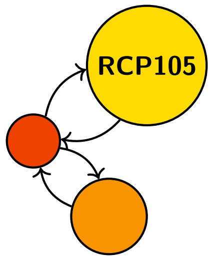
Sommaire



Généralités sur les graphes

1. Préambule

Beaucoup de problèmes se modélisent par des objets et un graphe de relations entre objets, comme par exemple :

- *les chemins plus courts sur un réseau routier,*
- *l' adressage dans un réseau web,*
- *la bande passante d'un réseau informatique .*

Parmi les grands théoriciens qui ont travaillé sur le sujet, il y a Euler, Hamilton, Kirchho, Konig, Edmonds, Berge, Lovasz, Seymour, etc...

Les graphes sont omniprésents en informatique. ils sont utilisés :

- *dans la norme Unified Modeling Language (UML) pour représenter les diagrammes de classes et d'activités,*
- *pour spécifier les protocoles de communications,*
- *pour formaliser la logique séquentielle et les séquences de pilotage.*

2. Généralités sur les graphes

2.1. Définitions

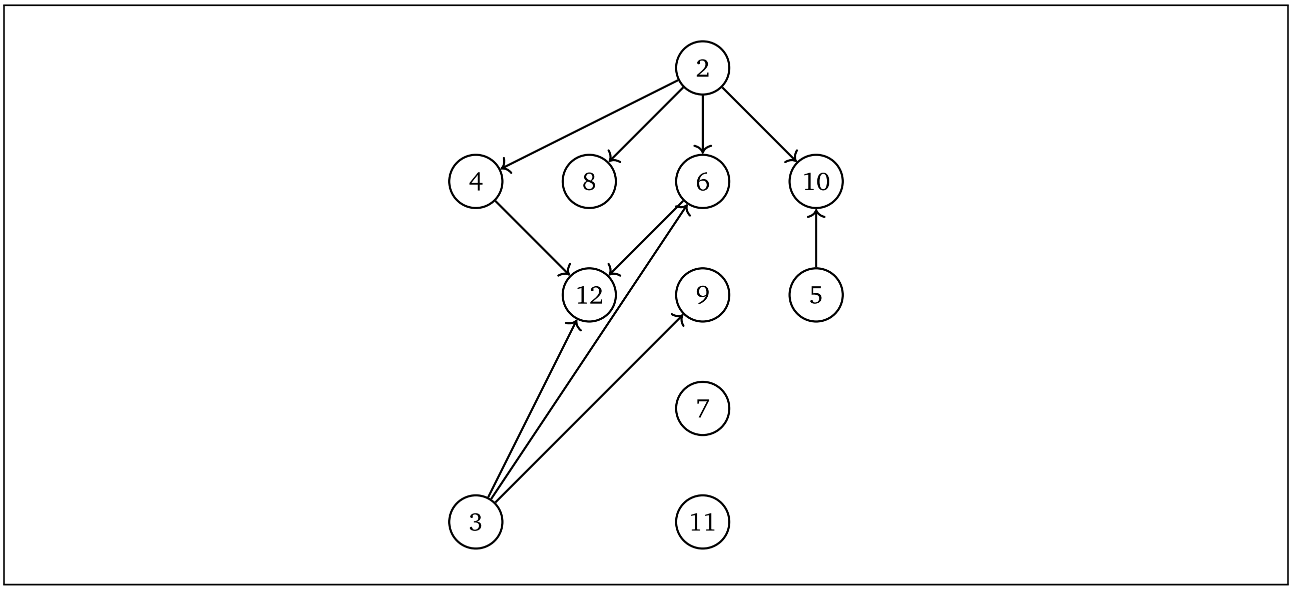


FIGURE 1.1 – Graphe orenté

Définition 1 (Graphe symétrique et orienté).

- Un *graphe* G=(X,A) est donné par un ensemble X de *sommets*, et par un sous-ensemble A du produit cartésien (XxX) appelé l'ensemble des *arcs* de G.
- Un *arc* a = (x,y) a pour origine le sommet x et pour extrémité le sommet y. On note $org(a) = x$ et $ext(a) = y$.

Remarque 1 (Graphes finis).
Dans la suite, on suppose que tous les ***graphes*** considérés sont ***finis***, ainsi X et par conséquent A sont des ensembles finis.

