L3-INFO – INF6B
Structures discrètes
2020-2021

Semaine #3 – Parcours, Composantes connexes et Tri topologique

---

1. Introduction
2. Parcours en profondeur *(DFS : Depth First Search)*
3. Parcours en largeur *(BFS : Breath First Search)*
4. Application des parcours
   (1) Composantes connexes d'un graphe non-orienté
   (2) Tri topologique des sommets d'un graphe orienté sans circuit
   (3) Ordonnancement : Méthode Potentiels/Tâches

Introduction

– Le *parcours en largeur* consiste à explorer les sommets du graphe niveau par niveau, à partir d'un sommet donné.

– Le *parcours en profondeur* consiste, à partir d'un sommet donné, à suivre un chemin le plus loin possible (jusqu'à un cul-de-sac ou un cycle), puis à faire des retours en arrière pour reprendre tous les chemins ignorés précédemment.

Marquage des sommets

– Initialement, tous les sommets sont ***blancs*** (un sommet blanc n'a pas encore été découvert).

– Lorsqu'un sommet est "découvert" (i.e. quand on arrive pour la première fois sur ce sommet), il devient ***gris***.

Un sommet demeure gris tant qu'il reste des successeurs de ce sommet qui sont blancs (i.e. qui n'ont pas encore été découverts).

– Un sommet devient ***noir*** lorsque tous ses successeurs sont gris ou noirs (i.e. lorsqu'ils ont tous été découverts).

Arbre/Forêt Recouvrant.e

On parcourt un graphe à partir d'un sommet source s.

Ce parcours va permettre de découvrir tous les sommets accessibles depuis s, i.e. tous les sommets t pour lesquels il existe un chemin depuis s.

Au passage, on construit l'arborescence des sommets accessibles depuis s, appelée ***ARBRE RECOUVRANT*** de racine s.

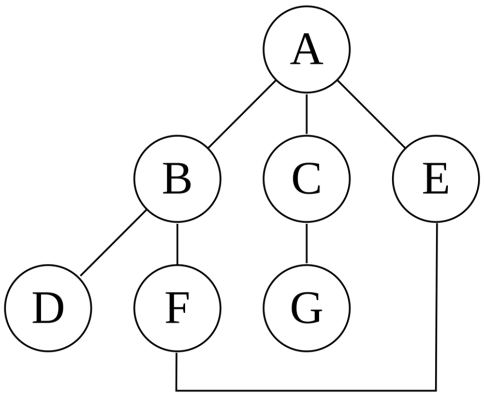
- Chaque sommet a au plus un prédécesseur à partir duquel il a été découvert.

- La racine de cette arborescence est s, le sommet à partir duquel on a commencé le parcours.

- L'arbre recouvrant associé à un parcours de graphe sera mémorisé dans un tableau PRED[t].

Enfin, si le graphe n'est pas connexe, alors il faudra parcourir chaque partie (composante) à partir d'un sommet de celle-ci, et ainsi potentiellement construire plusieurs arbres recouvrants. On parle alors de ***FORET RECOUVRANTE***.

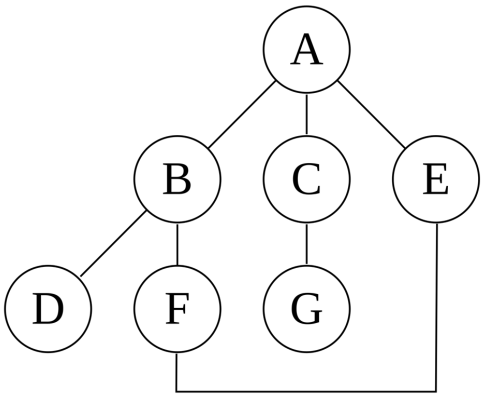
Parcours en profondeur à partir du sommet source A



Ordre de visite des sommets : A, B, D, F, E, C, G

Quel est l'arbre recouvrant associé à ce parcours ?

