ALGORITHMIQUE DES GRAPHES

Jean-Michel Hélary
IFSIC[1]

Cours C66
Juin 2004

Table des matières

Chapitre 1

Introduction, exemples

1.1 Historique

La théorie des graphes a pris naissance en 1736 : le mathématicien allemand L. Euler apporta une réponse au problème que se posaient les habitants de la ville de Koenigsberg ; à savoir : *comment traverser les sept ponts de cette ville sans jamais passer deux fois par le même.*

Jusqu'en 1946, la théorie des graphes reste du domaine des mathématiques; les mathématiciens ayant associé leurs noms aux travaux les plus marquants sont, entre autres :

- au 19ème siècle : Kirchoff, Hamilton, Sylvester, Kempe, Lucas, Petersen, Tarry
- au 20ème siècle (1ère moitié) : Poincare, Sainte-Lagüe, Kuratowski, Hall, Polya, König, Whitney, Tutte.

Des recherches militaires liées au conflit mondial de 1939-45 naît la *Recherche Opérationnelle* provoquant un développement de la *théorie des graphes* comme modèles de problèmes concrets ; cet aspect est encore renforcé, dans les années qui suivent, par le développement des Sciences Économiques et de Gestion. Les grands spécialistes de cette nouvelle orientation des graphes seront des auteurs tels que Kühn, Dantzig, Ford, Fulkerson, Ghouila-Houry, Harary, Saaty, Bellmann, Roy, Berge, Faure, Kaufmann, etc... Une troisième époque a été ouverte dans les années 1960, avec le développement des sciences de l'information et de la communication. Des informaticiens tels que Dijkstra, Knuth, Wirth, Sakharovitch, Gondran et beaucoup d'autres, participent au développement de l'algorithmique des graphes autant qu'à la modélisation de problèmes nés de ces nouvelles sciences en terme de graphes. Une approche historique se trouve dans l'introduction de [GM79], qui renvoie d'ailleurs à un ouvrage historique complet sur la première époque de la théorie des graphes (1746-1936) [BLW76].

1.2 Relations graphes et informatique

Les relations entre graphes et informatique sont à double sens :

1. l'informatique, outil de résolution de problèmes de graphes
2. les graphes, outil de modélisation de problèmes informatiques.

On aboutit donc, dans certaines situations, à l'informatique comme outil de résolution de ses propres problèmes (via la modélisation en termes de graphes). Ce cours s'intéresse essentiellement

à la partie 1. Outre la nécessité qu'il y a, pour un informaticien, de connaître les algorithmes adaptés aux principaux types de problèmes de graphe et la manière de les programmer, il faut mettre l'accent sur un aspect essentiel : l'étude des techniques de programmation d'algorithmes de graphe. Cette étude est enrichissante par elle- même car elle donne l'occasion d'aborder sur des exemples non triviaux des notions d'algorithmique aussi fondamentales que les structures de données et l'adéquation de la représentation au problème traité, les niveaux de programmation (types abstraits et mise en oeuvre), la complexité, etc..

Cependant la modélisation de problèmes en terme de graphes ne doit pas être négligée. Nous donnons dans la section suivante trois exemples de situations– d'origine informatique ou non – qui se prêtent à une telle modélisation.

1.3 Exemples

1.3.1 Le problème du chou, de la chèvre et du loup

L'énoncé

Un passeur, disposant d'une barque, doit faire traverser une rivière à un chou, une chèvre et un loup. Outre le passeur, la barque peut contenir au plus une des trois unités qui doivent traverser. D'autre part, pour des raisons de sécurité, le passeur ne doit jamais laisser seuls la chèvre et le chou, ou le loup et la chèvre. (Par contre, le loup ne mangera pas le chou... et réciproquement). Comment le passeur doit-il s'y prendre ?

Modélisation

Il s'agit d'une situation très classique d'un système à états possédant un état initial, un état final, et des transitions autorisées. Chaque état se traduit par un sommet, ou point. Chaque transition autorisée d'un état à un autre se traduit par un arc, allant du premier état vers le second. Dans cet exemple, un état est une configuration possible sur la rive de départ. Désignons par `P` le passeur, `L` le loup, `X` la chèvre, `C` le chou. Une configuration sera alors désignée par l'ensemble des présents sur la rive de départ. L'état initial est `PLXC` et l'état final $\emptyset$. Chaque arc, représentant une transition possible, sera libellé –pour la clarté de la modélisation– par la description de l'action entreprise (`A` signifie : *aller* , `R` signifie *retour*. On obtient alors le graphe suivant :

Problème

Chaque solution correspond à un chemin dans le graphe, de `PLXC` à $\emptyset$ et alternant les étiquettes `A` et `R`. Un chemin correspond à une séquence d'actions enchaînables les unes aux autres (l'état initial d'une action est égal à l'état final de la précédente).

Résolution

Pour l'instant, "à la main" (ou "au pif", comme on veut !), on trouve deux solutions :
soit 7 traversées simples, quelle que soit la solution. Laissons au lecteur le soin de fournir au passeur, en langage naturel, les explications nécessaires à l'accomplissement de sa mission.

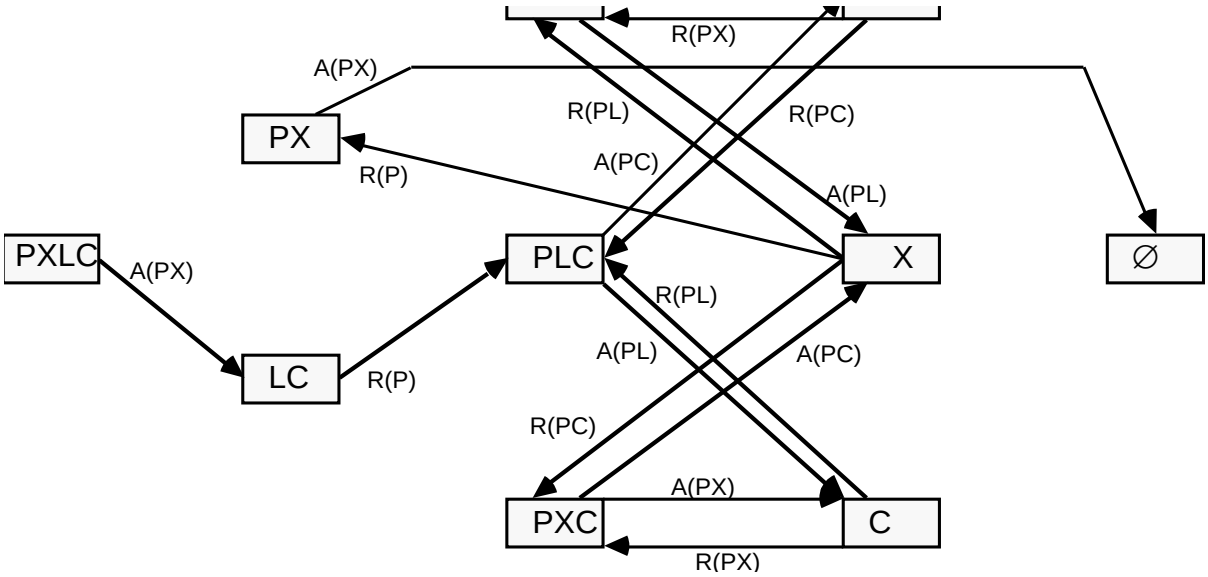


FIG. 1.1 – *graphe des transitions*

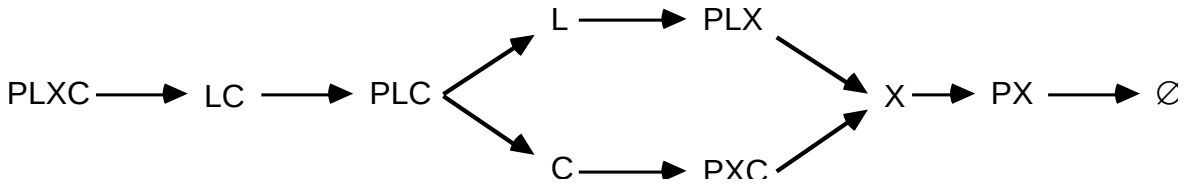


FIG. 1.2 – *Solution du problème des traversées*

1.3.2 Blocage mutuel dans un partage de ressources

3 philosophes chinois `A, B, C` possèdent, à eux trois, trois baguettes `R, S, T`. Pour manger, chacun a besoin de 2 baguettes. Lorsqu'un des philosophes désire manger, il requiert deux baguettes et ne les libère que lorsqu'il a fini son repas[1]. Par contre, tant qu'il n'a pas obtenu les deux baguettes, il ne peut rien faire qu'attendre : même s'il a eu la chance d'obtenir une baguette, il ne la rend pas tant qu'il n'a pas pu aller au bout de son repas.

Considérons alors la situation suivante : les 3 philosophes ont envie de manger exactement au même moment, et chacun se précipite sur les baguettes... mais chacun n'en obtient qu'une. Pour fixer les idées :

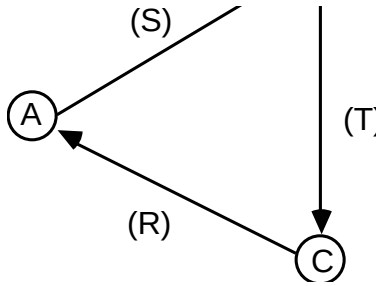
```
A possède R et requiert S
```

1. après les avoir soigneusement lavées!

B possède S et requiert T
C possède T et requiert R

Bigre ! Il va bien falloir faire preuve de patience... et de philosophie. Ils ne peuvent pas sortir de ce blocage mutuel, et vont donc mourir de faim, sauf si quelqu'un (c'est-à-dire l'un d'eux, ou un personnage extérieur) se rend compte de la situation et intervient autoritairement, par exemple en libérant de force une des baguettes pour l'attribuer à un des deux autres.

Cette situation est un exemple de ce qui peut se passer dans les systèmes informatiques complexes, où plusieurs entités (processeurs, périphériques,etc.) se partagent des ressources. Les entités en question étant *a priori* moins intelligentes que les philosophes chinois, il faut pallier cette lacune en étant capable de détecter automatiquement les situations d'interblocage. Quel que soit le nombre d'entités et de ressources, les configurations de possession/requête des ressources par les entités peuvent être modélisées par un graphe de la manière suivante :
. chaque entité est représenté par un sommet
. un arc existe, depuis le sommet `X` vers le sommet `Y` si `X` requiert une ressource que possède `Y`.
Ainsi, dans l'exemple des philosophes et des baguettes, on obtient le graphe dessiné figure 1.3 :



Le libellé sur l'arc désigne la ressource demandée (par l'origine) et possédée (par l'extrémité)

FIG. 1.3 – *graphe des attentes*

Un tel graphe est dynamique dans la mesure où les ressources sont libérées et ré-allouées au cours du temps. Mais à chaque instant, il y a un graphe et un seul et de plus, si une situation d'interblocage apparaît, le graphe n'évolue plus. Maintenant, il est clair que l'interblocage entre plusieurs entités se produit si et seulement si les sommets correspondants sont situés sur un circuit, c'est à dire un chemin dont l'extrémité coïncide avec l'origine. C'est donc un *algorithme de détection de circuit* qui permettra, ici, de résoudre le problème de détection de l'interblocage.

Le problème présenté (interblocage) est un exemple d'un type de situation très courant mettant en jeu des objets (les entités) et des relations entre ces objets (la requête d'une ressource possédée par un autre).

1.3.3 Fiabilité dans les réseaux

Un réseau est un système de communication dans lequel des sites communiquent entre eux soit directement, soit par l'intermédiaire d'autres sites, en envoyant des messages qui circulent le long de lignes de communication. La représentation de tels réseaux par des graphes est ici évidente, les sites et les lignes correspondant trivialement aux sommets et aux arcs. Mais selon

le genre d'études que l'on souhaite effectuer, le graphe modélisant le réseau devra être précisé; par exemple :

- les communications sont-elles uni- ou bidirectionnelles?
- Un message émis parvient-il sûrement ou de manière aléatoire à destination?
- Les lignes de communication peuvent-elles acheminer une quantité illimitée ou non d'informations par unité de temps? (débit d'une ligne)

etc...

Autant de problèmes, autant de solutions que l'on cherche à exprimer en termes d'algorithmes de graphe. A titre d'exemple, nous montrons deux problèmes liés à des études de fiabilité.

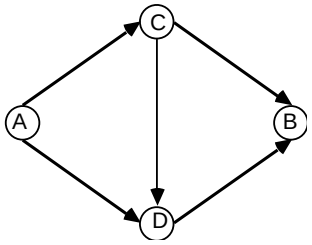


FIG. 1.4 – *réseau 1*

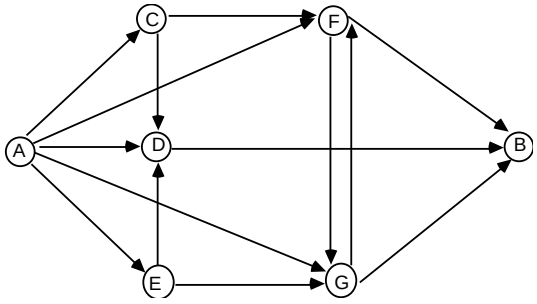


FIG. 1.5 – *réseau 2*

Exemple 1. Résistance aux pannes Dans un réseau, il peut se produire qu'une ligne de communication soit coupée, ne transmettant plus aucun message. Il en est de même pour un site dont la panne peut être assimilée à la coupure de toutes les lignes qui y aboutissent ou qui en partent. Supposons que, dans un réseau, on achemine des informations d'un sommet `A` vers un sommet `B`, éventuellement via des sites intermédiaires servant de relais. De telles informations doivent emprunter des chemins allant de `A` à `B`. Ainsi, une information issue de `A` parviendra à `B` si et seulement si au moins un chemin est valide (c'est à dire la séquence d'arcs qui le constitue ne comporte que des arcs non détériorés). Le problème posé est alors le suivant : quel est le nombre maximum de pannes auquel cet acheminement peut résister? Autrement dit, quel est le nombre maximum de chemins indépendants existant entre A et B? (deux chemins sont indépendants si et seulement si ils sont disjoints au sens des arcs et des sommets intermédiaires).

Dans le réseau 1, ce nombre est égal à 2 car, s'il n'y a qu'une panne, quelle que soit sa localisation (site ou ligne) il reste toujours un chemin valide. Par contre il existe des configurations à 2 pannes (lignes `AC` et `DB` par exemple) auxquelles l'acheminement ne résiste pas.

Dans le réseau 2, ce nombre est égal à 3 (sites `D, F, G`).

Exemple 2. Routage de fiabilité maximale Dans ce modèle, chaque ligne (arc) est munie d'une valeur, égale à la probabilité qu'un message émis à l'origine parvienne sans altération à l'extrémité. Un émetteur `A` émet vers un récepteur `B`, plusieurs parcours d'acheminement étant possibles. Si on suppose que les altérations des messages sont des évènements indépendants, la probabilité de non altération le long d'un chemin est donc égale au produit des probabilités de chaque arc le constituant. On cherche donc à déterminer un routage de fiabilité maximale de `A` à `B` (c'est à dire un chemin de `A` à `B` de probabilité maximale), représenté de la manière suivante : à chaque sommet de ce chemin, on associe le sommet suivant.

Chapitre 2

Graphes : un modèle mathématique

2.1 Définitions mathématiques et notations

2.1.1 L'objet mathématique

2.1.1.1 Vision ensembliste

Définition 2.1 *Un graphe est un doublet* $G = (X,\Gamma)$*, où*
X *= ensemble –en général fini– appelé ensemble des* **sommets**
Γ*=ensemble de couples* $(x,y) \in X^2$*, appelé ensemble des* **arcs**.

Ainsi, $\Gamma \subseteq X^2$. Par définition d'un ensemble, entre deux sommets donnés il existe au plus un arc; on peut toutefois, dans certains cas, considérer des multi-graphes dans lesquels plusieurs arcs distincts existent entre deux sommets, mais il faut alors adopter une définition différente pour Γ.

Exemple : graphe G1
$X = \{1,2,3,4,5,6,7,8,9\}$

$\Gamma = \{(1,2),(1,3),(2,5),(3,3),(3,4),(3,6),(4,2),(4,5),(5,6),(5,7),(5,8),(6,3),(8,7)\}$

2.1.1.2 Vision fonctionnelle

Définition 2.2 *Un graphe est un doublet* $G = (X,\Gamma)$*, où*
X *= ensemble des sommets*
Γ*= fonction de* X *dans l'ensemble* 2^X *des parties de* X.

Ainsi, à tout sommet x est associé l'ensemble $\Gamma(x) \subseteq X$. La notation 2^X pour désigner l'ensemble des parties de X vient du fait que, si X est de cardinal fini $|X|$, le nombre de sous-ensembles

(parties) de X est égal à $2^{|X|}$. En effet, si $X = \{x_1,x_2,\ldots,x_n\}$ la construction d'un sous-ensemble de X revient à décider, pour chaque x_i, si cet élément fait partie ou non du sous-ensemble. Il y a donc n décisions binaires indépendantes, soit $2 \times 2 \times \ldots \times 2 = 2^n = 2^{|X|}$ constructions possibles.

Exemple : graphe G1
$X = \{1,2,3,4,5,6,7,8,9\}$

$\Gamma(1) = \{2,3\}$; $\Gamma(2) = \{5\}$; $\Gamma(3) = \{3,4,6\}$;
$\Gamma(4) = \{2,5\}$; $\Gamma(5) = \{6,7,8\}$; $\Gamma(6) = \{3\}$;
$\Gamma(7) = \emptyset$; $\Gamma(8) = \{7\}$; $\Gamma(9) = \emptyset$

2.1.1.3 Vision relationnelle

A tout graphe $G = (X,\Gamma)$ est associée une relation binaire $\mathcal{R}$ sur X, et réciproquement, par l'équivalence suivante :

$$x\mathcal{R}y \Longleftrightarrow (x,y) \in \Gamma$$

Nous verrons ultérieurement qu'à chaque propriété des relations binaires correspond une propriété du graphe.

Équivalences et ordres Parmi les relations binaires, celles qui vérifient les propriétés de *réflexivité* et *transitivité* sont particulièrement importantes. On les appelle des *pré-ordres*, car elles elles expriment en particulier les relations de classement avec ex-aequo.
Exemple de pré-ordre Soit E un ensemble d'éléments. Chaque élément x est muni d'une estampille entière $n(x)$. Il est facile de voir que la relation $x \sim y \Leftrightarrow n(x) \leq n(y)$ est un pré-ordre. Deux éléments distincts x et y sont ex-aequo si $n(x) = n(y)$. La figure 2.1.a montre le graphe d'une telle relation.

Un pré-ordre est spécialisé en *équivalence* ou en *ordre* selon qu'il est symétrique ou anti-symétrique. Ainsi, la relation d'ex-aequo est une équivalence car elle est symétrique (la figure 2.1.b en montre le graphe) tandis que la relation $x \prec y \Leftrightarrow x = y \vee n(x) < n(y)$ est un ordre car elle est antisymétrique (figure 2.1.c). Rappellons qun ordre est dit *total* si pour tout couple d'éléments x et y, on a toujours $x\mathcal{R}y$ ou $y\mathcal{R}x$. Un ordre non complet est dit *partiel*. C'est le cas notamment de l'exemple de la figure 2.1.c), car certains éléments ne sont pas comparables.

2.1.2 Vocabulaire

Pour un arc $(x,y) \in \Gamma$, x est l'**origine** et y est l'**extrémité** de l'arc
On dit aussi que y est **successeur** de x, ou que x est **prédécesseur** de y.
On note :

$\Gamma(x) = \{y \in X \mid (x,y) \in \Gamma\}$ l'ensemble des successeurs de x

$\Gamma^{-1}(x) = \{z \in X \mid (z,x) \in \Gamma\}$ l'ensemble des prédécesseurs de x

Élément	a	b	c	d	e	f	g	h	i	j	k
Estampille	10	5	4	5	7	7	3	4	5	2	10

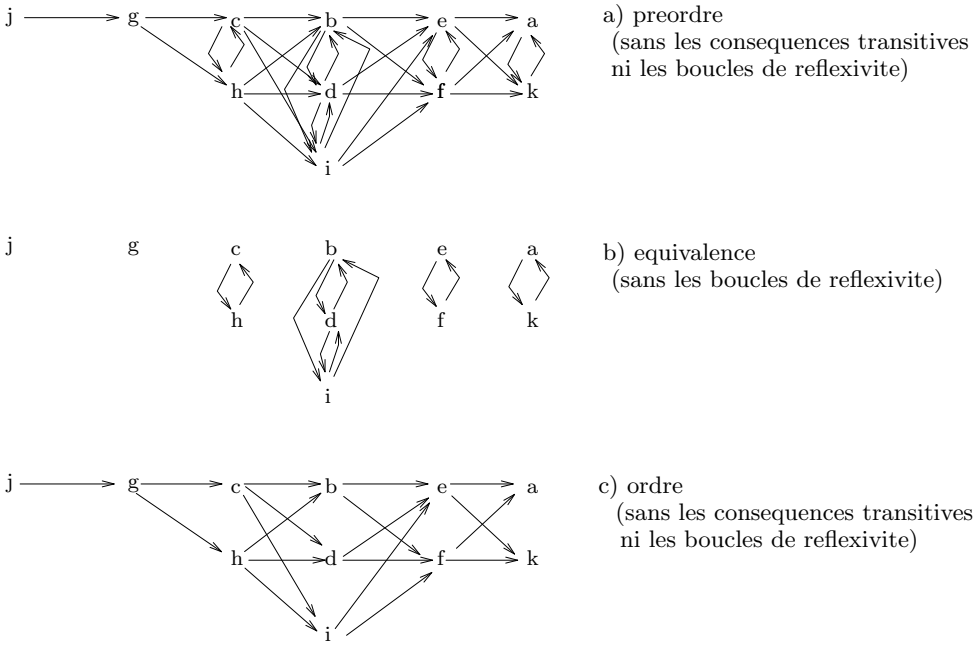


FIG. 2.1 – *Pré-ordre, équivalence et ordre*

Si $A \subset X$:

$$\Gamma(A) = \bigcup_{x \in A} \Gamma(x)$$

$$\Gamma^{-1}(A) = \bigcup_{x \in A} \Gamma^{-1}(x)$$

$d^+(x) = card(\Gamma(x))$ est le $\frac{1}{2}$ **degré extérieur** de x

$d^-(x) = card(\Gamma_{-1}(x))$ est le $\frac{1}{2}$ **degré intérieur** de x

$d(x) = d^+(x) + d^-(x)$ est le **degré** de x

2.1.3 Propriétés de graphes

On montre, pour quelques propriété des relations binaires, une propriété caractéristique du graphe correspondant. La caractérisation de la transitivité ne sera abordée qu'au chapitre 5 (fermeture transitive).

Dans ce qui suit, on pose $\Delta = \{(x,x) | x \in X\}$ (ensemble des "boucles").

Réflexivité $\forall x \in X : (x,x) \in \Gamma$.
Boucle sur chaque sommet .
$\Delta \subset \Gamma$

Anti-réflexivité $\forall x \in X : (x,x) \notin \Gamma$.
Graphe sans boucle .
$\Delta \cap \Gamma = \emptyset$

Symétrie $\forall x \in X, \forall y \in X : (x,y) \in \Gamma \Longrightarrow (y,x) \in \Gamma$.
L'existence d'un arc implique l'existence de l'arc opposé.
$\Gamma = \Gamma^{-1}$

Anti-symétrie $\forall x \in X, \forall y \in X : (x,y) \in \Gamma \wedge (y,x) \in \Gamma \Longrightarrow x = y$.
L'existence d'un arc interdit l'existence de l'arc opposé, sauf dans le cas d'une boucle.
$\Gamma \cap \Gamma^{-1} \subset \Delta$

2.2 Notion de chemin

Définition 2.3 *Un chemin de G est une suite de sommets*
$[x_{i_1}, x_{i_2}, \ldots, x_{i_k}]$
telle que :
$(x_{i_1}, x_{i_2}) \in \Gamma, \quad (x_{i_2}, x_{i_3}) \in \Gamma, \quad \ldots, \quad (x_{i_{k-1}}, x_{i_k}) \in \Gamma$
x_{i_1} *en est l'*origine, x_{i_k} *l'*extrémité *et les* x_{i_j} $(2 \leq j \leq k-1)$, *s'ils existent, les* sommets intermédiaires.

Un chemin allant de x à y, dont on ne précise pas les sommets intermédiaires, sera noté :

$\boldsymbol{u} = [x * y]$

Si l'on souhaite désigner des sommets intermédiaires, on utilisera la notation telle que :

$\boldsymbol{u} = [x_1 * x_p * x_q * x_k]$

y est alors un *descendant* de x et x un *ascendant* de y.

La *longueur* du chemin $\boldsymbol{u}$, ou $|\boldsymbol{u}|$, est le nombre d'arcs qui le composent.
$\boldsymbol{u} \cdot \boldsymbol{v}$ désigne la *concaténation* de deux chemins tels que
extrémité de $\boldsymbol{u}$ = origine de $\boldsymbol{v}$; on a alors $|\boldsymbol{u} \cdot \boldsymbol{v}| = |\boldsymbol{u}| + |\boldsymbol{v}|$.
Un *circuit* est un chemin dont l'extrémité coïncide avec l'origine.
Un chemin (resp. circuit) *élémentaire* est défini par une suite de sommets sans répétition (sauf l'origine et l'extrémité pour un circuit)

2.3 Représentations de l'objet mathématique

On s'intéresse aux représentations visuelles, destinées au traitement "à la main". On peut utiliser :

a)- le graphique (diagramme sagittal) Exemple : graphe G1

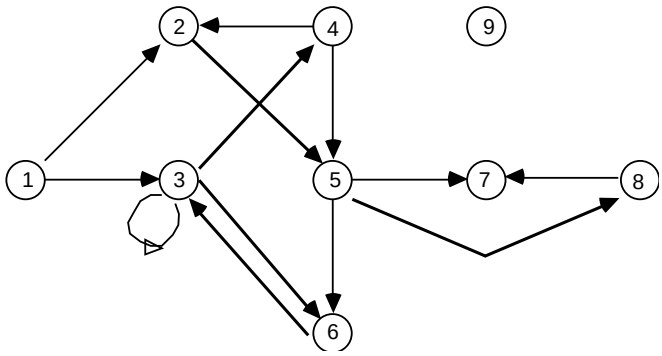


FIG. 2.2 – *Diagramme Sagittal*

Cette représentation pose des problèmes de topologie (croisements d'arcs) et de lisibilité lorsqu'il y a beaucoup de sommets et/ou d'arcs. Leurs attributs sont portés sur le graphique, ce qui peut aussi être source d'ambiguïté. Elle n'est adaptée en général que pour les "petits" graphes ($card(X) \leq 10$).

b)- un tableau d'attribution des arcs. S'il s'agit de représenter l'existence des arcs, ce tableau sera une *matrice booléenne*.

extrémités → *origines* ↓	1	2	3	4	5	6	7	8	9
1		1	1						
2					1				
3			1	1		1			
4		1			1				
5						1	1	1	
6			1						
7									
8							1		
9									

c)- dictionnaire des successeurs

1	2, 3
2	5
3	3, 4, 6
4	2, 5
5	6, 7, 8
6	3
7	/
8	7
9	/

resp. des prédécesseurs

1	/
2	1, 4
3	1, 3, 6
4	3
5	2, 4
6	3, 5
7	5, 8
8	5
9	/

Un seul de ces deux tableaux suffit à spécifier complètement le graphe.

Chapitre 3

Aspect algorithmique : un type de données abstrait

Dans ce chapitre on donne d'abord trois exemples de représentation algorithmique. Puis on fait *abstraction* des représentations particulières en définissant le type de données abstrait *GRAPHE*. Le problème est de représenter, à l'aide de structures de données algorithmiques, le doublet mathématique (X,Γ) avec $\Gamma \subseteq X^2$.

3.1 Les sommets et les arcs

La définition des types Sommet et Arc est utilisée dans toutes les représentations de graphes. Elle est présentée ci-dessous.

3.1.1 Le type Sommet

Tout sommet est identifié par un numéro (entier), qui sera désigné par *num*. En général, on suppose que *num*≥ 1. La notation algorithmique est la suivante :

```
ENT SOMMET::num ;
```

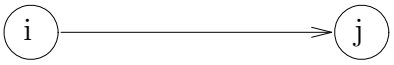
signifiant que les objets de type *SOMMET* ont un attribut de nom *num* et de type entier. L'expression *x.num* désigne alors le numéro du sommet *x*.

3.1.2 Le type Arc

Tout arc possède deux sommets (son origine et son extrémité).

```
SOMMET ARC::ori ;
SOMMET ARC::ext ;
```

Si γ est un objet de type arc, l'expression γ*.ori* (resp. γ*.ext*) désigne le sommet origine (resp. extrémité) de l'arc γ, et γ*.ori.num* (resp. γ*.ext.num*) le numéro du sommet origine (resp. extrémité) de γ.



Dans une représentation graphique on utilisera souvent le schéma ci-dessous pour indiquer que l'arc dessiné relie le sommet de numéro `i` au sommet de numéro `j`.

Par commodité de notation, on désignera souvent par `(x,y)` l'arc γ tel que γ`.ori=x` et γ`.ext=y`, ou par `(i,j)` l'arc γ tel que γ`.ori.num=i` et γ`.ext.num=j`.

3.2 Trois exemples de représentation concrète

Ces exemples utilisent les structures de données supposées connues `TABLEAU[G], TABLEAU2[G], LISTE[G]`. La première décrit les tableaux à une dimension, la deuxième les tableaux à deux dimensions et la troisième les listes chaînées. L'identificateur `G` désigne le type des éléments contenus dans ces structures (type générique formel).

3.2.1 Représentation par tableaux

```
ENT GRAPHE::nmax ;
 -- numéro maximum de sommet. On représente les graphes dont les sommets peuvent être num
TABLEAU[BOOL] GRAPHE::vs ;
   -- G.vs[i]  indique l'existence du sommet de numéro i, 1 ≤ i ≤ G.nmax
TABLEAU2[BOOL] GRAPHE::m ;
   -- G.m[i,j]  indique l'existence de l'arc (i,j), 1 ≤ i, j ≤ G.nmax
```

On a toujours l'implication ∀i ∀j : G.m[i,j] ⇒ G.vs[i] ∧ G.vs[j] (un arc ne peut relier que des sommets existants).

Dans cette représentation de $G = (X,\Gamma)$, les ensembles X et Γ sont représentés de la manière suivante :

$x \in X \Leftrightarrow$ `G.vs[x.num]`

$(x,y) \in \Gamma \Leftrightarrow$ `G.m[x.num,y.num]`

Exemple : Graphe G1 `nmax = 10` ;

`vs`

1	2	3	4	5	6	7	8	9	10
v	v	v	v	v	v	v	v	v	f

m

	1	2	3	4	5	6	7	8	9	10
1		v	v							
2					v					
3			v	v		v				
4		v			v					
5						v	v	v		
6			v							
7										
8							v			
9										
10										

les cases vides ont des valeurs *faux*

3.2.2 Représentation par tableaux et listes

Les attributs *nmax* et *vs* sont définis comme précédemment.

TABLEAU[LISTE[SOMMET]] GRAPHE::ts ;
-- *G.ts[i]* désigne la liste des successeurs du sommet de numéro *i*, $1 \leq i \leq G.nmax$.

On a toujours : si *G.vs[i]* est faux, alors *G.ts[i]* est nécessairement vide (en fait, cette liste n'est même pas définie). Dans cette représentation de $G = (X,\Gamma)$, les ensembles X et Γ sont représentés de la manière suivante :

$x \in X \Leftrightarrow$ *G.vs[x.num]*

$(x,y) \in \Gamma \Leftrightarrow$ *G.ts[x.num].contient(y)*

Exemple : Graphe G1 *nmax = 10* ;

vs

1	2	3	4	5	6	7	8	9	10
v	v	v	v	v	v	v	v	v	f

3.2.3 Représentation décentralisée par listes

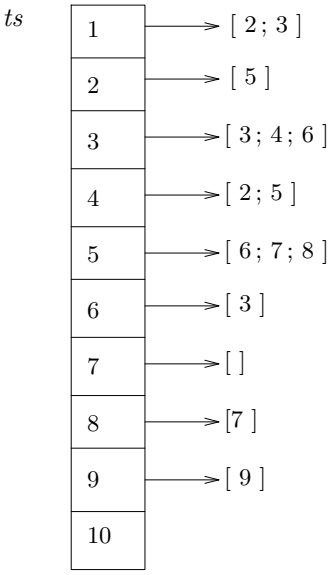
Le type *SOMMET* est enrichi : chaque sommet a maintenant comme attribut la liste de ses successeurs. Le graphe se réduit alors à la liste de ses sommets.

ENT SOMMET::num ; -- comme précédemment
LISTE[SOMMET] SOMMET::ls ;
-- *x.ls* est la liste des sommets successeurs de *x*.

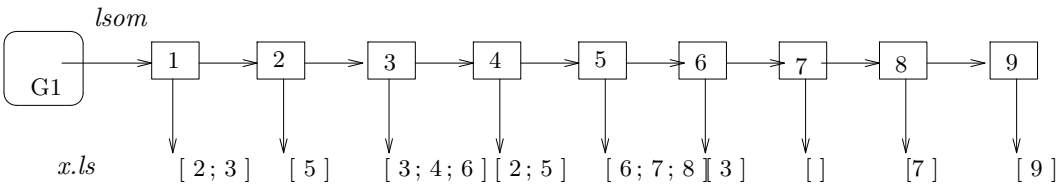
LISTE[SOMMET] GRAPHE::lsom ;
-- *G.lsom* est la liste des sommets du graphe *G*.

Dans cette représentation du graphe $G = (X,\Gamma)$, les ensembles X et Γ sont représentés de la manière suivante :

$x \in X \Leftrightarrow$ *G.lsom.contient(x)*



$(x,y) \in \Gamma \Leftrightarrow$ *G.lsom.contient(x) et G.lsom.contient(y) et x.ls.contient(y)*



Exemple : Graphe G1

3.3 Le type abstrait Graphe : les méthodes

La plupart des algorithmes de résolution de problèmes modélisés par des graphes peuvent être décrits à l'aide de traitements plus ou moins élémentaires sur le graphe modélisant le problème. Cette description –enchaînement contrôlé des actions– est évidemment, à un certain niveau, intrinsèque, c'est à dire indépendante de la représentation choisie pour le graphe.

Par exemple, l'algorithme de saisie ci-après construit un graphe à partir de données lues dans un fichier. Le format de présentation des données est le suivant :

- Séquence des arcs, chaque arc étant donné comme un couple d'entiers (numéro d'origine, numéro d'extrémité), terminée par (0,0);
- séquence des numéros des sommets isolés, terminée par 0.

Exprimé en faisant abstraction d'une représentation concrète particulière, on obtient le texte:

```
GRAPHE construction c'est
    local ENT i, j ;
```

```
        GRAPHE G ;
  début
  --    initialisation au graphe vide
  G. creer ;

  --   lecture de la liste des arcs
  depuis i ← lire ; j ← lire
  jusqu a i=0 et j=0
  faire
     G. ajoutarc(i, j) ;
     i ← lire ; j ← lire
  fait ;

  --   lecture de la liste des sommets isolés
  depuis i ← lire
  jusqua i=0
  faire
     G. ajout_sommet(i) ;
     i ← lire
  fait ;
  Result ← G
    --   Result est une variable exprimant le résultat d'une fonction
fin
```

L'algorithme, présenté sous cette forme, peut s'appliquer à n'importe quelle représentation interne du graphe : il suffit, dans chaque cas, de "traduire" convenablement les opérations indiqués *en italique*.

Dans ce texte, le type `GRAPHE` n'est pas défini concrètement : on l'appelle **type abstrait**. Cela signifie, qu'à ce niveau d'expression, on n'a pas besoin d'en savoir plus sur les structures de données qui seront effectivement utilisées pour représenter concrètement l'objet `G`.

Les traitements élémentaires *en italique* ont été dénommés. On les appelle des *méthodes* du type abstrait `GRAPHE`, et tout ce que l'on a besoin de connaître est leur sémantique. Ainsi, on aura défini une fois pour toutes que :

`creer` : procédure qui initialise le graphe courant au graphe vide ($\emptyset$,$\emptyset$).

`ajoutarc(ENT i; ENT j)` : ajout au graphe courant de l'arc (`i, j`) (et au besoin des sommets de numéro `i` et `j`)

`ajout_sommet(ENT i)` : ajout au graphe courant du sommet de numéro `i`

La liste complète des accès, avec leur sémantique, est donnée en Annexe.

On peut poursuivre cette démarche. En effet, les structures de données utilisés dans les représentations "concrètes" des graphes peuvent être elles aussi considérées comme des types de données abstraits (c'est le cas par exemple du type générique `LISTE[T]`). Ces types de données abstraits peuvent à leur tour être manipulés à travers leurs méthodes, et mis en oeuvre selon diverses représentations concrètes, etc. Dans le cas du type `LISTE[T]`, par exemple, la méthode

`BOOL contient(T x)` est telle que ℓ`.contient(x)` renvoie vrai si et seulement si la liste ℓ contient un élément de type `T`, de valeur `x`.

Nous ne détaillerons pas ici ces représentations, qui sont étudiées dans les cours de techniques de programmation ou structures de données. Il faut néanmoins mettre l'accent sur le fait essentiel que la démarche est absolument identique dans le cas des graphes, des listes, etc... : séparer la partie intrinsèque d'un algorithme des détails de mise en oeuvre, ou d'implémentation. On peut toujours définir de nouveaux niveaux d'abstraction, à condition de bien préciser les types abstraits manipulés et leurs méthodes. Remarquons enfin que les types prédéfinis, même élémentaires (**réel, entier, caractère, booléen**) utilisés dans les langages de haut niveau sont eux aussi des types abstraits par rapport à la machine, leur implémentation concrète en mémoire dépendant du matériel utilisé ; les méthodes sont alors, dans un langage donné, les opérations prédéfinies du langage telles que : lecture/écriture, opérations arithmétiques ou logiques, comparaisons, etc...

Cette démarche de programmation par abstractions successives est réalisée, notamment, dans les langages dits de "programmation par objets", appelés encore, improprement, "langages orientés objets". Citons, parmi les plus connus, EIFFEL, JAVA, C++, dans une moindre mesure TURBO-PASCAL (à partir de la version 6) ou encore, plus anciens, SIMULA et SMALLTALK. Dans ces langages, les types sont décrits dans des *classes*, contenant tout à la fois les *valeurs* des objets du type (appelés *attributs*) et les *méthodes* permettant de manipuler ces valeurs (en lecture ou en écriture). Nous renvoyons, pour plus de détail, aux cours de programmation s'appuyant sur la démarche par objets. Dans la suite de ce polycopié, les notations utilisées dans la description des algorithmes seront proches de celles utilisées dans de tels langages, sans pour autant être liée à l'un d'entre eux en particulier. Le `.` est l'opérateur d'application de méthode ou d'attribut à l'objet désigné. Par exemple, `G.ajout_arc(i,j)` est une *instruction*, signifiant que l'on applique l'opération `ajout_arc`, avec les deux paramètres effectifs `i` et `j` à l'objet de type `GRAPHE` désigné par `G`. De même, `G.valid_arc(i,j)` est une *expression* dont l'évaluation renvoie une valeur booléenne, valant `vrai` si et seulement si l'objet de type `GRAPHE` désigné par `G` possède l'arc `(i, j)`.

3.4 Notations

L'approche objet interdit en général d'accéder *directement* aux attributs d'un objet distant, que ce soit en écriture (c'est-à-dire pour *affecter* directement sa valeur) ou même en lecture (pour obtenir sa valeur). Dans ce cas, des écritures telles que

`x.att := valeur` (affectation "à distance", c'est-à-dire via `x`, de l'attribut `att`), ou même

`v:=x.att` (lecture "à distance", c'est-à-dire via `x` de l'attribut `att`)

sont souvent interdites; de telles opérations ne peuvent en effet être réalisées que par des méthodes (procédures ou fonctions) équipant le type de l'objet désigné par `x`.

Par exemple, avec la déclaration

```
REEL SOMMET::lambda ;
```

la mise à jour de l'attribut `lambda` d'un objet de type `SOMMET` désigné par `x` ne pourra se faire que si le type `SOMMET` possède une méthode

affecter_lambda(REEL v)

post *lambda=v*

qui effectue, de manière interne, l'affectation *lambda:=v*. L'instruction externe réalisant une telle mise à jour s'écrira alors :

x.affecter_lambda(4.18)

Toutefois, afin d'alléger l'écriture des algorithmes et d'en faciliter la lecture, nous utiliserons le symbole ← ayant la signification suivante:

x.lambda ←*4.18* suppose que le type *SOMMET* est équipé d'une méthode réalisant cette affectation. L'écriture ci-dessus est alors équivalente à l'instruction d'appel *x.affecter_lambda(4.18)*. Dans un même souci de simplification, une expression telle *x.lambda* délivre la valeur de l'attribut *lambda* de l'objet désigné par *x*, en supposant que le type de *x* est équipé d'une méthode permettant cette lecture (certains langages, comme Eiffel, permettent de telles lectures directes, à condition que l'attribut soit "exportable").

Enfin, toujours pour alléger le formalisme, si *a* et *b* désignent deux objets de même type, l'écriture *a* ←*b* dénote l'instruction de copie de l'objet désigné par *b* dans l'objet désigné par *a* (affectation de valeurs).

Finalement, dans les parties "mathématiques" (propriétés, démonstrations, etc.) nous utiliserons des notations fonctionnelles plus classiques telles que $\lambda(x)$ ou $pred(x)$ (au lieu des notations "algorithmiques" *x.lambda* ou *x.pred*). D'une manière générale, *x* désignant un objet d'un type $\mathcal{C}$, et *att* un attribut de ce type, les deux notations $att(x)$ et *x.att* seront considérées comme ayant la même signification.

Chapitre 4

Relations et opérateurs entre graphes

Dans ce qui suit, nous désignerons par $\mathcal{G}(X)$ l'ensemble des graphes ayant X comme ensemble de sommets.

Sur $\mathcal{G}(X)$, nous pouvons considérer un certain nombre de relations ou opérations; dans ce chapitre, nous en étudions quelques unes. On note :

$$G_1 = (X,\Gamma_1), G_2 = (X,\Gamma_2)$$

deux éléments de $\mathcal{G}(X)$.

4.1 Relation d'ordre

La relation
$G_1 \preceq G_2 \Longleftrightarrow \Gamma_1 \subset \Gamma_2$ qui se lit : "G_1 est un graphe partiel de G_2"
est une relation d'ordre partiel sur $\mathcal{G}(X)$. Pour cette relation,

$\mathcal{G}(X)$ a un *minimum* : $(X,\emptyset)$
maximum : (X,X^2)

Toute partie de $\mathcal{G}(X)$ possède d'ailleurs *au moins* un minorant et un majorant, comme c'est le cas de toute partie finie d'un ensemble ordonné.

4.2 Union

$$G_1 \cup G_2 = (X,\Gamma_1 \cup \Gamma_2)$$

Cet opérateur possède toutes les propriétés de l'union ensembliste.

4.3 Composition

4.3.1 Définition

$G_3 = G_1 \circ G_2 = (X,\Gamma_3)$

est défini par :

$\Gamma_3 = \{(x,y) \mid \exists z \in X : (x,z) \in \Gamma_1 \wedge (z,y) \in \Gamma_2\}$

L'opérateur $\circ$ n'est pas commutatif. Toutefois, il est associatif et possède un élément neutre, à savoir le graphe "diagonal" :

$D = (X,\Delta)$ où $\Delta = \{(x,x) \mid x \in X\}$

En effet, $\forall G \in \mathcal{G}(X) : G \circ D = D \circ G = G$.
Par abus de notation, on écrira parfois
$\Gamma_3 = \Gamma_1 \circ \Gamma_2$ lorsque $G_3 = G_1 \circ G_2$

Proposition 4.1 $\circ$ *est distributif par rapport à* $\cup$
Autrement dit :

$\forall G_1,G_2,G_3 \in \mathcal{G}(X)$:

$$(G_1 \cup G_2) \circ G_3 = (G_1 \circ G_3) \cup (G_2 \circ G_3)$$
$$G_1 \circ (G_2 \cup G_3) = (G_1 \circ G_2) \cup (G_1 \circ G_3)$$

(Démonstration laissée au lecteur).

Par contre, $\cup$ n'est pas distributif par rapport à $\circ$:

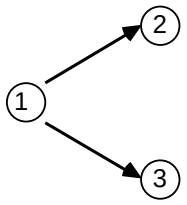
$(G_1 \circ G_2) \cup G_3 \neq (G_1 \cup G_3) \circ (G_2 \cup G_3)$

contre-exemple

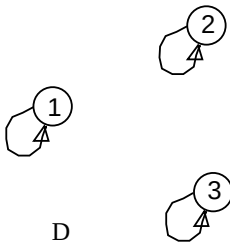
$G_1 = G_2$
$G_3 = D$

$G_1 \circ G_2 = (X,\emptyset)$
$(G_1 \circ G_2) \cup G_3 = D$

D'autre part :



G1=G2



D

$(G_1 \cup D) \circ (G_2 \cup D) = G_1 \cup D$

Proposition 4.2 *L'opérateur* $\circ$ *est compatible à gauche et à droite avec la relation d'ordre définie en 3.1*

Démonstration : Cet énoncé signifie que

$$\forall G_1,G_2,G_3 \in \mathcal{G}(X) \ : \ G_1 \preceq G_2 \ \Rightarrow \ (G_1 \circ G_3 \ \preceq \ G_2 \circ G_3) \ \wedge \ (G_3 \circ G_1 \ \preceq \ G_3 \circ G_2)$$

Si $G_i = (X,\Gamma_i)$ $(i = 1,2,3)$, on a :

$$\begin{aligned} (x,y) \in \Gamma_1 \circ \Gamma_3 &\Rightarrow \exists z \in X \ : (x,z) \in \Gamma_1 \wedge (z,y) \in \Gamma_3 \\ &\Rightarrow \exists z \in X \ : (x,z) \in \Gamma_2 \wedge (z,y) \in \Gamma_3 \\ &\Rightarrow (x,y) \in \Gamma_2 \circ \Gamma_3 \end{aligned}$$

ce qui montre que $G_1 \circ G_3 \preceq G_2 \circ G_3$.

La deuxième inégalité se montre de manière analogue.□

4.3.2 Algorithme

Nous donnons ci-dessous un algorithme de construction de $G_3 = G_1 \circ G_2$, en supposant pour simplifier que $X = [1..n]$

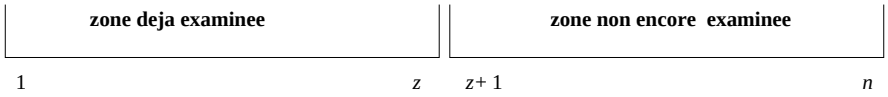
4.3.2.1 Analyse

Pour tout couple $(x,y) \in X^2$, il faut déterminer si (x,y) est ou non un arc de G_3. Par définition,

$(x,y) \in \Gamma_3 \Longleftrightarrow \exists z \ (1 \leq z \leq n) \wedge ((x,z) \in \Gamma_1) \wedge ((z,y) \in \Gamma_2)$

Notons $\mathcal{P}(z)$ la proposition : $((x,z) \in \Gamma_1) \wedge ((z,y) \in \Gamma_2)$. Il s'agit d'évaluer le prédicat :

$\exists t \ 1 \leq t \leq n \ : \ P(t)$. Cette évaluation est mise en œuvre par un calcul itératif. La figure ci-dessous en détermine l'invariant :



c'est-à-dire, plus formellement :

$I \equiv (0 \leq z \leq n) \wedge (b = \exists t \ 1 \leq t \leq z \ : \ P(t))$

Le calcul sera terminé lorsque :

$(z = n) \vee b$

et, lorsque ceci se produit :

- si b alors $(\exists t \ 1 \leq t \leq z \ : \ P(t)) \wedge (z \leq n)$, c'est-à-dire $(\exists t \ 1 \leq t \leq n \ : \ P(t))$ (*succes*)
- si $\neg b$ alors $(z = n) \wedge \neg(\exists t \ 1 \leq t \leq n \ : \ P(t))$ (* échec *)

Le résultat est donc égal à b

Si le calcul n'est pas terminé, on progresse en mettant à jour les variables z et b de la manière suivante :

$z \leftarrow z + 1 \ ; \ b \leftarrow P(z)$

Les valeurs initiales doivent satisfaire l'invariant ; par exemple, on peut prendre :

$z \leftarrow 0$; $b \leftarrow false$

De cette analyse découle le texte de l'algorithme.

4.3.2.2 Texte de l'algorithme

Il est donné page 32.

