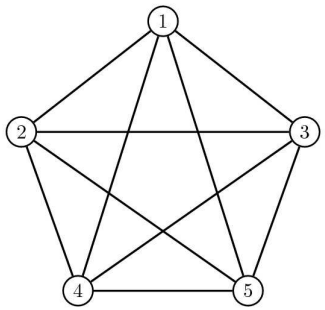
Théorie des graphes

DUT Informatique, semestre 2

Version 2.0

3 février 2014



Ph. Roux

Table des matières

Avertissement

Pour bien utiliser ce polycopié, il faut le lire au fur et à mesure de l'avancement du cours magistral, et prendre le temps de refaire les exercices types qui y sont proposés.

- Les définitions et théorèmes sont numérotés suivant le même ordre que dans le cours magistral.
  Théorème 0.0.0 les théorèmes apparaissent toujours dans un cadre grisé comme celui-ci et sont en général suivis de leur démonstration, signalée par une barre dans la marge et un □ à la fin comme ci-dessous :
  Preuve : Début de la démonstration . . .
  . . .fin de la démonstration
  □
- La table des matières et l'index (à la fin du document) permettent de retrouver une notion précise dans ce polycopié.
- Les méthodes et techniques qui seront approfondies en TD sont signalées par un cadre (sans couleurs)
- Des exercices types corrigés, *rédigés comme vous devriez le faire en* DS, sont signalés par le symbole :

- Les erreurs et les confusions les plus fréquentes sont signalées dans des cadres rouges avec le symbole :

-

Origines historiques

Les mathématiques fournissent de puissants outils pour modéliser des problèmes de toutes sortes :

- les structures booléennes pour les problèmes de logique ($\wedge$, $\vee$, $\neg$, $\Longrightarrow$, ...)
- les ensembles pour représenter des collections d'objets $\mathbb{N}$, $\mathbb{R}$, $\mathbb{R}^2$, $\mathscr{M}_{p,n}(\mathbb{R})$ ...
- les fonctions, dérivées, intégrales pour réaliser des calculs ...

Mais ses outils sont insuffisants, même à notre niveau, pour pouvoir modéliser des problèmes d'apparence pourtant assez simple. Un bon exemple de ce type de problème peut être trouvé dans le domaine des bases de données :

« *comment modéliser les liens entre des objets pris dans différents ensembles ?* »

Pour cela nous avons besoin d'un nouveau type d'objet mathématiques : **les graphes**.

Par rapport aux autres théories mathématiques étudiées à l'IUT, la théorie des graphes est assez récente. L'article considéré comme fondateur de la théorie des graphes fut présenté par le mathématicien suisse Leonhard Euler à l'Académie de Saint Pétersbourg en 1735, puis publié en 1741, et traitait du problème des sept ponts de Königsberg. Le problème consistait à trouver une promenade à partir d'un point donné qui fasse revenir à ce point en passant une fois et une seule par chacun des sept ponts de la ville de Königsberg.

Au milieu du $XIX^{\text{ième}}$, c'est le « théorème des quatre couleurs » qui va populariser dans le monde des mathématiques cette théorie peu connue jusque là. Ce théorème affirme qu'on a besoin que de quatre couleurs différentes pour colorier n'importe quelle carte géographique de telle sorte que deux régions limitrophes (ayant toute une frontière commune) reçoivent toujours deux couleurs distinctes. Le résultat fut conjecturé en 1852 par Francis Guthrie, intéressé par la coloration de la carte des régions d'Angleterre, mais ne fût démontré qu'en 1976 par deux Américains Kenneth Appel et Wolfgang Haken. Leur démonstration de ce théorème fut la première à utiliser un ordinateur pour étudier les 1478 cas particulier aux quels se ramène le problème des quatre couleurs critiques ce qui nécessita plus de 1200 heures de calcul !

C'est donc au $XX^{\text{ième}}$ que cette théorie va connaître son véritable essor avec l'utilisation croissante dans la vie quotidienne des réseaux dont il faut optimiser l'utilisation constamment :

- réseaux de transport routier, transport d'eau, d'électricité
- réseaux de transport de données (réseau de téléphonie fixe, GSM, wifi ...)
- réseaux d'informations (bases de données, web, réseaux sociaux ...)

Cette théorie est devenue fondamentale en informatique car elle fournit de nombreux algorithmes pour résoudre des problèmes complexes représentés par des graphes de très grande taille (plusieurs centaines, milliers,... de sommets et d'arcs !).

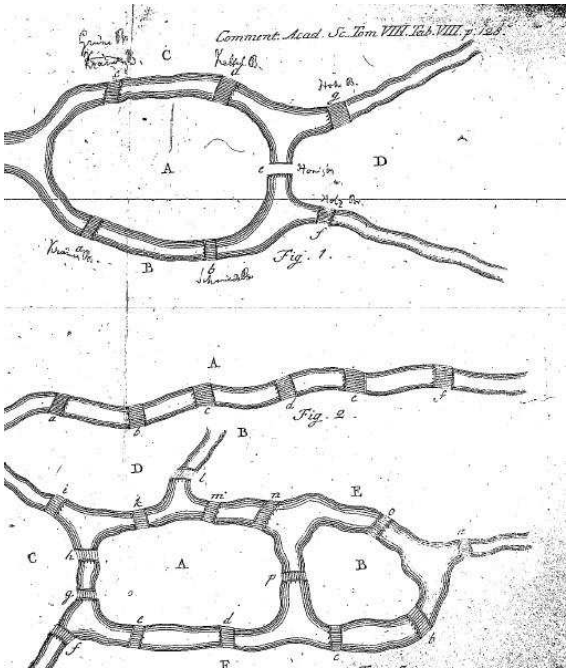


FIGURE 1 – les 7 ponts de Königsberg

FIGURE 2 – une carte géographique coloriée avec 4 couleurs seulement

1 Différentes notions de graphes

1.1 Relations binaires

La notion de graphe repose avant tout sur la notion de **relation binaire**, pour l'introduire nous allons commencer par prendre un exemple de la vie courante.

✎ **1.1 Emploi du temps** *Un emploi du temps met en relation des jours (ou des créneaux horaires) et des matières (et éventuellement des enseignants, des salles ...) :*

Lundi	*Mardi*	*Mercredi*	*Jeudi*	*Vendredi*	*Samedi*	*Dimanche*
Archi	*Système*	*Algo*	*Maths*	*EGO*		
Algo	*Système*	*Anglais*	*EC*	*EGO*		
Archi	*Archi*	*Algo*	*Algo*	*Maths*		
Algo			*EC*	*Maths*		

On est donc en présence de deux ensembles de données dans cet exemple : les jours de la semaine et Les matières enseignées. Mais il y a une donnée supplémentaire qu'on ne peut pas représenter par un ensemble : la relation qui existe entre les jours et les matières. On peux l'exprimer simplement par la phrase :

« une matière est en relation avec les jours de la semaine où elle est enseignée »

On peut essayer de représenter ces liens sur un diagramme en les représentant par des flèches comme sur la figure FIG.3. On se rend alors facilement compte qu'on ne peut pas modéliser ces liens en utilisant des fonctions ou des applications d'un des ensembles vers l'autre. On a besoin d'une notion plus générale ...

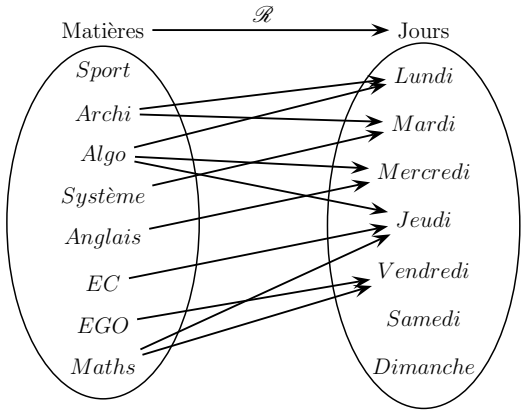


FIGURE 3 – Relations entre les matières et les où elles sont enseignées

Définition 1.1 (Relation binaire) *Soient E et F deux ensembles alors*

- *« $\mathscr{R}$ est une relation de E vers F » si $\mathscr{R}$ est la donnée d'un triplet d'ensembles (E, F, U) tel que $U \subset E \times F$.*
- *On dit que « x est en relation avec y » si et seulement si $(x, y) \in U$ ce qui sera noté $x\mathscr{R}y$*
- *Au contraire si « x n'est pas en relation avec y » on écrira $x \not\mathscr{R} y$*
- *On représentera une relation $\mathscr{R} = (E, F, U)$ par un diagramme sagittal (ou diagramme fléché) pour cela :*
  - *on dessine les diagrammes de Venn des ensembles E et F*
  - *chaque couple $(x, y) \in U$ est représenté par une flèche allant de x à y*

Pour des raisons pratiques on utilisera le vocabulaire suivant pour désigner les différents ensembles associées à la définition de relation binaire :

Définition 1.2 (Lexique de la théorie des graphes)
Pour une relation $\mathscr{R}$, définie par le triplet (E, F, U), telle que $x\mathscr{R}y$ on dira :

- *E est l'ensemble de départ, F celui d'arrivé et U celui des arcs.*
- *x est le prédécesseur de y, on dit aussi l'origine de (x, y)*
- *l'ensemble des prédécesseurs de y est $\Gamma^-(y)$*
- *$d^-(y) = \text{Card}\Gamma^-(y)$ est le degré entrant en y*
- *le domaine de $\mathscr{R}$: $\mathscr{D}_{\mathscr{R}} = \{x \in E \mid \exists y \in F, \quad (x, y) \in U\}$*
- *y est le successeur de x, on dit aussi l'extrémité de (x, y)*
- *l'ensemble des successeurs de x est $\Gamma^+(x)$*
- *$d^+(x) = \text{Card}\Gamma^+(x)$ est le degré sortant de x*
- *l'image de $\mathscr{R}$: $\text{Im}_{\mathscr{R}} = \{y \in F \mid \exists x \in E, \quad (x, y) \in U\}$*

Il est très facile de retenir le sens de certaines de ces notions en pensant à la représentation graphique de la relation par un diagramme sagittal (ensembles de départ et d'arrivé, successeurs, prédécesseur) Mais d'autres sont moins faciles à retenir (domaine, image, degré). Il faut donc bien retenir ces définitions dès maintenant.

✎ **1.2 Exprimer ces différents ensembles pour la relation de la FIG.3**

- *Départ :* $E = \{Sport; Archi; Algo; \ldots; Maths\}$
- *Arrivée :* $F = \{Lundi; Mardi; Mercredi; \ldots; Dimanche\}$
- *Arcs :* $U = \{(Archi, Lundi); (Archi, Mardi); \ldots; (Maths, Vendredi)\}$
- *Domaine :* $\mathscr{D}_{\mathscr{R}} = E \setminus \{sport\}$
- *Image :* $\text{Im}_{\mathscr{R}} = F \setminus \{Samedi; Dimanche\}$
- $\Gamma^+(Maths) = \{Jeudi; Vendredi\} \Longrightarrow d^+(Maths) = 2$
- $\Gamma^-(Jeudi) = \{Maths; EC; Algo\} \Longrightarrow d^-(Jeudi) = 3$
- ...

On peut aussi rapprocher ce vocabulaire du vocabulaire utilisés pour les fonctions et applications qui sont en fait des cas particuliers de relations !

✎ **1.3 Cas des fonctions et applications** *reprendre les définitions du cours de théorie des ensembles concernant les fonctions et applications du point de vue des relations binaires :*

Fonctions *une fonction $f : E \longrightarrow F$ est une relation*

$$x\mathscr{R}y \Longleftrightarrow f(x) = y$$

Réciproquement, une relation $\mathscr{R}$ est une fonction si chaque élément de E à au plus un successeur

$$\forall x \in E, \quad \text{Card}(\Gamma^+(x)) \leq 1 \Longleftrightarrow \forall x \in E, \quad d^+(x) \leq 1$$

Applications *Une relation $\mathscr{R} : E \longrightarrow F$ est une application si et seulement si :*

- *$\mathscr{R}$ est une fonction*
- *Son domaine de définition est égal à E*

ce qui s'exprime en langage des relations binaires par

$$\forall x \in E, \quad \text{Card}(\Gamma^+(x)) = 1 \Longleftrightarrow \forall x \in E, \quad d^+(x) = 1$$

Application injective surjective, bijective *Soit $f : E \longrightarrow F$ une application, on dit que f est*

- *injective si chaque élément de F à au plus un prédécesseur*

$$\forall y \in F, \quad \text{Card}(\Gamma^-(y)) \leq 1 \Longleftrightarrow \forall y \in F, \quad d^-(y) \leq 1$$

- *surjective si chaque élément de F à au moins un prédécesseur*

$$\forall y \in F, \quad \text{Card}(\Gamma^-(y)) \geq 1 \Longleftrightarrow \forall y \in F, \quad d^-(y) \geq 1$$

- *bijective si elle est injective et surjective*

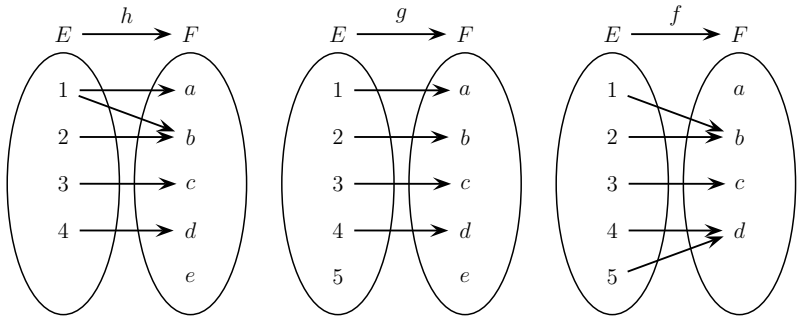
$$\forall y \in F, \quad \text{Card}(\Gamma^-(y)) = 1 \Longleftrightarrow \forall y \in F, \quad d^-(y) = 1$$

Ces définitions ont une grande importance en base de données, elles sont directement liées aux cardinalités qui apparaissent dans un MCD. Elles permettent d'expliquer pourquoi :

- Une relation fonctionnelle qui apparaît dans un MCD n'aura pas de table propre
- Une relation bijective ne devrait jamais apparaître dans un MCD

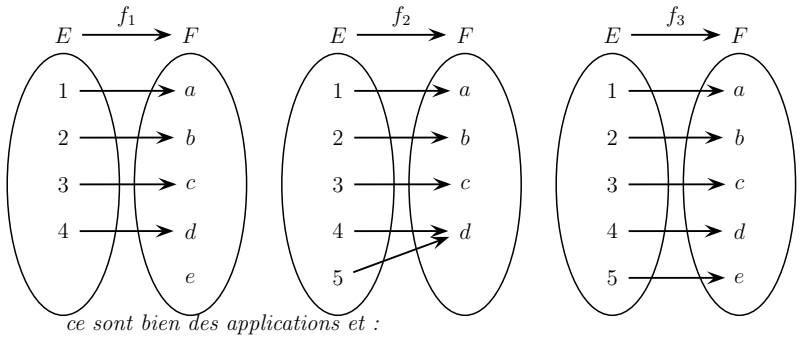
✎ **1.4 Reconnaître à partir des diagramme sagittaux les définitions précédentes :**

reconnaître les fonctions et les applications des autres relations *:*



- *g et f sont des fonctions*
- *f est une application mais pas g (car $d^+(5) = 0$ ou encore $\mathscr{D}_g = E \setminus \{5\} \neq E$)*
- *et h n'est pas une fonction (car $d^+(5) = 2$) donc pas une application*

reconnaître injectivité, surjectivité et bijectivité *:*



ce sont bien des applications et :

- *f_1 est injective mais pas surjective (à cause de $d^-(e) = 0$)*
- *f_2 est surjective mais pas injective (à cause de $d^-(d) = 2$)*
- *seule f_3 est bijective*

On retrouve sur ces exemples les résultats du théorème suivant :

Théorème 1.3
Soient E, F des ensembles finis et $f : E \longrightarrow F$ une application alors

- *si f est injective alors* $\text{Card}(E) \leq \text{Card}(F)$
- *si f est surjective alors* $\text{Card}(E) \geq \text{Card}(F)$
- *si f est bijective alors* $\text{Card}(E) = \text{Card}(F)$

Le diagramme sagittal permet de détecter de nombreuses propriétés d'une relation binaire, à condition qu'il n'y ait pas trop d'arc ni d'éléments. Pour pouvoir analyser les propriétés de graphes de grandes tailles nous avons besoin d'une représentation qui permette de faire des calculs : une matrice.

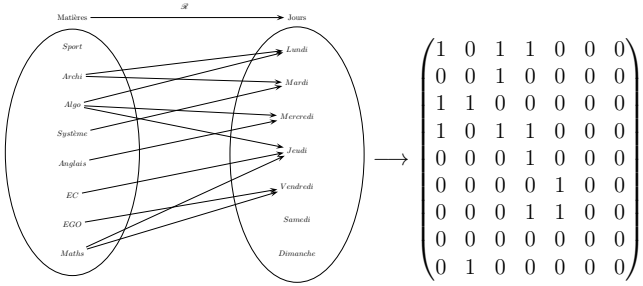
Définition 1.4 (matrice d'adjacence)
Soient $E = \{x_1; x_2; \dots; x_p\}$, $F = \{y_1; y_2; \dots; y_n\}$ et $\mathscr{R} = (E, F, U)$ une relation alors on appelle matrice d'adjacence de $\mathscr{R}$ la matrice booléenne $M_{\mathscr{R}} \in \mathscr{M}_{p,n}(\mathbb{B})$ telle que

$$M_{\mathscr{R}} = (m_{i,j}) \quad avec \quad m_{i,j} = \begin{cases} 1 & si \quad (x_i, y_j) \in U \\ 0 & sinon \end{cases}$$

Pour pouvoir écrire la matrice d'adjacence d'une relation il faut avoir choisi un ordre pour les éléments des ensembles E et F (il sont numérotés $x_1, x_2, \dots$ pour E et $y_1, y_2, \dots$ pour F). Ce choix est arbitraire, mais il n'est pas indiqué dans la matrice d'adjacence ! **On prendra donc (sauf mention contraire) l'ordre lexicographique pour ordonner les éléments de E et F.** Une fois qu'on a ordonné les éléments des ensembles E et F chaque relation est représentée par une matrice et chaque matrice représente une relation

1.5 Représentation d'une relation à l'aide d'une Matrice d'adjacence

Comme il n'y a pas à priori d'ordre sur les éléments d'un ensemble il faut souvent faire attention pour remplir la matrice d'adjacence à partir d'un diagramme sagittal où les éléments ne sont pas forcément classé dans l'ordre lexicographique[1]*. Ici on choisit l'ordre lexicographique pour l'ensemble des matières mais pas pour l'ensemble des jours (où il y a un ordre plus naturel) ce qui donne :*



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

Pour mieux comprendre la matrice d'adjacence on peut y faire apparaître les éléments des ensembles E et F

1. ça ne donnerait pas forcément un diagramme très lisible

	lundi	*mardi*	*mercredi*	*jeudi*	*vendredi*	*samedi*	*dimanche*
Algo	1	0	1	1	0	0	0
Anglais	0	0	1	0	0	0	0
Archi	1	1	0	0	0	0	0
Algo	1	0	1	1	0	0	0
EC	0	0	0	1	0	0	0
EGO	0	0	0	0	1	0	0
Maths	0	0	0	1	1	0	0
Sport	0	0	0	0	0	0	0
Système	0	1	0	0	0	0	0

On se rappellera que dans la matrice d'adjacence :

- les **lignes** correspondent aux éléments de l'**ensemble de départ**
- les **colonnes** correspondent aux éléments de l'**ensemble de arrivée**

Le défaut de la matrice d'adjacence est quelle contient beaucoup de 0. D'un point de vue informatique cela représente un gaspillage de place mémoire. C'est pourquoi on représente parfois ces matrices sous forme de « **matrice creuse** » , c'est à dire en donnant seulement la position de chaque coefficient non-nul de la matrice (ainsi que sa taille). C'est la représentation qui est utilisée en base de données pour représenter la table associée à un TA. Dans le cas de la relation représentée FIG.3 la matrice d'adjacence sera représentée comme ci-contre :

i	j	m_{ij}
1	1	1
1	3	1
1	4	1
2	3	1
3	1	1
3	2	1
4	1	1
4	3	1
4	4	1
5	4	1
6	5	1
7	4	1
7	5	1
9	2	1

La matrice d'adjacence permet de faire de nombreux calculs, comme par exemples compte le nombre de relations entre deux ensembles.

Théorème 1.5 (Nombres de relations entre 2 ensembles)
Si E et F sont des ensembles finis alors le nombre de relations de E vers F est

$$2^{\text{Card}(E\times F)} = 2^{\text{Card}(E)\times\text{Card}(F)}$$

Preuve : Il suffit de compter le nombre de Matrices d'adjacences :

- la matrice d'adjacence est de taille $\text{Card}(E) \times \text{Card}(F)$
- Chaque case peut être remplie de 2 manières 0 ou 1
- On a donc au total $\underbrace{2 \times 2 \times \cdots \times 2}_{\text{Card}(E)\times\text{Card}(F) \text{ répétitions}} = 2^{\text{Card}(E)\times\text{Card}(F)}$ possibilités

□

Mais surtout la matrice d'adjacence permet de calculer de nouvelles relations à partir de relations connues. Commençons par la composition des relations.

Définition 1.6 (Composition de relations) *Soient $\mathscr{R}_1 = (E, F, U_1)$ et $\mathscr{R}_2 = (F, G, U_2)$ 2 relations alors on appelle $\mathscr{T} = \mathscr{R}_2 \circ \mathscr{R}_1 = (E, G, U)$ la relation dont les arcs sont :*

$$U = \{(x, z) \in E \times G \mid \exists y \in F, \quad (x, y) \in U_1 \quad et \quad (y, z) \in U_2\}$$

✎ **1.6 composition de deux relations** *$\mathscr{T} = \mathscr{R}_2 \circ \mathscr{R}_1$ il y a un arc joignant un élément de E et un élément de G dans le diagramme de la relation $\mathscr{T}$ si on trouve dans les diagrammes de $\mathscr{R}_1$ et $\mathscr{R}_2$ un chemin entre ces éléments passant par un élément de F :*

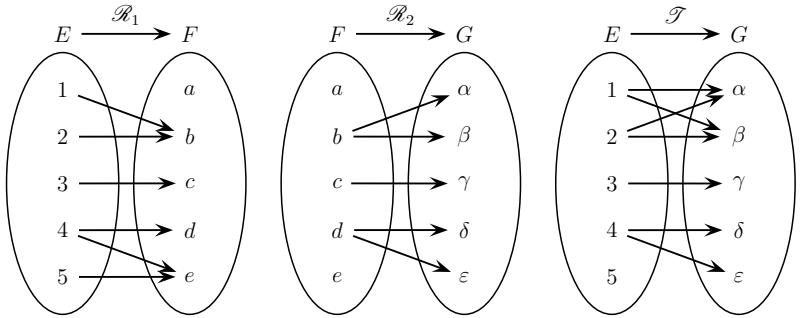


FIGURE 4 – Composition de relations

La composition de deux relations correspond au produit matriciel des matrices d'adjacence.

Théorème 1.7
Si E, F, G sont des ensembles finis alors la matrice d'adjacence de $\mathscr{T} = \mathscr{R}_2 \circ \mathscr{R}_1$ est donnée par le produit matriciel suivant :

$$M_{\mathscr{T}} = M_{\mathscr{R}_2 \circ \mathscr{R}_1} = M_{\mathscr{R}_1} \times M_{\mathscr{R}_2}$$

Ce produit matriciel est effectué avec les opérations de l'algèbre de Boole binaire !

Preuve : On note

$$E = \{x_1; \ldots; x_p\}; \quad F = \{y_1; \ldots; y_l\}; \quad G = \{z_1; \ldots; z_n\}$$

On peut déjà remarquer que la relation composée $\mathscr{T}$ va de E vers G

$$\mathscr{T} = \mathscr{R}_2 \circ \mathscr{R}_1 : E \xrightarrow{\mathscr{R}_1} F \xrightarrow{\mathscr{R}_2} G$$

donc sa matrice doit être de taille $p \times n$ ce qui correspond bien à la taille du résultat du produit matriciel $M_{\mathscr{R}_1} \times M_{\mathscr{R}_2}$. Ensuite en reprenant la formule du produit matriciel :

$$M_{\mathscr{T}}(i, j) = \sum_{k=1}^{l} M_{\mathscr{R}_1}(i, k) \times M_{\mathscr{R}_2}(k, l)$$

la somme et le produit sont faits dans l'algèbre de Boole binaire $\mathbb{B}$ donc il suffit qu'un seul des termes du produit soit non-nul (égal à 1) pour que $M_{\mathscr{T}}(i, j) = 1$ cela veut dire qu'il existe k tel que $M_{\mathscr{R}_1}(i, k) = 1$ et $M_{\mathscr{R}_2}(k, l) = 1$. Cela revient à dire qu'il existe un arcs (x_i, y_k) dans $\mathscr{R}_1$ et un autre (y_k, z_j) dans $\mathscr{R}_2$, il y a donc bien un arc (x_i, z_j) dans $\mathscr{T}$ □

Attention à l'ordre des termes dans le produit matriciel, la matrice d'adjacence de $\mathscr{R}_2 \circ \mathscr{R}_1$ est $M_{\mathscr{R}_1} \times M_{\mathscr{R}_2}$ et pas $M_{\mathscr{R}_2} \times M_{\mathscr{R}_1}$!

✎ **1.7 Vérifier que la matrice $M_{\mathscr{R}_1} \times M_{\mathscr{R}_2}$ est bien la matrice d'adjacence de $M_{\mathscr{T}}$ FIG.4**

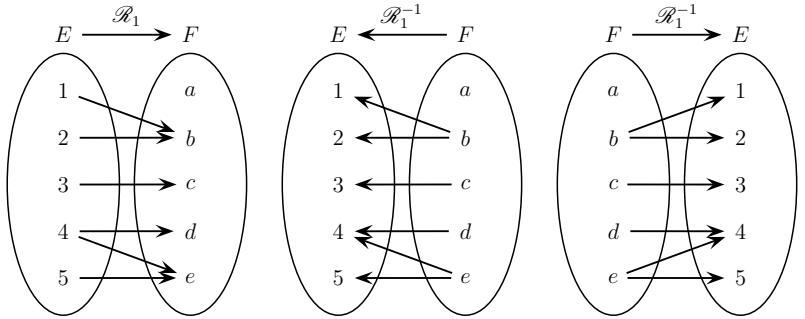
$$M_{\mathscr{R}_1} = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}, \quad M_{\mathscr{R}_2} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix} M_{\mathscr{T}} = \begin{pmatrix} 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

La deuxième formule matriciel qu'il faut connaître est celle sur la réciproque d'une relation.

Définition 1.8 (Réciproque d'une relation) *Soit $\mathscr{R} = (E, F, U)$ une relation alors on appelle la relation réciproque de $\mathscr{R}$, notée $\mathscr{R}^{-1} = (F, E, U')$, la relation dont les arcs sont :*

$$U' = \{(y, x) \in F \times E \mid \quad (x, y) \in U\}$$

✎ **1.8 Calculer la réciproque de la relation $\mathscr{R}$**
calculer la réciproque d'une relation revient à inverser le sens des flèches sur le diagramme sagittal :



La encore, on a une formule qui permet de calculer la matrice d'adjacence de la réciproque à partir de la matrice d'adjacence de la relation de départ.

Théorème 1.9 *Si E, F sont des ensembles finis alors la matrice d'adjacence de $\mathscr{R}^{-1}$ est la transposée de la matrice d'adjacence de $\mathscr{R}$:*

$$M_{\mathscr{R}^{-1}} = {}^t M_{\mathscr{R}}$$

Preuve : On a dit que dans la matrice d'adjacence les lignes correspondent aux éléments de l'ensemble de départ et les colonnes aux éléments de l'ensemble d'arrivé. Pour échanger les rôles des ensembles de départ et d'arrivé il suffit donc d'échanger les rôles des lignes et des colonnes de la matrice d'adjacence ce qui revient à transposer cette matrice. □

✎ **1.9 Calculer la matrice d'adjacence de la réciproque de la relation $\mathscr{R}_1$**

$$M_{\mathscr{R}_1} = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \Longrightarrow M_{\mathscr{R}_1^{-1}} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 \end{pmatrix}$$

Attention, la matrice d'adjacence de la réciproque d'une relation n'est pas l'inverse de la matrice d'adjacence de la relation de départ :

$$M_{\mathscr{R}^{-1}} \neq (M_{\mathscr{R}})^{-1}$$

D'ailleurs l'inverse d'une matrice d'adjacence n'existe pas forcément alors que la réciproque existe toujours.

Les produits matriciels qui apparaissent dans ce cours sont fait en utilisant les opérations $+$ et $\times$ de l'algèbre de Boole binaire $\mathbb{B}$ ce qui ne donne pas le même résultat que s'ils sont fait avec les opérations $+$ et $\times$ de $\mathbb{R}$. Par exemple dans $\mathscr{M}_5(\mathbb{B})$ on a

$$M_{\mathscr{S}} \times M_{\mathscr{R}} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 \end{pmatrix} \times \begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 \end{pmatrix}$$

alors que dans $\mathscr{M}_5(\mathbb{R})$ on a

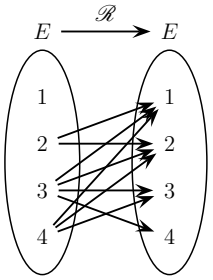
$$M_{\mathscr{S}} \times M_{\mathscr{R}} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 \end{pmatrix} \times \begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 2 \end{pmatrix}$$

quand on fait les calculs dans $\mathbb{R}$ le nombre obtenu en position (i, j) dans la matrices donne le nombre de manières de construire l'arc (i, j) dans la relation $\mathscr{R} \circ \mathscr{S}$.

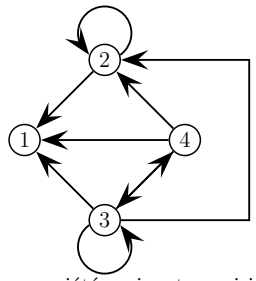
1.2 Relation dans un ensemble

Les définitions de la partie précédente s'appliquent bien entendu aussi dans le cas où l'ensemble de départ et l'ensemble d'arrivé d'une relation sont les mêmes, dans ce cas on parle de relation dans un ensemble que l'on peut représenter par une matrice d'adjacence ou par différents diagrammes sagittaux.

✎ **1.10 Diagramme sagittal d'une relation $\mathscr{R}$ avec un ou deux ensembles**



$$M_{\mathscr{R}} = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix}$$



dans ce cas on s'intéresse souvent aux quatre propriétés suivantes qui jouent un rôle très important en théorie des graphes.

Définition 1.10 *Soit $\mathscr{R}$ une relation dans un ensemble E alors on dit que $\mathscr{R}$ est*

- *Réflexive si $\forall x \in E \quad x \mathscr{R} x$*
- *Symétrique si $\forall x, y \in E, \quad x \mathscr{R} y \Longleftrightarrow y \mathscr{R} x$*
- *Anti-symétrique si $\forall x, y \in E, \quad x \mathscr{R} y \quad et \quad y \mathscr{R} x \Longrightarrow x = y$*
- *Transitive si $\forall x, y, z \in E, \quad x \mathscr{R} y \quad et \quad y \mathscr{R} z \Longrightarrow x \mathscr{R} z$ on dit que $\mathscr{R}$ est*

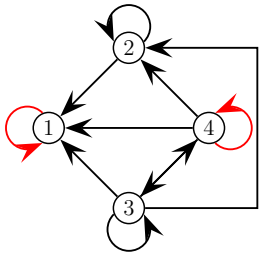
Il est assez facile de se représenter ces définitions sur un diagramme sagittal à un ensemble de la relation.

Proposition 1.11 (caractérisation par le diagramme sagittal)
Soit $\mathscr{R}$ une relation dans un ensemble E alors $\mathscr{R}$ est

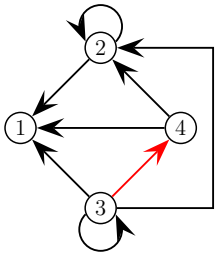
- *Réflexive si tout élément de S possède une boucle,*
- *Symétrique si tous les arcs sont à double sens,*
- *Anti-symétrique si aucun arc n'est à double sens,*
- *Transitive si pour chaque couple d'arcs adjacents le "raccourci" est aussi un arc du graphe.*

Pour l'anti-symétrie et la symétrie on ne tient pas compte des boucles, c'est pour cette raison que Anti-symétrique n'est pas la négation logique de symétrique. Il existe donc des relations qui sont anti-symétrique et symétrique ou, au contraire ni anti-symétrique ni symétrique !

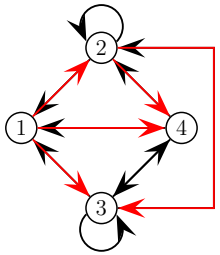
✎ **1.11 Modifier la relation** $\mathscr{R}$ *pour qu'elle soit, successivement, Réflexive, Anti-symétrique, Symétrique*



Réflexif



Anti-symétrique



Symétrique

On pourrait croire que la relation $\mathscr{R}$ est déjà transitive, pourtant il n'en est rien, mais trouver ce qu'il faut ajouter au graphe pour la rendre transitive demande beaucoup plus de calculs ... C'est là que la matrice d'adjacence va nous aider !

Proposition 1.12 (caractérisation par la matrice d'adjacence)
Soit $\mathscr{R}$ une relation dans un ensemble E de matrice d'adjacence M alors $\mathscr{R}$ est :

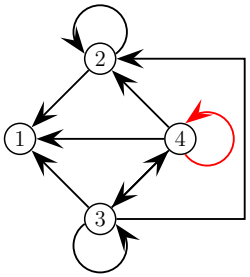
- *Réflexive si $M = M + Id_n$ (Id_n = matrice identité de taille n)*
- *Symétrique si $M = {}^tM$ (^tM = transposée de la matrice M)*
- *Anti-symétrique $M * {}^tM + Id_n = Id_n$ ($*$ = produit terme à terme des matrices)*
- *Transitive $M^2 + M = M$ ($M^2 = M \times M$ produit matriciel)*

Preuve :

- $M+Id_n$ ajoute des « 1 » sur la diagonale qui correspondent donc à des boucles sur chaque sommet, donc si $M = M + Id_n$ on a déjà toutes les boucles.
- $M = {}^tM$ signifie que $M_{ij} = 1 \Longrightarrow M_{ji} = 1$ donc chaque arc de i vers j on doit avoir un arc de j vers i
- $M * {}^tM + Id_n = Id_n$ signifie que $M * {}^tM$ à tous ces coefficients nuls sauf sur la diagonale éventuellement (puis qu'on rajoute Id_n). Donc si $M_{ij} = 1 \Longrightarrow M_{ji} = 0$ pour que le produit terme à terme donne un zéro dans la case (i, j) de la matrice (et aussi dans la case (j, i))
- Soit $\mathscr{R}$ la relation associée au graphe G et M sa matrice d'adjacence alors M^2 est la matrice d'adjacence de la relation $\mathscr{R} \circ \mathscr{R}$. Les « 1 » de cette matrice correspondent donc à des arcs composés de deux arcs adjacents du graphe G. Si ces arcs sont déjà dans G, les « 1 » correspondant sont déjà dans la matrice M et on a bien que $M^2 + M = M$.

□

✎ **1.12 Modifier la relation $\mathscr{R}$ pour qu'elle soit transitive**



$$car\ M^2 + M = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & \mathbf{1} \end{pmatrix} \neq \begin{pmatrix} 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix}$$

On verra plus loin comment rendre un graphe transitif en ajoutant des arcs (cf. fermeture transitive).