Parcours en largeur à partir du sommet source A

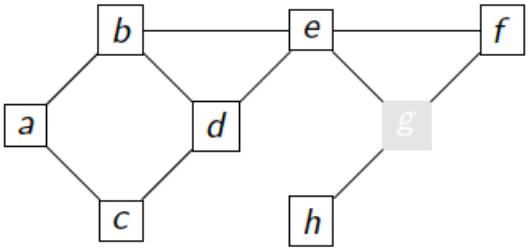


Ordre de visite des sommets : A, B, C, E, D, F, G

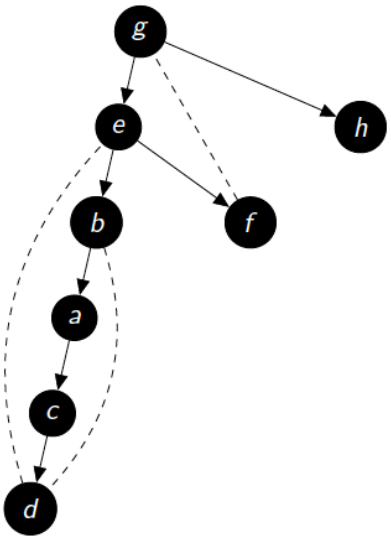
Quel est l'arbre recouvrant associé à ce parcours ?

Parcours en Profondeur *(Depth First Search)*

Exemple. Soit le graphe G ci-dessous avec comme source le sommet g



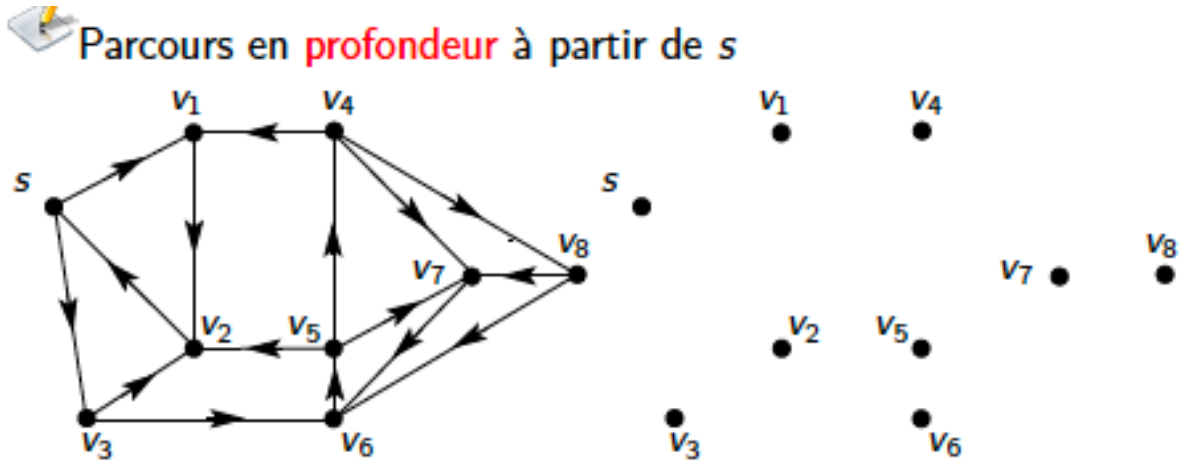
Parcours DFS : g e b a c d f h

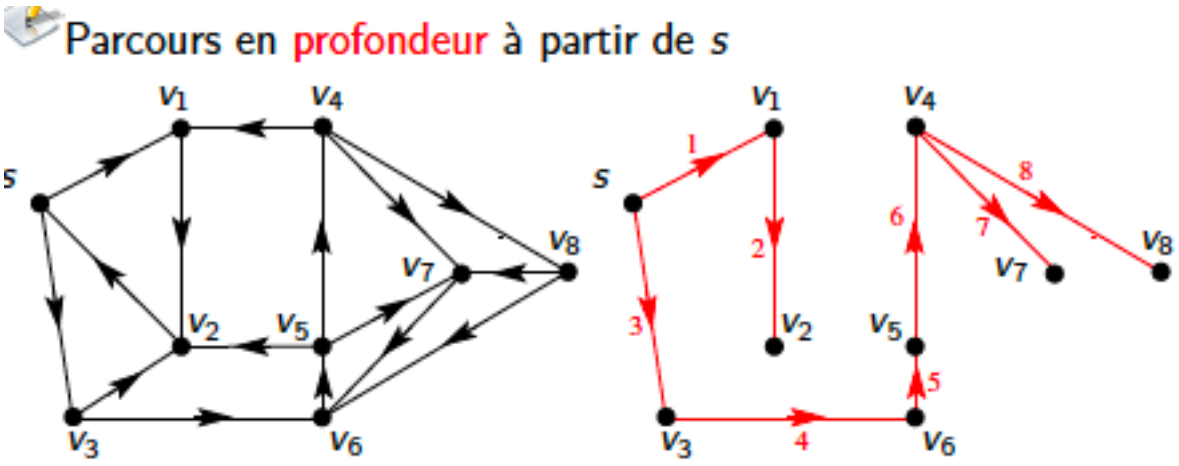


Arbre Recouvrant

Faire le parcours slides 49 à 76 puis 78 à 95 du support Jean-Manuel Méry

EXO Source slides pp 32-35





On considère le parcours en profondeur à partir d'un sommet source. L'algorithme se formule naturellement sous *forme récursive.*