ALGORITHME DE CONSTRUCTION DE G1∘G2

```
GRAPHE composer(GRAPHE G1, G2)  c'est
--   pré : G1 et G2 ont le même ensemble de sommets :
--         G1.ens_som=G2.ens_som={1,...,n}
    local SOMMET x,y,z; BOOL b;
 début
   Result.creer;
--   Result est une variable implicite de type GRAPHE, exprimant
le résultat de la fonction
    pour tout x  de G1.ens_som
      pour tout y  de G1.ens_som
        depuis z ← 0 ; b ← false
        jusqua z ≥ n  ou b
        faire
          z ← z+1;
          b ← G1.validarc(x,z)  et G2.validarc(z,y)
        fait;
        si b  alors Result.ajoutarc(x,y)  fsi
      fpour
    fpour
 fin
```

4.3.3 Complexité

La boucle interne **depuis** ... **jqa** comporte au plus n pas , et elle est exécutée exactement n^2 fois. La complexité **maximum** est donc :

$2n^3$ accès *validarc*
n^3 opérations **et**
n^2 accès *ajoutarc*

On dit que c'est un algorithme en $O(n^3)$, en termes d'accès *validarc*.

Remarques

1) La complexité de mise en œuvre de chacun des accès dépend évidemment de la représentation choisie pour chacun des 3 graphes `G1, G2, Result`.

2) Si les graphes sont tous les trois représentés par leur matrice booléenne, alors il est facile de voir que

$M_3 = M_1 \cdot M_2$ où le $\cdot$ désigne le produit matriciel booléen de deux matrices :

$$\forall 1 \leq i,j \leq n \; : m3_{ij} = \bigvee_{k=1}^{n} (m1_{ik} \wedge m2_{kj})$$

4.4 Puissances d'un graphe

On définit par récurrence sur $p \in \mathbb{N}$ le graphe $G^{[p]} = (X,\Gamma^{[p]})$ de la manière suivante :

Définition 4.3

$$G^{[0]} = D$$

$$\forall p \geq 1 \; : G^{[p]} = G^{[p-1]} \circ G = G \circ G^{[p-1]}$$

propriétés : $G^{[k]} \circ G^{[l]} = G^{[k+l]}$

$(G^{[k]})^{[l]} = G^{[kl]}$

(identiques aux puissances des nombres réels)

Nous allons interpréter les puissances d'un graphe.

Proposition 4.4 *Soit $G^{[p]} = (X,\Gamma^{[p]})$ la puissance $p^{ème}$ d'un graphe $G = (X,\Gamma)$. Alors, $(x,y) \in \Gamma^{[p]}$* **si et seulement si** *il existe, dans G, un chemin de longueur p, allant de x à y (ce chemin est un circuit si $x = y$).*

Démonstration Par récurrence sur p :

- Pour p= 1, le résultat est évident : $G^{[1]} = (X,\Gamma)$ et les arcs sont les chemins de longueur 1.
- Supposons démontré jusqu'à p-1.

$$\begin{aligned} & (x,y) \in \Gamma^{[p]} = \Gamma^{[p-1]} \circ \Gamma \\ \iff \; & \exists z \in X \; : (x,z) \in \Gamma^{[p-1]} \wedge (z,y) \in \Gamma \end{aligned}$$

$$\iff \exists z \in X : \exists \mathbf{u} = [x * z],\ |\ \mathbf{u}\ | = p - 1 \wedge \exists \mathbf{v} = [zy],\ |\ \mathbf{v}\ | = 1$$
$$\iff \exists \mathbf{w} = \mathbf{u}.\mathbf{v} = [x * y],\ |\ \mathbf{w}\ | = |\ \mathbf{u}\ | + |\ \mathbf{v}\ | = p$$

ce qui démontre le résultat pour p □

4.5 Graphe transposé

Définition 4.5 *Le graphe transposé de* $G = (X,\Gamma)$ *est :*

$G^t = (X,\Gamma^t\ )$ avec

$(x,y) \in \Gamma^t \iff (y,x) \in \Gamma$

C'est donc le graphe déduit de G par inversion du sens de tous les arcs. On utilise parfois la notation Γ^{-1}, lorsqu'on veut exprimer le fait que l'application multivoque associée à G^t est l'inverse de l'application multivoque Γ liée à G. Remarquons toutefois que les relations $\Gamma \circ \Gamma^{-1} = \Delta$ ou $\Gamma^{-1} \circ \Gamma = \Delta$ ne sont en général pas vérifiées. Ainsi, Γ^{-1} ne doit pas être considéré comme la "puissance moins un" du graphe, mais bien comme une notation désignant l'ensemble des arcs du graphe transposé.

4.6 Sous-graphe

Dans ce paragraphe, les graphes considérés ne sont pas tous construits sur le même ensemble de sommets. En effet, si nous avons un graphe $G = (X,\Gamma)$, certaines opérations sur G peuvent amener à supprimer des sommets. Or, si un sommet est ôté du graphe, il va de soi que les éventuels arcs adjacents à X doivent aussi être supprimés : la suppression d'un objet entraîne la suppression de toutes les relations relatives à cet objet ! Cette remarque préliminaire conduit à la définition suivante :

Définition 4.6 *Soit* $G = (X,\Gamma) \in \mathcal{G}(X)$ *et* $A \subset X$.
Le sous-graphe engendré par A *est défini par :*

$G_A = (A,\Gamma_A) \in \mathcal{G}(A)$,avec

$\Gamma_A = \{(x,y) \mid x \in A, y \in A, (x,y) \in \Gamma\}$

On peut donner un algorithme de construction du sous-graphe engendré par A :

```
sous_graphe(ENS[SOMMET] A)  c'est
  local SOMMET z
```

```
début
   pourtout z de ens_som faire
      si z ∉ A alors oter_sommet(z) fsi
   fpourtout
fin
```

Chapitre 5

Fermeture transitive

Le problème étudié dans ce chapitre est celui de l'existence des chemins entre tous les couples de sommets. Plus précisément :

Étant donné un graphe $G = (X,\Gamma)$, construire le graphe $G^+ = (X,\Gamma^+)$ tel que $(x,y) \in \Gamma^+ \Leftrightarrow$ *il existe un chemin de* G *allant de* x *à* y. G^+ s'appelle la *fermeture transitive* de G.

Nous allons voir successivement les bases théoriques, puis deux algorithmes permettant de résoudre le problème, et enfin des exemples d'application.

5.1 Retour sur la transitivité

Un graphe est *transitif* si la relation binaire qui lui est associée est transitive. En d'autres termes :

$\forall x,y,z \in X \ : (x,y) \in \Gamma \wedge (y,z) \in \Gamma \Longrightarrow (x,z) \in \Gamma$.

Voici une propriété caractéristique des graphes transitifs:

Proposition 5.1 *Un graphe* G *est transitif si et seulement si tout chemin de longueur deux est sous-tendu par un arc, c'est-à-dire :* $\Gamma^{[2]} \subset \Gamma$

Démonstration

1. Supposons G transitif.

$$\begin{array}{lcl} G \text{ transitif} & \Rightarrow & \forall x \forall y \forall z \left((x,y) \in \Gamma \wedge (y,z) \in \Gamma \Rightarrow (x,z) \in \Gamma\right) \\ & \Rightarrow & \forall x \forall z \left(\exists y \left((x,y) \in \Gamma \wedge (y,z) \in \Gamma\right) \Rightarrow (x,z) \in \Gamma\right) \\ & \Rightarrow & \forall x \forall z \left((x,z) \in \Gamma^{[2]} \Rightarrow (x,z) \in \Gamma\right) \\ & \Rightarrow & \Gamma^{[2]} \subseteq \Gamma \end{array}$$

2. Réciproquement, supposons $\Gamma^{[2]} \subseteq \Gamma$.

$$\begin{array}{lcl} (x,z) \in \Gamma \wedge (z,y) \in \Gamma & \Rightarrow & (x,y) \in \Gamma^{[2]} \\ & \Rightarrow & (x,y) \in \Gamma \\ & \Rightarrow & G \text{ est transitif} \end{array}$$

□

Cette propriété admet un corollaire évident:

Corollaire 5.2 *Un graphe G est transitif si et seulement si tout chemin est sous-tendu par un arc, c'est-à-dire :* $\forall k \geq 2 \ : \ \Gamma^{[k]} \subset \Gamma$

Démonstration Par récurrence sur k :

- vrai pour $k = 2$, d'après la proposition précédente.
- Hypothèse de récurrence : $\Gamma^{[k-1]} \subseteq \Gamma$. Cela signifie: $G^{[k-1]} \preceq G$. On a alors :

$$\begin{aligned} G^{[k]} &= G^{[k-1]} \circ G \\ &\preceq G \circ G \text{(hypothèse de récurrence et proposition 4.2)} \\ &\preceq G^{[2]} \\ &\preceq G \text{ (proposition 5.1)} \end{aligned}$$

□

L'interprétation "concrète" de ce corollaire est la suivante : dans un graphe transitif, la notion de chemin se confond avec la notion d'arc.

Nous allons maintenant aborder les caractérisations et les algorithmes de calcul de la fermeture transitive. Le terme de *fermeture transitive* provient du fait que, d'une part, G^+ est transitif (par construction!), et d'autre part, G^+ est le *plus petit graphe transitif supérieur ou égal à* G (au sens de la relation $\preceq$). Plus précisément :

- G^+ est transitif,
- $G \preceq G^+$
- Si H est transitif et $G \preceq H$, alors $G^+ \preceq H$

Intuitivement, on obtient G^+ à partir de G en rajoutant "juste ce qu'il faut" d'arcs pour rendre le graphe transitif. La démonstration de ces propriétés ne présente aucune difficulté. Elle est laissée en exercice au lecteur.

5.2 Caractérisation du graphe G^+

Proposition 5.3 $G^+ = G \cup G^{[2]} \cup \ldots \cup G^{[k]} \cup \ldots$

Démonstration Posons $G^+ = (X,\Gamma^+)$ et, $\forall k \geq 1$, $G^{[k]} = (X,\Gamma^{[k]})$. Par définition,

$$\begin{aligned} (x,y) \in \Gamma^+ &\Leftrightarrow \exists \mathbf{u} = [x \star y] \text{ dans } G \\ &\Leftrightarrow (x,y) \in \Gamma^{[k]}, \text{où } k = |\mathbf{u}| \\ &\Leftrightarrow (x,y) \in \Gamma \cup \Gamma^{[2]} \cup \ldots \cup \Gamma^{[k]} \cup \ldots \end{aligned}$$

□

La proposition suivante montre que le problème se réduit à l'existence des chemins élémentaires.

Proposition 5.4 *Les deux propositions suivantes sont équivalentes :*
i) *Dans le graphe G, il existe un chemin de x à y,*
ii) *Dans le graphe G, il existe un chemin* élémentaire *de x à y*

Démonstration ii) ⇒ i). Évident puisque tout chemin élémentaire est un chemin.

i) ⇒ ii). Supposons qu'il existe un chemin $\mathbf{u} = [x * y]$. S'il est élémentaire, la démonstration est terminée; sinon, la suite de sommets le définissant comporte au moins *une* répétition :

$\mathbf{u} = [x * z * z * y]$ (éventuellement, $z = x$ ou $z = y$)

On a donc :

$\mathbf{u = u_1 \cdot u_2 \cdot u_3}$

avec $\mathbf{u_1} = [x * z]$,$\mathbf{u_2} = [z * z]$,$\mathbf{u_3} = [z * y]$ et **u'** $= \mathbf{u_1 \cdot u_3}$ est un chemin de x à y, obtenu à partir de **u** en "court-circuitant" le circuit $[z * z]$. En itérant ce raisonnement tant qu'il y a des répétitions (circuits contenus dans le chemin), on obtient nécessairement un chemin élémentaire reliant x à y. □

Des deux propositions précédentes résulte la caractérisation suivante :

Corollaire 5.5 *Soit* $G = (X,\Gamma)$ *et* $n = \mid X \mid$. *On a :*

$$G^+ = G \cup G^{[2]} \cup \ldots \cup G^{[n]}$$

Démonstration Le résultat découle des deux propositions précédentes et du fait que *tout chemin élémentaire est de longueur* $\leq n$ (et même $\leq n-1$ si ce n'est pas un circuit). □

On peut encore être plus précis :

Proposition 5.6 *Soit k la longueur maximum des chemins élémentaires de G. Alors :*
i) $k \leq n$,
ii) $\bigcup_{j=1}^{k} G^{[j]} = \bigcup_{j=1}^{k+1} G^{[j]}$
iii) $G^+ = G \cup G^{[2]} \cup \ldots \cup G^{[k]}$

Démonstration i) : par définition des chemins élémentaires (longueur $\leq n$)

ii) Si $(x,y) \in \bigcup_{j=1}^{k+1} G^{[j]}$ alors il existe dans G un chemin **u** de longueur $\leq k+1$, allant de x à y. D'après la proposition 5.4, il existe dans G un chemin élémentaire allant de x à y; autrement dit,

il existe un entier $\ell \leq k$ tel que $(x,y) \in G^{[\ell]}$. Comme $G^{[\ell]} \subseteq \bigcup_{j=1}^{k} G^{[j]}$ on a bien $\bigcup_{j=1}^{k} G^{[j]} = \bigcup_{j=1}^{k+1} G^{[j]}$.

iii) Immédiat D'après le point précédent et le corollaire 5.5 □

5.3 Algorithme des puissances

L'algorithme des puissances calcule G^+ en se basant sur la proposition 5.6. Il est construit en maintenant invariante la proposition ci-dessous :

$$(G^+ = G \cup ... \cup G^{[k]}) \wedge (H = G \cup ... \cup G^{[k-1]}) \wedge (1 \leq k \leq n)$$

Le calcul sera donc terminé lorsque $H = G^+$
Si le calcul n'est pas terminé, la progression consiste à mettre à jour les variables H,G^+:

$$\begin{array}{rcl} H & \leftarrow & G^+; \\ G^+ & \leftarrow & G \cup (G^+ \circ G) \end{array}$$

En effet :

si $G^+ = \bigcup_{j=1}^{k} G^{[j]}$ **alors**

$G^+ \circ G = \bigcup_{j=2}^{k+1} G^{[j]}$ (distributivité de $\circ$ par rapport à $\cup$) d'où

$G \cup (G^+ \circ G) = \bigcup_{j=1}^{k+1} G^{[j]}$

Les valeurs initiales doivent vérifier l'invariant et être faciles à évaluer en fonction des données :

$G^+ \leftarrow G$, H arbitraire (mais $\neq G^+$, par exemple : $H \leftarrow (\emptyset,\emptyset)$).

Le texte de l'algorithme en découle.

5.3.1 Texte de l'algorithme

Il est donné page 41

FERMETURE TRANSITIVE : PUISSANCES

```
GRAPHE fermeture-transitive-puissance(GRAPHE G) c'est
  local GRAPHE H
  début
    depuis Result ← G ; H.creer
    jusqua H=Result
    faire
      H ← Result ;
      Result ← G ∪(G ∘ Result)
    fait
  fin
```

5.3.2 Complexité

Au maximum, n pas d'itération sont effectués. A chaque pas, sont effectués :

- 2 affectations de graphe, en $O(n^2)$ accès élémentaires,
- 1 comparaison de 2 graphes, en $O(n^2)$ accès élémentaires,
- 1 opération $\circ$, en $O(n^3)$ accès élémentaires (chapitre 4, section4.3.3)
- 1 opération $\cup$, en $O(n^2)$ accès élémentaires.

C'est donc un algorithme en $O(n^4)$ accès élémentaires.

De plus, l'espace utilisé est de 3 graphes.

5.4 Algorithme de ROY-WARSHALL

Par rapport à l' algorithme précédent, basé sur les calculs de puissances, celui-ci présente les avantages suivants :

- sa complexité est en $O(n^3)$ accès,
- l'espace utilisé n'est que de 1 graphe,
- il offre la possibilité de calculer des routages, c'est à dire d'obtenir le tracé de chemins élémentaires.

5.4.1 Principe de l'algorithme

L'algorithme est basé sur l'opération élémentaire Θ suivante :

Définition 5.7 *Soit $x \in X$. Si $G = (X,\Gamma) \in \mathcal{G}(X)$, alors $\Theta_x(G) \in \mathcal{G}(X)$ a pour ensemble d'arcs :*

$$\Theta_x(\Gamma) = \Gamma \cup \{(y,z) \mid (y,x) \in \Gamma \wedge (x,z) \in \Gamma\}$$

Autrement dit, Θ_x enrichit le graphe en lui ajoutant les arcs reliant les prédécesseurs de x aux successeurs de x.
L'algorithme est basé sur une nouvelle caractérisation de la fermeture transitive :

$$G^+ = \prod_{x \in X} \Theta_x(G)$$

où $\prod$ désigne le produit (composition) des opérations Θ_x, c'est à dire, si $X = \{1,2,\ldots,n\}$ est une numérotation des sommets :

$$G^+ = \Theta_n(\Theta_{n-1}(\ldots(\Theta_1(G))\ldots))$$

Lemme 5.8 *Soit V_i l'ensemble des couples $(y,z) \in X^2$ tels qu'il existe dans G un chemin élémentaire de longueur 1 ou dont tous les sommets intermédiaires sont de numéro inférieur ou égal à i. On a alors :*

$$\forall 1 \leq i \leq n \ : \Theta_i \cdot \Theta_{i-1} \cdot \ldots \cdot \Theta_1(\Gamma) = V_i$$

Démonstration Par récurrence sur i :

- Pour i= 1,

$$\Theta_1(\Gamma) = \Gamma \cup \{(y,z) \mid (y,1) \in \Gamma \wedge (1,z) \in \Gamma\}$$

Γ est l'ensemble des couples reliés par un chemin de longueur 1, $\{(y,z) \mid (y,1) \in \Gamma \wedge (1,z) \in \Gamma\}$ est l'ensemble des couples reliés par un chemin élémentaire de longueur 2 de sommet intermédiaire numéro 1. La propriété est donc vérifiée.

- Supposons la vraie pour i-1 et montrons qu'elle est vraie pour i.

Hypothèse : $\Theta_{i-1} \cdot \ldots \cdot \Theta_1(\Gamma) = V_{i-1}$

Il faut montrer : $\Theta_i \cdot \Theta_{i-1} \cdot \ldots \cdot \Theta_1(\Gamma) = V_i$.

D'après l'hypothèse de récurrence, cela revient à montrer que :$\Theta_i(V_{i-1}) = V_i$.

Mais, par définition de Θ_i :

$$\Theta_i(V_{i-1}) = V_{i-1} \cup \{(y,z) \mid (y,i) \in V_{i-1} \wedge (i,z) \in V_{i-1}\}$$

Soit $(y,z) \in \Theta_i(V_{i-1})$. Il y a deux possibilités:

1. Si $(y,z) \in V_{i-1}$ alors $(y,z) \in V_i$ (car $V_{i-1} \subseteq V_i$)

2. Si $(y,i) \in V_{i-1} \wedge (i,z) \in V_{i-1}$ alors il existe un chemin de y à z dont tous les sommets intermédiaires sont de numéro inférieur ou égal à i, donc (proposition 5.4) il existe un chemin *élémentaire* de y à z dont tous les sommets intermédiaires sont de numéro inférieur ou égal à i, c'est-à-dire $(y,z) \in V_i$.

On a donc montré $\Theta_i(V_{i-1}) \subseteq V_i$.

Réciproquement, soit $(y,z) \in V_i$. Il existe donc un chemin élémentaire u de y à z, dont tous les sommets intermédiaires sont de numéro inférieur ou égal à i. Il y a deux cas :

1. u ne passe pas par le sommet i. Dans ce cas, tous les sommets intermédiaires sont de numéro inférieur ou égal à $i-1$ donc $(y,z) \in V_{i-1} \subseteq \Theta_i(V_{i-1})$
2. u passe par le sommet i. Comme il est élémentaire, il se décompose en deux chemins u_1 et u_2, avec $u_1 \in V_{i-1}$ et $u_2 \in V_{i-1}$. Par conséquent, $(y,i) \in V_{i-1}$ et $(i,z) \in V_{i-1}$ d'où $(y,z) \in \Theta_i(V_{i-1})$.

On a donc montré $V_i \subseteq \Theta_i(V_{i-1})$. □

Proposition 5.9 $G^+ = \Theta_n \cdot \Theta_{n-1} \cdot \ldots \cdot \Theta_1(G)$

Démonstration Tout chemin de G est de longueur 1, ou possède des sommets intermédiaires $\in \{x_1, \ldots, x_n\} = X$. Le théorème découle donc du lemme précédent, avec $i = n$. □

5.4.2 Texte de l'algorithme

Il est donné page 44

5.4.3 Complexité

Pour chaque opération Θ, il y a au plus :

- $n + n^2$ accès *validarc*
- n^2 accès *ajoutarc*

soit, globalement :

- au plus $n^2 + n^3$ accès *validarc*
- au plus n^3 accès *ajoutarc*

d'où un algorithme en $O(n^3)$ en termes d'accès élémentaires.

De plus, l'espace mémoire n'est que de 1 graphe (le graphe courant `Result`). En effet, le graphe donné `G` n'est utilisé que pour "initialiser" `Result`, il est donc inutile de le garder en mémoire.

ALGORITHME DE ROY-WARSHALL

```
GRAPHE roy-warshall(GRAPHE G)  c'est
   local SOMMET x,y,z ; ENS[SOMMET] X;
 début

  Result ← G ; X ← G.lst_som ;
   pour tout x  de X
      pour tout y  de X
         si Result.validarc(y,x)  alors
            pour tout z  de X
               si Result.validarc(x,z)
                  alors Result.ajoutarc(y,z)
               fsi
            fpourtout
         fsi
      fpourtout
    fpourtout
 fin
```

5.4.4 Routage

5.4.4.1 Définition

L'idée du routage est la suivante: si l'on calcule la fermeture transitive G^+ d'un graphe G, on est capable de répondre à la question : quels que soient les sommets x et y, existe-t-il dans G un chemin de x à y? Une information de routage doit permettre de compléter cette réponse, en fournissant le *tracé* d'un tel chemin (lorsque celui-ci existe). Il y a plusieurs manières d'obtenir un routage. Nous en décrivons une, basée sur la notion de table. Rappelons qu'une table est une structure de données mettant en oeuvre une correspondance entre un ensemble de *clefs* et un ensemble de *valeurs*. Le type abstrait `TABLE[K,E]` décrit les tables dont l'ensemble des clefs est `K` et l'ensemble des valeurs `E` (mathématiquement, ce sont les applications de `K` dans `E`). Une table est déclarée `TABLE[K,E] T`; elle est alors accédée par les opérations suivantes :

- `T[x]` : délivre la valeur `v` de type `E` correspondant à la clef `x` (de type `K`).
- `T[x]` ←`v` affecte la valeur `v` de type `E` à l'élément correspondant à la clef `x` (de type `K`).

Les objets de type `SOMMET` sont alors munis d'une table, selon la déclaration ci-dessous :

```
TABLE[SOMMET,SOMMET] SOMMET::R
```

avec la signification suivante:

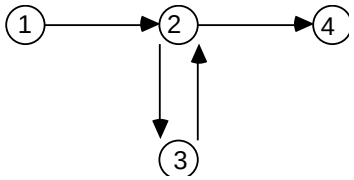
Pour tout sommet y, pour tout sommet z,

$$y.R[z] = \begin{cases} \textbf{nil si } z \notin \Gamma^+(y) \textit{ où } \textbf{nil} \textit{ signifie "pas de successeur"} \\ \\ \text{successeur de } y \text{ sur un chemin de } y \text{ à } z \text{ dans } G \textbf{ si} \\ z \in \Gamma^+(y) \end{cases}$$

La connaissance des sommets $y.R[z]$ permet-elle de déterminer le tracé d'un chemin de y à z, à partir de n'importe quel y? La réponse n'est pas toujours positive ! En effet, l'algorithme de tracé a l'allure suivante (si $y.R[z] \neq$ **nil**) :

```
début
   local SOMMET somcour;
   depuis  ecrire ← y ; somcour ← y.R[z];
   jusqua somcour = z
   faire
      ecrire ← somcour ;
     somcour ← somcour.R[z];
   fait;
   ecrire ← z
fin
```

Mais rien ne prouve que cet algorithme se termine ; par exemple :



avec, pour `R[4]` :

`y`	`y.R[4]`
1	2
2	3
3	2
4	**nil**

A partir des sommets 1, 2 ou 3, l'algorithme boucle, et pourtant les valeurs `R[4]` sont bien conformes à la définition.

Nous allons donc renforcer la définition, afin d'éviter cette difficulté, en imposant à `y.R[z]`, si $z \in \Gamma^+(y)$, d'être le successeur de y sur un chemin *élémentaire* de y à z. Il est alors facile de vérifier que l'algorithme de tracé se termine, puisque les valeurs successives de la variable `somcour` sont distinctes. (Sauf éventuellement $y = z$ pour un circuit élémentaire).

Pour tout sommet y, pour tout sommet z,

$$y.R[z] = \begin{cases} \textbf{nil si } z \notin \Gamma^+(y) \\ \\ \text{successeur de } y \text{ sur un chemin élémentaire de } y \text{ à } z \\ \text{dans } G \textbf{ si } z \in \Gamma^+(y) \end{cases}$$

Dans l'exemple précédent, les valeurs `y.R[4]` deviennent :

y	*y.R[4]*
1	2
2	4
3	2
4	**nil**

et l'algorithme de tracé vers le sommet `4`, exécuté à partir de `y=1`, affiche `1 2 4`.

5.4.4.2 Construction des routages par l'algorithme de Roy-Warshall

$\forall y \in X$, on considère les valeurs initiales de la table `y.R` :

$$\forall z \in X \ : y.R[z] = \begin{cases} \textbf{nil si } (y,z) \notin \Gamma \\ z \textbf{ si } (y,z) \in \Gamma \end{cases}$$

C'est donc l'attribut de routage vers z lorsqu'on se restreint aux chemins de longueur 1. On modifie alors les opérations Θ_x, afin qu'elles mettent à jour les valeurs des tables `R` : dans le cas où Θ_x rajoute un arc entre y et z, alors `y.R[z]` ←`y.R[x]`. En effet, le chemin élémentaire de y à z qui vient d'être repéré passe par x : pour aller de y à z, on doit d'abord se diriger vers x.

D'où le texte de l'algorithme, modifié routage :

ALGORITHME DE ROY-WARSHALL avec ROUTAGE

```
GRAPHE roy-warshall(GRAPHE G)  c'est
   local SOMMET x,y,z ; ENS[SOMMET] X;
 début

  Result ← G ; X ← G.lst_som ;
  --   initialisation des tables de routage
   pour tout y  de X
      pour tout z  de X
           si Result.validarc(y,z)  alors y.R[z] ← z  sinon y.R[z] ← nil  fsi
      fpourtout
   fpourtout ;
  --   opérations Θ avec gestion des tables de routage
   pour tout x  de X
      pour tout y  de X
         si Result.validarc(y,x)  alors
            pour tout z  de X
               si Result.validarc(x,z)
                  alors
                    si  non Result.valid_arc(y,z)
                        alors Result.ajoutarc(y,z);
                                  y.R[z] ← y.R[x]
                    fsi
```

```
            fsi
          fpourtout
        fsi
      fpourtout
    fpourtout
fin
```

5.4.4.3 Exemple d'exécution

A la main, la représentation la mieux adaptée est matricielle. Pour gérer le routage, au lieu d'utiliser la matrice booléenne, nous utilisons la matrice de routage R, telle que

$$\forall y\ \forall z\ : R[y,z] = \begin{cases} y.R[z] \text{ si } (y,z) \in \Gamma \\ 0 \text{ si } (y,z) \notin \Gamma \end{cases}$$

Nous traitons l'exemple suivant (les cases de valeur 0 sont laissées vides):

	1	2	3	4	5	6	7	8	9
1						6			
2									
3		2					7	8	
4	1				5				
5							7		
6				4					
7								8	
8			3						
9	1	2							

Opération θ_1: il s'agit de parcourir la *colonne* numéro 1, et de s'arrêter sur toute ligne de valeur $\neq 0$ (correspondant à `Result.validarc(y,1)`). Pour une telle ligne, on effectue la "recopie" des éléments non nuls de la ligne 1 sur la ligne y, en mettant la valeur $R[y,1]$ en toute case nulle $R[y,z]$ telle que $R[1,z] \neq 0$ (ce qui correspond à:

```
si Result.validarc(1,z)
      alors
         si  non Result.valid_arc(y,1)
                 alors Result.ajoutarc(y,z);
                       y.R[z] ← y.R[1]
         fsi
fsi
```

L'application de θ_1 ne modifie que les lignes 4 et 9, et donne le tableau suivant :

	1	2	3	4	5	6	7	8	9
1						6			
2									
3		2					7	8	
4	1				5	**1**			
5							7		
6				4					
7								8	
8			3						
9	1	2				**1**			

Le détail des différentes étapes serait trop long; notons cependant que θ_2 ne rajoute aucun arc (le sommet 2 est un point de sortie, c'est-à-dire un sommet sans successeur); de même θ_9, car le sommet 9 est un point d'entrée (c'est-à-dire un sommet sans prédécesseur). On donne ci-dessous le résultat final :

	1	2	3	4	5	6	7	8	9
1	6	6	6	6	6	6	6	6	
2									
3		2	8				7	8	
4	1	5	5	1	5	1	5	5	
5		7	7				7	7	
6	4	4	4	4	4	4	4	4	
7		8	8				8	8	
8		3	3				3	3	
9	1	2	1	1	1	1	1	1	

A titre d'exemple, on va chercher le tracé d'un chemin du sommet 1 au sommet 2; on obtient successivement:

R[1,2] = 6; R[6,2] = 4; R[4,2] = 5; R[5,2] = 7; R[7,2] = 8; R[8,2] = 3; R[3,2] = 2

d'où le chemin : [1,6,4,5,7,8,3,2]

5.4.4.4 Preuve

Pour démontrer la validité de cet algorithme, nous établissons les lemmes suivants :

Lemme 5.10 *Si pour un couple (y, z) on a, à l'issue de l'algorithme, $y.R[z] \neq$* **nil***, alors $y.R[z]$ est successeur de y sur le premier chemin allant de y à z rencontré au cours de l'algorithme.*

Démonstration Pour tout y et tout z, la valeur $y.R[z]$ est mise à jour au plus une fois. En effet, si $(y,z) \in \Gamma$, cette affectation a lieu à l'initialisation, et de plus: `Result.validarc(y,z)` est vérifié. Si $(y,z) \notin \Gamma$, la précondition de cette affectation implique :

$\neg$ `Result.validarc(y,z)`, et sa postcondition implique `Result.validarc(y,z)` □

Lemme 5.11 *Si $z \in \Gamma^+(y)$ alors le premier chemin allant de y à z rencontré au cours de l'algorithme est un chemin élémentaire.*

Démonstration Si $(y,z) \in \Gamma$, ce chemin est un arc et donc il est élémentaire. Supposons que tous les chemins repérés jusqu'à l'opération Θ_{i-1} incluse soient élémentaires. Soit alors $\mathbf{u} = [y * x_i] \cdot [x_i * z]$ un chemin repéré par l'opération Θ_i, c'est-à-dire tel que l'affectation `y.R[z]` $\leftarrow$`y.R[`x_i`]` ait lieu ; si $\mathbf{u}$ n'est pas élémentaire, il existe $j < i$ tel que

$$\mathbf{u} = [y * x_j * x_i] \cdot [x_i * x_j * z]$$

et tous les autres sommets intermédiaires sont dans $\{x_1, \ldots, x_{i-1}\}$ et différents de x_j, d'après l'hypothèse de récurrence. Mais cela signifie qu'il existe un chemin

$$\mathbf{u'} = [y * x_j] \cdot [x_j * z] \text{ avec } * \in \{x_1, \ldots, x_{i-1}\}$$

et donc

$$\exists \ell \leq i-1 \ : (y,z) \in V_\ell$$

Au cours du pas d'itération n$^o\ell$ l'affectation `y.R[z]` $\leftarrow$`y.R[`x_ℓ`]` a donc lieu, et cela contredit l'affectation de `y.R[z]` lors du pas n$^o i > \ell$. □

De ces deux lemmes on déduit le résultat :

Théorème 5.12 *A l'issue de l'algorithme,*

$\forall z \in X$: *les valeurs $R[z]$ définissent un routage vers z*

5.5 τ-minimalité

La *τ-minimalité* constitue une notion intéressante du point de vue des possibilités de cheminement. Il s'agit de mettre en évidence un graphe partiel de G, ayant la même fermeture transitive que G – donc les mêmes possibilités de cheminement – mais *minimal* au sens où la suppression d'un arc quelconque de ce graphe partiel supprime au moins une possibilité de cheminement. Il n'y a en général pas unicité de la solution.

Définition 5.13 *La relation binaire sur $\mathcal{G}(X)$:*

$$G \tau G' \Leftrightarrow G^+ = G'^+$$

est une relation d'équivalence, appelée τ-équivalence.
La classe de G sera notée : $\tau(G)$

Définition 5.14 *On appelle graphe τ-minimal de G un élément minimal du sous-ensemble $\tau(G)$, vis-à-vis de l'ordre $\preceq$ (chapitre 4).*

Nous verrons au chapitre 7 des techniques pour déterminer un graphe τ-minimal de G, soit de manière algorithmique si G est sans circuit, soit de manière mi-heuristique, mi-algorithmique, en utilisant la décomposition de G en composantes fortement connexes, si G possède des circuits.

Chapitre 6

Méthodes d'exploration

6.1 Développement arborescent issu d'un sommet donné

Définition 6.1 *On appelle arborescence un graphe* $A = (S,\Theta)$ *tel que :*

i) $\exists$ *un sommet* $s_0 \in S$ *et un seul tel que* $\Theta^{-1}(s_0) = \emptyset$.
ii) $\forall s \neq s_0 : \Theta^{-1}(s)$ *possède un seul élément.*
iii) *A est sans circuit.*

s_0 s'appelle *racine* de A, et tout sommet s sans successeur s'appelle une *feuille* de l'arborescence. Un chemin de la racine à une feuille s'appelle *branche* de l'arborescence.

Soit maintenant $G = (X,\Gamma)$ un graphe, et $x \in X$. On construit une arborescence A_x de la manière suivante :

- x est la racine ,
- Itérativement sur $k \geq 1 : \forall y \in \Gamma^{[k]}(x) : \Theta(y) = \Gamma(y)$

A_x s'appelle *développement arborescent* issu d'un sommet donné ; chaque branche de l'arborescence correspond à un chemin issu de x (certaines branches peuvent être infinies).

Exemple 5.1

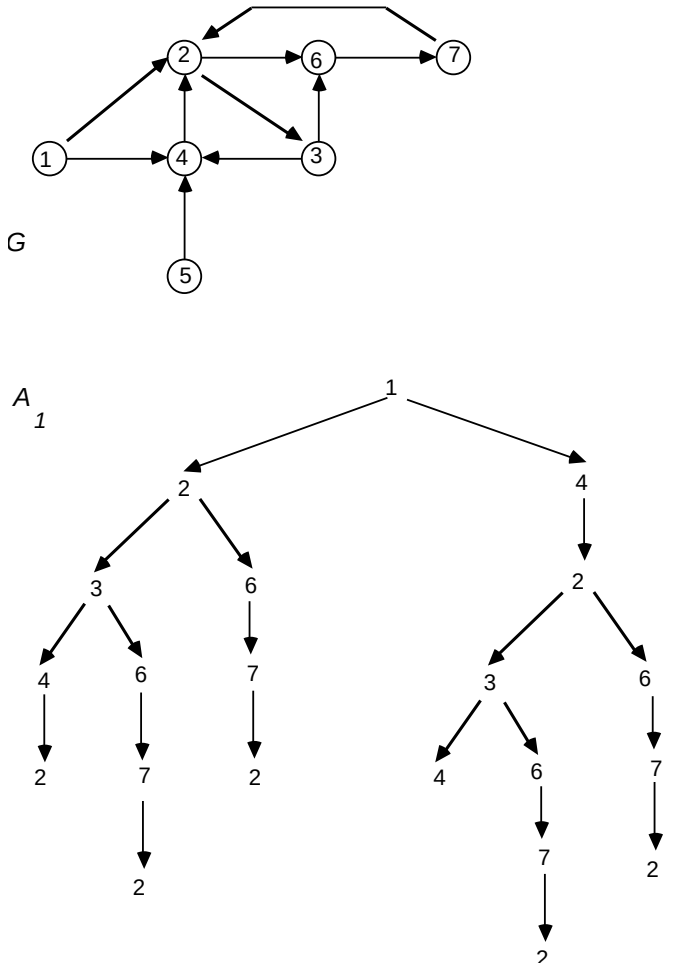


FIG. 6.1 – *Un graphe et son développement arborescent issu du sommet 1*

Les problèmes de cheminement issus de x que nous allons voir correspondent tous à des parcours d'une partie de l'arborescence A_x, selon différentes stratégies. De tels parcours sont encore connus sous le nom d'*exploration de la descendance de x*

6.2 Exploration de la descendance de x

6.2.1 Spécification du résultat

Explorer la descendance d'un sommet x, c'est effectuer un parcours du graphe $G_x = (\Gamma^*(x),\Gamma_x)$ où

$\Gamma^*(x)$ est l'ensemble des descendants de x (rappelons que $x \in \Gamma^*(x)$) et

$\Gamma_x = \{(y,z) \mid (y,z) \in \Gamma \wedge y \in \Gamma^*(x)\}$.

(Remarquons que $(y,z) \in \Gamma_x \Rightarrow z \in \Gamma^*(x)$).

Ce parcours doit permettre de **visiter** *au moins une fois* chaque sommet $y \in \Gamma^*(x)$ et *exactement une fois* chaque arc $(y,z) \in \Gamma_x$.

A chaque sommet (resp. arc) on attribue un booléen *sommet_visité* (resp. *arc_visité*) : $y.sommet\_visité$ (resp. $(y,z).arc\_visité$) est vrai si et seulement si le sommet y (resp. l'arc (y,z)) a été visité. La cohérence des états visités est exprimée par les axiomes suivants :

(1) $x.sommet\_visité$

(2) $\forall z \neq x : z.sommet\_visité \Rightarrow \exists y\ (y,z).arc\_visité \wedge z \in \Gamma^*(x)$

(3) $\forall y\ \forall z : (y,z).arc\_visité \Rightarrow y.sommet\_visité \wedge z.sommet\_visité \wedge (y,z) \in \Gamma_x$

Dans l'état initial, on a

$x.sommet\_visité$

$\forall z \neq x\ \neg z.sommet\_visité$

$\forall y\ \forall z : \neg(y,z).arc\_visité$

et dans l'état final, on doit avoir :

$\forall z : z.sommet\_visité \Leftrightarrow z \in \Gamma^*(x)$

$\forall y\ \forall z : (y,z)z.arc\_visité \Leftrightarrow (y,z) \in \Gamma_x$

6.2.2 Principe des méthodes d'exploration

Invariant. Au cours d'une exploration issue de x,

- tout sommet $y \in X$ est dans l'un des trois états suivants :

  dehors : $\neg y.visité$

  en_attente : $y.visité \wedge \exists z\ (y,z) \in \Gamma \wedge \neg(y,z).visité$

  terminé $y.visité \wedge \forall z\ (y,z) \in \Gamma \Rightarrow (y,z).arc\_visité.$

- tout arc est dans l'état *visité* ou *non visité*

Si l'on désigne par D (resp. A, T) l'ensemble des sommets d'état *dehors* (resp *en_attente*, *terminé*) et V l'ensemble des arcs d'état *visité*, les axiomes se traduisent par :

A1 $x \in A \cup T$

A2 $\forall z \neq x : z \in A \cup T \Rightarrow (\exists y(y,z) \in V) \wedge z \in \Gamma^*(x)$
A3 $\forall y \; \forall z \; (y,z) \in V \Rightarrow y \in A \cup T \wedge z \in A \cup T \wedge (y,z) \in \Gamma_x$

Condition d'arrêt

Proposition 6.2 $A = \emptyset \Leftrightarrow (T = \Gamma^*(x) \wedge V = \Gamma_x)$

Démonstration
i) Supposons $A = \emptyset$. Montrons que $T = \Gamma^*(x)$ et $V = \Gamma_x$.
- Comme $A \cup T \subseteq \Gamma^*(x)$, on a $T \subseteq \Gamma^*(x)$.
- Soit $y \in \Gamma^*(x)$. Il existe un chemin $\mathbf{u} = [x,z_1,\ldots,z_k,y](k \geq 0)$. Mais $x \in T \Rightarrow$ *arc_visité*$(x,z_1) \Rightarrow$ *sommet_visité*$(z_1) \Rightarrow z_1 \in A \cup T = T$. En répétant ce raisonnement, on obtient $z_k \in T \Rightarrow$ *arc_visité*$(z_k,y) \Rightarrow$ *sommet_visité*$(y) \Rightarrow y \in A \cup T = T$. Donc $\Gamma^*(x) \subseteq T$.
- $V \subseteq \Gamma_x$, par construction.
- Soit $(y,z) \in \Gamma_x$. On a donc $y \in \Gamma^*(x)$, c'est-à-dire $y \in T$ d'où *arc_visité*(y,z) soit $(y,z) \in V$. Donc, $\Gamma_x \subseteq V$.

ii) Réciproquement, supposons $T = \Gamma^*(x) \wedge V = \Gamma_x)$. Si $A \neq \emptyset$, il existe $y \in A$. On a donc $y \in \Gamma^*(x)$ (car $A \subseteq \Gamma^*(x)$) et $\exists z \; (y,z) \in \Gamma$, $(y,z) \notin V$. Mais $y \in \Gamma^*(x) \Rightarrow (y,z) \in \Gamma_x = V$. Contradiction. Donc $A = \emptyset$. □

Progression (* $A \neq \emptyset$ *)

Soit y un sommet de A
Soit S un ensemble de successeurs de y tel que $\forall z \in S : (y,z) \notin V$.
Pour tout $z \in S$, l'état de (y,z) devient *visité*
si z est dans l'état *dehors*
  alors (première visite de z)
    si $\Gamma(z) = \emptyset$
      alors z passe dans l'état *terminé*
      sinon z passe dans l'état *en_attente*
    fsi
  sinon
    z a déjà été visité à partir d'un autre de
    ses prédecesseurs
fsi
si $\forall z \in \Gamma(y)$ l'état de (y,z) est *visité* **alors** l'état de y devient *terminé* **fsi**

Valeurs initiales

$D = X \setminus \{x\}$
$A =$ **si** $\Gamma(x) = \emptyset$ **alors** $\emptyset$ **sinon** $\{x\}$
$T =$ **si** $\Gamma(x) = \emptyset$ **alors** $\{x\}$ **sinon** $\emptyset$
$V = \emptyset$

6.2.3 Preuve

Proposition 6.3 *Les valeurs initiales satisfont l'invariant*

Démonstration

A1 $x \in A \cup T$

A2 Si $z \neq x$ on a $z \in D$ donc $z \notin A \cup T$.

A3 $V = \emptyset$

□

Proposition 6.4 *La progression maintient l'invariant*

Démonstration Soit D, A , T , V les valeurs au début d'une étape de l'itération, y le sommet et S l'ensemble des successeurs de y sélectionnés au début de cette étape; par hypothèse, ces valeurs satisfont (A1) à (A3). Soit D', A', T', V' les valeurs à la fin du pas. Prouvons qu'elles vérifient les trois axiomes.

A1 $A \cup T \subseteq A' \cup T'$ donc $x \in A' \cup T'$

A2 Soit $z \neq x$. Si $z \in A' \cup T'$. Il y a deux cas :

1) $z \in A \cup T$. Comme (A2) est satisfait au début de l'étape, on a $(\exists y (y,z) \in V) \wedge z \in \Gamma^*(x)$. Comme $V \subseteq V'$, on a bien $(\exists y (y,z) \in V') \wedge z \in \Gamma^*(x)$.

2) $z \in D$. Alors z a été visité à partir de $y \in A \cup T$, donc (y,z) a été visité lors de cette étape : $V' = V \cup \{(y,z)$. De plus, d'après (A2) au début de l'étape, $y \in \Gamma^*(x)$ ce qui implique $z \in \Gamma^*(x)$. Donc on a bien $(\exists y (y,z) \in V') \wedge z \in \Gamma^*(x)$.

A3 Soit $(y,z) \in V'$. Il y a deux cas :

1) $(y,z) \in V$. Comme (A3) est satisfait au début de l'étape, on a $y \in A \cup T \wedge z \in A \cup T \wedge (y,z) \in \Gamma_x$. Comme $A \cup T \subseteq A' \cup T'$, on a aussi $y \in A' \cup T' \wedge z \in A' \cup T' \wedge (y,z) \in \Gamma_x$.

2) $(y,z) \notin V$. Cet arc a donc été visité lors de cette étape, donc $y \in A \subseteq A' \cup T'$ et, à l'issue de la visite, $z \in A' \cup T'$. De plus, (A2) étant vrai au début de l'étape, $y \in A \Rightarrow y \in \Gamma^*(x)$ et donc $(y,z) \in \Gamma_x$. On a donc bien $y \in A' \cup T' \wedge z \in A' \cup T' \wedge (y,z) \in \Gamma_x$.

□

Proposition 6.5 *L'invariant et la condition d'arrêt impliquent le résultat*

Démonstration D'après la proposition 6.2 □

Proposition 6.6 *L'algorithme s'arrête après un nombre fini d'étapes*

Démonstration La règle de progression fait décroître strictement la quantité entière card(Γ)-card(V) et cette quantité est minorée par 0. □

6.2.4 Texte de l'algorithme général d'exploration

L'état des sommets est mis en œuvre à l'aide des trois ensembles D, A, T et celui des arcs à l'aide de l'ensemble V; en outre, à chaque sommet de $y \in A$ est associé un ensemble de sommets *a_visiter* contenant les successeurs z de y tels que $(y,z) \notin V$ (si $y \in D$, *a_visiter* n'est pas défini et, si $y \in T$, on a $a\_visiter = \emptyset$).

La fonction *sous_ensemble*, appliquée à un ensemble non vide, délivre un sous-ensemble de cet ensemble. La fonction *élément*, appliquée à un ensemble non vide, délivre un élément quelconque de cet ensemble (sans le retirer).

Le texte de l'algorithme est donné page 57

6.2.5 Complexité

Chaque sommet de $\Gamma^*(x)$ passe une fois et une seule dans l'état *en_attente*. Lorsqu'un sommet z passe de l'état *dehors* à l'état *en_attente*, l'ensemble $a\_visiter(z)$ est établi, par l'appel de l'accès *lst_succ*. Il y a donc

card($\Gamma^*(x)$) appels de l'accès *lst_succ*

et, chaque accès étant, selon sa mise en œuvre, au pire en $O(n)$, on a un algorithme en $O(n^2)$ (en termes d'opérations du type *validarc*)

6.3 Stratégies d'exploration particulières : largeur et profondeur

Les stratégies d'exploration que nous allons maintenant développer dépendent de la gestion de l'ensemble A, du critère de choix d'un élément y dans cet ensemble (celui à partir duquel on tente de prolonger l'exploration lors d'un pas d'itération) et du choix de l'ensemble S. Nous en développons deux, appelées respectivement :

1. recherche en largeur d'abord (*breadth-first* ou *width first* search)
2. recherche en profondeur d'abord (*depth-first* search)

D'autre part, le canevas d'exploration de la descendance d'un sommet x peut être utilisé pour effectuer un calcul particulier sur le graphe, par exemple :

- Ensemble des descendants de x
- Énumération des chemins et circuits élémentaires issus de x
- développement arborescent de hauteur minimum issu de x
- Arborescence de chemins de valeur optimum issus de x dans un graphe valué (ce problème sera traité au chapitre 8)
- etc.

Dans la suite de ce chapitre, les deux premiers exemples seront examinés; la stratégie d'exploration en largeur et une expression récursive de la stratégie d'exploration en profondeur seront explicitées sur le premier d'entre eux, et l'exploration en profondeur sur le second.

ALGORITHME GENERAL D'EXPLORATION

```
ENS[SOMMET] D, T; ENS[ARC] V ;
exploration-generale(GRAPHE G; SOMMET x)  c'est
   local ENS[SOMMET] A, S;
         ENS[SOMMET] SOMMET::a_visiter ;
         SOMMET y, z ;
 début
   depuis D ← G.lst_som\{x} ; V ← ∅ ;
         x.a_visiter ← G.lst_succ(x) ;
          si x.a_visiter = ∅
             alors A ← ∅ ; T ← {x}
             sinon A ← {x} ; T ← ∅
          fsi
   jusqua A = ∅
   faire
     y ← A.element ;
     S ← y.a_visiter.sous_ensemble;
      --   mise à jour du statut de y
     y.a_visiter ← y.a_visiter\ S;
      si y.a_visiter=∅  alors A ← A\{y} ; T ← T ∪ {y}  fsi ;
      pour tout z  de S
        V ← V ∪ {(y,z)} ;
        --   mise à jour du statut de z
         si z ∈ D  alors  D ← D\{z} ;
                          z.a_visiter ← lst_succ(G,z) ;
                           si z.a_visiter = ∅
                              alors T ← T ∪ {z}
                              sinon A ← A ∪ {z}
                           fsi
         fsi
      fpourtout
   fait
  --   exploration terminée : Γ*(x) = T , Γ_x = V
 fin
```

6.3.1 Exploration en largeur d'abord

6.3.1.1 Une structure de données : la file

Cette stratégie consiste à sélectionner dans A le sommet y *le plus anciennement ajouté* et à choisir $S = \Gamma(y)$. Au niveau de l'arborescence, cela revient à effectuer un parcours *en largeur* (niveau par niveau) : x est visité, puis tous les arcs issus de x; ensuite, tous les successeurs de x et les arcs qui en sont issus, puis tous les sommets de $\Gamma^{[2]}(x)$ – qui n'ont pas été visités – , et ainsi de suite. L'algorithme qui résulte de cette stratégie visite donc les éléments de $\Gamma^*(x)$ dans l'ordre suivant :

$$x;\quad \Gamma(x) \setminus \{x\};\quad \Gamma^{[2]}(x) \setminus \Gamma(x) \setminus \{x\};\quad \ldots\quad ;\quad \Gamma^{[k]}(x) \setminus \bigcup_{j=0}^{k-1} \Gamma^{[j]}(x);\quad \ldots$$

Pour cette raison, il est parfois qualifié de "*concentrique*".