Ces propriétés sont à la base de relations que vous utilisez déjà depuis vos premières études de mathématiques : les relations d'équivalences et les relations d'ordre.

Définition 1.13 (relation d'équivalence)
Soit $\mathscr{R} = (E, E, A)$ une relation dans un ensemble E alors $\mathscr{R}$ est une « relation d'équivalence » si et seulement si elle est

- *Réflexive,*
- *Symétrique,*
- *Transitive.*

Dans ce cas on appelle « classe d'équivalence de x » le sous ensemble de E :

$$\mathrm{Cl}(x) = \{y \in E \mid x \mathscr{R} y\}$$

L'ensemble des classes d'équivalences de $\mathscr{R}$ est noté $E/\mathscr{R}$ et appelé « ensemble quotient de E par $\mathscr{R}$ ».

✎ **1.13 Exemples de relations d'équivalences**

- *l'égalité dans un ensemble E car*
  - $\forall x \in E,\ x = x$
  - $\forall x, y \in E,\ x = y \Longleftrightarrow y = x$
  - $\forall x, y, z \in E,\ x = y$ *et* $y = z \Longleftrightarrow x = z$
- *pour $x_0 \in I$ un intervalle de $\mathbb{R}$, la relation*

  $$f \underset{x_0}{\sim} g \quad \textit{dans l'ensemble} \quad E = \{f : I \to \mathbb{R} | f \textit{ application}\}$$

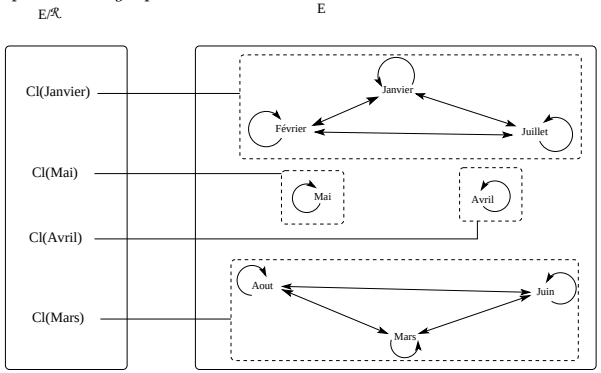
  est une relation l'équivalence car
  - $\forall f \in E,\ f \underset{x_0}{\sim} f$
  - $\forall f, g \in E,\ f \underset{x_0}{\sim} g \Longleftrightarrow g \underset{x_0}{\sim} f$
  - $\forall f, g, h \in E,\ f \underset{x_0}{\sim} g$ *et* $g \underset{x_0}{\sim} h \Longleftrightarrow f \underset{x_0}{\sim} h$
- *Sur l'ensemble $E = \{$Janvier ; Février ; Mars ;Avril ; Mai ; Juin ; Juillet ; Août$\}$ on considère la relation :*

  $mois_1 \mathscr{R} mois_2 \Longleftrightarrow mois_1$ *et* $mois_2$ *s'écrivent avec le même nombre de lettres*

  en effet si $mois_1$, $mois_2$ *et* $mois_3$ *s'écrivent avec* n_1, n_2 *et* n_3 *lettres on a*

— $mois_1 \mathscr{R} mois_1$ *car* $n_1 = n_1$
— *si* $mois_1 \mathscr{R} mois_2$ *donc* $n_1 = n_2$ *donc* $n_2 = n_1$ *donc* $mois_2 \mathscr{R} mois_1$ *et inversement*
— *si* $mois_1 \mathscr{R} mois_2$ *et* $mois_2 \mathscr{R} mois_3$ *alors* $n_1 = n_2 = n_3$ *donc* $n_1 = n_3$ *et* $mois_1 \mathscr{R} mois_3$

donc c'est bien une relation d'équivalence. Son diagramme sagittal est représenté ci-dessous avec les classes d'équivalences et l'ensemble quotient $E/\mathscr{R}$. Sur cet exemple on comprend facilement que les classe d'équivalence représentent en fait des parties du graphe isolées les unes des autres.



Proposition 1.14 *Si $\mathscr{R}$ est une relation d'équivalence sur E alors*
- $\forall x \in E, \quad \mathrm{Cl}(x) \neq \emptyset \quad (\ car\ x \in \mathrm{Cl}(x))$.
- $\forall (x,y) \in E^2, \quad x \mathscr{R} y \iff \mathrm{Cl}(x) = \mathrm{Cl}(y)$.
- $\forall x \in E, \quad \bigcup_{x \in E} \mathrm{Cl}(x) = E$.

L'ensemble des classes d'équivalences de $\mathscr{R}$ (i.e. $E/\mathscr{R}$) forment une partition de E.

Il ne faut pas confondre les relations d'équivalence avec les relations d'ordre.

Définition 1.15 (relation d'ordre)
Soit $\mathscr{R} = (E, E, A)$ une relation dans un ensemble E alors $\mathscr{R}$ est une « relation d'ordre » si et seulement si elle est
- *Réflexive,*
- *Anti-symétrique,*
- *Transitive.*

On dit qu'une relation $\mathscr{R}$ est une relation d'ordre total sur S si en plus :

$$\forall (x,y) \in E^2, \quad x \mathscr{R} y \quad ou \quad y \mathscr{R} x$$

dans le cas contraire on parle de relation d'ordre partiel.

On représente souvent les relations d'ordre par un diagramme plus simple que le diagramme sagittal habituel.

Proposition 1.16 (Diagramme de Hasse) *Soit $\mathscr{R} = (E, E, A)$ une relation d'ordre, on peut représenter cette relation par un « diagramme de Hasse » qui est un diagramme sagittal dans lequel*
- *on positionne les sommets du plus petit au plus grand (au sens de $\mathscr{R}$)*
- *on omet toutes les boucles (sous-entendues par réflexivité)*
- *on ne trace pas les raccourcis (sous-entendus par transitivité)*
- *on ne met pas de sens aux arcs (sous-entendus par anti-symétrie) par convention la relation va de bas en haut ou de gauche à droite*

1.14 exemples de relations d'ordre

- *la relation $\leq$ sur un ensemble de nombres $E = \mathbb{N}, \mathbb{Z}, \mathbb{Q}, \mathbb{R}$*
  — $\forall x \in E,\ x \leq x$
  — $\forall x, y \in E,\ x \leq y$ *et* $y \leq x \implies x = y$
  — $\forall x, y, z \in E,\ x \leq y$ *et* $y \leq z \implies x \leq z$

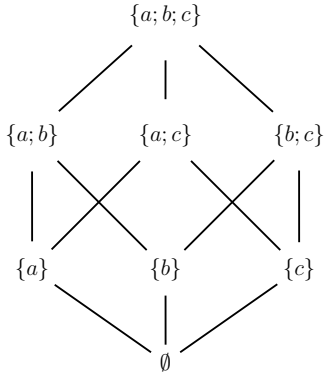
  son diagramme de Hasse (sur $\mathbb{N}$ par exemple) montre que c'est une relation d'ordre total :

  $0 \text{ —— } 1 \text{ —— } 2 \text{ —— } 3 \text{ —— } \cdots$

- *La relation $\subset$ sur les parties d'un ensemble $\mathscr{P}(E)$*

  — $\forall A \subset E,\ A \subset A$
  — $\forall A, B \subset E,\ A \subset B$ *et* $B \subset A$ $\implies A = B$
  — $\forall A, B, D \subset E,$ $A \subset B$ *et* $B \subset D \implies A \subset D$

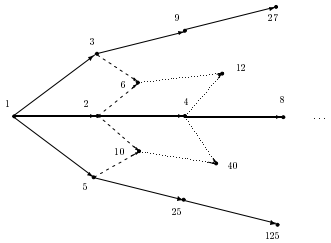
  son diagramme de Hasse (sur $\mathscr{P}(\{a,b,c\})$ par exemple) montre que ce n'est pas une relation d'ordre total (en général, pourtant il y a un plus grand élément ($\{a;b;c\}$) et un plus petit élément ($\emptyset$) dans $\mathscr{P}(\{a,b,c\})$ pour cette relation.



- *La relation $a|b \iff$ « a divise b » sur $\mathbb{N}^*$*

  — $\forall a \in \mathbb{N}^*,\ a|a$
  — $\forall a, b \in \mathbb{N}^*,\ a|b$ *et* $b|a \implies a = b$
  — $\forall a, b, c \in \mathbb{N}^*,\ a|b$ *et* $b|c \implies a|c$

  son diagramme de Hasse est beaucoup plus compliqué que les deux relations précédentes, il y a un plus petit élément (1) mais pas de plus grand élément :



1.3 Graphes

Dans la suite nous appellerons “graphe orienté ” une relation dans un ensemble.

Définition 1.17 (Graphe orienté) *Un graphe orienté G est la donnée d'un couple d'ensembles (S, A) tels que $A \subset S \times S$ de telle sorte que G peut être vu comme la relation binaire $\mathscr{R} = (S, S, A)$ entre l'ensemble S et lui même. On appellera aussi :*

sommets du graphe *les éléments $x \in S$*

arcs du graphe *les éléments $(x, y) \in A$*

degré d'un sommet *somme des degrés entrants et sortant $d(x) = d^+(x) + d^-(x)$*

ordre du graphe *le nombre de sommets $n = \mathrm{Card}(S)$*

taille du graphe *le nombre d'arcs $m = \mathrm{Card}(A)$*

boucle *tout arc de la forme (x, x), c'est à dire dont l'origine est aussi son extrémité*

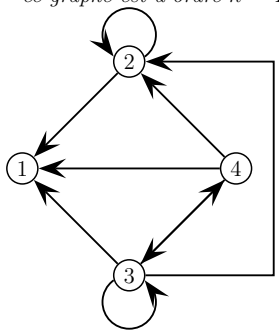
arcs adjacents *deux arcs de la forme (x, y) et (y, z), c'est à dire dont l'origine de l'un est l'extrémité de l'autre.*

Comme pour les relations binaires, G possède une matrice d'adjacence M_G qui est une matrice carrée $M_G \in \mathscr{M}_n(\mathbb{B})$ (où $n = \mathrm{Card}(S)$) et peux être représenté par un diagramme sagittal à un seul ensemble.

✎ **1.15 Construire le graphe $G = (S, A)$ suivant**

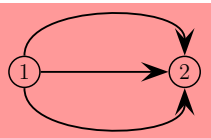
$S = \{1; 2; 3; 4\}$ $A = \{(2,1); (2,2); (3,1); (3,2); (3,3); (3,4); (4,1); (4,2); (4,3)\}$

ce graphe est d'ordre $n = 4$ et de taille $m = 9$



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

Dans ce cours on ne considérera pas de graphes avec des liens multiples, c'est à dire avec plusieurs arcs (x, y) différents ! Ce type de graphes est appelé « graphe multiple »



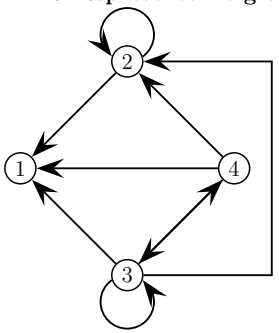
Définition 1.18 (listes d'adjacence) *Soit $G = (S, A)$ un graphe orienté d'ordre n et de taille m dont les sommets $x_1; x_2; \ldots x_n$ sont ordonnés. Le graphe G peut être représenté par des listes d'adjacence (LS, TS) qui sont définies par :*

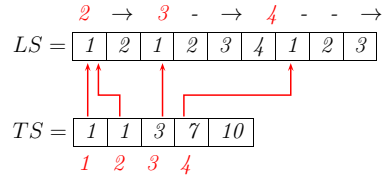
- *LS = liste de longueur m appelée « liste des successeurs », elle contient les successeurs du sommet 1 (rangé dans l'ordre croissant) puis du sommet 2 ...et si un sommet n'a pas de successeur on passe au sommet suivant.*
- *TS = liste de longueur $n+1$ appelée « liste des têtes successeurs » qui indique la position du premier successeur de chaque sommet dans LS*

la liste TS est définie comme suit :

- $TS(1) = 1$
- *pour $x \in S$*
  - *si x à des successeurs alors $TS(x)$ = numéro de la case de LS du premier successeur de x*
  - *sinon $TS(x) = TS(x+1)$*
- $TS(n+1) = m+1$

✎ **1.16 Représenter le graphe G par des listes d'adjacence**





La liste des « tête successeurs » est une liste de pointeurs qui permettent de faire apparaître la liste des successeurs de chaque sommet dans »liste des successeurs ». Par exemple, en faisant apparaître ces liens de TS vers LS (rouge ci-dessus) on retrouve facilement la liste des arcs (et toute la structure du graphe) à partir des listes d'adjacence :

- *1 n'a pas de successeurs*
- *les successeurs de 2 sont 1 et 2 ce qui donne les arcs $(2,1)$ et $(2,2)$*
- *les successeurs de 3 sont $1, 2, 3$ et 4 ce qui donne les arcs $(3,1), (3,2), (3,3), (3,4)$*
- *les successeurs de 4 sont $1, 2$ et 3 ce qui donne les arcs $(4,1), (4,2)$ et $(4,3)$*

Les listes d'adjacences occupent une place mémoire de taille $n + m + 1$ c'est le minimum d'informations pour représenter un graphe comparé à la matrice d'adjacence occupe une place n^2 et la liste des arcs occupe une place $2m$

Le fait que l'ensemble de départ et d'arrivé de la relation associée au graphe soient les mêmes impose une contrainte, sur les degrés des sommets, appelée « lemme des poignées de mains ».

Théorème 1.19 (Lemme des poignées de mains) *Soit $G = (S, A)$ un graphe de taille m alors les sommes des degrés entrants et sortants des sommets de G sont égales au nombre d'arcs*

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

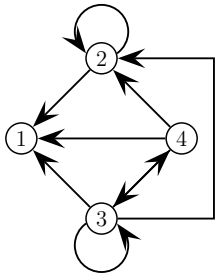
en conséquence la somme des degrés est égale au double de nombre d'arcs

$$\sum_{x \in S} d(x) = \sum_{x \in S} d^+(x) + \sum_{x \in S} d^-(x) = 2m$$

Preuve : chaque arc (x, y) compte deux fois dans la somme des degrés : une fois dans $d^+(x)$ et une fois dans $d^-(y)$, d'où le résultat :

$$\sum_{x \in S} d^+(x) = \sum_{x \in S} d^-(x) = \sum_{i=1}^{m} 1 = m \Longrightarrow \sum_{x \in S} d(x) = \sum_{x \in S} d^+(x) + \sum_{x \in S} d^-(x) = 2m \ \square$$

✎ **1.17 Vérifier le lemme des poignées de mains sur le graphe G de taille 9**



s	$d^+(s)$	$d^-(s)$	$d(s)$
1	*0*	*3*	*3*
2	*2*	*3*	*5*
3	*4*	*2*	*6*
4	*3*	*1*	*4*
$\sum$	*9*	*9*	*18*

1.4 Autres types de graphes

On rencontrera d'autres définitions de graphes, qui sont des cas particulier de celle vue dans la partie précédente, qui permettent de modéliser des problèmes concrets avec des graphes plus simples. Le premier exemple est celui des arbres

Définition 1.20 (arbre) *Un arbre est un graphe orienté $G = (S, A)$ tel que :*

- *un seul sommet de G n'a pas de prédécesseur, c'est la racine de l'arbre,*
- *tous les autres sommets ont exactement 1 prédécesseur*

On appellera

« **racine de l'arbre** » *le seul sommet de G qui n'a pas de prédécesseur,*

« **feuilles de l'arbre** » *les sommets qui n'ont pas de successeur,*

« **nœuds de l'arbre** » *tous les autres sommets,*

« **branche de l'arbre** » *tout chemin de la racine vers une feuille,*

« **descendant de x** » *les successeurs de x,*

« **ascendant de x** » *le prédécesseur de x,*

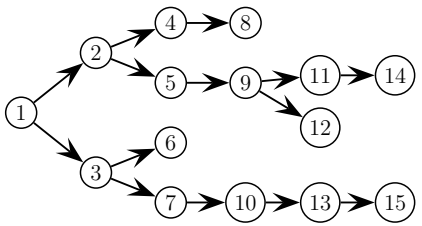
Lorsque chaque sommet à au plus 2 successeurs on parle aussi d'arbre binaire.

Un graphe d'ordre n qui est un arbre peut être représenté avec seulement n places mémoire ! C'est beaucoup moins que pour un graphe quelconque, qui nécessite une place mémoire $n + m + 1$ en utilisant les listes d'adjacence ou n^2 cases mémoire avec une matrice d'adjacence.

Proposition 1.21 (liste des prédécesseurs) *Un arbre à n sommets peut être définit par une liste de n éléments, appelé* **liste des prédécesseurs**, *qui contient le prédécesseur de chaque sommet (où 0 pour la racine de l'arbre) :*

$$\forall y \in S, \quad pred(y) = \begin{cases} \emptyset & si \quad y \text{ est la racine de } G \\ x & tel\ que \quad (x, y) \in A \text{ sinon} \end{cases}$$

✎ **1.18 Représenter l'arbre ci-dessous par une liste de prédécesseurs**



- $Pred =$ | *0* | *1* | *1* | *2* | *2* | *3* | *3* | *4* | *5* | *7* | *9* | *9* | *10* | *11* | *13* |
- *la racine du graphe est le sommet* 1
- *les feuilles sont les sommets* $8, 14, 12, 6$ *et* 15
- *une branche de l'arbre* $C = (1, 2, 5, 9, 12)$ *(chemin de la racine jusqu'à* 12*)*

Pour traiter le cas de plusieurs sommets sans prédécesseurs (qu'on rencontrera lors du parcours d'un graphe) on parlera plus généralement de forêt.

Définition 1.22 (forêt) *Un graphe $G = (S, A)$ composé de plusieurs sous-graphes qui sont tous des arbres est appelée* **forêt**. *Comme pour un arbre la structure d'une forêt peut être entièrement reconstruite par la liste des prédécesseurs, il y aura juste plusieurs sommets $x \in S$ tels que $pred(x) = 0$.*

On verra aussi des graphes où le sens des arcs n'a pas de signification, on parle dans ce cas de graphe non-orienté.

Définition 1.23 (graphe non-orienté) *Un graphe non-orienté G est un couple d'ensembles (S, A) où A est un ensemble de parties de S ayant 1 ou 2 éléments :*

$$A \subset \{B \in \mathscr{P}(S) | \text{Card}(B) = 1 \quad ou \quad 2\}$$

On appellera

- *S l'ensemble des sommets de G*
- *A l'ensemble des arêtes de G*

La relation $\mathscr{R}$ associée au graphe G est la relation de S dans S contenant tous les arcs (x, y) correspondant à chaque arête $\{x; y\}$. Mathématiquement cela correspond à une relation $\mathscr{R}$ définie par :

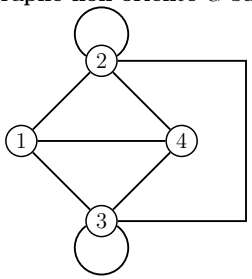
$$\mathscr{R} = (S, S, \widetilde{A}) \quad avec \quad \widetilde{A} = \{(x, y) \in S^2 | \{x; y\} \in A\}$$

ou encore

$$x \mathscr{R} y \Longleftrightarrow \{x; y\} \in A$$

On représentera un graphe non-orienté par un diagramme sagittal où l'on ne met pas de sens aux flèches et par la matrice d'adjacence de $\mathscr{R}$.

✎ **1.19 Représenter la matrice d'adjacence et l'ensemble des arêtes du graphe non-orienté G suivant**



$A = \{\{1; 2\}; \{1; 3\}; \{2; 3\}; \{1; 4\}; \{2; 4\}; \{3; 4\}; \{2\}; \{3\}\}$

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

La relation associée à un graphe non-orienté est forcément symétrique, donc sa matrice d'adjacence sera forcément symétrique.

Dans un graphe non-orienté les notions de successeurs/prédécesseur ou de degrés entrant/sortant n'ont plus de signification. Cela crée un piège au niveau du lemme des poignées de mains.

Dans un graphe non-orienté $G = (S, A)$ le degré d'un sommet est égal au nombre d'arêtes qu'on peut compter sur le diagramme saggital, en particulier

Chaque boucle compte **deux fois** *dans le degré du sommet considéré !*

de telle sorte que le lemme des poignées de mains reste vrai : $\sum_{x \in S} d(x) = 2m$

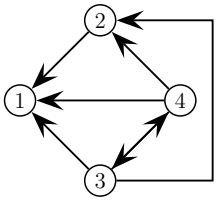
Passons ensuite au cas où les boucles n'ont pas de signification, on parle dans ce cas de graphe simple.

Définition 1.24 (graphe simple) *Un graphe simple G est un graphe sans boucles.*

Un graphe simple peut être orienté ou non-orienté !

✎ **1.20 Exemples de graphes simples** *calculer la liste des arêtes/arcs et la matrice d'adjacence des graphes suivants :*

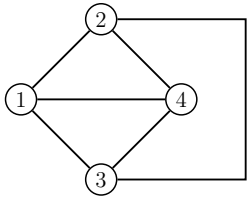
graphe simple orienté



$A = \{(1; 2); (1; 3); (2; 3); (1; 4); (2; 4); (3; 4); (4; 3)\}$

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

graphe simple non-orienté



$A = \{\{1; 2\}; \{1; 3\}; \{2; 3\}; \{1; 4\}; \{2; 4\}; \{3; 4\}\}$

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

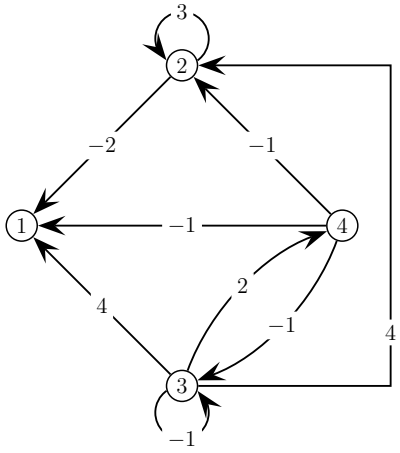
Dans les applications de la théorie des graphes les arcs représentent souvent des données numériques (valeur d'un flux, une durée,...) on a donc besoins de définir des graphes valués (*i.e.* avec des valeurs numérique associées à chaque arc ou arête).

Définition 1.25 (graphe valué) *Un graphe valué $G = (S, A, \nu)$ est un graphe (S, A) (orienté ou non-orienté) muni d'une application $\nu : A \longrightarrow \mathbb{R}$ Le graphe peut être représenté par la matrice des valuations :*

$$W \in \mathscr{M}_n(\mathbb{R}) \quad telleque \quad W_{i,j} = \begin{cases} \infty & si \quad (x_i, y_j) \notin A \\ \nu((x_i, y_j)) & si \quad (x_i, y_j) \in A \end{cases}$$

Pour un graphe valué on ajoutera sur le diagramme sagittal les valuations de chaque arc ou arête.

✎ **1.21 Exemple de graphes valués** *calculer la matrice d'adjacence et la matrice des valuations des graphes suivants :*



graphe valué orienté

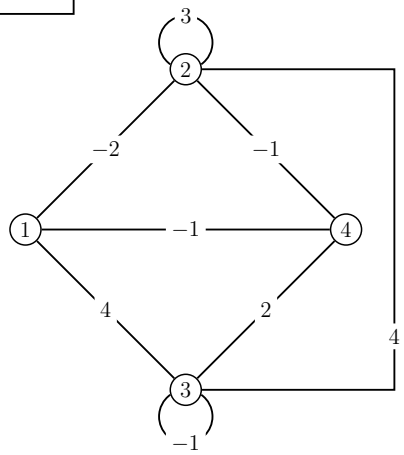
$$W = \begin{pmatrix} \infty & \infty & \infty & \infty \\ -2 & 3 & \infty & \infty \\ 4 & 4 & -1 & 2 \\ -1 & -1 & -1 & \infty \end{pmatrix}$$

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

graphe valué non-orienté

$$W = \begin{pmatrix} \infty & -2 & 3 & 4 \\ -2 & -1 & 4 & -1 \\ 3 & 4 & -1 & 2 \\ 4 & -1 & 2 & \infty \end{pmatrix}$$

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



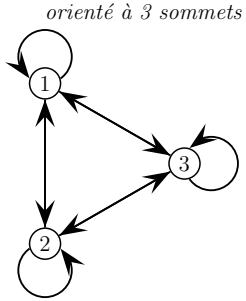
On rencontrera dans les applications de la théorie des graphes plusieurs types de valuations associées aux arcs d'un graphe : poids, longueur, coût, capacité,...

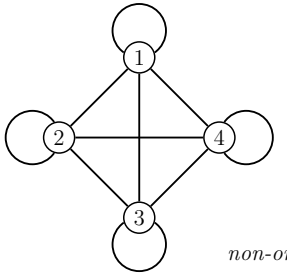
Dans un graphe valué il faut obligatoirement dé-doubler les arcs à double sens pour pouvoir indiquer les deux valuations !

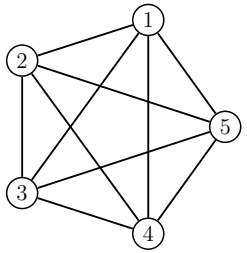
Définition 1.26 (graphe complet) *On appelle « graphe complet à n sommets », souvent noté K_n, le graphe d'ordre n ayant le plus d'arcs/arêtes possibles.*

✎ **1.22 Dessiner les graphes complets**

orienté à 3 sommets *non-orienté à 4 sommets*







non-orienté et simple à 5 sommets

Il y a deux manières de prendre « une partie d'un graphe » suivant qu'on élimine des sommets ou des arcs/arêtes, ce sont les définitions de sous-graphes et graphe partiel.

Définition 1.27 (sous-graphe et graphe partiel) *Soit $G = (S, A)$ un graphe (orienté ou pas) alors*

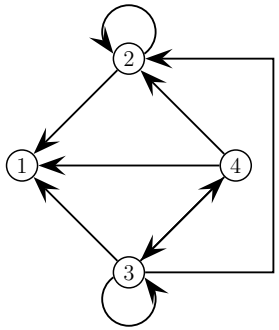
- *un graphe partiel de G est un graphe G' ayant pour sommets* **tous les sommets de** *G et pour arcs/arêtes seulement un sous-ensemble de A, ce qui s'écrit :*

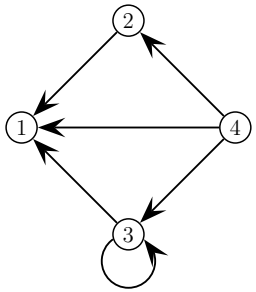
$$G' = (S, A') \quad avec \quad A' \subset A$$

- *un sous-graphe de G est un graphe G' ayant pour sommets* **un sous-ensemble** *S'* **des sommets de** *G et en ne conservant que les arcs/arêtes joignant les sommets de S' ce qui s'écrit :*

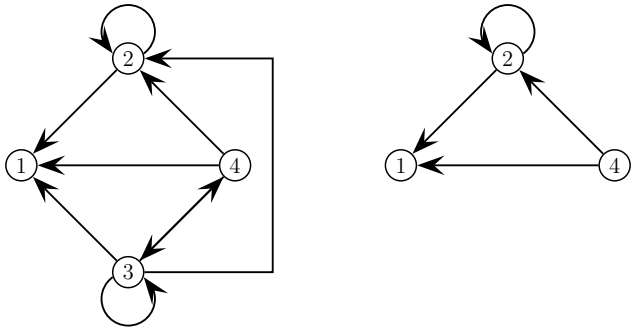
$$G' = (S', A') \quad avec \quad S' \subset S \quad et \quad A' = \{(x, y) \in A | x \in S' \ \ et \ \ y \in S'\}$$

✎ **1.23 graphe partiel de G induit par $A' = A \setminus \{(2,2); (3,2); (4,3)\}$**





✎ **1.24 sous-graphe de G induit par $S' = \{1; 2; 4\}$**



en supprimant le sommet 3 on supprime les arcs $\{(3,1); (3,2); (3,3); (3,4); (4,3)\}$.

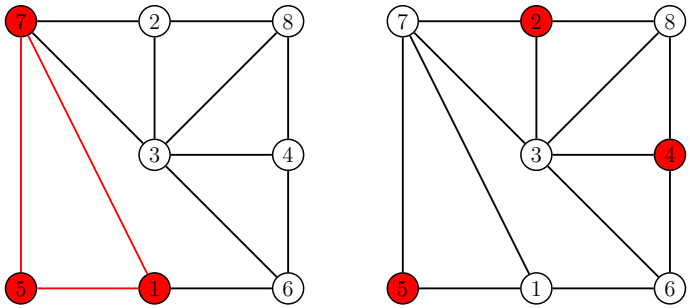
deux exemples importants de sous-graphe et de graphe partiel sont les cliques et les stables :

Définition 1.28 (clique et stable) *Soit $G = (S, A)$ un graphe (orienté ou pas) alors*

- *une clique est un sous-graphe complet de G*
- *un stable est un sous-graphe de G sans arcs/arêtes*

la recherche du plus grand stable ou de la plus grande clique d'un graphe est un problème très important en théorie des graphes

✎ **1.25 Trouver le plus grand stable et la plus grande clique d'un graphe**



Dans la graphe G l'ensemble de sommets $\{1; 5; 7\}$ *induit une clique maximale alors que* $\{2; 4; 5\}$ *induit un stable maximal (il y en a d'autres).*

1.5 Quelques problèmes courants de théorie des graphes

Les notions précédentes jouent un rôle fondamental dans un certain nombre de problèmes « type » de théorie des graphes. Nous allons en donner quelques uns en exemple.

Définition 1.29 (coloriage d'un graphe) *Soit $G = (S, A)$ un graphe simple non-orienté, colorier le graphe G consiste à assigner une couleur (ou un nombre) à chaque sommet du graphe de telle sorte que deux sommets reliées par un arc/arête aient des couleurs différentes* **en utilisant le moins de couleurs possibles**. *Le nombre minimal de couleur est appelé $\gamma(G)$ =nombre chromatique du graphe G.*

De même colorier les arêtes du graphe G consiste à assigner une couleur (ou un nombre) à chaque arête du graphe de telle sorte que deux arêtes reliées à un même sommet aient des couleurs différentes **en utilisant le moins de couleurs possibles**. *Le nombre minimal de couleur est appelé $\gamma'(G)$ =indice chromatique du graphe G.*

L'algorithme « glouton »est le plus simple pour colorier un graphe :

fonction G = Coloriage(G)
  couleur courante = 1
  pour tout x sommet de G **faire**
    V = liste des voisins de x
    couleur = plus petite couleur non encore utilisée dans V
    si *couleur* $\leq$ *couleur courante* **alors** colorier x avec cette couleur
      sinon incrémenter la *couleur courante* et colorier x avec
    fin
  fin faire

Il existe un autre algorithme intéressant pour colorier un graphe : l'algorithme de Welsh-Powell. Cet algorithme est plus compliqué mais souvent moins long à mettre en œuvre.

fonction G = Welsh(G)
  L = liste des sommets classés dans l'ordre décroissant de leur degré
  couleur courante = 0
  tant que $L \neq \emptyset$ **faire**
    incrémenter la *couleur courante*
    Colorier s le premier sommet de L avec la *couleur courante*
    éliminer s de L
    V = liste des voisins de s
    pour tout x dans L **faire**
      si $x \notin V$ **alors** colorier x avec la *couleur courante*
        ajouter les voisins de x à V
      fin
    fin faire
    éliminer les sommets coloriés de L
  fin faire

Attention, les deux algorithmes ne donne pas toujours le nombre minimal de couleurs ! L'algorithme Glouton peut être amélioré en traitant les sommets dans l'ordre décroissant de leur degré (comme dans l'algorithme de Welsh-Powell). Enfin notons que la structure du graphe impose certaines contraintes sur le nombre chromatique :

- les sommets d'une même **clique** doivent être coloriés d'une **couleur différente**
- les sommets d'un même **stable** peuvent tous être coloriés de la **même couleur**

cela permet d'encadrer le nombre Chromatique de G :

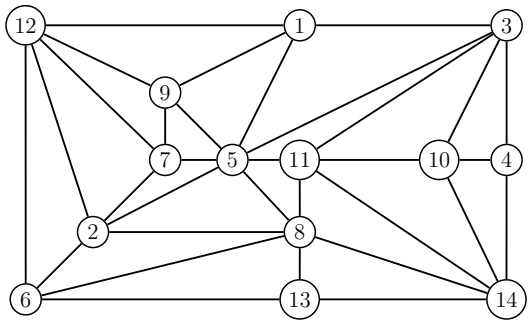
- obtenir un coloriage à k couleurs permet d'affirmer que $\gamma(G) \leq k$
- trouver une clique à k sommets permet d'affirmer que $\gamma(G) \geq k$

Le coloriage d'un graphe permet de résoudre de nombreux problèmes d'incompatibilité.

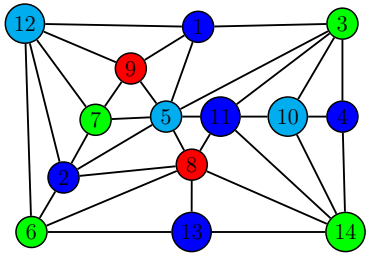
✎ **1.26 Un problème de coloriage**
Dans un groupe de TP de 14 étudiants on doit former des groupes de PPP de quelques étudiants en faisant en sorte que les étudiants d'un même groupe ne s'entendent pas trop mal. On connaît pour chaque étudiant les membres du groupe avec lesquels il ne s'entend pas :

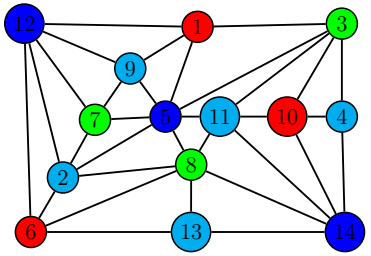
l'étudiant	*1*	*2*	*3*	*4*	*5*	*6*	*7*
ne s'entend pas avec	*3 ;5 ; 9 ;12*	*5 ;6 ;7 ; 8 ;12*	*1 ;4 ;5 ; 10 ;11*	*3 ;10 14*	*1 ;2 ;3 ;7 ; 8 ;9 ;11*	*2 ;8 ; 12 ;13*	*2 ;5 ; 9 ;12*
l'étudiant	*8*	*9*	*10*	*11*	*12*	*13*	*14*
ne s'entend pas avec	*2 ;5 ;6 ; 11 ;13 ;14*	*1 ;5 ; 7 ;12*	*3 ;4 ; 11 ;14*	*3 ;5 ;8 ; 10 ;14*	*1 ;2 ;6 ; 7 ;9*	*6 ;8 ; 6 ;8 ;*	*4 ;8 ;10 ; 11 ;13*

On représente la situation par un graphe simple non-orienté où les sommets représentent les étudiants. On trace une arête quand deux étudiants ne s'entendent pas :



Pour trouver comment former les groupes il suffit de colorier le graphe (chaque couleur constituera un groupe). Suivant l'algorithme utilisé on trouve plusieurs solutions à 4 groupes :





Coloriage avec l'algorithme glouton :

- $s = 1$, $d(s) = 4$, *couleur bleu*
- $s = 2$, $d(s) = 5$, *couleur bleu*
- $s = 3$, $d(s) = 5$, *couleur vert*
- $s = 4$, $d(s) = 3$, *couleur bleu*
- $s = 5$, $d(s) = 7$, *couleur cyan*
- $s = 6$, $d(s) = 4$, *couleur vert*
- $s = 7$, $d(s) = 4$, *couleur vert*
- $s = 8$, $d(s) = 6$, *couleur rouge*
- $s = 9$, $d(s) = 4$, *couleur rouge*
- $s = 10$, $d(s) = 4$, *couleur cyan*
- $s = 11$, $d(s) = 5$, *couleur bleu*
- $s = 12$, $d(s) = 5$, *couleur cyan*
- $s = 13$, $d(s) = 3$, *couleur bleu*
- $s = 14$, $d(s) = 5$, *couleur vert*

avec l'algorithme de Welsh-Powell :

- $s = 5$, $d(s) = 5$, *couleur bleu ainsi que les sommets = [14 ;12],*
- $s = 8$, $d(s) = 4$, *couleur vert ainsi que les sommets = [3 ;7],*
- $s = 2$, $d(s) = 6$, *couleur cyan et les sommets = [11 ;9 ;13 ;4],*
- $s = 6$, $d(s) = 5$, *couleur rouge ainsi que les sommets = [10 ;1],*

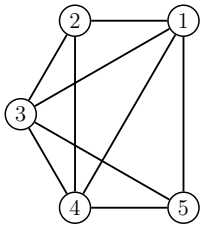
Si dans l'algorithme glouton on prend les sommets dans l'ordre décroissant de leur degré on trouverai le même résultat qu'avec l'algorithme de Welsh-Powell.

