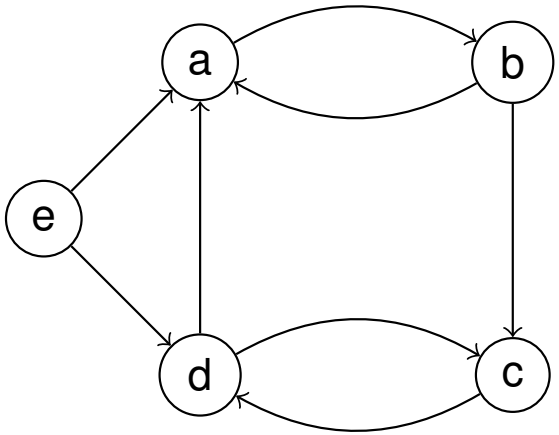
Un graphe non-orienté ***complet*** *possède des arêtes reliant chaque paire de sommets : s'il possède n sommets, il possède* $\binom{n}{2} = \dfrac{n(n-1)}{2}$ *arêtes. Et plus généralement, le nombre d'arêtes d'un graphe non-orienté à n sommets est toujours en* $\mathrm{O}(n^2)$*.*

Graphes orientés

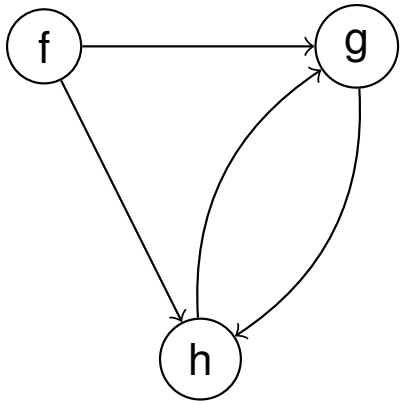
Définition (Graphe orienté)

*Un **graphe orienté** est la donnée d'un couple de la forme $G = (S, A)$, où :*

- *S est un ensemble non vide, dont les éléments sont les **sommets** (ou **nœuds**) ;*
- *A est un ensemble de **couples** d'éléments de S, dont les éléments sont les **arcs**.*



$$S = \{a, b, c, d, e, f, g, h\}$$



$$A = \{(a,b),(b,a),(b,c),(c,d),(d,a),(d,c),(e,a),(e,d),(f,g),(f,h),(g,h),(h,g)\}$$

Graphes orientés

Remarques

- *Les boucles jouent un rôle un peu particulier, et la plupart des graphes que l'on traitera seront sans boucles. L'idée est qu'on vise dans ce cours à faire des **parcours** de graphes, et les boucles ne présentent que peu d'intérêt dans ce cadre.*
- *On ne considèrera pas non plus les **multiarcs/multiarêtes** (des arcs/arêtes qui sont comptées plusieurs fois).*
- *On peut toujours écrire un graphe non-orienté comme un graphe orienté : il suffit, pour l'arête $\{a, b\}$, de considérer les arcs (a, b) et (b, a). À l'inverse, on peut "désorienter" un graphe orienté en transformant tout arc (a, b) en l'arête $\{a, b\}$, mais le graphe obtenu est alors différent.*

Exemple

Un graphe orienté à 3 sommets, notés a, b, c, est déterminé par ses arcs qui forment un sous-ensemble de S^2. Si l'on exclu les boucles, cela laisse 6 arcs possibles, et donc 2^6 graphes orientés possibles.

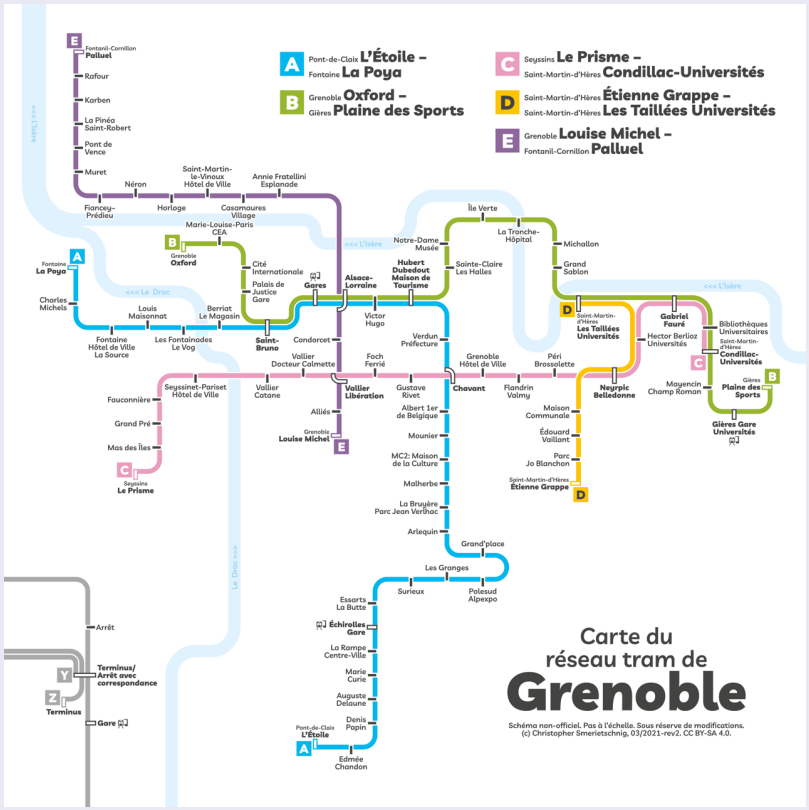
Plus généralement : si on fixe $S = \{a_1, \ldots, a_n\}$ les sommets d'un graphe, il y a $n(n-1)$ couples d'éléments de S dont les coordonnées sont différentes, et donc $2^{n(n-1)}$ graphes orientés (sans boucle) possibles.

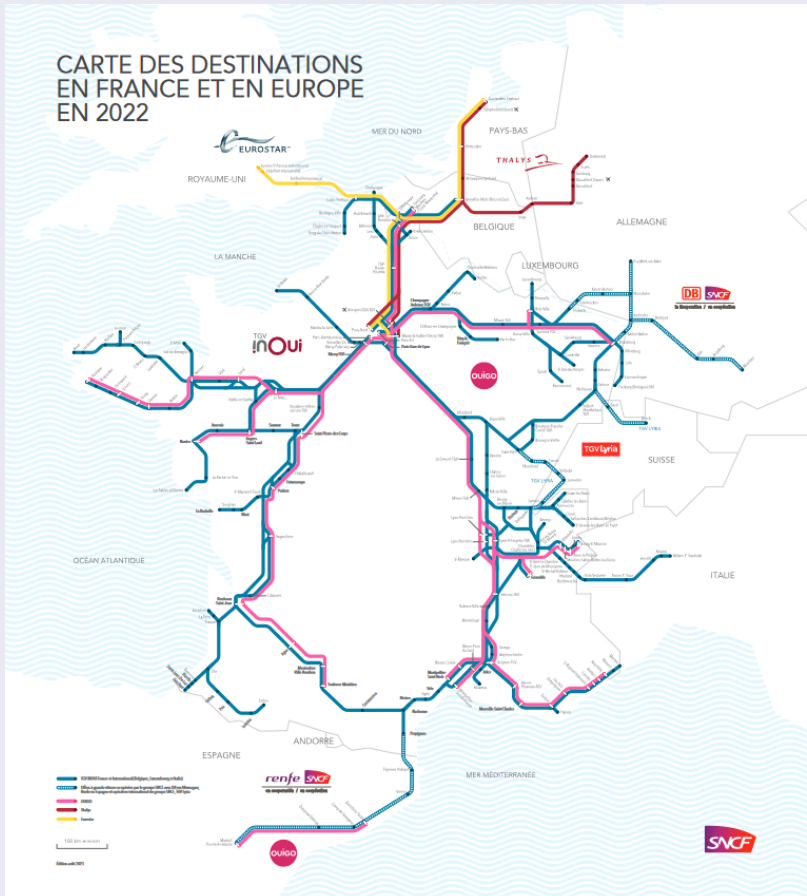
Modélisation et graphes pondérés

Les graphes (orientés ou non) servent à modéliser des situations dans lesquelles on peut **transiter** d'un état à un autre. Les arcs/arêtes sont là pour montrer si l'on peut passer d'un état à un autre.

Exemple

Modélisation d'un réseau (train ou tram) : sommets = stations/villes et arêtes = trajets directs entre stations/villes.





Modélisation et graphes pondérés

Définition

*Un **graphe pondéré** est la donnée d'un graphe, orienté ou non, dont les arêtes/arcs sont munies de **valuation** appelée **poids**.*

Remarques

Les graphes pondérés généralisent les graphes, dans le sens où un graphe non pondéré revient à un graphe dont toutes les valuations valent 1*.*
*Les graphes (pondérés ou non) permettent de représenter différentes situations de **transitions** (modélisées par des arcs/arêtes) entre des **états** (modélisés par des sommets).*
Les pondérations permettent de :

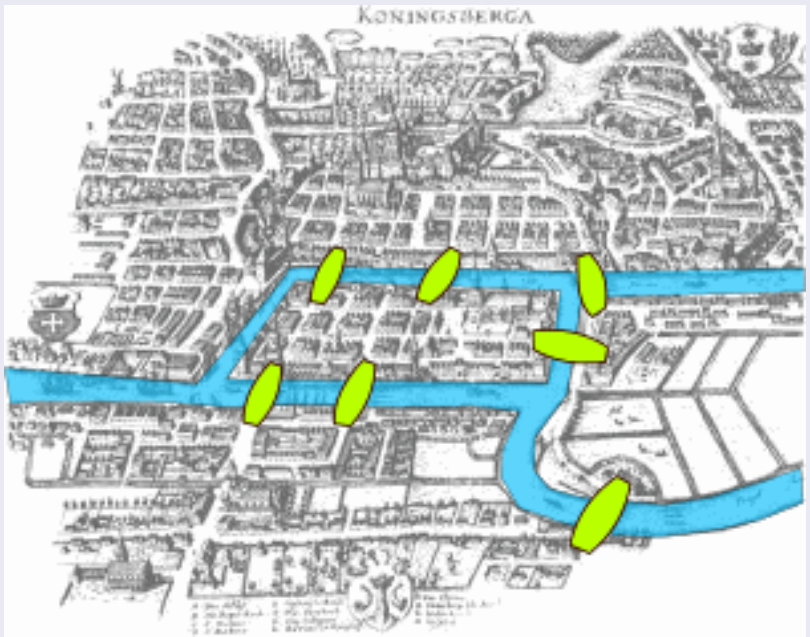
- *mettre en évidence des dissymétries dans les arcs/arêtes (exemple : distances différentes)*
- *avoir des schémas plus lisibles en ne les faisant pas nécessairement à l'échelle*
- *gagner en simplicité de lecture/calculs en retirant éventuellement des sommets*

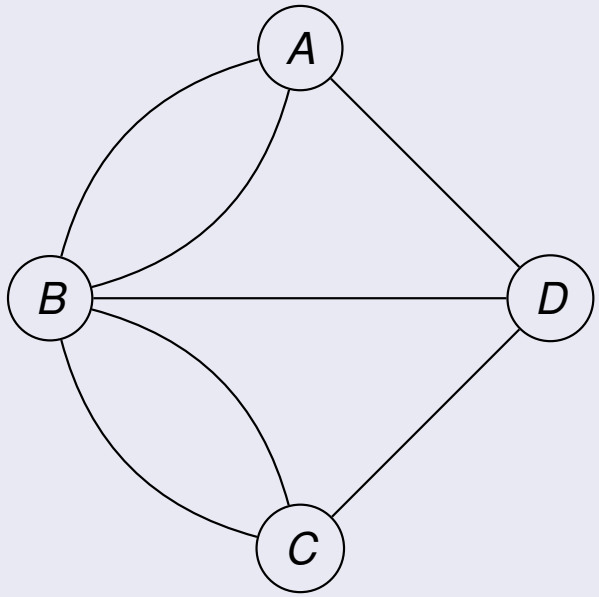
Une pondération n'est pas toujours utile, mais elle est toujours possible.

Modélisation et graphes pondérés

Exemple

La ville de Kaliningrad (anciennement Konigsberg), est traversée par la Pregolia, dans le cours de laquelle se trouvent deux îles. On y a construit sept ponts. On veut savoir si l'on peut se promener entre les îles de sorte à passer une et une seule fois par chaque pont.





C'est un problème de parcours de graphes résolu par Euler : un tel parcours est impossible !