Nous allons mettre en oeuvre cette technique avec une structure de données qui s'appelle une **file**. Une file est une liste sur laquelle les seules modifications possibles sont :

1. ajout d'un élément en queue (*mettreenfile*)
2. retrait de l'élément de tête (*oterdefile*)

On dit encore que cette liste est gérée selon la discipline **P.A.P.S.** (**P**remier **A**rrivé **P**remier **S**orti) ou **F.I.F.O.** (**F**irst **I**n **F**irst **O**ut).

Nous considérons une file comme un type abstrait et la déclaration
`FILE[E] f` déclare un objet `f` de type *file* d'objets de type `E`.

Les primitives d'accès sont :

`creer`	crée une nouvelle file, vide
`BOOL filevide`	prédicat à valeur `vrai` si et seulement si la file courante est vide
`mettreenfile(E v)`	ajoute l'objet désigné par `v` en queue de file
`E oterdefile`	délivre le premier élément et l'enlève de la file.

Le principe de mise en œuvre est alors le suivant : pour le sommet y en tête de file, on examine séquentiellement tous les successeurs et chacun d'eux est immédiatement ajouté en queue de file s'il n'a pas encore été visité; puis y est ôté de la file puisqu'il ne peut plus apporter d'information nouvelle. Il est donc inutile, avec cette stratégie, de gérer les ensembles de successeurs *a_visiter*, puisque tous les arcs issus du sommet sélectionné (tête de file) sont visités lors du même pas d'itération.

6.3.1.2 Texte de l'algorithme

Le texte de l'algorithme est donné ci-après, page 59.

6.3.1.3 Exemple d'exécution

La figure 6.2 montre le résultat de l'exploration *en largeur d'abord* issue du sommet 1 dans le graphe de la figure 6.1

EXPLORATION LARGEUR D'ABORD

```
ENS[SOMMET] D, T;
ENS[ARC] V;

exploration-largeur(GRAPHE G; SOMMET x) c'est
 local ENS[SOMMET] S; --  ensemble des successeurs de la tête de file
       FILE[SOMMET] A;
       SOMMET y, z;
 début
    depuis D ← G.lst_som\{x} ;
           A.creer ; A.mettreenfile(x) ;
           T ← ∅ ; V ← ∅ ;
    jusqua A.filevide
    faire
      y ← A.oterdefile ; T ← T ∪ {y} ;
      S ← G.lst_succ(y) ;
      pour tout z de S
         V ← V ∪ {(y,z)} ;
          si z ∈ D
             alors D ← D\{z} ; A.mettreenfile(z)
          fsi
      fpourtout
    fait
--  exploration terminée
 fin
```

6.3.2 Recherche en profondeur d'abord

6.3.2.1 Une structure de données : la pile

Cette recherche consiste à sélectionner dans A le sommet y *le plus récemment ajouté*, et pour S *un* arc non visité issu de y. Au niveau de l'arborescence, cela revient à effectuer un parcours en descendant le plus loin possible sur chaque branche. Lorsqu'un sommet est rencontré :

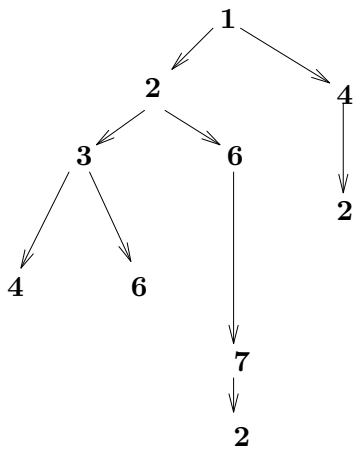
- s'il est dans l'état *dehors*, il devient *en_attente*
- s'il n'est pas dans l'état *dehors*, il marque l'arrêt de la descente, et la recherche reprend à partir de son prédécesseur dans l'arborescence.

Pour mettre en oeuvre cette technique, on utilise une structure de données qui s'appelle une **pile**; il s'agit d'une liste sur laquelle les seules modifications possibles sont :

1. ajout d'un élément en queue de liste *(empiler)*
2. retrait du dernier élément de la liste *(dépiler)*.

file	**D**	**R**
1	2, 3, 4, 5, 6, 7	$\emptyset$
2, 4	3, 5, 6, 7	1
4, 3, 6	5, 7	1, 2
3, 6	5, 7	1, 2, 4
6	5, 7	1, 2, 4, 3
7	5	1, 2, 4, 3, 6
ϵ	5	1, 2, 4, 3, 6, 7

arborescence



$R = \Gamma^{\star}(1) = \{1,2,3,4,6,7\} = X \setminus \{5\}$

$V = \Gamma_1 = \{(1,2),(1,4),(2,3),(2,6),(3,4),(3,6),(4,2),(6,7),(7,2)\} = \Gamma \setminus \{(5,4)\}$

FIG. 6.2 – *exploration en largeur*

L'élément de queue de cette liste est aussi désigné sous le nom de *sommet de pile*.

On dit encore que cette liste est gérée selon la discipline **DAPS** (**D**ernier **A**rrivé **P**remier **S**orti) ou encore **LIFO** (**L**ast **I**n **F**irst **Out**).

Nous considérons donc une pile comme un type abstrait, et la déclaration

`PILE[E]` ℓ déclare une *liste d'objets de type* `E` gérés en pile.

Les opérations sur les piles sont alors :

`creer`	crée une nouvelle pile vide
`BOOL pilevide`	prédicat à valeur `vrai` si et seulement si la pile est vide
`empiler(E z)`	ajout de `z` en queue
`dépiler`	retrait de l'élément de queue (erreur si la pile est vide)
`E sommetpile`	délivre la valeur du sommet de pile (consultation sans dépiler)

6.3.2.2 Texte de l'algorithme

Il est donné ci-après, page 61

6.3.2.3 Exemple d'exécution

La figure 6.3 montre le résultat de l'exploration *en profondeur d'abord* issue du sommet 1 dans le graphe de la figure 6.1

EXPLORATION PROFONDEUR D'ABORD

```
ENS[SOMMET] D, T;
ENS[ARC] V;

recherche-profondeur(GRAPHE G; SOMMET x)  c'est
 local PILE[SOMMET] A;
       ENS[SOMMET] SOMMET::a_visiter ;
       SOMMET y; --   sommet de pile
       SOMMET z; --   sommet courant
 début
    depuis
       D ← G.lst_som\{x} ; V ← ∅
       A.creer ; x.a_visiter ← G.lst_succ(x) ;
        si x.a_visiter = ∅
           alors T ← {x}
           sinon A.empiler(x); T ← ∅
        fsi ;
    jusqua A.pilevide
    faire
      y ← A.sommetpile ;
       si  y.a_visiter ≠ ∅
          alors
             z ← (y.a_visiter).element ;
             y.a_visiter ← y.a_visiter\{z} ;
             V ← V ∪ {(y,z)} ;
              si z ∈ D  alors
                 D ← D\{z} ;
                 z.a_visiter ← G.lst_succ(z) ;
                  si z.a_visiter  = ∅
                     alors T ← T ∪ {z}
                     sinon A.empiler(z)
                  fsi
              fsi
          sinon T ← T ∪ {y} ; A.depiler
       fsi
    fait
--   exploration terminée
 fin
```

pile	D	R
1	2, 3, 4, 5, 6, 7	∅
1, 2	3, 4, 5, 6, 7	∅
1, 2, 3	4, 5, 6, 7	∅
1, 2, 3, 4	5, 6, 7	∅
1, 2, 3, 4	5, 6, 7	∅
1, 2, 3	5, 5, 7	4
1, 2, 3, 6	5, 7	4
1, 2, 3, 6, 7	5	4
1, 2, 3, 6, 7	5	4
1, 2, 3, 6	5	4, 7
1, 2, 3	5	4, 7, 6
1, 2	5	4, 7, 6, 3
1, 2	5	4, 7, 6, 3
1	5	4, 7, 6, 3, 2
1	5	4, 7, 6, 3, 2
ϵ	5	4, 7, 6, 3, 2, 1

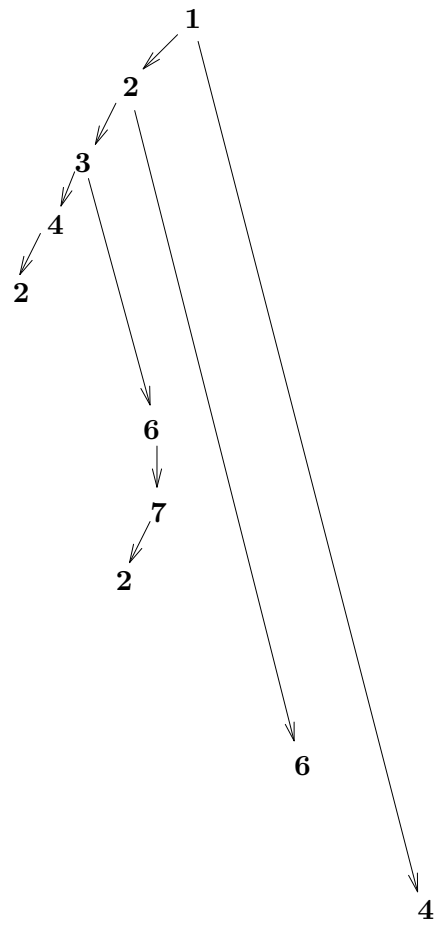


FIG. 6.3 – *exploration en profondeur*

6.4 Exemple d'application : calcul de l'ensemble des descendants

L'algorithme général de recherche permet d'obtenir l'ensemble des descendants du sommet de départ : en effet, cet ensemble est égal, à l'issue de l'algorithme, à T. Toutefois, l'exploration peut être simplifiée, dans la mesure où l'on ne cherche pas à calculer l'ensemble V des arcs du graphe G_x, mais seulement l'ensemble $\Gamma^*(x)$ de ses sommets. Quelle que soit la stratégie d'exploration retenue, la simplification est la suivante :

- Suppression de la variable V
- Le statut des sommets peut être mémorisé à l'aide de 2 ensembles seulement : A et $DESC$,

ce dernier contenant à tout instant du calcul l'ensemble des sommets reconnus comme descendants de x, c'est-à-dire $A \cup T$. En effet,

- $y \in D \Leftrightarrow y \notin DESC$
- $y \in T \Leftrightarrow y \notin A \wedge y \in DESC$

6.4.1 Cas de la recherche en largeur

Nous donnons le texte de l'algorithme simplifié dans le cas de l'exploration en largeur (page 63).

CALCUL DES DESCENDANTS : LARGEUR D'ABORD

```
ENS[SOMMET] DESC;

descendants-largeur(GRAPHE G;SOMMET x) c'est
 local  ENS[SOMMET] S;
        FILE[SOMMET] A;
        SOMMET y, z ;
 début
    depuis
      DESC ← {x} ; A.creer ; A.mettreenfile(x)
    jusqua A.filevide
    faire
      y ← A.oterdefile ;
      S ← G.lst_succ(y) ;
       pour tout z  de S
            si z ∉ DESC
               alors
                  DESC ← DESC ∪ {z} ;
                  A.mettreenfile(z)
            fsi
       fpourtout
    fait
 fin
```

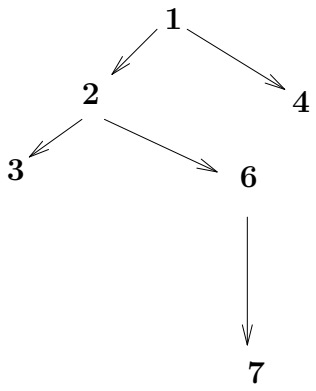
La figure 6.4 montre, à titre de comparaison avec l'exploration complète (figure 6.2), le résultat de la recherche des descendants du sommet 1 dans le graphe de la figure 6.1

6.4.2 Cas de la recherche en profondeur : expression récursive

Le texte de cet algorithme est donné page 65

file	**DESC**
1	1
2, 4	1, 2, 4
4, 3, 6	1, 2, 4, 3, 6
3, 6	1, 2, 4, 3, 6
6	1, 2, 4, 3, 6
7	1, 2, 4, 3, 6, 7
ϵ	1, 2, 4, 3, 6, 7

arborescence



$$\text{DESC} = \Gamma^\star(1) = \{1,2,3,4,6,7\} = X \setminus \{5\}$$

FIG. 6.4 – *calcul des descendants (largeur)*

On peut alors vérifier que le mode de recherche correspond à une recherche profondeur d'abord. Dans une expression itérative de cet algorithme récursif, la manipulation de la pile serait alors explicite, via la variable A (cette manipulation correspond à la mémorisation des contextes de retour des différents appels de la procédure *desc_local*).

6.5 Énumération des chemins élémentaires issus de x

Le problème posé ici est plus complexe, puisqu'il s'agit d'obtenir les tracés de tous les chemins élémentaires issus du sommet x. Il faut donc parcourir une sous-arborescence plus importante, puisqu'un même sommet peut apparaître plusieurs fois (comme extrémité de plusieurs chemins distincts de G). En fait, les chemins élémentaires issus de x sont les branches de l'arborescence, parcourues jusqu'à trouver une répétition dans la suite des sommets situés sur une branche. Dans l'exemple de la figure 6.1, il s'agit de l'arborescence représentée graphiquement.

Le choix d'une exploration en profondeur s'explique par la propriété suivante :

Proposition 6.7 *Lors d'une exploration en profondeur, l'état de la pile est un chemin élémentaire d'origine x et d'extrémité le sommet de pile.*

Démonstration Elle se fait par induction sur l'état de la pile.

– Le premier état non vide de la pile est x, donc vérifie la propriété.

– Supposons la propriété vérifiée au début d'un pas d'itération, et soit

CALCUL DES DESCENDANTS : FORMULATION RECURSIVE

```
ENS[SOMMET] DESC;

desc_local (SOMMET y)  c'est
 local  ENS[SOMMET] S;
        SOMMET z;
 début
    DESC ← DESC ∪ {y} ;
    S ← G.lst_succ(y) ;
     pour tout z  de S  faire
        si z ∉ DESC  alors desc_local(z)  fsi
     fpourtout
 fin


descendants-recursif(GRAPHE G; SOMMET x)  c'est
 début
   DESC ← ∅ ; desc_local(x)
 fin
```

$x \ \ldots \ t \ y$ l'état de la pile. Montrons qu'elle reste vérifiée à la fin du pas.

cas 1 Le pas courant ne modifie pas l'état de la pile. Ce cas est trivial;

cas 2 Le pas courant effectue une opération *dépiler*; le résultat est évident puisqu'un chemin élémentaire privé de son dernier sommet reste un chemin élémentaire; ici, le chemin obtenu va de x à t, ce dernier sommet étant le sommet de pile;

cas 3 Le pas courant effectue une opération *empiler*. Soit z le sommet empilé. Par construction, z est un successeur de y, donc le nouvel état de la pile $x \ \ldots \ t \ y \ z$ est bien un chemin de x au sommet de pile z. De plus le sommet z étant empilé à cette étape, il appartenait à D, donc ne figurait pas dans la pile au début du pas; par conséquent, l'état de la pile à la fin du pas est un chemin *élémentaire*.

□

Par rapport à l'algorithme d'exploration vu au §6.3.2, il faut considérer toutes les occurrences possibles d'un sommet dans l'arborescence d'exploration : chacune d'elles, provenant d'un prédécesseur différent, engendre un nouvel ensemble de chemins via sa descendance. Ainsi, la progression doit être modifiée de la manière suivante :

Progression Soit y le sommet de la pile; sélectionner un successeur de y, non encore visité. S'il n'y en a pas, alors dépiler y, qui passe dans l'état *terminé*; s'il y en a un soit z ; si $z \in A$

alors il ne définit pas de nouveau chemin (sauf si $z = x$: circuit élémentaire); si $z \in D$, on l'empile (c'est la première fois qu'il est visité), et donc tous ses arcs sortants deviennent à visiter ($a\_visiter(z) \leftarrow lst\_succ(G,z)$); si $z \in T$ (il est déjà passé dans la pile puis en est ressorti), il faut procéder de même puisque l'état de la pile est un chemin élémentaire de x à z, et tous les chemins élémentaires ayant cet état de pile comme préfixe vont alors être reconnus.

Un exemple d'exécution est donné page 67. Le texte de l'algorithme est donné page 68, et une version récursive page 69. **N.B.** : l'accès `ecrirechemin` permet d'afficher l'état d'une pile, qui correspond alors à un chemin élémentaire issu de `x`. Un circuit élémentaire passant par un sommet `z` présent dans la pile est obtenu par l'accès `ecrirecircuit(z)` qui affiche l'état de la pile depuis le sommet `z`, suivi du sommet `z`. Ces accès permettent d'ailleurs d'incorporer des restrictions sur la nature des chemins que l'on souhaite énumérer : conditions sur la longueur, sur les sommets traversés, etc.

6.5.1 Algorithmes gloutons et non gloutons

Les algorithmes d'exploration présentés dans ce chapitre se classent en *gloutons* et *non gloutons*. L'exploration de la descendance rentre dans la catégorie des gloutons, ce qui signifie que certains résultats sont définitivement acquis avant la fin de l'exécution de l'algorithme : c'est le cas, ici, lorsqu'un sommet passe de l'état *en_attente* à l'état *terminé* : on enregistre ainsi le fait que *toute l'information que l'on peut obtenir à partir de ce sommet a été obtenue.*

Au contraire, l'énumération des chemins élémentaires est un algorithme non glouton, car aucun résultat ne peut être considéré comme définitif avant d'avoir rencontré la condition d'arrêt; ce qui se traduit, ici, par le fait que *le statut terminé d'un sommet peut être remis en cause* - lorsque ce sommet est à nouveau visité à partir d'un nouveau prédécesseur, toute l'exploration issue de ce sommet doit être refaite, et donc le sommet "régresse" dans l'état où il était lors de son passage *dehors* → *en_attente.*

Nous retrouverons au chapitre 8 cette distinction entre algorithmes gloutons et non gloutons, à l'occasion des calculs de chemins de valeur optimale.

Exemple d'exécution Énumération des *chemins* et *circuits* élémentaires issus du sommet 1 dans le graphe de la figure 6.1 :

pile	**chemin**	**circuit**
1		
1,2	1,2	
1,2,3	1,2,3	
1,2,3,4	1,2,3,4	
1,2,3,4,2		2,3,4,2
1,2,3,4		
1,2,3		
1,2,3,6	1,2,3,6	
1,2,3,6,7	1,2,3,6,7	
1,2,3,6,7,2		2,3,6,7,2
1,2,3,6,7		
1,2,3,6		
1,2,3		
1,2		
1,2,6	1,2,6	
1,2,6,7	1,2,6,7	
1,2,6,7,2		2,6,7,2
1,2,6,7		
1,2,6		
1,2		
1		
1,4	1,4	
1,4,2	1,4,2	
1,4,2,3	1,4,2,3	
1,4,2,3,6	1,4,2,3,6	
1,4,2,3,6,7	1,4,2,3,6,7	
1,4,2,3,6,7,2		2,3,6,7,2
1,4,2,3,6,7		
1,4,2,3,6		
1,4,2,3		
1,4,2,3,4		4,2,3,4
1,4,2,3		
1,4,2		
1,4,2,6	1,4,2,6	
1,4,2,6,7	1,4,2,6,7	
1,4,2,6,7,2		2,6,7,2
1,4,2,6,7		
1,4,2,6		
1,4,2		
1,4		
1		
ϵ		

ENUMERATION DES CHEMINS ELEMENTAIRES

```
ENS[SOMMET] D, T
ENS[ARC] V;

enumeration-chemins(GRAPHE G; SOMMET x)  c'est
 local  PILE[SOMMET] A;
        ENS[SOMMET] SOMMET.a_visiter ;
        SOMMET y; --   sommet de pile
        SOMMET z; --   sommet courant
 début
    depuis
     D ← G.lst_som\{x} ;
     A.creer;
     x.a_visiter ← G.lst_succ(x)\{x} ;
      si x.a_visiter ≠ ∅  alors A.empiler(x)  fsi
    jusqua A.pilevide
    faire
      y ← A.sommetpile ;
       si  y.a_visiter ≠ ∅
          alors
            z ← (y.a_visiter).element ;
            y.a_visiter ← y.a_visiter\{z} ;
             si z ∉ A
                alors
                   si z ∈ D  alors D← D\{z}  fsi ;
                  z.a_visiter ← G.lst_succ(z) ;
                  A.empiler(z) ;
                  A.ecrirechemin
                sinon A.ecrirecircuit(z)
             fsi
          sinon T ← T ∪ {y} ; A.depiler
       fsi
    fait
 fin
```

ENUMERATION DES CHEMINS ELEMENTAIRES : VERSION RECURSIVE

```
ENUM[dehors, en_attente, terminé] SOMMET::etat ;--   type énuméré
PILE[SOMMET] chemin_courant ;

enumerer(SOMMET y)  c'est
--   pre : y.etat=en_attente
    local SOMMET z
 début
   pour tout  z  de G.lst_succ(y)
   --   y est le  père de z dans l'arborescence d'exploration
      cas
       z.etat= dehors,terminé →
         chemin_courant.empiler(z) ;
         chemin_courant.ecrirechemin ;
         z.etat ← en_attente ;
         enumerer(z)
       z.etat=en_attente →
         chemin_courant.ecrirecircuit(z)
      fcas
   fpourtout;
  chemin_courant.depiler ;
   --   le sommet y est dépilé
  y.etat ← terminé
 fin --   enumerer

enumeration-recursive(GRAPHE G; SOMMET x)  c'est
 début
  chemin_courant.creer ;
   pour tout y  de G.lst_som
     y.etat ← dehors
   fpourtout ;
  x.etat ← en_attente ;
  chemin_courant.empiler(x);
  enumerer(x)
 fin
```

Chapitre 7

Circuits. Composantes fortement connexes

7.1 Détermination de l'existence de circuits

7.1.1 Principe

Dans de nombreuses applications, on souhaite savoir si un graphe donné possède ou non des circuits (cf. exemple de l'interblocage). Certains des algorithmes que nous avons vus précédemment (ROY-WARSHALL, DESCENDANTS) pourraient être utilisés :

- ▷ avec ROY-WARSHALL on obtient l'ensemble des sommets situés sur un circuit, puisque de tels sommets vérifient $(x,x) \in G^+$.
- ▷ si on se restreint à l'existence de circuits situés dans le sous-graphe engendré par $\Gamma^*(x)$, x étant un sommet donné, on peut utiliser l' algorithme de recherche des descendants de x en profondeur, sachant qu'un circuit existe dès lors qu'on trouve un sommet successeur du sommet de pile, et appartenant déjà à *desc*.

Mais, avec ROY-WARSHALL, la complexité est élevée (cet algorithme résout le problème plus général de la fermeture transitive), et avec la recherche des descendants de x, on ne détecte que les circuits passant par les sommets de $\Gamma^*(x)$. Nous allons voir un algorithme spécifique, appelé algorithme de MARIMONT, permettant de résoudre le problème avec une complexité moindre que celle de ROY-WARSHALL. Cet algorithme est fondé sur la propriété suivante :

Proposition 7.1 *Les propositions suivantes sont équivalentes :*
i) G est sans circuit,
ii) G et tous ses sous-graphes possèdent au moins un point d'entrée (sommet sans prédécesseur) et un point de sortie (sommet sans successeur).

Démonstration. *i)* $\Rightarrow$ *ii)* Supposons $G = (X,\Gamma)$ sans circuit, et que G n'a pas de point de

sortie. Alors :

$$\forall x \in X, \exists y \in X \ : y \in \Gamma(x)$$

Autrement dit, partant d'un sommet x_i quelconque, on peut construire un chemin $x_i, x_{i_1}, x_{i_2}, \ldots, x_{i_k}, \ldots$ de longueur aussi grande que l'on veut ; et dès que la longueur est $> n$, on obtient un circuit. D'où contradiction avec l'hypothèse. On montre de manière identique que G a au moins un point d'entrée. Enfin, si G est sans circuit, tout sous-graphe engendré par une partie de X est aussi sans circuit, ce qui montre la première partie de la proposition.
ii) $\Rightarrow$ *i*) Supposons que G ait un circuit γ et soit G_γ le sous-graphe engendré par les sommets de ce circuit. Par construction, G_γ n'a ni point d'entrée ni point de sortie, ce qui contredit l'hypothèse. □

Le principe de l'algorithme est alors très simple : si G ne possède pas de points d'entrée ou de points de sortie, on peut conclure que G a des circuits. Sinon, on ôte les points d'entrée et les points de sortie, et on recommence sur le sous-graphe engendré. Si on aboutit au graphe vide (tous les sommets de G ont été ôtés), on conclut que G est sans circuit. Remarquons que cet algorithme peut être mis en oeuvre uniquement à partir des points d'entrée, ou des points de sortie, ou à partir des deux. Nous ne donnons, ci-dessous, que cette dernière variante.

7.1.2 Algorithme de Marimont

7.1.2.1 Analyse

Invariant : H sous-graphe de G ;
E = ensemble des points d'entrée de H ;
S = ensemble des points de sortie de H.

Arrêt : H vide **ou** $E = \emptyset$ **ou** $S = \emptyset$
Traitement de sortie :
il y a des circuits si et seulement si H est non vide.

Progression : (H non vide **et** $E \neq \emptyset$ **et** $S \neq \emptyset$)
$H \leftarrow$ôter de H les points d'entrée et les points de sortie.

Valeurs initiales : $H \leftarrow G$

Pour améliorer l'efficacité au niveau de la programmation, nous mémorisons dans une variable booléenne *h_vide* le résultat du test *vide(H)*

7.1.2.2 Texte de l'algorithme

Il est donné page 73

EXISTENCE DE CIRCUITS : MARIMONT

```
BOOL marimont(GRAPHE G) c'est
    local GRAPHE H ; --   graphe de travail
          ENS[SOMMET] E, S; --   points d'entrée, de sortie
          BOOL h_vide ;
 début
    depuis
      H ← G ; h_vide ← H.vide ;
       si non h_vide  alors
          E ← H.points_entree ;
          S ← H.points_sortie
       fsi ;
    jusqua h_vide  ou E= ∅  ou S =∅
    faire
      H ← H.sous_graphe(G.lst_som \ E ∪ S) ; h_vide ← H.vide ;
       si non h_vide  alors
         E ← H.points_entree;
         S ← H.points_sortie
       fsi
    fait ;
  --   ici, h_vide  ou E = ∅  ou S = ∅
   Result ←  non h_vide
 fin
```

7.1.2.3 Complexité

Dans le cas le pire (tous les sommets de G sont ôtés, et seulement 2 sommets à chaque pas d'itération), il y a $\lceil n/2 \rceil$ pas d' itération. Le pas n^o k nécessite
1 accès *points_entrée*
1 accès *points_sortie*
sur un sous-graphe à $n - 2(k-1)$ sommets, soit :

$n - 2k + 2$ accès *lst_pred*

$n - 2k + 2$ accès *lst_succ*

Au total, il y a donc *au pire* :

$\sum_{k=1}^{\lceil n/2 \rceil} (n - 2k + 2)$ accès *lst_pred* et autant d'accès *lst_succ*.

Mais :

$$\sum_{k=1}^{\lceil n/2 \rceil} (n-2k+2) = \begin{cases} \frac{n(n+2)}{4} & \textbf{si } n \text{ est pair,} \\ \frac{(n+1)^2}{4} & \textbf{si } n \text{ est impair} \end{cases}$$

soit $O(n^2)$ accès *lst_pred, lst_succ.*

7.2 Application des graphes sans circuit : fonction ordinale

Définition 7.2 *On appelle* **fonction ordinale** *d'un graphe* $G = (X,\Gamma)$ *une fonction* $f : X \to \mathbb{N}$ *telle que*

$$(x,y) \in \Gamma \Rightarrow f(x) < f(y)$$

On peut aussi définir une *fonction ordinale inverse* f^t, qui n'est autre qu'une fonction ordinale du graphe transposé :

$(x,y) \in \Gamma \Rightarrow f^t(x) > f^t(y)$

Théorème 7.3 *Un graphe possède une fonction ordinale si et seulement si il est sans circuit.*

Démonstration. Supposons que G possède une fonction ordinale, et qu'il ait un circuit :

$\gamma = [x_{i_1},x_{i_2},\ldots,x_{i_k},x_{i_1}]$

On a :

$$f(x_{i_1}) < f(x_{i_2}) < \ldots < f(x_{i_k}) < f(x_{i_1})$$

d'où, par transitivité :

$$f(x_{i_1}) < f(x_{i_1})$$

ce qui est absurde.

Réciproquement supposons G sans circuit et considérons l'algorithme de MARIMONT ne prenant en compte que les points d'entrée. Au pas d'itération n^o k, le sous-graphe courant a des points d'entrée E. On pose alors, par définition :

$$\forall x \in E \ : f(x) = k \quad (k = 1,2,\ldots,p)$$

où p est le nombre de pas, majoré par n.

f définit bien une fonction ordinale : si $(y,z) \in \Gamma$ alors le sommet y est ôté avant le sommet z, car z ne peut pas être point d'entrée d'un sous-graphe possédant le sommet y ; donc : $(y,z) \in \Gamma \Rightarrow f(y) < f(z)$.

De plus, tous les sommets étant ôtés, f est bien définie sur X. □

La démonstration du théorème montre que l'on peut obtenir une fonction ordinale en appliquant l'algorithme de MARIMONT à partir des points d'entrée, à condition d'introduire un compteur de marquage des sommets. Nous laissons au lecteur le soin de ré-écrire l'algorithme.

Remarquons qu'une fonction ordinale inverse peut aussi être obtenue en opérant à partir des points de sortie.

Enfin, ces deux fonctions ordinales s'interprètent concrètement comme suit :

• **à partir des points d'entrée** :

$\forall x \in X \ : f(x) = 1 +$ longueur maximum des chemins issus d'un point d'entrée et aboutissant à x

La relation suivante est vérifiée :

$$\forall x \in X \ : f(x) = \begin{cases} 1 & \textbf{si } \Gamma^{-1}(x) = \emptyset \\ \max_{y \in \Gamma^{-1}(x)}(f(y)) + 1 & \textbf{si } \Gamma^{-1}(x) \neq \emptyset \end{cases}$$

• **à partir des points de sortie** :

$\forall x \in X \ : f^t(x) = 1 +$ longueur maximum des chemins issus de x et aboutissant à un point de sortie

La relation suivante est vérifiée :

$$\forall x \in X \ : f^t(x) = \begin{cases} 1 & \textbf{si } \Gamma(x) = \emptyset \\ \max_{y \in \Gamma(x)}(f(y)) + 1 & \textbf{si } \Gamma(x) \neq \emptyset \end{cases}$$

Une fonction ordinale n'est pas nécessairement injective : deux sommets distincts $x \neq y$ peuvent avoir une même image $f(x) = f(y)$. Une fonction non injective ne convient donc pas pour *numéroter* les sommets. C'est pourquoi l'on introduit la notion de *numérotation conforme*, qui n'est autre qu'une fonction ordinale injective.

Définition 7.4 *Soit $G = (X,\Gamma)$ avec card(X) = n. On appelle* **numérotation conforme** *de X une injection*

$num \ : X \Rightarrow \mathbb{N}$

telle que $(x,y) \in \Gamma \Rightarrow num(x) < num(y)$

Théorème 7.5 *Un graphe G possède une numérotation conforme si et seulement si il est sans circuit.*

Démonstration. On peut démontrer ce théorème comme corollaire du théorème précédent 7.3. Toutefois, nous préférons en donner une démonstration directe: celle-ci montre comment obtenir une telle numérotation à partir d'une exploration en profondeur.

i) Si G possède une numérotation conforme, il est clairement sans circuit.

ii) Pour montrer la réciproque, on suppose le graphe sans circuit, et on utilise l'algorithme suivant, basé sur l' exploration en profondeur: partant d'un sommet *quelconque*, on explore la descendance de ce sommet, en numérotant les sommets visités de manière *décroissante* lors de leur *sortie* de la pile. Tant qu'il reste des sommets non numérotés, on réitère le procédé, en se limitant au sous-graphe engendré par les sommets non encore numérotés. L'analyse de cet algorithme est donnée ci-dessous:

Invariant :

T=ensemble des sommets déjà numérotés, $D = X \setminus T$
Le sous-graphe engendré par T possède une numérotation conforme,
der_num=plus petit numéro de sommet dans T (dernier numéro attribué).

Arrêt : $D = \emptyset$.

Progression : Soit $x \in D$. Explorer en numérotant la descendance de x (voir détail de ce traitement ci-après).

Valeurs initiales : $D = X$, $T = \emptyset$, *der_num*=une valeur quelconque telle que *der_num* $> n(= \text{card}(X))$.

Explorer en numérotant: effectuer l'exploration en profondeur de la descendance de x dans le sous-graphe engendré par les sommets de D (non encore numérotés). La numérotation des nouveaux sommets visités se fait lors de leur sortie de pile, en faisant décroître le compteur *der_num*. Ce traitement se termine lorsque l'exploration est finie. A ce moment, tous les descendants de x dans le graphe sont numérotés, et cette numérotation est *conforme*.

Avant de démontrer que cet algorithme permet bien d'obtenir une numérotation conforme, nous en donnons un exemple afin de faciliter la compréhension.

1. sommet de départ: h (choix arbitraire)

pile	
h	
h k	
h k l	
h k l o	
h k l	num(o)=15
h k	num(l)=14
h k m	
h k	num(m)=13
h k n	
h k	num(n)=12
h	num(k)=11
ε	num(h)=10

fin de la première étape; *der_num* = 10

2. sommet de départ: d

pile	
d	
d g	
d g j	
d g	num(j)=9
d	num(g)=8
ε	num(d)=7

fin de la deuxième étape; *der_num* = 7

3. sommet de départ: c

pile	
c	
c e	
c e f	
c e	num(f)=6
c e i	
c e	num(i)=5
c	num(e)=4
ε	num(c)=3

fin de la troisième étape; *der_num* = 3

4. sommet de départ: a

pile	
a	
ε	num(a)=2

fin de la quatrième étape; *der_num* = 2

5. sommet de départ: b

pile	
b	
ε	num(b)=1

fin de la numérotation

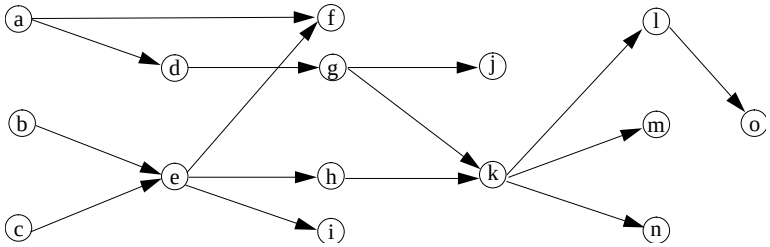


FIG. 7.1 – *Construction d'une numérotation conforme*

Preuve. Il faut montrer que $(x,y) \in \Gamma \Rightarrow num(x) < num(y)$. pour cela, il suffit de montrer que $(x,y) \in \Gamma \Rightarrow y$ sort de la pile *avant* x.

Soit donc $(x,y) \in \Gamma$. Lorsque y passe de l'état *dehors* à l'état *en_attente*, c'est-à-dire lorsque y est empilé, trois cas sont à considérer selon le statut de x:

1. x est *dehors*. Cela veut dire que x n'a pas encore été empilé. Mais, par hypothèse, le graphe est sans circuit, et comme $(x,y) \in \Gamma$, on a nécessairement $(y,x) \notin \Gamma^+$. Donc, x ne sera pas empilé tant que y est dans la pile (les sommets empilés pendant que y est dans la pile sont nécessairement des descendants de y). Il en résulte que y sort de la pile *avant* que x soit empilé, donc a fortiori *avant* que x soit dépilé.
   Dans l'exemple, ce cas se produit notamment pour x=e, y=h.
2. x est *en_attente*. Cela veut dire que x est dans la pile lorsque y est empilé. D'après les propriétés structurelles de la pile, y sera nécessairement dépilé *avant* x.
   Dans l'exemple, ce cas se produit notamment pour x=k, y=l ou m ou n.
3. x est *terminé* (c'est-à-dire déjà numéroté). Ce cas ne peut se produire, car il impliquerait que x ait été empilé, puis dépilé, sans que son successeur y n'ait été visité.

□

Remarques.

1. Si, dans l'algorithme précédent, on avait numéroté les sommets dans l'ordre de leur sortie de pile (et non pas en décroissant) on aurait obtenu une *numérotation conforme inverse* c'est-à-dire une fonction ordinale inverse injective, vérifiant

$(x,y) \in \Gamma \Rightarrow num(x) > num(y)$

2. Une numérotation conforme peut aussi être obtenue pratiquement à partir de la fonction ordinale des points d'entrée, en numérotant arbitrairement les sommets de niveau 1, puis ceux de niveau 2, etc., avec une numérotation strictement croissante.

7.3 Application : fermeture anti-transitive et τ-minimalité

A la section 5.5 nous avons mis en évidence la notion de τ-**minimalité**, tout en faisant remarquer qu'un graphe donné pouvait posséder plusieurs équivalents τ-minimaux.

Par contre, comme va le montrer le thèorème suivant, un graphe sans circuit G possède *un seul* équivalent τ-mimimal, noté $\overset{\vee}{G}$. L'intérêt de $\overset{\vee}{G}$ provient du fait que c'est le seul graphe offrant les mêmes possibilités de cheminement que G, sans qu'aucun arc ne soit redondant.

On précise d'abord la notion de graphe *fortement* anti-transitif :

Définition 7.6 *Un graphe est fortement anti-transitif si* $(\exists u = [x * y] \wedge |u| \geq 2) \Rightarrow (x,y) \notin \Gamma$.

Autrement dit, aucun chemin de longueur supérieure ou égale à 2 ne peut être sous-tendu par un arc.

On a le résultat suivant :

Proposition 7.7 *Soit* $G = (X,\Gamma)$ *un graphe sans circuit. Il existe un seul graphe partiel de* G τ*-minimal* τ*-équivalent à* G *(Déf. 5.13 et 5.14). Ce graphe* $\overset{\vee}{G}$ *est un élément maximal de l'ensemble des graphes partiels de* G *fortement anti-transitifs.*

Démonstration. Soit $\tau(G)$ la classe d'équivalence de G. Si $H \in \tau(G)$, alors H est sans circuit. En effet, si H possédait un circuit passant par x, alors (x,x) serait une boucle de $H^+ (= G^+)$ et G aurait un circuit passant par x.

Supposons alors que $\tau(G)$ ait deux éléments minimaux $G_1 = (X,\Gamma_1)$ et $G_2 = (X,\Gamma_2)$ et soit $(x,y) \in \Gamma_1$ et $(x,y) \notin \Gamma_2$

On a donc $(x,y) \in \Gamma_1^+ = \Gamma_2^+$ d'où :

$$\exists z \in X, z \neq x, z \neq y \ : \ (x,z) \in \Gamma_2^+ \wedge (z,y) \in \Gamma_2^+$$

soit encore :

$$(x,z) \in \Gamma_1^+ \wedge (z,y) \in \Gamma_1^+$$

Mais aucun chemin allant de x à z dans G_1 ne peut passer par y, sans quoi on aurait :

$\big((y,z) \in \Gamma_1^+ \wedge (z,y) \in \Gamma_1^+\big) \Rightarrow (y,y) \in \Gamma_1^+$ donc G_1 aurait un circuit.

De même, aucun chemin allant de z à y dans G_1 ne peut passer par x.

Il existe donc, dans G_1, un chemin allant de x à y, distinct de l'arc (x,y), ce qui contredit la τ-minimalité de G_1. L'hypothèse $(x,y) \in \Gamma_1$ et $(x,y) \notin \Gamma_2$ est donc absurde, d'où $G_1 = G_2 = \overset{\vee}{G}$.

D'autre part on sait que dans une partie finie, l'unicité de l'élément minimal en fait un minimum, c'est à dire un élément comparable à tous les éléments de cette partie. $\overset{\vee}{G}$ est donc un graphe partiel de G.

Enfin, montrons que $\overset{\vee}{G}$ est un élément maximal de l'ensemble des graphes partiels de G, fortement anti-transitifs. On remarque d'abord que $\overset{\vee}{G}$ est le seul graphe fortement anti-transitif de $\tau(G)$. En effet, soit $H = (X,\Sigma)$, $H \in \tau(G)$ et H fortement anti-transitif. Alors : $\forall H' = (X,\Sigma') \prec H$, H' est fortement anti-transitif. Soit $(x,y) \in \Sigma$ et $(x,y) \notin \Sigma'$. Alors $(x,y) \notin \Sigma'^+$ et par conséquent, $H' \notin \tau(G)$ d'où on conclut que H est minimal dans $\tau(G)$, et donc $H = \overset{\vee}{G}$.

Enfin, si G' est un graphe partiel de G, fortement anti-transitif, on a :

$G' \preceq G$ et donc :

(1) $G'^{+} \preceq G^{+}$

Si par ailleurs $\overset{\vee}{G} \preceq G'$ alors $\overset{\vee}{G}^{+} \preceq G'^{+}$ mais comme $\overset{\vee}{G}^{+} = G^{+}$ on en déduit :

(2) $G^{+} \preceq G'^{+}$. De (1) et (2) découle :

$G' \in \tau(G)$ c'est à dire $G' = \overset{\vee}{G}$

□

Nous proposons ci-dessous (texte page 80) un algorithme, adapté de l'algorithme de Roy-Warshall, permettant d'obtenir $\overset{\vee}{G}$. Il consiste à construire la fermeture transitive de G et à repérer les arcs nouveaux introduits par les opérations Θ. On travaille sur deux copies de G, et à l'issue de l'algorithme on vérifie, sur G^{+}, que G était bien sans circuit. (On se place dans le cas où $X = 1..n$).

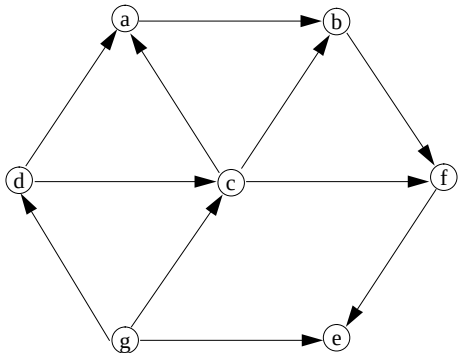


Fig. 7.2 – *Un graphe sans circuit*

Exemple Soit le graphe représenté figure 7.2. Les opérations Θ successives produisent les résultats suivants:

Opération	ajout dans G_2	retrait dans G_1
Θ_a	(d,b)	(c,b)
Θ_b	(a,f),(d,f)	(c,f)
Θ_c	(d,e),(g,a),(g,b),(g,f)	(d,a)
Θ_d		(g,c),(g,e)
Θ_e		
Θ_f	(a,e),(b,e)	(c,e),(c,f)
Θ_g		

D'où la fermeture τ-minimale représentée figure 7.3

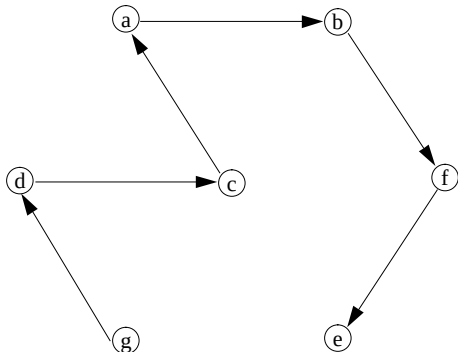


FIG. 7.3 – *Fermeture τ-mininale*

GRAPHE SANS CIRCUIT : τ-MINIMALITE

```
GRAPHE tau-minimalite(GRAPHE G)  c'est
 local  GRAPHE G1, G2 ;
        SOMMET x, y, z; ENS[SOMMET] X;
 début
   G1 ← G ; G2 ← G ; X ← G.lst_som ;
    pour tout  x  de X
       pour tout  y  de X
          si  G2.validarc(y, x)  alors
             pour tout  z  de X
                si  G2validarc(x, z)  alors
                  G1.oter_arc(y,z) ; G2.ajout_arc(y,z)
                fsi
             fpourtout
          fsi
       fpourtout
    fpourtout ;
    si  existe x  de G2.lst_som  telque G2.validarc(x, x)
       alors  --   G a un circuit : pas de fermeture τ-minimale
       sinon Result ← G1
    fsi
 fin
```

7.4 Composantes fortement connexes

7.4.1 Définition

Définition 7.8 *Soit $G = (X,\Gamma)$ un graphe. On considère la relation binaire sur X :*

$$x\mathcal{R}y \equiv (x \in \Gamma^*(y)) \wedge (y \in \Gamma^*(x))$$

C'est une relation d'équivalence dont les classes s'appellent composantes fortement connexes de G (en abrégé : c.f.c.).

Proposition 7.9 *Deux sommets appartiennent à la même c.f.c. si et seulement si ils sont situés sur un même circuit.*

Démonstration. Évident, puisque $(x \in \Gamma^*(y)) \wedge (y \in \Gamma^*(x)) \Leftrightarrow x$ et y sont sur un même circuit. □

Proposition 7.10 *Si deux sommets x et y appartiennent à une même c.f.c. C, alors tout chemin de x à y est entièrement inclus dans C.*

Démonstration. Soit $\mathbf{u}$ un chemin de x à y (par hypothèse, de tels chemins existent), et z un sommet de ce chemin. On a donc : $\mathbf{u} = \mathbf{u}_1 \cdot \mathbf{u}_2$, avec $\mathbf{u}_1$ chemin de x à z, et $\mathbf{u}_2$ chemin de z à y.Par hypothèse, on a aussi : il existe un chemin $\mathbf{u}'$ de y à x. Considérant $\mathbf{u}_2 \cdot \mathbf{u}' \cdot \mathbf{u}_1$, on obtient un circuit de z à z, passant par x et y. Donc z appartient à la même c.f.c. que x et y.

Définition 7.11 *Un graphe est* fortement connexe *s'il possède une seule composante fortement connexe.*

On remarque donc que, si $|X| = n$, un graphe $G \in \mathcal{G}(X)$ possède

- *au plus* n c.f.c.; ce cas extrême est obtenu si et seulement si G est *sans circuit* (chacune c.f.c. est alors réduite à un seul sommet),

- *au moins* une c.f.c. : ce cas extrême est obtenu lorsque le graphe est fortement connexe (tous les sommets sont sur un même circuit, ou encore, chaque couple de sommets est relié par au moins un chemin).

Définition 7.12 *Soit $\mathcal{R}$ une relation d'équivalence sur X. Le graphe réduit selon les classes de $\mathcal{R}$ est défini par :*

$G_{\mathcal{R}} = (X/\mathcal{R},\Gamma_{\mathcal{R}})$, où

$X/\mathcal{R}$ = ensemble des classes de la relation $\mathcal{R}$ (ensemble quotient)
$(C_i,C_j) \in \Gamma_{\mathcal{R}} \Leftrightarrow \exists x \in C_i, \exists y \in C_j \ : \ ((x,y) \in \Gamma) \wedge (C_i \neq C_j)$

Autrement dit, tous les sommets situés dans une même classe sont confondus en un seul sommet (on ne les distingue plus les uns des autres), et un arc relie une classe à une autre chaque fois que, dans le graphe initial, un représentant de la première classe était prédécesseur d'un représentant de la seconde classe. Si on prend pour $\mathcal{R}$ la relation ayant servi à définir les composantes fortement connexes, on obtient le **graphe réduit selon les c.f.c.**.