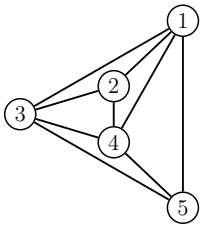
En cherchant un peu, on peut voir qu'il n'est pas possible de trouver une solution à 3 groupes.

Les notions de graphe complet, de stable et de clique jouent un rôle très important dans le coloriage d'un graphe. Mais ce problème est aussi relié à un autre problème d'apparence plus complexe : le problème des graphes planaires.

Définition 1.30 (graphe planaire) *Un graphe $G = (S, A)$ est dit planaire s'il existe un diagramme sagittal de ce graphe où aucun arc/arêtes n'en coupe d'autre.*

✎ **1.27 Rendre le graphe suivant planaire** *il faut déplacer les sommets* 2 *et* 4





Est-ce encore possible si on ajoute l'arête $\{2; 5\}$ au graphe ?

Le théorème suivant fait le lien entre graphe planaire et coloriage d'un graphe.

Théorème 1.31 (des quatre couleurs)
Tout graphe planaire peut être colorié avec au plus quatre couleurs

Un graphe ayant pour nombre chromatique $\gamma(G) = 5$ ne peut donc pas être planaire. Mais attention, la réciproque de ce théorème est fausse : un graphe avec $\gamma(G) = 4$ n'est pas forcément planaire !

Ce problèmes a de nombreuses applications pratiques. Par exemple en électronique, on peut représenter un circuit imprimé par un graphe non-orienté dont les sommets sont des composants électroniques et les arêtes sont des pistes en cuivre. Si le graphe est planaire on pourra graver le circuit imprimé sur une seule face. Dans le cas contraire on devra utiliser un circuit « bicouche » ou des « straps » qui fragilisent le circuit. Dans un autre domaine, lorsqu'on représente un réseau informatique, mieux vaut représenter la situation par un graphe planaire quand c'est possible. Cela permet de mieux repérer les parties d'un réseau qui peuvent se retrouver isolées lorsqu'une connexion est coupé.

La caractérisation des graphes planaires à été obtenue vers 1928 :

Théorème 1.32 (Kuratowski)
un graphe fini est planaire si et seulement si il ne contient pas de sous-graphe qui est une expansion de K_5 (la clique à 5 sommets) ou $K_{3,3}$ (le graphe complet biparti à 3+3 sommets)

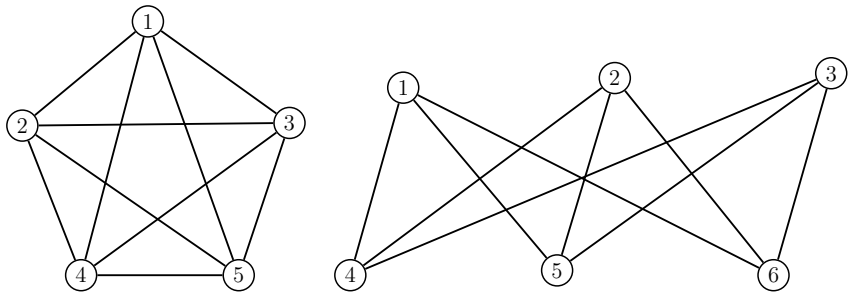


FIGURE 5 – les graphes K_5 et $K_{3,3}$

Pour ceux qui aiment jouer, allez voir le site [5]. But du jeu : rendre des graphes de plus en plus complexe planaires. Un petit algorithme pour vous aider à rendre un graphe planaire dans ce jeu :


```
fonction G = planaire(G)
  tant que G n'est pas planaire faire
        pour tout x sommet de G faire
                V = liste des voisins de x
                M = barycentre des sommets de V
                si on diminue le nombre d'intersections d'arcs
                   alors placer x en M
                fin
          fin faire
        fin faire
```

2 Chemins dans un graphe

2.1 Définitions et premiers exemples

La majeure partie des problèmes modélisés en théorie des graphes repose sur la notion de chemin. Cette notion est tout à fait intuitive, mais nous allons lui donner un sens mathématique très précis.

Définition 2.1 (chemin) *Soit $G = (S, A)$ un graphe orienté (resp. non-orienté) alors un chemin (resp. une chaîne) dans G est une liste de sommets $C = (x_0, x_1, x_2, \ldots, x_k)$ telle qu'il existe un arc (resp. une arête) entre chaque couple de sommets successifs de C. Ce qui s'écrit :*

- *si $G = (S, A)$ est orienté alors $\forall i = 0, 1, \ldots k-1$ $(x_i, x_{i+1}) \in A$*
- *si $G = (S, A)$ est non-orienté alors $\forall i = 0, 1, \ldots k-1$ $\{x_i, x_{i+1}\} \in A$*

On appellera

longueur du chemin *le nombre d'arcs/arêtes du chemin*

chemin/chaîne simple *un chemin/chaîne dont tous les arcs/arêtes sont différents*

chemin/chaîne élémentaire *un chemin/chaîne dont tous les sommets sont différents sauf peut être le départ et l'arrivée (pour autoriser les circuits/-cycles)*

circuit *dans un graphe orienté un chemin simple finissant à sont point de départ*

cycle *dans un graphe non-orienté un chemin simple finissant à sont point de départ*

La notion intuitive de chemin correspond donc plutôt à celle de chemin/chaîne simple ou élémentaire. Heureusement le lemme suivant nous assure qu'on peut toujours s'y ramener.

Théorème 2.2 (lemme de König) *Soient x et y deux sommets distincts d'un graphe G. S'il existe un chemin de G reliant x à y alors il existe un chemin élémentaire de x à y.*

Preuve : On considère un chemin de x à y qui n'est pas élémentaire, il passe donc deux fois par un même sommet s, on est donc dans la situation suivante :

$$C = (x, x_1, \ldots, \underbrace{s, \ldots, s}_{\text{à supprimer}}, \ldots, x_{n-1}, y)$$

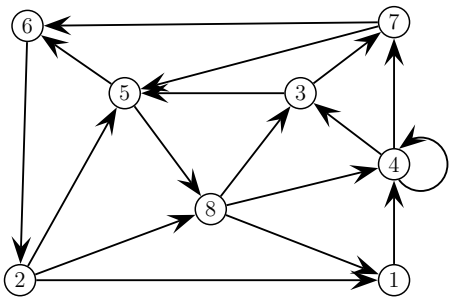
la partie qu'on a supprimé est un cycle/circuit. Ensuite il suffit de recommencer tant que le chemin n'est pas élémentaire. □

Les circuits les plus simples sont les boucles et les doubles flèches :

- une boucle (x, x) est le plus court circuit possible
- une double flèche correspondant à deux arcs (x, y) et (y, x) donne aussi un circuit (x, y, x)

le chemin $C = (x)$ est de longueur nulle ! Il ne correspond donc à aucun arc (c'est cohérent puisque la boucle $(x, x) \neq C$) on parle alors plutôt de chemin « nul ».

✎ **2.1 Dire si les chemins du graphe suivant sont simple, élémentaire, circuit(cycle)et donner leur longueur**



- $(8, 4, 4)$ *: chemin simple mais pas élémentaire, longueur 2*
- $(1; 4; 3; 5; 8; 1)$ *: circuit élémentaire et simple, longueur 5*
- $(1; 4; 7; 6; 2)$ *: chemin élémentaire et simple, longueur 4*
- $(7; 6; 2; 1; 4; 3; 5; 8; 1; 4; 7)$ *: circuit ni simple ni élémentaire, longueur 10*

La notion de longueur de chemin nous permet ensuite de définir la notion de distance dans un graphe.

Définition 2.3 (distance et diamètre) *Dans un graphe $G = (S, A)$ on appelle*

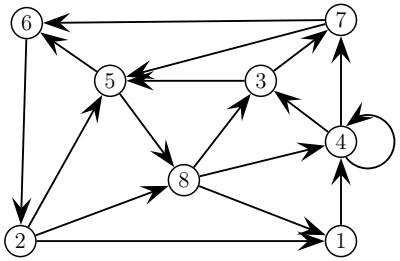
distance *d'un sommet à un autre la longueur du plus court chemin d'un sommet à l'autre, ou ∞ s'il n'y a pas de tel chemin :*

$$\forall x, y \in S, \quad d(x, y) = \begin{cases} k & \text{si plus court chemin de } x \text{ vers } y \\ & \text{est de longueur } k \\ \infty & \text{sinon} \end{cases}$$

diamètre *du graphe la plus grande distance entre deux sommets*

Dans un graphe non-orienté $d(x, y) = d(y, x)$ mais ce n'est pas forcément vrai dans un graphe non-orienté ! À noter aussi, la distance d'un sommet à lui même est toujours nulle $(d(x, x) = 0)$ puisque le chemin « nul » $(C = (x))$ va de x à x !

✎ **2.2 Calculer les distances de $d(1, 8)$ et $d(8, 1)$ et le diamètre du graphe**



$d(8, 1) = 1$

$d(1, 8) = 4$

Dans cet exemple nous sommes obligé de calculer tous les chemins possibles pour être sûr de trouver le plus court/long ce qui même sur un petit graphe est vite très difficile ! Nous consacrerons (plus loin) un chapitre entier de ce cours au problème de la recherche du plus court chemin dans un graphe.

2.2 graphes Euleriens et Hamiltoniens

Nous allons maintenant revenir sur le problème à l'origine de la théorie des graphes.

Définition 2.4 (graphe Eulerien et Hamiltonien) *Soit $G = (S, A)$ un graphe, on dit que*

- *G est Eulerien s'il existe un circuit/cycle $C = (x_0, x_1, \dots x_0)$ passant par tous les arc/arêtes du graphe*
- *G est semi-Eulerien s'il existe un chemin $C = (x_0, x_1, \dots x_k)$ $(x_0 \neq x_k)$ passant par tous les arc/arêtes du graphe*
- *G est Hamiltonien s'il existe un circuit/cycle $C = (x_0, x_1, \dots x_0)$ passant par tous les sommets du graphe*
- *G est semi-Hamiltonien s'il existe un chemin $C = (x_0, x_1, \dots x_k)$ $(x_0 \neq x_k)$ passant par tous les sommets du graphe*

un cycle ou chemin C du graphe correspondant à à l'une de ces définitions sera appelé suivant le cas :
cycle Eulerien, chemin semi-Eulerien, cycle Hamiltonien, chemin semi-Hamiltonien

✎ 2.3 Problème des sept ponts de Königsberg

La ville de Königsberg (aujourd'hui Kaliningrad en Russie) est construite sur l'estuaire de la Pregel et englobe deux îles. Six ponts relient le continent à l'une ou l'autre des deux îles elles mêmes reliées par septième pont (voir aussi le plan FIG.1 dessiné par Euler dans [4]). Euler aimant se promener sur ces ponts se posa un jour la question suivante :
« existe-t-il une promenade dans les rues de Königsberg permettant, à partir d'un point de départ au choix, de passer une et une seule fois par chaque pont, et de revenir à son point de départ ? »

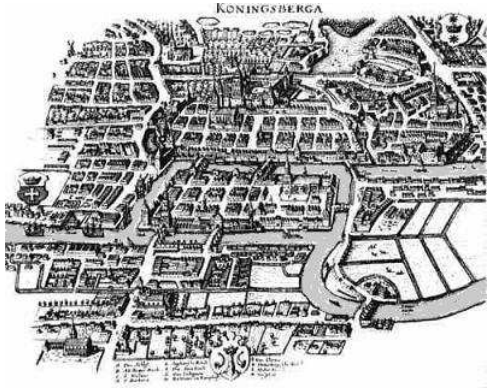
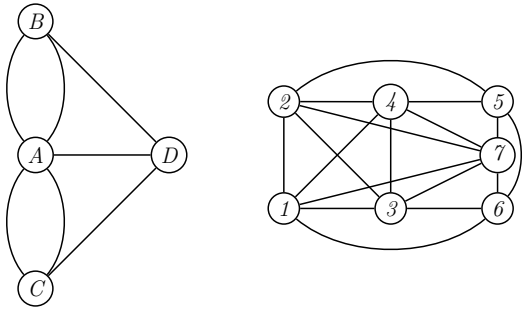


FIGURE 6 – plan de la ville de Königsberg

On peut représenter la situation par les graphes ci-dessous suivant que les ponts sont modélisés par les arêtes[2] *(les sommets représentent les 2 îles et les berges) ou par les sommets (les arêtes connectent deux ponts ayant une berge commune) :*



Résoudre le problème revient à trouver un cycle Eulerien pour le premier graphe, Hamiltonien pour le second graphe. En utilisant la première modélisation, Euler à découvert que ce problème n'avait pas de solution.

Théorème 2.5 (d'Euler) *Soit $G = (S, A)$ un graphe* **connexe** *alors*

- *si G est orienté alors G est Eulerien si et seulement si pour tout sommet le degré sortant est égal au degré entrant :*

$$\forall x \in S, \quad d^+(x) = d^-(x)$$

- *si G est non-orienté alors G est Eulerien si et seulement si pour tout sommet est de degré pair :*

$$\forall x \in S, \quad \exists k \in \mathbb{N}, \quad d(x) = 2k$$

Dans tous les cas (G orientés ou pas) si seulement **deux** *sommets ne vérifient pas les conditions précédentes alors G est semi-Eulerien.*

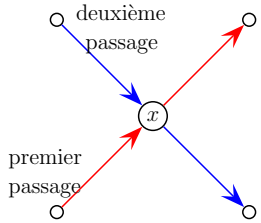
Si on applique ce théorème au premier graphe (non-orienté) représentant les sept pont de Königsberg on voit que les quatre sommets A, B, C, D sont de degré impair, il n'existe donc pas de cycle Eulerien dans ce graphe.

Savoir si un graphe est Eulerien est assez facile (et trouver un cycle Eulerien n'est en général pas très dur), par contre savoir si un graphe est Hamiltonien est un problème plus complexe, en particulier la question de trouver un cycle Hamiltonien minimal est connu sous le nom de :
problème du voyageur de commerce

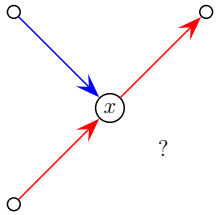
2. Attention, il s'agit d'un multigraphe !

Preuve : Raisonnons sur un sommet x quelconque du graphe :

Pour qu'il existe un cycle Eulerien il faut qu'en chaque sommet lorsqu'on arrive par un arc/arête on puisse repartir par un autre arc/arête. Suivant que le graphe est orienté ou pas on obtient donc que $d^+(x) = d^-(x)$ ou $d(x)$ pair (si le graphe est non-orienté).



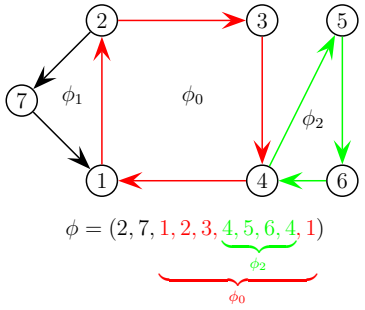
Au contraire si un sommet est de degré impair alors il constitue forcément un « cul de sac» ce qui empêche l'existence d'un cycle. Par contre si seulement deux sommets sont de degré impair il peuvent servir de point de départ et d'arrivé d'un chemin passant par tous les arcs/arêtes du graphe, le graphe est donc semi-Eulerien.



Inversement si chaque sommet est de degré pair on raisonne par récurrence sur le nombre de sommets pour démontrer le théorème :

- un graphe à $n = 1$ seul sommet est évidement Eulerien !
- pour un graphe à n sommets on considère le graphe partiel H constitué des arcs/arêtes en dehors du circuit/cycle ϕ_0 (à éventuellement un seul sommet et sans arc/arête) qui a donc strictement moins de n sommets. Les sommets de H vérifient encore la propriété sur les degrés puisque quand on enlève le circuit/cycle
  - si G est orienté on enlève on enlève 1 arc entrant et 1 arc sortant pour chaque sommet donc les degrés entrant et sortant restent égaux,
  - si G est non-orienté on enlève 2 arêtes incidentes pour chaque sommet donc le degré reste pair,

Par induction chaque composante connexe de H est un graphe Eulérien, et admet donc un circuit/cycle Eulérien ϕ_i. Pour reconstruire un cycle eulérien sur G, il nous suffit de fusionner les circuits/cycles trouvés sur H avec le circuit/cycle enlevé au début. Dans l'exemple ci-contre, une fois ϕ_0 enlevé du graphe on trouve un circuit Eulerien ϕ_2 et un autre semi-Eulerien ϕ_1. Il ne reste plus qu'à les connecter pour retrouver le circuit



□

Proposition 2.6 *Soit G un graphe non-orienté, si G n'est pas (semi-)Eulerien on peut le transformer en un graphe (semi-)Eulerien en lui ajoutant des arcs/arêtes voir, éventuellement, des sommets.*

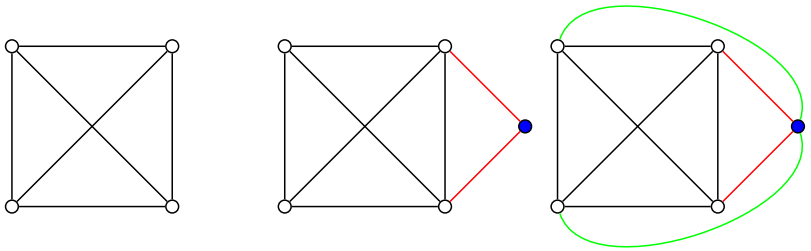
Preuve : Le problème est de modifier le degré des sommets de degré impair. On se contente de démontrer le résultat dans le cas des graphes non-orientés. Soit x, y deux sommets de degré impair :

- s'il n'y a pas d'arête entre x et y alors on en ajoute une arête entre x et y leurs degrés sont augmentés de 1 et deviennent pairs
- s'il y a déjà une arête entre x et y alors on ajoute un sommet z et deux arêtes entre x et z et entre y et z, les degrés de x et y sont augmentés de 1 et deviennent pairs, alors que z est de degré 2 donc pair

on recommence jusqu'à ce qu'il n'y ait plus de sommet de degré impair (comme vu en TD, le nombre de sommets de degré impair est pair donc on y arrivera).

La même démonstration marche dans le cas où G est orienté en considérants les sommets tels que $d^+(x) \neq d^-(x)$ et en leur ajoutant des arcs (bien orientés). □

✎ 2.4 (contre-)exemples de graphes (semi-)Eulerien



le premier graphe est non-Eulerien (sommets de degré 3), en lui ajoutant 1 sommet et 2 arêtes il devient semi-Eulerien, et en ajoutant encore 2 arêtes il devient Eulerien (sommets de degré 4).

Une notion associée à celle de chemin est la notion de connexité.

Définition 2.7 (connexité) *Soit $G = (S, A)$ un graphe tel que pour tout couple de sommets (x, y) il existe un chemin de x vers y :*

$$\forall x, y \in S, \quad \exists x_i \in S, i = 0, \ldots, k, \quad c = (x_0, \ldots, x_k) \text{chemin } G \text{ et } [x_0 = x \text{ et } x_k = y]$$

alors on dira que

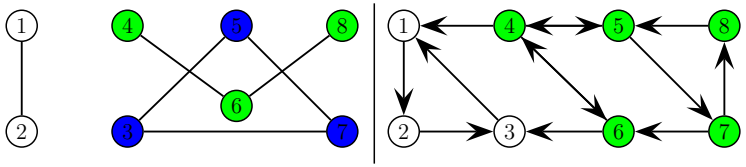
- *G est* **connexe** *si G est non-orienté*
- *G est* **fortement connexe** *si G est orienté*

On appelle aussi **composante (fortement) connexe** *un sous-graphe de G de taille maximale qui est (fortement) connexe.*

Trouver les composantes connexes d'un graphe est assez facile visuellement (si celui-ci est planaire) mais trouver les composantes fortement connexes est plus difficile. On pourra utiliser l'algorithme suivant :

fonction $G =$ composantes_fortement_connexe(G)
  tant que tous les sommets de G ne sont pas marqués **faire**
    $V =$ premier sommet non-marqué
    tant que tous les sommets de V ne sont pas marqués **faire**
      marquer x premier sommet non-marqué de V
      si $\exists$un cycle C passant par x
        alors ajouter les sommets de C à V
      fin
    fin faire
    V est une composante fortement connexe de G
  fin faire

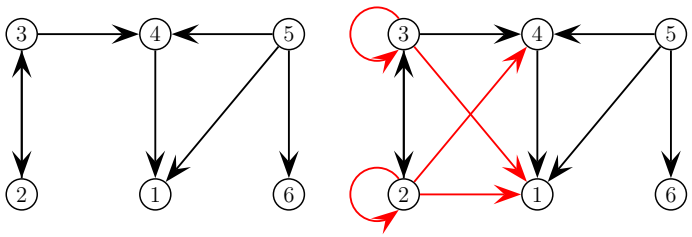
✎ **2.5 trouver les composantes (fortement) connexes des graphes suivants**



La notion de chemin dans un graphe est directement reliée à la notion de transitivité que nous avons déjà évoquée. Rendre un graphe transitif alors qu'il ne l'est pas consiste à ajouter des arcs raccourcis de chemins existants. Cela s'appelle prendre la « fermeture transitive » du graphe.

Définition 2.8 (fermeture transitive) *On appelle fermeture transitive d'un graphe $G = (S, A)$ le plus petit graphe G^* transitif et « contenant » G (c'est à dire tel que G soit un graphe partiel de $G^* = (S, B)$ avec $A \subset B$).*

✎ **2.6 Calculer la fermeture transitive du graphe**



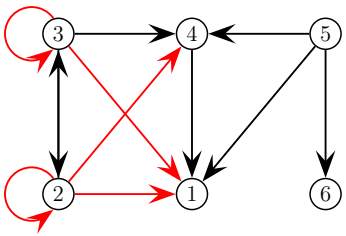
La fermeture transitive d'un graphe peut être obtenue à partir de la matrice d'adjacence.

Théorème 2.9 (fermeture transitive) *Soit $G = (S, A)$ un graphe de matrice d'adjacence M et de fermeture transitive G^* alors la matrice d'adjacence M^* de G^* vérifie*

$$M^* = \sum_{k=1}^{\infty} M^k$$

où la somme (au sens de l'algèbre de boole binaire) est en fait une somme finie.

✎ **2.7 Calcul de la fermeture transitive via la matrice d'adjacence :**



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

$$M + M^2 = \begin{pmatrix} 0&0&0&0&0&0\\ 0&\mathbf{1}&1&\mathbf{1}&0&0\\ \mathbf{1}&1&\mathbf{1}&1&0&0\\ 1&0&0&0&0&0\\ 1&0&0&1&0&1\\ 0&0&0&0&0&0 \end{pmatrix}, \quad M^* = M + M^2 + M^3 = \begin{pmatrix} 0&0&0&0&0&0\\ \mathbf{1}&1&1&1&0&0\\ 1&1&1&1&0&0\\ 1&0&0&0&0&0\\ 1&0&0&1&0&1\\ 0&0&0&0&0&0 \end{pmatrix},$$

$$\textit{car } M^* + M^4 = \begin{pmatrix} 0&0&0&0&0&0\\ 1&1&1&1&0&0\\ 1&1&1&1&0&0\\ 1&0&0&0&0&0\\ 1&0&0&1&0&1\\ 0&0&0&0&0&0 \end{pmatrix} = M^* \textit{ donc plus de changement après.}$$

Preuve : Soit $\mathscr{R}$ la relation associée au graphe G et M sa matrice d'adjacence alors :

- M^k est la matrice d'adjacence de la relation $\underbrace{\mathscr{R} \circ \mathscr{R} \circ \cdots \circ \mathscr{R}}_{k \text{ compositions}}$ donc chaque « 1 » de cette matrice représente une suite de k arcs adjacents dans le graphe de G.
- Lorsqu'on additionne les différentes matrices M^k (au sens de l'addition des booléens) on ajoute donc des « 1 » dans la matrice M qui correspondent à des arcs n'existant pas dans le graphe de G mais qui sont des « raccourcis » dans ce graphe. On en déduit donc que $M^* = \sum_{k=1}^{\infty} M^k$ va contenir tous les raccourcis possibles dans G en plus des arcs déjà dans G.
- Il reste à voir qu'au bout d'un moment tous les « 1 » dans la matrice M^{K+1} ont déjà été ajouté dans la somme $M^* = \sum_{k=1}^{K} M^k$. Cela arrive dès qu'on a fait le chemin élémentaire le plus long possible dans G.

□

On rencontrera aussi les deux notions suivantes :

Définition 2.10 *Graphes τ-équivalents et τ-minimaux deux graphes orientés G_1 et G_2 sont dits τ-équivalents s'ils ont la même fermeture transitive :*

$$G_1 \tau G_2 \Longleftrightarrow G_1^* = G_2^*$$

Un graphe est dit τ-minimal si aucun graphe partiel de G n'a la même fermeture transitive que G

$$\forall\ G' \text{ graphe partiel de } G,\quad G'^* \neq G^*$$

On remarquera que :

Proposition 2.11 *τ est une relation d'équivalence sur l'ensemble des graphes.*

Preuve :

- τ réflexive car $G^* = G^*$
- τ symétrique car $G_1^* = G_2^* \Longleftrightarrow G_2^* = G_1^*$
- τ transitive car $G_1^* = G_2^*$ et $G_2^* = G_3^* \Longrightarrow G_1^* = G_3^*$

□

La fermeture transitive est un outil de calcul assez puissant en théorie des graphes. Par exemple elle permet de repérer les composantes (fortement-)connexe d'un graphe.

Proposition 2.12 *Soit G un graphe et G^* sa fermeture transitive, le sous-graphe induit sur G^* par une composante (fortement-)connexe est un graphe complet.*

La fermeture transitive permet aussi de repérer l'absence de circuits dans un graphe.

Proposition 2.13 *Un graphe $G = (S, A)$ est sans circuit si la relation $\mathscr{R}^*$ associée à sa fermeture transitive G^* est une relation d'ordre.*

Les graphes orientés sans circuits possèdent des propriétés spécifique, en particulier il y a dans un graphe sans circuit une notion de hiérarchie entre les sommets. C'est ce qu'on appelle la décomposition en niveaux ou aussi un « tri topologique ».

Proposition 2.14 (décomposition en niveau) *Si G est un graphe sans circuit alors on peut définir pour chaque sommet un* **niveau** *de la manière suivante :*

- *Les sommets sans prédécesseurs sont de niveau 0*
- *tout sommet x a un niveau supérieur aux niveaux de ses prédécesseurs :*

$$niveau(x) = \max_{y \in \Gamma^-(x)} niveau(y) + 1$$

On peut ensuite re-dessiner le graphe G en disposant les sommets de gauche à droite dans l'ordre croissant des niveaux.

Si G possède un circuit $C = (x_0, x_1, \ldots, x_{n-1}, x_0)$ il ne peut pas avoir de niveau puisque dans ce cas on aurait :

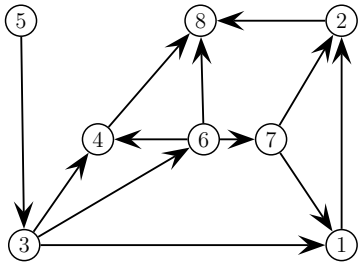
$$niveau(x_0) < niveau(x_1) < \cdots < niveau(x_{n-1}) < niveau(x_0)$$

Pour décomposer en niveau un graphe G on utilisera l'algorithme suivant :

fonction *niveau* = decomp_niveau(G)
  n = nombre de sommets du graphe
  niveau = matrice nulle de taille $1 \times n$
  degre = matrice de taille $1 \times n$ contenant les degrés entrants $d^-(x)$
  tant que tous les sommets ne sont pas marqués **faire**
    L = liste des sommets non-marqués de degré entrant nul
    pour tout $x \in L$ **faire**
      pour tout y successeur de x dans G **faire**
        si $niveau(y) < niveau(x) + 1$
          alors $niveau(y) = niveau(x) + 1$
        fin
        $degre(y) = degre(y) - 1$ (revient à éliminer l'arc (x, y))
      fin faire
    fin faire
    on marque les sommets de L
  fin faire

Cette technique nous resservira plus loin dans la recherche des chemins les plus courts dans un graphe, il faut donc savoir décomposer un graphe en niveaux.

2.8 Décomposition en niveau d'un graphe sans circuit :

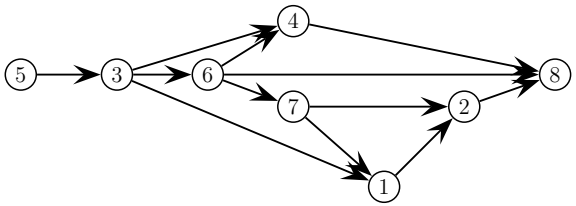


- $d^-(5) = 0$ *donc niveau 0 et on élimine* $(5, 3)$
- $d^-(3) = 0$ *donc niveau 1 et on élimine* $(3, 1)$, $(3, 4)$, $(3, 6)$
- $d^-(6) = 0$ *donc niveau 2 et on élimine* $(6, 4)$, $(6, 8)$, $(6, 7)$
- $d^-(4) = d^-(7) = 0$ *donc niveau 3 et on élimine* $(4, 8)$, $(7, 1)$, $(7, 8)$
- ...

au final niveau =

4	*5*	*1*	*3*	*0*	*2*	*3*	*6*

et on peut redessiner le graphe :



Le graphe associé à une relation d'ordre admet une décomposition en niveaux (si on ne tient pas compte des boucles). Faire le diagramme de Hasse d'une relation d'ordre revient à re-dessiner le graphe en disposant les sommets du graphe dans l'ordre croissant des niveaux calculés.

Un cas très important de graphe sans circuit est le cas des arbres car dans un arbre il est très facile de retrouver un chemin entre la racine et un sommet quelconque :

Théorème 2.15 *Soit $G = (S, A)$ un arbre de racine r, alors pour tout sommet x il existe un unique chemin de x vers r.*

Preuve : Pour démontrer ce théorème il suffit d'indiquer comment construire ce chemin. Pour trouver ce chemin il suffit de « remonter » le long d'une branche en lisant la liste des prédécesseurs. Le prédécesseur d'un sommet dans l'arbre étant à chaque fois unique il n'y pas de choix et donc le chemin trouvé est bien unique □

Cette démonstration nous donne directement un algorithme pour calculer le chemin entre la racine et un sommet de l'arbre :

fonction $C =$ chemin_arbre(G, x)
  $P =$ liste des prédécesseurs de l'arbre G
  si $P(x) = \emptyset$ **alors** $C = (x)$
    sinon $C =$ (chemin_arbre$(G, P(x)), x)$
  fin

2.3 Parcours de graphes orientés

Le parcours d'un graphe est un problème type de théorie des graphes auquel peuvent se ramener de nombreux autres problèmes d'algorithmique.

Définition 2.16 (parcours d'un graphe) *Soit $G = (S, A)$ un graphe et $x \in S$ un sommet, un parcours du graphe G à partir de x est une* **visite de chaque sommet accessible depuis** *x. Le résultat d'un parcours est un ensemble de chemins partants de x allant vers les sommets accessibles depuis x.* **Un parcours peut être représenté par** *sous-graphe de G qui est* **un arbre de racine** *x (ou une forêt si certains sommets de G ne sont pas accessibles depuis x).*

D'un point de vue algorithmique, un parcours correspond à la procédure suivante :

procédure parcours(G, x)
  $L =$ liste des sommets à traiter (vide au départ)
  mettre x dans L (début du traitement de x)
  tant que $L \neq \emptyset$ **faire**
    sortir le 1er sommet y de L (y en cours de traitement)
    $V =$ successeurs non traités de y
    pour tout $z \in V$ **faire**
      mettre z dans L (début du traitement de z)
    fin faire
    fin du traitement de y
  fin faire

à chaque étape de la boucle **tant que** *un seul sommet y est traité qui engendre le début du traitement d'un ou plusieurs autre sommets . . .*
On peut définir un **ordre de parcours** *en numérotant à un endroit précis de la boucle* **tant que** *le sommet y en cours de traitement.*

Les algorithmes de type « glouton » qui consistent à traiter chaque sommet du graphe peuvent souvent être écrit en utilisant en utilisant le modèle d'un parcours (ce qui parfois conduit à des améliorations de l'algorithme). On peut par exemple voir les algorithmes de coloriage de graphes ou de décomposition en niveau sous forme de parcours de graphes (reprendre l'algorithme de Weslh sous cet angle).

Lors du parcours d'un graphe G, au cours du traitement d'un sommet x celui-ci doit être **marqué**, selon le moment où l'on décide de marquer le sommet (en début ou en fin de traitement), l'ordre dans lequel on décide de traiter les successeurs de ce sommet (par ordre de numéro croissant ou décroissant par exemple), on obtiendra des parcours très différents.

Dans la pratique on s'intéressera surtout à récupérer la liste des prédécesseurs correspondant à l'arbre de parcours obtenu.

Les parcours que nous allons étudier s'appliquent aussi aux graphes non-orientés, Dans ce cas on considère que les arêtes sont des arcs à double sens.

Commençons par étudier un premier type de parcours : le parcours en largeur.