- On dit que le sommet y est un *successeur* de x si $(x, y) \in A$, on dit que x est un *prédécesseur* de y.
- Un *chemin* f du graphe G=(X,A) est une suite d'arcs $a_1, a_2, \ldots a_p$ telle que $\forall i \in [1, p], org(a_i + 1) = ext(a_i)$. L'origine d'un chemin f, aussi notée org(f), est celle de son premier arc a_1 et son extrémité, notée ext(f) est celle de son dernier arc a_p. La *longueur* du chemin est égale au nombre d'arcs qui le composent, c'est-à-dire p.
- On distingue les *graphes orientés* dans lesquels les *arcs* sont parcourus dans un sens déterminé (de x vers y mais pas de y vers x) et les *graphes symétriques* (ou *non-orientés*) dans lesquels les arcs (*arêtes*) peuvent être parcourus dans les deux sens.

Remarque 2 (Graphe symétrique vs orienté).

Pour les ***graphes symétriques*** on parle de *chaîne* au lieu de *chemin*, et d'***arêtes*** au lieu d'***arcs***.

Les algorithmes de parcours pour les ***graphes orientés*** s'appliquent en particulier aux ***graphes symétriques*** : il suffit de construire à partir d'un graphe symétrique G, le graphe orienté G' comportant pour chaque arête (x,y) de G, 2 arcs opposés, l'un de x vers y et l'autre de y vers x.

2.2. Terminologie

- Un chemin f tel que $org(f) = ext(f)$ est appelé un *circuit* pour un graphe ***orienté***. Pour les graphes ***symétriques*** on parle de *cycle*.
- Un *circuit (cycle)* est *élémentaire* s'il ne contient pas 2 fois le même ***sommet***.
- Un *circuit (cycle)* est *simple* si il ne contient pas 2 fois le même ***arc***.
- L'*ordre* d'un graphe est le nombre de ses ***sommets***.
- Une *boucle* est un ***arc*** ou une ***arête*** reliant un sommet à lui-même.
- Un graphe est dit *simple* s'il ne comporte pas de boucle et s'il ne comporte jamais plus d'une arête entre 2 sommets. Un graphe ***symétrique*** qui n'est pas ***simple***, est un *multi-graphe*.
- Un graphe ***orienté*** est dit *élémentaire* s'il ne contient pas de boucle.
- Un graphe ***orienté*** est un *p-graphe*, s'il comporte au plus ***p arcs*** entre deux sommets.
- Un *graphe partiel* d'un graphe orienté (resp symétrique) est le graphe obtenu en supprimant certains ***arcs*** (resp ***arêtes***).
- Un *sous-graphe* d'un graphe orienté (resp symétrique) est le graphe obtenu en supprimant certains ***sommets*** et tous les ***arcs*** (resp ***arêtes***) incidents aux sommets supprimés.
- Un graphe orienté est dit *complet* s'il comporte un ***arc*** (s_i, s_j) et un ***arc*** (s_j, s_i) pour tout couple de ***sommets*** différents s_i, s_j. De même, un graphe ***symétrique*** est dit complet, s'il comporte une ***arête*** (s_i, s_j) pour toute paire de ***sommets*** différents s_i, s_j.

2.3. Adjacence entre sommets

Définition 2 (Adjacence).

- Dans un graphe ***symétrique***, un sommet s_i est dit *adjacent* à un autre sommet s_j, s'il existe une arête entre s_i et s_j. L'ensemble des sommets adjacents à un sommet s_i est défini par $adj(s_i) = \{s_j/(s_i, s_j)\}$.
- Dans un graphe ***orienté***, on distingue les sommets *successeurs* des sommets *prédécesseurs* : $succ(s_i) = \{s_j/(s_i, s_j)\}, pred(s_i) = \{s_j/(s_j, s_i)\}$.
- Dans un ***graphe symétrique***, le *degré d'un sommet* s, noté $d(s)$, est le nombre d'arêtes *incidentes* à ce

sommet. Pour un graphe simple, $d(s) = card(adj(s))$.

- Dans un ***graphe orienté***, le ***demi-degré extérieur*** d'un sommet s, noté $d_+(s)$, est le nombre d'arcs partant de s (dans le cas d'un 1-graphe, on aura $d_+(s) = card(succ(s))$). De même, le ***demi-degré intérieur*** d'un sommet s, noté $d_-(s)$, est le nombre d'arcs arrivant à s. Pour un 1-graphe, $d_-(s) = card(pred(s))$.