Proposition 7.13 *Le graphe réduit selon les composantes fortement connexes est sans circuit.*

Démonstration. Si il existait un circuit
$C_{i_1},\ldots,C_{i_k},C_{i_1}$ dans $G_{\mathcal{R}}$, alors il existerait un circuit
$x_{i_1} \ * \ldots * \ x_{i_k} \ * \ x_{i_1}$ dans G, avec
$x_{i_1} \in C_{i_1},\ldots,x_{i_k} \in C_{i_k}$ et ceci impliquerait
$C_{i_1} = \ldots = C_{i_k}$ □

La propriété précédente signifie en particulier que, en suivant un chemin de G, *si on sort d'une c.f.c. on ne peut plus y revenir.*

Définition 7.14 *On appelle composante fortement connexe terminale de G un point de sortie du graphe réduit.*

Autrement dit, si en parcourant un chemin de G on atteint une c.f.c. terminale, *on ne peut plus en sortir.*

Proposition 7.15 *Soit C un sous-ensemble de sommets tel que, $\forall x \in C$, $\forall y \in C$, $(x \in \Gamma^*(y)) \wedge (y \in \Gamma^*(x))$. Si C n'a pas d'arc sortant, alors C est une c.f.c. terminale.*

Démonstration. Clairement, tous les sommets de C sont dans une même c.f.c. C'. On a donc $C \subseteq C'$. Supposons que $C \neq C'$. Cela signifie qu'il existe un sommet z, tel que $z \in C'$ et $z \notin C$. Puisque $x \in C$ on a $x \in C'$ et donc il existe un chemin de x à z. Comme $x \in C$ et $z \notin C$, ce chemin comporte un arc sortant de C, ce qui contredit l'hypothèse. Donc, $C = C'$, c'est-à-dire C est une c.f.c. Comme elle n'a pas d'arc sortant, elle est terminale. □

Dans les sections suivantes, on donne trois algorithmes permettant de déterminer les c.f.c. d'un graphe. Puis nous verrons quelles applications on peut tirer de cette décomposition.

7.4.2 Algorithme de FOULKES

Cet algorithme s'applique à la fermeture transitive G^+ de G. (Il suppose donc celle-ci déjà déterminée). Il est basé directement sur la définition, qui peut s'écrire encore :

$$x\mathcal{R}y \Leftrightarrow (x = y) \vee \big((x,y) \in \Gamma^+ \wedge (y,x) \in \Gamma^+\big)$$

7.4.2.1 Principe

Il est très simple, et s'appuie sur trois constatations :

- un sommet appartient à sa propre classe
- un sommet ne peut appartenir qu'à une seule classe
- si un sommet n'est pas son propre descendant, il est seul dans sa classe (il n'est pas situé sur un circuit).

Les sommets sont examinés séquentiellement; soit x le sommet courant. Trois cas sont à distinguer quant à la situation de x:

1. x a déjà été rangé comme élément de la classe d'un sommet y examiné avant lui (c'est-à-dire $y \leq x$). Dans ce cas, $cl(x) = cl(y)$ (déjà calculée)
2. x n'est pas encore classé et n'est pas son propre descendant, c'est-à-dire $(x,x) \notin \Gamma^+$. Dans ce cas, $cl(x) = \{x\}$.
3. x n'est pas encore classé, et $(x,x) \in \Gamma^+$. Dans ce cas, les éléments de $cl(x)$, outre x lui-même, sont à rechercher parmi les y tels que $y > x$; en effet, s'il existait $y \leq x$ avec $y \in cl(x)$, x aurait été classé lors de l'examen de y, antérieur à celui de x.

L'analyse de l'algorithme en découle immédiatement :

Invariant $(1 \leq x \leq n) \wedge$
$PRIS = \{y \mid cl(y) \text{ est calculé } \} \wedge$
$\{1 \,..\, x-1\} \subseteq PRIS$

Condition d'arrêt $x = n$

Progression

cas $x \in PRIS \quad \Rightarrow$
  $cl(x) \leftarrow cl(y)$
$(x \notin PRIS) \wedge ((x,x) \notin \Gamma^+) \quad \Rightarrow$
  $cl(x) \leftarrow \{x\}$
$(x \notin PRIS) \wedge ((x,x) \in \Gamma^+) \quad \Rightarrow$
  $cl(x) \leftarrow \{x\}$;
  pourtout y **de** $[x+1..n]$ **faire**
    si $((x,y) \in \Gamma^+) \wedge ((y,x) \in \Gamma^+)$ **alors**
      $cl(x) \leftarrow cl(x) \cup \{y\}$;
    fsi
  fpour
fcas ;
$PRIS \leftarrow PRIS \cup cl(x)$;

$x \leftarrow x + 1$

Valeurs initiales $x \leftarrow 1$; $PRIS \leftarrow \emptyset$

7.4.2.2 Texte de l'algorithme

Il est donné page 84

ALGORITHME DE FOULKES

```
ENS[SOMMET] SOMMET::cl : ;

foulkes(GRAPHE G+) c'est
 local ENS[SOMMET] PRIS;
       SOMMET x, y;
 début
   PRIS ← ∅ ;
    pour x depuis 1 jqa n faire
      si x ∉ PRIS alors
        x.cl ←{x} ;
         si G+.validarc(x, x) alors
           --   x est sur un circuit ; recherche des sommets de sa classe
            pour y depuis x + 1 jqa n faire
               si G+.validarc(x, y) et G+.validarc(y, x)
                  alors x.cl ← x.cl ∪{y}
               fsi
            fpour ;
            pour tout y de x.cl faire y.cl ← x.cl fpourtout ;
           PRIS ← PRIS ∪ x.cl
         fsi
      fsi
    fpour
 fin
```

7.4.2.3 Complexité

- construction de la fermeture transitive : au moins $O(n^3)$
- le nombre d'accès *validarc* est $\leq n^2$

7.4.3 Algorithme descendants-ascendants

7.4.3.1 Principe

Il opère directement sur le graphe donné G, et utilise les algorithmes de recherche des ascendants et des descendants d'un sommet donné.

Si on désigne par $asc(x)$ (resp. $desc(x)$) la fonction délivrant l'ensemble des ascendants (resp. descendants) de x, on peut écrire :

$$cl(x) = asc(x) \cap desc(x)$$

En fait, lorsqu'on calcule la c.f.c. d'un sommet x, il n'est pas nécessaire de calculer l'ensemble de tous les ascendants (resp. descendants) de x. En effet, comme le montre la proposition suivante, lors de l'exploration de la descendance de x (calcul de $desc(x)$), si on rencontre un sommet y déjà classé, alors aucun descendant de y ne peut être dans la classe de x. (Même chose en remplaçant *descendant* par *ascendant*).

Par conséquent, dans la procédure de recherche des descendants de x (resp. ascendants) on peut se limiter au sous-graphe engendré par les sommets de G non encore classés.

Proposition 7.16 *Soit* $y \in desc(x)$*; si* $y \notin cl(x)$ *alors* $(desc(y) \cap cl(x) = \emptyset)$
Soit $y \in asc(x)$*; si* $y \notin cl(x)$ *alors* $(asc(y) \cap cl(x) = \emptyset)$

Démonstration. Soit $y \in \Gamma^*(x) \wedge y \notin cl(x)$; supposons que $z \in \Gamma^*(y) \cap cl(x)$. On a alors :

$$z \in \Gamma^*(y) \wedge x \in \Gamma^*(z) \Rightarrow x \in \Gamma^*(y)$$

et comme $y \in \Gamma^*(x)$ ceci contredit $y \notin cl(x)$.

On démontre de même la deuxième partie de la proposition (ascendants) □

Ici, on donne une version récursive du calcul des ascendants, mais les autres versions (largeur ou profondeur itératives) conviendraient.

7.4.3.2 Texte de l'algorithme

Le texte est donné page 86

7.4.3.3 Complexité

Chaque recherche de *descendants-non-pris* et *ascendants-non-pris* nécessite exactement
$\text{card}(\Gamma^*(x) \cap (X \setminus PRIS)) \leq n - \#(\text{sommets classés})$ accès *lst_succ*
$\text{card}\big((\Gamma^{-1})^*(x) \cap (X \setminus PRIS)\big) \leq n - \#(\text{sommets classés})$ accès *lst_pred*

Or à l'examen de x, il y a au moins $x - 1$ sommets classés, et donc la complexité est, au plus :

$\frac{n(n+1)}{2}$ accès *lst_succ*, autant d'accès *lst_pred*, soit

$O(\frac{n^2}{2})$ accès *lst_succ*, *lst_pred*

COMPOSANTES FORTEMENT CONNEXES : ASCENDANTS-DESCENDANTS

```
ENS[SOMMET] SOMMET::cl : ;
ascendants-descendants(GRAPHE G)  c'est
   local  ENS[SOMMET] D, A, PRIS;
 début
  --   initialisation
   PRIS ← ∅ ;
  --   classement
   pour tout x  de G.lst_som
     si x ∉ PRIS  alors
      A ← asc_non_pris(x) ; D ← desc_non_pris(x) ;
      x.cl ← A ∩ D ;
       pour tout y  de x.cl  faire y.cl ← x.cl  fpourtout ;
      PRIS ← PRIS ∪ x.cl
     fsi
   fpourtout
 fin

ENS[SOMMET] S;
ENS[SOMMET] asc_non_pris (SOMMET x)  c'est
   début
     S ← ∅ ; anp(x) ; Result ← S
   fin;

 anp(SOMMET y)  c'est
     local ENS[SOMMET] E;
           SOMMET z;
   début
     S ← S ∪{y} ; E ← G.lst_pred(y) ;
     E ← E \ PRIS ;
     pour tout z  de E  faire
        si z ∉ S  alors anp(z)  fsi
     fpourtout
   fin;;

ENS[SOMMET] desc_non_pris (SOMMET x)  c'est ;
   --   identique à asc_non_pris en remplaçant lst_pred par lst_succ
```

7.4.4 Algorithme de TARJAN

Cet algorithme fournit non seulement la décomposition du graphe en composantes fortement connexes, mais de plus il délivre les composantes avec une numérotation conforme (par rapport au graphe réduit selon ses composantes fortement connexes).

7.4.4.1 Principe de l'algorithme de TARJAN

Cet algorithme généralise aux graphes quelconques l'algorithme de numérotation conforme des graphes sans circuit, basé sur l'exploration en profondeur, vu en 6.3.2. Il est basé sur l'observation suivante : si le graphe était sans circuit, les c.f.c. se confondraient avec les sommets, et l'algorithme de numérotation conforme des sommets répond à la question. Si le graphe possède des circuits, on se ramène à un graphe sans circuit en considérant le graphe réduit selon les c.f.c., ce qui revient à traiter des sous-ensembles de sommets situés sur un même circuit comme un seul "macro-sommet". Or, lors d'une exploration en profondeur, on est capable de déterminer des "segments" de pile dont tous les éléments appartiennent à un même circuit : chaque fois que l'on visite un arc allant du sommet de pile y à un sommet z appartenant à la pile, tous les sommets situés dans la pile depuis z jusqu'à y sont sur un même circuit (donc appartiennent à une même c.f.c.). De plus, d'après la proposition 7.15, on saura qu'un tel ensemble de sommets constitue une c.f.c. (terminale dans le sous-graphe engendré par les sommets non encore classés) dès que tous les arcs sortants de ce sous-ensemble sont visités.

Pour mettre en oeuvre cette idée, il suffit de remplacer les sommets par des sous-ensembles de sommets dans l'algorithme de numérotation conforme. On va donc travailler sur un graphe qui, à chaque étape, sera une réduction du graphe initial selon des sous-ensembles de sommets reconnus comme faisant partie d'une même c.f.c.

Invariant

- X est partitionné en $C_1, \ldots, C_k$:

$$X = C_1 \cup \ldots C_k, \text{ et } \forall i \neq j \ : \ C_i \cap C_j = \emptyset$$

- Chaque sous-ensemble C est dans l'un des trois états suivants:
  - *dehors*, et dans ce cas, $|C| = 1$, et le sommet constituant C n'a pas été visité;
  - *en_attente* (dans la pile): $\forall y \in C$, y est visité et au moins un arc sortant de C n'est pas visité;
  - *terminé*: $\forall y \in C$, y est visité et tous les arcs sortants de C ont été visités. Dans ce cas, C est une c.f.c.

Arrêt Pile vide (pas de sous-ensemble en attente). Ceux qui sont terminés constituent les c.f.c. déjà déterminées. S'il reste des sous-ensembles dehors, on recommence une exploration à partir de l'un des ces sous-ensembles. Sinon, toutes les c.f.c. ont été obtenues.

Initialement Soit $\{x\}$ le sous-ensemble (à un sommet) dont est issu l'exploration

- $X = \{1\} \cup \{2\} \cup \ldots \{n\}$ (en supposant que les sommets sont numérotés de 1 à n.

- num=numéro initial de c.f.c
- $\{x\}$ est empilé si $\Gamma(x) \neq \emptyset$, terminé sinon (dans ce cas, c'est la c.f.c. numéro num).
- Tous les autres sous-ensembles sont dehors.
- Aucun arc n'a été visité.

Cette situation initiale vérifie bien l'invariant. En particulier, si $\Gamma(x) = \emptyset$, $\{x\} = cfc(x)$ puisque x est alors point de sortie.

Progression Soit C le sommet de pile.

- Si C n'a pas d'arc sortant non visité, alors
  - *A.depiler* ; mettre C dans T, ensemble des sommets terminés ;
  - $num \leftarrow num - 1$ (C constitue une c.f.c, on lui attribue le numéro num)
- Sinon, soit (y,z) un arc sortant de C, non visité, et C_z le sous-ensemble contenant z (par construction, il y en a un et un seul, cf. invariant).
  - l'arc (y,z) devient visité;
    - Si $C_z \in D$ (dans ce cas, $C_z = \{z\}$), alors $A.empiler(C_z)$; enlever C_z de D
    - Si $C_z \in A$, fusionner dans C tous les sous-ensembles qui sont dans la pile depuis C_z jusqu'à C (tous les sommets de ces sous-ensembles sont sur un même circuit)
    - Si $C_z \in T$, pas d'action

Nous ne démontrerons pas formellement que la progression maintient l'invariant, mais le lecteur peut aisément s'en convaincre.

7.4.4.2 Exemple

Nous allons d'abord montrer sur l'exemple de la figure 7.4 le fonctionnement de l'algorithme.

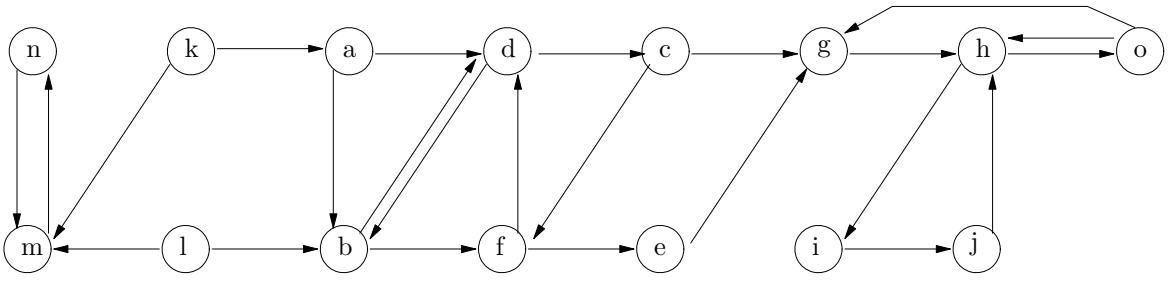


FIG. 7.4 – *Exemple pour l' algorithme de TARJAN*

PILE	
$\{a\}$	
$\{a\}$ $\{b\}$	
$\{a\}$ $\{b\}$ $\{d\}$	
$\{a\}$ $\{b\}$ $\{d\}$ (b)	Circuit détecté vers b
$\{a\}$ $\{b, d\}$	
$\{a\}$ $\{b, d\}$ $\{c\}$	
$\{a\}$ $\{b, d\}$ $\{c\}$ $\{g\}$	
$\{a\}$ $\{b, d\}$ $\{c\}$ $\{g\}$ $\{h\}$	
$\{a\}$ $\{b, d\}$ $\{c\}$ $\{g\}$ $\{h\}$ $\{o\}$	
$\{a\}$ $\{b, d\}$ $\{c\}$ $\{g\}$ $\{h\}$ $\{o\}$ (h)	Circuit détecté vers h
$\{a\}$ $\{b, d\}$ $\{c\}$ $\{g\}$ $\{h, o\}$	
$\{a\}$ $\{b, d\}$ $\{c\}$ $\{g\}$ $\{h, o\}$ (g)	Circuit détecté vers g
$\{a\}$ $\{b, d\}$ $\{c\}$ $\{g, h, o\}$	Tous les arcs issus de o sont visités: on "remonte" à h
$\{a\}$ $\{b, d\}$ $\{c\}$ $\{g, h, o\}$ $\{i\}$	
$\{a\}$ $\{b, d\}$ $\{c\}$ $\{g, h, o\}$ $\{i\}$ $\{j\}$	
$\{a\}$ $\{b, d\}$ $\{c\}$ $\{g, h, o\}$ $\{i\}$ $\{j\}$ (h)	Circuit détecté vers h
$\{a\}$ $\{b, d\}$ $\{c\}$ $\{g, h, o, i, j\}$	Tous les arcs issus de j sont visités: on "remonte" à i puis à h puis à g Tous les arcs sortant de {**g**,**h**,**o**,**i**,**j**} sont visités: on dépile ce sous-ensemble, qui est la c.f.c. $C_{15} = \{g,h,o,i,j\}$
$\{a\}$ $\{b, d\}$ $\{c\}$	
$\{a\}$ $\{b, d\}$ $\{c\}$ $\{f\}$	
$\{a\}$ $\{b, d\}$ $\{c\}$ $\{f\}$ $\{e\}$	
$\{a\}$ $\{b, d\}$ $\{c\}$ $\{f\}$ $\{e\}$ (g)	g est déjà classé; tous les arcs sortant de {**e**} sont visités: on dépile cette c.f.c $C_{14} = \{e\}$
$\{a\}$ $\{b, d\}$ $\{c\}$ $\{f\}$ (d)	Circuit détecté vers d
$\{a\}$ $\{b, d, c, f\}$	Tous les arcs issus de f sont visités: on "remonte" à c puis à d puis à b
$\{a\}$ $\{b, d, c, f\}$ (f)	f appartient déjà au sommet d e pile: pas de nouveau circuit.
$\{a\}$ $\{b, d, c, f\}$	Tous les arcs sortants de {**b**,**d**,**c**,**f**} sont visités : on dépile cette c.f.c $C_{13} = \{b,d,c,f\}$
$\{a\}$	
$\{a\}$ (d)	d est déjà classé; tous les arcs sortant de {**a**} sont visités: on dépile cette c.f.c $C_{12} = \{a\}$
ε	Pile vide : toute la descendance de f a a été classée.

	Repartir d'un sommet non classé.
$\{k\}$	
$\{k\}$ (a)	a est déjà classé.
$\{k\}$ $\{m\}$	
$\{k\}$ $\{m\}$ $\{n\}$	
$\{k\}$ $\{m\}$ $\{n\}$ (m)	Circuit détecté vers m
$\{k\}$ $\{m,\ n\}$	Tous les arcs issus de n sont visités: on "remonte" à m
	Tous les arcs sortant de {**m**,**n**} sont visités:
	on dépile cette c.f.c $C_{11} = \{m,n\}$
$\{k\}$	Tous les arcs sortant de {**k**} sont visités:
	on dépile cette c.f.c $C_{10} = \{k\}$
ε	
$\{\ell\}$	
$\{\ell\}$ (b)	b est déjà classé.
$\{\ell\}$ (m)	m est déjà classé.
	Tous les arcs sortant de {**l**} sont visités:
	on dépile cette c.f.c $C_9 = \{\ell\}$
ε	Pile vide **et tous les sommets sont classés : EXÉCUTION TERMINÉE**.

On a obtenu 7 c.f.c, numérotées de 9 à 15. On effectue une translation de numéros, en retranchant 8, pour avoir une numérotation contiguë de 1 à 7. On obtient alors le graphe réduit (figure 7.5) qui montre qu'on a bien une *numérotation conforme*

$(C_i,C_j) \in \Gamma_{\mathcal{R}} \Rightarrow i < j$

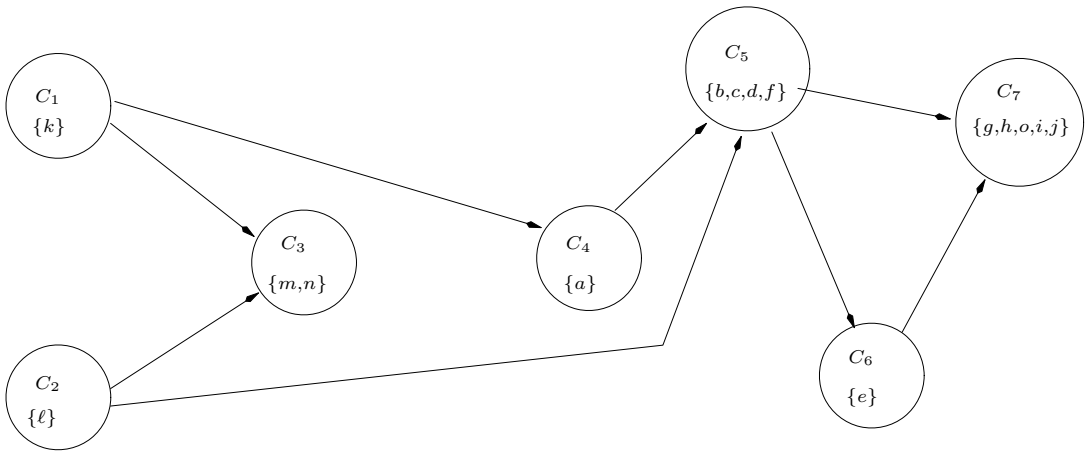


FIG. 7.5 – *Graphe réduit de la figure 7.4*

7.4.4.3 Mise en œuvre et texte de l'algorithme

Pour réaliser l'algorithme ci-dessus, en plus de la pile, on attache à chaque sommet $y \in X$ les attributs suivants :

ENT SOMMET::pospile	définit la position de *y* dans la pile (non défini si *y* n'a pas encore été atteint).
ENT SOMMET::marque	- 0 si *y* n'a pas encore été atteint, - *y.pospile* lorsque *y* est empilé, - puis, éventuellement diminuée lorsque *y* est repéré comme appartenant à la c.f.c. d'un sommet empilé avant lui.
SOMMET SOMMET::pred	père du sommet *y* dans l'arborescence d'exploration.
ENS[SOMMET] SOMMET::S	ensemble des successeurs de *y*, dans le sous- graphe partiel des sommets non classés, qui ne sont pas encore empilés (comme dans l'algorithme d'énumération des chemins élémentaires).
ENT SOMMET::ncl	numéro de la c.f.c. à laquelle appartient *y* (indéfini tant que *y* n'est pas classé).

Le texte de l'algorithme est donné page 92 et suivantes.

ALGORITHME DE TARJAN

```
tarjan(GRAPHE G) c'est
  local PILE[SOMMET] L;
        ENS[SOMMET] NC;  --   sommets non classés
        SOMMET finchemin, z ;
        ENT compt ;  --   compteur du numéro de la cfc courante

 début
    depuis NC ← G.lst_som ; L.creer ; compt ← 0 ;
       pourtout y de NC y.marque ← 0 fpourtout ;
    jusqua NC = ∅
    faire
      finchemin ← élément(NC) ; L.empiler(finchemin) ;
      finchemin.S ← G.lst_succ(finchemin) ;
      finchemin.S ← finchemin.S ∩ NC ;
      finchemin.marque ← L.taille ;
      finchemin.pospile ← L.taille ;
       depuis
       jusqua L.vide
       faire
          si finchemin.S ≠ ∅
             alors z ← élément(finchemin.S) ;
                  si z.marque = 0
                    alors L.empiler(z) ;
                         z.S ← G.lst_succ(z) ;
                         z.S ← z.S ∩ NC ;
                         z.marque ← L.taille ;
                         z.pospile ← L.taille ;
                         z.pred ← finchemin ;
                         finchemin ← z
                    sinon si finchemin.marque > z.marque
                                alors finchemin.marque ← z.marque
                            fsi
                 fsi
            sinon si finchemin.marque ≠ finchemin.pospile
               alors z ← finchemin ;
                    finchemin ← z.pred ;
                    finchemin.marque ← z.marque
               sinon --   sortie d'une nouvelle cfc
                  depuis compt ← compt + 1 ;
                  jusqua L.taille < finchemin.pospile
```

```
                    faire
                      z ← L.sommetpile ;
                      z.ncl ← compt ; NC ← NC \ {z} ;
                      L.dépiler
                    fait ;
                  finchemin ← finchemin.pred
                fsi
            fsi
        fait
    fait
fin
```

7.4.4.4 Complexité

Elle est du même ordre que l'algorithme de recherche *profondeur d'abord*, puisque chaque sommet de `X` est empilé exactement une fois, et l'accès `lst_succ` est appelé une fois pour chaque sommet. C'est donc un algorithme en $0(n)$ accès `lst_succ`.

7.4.5 Applications

7.4.5.1 Recherche d'un graphe minimal dans $\tau(G)$

Nous avons précédemment montré (§7.3) que, si G est sans circuit, il possède une fermeture fortement anti-transitive $\overset{\vee}{G}$ (élément minimum de $\tau(G)$). Il n'en est pas nécessairement de même lorsque G possède des circuits. On peut cependant mettre en évidence un graphe minimal dans $\tau(G)$, c'est à dire un graphe G' tel que :

- $G'^{+} = G^{+}$ (mêmes possibilités de cheminement)
- la suppression d'un arc quelconque de G' supprime une possibilité de cheminement.

Pour cela, on peut procéder comme suit :

1. Déterminer les c.f.c. de G, et construire le graphe réduit $G_{\mathcal{R}}$.
2. Déterminer la fermeture anti-transitive $\overset{\vee}{G}_{\mathcal{R}}$. Tout arc (C_i,C_j) ainsi supprimé dans $G_{\mathcal{R}}$ entraîne la suppression de tous les arcs (x,y) de G tels que $x \in C_i$ et $y \in C_j$.
3. Pour les couples $(C_i,C_j) \in \overset{\vee}{G}_{\mathcal{R}}$, ne conserver qu'un seul arc (x,y) dans G, tel que $x \in C_i$ et $y \in C_j$.
4. A l'intérieur de chaque c.f.c., on ordonne arbitrairement les sommets, ce qui définit un ensemble d'arcs formant un circuit hamiltonien (passant une fois et une seule par chaque sommet).

Le lecteur vérifiera qu'un graphe G' ainsi obtenu vérifie bien les propriétés voulues. Remarquer que G' n'est pas nécessairement un graphe partiel de G (à cause du point 4). Si l'on souhaite en outre que $G' \leq G$, le traitement à effectuer à l'intérieur de chaque c.f.c. est heuristique, et il est difficile d'assurer la τ-minimalité. (Il faut obtenir un circuit passant par tous les sommets, en n'empruntant que des arcs de G).

Exemple voir figures 7.6 et 7.7

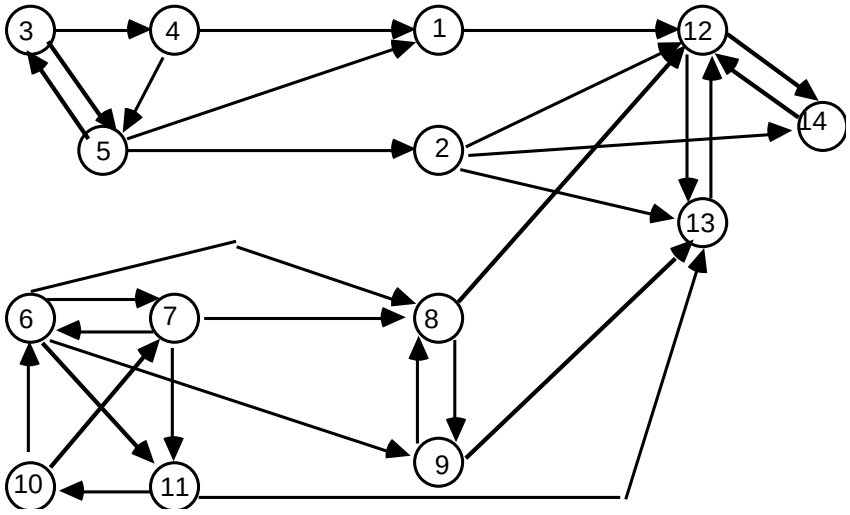


FIG. 7.6 – *Graphe donné*

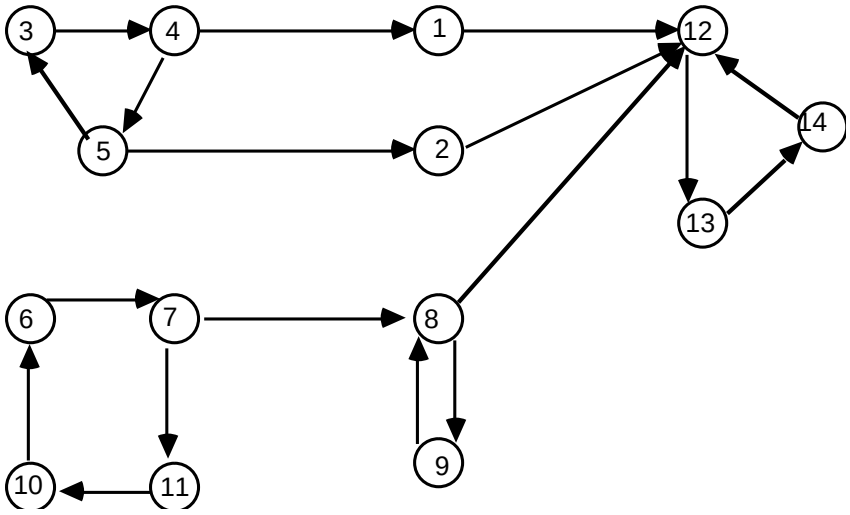


FIG. 7.7 – *Graphe G', minimal dans $\tau(G)$*

7.4.5.2 Calcul rapide de la fermeture transitive

Nous avons vu (§5.4) que l'algorithme de ROY-WARSHALL calcule la fermeture transitive d'un graphe quelconque en $O(n^3)$. Dans le cas où G est sans circuit, l'utilisation d'une numérotation conforme (ou conforme inverse) réduit la complexité ; l'algorithme simplifié est donné page 95.

GRAPHE SANS CIRCUIT : ROY-WARSHALL SIMPLIFIE

```
GRAPHE roy_warshall_conforme(GRAPHE G) c'est
   local SOMMET x, y, z
début
  Result ← G ;
   pour x depuis 2 jqa n-1
      pour y depuis 1 jqa x-1
         si Result.validarc(y, x) alors
            pour z depuis x+1 jqa n
               si Result.validarc(x, z) alors Result.ajoutarc(y, z) fsi
            fpour
         fsi
      fpour
   fpour
fin
```

Le nombre maximum d'opérations est alors :

$$\sum_{x=2}^{n-1}(x-1)(n-x) = \frac{n(n-1)(n-2)}{6} \simeq \frac{n^3}{6}$$

en termes d'accès *validarc, ajoutarc*

Considérons alors le procédé suivant :

1. Appliquer l'algorithme de TARJAN pour obtenir le graphe réduit $G_{\mathcal{R}}$, avec une numérotation conforme inverse .
2. Appliquer l'algorithme de ROY-WARSHALL amélioré au graphe $G_{\mathcal{R}}$
3. Pour tout couple $(C_i,C_j) \in G^+_{\mathcal{R}}$ tel que $card(C_i) > 1$ ou $card(C_j) > 1$ rajouter à G **tous** les arcs (x,y) avec $x \in C_i, y \in C_j$
4. A l'intérieur de chaque c.f.c., créer **tous** les arcs (x,y).

S'il y a p composantes $C_1,C_2,\ldots,C_p$, de cardinal respectif $k_1,k_2,\ldots,k_p$, et si on désigne par m le nombre d'arcs de G, la complexité est en :

- $O(n)$ accès *lst_succ* pour le 1)
- $\frac{p^3}{6}$ pour le 2)
- au plus

$$\left(\frac{n}{p}\right)^2 \sum_{i=1}^{p-1}(p-i) = \frac{n^2(p-1)}{2p}$$

pour le 3)

[maximum obtenu lorsque $k_1 = k_2 = \ldots = k_p = \frac{n}{p}$]

– $k_1^2 + k_2^2 + \ldots + k_p^2 \leq n^2$ pour le 4)

On peut observer que, lorsque $p \to 1$ (graphe *fortement connexe*), la complexité est en $O(n^2)$ (opération 4) prépondérante), et lorsque $p \to n$ (graphe *sans circuit*) c'est l'opération 2) qui domine, et la complexité est alors en $O(\frac{n^3}{6})$.

Chapitre 8

Chemins de valeur optimale

8.1 Définitions et problèmes posés

Nous avons vu au chapitre 2 que les arcs d'un graphe peuvent être munis d'attributs de différents types: pour un graphe non valué, ces attributs sont de type *booléen* (l'arc existe ou n'existe pas), pour un graphe modélisant des distances, ce seront des attributs de type *numérique*, etc. Les attributs peuvent être *statiques* (donnés avec le graphe) ou *dynamiques* (calculés lors de l'exécution d'un algorithme: cf. par exemple le chapitre 10 sur les flots). Dans ce chapitre et les suivants, nous allons manipuler des graphes valués sur les arcs, c'est-à-dire des graphes dont un attribut d'arc est précisé dans la définition du graphe, avec sa valeur (attribut statique).

Définition 8.1 *On appelle graphe valué un doublet $G = (X,v)$, où v est une fonction définie sur $X \times X$ et à valeurs dans un ensemble E.*

Nous supposerons que l'ensemble E possède une loi de composition $\otimes$, associative et munie d'un élément neutre e et d'un élément absorbant α :

$$\begin{array}{lllllll} \forall w \in E: & w \otimes e & = & e \otimes w & = & w \\ \forall w \in E: & w \otimes \alpha & = & \alpha \otimes w & = & \alpha \\ & w_1 \otimes w_2 & = & \alpha & \Rightarrow & w_1 = \alpha \vee w_2 = \alpha \end{array}$$

L'élément absorbant α permet de caractériser l'ensemble Γ des arcs de la manière suivante: $(x,y) \in \Gamma \Leftrightarrow v(x,y) \neq \alpha$

Exemples:

	E	$\otimes$	e	α	
1)	$\mathcal{B}$={**vrai, faux**}	$\wedge$	**vrai**	**faux**	(valeurs d'existence)
2)	$\mathbb{R}$	$+$	0	∞	(valeurs additives)
3)	[0 , 1]	$\times$	1	0	(probabilités)
4)	$\mathbb{N} \cup \{\infty\}$	MIN	∞	0	(capacités)

L'associativité de la loi $\otimes$ permet de prolonger v à l'ensemble des suites de sommets de longueur ≥ 2:

$$v[x_1, \ldots, x_p] = \bigotimes_{j=1 \ldots p-1} v(x_j, x_{j+1})$$

De plus, l'élément α caractérise les chemins de la manière suivante:

Proposition 8.2 *Soit* $[x_1,\ldots,x_p]$ $(p \geq 2)$ *une suite de sommets de* X. *Cette suite définit un chemin de* G *si,et seulement si:*

$$v[x_1,\ldots,x_p] \neq \alpha$$

Démonstration

$$\begin{aligned} v[x_1,\ldots,x_p] = \quad & \bigotimes_{j=1\ldots p-1} v(x_j,x_{j+1}) \neq \alpha \Leftrightarrow \\ & \forall j,\ 1 \neq j \neq p-1 \Rightarrow (x_j,x_{j+1}) \in \Gamma \Leftrightarrow \\ & [x_1,\ldots,x_p] \text{ est un chemin de } G \end{aligned}$$

□

Le but de ce chapitre étant l'étude de chemins de valeur optimale, il nous faut définir un critère de comparaison sur les valeurs des chemins. Autrement dit, étant donné deux sommets x et y et deux chemins reliant x à y, lequel des deux est préférable? Pour cela, l'ensemble E doit aussi être muni d'une relation d'ordre totale, notée $\preceq$, telle que:

i) $\preceq$est compatible avec $\otimes$, c'est-à-dire:

$$\begin{aligned} w_1 \preceq w_2 \quad \Rightarrow \quad & w_1 \otimes w_3 \preceq w_2 \otimes w_3 \wedge \\ & w_3 \otimes w_1 \preceq w_3 \otimes w_2 \end{aligned}$$

ii) α est *maximum* pour $\preceq$:

$$\forall w \in E :\ w \preceq \alpha$$

Soit alors deux suites $\mathbf{u_1} = [x,y_1,\ldots,y_k,y]$, $\mathbf{u_2} = [x,z_1,\ldots,z_p,y]$. On dira que $\mathbf{u_1}$ est *meilleur que* $\mathbf{u_2}$ si, et seulement si: $v(\mathbf{u_1}) \preceq v(\mathbf{u_2})$.

En particulier, une suite définissant un chemin sera toujours préférée à une suite ne définissant pas de chemin, grâce à la propriété ii) ci-dessus.

Exemples de relations d'ordre:

E	$\otimes$	α	$\preceq$	
$\mathcal{B}$=**{vrai, faux}**	$\wedge$	**faux**	**vrai**<**faux**	existence
$\mathbb{R}$	$+$	∞	$\leq$	valeurs additives minimales
[0,1]	$\times$	0	$\geq$	probabilités maximales
$\mathbb{N} \cup \{\infty\}$	MIN	0	$\geq$	capacités maximales

Ces définitions étant données, nous posons les problèmes:

Problème 8.1 *(x-optimal-y) Soit* x,y *deux sommets donnés de* $G = (X,\Gamma,v)$. *Déterminer un chemin* $\mathbf{u} = (x * y)$ *de* G, *de* **valeur optimale** *parmi tous les chemins de* x *à* y *dans* G.

Problème 8.2 *(x-optimal ou optimal-y) Il s'agit du problème x-optimal-y pour tout* $y \in X$ *(resp. x-optimal-y pour tout* $x \in X$*) : chemins de valeur optimale issus de* x *(resp. aboutissant à* y*)*

Problème 8.3 *(∗-optimal-∗) Il s'agit du problème x-optimal pour tout $x \in X$ (chemins de valeur optimale entre tous les couples de sommets).*

L'optimalité est à prendre, bien sûr, au sens de la relation d'ordre sur E.

Dans ce qui suit, et jusqu'au paragraphe 8.8, nous prendrons :

$$E \equiv \mathbb{R}; \otimes \equiv +; \preceq \equiv \leq$$

c'est-à-dire le cas des *valeurs additives minimales*. Ceci n'est pas une restriction car nous verrons au §8.8 comment adapter les résultats et algorithmes aux autres cas, sous réserve d'une structure adéquate de $(E, \otimes, \preceq)$

Nous étudions successivement les problèmes de type 8.2 (et 8.1 comme cas particulier) avec plusieurs algorithmes, puis les problèmes de type 8.3 dans le cadre de la transformation d'algorithmes.

8.2 Chemins de valeur additive minimale issus d'un sommet donné

Nous supposons, dans cette section, que les sommets sont numérotés de 1 à n, et que le sommet de départ est le sommet de numéro 1 (quitte à renuméroter).

8.2.1 Existence

Proposition 8.3 *Le problème 1-minimal admet des solutions si et seulement si le sous-graphe $G_1 = (\Gamma^*(1), \Gamma_1)$ ne contient pas de circuit de valeur < 0.*

Démonstration. Soit $c = [x * x]$ un circuit de valeur $\rho < 0$ dans G_1. Comme $x \in \Gamma^*(1)$, $\exists \gamma = [1 * x]$ de valeur ω; par suite, $\gamma.c$ a pour valeur $\omega + \rho$, $\gamma.c.c$ a pour valeur $\omega + 2\rho$, ..., $\forall k \in \mathbb{N}$: $\gamma.c^k$ a pour valeur $\omega + k\rho$ d'où :

$$\lim_{k \to \infty} (\omega + k\rho)] = -\infty$$

Donc la valeur des chemins de 1 à x n'est pas minorée, et il en est de même pour tous les descendants de x.

Réciproquement, si G_1 n'a pas de circuit de valeur < 0, alors : si $\mathbf{u}$ est un chemin de 1 à x et $\mathbf{u}^e$ un chemin élémentaire, inclus dans $\mathbf{u}$, de 1 à x, on a : $v(\mathbf{u}) \geq v(\mathbf{u}^e) \geq \min v(\mathbf{u}^e)$ (où $\min v(\mathbf{u}^e)$ dénote le minimum des valeurs des chemins élémentaires). Comme l'ensemble des chemins élémentaires est fini, $\min v(\mathbf{u}^e)$ est défini et de plus $\min v(\mathbf{u}) \geq \min v(\mathbf{u}^e)$ ((où $\min v(\mathbf{u})$ dénote le minimum des valeurs des chemins). D'autre part, un chemin élémentaire est un chemin,

et donc $\min v(\mathbf{u}) = \min v(\mathbf{u}^e)$

Définition 8.4 *Un circuit de valeur < 0 s'appelle* **circuit absorbant** *(sous entendu pour les valeurs additives minimales).*

8.2.2 Caractérisation

Nous donnons maintenant un théorème de caractérisation des valeurs minimales, dans le cas où ces valeurs existent. On note $\mathcal{C}_{1i}$ l'ensemble de tous les chemins de 1 à i. On considère les attributs de sommets suivants :

$\forall i \in X$: $\lambda(i) = \min_{u \in \mathcal{C}_{1i}} v(u)$, avec $\lambda(i) = +\infty$ si $\mathcal{C}_{1i} = \emptyset$ (valeur minimale des chemins de 1 à i).

Si $\lambda(i) < +\infty$, $pred(i)$ = prédécesseur de i sur un chemin élémentaire de valeur optimale de 1 à i. ($pred(1)$ n'est pas défini). En abrégé, $pred(i)$ sera désigné comme *prédécesseur optimal* de i.

Le couple ($\lambda(i)$,$pred(i)$) constitue les attributs minimaux (relativement au problème posé) du sommet i.

On montre d'abord que les attributs $\lambda(i)$ ($i = 1, \ldots, n$) sont solutions d'un système d'inéquations et d'un système d'équations.

Proposition 8.5 *Les attributs $\lambda(i)$ ($i = 1, \ldots, n$) sont solution du système: $\forall i \ \forall j$: $x_i \leq x_j + v(j,i)$*

Démonstration Il y a trois cas à considérer :

1. $(j,i) \notin \Gamma$. Alors $v(j,i) = +\infty$ et la proposition est vraie.
2. $j \notin \Gamma^*(1)$. Alors $\lambda(j) = +\infty$ et la proposition est vraie.
3. $(j,i) \in \Gamma \wedge j \in \Gamma^*(1)$. Soit $u \in \mathcal{C}_{1j}$ tel que $v(u) = \lambda(j)$. Alors $u' = u \cdot (j,i) \in \mathcal{C}_{1i}$ donc $\lambda(i) \leq v(u') = v(u) + v(j,i) = \lambda(j) + v(j,i)$. La proposition est donc vraie.

□

Proposition 8.6 *Les attributs $\lambda(i)$ ($i = 1, \ldots, n$) sont solution du système: $\forall i$: $x_i = \min_j(x_j + v(j,i))$*

Démonstration D'après la proposition 8.5,

$$\forall i \ : \ \lambda(i) \leq \min_j(\lambda(j) + v(j,i)) \tag{8.1}$$

Il y a deux cas à considérer:

– $i \in \mathcal{C}_{1i}$. Comme $\lambda(i) = \min_{u \in \mathcal{C}_{1i}} v(u)$, il existe $\overline{u} \in \mathcal{C}_{1i}$ tel que $v(\overline{u}) = \lambda(i)$. Soit p l'avant-dernier sommet de $\overline{u}$. On a donc $\overline{u} = u' \cdot (p,i)$ où $u' \in \mathcal{C}_{1p}$. On peut donc écrire :

$\lambda(i) = v(\overline{u}) = v(u') + v(p,i) \geq \lambda(p) + v(p,i) \geq \min_j(\lambda(j) + v(j,i)$, d'où

$$\lambda(i) \geq \min_j(\lambda(j) + v(j,i)) \tag{8.2}$$

– $i \notin \mathcal{C}_{1i}$. Dans ce cas, $\lambda(i) = +\infty \geq \min_j(\lambda(j) + v(j,i))$, on obtient donc aussi l'équation 8.2
Des équations 8.1 et 8.2 on déduit $\lambda(i) = \min_j(\lambda(j) + v(j,i))$ □

Toutefois, les conditions énoncées dans les deux propositions précédentes ne sont pas *suffisantes*, comme le contre-exemple ci-après (figure 8.1) va le montrer:

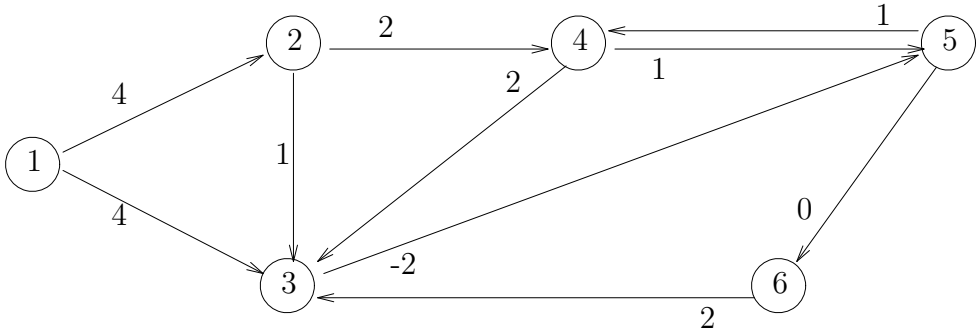


FIG. 8.1 – *Contre-exemple*

Le système d'équations de la proposition 8.6 s'écrit :

$$\begin{cases} x_1 &= 0 \\ x_2 &= \min(x_1 + 2) = x_1 + 2 \\ x_3 &= \min(x_1 + 4, x_2 + 1, x_4 + 2, x_6 + 2) \\ x_4 &= \min(x_2 + 2, x_5 + 1) \\ x_5 &= \min(x_3 - 2, x_4 + 1) \\ x_6 &= \min(x_5 + 0) = x_5 \end{cases}$$

Les valeurs $\mu_1 = 0, \mu_2 = 2, \mu_3 = 2, \mu_4 = 1, \mu_5 = 0, \mu_6 = 0$ satisfont ce système. Et pourtant, ce ne sont pas les valeurs minimales des chemins issus du sommet 1. Par exemple, la valeur minimale des chemins de 1 à 3 est égale à 3, alors que $\mu_3 = 2$. Les valeurs proposées ici constituent bien des *minorants* des valeurs de chemins, mais pas nécessairement des *minima* : elles peuvent être *trop petites*.

Le théorème de caractérisation suivant montre en effet que les valeurs minimales de chemins issus de 1 constituent LA solution *maximale* du système d'équations.

Théorème 8.7 *Les attributs* $\lambda(i)$ $(i = 1, \ldots, n)$ *constituent la solution maximale du système:*

$$x_i = \begin{cases} 0 \text{ si } i = 1 \\ \min_{x_j \in \Gamma^{-1}(x_i)}(x_j + v(j,i)) \text{si } i = 2, \ldots, n \end{cases}$$

Démonstration Soit $\mu(i)$ $(i = 1, \ldots, n)$ la solution maximale du système d'équations. Il faut montrer : $\forall i$ $(1 \leq i \leq n)$: $\lambda(i) = \mu(i)$.

– D'après la proposition 8.6, les $\lambda(i)$ constituent une solution du système et donc :

$$\forall i \ (1 \leq i \leq n) \ : \ \lambda(i) \leq \mu(i) \tag{8.3}$$

– Pour montrer les inégalités contraires, on considère les deux cas possibles :

1. $i \notin \Gamma^\star(1)$. Dans ce cas, $\lambda(i) = +\infty$ et donc $\lambda(i) \geq \mu(i)$.
2. $i \in \Gamma^\star(1)$. Dans ce cas, il existe dans G un chemin $\mathbf{u} = [1,i_1,\ldots,i_k,i]$, de valeur $\lambda(i)$. Puisque les $\mu(i)$ sont solution du système, les inégalités suivantes sont vérifiées:

$$\begin{cases} \mu(i_1) & \leq \mu(1) + v(1,i_1) \\ \mu(i_2) & \leq \mu(i_1) + v(i_1,i_2) \\ & \ldots \\ \mu(i) & \leq \mu(i_k) + v(i_k,i) \end{cases}$$

En sommant membre à membre, et comme $\mu(1) = 0$, on obtient:

$$\mu(i) \leq v(1,i_1) + v(i_1,i_2) + \ldots + v(i_k,i) = v(\mathbf{u}) = \lambda(i)$$

Regroupant les deux cas, on obtient :

$$\forall i \ (1 \leq i \leq n) \ : \ \lambda(i) \geq \mu(i) \tag{8.4}$$

– Des équations 8.3 et 8.4 on déduit :

$$\forall i \ (1 \leq i \leq n) \ : \ \lambda(i) = \mu(i)$$

□

Remarque. Considérons le graphe $G_{min}(1)$ dont l'ensemble des sommets est l'ensemble des descendants du sommet 1, et l'ensemble des arcs est l'ensemble des couples $(pred(i),i)$ pour tout i descendant du sommet 1. Formellement :

$$\begin{aligned} G_{min}(1) &= (\Gamma^*(1),\{(pred(i),i) \mid i \in \Gamma^*(1)\}) \\ &= (\{i \mid \lambda(i) < +\infty\},\{(i,j) \mid \lambda(j) = \lambda(i) + v(i,j)\} \end{aligned}$$

Les caractérisations précédentes montrent que ce graphe est une arborescence de racine 1. Les algorithmes de calcul des chemins minimaux issus de 1 que nous étudions ci-après pourront donc être vus comme des algorithmes d'exploration de la descendance de 1, avec une stratégie de sélection spécifique à chacun d'eux.