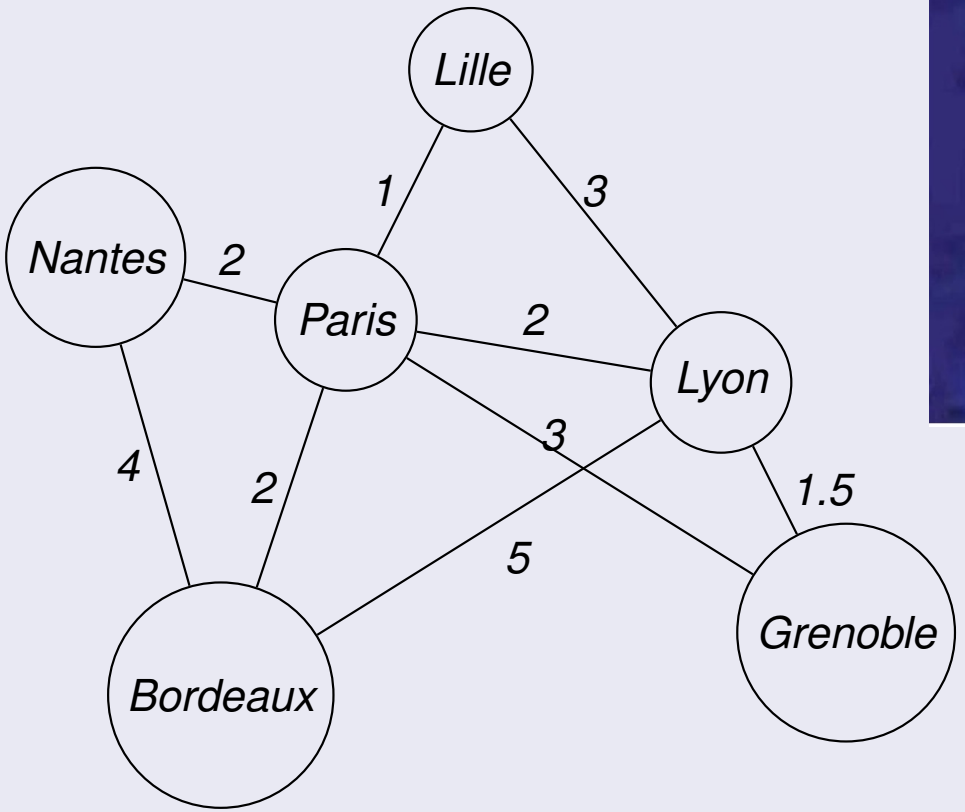
Remarque

On a ici des multi-arêtes...

Modélisation et graphes pondérés

Exemple

*La pondération peut être utile pour des distances : si on s'intéresse seulement aux trajets **directs** en train entre les villes de Paris, Lille, Nantes, Bordeaux, Lyon et Grenoble, on a le graphe suivant (pondéré par les temps de trajets).*



Modélisation et graphes pondérés

Définition

*Un **processus de Markov** (discret) est une succession d'états tel que la probabilité d'être dans un certain état à un instant ne dépend que de l'état dans lequel on était à l'instant précédent. On le modélise par un graphe (orienté pondéré) appelé **graphe probabiliste** dont :*

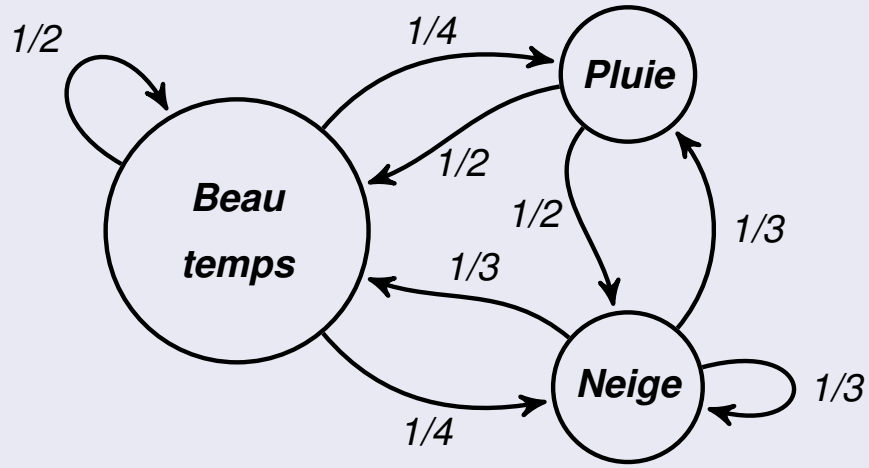
- *les sommets sont les différents états possibles*
- *l'arc entre les états A et B est pondéré par la probabilité de passer de l'état A (à un instant donné) à l'état B (à l'instant suivant)*

Exemple

On considère que l'on a trois type de temps (beau temps, pluie ou neige) et que :

- *il ne pleut jamais deux jours de suite*
- *s'il fait beau un jour, il y a une chance sur deux qu'il fasse beau le lendemain*
- *les autres situations sont équiprobables*

alors on a le graphe suivant.



Remarque

Les boucles sont souvent nécessaires pour les graphes probabilistes.

Modélisation et graphes pondérés

Exemple

*L'algorithme **pagerank** repose sur des graphes probabilistes : l'idée (simplifiée) est de considérer un graphe pondéré orienté dont :*

- *les sommets sont les pages web*
- *les arcs correspondent au lien d'une page vers une autre (pondéré par le nombre de lien)*

*L'algorithme consiste à pondérer les pages (les sommets) par un **indice de popularité** en cherchant à modéliser une balade sur internet comme un déplacement aléatoire dans le graphe précédent, en utilisant que l'indice de popularité d'une page est d'autant plus grand qu'un grand nombre de **pages populaires** on un **lien** vers elle.*
On a une définition récursive, qui fonctionne bien en se disant que :

- *initialement un utilisateur est sur n'importe quelle page avec équiprobabilité*
- *à l'instant suivant il va vers une autre page avec probabilité d'autant plus grande que le nombre de référencement est grand*

et on répète à l'infini (jusqu'à convergence) ce processus pour avoir l'indice de popularité d'une page.

Plan

Voisins et degrés

Définition

*Dans un graphe non-orienté, on dit que deux de ses sommets sont **voisins** (ou qu'ils sont **adjacents**) s'ils sont les extrémités d'une même arête.*
*Pour un sommet s donné, on appellera **degré** de s, noté $d(s)$, le nombre d'arêtes qui possèdent s comme extrémité.*

Définition

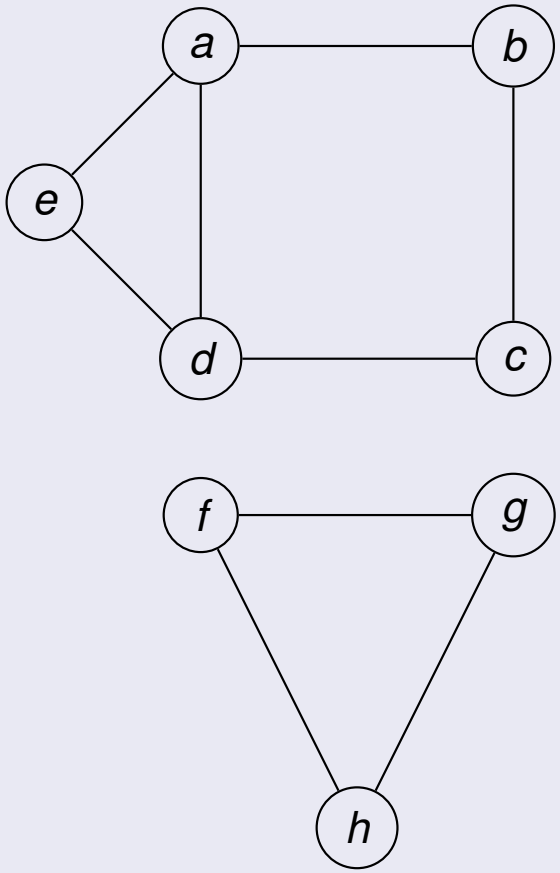
*Dans un graphe orienté, on appelle **degré entrant** (resp. **degré sortant**) d'un sommet s, noté $d_-(s)$ (resp. $d_+(s)$) le nombre d'arcs qui arrivent en (resp. qui partent de) s.*

Remarques

- *On évite de parler de voisins pour les graphes orientés, comme les rôles des sommets dans un arcs ne sont pas symétriques.*
- *Un sommet est son voisin si, et seulement si, ce sommet présente une boucle.*
- *L'absence de multi-arêtes permet de faire le lien entre arêtes et voisins pour exprimer le degré d'un sommet.*
- *Les degrés des sommets sont a priori différents, mais s'ils ont tous même degré on l'appellera le degré du graphe.*

Voisins et degrés

Exemple



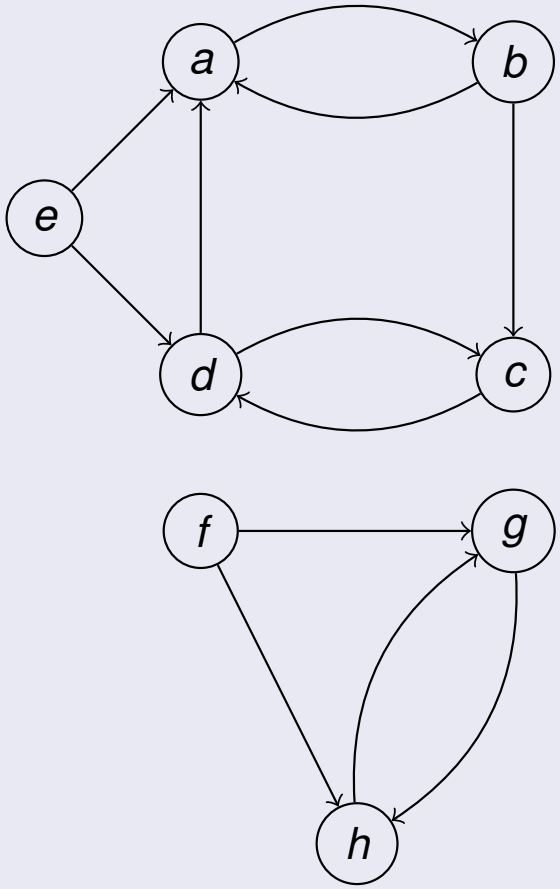
sommet	*a*	*b*	*c*	*d*	*e*	*f*	*g*	*h*
degré	*3*	*2*	*2*	*3*	*2*	*2*	*2*	*2*

Exemple

Dans un graphe non-orienté complet à n sommets, chaque sommet a pour degré $n - 1$.

Voisins et degrés

Exemple



sommet	*a*	*b*	*c*	*d*	*e*	*f*	*g*	*h*
degré entrant	*3*	*1*	*2*	*2*	*0*	*0*	*2*	*2*
degré sortant	*1*	*2*	*1*	*2*	*2*	*2*	*1*	*1*

Exemple

Dans un graphe orienté complet à n sommets, chaque sommet a pour degré entrant ou sortant $n-1$.

Voisins et degrés

Proposition (Formule des degrés)

Si $G = (S, A)$ est un graphe ***non-orienté****, alors :*

$$\sum_{s \in S} d(s) = 2\text{card}(A).$$

Démonstration.

On calcule de deux manières le nombre d'extrémités des arêtes. Il y en a :

- deux fois plus que d'arêtes
- autant que la somme des degrés

□

Remarque

Si $G = (S, A)$ est un graphe ***orienté****, on trouve de la même manière :*

$$\sum_{s \in S} d_+(s) = \sum_{s \in S} d_-(s) = \text{card}(A).$$

Voisins et degrés

Corollaire (Lemme des poignées de main)

Dans un graphe, il y a toujours un nombre pair de sommets de degré impair.

Démonstration.