Exemple 1.
La figure 1.2 représente le graphe G suivant :

- X = 1,2,3,4,5
- A = (1,2),(1,4),(2,2),(2,3),(2,4),(3,5),(4,3),(5,3) :

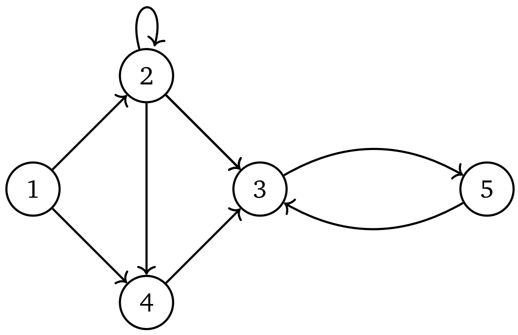


FIGURE 1.2 – Graphe orienté G

- $d_+(2) = 3$, $d_-(4) = 2$
- pred(4) = {1, 2}, succ(2) = {2, 3, 4}

3. Représentation des graphes

Il existe deux façons de représenter un graphe, par une ***matrice d'adjacence*** ou par un ensemble de ***listes d'adjacence***.

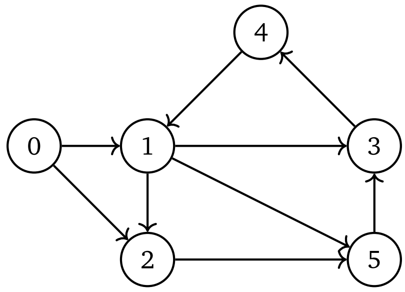
3.1. Représentation par matrice d'adjacence

Définition 3 (Matrice d'adjacence).
Une structure de données simple pour représenter un graphe G est la ***matrice d'adjacence*** M. C'est une matrice carrée de taille ***n*** le nombre de ***sommets*** du graphe. Le coefficient de la matrice $M_{i,j}$ est un entier correspondant au ***nombre d'arcs*** entre les ***sommets i et i***.

Remarque 3 (Successeurs et prédécesseurs).
Lecture de la matrice :

- La ***ligne*** i donne tous les ***successeurs*** du sommet i.
- La ***colonne*** j tous les ***prédécesseurs*** sur sommet j.

Si G est simple, M est une matrice ***booléenne*** carrée $n * n$ dont les coefficients sont V (ou 1) et F (ou 0) telle que $M_{i,j} = V, si\ (x_i, x_j) \in A, M_{i,j} = F, sinon.$



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

FIGURE 1.3 – Matrice d'adjacence - Graphe simple - Sommets par ordre croissant

```
class GrapheMat{
  boolean[][] m; // la matrice M d'adjacence,
  GrapheMat(int n) {
      m=new boolean[n][n];
  }
}
```

FIGURE 1.4 – Algorithme - Classes représentation par matrice d'un graphe

3.2. Complexité

Définition 4 (Complexité).

La ***complexité*** d'un algorithme est la quantité de ***ressources nécessaires*** pour traiter des entrées. C'est une ***fonction*** de n, la taille de l'entrée, ***paramètre*** dimensionnant.

Les principales ressources mesurées sont

- le ***temps*** : nombre d'instructions utilisées.
- l'***espace*** : quantité d'espace mémoire nécessaire.

Notation 1 (Ordre).
Si n est un paramètre ***dimensionnant*** de l'algorithme, on dira que la ***complexité*** (spatiale ou temporelle) est ***en*** ou de l'***ordre*** de $O(f(n))$, avec f fonction. Ce qui signifie, que le nombre d'***instructions*** ou d'unités espace ***mémoire***, peut être encadré à des facteurs près, par $f(n)$, autrement dit, que la ***complexité croit*** comme $f(n)$.

Classes de complexités classiques

Définition 5 (Classes de complexité).
On rencontre typiquement les ***classes*** de complexité suivantes :

- $O(log(n))$: ce sont des algorithmes très rapides, cf recherche dichotomique, exponentiation rapide, etc ...
- $O(n)$: ***linéaire***, typiquement quand on parcourt un tableau ou une liste un nombre borné de fois, cf recherche dans un tableau, minimum d'une liste, etc ...
- $O(n * log(n))$: c'est la complexité des algorithmes de tri-fusion, tri-par-tas, etc ... avec n la taille du tableau.
- $O(n^2)$: ***quadratique***, c'est la complexité des algorithmes de somme de deux matrices, transposée d'une

matrice, tri-insertion, tri-bulles, etc ...