8.3 Algorithme de FORD : exploration

8.3.1 Principe

On note x_0 le sommet initial. Le principe de cet algorithme consiste à ajuster progressivement les valeurs des attributs λ jusqu'à ce qu'ils vérifient le système caractéristique. Pour assurer qu'on obtient bien la solution maximale, on part de valeurs initiales dont on sait *a priori* qu'elles majorent les valeurs λ. Pour cela, nous allons tester tous les arcs du graphe selon un ordre de parcours induit par une exploration *en profondeur* analogue à celle réalisée pour l'énumération des chemins élémentaires issus de x_0 (cf. §6.3).

Un arc (y,z) est testé lorsque y est sommet de pile et z est le successeur courant de y; en effet, y est alors reconnu comme descendant de x_0, et l'état de la pile définit le chemin courant de x_0

à y : ($\lambda(y)$, $pred(y)$) ont été initialisés ou mis à jour lorsque y a été empilé pour la première fois. Quant à z, plusieurs cas peuvent se présenter selon son état relatif à l'exploration (cf terminologie du chapitre 5) :

1er cas $\triangleright z$ est *dehors*. Il devient alors *en_attente* (et va être empilé), et ($\lambda(z)$, $pred(z)$) sont initialisés :

$$\begin{aligned} \lambda(z) &\leftarrow \lambda(y) + v(y,z); \\ pred(z) &\leftarrow y \end{aligned}$$

2ème cas $\triangleright$ z est *terminé*; cela signifie que z a été précédemment empilé à partir d'un prédécesseur $\neq y$ puis dépilé – après que toute sa descendance ait été explorée; les attributs de z ont donc déjà une valeur, mais celle-ci peut être remise en cause par l'arc (y,z) qui doit donc être testé:

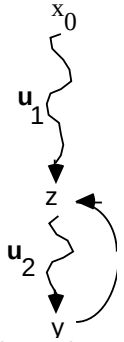
si $\lambda(y) + v(y,z) < \lambda(z)$

alors $\lambda(z) \longleftarrow \lambda(y) + v(y,z)$; $pred(z) \longleftarrow y$;
z est empilé car la modification de $\lambda(z)$ peut remettre en cause les valeurs précédemment attribuées aux descendants de z, d'où la nécessité de refaire l'exploration à partir de z.

sinon inutile d'empiler z à nouveau : ses attributs ne sont pas modifiés donc ceux de ses descendants non plus.

fsi

3ème cas $\triangleright$ z est *en_attente* (il appartient donc à la pile.)



Cela signifie qu'un circuit passant par y et z est rencontré, puisque z est à la fois antécédent et successeur de y. Le test de mise à jour des attributs de z permet alors de détecter si ce circuit est absorbant; examinons la figure ci-contre :

Proposition : le circuit $\mathbf{u}_2 \cdot (y,z)$ est de valeur <0 si et seulement si :

$\lambda(y) + v(y,z) < \lambda(z)$

Démonstration Il existe un chemin $\mathbf{u}_1 = [1 * z]$, de valeur $\lambda(z)$. Comme $\mathbf{u}_1 \cdot \mathbf{u}_2$ est le chemin courant de 1 à y, on a :

(1) $\lambda(y) \leq v(\mathbf{u}_1) + v(\mathbf{u}_2) = \lambda(z) + v(\mathbf{u}_2)$

i) $\triangleright$ Supposons $v(\mathbf{u}_2) + v(y,z) < 0$. D'après (1), on a :

$$\begin{aligned} \lambda(y) + v(y,z) &\leq \lambda(z) + v(\mathbf{u}_2) + v(y,z) \\ &< \lambda(z) \quad \text{d'après l'hypothèse.} \end{aligned}$$

ii) $\triangleright$ Réciproquement, supposons $\lambda(y) + v(y,z) < \lambda(z)$. L'inégalité (1) ne peut pas être stricte : en effet, si $\lambda(y) < \lambda(z) + v(\mathbf{u}_2)$, cela signifie que y a été visité précédemment comme extrémité d'un chemin ne passant pas par z; y a alors été empilé et sa descendance explorée. L'arc (y,z)

a donc été testé et on doit avoir $\lambda(z) \leq \lambda(y) + v(y,z)$, ce qui contredit (1).
Par conséquent on a nécessairement :

$\lambda(y) = \lambda(z) + v(\mathbf{u}_2)$, ce qui, avec l'hypothèse, permet de déduire :

$\lambda(y) > \lambda(y) + v(y,z) + v(\mathbf{u}_2)$ d'où : $0 > v(y,z) + v(\mathbf{u}_2)$

□

La détection d'un circuit absorbant est un critère d'arrêt de l'algorithme; il suffit de gérer une variable booléenne `absorbant`, initialement à `faux` et qui passe à `vrai` lorsqu'un tel circuit est détecté.

8.3.2 Texte de l'algorithme

Celui-ci est une adaptation de l'algorithme d'énumération des chemins élémentaires vu au 6.5. Le texte est donné page 105 (voir la signification des accès à l'annexe A).

FORD : EXPLORATION EN PROFONDEUR

```
REEL SOMMET::λ ; SOMMET SOMMET::pred ;
ford-profondeur(GRAPHE G, SOMMET x)  c'est
   local  ENS[SOMMET] SOMMET::a_visiter: ;
          ENUM[dehors, en_attente, terminé] SOMMET::etat ;
          PILE[SOMMET] A; BOOL absorbant ; SOMMET y, z ;
  début
    depuis
       pourtout y  de G.lst_som  faire y.état ← dehors  fpourtout ;
         absorbant ←  faux ; A.creer ; A.empiler(x) ;
         x.état  ← en_attente ; x.λ ← 0 ; x.a_visiter ← G.lst_succ(x) ;
    jusqua A.vide  ou absorbant
    faire
      y  ←  A.sommetpile ;
       si y.a_visiter = ∅
          alors y.état ← terminé ; A.depiler
          sinon
            z ← y.a_visiter.element ; y.a_visiter ← y.a_visiter \{z} ;
             cas z.état =
                       dehors ⟶
                          z.état ← en_attente ; z.λ ← y.λ+  G.v(y,z) ; z.pred ← y ;
                          A.empiler(z) ; z.a_visiter ← G.lst_succ(z)
                       en_attente ⟶
                          absorbant ← (z.λ  > y.λ + G.v(y,z))
                       terminé ⟶
                           si z.λ > y.λ+ G.v(y,z)
                              alors z.état  ← en_attente ;
                                    z.λ ← y.λ+ G.v(y,z) ; z.pred ← y ;
                                    A.empiler(z) ; z.a_visiter  ← G.lst_succ(z)
                           fsi
             fcas
       fsi
    fait ;
  --   si absorbant alors  il y a un circuit absorbant
                  sinon les sommets ∉ en_attente  sont inaccessibles
 fin
```

8.3.3 Une heuristique d'amélioration

Dans la stratégie d'exploration en profondeur utilisée, le choix d'un successeur du sommet de pile est laissé arbitraire. Il est posible de définir un critère de choix d'un tel successeur, tenant compte de la nature du problème à résoudre, à savoir la recherche de chemins de valeur *minimale* sur un graphe dont les arcs sont *valués*. Un tel critère doit laisser espérer une amélioration du temps d'exécution de l'algorithme, mais sans que le prix à payer pour sa mise en œuvre ne devienne prohibitif: ce critère ne devra donc prendre en compte que des informations *locales*, c'est-à-dire relatives à l'*état courant* du *sommet courant*. Par contre, l'amélioration censée résulter de l'application de ce critère ne pourra être établie sûrement; dans certains cas, il pourrait même en résulter une détérioration du comportement, mais en moyenne une amélioration doit intervenir.

Un critère possible et simple à mettre en œuvre consiste à choisir, parmi les successeurs possibles du sommet de pile y, un sommet z tel que $v(y,z)$ soit minimum. L'amélioration attendue réside dans une diminution du nombre de *retours dans la pile*, donc de reprise de parcours déjà effectués. Le prix à payer est un tri sur l'ensemble des valeurs des arcs issus de y.

8.4 Algorithme de BELLMANN-KALABA

8.4.1 Principe

Il s'applique à tout graphe valué, et permet de détecter d'éventuels circuits absorbants. Nous poserons, par convention :

$v(1,1) = 0$ et, si $\mid \mathbf{u} \mid = 0, v(\mathbf{u}) = 0$.

Pour tout $x \in X$ et tout entier $k \geq 1$, on notera $C_x^{(k)}$ l'ensemble des chemins de 1 à x, de longueur $\leq k$. Remarquer que $C_x^{(k)} = \emptyset$ si et seulement si tout chemin de 1 à x, s'il en existe, est de longueur $> k$.

L'algorithme de BELLMANN-KALABA est fondé sur le résultat suivant :

Proposition 8.8 *On considère les valeurs $\lambda_x^{(k)}$ définies par :*

$$\lambda_x^{(1)} = \begin{cases} v(1,x) & \textbf{si} \quad (1,x) \in \Gamma \\ 0 & \textbf{si} \quad x = 1 \\ +\infty & \textbf{si} \quad (1,x) \notin \Gamma \end{cases}$$

$$\forall k \geq 2 \ : \lambda_x^{(k)} = \min_{y \in \Gamma^{-1}(x)} [\lambda_y^{(k-1)} + v(x,y)]$$

Alors $\lambda_x^{(k)} = \min_{\mathbf{u} \in C_x^{(k)}} v(\mathbf{u})$ ($+\infty$ si $C_x^{(k)} = \emptyset$).

Démonstration. Par récurrence sur k :
Base : Vérifié pour $k = 1$, par définition.
Induction : Supposons la proposition vérifiée pour $k - 1 \geq 1$
1^{er} cas : $C_x^{(k)} \neq \emptyset$.

Soit $\mathbf{u} \in C_x^{(k)}$ et y le prédécesseur de x sur ce chemin. On a donc

$\mathbf{u} = \mathbf{u}_1.(y,x)$ avec $\mathbf{u}_1 \in C_y^{(k-1)}$

Avec la convention $\mid \mathbf{u}_1 \mid = 0 \Longrightarrow v(\mathbf{u}_1) = 0$, on obtient, d'après l'hypothèse de récurrence et la définition de $\lambda_x^{(k)}$:

$v(\mathbf{u}) = v(\mathbf{u}_1) + v(x,y) \geq \lambda_y^{(k-1)} + v(x,y) \geq \lambda_x^{(k)}$

Donc, $\lambda_x^{(k)} \leq \min_{\mathbf{u} \in C_x^{(k)}} v(\mathbf{u})$.

D'autre part, par définition de $\lambda_x^{(k)}$, il existe y_0 tel que

$$\lambda_x^{(k)} = \lambda_{y_0}^{(k-1)} + v(y_0,x)$$

et d'après l'hypothèse de récurrence, il existe

$$\overline{u}_1 \in C_{y0}^{(k-1)} \text{ avec } v(\overline{\mathbf{u}}_1) = \lambda_{y_0}^{(k-1)}$$

d'où $\overline{\mathbf{u}} = \overline{\mathbf{u}}_1 \cdot (y_0,x)$ vérifie

$$\overline{\mathbf{u}} \in C_x^{(k)} \text{ et } v(\overline{\mathbf{u}}) = \lambda_x^{(k)}$$

Donc, dans ce cas, $\lambda_x^{(k)} = \min_{\mathbf{u} \in C_x^{(k)}} v(\mathbf{u})$.

$2^{ème}$ cas : $C_x^{(k)} = \emptyset$. Pour tout sommet $y \in \Gamma^{-1}(x)$ on a $C_y^{(k-1)} = \emptyset$, et donc $\lambda_y^{(k-1)} = +\infty$ (hypothèse de récurrence). D'où, par définition, $\lambda_x^{(k)} = +\infty$. □

Corollaire 8.9 *Il existe $k \leq n$ tel que*
$\forall x \in X \ : \lambda_x^{(k)} = \lambda_x^{(k-1)}$
si et seulement si le sous-graphe G_0 ne comporte pas de circuit absorbant

Démonstration.
▷ Supposons que $\exists x \in X \ : \lambda_x^{(n)} < \lambda_x^{(n-1)}$ Alors : $\exists \mathbf{u} \in C_x^{(n)}$ tel que $\lambda_x^{(n)} = v(\mathbf{u})$ et, pour tout chemin $\mathbf{u}' \in C_x^{(n-1)}$, on a : $v(\mathbf{u}) < v(\mathbf{u}')$.

Par conséquent, $\mid \mathbf{u} \mid = n$ et donc $\mathbf{u}$ n'est pas élémentaire, d'où l'on déduit l'existence d'un circuit de valeur < 0 inclus dans $\mathbf{u}$, donc dans le sous-graphe G_0.
▷ Réciproquement, d'après la proposition 8.8, le problème 1-minimal admet alors une solution, ce qui prouve l'inexistence de circuit absorbant dans le sous-graphe G_0 . □

Corollaire 8.10 $\forall k \geq 1, \forall x \in \Gamma^*(1)$, *le prédécesseur de x sur un chemin de $C_x^{(k)}$ de valeur minimale est un sommet y^k pour lequel le minimum est atteint lors du calcul de $\lambda_x^{(k)}$*

Démonstration. Elle résulte immédiatement de la démonstration de la proposition 8.8 □

8.4.2 Analyse

Invariant $1 \leq k \leq n \wedge$
$A = \cup_{j=1}^{k} \Gamma^{[j]}(1) \wedge$
$\forall x \in A$ $\lambda\_anc(x)$ = valeur minimale des chemins de $C_x^{(k-1)} \wedge$
$\lambda(x)$ = valeur minimale des chemins de $C_x^{(k)} \wedge$
$pred(x)$ = prédécesseur optimal de x sur de tels chemins $\wedge$
$stab \equiv \lambda = \lambda\_anc$

Terminé $(k = n) \vee stab$
resultat si $stab$ **alors** (λ, $pred$)
sinon il y a des circuits absorbants **fsi**

Progression (* $k < n \wedge \neg stab$ *)
$\lambda\_anc \leftarrow \lambda$; $stab \leftarrow$ **vrai** ;
$\forall x \in X$: $\lambda(x) \leftarrow \min\{\lambda\_anc(y) + v(y,x) \mid y \in \Gamma^{-1} \cap A\}$;
$pred(x) \leftarrow$ valeur de y pour laquelle ce min est atteint ;
si $x \in A$ **alors** $stab \leftarrow (stab \wedge \lambda(x) = \lambda\_anc(x))$
sinon $stab \leftarrow$ **faux** ; $A \leftarrow A \cup \{x\}$
fsi ;
$k \leftarrow k + 1$

Valeurs initiales $\lambda(1) \leftarrow 0$;
$\forall x \in \Gamma(1) : \lambda(x) \leftarrow v(1,x)$;
$pred(x) \leftarrow 1$;
$A \leftarrow \{1\} \cup \Gamma(1)$;
$k \leftarrow 1$

8.4.3 Texte de l'algorithme

Il est donné page 109.

8.4.4 Preuve

Nous ne la détaillerons pas, la laissant en exercice au lecteur. Les quatre résultats suivants doivent être montrés :

Lemme 8.11 *Les valeurs initiales satisfont l'invariant.*

Démonstration. Évidente, grâce à la convention $v(1,1) = 0$

ALGORITHME DE BELLMANN-KALABA

```
REEL SOMMET::λ ;
SOMMET SOMMET::pred ;
REEL SOMMET::λ_anc ;

bellmann-kalaba(GRAPHE G, SOMMET x) c'est
  local ENS[SOMMET] A, E;
        BOOL stab;
        ENT k;
        SOMMET x, y ;
 début
  depuis
    k ← 1 ; stab ← faux ; x.λ ← 0 ; A ← {x0} ; E ← G.lst_succ(x) ;
     pourtout x de E
       x.λ ← G.v(x,x) ; A ← A ∪{x}
     fpourtout ;
  jusqua stab ou k ≥ n
  faire
    k ← k+1 ;
     pourtout x de G.lst_som faire x.λ_anc ← x.λ fpourtout ;
    stab ← vrai ;
     pourtout x de G.lst_som faire
       E ← G.lst_pred(x) ; E ← E ∩ A ;
        si E ≠ ∅ alors
          (x.λ, x.pred) ← MIN(y.λ_anc + G.v(y,x), E) ;
           si x ∈ A
             alors stab ← stab et (x.λ =x.λ_anc)
             sinon stab ← faux ; A ← A ∪{x}
           fsi
        fsi
     fpourtout ;
  fait ;
  --   si k < n alors Resultat : ← (λ , pred)
     --   sinon circuit absorbant détecté
 fin
```

$(w,y0) \leftarrow$ *min* $(f(y),E)$ est une primitive, signifiant que
$w = \min\,(f(y) \mid y \in E) = f(y0)$, où f est une fonction $E \rightarrow \mathbb{R}$

Lemme 8.12 *La progression maintient l'invariant.*

Démonstration. Utiliser :
- La proposition 8.8 pour les valeurs λ
- Le corollaire 8.10 pour les valeurs *pred*
- Le corollaire 8.9 pour le booléen *stab*

Proposition 8.13 *(Correction partielle). Lorsque l'algorithme s'arrête, on a l'alternative :*
- **ou bien** *stab et dans ce cas,* $\forall x \in \Gamma^*(1)$ *les attributs* $(\lambda(x), pred(x))$ *sont minimaux*
- **ou bien non** *stab* **et** $k = n$ *et dans ce cas, il y a des circuits absorbants*

Démonstration. Conséquence de ($stab\wedge$ Invariant) $\vee$ ($k = n \wedge \neg stab\wedge$ Invariant)

Proposition 8.14 *(Terminaison). L'algorithme s'arrête en un nombre fini d'étapes*

Démonstration. Considérer la fonction $n - k$

8.4.5 Complexité

L'algorithme exécute au plus $n - 1$ pas d'itération, et pour chacun d'eux :

n accès *lst_pred*

n calculs de MIN, sur un ensemble E, impliquant chacun :

$card(E)$ accès valarc,

$card(E)$ comparaisons et additions de réels;

comme $card(E) \leq n$, on obtient un algorithme en $0(n^3)$.

Remarque. L'algorithme de BELLMANN-KALABA met en œuvre, au niveau du problème 8.2, le principe d'optimalité de BELLMANN qui fonde les méthodes de programmation dynamique : les *étapes* sont les longueurs des chemins examinés, les *états* à l'étape k sont les sommets qui peuvent être atteints par un chemin de longueur k et les transitions possibles de l'étape k-1 à l'étape k sont définies par les arcs issus des sommets états à l'étape k-1. La relation

$$\lambda_x^{(k)} = \min_{y \in \Gamma^{-1}(x)} (\lambda_y^{(k-1)} + v(x,y))$$

n'est autre, dans ce contexte, que la traduction du principe d'optimalité.

8.4.6 Accélération de l'algorithme

Il est possible d'accélérer l'algorithme en utilisant, dans le calcul de $\lambda_x^{(k)}$, les valeurs $\lambda_y^{(k)}$ (avec $y < x$), dès leur mise à jour dans l'étape courante. En effet, on a les résultats suivants :

Proposition 8.15 *On définit, par récurrence :*

$\lambda_x^{(1)}$ *comme dans la proposition 8.8*

$$\forall k \geq 2 \ : \lambda_x^{(k)} = \min_{y \in \Gamma^{-1}(x)} (\mu_y^{(k)} + v(y,x))$$

avec :

$$\mu_y^{(k)} = \begin{cases} \lambda_y^{(k)} & \textbf{si} \quad y < x, \\ \lambda_y^{(k-1)} & \textbf{si} \quad y \geq x \end{cases}$$

On a alors :
si $C_x^{(k)} \neq \emptyset$ *:*

i) $\lambda_x^{(k)}$ *minore la valeur minimale des chemins de* $C_x^{(k)}$.

ii) *il existe un chemin de* 1 *à* x*, ayant pour valeur* $\lambda_x^{(k)}$.

Démonstration Elle s'effectue par récurrence sur k, et pour chaque k, par récurrence sur x.
▷ Pour $k = 1$, la proposition est vérifiée.
▷ Supposons là vérifiée pour k - 1, et soit x tel que $C_x^{(k)} \neq \emptyset$.
Il existe $\mathbf{u} \in C_x^{(k)}$ et soit y le prédécesseur de x sur $\mathbf{u}$. Alors :

$$\mathbf{u} = \mathbf{u}'.(y,x),\ \mathbf{u}' \in C_y^{(k-1)},\ v(\mathbf{u}) = v(\mathbf{u}') + v(y,x)$$

c'est à dire, d'après l'hypothèse de récurrence :

$$v(\mathbf{u}) \geq \lambda_y^{(k-1)} + v(y,x)$$

Montrons, par récurrence sur x, que $\lambda_x^{(k)} \geq v(\mathbf{u})$
Pour x=1, on a $y \geq 1$ donc $\lambda_y^{(k-1)} = \mu_y^{(k-1)}$ et par conséquent :

$$v(\mathbf{u}) \geq \mu_y^{(k)} + v(y,x) \geq \lambda_x^{(k)}$$

Supposons que cette inégalité est vérifiée jusqu'à x-1. On a alors :

$$\textbf{si } y < x \textbf{ alors } v(\mathbf{u}') \geq \min_{\gamma \in C_y^{(k-1)}} v(\gamma) \geq \min_{\gamma \in C_y^{(k)}} v(\gamma) \geq \lambda_y^{(k)} = \mu_y^{(k)}$$

d'où
$v(\mathbf{u}) \geq \mu_y^{(k)} + v(x,y) \geq \lambda_y^{(k)}$ tandis que si $y \geq x$:

$\lambda_y^{(k-1)} = \mu_y^{(k)}$ et par conséquent :

$v(\mathbf{u}) \geq \mu_y^{(k)} + v(x,y) \geq \lambda_y^{(k)}$

Ceci montre la partie **i)** de la proposition.

Pour montrer la partie **ii)**, considérons la suite de sommets
$x = y_0 > y_1 > \ldots > y_\nu$ et $y_\nu \geq y_{\nu+1}$ telle que :

$$\begin{cases} \lambda_x^{(k)} & = & \lambda_{y_1}^{(k)} + v(y_1,x) \\ \lambda_{y_1}^{(k)} & = & \lambda_{y_2}^{(k)} + v(y_1,y_2) \\ & \vdots & \\ \lambda_\nu^{(k)} & = & \lambda_{\nu+1}^{(k)} + v(y_{\nu+1},y_\nu) \end{cases}$$

y_ν existe nécessairement (éventuellement $\nu = 0$), puisque la suite est décroissante et minorée par 1. Si on somme cette suite d'égalités, il vient :

$$\lambda_x^{(k)} = \lambda_{y_{\nu+1}}^{(k-1)} + v(y_{\nu+1},y_\nu) + ... + v(y_1,x)$$

D'après l'hypothèse de récurrence, il existe un chemin $\mathbf{u'} = [1 * y_{\nu+1}]$ avec $v(\mathbf{u'}) = \lambda_{y_{\nu+1}}^{(k-1)}$. Le chemin $\mathbf{u'} \cdot (y_{\nu+1},y_\nu,\ldots,y_1,x)$ a donc pour valeur $\lambda_x^{(k)}$. □

Corollaire 8.16 *Il existe $k \leq n$ tel que*
$\forall x \in X \ : \lambda_x^{(k)} = \lambda_x^{(k-1)}$
si et seulement si le sous-graphe G_0 ne comporte pas de circuit absorbant

La démonstration est identique à celle du corollaire 8.9

Nous ne décrivons pas l'algorithme accéléré en entier, mais seulement les modifications à apporter à l'algorithme précédent : le tableau λ_anc devient inutile ; on le remplace par une simple variable réelle δ. (L'affectation $\lambda\_anc \leftarrow \lambda$ est donc à supprimer).

Remarque. Comme pour l'algorithme de FORD, l'algorithme de BELLMANN-KALABA rentre dans une classe d'algorithmes par ajustements successifs, et donc n'apporte pas de solution spécifique au problème 8.1. Par contre, l'algorithme dans sa version non accélérée permet de restreindre le calcul aux chemins ayant une longueur maximum donnée.

Les deux algorithmes qui vont suivre rentrent dans la classe des algorithmes *gloutons*, c'est à dire qu'ils calculent le résultat définitif de chaque sommet, examinés dans un certain ordre, sans jamais revenir en arrière - c'est à dire sans jamais remettre en cause les résultats partiels déjà acquis. A ce titre, ils peuvent être adaptés au problème 8.1. Toutefois, ils ne s'appliquent qu'à des graphes ayant des propriétés particulières : valeurs positives ou nulles pour le premier, graphes sans circuit pour le second.

8.5 Cas des arcs de valeur positive ou nulle : algorithme de DIJKSTRA

$G = (X,\Gamma,v)$ est un graphe valué, avec $\forall \gamma \in \Gamma \ : v(\gamma) \geq 0$ si bien que G ne possède aucun circuit absorbant.

8.5.1 Principe de l'algorithme de DIJKSTRA

L' algorithme de DIJKSTRA est une stratégie particulière de l'algorithme d'exploration pour le calcul de la **descendance** du sommet 1 vu au 6.2. Il est donc plus simple que l'algorithme d'exploration de FORD du §8.3, basé sur l'exploration de **tous** les chemins élémentaires issus de 1 : ceci est du à l'hypothèse plus restrictive sur les valeurs des arcs.

Par rapport au schéma général du §6.2, les précisions et modifications suivantes sont apportées :

- Lorsqu'un sommet z devient *en_attente*, ses attributs sont initialisés :
  $\lambda(z) \leftarrow \lambda(y) + v(y,z)$; $pred(z) \leftarrow y$, où y est son père dans l'arborescence d'exploration.
- Lorsqu'un sommet z est *terminé*, ses attributs ($\lambda(z)$, $pred(z)$) sont définitifs (leur valeur est correcte)
- La sélection d'un sommet d'état *en_attente* se fait selon le critère suivant : on choisit le sommet y d'état *en_attente* dont l'attribut $\lambda(y)$ est minimum. Le sommet sélectionné devient alors *terminé* et, pour tout $z \in \Gamma(y)$:
  - si z est *dehors*, il devient *en_attente* et ses attributs sont initialisés
  - si z est *en_attente* ses attributs sont réévalués de la manière habituelle :

    si $\lambda(y) + v(y,z) < \lambda(z)$
    alors $\lambda(z) \leftarrow \lambda(y) + v(y,z)$; $pred(z) \leftarrow y$
    fsi
  - si z est *terminé*, il est ignoré.

Le fait que les attributs des sommets puissent atteindre leur valeur finale avant la fin de l'algorithme, et que l'on sache à quel moment de l'exécution ce fait se produit, indique que cet algorithme rentre dans la classe des algorithmes gloutons.

Remarque. Au §6.2 nous avions mis en évidence les stratégies *en largeur d'abord* et *en profondeur d'abord*. La stratégie utilisé ici peut être qualifiée, elle, de *meilleur d'abord (best-first search)*. Relativement au problème traité, cette stratégie est *informante*, c'est-à-dire qu'elle tient compte des informations déjà obtenues sur les valeurs de chemins (informations *sémantiques*), alors que les deux premières sont *aveugles*, c'est-à-dire purement *syntaxiques*.

La gestion de l'ensemble des sommets d'état *en_attente*, noté A, va être effectuée selon la structure de données appelée **tas**, qui est une structure ordonnée dans laquelle l'accès à un élément de valeur *minimum* parmi tous les éléments de l'ensemble se fait *en temps constant*, c'est-à-dire *indépendant du nombre d'éléments contenus dans cette structure*. De manière analogue aux *files* (§6.3.1) et aux *piles* (§6.3.2), on considère le type tas comme un type de données abstrait. La déclaration

`TAS[E_ORD] t` déclare un *tas d'objets de type* `E_ORD`, où `E_ORD` est un type dont les valeurs sont ordonnées.

Les opérations sur les tas sont alors :

`creer`	crée un nouveau tas vide
`BOOL tasvide`	prédicat à valeur `vrai` si et seulement si le tas est vide
`mettre_en_tas(E_ORD v)`	ajout dans le tas de l'élément de valeur `v`

`ôter_de_tas`	retrait d'un élément de valeur minimum (erreur si le tas est vide)
`E_ORD mintas`	délivre un élément du tas de valeur minimum, sans modifier le tas (erreur si le tas est vide)
`reorg(E_ORD v1, v2)`	remplacement dans le tas de la valeur `v1` par la valeur `v2` (erreur si la valeur `v1` est absente). nb: `t.reorg(v1,v2)` est équivalent à la séquence: `t.ôter_de_tas(v1); t.mettre_en_tas(v2)`

La mise en œuvre concrète d'une telle structure abstraite et des accès est vue dans les cours de techniques ou méthodes de programmation (elle se fait sous forme d'arbre binaire ordonné, complet à gauche, souvent représenté en tableau). Noter que le terme *liste de priorité (priority queue)* est aussi utilisé pour désigner un tas.

8.5.2 Analyse de l'algorithme

L'analyse correspondant au principe exposé ci-dessus est donc dérivée de celle de l'algorithme général d'exploration de la descendance de 1. Les éléments du tas sont ici des objets à deux champs `(SOMMET, RÉEL)`, où à chaque sommet x du graphe valué est associé son attribut $\lambda(x)$ (lorsque celui-ci a été initialisé). Les valeurs du type de base sont ordonnées par le champ `RÉEL` (attributs λ)

Invariant : (I1) : A = sommets d'état en_attente $\subseteq \Gamma^*(1)$
(I2) : T = sommets d'état terminé $\subseteq \Gamma^*(1)$
(I3) : $\forall x \in T$: $\Gamma(x) \subseteq T \cup A$
(I4) : $\forall x \in T$:
$(\lambda(x), pred(x))$ = attributs minimaux de x
(I5) : $\forall x \in A$:
$\lambda(x) = \min_{y \in T \cap \Gamma^{-1}(x)}(\lambda(y) + v(y,x))$
$pred(x)$ = sommet pour lequel ce minimum est atteint

Terminé : $A.tasvide$

Progression :
$(nouv, \lambda) \leftarrow A.mintas$;
$A.oter\_de\_tas$;
$T \leftarrow T \cup \{nouv\}$;
$MOD \leftarrow \Gamma(nouv)$;
$\forall x \in MOD$:
si $x \in A$
alors mise à jour de $(\lambda(x), pred(x))$
sinon initialisation de $(\lambda(x), pred(x))$;
$A.mettre\_en\_tas(x, \lambda(x))$
fsi

Valeurs initiales : $T \leftarrow \emptyset$; $A.creer$
$\lambda(x_0) \leftarrow 0$;
$A.mettre\_en\_tas((x_0,0))$

8.5.3 Exemple d'exécution

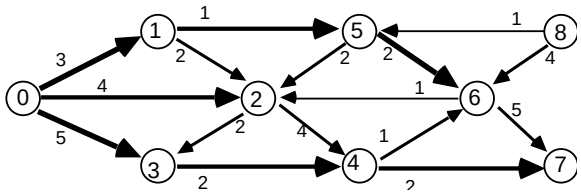


FIG. 8.2 – *Exemple pour l'algorithme de* DIJKSTRA

Etape	nouv	MOD	tas	Terminés
init			(0,0)	
1	0	1,2,3	(1,3),(2,4),(3,5)	0
2	1	2,5	(2,4),(5,4),(3,5)	0,1
3	2	3,4	(5,4),(3,5),(4,8)	0,1,2
4	5	2,6	(3,5),(4,8),(6,6)	0,1,2,5
5	3	4	(6,6),(4,7)	0,1,2,5,3
6	6	2,7	(4,7),(7,11)	0,1,2,5,3,6
7	4	-,7	(7,9)	0,1,2,5,3,6,4
8	7	$\emptyset$	$\emptyset$	0,1,2,5,3,6,4,7

fin de l'algorithme. Le sommet 8 est **inaccessible**

8.5.4 Texte de l'algorithme

8.5.5 Preuve

Lemme 8.17 *Les valeurs initiales vérifient l'invariant*

Démonstration. Evident.

Lemme 8.18 *L'invariant est maintenu par la progression.*

Démonstration. Les parties I1, I2 et I3 de l'invariant sont maintenues par la progression : seul

ALGORITHME DE DIJKSTRA

```
REEL SOMMET::λ ; SOMMET SOMMET::pred ;
dijkstra(GRAPHE G, SOMMET x)  c'est
   local ENUM[dehors,en_attente,terminé] SOMMET::état ;
         TAS[DOUBLET[SOMMET, REEL]] A;
         ENS[SOMMET] MOD --     sommets modifiables
         SOMMET z, nouv ;
         REEL w;
 début
   depuis
     pourtout x  de G.lst_som  faire x.état  ← dehors  fpourtout ;
    A.creer ;
    x.état ← en_attente ; x.λ ← 0 ;
    A.mettre_en_tas((x,0)) ;
   jusqua A.vide
   faire
     (nouv,w) ← A.mintas ; --   w=nouv.λ
      A.ôter_de_tas;
      nouv.état ← terminé ;MOD  ← G.lst_succ(nouv) ;
      --   mise à jour des marques des sommets modifiables
       pourtout z  de MOD
          cas z.état = dehors ⟶ z.état  ← en_attente ;
                                  z.λ  ← w +  G.v(nouv, z) ;
                                  z.pred  ← nouv ;
                                  A.mettre_en_tas((z, z.λ))
               z.état = en_attente ⟶  si w + G.v(nouv, z) < z.λ
                                        alors
                                          A.reorg((z, z.λ),(z, w + G.v(nouv, z)));
                                          z.λ  ← w + G.v(nouv, z) ;
                                          z.pred  ← nouv
                                        fsi
               z.état = terminé ⟶  rien
          fcas
       fpourtout
   fait
--   les sommets d'état dehors sont inaccessibles
 fin
```

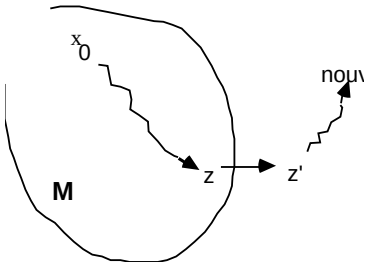
le sommet *nouv* passe de l'état *en_attente* à l'état *terminé*; il est ôté de A, ajouté à T et il est descendant de 1. D'autre part, les nouveaux éléments de A sont successeurs de *nouv*, donc

descendants de 1.

Pour I4 et I5, il faut montrer :

1. le sommet *nouv* vérifie I4
2. les sommets $z \in A$, dont les attributs sont initialisés ou modifiés, vérifient I5

▷ **Maintien de I4** : ici, T désigne la valeur de l'ensemble des sommets d'état *terminé* au début du pas d'itération (avant l'adjonction de *nouv*).



Soit **u** un chemin quelconque de 1 à *nouv*. Comme $1 \in T$ et $nouv \notin T$, ce chemin "sort" de T : soit z le dernier sommet de **u**, faisant partie de T, et z' son successeur sur **u**, qui appartient donc à A par construction (éventuellement, $z' = nouv$). Le chemin **u** se décompose en :

$$\mathbf{u} = [1 * z] \cdot (z,z') \cdot [z' * nouv],$$

et donc :

$$\begin{array}{rcll} v(\mathbf{u}) & = & v[1*z] + v(z,z') + v[z'*nouv] & \\ & \geq & v[1*z] + v(z,z') & (\text{valeurs des arcs } \geq 0) \\ & \geq & \lambda(z) + v(z,z') & (z \in T) \text{ et I4} \\ & \geq & \lambda(z') & (z \in \Gamma^{-1}(z') \cap T \text{ et I5}) \\ & \geq & \lambda(nouv) & (\text{critère de sélection de } nouv) \end{array}$$

$\lambda(nouv)$ minore donc la valeur des chemins de 1 à *nouv*.

D'autre part, soit $y = pred(nouv)$. Par construction :

$$\lambda(nouv) = \lambda(y) + v(y,nouv)$$

Comme $y \in T$, il existe un chemin $[1 * y]$, de valeur $\lambda(y)$. Le chemin $[1 * y].(y,nouv)$ a donc pour valeur $\lambda(nouv)$, ce qui montre le point 1 (maintien de I4).

▷ **Maintien de I5** : Soit $x \in \Gamma(nouv)$.

- Si x est dans l'état *dehors*, alors $\Gamma^{-1}(x) \cap T = \{nouv\}$, sans quoi x serait déjà dans A au début de l'étape. Par construction, les attributs de x vérifient bien I5
- Si x est dans l'état *en_attente*, alors *nouv* est ajouté à $\Gamma^{-1}(x) \cap T$, et le test sur $\lambda(x)$ montre que les nouveaux attributs de x vérifient toujours I5

□

Proposition 8.19 *(Correction partielle). Si l'algorithme s'arrête, on a à l'arrêt :*
$\forall x \in \Gamma^*(1)$*), les valeurs* $(\lambda(x), pred(x))$ *constituent les attributs minimaux de* x

Démonstration. Il suffit de montrer que, lorsque l'algorithme est terminé, on a : $T = \Gamma^*(1)$; le résultat découle alors de I4.

D'après I1, on a $T \subseteq \Gamma^*(1)$

Réciproquement, d'après I3 et la condition d'arrêt $A = \emptyset$, tout successeur d'un sommet de T appartient à T; donc tout descendant d'un sommet de T appartient aussi à T, ce qui montre l'inclusion inverse $\Gamma^*(1) \subseteq T$ □

Proposition 8.20 *(Terminaison) L'algorithme s'arrête au bout d'un nombre fini d'itérations*

Démonstration. La fonction card(T) est strictement croissante, à valeurs entières positives ou nulles et majorée par la valeur finie card($\Gamma^*(1)$); la condition d'arrêt $(A = \emptyset) \equiv (T = \Gamma^*(1))$ est donc obtenue en un nombre fini d'étapes. □

8.5.6 Complexité

L'algorithme effectue au plus n pas d'itérations, et à l'étape n^o k, on a :

card$(T) = k$,

card$(A) \leq n - k$,

card$(MOD) \leq d^+(nouv_k)$, où $d^+(x) =$ card$(\Gamma(x))$, et $nouv_k$ le $k^{ème}$ sommet terminé.

La boucle sur MOD comporte donc **au plus** $d^+(nouv_k)$ additions, comparaisons et accès *mettre_en_tas* ou *reorg* (exclusivement l'un de l'autre).

De plus, 1 accès *min*, 1 accès *ôter_de_tas* et un accès *lst_succ* sont effectués à chaque étape.

Au total, on a donc au plus (m désigne le nombre d'arcs) :

$\sum_{k=1}^{n} d^+(nouv_k) = m$	additions, comparaisons;
m	accès *mettre_en_tas* ou *reorg*
n	accès *min*, *ôter_de_tas*;
n	accès *lst_succ*

Or, on démontre (dans les cours sur les structures de données ou sur les méthodes de tri) que la gestion d'un tas de p éléments nécessite $O(\log_2 p)$ comparaisons pour la mise en œuvre des accès *mettre_en_tas, reorg, ôter_de_tas* et $O(1)$ pour *min*.

La complexité de l'algorithme est donc finalement :

$O(m + n \log_2 n)$	comparaisons
$O(n)$	accès *lst_succ*

Remarque. Supposons qu'il existe une constante d telle que $\forall x \in X \ : d^+(x) \leq d$ (ce qui implique notamment $m \leq d.n$). On a alors $O(m) = O(n)$ et, de plus, la complexité de mise en œuvre d'un accès *lst_succ* est $O(1)$. L'algorithme est alors en $O(n \log_2 n)$

(La même remarque vaut si le graphe est peu dense, c'est-à-dire m est beaucoup plus petit que n^2)

8.6 Cas des graphes sans circuit : algorithme ORDINAL

8.6.1 Principe

Comme l'algorithme de DIJKSTRA, cet algorithme rentre dans la catégorie des algorithmes gloutons; c'est l'hypothèse restrictive d'absence de circuit qui rend cette simplification possible.

(Par contre, les valeurs des arcs peuvent être de signe quelconque).

Dans un premier temps, nous présentons un algorithme valable dans le cas où le sommet 1 est racine de G, c'est à dire :

$$\Gamma^{-1}(1) = \emptyset \textbf{ et } \Gamma^*(1) = X$$

Nous verrons ensuite comment lever cette hypothèse.

Le parcours de la liste des arcs est ici organisé selon la méthode employée lors du calcul de la fonction ordinale (§7.2) avec, en plus, la prise en compte des valeurs des arcs, d'où le nom donné à cet algorithme. Ainsi, un arc (x,y) est examiné lorsque x est point d'entrée du graphe courant, sachant que les attributs définitifs ($\lambda(x)$, $pred(x)$) ont déjà été calculés.

La mise en œuvre de cet algorithme se fait alors selon l'analyse qui suit.

8.6.2 Analyse

Invariant : H sous-graphe de G ;
E = ensemble des points d'entrée de H ;
$M = G.ens\_som \setminus H.ens\_som$ (sommets qui ne sont pas dans H).
$\forall x \in M$: ($\lambda(x)$, $pred(x)$) = attributs minimaux de x

Arrêt : H vide

Progression : (H non vide, donc E non vide puisque H n'a pas de circuit)
$x \leftarrow element(E)$; -- tous les prédécesseurs de x sont dans M
$\lambda(x) \leftarrow \text{MIN}_{y \in \Gamma^{-1}(x)}(\lambda(y) + v(y,x))$
$pred(x)$=sommet pour lequel ce min est atteint
$M \leftarrow M \cup \{x\}$;
$H.oter\_sommet(x)$;
$E \leftarrow H.points\_entree$

Valeurs initiales : $H \leftarrow G$; $H.oter\_som(1)$;
$M \leftarrow \{1\}$;
$\lambda(1) = 0$; $pred(1)$ non défini;
$E \leftarrow H.points\_entree$

8.6.3 Preuve

Lemme 8.21 *Les valeurs initiales satisfont l'invariant.*

Démonstration Évident, par construction. □

Lemme 8.22 *La progression maintient l'invariant.*

Démonstration Notons H',M',E' les valeurs au début de l'étape. Par hypothèse, l'invariant est vérifié pour ces valeurs. Soit H, M, E les valeurs à la fin de l'étape, et x le sommet sélectionné durant l'étape.

- $H = H'$ privé de x, donc H est un sous-graphe de G.
- $M = M' \cup \{x\}$ donc M est l'ensemble des sommets qui ne sont pas dans H.
- $E =$ ensemble des points d'entrée de H, par construction.
- $\forall y \in M'$: $(\lambda(y), pred(y)) =$ attributs minimaux de y. Il reste donc à montrer que $(\lambda(x), pred(x)) =$ attributs minimaux de x.

On peut écrire :

$$\mathcal{C}_x = \bigcup_{y \in \Gamma^{-1}(x)} \mathcal{C}_{y,x}$$

où $\mathcal{C}_{y,x}$ est l'ensemble de chemins de 1 à x dont l'avant-dernier sommet est y. On a donc, par associativité du minimum :

$$\begin{aligned}
\min_{\mathbf{u} \in \mathcal{C}_x} v(\mathbf{u}) &= \min_{y \in \Gamma^{-1}(x)} \big(\min_{\mathbf{u} \in \mathcal{C}_{y,x}} v(\mathbf{u}) \big) \\
&= \min_{y \in \Gamma^{-1}(x)} \big(\min_{\mathbf{u} \in \mathcal{C}_{y,x}} (v[1 \star y] + v(y,x)) \big) \\
&= \min_{y \in \Gamma^{-1}(x)} \big(\min_{\mathbf{u}' \in \mathcal{C}_y} (v(\mathbf{u}') + v(y,x)) \big) \\
&= \min_{y \in \Gamma^{-1}(x)} (v(y,x) + \min_{\mathbf{u}' \in \mathcal{C}_y} v(\mathbf{u}')) \text{ et, comme } y \in M' : \\
&= \min_{y \in \Gamma^{-1}(x)} (v(y,x) + \lambda(y)) \\
&= \lambda(x)
\end{aligned}$$

De plus, $\lambda(x) = v(\overline{y},x) + \lambda(\overline{y})$, avec $\overline{y} = pred(x)$. Donc $pred(x)$ est bien le prédécesseur de x sur un chemin de 1 à x, de valeur $\lambda(x)$. Enfin, par hypothèse sur G, tout chemin est élémentaire. Donc, $(\lambda(x), pred(x))$ satisfont la définition des attributs minimaux de x. □

Proposition 8.23 *(Correction partielle) A l'arrêt de l'algorithme : $\forall x \in X$ les attributs $(\lambda(x), pred(x))$ sont minimaux.*

Démonstration A l'arrêt, on a $H.vide$ donc $M = G.ens\_som$. Le résultat découle de l'invariant. □

Proposition 8.24 *(Terminaison) L'algorithme s'arrête au bout d'un nombre fini d'étapes.*

Démonstration A la fin de chaque étape, on a $\neg H.vide \Rightarrow E \neq \emptyset$ car le graphe est sans circuit (cf. proposition 7.1). Donc, dans chaque étape, la sélection d'un sommet x est possible. Ce sommet étant ôté de M lors de l'étape, la fonction card$(X \setminus M)$, minorée par 0, décroît strictement d'une unité à chaque pas d'itération. □

8.6.4 Texte de l'algorithme

Il est donné page 121

ALGORITHME ORDINAL RACINE

```
REEL SOMMET::λ ; SOMMET SOMMET::pred ;

ordinal_racine(GRAPHE G; SOMMET x) c'est
   local ENS[SOMMET] M; --   sommets marqués
         ENS[SOMMET] E; --   points d'entrée courants
         GRAPHE H ; --   graphe courant
         SOMMET x, y;
début
  depuis
     x.λ ← 0 ; H ← G; H.otersom(x) ;
     M ← {x0} ;
     E ← H.points_entree
 jusqua H.vide ou sinon E = ∅
  faire
     --   selection d'un sommet de E
     x ← E.element ;
     (x.λ, x.pred) ←  MIN(y.λ+G.v(y,x), G.lst_pred(x)) ;
     M ← M ∪ {x} ;
     H.oter_sommet(x) ;
     E ← H.points_entree
  fait ;
--   si E=∅ alors G a des circuits : algorithme inapplicable
 fin
```

MIN a la même signification que dans l'algorithme de BELLMANN-KALABA (cf. page 109).

8.6.5 Complexité

Il y a exactement n étapes. A chaque étape, un sommet est définitivement "marqué" (mis dans `M`). Si `x` désigne ce sommet, le calcul de x.λ est un calcul de minimum sur l'ensemble des prédécesseurs de `x`, qui nécessite $\mid \Gamma^{-1}(x) \mid -1$ comparaisons. L'algorithme effectue donc $\sum_{x \in X} (\mid \Gamma^{-1}(x) \mid -1) = m - n$ comparaisons (m est le nombre d'arcs de G, avec $m \leq n^2$). C'est donc un algorithme en $O(n^2)$.

8.6.6 Adaptations et améliorations

8.6.6.1 Calcul progressif des attributs

Au lieu de calculer les attributs ($\lambda(x)$, $pred(x)$) d'un seul coup (lorsque x est ôté du graphe), on peut les calculer progressivement en maintenant l'invariant suivant:

$\forall x \in E$: ($\lambda(x)$, $pred(x)$)=attributs minimaux de x. Pour cela, on modifie la progression ainsi(x désigne le sommet sélectionné à l'étape courante) :

```
pour tout y de H.lst_succ(x)
   si x.λ+H.v(x,y) < y.λ
      alors y.λ ← x.λ+H.v(x,y) ;
            y. pred ← x
   fsi
fpour
```

Il est alors facile de vérifier que tout sommet devenant point d'entrée de `H` aura déjà ses attributs minimaux.

8.6.6.2 Mise à jour de l'ensemble des points d'entrée

Dans la progression, l'ensemble des points d'entrée E est recalculé. Or, si H' et E' (resp. H et E) désignent les valeurs au début (resp. à la fin) d'une étape, on a :

H est obtenu en enlevant de H' un sommet $x \in E'$,

E est l'ensemble des points d'entrée de H.

On pourrait exploiter la connaissance de E' pour recalculer E :

$E = E' \setminus \{x\} \cup \{$nouveaux points d'entrée$\}$. Or, les nouveaux points d'entrée ne peuvent être que parmi les *successeurs* de x. On associe à chaque sommet y l'attribut entier nb tel que $nb(y)=$ nombre de prédécesseurs de y dans H. Ces attributs sont gérés de la manière suivante :

- Dès que `y` est "visité" (c'est-à-dire lorsqu'un de ses prédécesseurs est sélectionné) : `y.nb` ←`H.nb_pred(y)` (initialisation).
- Lorsqu'un sommet `x` est ôté de `H`:

```
∀ y ∈ H.lst_succ(x) : y.nb ← y.nb-1;
si y.nb =0 alors E ← E ∪ {y}
```

8.6.6.3 Nouvelle version de l'algorithme

On peut mettre en œuvre l'une ou l'autre de ces deux adaptations. On donne le texte de l'algorithme prenant en compte les deux à la fois.

