Théorie des graphes

L. Sais

Algorithmique & Programmation 5

21 mars 2011

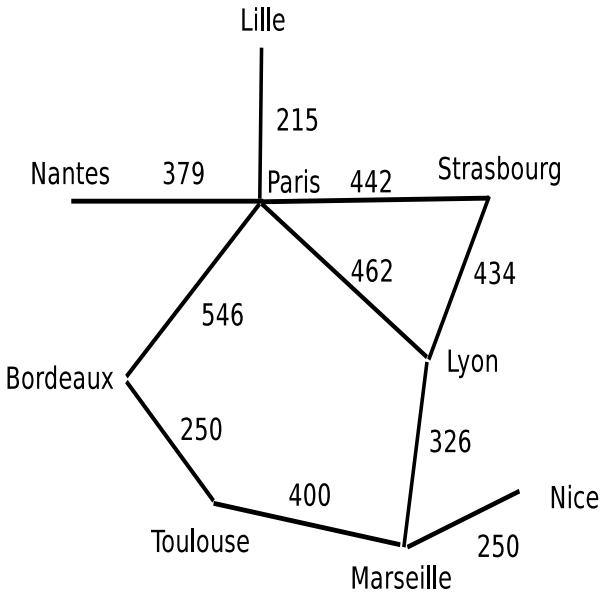
Graphes et algorithme : présentation

Introduction

De nombreux problèmes courants, en informatique, en ingénierie, en sciences sociales, en intelligence artificielle, peuvent se représenter en termes de relations (binaires) entre objets.

On peut citer par exemple :

- réseaux de communications : routiers, ferroviaires, machines en réseaux, circuits intégrés, ...

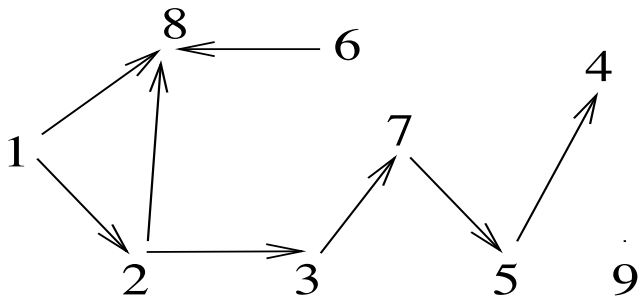


On distingue les sommets (villes, carrefours, soudures) et les arcs ou arêtes (communication entre sommets) qui mettent en relation les sommets. On associe parfois aux arcs des caractéristiques, qui permettent d'exprimer des distances, des coûts, des flux, ...

Le type des problèmes que l'on peut se poser alors concerne par exemple la recherche d'itinéraires optimaux (problème classique du voyageur de commerce : visiter toutes les villes avec un cheminement optimal).

- relations sociales : familiales, hirarchiques, d'incompatibilité, ...
  Ici, les sommets correspondent aux individus, les arcs aux relations entre individus.

- ordonnancement de tâches : atelier, systmes d'informations, ...
  Une première possibilité consiste représenter les instants de début et de fin d'activité par les sommets, et les activités par les arcs ; on peut également représenter les tâches par les sommets et les relations de précédance par les arcs.



1 : caleçon	4 : veste	7 : chemise
2 : pantalon	5 : cravate	8 : chaussures
3 : ceinture	6 : chaussettes	9 : montre

- représentation d'algorithmes (organigrammes) :
  Les sommets correspondent aux instructions ou tests, les arcs correspondent aux enchanements possibles entre instructions ou tests.
  On retrouvera également :
  * les emplois du temps
  * les automates à états finis
  * les problèmes de coloration de cartes (2 pays voisins doivent avoir 2 couleurs différentes)
  * etc.

Remarque

Le formalisme des graphes permet de décrire de nombreux problèmes souvent de manière simple, mais qui peuvent s'avérer difficile à résoudre.

Dans le cadre de ce cours, on traitera des problèmes classiques où de bons algorithmes sont connus (il existe des problèmes où l'on ne connait pas de bons algorithmes) :

- étant donné un graphe, vérifier s'il possède certaines propriétés
- étant donné un graphe, déterminer une partie possédant certaines propriétés.
- construire un, plusieurs, ou tous les graphes possédant certaines propriétés.