AU PREALABLE : initialiser PRED et marquer sommets en blanc ;

```
Procédure DFS_rec (G : graphe, s : sommet source) ;
début
        marquer le sommet s en gris ;
        pour tout sommet t successeur de s faire
                si t est de couleur blanche
                        alors
                                PRED[t] := s ;
                                DFS_rec (G, t)
        fpour
        marquer s en noir
fin
```

Rappel de la définition de DFS_rec

```
Procédure DFS_rec (G : graphe, s : sommet source) ;
début
        marquer le sommet s en gris ;
        pour tout sommet t successeur de s faire
                si t est de couleur blanche
                        alors
                                PRED[t] := s ;
                                DFS_rec (G, t)
        fpour
        marquer s en noir
fin
```

Pour visiter tous les sommets du graphe et ainsi construire une forêt recouvrante.

```
Procédure DFS (G : graphe) ;            % DFS : Depth First Search
début
        pour tout sommet t de S faire
                PRED[t] := nil ;
                marquer t en blanc
        fpour
        tantque il existe (au moins) un sommet s de couleur blanche faire
                DFS_rec(G, s) ;
fin
```

Parcours en profondeur à partir d'un sommet source sous *forme itérative.*

Le parcours en profondeur peut aussi se formuler *sous forme itérative* en utilisant une *pile*.

```
Procédure DFS_iteratif (G : graphe, s : sommet source)
début
        initStack(P) ;
        marquer s en gris ;
        push(s, P) ;
        tantque non emptyStack(P) faire
                pop(u, P) ;
                pour tout sommet v successeur de u faire
                        si v est de couleur blanche
                                alors
                                        push(v, P) ;
                                        marquer v en gris ;
                                        PRED[v] := u
                fin pour ;
                marquer u en noir
        finttque
fin
```

Pour visiter tous les sommets du graphe et ainsi construire une forêt recouvrante.

```
Procédure DFS (G : graphe) ;            % DFS : Depth First Search
début
pour tout sommet t de S faire PRED[t] := nil ; marquer t en blanc fpour
tantque il existe (au moins) un sommet s de couleur blanche faire
        DFS_itératif(G, s) ;
fin
```

Complexité temporelle de DFS

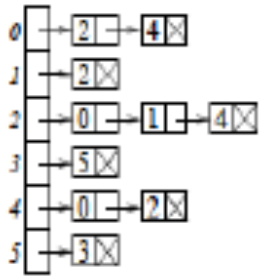
Chaque sommet (accessible depuis s) est empilé (puis dépilé) une fois et une seule.

A chaque fois qu'on enlève un sommet de la pile, on parcourt tous ses successeurs

→ chaque arc (ou arête) du graphe sera utilisé une fois et une seule dans l'algorithme.

Graphe non orienté : $S = \{0, 1, 2, 3, 4, 5\}$, $A = \{\{0, 2\}, \{0, 4\}, \{1, 2\}, \{2, 4\}, \{3, 5\}\}$

	0	1	2	3	4	5
0	0	0	1	0	1	0
1	0	0	1	0	0	0
2	1	1	0	0	1	0
3	0	0	0	0	0	1
4	1	0	1	0	0	0
5	0	0	0	1	0	0

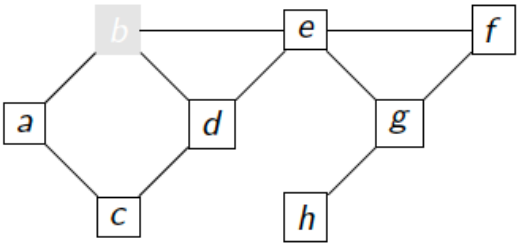


Si le graphe contient n sommets (accessibles à partir de s) et m arcs/arêtes, alors :