ALGORITHME ORDINAL RACINE (deuxième version)

```
REEL SOMMET::λ ; SOMMET SOMMET::pred ;

ordinal_racine(GRAPHE G, SOMMET x) c'est
   local ENS[SOMMET] D; --   sommets dehors
         ENS[SOMMET] E; --   points d'entrée courants
         GRAPHE H ; --   graphe courant
         ENT SOMMET::nb ;
         SOMMET x, y;
début
  depuis
     x.λ ← 0 ; H ← G;
     D ← G.ens_som \ {x0} ;
     E ← {x0}
 jusqua H.vide ou sinon E = ∅
  faire
     --   selection d'un sommet de E
     x ← E.element ;
     pour tout y de H.lst_succ(x)
        si y ∈ D
          alors y.nb ← H.nb_pred(y) ;
                y.λ ← x.λ+H.v(x,y) ;
                y.pred ← x
          sinon
            si x.λ+H.v(x,y) < y.λ
              alors y.λ ← x.λ+H.v(x,y) ;
                    y.pred ← x
            fsi ;
            y.nb ← y.nb - 1 ;
            si y.nb=0 alors E ← E ∪ {y} fsi
     fpourtout ;
     E ← E \ {x}
   fait ;
--   si E=∅ alors G a des circuits : algorithme inapplicable
 fin
```

8.6.6.4 Généralisation au cas où 1 n'est pas nécessairement point d'entrée

Il faut alors tenir compte des sommets $x \notin \Gamma^*(1)$, pour lesquels $\lambda(x)$ ne doit pas être défini. L'algorithme doit alors mettre en évidence, parmi les sommets "marqués", ceux qui sont descendants de 1. Le calcul des attributs ($\lambda(x)$, $pred(x)$) se fera en ne tenant compte que des descendants de 1, géré comme dans les algorithmes d'exploration.

Nous laisserons au lecteur le soin d'écrire le texte de l'algortihme, d'établir la preuve et d'étudier la complexité, de manière analogue à l'algorithme ORDINAL-RACINE.

8.7 Valeurs optimales de tout sommet à tout sommet

8.7.1 De la fermeture transitive aux valeurs optimales

Le but de ce problème est de calculer les nombres

λ_{ij}= valeurs minimales des chemins de i à j pour tous couples (i,j) (si $j \notin \Gamma^+(i)$, on peut poser, arbitrairement, $\lambda_{ij} = +\infty$).

Si on revient au problème de la fermeture transitive, il s'agissait alors de calculer les booléens :

$$b_{ij} = (j \in \Gamma^+(i))$$

Il est clair que ces booléens vérifient :

$$(2)\ \forall i \in X, \forall j \in X \ : b_{ij} = \bigvee_{k \in \Gamma^{-1}(j)} [b_{ik} \wedge v(k,j)]$$

où $v(k,j)$ est la **valeur d'existence** (booléenne) de l'arc(k,j).

La comparaison de (1) et (2) montre qu'il y a une analogie entre les deux problèmes, aux transformations suivantes près :

valeur des arcs et des chemins :	**réel** $\cup\{+\infty\}$	**booleen**
opérations sur les arcs :	$+$	$\bigwedge$
opérations sur les chemins :	min	$\bigvee$
valeurs remarquables :	0 (neutre de $+$) $+\infty$ (neutre de min, absorbant de $+$)	**vrai** (neutre de $\bigwedge$) **faux** (neutre de $\bigvee$, absorbant de $\bigwedge$

Ceci suggère l'idée suivante : de tout algorithme d'existence de chemin on peut déduire un algorithme de chemin de valeur minimale.

En particulier, les algorithmes de calcul de la fermeture transitive ont leur équivalent.

8.7.2 Transformation de l'algorithme des puissances

Pour l'algorithme des puissances, il suffit de remplacer :

- L'opération $\circ$ par $\bigwedge+$, dont nous donnons l'algorithme ci-dessous page 125
- L'opération $\bigcup$ par l'opération MIN telle que :

  $G3 = (X,\Gamma 3,V3) = \min(G1,G2)$ est défini par :

$V3(i,j) = \min[V1(i,j), V2(i,j)]$

(par convention, $(i,j) \in \Gamma \Leftrightarrow v(i,j) < +\infty$)

ALGORITHME G1 $\bigwedge$+ G2

```
GRAPHE composer-general(GRAPHE G1, G2)  c'est
 --   pré : G1.lst_som = G2.lst_som; G1 et G2 ont même type de valeurs d'arcs
   local SOMMET x, y, z ; ENS[SOMMET] X
         REEL w1, w2, w3 ;
 début
   Result.creer ; X ← G1.lst_som ; --   ou G2.lst_som
    pourtout x  de X
       pourtout y  de X
         w3 ← Result.v(x,y) ;
         pourtout z  de X
           w1 ← G1.v(x,z) ;
           w2 ← G2.v(z,y) ;
              si w1 ≠  nil  et w2 ≠  nil
                 alors  si w3 ≠  nil
                             alors w3 ← min(w1 + w2, w3)
                             sinon w3 ← w1 + w2
                        fsi
              fsi
         fpourtout ;
         si w3 ≠  nil  alors Result.modif_val(x,y,w3)  fsi
       fpourtout
    fpourtout
 fin
```

La validité de l'algorithme des puissances (adapté du §5.3) découle des propriétés algébriques suivantes, faciles à vérifier :

MIN est associative, commutative

$\bigwedge$+ est associative, possède l'élément neutre $D = (X,\Delta)$ valué par 0, et est distributive par rapport à MIN.

Enfin, $\forall k$: la valeur $v^{(k)}$ de tout arc $(x,y) \in \Gamma^{(k)}$ est égale à la valeur minimale des chemins reliant x à y dans G, de longueur k.

Toutefois, la possibilité d'existence de circuits absorbants oblige à faire les vérifications supplémentaires suivantes pour l'algorithme des puissances (§5.3) : il faut rajouter la condition d'arrêt $k = n$ qui, si elle est obtenue, est équivalente à l'existence d'un circuit absorbant ;

en effet, on a alors :

$$\min(G,\ldots,G^{(n)}) \neq \min(G,\ldots,G^{(n-1)})$$

ce qui prouve qu'il existe un chemin de longueur n (circuit hamiltonien ou chemin non élémentaire), de valeur $v <$ aux valeurs de tous les chemins élémentaires.

8.7.3 Transformation de l'algorithme de ROY-WARSHALL

En nous inspirant des mêmes principes, nous pouvons adapter l'algorithme de ROY-WARSHALL (appelé WARSHALL-FLOYD); nous donnons ci- dessous (page 127) la version avec routage procéduré, suivie des procédures de routage.

Nous ne redémontrerons pas en détail cet algorithme, la démonstration étant en tous points analogue à celle de l'algorithme de ROY- WARSHALL avec routage ; on a les 3 lemmes suivants :

1. A l'issue de l'algorithme,
   $$\forall z \in X, \forall y \in X \ : y.R[z] \neq \textbf{nil si et seulement si } (y,z) \in \Gamma^+$$
2. Si G n'a pas de circuit absorbant alors, pour tout couple $(y,z) \in \Gamma^+$ on a, à l'issue de l'algorithme :
   $y.R[z]$ est successeur de y sur un chemin de valeur minimale de y à z.
3. Si G n'a pas de circuit absorbant alors, pour tout couple $(y,z) \in \Gamma^+$, tout chemin de valeur minimale de y à z est *élémentaire*

On déduit de ces trois lemmes que : si G n'a pas de circuit absorbant, les tables de routage R permettent d'obtenir les tracés de chemins de valeur minimale entre tout couple de sommets.

Cas où G a des circuits absorbants

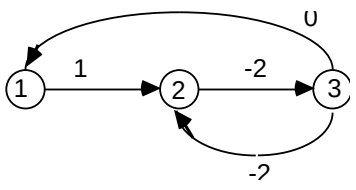
Cette situation peut être détectée à l'examen du graphe `Result` à l'issue de l'algorithme. En effet, l'existence d'un circuit absorbant passant par un sommet x se manifestera par la propriété : `Result.v(x,x)` < 0

Si tel est le cas, les valeurs des arcs de `Result` ne peuvent être interprétées comme des valeurs minima, puisque les valeurs des chemins ne sont pas nécessairement bornées inférieurement. Toutefois, les fonctions de routage permettent d'obtenir le tracé de circuits absorbants, et donc d'en éliminer. En effet, soit $x \in X$ tel que `Result.v(x,x)` < 0. La suite définie par

$$x_1 = x.R[x], x_2 = x_1.R[x], \ldots, x_j = x_{j-1}.R[x]$$

comporte une répétition, laquelle correspond à un circuit absorbant.

Exemple :



Sur une mise en œuvre matricielle ($y.R[z] = R[y,z]$) on obtient, par applications successive de θ_1, θ_2, θ_3 (à gauche : matrice des valeurs, à droite matrice R) :

Algorithme WARSHALL-FLOYD

```
TABLE[SOMMET,SOMMET] SOMMET::R ;
GRAPHE warshall_floyd(GRAPHE G)  c'est
   local SOMMET x, y, z ; ENS[SOMMET] X ;
         REEL vy, vz, w ;
 début
   Result  ← G ; X ← G.lst_som ;
   initroutage ;
    pourtout x  de X
      pourtout y  de X
         vy  ← Result.v(y,x) ;
        si vy ≠  nil alors
           pourtout z  de X
             vz ← Result.v(x,z) ;
              si vz ≠  nil alors
                w ← Result.v(y,z) ;
                 si w ≠  nil
                    alors  si vy + vz < w
                                  alors w ← vy + vz ;
                                        maj_routage(y,z,x)
                           fsi
                    sinon w ← vy + vz ; maj_routage(y,z,x)
                 fsi ;
                Result.modif_val(y,z,w)
              fsi
           fpourtout
        fsi
      fpourtout
    fpourtout
 fin
```

avec :

$$initroutage \ : \forall z \in X, \forall y \in X \ : y.R[z] \leftarrow \begin{cases} \textbf{nil} & \textbf{si} \quad (y,z) \notin \Gamma \\ z & \textbf{si} \quad (y,z) \in \Gamma \end{cases}$$

$$majroutage(y,z,x) \ : y.R[z] \leftarrow y.R[x]$$

	1	2	3
1		1	
2			-2
3	0	-2	

	1	2	3
1		2	
2			3
3	1	2	

$\theta_1 \downarrow$

	1	2	3
1		1	
2			-2
3	0	-2	

	1	2	3
1		2	
2			3
3	1	2	

$\theta_2 \downarrow$

	1	2	3
1		1	-1
2			-2
3	0	-2	-4

	1	2	3
1		2	2
2			3
3	1	2	2

$\theta_3 \downarrow$

	1	2	3
1	-1	-3	-5
2	-2	-4	-6
3	-4	-6	-8

	1	2	3
1	2	2	2
2	3	3	3
3	2	2	2

d'où les suites (à partir de 1, 2 et enfin 3) :

1, R[1,1]=2, R[2,1]=3, R[3,1]=2 $\longrightarrow$ circuit 2,3,2

2, R[2,2]=3, R[3,2]=2 $\longrightarrow$ circuit 2,3,2

3, R[3,3]=2, R[2,3]=3 $\longrightarrow$ circuit 3,2,3

Le circuit absorbant 2,3,2 (valeur -4) est détecté, mais pas le circuit 1,2,3,1 (valeur -1).

8.8 Algèbres de chemins : transformations d'algorithmes

Au paragraphe précédent, nous avons mis en évidence la possibilité de déduction d'un algorithme de chemin de valeur minimale à partir d'un algorithme d'existence de chemin, grâce à la transformation :

$$(\{\textbf{vrai},\textbf{faux}\}, \vee, \wedge) \longrightarrow (\mathbb{R} \cup \{+\infty\}, \min, +)$$

On peut dégager une structure abstraite, appelée **dioïde** ou encore **semi-anneau**, à partir de laquelle d'autres transformations pourront être déduites.

Définition 8.25 *On appelle dioïde un triplet* $(S, \oplus, \otimes)$ *où* S *est en ensemble, muni des deux lois de composition internes* $\oplus$ *et* $\otimes$ *, et satisfaisant aux axiomes suivants :*

A1 $\oplus$ *est associative, commutative et possède un élément neutre* α*.*

A2 $\otimes$ *est associative et possède un élément neutre* e*.*

A3 $\otimes$ *est distributive à droite et à gauche par rapport à* $\oplus$*.*

A4 α *est élément absorbant de* $\otimes$*.*

La structure $(E, \otimes, \preceq)$ introduite au paragraphe 8.1 rentre dans ce cadre. En effet, la relation d'ordre $\preceq$ définit une loi $\oplus$, de la manière suivante:

$$\begin{aligned} w_1 \oplus w_2 \quad = \quad & w_1 \text{ si } w_1 \preceq w_2 \\ & w_2 \text{ si } w_2 \preceq w_1 \end{aligned}$$

Les axiomes A1 à A4 sont alors vérifiés sur $(E, \oplus, \otimes)$.

pour A1: – La vérification de l'associativité et de la commutativité de $\oplus$ sont immédiates

– α est maximum pour $\preceq$, donc $w \oplus \alpha = w$

pour A2: les hypothèses sur $\otimes$ sont identiques

pour A3: $$w_1 \otimes (w_2 \oplus w_3) = \begin{cases} w_1 \otimes w_2 & \text{si} \quad w_2 \preceq w_3 \text{ compatibilté } \preceq \text{ avec } \otimes \\ w_1 \otimes w_3 & \text{si} \quad w_3 < w_2 \end{cases}$$

D'autre part,

$$\begin{aligned} w_2 \preceq w_3 \quad \Rightarrow \quad & w_1 \otimes w_2 \preceq w_1 \otimes w_3 \\ \Rightarrow \quad & (w_1 \otimes w_2) \oplus (w_1 \otimes w_3) = w_1 \otimes w_2 \end{aligned}$$

et de même

$$\begin{aligned} w_3 \preceq w_2 \quad \Rightarrow \quad & w_1 \otimes w_3 \preceq w_1 \otimes w_2 \\ \Rightarrow \quad & (w_1 \otimes w_2) \oplus (w_1 \otimes w_3) = w_1 \otimes w_3 \end{aligned}$$

Dans les deux cas, on a donc :

$$w_1 \otimes (w_2 \oplus w_3) = (w_1 \otimes w_2) \oplus (w_1 \otimes w_3)$$

On vérifie de même la distributivité à droite.

Par contre, la structure de dioïde $(E, \oplus, \otimes)$ ne rentre pas toujours dans le cadre de la structure définie au §8.1: on ne peut pas toujours déduire une relation d'ordre total de la loi de composition $\oplus$. Ceci est illustré notamment par les exemples 8 et 9 ci-après. La structure de dioïde est donc plus abstraite, mais aussi plus riche - puisqu'elle permet de rendre compte de situations plus nombreuses - que celle d'ensemble totalement ordonné.

Dans la structure de dioïde, la notion de circuit absorbant se généralise comme suit : un circuit est dit **absorbant** si sa valeur v vérifie :

$$v \oplus e \neq e$$

En fonction du choix du dioïde $(S, \oplus, \otimes)$ on peut interpréter concrètement la valeur (dans S) $v(x,y)$ et la valeur (dans S) des expressions :

$$\forall x \in X, \forall y \in X \ : \lambda(x,y) = \bigoplus_{\{\mathbf{u} \in X^* | \mathbf{u}=[x*y]\}} v(\mathbf{u}) = \bigoplus_{\{\mathbf{u} \in X^* | \mathbf{u}=[x*y] \wedge v(\mathbf{u}) \neq \alpha\}} v(\mathbf{u})$$

où, si $\mathbf{u} = (x_1, \ldots, x_p)$, on a posé

$$v(\mathbf{u}) = \bigotimes_{j=1,\ldots,p-1} v(x_j, x_{j+1})$$

Exemples

1) Existence d'au moins un chemin

$S \equiv \{\mathbf{faux}, \mathbf{vrai}\}; \oplus \equiv \vee; \otimes \equiv \wedge; \alpha = \mathbf{faux}; e = \mathbf{vrai}$

$v(x,y) = \mathbf{si}(x,y) \in \Gamma$ **alors vrai sinon faux**;

$\forall s \in S, s \vee \mathbf{vrai} = \mathbf{vrai}$ donc il n'y a pas de circuit absorbant ;

$\lambda(x,y) =$ **vrai si et seulement si** il existe un chemin de x à y dans G.

2) Chemin de valeur minimale

$S = \mathbb{R} \cup \{+\infty\}; \oplus = \min; \otimes = +; \alpha = +\infty; e = 0;$

$v(x,y) = \mathbf{si}(x,y) \in \Gamma$ **alors** valeur numérique de l'arc
sinon $+\infty$

Un circuit de valeur v telle que $\min(v,0) \neq 0$ (c'est à dire $v < 0$) est absorbant.

$\lambda(x,y) =$ valeur minimale des chemins de x à y (s'il n'y a pas de circuit absorbant)

3) chemin de valeur maximale

$S = \mathbb{R} \cup \{-\infty\}; \oplus = \max; \otimes = +; \alpha = -\infty; e = 0;$

$v(x,y) = \mathbf{si}\ (x,y) \in \Gamma$ **alors** valeur numérique de l'arc
sinon $-\infty$

Un circuit de valeur v telle que $\max(v,0) \neq 0$ (c'est à dire $v > 0$) est absorbant.

$\lambda(x,y) =$ valeur maximale des chemins de x à y (s'il n'y a pas de circuit absorbant).

4) Chemin de fiabilité maximale

$S = [0,1]; \oplus = \max; \otimes = \times; \alpha = 0; e = 1;$

$v(x,y) =$ **si** $(x,y) \in \Gamma$ **alors** fiabilité de l'arc
sinon 0

Toute valeur $p \in S$ vérifie $\max(p,1) = 1$ donc il n'y a pas de circuit absorbant.

$\lambda(x,y) =$ fiabilité maximale des chemins de x à y.

5) Chemin de capacité maximale

$S = \mathbb{R}^+ \cup \{+\infty\}; \oplus = \max; \otimes = \min; \alpha = 0; e = +\infty;$

$v(x,y) =$ **si** $(x,y) \in \Gamma$ **alors** capacité de l'arc
sinon 0

Toute valeur $c \in S$ vérifie $\max(c, +\infty) = +\infty$ donc il n'y a pas de circuit absorbant.

$\lambda(x,y) =$ capacité maximale des chemins de x à y.

6) Chemin de fiabilité minimale

$S = [0,1] \cup \{\alpha\} (\alpha \notin [0,1]);$

$\oplus = \min$ sur $[0, 1]$, prolongée par $\forall x \in S \ : x \oplus \alpha = x;$

$\otimes = \times$ sur $[0, 1]$, prolongée par $\forall x \in S \ : x \otimes \alpha = \alpha; e = 1;$

$v(x,y) =$ **si** $(x,y) \in \Gamma$ **alors** fiabilité de l'arc
sinon α

Toute valeur $p \in S, p \neq 1, p \neq \alpha$ vérifie $p \oplus e = \min(p,1) = p$ donc tout circuit de fiabilité < 1 est absorbant.

$\lambda(x,y) =$ fiabilité minimale des chemins de x à y (s'il n'y a pas de circuit absorbant).

7) Chemin de capacité minimale

$S = \mathbb{R}^+ \cup \{+\infty\} \cup \{\alpha\} (\alpha \notin \mathbb{R}^+ \cup \{+\infty\});$

$\oplus = \min$ sur $\mathbb{R}^+ \cup \{+\infty\}$, prolongée par $\forall x \in S \ : x \oplus \alpha = x;$

$\otimes = \min$ sur $\mathbb{R}^+ \cup \{+\infty\}$, prolongée par$\forall x \in S \ : x \otimes \alpha = \alpha;$

$e = +\infty;$

$v(x,y) =$ **si** $(x,y) \in \Gamma$ **alors** capacité de l'arc
sinon α

Toute valeur $p \in S, p \neq +\infty, p \neq \alpha$, vérifie $p \oplus e = \min(p, +\infty) = p$ donc tout circuit de capacité $< \infty$ est absorbant. (toutefois, $p^2 \oplus p \oplus e = p \oplus e$, où $p^2 = p \otimes p$, donc un circuit n'est absorbant "qu'une fois").

$\lambda(x,y) =$ capacité minimale des chemins de x à y (s'il n'y a pas de circuit absorbant).

8) Dénombrement des chemins

$S = \mathbb{N}; \oplus = +; \otimes = \times; \alpha = 0; e = 1$

$v(x,y) = \textbf{si } (x,y) \in \Gamma \quad \textbf{alors } 1$
$\qquad\qquad\qquad\qquad\quad \textbf{sinon } 0$

Toute valeur $n \in \mathbb{N}$ vérifie $n + 1 \neq 1$ donc tout circuit est absorbant.

$\lambda(x,y) =$ nombre de chemins distincts de x à y (s'il n'y a pas de circuit).

9) Énumération des chemins élémentaires (exemple difficile) :

Soit $\overline{X}^+$ l'ensemble de toutes les suites sans répétition d'éléments de X, ayant au moins un élément. Notons $n = |X|$. S est alors l'ensemble constitué des $n(n-1)$ éléments $s_{x,y}$, $x \in X$, $y \in X$, $x \neq y$, où $s_{x,y}$ est un ensemble d'éléments de $\overline{X}^+$ de la forme $[x*y]$, des n éléments $s_x = \{[x]\}$ $(x \in X)$ et de l'élément $e = \bigcup_{x \in X} s_x$ (identifié à X). On a donc $|S| = n^2 + 1$. Notons qu'on peut avoir $s_{x,y} = \emptyset$ pour certains couples (x,y), $x \neq y$. En outre,

$\oplus = \cup; \otimes =$ multiplication latine ; $\alpha = \emptyset$

La multiplication latine est définie comme suit :

- $\forall s \in S \ : s \otimes \alpha = \alpha \otimes s = \alpha$ ($\alpha = \emptyset$ est donc l'élément absorbant de S).
- $\forall x, \forall y, \forall z, \forall t, \ z \neq t \ : s_{x,z} \otimes s_{t,y} = \alpha$.
- $\forall x, \forall z \ : s_{x,z} \otimes s_{z,x} = \alpha$
- $\forall x, \forall y \ ,\forall z \ , \ y \neq z \ : s_{x,y} \otimes s_z = \alpha$
- $\forall x, \forall y \ ,\forall z \ , \ x \neq z \ : s_z \otimes s_{x,y} = \alpha$
- $\forall x, \forall y \ : s_x \otimes s_{x,y} = s_{x,y}$
- $\forall x, \forall y \ : s_{x,y} \otimes s_y = s_{x,y}$
- $\forall x, \forall y, \forall z \ , \ x \neq y \ : s_{x,z} \otimes s_{z,y} = \{u \in \overline{X}^+ | u = [x*z*y], [x*z] \in s_{x,z}, [z*y] \in s_{z,y}$

Exemple :

si $s_{1,4} = \{[1,4],[1,2,4]\}$ et $s_{4,5} = \{[4,5],[4,2,5],[4,3,5]\}$ on a :
$s_{1,4} \otimes s_{4,5} = \{[1,4,5],[1,4,2,5],[1,4,3,5],[1,2,4,5],[1,2,4,3,5]\}$.

Il est facile de vérifier que l'élément neutre de $\otimes$ est e.

La valuation du graphe est alors :

$v(x,y) = \textbf{si}(x,y) \in \Gamma \quad \textbf{alors } \{[xy]\}$
$\qquad\qquad\qquad\qquad\quad \textbf{sinon } \emptyset$

Par ailleurs, d'après la définition de $\otimes$, la valeur de tout circuit est égale à α, donc vérifie $\alpha \cup e = e$. Il n'y a donc pas de circuit absorbant.

Enfin,

$\lambda(x,y) =$ ensemble des chemins élémentaires de x à y

Le lecteur intéressé trouvera, dans ([GM79], chapitre 3) d'autres exemples et d'intéressants développements sur les algèbres de chemin.

Pour conclure, on peut se poser la question de savoir quels sont les algorithmes de cheminement optimal qui peuvent être transformés, sur la base de la structure de dioïde appropriée, pour résoudre les problèmes correspondants. Sans entrer dans les détails, il s'avère que les algorithmes n'utilisant pas une procédure de sélection d'un élément réalisant un "optimum" peuvent être directement adaptés, en utilisant les opérations $\otimes$ (à la place de $+$) et $\oplus$ (à la place de min), ainsi que α (à la place de $+\infty$) et e (à la place de 0). Ainsi, on peut adapter directement Ford, Bellmann-Kalaba, Ford Ordinal, Puissances, Warshall. Mais ce n'est pas le cas de Dijkstra : en effet, cet algorithme nécessite que la loi $\oplus$ vérifie : $\forall a, \forall b \ : a \oplus b \in \{a,b\}$ (sélection du "meilleur" élément en attente). C'est bien le cas lorsque $\oplus \equiv ou$, ou $\oplus \equiv \min$, ou $\oplus \equiv \max$ (exemples 1 à 7) mais ce n'est plus le cas pour $\oplus \equiv +$ ou $\oplus \equiv \cup$ (exemples 8 et 9).

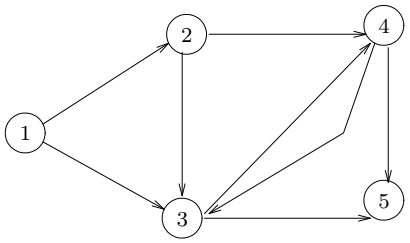
De plus, lorsque la loi $\oplus$ a cette propiété, la condition d'application de l'algorithme de DIJKSTRA s'exprime par :

$$\forall (x,y) \in \Gamma \ : v(x,y) \oplus e = e$$

Elle est donc toujours vérifiée pour les exemples 1,4,5 ;
vérifiée si $v(x,y) \geq 0$ pour l'exemple 2
$v(x,y) \leq 0$ pour l'exemple 3
$v(x,y) = 1$ pour l'exemple 6
$v(x,y) = +\infty$ pour l'exemple 7 (donc irréalistes pour ces deux exemples)

Pour illustrer la puissance de ces transformations d'algorithme, nous donnons un exemple d'application de l'algorithme de Bellmann-Kalaba pour l'énumération des chemins élémentaires issus du sommet 1 vers chacun des autres sommets (exemple 9). On doit donc calculer : $\forall i$, $i \neq 1$: λ_i=ensemble des chemins élémentaires de 1 à i. La relation de récurrence est ici :

$$\begin{aligned} \lambda_1 &= \{[1]\} \\ \forall k, \forall i \ : \ \lambda_i^k &= \bigcup_{j=1}^{n} \lambda_j^{k-1} \otimes v(j,i) \end{aligned}$$



La matrice du graphe est :

	1	2	3	4	5
1	{[1]}	{[1,2]}	{[1,3]}	α	α
2	α	α	{[2,3]}	{[2,4]}	α
3	α	α	α	{[3,4]}	{[3,5]}
4	α	α	{[4,3]}	α	{[4,5]}
5	α	α	α	α	α

L'exécution de l'algorithme donne alors :

1	2	3	4	5
{[1]}	{[1,2]}	{[1,3]}	α	α
{[1]}	{[1,2]}	{[1,3]} [1,2,3]}	{[1,2,4] [1,3,4]}	{[1,3,5]}
{[1]}	{[1,2]}	{[1,3]} [1,2,3] [1,2,4,3]}	{[1,2,4] [1,3,4] [1,2,3,4]}	{[1,3,5] [1,2,3,5] [1,2,4,5] [1,3,4,5]}
{[1]}	{[1,2]}	{[1,3]} [1,2,3] [1,2,4,3]}	{[1,2,4] [1,3,4] [1,2,3,4]}	{[1,3,5] [1,2,3,5] [1,2,4,5] [1,3,4,5] [1,2,4,3,5] [1,2,3,4,5]}
{[1]}	{[1,2]}	{[1,3]} [1,2,3] [1,2,4,3]}	{[1,2,4] [1,3,4] [1,2,3,4]}	{[1,3,5] [1,2,3,5] [1,2,4,5] [1,3,4,5] [1,2,4,3,5] [1,2,3,4,5]}

stabilité atteinte.

Chapitre 9

Problèmes d'ordonnancement

9.1 Nature des problèmes

Un projet peut être décomposé en n tâches A_1,A_2, ... ,A_n telles que :

- Chaque tâche est indivisible
- Chaque tâche est munie d'une durée *connue* : d_i = durée de A_i
- Chaque tâche est soumise à un ensemble de *contraintes temporelles* qui peuvent toujours s'exprimer en fonction de contraintes sur la date de début d'exécution de la tâche

Ces contraintes sont de deux types :

a) Contraintes *"déterministes"* **(ou** *"potentiel"***)** :

La date de début t_i de la tâche A_i intervient *au plus tôt* (resp. *au plus tard*) à une date donnée *a priori* ou liée à la réalisation d'autres tâches du projet.

Exemples

. la tâche A_i ne peut pas commencer avant la date T :

$$t_i \geq T$$

. la tâche A_j ne peut pas commencer avant l'achèvement de la tâche A_i :

$t_j \geq t_i + d_i$ soit encore : $t_j - t_i \geq d_i$

L'ensemble des contraintes "potentiel" définit un ensemble conjonctif de contraintes, qui s'exprime sous la forme d'un système simultané d'inéquations du premier degré de la forme

$$t_j - t_i \geq a_{i,j} \quad (i,j \in \{0,\ldots,n+1\})$$

Par convention, $t_0 = 0$: date de *début du projet*

t_{n+1} : date de fin d'exécution du projet
(lorsque toutes les tâches sont terminées)
= durée totale d'exécution du projet

b) Contraintes *"non déterministes"* **(ou** *"disjonctives"***)** :

Elles expriment des incompatibilités entre sous-ensembles de l'ensemble des tâches, mais laissent un degré de choix quant à leur séquencement : chaque contrainte est exprimée par un système d'inéquations, d'inconnues $(t_i)_{0\leq i\leq n+1}$ liées par des **ou** (une des inéquations du système doit être vérifiée)

Exemples

. la tâche A_i et la tâche A_j doivent être totalement disjointes :

$$t_i \geq t_j + d_j \textbf{ ou } t_j \geq t_i + d_i$$

. une des 4 tâches A_i,A_j,A_k,A_ℓ ne peut être effectuée qu'après la date T:

$$t_i \geq T \textbf{ ou } t_j \geq T \textbf{ ou } t_k \geq T \textbf{ ou } t_\ell \geq T$$

Définition 9.1 *On appelle ordonnancement compatible (sous-entendu "avec les contraintes") un vecteur de* $\mathbf{N}^{n+2}$ *:*
$T = (t_0,t_1,\ldots,t_n,t_{n+1})$*, où :*
t_i *= date de début d'exécution de la tâche* $A_i (i = 1,\ldots,n)$
t_0 et t_{n+1} *respectivement dates de début et de fin d'exécution du projet, respectant les contraintes de chacune des tâches.*

Problème 9.1 *Parmi tous les ordonnancements compatibles avec un ensemble de tâches et de contraintes données, en trouver un de durée minimale, c'est à dire tel que* t_{n+1} *soit minimum.*

Problème 9.2 *Parmi tous les ordonnancements compatibles avec un ensemble de tâches et de contraintes données, en trouver un de durée donnée, c'est à dire tel que* $t_{n+1} = T$ *donnée.*

Dans la suite de ce chapitre, on ne s'intéresse qu'aux contraintes *"déterministes"* ; les autres types de contraintes nécessitent l'utilisation de techniques combinatoires, par exemple de recherche arborescente telles que les Procédures de Séparation et Évaluation Progressives (P.S.E.P.).

9.2 Modélisation à l'aide de graphe

9.2.1 graphe potentiel-tâche

On associe au problème un graphe valué, défini de la manière suivante :

– $X = \{x_0,x_1, \ldots ,x_n,x_{n+1}\}$ où $x_i \equiv A_i \ (1 \leq i \leq n)$, $x_0 \equiv$ début, $x_{n+1} \equiv$ fin

– Γ = ensemble des contraintes, au sens suivant :

si $t_j - t_i \geq a_{i,j}$ est une contrainte, alors $(x_i,x_j) \in \Gamma$ avec la valeur $a_{i,j}$

Exemples de contraintes et d'arc correspondant

▷ Contrainte "*au plus tôt*" sur A_j :

$$t_j - t_i \geq a_{i,j} \quad (a_{i,j} \geq 0)$$

$$x_i \xrightarrow{a_{i,j}} x_j$$

▷ Contrainte "*au plus tard*" sur A_i :

$$t_i - t_j \leq a_{i,j} \quad (a_{i,j} \geq 0), \text{ c'est-à-dire : } t_j - t_i \geq -a_{i,j}$$

$$x_i \overset{-a_{i,j}}{\longrightarrow} x_j$$

▷ Contraintes implicites relatives au début :
$\forall i (1 \leq i \leq n+1) : t_i \geq 0 \Longleftrightarrow (x_0, x_i) \in \Gamma$, de valeur 0

▷ Contraintes implicites relatives à la fin :
$\forall i (1 \leq i \leq n) : t_{n+1} \geq t_i + d_i \Longleftrightarrow (x_i, x_{n+1}) \in \Gamma$, de valeur d_i

Parmi toutes ces contraintes, certaines peuvent être redondantes, c'est-à-dire conséquences d'autres contraintes, de manière plus ou moins immédiate; ceci sera plus particulièrement vu au §9.6 de ce chapitre

9.2.2 Exemple

Soit un projet composé de 6 tâches :

tâche	A_1	A_2	A_3	A_4	A_5	A_6
durée	10	12	6	20	12	7

avec les contraintes :

A_1 : peut commencer dès le début des travaux.

A_2 : commence au plus tôt 7 u.t. après le début de A_1.

A_3 : enchaînement sans délai avec A_2.

A_4 : commence au plus tôt 3 u.t. après l'achèvement de A_5 ; la seconde moitié de A_4 commence au plus tôt 13 u.t. après l'achèvement de A_3.

A_5 : commence au plus tôt lorsque les 3/4 de A_2 sont achevés, au plus tard 6 u.t. après l'achèvement de A_2 et au plus tôt 3 u.t. après l'achèvement de A_6.

A_6 : commence au plus tôt 20 u.t. après le début des travaux.

Liste des inéquations traduisant ces contraintes :

$t_1 \geq 0$;

$t_2 - t_1 \geq 7$;

$t_3 \geq t_2 + 12$ **et** $t_3 \leq t_2 + 12 \Longleftrightarrow t_3 - t_2 \geq 12$ **et** $t_2 - t_3 \geq -12$;

$t_4 - t_5 \geq 15; t_4 + \frac{1}{2} \times 20 \geq t_3 + 19 \Longleftrightarrow t_4 - t_3 \geq 9$;

$t_5 \geq t_2 + \frac{3}{4} \times 12 \Longleftrightarrow t_5 - t_2 \geq 9$;

$t_5 \leq t_2 + 18 \Longleftrightarrow t_2 - t_5 \geq -18; t_5 - t_6 \geq 10$;

$t_6 \geq 20$

et les contraintes "implicites" :

$t_i \geq 0 \quad (i = 2,3,4,5)$;

$t_7 - t_i \geq d_i \ (i = 1, \ \ldots \ ,6)$

Dans cet exemple, les contraintes $(x_0,x_i),i = 2,3,4,5$ ainsi que les contraintes $(x_i,x_7),i = 2,3,5,6$ sont évidemment superflues (cf §9.6 ci-après).

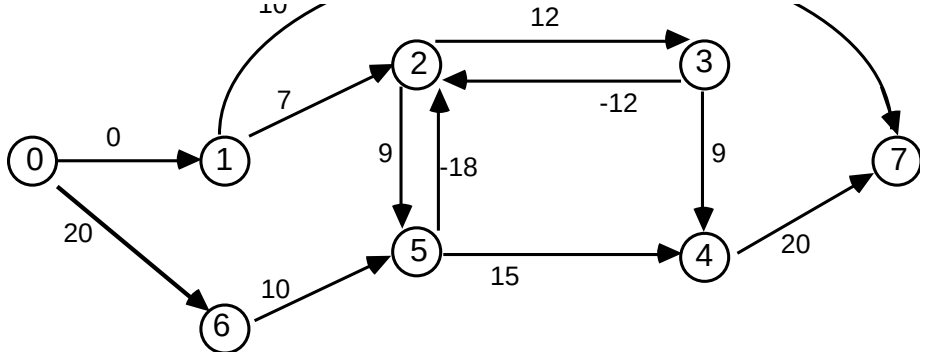


FIG. 9.1 – *graphe potentiel-tâche*

9.3 Ordonnancements particuliers; chemin critique; marges.

Théorème 9.2 *Soit* $\Lambda = (\lambda_0,\lambda_1, \ \ldots \ ,\lambda_n,\lambda_{n+1})$ *l'ordonnancement défini par :*

$\lambda_0 = 0$

$\lambda_i =$ *valeur maximale des chemins de* x_0 *à* x_i $(i = 1,\ldots,n+1)$.

C'est un ordonnancement compatible, vérifiant :

$\lambda_i \leq t_i (i = 0,\ldots,n+1)$

quelque soit l'ordonnancement compatible

$T = (t_0 = 0,t_1, \ \ldots \ ,t_n,t_{n+1})$.

En particulier, il est de durée minimale.

Démonstration D'après le théorème 8.7, appliqué au problème des valeurs additives *maximales*, les attributs λ_i $(i = 0,\ldots,n+1)$ constituent la solution minimale du système d'équations:

$$x_i = \begin{cases} 0 \text{ si } i = 0 \\ \max_{x_j \in \Gamma^{-1}(x_i)}(x_j + v(j,i)) \text{si } i = 2,\ldots,n \end{cases}$$

Il en résulte, d'une part : $\lambda_j - \lambda_i \geq a_{i,j}$ sur tout arc (x_i,x_j), donc l'ordonnancement est compatible.

D'autre part, tout ordonnancement compatible $T = (t_0, \ldots, t_{n+1})$ est aussi solution de ce système, et donc :

$\forall i\ (1 \leq i \leq n)$, $\lambda_i \leq t_i$

En particulier, $\lambda_{n+1} \leq t_{n+1}$. □

L'ordonnancement Λ s'appelle **ordonnancement "au plus tôt"** car λ_i est la date de début au plus tôt compatible avec les contraintes pour la tâche A_i.

Définition 9.3 *Un chemin de valeur maximale entre x_0 (début) et x_{n+1} (fin) s'appelle* **chemin critique**, *et les sommets qui y appartiennent* **tâches critiques**.

En effet, si une tâche située sur ce chemin est retardée alors la date finale du projet est retardée.

Soit maintenant une date finale F fixée à priori. On cherche les dates au plus tard, λ'_i, d'un ordonnancement compatible permettant l'achèvement des travaux à la date fixée.

Théorème 9.4 *Si $F < \lambda_{n+1}$, il n'existe pas d'ordonnancement compatible se terminant à la date F.*
Si $F \geq \lambda_{n+1}$, alors l'ordonnancement $\Lambda' = (\lambda'_0, \ldots, \lambda'_{n+1})$ défini par

$$\begin{aligned} \lambda'_0 &= 0 \\ \lambda'_i &= F - \alpha_i (i = 1, \ldots, n+1). \end{aligned}$$

où α_i = valeur maximale des chemins de x_i à x_{n+1},
répond à la question.

Démonstration Si $F < \lambda_{n+1}$, le résultat est évident puisque λ_{n+1} est la date au plus tôt d'achèvement des travaux.

Supposons donc $F \geq \lambda_{n+1}$. Les problèmes de cheminement dans G *aboutissant* au sommet x_{n+1} se ramènent aux problèmes de cheminement *issus du sommet* x_{n+1} dans le graphe transposé G^t. Par conséquent, d'après le théorème 8.7, les attributs α_i $(i = 0, \ldots, n+1)$ constituent la solution minimale du système d'équations:

$$x_i = \begin{cases} 0 \text{ si } i = n+1 \\ \max_{x_j \in \Gamma(x_i)}(x_j + v(i,j)) \text{si } i = 2, \ldots, n \end{cases}$$

Il en résulte, d'une part : $\alpha_i - \alpha_j \geq a_{i,j}$ sur tout arc (x_i,x_j), c'est-à-dire, d'après la définition des λ'_i : $\lambda'_j - \lambda'_i \geq a_{i,j}$ sur tout arc (x_i,x_j), donc l'ordonnancement est compatible.

D'autre part, pour tout ordonnancement compatible $T = (t_0 \ \ldots \ t_{n+1})$, avec $t_{n+1} = F$, les valeurs $\beta_i = F - t_i$ sont aussi solution de ce système, et donc : $\forall i\ (1 \leq i \leq n)$, $\alpha_i \leq \beta_i$ d'où :

$\forall i\ (1 \leq i \leq n)$, $\lambda'_i \geq t_i$

□

Définition 9.5 *Soit Λ l'ordonnancement au plus tôt et Λ' l'ordonnancement au plus tard relatif à une date finale $F \geq \lambda_{n+1}$. On appelle* **marge totale** *de la tâche A_i (relativement à F) la différence $m_i = \lambda'_i - \lambda_i$. C'est le retard maximum que peut prendre l'achèvement de A_i sans compromettre la date de réalisation du projet prévue $\lambda'_{n+1} = F$.*

Remarque. Si la date d'achèvement de A_i est retardée d'une durée $\leq m_i$, ce retard peut modifier l'ordonnancement des tâches ultérieures sur tout chemin de x_i à x_{n+1}. Les marges totales des tâches ne sont donc pas indépendantes. Il vaut donc mieux parler de **marge par chemin** traduisant le retard global tel que la somme des retards des tâches situées sur ce chemin reste inférieure à ce retard global sous peine de retarder la date λ'_{n+1}. Pour un chemin $\mathbf{u}$, $m(\mathbf{u}) = \max_{x_i \in \mathbf{u}}(m_i)$

Définition 9.6 *Soit $T = (t_0, \ \ldots \ , t_{n+1})$ un ordonnancement compatible donné. La* **marge libre** *de A_i , relativement à T , est définie par :*

$$\mu_i = \min_{x_j \in \Gamma(x_i)} (t_j - t_i - a_{i,j})$$

C'est le retard maximum que peut prendre l'achèvement de A_i sans compromettre la date de début prévue pour les autres tâches.

9.4 Exemple

Nous traitons complètement l'exemple donné au §9.2 (figure 9.1)

tâche	A_1	A_2	A_3	A_4	A_5	A_6	A_7
Λ	0	12	24	45	30	20	65
α_i	51	44	32	20	35	45	0
$\Lambda' \mid F = 65$	14	21	33	45	30	20	65
$m_i \mid F = 65$	14	9	9	0	0	0	/
$\mu_i(\Lambda)$	5	0	0	0	0	0	/

Marges totales par chemin :

$x_0\ x_1\ x_2\ x_3\ x_4\ x_7 \quad = \quad 14$
$x_0\ x_1\ x_2\ x_5\ x_4\ x_7 \quad = \quad 14$
$x_0\ x_6\ x_5\ x_2\ x_3\ x_4\ x_7 \quad = \quad 9$
$x_0\ x_6\ x_5\ x_4\ x_7 \quad = \quad 0$ (chemin critique)

9.5 Choix des algorithmes de résolution

La recherche des ordonnancements "au plus tôt" ou "au plus tard" relatif à une date F nécessite le calcul des valeurs maximales des chemins issus du sommet x_0 ou aboutissant au sommet x_{n+1}. Au chapitre précédent, nous avons décrit quelques algorithmes résolvant ces

problèmes. Nous allons donner quelques indications sur le choix à effectuer, compte-tenu de la nature des contraintes.

1er cas. Il n'y a que des contraintes "au plus tôt". Dans ce cas, le graphe obtenu ne doit pas comporter de circuit, sans quoi il y aurait des contraintes incompatibles : il n'y aurait alors pas d'ordonnancement compatible, puisque tout circuit est un circuit absorbant (les arcs étant tous de valeur positive ou nulle).

Comme par ailleurs le sommet x_0 est racine, et le sommet x_{n+1} anti-racine (racine du graphe transposé), l'algorithme ORDINAL-RACINE est le mieux adapté :

- pour l'ordonnancement au plus tôt : avec x_0 comme sommet de départ, il suffit de remplacer MIN par MAX dans le calcul d'une marque.
- pour l'ordonnancement au plus tard : avec x_{n+1} comme sommet de départ, il suffit de remplacer MIN par MAX dans le calcul d'une marque et de permuter les opérations *lst_pred* et *lst_succ* (puisque l'on cherche les chemins de valeur maximale issus de x_{n+1} dans le graphe transposé).

Remarque : Si $x.\alpha$ est la valeur maximale des chemins de x à x_{n+1} , ces valeurs sont caractérisées, d'après le théorème 9.4, par :

$$\forall x \in X : x.\alpha = \max_{y \in \Gamma(x)} (y.\alpha + v(x,y))$$

La date au plus tard de x, relative à la date finale T, est donnée par $x.\lambda' = T - x.\alpha$ (théorème 9.4).

Par conséquent, ces dates sont caractérisées par les relations :

$$\forall x \in X \setminus \{x_{n+1}\} : x.\lambda' = \min_{y \in \Gamma[x]} \big(y.\lambda' - v(x,y)\big) \ , \ x_{n+1}.\lambda' = T$$

On peut donc obtenir directement ces dates en adaptant l'algorithme ORDINAL-RACINE comme suit :

sommet de départ : x_{n+1} ; initialiser $x_{n+1}.\lambda$ à T;

permuter les opérations *lst_pred* et *lst_succ* ;

marquer un nouveau sommet x par : $x.\lambda' \longleftarrow$ MIN$(y.\lambda' - v(x,y), G.lst_succ(x))$

2ème cas. Les contraintes sont quelconques. Le graphe possède alors des circuits, et les arcs ont des valeurs de signe quelconque. S'il n'y a pas d'incompatibilité dans les contraintes, il ne doit pas y avoir de circuit absorbant (valeur > 0). On peut donc envisager l'algorithme d'exploration en profondeur de FORD (algorithme 8.3.2) ou bien celui de BELLMANN-KALABA (8.4.3). Les adaptations à apporter sont les mêmes que dans le cas précédent : remplacer MIN par MAX.

Il existe cependant une technique spécifique, qui en général s'avère plus efficace (heuristique). Ceci est basé sur la remarque suivante: si on enlève du graphe tous les arcs de valeur ≤ 0, le graphe partiel obtenu doit être sans circuit (autrement, il y aurait des contraintes incompatibles: voir cas précédent). On peut toutefois conserver les arcs issus du sommet x_0 ainsi que ceux aboutissant au sommet x_{n+1}, car ces arcs ne peuvent pas appartenir à des circuits (x_0 est point d'entrée, et x_{n+1} est point de sortie). Cette remarque incite à procéder en deux phases.

Première phase – Ignorer tous les arcs non issus de x_0 ou aboutissant à x_{n+1}, qui sont de valeur ≤ 0.

– Calculer les valeurs λ_i provisoires, en utilisant l'algorithme ORDINAL-RACINE (si cet algorithme met en évidence des circuits, alors il y a des contraintes incompatibles).

Deuxième phase : ajustements Réintroduire un par un chacun des arcs que l'on a ignorés lors de la première phase. Soit (x_i,x_j,a_{ij}) un tel arc. On teste alors la validité de la contrainte correspondant à cet arc:

– si $\lambda_j < \lambda_i + a_{ij}$ alors $\lambda_j \leftarrow \lambda_i + a_{ij}$.

– Pour tout sommet x_j dont l'attribut a ainsi été augmenté, il faut recommencer le test sur tous les arcs issus de x_j. Ce procédé correspond en fait à une exploration en profondeur de la descendance de x_j, mais avec arrêt de la "descente" à partir de sommets non modifiés.

L'intérêt de cette technique vient de ce que, en général, la seconde phase ne provoque que peu de modifications, car les mises à jour de la descendance des sommets modifiés s'éteignent assez vite.

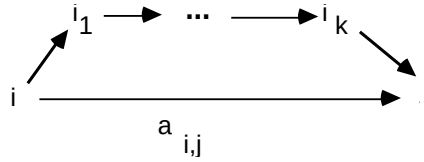
Des illustrations de cette méthode seront vues en TD.

Pour le calcul d'un ordonnancement au plus tard, tout ce qui précède reste valable, en travaillant dans le graphe transposé (c'est-à-dire en considérant x_{n+1} comme sommet de départ, et en permutant les accès "successeurs" et prédécesseurs").

9.6 Élimination de contraintes redondantes

Définition 9.7 *Une contrainte est dite "redondante" si elle est impliquée par un ensemble de contraintes dont elle ne fait pas partie.*

Cette situation se produit dans le cas suivant : la contrainte $t_j - t_i \geq a_{i,j}$ est redondante si et seulement si il existe, dans G, un chemin (différent de l'arc (i,j)) de x_i à x_j , et de valeur $v \geq a_{i,j}$.