Graphes et algorithme : présentation

Définitions et terminologie

Définition

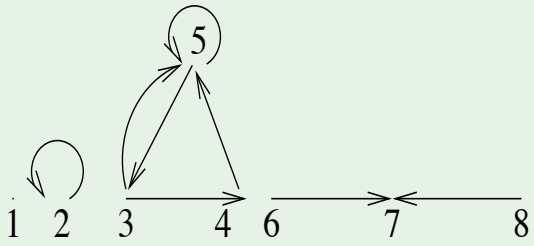
Un graphe orienté est un couple G=(S,A) où :
- S est un ensemble fini d'éléments appelés sommets
- A est un ensemble fini de couples de sommets appelés arcs

on a $A \subset S \times S$

Remarque

S'il existe plusieurs arcs entre deux sommets, on parle d'un multigraphe (arcs multiples)

Exemple



S={1..8}
A={(2,2),(3,5),(4,5),(3,4),(5,5),(5,3),(6,7),(8,7)}

Définitions

Etant donné un graphe G=(S,A) et $x \in S$

- si $(x, y) \in A$, alors y est un successeur de x
- si $(y, x) \in A$, alors y est un prédécesseur de x
- x et y sont adjacents si y est un prédécesseur et/ou un successeur de x
- l'ensemble des successeurs de x est noté $\Gamma^+(x)$
  $\Gamma^+(x) = \{y | (x, y) \in A\}$
- l'ensemble des prédécesseurs de x est noté $\Gamma^+(x)$
  $\Gamma^-(x) = \{y | (y, x) \in A\}$
- l'ensemble des voisins de x est noté $\Gamma(x)$
  $\Gamma(x) = \Gamma^+(x) \cup \Gamma^-(x)$

- le degré intérieur de x est noté $d^-(x)$
  $d^-(x) = |\Gamma^-(x)|$
- le degré extérieur de x est noté $d^+(x)$
  $d^+(x) = |\Gamma^+(x)|$
- le degré de x est noté $d(x)$
  $d(x) = |\Gamma(x)|$

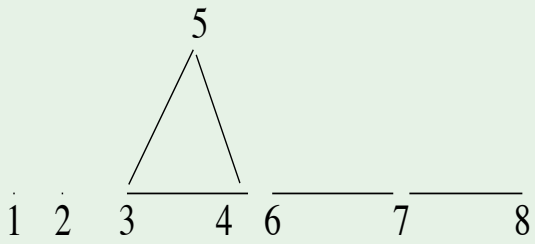
Définition

Un graphe non orienté est un couple G=(S,A) où :
- S est un ensemble fini d'éléments appelés sommets
- A est un ensemble fini de paires de sommets appelés arêtes

Une arête est notée $\{x,y\}$

Dans le cas de ces graphes non-orientés, on parle simplement de sommets voisins et de degré.

Exemple



S={1..8}
A={{3,5},{4,5},{3,4},{6,7},{8,7}}

Définition (graphe complet)

- Un graphe orienté G=(S,A) est dit complet si $A=S^2$
- Un graphe non orienté G=(S,A) est dit complet si toute paire de sommets figure dans A

Définition

Etant donné un graphe orienté G=(S,A) :

- le graphe réciproque de G est le graphe $G^{-1}=(S,A^{-1})$ où $A^{-1}=\{(x,y) \in S \times S \mid (y,x) \in A\}$
- le graphe symétrisé associé à G est le graphe $G_S=(S,A \cup A^{-1})$

Définition

Un graphe valué est un graphe orienté ou non orenté G=(S,A) auquel on associe une fonction de coût c : A->R , un tel graphe est noté (S,A,c)

Exemple

Voir l'exemple sur le réseau routier (villes)

Notations

Si G=(S,A) On a |S|=card(S)=n et |A|=card(A)=m

Rapports etre n et m : (cas des graphes orientés)

- Si G=(S,A) n'est pas un multigraphe, on a $A \in S^2$
- graphes tels que $m=n^2$: graphes complets avec boucles
- graphes complets sans boucle : $m=n(n-1)$

Relations entre d^+, d^-, d et m :