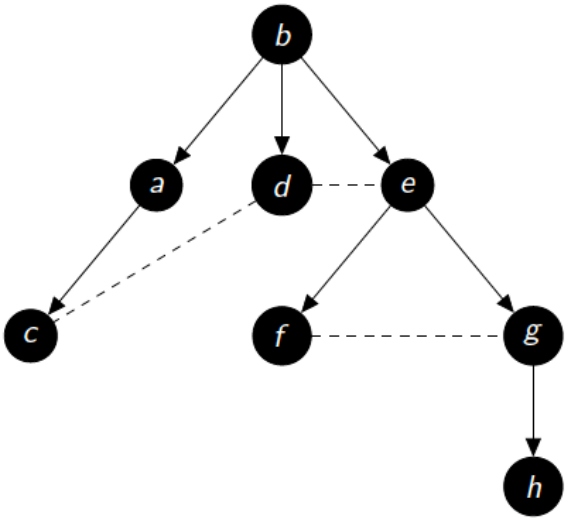
- pour une représentation par matrice d'adjacence en *O(n^2)*
- pour une représentation par listes de successeurs en *O(n + m)*

Parcours en Largeur *(Breath First Search)*

Exemple. Soit le graphe G ci-dessous avec comme source le sommet b



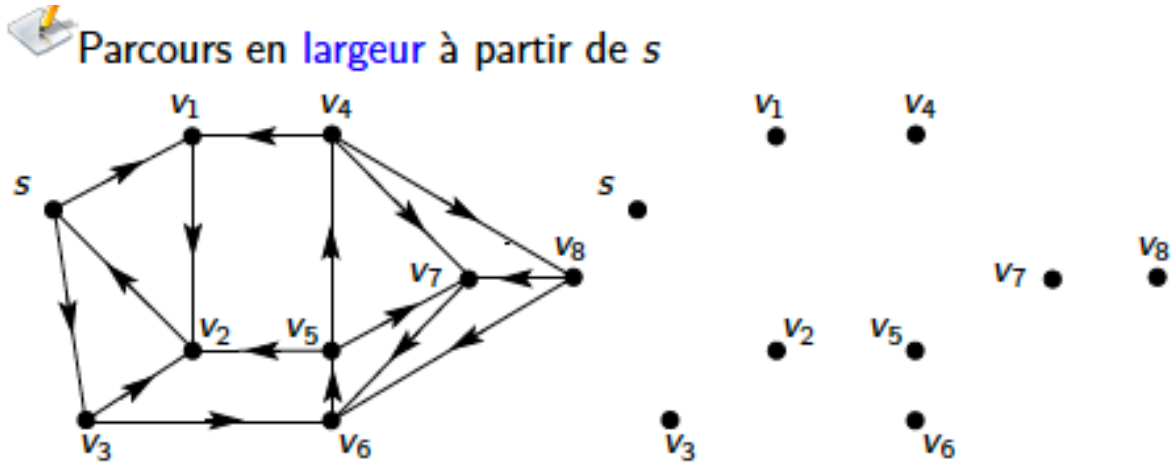
Parcours BFS : b a d e c f g h

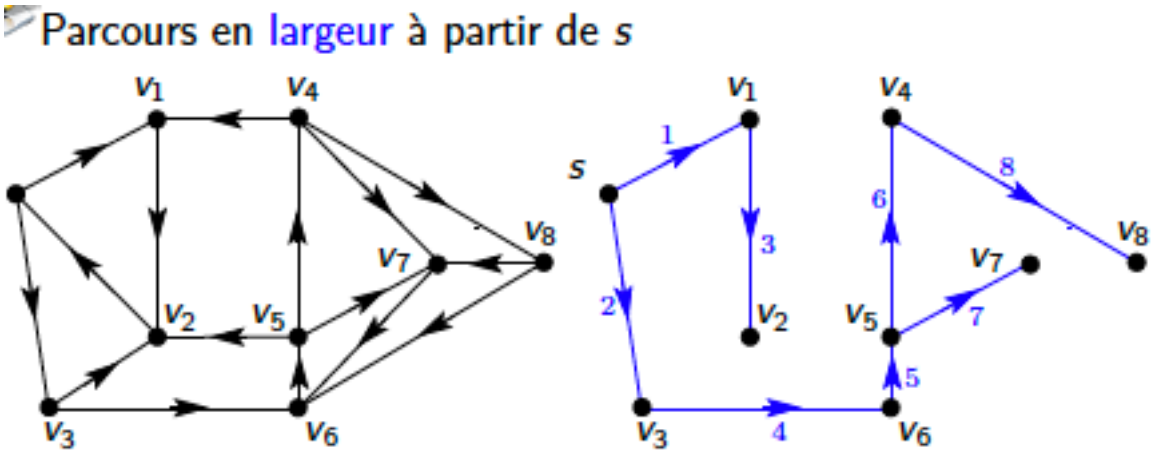


Arbre Recouvrant

Faire le parcours slides 13 à 28 puis 31 à 42 su support Jean-Manuel Méry

EXO Source slides pp 32-35





Le parcours en largeur se formule *sous forme itérative* en utilisant une *file.*

```
Procédure BFS  (G : graphe) ;                % BFS : Breath First Search
début
pour tout sommet t de S faire PRED[t] := nil ; marquer t en blanc fpour
tantque il existe (au moins) un sommet s de couleur blanche faire
                BFS_itératif(G, s) ;
fin

Procédure BFS_iteratif (G : graphe, s : sommet source)
début
        initQueue(Q) ;
        marquer s en gris ;
        enqueue(s, Q) ;
        tantque non emptyQueue(Q) faire
                dequeue(u, Q) ;
                pour tout sommet v successeur de u faire
                        si v est de couleur blanche
                                alors
                                        enqueue(v, Q) ;
                                        marquer v en gris ;