En effet, le long d'un tel chemin [i,i_1, ... ,i_k,j] on aura :

$$\begin{cases} t_{i_1} - t_i & \geq & a_{i,i_1} \\ & \vdots & \\ t_j - t_{i_k} & \geq & a_{i_k,i} \end{cases} \Longrightarrow t_j - t_i \geq a_{i,i_1} + \ \ldots \ + a_{i_k,i} \geq a_{i,j}$$

Nous allons ci-dessous examiner dans quels cas certaines contraintes peuvent être perçues comme redondantes *au seul vu des données du problème*, c'est-à-dire sans le secours d'un algorithme de cheminement (qui pourrait s'avérer plus coûteux que le gain obtenu par la suppression éventuelle de contraintes). Il s'agit essentiellement des contraintes implicites, c'est à dire :

$\forall i \quad t_i \geq 0$ (liées au début du projet)
$\forall i \quad t_{n+1} - t_i \geq d_i$ (liées à la fin du projet)

- **contraintes de début** :
  S'il existe un arc (x_j,x_i) de valeur $a_{j,i} \geq 0$ et $j \neq 0$, alors $t_i \geq 0$ est redondante. En effet, on a : $t_i \geq t_j + a_{j,i}$.
- **contraintes de fin** :
  S'il existe un arc (x_i,x_j) de valeur $a_{i,j} \geq d_i - d_j$ et $j \neq n+1$ alors $t_{n+1} - t_i \geq d_i$ est redondante. En effet, on a :

  $t_j \geq t_i + a_{i,j}$ et $t_{n+1} \geq t_j + d_j$ ce qui implique :

  $t_{n+1} \geq t_i + a_{i,j} + d_j \geq t_i + d_i$

Chapitre 10

Arbres ; arbre partiel de poids optimum

10.1 Définition et propriétés caractéristiques

Soit $G = (X,U)$ un graphe non orienté, c'est à dire pour lequel U est un ensemble de couples non ordonnés, appelés **arêtes**. Dans un graphe non orienté, la terminologie suivante est utilisée : un chemin devient une **chaîne** un circuit devient un **cycle**.

Les successeurs et prédécesseurs d'un sommet deviennent les **voisins**, et le nombre d'arêtes adjacentes à un sommet s'appelle le **degré** du sommet.

Enfin, la relation binaire sur X :

$$xRy \iff \text{il existe une chaîne de } x \text{ à } y \textbf{ ou } x = y$$

est une relation d'équivalence, dont les classes s'appellent **composantes connexes** de G. Un graphe connexe est un graphe ne possédant qu'une seule composante connexe.

Cette terminologie établie, nous posons les définitions suivantes :

Définition 10.1 *Un arbre est un graphe non orienté, connexe et sans cycle.*

Nous allons établir un certain nombre de définitions équivalentes, grâce aux deux lemmes suivants :

Lemme 10.2 *Un graphe connexe de n sommets possède au moins $n-1$ arêtes.*

Démonstration. Soit $G = (X,U)$, ayant n sommets, m arêtes et p composantes connexes, et posons $\nu(G) = m - n + p$ (**nombre cyclomatique** de G). Définissons G' en enrichissant G par une nouvelle arête $w = (x,y)$. Si x et y appartiennent à la même composante connexe, on a

alors :

$n' = n$, $m' = m + 1$, $p' = p$ d'où $\nu(G') = \nu(G) + 1$

Dans le cas contraire, les composantes connexes (distinctes) de x et y fusionnent en une seule, d'où

$n' = n$, $m' = m + 1$, $p' = p - 1$ et $\nu(G') = \nu(G)$

L'adjonction de toute nouvelle arête à un graphe ne peut donc pas diminuer la valeur de $\nu(G)$. Or, pour un graphe à n sommets et 0 arête, on a $p = n$ d'où $\nu(G) = 0$. On en déduit que pour tout graphe, $\nu(G) \geq 0$.

Si G est connexe, on a $p = 1$ d'où $m \geq n - 1$. □

Lemme 10.3 *Un graphe sans cycle de n sommets possède au plus $n-1$ arêtes.*

Démonstration. Soient u_1,u_2,...,u_m les arêtes de G, numérotées de manière quelconque. Considérons la suite de graphes

$G_0 < G_1 < \ldots < G_{m-1} < G_m$ avec

$G_0 = (X,\emptyset); G_j = (X,U_j)$ et $U_j = U_{j-1} \cup \{u_j\}$ et donc $G_m = G$

Ces graphes sont sans cycle, donc on passe de G_j à G_{j+1} en ajoutant à G_j l'arête u_{j+1} entre deux sommets non reliés par une chaîne dans G_j . Comme dans la démonstration du lemme précédent, on en déduit :

$\nu(G_j) = \nu(G_{j+1})$, d'où $0 = \nu(G_0) = \nu(G_m)$.

Par conséquent, $m = n - p$ et comme $p \geq 1$, on a $m \leq n - 1$. □

Théorème 10.4 *Les propriétés suivantes, caractérisant un arbre, sont équivalentes :*

i) *$G = (X,U)$ est connexe et sans cycle.*

ii) *Tout couple de sommets est relié par une chaîne unique de G.*

iii) *G est sans cycle et l'adjonction de toute nouvelle arête crée un cycle unique.*

iv) *G est sans cycle et possède $n - 1$ arêtes.*

v) *G est connexe et la suppression de toute arête de G crée deux composantes connexes, telles que toute autre arête de G est incluse dans l'une ou l'autre de ces composantes.*

vi) *G est connexe et possède $n - 1$ arêtes.*

Démonstration

i) $\Longrightarrow$ ii) Tout couple de sommets est relié par une chaîne (connexité), et si deux chaînes distinctes les reliaient, leur réunion créerait un cycle.

ii) $\Longrightarrow$ iii) Si G avait un cycle, deux sommets de ce cycle seraient reliés par deux chaînes distinctes ; l'adjonction d'une nouvelle arête entre deux sommets x et y crée un cycle avec l'unique chaîne les reliant.

iii) $\Longrightarrow$ iv) Puisque l'adjonction de toute nouvelle arête crée un cycle, G est connexe donc possède au moins $n-1$ arêtes (lemme 10.2). Comme il est sans cycle, il a au plus n-1 arêtes (lemme 10.3).

iv) $\Longrightarrow$ v) Si G n'était pas connexe, on pourrait lui ajouter une nouvelle arête sans créer de cycle, ce qui donnerait un graphe sans cycle à n arêtes, d'où une contradiction (lemme 10.3). La suppression d'une arête u laisse un graphe partiel à $n-2$ arêtes, donc non connexe ; de plus, ce graphe partiel possède deux composantes connexes C et $X \setminus C$ sans quoi G ne serait pas connexe. Enfin, si $\exists v \in U$ reliant C et $X \setminus C$, G possède un cycle passant par les extrémités de u et celles de v.

v) $\Longrightarrow$ vi) G possède au moins $n-1$ arêtes (lemme 10.2) ; G possède au plus $n-1$ arêtes sinon il aurait un cycle (lemme 10.3) et la suppression d'une arête de ce cycle ne romprait pas la connexité.

vi) $\Longrightarrow$ i) Si G avait un cycle, la suppression d'une arête de ce cycle laisserait un graphe connexe, donc ayant au moins $n-1$ arêtes, ce qui contredit l'hypothèse sur le nombre d'arêtes de G. □

Proposition 10.5 *Tout arbre possède au moins deux sommets de degré 1 (appelés sommets pendants).*

Démonstration Un arbre possède au moins 1 sommet pendant, sans quoi il aurait un cycle.

D'autre part, si un arbre ayant n sommets et $n-1$ arêtes ($n \geq 4$) possédait un seul sommet pendant, le sous-graphe engendré par la suppression de ce sommet possèderait au plus 1 sommet pendant, $n-1$ sommets et $n-2$ arêtes. En répétant le procédé, on aboutirait nécessairement à un sous-graphe ayant 3 sommets, 2 arêtes donc 2 sommets pendants, ce qui est une contradiction. □

10.2 Arbres partiels dans un graphe non orienté valué

Considérons le problème pratique suivant, qui se rencontre dans de nombreux domaines : soit un ensemble de sites $\{X_1, X_2, \ldots, X_n\}$. On souhaite les relier entre eux, de telle sorte que le réseau ainsi constitué soit :

1. connexe (tout site est relié, éventuellement via d'autres sites, à tout autre site)
2. sans liaison redondante : une seule chaîne suffit, de tout site à tout site ; ceci conduit à chercher un réseau sans cycle
3. de coût minimum, sachant qu'à chaque liaison directe bilatérale (X_i, X_j) est associée un coût $w_{i,j}$.

D'après les résultats du théorème 10.4, il s'agit donc de construire un arbre sur les n sommets donnés, celui-ci étant de poids minimum parmi tous les arbres possibles (il y en a *a priori* $\binom{n-1}{\frac{n(n-1)}{2}}$. Le poids d'un arbre est alors égal, par définition, à la somme des valeurs des

$n-1$ arêtes qui en font partie. D'autre part, on peut toujours introduire comme contrainte supplémentaire l'interdiction a priori de certaines arêtes : il suffit de les affecter d'une valeur $+\infty$.

On obtient finalement le problème de détermination d'un arbre partiel de poids optimum :

Problème 10.1 *Étant donné un graphe $G = (X,U,w)$ valué non orienté et connexe, déterminer un arbre partiel de G, de poids optimum.*

Remarque. Si G n'est pas connexe, il suffit de résoudre le problème séparément dans chaque composante connexe.

Dans ce qui suit, nous supposerons – pour fixer les idées – que optimum signifie minimum. Nous établissons maintenant un théorème caractéristique – et sa version "duale" – d'un arbre partiel de poids minimum.

Théorème 10.6 *Une condition nécessaire et suffisante pour que $A = (X,H)$ soit un arbre de poids minimum de $G = (X,U,w)$ est la suivante : $\forall u \in U \setminus H$: le long du cycle (unique) μ formé par u avec des arêtes de H, on a $\forall v \in \mu \ : \ w(v) \leq w(u)$.*

Démonstration.

▷ La condition est nécessaire : sinon, en substituant u à v dans H, où $w(v) > w(u)$, on obtiendrait un nouvel arbre de poids strictement inférieur.

▷ Réciproquement, considérons un arbre $A^* = (X,H^*)$, de poids minimal.

Si $A^* \neq A$, considérons une arête $u^* = (x,y)$ telle que $u^* \in H^*$ et $u^* \notin H$ (figure 10.1).

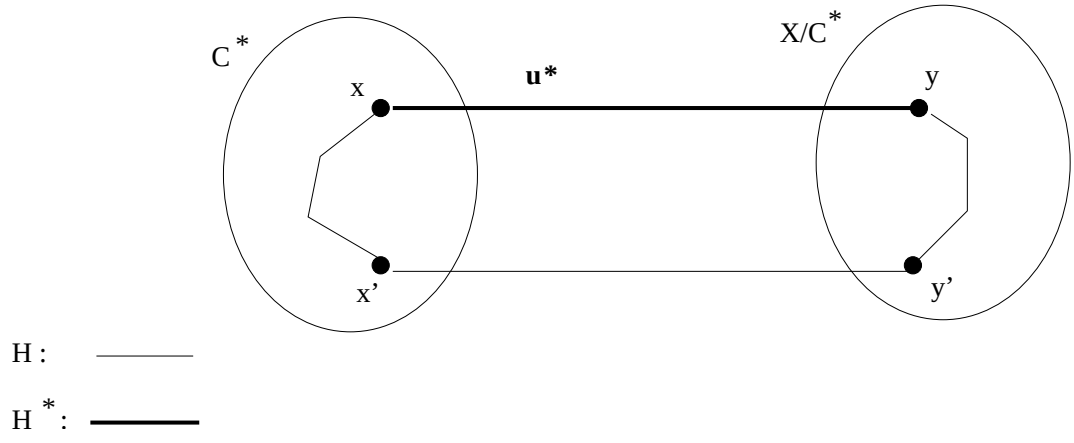


FIG. 10.1 – *arbres H et H^**

D'une part, x et y sont reliés dans H par une chaîne γ, ne contenant pas u^*.

D'autre part, la suppression de u^* dans H^* définit un graphe partiel ayant deux composantes connexes C^* et $X \setminus C^*$, et u^* est la seule arête de H^* reliant C^* à $X \setminus C^*$.

Il en résulte que γ relie $x \in C^*$ à $y \in X \setminus C^*$ sans passer par u^*, donc contient au moins une arête $u = (x',y')$, avec $x' \in C^*$,$y' \in X \setminus C^*$,$u \in H$,$u \notin H^*$.

On en déduit que :
(1) $w(u) \leq w(u^*)$ d'après l'hypothèse sur A
(2) la chaîne γ^* qui relie x' et y' dans H^* passe nécessairement par u^* (puisque u^* est la seule arête de H^* reliant C^* à $X \setminus C^*$) et donc, A^* étant minimal : $w(u^*) \leq w(u)$ (condition nécessaire).

(1) et (2) montrent que : à chaque arête $u^* \in H^*$,$u^* \notin H$ on peut faire correspondre une arête $u \in H$,$u \notin H^*$, telle que $w(u^*) = w(u)$; par conséquent, H et H^* sont de même poids. □

Nous établissons maintenant une version "duale" du théorème de caractérisation 10.6 :

Théorème 10.7 *Une condition nécessaire et suffisante pour que $A = (X,H)$ soit un arbre de poids minimum de $G = (X,U,w)$ est la suivante : $\forall u \in H$: pour toute arête $v \neq u$ reliant C_u et $X \setminus C_u$ (les deux composantes connexes créées dans A par la suppression de u), on a : $w(u) < w(v)$.*

Démonstration.

▷ La condition est nécessaire : en substituant $v(\notin H)$ à u dans H, où $w(u) > w(v)$, on obtiendrait un nouvel arbre, de poids strictement inférieur.

▷ Réciproquement, soit $A = (X,H)$ un arbre satisfaisant les conditions du théorème. Pour tout $u \notin H$, considérons le cycle γ_u que u forme avec H. On a alors :

$\forall v \in \gamma_u$, u relie C_u et $X \setminus C_u$ et donc : $w(v) < w(u)$; par suite A satisfait les conditions du théorème 10.6. □

10.3 Algorithme de KRUSKAL

10.3.1 Principe

Du théorème précédent 10.6 découle immédiatement l'algorithme de KRUSKAL. Le principe est le suivant : on organise un parcours des arêtes dans l'ordre des valeurs croissantes. Initialement, le graphe partiel A est vide. On ajoute l'arête courante à A si et seulement si elle ne crée pas de cycle dans A, et on s'arrête en fin de parcours, ou dès qu'on a obtenu $n-1$ arêtes dans A.

A l'issue de l'algorithme, on a bien un arbre partiel de poids minimum : en effet, on a construit un graphe partiel sans cycle, ayant au moins $n-1$ arêtes (par hypothèse, le graphe G est connexe), et d'autre part, toute arête u non retenue forme un cycle avec des arêtes situées avant elle dans la liste ordonnée des arêtes, donc vérifie la condition suffisante du théorème 10.6.

Pour faciliter le test : u ne crée pas de cycle, il suffit de gérer les composantes connexes de A au fur et à mesure de sa construction. Une arête pourra être ajoutée si et seulement si ses deux extrémités n'appartiennent pas à la même composante. Les composantes des deux extrémités

sont alors fusionnées en une seule. Enfin, le parcours des arêtes ordonnées par valeurs croissantes sera géré par les accès :

`pp_arete : --` *pré : non hors_arete*
délivre la première arête du parcours ordonné
(arête de valeur minimum)

`pp_arete_suiv : --` *pré : non hors_arete*
délivre la prochaine arête du parcours ordonné

`hors_arete : vrai` si toutes les arêtes ont été visitées

A noter que `G.vide` $\Rightarrow$ `G.hors_arete`.

10.3.2 Texte de l'algorithme

Il est donné page 150

ALGORITHME DE KRUSKAL

```
ENS[SOMMET] SOMMET::C ; --   x.C = composante connexe de x

ARBRE kruskal(GRAPHESYM G): c'est
  local ARêTE arcour;
        ENT k; --   compteur du nombre d'arêtes
        SOMMET x, y;
 début
    depuis
       pourtout x de G.lst_som faire x.C ← {x} fpourtout ;
      Result ← creer ; k ← 0 ;
      arcour ← G.pp_arete ; x ← arcour.ori ; y ← arcour.ext ;
    jusqua k ≥ n-1 ou x = nil
    faire
       si y ∉ x.C alors
          Result.ajoutarete(x,y) ;
          x.C ← x.C ∪ y.C ; y.C ← x.C ;
          k ← k + 1
       fsi ;
      arcour ← G.pp_arete_suiv ; x ← arcour.ori ; y ← arcour.ext
    fait ;
    si k < n-1 alors --   erreur : le graphe donne G n'est pas connexe
 fin
```

10.3.3 Exemple

Considérons le graphe non orienté, valué, de la figure 10.2

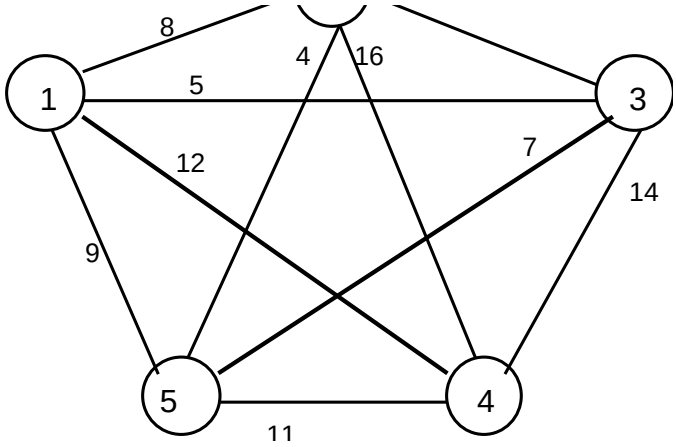


FIG. 10.2 – *Exemple*

La liste ordonnée des arêtes est donnée dans le tableau de gauche :

		comp. connexes : {1} {2} {3} {4} {5}
2,5	4 *	arête (2,5) $\longrightarrow$ c.c. {1} {2,5} {3} {4}
1,3	5 *	arête (1,3) $\longrightarrow$ c.c. {1,3} {2,5} {4}
3,5	7 *	arête (3,5) $\longrightarrow$ c.c. {1,2,3,5} {4}
2,3	8	
1,2	8	
1,5	9	
4,5	11 *	arête (4,5) $\longrightarrow$ c.c. {1,2,3,4,5}
1,4	12	
3,4	14	
2,4	16	

poids : $4 + 5 + 7 + 11 = 27$

10.3.4 Complexité

Lorsque le parcours est organisé, on effectue au plus $n - 1$ pas d'itérations. L'algorithme est donc en O(n) + complexité maximale du tri des arêtes. Ce dernier terme dépend de la technique utilisée :

- tri préalable (les meilleurs algorithmes connus sont en $O(m \log m)$)
- recherche séquentielle dans la liste des arêtes, à chaque pas d'itération (complexité totale de $\sum_{k=1}^{n-1} O(m-k)$ – puisqu'à l'étape k il reste $m - k$ arêtes)
- organisation "en tas" de la liste des arêtes et réorganisation à chaque étape :
  - $O(m \log m)$ pour l'organisation initiale

– $O(\log(m-k))$ pour la réorganisation à l'étape k (nombre d'arêtes restant $= m-k$)

d'où, globalement : $0(m \log m) + \sum_{k=1}^{n-1} 0(\log m)$, le terme $0(m \log m)$ étant prépondérant puisque $m \geq n-1$ (graphe connexe).

10.4 Algorithme de PRIM

10.4.1 Principe de l'algorithme

C'est aussi un algorithme d'extension, mais au lieu d'engendrer des graphes partiels successifs, sans cycle, croissant jusqu'à la connexité, on engendre des sous-arbres partiels croissants jusqu'à ce qu'on obtienne un arbre partiel. La propriété maintenue invariante tout au long de l'algorithme est la suivante :

– $1 \leq k \leq n$;
– $X_k = \{x_1, \ldots, x_k\}$;
– $H_k = \{u_1, \ldots, u_{k-1}\}$;
– $A_k = (X_k, H_k)$ est un arbre couvrant X_k;
– Il existe un arbre $A = (X,H)$, couvrant X, de poids minimum, dont la restriction à X_k est A_k (la restriction de A à X_k est le sous-graphe de A engendré par les sommets de X_k).

La **condition d'arrêt** est donc $k = n$. Si l'invariant est vérifié à l'arrêt, A_n est bien un arbre couvrant de poids minimum (évident).

Initialement $k = 1, X_1 = \{x_1\}$ (sommet quelconque), $H_1 = \emptyset$. Les valeurs initiales satisfont l'invariant (évident).

Il reste à établir la **progression**, de manière à maintenir l'invariant. Le principe est très simple : soit $u_k = (x_i, x_{k+1})$ (avec $x_i \in X_k$ et $x_{k+1} \notin X_k$) une arête *sortante* de X_k, de poids minimum parmi les arêtes sortantes. On pose:

$H_{k+1} \leftarrow H_k \bigcup \{u_k\}$
$X_{k+1} \leftarrow X_k \bigcup \{x_{k+1}\}$

Proposition 10.8 *La progression maintient l'invariant*

Démonstration Soit $A = (X,H)$ un arbre couvrant X, de poids minimal, dont la restriction à X_k est A_k (un tel arbre existe par hypothèse).

premier cas : $u_k \in H$. Dans ce cas, la restriction de A à X_{k+1} est A_{k+1}, et la propriété est démontrée.

deuxième cas (voir figure 10.3) : $u_k \notin H$. Dans ce cas, il existe une chaîne γ de H, reliant x_i à x_{k+1} et, puisque A est de poids minimum, on a :

$\forall v \in \gamma \ : \ c(v) \leq c(u_k)$.

La chaîne γ contient une arête $v_k = (x_a, x_b)$ telle que $x_a \in X_k, x_b \notin X_k$; puisque $v_k \in \gamma$, on a $c(v_k) \leq c(u_k)$; et, d'après le critère de sélection de u_k (arête sortante de X_k de poids minimum), on a aussi $c(u_k) \leq c(v_k)$; d'où $c(v_k) = c(u_k)$.

Examinons le graphe $A' = (X, H \setminus \{v_k\} \bigcup \{u_k\})$:

– il a n sommets et $n-1$ arêtes,

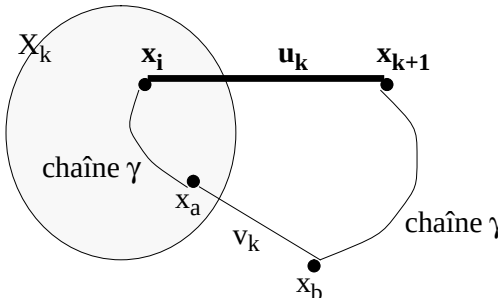


FIG. 10.3 – *passage de A_k à A_{k+1}*

– il est connexe; en effet, soit s et t deux sommets de X; ils sont reliés dans A par une chaîne κ. Si cette chaîne ne contient pas v_k, alors c'est une chaîne de A', par construction; si au contraire κ contient v_k, on remplace cette dernière arête par la chaîne $[x_a,\ldots,x_i]\cdot u_k\cdot[x_{k+1},\ldots,x_b]$ qui est une chaîne de A' par construction (figure 10.3).

A' est donc un arbre couvrant X, de poids $c(A') = c(A) - c(v_k) + c(u_k) = c(A)$, et donc de poids minimal; il contient u_k et donc, sa restriction à X_{k+1} est bien A_{k+1}. □

Corollaire 10.9 *A chaque étape, l'arbre partiel $A_k = (X_k,H_k)$ est un arbre couvrant X_k, de poids minimum.*

Évident puisque c'est un arbre couvrant X_k, par construction, et c'est la restriction d'un arbre de poids minimum d'après la proposition précédente. □

10.4.2 Mise en œuvre de l'algorithme

La sélection d'une nouvelle arête peut être effectuée selon une méthode tout à fait analogue à celle de l'algorithme de DIJKSTRA (chapitre 8.5). En effet, soit M l'ensemble des sommets déjà "couverts" par le sous-arbre partiel. On désigne par A l'ensemble des sommets "limitrophes" de M, c'est à dire :

$$z \in A \Leftrightarrow z \notin M \quad \textbf{et } \exists x \in M \ : (x,z) \in U$$

Pour tout $z \in A$, on pose :

$$\delta_z = \min_{\substack{x\in M\\(x,z)\in U}} w(x,z)$$

atteint pour un sommet dénoté p_z, puis on sélectionne $y \in A$ tel que

$$\delta_y = \min_{z\in A} \delta_z$$

L'arête (p_y,y) est alors ajoutée au sous-arbre partiel. A chaque pas d'itération, l'ensemble A est donc modifié de la manière suivante : y est ôté de A, et les voisins de y, n'appartenant pas

à M, sont soit introduits dans A – pour de tels sommets z, on a alors $\delta_z = w(y,z)$ – soit, s'ils appartenaient déjà à A, leur marque est réévaluée par : $\delta_z = \min(\delta_z, w(y,z))$.

De plus, comme pour l'algorithme de DIJKSTRA, l'ensemble des valeurs $(\delta_z)_{z \in A}$ peut être géré en "tas".

10.4.3 Texte de l'algorithme

Le texte de l'algorithme, donné ci-après page 155, montre qu'en fait on retrouve l'algorithme de DIJKSTRA avec les différences suivantes :

- les marques des sommets passent à 0 lorsque ceux-ci rentrent dans M;
- les accès *lst_succ* et *valarc* sont remplacés respectivement par *lst_voisins* et *valarête*, adaptés aux graphes non orientés;
- on mémorise, en plus, le sous-arbre partiel de poids minimum, A.

10.4.4 Exemple

on reprend le graphe de la figure 10.2

M	$\{(x.p, x, x.\delta) \mid x \in A\}$	arête sélectionnée	nbar
1	(1,3,5) (1,2,8) (1,5,9) (1,4,12)	(1,3) poids 5	1
1,3	(3,5,7) (1,2,8) (1,4,12)	(3,5) poids 7	2
1,3,5	(5,2,4) (5,4,11)	(5,2) poids 4	3
1,3,5,2	(5,4,11)	(5,4) poids 11	4

ALGORITHME DE PRIM

```
REEL SOMMET::δ : ; SOMMET SOMMET::p ;
ENUM[dehors,atteint,recouvert] SOMMET::état ;
ARBRE prim(GRAPHESYM G)  c'est
    local  ENT nbar ; --   nombre d'arêtes
          TAS[(SOMMET,REEL)] A; ENS[SOMMET]  X, MOD;
          SOMMET nouv ; REEL w;
 début
    depuis
      nbar ← 0 ;
      Result ← creer ; X ← G.lst_som ;
       pourtout x  de X  faire x.état  ← dehors  fpourtout ;
      A ← creer ;
      x0 ← X.element ; --   sommet initial quelconque
      x0.état ← recouvert ; MOD ←  G.lst_voisins(x0) ;
       pourtout z  de MOD
         z.état ←  atteint ; z.δ ←  G.valarête(x0,z) ; z.p ←  x0 ;
         A.mettre_en_tas((z,z.δ)) ;
       fpourtout
    jusqua nbar ≥ n-1
    faire
      (nouv,w) ← A.mintas ; A.ôter_de_tas ;
      nouv.état ← recouvert ; MOD  ← G.lst_voisins(nouv) ;
      --   mise à jour des marques des sommets modifiables
       pourtout z  de MOD  faire
         cas
           z.état = dehors ⟶ z.état  ← atteint ;
                                z.δ ← G.valarête(nouv, z) ; z.p ← nouv ;
                                A.mettre_en_tas((z,z.δ))
           z.état = atteint ⟶  si G.valarête(nouv, z) < z.δ  alors
                                      A.reorg((z,z.δ),(z, G.valarête(nouv, z)));
                                      z.δ  ← G.valarête(nouv, z) ; z.p ← nouv
           z.état = recouvert ⟶  rien
         fcas
       fpourtout ;
      Result.ajoutarête(nouv,nouv.p) ; nbar ←  nbar+1
    fait
 fin
```

Chapitre 11

Flots dans un réseau de transport

11.1 Exemple introductif

Problème de l'approvisionnement et de la demande. Une ressource est disponible sur 3 sites A_1,A_2,A_3 en quantités respectives a_1,a_2,a_3. Cette ressource est demandée en 4 sites B_1,B_2,B_3,B_4 en quantités respectives b_1, b_2, b_3, b_4. Les possibilités d'acheminement des ressources des A_i vers les B_j sont modélisées par le graphe valué de la figure 11.1; dans ce graphe, les sites C_1,C_2 sont des sites intermédiaires de transfert, et la valeur d'un arc est une capacité, égale à la quantité maximale de ressource qui peut transiter sur cet arc.

Le problème posé est alors le suivant : *est-il possible de satisfaire les demandes des B_j à partir des disponibilités des A_i et des possibilités de transport?*

Pour prendre en compte l'ensemble des données sur le graphe, on "agrandit" ce dernier en introduisant un site fictif d'entrée e, un site fictif de sortie s, et les arcs (représentés en gras sur la figure):

- (e,A_i), de capacité a_i
- (B_j,s) de capacité b_j

Un tel graphe s'appelle *réseau de transport*.

Le fait d'attribuer à un arc d'entrée (e,A_i) une capacité s'interprète bien: cela signifie que a_i est la quantité maximum que l'on peut acheminer depuis A_i; par contre, la capacité d'un arc de sortie B_j signifie que l'on s'interdit de consommer en B_j plus que la ressource demandée.

Pour résoudre le problème, on considère une fonction φ, définie sur les arcs et à valeur dans $\mathbf{N}$, telle que:

- arc d'entrée: $\varphi(e,A_i)$ = quantité de ressource enlevée au site A_i,
- arc de sortie: $\varphi(B_j,s)$ = quantité de ressource amenée au site B_j,
- arc de transport: $\varphi(x,y)$ = quantité de ressource acheminée sur l'arc (x,y).

La valeur de φ sur un arc représente donc un *flux*, et doit satisfaire les *conditions de conservation* (ou lois de Kirchoff) en tout nœud du réseau (autre que e et s):

(1) $\forall x,\ x \neq e,\ x \neq s : \sum_{z \in \Gamma^{-1}(x)} \varphi(z,x) = \sum_{y \in \Gamma(x)} \varphi(x,y)$

De plus, sur tout arc, la condition de capacité doit être respectée, c'est-à-dire

(2) $\forall u \in \Gamma :\ \varphi(u) \leq c(u)$, où $c(u)$ est la capacité de l'arc u

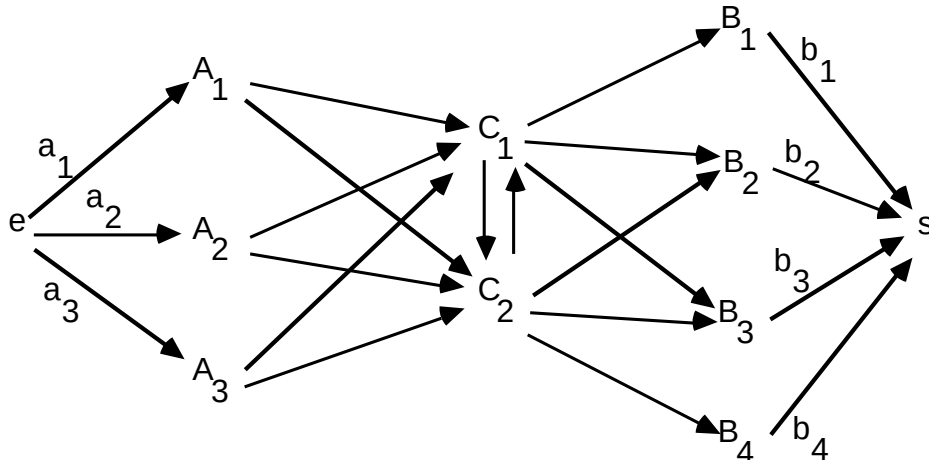


FIG. 11.1 – *Réseau de transport*

Une telle fonction s'appelle un *flot* (contraintes 1) *compatible* (contraintes 2)

Le problème posé admet une solution s'il existe un flot compatible tel que, pour tout site demandeur B_j, on ait $\varphi(B_j,s) = b_j$ (*saturation des arcs de sortie*). En pratique, nous verrons, grâce aux résultats du §11.2, que la résolution de ce problème revient à chercher, parmi les flots compatibles, un flot de *valeur maximum*, c'est-à-dire acheminant la plus grande quantité globale de ressource. Nous allons dans ce qui suit définir précisément les notions de *réseau de transport*, *flot compatible*, *valeur d'un flot*, et les résultats généraux liés à ces notions. Puis nous donnerons un algorithme de calcul de flot de valeur maximum, dû à FORD et FULKERSON.

11.2 Définitions et propriétés générales

Réseau de transport.

Définition 11.1 *Un réseau de transport est un graphe valué par des entiers $G = (X,e,s;\Gamma;c)$ où*

- *$X \cup \{e\} \cup \{s\}$ est l'ensemble des sommets,*
- *$c : \Gamma \longrightarrow \mathbf{N}$ est la fonction capacité.*
- *le sommet e s'appelle l'entrée (ou source), le sommet s la sortie (ou puits) et ils sont respectivement racine et anti-racine du graphe.*
- *les arcs $(e,*)$ et $(*,s)$ s'appellent respectivement arcs d'entrée et arcs de sortie du réseau*

Flot compatible.

Définition 11.2 *Un flot sur un réseau de transport* $(X,e,s;\Gamma;c)$ *est une fonction* $\varphi : \Gamma \longrightarrow \mathbf{N}$ *vérifiant les lois de Kirchoff (définies au §11.1) Un flot compatible vérifie en outre* $\forall u \in \Gamma : \ 0 \leq \varphi(u) \leq c(u)$

Quelques notations. Soit $A \subseteq X$. On note:

$\Gamma^S(A) = \{(x,y) \in \Gamma \mid x \in A \wedge y \notin A\}$ arcs sortants de A

$\Gamma^E(A) = \{(x,y) \in \Gamma \mid x \notin A \wedge y \in A\}$ arcs entrants dans A

puis

$$\varphi^S(A) = \sum_{u \in \Gamma^S(A)} (\varphi(u)) = \text{flux sortant de } A \text{ et}$$

$$\varphi^E(A) = \sum_{u \in \Gamma^E(A)} (\varphi(u)) = \text{flux entrant dans } A$$

(de manière analogue, on aura $c^S(A)$, $c^E(A)$). Avec ces notations, par exemple, les conditions de conservation au nœud $x(\neq e, \neq s)$ s'expriment simplement :

$\varphi^E(x) = \varphi^S(x)$ où x est mis pour l'ensemble $\{x\}$

Sur le plan algorithmique, un réseau de transport est un graphe valué, muni des accès suivants :

`ENT ARC::c --` *capacité*
`ENT ARC::φ --` *flot*

avec les notations habituelles : si `(x,y)` est un arc, alors `c(x,y)` et `φ(x,y)` désignent respectivement la capacité et le flux de l'arc `(x,y)`.

Valeur d'un flot compatible.

Définition 11.3 *La valeur d'un flot compatible est la quantité* $v(\varphi) = \varphi^S(e)$ *(flux sortant de la "source" e).*

La proposition suivante et son corollaire vont montrer, entre autres, que cette valeur représente bien la quantité globale de flux qui traverse le réseau de la source vers le puits.

Proposition 11.4 *Soit* $A \subseteq X \cup \{e\}$ *et* φ *un flot.*
- *si* $e \notin A$ *alors* $\varphi^S(A) - \varphi^E(A) = 0$
- *si* $e \in A$ *alors* $\varphi^S(A) - \varphi^E(A) = v(\varphi)$

Démonstration.
i) Supposons que $e \notin A$. Pour tout x de A, les conditions de conservation permettent d'écrire:

$$\sum_{\substack{y\in\Gamma(x)\\ y\notin A}} \varphi(x,y) + \sum_{\substack{y\in\Gamma(x)\\ y\in A}} \varphi(x,y) = \sum_{\substack{z\in\Gamma^{-1}(x)\\ z\notin A}} \varphi(z,x) + \sum_{\substack{z\in\Gamma^{-1}(x)\\ z\in A}} \varphi(z,x)$$

Sommons ces égalités sur A. Il vient:

$$\varphi^S(A) + \sum_{\substack{x\in A\\ y\in\Gamma(x)\\ y\in A}} \sum \varphi(x,y) = \varphi^E(A) + \sum_{\substack{x\in A\\ z\in\Gamma^{-1}(x)\\ z\in A}} \sum \varphi(z,y), \text{ soit}$$

$$\varphi^S(A) - \varphi^E(A) = -\sum_{\substack{x\in A\\ y\in\Gamma(x)\\ y\in A}} \sum \varphi(x,y) + \sum_{\substack{x\in A\\ z\in\Gamma^{-1}(x)\\ z\in A}} \sum \varphi(z,x)$$

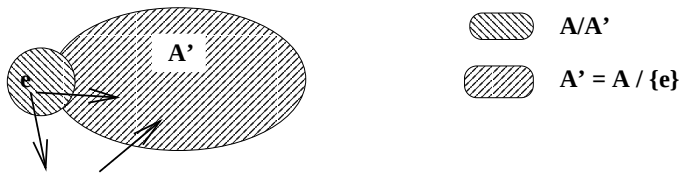
Si dans la deuxième somme double on fait le changement de variable: $(x,z) \to (y,x)$, on obtient:

$$\varphi^S(A) - \varphi^E(A) = -\sum_{\substack{x\in A\\ y\in\Gamma(x)\\ y\in A}} \sum \varphi(x,y) + \sum_{\substack{y\in A\\ x\in\Gamma^{-1}(y)\\ x\in A}} \sum \varphi(x,y) = 0$$

ce qui démontre la proposition dans ce cas.
ii) Supposons que $e \in A$, et posons $A' = A \setminus \{e\}$. On a alors:

$$\varphi^S(A') = \varphi^S(A) - \sum_{\substack{y\in\Gamma(e)\\ y\notin A'}} \varphi(e,y)$$

$$\varphi^E(A') = \varphi^S(A) + \sum_{\substack{y\in\Gamma(e)\\ y\in A'}} \varphi(e,y)$$



et, d'après le point i): $\varphi^S(A') - \varphi^E(A') = 0$, ce qui donne:

$$\varphi^S(A) - \varphi^E(A) = \sum_{y\in\Gamma(e)} \varphi(e,y) = \varphi^S(e) = v(\varphi)$$

□

Corollaire 11.5 *Soit φ un flot. On a: $v(\varphi) = \varphi^S(e) = \varphi^E(s)$*

Démonstration. En effet, si on applique la proposition précédente avec $A = \{e\} \cup X$, on obtient:

$$\varphi^S(A) = \varphi^E(s),\ \varphi^E(A) = 0 \quad \text{d'où} \quad v(\varphi) = \varphi^E(s)$$

Définition 11.6 *On appelle coupe du réseau tout ensemble d'arcs* $\Gamma^S(A)$ *où* $A \subseteq X \cup \{e\}$ *avec* $e \in A$. *La quantité* $c^S(A)$ *s'appelle la capacité de la coupe* $\Gamma^S(A)$.

Remarque. Ce nom de *coupe* vient de ce que tout chemin de e à s emprunte au moins un arc de la coupe; autrement dit, si on supprime les arcs de la coupe, on "déconnecte" le réseau entre e et s, c'est-à-dire on empêche tout passage de e à s.

Nous énonçons un lemme, dû à FORD et FULKERSON, fondamental pour l'obtention de l'algorithme de calcul d'un flot maximal.

Lemme 11.7 *Soit* $A \subseteq X \cup \{e\}$ *avec* $e \in A$, *et* φ *un flot compatible. On a l'inégalité:*

$$v(\varphi) \leq c^S(A)$$

Démonstration. D'après la proposition 11.4 on a:

$$v(\varphi) = \varphi^S(A) - \varphi^E(A)$$

mais, le flot étant compatible:

$$\varphi^S(A) \leq c^S(A), \quad \varphi^E(A) \geq 0$$

d'où le résultat □

Ce lemme s'énonce encore sous la forme suivante : "*La valeur de tout flot compatible est inférieure ou égale à la capacité de toute coupe*"

Corollaire 11.8
$$\max_{\varphi \text{ compatible}} v(\varphi) \leq \min_{\substack{A \subseteq X \cup \{e\} \\ e \in A}} c^S(A)$$

Évident d'après le lemme précédent.

Dans le paragraphe suivant, nous nous intéressons au calcul d'un flot compatible de valeur maximum. En fait, nous allons montrer un résultat de *dualité*: dans le corollaire précédent, l'inégalité est une *égalité*: ce résultat est connu sous le nom de *théorème de* FORD-FULKERSON, et sa preuve repose sur la construction explicite, par un algorithme, d'un flot $\overline{\varphi}$ et d'une coupe $\Gamma^S(\underline{A})$ tels que $v(\overline{\varphi}) = c^S(\underline{A})$

11.3 Recherche d'un flot compatible de valeur maximum

11.3.1 Schéma général

Dans ce paragraphe, nous décrivons un algorithme dû à FORD et FULKERSON; le principe en est le suivant: partant d'un flot compatible initial, on teste si ce flot est de valeur maximum, et s'il ne l'est pas, on le modifie pour obtenir un nouveau flot compatible de valeur *strictement supérieure* à la valeur du flot précédent, puis on recommence. Le test et l'augmentation du flot

sont effectués grâce à une procédure de *marquage* que nous détaillons ci-après. Le schéma de l'algorithme est donc itératif:

depuis `G.`φ $\longleftarrow$ *flot compatible;*
jusqua $\neg$ *marquage(G)*
faire
  amélioration(G)
fait

11.3.2 Marquage

Le rôle de la fonction *marquage* consiste à déterminer un sous-ensemble A de $X \cup \{e\} \cup \{s\}$, ce sous-ensemble devant être le plus grand possible, tel que chaque sommet de A puisse être marqué en appliquant les règles ci-après:

1. e est marqué $(+)$
2. si x est marqué:
   (a) marquage *en avant*: marquer $(+x)$ tout successeur y de x tel que $y \notin A$ et $\varphi(x,y) < c(x,y)$ (arc *non saturé*)
   (b) marquage *en arrière*: marquer $(-x)$ tout prédécesseur z de x tel que $z \notin A$ et $\varphi(z,x) > 0$ (arc *chargé*)
3. arrêt lorsque s est marqué ou lorsqu'aucun nouveau sommet ne peut plus être marqué. La fonction rend le résultat **vrai** si, et seulement si, s n'a pas pu être marqué. Le marquage est réalisé comme "*effet de bord*" de la fonction.

Intuitivement, ce procédé de marquage repose sur l'idée suivante: on essaye d'augmenter la valeur de φ; pour cela, il faut d'abord trouver un arc d'entrée (e,x) non saturé; c'est le sens du marquage de e, puis du marquage $(+e)$ de tout successeur x de e tel que (e,x) ne soit pas saturé. Si un tel arc existe, on envoie une unité de flux supplémentaire, qui parvient donc à x; le flux entrant en x ayant augmenté de 1, le maintien des conditions de conservation nécessite donc, soit d'augmenter le flux sortant de 1, soit de diminuer le flux sur un autre arc entrant de 1; l'ensemble des arcs issus de x sur lesquels on peut envoyer l'unité de flux supplémentaire est donc repéré grâce au marquage "*en avant*", et l'ensemble des arcs aboutissant à x sur lesquels on peut diminuer d'une unité le flux entrant est repéré grâce au marquage "*en arrière*". Selon les possibilités, on parvient ainsi à compenser l'envoi de l'unité de flux supplémentaire le long d'une suite de sommets; si l'on parvient à un sommet x prédécesseur de s tel que l'arc (x,s) n'est pas saturé, on peut maintenir les conditions de conservation en x en envoyant l'unité de flux supplémentaire sur cet arc; la valeur du flot a ainsi augmenté d'une unité, puisqu'une unité supplémentaire issue de e a pu être acheminée jusqu'à s tout en maintenant les conditions de conservation et les contraintes de compatibilité.

Du point de vue algorithmique, ce procédé de marquage n'est rien d'autre qu'une recherche de descendants du sommet e, dans un graphe "auxiliaire" engendré par les arcs non saturés du réseau et par les arcs chargés du réseau transposé; cette recherche pourra être mise en œuvre par une des méthodes vues dans les chapitres précédents, notamment une méthode d'exploration.

Donnons maintenant un premier résultat lié à l'application des règles de marquage.

Proposition 11.9 *Si le sommet s n'a pu être marqué par la fonction* `marquage(G)` *alors* `G`.φ *est de valeur maximum.*

Démonstration. Soit A l'ensemble des sommets marqués à l'issue de la procédure. Par construction $e \in A$ et, par hypothèse, $s \notin A$. Par conséquent, $\Gamma^S(A)$ est une coupe. D'après la proposition 11.4, on a:

(1) $v(\varphi) = \varphi^S(A) - \varphi^E(A)$. Mais, si $x \in A$ et $y \notin A$, l'arc (x,y) est saturé, d'où $\varphi(x,y) = c(x,y)$; donc:

(2) $\varphi^S(A) = c^S(A)$. De même, si $z \notin A$ et $x \in A$, l'arc (z,x) n'est pas chargé, c'est-à-dire $\varphi(z,x) = 0$; donc:

(3) $\varphi^E(A) = 0$. Les égalités (1), (2), (3) impliquent: $v(\varphi) = c(A)$ et par conséquent, d'après le lemme de 11.7 Ford-Fulkerson, φ est de valeur maximum.

Corollaire 11.10 *Si le sommet s n'a pu être marqué par la fonction* `marquage(G)`*, alors la coupe $\Gamma^S(A)$ définie par l'ensemble des sommets marqués est de capacité minimum.*

Théorème 11.11 *(Ford-Fulkerson) La valeur maximum des flots compatibles est égale à la capacité minimum des coupes.*

Ce théorème est un énoncé équivalent aux deux propositions précédentes. Il répond à la question posée à la fin du §11.2.

11.3.3 Amélioration

Nous établissons maintenant la réciproque de ce qui précède, à savoir le fait que si s a pu être marqué par la fonction `marquage(G)`, alors le flot `G`.φ n'est pas de valeur maximum; de plus, le marquage réalisé permet de définir un nouveau flot compatible de valeur strictement supérieure à celle de φ.

Soit $e(= x_0),x_1,x_2,\ldots,x_n,s(= x_{n+1})$ une séquence de sommets telle que x_1 soit marqué $(+e)$, x_2 marqué $(\pm x_1)$, ... , x_i marqué $(\pm x_{i-1})$, ... ,s marqué $(+x_n)$.

Un couple (x_{i-1},x_i) tel que x_i est marqué $(+x_{i-1})$ correspond à un arc non saturé (x_{i-1},x_i); posons $\varepsilon_i = c(x_{i-1},x_i) - \varphi(x_{i-1},x_i)$ (*capacité résiduelle de l'arc*, > 0). Cette quantité est le nombre maximum d'unités de flux supplémentaires pouvant circuler sur cet arc. On dira que l'arc (x_i,x_{i-1}) peut être ε_i-augmenté.

Un couple (x_{i-1},x_i) tel que x_i est marqué $(-x_{i-1})$ correspond à un arc chargé (x_i,x_{i-1}); posons $\varepsilon_i = \varphi(x_{i-1},x_i)$ (*flux de l'arc*, > 0). Cette quantité est le nombre maximum d'unités de flux pouvant être ôtées de cet arc. On dira que l'arc (x_i,x_{i-1}) peut être ε_i-diminué.

Définissons la fonction $\hat{\varphi}$ de la manière suivante: Soit $\varepsilon = \min_{1\le i\le n+1} \varepsilon_i$

– sur tout arc (x_{i-1},x_i) de la chaîne pouvant être augmenté (c'est-à-dire x_i marqué $(+x_{i-1})$): $\hat{\varphi}(x_{i-1},x_i) = \varphi(x_{i-1},x_i) + \varepsilon$

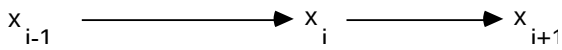
– sur tout arc (x_i,x_{i-1}) de la chaîne pouvant être diminué (c'est-à-dire x_i marqué $(-x_{i-1})$: $\hat{\varphi}(x_i,x_{i-1}) = \varphi(x_i,x_{i-1}) - \varepsilon$
– sur tout autre arc (y,z): $\hat{\varphi}(y,z) = \varphi(y,z)$

On a le résultat suivant:

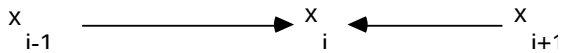
Proposition 11.12 *$\hat{\varphi}$ est un flot compatible, et $v(\hat{\varphi}) = v(\varphi)+\varepsilon > v(\varphi)$*

Démonstration. 1) vérifions que $\hat{\varphi}$ est un flot. Il suffit de vérifier que les conditions de conservation sont maintenues en tout sommet de la chaîne ($\neq e$, $\neq s$). Quatre cas sont possibles pour un sommet $x_i (1 \leq i \leq n)$

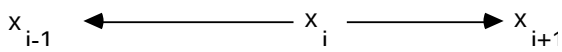
1. $marque(x_i) = (+x_{i-1})$ **et** $marque(x_{i+1}) = (+x_i)$.