- $O(n^3)$: Produit de matrice par exemple.

Exemple 2.
Taille ***mémoire*** nécessaire à la ***matrice d'adjacence*** d'un graphe ayant n sommets est de l'***ordre*** $O(n^2)$. Pour tester l'existence d'un arc ou d'une arête avec une représentation par ***matrice d'adjacence***, il suffit de lire directement la case correspondante de la matrice.

Pour calculer le ***degré*** d'un sommet, ou lister les ***successeurs*** d'un sommet, il faut parcourir la ligne ou la colonne du sommet ordre $O(n)$.

Le parcours de l'ensemble des arcs/arêtes nécessite la consultation de la totalité de la matrice, et est de l'***ordre*** de $O(n^2)$.

Remarque 4 (Matrice sous-optimale)**.**
Si le nombre d'arcs est très inférieur à n^2, cette représentation n'est pas optimale.

Définition 6 (Complexité NP-complet)**.**
Un problème difficile à résoudre dans un temps raisonnable dans le cas général, mais dont il est possible de vérifier une solution efficacement avec une complexité polynomiale est dit *NP-complet*. *NP* signifie « *Non deterministic Polynomial time* ». Par exemple le problème algorithmique du chemin hamiltonien qui passe par tous les sommets 1 fois.

3.3. Représentation par liste d'adjacence

Définition 7 (liste d'adjacence)**.**
La ***matrice d'adjacence*** peut être très ***creuse***. Une façon plus compacte de représenter un graphe consiste à créer une *liste d'adjacence* en associant à chaque sommet x la ***liste chaînée*** de ses ***successeurs***, notée ***succ***.

Remarque 5.
Pour un graphe G=(X,A), la taille ***mémoire*** est de l'ordre $O(|X|+|A|)$, où |.| désigne le *cardinal*.

La ***liste d'adjacence*** correspond à la ***matrice d'adjacence*** $M = \begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$ est représentée figure 1.5.

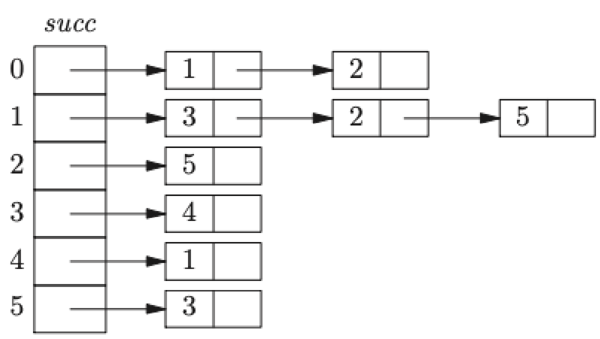


FIGURE 1.5 – Liste d'adjacence

```
// Graphe= Tableau de n listes chainees de successeurs des n sommets,
// correspondant aux n lignes de la matrice d'ajacence.
class Graphe {
```

```
  Liste[ ] succ;
  Graphe(int n) {succ=new Liste[n];}
}

// LIste chainee de successeurs d'une meme origine.
// Une liste correspond a 1 arc.
// Pour l'extremite de la liste, suivant=null.
class Liste {
  int val;  // Numéro du sommet.
  Liste suivant; // Sucesseur de val.
  Liste (int x,Liste ls) {val=x;suivant=ls;} // Initialisation
  Liste () {val=null;suivant=null;} // Default
}

// Traiter tous les successeurs de x du graphe g
Graphe g=new Graphe(n);
for (Liste ls=g.succ[x]; ls!=null; ls=ls.suivant){
  int y = ls.val;
  //Traitement de y
}

// Matrice donne liste
Graphe(GrapheMat g) {
int n=g.m.length;
// succ[i]=[]
succ=new Liste[n];
for (int i=0;i<n;++i){
  for (int j=n-1;j>=0;--j){
          if (g.m[i][j]) succ[i]=new Liste(j, succ[i]);
       }
  }
}

// Liste donne Matrice
GrapheMat(Graphe g) {
   int n=g.succ.length;
   m=new boolean[n][n];
   //m[x][y] = false par defaut
    for (int x=0;x<n;++x) {
     for (int y=0;x<n;++x) {
       m[x][y]=false
    }
  }
  for (int x=0;x<n;++x) {
       for (Liste ls=g.succ[x]; ls !=null; ls=ls.suivant) {
             int y=ls.val;
             m[x][y]=true;
       }
  }
}
```