- pour le cas des graphes orientés sans boucles :

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

- pour le cas des graphes non orientés :

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

Définition

Etant donné un graphe orienté G=(S,A) et un sous-ensemble de sommets X, le sous-graphe de G induit (engendré) par X est G(X)=(X,E)
où E=$\{(x, y) \in A | x, y \in X\}$

La définition est identique dans le cas non orienté

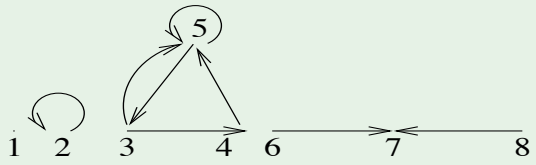
Définition

Etant donné un graphe G=(S,A) et un sous ensemble d'arcs ou d'arêtes E, le graphe partiel de G induit par E est le graphe G=(S,E)

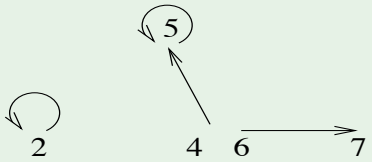
Exemple

Un graphe, le sous-graphe induit par X ={2,4,5,6,7}
et le graphe partiel induit par l'ensemble d'arcs
E={(2,2),(5,3),(4,5),(3,4),(8,7)}

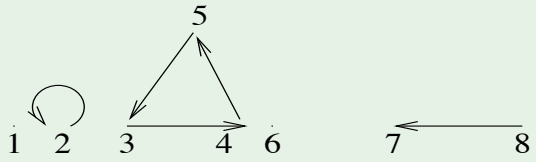
graphe



sous-graphe



graphe partiel



Définition

Etant donné un graphe G=(S,A), une partition p=$\{S_1,...,S_p\}$ de S,
le graphe quotient induit par P est la graphe (V,E) où :

- V=$\{s_1,...,s_p\}$ (à chaque partie S, on associe un sommet S_i)
- E=$\{(s_i,s_j)$ / $\exists$ (x,y)$\in$A , x$\in S_i$ et y$\in S_j\}$

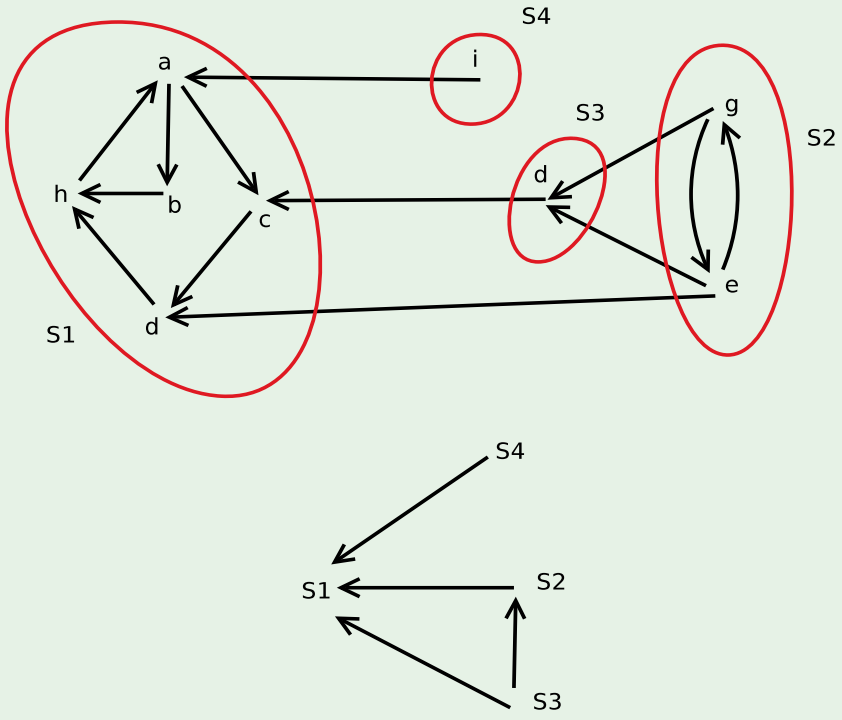
Rappel

p=$\{S_1,...,S_p\}$ est une partition de S ssi

$$\bigcup_{1\leq i\leq p} S_i = S \quad \text{et} \quad \forall i \neq j,\ S_i \bigcap S_j = \emptyset$$