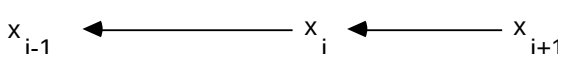
   On a alors $\hat{\varphi}^E(x_i) = \varphi^E(x_i) + \varepsilon$ et $\hat{\varphi}^S(x_i) = \varphi^S(x_i) + \varepsilon$ d'où
   $\hat{\varphi}^E(x_i) = \hat{\varphi}^S(x_i)$
2. $marque(x_i) = (+x_{i-1})$ **et** $marque(x_{i+1}) = (-x_i)$



   On a alors $\hat{\varphi}(x_{i-1},x_i) = \varphi(x_{i-1},x_i) + \varepsilon$ et $\hat{\varphi}(x_{i+1},x_i) = \varphi(x_{i+1},x_i) - \varepsilon$ soit
   $\hat{\varphi}^E(x_i) = \varphi^E(x_i)$ et $\hat{\varphi}^S(x_i) = \varphi^S(x_i)$ d'où
   $\hat{\varphi}^E(x_i) = \hat{\varphi}^S(x_i)$
3. $marque(x_i) = (-x_{i-1})$ **et** $marque(x_{i+1}) = (+x_i)$.



   On a alors $\hat{\varphi}(x_i,x_{i-1}) = \varphi(x_i,x_{i-1}) - \varepsilon$ et $\hat{\varphi}(x_i,x_{i+1}) = \varphi(x_i,x_{i+1}) + \varepsilon$ soit
   $\hat{\varphi}^E(x_i) = \varphi^E(x_i)$ et $\hat{\varphi}^S(x_i) = \varphi^S(x_i)$ d'où
   $\hat{\varphi}^E(x_i) = \hat{\varphi}^S(x_i)$
4. $marque(x_i) = (-x_{i-1})$ **et** $marque(x_{i+1}) = (-x_i)$.



   On a alors $\hat{\varphi}^S(x_i) = \varphi^S(x_i) - \varepsilon$ et $\hat{\varphi}^E(x_i) = \varphi^E(x_i) - \varepsilon$ d'où
   $\hat{\varphi}^E(x_i) = \hat{\varphi}^S(x_i)$

ce qui montre que $\hat{\varphi}$ est un flot.

Montrons que $\hat{\varphi}$ est compatible; c'est immédiat par construction, puisque

1. sur un arc augmentant (x_{i-1},x_i), $\hat{\varphi}$ a été augmenté de $\varepsilon \leq \varepsilon_i = c(x_{i-1},x_i) - \varphi(x_{i-1},x_i)$, donc $\hat{\varphi}(x_{i-1},x_i) \leq c(x_{i-1},x_i)$
2. sur un arc diminuant (x_i,x_{i-1}), $\hat{\varphi}$ a été diminué de $\varepsilon \leq \varphi(x_i,x_{i-1})$ et donc $\hat{\varphi}(x_i,x_{i-1}) \geq 0$

Enfin, $\hat{\varphi}$ a été augmenté sur un arc d'entrée (e,x_1), de la quantité ε, donc $v(\hat{\varphi}) = v(\varphi) + \varepsilon$ et comme $\varepsilon > 0$, on a bien $v(\hat{\varphi}) > v(\varphi)$ □

11.4 Algorithme de Ford-Fulkerson

La mise en œuvre algorithmique des fonctions de marquage et d'amélioration se fait par une méthode d'exploration issue de e. Cette exploration peut être menée avec n'importe quelle stratégie: largeur, profondeur, etc. Quelle que soit la stratégie retenue, l'exploration est adaptée de manière à:

1. opérer sur le graphe valué $G^{\varepsilon}(\varphi)$ (appelé *graphe d'écart*) constitué de l'union du graphe partiel de G constitué des arcs non saturés et du graphe partiel de G^t (transposé de G) constitué des arcs chargés, soit, formellement:

   $G^{\varepsilon}(\varphi) = (X \cup \{e\} \cup \{s\}, \Gamma^{\varepsilon}, c(\varphi))$ avec

   $$\begin{aligned}\Gamma^{\varepsilon}(\varphi) &= \{(x,y) \mid (x,y) \in \Gamma \wedge \varphi(x,y) < c(x,y)\} \cup \\ &\quad \{(x,y) \mid (y,x) \in \Gamma \wedge \varphi(y,x) > 0\} \text{ et}\end{aligned}$$

   $$c(\varphi)(x,y) = \begin{cases} c(x,y) - \varphi(x,y) & \text{si} \quad (x,y) \in \Gamma \wedge \varphi(x,y) < c(x,y) \\ \varphi(y,x) & \text{si} \quad (y,x) \in \Gamma \wedge \varphi(y,x) > 0 \end{cases}$$

   Pour fixer les idées, la figure 11.2 donne un exemple de graphe d'écart associé à un flot.
2. associer à chaque sommet y visité un attribut ε tel que $\varepsilon(y)$ est la valeur du chemin de G^{ε} (égale au minimum des valeurs des arcs) à l'issue duquel y a été visité pour la première fois,
3. rajouter la condition d'arrêt s visité.

11.4.1 Fonction marquage

On en donne une version récursive, analogue au calcul récursif de l'ensemble des descendants d'un sommet donné (algorithme 6.4.2). Ici, l'exploration est celle des descendants du sommet e, dans le graphe d'écart. Sur l'exploration proprement dite, la fonction greffe un calcul, en affectant à chaque sommet marqué sa marque (chaîne de caractères) et son écart d'amélioration (l'entier ε). Ces deux informations seront exploitées par la procédure d'amélioration. Enfin, la condition d'arrêt est modifiée: le marquage s'arrête lorsque l'exploration ne permet plus de marquer de nouveaux sommets (comme dans le calcul des descendants) ou lorsque le sommet s est marqué.

11.4.2 Procédure amélioration

Précondition: s est marqué ($s \in M$).

Invariant:
debchaine=tête de la chaîne le long de laquelle φ est modifié $\wedge$
flot modifié entre *debchaine* et s

Condition d'arrêt: $debchaine = e$

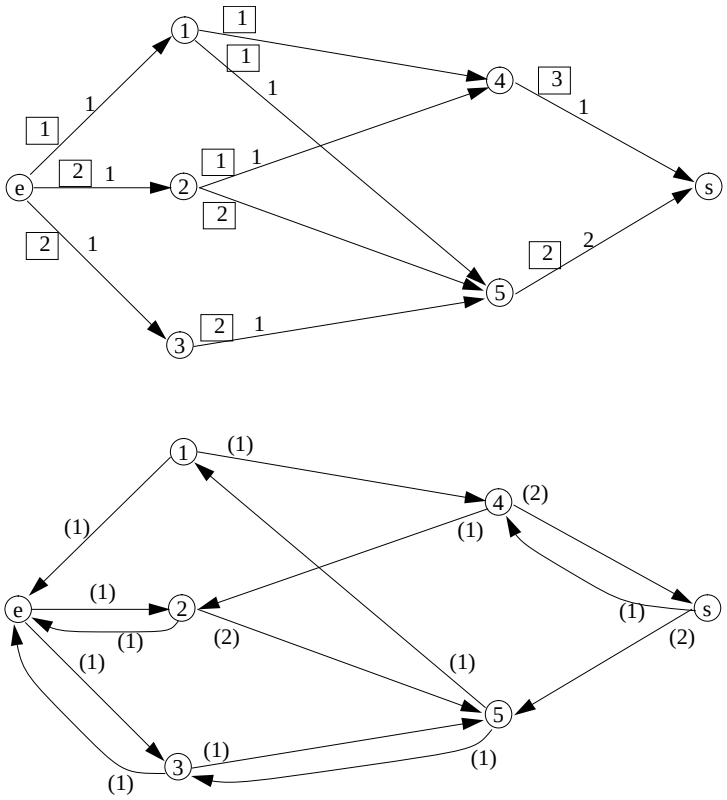


FIG. 11.2 – *Flot et graphe d'écart associé*

11.4.3 Algorithme

Invariant:
G est muni d'un flot compatible G.φ.

Condition d'arrêt $marquage(G)$

Progression:
$amélioration(G)$;

Valeurs initiales: Ici, il s'agit de calculer un flot compatible initial. Une première solution, triviale, consiste à prendre G.$\varphi \equiv 0$, qui répond évidemment à la question.

Une seconde solution, qui sera intéressante lorsqu'on résoud le problème "à la main", notamment sur une représentation graphique, consiste à chercher d'abord un flot *complet*:

Définition 11.13 *Un flot compatible est dit complet si tout chemin de e à s possède au moins un arc saturé.*

Autrement dit, un flot compatible n'est pas complet si, et seulement si, la fonction `marquage` permet de marquer s en n'utilisant que des marquages avant.

La construction d'un flot complet s'effectue en explorant les chemins de e à s et en affectant uniformément sur chaque chemin un flux égal à la plus petite capacité résiduelle des arcs du chemin. Les flux ainsi obtenus sont superposés (c'est-à-dire leurs valeurs additionnées sur chaque arc) et lorsqu'aucun chemin ne peut plus recevoir de flux supplémentaire, on a un flot compatible complet.

Il est clair qu'un flot de valeur maximum est complet, mais que la réciproque est fausse, comme l'exemple traité dans le §11.5 suivant va le montrer.

Le texte complet de l'algorithme est donné pages 173.

11.4.4 Exemple

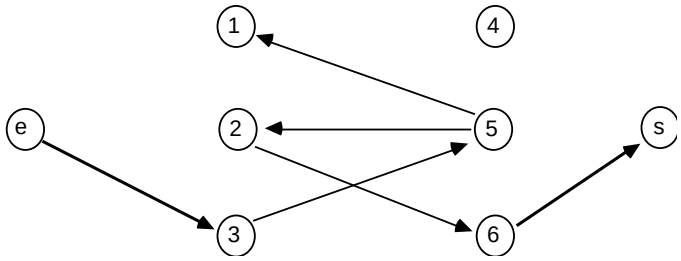
Considérons le réseau de transport de la figure 11.3

Initialisation Recherche d'un flot complet

chemin (e,1,4,s); flux = 1 ; arc (1,4) saturé
chemin (e,1,5,s); flux = 1 ; arcs (e,1),(1,5) saturés
chemin (e,2,4,s); flux = 1 ; arcs (2,4),(4,s) saturés
chemin (e,2,5,s); flux = 1 ; arcs (e,2),(2,5),(5,s) saturés
chemin (e,3,6,s); flux = 1 ; arc (3,6) saturé
Flot représenté sur la figure 11.4 (les arcs saturés sont en gras)

marquage: Arborescence en largeur

$epsil = 1$



marquage reporté sur la figure 11.4

premier pas d'itération amélioration

$\hat{\varphi}(6,s)\leftarrow 2; \hat{\varphi}(2,6)\leftarrow 2; \hat{\varphi}(2,5)\leftarrow 0; \hat{\varphi}(3,5)\leftarrow 2; \hat{\varphi}(e,3)\leftarrow 2$

marquage: $e(+)$ stop; ce flot est donc maximum

11.4.5 Complexité

La complexité de l'algorithme de Ford-Fulkerson n'est pas polynomiale. En effet, le nombre d'étapes de l'algorithme est majoré par la capacité minimale $\underline{c}^+(A)$ des coupes du réseau, puisqu'à chaque étape la valeur de φ augmente au moins de 1, et que d'après le lemme 11.7 on a $v(\varphi) \leq \underline{c}^+(A)$. Or cette capacité ne peut être majorée à partir de la taille du graphe (exprimée en fonction de n et m, nombre de sommets et d'arcs). A une même configuration topologique de réseau peuvent en effet être attribuées des capacités qui sont des nombres entiers quelconques, donc *a priori* non bornés.

Signalons que des algorithmes de complexité polynomiale en fonction du nombre n de sommets ont été proposés: algorithmes de Edmonds-Karp, de Dinic, de Karzanov (ce dernier ayant une complexité globale en $O(n^3)$) [Din70, EK72, Kar74, MKM78].

11.5 Cas des réseaux bi-partis : mise en œuvre par tableaux

Dans ce paragraphe, nous considérons le cas particulier de réseaux de la forme

$G = (X,Y,e,s\ ;\ \Gamma_e,\Gamma,\Gamma_s\ ;\ c)$ où

$X \cap Y = \emptyset$, $e \notin X \cup Y$, ,$s \notin X \cup Y$

$\Gamma_e \subseteq \{(e,x), x \in X\}$, $\Gamma \subseteq X \times Y$, $\Gamma_s \subseteq \{(y,s), y \in Y\}$

Supposons que $X = \{x_1,x_2,\ldots,x_n\}$ et $Y = \{y_1,y_2,\ldots,y_p\}$. Un tel réseau peut alors être représenté par un tableau C de format $(n+1) \times (p+1)$, avec

$c_{i,j}$	=	$c(x_i,y_j)$,	$1 \leq i \leq n, 1 \leq j \leq p$
$c_{i,p+1}$	=	$c(e,x_i)$,	$1 \leq i \leq n$
$c_{n+1,j}$	=	$c(y_j,s)$,	$1 \leq j \leq p$
$c_{n+1,p+1}$	non défini		

Exemple

	y_1	y_2	y_3	y_4	y_5	y_6	
x_1	20	10	30	0	20	0	30
x_2	30	20	0	30	30	0	70
x_3	0	20	20	20	0	20	80
x_4	0	10	10	10	0	40	60
x_5	30	0	20	0	0	20	60
	40	20	40	40	30	70	

Mise en œuvre de Ford-Fulkerson. Cette mise en œuvre peut être faite directement à l'aide d'un tel tableau: la fonction φ est représentée dans un tableau de même format. Pour que

φ soit un flot compatible, il faut et il suffit d'assurer:

1) $\forall i\ 1 \leq i \leq n \Rightarrow \sum_{j=1}^{p} \varphi_{i,j} = \varphi_{i,p+1}$
(conservation au nœud x_i)
$\forall j\ 1 \leq j \leq p \Rightarrow \sum_{i=1}^{n} \varphi_{i,j} = \varphi_{n+1,j}$
(conservation au nœud y_j)

2) $1 \leq i \leq n+1, 1 \leq j \leq p+1 \Rightarrow 0 \leq \varphi_{i,j} \leq c_{i,j}$
(compatibilité)

De plus, la valeur de φ est donnée par

$$v(\varphi) = \sum_{i=1}^{n} \varphi_{i,p+1} = \sum_{j=1}^{p} \varphi_{n+1,p}$$

Les différentes étapes sont alors :

Obtention d'un flot complet initial. Saturer le tableau φ ligne par ligne (méthode dite du *coin Nord-ouest*), c'est-à-dire:

pour toute `ligne i` **depuis** `1` **jqa** `n`
  $\varphi_{i,p+1} \leftarrow$ `0 ;`
  pour toute `colonne j,` **tant que** $\varphi_{i,p+1} < c_{i,p+1}$
    $\varphi_{i,j} \leftarrow \max(c_{i,j},\ c_{i,p+1} - \varphi_{i,p+1},\ c_{n+1,j} - \varphi_{n+1,j})$;
    $\varphi_{i,p+1} \leftarrow \varphi_{i,p+1} + \varphi_{i,j}$;
    $\varphi_{n+1,j} \leftarrow \varphi_{n+1,j} + \varphi_{i,j}$
  fpour `colonnes`
fpour `lignes`

Si $\varphi_{i,p+1} = c_{i,p+1}$ on dira que la ligne i est *saturée* (cela correspond à la saturation de l'arc d'entrée (e,x_i)). De même si $\varphi_{n+1,j} = c_{n+1,j}$ pour la colonne j (saturation de l'arc de sortie (y_j,s)).

Marquage Le marquage consiste ici à alterner des phases de marquage *avant* et marquages *arrière*, où

- une phase de marquage avant consiste à marquer de nouvelles colonnes à partir des lignes déjà marquées,
- une phase de marquage arrière consiste à marquer de nouvelles lignes à partir des colonnes déjà marquées.

Initialement, toute ligne i non saturée est marquée $(+e)$
Ensuite, les phases avant et arrière se déroulent comme suit:

- phase de marquage avant :

pour toute `ligne i` *marquée à la phase précédente*
  pour toute `colonne j` **depuis** `1` **jqa** `p`
    si `j` non marquée **et** $\varphi_{i,j} < c_{i,j}$
      alors `marquer (+i) la colonne j`
    fsi

fpour *colonne*
fpour *ligne*

– phase de marquage arrière :

pour toute *colonne j* *marquée à la phase précédente*
pour toute *ligne i* **depuis** *1* **jqa** *n*
si *i* non marquée **et** $\varphi_{i,j} > 0$
alors *marquer (-j) la ligne i*
fsi
fpour *ligne*
fpour *colonne*

Arrêt: les deux situations d'arrêt se produisent lorsque:

1: s **est marqué** : lors d'une phase *avant* une colonne j *non saturée* est marquée. Cela correspond au sommet y_j marqué et l'arc (y_j,s) non saturé.

2: blocage : lors d'une phase (avant ou arrière) aucune nouvelle rangée (ligne ou colonne selon les cas) n'a pu être marquée **et** toutes les colonnes marquées sont saturées.

Dans le premier cas, *marquage(G)* rend *faux* (G.φ non maximum), dans le deuxième cas, *marquage(G)* rend *vrai* (G.φ maximum)

Amélioration(G) (dans le cas où s est marqué).

Partant de la colonne marquée non saturée j_k qui a provoqué l'arrêt du marquage, on remonte la chaîne de marquage jusqu'à trouver une ligne i_1 marquée $(+e)$: c'est une chaîne de marques alternées:

$$(+i_k),(-j_{k-1}),(+ik-1),\ldots,(+i_2),(-j_1),(+e)$$

Simultanément, on calcule $\varepsilon = \min(\varepsilon_+,\varepsilon_-)$, où

$$\varepsilon_+ = \min_{\ell=1,\ldots,k}(c_{i_\ell,j_\ell} - \varphi_{i_\ell,j_\ell})$$

$$\varepsilon_- = \min_{\ell=2,\ldots,k} \varphi_{i_\ell,j_{\ell-1}} \text{(voir figure ci-dessous)}$$

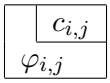
puis on ajuste φ en ajoutant/retranchant alternativement ε:

$$\varphi_{i_\ell,j_\ell} \leftarrow \varphi_{i_\ell,j_\ell} + \varepsilon,\ \varphi_{i_\ell,j_{\ell-1}} \leftarrow \varphi_{i_\ell,j_{\ell-1}} - \varepsilon$$

puis

$$\varphi_{n+1,j_k} \leftarrow \varphi_{n+1,j_k} + \varepsilon,\ \varphi_{i_1,p+1} \leftarrow \varphi_{i_1,p+1} + \varepsilon$$

Exemple Nous traitons maintenant l'exemple donné au début de ce paragraphe. Un seul tableau contient les flux et les capacités. Chaque case se présente comme suit:



Flot complet initial et marquage correspondant

		$(+x_5)$	$(+x_3)$	$(+x_5)$	$(+x_3)$	$(+x_1)$	$(+x_5)$	
		y_1	y_2	y_3	y_4	y_5	y_6	
$(-y_1)$	x_1	20 / 20	10 / 10	30 / 0	0 / 0	20 / 0	0 / 0	30 / 30
$(-y_1)$	x_2	30 / 20	20 / 10	0 / 0	30 / 30	30 / 10	0 / 0	70 / 70
$(+e)$	x_3	0 / 0	20 / 0	20 / 20	20 / 10	0 / 0	20 / 20	80 / 50
$(+e)$	x_4	0 / 0	10 / 0	10 / 10	10 / 0	0 / 0	40 / 40	60 / 50
$(+e)$	x_5	30 / 0	0 / 0	20 / 10	0 / 0	0 / 0	20 / 10	60 / 20
		40 / 40	20 / 20	40 / 40	40 / 40	30 / 10	70 / 70	v=220

Amélioration du flot: La chaîne de marquage est:

(y_5,s)	de	capacité résiduelle	20
(x_1,y_5)		” ”	20
(x_1,y_1)		” ”	-20
(x_5,y_1)		” ”	30
(e,x_5)		” ”	40

soit un flux à transférer de 20.

Texte de la fonction

```
ENT epsil ;
ENT SOMMET::ε ;
STRING SOMMET::marq ;
 local ENS[SOMMET] M
BOOL marquage(GRAPHE G)  c'est
 début
  M ← ∅ ; epsil ← 0 ;
  marquer(e, ''+'', +∞) ;
  si s ∈ M alors epsil ← s.ε fsi ;
  Result ← s ∉ M
 fin


marquer(SOMMET x; STRING m; ENT ecart)  c'est
   precondition x ∉ M
   local ENT e
 début
   x.marq ← m ;
   x.ε ← ecart ;
   M ← M ∪ { x } ;
   si x ≠ s
    alors
     --   marquages avant
     pour tout y de G.lst_succ(x) faire
       si y ∉ M et φ(x,y) < c(x,y)
         alors e ← min(x.ε,c(x,y)-φ(x,y)) ;
               marquer(y, ''+x'' , e)
      fsi
    fpourtout ;
     --   marquages arriere
     pour tout z de G.lst_pred(x) faire
       si z ∉ M et φ(z,x) > 0
         alors e ← min(x.ε,φ(z,x)) ;
               marquer(z, ''-x'' , e)
      fsi
    fpourtout
  fsi
 fin
--   à la sortie de cette procédure, soit s est marqué (en avant à partir de x) soit tous les sommets marquables à partir de x sont marqués.
```

Texte de la procédure

```
amelioration(GRAPHE G) ;
   local SOMMET debchaine ;
 début
    depuis debchaine ← s ;
    jusqua debchaine = e
    faire
       si debchaine.marq = ''+x''
          alors G.φ(x,debchaine) ← G.φ(x,debchaine) + G.epsil
          sinon G.φ ← G.φ(debchaine,x) - G.epsil
       fsi ;
      debchaine ← x
    fait ;
 fin
```

Algorithme de FORD-FULKERSON

```
ford-fulkerson(GRAPHE G)  c'est
 début
    depuis G.φ ← initial --   délivre un flot initial compatible; le flot ≡ 0 convient
    jusqua marquage(G)
    faire
      amelioration(G)
    fait
 fin
```

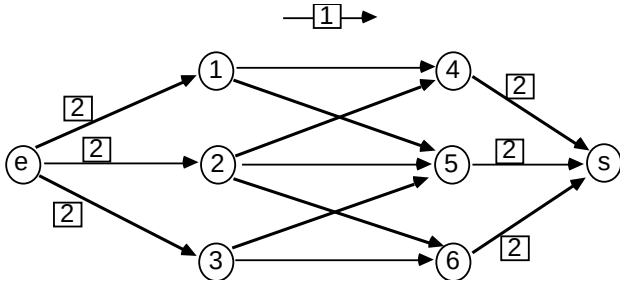


FIG. 11.3 – *Exemple d'application de Ford-Fulkerson*

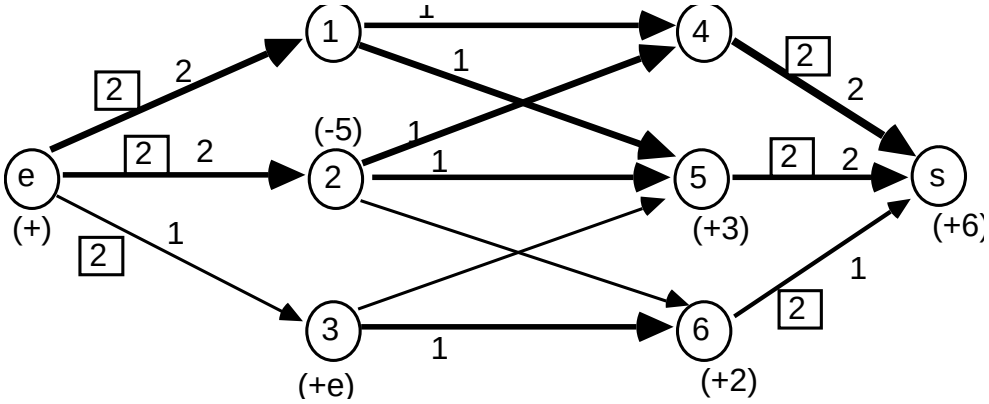


FIG. 11.4 – *flot complet et marquage*

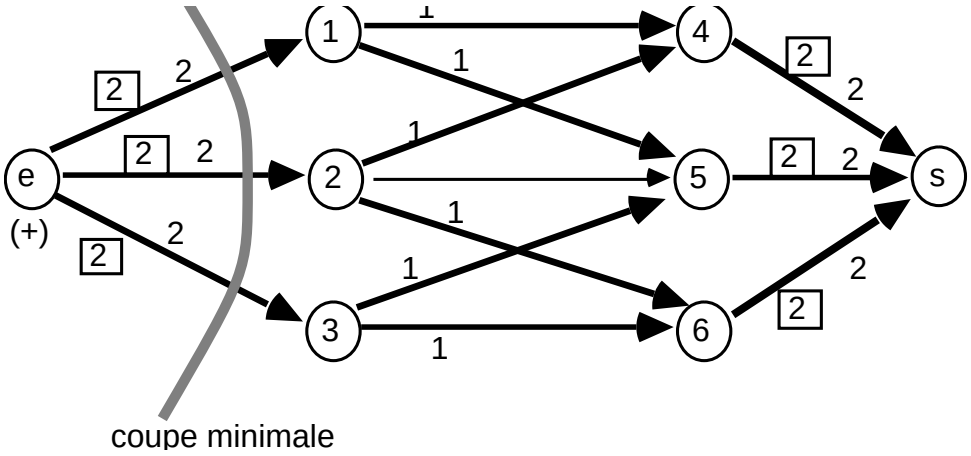


FIG. 11.5 – *deuxième flot et marquage*

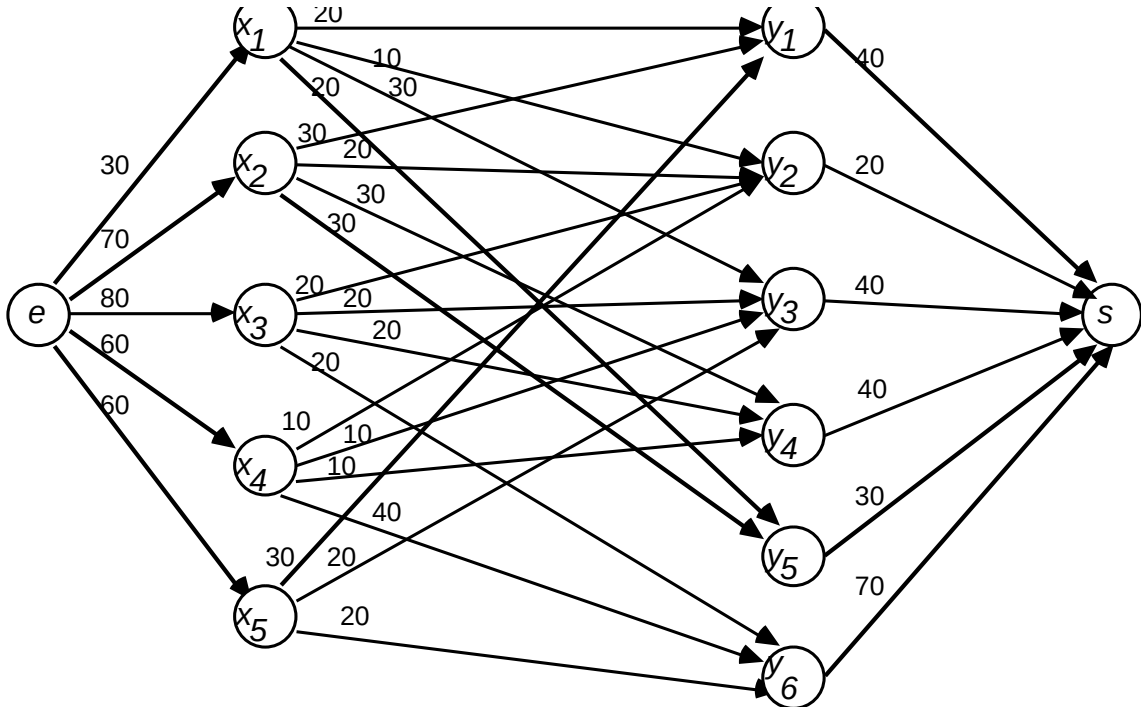


FIG. 11.6 – *Réseau bi-parti*

Deuxième flot et marquage correspondant

		$(+x_5)$	$(+x_3)$	$(+x_5)$	$(+x_3)$	$(+x_2)$	$(+x_5)$	
		y_1	y_2	y_3	y_4	y_5	y_6	
$(-y_2)$	x_1	20 / 0	10 / 10	30 / 0	0 / 0	20 / 20	0 / 0	30 / 30
$(-y_1)$	x_2	30 / 20	20 / 10	0 / 0	30 / 30	30 / 10	0 / 0	70 / 70
$(+e)$	x_3	0 / 0	20 / 0	20 / 20	20 / 10	0 / 0	20 / 20	80 / 50
$(+e)$	x_4	0 / 0	10 / 0	10 / 10	10 / 0	0 / 0	40 / 40	60 / 50
$(+e)$	x_5	30 / 20	0 / 0	20 / 10	0 / 0	0 / 0	20 / 10	60 / 40
		40 / 40	20 / 20	40 / 40	40 / 40	30 / 30	70 / 70	v=240

Ce flot est maximal car il sature les sorties.

11.6 Application: problèmes de couplage des graphes bi-partis

Les problèmes de couplage de cardinalité maximale constituent une des applications les plus classiques des modèles de flots. Ils interviennent aussi comme étape dans la résolution de problèmes d'affectation à coût optimal, largement utilisés en optimisation combinatoire. Historiquement, la théorie du couplage maximum d'un graphe bi-parti a précédé la théorie des flots; elle apparaît maintenant comme une application de cette dernière aux aux graphes bi-partis. En fait, le résultat fondamental, du à König et Hall, résulte de l'application aux réseaux bi-partis modélisant le problème du lemme de Ford-Fulkerson, de même l'algorithme résolvant le problème (algorithme de Evegary et Kühn) est l'adaptation de l'algorithme de Ford-Fulkerson.

11.6.1 Le problème

Soit Y et Z deux ensembles finis et Γ une relation sur $Y \times Z$, ou encore, ce qui est équivalent, une application de Y dans 2^Z (ensemble des parties de Z). Le triplet (Y,Z,Γ) est en fait un graphe bi-parti G, avec $Y \cup Z$ comme ensemble de sommets, et Γ comme ensemble d'arcs.

Définition 11.14 *On appelle couplage du graphe bi-parti G une partie $A \subseteq Y$ et une bijection $\varphi : A \to \varphi(A)$ telle que $\varphi(A) \subseteq Z$ et $\forall x \in A$: $\varphi(x) \in \Gamma(x)$.*

Définition 11.15 *Si (A,φ) est un couplage de G alors $|A| = |\varphi(A)|$ s'appelle le cardinal (ou valeur) du couplage.*

Remarque Un couplage peut aussi être vu comme un sous-ensemble W de l'ensemble des arcs Γ du graphe bi-parti $G = (Y,Z,\Gamma)$ tel que deux arcs distincts de W n'aient ni origine ni extrémité commune (figure 11.7).

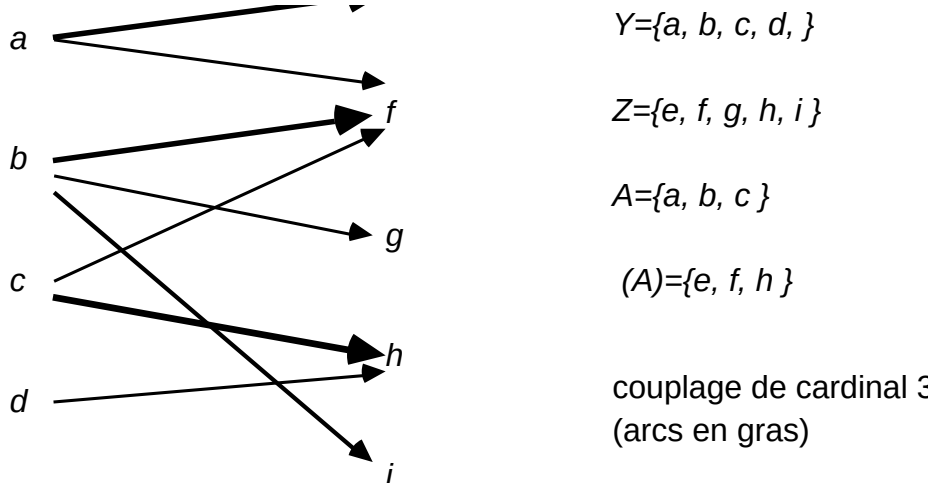


FIG. 11.7 – *Couplage du graphe G*

Problème 11.1 *Étant donné un graphe bi-parti* $G = (Y,Z,\Gamma)$, *déterminer un couplage de* G, *de cardinalité maximum*

Remarque S'il existe un couplage de valeur $|Y|$, on dit qu'*on peut coupler* Y *dans* Z.

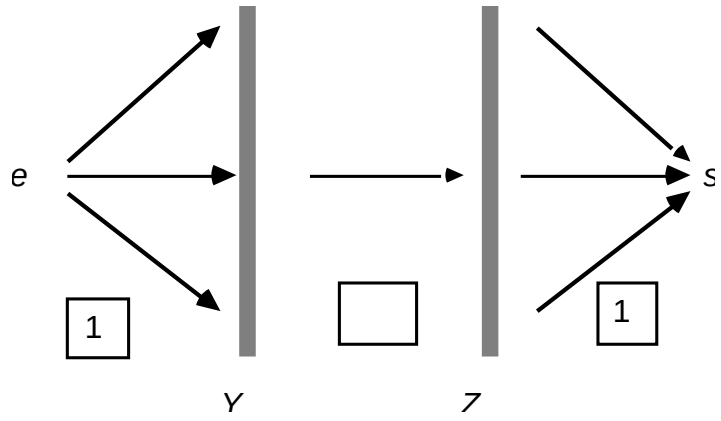
11.6.2 Modélisation en réseau bi-parti

Au graphe bi-parti $G = (Y,Z,\Gamma)$ on associe le réseau de transport

$G' = (Y,Z,e,s;\Gamma';c')$ avec

$$\Gamma' = \Gamma \cup \{(e,y),y \in Y\} \cup \{(z,s),z \in Z\}$$

$$\forall y \in Y,\forall z \in Z \begin{cases} c(e,y) &= c(z,s) = 1 \\ c(y,z) &= +\infty \end{cases}$$



On vérifie facilement que tout flot compatible sur G' est un couplage de G et réciproquement, dont les valeurs sont égales. Il suffit pour cela de considérer la correspondance

$z = \varphi(y) \Leftrightarrow \varphi(y,z) = 1$

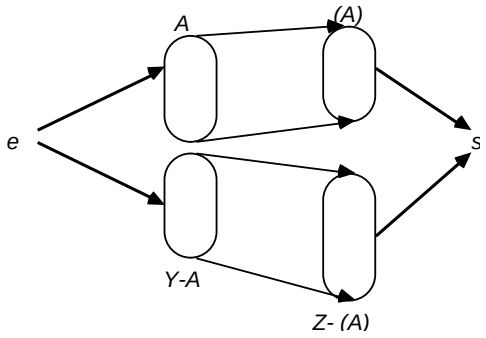
Nous donnons le résultat fondamental de la théorie du couplage maximum, du à König et Hall:

Théorème 11.16 *Soit $G = (Y,Z,\Gamma)$ un graphe bi-parti. Il existe un couplage de valeur Y si, et seulement si,*

$\forall A \subseteq Y : \quad |A| \leq |\Gamma(A)|$

Démonstration. i) Supposons qu'il existe un couplage φ de valeur $|Y|$, et considérons le réseau de transport associé. Soit $A \subseteq Y$. Alors $A' = \{e\} \cup A \cup \Gamma(A)$ définit une coupe $\Gamma'^S(A')$. Calculons sa capacité:

les arcs sortants de A' sont:
- (e,y) avec $y \in Y \setminus A$
- (z,s) avec $z \in \Gamma(A)$



et comme $(y,z) \in \Gamma \wedge y \in A \Rightarrow z \in \Gamma(A)$, aucun arc de Γ ne sort de A'. D'où:

$c^S(A') = |Y \setminus A| + |\Gamma(A)| = |Y| - |A| + |\Gamma(A)|$

D'après le lemme de Ford-Fulkerson,

$v(\varphi) = |Y| \leq c^S(A') = |Y| - |A| + |\Gamma(A)|$

c'est-à-dire $|A| \leq |\Gamma(A)|$

ii) Réciproquement, soit φ un flot de valeur maximale sur le réseau de transport associé à G, et soit $A' = \{e, A \subseteq Y, B \subseteq Z\}$ l'ensemble des sommets marqués par le marquage associé à $G.\varphi$. D'après le théorème de Ford-Fulkerson, $v(\varphi) = c^S(A') < \infty$ (car $v(\varphi) \leq |Y| < \infty$) donc la coupe $\Gamma'^+(A')$ ne comporte aucun arc $(y,z) \in \Gamma$. Par conséquent $\Gamma(A) \subseteq B$ soit encore $|\Gamma(A)| \leq |B|$ et aussi :

$$c^S(A') = |Y| - |A| + |B| \geq |Y| - |A| + |\Gamma(A)|$$

Comme, par hypothèse, $|A| \leq |\Gamma(A)|$, il vient $v(\varphi) = c^S(A') \geq |Y|$ d'où, par double inégalité, $v(\varphi) = |Y|$

□

La condition nécessaire et suffisante établie par ce théorème est évidemment impraticable pratiquement, puisqu'il faudrait la vérifier sur les $2^{|Y|}$ parties de Y. Pratiquement, il suffit de calculer un couplage de valeur maximum en appliquant l'algorithme de Ford-Fulkerson adapté au réseau bi-parti modélisant le problème (§11.5) (ou un autre algorithme de calcul de flot maximal), puis de comparer la valeur de ce couplage à $|Y|$.

Du point de vue algorithmique, la mise en œuvre de l'algorithme de Ford-Fulkerson bi-parti est ici plus simple que pour un réseau bi-parti quelconque: la colonne et la ligne supplémentaires portant les capacités d'entrée et de sortie deviennent inutiles (puisque ces capacités sont implicitement égales à 1); de même, les capacités des arcs de Γ n'ont pas besoin d'être indiquées: toute case du tableau est soit *interdite* si elle ne correspond pas à un arc de Γ (capacité 0), soit *autorisée*, c'est-à-dire de capacité ∞. Enfin, les flux ne peuvent prendre que les valeurs 0 ou 1: il suffira donc de "marquer" les cases autorisées avec un signe indiquant si elles sont sélectionnées (flux 1) ou non (flux 0).

En fait, le problème revient à sélectionner, parmi les cases autorisées, un ensemble de cases tel qu'au plus une case par rangée soit sélectionnée (*couplage*) et, parmi tous les ensembles répondant à la question, en donner un de cardinal maximum (*couplage maximum*). Une rangée sera dite *saturée* si une case de cette rangée est sélectionnée (cela correspond à la saturation d'un arc d'entrée si la rangée est une ligne, ou d'un arc de sortie si la rangée est une colonne). Enfin, une case autorisée ne peut jamais être saturée (car elle est de capacité ∞).

Pour illustrer l'intérêt des modèles de couplage et la mise en œuvre pratique de l'algorithme de Ford-Fulkerson, nous traitons ci-après un exemple concret de calcul de *système de représentants distincts*.

11.6.3 Exemple: système de représentants distincts

Soit $X = \{x_1, \ldots, x_n\}$ un ensemble fini, et $Y_1, \ldots, Y_m$ des parties de X: chaque Y_j regroupe des éléments de X selon un certain critère. On appelle *système de représentants distincts* (srd) un sous-ensemble $E \subseteq X$ dont chaque élément représente *une* et *une seule* partie Y_j. On a donc: $E = \{x_{i_1}, \ldots, x_{i_m}\}$ avec

$$k \neq \ell \Rightarrow x_{i_k} \neq x_{i_\ell} \text{ et } x_{i_k} \in Y_k, k = 1, \ldots, m.$$

On se ramène au problème de couplage d'un graphe bi-parti de la manière suivante: soit $Y = \{Y_1, \ldots, Y_m\}$, $Z \equiv X$ et $(Y_j, x_i) \in \Gamma \Leftrightarrow x_i \in Y_j$.

Tout couplage sur (Y,Z,Γ) définit donc une bijection d'une partie de Y sur une partie de Z, c'est-à-dire qu'à tout Y_j correspond un élément bien défini de $Z \equiv X$ (son "représentant"), et deux Y_j distincts ont deux représentants distincts.

Considérons l'exemple suivant:
X = {Jean, Anita, Louis, Mireille, Albert, Patricia, Florent} et

Y_1	=	{Jean, Anita, Louis}	(moins de 20 ans)
Y_2	=	{Mireille, Albert }	(chômeurs)
Y_3	=	{Anita, Louis, Patricia }	(amateurs de théâtre)
Y_4	=	{Mireille, Patricia }	(mères de famille)
Y_5	=	{Jean, Anita }	(étudiants)
Y_6	=	{Jean, Louis }	(membres de l'association sportive)
Y_7	=	{Mireille, Florent }	(membres du comité d'action sociale)

		($+Y_5$) Jean	($+Y_5$) Anita	($+Y_1$) Louis	($+Y_4$) Mireille	($+Y_2$) Albert	($+Y_3$) Patricia	Florent
(-Je)	Y_1	⊗	×	×				
(-Mi)	Y_2				⊗	×		
(-An)	Y_3		⊗	×			×	
(-Pa)	Y_4				×		⊗	
($+e$)	Y_5	×	×					
(-Lo)	Y_6	×		⊗				
	Y_7				×			⊗

Couplage initial (méthode du coin Nord-ouest): $v = 6$. Ce couplage initial ne permet de choisir que 6 représentants (les étudiants ne sont pas représentés). Le marquage, effectué sur le tableau ci-dessus, permet de marquer Albert, colonne non saturée. Ce couplage n'est donc pas maximal. La phase d'amélioration donne le couplage ci-dessous, qui, lui, est maximal (valeur $|Y|$=7)

	Jean	Anita	Louis	Mireille	Albert	Patricia	Florent
Y_1	⊗	×	×				
Y_2				×	⊠		
Y_3		×	×			⊠	
Y_4				⊠		×	
Y_5	×	⊠					
Y_6	×		⊗				
Y_7				×			⊗

(en ⊠ nous avons indiqué les nouvelles cases sélectionnées dans cette solution).

Jean	représente	les moins de 20 ans
Anita	...	les étudiants
Louis	...	l'association sportive
Mireille	...	les mères de famille
Albert	...	les chômeurs
Patricia	...	les amateurs de théâtre
Florent	...	le comité d'action sociale

Annexe A

Les opérations de la classe GRAPHE et leur sémantique

Dans ce qui suit, un graphe non valué `G` est un doublet `(X, Γ)` et un graphe valué `G` est un triplet `(X, Γ), v)`

Constructeur

creer

post *vide*

`G.creer` *crée un nouvel objet de type graphe, désigné par G, vide*

Opérations de consultation (lecture)

Fonctions à résultat booléen

BOOL vide

post *Result* $\Rightarrow$ *nm_som=0* $\wedge$ *nb_arc=0*

BOOL validsom(SOMMET x)

renvoie `vrai` **ssi** `x` *est un sommet du graphe.*

BOOL validarc(SOMMET x, y)

post *validarc(x,y)* $\Rightarrow$ *validsom(x)* $\wedge$ *validsom(y)*

renvoie `vrai` **ssi** `(x,y)` *est un arc du graphe*

Graphes valués

E v(SOMMET x, y) : E

pre *valid_arc(x,y)*

`v(x,y)` = *valeur de l'arc* `(x,y)`.

Fonctions à résultat entier

ENT nb_som
: *nombre de sommets*

ENT nb_arc
: *nombre d'arcs*

ENT nb_succ(SOMMET x)
: **pre** *validsom(x)*
renvoie le nombre de successeurs de x.

ENT nb_pred(SOMMET x)
: **pre** *validsom(x)*
renvoie le nombre de prédécesseurs de x.

Parcours de l'ensemble des sommets

BOOL som_termine
: *som_termine* = vrai **ssi** *le parcours de l'ensemble des sommets est épuisé.*

premier_som
: *premier_som positionne le sommet courant sur le premier sommet.*

prochain_som
: **pre non** *som_termine*
prochain_som avance le sommet courant au prochain sommet.

SOMMET som_cour
: **pre non** *som_termine*
renvoie le sommet courant du parcours.

Parcours de l'ensemble des arcs

BOOL arc_termine
: vrai **ssi** *le parcours de l'ensemble des arcs est épuisé.*

premier_arc
: *positionne l'arc courant sur le premier arc de l'ensemble des arcs.*

prochain_arc
: **pre non** *arc_termine*
avance l'arc courant au prochain arc.

ARC arc_cour

pre non *arc_termine*
renvoie l'arc courant du parcours.

Fonctions à résultat ensemble de sommets

ENS[SOMMET] ens_som
G.ens_som = ensemble des sommets de G.

ENS[SOMMET] lst_succ(SOMMET x)
pre *validsom(x)*
G.lst_succ(x) = $\Gamma(x)$.

ENS[SOMMET] lst_pred(SOMMET x)
pre *validsom(x)*
G.lst_pred(x) = $\Gamma^{-1}(x)$.

ENS[SOMMET] points_entrée
post x ∈ points_entrée ⇒ lst_pred(x)=∅
points_entrée = $\{x \in X \mid \Gamma^{-1}(x) = \emptyset\}$.

ENS[SOMMET] points_sortie
post x ∈ points_sortie ⇒ lst_succ(x)=∅
G.points_sortie = $\{x \in X \mid \Gamma(x) = \emptyset\}$.

Procédures de modification (écriture)

Graphes non valués

ajout_sommet(SOMMET x)
post *validsom(x)*
G.ajout_sommet(x) ajoute le sommet x au graphe G.
Sans effet si x est déjà un sommet.

ajout_arc(SOMMET x,y)
post *validarc(x,y)*
G.ajout_arc(x,y) ajoute les sommets x, y et l'arc (x,y) au graphe G.
Sans effet si (x,y) est déjà un arc.

oter_sommet(SOMMET x)
pre non *vide*
post non *validsom(x)*
G.oter_sommet(x) ôte le sommet x et tous les arcs adjacents à x .

oter_arc(SOMMET x,y)

pre non *vide*
post non *validarc(x,y)*
G.oter_arc(x,y) ôte l'arc (x,y).

sous_graphe(ENS[SOMMET] E)
pre non *vide*
post $x \in E \Rightarrow$ **non** *validsom(x).*
G.sous_graphe(E) ôte de G tous les sommets appartenant à E.

Graphes valués

ajout_val(SOMMET x, y; E ϖ)
post *valid_arc(x,y) ; v(x,y) = ϖ*
ajout_val(x, y ,ϖ) ajoute l'arc (x,y) de valeur ϖ.
Si l'arc existe déjà, remplace sa valeur

modif_val(SOMMET x, y; E ϖ)
pre *valid_arc(x,y)*
post *v(x,y) = ϖ*
modif_val(x, y, ϖ) modifie la valeur de l'arc.

Invariants

vide $\Rightarrow$ *arcvide*
arcvide $\Rightarrow$ *hors_arc*

Bibliographie

[Ber70] C. Berge. *Graphes et Hypergraphes.* Dunod, 1970.

[BLW76] N. L. Biggs, E. K. Llyod and R. J. Wilson. *Graph Theory 1736-1936.* Oxford University Press, 1976.

[CP84] A. Couvert, R. Pedrono. *Techniques de Programmation.* Cours C 45, IFSIC, Université de Rennes, 1984.

[Cri75] N. Christofides. *Graph Theory ; an algorithmic approach.* Academic Press, 1975.

[Din70] Dinic, G. A. Algorithm for solution of a problem of maximum flow in a network with lower estimation. *Soviet. Math. Dokl.*, 11:1277-1280, 1970.

[EK72] Edmonds, J. and Karp, R. M. Theoretical improvements in algorithmic efficiency for network flow problems. *J. ACM* 19(2):248-264, 1972.

[GM79] M. Gondran, M. Minoux. *Graphes et Algorithmes.* Eyrolles, 1979.

[HP82] J. M. Hélary, R. Pedrono. *Recherche Opérationnelle : Exercices Corrigés.* Hermann, 1982.

[Kar74] Karzanov, A.V. Determining the maximal flow in a network by the method of preflows. *Soviet. Math. Dokl.*, 15:434-437, 1974.

[MKM78] Malhorta, M.V., Kumar, M.P., Mahes Hwari S.N. An algorithm for finding maximum flow in network. *Inf. Proc. Letters*, 7(6):277-278, 1978.

[Ros83] ROSEAUX. *Exercices et Problèmes résolus de Recherche Opérationnelle, tome 1.* Masson, 1983.

[Roy70] B. Roy. *Algèbre Moderne et Théorie des Graphes, tomes 1 & 2.* Dunod, 1970.

Index