FIGURE 1.6 – Algorithme - Classes représentation par liste d'un graphe

Mini-exercices.
Si on applique la fonction « *Graphe(GrapheMat g)* » à la matrice M ci-dessus :

1) Représenter la liste « *succ* » quand i=1 et j=5.
2) Modifiez la fonction `GrapheMat()`, pour traiter un graphe qui n'est pas simple en utilisant une matrice d'adjacence d'entiers.

4. Problèmes de graphes

4.1. Nombre de chemins de longueur p entre 2 sommets

Théorème 1 (*Nombre de chemins de longueur p - Graphe orienté*).
Soit M^p la puissance **p-ième** *de la matrice M, le coefficient $M^p_{i,j}$ de la* **ligne i** *et* **colonne j**, *est égal au nombre de* **chemins** *de* **longueur p** *de G dont l'origine est le sommet x_i et dont l'extrémité est le sommet x_j.*

Remarque 6 (Matrice d'entiers).
On considère des coefficients ***entiers*** (pas booléens).

Démonstration. [2] Par récurrence sur p. Pour $p = 1$, le résultat est immédiat car un chemin de longueur 1 est un arc du graphe. Pour $p > 1$, le calcul de M^p donne :

$$M^p = M^{p-1} * M \implies M^p_{i,j} = \sum_{k=1}^{n} M^{p-1}_{i,k} * M_{k,j}$$

Or tout chemin de longueur p entre x_i et x_j se décompose en un chemin de longueur p-1 entre x_i et un certain x_k suivi d'un arc reliant x_k et x_j. Le résultat découle alors de l'hypothèse de récurrence suivant laquelle $M^{p-1}_{i,j}$ est le nombre de chemins de longueur $p-1$ joignant x_i à x_k. □

Remarque 7 (Taille n).
L'algorithme dénombre tous les chemins de taille comprise entre 1 et n. Au delà, c'est une combinaison de chemins de longueur inférieure à n.

Complexité :
Le nombre d'opérations effectuées par l'algorithme est de l'***ordre*** n^4, car le produit de 2 matrices carrées de taille n demande n^3 opérations et l'on peut avoir à effectuer n produits de matrices (chemins longueur n max).

Remarque 8 (Pour 2 sommets).
Si on se limite à la recherche de l'existence d'un chemin entre 2 sommets x et y donnés, pour diminuer la complexité, on peut ne calculer que la ligne x de la matrice $M^p_x = \sum_{k=1}^{n} M^{p-1}_{x,k} * M_{k,y}$ ($O(n^3)$).

```
// Algorithme pour tester l'existence d'un chemin entre x et y
boolean existeChemin(GrapheMat g, int x, int y) {
  int n = g.m.length;
  boolean r[ ][ ]=new boolean[n][n];
  copie(r, g.m);
  for (int p=1;!r[x][y]&&(p<n);++p) {
    multiplier(r, r, g.m); // r=r*m
    additionner(r, r, g.m); // r=m+r*m
  }
return r[x][y];
}
```

Remarque 9.

Les fonctions « *multiplier(r,a,b)* » et « *additionner(r,a,b)* » du code ci-dessus sont respectivement des fonctions qui multiplient et ajoutent les 2 matrices booléennes a et b (de dimension $n * n$) en rangeant le résultat dans la matrice r.

La fonction copie(r,m) fait une copie de m dans la matrice r.

Mini-exercices.
On veut appliquer la fonction `existeChemin(M,3,3)` à la matrice ci-dessus. les sommets sont numérotés de 0 à 5.

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

1) Représentez le graphe correspondant à la matrice.
2) Donnez la formule littérale de r en fonction de m après p itérations ligne 6. Interprétez cette formule.
3) Calculez r quand p=1. Pourquoi s'arrêter à p=n ?
4) Quelle est la valeur de p en sortie de boucle de `existeChemin(M,3,3)` ? Que vaut r ? Que pouvez-vous en conclure concernant le nombre de chemins de taille inférieure à p du graphe ?
5) Connaissant la complexité du produit de 2 matrices de taille n, en déduire la complexité de la fonction `existeChemin()`.