Par formule des degrés sur un graphe $G = (S, A)$:

$$\underbrace{2\text{card}(A)}_{\text{pair}} = \sum_{s \in S} \deg(s) = \underbrace{\sum_{\substack{s \in S \\ \deg(s) \text{ pair}}} \underbrace{\deg(s)}_{\text{pair}}}_{\text{pair}} + \underbrace{\sum_{\substack{s \in S \\ \deg(s) \text{ impair}}} \underbrace{\deg(s)}_{\text{impair}}}_{\text{pair}}$$

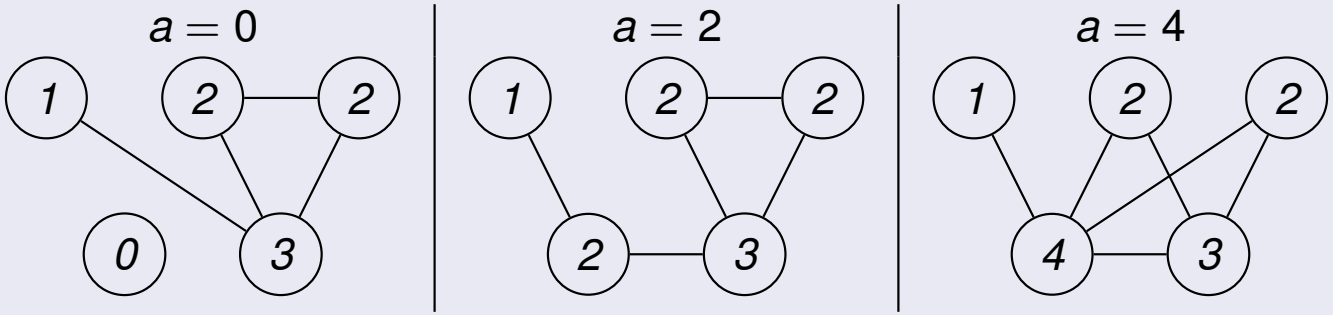
□

Voisins et degrés

Exemple

Pour quelle valeur de $a \in \mathbb{N}$ *existe-t-il un graphe à* 5 *sommets dont les degrés valent* $1, 2, 2, 3, a$ *?*

Nécessairement, a *est pair, et vaut au plus* 4*. Donc* $a \in \{0, 2, 4\}$*. Les graphes suivants donnent les bons degrés selon les valeurs de* a*.*



Représentation par adjacence

Définition

Si $G = (S, A)$ est un graphe (orienté ou non) dont l'ensemble des sommets S est fini de cardinal $n \in \mathbb{N}^*$, identifié à $\{0, \dots, n-1\}$, on lui associe sa ***matrice d'adjacence*** *comme la matrice $M_G = (m_{i,j}) \in \mathcal{M}_n(\mathbb{R})$ définie par :*

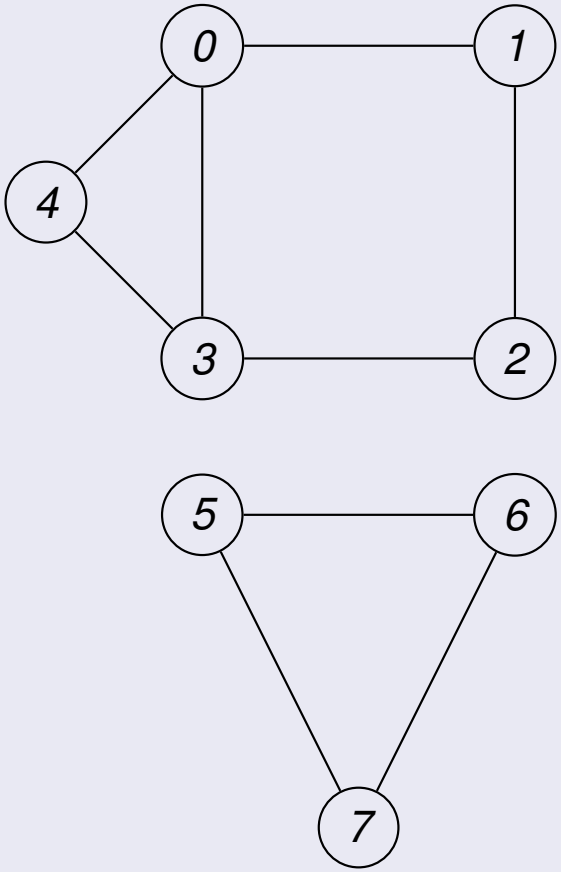
$$\forall i, j \in \{0, \dots, n-1\}, m_{i,j} = \begin{cases} 1 & \text{si } \{i,j\} \text{ ou } (i,j) \in A \\ 0 & \text{sinon} \end{cases}.$$

Remarques

- *La matrice d'un graphe non-orienté est toujours symétrique. Celle d'un graphe orienté l'est seulement s'il provient d'un graphe non-orienté.*
- *On définit de même les matrices d'adjacences de graphes pondérés, en affectant à $m_{i,j}$ le poids de l'arc (i, j). Pour un graphe probabiliste, on parle de* ***matrice de transition****.*
- *L'existence de boucles se lit directement sur les coefficients diagonaux : il y a une boucle sur le sommet i si, et seulement si, $m_{i,i}$ est non nul.*

Représentation par adjacence

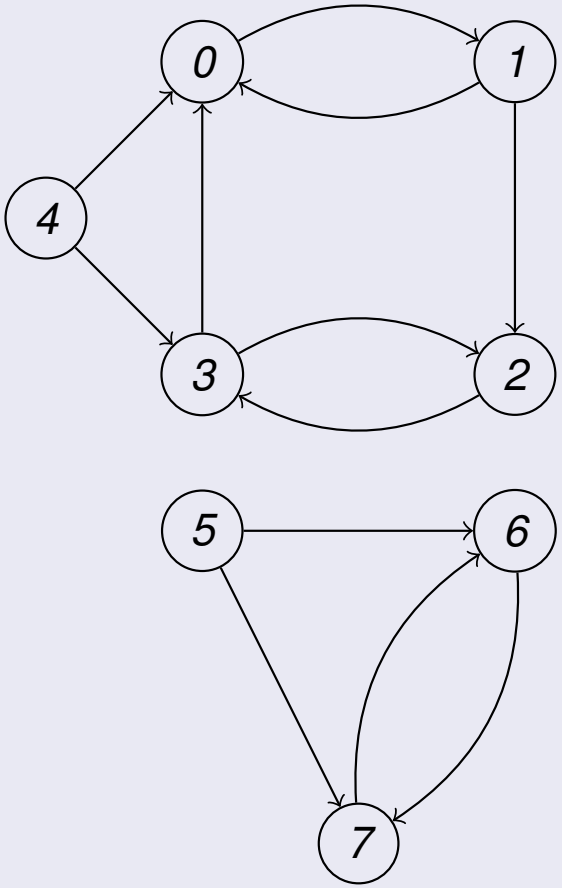
Exemple



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

Représentation par adjacence

Exemple



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

Représentation par adjacence

Définition

*Soit $G = (S, A)$ un graphe (orienté ou non) dont l'ensemble des sommets S est fini de cardinal $n \in \mathbb{N}^*$, identifié à $\{0, \dots, n-1\}$. On lui associe sa **liste d'adjacence** qui est la liste des listes des voisins de ses sommets, c'est-à-dire que c'est une liste à n éléments, numérotés de 0 à $n-1$, dont le i-ème est la liste (éventuellement vide) des $j \in \{0, \dots, n-1\}$ tels que :*

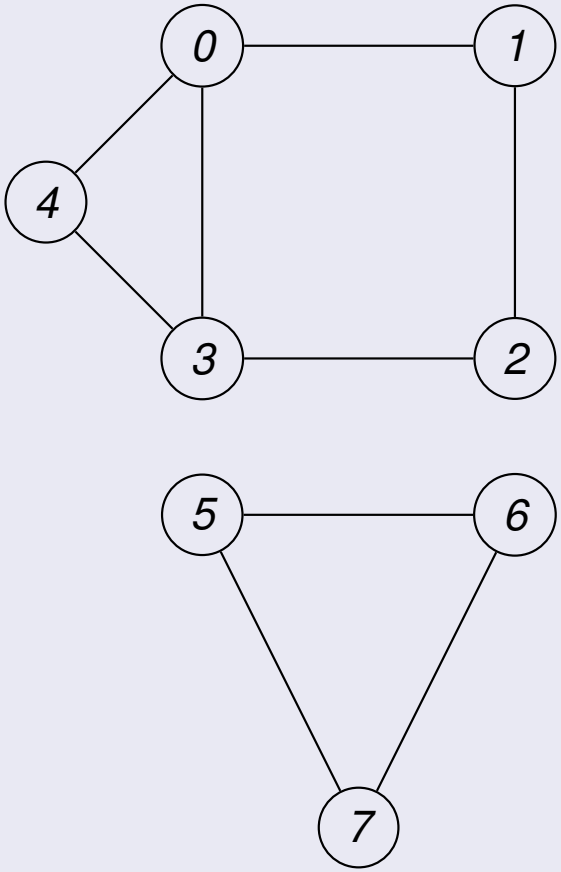
- *si G est non-orienté : $\{i, j\} \in A$;*
- *si G est orienté : $(i, j) \in A$.*

Remarques

- *L'intérêt est qu'une liste adjacence peut simplifier certains calculs : la structure d'une liste se prête mieux à la suppression ou au rajout d'un sommet dans un graphe ; la structure est moins complexe (en mémoire) lorsque le graphe possède peu de sommets (alors que la matrice d'adjacence a même taille, mais possèdera de nombreux 0).*
- *Pour des graphes infinis (ou dont on ne maîtrise pas bien les sommets), on peut construire une **fonction d'adjacence**, qui est une application de S dans $\mathcal{P}(S)$ qui, à tout sommet, associe la liste de ses voisins.*

Représentation par adjacence

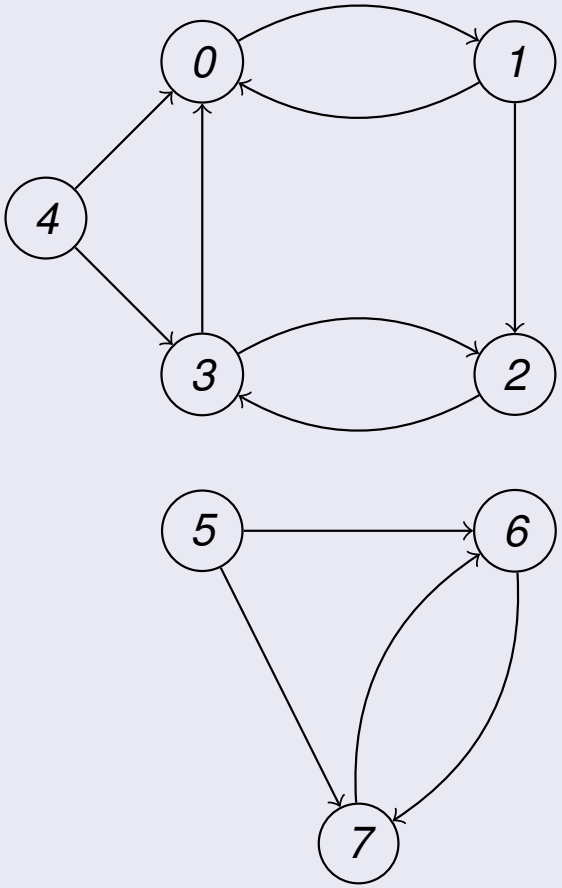
Exemple



$$\begin{bmatrix} [1,3,4], \\ [0,2], \\ [1,3], \\ [0,2,4], \\ [0,3], \\ [6,7], \\ [5,7], \\ [5,6] \end{bmatrix}$$

Représentation par adjacence

Exemple



$$
\begin{bmatrix}
[1], \\
[0, 2], \\
[3], \\
[0, 2], \\
[0, 3], \\
[6, 7], \\
[7], \\
[6]
\end{bmatrix}
$$

Représentation par adjacence

Exemple

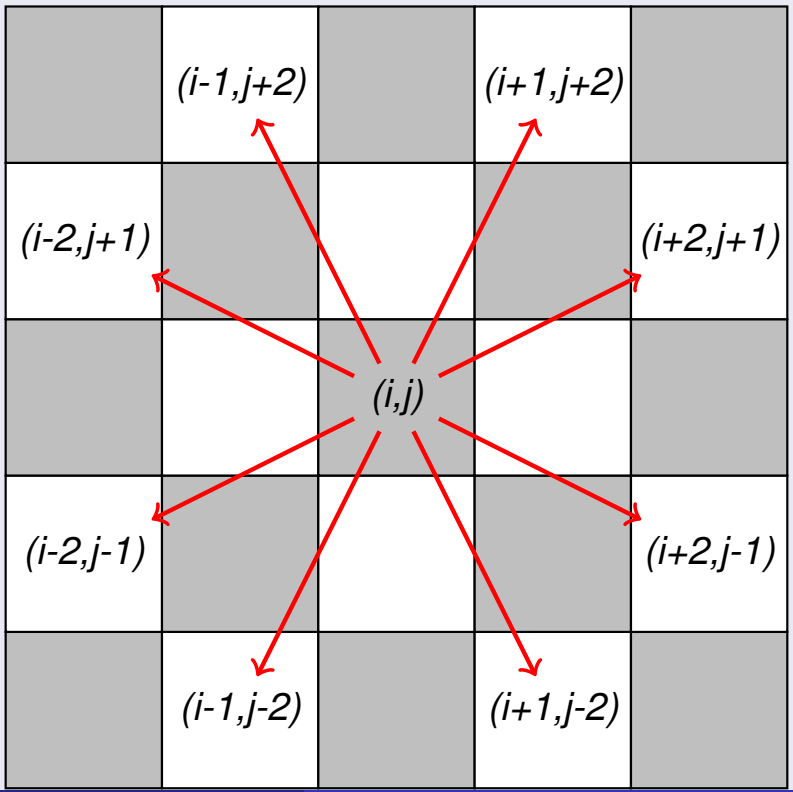
Si on considère une pièce d'échec (fixée une fois pour toutes) sur un échiquier de taille $n \times n$, on peut considérer le graphe (non-orienté) dont :