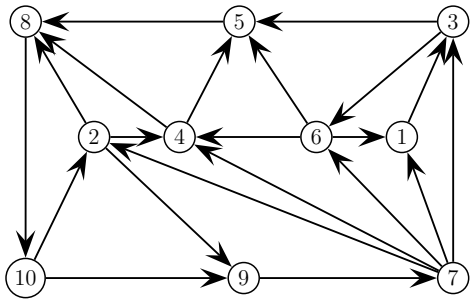
Définition 2.17 (parcours en largeur) *Soit $G = (S, A)$ un graphe et $x \in S$ un sommet, un parcours en largeur du graphe G à partir de x est un parcours dans lequel un sommet y est marqué* **avant** *le début de traitement de ses successeurs.*

Si on veut récupérer la liste des prédécesseurs P qui permet de retrouver l'arbre de parcours en largeur depuis le sommet x on utilisera l'algorithme suivant :

fonction $P =$ parcours_largeur(G, x)
$L =$ file des sommets à traiter
$P =$ liste des prédécesseurs de l'arbre de parcours
marquer le sommet x et le mettre dans L
tant que $L \neq \emptyset$ **faire**
sortir le 1er sommet y de L
$V =$ successeurs non traités de y
pour tout $z \in V$ **faire**
marquer $z; P(z) = y$
mettre z à la fin de L
fin faire
fin faire

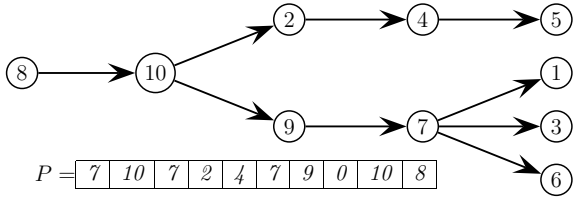
La liste L des sommets à traiter est l'exemple type d'une file de type FIFO[3] :
- on ajoute les éléments « par le bas » de la file
- on retire les éléments « par le haut » de la file

✎ **2.9 Parcours du graphe G en largeur, par ordre croissant des sommets, depuis le sommet 8 :**



3. First In, First Out

Donne l'arbre de parcours :



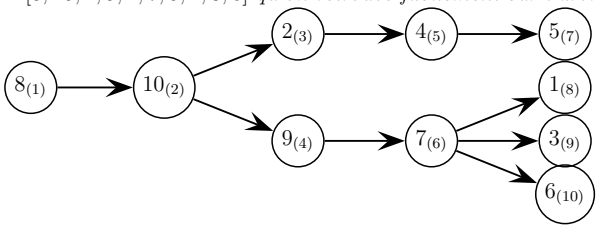
$P =$

7	10	7	2	4	7	9	0	10	8

Détail de l'algorithme :

x	$\Gamma^+(x)$	*marqués*	L
8	{*10*}	*[8 ;10]*	*[10]*
10	{*2 ;9*}	*[8 ;10 ;2 ;9]*	*[2 ;9]*
2	{*4 ;8 ;9*}	*[8 ;10 ;2 ;9 ;4]*	*[9 ;4]*
9	{*7*}	*[8 ;10 ;2 ;9 ;4 ;7]*	*[4 ;7]*
4	{*5 ;8*}	*[8 ;10 ;2 ;9 ;4 ;7 ;5]*	*[7 ;5]*
7	{*1 ;2 ;3 ;4 ;6*}	*[8 ;10 ;2 ;9 ;4 ;7 ;5 ;1 ;3 ;6]*	*[5 ;1 ;3 ;6]*
5	{*8*}	*[8 ;10 ;2 ;9 ;4 ;7 ;5 ;1 ;3 ;6]*	*[1 ;3 ;6]*
1	{*3*}	*[8 ;10 ;2 ;9 ;4 ;7 ;5 ;1 ;3 ;6]*	*[3 ;6]*
3	{*5 ;6*}	*[8 ;10 ;2 ;9 ;4 ;7 ;5 ;1 ;3 ;6]*	*[6]*
6	{*1 ;4 ;5*}	*[8 ;10 ;2 ;9 ;4 ;7 ;5 ;1 ;3 ;6]*	*[ ]*

L'ordre de parcours des sommets du graphes est donné par la liste des sommets marqués : $[8; 10; 2; 9; 4; 7; 5; 1; 3; 6]$ *qu'on retrouve facilement sur l'arbre :*



Le parcours en largeur est très utile pour trouver les distances depuis un sommet donné dans un graphe.

Théorème 2.18 (parcours en largeur et distances) *Soit $G = (S, A)$ un graphe et x un sommet, alors les niveaux des sommets dans le parcours en largeur depuis le sommet x sont exactement les distances de x à ces sommets.*

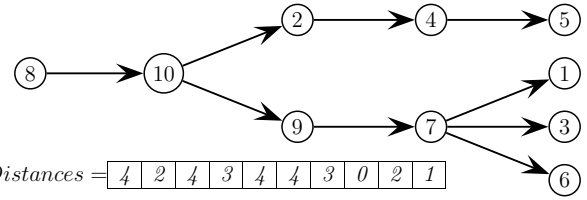
Preuve : La démonstration se fait par récurrence
- la racine de l'arbre est de niveau 0 et on a bien $d(x, x) = 0$
- par hypothèse pour tous les sommets y de niveau k on a bien $d(x, y) = k$, examinons maintenant la distance à x d'un successeur z d'un des sommets y :
  — son niveau est $k + 1$ (successeur de y de niveau k)
  — sa distance à x vérifie $d(x, z) \geq d(x, y) + d(y, z) = k + 1$

— si sa distance à x vérifiait $d(x,z) < k+1$ il y aurait alors un autre chemin de x à z dans le graphe $(x, \ldots, y', z)$ mais alors

$$d(x,z) \leq d(x,y') + d(y',z) \Longrightarrow d(x,y') \leq d(x,z) - d(y',z) < k$$

donc z est le successeur d'un sommet de niveau $< k$ et z aurait du être traité au pire au niveau k et pas au niveau $k+1$! Donc $d(x,z) \leq k+1$
conclusion $d(x,z) = k+1$ □

✎ **2.10 Calcul des distances du sommet 8 dans G**



Distances =

4	*2*	*4*	*3*	*4*	*4*	*3*	*0*	*2*	*1*

Passons maintenant au parcours en profondeur :

Définition 2.19 (parcours en profondeur) *Soit $G = (S, A)$ un graphe et $x \in S$ un sommet, un parcours en profondeur du graphe G à partir de x est un parcours dans lequel un sommet y n'est marqué qu'***après** *le début du traitement de ses successeurs*

Comme pour le parcours en largeur on peut écrire précisément l'algorithme permettant de récupérer l'arbre de parcours en profondeur :

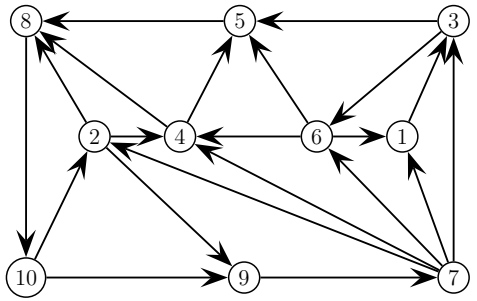
fonction $P =$ parcours_profondeur(G, x)
  $L = [x]$ (Pile des sommets à traiter)
  $P =$ liste des prédécesseurs de l'arbre de parcours
  tant que $L \neq \emptyset$ **faire**
    sortir le 1er sommet y de L
    $V =$ successeurs non traités de y
    pour tout $z \in V$ **faire**
      $P(z) = y$
      mettre z au début de L
    fin faire
    marquer le sommet y
  fin faire

La liste L des sommets à traiter est l'exemple type d'une Pile de type LIFO[4] :
- on ajoute les éléments « par le haut » de la pile
- on retire les éléments « par le haut » de la pile

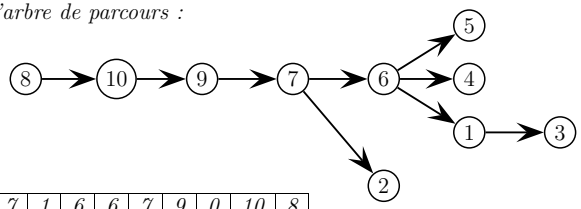
Cet algorithme admet aussi une formulation récursive plus simple à programmer :

fonction $P =$ parcours_profondeur(G, x)
  marquer x (début du traitement de x)
  $V =$ successeurs non traités de x
  pour tout $y \in V$ **faire**
    $P(y) = x$
    $P =$ parcours_profondeur(G, y)
  fin faire
  fin du traitement de x

✎ **2.11 Parcours du graphe G en profondeur, par ordre décroissant des sommets, depuis le sommet 8 :**



Donne l'arbre de parcours :



$P =$

6	*7*	*1*	*6*	*6*	*7*	*9*	*0*	*10*	*8*

Le parcours en profondeur ne permet pas de calculer les distances depuis un sommet ! Au contraire le parcours en profondeur essaye de construire les chemins les plus long possibles depuis un sommet donné.

4. Last In, First Out

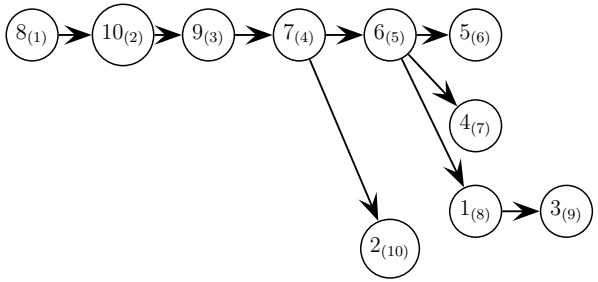
Définition 2.20 (numérotation préfixe et suffixe) *On appelle numérotation* **préfixe** *et numérotation* **suffixe** *les numérotations des sommets du graphe correspondants à l'ordre de traitement des sommets du graphe lors du parcours en profondeur suivant qu'on numérote un sommet* **avant** *ou* **après** *le traitement de ses successeurs :*

fonction $P =$ parcours_profondeur(G, x)
  marquer x
  numérotation **préfixe** de x
  $V =$ successeurs non traités de x
  pour tout $y \in V$ **faire**
    $P(y) = x$
    $P =$ parcours_profondeur(G, y)
  fin faire
  numérotation **suffixe** de x

✎ **2.12 numérotation des sommets dans le parcours en profondeur :**

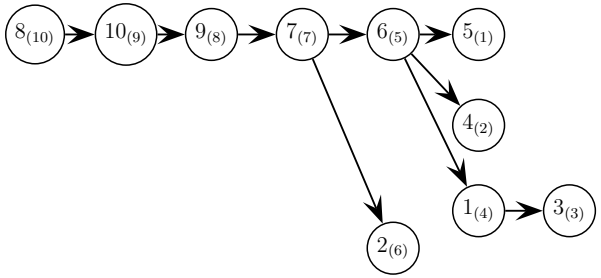
numérotation préfixe =

8	10	9	7	6	5	4	1	3	2



numérotation suffixe =

4	6	3	2	1	5	7	10	8	9



Le parcours en profondeur sert, entre autre, à classer les arcs en différentes catégories.

Définition 2.21 (classification des arcs) *le parcours en profondeur d'un graphe G depuis le sommet x permet de définir quatre types d'arc :*

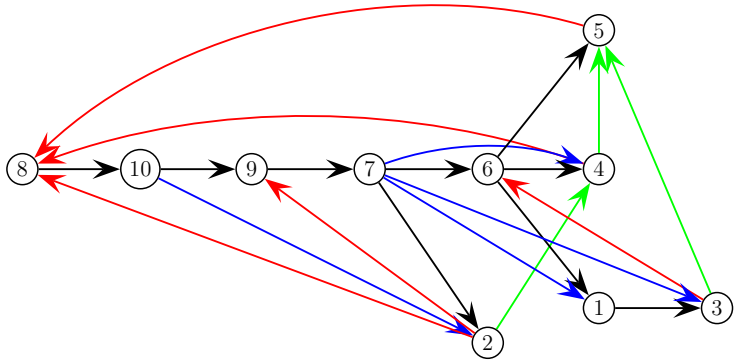
arcs couvrants : *les arcs retenus pour le parcours en profondeur*

arcs directs : *les arcs n'appartenant pas au parcours en profondeur mais reliant un sommet à un descendant*

arcs rétrograde : *les arcs n'appartenant pas au parcours en profondeur mais reliant un sommet à un ascendant (ou à lui même)*

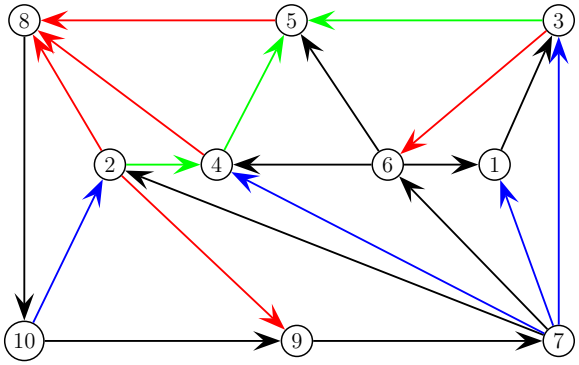
arcs traversiers : *les arcs n'appartenant pas au parcours en profondeur mais reliant deux branches distinctes de l'arbre*

✎ **2.13 Classement des arcs du graphe G d'après le parcours en profondeur, par ordre décroissant des sommets, depuis le sommet 8 :**



arcs couvrants : **arcs rétrograde :**

arcs directs : **arcs traversiers :**



Cette classification permet de détecter des circuits dans un graphe.

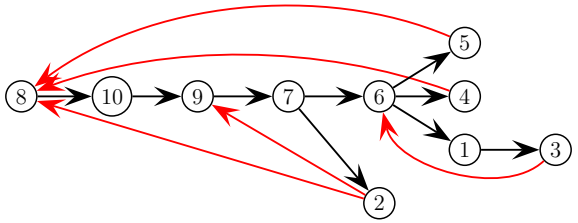
Théorème 2.22 (détection des circuits) *Dans un graphe $G = (S, A)$ et x un sommet. Si dans le parcours en profondeur de G à partir de x il existe un arc rétrograde alors il existe au moins un circuit dans le graphe G.*

Preuve : Soit (y, z) un arc rétrograde du graphe G (lors du parcours depuis x). Soit $y = z$ et alors cet arc est une boucle qui est le plus simple des circuits. Sinon il existe un chemin dans l'arbre de parcours qui va de x à y passant par z, donc on a un chemin C' de z à y qui permet de construire un circuit C'' :

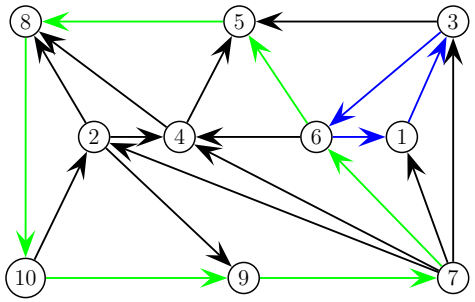
$$C = (x, \ldots, \underbrace{z, \ldots\ldots\ldots, y}_{C' = \text{chemin de } z \text{ à } y}) \Longrightarrow C'' = (y, \underbrace{z, \ldots, y}_{C'}) \text{est un circuit de } G$$

□

✎ **2.14 Exemple de détection de circuit du graphe G d'après le parcours en profondeur, par ordre décroissant des sommets, depuis le sommet 8 :**



- *l'arc rétrograde* $(5, 8)$ *permet de retrouver le circuit :*$(8, 10, 9, 7, 6, 5, 8)$
- *l'arc rétrograde* $(3, 6)$ *permet de retrouver le circuit :*$(3, 6, 1, 3)$



3 Problèmes d'optimisation pour des graphes valués

3.1 Arbre couvrant optimal

Dans le cas des graphes non-orientés on s'intéresse aussi aux arbres couvrants :

Définition 3.1 (arbre couvrant) *Soit $G = (S, A)$ un graphe on appelle « arbre couvrant de G »un graphe $G' = S, A')$ graphe partiel de G qui est un arbre.*

De nombreux problèmes associés aux graphes valués peuvent être résolus par la recherche d'un arbre couvrant de poids minimal.

Proposition 3.2 *Soit $G = (S, A, \nu)$ un graphe non-orienté et valué parmi tous les arbres couvrant de G il en existe un $G' = (S, A')$ telle que $\sum_{\{x;y\} \in A'} \nu(\{x; y\})$ (la somme des poids des arêtes de G') soit maximale (resp. minimale) . On appelle cet arbre l'* **arbre couvrant de G de poids maximal (resp. minimal)** .

Cet arbre peut être obtenu par l'algorithme de Kruskal ou l'algorithme de Prim. À chaque fois on part d'un sommet quelconque (1par exemple) et suivant l'algorithme :

Algorithme de Prim consiste à construire l'arbre en prenant choisissant à chaque étape une arête joignant les sommets connectés à l'arbre aux autres sommets en prenant l'arête de poids optimal (maximal ou minimal suivant le cas)

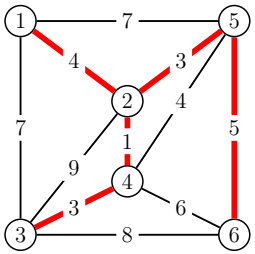
fonction $T = \mathsf{prim}(G)$
  poids = poids total de l'arbre couvrant (initialisé à 0)
  marquer le sommet 1
  tant que il reste des sommets non-marqués **faire**
    $\{x; y\}$ = arête de coût minimal joignant un sommet marqué x et un sommet non-marqué y
    marquer le sommet y et ajouter l'arête $\{x; y\}$ à T
    $poids = poids + W(x, y)$
  fin faire

Algorithme de Kruskal consiste à balayer les arêtes triées dans l'ordre (croissant ou ndécroissant suivant le cas) et à choisir l'arête si les sommets ne sont pas déjà connectés :

fonction $T = \mathsf{Kruskal}(G)$
  initialisation *poids* = poids total de l'arbre couvrant (initialisé à 0)
  pour chaque sommet $x \in S$ **faire**
    $E(x) = \{$ sommets connectés à $x\} = \{x\}$ **fin faire**
  traitement
  pour chaque arête $(x, y) \in A$ (par ordre de poids décroissant) **faire**
    si $E(x) \neq E(y)$ **alors** ajouter l'arête (x, y) à l'arbre T
      $poids = poids + W(x, y)$
      $F = E(x) \cup E(y)$
      pour chaque sommet $z \in F$ **faire**
        $E(z) = F$
      fin faire
    fin
  fin faire

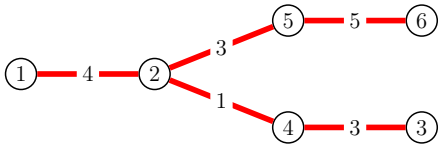
Quelques deux exemples pour voir le fonctionnement de ces algorithmes :

✎ 3.1 Arbre couvrant de poids minimal avec l'algorithme de Prim



sommets marqués	*arêtes sortantes*	*poids minimal*
1	{1;2} {1;3} {1;5}	*4*
1,2	{1;3} {1;5} {2;3} {2;5} {2;4}	*1*
1,2,4	{1;3} {2;3} {4;3} {1;5} {2;5} {4;5} {4;6}	*3*
1,2,4,3	{1;5} {2;5} {4;5} {3;6} {4;6}	*3*
1,2,4,3,5	{3;6} {4;6} {5;6}	*5*

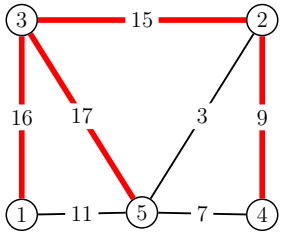
Poids total de l'arbre = 16

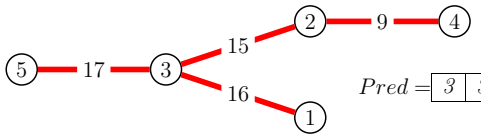


Pred = | 0 | 1 | 4 | 2 | 2 | 5 |

✎ 3.2 Arbre couvrant de poids maximal avec l'algorithme de Kruskal

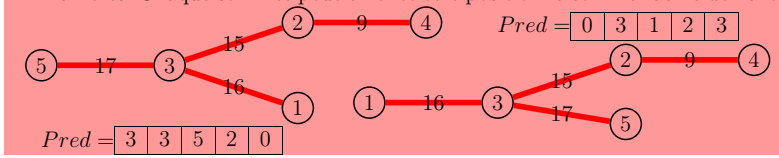
- *arête* $\{5,3\}$ ⇒ *poids=0+17*
- *arête* $\{3,1\}$ ⇒ *poids=17+16*
- *arête* $\{2,3\}$ ⇒ *poids=33+15*
- *arête* $\{1,5\}$ *1 et 5 déjà connectés*
- *arête* $\{4,2\}$ ⇒ *poids=48+9*
- *arête* $\{5,4\}$ *5 et 4 déjà connectés*
- *arête* $\{2,5\}$ *2 et 5 déjà connectés*
- *Poids total de l'arbre = 57*





Pred = | 3 | 3 | 5 | 2 | 0 |

Dans le cas d'un graphe non-orienté, un arbre peut être représenté de nombreuse manières. Chaque sommet peut en effet être positionné comme racine de l'arbre :



3.2 Problème du plus court chemin

Le problème d'optimisation suivant est celui du chemin optimal, il intervient directement dans le fonctionnement des routeurs d'un réseau informatique(protocole OSPF par exemple). Nous avons déjà étudié cette question dans la partie 2.1 de ce cours, où nous avons défini la longueur des chemins. Ici nous avons besoin d'étudier une notion plus générale de longueur qui puisse être appliquée à un graphe valué.

Définition 3.3 (longueur et distance) *Dans un graphe orienté valué* $G = (S, A, f)$ *on appellera longueur d'un chemin* $C = (x_0, x_1, \ldots, x_{p-1}, x_p)$ *relativement à* f *la valeur*

$$\text{Longueur}_f(C) = \sum_{i=0}^{p-1} f(x_i, x_{i+1})$$

on appellera distance de x *à* y *par rapport à* f *la longueur (relativement à* f*) du plus court chemin de* x *à* y

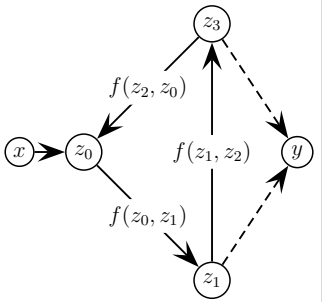
$$Dist_{\min}(x,y) = \min_{C=(x,\ldots,y)} \text{Longueur}_f(C), \quad et \quad Dist_{\max}(x,y) = \max_{C=(x,\ldots,y)} \text{Longueur}_f(C)$$

Ces définitions généralisent les définitions de longueur et distance dans un graphe orienté (mais non-valué). On retrouve la définition de la partie 2.1 en prenant la valuation $f(x,y) = 1, \ \forall (x,y) \in A$ (tous les arcs sont de longueur 1).

Proposition 3.4 (existence du chemin optimal) *Dans un graphe orienté valué* $G = (S, A, f)$ *il existe un plus court (resp. long) chemin entre tout couple de sommets si et seulement si il n'existe pas de circuit de longueur négative (resp. positive) relativement à* f*.*

Preuve :

Supposons qu'il existe un circuit $(z_0, z_1, \ldots, z_0)$ dans le graphe et notons sa longueur (relativement à f) $l = \sum f(z_i, z_{i+1})$. Alors on peut trouver deux sommets x, y tel qu'un chemin optimal (de longueur L relativement à f) entre ces deux sommets passe par un des sommets z_i (prendre $x = z_0$!) ce chemin s'écrit donc $C = (x, \ldots, z_0, \ldots, y)$. Dans ce cas en rajoutant autant de tours de circuits qu'on le souhaite on obtient un nouveau chemin



$$C_k = (x, \ldots, \underbrace{z_0, \ldots, z_0}_{1^{\text{er}} \text{ tour}}, \ldots, \underbrace{z_0, \ldots, z_0}_{k^{\text{ième}} \text{ tour}}, \ldots, y) \Rightarrow L_k = L + k \times \left(\sum_{i=0,\ldots} f(z_i, z_{i+1}) \right) = L + k \times l$$

en faisant tendre $k \to \infty$ on obtient, selon le signe de l, que la longueur de C_k tend vers $\pm\infty$. Il ne peut donc pas exister de distance minimale entre x et y si $l < 0$ et il ne peut donc pas exister de distance maximale entre x et y si $l > 0$. □

La premier algorithme pour calculer les distances minimales consiste à calculer toutes les distances entres couples de sommets (i, j) en essayant de les diminuer en passant par un autre sommet k. C'est l'algorithme de Floyd-Warshall-Roy.

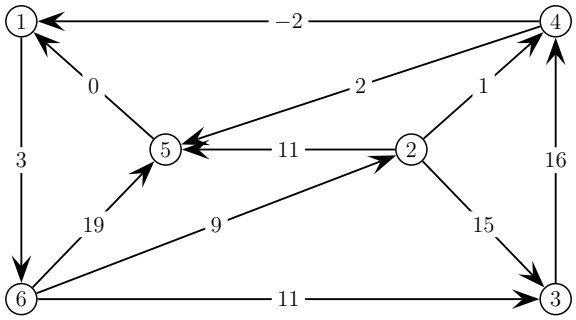
Définition 3.5 (algorithme de Floyd-Warshall-Roy) *Dans un graphe orienté valué $G = (S, A, f)$, d'ordre n et de taille m. L'algorithme de Floyd-Warshall calcule deux matrices de taille $n \times n$*

- *$Dist$ matrice des distances telle que $Dist(x, y) =$ distance optimale de x à y*
- *$Pred$ matrice des prédécesseurs telle que $Pred(x, y) =$ prédécesseur de y dans le chemin optimal depuis x*

Pour le plus court chemin l'algorithme s'écrit :

fonction $[Dist, Pred] =$ FLOYD(G)
Initialisation :
$n =$ nombre de sommets de G
$Dist =$ matrice des poids relativement à f
initialisée à 0 sur la diagonale $Dist(s, s) = 0, \forall s = 1, \ldots, n$
$Pred =$ matrice des prédécesseurs initialisée à
$Pred(i, j) = i$ si l'arc (i, j) existe, 0 sinon
Traitement :
pour $z = 1$ **jusqu'à** n **faire**
  pour $x = 1$ **jusqu'à** n **faire**
    pour $y = 1$ **jusqu'à** n **faire**
      si $Dist(x, z) + Dist(z, y) < Dist(x, y)$
        alors modifier $Dist(x, y)$ et $Pred(x, y) = Pred(z, y)$
      fin
    fin faire
  fin faire
fin faire

✎ **3.3 Appliquer l'algorithme de Floyd-Warshall-Roy pour trouver les plus courts chemins du graphe :**



Ce graphe possède une valuation négative ($f(4, 1) = -2$) et des circuits de longueur positive (comme $(1, 6, 5, 1)$). En particulier il n'y a pas de chemins les plus

long dans ce graphe.

- *étape $k = 0$*

$$Dist_0 = \begin{pmatrix} 0 & \infty & \infty & \infty & \infty & 3 \\ \infty & 0 & 15 & 1 & 11 & \infty \\ \infty & \infty & 0 & 16 & \infty & \infty \\ -2 & \infty & \infty & 0 & 2 & \infty \\ 0 & \infty & \infty & \infty & 0 & \infty \\ \infty & 9 & 11 & \infty & 19 & 0 \end{pmatrix}$$

- *étape $k = 1$*

$$Dist_1 = \begin{pmatrix} 0 & \infty & \infty & \infty & \infty & 3 \\ \infty & 0 & 15 & 1 & 11 & \infty \\ \infty & \infty & 0 & 16 & \infty & \infty \\ -2 & \infty & \infty & 0 & 2 & 1 \\ 0 & \infty & \infty & \infty & 0 & 3 \\ \infty & 9 & 11 & \infty & 19 & 0 \end{pmatrix}$$

- *étape $k = 2$*

$$Dist_2 = \begin{pmatrix} 0 & \infty & \infty & \infty & \infty & 3 \\ \infty & 0 & 15 & 1 & 11 & \infty \\ \infty & \infty & 0 & 16 & \infty & \infty \\ -2 & \infty & \infty & 0 & 2 & 1 \\ 0 & \infty & \infty & \infty & 0 & 3 \\ \infty & 9 & 11 & 10 & 19 & 0 \end{pmatrix}$$

- *étape $k = 3$*

$$Dist_3 = \begin{pmatrix} 0 & \infty & \infty & \infty & \infty & 3 \\ \infty & 0 & 15 & 1 & 11 & \infty \\ \infty & \infty & 0 & 16 & \infty & \infty \\ -2 & \infty & \infty & 0 & 2 & 1 \\ 0 & \infty & \infty & \infty & 0 & 3 \\ \infty & 9 & 11 & 10 & 19 & 0 \end{pmatrix}$$

- *étape $k = 4$*

$$Dist_4 = \begin{pmatrix} 0 & \infty & \infty & \infty & \infty & 3 \\ -1 & 0 & 15 & 1 & 3 & 2 \\ 14 & \infty & 0 & 16 & 18 & 17 \\ -2 & \infty & \infty & 0 & 2 & 1 \\ 0 & \infty & \infty & \infty & 0 & 3 \\ 8 & 9 & 11 & 10 & 12 & 0 \end{pmatrix}$$

- *étape $k = 5$*

$$Dist_5 = \begin{pmatrix} 0 & \infty & \infty & \infty & \infty & 3 \\ -1 & 0 & 15 & 1 & 3 & 2 \\ 14 & \infty & 0 & 16 & 18 & 17 \\ -2 & \infty & \infty & 0 & 2 & 1 \\ 0 & \infty & \infty & \infty & 0 & 3 \\ 8 & 9 & 11 & 10 & 12 & 0 \end{pmatrix}$$

- *étape $k = 6$*

$$Dist_6 = \begin{pmatrix} 0 & 12 & 14 & 13 & 15 & 3 \\ -1 & 0 & 13 & 1 & 3 & 2 \\ 14 & 26 & 0 & 16 & 18 & 17 \\ -2 & 10 & 12 & 0 & 2 & 1 \\ 0 & 12 & 14 & 13 & 0 & 3 \\ 8 & 9 & 11 & 10 & 12 & 0 \end{pmatrix}$$

- *la liste des prédécesseurs correspondante est*

$$Pred = \begin{pmatrix} 0 & 6 & 6 & 2 & 4 & 1 \\ 4 & 0 & 6 & 2 & 4 & 1 \\ 4 & 6 & 0 & 3 & 4 & 1 \\ 4 & 6 & 6 & 0 & 4 & 1 \\ 5 & 6 & 6 & 2 & 0 & 1 \\ 4 & 6 & 6 & 2 & 4 & 0 \end{pmatrix}$$

L'algorithme de Floyd-Warshall-Roy fonctionne sur tous les graphes mais a deux défauts :

- son temps de calcul est très long ($\sim n^3$) et il consomme beaucoup de mémoire
- il calcule l'ensemble des distances entre tout couple de sommets alors qu'on peut souvent se contenter des distance depuis un sommet particulier

Les défauts de l'algorithme de Floyd-Warshall-Roy, nous obligent à chercher d'autres algorithmes moins coûteux. Le premier d'entre eux est l'algorithme de Bellman-Ford.

Définition 3.6 (algorithme de Bellman-Ford-Kalaba) *Soit un graphe orienté valué $G = (S, A, f)$, d'ordre n et de taille m, et x un sommet de G. L'algorithme de Bellman calcule deux matrices de taille $1 \times n$*

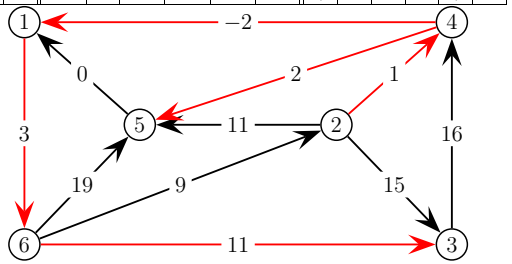
- *Dist matrice des distances telle que $Dist(y)$ = distance optimale de x à y*
- *Pred matrice des prédécesseurs telle que $Pred(y)$ = prédécesseur de y dans le chemin optimal depuis x*

Pour le plus court chemin l'algorithme s'écrit :

fonction $[Dist, Pred] =$ BELLMAN(G, s)
Initialisation : $n =$ nombre de sommets de G
$Pred =$ tableau des prédécesseurs initialisé à 0
$Dist =$ tableau des distances initialisé à $+\infty$ (sauf $Dist(s) = 0$)
$W =$ matrice des poids des arcs (∞ si l'arc n'existe pas)
Traitement : $k = 1$
tant que $k \leq n$ et il y a eu des modifications à l'étape précédente **faire**
  pour tout sommet x **faire**
    pour tout y successeur de x **faire**
      si $Dist(x) + W(x, y) < Dist(y)$
        alors modifier $Dist(y)$ et $Pred(y) = x$
      fin
    fin faire
  fin faire
  $k = k + 1$
fin faire

✎ **3.4 Appliquer l'algorithme de Bellman-Ford-Kalaba :** *pour trouver les plus courts chemins du graphe G depuis le sommet 2 et faire apparaître l'arbre de parcours sur le graphe*

		Dist						*Pred*					
k	x	*1*	*2*	*3*	*4*	*5*	*6*	*1*	*2*	*3*	*4*	*5*	*6*
0	*0*	∞	*0*	∞	∞	∞	∞	*0*	*0*	*0*	*0*	*0*	*0*
1	*2*	∞	*0*	*15*	*1*	*11*	∞	*0*	*0*	*2*	*2*	*2*	*0*
1	*4*	*-1*	*0*	*15*	*1*	*3*	∞	*4*	*0*	*2*	*2*	*4*	*0*
2	*1*	*-1*	*0*	*15*	*1*	*3*	*2*	*4*	*0*	*2*	*2*	*4*	*1*
2	*6*	*-1*	*0*	*13*	*1*	*3*	*2*	*4*	*0*	*6*	*2*	*4*	*1*



l'algorithme de Bellman-Ford **fonctionne sur tout les graphes** mais a un temps de calcul encore relativement long ($\sim n^2 \times m \approx n^3$) mais consomme moins de mémoire.

L'algorithme de Bellman-Ford-Kalaba reste encore coûteux et complexe. Dans de nombreux cas on peut simplifier la recherche d'un chemin optimal à condition que le graphe possède certaines propriétés. Le premier exemple d'une telle situation est l'algorithme de Dijkstra, que l'on peut utiliser pour la recherche de chemins minimaux dans un graphe à valuations positives.

Définition 3.7 (algorithme de Dijkstra-Moore) *Soit un graphe orienté valué $G = (S, A, f)$, d'ordre n et de taille m, et x un sommet de G. L'algorithme de Dijkstra calcule deux matrices de taille $1 \times n$*

- *Dist matrice des distances telle que $Dist(y)$ = distance optimale de x à y*
- *Pred matrice des prédécesseurs telle que $Pred(y)$ = prédécesseur de y dans le chemin optimal depuis x*

Pour le plus court chemin l'algorithme s'écrit :

fonction $[Dist, Pred] =$ DIJKSTRA(G, s)
Initialisation :
$n =$ nombre de sommets de G
$Pred =$ tableau des prédécesseurs initialisé à 0
$Dist =$ tableau des distances initialisé à $+\infty$ (sauf $Dist(s) = 0$)
$W =$ matrice des poids des arcs (∞ si l'arc n'existe pas)
$C = \{1; 2; \ldots; n\}$ (liste des sommets restant à traiter)
$D = \emptyset$ (liste des sommets déjà traités)
Traitement :
tant que $C \neq \emptyset$ **faire**
  $x =$ sommet de C le plus proche de s
  retirer x de C et le mettre dans D
  pour tout sommet $y \in C$ **faire**
    si $Dist(x) + W(x, y) < Dist(y)$
      alors modifier $Dist(y)$ et $Pred(y) = x$
    fin
  fin faire
fin faire

L'algorithme de Dijkstra a un temps d'exécution assez rapide ($\sim n^2$) mais a deux défauts :

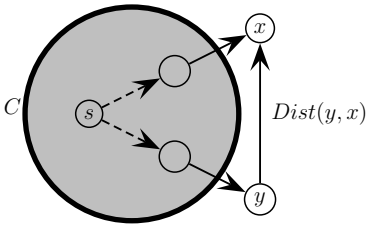
- il ne s'applique qu'aux graphes à **valuations positives**
- il ne marche que pour trouver **les plus courts chemins**

Preuve : L'algorithme de Dijkstra est basé sur le fait qu'à chaque étape de l'algorithme, pour tout élément x de C, $Dist(x)$ est la plus petit distance de s à x. L'efficacité de l'algorithme de Dijkstra peut donc se démontrer par récurrence :

- **hypothèse de récurrence** $\mathscr{P}_k$ à une étape k de l'algorithme : « pour tout élément x de D, $Dist(x)$ est la plus petit distance de s à x »
- $\mathscr{P}_0$ **est vraie** : $D = \{s\}$ et $Dist(s) = 0$ (par initialisation) est bien minimale.
- $\mathscr{P}_k \Rightarrow \mathscr{P}_{k+1}$ à l'étape k de l'algorithme les distances dans $Dist$ sont minimales. Soit x le $k+1^{\text{ième}}$ sommet traité par l'algorithme, x est ajouté à D. Supposons qu'il existe un plus court chemin de la source à x passant par $y \notin D$, on aurait donc :

$$Dist(s,y) + Dist(y,x) \leq Dist(s,x)$$

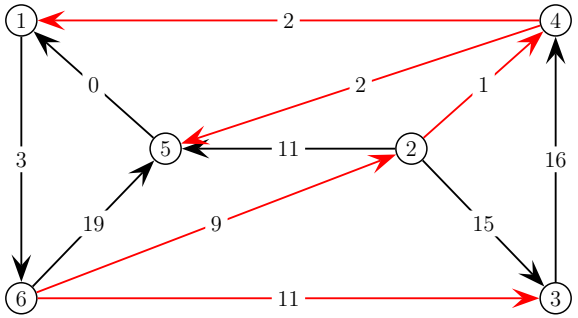
si les valuations sont toutes positives $Dist(y,x) \geq 0$ d'où l'on tire que $Dist(s,y) \leq Dist(s,x)$. On aurait donc du traiter y avant x dans l'algorithme!!! Ce qui n'est pas possible.