                                        PRED[v] := u
                fin pour ;
                marquer u en noir
        finttque
fin
```

Complexité temporelle de BFS

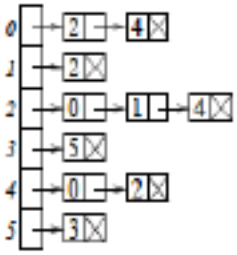
Le raisonnement est analogue à celui tenu pour le parcours en profondeur.

Chaque sommet (accessible depuis s) est mis, puis enlevé, une fois et une seule dans la file.

A chaque fois qu'on enlève un sommet de la file, on parcourt tous ses successeurs
→ chaque arc (ou arête) du graphe sera utilisé une fois et une seule dans l'algorithme.

Graphe non orienté : $S = \{0,1,2,3,4,5\}$, $A = \{\{0,2\},\{0,4\},\{1,2\},\{2,4\},\{3,5\}\}$

	0	1	2	3	4	5
0	0	0	1	0	1	0
1	0	0	1	0	0	0
2	1	1	0	0	1	0
3	0	0	0	0	0	1
4	1	0	1	0	0	0
5	0	0	0	1	0	0



Si le graphe contient n sommets (accessibles à partir de s) et m arcs/arêtes, alors :

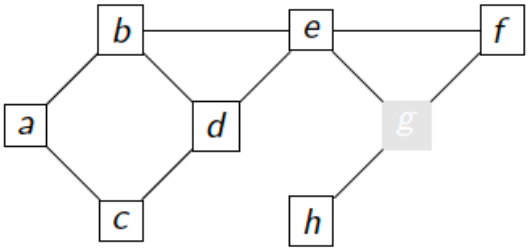
- pour une représentation par matrice d'adjacence en *O(n^2)*
- pour une représentation par listes de successeurs en *O(n + m)*

Etiquetage des sommets lors du parcours en profondeur

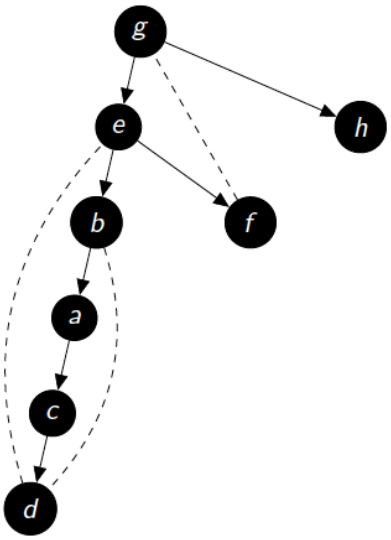
Le temps est discrétisé et représenté par la variable Temps initialisée à 0. On « tamponne » deux fois chaque sommet t en fonction du temps :

- DEB[t] = date de découverte (1ère visite) du sommet t (t devient gris)
- FIN[t] = date à laquelle se termine le traitement de t (t devient noir)