- *les sommets sont les cases de l'échiquier*
- *les voisins d'une sont les cases atteignables en un coup depuis cette case*

ce qui se modélise bien par fonction d'adjacence.
Par exemple, si on considère un cavalier, la fonction d'adjacence est :

$$(i,j) \mapsto \{(i \pm 1, j \pm 2), (i \pm 2, j \pm 1)\}$$

(à suppression près des cases qui sortent de l'échiquier)



Représentation par adjacence

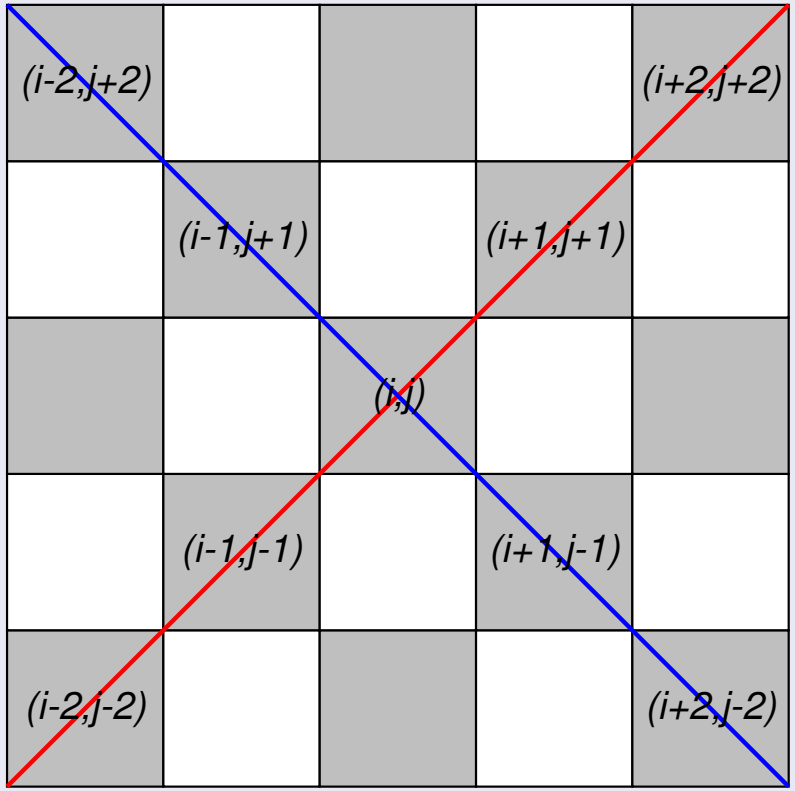
Exemple

Pour un fou sur la case (i, j), il peut aller en un coup sur la case (k, l) dès lors que :

- *elle est sur la diagonale montante passant par (i, j), c'est-à-dire que $k + l = i + j$ (en rouge)*
- *ou sur la diagonale montante passant par (i, j), c'est-à-dire que $k - l = i - j$ (en bleu)*

$$(i, j) \quad \mapsto \quad \{(k, l) \mid k + l = i + j \text{ ou } k - l = i - j\} = \{(i \pm h, j \pm h)\}$$

(à suppression près des cases qui sortent de l'échiquier)



Représentation par adjacence

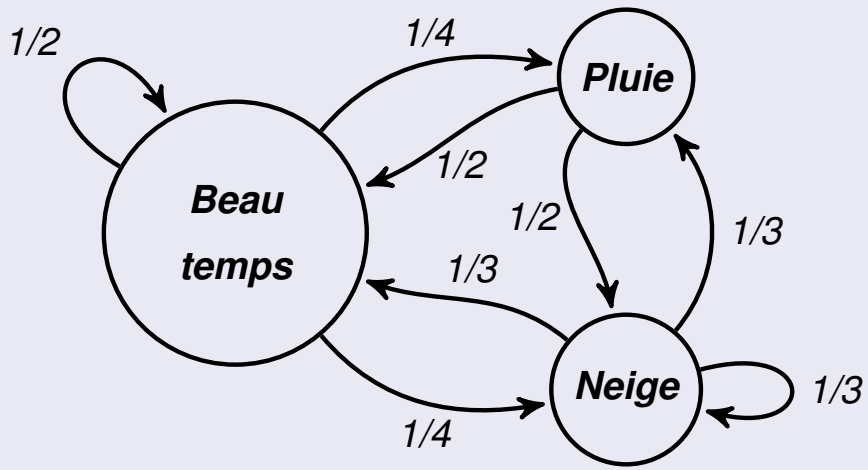
Remarques

- *On préfère utiliser une matrice d'adjacence pour des graphes **denses** (avec beaucoup d'arêtes/arcs) et des listes d'adjacences pour des graphes **creux** (avec peu d'arêtes/arcs).*
- *On perd certaines propriétés liées aux matrices : le fait qu'un graphe soit orienté ou non n'est pas immédiat ; et quelques calculs avec matrices seront utiles pour étudier un graphe et ne sont plus possibles (aussi directement) avec des listes.*
- *La numérotation des sommets de* 0 *à* $n-1$ *est seulement utile pour numéroter les éléments de la liste. On pourrait donc prendre des dénominations quelconques et travailler avec des **dictionnaires** : chaque sommet est une clé du dictionnaire, dont la définition est la liste de ses voisins.*
- *Pour un graphe pondéré, on peut adapter la notation en remplaçant la liste des voisins par la liste des couples de la forme* (s, p) *où* s *est un voisin, et* p *le poids de l'arc associé. L'utilisation d'un dictionnaire est encore plus efficace : on associe à chaque sommet un dictionnaire, dont les clés sont ses voisins et les définitions sont les pondérations des arcs correspondant.*

Représentation par adjacence

Exemple

On peut traiter la liste d'adjacence du graphe probabiliste de la météo par dictionnaire :



$$\begin{pmatrix} 1/2 & 1/4 & 1/4 \\ 1/2 & 0 & 1/2 \\ 1/3 & 1/3 & 1/3 \end{pmatrix}$$

$$\begin{array}{rcl} \{\text{"}\textit{Beau temps}\text{"} & : & \{\text{"}\textit{Beau temps}\text{"} : 1/2, \text{"}\textit{Pluie}\text{"} : 1/4, \text{"}\textit{Neige}\text{"} : 1/4\} \\ \text{"}\textit{Pluie}\text{"} & : & \{\text{"}\textit{Beau temps}\text{"} : 1/2, \text{"}\textit{Neige}\text{"} : 1/2\} \\ \text{"}\textit{Neige}\text{"} & : & \{\text{"}\textit{Beau temps}\text{"} : 1/3, \text{"}\textit{Pluie}\text{"} : 1/3, \text{"}\textit{Neige}\text{"} : 1/3\}\} \end{array}$$

Représentation par adjacence

Exemple

En Python, on peut représenter un graphe de l'une des trois manières suivantes :

- *par l'ensemble (sous forme de liste ou dictionnaire) des sommets et des arcs/arêtes*
- *par matrice d'adjacence (après numérotation des sommets)*
- *par liste d'adjacence*

Les algorithmes pour passer d'une forme à l'autre sont par exemple :

```
1  def matrice_vers_liste(M):
2      adj=[]
3      n=M.shape[0]
4      for i in range(n):
5          voisi=[]
6          for j in range(n):
7              if M[i][j]!=0 :
8                  voisi.append(j)
9          adj.append(voisi)
10     return adj
```

```
1  def liste_vers_matrice(adj):
2      n=len(adj)
3      M=np.zeros((n,n))
4      for i in range(n):
5          for j in adj[i]:
6              M[i][j]=1
7      return M
```

Représentation par adjacence

Exemple

```
def matrice_vers_ens(M):
    ens=[]
    n=M.shape[0]
    for i in range(n):
        for j in range(n):
            if M[i][j]!=0 :
                ens.append([i,j])
    return ens
```

```
def liste_vers_ens(adj):
    n=len(adj)
    ens=[]
    for i in range(n):
        for j in adj[i]:
            ens.append([i,j])
    return ens
```

```
def ens_vers_matrice(ens):
    n=0
    for a in ens :
        if a[0]> n :
            n=a[0]
        if a[1]>n :
            n=a[1]
    M=np.zeros((n,n))
    for a in ens :
        M[a[0]][a[1]]=1
    return M
```

```
def ens_vers_liste(ens):
    n=0
    for a in ens :
        if a[0]> n :
            n=a[0]
        if a[1]>n :
            n=a[1]
    adj=[[] for i in range(n)]
    for a in ens :
        adj[a[0]].append(a[1])
    return adj
```

Chemins, cycles et connexité

Définition (Chemin)

*Dans un graphe $G = (S, A)$ (orienté ou non), un **chemin** entre deux sommets $s, s' \in S$ est une suite d'arcs/arêtes consécutifs qui relient s et s'.*
*La **longueur** d'un chemin est le nombre des arcs/arêtes qui le composent.*

Remarques

- *Un chemin qui passe par n sommets est de longueur $n - 1$.*
- *Un **cycle** est un chemin non trivial qui relie un sommet à lui-même.*
- *On peut aussi parler de chemins dans des graphes pondérés, et leur longueur est alors la somme des pondérations des arcs/arêtes.*

Définition (Distance)

*La **distance** entre deux sommets d'un graphe est la plus petite longueur d'un chemin qui les relie. Par convention, on dira que la distance entre deux sommets qui ne peuvent être reliés par un chemin vaut ∞ et que la distance entre un sommet et lui-même est nulle.*

Chemins, cycles et connexité

Proposition

Dans un graphe non-orienté $G(S, A)$, la distance définie comme ci-dessus est une distance au sens mathématiques sur S, c'est-à-dire qu'elle vérifie les propriétés suivantes :

- ***symétrie*** *:* $\forall a, b \in S,\ \mathrm{d}(a, b) = \mathrm{d}(b, a)$
- ***séparation*** *:* $\forall a, b \in S,\ \mathrm{d}(a, b) = 0 \Leftrightarrow a = b$
- ***inégalité triangulaire*** *:* $\forall a, b, c \in S,\ \mathrm{d}(a, c) \leq \mathrm{d}(a, b) + \mathrm{d}(b, c)$

Démonstration.

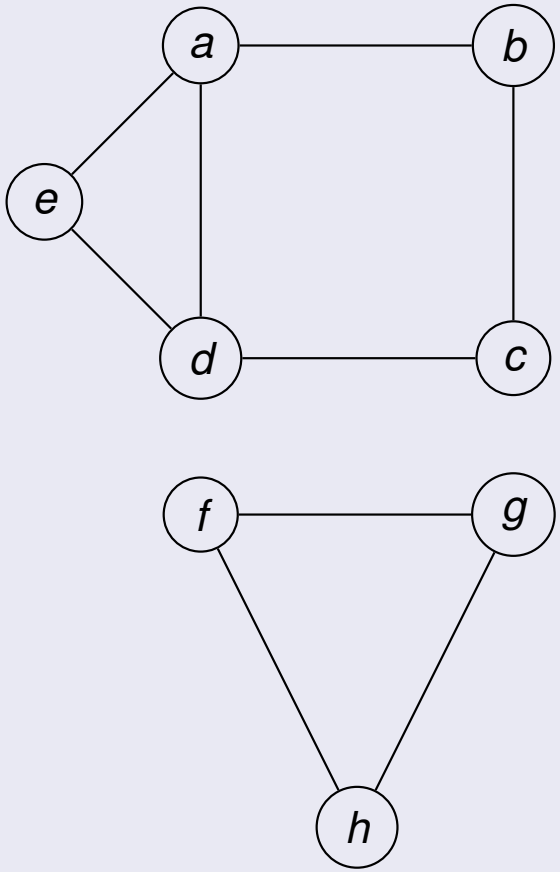
La symétrie vient du fait que le graphe est non-orienté (et légitime de travailler avec de tels graphes)
La séparation vient de la convention choisie.
L'inégalité triangulaire vient du fait que :

- s'il existe un chemin de a à b et un autre de b à c : on peut les recoller pour former un chemin de a à c, dont la longueur est la somme des longueurs des chemins initiaux. Comme il pourrait exister d'autres chemins, on n'a qu'une inégalité.
- s'il n'existe pas de chemin de a à b **ou** de b à c : le membre de droite de l'inégalité à montrer vaut $+\infty$, et l'inégalité est vérifiée.