□

L'algorithme de Dijkstra ne peut pas être utilisé sur le graphe G à cause de la valuation négative $f(4,1) = -2$. Si on change cette valuation en un nombre positif on peut alors calculer les distances par l'algorithme de Dijkstra.

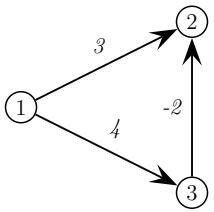
✎ **3.5 Appliquer l'algorithme de Dijkstra-Moore sur le graphe** G_D *pour calculer les chemins les plus courts depuis le sommet 6*



x	D	C	$Dist$						$Pred$					
			1	*2*	*3*	*4*	*5*	*6*	*1*	*2*	*3*	*4*	*5*	*6*
0	{}	{*1*;*2*;*3*;*4*;*5*;*6*}	∞	∞	∞	∞	∞	*0*	*0*	*0*	*0*	*0*	*0*	*0*
6	{*6*}	{*1*;*2*;*3*;*4*;*5*}	∞	*9*	*11*	∞	*19*	*0*	*0*	*6*	*6*	*0*	*6*	*0*
2	{*6*;*2*}	{*1*;*3*;*4*;*5*}	∞	*9*	*11*	*10*	*19*	*0*	*0*	*6*	*6*	*2*	*6*	*0*
4	{*6*;*2*;*4*}	{*1*;*3*;*5*}	*12*	*9*	*11*	*10*	*12*	*0*	*4*	*6*	*6*	*2*	*4*	*0*
3	{*6*;*2*;*4*;*3*}	{*1*;*5*}	*12*	*9*	*11*	*10*	*12*	*0*	*4*	*6*	*6*	*2*	*4*	*0*
1	{*6*;*2*;*4*;*3*;*1*}	{*5*}	*12*	*9*	*11*	*10*	*12*	*0*	*4*	*6*	*6*	*2*	*4*	*0*
5	{*6*;*2*;*4*;*3*;*1*;*5*}	{}	*12*	*9*	*11*	*10*	*12*	*0*	*4*	*6*	*6*	*2*	*4*	*0*

✎ **3.6 contre-exemple pour l'algorithme de Dijkstra pour des valuations négatives** *Pour le graphe suivant il est facile de voir que Dist =* | *0* | *2* | *4* |

contrairement à ce que donne l'algorithme de Dijkstra :



s	D	C	$Dist$			$Pred$		
			1	*2*	*3*	*1*	*2*	*3*
0	{}	{*1*;*2*;*3*}	*0*	∞	∞	*0*	*0*	*0*
1	{*1*}	{*2*;*3*}	*0*	*3*	*4*	*0*	*1*	*1*
2	{*1*;*2*}	{*3*}	*0*	*3*	*4*	*0*	*1*	*1*
3	{*1*;*2*;*3*}	{}	*0*	*3*	*4*	*0*	*1*	*1*

Le protocole de routage **OSPF** (Open Shortest Path First), de type *"link state"*, utilise l'algorithme de Dijkstra pour calculer le chemin optimal dans un réseau.

De même pour un graphe décomposable en niveaux, quelque soit ses valuations et quelque soit le type de chemin recherché, on pourra utiliser un autre algorithme : l'algorithme de Bellman-Kalaba.

Définition 3.8 (algorithme de Bellman simplifié) *Soit un graphe orienté valué $G = (S, A, f)$, d'ordre n et de taille m, et x un sommet de G. L'algorithme de Bellman simplifié calcule deux matrices de taille $1 \times n$*

- *Dist matrice des distances telle que $Dist(y)$ = distance optimale de x à y*
- *Pred matrice des prédécesseurs telle que $Pred(y)$ = prédécesseur de y dans le chemin optimal depuis x*

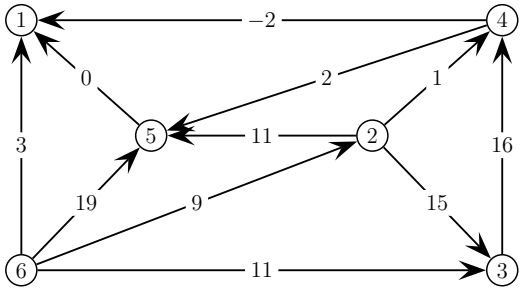
Pour le plus court chemin l'algorithme s'écrit :

fonction $[Dist, Pred] =$ BELLMAN_SIMPLE(G, s)
Initialisation :
$Pred$ = tableau des prédécesseurs initialisé à 0
$Dist$ = tableau des distances initialisé à $+\infty$ (sauf $Dist(s) = 0$)
W = matrice des poids des arcs (∞ si l'arc n'existe pas)
Faire la décomposition en niveau de G
Traitement :
pour tout $k = niveau(s) + 1$ **jusqu'à** niveau maximum **faire**
pour tout sommet x du niveau k **faire**
pour tout sommet y prédécesseur de x **faire**
si $Dist(y) + W(y,x) < Dist(x)$
alors modifier $Dist(x)$ et $Pred(x) = y$
fin
fin faire
fin faire
fin faire

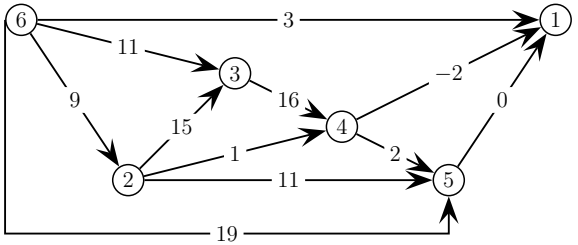
l'algorithme de Bellman simplifié a un temps d'exécution assez rapide ($\sim n^2$) mais ne s'applique qu'aux graphes **décomposable en niveaux** (donc sans circuits)

On ne peut pas appliquer l'algorithme de Bellman simplifié sur le graphe de départ à cause des circuits $(6,3,4,1,6)$, $(6,5,1,6)$,.... Si on inverse l'arc $(1,6)$ on aura plus aucun circuit et là on peut appliquer notre algorithme.

✎ **3.7 Appliquer l'algorithme de Bellman simplifié** *sur le graphe G_K pour calculer les chemins les plus courts depuis le sommet 2*



la décomposition en niveaux donne :



ce qui donne :

	Dist						*Pred*					
niv	*1*	*2*	*3*	*4*	*5*	*6*	*1*	*2*	*3*	*4*	*5*	*6*
1	∞	*0*	∞	∞	∞	∞	*0*	*0*	*0*	*0*	*0*	*0*
2	∞	*0*	*15*	∞	∞	∞	*0*	*0*	*2*	*0*	*0*	*0*
3	∞	*0*	*15*	*1*	∞	∞	*0*	*0*	*2*	*2*	*0*	*0*
4	∞	*0*	*15*	*1*	*3*	∞	*0*	*0*	*2*	*2*	*4*	*0*
5	*-1*	*0*	*15*	*1*	*3*	∞	*4*	*0*	*2*	*2*	*4*	*0*

Les algorithmes de Floyd-Warshall-Kalaba et de Bellman (mais pas l'algorithme de Dijkstra) permettent aussi de calculer les **chemins de longueur maximale** dans un graphe. Dans ce cas il suffit de faire les modifications suivantes dans les algorithmes données :

- changer $Dist(y)+W(y,x) < Dist(x)$ en $Dist(y)+W(y,x) > Dist(x)$ dans les conditionnelles
- initialiser les distances à $-\infty$ au lieu de à $+\infty$

Dans la pratique la liste *Pred* calculé par ces différents algorithmes correspond à la table de routage décrivant les meilleurs chemins.

3.3 Ordonnancement et gestion de projet

La planification d'un projet regroupe l'ensemble des méthodes permettant de trouver l'organisation optimale du projet (durée minimale, identification des tâches critiques, ...). Ce problème peut être modélisé à l'aide d'un **réseau PERT** (« *project evaluation and review technique* ») et résolu à l'aide de la théorie des graphes. Historiquement le PERT a été créé en 1956 à la demande de la marine américaine, pour planifier la durée de son programme de construction de missiles balistiques nucléaires miniaturisés Polaris (qui nécessitait l'intervention de 9000 sous-traitants et 250 fournisseurs) afin de rattraper le retard en matière de balistique, après le choc de la « crise de Spoutnik », par rapport à l'URSS. L'utilisation de la théorie des graphes a permis de réduire la durée du projet à 4 ans alors que le délai initial était estimé à 7 ans !

Définition 3.9 (projet) *un projet $\mathscr{P}$ est constitué par :*

- *un ensemble de tâches à réaliser $(\mathscr{A}_i)_{i=1,\dots,n}$ avec, pour chaque tâche, une date de commencement t_i et une durée d_i,*
- *un ensemble de « contraintes » sur les tâches du projets, contraintes qui peuvent s'exprimer par des inégalités faisant intervenir les dates $(t_i)_{i=1,\dots,n}$ et les durées $(d_i)_{i=1,\dots,n}$*

Trouver un **ordonnancement** *pour le projet consister à calculer, à partir des durées $(d_i)_{i=1,\dots,n}$, des dates de commencement $(t_i)_{i=1,\dots,n}$ qui soient* **compatibles** *avec les contraintes du projet.*

✎ **3.8 un exemple de projet et de contraintes :** *On le présente en général sous forme d'un tableau :*

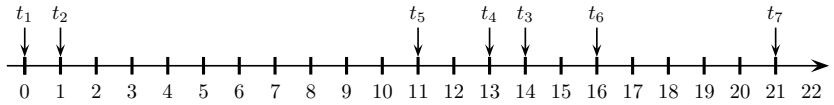
Tâches	*Opérations et contraintes*	*Durée en jours*
1	**Début du projet**	*0*
2	*aucune contrainte*	*11*
3	*commence au plus tôt 1 jour après le début du projet* *commence au plus tard 8 jours après le début de (4)*	*5*
4	*commence au plus tôt 1 jours après la fin de (2)*	*8*
5	*commence au plus tôt 1 jours avant la fin de (2)*	*5*
6	*commence au plus tôt après le début de (4) et après la fin de (5)*	*4*
7	**Fin projet**	*0*

Il est difficile de concevoir un planning directement à partir d'une liste de contraintes, par exemple :

- *l'ordonnancement*

i	*1*	*2*	*3*	*4*	*5*	*6*	*7*
t_i	*0*	*1*	*14*	*13*	*11*	*16*	*21*

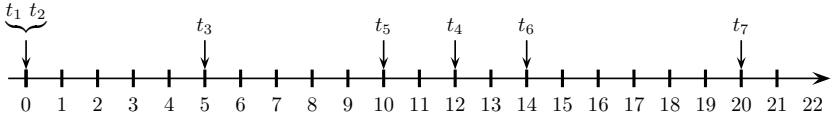
est compatible



- *l'ordonnancement*

i	1	2	3	4	5	6	7
t_i	0	0	5	12	10	14	20

n'est pas compatible



car on doit avoir plus de 5 jours entre le début de la tâche 5 et le début de la tâche 6

Pour rendre tout cela plus lisible nous allons traduire les contraintes sous forme d'inégalités entre les dates de commencement de chaque tâche. Il faudra pour cela bien identifier les différents types de contrainte que l'on rencontre.

Proposition 3.10 (type de contraintes) *On traduira les différentes contraintes en inégalités de la manière suivante :*

contrainte au plus tôt « $\mathscr{A}_j$ *commence au plus tôt* δ *après le début de* $\mathscr{A}_i$ »

$$t_j - t_i \geq \delta$$

contrainte au plus tard « $\mathscr{A}_j$ *commence au plus tard* δ *après le début de* $\mathscr{A}_i$ »

$$t_j - t_i \leq \delta \iff t_i - t_j \geq -\delta$$

contraintes implicites « *toute tâche* $\mathscr{A}_i$ *doit démarrer au plus tôt au début du projet* ($\mathscr{A}_1$) *et finir au plus tard à la fin du projet* ($\mathscr{A}_n$) »

$$t_i - t_1 \geq 0 \quad \text{et} \quad t_n - t_i \geq d_i$$

✎ **3.9 Traduire les contraintes du projet en inéquations :** *On essaie de traduduire chaque contrainte par une inégalité de la forme* $t_j - t_i \geq \delta$

contraintes	*équation*
(3) commence au plus tôt 1 jour après le début de (1)	$t_3 - t_1 \geq 1$
(3) commence au plus tard 8 jours après le début de (4)	$t_4 - t_3 \geq -8$
(4) commence au plus tôt 1 jours après la fin de (2)	$t_4 - t_2 \geq 11 + 1 = 12$
(5) commence au plus tôt 1 jours avant la fin de (2)	$t_5 - t_2 \geq 11 - 1 = 10$
(6) commence au plus tôt après le début de (4)	$t_6 - t_4 \geq 0$
(6) commence au plus tôt après la fin de (5)	$t_6 - t_5 \geq d_5 = 5$

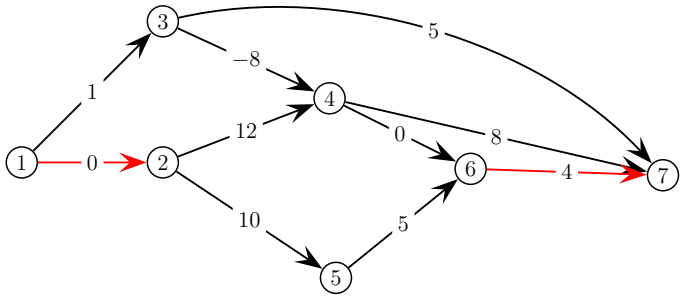
À partir des équations obtenues on va pouvoir représenter ce projet par un graphe orienté et valué.

Définition 3.11 (graphe potentiel-tâche, réseau PERT) *On associe à un problème d'ordonnancement un graphe orienté et valué* $G = (S, A, f)$ *tel que :*

- *Chaque tâche* $\mathscr{A}_i$ *du projet sera représenté par un sommet* x_i *du graphe*
- *chaque contrainte* $t_j - t_i \geq d$ *du projet sera représentée par un arc* (x_i, x_j) *de valuation* $f(x_i, x_j) = d$

On ajoutera éventuellement des **contraintes implicites** *pour que chaque tâche dans le graphe appartienne à au moins un chemin reliant la tâche de début de projet et celle de fin de projet.*

✎ **3.10 Représenter le projet par un graphe :**



ici on a du ajouter deux arcs correspondant à des contraintes implicites :

- *l'arc* $(1, 2)$ *avec le poids* $0 \iff t_2 - t_1 \geq 0$ *(*$\mathscr{A}_2$ *débute après le début)*
- *l'arc* $(6, 7)$ *avec le poids* $4 \iff t_7 - t_6 \geq d_6 = 4$ *(*$\mathscr{A}_7$ *commence au plus tôt à la fin de* $\mathscr{A}_6$*)*

Les valuations du graphe potentiel-tâche peuvent très bien être négatives. Les contraintes « au plus tard » feront apparaître des circuits dans le graphe potentiel-tâche.

Théorème 3.12 (ordonnancement au plus tôt) *L'ordonnancement au plus tôt d'un projet consiste à trouver les dates de commencement* $(t_i)_{i=1,\dots,n}$ *de chaque tâche telles que le projet soit fini le plus rapidement possible. Pour calculer cet ordonnancement il suffit de calculer les* **chemins les plus longs dans le graphe** G *(relativement à* f*) depuis le sommet correspondant au début du projet. Les distances obtenues donnent les dates de commencement de chaque tâche :*

$$t_i = Dist(x_i), \quad \forall i = 1, \dots, n$$

Si x_n *correspond à la tâche de fin du projet alors la durée totale du projet est* t_n*.*

✎ **3.11 Calculer l'ordonnancement au plus tôt :** *ici le graphe de* G *ne possède pas de circuits on peut utiliser l'algorithme de Bellman simplifié (le graphe est déjà décomposé en niveaux) ce qui donne :*

	Dist							*Pred*						
niv	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*1*	*2*	*3*	*4*	*5*	*6*	*7*
0	0	-∞	-∞	-∞	-∞	-∞	-∞	0	0	0	0	0	0	0
1	0	0	1	-∞	-∞	-∞	-∞	0	1	1	0	0	0	0
2	0	0	1	12	10	-∞	-∞	0	1	1	2	2	0	0
3	0	0	1	12	10	15	-∞	0	1	1	2	2	5	0
4	0	0	1	12	10	15	20	0	1	1	2	2	5	4

La durée minimale de réalisation du projet est donc de 20 jours qui correspond à l'ordonnancement :

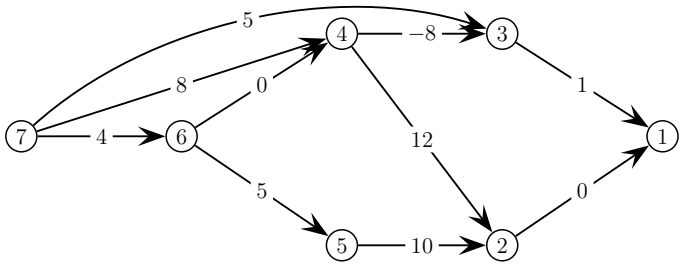
i	1	2	3	4	5	6	7
t_i	0	0	1	12	10	15	20

Lorsqu'il n'y a pas de circuits dans le graphe on peut utiliser l'algorithme de Bellman simplifié pour calculer l'ordonnancement correspondant. Dans les autres cas il faudra utiliser l'algorithme de Bellman général !

Théorème 3.13 (ordonnancement au plus tard) *Pour une durée T, l'ordonnancement au plus tard d'un projet consiste à trouver les dates les plus tardives de commencement $(t_i)_{i=1,\dots,n}$ de chaque tâche telles que la durée du projet soit au maximum de T. Pour calculer cet ordonnancement il suffit de calculer les* **chemins les plus longs dans le graphe réciproque** *G^{-1} (toujours relativement à f) depuis le sommet correspondant à la fin du projet. Les distances obtenues donnent le temps restant à partir du commencement d'une tâche jusqu'à la fin du projet, en d'autres termes la tâche i doit démarrer au plus tard à la date :*

$$t_i = T - Dist(x_i), \quad \forall i = 1, \dots, n$$

3.12 Calculer l'ordonnancement au plus tard pour $T = 22$ jours : *on calcule d'abord le graphe réciproque de G (en conservant les poids des arcs)*



on calcule les chemins les plus longs depuis le sommet 7 (avec l'algorithme de Bellman simplifié puisque G^{-1} est décomposé en niveaux) :

	Dist							*Pred*						
niv	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*1*	*2*	*3*	*4*	*5*	*6*	*7*
0	-∞	-∞	-∞	-∞	-∞	-∞	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*
1	-∞	-∞	-∞	-∞	-∞	*4*	*0*	*0*	*0*	*0*	*0*	*0*	*7*	*0*
2	-∞	-∞	-∞	*8*	*9*	*4*	*0*	*0*	*0*	*0*	*7*	*6*	*7*	*0*
3	-∞	*20*	*5*	*8*	*9*	*4*	*0*	*0*	*4*	*7*	*7*	*6*	*7*	*0*
4	*20*	*20*	*5*	*8*	*9*	*4*	*0*	*2*	*4*	*7*	*7*	*6*	*7*	*0*

on en déduit l'ordonnancement au plus tard pour la durée $T = 22$:

i	1	2	3	4	5	6	7
$Dist(i)$	20	20	5	8	9	4	0
$t_i = T - Dist(i)$	2	2	17	14	12	18	22

Si le graphe G n'a pas de circuits alors il en va de même pour le graphe réciproque G^{-1}. On peut donc utiliser l'algorithme de Bellman simplifié pour calculer ces ordonnancements, mais attention, les niveaux de G et G^{-1} ne sont pas forcément les mêmes !

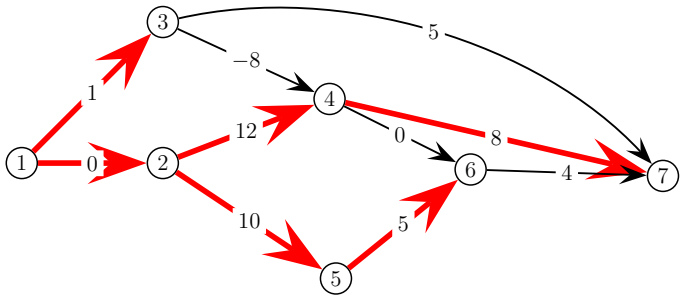
Les ordonnancements au plus tôt et au plus tard permettent de visualiser les tâches les plus critiques du projet, pour lesquelles on ne peut pas prendre de retard sans répercution sur la date de fin du projet.

Théorème 3.14 (chemin critique) *Si T est la durée minimale du projet (calculée via l'ordonnancement au plus tôt) le chemin le plus long depuis la tâche de début de projet jusqu'à celle de fin de projet est appelé* **chemin critique** *tout retard sur la réalisation d'une des tâche de ce chemin critique entraîne un allongement de la durée du projet. Pour les autres sommet du graphe potentiel-tâche il existe une* **marge** *sur la durée de réalisation de la tâche, marge définie par :*

$$\begin{aligned} marge = \ll & \text{ date de début pour l'ordonnancement au plus tard} \\ & - \text{ date de début pour l'ordonnancement au plus tôt} \gg \end{aligned}$$

3.13 Représenter le chemin critique et calculer les marges pour $T = 22$: *le chemin critique $(1, 2, 4, 7)$ s'obtient donc en faisant apparaître l'arbre de parcours correspondant à la liste de prédécesseurs calculée avec l'ordonnancement au plus tôt : $Pred =$*

0	1	1	2	2	5	4



On a donc une marge sur la réalisation des tâches 3 et 6 mais pas pour les tâches 2,4 et 5 comme le montre le calcul des marges :

i	*1*	*2*	*3*	*4*	*5*	*6*	*7*
au plus tôt	*0*	*0*	*1*	*12*	*10*	*15*	*20*
au plus tard	*2*	*2*	*17*	*14*	*12*	*18*	*22*
marge	*2*	*2*	*16*	*2*	*2*	*3*	*2*

La marge est donc au minimum de 2 jours (normal puisqu'on a fixée une durée $T = 22$ alors que la durée minimale est de 20 jours). En cas de besoin (absences, problèmes tecniques,...) on pourra donc prendre des ressources sur la réalisation de la tâche 3 pour aider sur les autres tâches critiques.

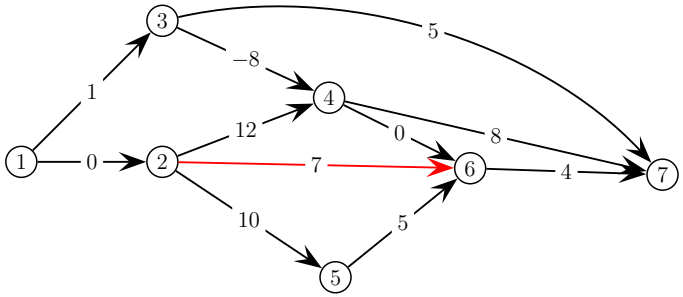
Il peut y avoir plusieurs chemins critiques dans un graphe potentiel tâche.

Définition 3.15 (contrainte redondante) *On dira qu'une contrainte d'un projet est redondante si dans le réseau PERT $G = (S, A, f)$ associé elle correspond à un arc (x_i, x_j) tel qu'il existe un chemin de x_i à x_j de longueur δ (relativement à f) telle que $f(x_i, x_j) < \delta$.*

3.14 Ajout d'une contrainte redondante au projet : *si on ajoute la contrainte :*

La tache (2) doit commencer au plus tôt 1 semaine avant la tâche (6)

on obtient un nouveau graphe :



pourtant on comprend bien que cette contrainte est inutile puisque la tâche 2 doit de toute façon démarrer au moins 15 jours avant la tâche 6 à cause du chemin $(2, 5, 6)$. Dans la pratique de nombreuses contraintes implicites sont ignorées car redondantes.

3.4 Flots dans les réseaux

L'une des applications les plus importantes de la théorie des graphes est l'optimisation des flots dans les réseaux. Pour étudier ce problème il faut d'abord définir ce qu'est un réseau de transport.

Définition 3.16 (réseau de transport) *Un réseau de transport est un quadruplet (G, s, t, C) où*

- *$G = (S, A)$ est un graphe orienté simple*
- *s est un sommet sans prédécesseur appelé « source »*
- *t est un sommet sans successeur appelé « puits »*
- *$C : A \longrightarrow \mathbb{R}^+$ est une valuation positive de G appelée « capacité »*

pour un ensemble de sommets $W \subset S$ on notera

arcs entrants en W : *l'ensemble d'arcs $W^- = \{(x, y) \in A | x \notin W \text{ et } y \in W\}$*

arcs sortants de W : *l'ensemble d'arcs $W^+ = \{(x, y) \in A | x \in W \text{ et } y \notin W\}$*

si $W = \{z\}$ est un singleton on notera ces ensembles simplement z^+ et z^-.

Le réseau de transport va donc servir de support à « un flot ». Ce flot est en fait une seconde valuation attachée au graphe représentant le réseau.

Définition 3.17 (flot) *Soit (G, s, t, C) un réseau de transport, alors un flot sur ce graphe $G = (S, A)$ est une valuation positive $f : A \longrightarrow \mathbb{R}^+$ qui vérifie :*

- *le flot ne dépasse pas la capacité*

$$\forall (x, y) \in A, \quad 0 \leq f(x, y) \leq C(x, y)$$

- *la* **loi des nœuds**

$$\forall z \in S \setminus \{s; t\}, \quad \sum_{(x,z) \in z^+} f(x, z) = \sum_{(z,y) \in z^-} f(z, y)$$

Pour tout sommet z on appelle :

flot entrant en z : *la quantité* $f^-(z) = \sum_{(x,z) \in z^-} f(x, z)$

flot sortant de z : *la quantité* $f^+(z) = \sum_{(z,y) \in z^+} f(z, y)$

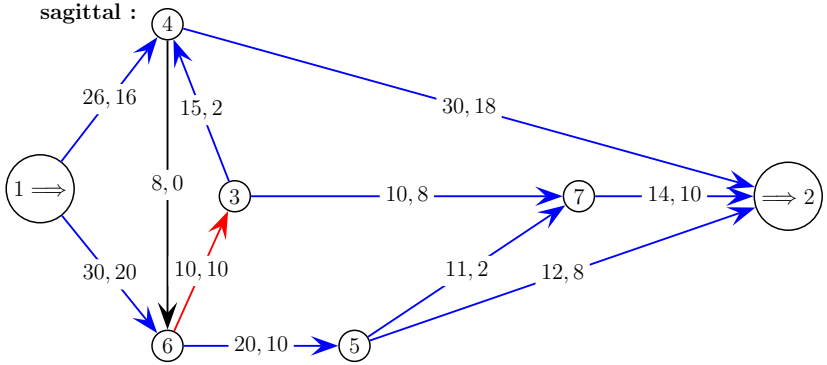
Et on dira que le flot f sur l'arc (x, y) est :

saturé : *si $f(x, y) = C(x, y)$*

nul : *si $f(x, y) = 0$*

La contrainte la plus importante pour un flot est la loi des nœuds. Elle a été posée par le physicien allemand Gustav Kirchhoff en 1845, lorsqu'il a établi les règles de calcul des intensités des courants dans un circuit électrique. Elle exprime simplement la conservation du flux :« le flux entrant= le flux sortant ».

✎ **3.15 Représenter un réseau de transport et un flot sur un diagramme sagittal :**



- *la source est le sommet* 1 *et le puits est le sommet* 2
- *sur chaque arc*
  - *le premier nombre indique la capacité*
  - *le second nombre indique le flot*

  par exemple l'arc $(3,7)$ *à une capacité* $C(3,7)=10$ *et un flot* $f(3,7)=8$
- *le flot de l'arc* $(6,3)$ *est saturé et le flot de l'arc* $(4,6)$ *est nul*
- *pour* $W=\{5;6\}$
  - $W^-=\{(1,6);(4,6)\}$ *mais ne contient pas* $(6,5)$ *!*
  - $W^+=\{(6,3);(5,7);(5,2)\}$
- *la loi des nœuds se vérifie en chaque sommet, par exemple pour* 3
  - $f^-(3)=10$ *car* $3^-=\{(6,3)\}$
  - $f^+(3)=8+2=10$ *car* $3^+=\{(3,4);(3,7)\}$

✎ **3.16 Exemples concrets de réseaux de transport et de flot :** *on peut facilement modéliser les réseaux qu'on rencontre dans la vie courante par un graphe orienté comportant une source et un puits, une capacité et un flot :*

réseau électrique : *les arcs sont des lignes électriques, les flots des quantités de courant, les sources sont les centrales (hydrauliques, nucléaire, éoliennes, à charbon,...) où est produite l'électricité, les puits sont des endroits où l'électricité est consommée, les capacités représentent l'intensité du courant à partir de laquelle une ligne va fondre !*

réseau de distribution de l'eau : *les arcs sont des canalisations, les flots des quantités d'eau, les capacités le débit maximal d'une canalisation, les sources peuvent être des nappes phréatiques, des barrages ..., les puits sont des endroits où l'eau est rejeté, une station d'épuration dans le meilleur des cas !*

réseau informatique : *les arcs peuvent représenter des câbles de transmission (câbles éthernet, câbles téléphoniques, fibre optique, où le vide pour le wifi !), les flots sont des quantités d'information, les capacités le débit maximal d'une connexion (Mo/s), les sommets sont des routeurs, la source une machine émettant des informations (un mail par exemple), le puits la machine destinataire de ces informations,*

réseau routier : *les arcs sont des routes, les flots des quantités de voitures, les sommets des carrefours, les capacités le nombre de voitures à partir duquel il va se former un bouchon, les sources les endroits d'où partent les véhicules (la zone Pégase par exemple), les puits les endroits où vont les voitures (maison/appartement des gens partant de la zone Pégase).*

Le vocabulaire utilisé dans ce chapitre est donc directement hérité de termes utilisés dans chacun des réseaux que nous rencontrons dans la vie courante.

il existe des définitions plus générales de flots et de réseau de transport. Par exemple, dans certaines référence, on parle parfois de réseaux de transport avec **plusieurs sources ou plusieurs puits**. On pourra toujours, dans ces cas, se ramener à la définition de ce cours (source et puits unique) en ajoutant :

- une source s' et des arcs vers toutes les autres sources du problème (s',s_i) avec une capacité égale à celle sortant de s_i $(\sum_{(s_i,x)\in s_i^+} C(s_i,x)=C(s',s_i))$
- un puits t' et des arcs depuis toutes les autres puits du problème (t_j,t') avec une capacité égale à celle entrant en t_j $(\sum_{(x,t_j)\in t_j^-} C(x,t_j)=C(t_j,t'))$

Dans d'autres définitions le **réseau de transport est muni de deux capacités** $C_{\max}$ **et** $C_{\min}$ (éventuellement négatives) avec un flot vérifiant (en plus de la loi des nœuds) :

$$\forall (x,y)\in A,\quad C_{\min}(x,y)\leq f(x,y)\leq C_{\max}(x,y)$$

si $C_{\min}$ vérifie la loi des nœuds on peut se ramener au cas traité dans ce cours en remplaçant :

- les capacités $C_{\min}$ et $C_{\max}$ par une capacité définie par

$$\forall (x,y)\in A,\quad C(x,y)=C_{\max}(x,y)-C_{\min}(x,y)$$

- le flot f par un nouveau flot $\tilde{f}$ défini par

$$\forall (x,y)\in A,\quad \tilde{f}(x,y)=f(x,y)-C_{\min}(x,y)$$

Dans les problèmes de réseau de transport les flots entrants et sortants d'un ensemble de sommets jouent un rôle très important, ce qui nous amène à définir la notion de coupe.

Définition 3.18 (coupe) *Soit* (G,s,t,C) *un réseau de transport, une coupe est un ensemble de sommets contenant la source :*

$$W\subset S,\quad s\in W$$

On appelle **capacité de la coupe** W *la quantité :*

$$C_W=\sum_{(x,y)\in W^+} C(x,y)$$

Une coupe est donc un ensemble qui « coupe » en deux le graphe avec d'un coté la source et de l'autre le puits. La coupe permet de définir la valeur du flot, c'est à dire la quantité qui a été transporté de la source jusqu'au puits. La capacité des différentes coupes impose une limite maximale au flot dans un réseau de transport.