Exemple. Soit le graphe G ci-dessous avec comme source le sommet g

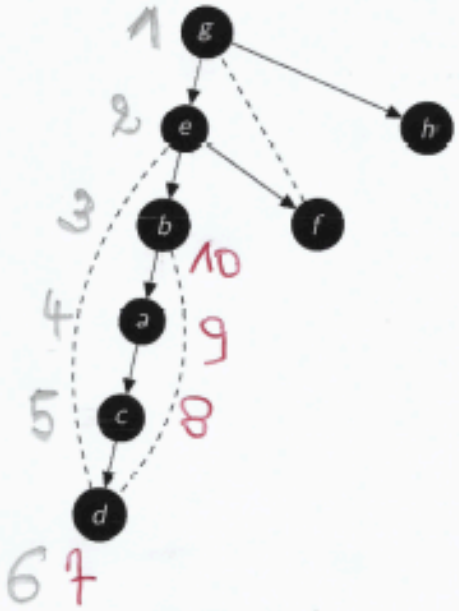


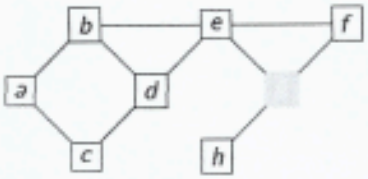
Parcours DFS : g e b a c d f h

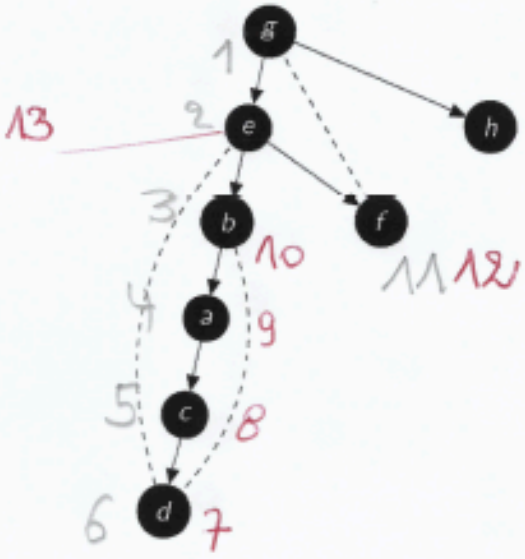


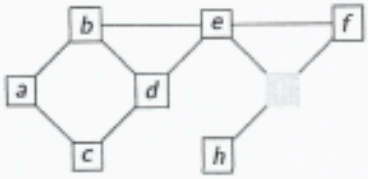
Arbre Recouvrant

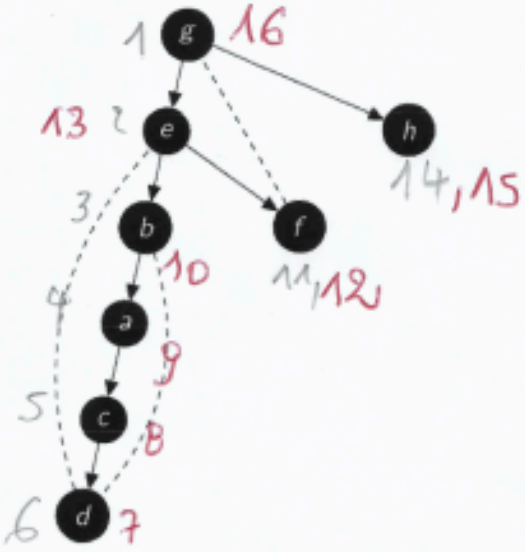

b
e
f
a
d
c
h


1
g
2
e
h
3
b
f
10
4
a
9
5
c
8
d
6
7


b
e
f
a
d
c
h


1
g
13
2
e
h
3
b
f
10
11
12
4
a
9
5
c
8
6
d
7


b
e
f
a
d
c
h


1
g
16
13
2
e
h
14,15
3
b
f
10
11,12
4
a
9
5
c
8
6
d
7

Le temps est discrétisé et représenté par la variable Temps initialisée à 0. On « tamponne » deux fois chaque sommet t en fonction du temps :

- DEB[t] = date de découverte (1ère visite) du sommet t (t devient gris)
- FIN[t] = date à laquelle se termine le traitement de t (t devient noir)