□

Chemins, cycles et connexité

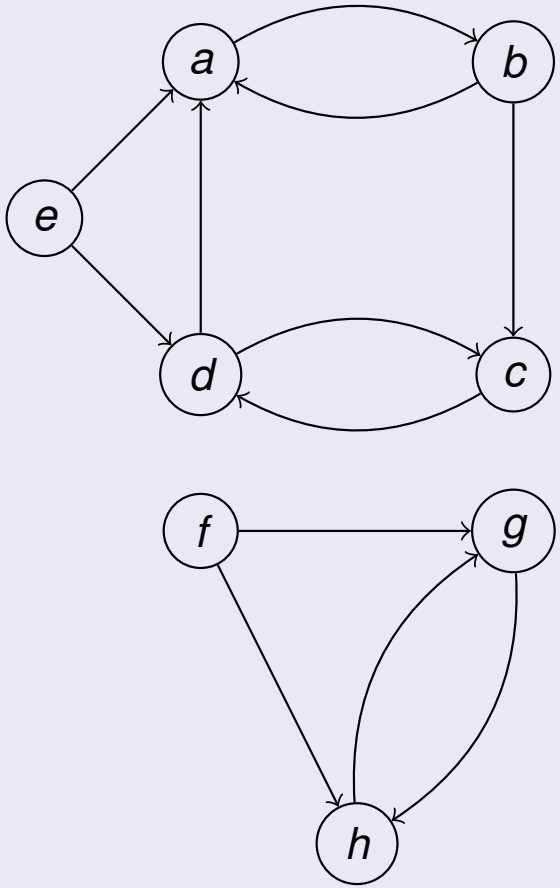
Exemple



	a	b	c	d	e	f	g	h
a	0	1	2	1	1	∞	∞	∞
b	1	0	1	2	2	∞	∞	∞
c	2	1	0	1	2	∞	∞	∞
d	1	2	1	0	1	∞	∞	∞
e	1	2	2	1	0	∞	∞	∞
f	∞	∞	∞	∞	∞	0	1	1
g	∞	∞	∞	∞	∞	1	0	1
h	∞	∞	∞	∞	∞	1	1	0

Chemins, cycles et connexité

Exemple

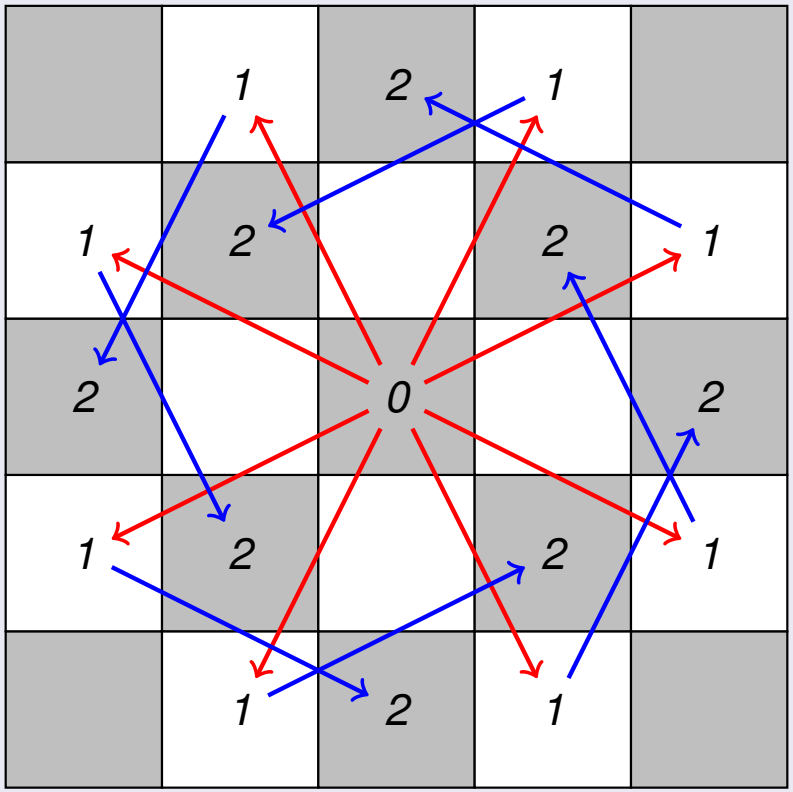


	a	b	c	d	e	f	g	h
a	0	1	2	3	∞	∞	∞	∞
b	1	0	1	2	∞	∞	∞	∞
c	2	3	0	1	∞	∞	∞	∞
d	1	2	1	0	∞	∞	∞	∞
e	1	2	2	1	0	∞	∞	∞
f	∞	∞	∞	∞	∞	0	1	1
g	∞	∞	∞	∞	∞	∞	0	1
h	∞	∞	∞	∞	∞	∞	1	0

Chemins, cycles et connexité

Exemple

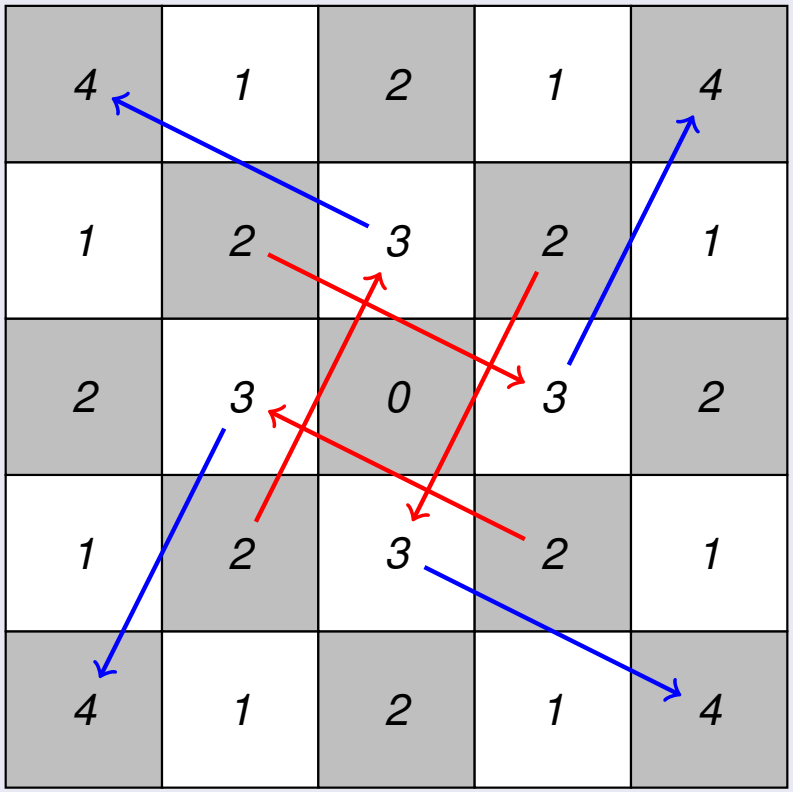
Si on considère cavalier sur un échiquier de taille 5×5 *initialement placé au centre, alors le cavalier peut atteindre n'importe quelle case en au plus* 4 *coups.*
En mettant sur chaque case le nombre (minimum) de coups nécessaire pour l'atteindre, on a la situation suivante :



Chemins, cycles et connexité

Exemple

Si on considère cavalier sur un échiquier de taille 5×5 *initialement placé au centre, alors le cavalier peut atteindre n'importe quelle case en au plus* 4 *coups.*
En mettant sur chaque case le nombre (minimum) de coups nécessaire pour l'atteindre, on a la situation suivante :



Chemins, cycles et connexité

Définition (Connexité)

*Un graphe **non-orienté** est dit **connexe** s'il existe toujours un chemin pour passer d'un sommet à un autre.*
*Dans le cas général, on appelle **composantes connexes** d'un graphe la partitionnement de ses sommets en graphes connexes **les plus grands possibles**.*

Définition

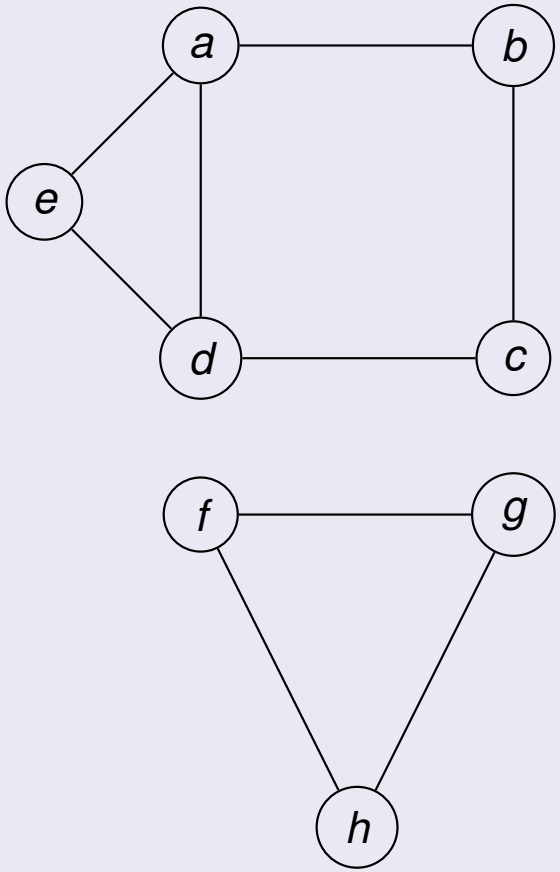
*Étant donné un graphe non-orienté, on appelle **diamètre** la plus grande distance entre deux de ses sommets.*
Par convention, on dira qu'un graphe non-connexe est de diamètre infini.

Remarques

1. *Un graphe est de diamètre* 1 *si, et seulement si, il est complet.*
2. *Un graphe fini est connexe si, et seulement si, il est de diamètre fini. Il existe en revanche des graphes connexes de diamètre infini : par exemple le graphe dont les sommets sont les entiers, et deux sommets sont adjacents si les entiers correspondants se succèdent.*
3. *Si on fixe un sommet d'un graphe non-orienté, le graphe est connexe si, et seulement si, toutes les distances à ce sommet sont finies. De plus, il est de diamètre fini si, et seulement si, il existe un majorant M de l'ensemble de ces distances : le diamètre est alors majoré par $2M$.*

Chemins, cycles et connexité

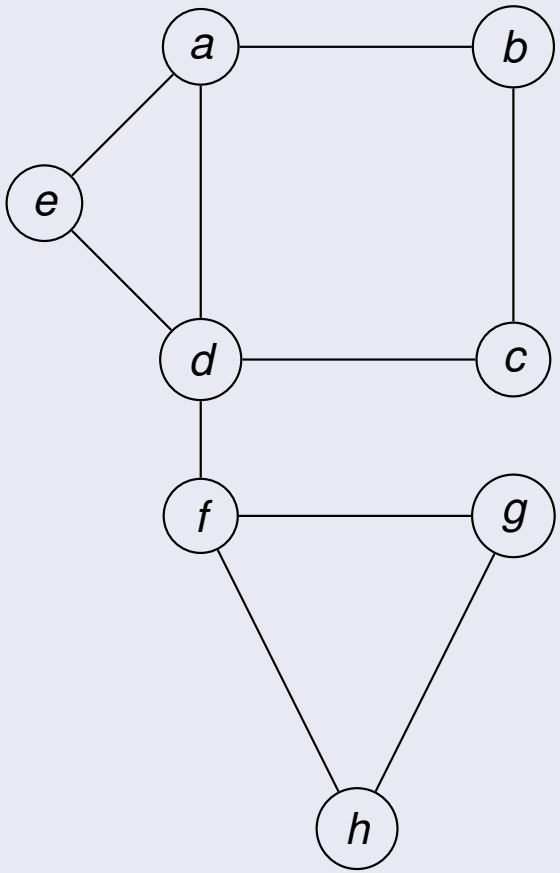
Exemple



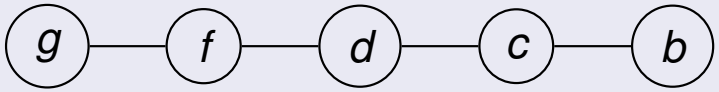
Le graphe n'est pas connexe. Il possède deux composantes connexes, qui partagent les sommets en $\{\{a, b, c, d, e\}, \{f, g, h\}\}$

Chemins, cycles et connexité

Exemple



Le graphe est connexe. Il est de diamètre 4 qui est la distance de b à g (ou à h), reliés par le chemin :



On pouvait regarder les distances, à d, ce qui donne le tableau :

sommet	*a*	*b*	*c*	*d*	*e*	*f*	*g*	*h*
distance	*1*	*2*	*1*	*0*	*1*	*1*	*2*	*2*

ce qui donnait déjà que le diamètre est au plus 2 (quitte à passer par d), avec nécessité de partir et d'arriver en b, g ou h pour réaliser ce maximum (par inégalité triangulaire).