Proposition 3.19 (Lemme de la coupe) *Soit (G, s, t, C) un réseau de transport et f un flot sur ce réseau alors pour toute coupe W on a*

$$f^+(s) = \left(\sum_{(x,y)\in W^+} f(x,y)\right) - \left(\sum_{(x,y)\in W^-} f(x,y)\right) = f^-(t)$$

cette valeur est appelée **valeur du flot** *f est notée $V(f)$ et pour toute coupe on a*

$$V(f) \leq C_W$$

Preuve : la démonstration se fait par récurrence sur le nombre de sommets de la coupe

au départ $W = \{s\}$ **:** donc $s^- = \emptyset$ car s n'a pas de prédécesseur

$$f^+(s) = \underbrace{\left(\sum_{(x,y)\in s^+} f(x,y)\right)}_{=f^+(s) \text{ par définition}} = \left(\sum_{(x,y)\in s^+} f(x,y)\right) - \underbrace{\left(\sum_{(x,y)\in s^-} f(x,y)\right)}_{=0 \text{ car } s^- = \emptyset}$$

$W' = W \cup \{z\}$ **:** on ajoute un sommet z à une coupe plus petite W

$$\begin{aligned}
&\left(\sum_{(x,y)\in W'^+} f(x,y)\right) - \left(\sum_{(x,y)\in W'^-} f(x,y)\right) \\
&= \left(\sum_{(x,y)\in W^+} f(x,y) + \sum_{(z,y)\in z^+} f(z,y)\right) - \left(\sum_{(x,y)\in W^-} f(x,y) + \sum_{(x,z)\in z^-} f(x,z)\right) \\
&= \underbrace{\left(\sum_{(x,y)\in W^+} f(x,y) - \sum_{(x,y)\in W^-} f(x,y)\right)}_{=f^+(s) \text{ pour la coupe } W} + \underbrace{\left(\sum_{(z,y)\in z^+} f(z,y) - \sum_{(x,z)\in z^-} f(x,z)\right)}_{=0 \text{ d'après loi des nœuds}} \\
&= f^+(s)
\end{aligned}$$

attention, lors du passage de la première à la deuxième ligne du calcul il faut remarquer que :

- certains arcs $(z,y) \notin W'^-$ mais dans ce cas $(z,y) \in z^+$ et $(z,y) \in W^-$ et les deux termes $f(z,y)$ s'annulent mutuellement dans la somme,
- certains arcs $(x,z) \notin W'^+$ mais dans ce cas $(x,z) \in z^-$ et $(x,z) \in W^+$ et les deux termes $f(x,z)$ s'annulent mutuellement dans la somme,

ce qui justifie l'égalité finale.

pour la dernière coupe $W = S \setminus \{t\}$ **:** comme $t^+ = \emptyset$ (car t n'a pas de successeur) on a :

$$f^+(s) = \underbrace{\left(\sum_{(x,y)\in W^+} f(x,y)\right)}_{=f^-(t) \text{ car } W^+ = t-} - \underbrace{\left(\sum_{(x,y)\in W^-} f(x,y)\right)}_{=0 \text{ car } t \text{ n'a pas de successeur}} = f^-(t)$$

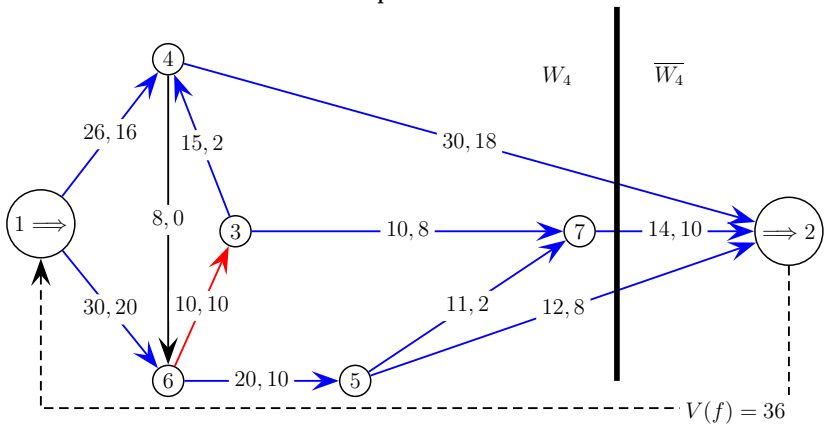
on peut donc définir la valeur du flot comme étant $V(f) = f^+(s) = f^-(t)$. Il ne reste plus qu'à justifier que la valeur du flot est majoré par la capacité de chaque coupe. Ce dernier résultat découle du contrôle du flot par les capacités, en effet pour une coupe W :

$$\begin{aligned}
V(f) &= \left(\sum_{(x,y)\in W^+} f(x,y)\right) + \left(\sum_{(x,y)\in W^-} -f(x,y)\right) \\
&\leq \underbrace{\left(\sum_{(x,y)\in W^+} C(x,y)\right)}_{\text{car } f(x,y)\leq C(x,y)} + \underbrace{\left(\sum_{(x,y)\in W^-} 0\right)}_{\text{car } -f(x,y)\leq 0} = C_W
\end{aligned}$$

□

Dans certains ouvrages sur les flots, on ajoute au graphe un arc fictif, partant du puits t pour rejoindre la source s, avec la valuation $f(s,t) = V(f)$, de telle sorte que s et t vérifient aussi la loi des nœuds.

3.17 Calcul du flot et coupe minimale :



la valeur du flot actuel est de 36 (sur l'arc fictif « de retour ») :

$$V(f) = 36 = f^+(1) = 16 + 20 = f^-(2) = 18 + 10 + 8$$

il y a plusieurs coupes de capacité minimale $C_W = 56$:

$$W_1 = \{1\}, \quad W_2 = \{1;6\}, \quad W_3 = \{1;3;6\}, \quad W_4 = \{1;3;4;5;6;7\}$$

Le flot maximum est donc limité par la coupe « minimale » du réseau. Le théorème suivant affirme que le flot maximum à justement pour valeur la capacité de la coupe « minimale ». Ce théorème a été énoncé par en 1956 par P. Elias, A. Feinstein et C.E. Shannon, et (indépendamment) par L.R. Ford, Jr. et D.R. Fulkerson.

Théorème 3.20 (flot-max/coupe-min)
Soit (G, s, t, C) un réseau de transport alors il existe un flot f sur ce réseau tel que

$$V(f) = \min_{\text{coupe } W} C_W$$

La démonstration de Ford et Fulkerson conduit à un procédé algorithmique qui permet, en partant d'un flot donnée (le flot nul $f(x, y) = 0$ par exemple), de l'augmenter jusqu'à la valeur maximale donnée par la capacité d'une coupe minimale ! C'est ce procédé que nous allons étudier maintenant. L'idée de Ford et Fulkerson est qu'il y a deux manières d'augmenter le flot dans un réseau entre deux sommets voisins x et y :

- si $(x, y) \in A$ et $f(x, y) < C(x, y)$ alors on peut augmenter le flot de x vers y de $C(x, y) - f(x, y)$ sur cet arc
- si $(y, x) \in A$ et $f(y, x) > 0$ alors on peut diminuer le flot jusqu'à 0 de y vers x ce qui revient à augmenter le flot de x vers y

Ainsi pour pouvoir augmenter le flot il suffit de trouver une suite de sommets du graphe allant de la source au puits et le long de laquelle on puisse augmenter le flot, c'est ce qu'on va appeler une chaîne augmentante.

Définition 3.21 (chaîne augmentante) *Dans un réseau de transport (G, s, t, C) muni d'un flot f on appelle* **augmentation du flot de x vers y** *la quantité :*

$$\Delta(x, y) = \max\left(C(x, y) - f(x, y), f(y, x)\right)$$

Une **chaîne augmentante** *est une suite de sommets $C = (x_0, x_1, \ldots, x_p)$ dans G telle que :*

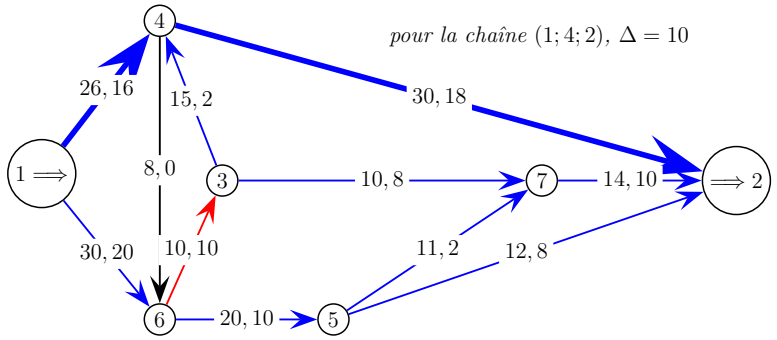
- *C est chemin du graphe de la source s jusqu'au puits t dans le graphe G considéré comme un* **graphe non-orienté** *:*

$$x_0 = s, \quad x_p = t, \quad et \quad [\forall i = 0, \ldots, p-1, \quad (x_i, x_{i+1}) \in A \quad ou \quad (x_{i+1}, x_i) \in A]$$

- *l'augmentation du flot de s vers t le long de C est* **strictement positive** *:*

$$\Delta_C = \min_{i=0,\ldots,p-1} \Delta(x_i, x_{i+1}) > 0$$

✎ **3.18 Chaîne augmentante** $(1; 4; 2)$ **avec** $\Delta = 10 = \min(10, 12)$ **:**



Le long d'une telle chaîne on peut augmenter le flot d'une quantité Δ_C.

Proposition 3.22 *Dans un réseau de transport (G, s, t, C) muni d'un flot f, s'il existe une chaîne augmentante $C = (x_0, x_1, \ldots, x_p)$ dans G alors on peut définir un nouveau flot $\tilde{f}$ sur le réseau par :*

$$\tilde{f}(x, y) = \begin{cases} f(x, y) = f(x, y) + \Delta_C & si \quad (x, y) = (x_i, x_{i+1}) \in A \\ f(x, y) = f(x, y) - \Delta_C & si \quad (x, y) = (x_{i+1}, x_i) \in A \\ f(x, y) & sinon \end{cases}$$

qui vérifie $V(\tilde{f}) = V(f) + \Delta_C$

Preuve : On vérifie facilement que $\tilde{f}$ est bien un flot, d'abord le nouveau flot $\tilde{f}$ ne dépasse pas la capacité et reste positif :

- si$(x_i, x_{i+1}) \in A$ alors

$$\tilde{f}(x_i, x_{i+1}) = f(x_i, x_{i+1}) + \Delta_C > f(x_i, x_{i+1}) + 0 \geq 0$$

et

$$\tilde{f}(x_i, x_{i+1}) = f(x_i, x_{i+1}) + \Delta_C \leq f(x_i, x_{i+1}) + C(x_i, x_{i+1}) - f(x_i, x_{i+1}) = C(x_i, x_{i+1})$$

- si $(x_{i+1}, x_i) \in A$ alors

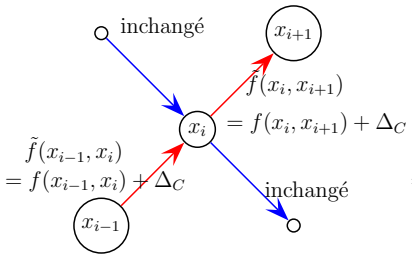
$$\tilde{f}(x_{i+1}, x_i) = f(x_{i+1}, x_i) - \Delta_C > f(x_{i+1}, x_i) - f(x_{i+1}, x_i) = 0$$

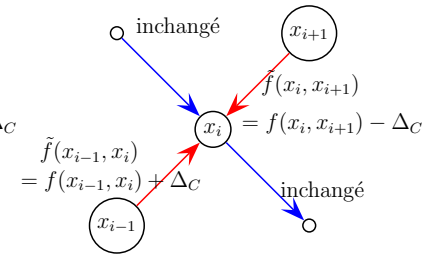
et

$$\tilde{f}(x_i, x_{i+1}) = f(x_i, x_{i+1}) - \Delta_C < f(x_i, x_{i+1}) - 0 \leq C(x_i, x_{i+1})$$

- sinon $\tilde{f}(x, y) = f(x, y)$ reste compris entre 0 et $C(x, y)$

Ensuite $\tilde{f}$ vérifie bien la loi des nœuds. Soit x_i un sommet de la chaîne augmentante la vérification est simple sur un diagramme mais peut être compliquée à écrire :





- Dans le premier cas si (x_{i-1}, x_i) et $(x_i, x_{i+1}) \in A$, $\tilde{f}^-(x_i) = \tilde{f}^+(x_i)$ car :

$$\tilde{f}^-(x_i) = \sum_{(x,x_i)\in x_i^-} \tilde{f}(x, x_i) = \left(\sum_{\substack{(x,x_i)\in x_i^- \\ x \neq x_{i-1}}} f(x, x_i) \right) + (f(x_{i-1}, x_i) + \Delta_C) = f^-(x_i) + \Delta_C$$

$$\tilde{f}^+(x_i) = \sum_{(x_i,x)\in x_i^+} \tilde{f}(x,x_i) = \left(\sum_{\substack{(x_i,x)\in x_i^+ \\ x\neq x_{i+1}}} f(x_i,x) \right) + (f(x_i,x_{i+1}) + \Delta_C) = f^+(x_i) + \Delta_C$$

- Dans le second cas si $(x_{i-1},x_i)\in A$ et $(x_{i+1},x_i)\in A$, $\tilde{f}^-(x_i) = \tilde{f}^+(x_i)$ car

$$\tilde{f}^-(x_i) = \sum_{(x,x_i)\in x_i^-} \tilde{f}(x,x_i) = \left(\sum_{\substack{(x,x_i)\in x_i^- \\ x\neq x_{i-1},x_{i+1}}} f(x,x_i) \right) \dots$$
$$\dots + (f(x_{i-1},x_i) + \Delta_C) + (f(x_i,x_{i+1}) - \Delta_C) = f^-(x_i) + 0$$

$$\tilde{f}^+(x_i) = \sum_{(x_i,x)\in x_i^+} \tilde{f}(x_i,x) = \sum_{(x_i,x)\in x_i^+} f(x_i,x) = f^+(x_i) \text{ car } x_{i+1}, x_{i-1} \notin x_i^+$$

- on démontre de même que $\tilde{f}^-(x_i) = \tilde{f}^+(x_i)$ quand $(x_i,x_{i-1})\in A$ et $(x_{i+1},x_i)\in A$ et quand $(x_i,x_{i-1})\in A$ et $(x_{i+1},x_i)\in A$ ...

enfin le flot à bien été augmenté de Δ_C puisque pour s (premier sommet de la chaîne augmentante) on a :

$$V(\tilde{f}) = \tilde{f}(s) = \sum_{(s,y)\in s^+} \tilde{f}(s,y) = \left(\sum_{(s,y)\in s^+, y\neq x_1} f(s,y) \right) + \underbrace{f(s,x_1) + \Delta_C}_{\tilde{f}(s,x_1)}$$
$$= \left(\sum_{(s,y)\in s^+} f(s,y) \right) + \Delta_C = f^+(s) + \Delta_C = V(f) + \Delta_C$$

□

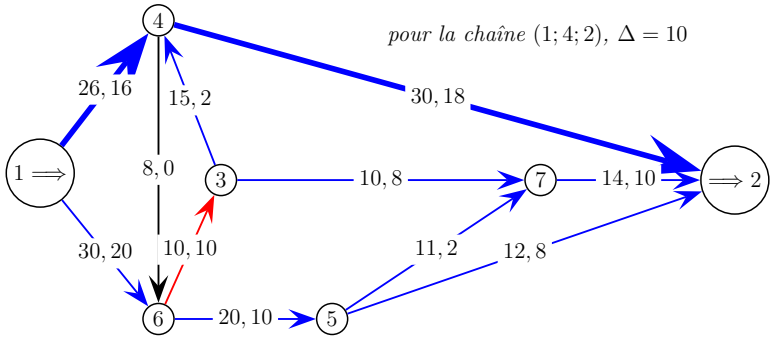
On en déduit facilement l'algorithme de Ford-Fulkerson pour calculer le flot maximum sur un réseau de transport.

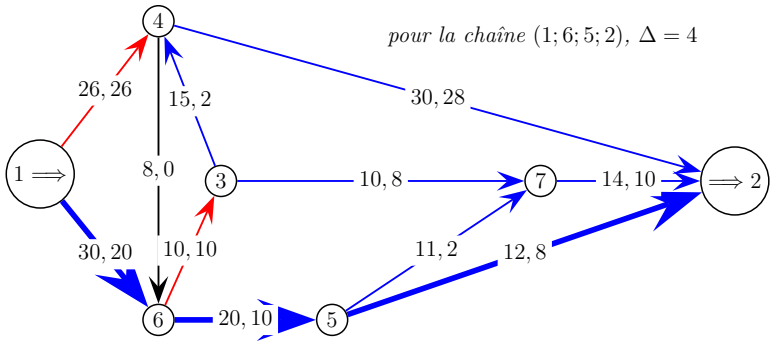
Théorème 3.23 (algorithme de Ford-Fulkerson) *Dans un réseau de transport (G,s,t,C), on obtient un flot maximum en partant de n'importe flot f, l'augmentant de Δ_C pour chaque chaîne augmentante C du réseau. Ce qui donne l'algorithme :*

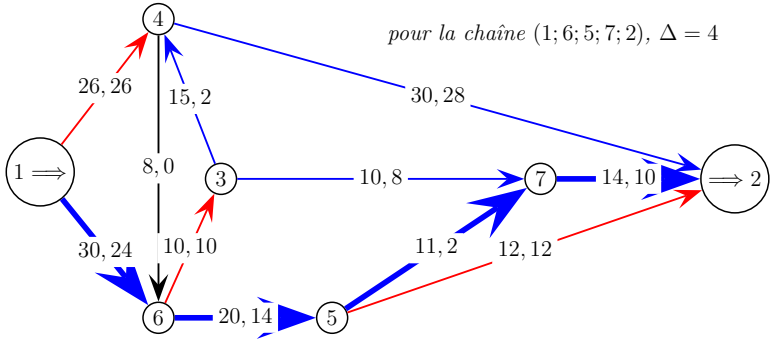
fonction f = Ford_Fulkerson(G,s,t,C,f)
f = flot de départ (éventuellement nul)
tant que $\exists C$ chaîne augmentante **faire**
augmenter f de Δ_C le long de C
fin faire

Pour comprendre le mieux est d'appliquer cet algorithme sur un exemple.

✎ **3.19 Construction du flot maximum avec l'algorithme de Ford-Fulkerson :**
au départ le flot est $V(f) = 36$ on cherche des chaînes augmentantes de la source au puits en commençant par chercher des chemins sans arcs saturés :

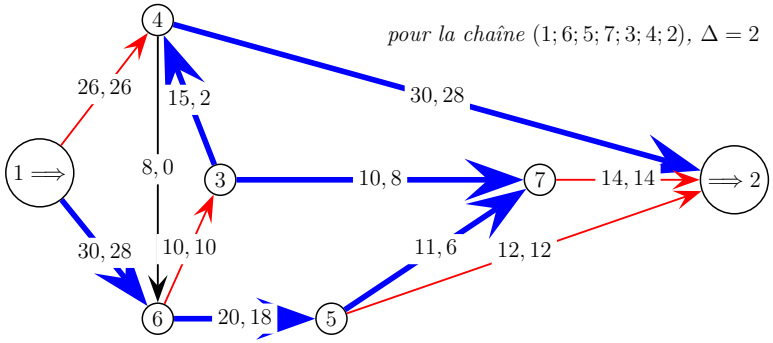


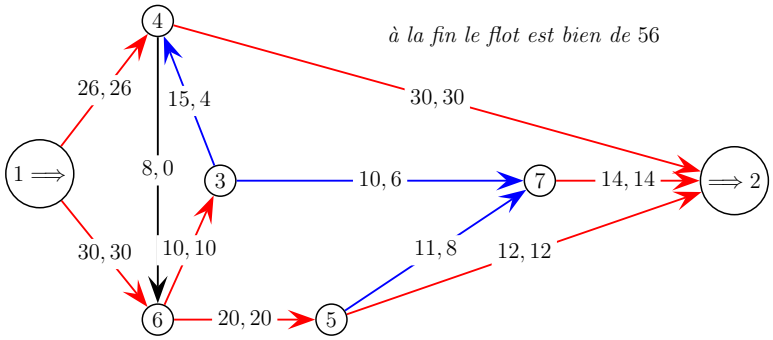




À partir de maintenant on ne trouve plus de chaîne augmentant qui soit un chemin de la source au puits, on est donc obligé de prendre un ou des arcs à contre-sens

pour trouver d'autre chaîne augmentante :





À chaque fois qu'on modifie le flot le long d'une chaîne augmentante qui est un chemin de la source ou puits **on sature un arc**.

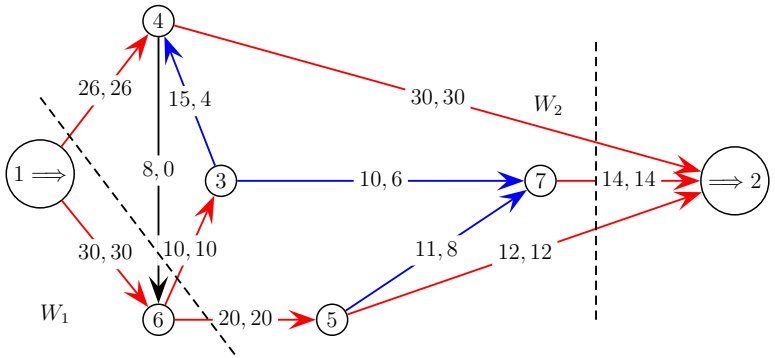
À la fin il faut éventuellement « détourner » le flot de certains points d'**engorgement** ce qui peut rendre nul le flot sur un arc.

Enfin sur le dernier diagramme les **coupes minimales** apparaissent clairement, ce sont des ensembles de sommets tels que le flux sortant est saturé (arcs en rouges) et le flux entrant est nul (arcs en noir) !

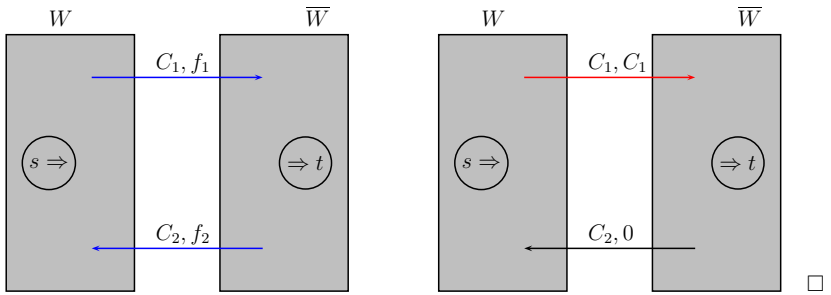
3.20 repérage des coupes minimales sur le flot maximum

il y a trois coupes minimales de capacité 56

$$W_0 = \{1\}, \quad W_1 = \{1;6\}, \quad W_2 = \{1;3;4;5;6;7\}$$



Preuve : théorème de Ford-Fulkerson Considérons une coupe minimale W, il y a forcément un arc, entre cette coupe et le reste des sommets, dont le flot est soit nul soit saturé. Il doit aussi exister un chemin de s à t passant par cet arc qui est une chaîne augmentante (sinon il existerait une coupe de capacité plus faible strictement plus petite que W). Augmenter le flot va donc tendre à saturer cet arc (si on le prend dans le sens direct) ou à le vider (si on le prend à contre-sens) :



□

Le protocole de routage IP appelé **RIP** (Routing Information Protocol), de type *"distance vector"*, est basé sur l'algorithme de Ford-Fulkerson. Chaque routeur communique aux autres la distance avec un réseau donné en comptant le nombre de sauts et diffuse toute sa table de routage. Ce protocole (maintenant assez ancien) est limité à 15 sauts et ne prend pas en compte la qualité des transmissions.

4 Notions de théorie des langages

Nous allons terminer ce cours avec un des prolongements de la théorie des graphes ayant le plus d'applications en informatiques : la théorie des langages. La théorie des langages permet, par exemple, de modéliser les langages de programmation et de les analyser. Elle s'est donc développée à partir des années 50 accompagnant les progrès de l'électronique puis de l'informatique. Cette théorie a permis de construire de nombreux outils d'analyse lexicale et syntaxique, comme *yacc* et *lex*, qui permettent de générer un interpréteur/compilateur pour un langage de programmation définit de manière formelle.

4.1 Alphabets, langages et grammaires formelles

Il faut commencer par donner une définition précise d'un langage.

Définition 4.1 (Langage formel) *Un langage L, défini sur un alphabet Σ, est un ensemble de mots (éventuellement vides) obtenus par concaténation des éléments de l'alphabet Σ. Pour être plus précis :*

un alphabet Σ *est un ensemble de symboles « élémentaires »,*

un mot sur Σ *est une suite de symboles appartenant à Σ (on dit aussi lexème),*

la concaténation $\cdot$ *est l'opération qui permet de fabriquer des mots à partir d'un alphabet Σ en les juxtaposant,*

la longueur d'un mot *est le nombre de symboles qui le compose,*

le mot vide *est noté ϵ est le seul mot de longueur nulle $|\epsilon| = 0$.*

En conséquence un « mot » au sens de la théorie des langages peut correspondre à ce que nous appellerions une « phrase ».

✎ **4.1 Exemples de langages simples** *quelques exemples tirés des cours de mathématiques :*

les entiers naturels $\mathbb{N}$ *forment un langage sur l'alphabet*

$$\Sigma = \{0; 1; 2; 3; 4; 5; 5; 6; 7; 8; 9\}$$

- $54321 \in \mathbb{N}$ *est un mot de longueur* 5
- *mais* 01234 *n'est pas un mot de ce langage, car l'écriture d'un entier ne commence jamais par 0 !*

les expressions booléennes *du cours de théorie des ensembles (semestre 1) forment un langage sur l'alphabet*

$$\{Vrai; Faux; \wedge; \vee; \neg; \Longrightarrow; \Longleftrightarrow; [;]; P; Q; R\}$$

- $[P \Longrightarrow Q] \Longleftrightarrow [\neg P \wedge Q]$ *est un mot de ce langage (de longueur* 12*)*
- *par contre* $P[\Longrightarrow \neg Q]$ *n'est pas un mot de ce langage, car le parenthésage n'est pas valide*

expressions arithmétiques définies dans $\mathbb{Q}$ *forment un langage sur l'alphabet*

$$\{0; 1; 2; 3; 4; 5; 5; 6; 7; 8; 9; +; \times; -; /; [;]\}$$

- $[56 - 34] \times [1 + 2]$ *est un mot de ce langage de longueur* 13
- $-[123/456] + [789/[10 - 2 \times 5]]$ *n'est pas un mot de ce langage à cause de la division par 0 !*

Attention un symbole de l'alphabet peut être composé de plusieurs caractères ! Il faut donc faire attention à la définition du langage, qui peut être ambiguë. Par exemple si on considère le **langage de tous les mots possibles sur l'alphabet $\Sigma = \{\mathbf{a}; \mathbf{b}; \mathbf{ab}\}$** alors il existe une **ambiguïté sur la nature du mot aab** qui peut être vu comme **a.a.b** ou comme **a.ab** ! En particulier on ne peut pas savoir si c'est un mot de longueur 2 ou 3 !
La notion de symbole « élémentaire », qui apparaît dans la définition de l'alphabet, doit donc être comprise dans le sens qu'aucun symbole de l'alphabet ne peut être décomposé en symboles appartenant eux aussi à l'alphabet.

Cette difficulté apparaît directement dans l'analyse des langages de programmation dans le processus d'identification des variables utilisateur. Par exemple dans la première définition du langage ALGOL (fin des années 50) les mots clés n'étaient pas interdits pour les noms de variables. On pouvait donc écrire :
`if else = then then if = else else then = if;`
Cela a vite été abandonné (à l'époque, la théorie des langages et des automates n'était pas très avancée...).

On peut fabriquer des langages à partir d'autres langages plus simple en utilisant quelques opérations de base.

Définition 4.2 (opérations sur les langages) *À partir de langages L, L_1, L_2, sur un alphabet Σ, on peut construire de nouveaux langages sur Σ en utilisant*

les opération de base sur les ensembles *union, intersection, complément*

$$L1 \cup L_2, \quad L_1 \cap L_2, \quad \overline{L}$$

la concaténation *définie par* $L_1.L_2 = \{w = u.v | u \in L_1 \text{ et } v \in L_2\}$

l'élévation à la puissance *définie par récurrence avec*

$$\forall n \geq 0, \quad L^n = L^{n-1}.L \quad \text{et} \quad L^0 = \{\epsilon\}$$

la fermeture de Kleene $L^* = \overset{\infty}{\underset{n=0}{\cup}} L^n$ *en particulier*
Σ^* *est l'ensemble de tous les mots possibles sur l'alphabet* Σ
On notera aussi $L^+ = \overset{\infty}{\underset{n=1}{\cup}} L^n$

✎ **4.2 Définir des langages sur l'alphabet** $\Sigma = \{1; 2; 3\}$ *à partir de*

$$L_1 = \{x \in \Sigma^* | x \text{ pair}\}, \quad L_2 = \{x \in \Sigma^* | x \leq 20\}, \quad L_3 = \{1; 3\}$$

alors $L_1 = \{2; 12; 22; 32; 112; 122; 132; \ldots\}$

- $\overline{L_1} = \{x \in \Sigma^* | x \text{ impair}\}$
- $L_1^+ = L_1$ *mais* $L_1^* = L_1 \cup \{\epsilon\}$
- $L_2 = \{1; 2; 3; 11; 12; 13\}$
- $L_1 \cap L_2 = \{2; 12\}$
- $L_1 \cup L_2 = L_1 \cup \{1; 3; 11; 13\}$
- $\Sigma^* L_3 = \{x \in \Sigma^* | x \text{ impair}\} = \overline{L_1}$

Un langage L sur Σ est donc forcément un sous-ensemble de Σ^* : $L \subset \Sigma^*$.

Grammaire formelle

Pour pouvoir représenter la construction des mots d'un langage, on a besoin de formaliser leurs règles de construction d'une manière simple. Pour cela nous avons besoin d'introduire le concept de grammaire.

Définition 4.3 (Grammaire formelle) *Une grammaire formelle (ou, simplement, grammaire) est constituée des quatre objets suivants :*

- *un ensemble fini de symboles V appelé « vocabulaire » (contenant l'alphabet Σ du langage) formé de 2 sous ensembles $V = T \cup N$:*
  - *des symboles terminaux $T = \Sigma \cup \{\epsilon\}$, notés conventionnellement par des minuscules,*
  - *des symboles non-terminaux N, notés conventionnellement par des majuscules,*
- *Un élément de l'ensemble des non-terminaux, appelé « axiome », noté conventionnellement S_0,*
- *Un ensemble de « règles » de la forme $\alpha \to \beta$ où α est un non-terminal et β une concaténation de terminaux et de non-terminaux*

pour des raisons de commodité on acceptera les simplifications suivantes dans l'écriture des règles :

$$[\alpha \to \beta;\ \alpha \to \gamma] \Longleftrightarrow \alpha \to \beta | \gamma$$

4.3 Exemple de grammaires formelles

identifier un langage à partir de la grammaire :

on considère la grammaire formelle sur l'alphabet $\Sigma = \{a; b; c\}$:

- *l'ensemble de symboles $V = \{a; b; c; S_0; S_1\}$*
- *symboles terminaux : a, b, c*
- *un axiome S_0*
- *et les règles* $S_0 \to a$; $S_0 \to S_1$; $S_1 \to bS_1$; $S_1 \to c$

analysons les mots que l'on peut former avec ce langage

- *d'après la première règle a est un mot du langage*
- *les trois autre règles permettent de former des mots du type $b \dots bc$*

ce langage est donc en fait $\{a; c; bc; bbc; bbbc; \dots\}$. On aurait pu aussi écrire les règles de cette grammaire sous la forme : $S_0 \to a|S_1$; $S_1 \to bS_1|c$

Retrouver la grammaire à partir du langage :

soit le langage contenant tous les noms des fiches d'exercice du cours de maths :

$$TD1.pdf,\quad TP1.pdf,\quad \dots, TD9.pdf,\quad TP9.pdf$$

peut être formalisé par une grammaire en posant :

- *l'alphabet $\Sigma = \{T; D; P; 1; \dots; 9; .; p; d; f\}$*
- *on ajoute un axiome S_0*
- *et les règles $S_0 \to TDS_1|TPS_1$;*

$$S_1 \to 1S_2|2S_2|3S_2|4S_2|5S_2|6S_2|7S_2|8S_2|9S_2;\quad S_2 \to .pdf$$

- *et l'ensemble de symboles $V = \{T; D; P; 1; \dots; 9; .; p; d; f; S_0; S_1; S_2\}$*

Pour décrire les règles d'un langage on utilisera dans le vocabulaire (en plus de l'alphabet) le mot vide ϵ. En particulier la règle $F \to \epsilon$ permet d'écrire simplement que l'écriture d'un mot est finie.

C'est un linguiste américain, Noam Chomsky, qui a le premier dégagé la notion de grammaire formelle, dans les années 50. Il en a proposé une classification appelée de nos jours hiérarchie de Chomsky.

Définition 4.4 (hiérarchie de Chomsky) *Soit L un langage formel, sur un alphabet $\Sigma = \{a; b; \dots\}$, défini par un vocabulaire $V = N \cup T$ ($T = \Sigma \cup \{\epsilon\}$ sous-ensemble des terminaux $N = \overline{T}$) et un ensemble de règles $\mathscr{R}$. Le langage L appartient à l'une des 4 catégories (de la plus restrictive à la plus large) suivantes :*

Les langages de type 3, ou langages rationnels :
ce sont les langages définis par une grammaire linéaire à gauche (resp. droite), grammaire dont chaque membre droit (resp. gauche) de règle commence par un non-terminal. En d'autres termes règles doivent pouvoir s'écrire sous la forme : *linéaire à gauche : $A \to aB|a$* *linéaire à droite : $A \to Ba|a$* *avec $A, B \in N$,* $a \in T = \Sigma \cup \{\epsilon\}$

Les langages de type 2, ou langages algébriques :
ce sont les langages définis par une grammaire formelle hors-contexte, grammaire dont les règles doivent pouvoir s'écrire sous la forme : $A \to a|aBb$ *avec* $A, B \in N$, $a, b \in T = \Sigma \cup \{\epsilon\}$

Les langages de type 1, ou langages contextuels :
ce sont les langages définis par une grammaire contextuelle, grammaire dont les règles doivent pouvoir s'écrire sous les formes des grammaires de type 2 ou 3 ou : $\alpha A\beta \to \alpha\gamma\beta$ *avec* $A \in N$, $\alpha, \beta, \gamma \in V, \gamma \neq \epsilon$

Les langages de type 0, ou langages récursivement énumérables :
Cet ensemble inclut tous les langages définis par une grammaire formelle.

$$\alpha \to \beta,\quad \alpha \in N,\quad \beta \neq \epsilon$$

Une grammaire de type $i+1$ est **forcément** aussi de type i mais une grammaire écrite sous la forme de type i peut en fait être une grammaire de type $i+1$!

✎ **4.4 Exemples de grammaires de types 3 écrite sous la forme d'une grammaire de type 2 : les expressions logiques**

On a déjà dit que le langage des expressions logiques, qu'on rencontre en calcul des prédicats[5]*, peut être formalisée par un langage sur l'alphabet*

$$\Sigma = \{Vrai; Faux; \wedge; \vee; \neg; \Longrightarrow; \Longleftrightarrow; [\ ;\ ]; P; Q; R\}$$

essayons d'écrire la grammaire de ce langage. Pour celà il faut ajouter des états S_i à l'alphabet Σ et expliquer comment on obtient un mot du langage par concaténation des symboles de Σ :

- *on démarre de l'axiome S_1*
- *on indique les règles d'utilisation du et du ou et du non :*

$$S_1 \to S_1 \ \vee \ S_1 | S_1 \ \wedge \ S_1 | \neg S_1 | S_1 \implies S_1 | S_1 \iff S_1$$

- *une expression se termine par une variable P, Q, R ou une valeur $Vrai$ ou $Faux$:*

$$S1 \to Vrai | Faux | P | Q | R$$

- *on a donc identifié $Vrai, Faux, P, Q, R$ comme terminaux de cette grammaire.*

Telle qu'elle cette grammaire est de type 2 mais elle peut être réécrite sous forme d'une grammaire de type 3 en rajoutant des états :

- *on part d'un axiome S_1 :*

$$S_1 \to Vrai\ S_2 | Faux\ S_2 | P\ S_2 | Q\ S_2 | R\ S_2 | \neg S_1$$

- *on peut terminer le mot en décidant que S_2 peut être remplacé par le ϵ ou ajouter un opérateur binaire*

$$S_2 \to \vee\ S_3 | \wedge\ S_3 | \implies S_3 | \iff S_3 | \epsilon$$

- *si on a ajouté un opérateur binaire on est obligé d'ajouter un symbole de l'alphabet (non vide) donc ensuite il faut revenir à l'état terminal S_2 en ajoutant un symbole correspondant à une variable ou un booléen :*

$$S_3 \to Vrai\ S_2 | Faux\ S_2 | P\ S_2 | Q\ S_2 | R\ S_2 | \neg\ S_3$$

- *le seul symbole terminal est donc S_2*

la grammaire est donc linéaire à droite et donc le langage est de type 3.

5. si on se limite à 3 propositions P, Q, R

4.2 Langages réguliers et automates Finis

Dans l'ensemble des langages sur un alphabet Σ l'ensemble des langages réguliers joue un rôle très important.