Pour cela, on adapte le pseudo code du parcours en profondeur (version récursive)

```
_____________________________________________________________________
Procédure ETIQUETER (G : graphe) ;                %
début
        pour tout sommet t de S faire PRED[t] := nil ; marquer t en blanc fpour
        Temps := 0 ;
        tantque il existe (au moins) un sommet s de couleur blanche faire
                DFS_rec_BIS(G, s) ;
fin

Procédure DFS_rec_BIS(G : graphe, s : sommet source) ;
début
        marquer le sommet s en gris ;
        Temps := Temps + 1 ;
        DEB[s] := Temps ;
        pour tout sommet t successeur de s faire
                si t est de couleur blanche
                        alors
                                PRED[t] := s ;
                                DFS_rec_BIS (G, t)
        fpour
        marquer s en noir ;
        Temps := Temps + 1 ;
        FIN[s] := Temps
fin
_____________________________________________________________________
```

(1) Calcul des composantes connexes d'un graphe non-orienté

Définitions

Un *graphe non orienté est connexe* si chaque sommet est accessible à partir de n'importe quel autre, i.e. si pour tout couple de sommets distincts (u, v) il existe une chaine entre u et v.

Une composante connexe d'un graphe non orienté G est un sous-graphe G' de G qui est connexe et maximal (c'est à dire qu'aucun autre sous-graphe connexe de G ne contient G').

Exemples

Le graphe ci-dessus n'est pas connexe car il n'existe pas d'arête entre le sommet a et le sommet e. Par contre le sous-graphe de sommets {a, b, c, d} et le sous-graphe de sommets {e, f, g} sont connexes.

Algorithme. Pour déterminer les composantes connexes d'un graphe non orienté, il suffit d'appeler itérativement la procédure DFS_rec(G, s) à partir des sommets blancs, jusqu'à ce que tous les sommets soient noirs.
Le nombre d'appels à la procédure DFS_rec(G, s) sera égal au nombre de composantes connexes du graphe.

```
Procédure Calcul_CC(G : graphe non orienté) ;  % Composantes connexes
début
        pour tout sommet s de S faire
                PRED[s] := nil ;
                marquer s en blanc
        fpour
        nbComposantes :=0 ;
        tantque il existe (au moins) un sommet t de couleur blanche faire
                DFS_rec(G, t) ;
                nbComposantes := nbComposantes + 1
fin
```

Exemple :

a) Il y a 2 composantes connexes

CC1 = {a, b, c, d} à gauche CC2 = {e, f, g} à droite

b) Forêt recouvrante composée de 2 arbres

arbre couvrant CC1 arbre couvrant CC2

Propriétés

- Une composante connexe d'un graphe est un sous-graphe connexe de ce graphe.
- Un graphe dont toutes les composantes connexes sont des arbres est une forêt.
- Un graphe connexe à n sommets possède au moins n-1 arêtes.
- Un graphe connexe à n sommets ayant exactement n-1 arêtes est un arbre.
- Un graphe à n sommets avec k composantes connexes possède au moins n-k arêtes.

Connexité et fermeture transitive

- La fermeture transitive d'un graphe connexe est un graphe complet.
- La fermeture transitive d'un graphe comportant k composantes connexes est un graphe contenant k sous-graphes complets (un pour chaque composante connexe).

(2) Tri topologique des sommets d'un graphe orienté sans circuit

Le tri topologique d'un graphe orienté sans circuit G = (S, A) consiste à ***ordonner linéairement tous ses sommets de sorte que, si G contient un arc (u, v), u apparaisse avant v dans le tri.***
(Si le graphe n'est pas sans circuit, aucun ordre linéaire n'est possible.)

Le tri topologique d'un graphe peut être vu comme un alignement de ses sommets le long d'une ligne horizontale de manière que tous les arcs soient orientés de gauche à droite. Les graphes orientés sans circuit sont utilisés dans de nombreuses applications pour représenter des précédences entre événements.

La figure 22.7 donne l'exemple du savant Cosinus qui s'habille le matin. Le professeur doit enfiler certains vêtements avant d'autres (par exemple les chaussettes avant les chaussures). D'autres peuvent être mis dans n'importe quel ordre (par exemple, les chaussettes et le pantalon). Un arc (u, v) du graphe orienté sans circuit de la figure 22.7(a) indique que le vêtement u doit être enfilé avant le vêtement v.

Exemple.