Chemins, cycles et connexité

Exemple

En modélisation, le diamètre et les distances de certains graphes sont devenus célèbres :

- *les **6 degrés de séparation** : le diamètre du graphe dont les sommets sont les individus sur terre, et les arêtes le fait de se connaître, est de diamètre 6, autrement dit :*

  “peu importe qui vous choisissez, je connais quelqu’un qui connaît quelqu’un qui connaît quelqu’un qui connaît quelqu’un qui connaît cette personne”

- *le **nombre de Dieu** est 20 : le diamètre du graphe dont les sommets sont les différentes positions du Rubik’s cube (classique), et les arêtes relie des positions espacées d’un coup, est de diamètre 20, autrement dit :*

  “peu importe dans quelle position on met un Rubik’s cube, il suffit de 20 coups pour le résoudre”

- *à la manière des degrés de séparation, on peut changer la relation “se connaître” en “avoir écrit ensemble un article” ou “avoir joué dans le même film” ou encore “avoir fait un morceau de musique ensemble” ce qui donne les nombres de Erdös–Bacon–Sabbath selon les distances respectives au mathématicien Paul Erdös, à l’acteur Kevin Bacon et au groupe Black Sabbath. Les nombres d’Erdös sont bien connus (tout mathématicien étant co-auteur en possède un), mais le simple fait d’avoir un nombre de Erdös–Bacon–Sabbath est exceptionnel et est réservé à peu d’élus (dont Brian May !)*

Calculs par adjacence

Remarque

Étant donné un graphe représenté par sa liste d'adjacence ou sa matrice d'adjacence, on peut directement déterminer : son nombre de sommets, son nombre d'arcs/arêtes, s'il est orienté ou non.

Proposition

Si G est un graphe orienté ou non, non pondéré, de matrice d'adjacence M_G, alors pour tout $k \in \mathbb{N}^*$, le coefficient d'indice (i,j) de la matrice M_G^k est le nombre de chemins de longueurs k pour aller du sommet i au sommet j.

Démonstration.

On procède par récurrence sur $k \in \mathbb{N}^*$:

- le cas $k = 1$ est évident : c'est la définition de M_G
- l'hérédité se déduit du fait que tout chemin de longueur $k+1$ reliant i à j est le recollement d'un chemin de longueur k reliant i à un sommet l, puis d'un chemin de longueur 1 reliant l à j. De tels chemins sont donc au nombre de :

$$\sum_l (M_G^k)_{i,l} \cdot (M_G)_{l,j} = \left(M_G^{k+1}\right)_{i,j}$$

ce qui conclut la récurrence.

Calculs par adjacence

Corollaire

*Avec les mêmes notations : Le graphe G est connexe si, et seulement si, il existe $k \in \mathbb{N}^*$ tel que la matrice $\mathrm{Id} + M_G + M_G^2 + \cdots + M_G^k$ a **tous ses coefficients** non nuls.*

Remarques

- *On peut adapter les résultats précédents pour détecter des cycles : le fait que $\left(M_G^k\right)_{i,i} \neq 0$ traduit qu'il existe un cycle de longueur k qui part (et arrive) au sommet i, donc regarder les coefficients diagonaux des puissances de M_G donnent une information sur les cycles. Le problème est qu'il faut aussi éliminer les chemins "triviaux" qui bouclent par un retour en arrière.*
- *On peut raffiner le résultat précédent : le fait que $\left(\mathrm{Id} + M_G + M_G^2 + \cdots + M_G^k\right)_{i,j} \neq 0$ traduit qu'il existe un chemin de longueur **au plus** k allant de i à j, et donc le plus petit $k \in \mathbb{N}^*$ tel que $\mathrm{Id} + M_G + M_G^2 + \cdots + M_G^k$ a tous ses coefficients non-nuls est le diamètre de G*

Calculs par adjacence

Définition

Dans un graphe pondéré, on appellera pondération d'un chemin le produit des pondérations des arcs/arêtes qui le composent.

Proposition

Si G est un graphe pondéré, orienté ou non, de matrice d'adjacence M_G, alors pour tout $k \in \mathbb{N}^*$, le coefficient d'indice (i, j) de la matrice M_G^k est la somme des pondérations des chemins de longueurs k pour aller du sommet i au sommet j.

Remarque

On a exactement les mêmes résultats qu'avant (cas non pondéré) comme on avait seulement utilisé la non-nullité des coefficients, donc le poids, tant qu'il est non nul, ne joue pas de rôle particulier.

Corollaire

Dans un graphe probabiliste G de matrice de transition T, si $k \in \mathbb{N}^*$, la probabilité, sachant que l'on est à l'état i à un instant donné d'être à l'état j dans k changements d'états vaut $\left(T^k\right)_{i,j}$

Démonstration.

Par formule des probabilités composées. □

Plan

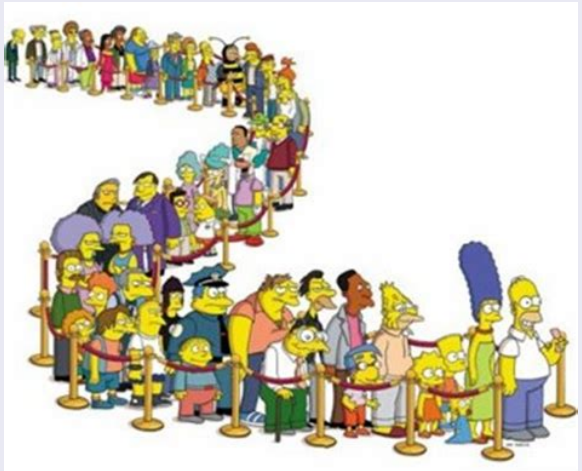
Piles et files

Définition

*Une **structure de données dynamique** est un objet informatique dans lequel on stocke une collection d'objets, dont le nombre peut varier.*
Elle est munie de deux opérations en temps constant : le rajout et l'extraction d'un objet. Selon la manière dont sont exécutées ces opérations, on a les structures dynamiques suivantes :

- *Une **pile** (LIFO = "last in first out") : on rajoute **au-dessus** et on extrait **au-dessus**, c'est-à-dire que c'est toujours le dernier élément rajouté que l'on extrait*

- *Une **file** (FIFO = "first in first out") : on rajoute **à la fin** et on extrait **au début**, c'est-à-dire que c'est toujours l'élément le plus ancien que l'on extrait*



Piles et files

Exemples

- *En Python, les listes sont des piles : on ajoute les éléments avec* `append` *à la fin, et on retire le dernier élément avec* `pop`*.*
- *Les instructions dans les éditeurs de texte sont sous forme de pile : l'extraction correspond au* `Ctrl+Z` *qui retire l'effet de la dernière instruction.*
- *Pour créer des files en Python, on utilise le module* `collections.deque` *:*
  - *l'instruction* `deque()` *crée la liste vide, et* `deque(L)` *transforme la liste* `L` *en une file*
  - *on rajoute l'élément* `a` *à (la fin de) la file* `q` *par l'instruction* `q.append(a)`
  - *on extrait (le premier élément) dans la file* `q` *avec l'instruction* `q.popleft()`

Remarque

Il existe aussi des ***files de priorité*** *: ce sont des listes d'objets pondérés, triés par poids croissants, où :*

- *l'ajout d'un élément se fait tout en maintenant la structure triée par poids (par un* `push`*)*
- *l'extraction se fait sur l'élément de poids minimal (par un* `pop`*)*

et on verra l'intérêt de telles structures pour les parcours de graphes.
Lorsque l'on connaît à l'avance les poids (par exemple si l'on sait que les poids seront des entiers naturels), on peut travailler avec des dictionnaires dont les clés sont les poids, et les définitions sont les files des objets de poids donné.

Généralités sur les parcours

Définition

*Un **parcours de graphe** est un algorithme consistant à explorer tous les sommets d'un graphe de proche en proche à partir d'un sommet déterminé.*

Remarques

- *L'idée d'un parcours est qu'on souhaite avoir **l'ordre** dans lequel on parcourt les sommets. Toute l'idée est donc de savoir quels sommets privilégier pour l'ordre de parcours.*
- *En pratique, on travaille avec deux structures pour faciliter le parcours : une liste qui va correspondre à la liste des sommets parcourus, et un tableau qui va permettre de dire si un sommet a déjà été parcouru ou non.*
- *On travaille surtout avec des graphes non-orientés, ce qui légitime les retours en arrière que l'on doit parfois faire.*
- *Les parcours se font en regardant les voisins de sommets déjà visités. Ainsi, un parcours va bien fonctionner dès lors que le graphe considéré est connexe. Sinon, il faudra fixer un nouveau voisin non visité (arbitrairement ou non) et relancer un parcours, jusqu'à avoir atteint tous les sommets.*
- *On confond parfois **parcours** et **chemin** : un chemin qui passerait par tous les sommets en revenant à son sommet de départ est dit **hamiltonien** tandis qu'un parcours qui passe une et une seule fois par chaque arête est dit **eulérien**.*

Parcours en profondeur

Définition

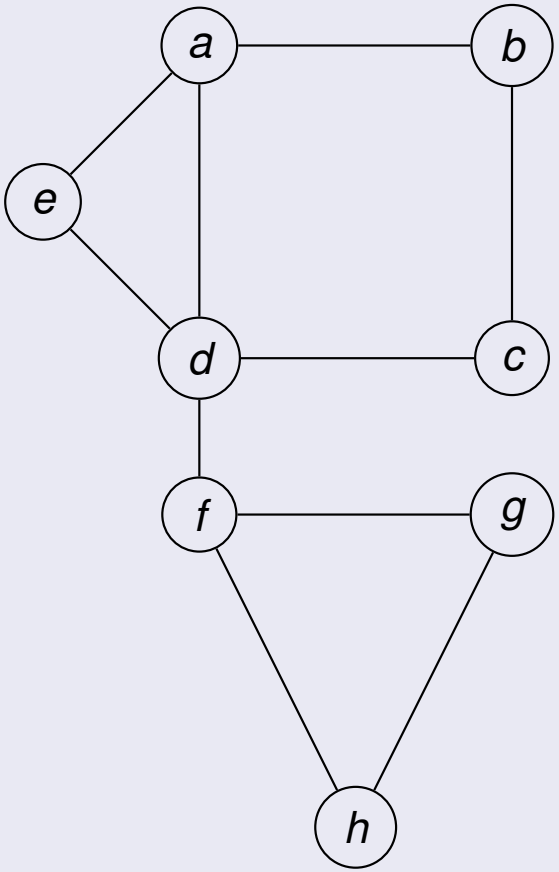
*Un **parcours en profondeur** (ou DFS pour Depth-First Search) est un parcours dans lequel on parcourt en priorité **un voisin** du dernier sommet parcouru.*

Remarques

- *En pratique, cela veut dire qu'on essaie d'aller le plus loin possible dans le graphe (d'où la dénomination). On doit revenir en arrière quand on n'a plus de voisins disponibles que l'on n'a pas déjà parcouru (ou traité). C'est d'ailleurs la méthode intuitive pour trouver la sortie d'un labyrinthe sans tourner en rond.*
- *L'ordre dans lequel on donne les voisins a son importance, et un même graphe, avec même sommet de départ, peut donner différents parcours en profondeur.*
- *L'utilisation d'une **pile** est naturelle pour un parcours en profondeur : on empile les voisins de chaque sommet qu'on visite, ce qui correspond bien à l'algorithme.*

Parcours en profondeur

Exemple



En partant du sommet d, on va :

- *regarder ses voisins (a,c,e,f) ce qui donne la pile* $[a, c, e, f]$*, et on va en f ;*
- *les voisins non traités de f sont g et h, donc la pile devient* $[a, c, e, g, h]$*, et on va en h ;*
- *h n'a pas de voisin non traité, donc on va en g ;*
- *idem pour g donc on va en e ;*
- *idem pour e donc on va en c ;*
- *c possède b comme seul voisin non traité, donc la pile devient* $[a, b]$ *et on va en b ;*
- *b n'a pas de voisin non traité, donc on va en a.*

Et finalement on a tout traité, avec comme parcours : $[d, f, h, g, e, c, b, a]$*.*

Parcours en profondeur

Exemple

Pour un graphe donné par liste d'adjacence, un parcours en profondeur à partir d'un sommet s donne :

```
def parcours_profondeur(G,s):
    n=len(G)
    traites = [False]*n
    pile = [s]
    traites[s]=True
    suivant = 0
    parcours = []
    while suivant<n:
        while len(pile)>0:
            t = pile.pop()
            parcours.append(t)
            for u in G[t]:
                if not traites[u]:
                    pile.append(u)
                    traites[u]=True
        while suivant<n and traites[suivant]:
            suivant+=1
        if suivant==n :
            return parcours
        pile = [suivant]
        traites[suivant]=True
```