Exemple

Un graphe et le graphe quotient induit par la partition p=$\{s_1,...,s_4\}$



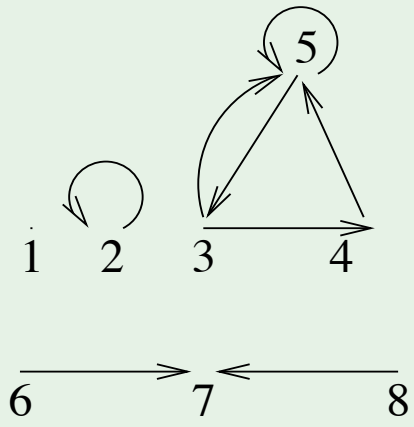
Graphes et algorithme : présentation

Représentation

Matrice d'adjacence

Le graphe est représenté par un tableau à deux dimensions indexé sur l'ensemble des sommets

Graphe et matrice d'adjacence



X	1	2	3	4	5	6	7	8
1	F	F	F	F	F	F	F	F
2	F	V	F	F	F	F	F	F
3	F	F	F	V	V	F	F	F
4	F	F	F	F	V	F	F	F
5	F	F	V	F	V	F	F	F
6	F	F	F	F	F	F	V	F
7	F	F	F	F	F	F	F	F
8	F	F	F	F	F	F	V	F

Les éléments du tableau peuvent être différents types selon le graphe considéré

- booléens : un élément [i,j] est vrai si l'arc (i,j) figure dans le graphe. Dans le cas non orienté si l'élément [i,j] est vrai, alors l'élément [j,i] est vrai
- réels : Dans le cas de graphes valués, on représente la fonction de coût, ainsi pour valeur c(i,j) et si l'arc (i,j) ne figure pas dans le graphe [i,j] prend une valeur conventionnelle.
- entiers : Dans le cas du multigraphe, [i,j] aura pôur valeur le nombre d'arcs (i,j) figurant dans le graphe.

en C :

```
define N_MAX

typedef struct{
  int g[N_MAX][N_MAX];
  int n;
}grapheM;
```

en pseudo-langage :

```
constante N_MAX = ... (* nombre maxi de sommets *)

type sommet = 1..N_MAX
     graphe = structure
                a:= tableau[sommet,sommet] de booleens
                n: 0..N_MAX
              fin
```

Avantages

- Accès direct aux arcs : tester l'existence d'un arc, ajouter, ou supprimer un arc sont des opérations réalisables en temps constant ($\theta(1)$)
- L'écriture des algos est généralement simplifiée :
  l'accès aux successeurs d'un sommet est aisé