Définition 4.5 (Langages réguliers)
L'ensemble $\mathscr{R}$ des langages réguliers sur Σ est le plus petit ensemble tel que

- $\emptyset \in \mathscr{R}$ *et* $\{\epsilon\} \in \mathscr{R}$
- $\forall a \in \Sigma\ \{a\} \in \mathscr{R}$
- $\forall A, B \in \mathscr{R}\ A \cup B, A.B, A^* \in \mathscr{R}$

Les langages réguliers peuvent donc être construit à partir d'un alphabet en utilisant uniquement des opérations simples (réunion, concaténation, fermeture de Kleene).

Théorème 4.6 (expression régulière)
Une expression régulière est une expression formée à partir des symboles $\Sigma \cup \{\epsilon\}$, des règles[6] *de réunion (|), concaténation (termes accolés !), puissance (^{2}, ^{3}, ...) et fermeture de Kleene (* et $^+$), le tout éventuellement parenthésé.*

$\mathscr{R}$ est langage régulier $\iff$ $\mathscr{R}$ peut être représenté par une expression régulière.

✎ **4.5 Exemple d'expressions régulières**

décrire les éléments d'un langage représenté par une expression régulière

sur l'alphabet $\Sigma = \{a; b; c\}$

- $a^* = \{\epsilon; a; aa; aaa; \ldots\}$
- $(a|b)^* = \{\epsilon; a; b; ab; ba; aab; aba; abb; baa; bab; bba; bbb; \ldots\}$
- $ab^* = \{a; ab; abb; abbb; \ldots\}$
- $a|b^*c = \{a; c; bc; bbc; bbbc; \ldots\}$ *et nous avons plus haut que ce langage est bien décrit par une grammaire de type 3* $\quad S_0 \to a|S_1; \quad S_1 \to bS_1|c$

trouver une expression régulière pour décrire un langage

exemple avec l'alphabet des caractères $\Sigma = \{a; b; \ldots; z; A; B; \ldots; Z; 0; \ldots 9; .\}$

- *L'ensemble des fiches pdf du cours de théorie des graphes (TD1 à TD9 et TP1 à TP9) peut être décrit par l'expression régulière*

$$\text{T(D|P)(1|2|3|4|5|6|7|8|9).pdf}$$

- *si on veut décrire dans une même expression régulière les différentes versions du polycopié (les fichiers cours.pdf et cours2.pdf) en plus des fiches de TD et TP alors on peut construire une autre expression régulière :*

$$\text{(cours}(\epsilon|2)\text{.pdf)|(T(D|P)(1|2|3|4|5|6|7|8|9).pdf)}$$

- *les fichiers scilab associés à chaque TP peuvent être décrit par l'expression régulière suivante*

$$\text{(TP(1|2|3|4|5|6|7|8|9).sce)|(exo(1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)}^*\text{G.graph)}$$

6. par ordre de priorité croissante

Les expressions régulières sont très importantes, en système elle permettent se simplifier le traitement des fichiers en grande quantité. Il existe de nombreuses implémentation des expressions régulières dans différents langages, par exemple, en Bourne shell on peut utiliser `grep`. Pour rechercher les fiches de TD/TP du cours dans un répertoire il suffira de faire :

```
ls | grep -e '^T[DP][1-9].pdf'
```

de même pour rechercher les fichiers graphes :

```
ls | grep -e '^exo[1-9][0-9]*G.graph'
```

On trouve aussi des fonctions comme `sed` qui permettent de faire des remplacements dans un fichier en utilisant des expressions régulières :

```
sed -e 's/regex1\(regex2\)regex3/\1/g' fichier.txt
```

cette commande substitue (option `s`), dans tout (option "global" `g`) le fichier (`fichier.txt`), les groupes de la forme `regex1\(regex2\)regex3` par uniquement `regex2` (symbolisé par `\1`) On retrouve le même genres de fonctionnalités en php avec `preg_match`, pour utiliser des expressions régulières dans des éditeurs de texte comme `notepad++` on pourra lire [6] ...

Pour construire des expressions régulières correspondant à des langages complexes il faut passer par un graphe permettant de représenter simplement le langage et qu'on appelle un automate.

Définition 4.7 (Automate à nombre fini d'états)
Un automate fini est un quintuplet $\mathscr{A} = (E, \Sigma, \delta, I, F)$ où :

- *Σ est un ensemble (non-vide) de symboles appelé alphabet*
- *E est un ensemble (non-vide) dont les éléments sont appelés états*
- *$\delta : E \times \Sigma \cup \{\epsilon\} \longrightarrow E$ est appelée fonction de transition*
- *$I \subset E$ un ensemble (non-vide) d'état appelés états initiaux*
- *$F \subset E$ un ensemble (non-vide) d'état appelés les états acceptant (terminaux)*

On représente $\mathscr{A}$ par le graphe d'un réseau de transport $G = (E, U, \delta)$ avec :

- *pour sommets l'ensemble E des états*
- *pour arcs l'ensemble des couples $(x, y) \in U \subset E \times E$ tels que*

$$\exists w \in \Sigma \cup \{\epsilon\}, \quad \delta(x, w) = y \quad (\text{donc } \delta(x, w) \text{ est bien définie})$$

- *pour valuation $C(x, y) = \{w | y = \delta(x, w)\}$*
- *plusieurs sources données par l'ensemble I des états initiaux*
- *plusieurs puits données par l'ensemble F des états acceptant*

Les états initiaux sont indiqués par des flèches, les finaux par des doubles cercles. À chaque chemin d'une source vers un puits de G on fait correspondre un mot obtenus en concaténant les symboles appartenant aux valuations des arcs du chemin dans l'ordre de parcours. Le **langage associé à un automate** *$\mathscr{A}$ est l'ensemble des mots obtenus en considérant tous les chemins possibles d'une source vers un puits.*

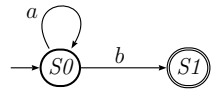
les états d'un automate correspondent grosso-modo aux éléments non-terminaux du vocabulaire dans la grammaire du langage L associé à ceci-près que :

- un non-terminal A de la grammaire admettant une règle de dérivation $A \to \epsilon$ deviendra un état final de l'automate $\mathscr{A}$
- on ajoutera des états terminaux (B ici) à l'automate $\mathscr{A}$ pour représenter les règles de dérivation du type $A \to a$ comme si on avait les règles $A \to aB, B \to \epsilon$

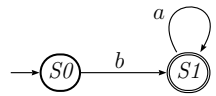
4.6 Premiers automates. *Les automates permettent de représenter un langage d'une manière plus explicite qu'une expression régulière et moins formelle qu'une grammaire. Il est donc très utile de pouvoir faire le lien entre un automate et le langage associé, soit sous forme d'expression régulières soit sous forme de grammaire :*

- *le langage a^* est décrit par la règle $S_0 \to aS_0|\epsilon$ le symbole terminal est a et l'axiome S_0. Il sera représenté par l'automate ci-contre. L'axiome S_0 est l'état initial et final de l'automate*

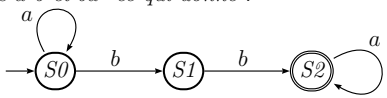
- *le langage a^*b consiste à enchaîner un mot de a^* puis le symbole b. Au niveau de la grammaire cela correspond aux règles $S_0 \to aS_0|b$ les symboles terminaux sont a, b et l'axiome S_0. Il sera représenté par l'automate ci-contre. L'axiome S_0 est l'état initial et S_1 l'état final de l'automate*



- *De même pour le langage ba^* mais on inverse l'ordre des transitions. le langage est donc décrit par les règles $S_0 \to bS_1$; $S_1 \to aS_1|\epsilon$ avec les symboles terminaux a, b et l'axiome S_0. Il sera représenté par l'automate ci-contre où l'axiome S_0 est l'état initial et S_1 l'état final de l'automate*

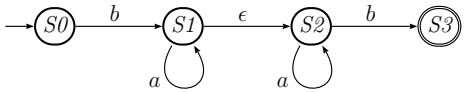


- *pour le langage a^*bba^* il suffit maintenant de concaténer les automates des langages a^*b et ba^* ce qui donne :*

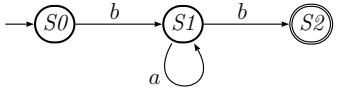


le langage est donc décrit par les règles $S_0 \to aS_0|bS_1$; $S_1 \to bS_2$; $S_2 \to aS_2|\epsilon$; les symboles terminaux sont a, b et l'axiome S_0. L'axiome S_0 est l'état initial et S_2 l'état final de l'automate.

- *De même le langage ba^*a^*b peut être représenté par l'automate :*

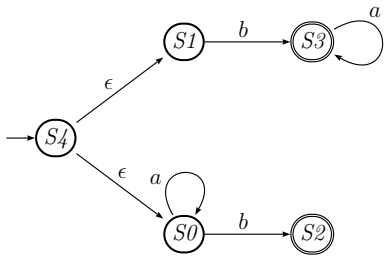


les états S_1 et S_2 étant très similaires on peut les regrouper en un seul (et supprimer la transition par le mot vide) :



cela revient à écrire que $ba^*a^*b = ba^*b$. On en déduit que le langage est décrit par les règles $S_0 \to bS_1$; $S_1 \to aS_1|bS_2; S2|\epsilon$ avec les symboles terminaux a, b et l'axiome S_0. L'axiome S_0 est l'état initial et S_2 l'état final de l'automate.

- *pour le langage $a^*b|ba^*$ il suffit d'ajouter un état qui permet de choisir si on veut aller vers l'automate de a^*b ou vers celui de ba^* :*

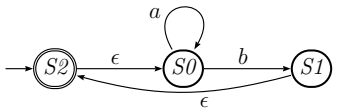


On en déduit que le langage associé est décrit par les règles

$$S_0 \to aS_0|bS_2; \quad S_1 \to bS_3; \quad S_3 \to aS_3|\epsilon; \quad S_4 \to S_1|S_2; \quad S_2 \to \epsilon$$

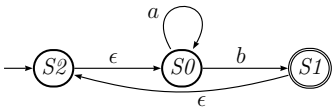
avec les symboles terminaux : a, b et l'axiome S_4. L'axiome S_4 est l'état initial et S_2, S_3 sont les états finaux de l'automate.

- *pour le langage $(a^*b)^*$ il faut ajouter un état à la fois initial et final qui va accepter le mot vide et permettre de « boucler » sur les mots de a^*b :*



ce langage est donc décrit par les règles $S_0 \to aS_0|bS_1$; $S_1 \to S_2$; $S_2 \to S_0|\epsilon$; les symboles terminaux étant a, b et l'axiome S_2 qui est donc l'état initial et aussi l'état final de l'automate.

- *pour le langage $(a^*b)^+$ il faut supprimer le mot vide de $(a^*b)^*$ ce qui peut se faire en changeant juste l'état final :*

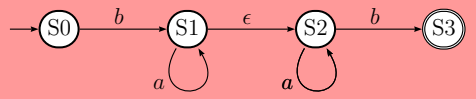


le langage est donc décrit par les mêmes règles que $(a^*b)^*$:

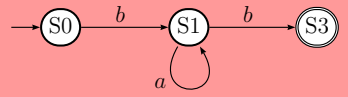
$$S_0 \to aS_0|bS_1; \quad S_1 \to S_2|\epsilon; \quad S_2 \to S_0;$$

mais cette fois L'axiome S_2 n'est que l'état initial alors que $S1$ est l'état final.

Un langage donnée peut être représenté par des automates en apparence différents ! Par exemple le langage ba^*a^*b a été représenté par l'automate :



qui peut être simplifié en :



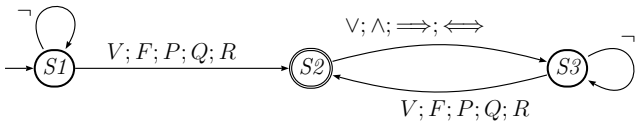
On parle alors d'automates équivalent. Simplifier un automate consiste à lui enlever des états ou des transitions **sans changer** le langage auquel il est associé !

Ces différents exemples montrent qu'on peut facilement construire un automate à partir d'automate plus simples en utilisant les opérations de base sur les langages concaténation (.) réunion (|) et étoile de Kleene (* ou $^+$).

Théorème 4.8 (Théorème de Kleene)
L'ensemble des langages qui peuvent être modélisés par un automate fini est exactement l'ensemble des langages de type 3, c'est à dire l'ensemble des langages réguliers (donc qui peuvent être décrits par une expression régulière). Vérifier qu'un mot appartient bien à un langage de type 3 s'appelle faire **l'analyse lexicale**.

4.7 Construire l'automate d'un langage.
Le langage des expressions booléennes (sans parenthésage) définit par la grammaire :
$S_1 \to S_1 \vee S_1|S_1 \wedge S_1|\neg S_1|S_1 \implies S_1|S_1 \iff S_1|Vrai|Faux|P|Q|R$
peut aussi être représenté par un automate, Pour cela il est plus facile de partir de la grammaire mise sous forme linéaire à gauche :

$$S_1 \to Vrai\ S_2|Faux\ S_2|P\ S_2|Q\ S_2|R\ S_2|\neg S_1$$
$$S_2 \to \vee\ S_3|\wedge\ S_3|\implies\ S_3|\iff\ S_3|\epsilon$$
$$S_3 \to Vrai\ S_2|Faux\ S_2|P\ S_2|Q\ S_2|R\ S_2|\neg\ S_3$$



S_1 est donc l'état initial et S_2 est l'état final.

L'automate permet de vérifier facilement si un mot appartient ou pas au langage associé puisque chaque mot correspond à un chemin depuis un état initial jusqu'à un état final. Tester si un mot appartient à un langage s'appelle faire l'analyse lexicale du langage

4.8 Analyse lexicale

- *le mot $P \Longrightarrow Q \Longleftrightarrow \neg P \wedge Q$ est bien dans le langage L car il peut être obtenu par le chemin :*

$$(S_1, S_2, S_3, S_2, S_3, S_3, S_2, S_3, S_2)$$

- *le mot $P \Longrightarrow Q \Longleftrightarrow \neg \wedge Q$ n'appartient pas au langage L car on ne peut pas atteindre un état final en partant de l'état initial, on bloque après $P \Longrightarrow Q \Longleftrightarrow \neg \ldots$ car on arrive en $S3$ et on ne peut pas rajouter le connecteur $\wedge$*

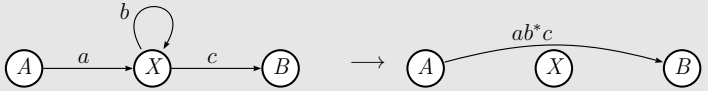
$$(S_1, S_2, S_3, S_2, S_3, S_3, ???)$$

En langage C, le programme Lex permet de faire l'analyseur lexical d'un langage de type 3 à partir de sa description formelle (sa grammaire). Il existe aussi une version GNU de Lex appelée Flex.

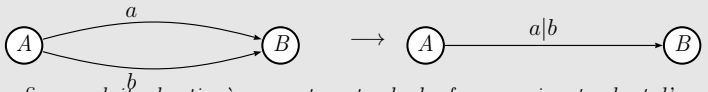
L'automate d'un langage permet de construire facilement une expression régulière du langage par ce qu'on appelle l'algorithme de Moore. Cet algorithme consiste à simplifier l'automate, en supprimant un état à chaque étape, jusqu'à ce qu'il n'ait plus que deux états (un initial et l'autre final) pour se ramener à un automate avec juste un état initial, un état final liés par un arc.

Proposition 4.9 (calcul d'expression régulière) *Soit un automate fini $\mathscr{A} = (E, \Sigma, \delta, I, F)$ pour calculer l'expression régulière correspondant au langage L associé à $\mathscr{A}$ on cherche à éliminer pas à pas les sommets et les transitions suivant 2 règles :*

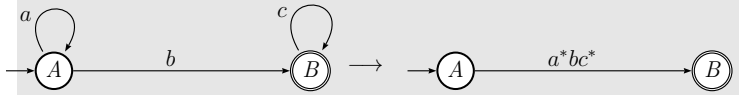
1. *supprimer les états intermédiaires*



2. *supprimer les transisssions multiples apparus après l'étape 1*



à la fin on doit aboutir à un automate de la forme suivante dont l'expression régulière associée est a^*bc^*



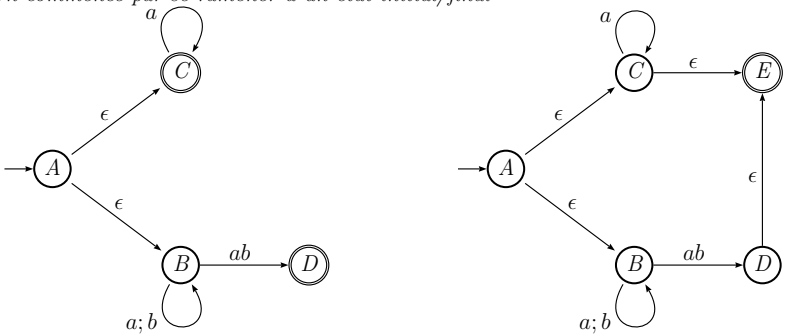
ce qui donne l'algorithme

fonction *regex* = automate_vers_regex($\mathscr{A}$)
  simplifier $\mathscr{A}$ pour qu'il n'ait qu'un seul état initial/final
  tant que $\mathscr{A}$possède au moins un états non-final et non-initial **faire**
    X = un état non-final et non-initial de $\mathscr{A}$
    pour tout A, B états de $\mathscr{A}$ adjacents à X **faire**
      simplifier $\mathscr{A}$ suivant la règle 1) :
      •créer la transition $A \to B$
      •supprimer les transitions $A \to X$ et $X \to B$
    fin faire
    supprimer X de l'automate
    simplifier les arc multiples suivant la règle 2)
  fin faire
  simplifier les éventuelles boucles restantes dans $\mathscr{A}$ suivant la règle 3)
fin faire
regex = valeur de l'unique transition de $\mathscr{A}$ (règle 3)

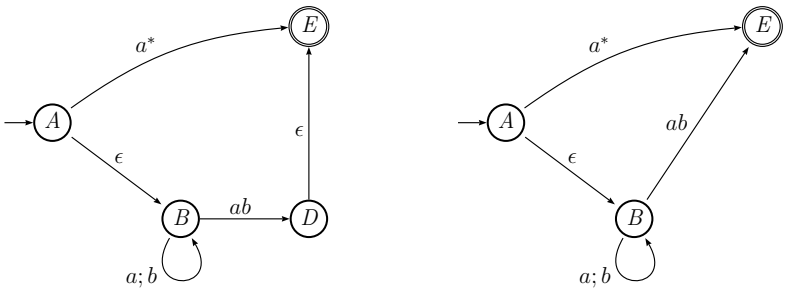
Selon l'automate utilisé pour décrire le langage on ne trouvera pas la même expression régulière !

✎ 4.9 calculer une expression régulière du langage

On commence par se ramener à un état initial/final



puis on déroule l'algorithme (peut importe l'ordre dans lequel on choisit les sommets)



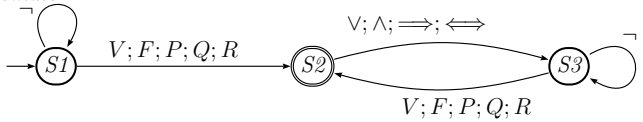
à la fin on arrive à seulement deux états (un initial un final) :

l'expression régulière est donc $a^*|((a|b)^*ab)$

Si l'automate de départ à plusieurs états finaux, les remplacer par un seul état final connecté aux précédents par des transition vides ϵ.

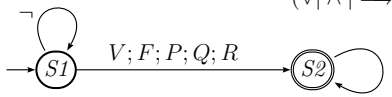
✎ 4.10 Trouver l'expression régulière du langage des expressions booléennes

Reprenons le langage des expressions booléennes (sans parenthésage) associé à l'automate :



la partie de l'automate entre S2 et S3 peut être vue comme une "boucle" sur S2où l'on répéterait le motif $(\vee|\wedge|\implies|\iff)(\neg)^*(V|F|P|Q|R)$*. Ceci permet de construire un automate équivalent mais avec moins d'états :*

$$(\vee|\wedge|\implies|\iff)(\neg)^*(V|F|P|Q|R)$$



Sur ce nouvel automate on peut remarquer que chaque boucle va correspondre à une fermeture de Kleene, dans l'expression régulière associée, qui entourent le motif apparaissant sur la transition $S1 \to S2$*. Ici il n'est pas difficile de trouver que le langage associé est défini par :*

$$\underbrace{(\neg)^*}_{\text{boucle } S1}\ \underbrace{(V|F|P|Q|R)}_{\text{arc } S1 \to S2}\ \underbrace{\Big((\vee|\wedge|\implies|\iff)(\neg)^*(V|F|P|Q|R)\Big)^*}_{\text{boucle } S2}$$

Les états initiaux et finaux de l'automate peuvent être le même état !

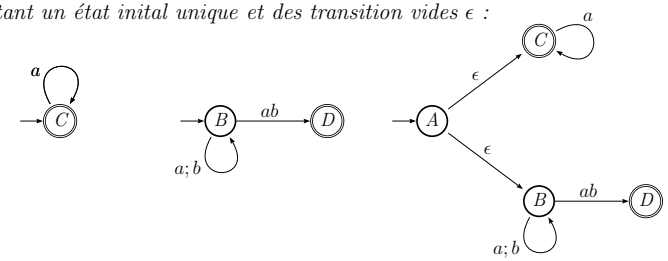
Pour de nombreux problèmes, on a en plus besoin d'imposer aux automates qu'on manipule d'être déterministe. C'est à dire qu'à partir d'un état donné l'état dans lequel on va se retrouver ensuite ne dépend que de l'unité lexicale suivante, et pas d'un choix fait par hasard (d'où le nom "déterministe").

Définition 4.10 (Automate déterministe)
Un automate fini $\mathscr{A} = (E, \Sigma, \delta, I, F)$ est dit déterministe si il existe un unique état initial et si pour tout mot du langage associé il existe un **unique** *chemin de l'état initial vers un état final.*

✎ **4.11 Automate non-déterministe :** *on considère le langage :*

$$L = \{w | w \text{ finit par } \ll ab \gg \text{ ou ne contenient que des } \ll a \gg\}$$

il contient exactement tous les mots du langage a^* et du langage $(a|b)^*ab$ donc il peut être représenté par l'expression régulière : $((a|b)^*ab)|a^*$. Si on veut construire un automate correspondant il suffit d'assembler les deux automates de a^* et $(a|b)^*ab$ en ajoutant un état inital unique et des transition vides ϵ :



ceci permet de trouver une grammaire linéaire à droite pour ce langage :

$$A \to B|C \qquad B \to a.B|b.B|D \qquad D \to ab \qquad C \to a.C|\epsilon$$

mais il est difficile de savoir si un mot est accepté par le langage : pour un mot commençant par a faut-il de A aller vers C ou vers B ?

un automate déterministe ne peut pas contenir plusieurs transitions avec le même symbole depuis un sommet donné. De même une transition avec le mot vide ϵ depuis un sommet n'est possible que s'il n'y a qu'une seule transition depuis ce sommet. Enfin il ne peut pas y avoir plusieurs états initiaux.

Quand on fabrique un automate à partir d'automates plus simple il est pratique d'utiliser des transition par le mot vide ϵ mais alors l'automate obtenu n'est en général plus déterministe. On dispose d'une méthode pour rendre n'importe quel automate déterministe.

Proposition 4.11 (rendre un automate déterministe)
Pour rendre un automate fini $\mathscr{A} = (E, \Sigma, \delta, I, F)$ déterministe on commence par :

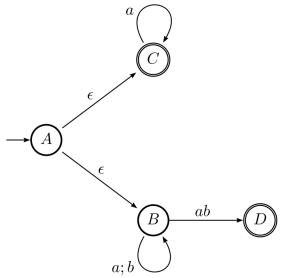
- *ajouter un état initial unique avec des transitions vides vers les états initiaux de départ*
- *éliminer les transitions multiples $S_i \to abS_j$ en ajoutant un état S_k pour les remplacer par les transitions $S_j \to aS_k$ et $S_k \to bS_j$*
- *éliminer les transitions vides $S_i \to \epsilon S_j$ et ajouter pour chaque transition $S_j \to aS_k$ la transition $S_i \to aS_k$ (des états peuvent devenir initiaux ou finaux à cette étape)*

ensuite pour chaque $S \subset E$ et $a \in \Sigma$ on calcule l'ensemble $T \subset E$ des états pour lesquels il existe une transition $S_i \to aT_j$. Le nouvel automate déterministe acceptant le même langage que $\mathscr{A}$ est alors $\mathscr{A}' = (E', \Sigma, \delta', I', F')$ où

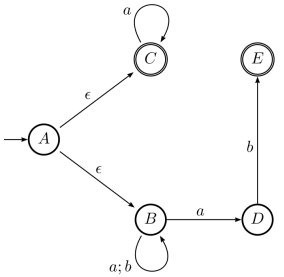
- $E' = \{S \subset E |$ *il existe des transition δ' de S vers un autre état de E'* $\}$
- $\delta'(S, a) = T \iff \exists S_i, T_j \in E, \quad \delta(S_i, a) = T_j$
- $I' = \{S \subset E | \exists S_0 \in I, \quad S_0 \in S\}$
- $F' = \{S \subset E | \exists S_f \in F, \quad S_f \in F\}$

✎ **4.12 Rendre un automate déterministe**
On commence par vérifier qu'il n'y a qu'un seul état initial puis on enlève les transitions multiples et les transitions vides :

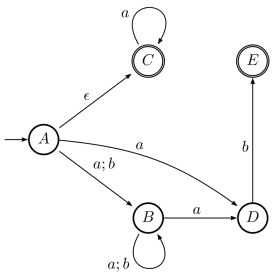
ajouter un nouvel état initial



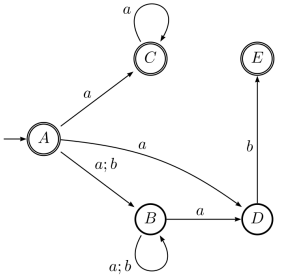
décomposer la transition $B \to abD$



éliminer la transition $A \to \epsilon B$



éliminer la transition $A \to \epsilon C$

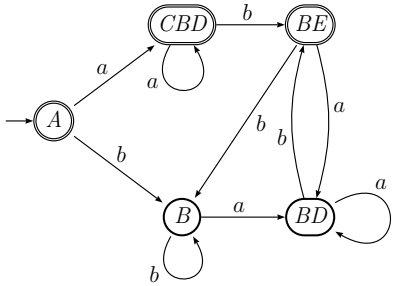


On a maintenant un état initial A et trois états finaux A, C et E. On applique ensuite la procédure pour calculer les nouveaux états en utilisant un tableau. On commence

par placer le singleton contenant l'état initial dans la file des ensembles à traiter et on calcule les ensembles d'états qu'on peut atteindre pour chaque transition possible. On ajoute les nouveaux ensembles obtenus à la File ...sur cet exemple on obtient :

File	a	b
$\{A\}$	$\{C;B;D\}$	$\{B\}$
$\{C;B;D\}$	$\{C;B;D\}$	$\{B;E\}$
$\{B\}$	$\{B;D\}$	$\{B\}$
$\{B;E\}$	$\{B;D\}$	$\{B\}$
$\{B;D\}$	$\{B;D\}$	$\{B;E\}$

Il ne reste plus qu'à représenter le nouvel automate, normalement l'automate possède un seul état initial A et tous les états contenant A, C ou E sont des états finaux soient A, CBD et BE :



La méthode pour rendre un automate déterministe consiste donc à ajouter des états à l'automate, ces états correspondent à des ensembles d'états de l'automate de départ.

4.3 Langages algébriques

L'exemple le plus simple de Langage de type 2, dans la hiérarchie de Chomsky, est celui des expressions bien parenthésées.

4.13 expressions booléennes avec parenthésage : *on peut expliquer le parenthésage d'une expression booléen en ajoutant à la grammaire :*
$S_1 \to Vrai\ S_2 | Faux\ S_2 | P\ S_2 | Q\ S_2 | R\ S_2 | \neg S_1$
$S_2 \to \vee\ S_3 | \wedge\ S_3 | \Longrightarrow\ S_3 | \Longleftrightarrow\ S_3 | \epsilon$
$S_3 \to Vrai\ S_2 | Faux\ S_2 | P\ S_2 | Q\ S_2 | R\ S_2 | \neg\ S_3$
les deux symboles [et] et les règles

$$S_1 \to [\ S_1\ ]; \qquad S_3 \to [\ S_3\ ];$$

mais il est impossible d'écrire ces règles sous la forme d'une grammaire linéaire à gauche (ou à droite). Ce langage est donc de type 2.

Un langage de type 2 peut être représenté par un automate plus complexe appelé **automate à pile**. Vérifier qu'un mot appartient bien à un langage de type 2 s'appelle faire **l'analyse syntaxique**.

Pour montrer qu'un langage n'est pas rationnel il ne suffit de l'écrire avec des règles de dérivation de la forme $A \to aBb$, mais montrer que ces règles ne peuvent pas se simplifier. Un outil très utile pour s'en sortir dans ce cas est le théorème ci-dessous appelé lemme de l'étoile

Théorème 4.12 (lemme de l'étoile)
Soit L un langage rationnel. Il existe un entier K tel que tout mot w de L de longueur $|w| \geq K$ possède une factorisation $w = xyz$ telle que
- $0 < |xy| \leq K$ *et*
- $xy^k z \in L$ *pour tout entier $k \geq 0$.*

en d'autre termes si L est régulier alors $xy^+z \subset L$.

Preuve : Le théorème n'a d'intérêt que si L contient une infinité de mots (sinon il existe un mot de longueur maximale $\max\{|w| \mid w \in L\}$ et il suffit de prendre $K = 0$, $x = y = \epsilon$ et $z = w$ de telle sorte que :

$$xy^k z = \epsilon\epsilon^k w = w \in L$$

Pour un langage contenant une infinité de mots mais décrit par une grammaire formelle finie. Soit n le nombre de symboles non-terminaux du langage L, alors il existe un mot w de longueur $l \geq K = n + 1 : w = a_1 \dots a_i \dots a_l$
Pour vérifier que ce mot appartient au langage on a besoin d'utiliser l règles de dérivations, on passe donc au moins 2 fois par le même non terminal puisque $l > n$:

$$S0 \xrightarrow{a_0} S1 \xrightarrow{a_i} Si \xrightarrow{a_{i+1}} S_{i+1} \longrightarrow \dots \xrightarrow{a_j} Si \longrightarrow \dots \longrightarrow a_l$$

donc le mot w possède un motif que l'on peut répéter k fois (k quelconque) en réutilisant la suite de dérivations $Si \longrightarrow \dots \longrightarrow^{a_j} Si$

$$w = \underbrace{a_1 \dots a_i}_{=x}\underbrace{a_{i+1} \dots a_j}_{=y}\underbrace{a_{j+1} \dots a_l}_{=z} \Longrightarrow \underbrace{a_1 \dots a_i}_{=x}\underbrace{(a_{i+1} \dots a_j)^k}_{=y^k}\underbrace{a_{j+1} \dots a_l}_{=z} = xy^k z \in L$$

il reste à vérifier que $|xy| \leq K$, c'est évident puisque le non-terminal répété ne peut pas l'être à la dernière dérivation qui donne forcément un terminal. □

Le lemme de l'étoile permet de montrer facilement par l'absurde qu'un langage ne peut pas être régulier.

✎ **4.14 Montrer qu'un langage avec parenthésage n'est pas régulier**
Soit un langage L s'écrivant avec pour grammaire

$$\Sigma = \{a;(;)\}, \quad I = \{S\} \quad S \to (S)|aS|\epsilon$$

qui contient des mots bien parenthésés comme :

$$(a), \quad (a(a)), \quad (a(a)aa((a))(aaa)), \quad ((((a)))a((a))), \ldots$$

soit K la valeur du lemme de l'étoile et w un mot du langage L, quitte a lui ajouter des parenthèses par la règle $S \to (S)$ on peut se ramener à ce qu'il commence par $K-2$ parenthèses :

$$w = \underbrace{(\ldots\ldots(}_{K-2 \text{ parenthèses}} \; a \; \underbrace{)\ldots\ldots)}_{K-2 \text{ parenthèses}} = \underbrace{\underbrace{(\ldots\ldots(}_{x} \; \underbrace{a)}_{=y}}_{\text{longueur}=K} \; \underbrace{)\ldots\ldots)}_{z}$$

de telle sorte que z ne contient que $K-3$ parenthèses et xy est de longueur K, mais alors on aurait que

$$w' = \underbrace{\underbrace{(\ldots\ldots(}_{x} \; \underbrace{a)\ldots\ldots a)}_{y^k}}_{} \; \underbrace{)\ldots\ldots)}_{z} \in L$$

alors que ce mot est mal parenthésé puisqu'il contient $K-2$ parenthèses ouvrantes et $K-2+k$ parenthèses fermantes ! D'où la contradiction.

La plupart des langages de programmation, sans être à proprement parler des langages algébriques, en sont assez proches pour que les techniques d'analyse des langages algébriques s'y adaptent. En langage C, le programme Yacc (acronyme de *Yet Another Compiler Compiler*) permet de faire l'analyseur syntaxique (parser en anglais) d'un langage de type 2 à partir de sa description formelle. Il existe aussi une version GNU de yacc appelée Bison.

5 Metanet

Pour illustrer ce cours de Théorie des graphes nous utiliserons en TP le logiciel *Scilab* et sa boîte à outils dédiée aux graphes : Metanet.

L'éditeur de graphes Metanet a subit des changements très importants entre les versions 4 et 5 de *Scilab* qui ne le rendent pas encore totalement fonctionnel pour la version 5. Dans ce cours **Nous utiliserons à la place le logiciel** *Scicoslab* qui est un clone de la version 4 de *Scilab* [10].