Parcours en profondeur

Remarque

On peut aussi traiter les sommets au fur et à mesure qu'on les rencontre (comme si l'on travaillait avec des files plutôt que des piles). La programmation se fait alors plus facilement avec de la récursivité.

```
1  def parcours_profondeur_rec(G, s,parcours=None,traites =None):
2      if parcours is None:
3          parcours = []
4      if traites is None:
5          traites = [False] * len(G)
6      n=len(G)
7      suivant = 0
8      def aux(t):
9          if not traites[t]:
10             traites[t] = True
11             parcours.append(t)
12             for u in G[t]:
13                 aux(u)
14     aux(s)
15     if len(parcours)==n :
16         return parcours
17     while traites[suivant]:
18         suivant+=1
19     return parcours_profondeur_rec(G, suivant,parcours,traites)
```

Parcours en largeur

Définition

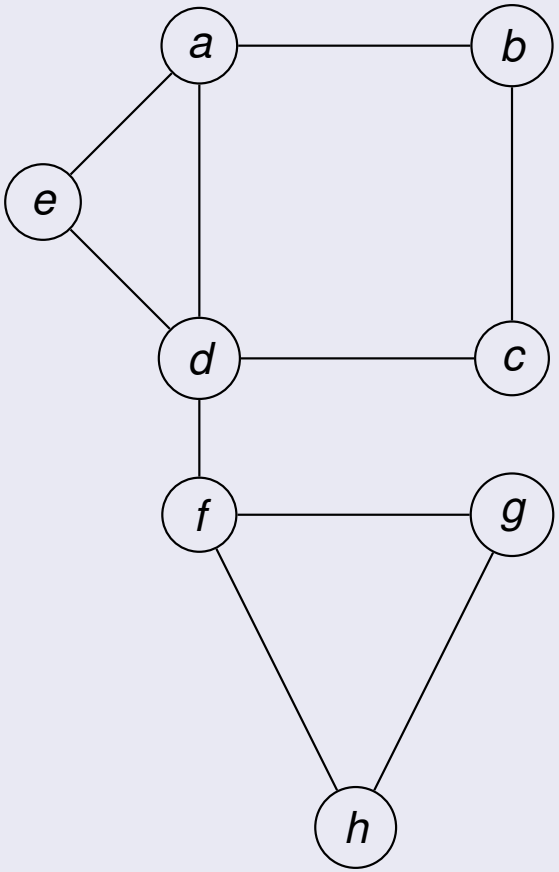
*Un **parcours en largeur** (ou BFS pour Breadth-First Search) est un parcours dans lequel on parcourt en priorité les sommets **les plus proches** du sommet de départ.*

Remarques

- *C'est l'idée inverse du parcours en profondeur : on ne va pas à l'étape suivante (c'est-à-dire à des chemins d'une distance donnée) tant qu'on n'est pas sur d'avoir parcouru **tous** les sommets de distance strictement inférieur.*
- *Ici encore, l'ordre dans lequel on donne les voisins a son importance, et un même graphe, avec même sommet de départ, peut donner différents parcours en profondeur. En revanche, le parcours a davantage de rigidité, puisque les sommets sont rangés par ordre croissant par rapport à la distance au sommet de départ.*
- *L'utilisation d'une **file** est naturelle pour un parcours en profondeur : on finit d'abord tous les voisins déjà considérés avant de passer aux suivants.*

Parcours en largeur

Exemple



En partant du sommet d, on va :

- *regarder ses voisins (a,c,e,f) ce qui donne la file* $[a, c, e, f]$*, et on va en a ;*
- *le seul voisin non traité de a est b, donc on stocke b dans la file qui devient* $[c, e, f, b]$*, et on va en c ;*
- *c n'a pas de voisin non traité, donc on va en e ;*
- *idem pour e donc on va en f ; et c'était le dernier voisin de d, donc tous les sommets suivants sont au moins à distance 2 ;*
- *f possède deux voisins non traités, qui sont g et h, donc la file devient* $[b, g, h]$ *et on va en b ;*
- *b n'a pas de voisin non traité, donc on va en g ;*
- *idem pour g donc on va en h ;*

Et finalement on a tout traité, avec comme parcours : $[d, a, c, e, f, b, g, h]$*.*

Parcours en largeur

Exemple

Pour un graphe donné par liste d'adjacence, un parcours en largeur à partir d'un sommet s donne :

```
def parcours_largeur(G,s):
    n=len(G)
    traites = [False]*n
    file = deque([s])
    traites[s]=True
    suivant = 0
    parcours = []
    while suivant<n:
        while len(file)>0:
            t = file.popleft()
            parcours.append(t)
            for u in G[t]:
                if not traites[u]:
                    file.append(u)
                    traites[u]=True
        while suivant<n and traites[suivant]:
            suivant+=1
        if suivant==n :
            return parcours
        file = deque([suivant])
        traites[suivant]=True
```

Parcours en profondeur

Remarque

On peut aussi faire un traitement avec des piles. L'idée est de différer le traitement en traitant simultanément deux listes : celle des éléments qu'on est en train de traiter (qui sont à distance n du sommet de départ) et ceux que l'on va traiter à la prochaine étape (qui sont à distance $n+1$) :

```
def parcours_largeur_piles(G,s):
    n=len(G)
    traites = [False]*n
    En_cours,A_suivre=[s],[]
    traites[s]=True
    suivant = 0
    parcours = []
    while suivant<n:
        while len(En_cours)+len(A_suivre)>0:
            if len(En_cours)==0 :
                En_cours,A_suivre=A_suivre,[]
            t = En_cours.pop()
            parcours.append(t)
            for u in G[t]:
                if not traites[u]:
                    A_suivre.append(u)
                    traites[u]=True
        while suivant<n and traites[suivant]:
            suivant+=1
        if suivant==n :
            return parcours
        En_cours,A_suivre=[suivant],[]
        traites[suivant]=True
```

Algorithme de Dijkstra

Définition

*L'**algorithme de Dijkstra** est un parcours dans un graphe non-orienté pondéré (positivement) qui consiste à traiter en priorité le sommet **le plus proche** du sommet fixé. Plus précisément, pour traiter un nouveau sommet dans ce parcours :*

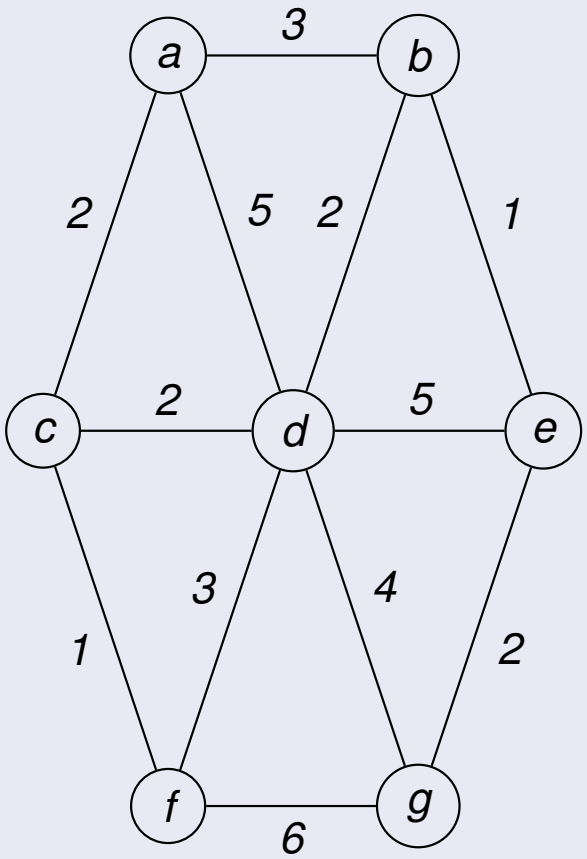
- *on choisit le sommet qui est le plus proche du sommet de départ en se restreignant aux chemins passant par les sommets étudiés ; on note d cette distance ;*
- *on regarde ses voisins : s'il est relié à un sommet par une arête de pondération p, alors :*
  - *ou bien le voisin n'était pas dans la file des sommets à traiter : on l'y rajoute avec pondération $d + p$;*
  - *ou bien le sommet était déjà dans cette file avec pondération P : on change sa pondération par $\min(P, d + p)$.*
- *et on recommence jusqu'à épuisement de la file des sommets à traiter.*

Remarques

- *Dans le cas d'un graphe non pondéré, l'algorithme de Dijkstra est tout simplement un parcours en largeur.*
- *L'algorithme se fait par des files de priorité : c'est une file dont les éléments sont rangés par ordre croissant par rapport à un poids (ici la distance au sommet de départ).*
- *En pratique, on travaille avec des graphes dont on connaît les sommets : dans ce cas on part d'une file de priorité avec des distances infinies partout sauf au sommet de départ (qu'on met à 0).*

Algorithme de Dijkstra

Exemple



En partant du sommet a, les distances dans la file de priorité prennent comme valeurs les différentes lignes du tableau suivant :

a	b	c	d	e	f	g	sommet visité
0	∞	∞	∞	∞	∞	∞	a
0	3	2	5	∞	∞	∞	c
0	3	2	4	∞	3	∞	b
0	3	2	4	4	3	∞	f
0	3	2	4	4	3	9	d
0	3	2	4	4	3	8	e
0	3	2	4	4	3	6	g

Et le parcours devient : $[a, c, b, f, d, e, g]$ *dont les distances au sommet a forment la dernière ligne du tableau, à savoir :* $[0, 2, 3, 3, 4, 4, 6]$ *(bien rangées par ordre croissant).*

Algorithme de Dijkstra

Remarques

- *Les distances au sommet de départ ne sont pas les distances au sens du graphe considéré : ce sont les distances pour le graphe qui a uniquement les sommets parcourus à un moment donné du parcours, mais mêmes arêtes que le graphe initial. D'où l'intérêt de remettre à jour les distances par le* $\min$ *dans l'algorithme.*
- *Le fait de travailler avec des distances positives en prenant à chaque fois le sommet le plus proche assure la correction de l'algorithme dans le sens où les distances rendues sont minimales. Avec des poids négatifs, dès lors que le graphe est non-orienté on peut faire des allers-retours sur une arête pour diminuer le poids du chemin (faisant tendre sa distance vers* $-\infty$*). Et avec un graphe orienté on peut trouver des raccourcis avec des poids négatifs, que Dijkstra n'anticipe pas. Il faut alors avoir recours à d'autres algorithmes (Bellman–Ford ou Floyd–Warshall par exemple). Et encore il ne faut pas de cycle de poids négatif.*

Applications à l'étude de graphes

Proposition

Les parcours permettent d'analyser la connexité d'un graphe de la manière suivante :

- *les parcours en profondeur ou en largeur permettent de déterminer si un graphe est connexe ;*
- *étant donné un sommet, les parcours en profondeur ou en largeur partant de ce sommet permettent de déterminer quels sommets du graphe lui sont reliés par un chemin, en donnant un chemin explicite.*

Remarques

- *Selon l'utilité qu'on a, on peut sensiblement simplifier les algorithmes de parcours. Le fait de rechercher seulement les sommets accessibles revient par exemple à travailler avec le graphe possédant les mêmes arêtes que le graphe initial, mais seulement les sommets accessibles, et se comporte donc comme si le graphe était connexe.*
- *Les chemins donnés n'ont pas de raison a priori de réaliser la distance. C'est le cas dans les parcours en largeur (ou dans l'algorithme de Dijkstra) comme on procède par distances croissantes. Pour les parcours en profondeur, cela reste vrai s'il n'y a pas de cycle (il y a alors unicité des chemins).*

Applications à l'étude de graphes

Proposition

Les parcours permettent d'analyser les distances dans un graphe de la manière suivante

- *les parcours en largeur permettent de déterminer les distances impliquant le sommet de départ, en donnant à chaque fois le chemin qui réalise la distance ;*
- *la répétition du parcours en largeur en partant de chaque sommet permet de déterminer le diamètre, en donnant les deux sommets les plus éloignés, ainsi que le chemin qui réalise le diamètre.*

Remarques

- *L'algorithme de Dijkstra généralise cette méthode aux graphes pondérés positivement.*
- *Pour chercher le chemin qui minimise la distance entre deux sommets, on peut aussi faire un double parcours en largeur : on part de deux sommets pour des parcours en largeur, et on s'arrête quand les deux parcours rencontrent un même sommet. Il suffit alors de recoller les chemins.*

Applications à l'étude de graphes

Exemple

Un parcours en largeur donne que, parmi les 3 674 160 *possibilités du Rubik's cube* $2 \times 2 \times 2$*, le nombre de sommets à distance donnée de n'importe quel sommet fixé est donné par le tableau suivant :*

distance	*0*	*1*	*2*	*3*	*4*	*5*
nombre de sommets	*1*	*9*	*54*	*321*	*1847*	*9992*
distance	*6*	*7*	*8*	*9*	*10*	*11*
nombre de sommets	*50136*	*227536*	*870072*	*1887748*	*623800*	*2644*