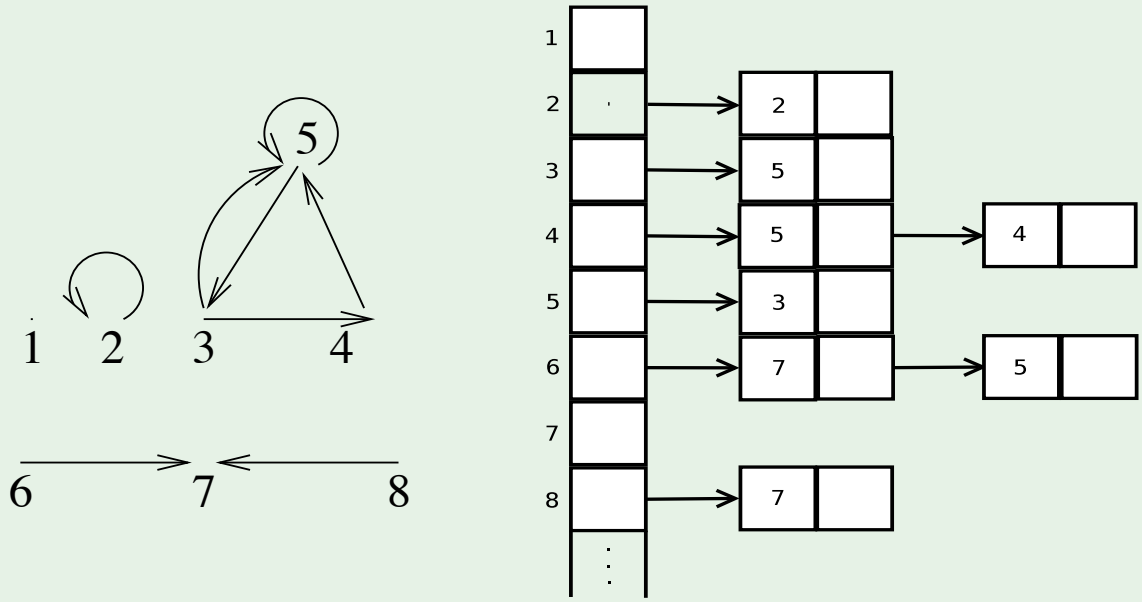
Inconvénients

- encombrement maximal de la mémoire :
  $\forall$ le nombre d'arcs figurant dans le graphe, la place mémoire nécessaire est en $\theta(\text{N\_MAX}^2)$; cas génant : $m \leq n^2$
- coût de l'initialisation : $\theta(n^2)$
- coût de la recherche des successeurs d'un sommet : $\theta(n)$, même s'il y'a pas de successeurs.
  l'accès aux successeurs d'un sommet est aisé

Listes d'adjacence

C'est généralement la représentation la plus utilisée, celle où l'on dispose des algos les plus performants.

Exemple



en C :

```
typedef struct cel{
  int st;
  struct cel* suiv;
}cellule;

typedef cellule* Liste;

typedef struct{
  Liste a[N_MAX];
  int n;
}grapheL;
```

en pseudo-langage :

```
type Liste = ^cellule
     cellule = structure
               st : sommet
               suiv : Liste
               fin
     grapheL = structure
               a := tableau [sommet,sommet] de Liste
               n :0..N_MAX
               fin
```

Avantages

- encombrement minimal de la mémoire : θ(N_MAX+m)
- l'accès au successeur d'un sommet x est en $\theta(d^+(x))$

Inconvénients

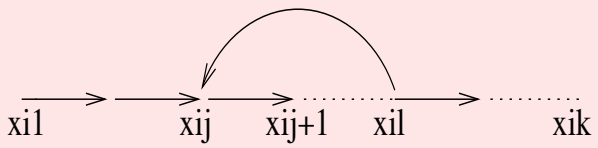
- cout d'accès à un arc l'origine x : $\theta(d^+(x))$
- l'accès aux prédécesseurs d'un sommet x impose le parcours de toutes les listes d'adjacence : θ(n+m)

Définition (Graphe orienté)

- un chemin de G=(S,A) est une séquence de sommets x_{i_1}, x_{i_2},...,x_{i_k}
  $k \geq 1$, $\forall j\ 1 \leq j \leq k\text{-}1$, $(x_{i_j},x_{i_{j+1}}) \in A$
  un chemin $x_{i_1},x_{i_2},...,x_{i_k}$ sera noté $[x_{i_1},x_{i_2},...,x_{i_k}]$
- un chemin $[x_{i_1},x_{i_2},...,x_{i_k}]$ est dit élémentaire si
  $\forall$ u,v , $1 \leq u \leq v \leq k$, $x_{i_u} \neq x_{i_v}$
  (ne contient pas deux fois le même sommet)
- la longueur d'un chemin est égale au nombre de sommets moins un.
  le chemin constitué du seul sommet x_{i_1} noté $[x_{i_1}]$ est de longueur nulle.

Propriété

si $c=[x_{i_1},...,x_{i_k}]$ est un chemin de G, alors il existe une sous-suite de c qui est un chemin élémentaire de x_{i_1} à x_{i_k}.



Preuve :

Par récurrence sur r, le nombre de répétitions de sommets dans c.

- si r=0, c'est un chemin élémentaire
- Supposons que la propriété est vraie au rang r-1
  Soit un chemin c=[x_{i_1},...,x_{i_k}] possédant r répétitions.
  Il existe u et v $1 \leq u \leq v \leq k$ tels que x_{i_u} à x_{i_v}.