Pour l'utilisation de *Scicoslab* en TP reportez vous aux documents suivants disponible sur le site [7]

- un tutoriel général sur *Scicoslab* [8]
- un tutoriel particulier pour metanet [9]

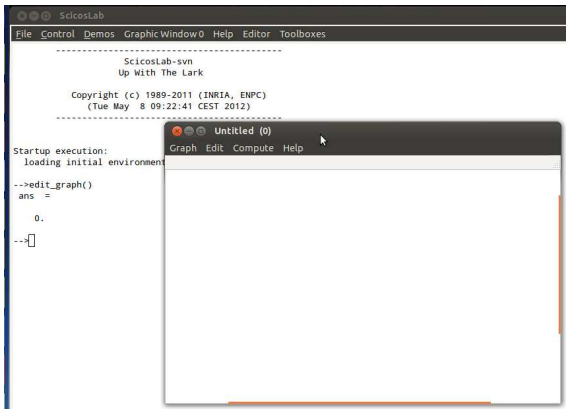
5.1 L'éditeur de graphes metanet

Scicoslab possède une interface graphique spécialement dédiée à la manipulation des graphes *metanet*. Nous allons voir comment l'utiliser pour construire un graphe :

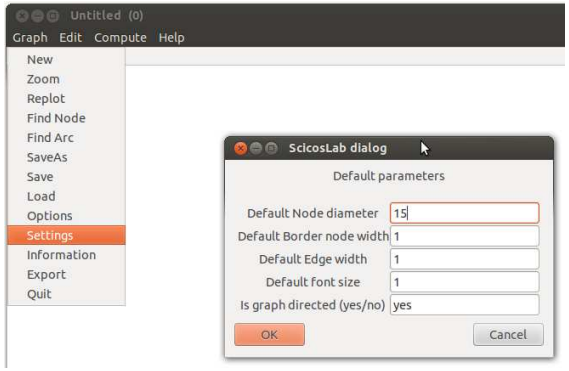
1. Lancer l'éditeur de graphes avec la commande

   ```
   --> edit_graph()
   ```

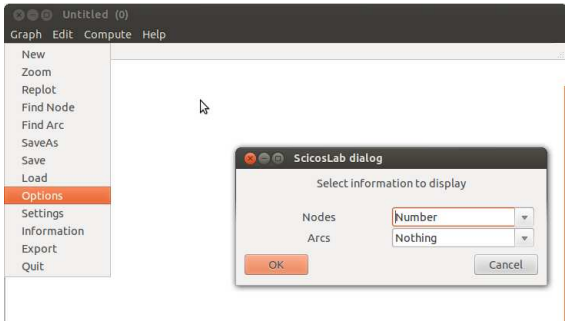
   une nouvelle fenêtre s'ouvre alors :



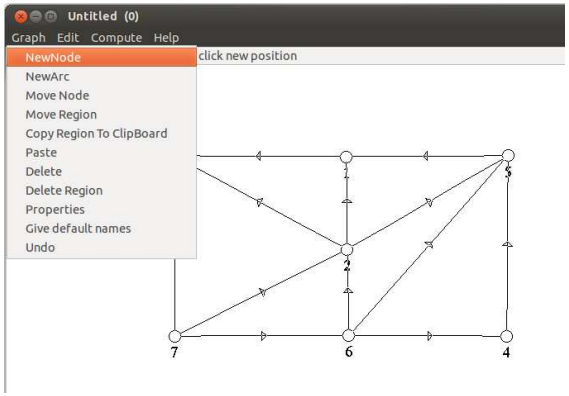
2. Avant de commencer on peut avoir besoin de paramétrer le comportement de cette fenêtre. Dans le menu `graph` de cette fenêtre choisir l'onglet `settings` permet de paramétrer la taille des sommets et l'épaisseur des arcs qui seront dessinés. Mais surtout le dernier paramètre *" is graph directed"* permet de définir le type de graphe qu'on va faire : orienté (*yes*) ou non-orienté (*no*) :



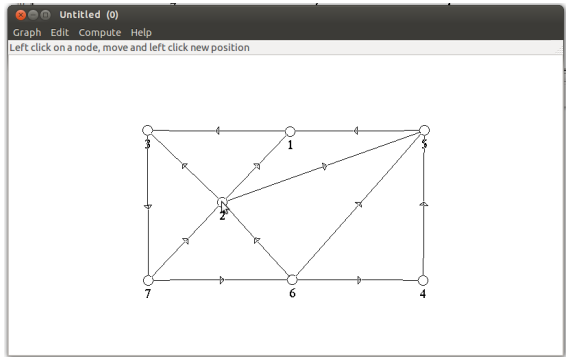
de même l'onglet `options` permet de choisir l'information qui sera indiquée à proximité d'un sommet (node en anglais) ou d'un arc. Je vous conseille de choisir pour le champ `Nodes` le paramètre `number` pour afficher automatiquement son numéro à coté de chaque sommet :



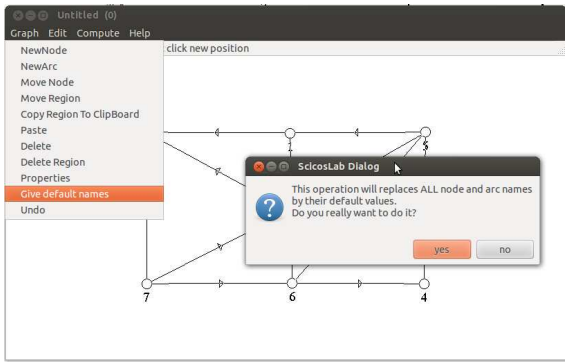
3. Ensuite on peut commencer la construction du graphe. Pour cela nous allons utiliser les fonctionnalités du menu `edit`. Pour créer les sommets choisir `New Node`, à chaque clic gauche vous créez un nouveau sommet à l'endroit du clic le numéro du nouveau sommet est incrémenté à chaque clic (et s'affiche si on l'a spécifié via l'onglet `options` du menu `graph`). Pour créer les arcs choisir `New Arc`, faire un clic gauche sur un sommet existant, pour définir l'origine, puis un autre clic gauche sur un sommet existant, pour définir l'extrémité de l'arc. L'arc s'affiche avec ou sans flèche suivant que le graphe est orienté ou pas (cela a été spécifié via l'onglet `settings` du menu `graph`).



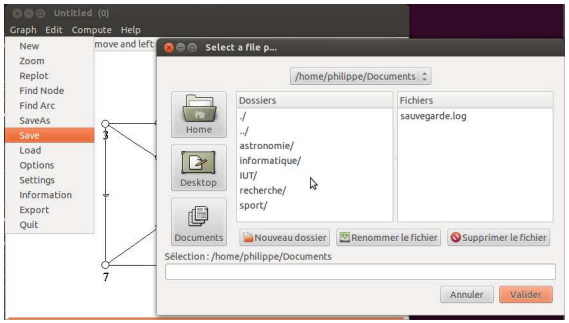
4. Une fois le graphe saisi vous pouvez modifier la position des sommets pour que les arcs ne cachent pas les informations affichées en utilisant la fonction `Move Node` du menu `edit`. Faire un clic gauche sur un sommet puis le déplacer avec la souris. Refaire un clic gauche à la nouvelle place désirée (le sommets et les arcs qui y sont attachés se déplacent en même temps que la souris).



5. Avant de pouvoir sauver le graphe, il faut indiquer à *Scicoslab* comment compléter un grand nombre d'informations relatives au graphe (couleurs des arcs et sommets, noms des sommets , etc...) en utilisant des valeurs par défaut. pour cela il faut choisir `Give default names` dans l'onglet `edit` et cliquer sur `yes` dans la fenêtre qui apparaît ensuite :

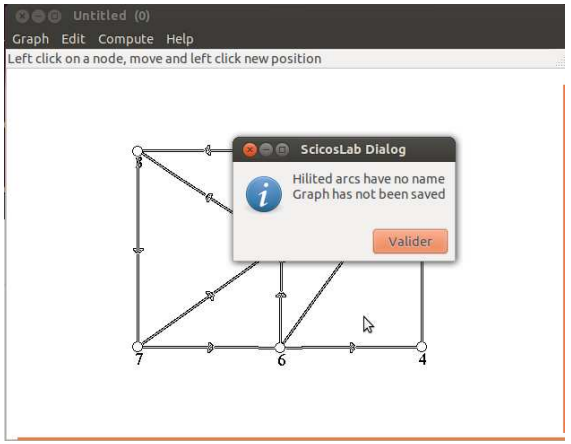


6. Vous pouvez maintenant sauver le graphe en utilisant le menu `SaveAs` dans l'onglet `graph` et choisir, dans la boite de dialogue qui apparaît, un nom de fichier avec l'extension `*.graph` pour sauver le graphe :



Ce fichier sera créé dans le répertoire que vous choisirez (répertoire courant par défaut) et contiendra toute la structure du graphe et va être utilisé dans la console de *Scicoslab* pour effectuer divers calculs sur le graphe.

si vous n'avez pas cliqué sur le menu `Give default names` à l'étape précédente vous ne pourrez pas sauver le graphe et vous aurez le message d'erreur suivant :



5.2 Chargement d'un graphe dans *Scicoslab*

Nous venons de sauver la structure d'un graphe, créé avec metanet, dans un fichier `*.graph`, inversement nous pouvons charger la structure d'un graphe dans *Scicoslab* à partir de fichier `*.graph`. Pour charger le graphe, contenu dans le fichier `G.graph`, dans l'environnement de travail il suffit maintenant d'appeler la commande :

```
--> G=load_graph('G.graph');
```

La commande `G=` sert à stocker le contenu du fichier `G.graph` dans la variable *Scicoslab* `G` (mais on aurait pu choisir tout autre nom de variable valide[7] comme `graphe` ou `monpremiergraphe` ...).

Pour que cela il faut que le fichier `*.graph` se trouve dans le répertoire courant de *Scicoslab* , sinon vous aurez une erreur lors du chargement du graphe :

```
-->G=load_graph('G.graph')
 !--error 9999
Graph file "./G.graph" does not exist
at line      10 of function load_graph called by :
G=load_graph('G.graph')
```

si ce n'est pas le cas n'oubliez pas de changer ce répertoire avec la commande `cd` par exemple :

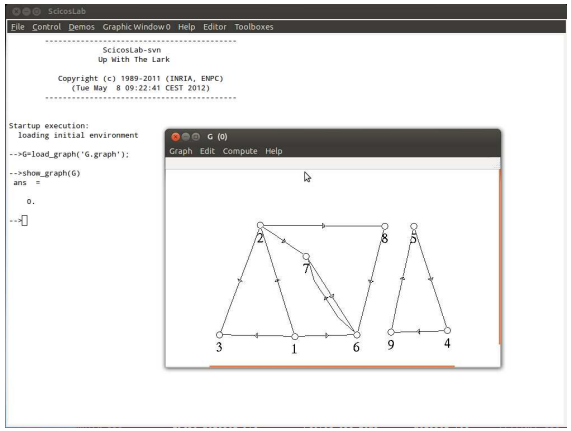
7. ne commençant pas par un chiffre ou un caractère spécial et de moins de 24 caractères

```
--> cd 'Z:/Graphes/' //nouveau répertoire courant Z:/Graphes/
```

Maintenant que le graphe est chargé dans une variable vous pouvez le ré-afficher dans metanet t avec la commande :

```
-->show_graph(G)
```

une nouvelle fenêtre de l'éditeur de graphes s'ouvre avec le graphe `G` dedans :



Attention, si l'éditeur de graphe n'a pas encore été ouvert (ou a été fermé) il faut le reparamétrer (menu `options` de l'onglet `graph`) pour afficher les informations relatives aux sommets et arcs.

Au niveau de l'affichage des graphes on remarquera que :

- les numéros de sommets apparaissent sous le sommet,
- si le graphe est non-orienté les arêtes apparaissent comme de simples traits rectilignes (sauf les boucles)
- si le graphe est orienté les arcs apparaissent sous forme de flèches droites, sauf :
  - — les boucles (qui apparaissent comme un petit cercle)
  - — les "doubles flèches" (qui apparaissent sous forme de deux flèches séparées légèrement courbées)

Il est possible d'utiliser plusieurs fenêtres graphiques différentes avec `show_graph()` en ajoutant l'option `'new'`. Il faudra alors faire attention à bien identifier la fenêtre graphique par défaut lors des appels à `show_graph`. Pour contrôler cela on pourra utiliser les fonctions `netwindows()` et `netwindow()` :

```
-->show_graph(G,'new')//affichage dans une nouvelle fenêtre
 ans  =
    1.
-->netwindows()//liste des fenêtres graphiques + N° de la fenêtre par défaut
```



```
 ans  =
       ans(1)
    0.    1.
       ans(2)
    1.
-->netwindow(0)// choisir la fenêtre graphique 0 par défaut
-->netwindows()//nouvelle liste des fenêtres graphiques
 ans  =
       ans(1)
    0.    1.
       ans(2)
    0.
-->netclose(1)//ferme la fenêtre 1
-->netwindows()//nouvelle liste des fenêtres graphiques
 ans  =
       ans(1)
    0.
       ans(2)
    0.
```

Pour imprimer un graphe (vers un fichier PS, BMP, GIF, ou vers une imprimante) utiliser le menu `export` du l'onglet `graph`.

5.3 Variable de type `graph` dans *Scicoslab*

La variable qui contient les informations du graphe est d'un type particulier, le type `graph`. Il s'agit d'une structure composée de 34 listes (des matrices à 1 ligne) d'ailleurs si vous ne mettez pas le point virgule après la commande `G=load_graph('G.graph')` ou si vous affichez `G` dans la console alors toutes ces informations seront affichées à l'écran ... et il y en a beaucoup et ce n'est pas très lisible :

```
-->G=load_graph('G.graph'); // chargement du graphe
-->typeof(G) // type de la variable G
 ans  =
 graph
-->G  // affichage des données du graphes
 G  =
       G(1)
         column  1 to 13
!graph  name  directed  node_number  tail  head  node_name  node_type  node_x  node_y
         column 14 to 22
!node_font_size  node_demand  edge_name  edge_color  edge_width  edge_hi_width  edge_f
```

```
         column 23 to 29
!edge_min_cap  edge_max_cap  edge_q_weight  edge_q_orig  edge_weight  default_node_
         column 30 to 34
!default_edge_width  default_edge_hi_width  default_font_size  node_label  edge_lab
      G(2)
 G
      G(3)
   1.
      G(4)
   9.
      G(5)
   1.    1.    1.    2.    2.    2.    7.    6.    4.    5.    9.    8.
[More (y or n) ?]
```

Ces 34 listes contiennent les propriétés du graphes, nous aurons besoin d'y accéder pour effectuer certains traitements sur les graphes. Pour un graphe stocké dans la variable `G`, chaque propriété du graphes est accessibles de 3 manières :

- `G(i)` où `i` est le numéro de la propriété
- `G.prop` où `prop` est le nom de la propriété
- `G('prop')` où `prop` est le nom de la propriété


On utilisera le plus souvent deuxième syntaxe `G.prop`. Voici la liste exhaustive que l'on peut aussi obtenir dans l'aide en ligne :

La description de toutes ces listes se trouve dans l'aide en ligne de *Scicoslab* :

```
--> help graph-list
```

en voici un bref récapitulatif :

n°	Nom	type	description
1	graph	`string`	vecteur ligne avec le nom du type `graph` puis les noms des propriétés 2 à 34
2	`name`	`string`	le nom du graphe. C'est une chaîne de caractères (longueur < 80).
3	`directed`	`constant`	flag donnant le type du graphe. Il est égal à 1 (graphe orienté) ou égal à 0 (graphe non-orienté).
4	`node_number`	`constant`	nombre de sommets
5	`tail`	`constant`	vecteur ligne des numéros des sommets origines

n°	Nom	type	description
6	`head`	`constant`	vecteur ligne des numéros des sommets extrémités
7	`node_name`	`string`	vecteur ligne des noms des sommets. Les noms des sommets doivent être différents. Par défaut les noms des sommets sont égaux à leurs numéros.
8	`node_type`	`constant`	vecteur ligne des types des sommets. Le type est un entier entre 0 et 2, 0 par défaut : 0 = sommet normal , 1= puits, 2= source
9	`node_x`	`constant`	vecteur ligne des coordonnées x des sommets. Valeur par défaut calculée.
10	`node_y`	`constant`	vecteur ligne des coordonnées y des sommets. Valeur par défaut calculée.
11	`node_color`	`constant`	vecteur ligne des couleurs des sommets, des entiers correspondants a la table de couleur courante.
12	`node_diam`	`constant`	vecteur ligne des diamètres des sommets en pixels, un sommet est dessiné sous forme d'un cercle. Par défaut, valeur de l'élément `default_node_diam` .
13	`node_border`	`constant`	vecteur ligne de l'épaisseur des bords des sommets. un sommet est dessiné sous forme d'un cercle, par défaut, valeur de l'élément `default_node_border` .
14	`node_font_size`	`constant`	vecteur ligne de la taille de la police utilisée pour afficher le nom du sommet. Les tailles possibles sont : 8, 10, 12, 14, 18 ou 24. Par défaut, valeur de l'élément `default_font_size` .
15	`node_demand`	`constant`	vecteur ligne des demandes des sommets, 0 par défaut ;
16	`edge_name`	`string`	vecteur ligne des noms d'arêtes. Il est souhaitable que les noms des arêtes soient différents, mais c'est n'est pas obligatoire. Par défaut les noms des arêtes sont leur numéros.
17	`edge_color`	`constant`	vecteur ligne des couleurs des arêtes. des entiers correspondants a la table de couleur courante.
18	`edge_width`	`constant`	vecteur ligne des épaisseurs des arêtes en pixels, par défaut, valeur de l'élément `default_edge_width` .

n°	Nom	type	description
19	`edge_hi_width`	`constant`	vecteur ligne des épaisseurs des arêtes mises en évidence (en pixels), par défaut, valeur de l'élément `default_edge_hi_width` .
20	`edge_font_size`	`constant`	vecteur ligne de la taille de la police utilisée pour afficher le nom des arêtes. Les tailles possibles sont : 8, 10, 12, 14, 18 ou 24. Par défaut, valeur de l'élément `default_font_size` .
21	`edge_length`	`constant`	vecteur ligne des longueurs des arêtes, 0 par défaut.
22	`edge_cost`	`constant`	vecteur ligne des coûts des arêtes, 0 par défaut.
23	`edge_min_cap`	`constant`	vecteur ligne des capacités minimum des arêtes, 0 par défaut.
24	`edge_max_cap`	`constant`	vecteur ligne des capacités maximum des arêtes, 0 par défaut.
25	`edge_q_weight`	`constant`	vecteur ligne des poids quadratiques des arêtes, 0 par défaut.
26	`edge_q_orig`	`constant`	vecteur ligne des origines quadratiques des arêtes, 0 par défaut.
27	`edge_weight`	`constant`	vecteur ligne des poids des arêtes, 0 par défaut.
28	`default_node_diam`	`constant`	diamètre par défaut des sommets du graphe, 20 pixels par défaut.
29	`default_node_border`	`constant`	épaisseur du bord des sommets, 2 pixels par défaut.
30	`default_edge_width`	`constant`	épaisseur par défaut des arêtes du graphe,1 pixel par défaut.
31	`default_edge_hi_width`	`constant`	taille par défaut des arêtes mises en évidence (en pixels), 3 pixels par défaut.
32	`default_font_size`	`constant`	taille par défaut de la police utilisée pour afficher le nom des sommets et arêtes. 12 par défaut
33	`node_label`	`string`	vecteur ligne des noms des sommets
34	`edge_label`	`string`	vecteur ligne des noms des arêtes

Les propriétés du graphe sont donc des matrices de réels (`constant`) ou de chaînes de caractères (`string`), à défaut de toutes les retenir on pourra se souvenir qu'elles se regroupent en 6 grandes catégories :

- la propriété 1 contient le nom du type `graph` puis les noms des propriétés 2 à 34,
- les propriétés 2 à 6 comportent les informations minimales pour créer un graphe, son nom,son type,le nombre de sommets, et les arcs sous forme de deux matrices `G.tail` et `G.head` à 1 ligne et m colonnes contenant les

extrémités de chaque arc :

```
-->G.directed // graphe orienté
 ans  =

    1.

-->G.node_number // nombre de sommets
 ans  =

    9.

-->[G.head;G.tail] // liste des arcs
 ans  =

    2.    3.    6.    7.    3.    8.    6.    7.    9.    4.    5.    6.
    1.    1.    1.    2.    2.    2.    7.    6.    4.    5.    9.    8.
```

- les propriétés 7 à 15 sont des vecteurs à 1 ligne et n colonnes décrivant les caractéristique de chaque sommet et dénommés `node_*`, par exemple `G.node_color` contient les couleurs des sommets :

```
-->n=G.node_number//nombre de sommets
 n  =

    9.
-->G.node_color// les couleurs des n sommets
 ans  =

   1.   1.   1.   1.   1.   1.   1.   1.   1.
-->G.node_color=[3 4 5 2 3 4 5 2 3];//on modifie les couleurs
-->G.node_color// nouvelles couleurs des sommets
 ans  =

   3.   4.   5.   2.   3.   4.   5.   2.   3.
```

- les propriétés 16 à 27 sont des matrices à 1 ligne et m colonnes décrivant les caractéristique de chaque arc et dénommés `edge_*`, par exemple `G.edge_color` contient les couleurs des arcs/arêtes :

```
-->G.edge_color//les couleurs des arcs
 ans  =

   1.   1.   1.   1.   1.   1.   1.   1.   1.   1.   1.   1.
-->G.edge_color=5*ones(G.edge_color);//couleurs des arcs à 5(=rouge)
-->G.edge_color//nouvelles couleurs des arcs
 ans  =

   5.   5.   5.   5.   5.   5.   5.   5.   5.   5.   5.   5.
-->G.default_node_border=5;//grossir la taille des sommets
```

- les propriétés 28 à 32 dénommées `default_*` sont des réels donnant certaines valeurs par défaut du graphe, ces valeurs qui peuvent se substituer aux valeurs indiquées par certaines propriétés `node_*` ou `edge_*` le cas échéant. par exemple pour changer la taille par défaut de l'épaisseur du bord d'un sommet :

```
-->G.node_border  // épaisseur du bord de chaque sommet
 ans  =
    0.    0.    0.    0.    0.    0.    0.    0.    0.
```

```
-->G.default_node_border  // épaisseur  par default
 ans  =
    1.
-->G.default_node_border=5;//grossir l'épaisseur du bord des sommets
```

- les propriétés 33 et 34, dénommées `*_label`, peuvent contenir, temporairement, une liste de chaînes de caractères associées aux sommets et aux arcs. Cela peut être pratique pour afficher certaines valeurs dans la fenêtre graphique de metanet (par exemple afficher 2 propriétés sur chaque arc) mais *ces valeurs ne peuvent pas être sauvées dans le fichier* `*.graph`!

Si vous orthographiez mal le nom d'une propriété alors vous aurez une erreur 144 avec un message disant que la fonction `%l_e` n'est pas définie. Exemple ci-dessous avec `edge_number` qui n'est pas une propriété de graphe.

```
-->G.edge_number  // il n'y a pas de propriété edge_number
    !--error 144
Undefined operation for the given operands
  check or define function %l_e for overloading
```

La modification des propriétés de G modifierons l'affichage du graphe lors du prochain appel de `show_graph(G)` :

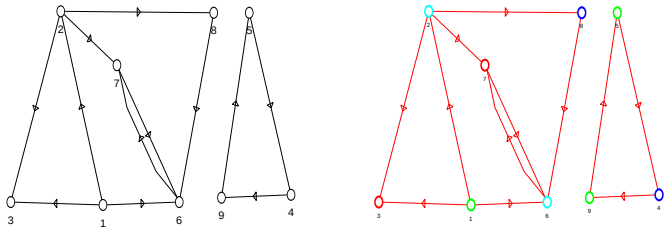


FIGURE 7 – modifications des propriétés graphiques du graphe `G`

La table des couleurs par défaut de *Scicoslab* est la suivante (utiliser `getcolor()`) :

numéro	1	2	3	4	5	6	7	8	...
couleur	noir	bleu	vert	cyan	rouge	magenta	rouge	blanc	...

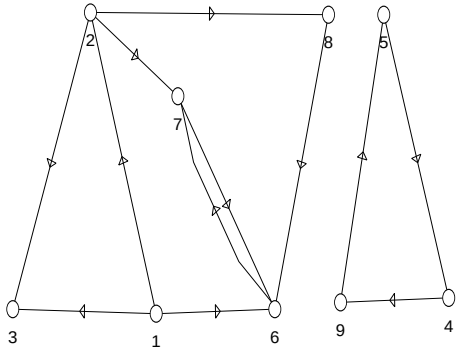
5.4 Quelques fonctions pour les graphes

Nous allons maintenant étudier quelques fonctions utiles qu'on peut appliquer sur un graphe avec *Scicoslab* .

Contrairement à la syntaxe que vous utilisez dans d'autres langages (Java par exemple) on applique une fonction (l'équivalent d'une méthode) à une instance de graphe `G` avec des `paramètres` optionnels en écrivant `fonction(G,paramètres)` et pas `G.fonction(paramètres)`.

Ces actions de base peuvent être faites directement sur les propriétés de la variable `G`, mais pour simplifier leur utilisation elles sont aussi codées sous forme de fonctions. Reprenons le dernier graphe de la partie précédente :

```
-->G=load_graph('G.graph');//chargement du graphe
-->show_graph(G);//affichage du graphe
```



On peut récupérer le nombre de sommets soit directement dans le graphe (avec la propriété `G.node_number` soit avec la fonction `node_number(G)`

```
-->G.node_number//lecture du nombre de sommets dans le graphe
 ans  =
    9.
-->node_number(G)//fonction équivalente
 ans  =
    9.
```

De même on peut récupérer le nombre d'arcs directement dans le graphe en calculant la longueur des listes `G.tail` et `G.head` ou en utilisant une fonction `edge_number(G)` ou `arc_number(G)` (suivant que le graphe est orienté ou pas) :

```
-->length(G.tail)//=calcul du nombre d'arêtes ou d'arcs
 ans  =
    12.
-->arc_number(G)//=nombre d'arcs d'un graphe orienté
 ans  =
    12.
-->edge_number(G)//=nombre d'arêtes pour un graphe non-orienté
 ans  =
    12.
```

pour un graphe non-orienté `arc_number(G)` renvoie 2 fois le nombre d'arêtes !

Ensuite il existe plusieurs fonctions pour rechercher les prédécesseurs, successeurs ou les voisins d'un sommet :

```
-->arcs=[G.head;G.tail]//liste des arcs
 arcs  =
 2.    3.    6.    7.    3.    8.    6.    7.    9.    4.    5.    6.
 1.    1.    1.    2.    2.    2.    7.    6.    4.    5.    9.    8.
-->predecessors(2,G)//prédécesseurs de 2
 ans  =
    1.
-->successors(2,G)//successeurs de 2
 ans  =
    7.    3.    8.
-->neighbors(2,G)//voisins de 2
 ans  =
    1.    3.    7.    8.
```

On évitera d'utiliser les fonctions `predecessors` et `successors` pour un graphe non-orienté (ou ces notions ne sont pas définies), mais on utilisera plutôt dans ce cas la fonction `neighbors`. Par contre pour un graphe orienté `neighbors` renvoie bien la liste des prédécesseurs et des successeurs.

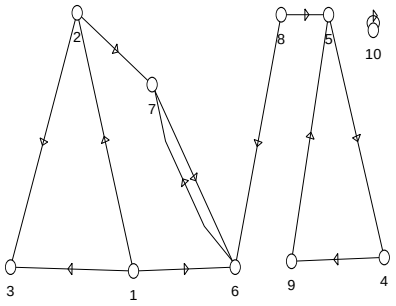
On peut créer un graphe à partir d'une liste d'arcs (deux listes `tail` et `head`) avec la commande `G = make_graph('G',oriente,n,tail,head)` .

On ne peut afficher un graphe `G` que s'il possède des coordonnées pour ses sommets c'est à dire si les propriétés `G.node_x` et `G.node_y` ne sont pas vides. Les graphes créé dans la fenêtre graphique possèdent des coordonnées pour chaque sommets mais pas ceux créé avec `make_graph`. Si on tente de les afficher aura une erreur 15.


```
-->G0=make_graph('G0',1,1,[1],[1]);
-->// G0 n'a pas de coordonnées
-->G0.node_x
 ans  =
     []
-->G0.node_y
 ans  =
     []
-->show_graph(G0)
 !--error 15
submatrix incorrectly defined
at line      31 of function ge_draw_loop_arcs called by :
at line      20 of function ge_drawarcs called by :
at line      10 of function ge_drawobjs called by :
at line       4 of function ge_do_replot called by :
at line      17 of function ge_show_new called by :
at line      49 of function show_graph called by :
show_graph(G0)
```

On peut aussi modifier la structure d'un graphe en enlevant/ajoutant des arcs ou des sommets avec les fonctions `delete_nodes`, `delete_arcs`, `add_node` et `add_edge` :

```
-->G=delete_arcs([2,8],G);//détruire un arc
-->G=add_edge(8,5,G);//ajouter un arc
-->n=node_number(G);//nombre de sommets
-->G=add_node(G,[500,300]);//ajouter un sommet
-->G=add_edge(n+1,n+1,G);//ajouter une boucle
-->show_graph(G)
```



Un graphe doit toujours contenir au minimum un sommet et un arc/arête. Si un algorithme aboutit à détruire tous les arcs/sommets d'un graphe il provoquera une erreur.

```
-->// plus petit graphe possible 1 sommet avec 1 boucle dessus
-->G0=make_graph('essai',1,1,[1],[1]);

-->delete_nodes(1,G0)
 !--error 10000
Cannot delete, a graph must have at least one edge
at line      32 of function delete_nodes called by :
delete_nodes(1,G0)

-->delete_arcs([1,1],G0)
 !--error 10000
Cannot delete, a graph must have at least one edge
at line      43 of function delete_arcs called by :
delete_arcs([1,1],G0)
```

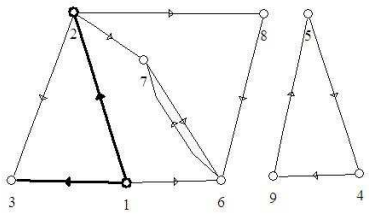
On a enfin des fonctions qui permettent de mettre en évidence (en gras) des sommets ou des arcs :

```
-->G=load_graph('G.graph');//chargement du graphe

-->show_graph(G);//affichage du graphe

-->show_arcs([1,2])//met en gras les arcs 1 et 2

-->show_nodes([1,2],'sup')  //met en gras les sommets 1 et 2
```



Dans beaucoup de cas (comme pour `show_arcs`) il faut comprendre que chaque arcs est numéroté par l'ordre dans le quel ses extrémités apparaissent dans les listes `tail` et `head`! Ici on peut vérifier que les arcs numérotés 1 et 2 sont bien les arcs $(1,2)$ et $(1,3)$:


```
-->arcs=[G.head;G.tail]//liste des arcs
 arcs  =
 2.    3.    6.    7.    3.    8.    6.    7.    9.    4.    5.    6.
 1.    1.    1.    2.    2.    2.    7.    6.    4.    5.    9.    8.
```

Pour finir on a une fonction `G = gen_net()` permet de générer un graphe planaire aléatoire (avec informations graphiques) l'intérêt de cette fonction est qu'elle détermine la position des sommets de telle sorte que les arcs ne se croisent pas. C'est un avantage pour générer facilement des exemples de graphes lisibles (malgré un bug mineur dans cette fonction)

5.5 Exercices

Pour finir quelques petits exercices pour comprendre comment fonctionnent les fonctions de base sur les graphes.

5.1 Voisins dans un graphe *Soit G un graphe simple orienté. Écrire les fonctions Scicoslab suivantes, sans utiliser les fonctions* `predecessors`, `successors`, `neighbors`, *mais en accédant directement aux propriétés du graphe G :*

- `L=predecesseurs(x,G)` *liste des prédécesseurs de* `x` *dans* `G`
- `L=successeurs(x,G)` *liste des successeurs de* `x` *dans* `G`
- `L=voisins(x,G)` *liste des voisins de* `x` *dans* `G`

solution : l'idée est de parcourir la liste des arcs (s,t) et quand l'un des sommets est égal à x l'autre est un prédécesseurs (ou un successeur suivant le cas) de x. Ensuite il y a de nombreuses manières de mettre en œuvre cette stratégie soit en utilisant des boucles (`for` ou `while`) soit en utilisant la fonction de recherche `find`, ce qui simplifie grandement l'écriture des fonctions.

```
function L=predecesseurs(x,G)
//solution classique avec une boucle
tail=G.tail,head=G.head//liste des arcs
m=length(tail)//nombre d'arcs
L=[]//liste des prédécesseurs
i=0
while i<m//parcours de la liste des arcs
      i=i+1
      if head(i)==x then L=[L, tail(i)]
      end
end
endfunction

function L=successeurs(x,G)
//solution en utilisant find
tail=G.tail,head=G.head//liste des arcs
position=find(tail==x)//find trouve les i où tail(i)==x
L=head(position)//liste des successeurs
endfunction

function L=voisins(x,G)
//solution mixte
tail=G.tail,head=G.head//liste des arcs
m=length(tail)//nombre d'arcs
L=[]//liste des voisins
for i=1:m//parcours de la liste des arcs
      //les prédécesseurs
      if head(i)==x then v=tail(i)
                  //on ajoute v à L s'il n'y est pas déjà
                  if find(L==v)==[] then L=[L, v]
                  end
      end
      //les successeurs
      if tail(i)==x then v=head(i)
                  //on ajoute v à L s'il n'y est pas déjà
                  if find(L==v)==[] then L=[L, v]
                  end
      end
end
endfunction
```


✎ **5.2 Numérotation des arcs** *Soit* `G` *un graphe simple, orienté ou non, et écrire une fonctions Scicoslab* `k=arc_2_num(x,y,G)` *qui calcule la liste des numéros* `k(i)` *de chacun des arcs* `(x(i),y(i))` *si ils existent, ou* `k=[]` *sinon.*

solution : la difficulté ici est double, il faut traiter plusieurs arcs dans la même fonction et traiter à la fois les graphes orientés et non-orientés.

```
function k=arc_2_num(x,y,G)
arcs=[G.tail;G.head]//liste des arcs
k=[]//initialisation de k
n=length(x)//nombre d'arc (x,y) à traiter
i=0
while i<n//parcours des listes x et y
      i=i+1
      X=x(i),Y=y(i)
      //ind=position de (x(i),y(i)) dans la liste des arcs
      ind=vectorfind(arcs,[X;Y],'c')
      k=[k ind]//on l'ajoute à la fin de k
      //pour les graphes non-orientés
      //on cherche aussi l'arc (y(i),x(i))
      if G.directed==0 then ind=vectorfind(arcs,[Y;X],'c')
                            k=[k ind]
      end
end
endfunction
```

✎ **5.3 Générateur de graphes planaires** *A partir de la fonction* `gen_net()` *créer une fonction* `gen_graph(n,directed)` *qui génère aléatoirement un graphe à n sommets orienté si* $n = 1$ *et non-orienté sinon.*

```
function G=gen_graph(n,varargin)
//n=nombre de sommets
//varargin=paramètre optionnel (direct) 0 ou 1
//G= graphe planaire à n sommets
//récupération de la variable direct
//direct=1 si G est orienté (par défaut) et 0 sinon
if length(varargin)>0 then direct=varargin(1)
                      else direct=1 //orienté par défaut
end
//initialisations
dt=getdate() //récupération de la date
//dt sert à fabriquer une nouvelle graine pour random
seed=(sum(dt([3 5]))-1)*prod(1+dt([2 6 7 8 9]))
v=[seed,n,1,1,0,20,50,50,0,20,100,100];
// paramètres pour gen_net
G=gen_net('G',direct,v)// génération du graphe
G(24)=null() //bug dans gen_net
//modification de certaines propriétés
m=length(G.tail)//nombre d'arc
G.node_type=zeros(1,n)//élimination des puits et sources
G.node_color=ones(1,n)//couleur noir pour tous les sommets
G.edge_color=ones(1,m)//couleur noir pour tous les arcs
//valeurs par défaut pour l'affichage du graphe
G.default_node_diam=10
G.default_node_border=5
G.default_edge_width=1
G.default_edge_hi_width=3
G.default_font_size=14
endfunction
```


Bibliographie

[1] Bertrand Hauchecorne, Daniel Surrateau *Des mathématiciens de A à Z*

[2] http://fr.wikipedia.org/wiki/Théorie_des_graphes

[3] http://fr.wikipedia.org/wiki/Théorie_des_langages

[4] http://math.dartmouth.edu/~euler/docs/originals/S053.pdf

[5] http://www.planarity.net/

[6] http://www.posteet.com/view/221

[7] http://perso.univ-rennes1.fr/philippe.roux/

[8] tutoriel pour scilab :http://perso.univ-rennes1.fr/philippe.roux/scilab/intro/fiche_scilab.pdf ou http://perso.univ-rennes1.fr/philippe.roux/scilab/intro/fiche_scilab.html

[9] tutoriel pour metanet :http://perso.univ-rennes1.fr/philippe.roux/scilab/metanet/fiche_metanet.pdf ou http://perso.univ-rennes1.fr/philippe.roux/scilab/metanet/fiche_metanet.html

[10] http://www.scicoslab.org/

Liste des exercices

Index