Ce qui donne que le diamètre est de 11*.*

Remarques

- *Une spécificité dans cet exemple est que tous les sommets ont des rôles symétriques : on a donc directement le diamètre en partant d'un sommet quelconque. Sinon, il faudrait recalculer les distances à partir de tous les sommets, ce qui prendrait beaucoup trop de temps.*
- *Pour la résolution du Rubik's cube* $2 \times 2 \times 2$*, on peut procéder suivant un parcours en largeur classique, qui donne le plus court chemin pour rejoindre la position de départ. On peut procéder de deux manières pour être efficace :*
  - *ou bien faire une fois pour toutes un parcours à partir de la version résolue : cela demande d'analyser les* 3 674 160 *positions accessibles, en faisant intégralement un parcours en largeur.*
  - *ou bien faire un parcours bilatère : on part de la position résolue et de la position mélangée pour initier deux parcours en largeur, ce qui limite aux sommets à distance* 5 *ou* 6*.*

Applications à l'étude de graphes

Proposition

Les parcours permettent de rechercher les cycles dans des graphes :

- *avec les parcours en profondeur : le fait de revenir sur un sommet déjà parcouru traduit directement la présence d'un cycle ;*
- *avec les parcours en largeur : un fois le parcours en largeur terminé, l'étude des distances permet de repérer des cycles dès lors que le graphes possède : soit deux sommets adjacents à même distance, soit un chemin de trois sommets où les distances ne croissent pas toujours de* 1*, ou ne décroissent pas toujours de* 1*.*

Exemple

Le parcours en profondeur récursif donne l'algorithme de détection de cycles suivant :

```
1  def cycle(G, s):
2      traites = [False] * len(G)
3      def aux(t,p):
4          if traites[t]:
5              return True
6          traites[t] = True
7          for u in G[t]:
8              if u!=p and aux(u,t):
9                  return True
10         traites[t]=False
11         return False
12     return aux(s,None)
```

Heuristique et parcours eulérien

Le problème des parcours développés est que les sommets choisis n'ont aucune raison de rapprocher de la solution :

- pour les parcours en profondeur : on pourrait très bien passer à côté du sommet que l'on cherche à rejoindre sans s'en rendre compte, et un sommet, même très proche du sommet initial, pourrait être rencontré à la toute fin ;
- pour les parcours en largeur : un sommet éloigné, même s'il existe un chemin assez naturel pour l'atteindre, demande de parcourir tous les sommets plus proches, quand bien même beaucoup d'entre eux seraient inutile pour le chemin final.

Définition (Heuristique)

*Méthode de calcul qui fournit **rapidement** une solution **réalisable**.*

Remarques

- *En aucun cas la solution qui sera trouvée sera optimale. Mais ce sera une solution, ce qui permettra d'éliminer certains débuts de solutions si ceux-ci sont déjà moins bons que la solution donnée par l'heuristique.*
- *Cette méthode s'inscrit dans le cadre plus général des **algorithmes gloutons** : ce sont des algorithmes qui essaient d'aller le plus vite vers la solution, sans se soucier du fait que l'on arrivera à une solution, ni à l'optimalité de la solution.*

Heuristique et parcours eulérien

Exemples

Un algorithme glouton classique consiste en la méthode suivante de rendu de monnaie : on rend la plus grosse valeur qui ne dépasse pas le montant visé, et on continue jusqu'à atteindre ce montant

- *avec un nombre illimité de pièces dont les valeurs font* 1, 2, 5, 10, 20, 50, 100, *etc. : la méthode gloutonne donne toujours une solution, et celle-ci est optimale !*
- *on peut ne même pas avoir de solution : par exemple, avec des pièces de valeurs* 5 *et* 2 *seulement, pour rendre* 6 *l'algorithme va rendre* 5 *puis être bloqué, alors que l'on pouvait rendre trois pièces de* 2 *;*
- *ou ne pas être optimal : avec des pièces de* 90, 50, 1, *pour rendre* 100, *on va rendre* 90 *et* 10 *pièces de* 1 *alors que deux pièces de* 50 *sont mieux.*

Remarque

*À l'inverse de l'**algorithme glouton** qui ne tente qu'une configuration qui devrait fonctionner, on trouve la **force brute** (ou **méthode naïve**) : celle-ci consiste à traiter **absolument toutes** les possibilités, pour en déduire la solution optimale (si elle existe).*
Le problème de la force brute vient du fait qu'elle demande beaucoup plus de temps pour être exécutée, et n'est souvent pas réalisable.

Heuristique et parcours eulérien

Définition

L'algorithme A (A **étoile** ou A **star**) est un algorithme de recherche de plus court chemin entre deux sommets qui repose sur une heuristique appliquée à l'algorithme de Dijkstra pour prioriser certains sommets durant le parcours.*

Remarques

- *On utilise donc des files de priorité, dont les pondérations prennent en compte la distance (comme pour Dijkstra) et l'heuristique.*
- *L'algorithme de Dijkstra (et donc les parcours en largeur) correspondent à une heuristique nulle.*

Exemples

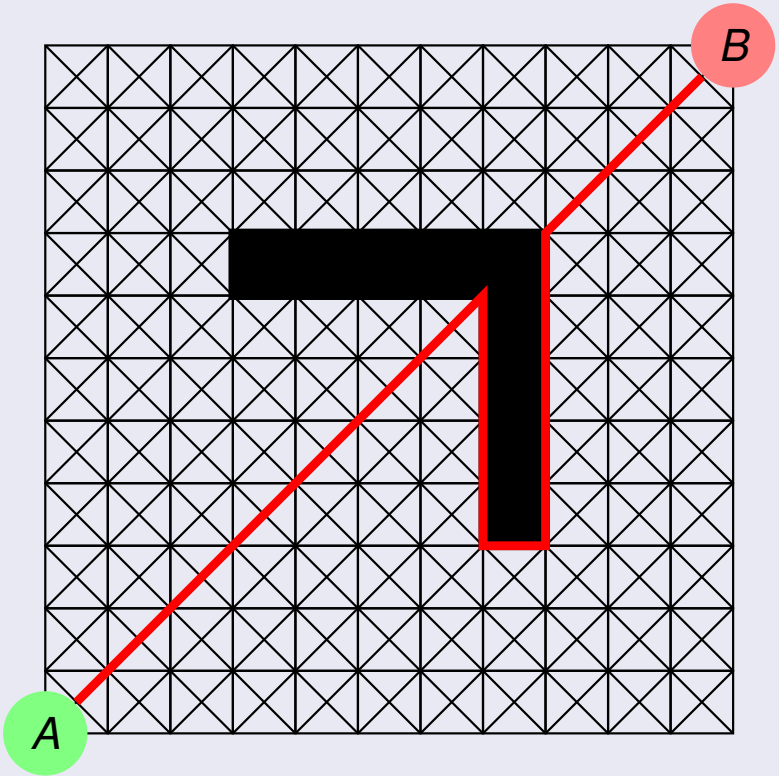
Quelques résolutions célèbres reposant sur une heuristique :

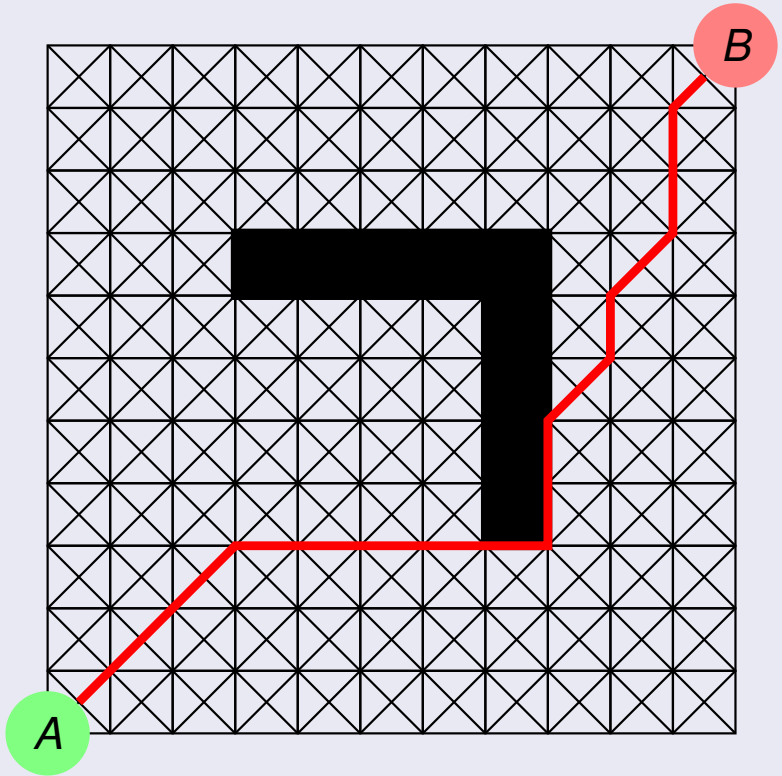
- *résolution de Taquin : on choisit comme heuristique la somme des distances horizontales et verticales entre la position de départ et la position actuelle (sorte de mesure de désordre) ;*
- *problème du voyageur : on cherche le chemin le plus court passant par tous les points d'un graphe et revenant au sommet de départ, en prenant comme heuristique la distance au dernier sommet visité (on vise le plus proche voisin) ;*
- *problème de percolation : on estime l'intérêt d'une case en fonction de son nombre de voisins non visités, et de sa distance à la paroi que l'on vise.*

Heuristique et parcours eulérien

Exemple

Si on cherche à relier deux sommets A et B placés dans un quadrillage, séparés par un mur, l'idée gloutonne est de se rapprocher le plus vite possible de B en partant en diagonale. En corrigeant cette méthode gloutonne, on a un chemin court mais pas optimal. Avec A on a un chemin optimal.*
L'heuristique est l'éloignement d'un sommet à la diagonale.





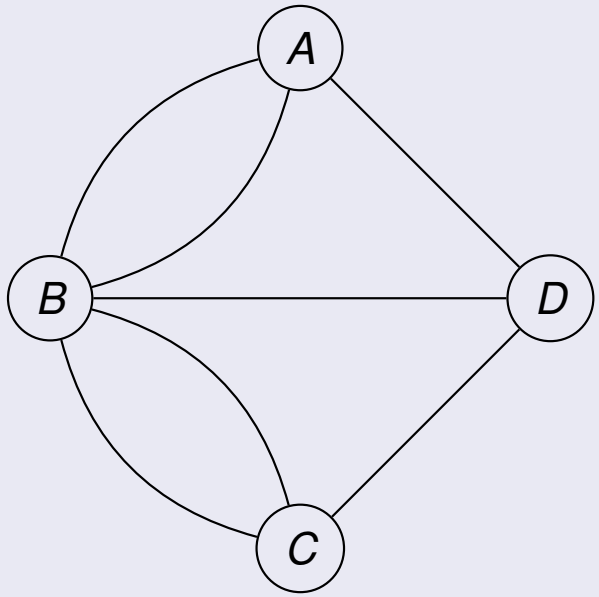
Heuristique et parcours eulérien

Définition

*Un **parcours eulérien** est un chemin dans un graphe non orienté qui passe une et une seule fois par chaque arête.*
*Si le chemin considéré est un cycle, on parlera de **cycle eulérien**.*

Exemple

Le problème des ponts de Konigsberg peut se reformuler comme : le graphe ci-contre admet-il un parcours eulérien ?



Heuristique et parcours eulérien

Théorème (Euler–Hierholzer)

Un graphe (non-orienté) connexe admet :

- *un parcours eulérien si, et seulement si, ses sommets sont tous de degré pair sauf au plus deux*
- *un cycle eulérien si, et seulement si, tous ses sommets sont de degré pair.*

Démonstration.

- condition nécessaire (Euler) : si un parcours existe, on le suit en supprimant les arêtes au fur et à mesure. Chaque sommet visité a son degré diminué de 2 (sauf éventuellement le début et la fin du parcours). Comme on supprime toutes les arêtes, on fait tomber les degrés à 0, qui sont donc pairs.
- condition suffisante (Hierholzer) : on part d'un sommet et on avance tant que c'est possible en supprimant les arêtes. Nécessairement, si on ne peut plus avancer, on est revenu au point de départ. Ou bien on a tout éliminé et on a un parcours eulérien. Ou bien ce n'est pas le cas, et alors on fait un autre cycle, qui se recole au premier et donne un plus grand cycle. Et on recommence jusqu'à épuisement.

□

Exemple

Le problème des ponts de Konigsberg n'admet pas de solution : ses sommets sont de degré 5, 3, 3 *et* 3. *Il suffit en revanche de retirer un pont (n'importe lequel) pour avoir un parcours eulérien.*