  Soit c' la sous-séquence de c obtenue en supprimant de c la sous-suite x_{i_u+1} ... x_{i_v}.

  Donc c'=[x_{i_1},...,x_{i_u},x_{i_v+1},...,x_{i_k}] est un chemin ayant au plus c-1 répétitions, d'où par hypothèse de récurrence c' qui est une sous-séquence de c posède une sous-séquence qui est un chemin élémentaire de x_{i_1} à x_{i_k} :

  $$c=[x_{i_1},...,(x_{i_u},...,x_{i_v}),x_{i_v+1},...,x_{i_k}]$$

Définitions (Graphe non orienté)

- une chaîne de G=(S,A) est une séquence de sommets $x_{i_1},\ldots,x_{i_k}$ $k \geq 1$ avec $\forall$ j $1 \leq u \leq v \leq k$

$$(x_{i_j},x_{i_{j+1}}) \in A \text{ ou } (x_{i_{j+1}},x_{i_j}) \in A$$

- une chaîne $(x_{i_1},\ldots,x_{i_k})$ est dite élémentaire si

$$\forall\ u,v \quad 1 \leq u \leq v \leq k \quad x_{i_u} \neq x_{i_v}$$

(ne contient pas deux fois le meme sommet)

- la longueur d'une chaine est égale au nombre de sommets moins un. La chaine constituée du seul sommet x_{i_1} sera notée (x_{i_1}) est de longueur nulle.

Définition

- un circuit de G=(S,A) est un chemin $[x_{i_1},\ldots,x_{i_k}]$ de G tel que $\forall k \geq 2$ et $x_{i_1}=x_{i_k}$.
- un cycle de G=(S,A) est une chaine $(x_{i_1},\ldots,x_{i_k})$ de G tel que $\forall k \geq 2$ et $x_{i_1}=x_{i_k}$.

Définition

Etant donné un graphe G=(S,A) et x∈S, on définit les ensembles de descendants et ascendants d'un sommet x :

- $desc_G(x)=\{y \in S \mid \exists$ [x,y] dans G$\}$
- $asc_G(x)=\{y \in S \mid \exists$ [y,x] dans G$\}$

=> $\forall$ x,y∈S, $y \in desc_G(x) <=> x \in asc_G(y)$

Propriété

Soit G=(S,A) un graphe

$$\forall x \in S,\ asc_G(x)=desc_{G^{-1}}(x)$$

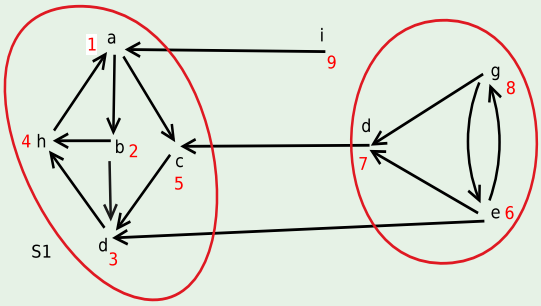
Définition

Une racine de G=(S,A) est un sommet r de S tel que $S=desc_G(r)$

Parcours en profondeur :

Exemple

Parcours en profondeur d'un graphe : les numéros correspondants aux sommets donnant l'ordre dans lequel les sommets sont visités.



algorithme :

```
procedure parcoursProfondeur(g : graphe);
    variable atteint : tableau[sommet] de booléens
    (atteint[x] <=> le sommet x a été atteint)
    x : sommet

debut
   pour tout sommet x de g faire
      atteint[x]:=faux
   pour tout sommet x de g faire
      si non atteint[x] alors RechercheProf(x)
fin
```

```
procedure RechercheProf(x : sommet);
{spécifications : étant donné un sommet x non atteint, cette
procédure marque x, et tous les sommets y descendants de x
tels qu'il existe un chemin [x,y] dont aucun sommet n'est
marqué}

   variable y : sommet

debut
   atteint[x]:=vrai
   pour tout successeur y de x faire
      si non atteint[y] alors RechercheProf(y)
fin
```

Complexité :

- La phase d'initialisation du tableau atteint est en $\theta(n)$.
- L'itération de l'algorithme principal est réalisé exactement en n étapes, pour chacune il y'a au moins un test réalisé.
- $\theta(n+m)$, soit $\theta(\max(n,m))$ dans le cas des listes d'adjacences.
- $\theta(n^2)$ dans le cas des matrices d'adjacence.