4.2. Existence de chemins entre 2 sommets

Problème : compilation conditionnelle dans un compilateur ?

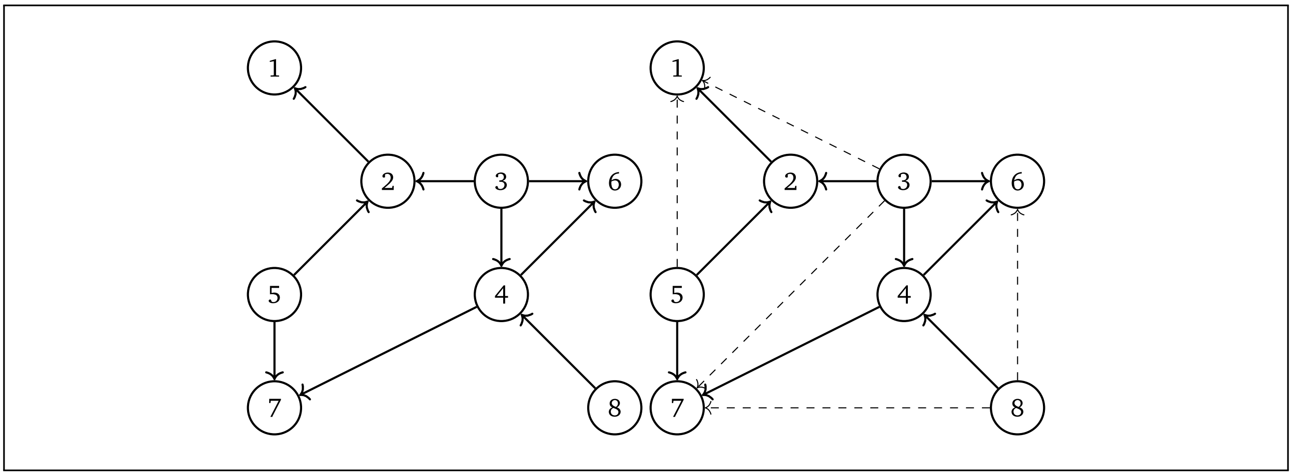


FIGURE 1.7 – Un graphe et sa fermeture transitive

4.3. Fermeture transitive - Algorithme de Roy (F) et Warshall (US) [2]

Définition 8 (Fermeture transitive).
La *fermeture transitive* d'un graphe $G = (X,A)$ est un ***graphe*** $G^* = (X,A^*)$, tel que $(x,y) \in A^*$, si et seulement si il existe un chemin f dans G d'origine x et d'extrémité y.

Remarque 10 (G simple).
G est un graphe ***simple*** et sa matrice d'adjacence est bouléenne. De même pour G^* la ***fermeture transitive***.

Remarque 11 (Existence d'un chemin).
On effectue un pré-traitement de G en calculant $G^* = (X,A^*)$, puis on répond en temps constant, O(1), à toute question sur l'existence de chemins entre x et y.

Définition 9 ($\phi_x(A)$).
Le calcul de (X,A^*) s'effectue par itération en partant de A, de l'opération de base $\phi_x(A)$, qui ajoute à A les arcs (y, z) tels que y est un prédécesseur de x et z un de ses successeurs. Soit