Ordre de traitement-Numérotation : (parcours en profondeur)

L'objet des parcours de graphes concerne des traitements que l'on souhaite opérer sur les graphes. Les traitements s'opèrent parfois sur les sommets visités. Il est alors possible d'opérer un traitement avant ou après la visite du sommet. Il y'a donc soit un traitement en préOrdre ou ordre préfixé, soit en postOrdre ou ordre postfixé. Ceci se traduit par une modification de la procédure de recherche en profondeur.

traitement en préOrdre

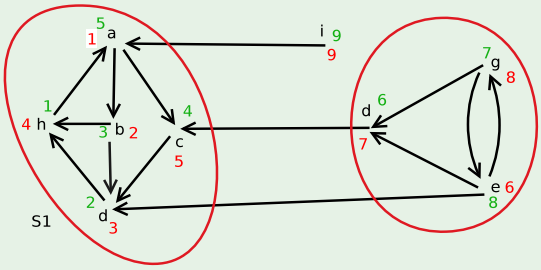
```
var y : sommet
debut
   atteint[x]:=vrai
   <traiter x>
   pour tout successeur y de x faire
      si non atteint[y] alors rechercheProf(y)
fin
```

traitement en postOrdre

```
var y : sommet
debut
   atteint[x]:=vrai
   pour tout successeur y de x faire
      si non atteint[y] alors rechercheProf(y)
   <traiter x>
fin
```

Exemple

numérotation en pré-ordre(rouge) et en post-ordre(vert) pour un parcours en profondeur



Mise en oeuvre en C du parcours en profondeur :

```
typedef struct cel{
  int st;
  struct cel* suiv;
}cellule

typedef cellule* Liste;

typedef struct{
  Liste a[N_MAX];
  int n;
}grapheL;

typedef int atteint[N_MAX];

/* variables globales déclarées:
   grapheL g;
   atteint m: */
```

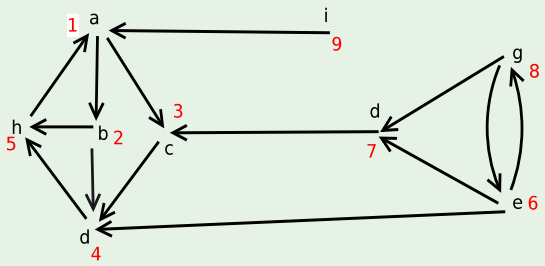
```
void parcoursProf(int x){
  Liste p;
  m[x]=1;
  p=g.a[x];
  while(p!=NULL){
    if(!m[p->st]) parcoursProf(p->st);
    p=p->suiv;
  }
}

void main(){
  int x;
  for(x=1;x<g.n;x++) m[x]=0;
  for(x=1;x<g.n;x++){
    if(!m[x]) parcoursProf(x);
  }
```

Parcours en largeur :

Dans ce parcours, lorsqu'un sommet x est atteint, tous ses successeurs y sont visités avant de visiter les autres descendants de x. Si l'on se réfère à la notion de distance, partant de x, le parcours va d'abord visiter tous les sommets situés à la distance 1 de x, puis tous les sommets situés à la distance 2 de x, et ainsi de suite jusqu'à ce que tous les sommets atteignables soient visités.

Exemple (parcours en largeur)

La numérotation indiquée correspond à un ordre de visite lors d'un parcours en largeur



principe de l'algo :

Il repose sur la notion de file.
Lors de la visite d'un sommet s, tous ses successeurs non encore atteint vont être rangés dans la file de manière à conserver la priorité liée aux distances depuis le sommet origine.

Mettons en oeuvre cet algo ! ! !

```
typedef struct{
  Liste tete,queue;
}File;

void enfiler(int x,File* f);
int defiler(File* f);
int fileVide(File* f);
```

```
void parcoursLarg(int x){
  Liste p;

  initFileVide(&f);
  enfiler(x,&f);
  m[x]=vrai;
  while(!fileVide(f)){
    x=defiler(f);
    p=g.a[x];
    while(p!=NULL){
if(!m[p->st]){
 m[p->st]=vrai;
enfiler(p->st,f);
}
      p=p->suiv;
    }
  }
}
```

```
void main(){
  int x;
  for(x=1;x<g.n;x++) m[x]=0;
  for(x=1;x<g.n;x++){
    if(!m[x]) parcoursLarg(x);
  }
}
```

complexité de l'algo :

- identique à celle du parcours en profondeur
- $\theta(n+m)$ pour les listes d'adjacence
- $\theta(n^2)$ pour les matrices d'adjacence

Quelques applications des parcours

- Obtention des descendants par un parcours en profondeur. Il suffit d'insérer dans un ensemble, les sommets dès qu'ils sont atteints. Outre le tableau atteint, on utilise donc une structure de données d'ensemble.

```
Type desc : ensemble de sommets
var atteint : tableau[sommet] de booléens

procedure descProf(x : sommet,desc : ens de sommets);
   var y : sommet;
   debut
      atteint[x]:=vrai
      desc:=desc U {X}
      pour tout successeur y de x faire
         si non atteint[y] faire descProf(y,desc)
   fin
```

complexité de l'algo :
identique à celle du parcours en profondeur du sous-graphe induit par les descendants de x, soit avec n_x et m_x le nombre de sommets et d'arcs pour ce sous-graphe.
- $\theta(n_x+m_x) -> $ listes d'adjacence pire cas : O(n+m)
- $\theta(n_x.m) -> $ matrice d'adjacence pire cas : $O(n^2)$

- Obtention des ascendants pour un parcours en profondeur.
  La recherche des ascendants d'un sommet peut se faire en utilisant la procédure descProf avec pour paramètre le sommet considéré et le graphe réciproque de G, soit G^{-1}

Application globale : Fermeture reflexo-transitive d'un graphe

Définition

Soit G=(S,A) un graphe, la fermeture réflexo-transitive de G est notée $G^t=(S,A^t)$ et elle vérifie la propriété suivante :

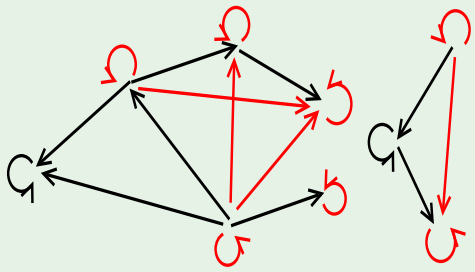
$$(x,y)\in A <=> x=y \text{ ou } \exists [x,y] \text{ dans } G$$

On peut vérifier que l'on a bien les propriétés suivantes :

- $(x,y)\in A => (x,y)\in A^t$
- $(x,y)\in A$ et $(y,z)\in A => (x,z)\in A^t$
- $\forall x\in S\ (x,x)\in A => (x,y)\in A^t$

Exemple

les arcs ajoutés par réfléxivité et transitivité



Calcul de G^t à partir de G :

principe :

Il s'agit d'ajouter les arcs (x,y) pour tout y tel que [x,y] est dans G ie pour tout élément y descendants de x. Il en découle que dans G^t les successeurs de x sont les descendants de x dans G.
On utilise ainsi la procédure de calcul des descendants.

```
procedure fermetureRefTrans(g : graphe,gt : graphe)
var atteint : tableau[sommet] de booléens
   desc     : ens de sommets
   x,y      :  sommet

debut
   {initialisations}
   pour tout sommet x de G faire atteint[x]:=faux
   desc:=ens_vide
   pour tout sommet x de G faire
      descProf(x,g,desc)
      pour tout sommet y de desc faire
         ajouterArc(x,y,gt)
         desc:=desc-{y}
         atteint[y]:=faux
      fin pour
   finpour
fin
```

complexité :

Nous donnons ici seulement une majoration de la complexité.
On étudie d'abord le cas d'une représentation matricielle :
on a vu que le cout de la procédure descProf(x,g,desc) est en $\theta(n_x.m)$. On réalise ce traitement n fois.
On obtient ainsi une majoration par $O(n^3)$

Dans le cas des listes d'adjacence, on obtient une majoration par $O(n(n+m))$