$$\phi_x(A) = A \cup (y,z) | (y,x) \in A\, et\, (x,z) \in A$$

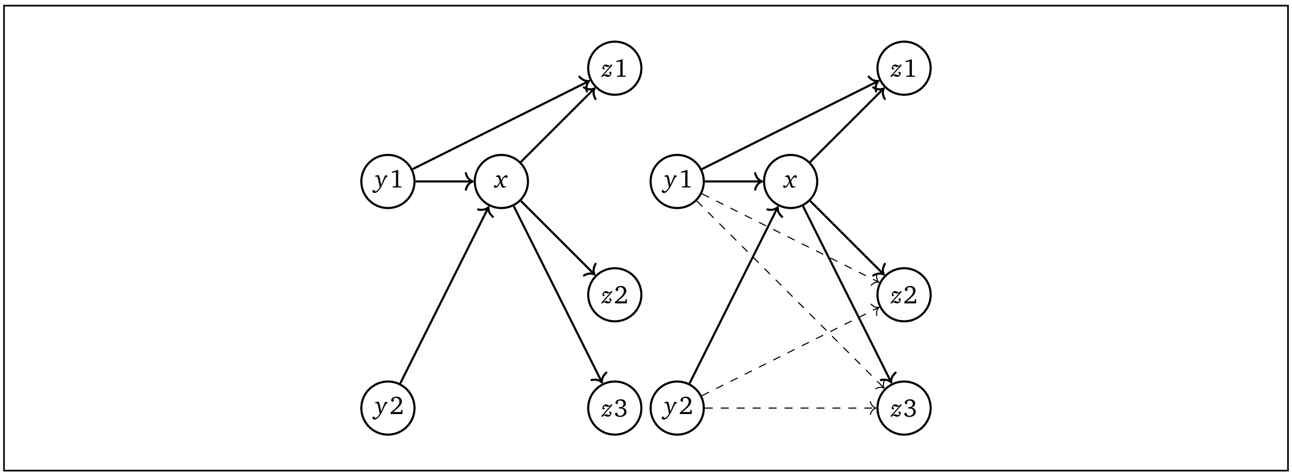


FIGURE 1.8 – L'effet de l'opération ϕ_x : les arcs ajoutés sont en pointillé

Proposition 1.
[2] *Pour tout sommet x, on a* $\phi_x(\phi_x(A)) = \phi_x(A)$ *et pour tout couple de sommets (x,y) :* $\phi_x(\phi_y(A)) = \phi_y(\phi_x(A))$.

Algorithme 1 (Roy et Warshall).
Algorithme de Roy (F) et Warshall (US) : La fermeture transitive A' est donnée par : $A' = \phi_{x_1}(\phi_{x_2}(...\phi_{x_n}(A)))$, x_i désignant les sommets de X.

Démonstration. La fermeture transitive A' contient les itérations $\phi_{x_1}(\phi_{x_2}(...\phi_{x_n}(A)))$. Inversement, si il existe un chemin de x à y de G s'écrivant $(x, y_1)(y_1, y_2)...(y_p, y)$, alors $\phi_{y_1}(\phi_{y_1}(...\phi_{y_p}(A)))$ contient effectivement un arc (x,y). □

```
void phi(GrapheMat g, int x) {
  int n=g.m.length;
  for (int i=0;i<n;++i) {
    for (int j=0; j<n;++j) {
      // i predecesseur de x
      // j successeur de x
      g.m[i][j]=g.m[i][j]||(g.m[i][x]&&g.m[x][j]);
    }
  }
}

// Calcul de G*
void fermetureTransitive(GrapheMat g) {
      GrapheMat r=copieGraphe(g);
      for (int k=0;k<r.m.length;++k) phi(r,k);
      return r;
}

GrapheMat copieGraphe(GrapheMat g) {
   int n = g.m.length;
   GrapheMat r=new GrapheMat (n);
   for (int i=0;i<n;++i)
     for (int j=0; j<n;++j) r.m[i][j]=g.m[i][j];
  return r;
}
```

FIGURE 1.9 – Algorithme Roy et Warshall - Fermeture transitive - Graphe orienté

Complexité :
L'algorithme effectue un nombre d'opérations que l'on peut majorer par n^3, chaque exécution de la fonction Φ pouvant

nécessiter n^2 opérations. Cet algorithme est donc meilleur que le calcul des puissances successives de la matrice d'adjacence.

Remarque 12 (En pratique)**.**
En pratique, il suffit de faire pour toute ligne i de 1 à n, un OU logique entre la ligne i et les lignes des prédécesseurs de i.

Mini-exercices.
On veut appliquer la fonction `phi(M,1)` ci-dessus à la matrice (les sommets sont numérotés de 0 à 5) : $M = \begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$

1) Que vaut g.m quand i=4 et j=5 ligne 7 ?
2) Calculez la complexité de la fonction phi().
3) En déduire la complexité de l'algorithme de la fermeture transitive.

Index

Auteurs du chapitre
Rédaction : Hervé Bailly

C'est une synthèse qui emprunte largement aux cours suivants :

- « *Initiation à la théorie des graphes* » de Thierry Brugère.[1]
- « *Informatique Fondamentale* » de Jean-Jacques Lévy. [2]

Design : « *Exo7* »

Version 3.4 – Mars 2025