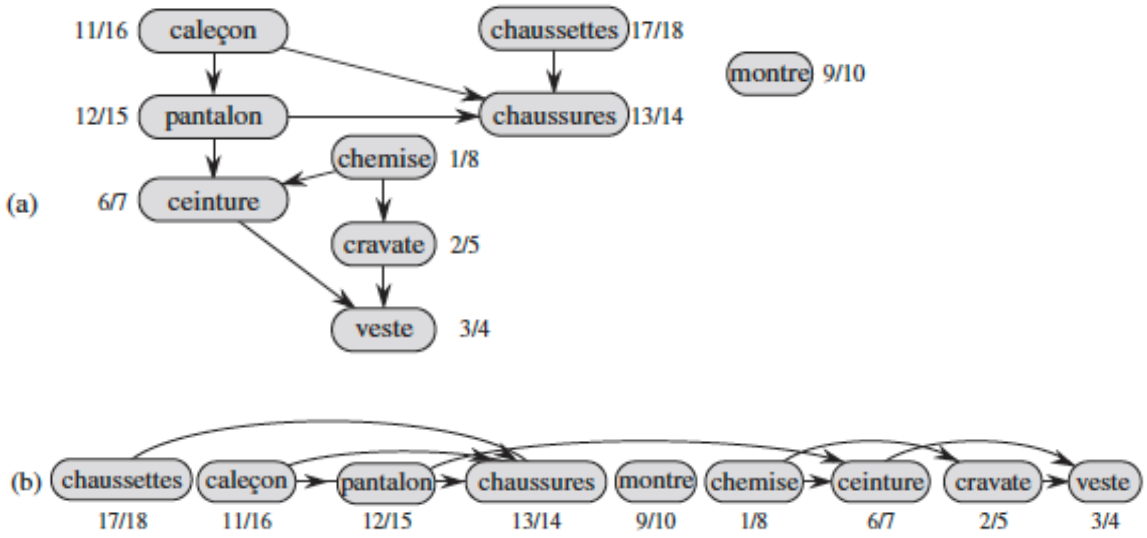
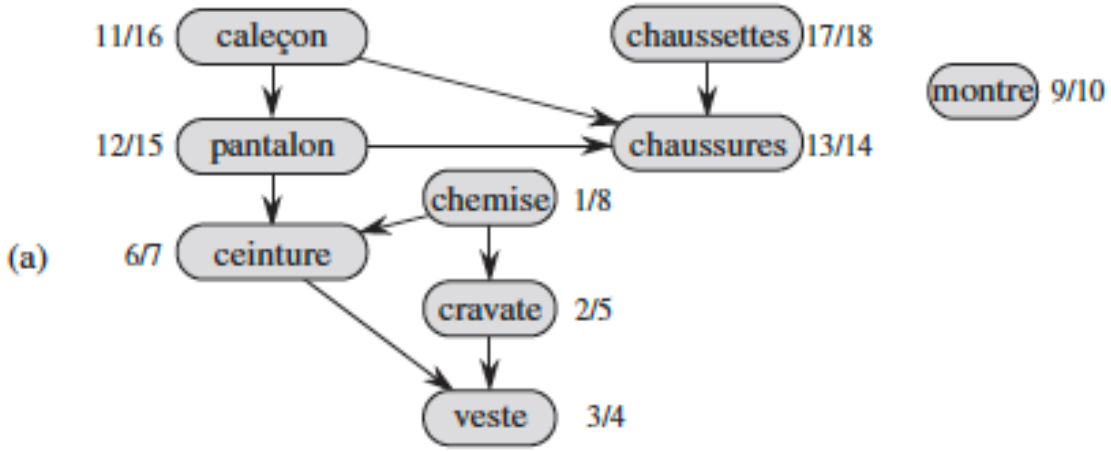


Figure 22.7 **(a)** Le savant Cosinus trie topologiquement ses vêtements quand il s'habille. Chaque arc (u, v) signifie que le vêtement u doit être enfilé avant le vêtement v. Les dates de découverte et de fin de traitement résultant d'un parcours en profondeur sont données à côté de chaque sommet. **(b)** Le même graphe trié topologiquement. Ses sommets sont ordonnés de gauche à droite par ordre décroissant des dates de fin de traitement. Notez que tous les arcs sont orientés de gauche à droite.



Le tri topologique de ce graphe orienté sans circuit donne donc un ordre permettant de s'habiller correctement. La figure 22.7(b) montre le graphe orienté sans circuit trié topologiquement comme une suite de sommets sur une ligne horizontale de telle façon que tous les arcs soient orientés de gauche à droite.

Donner un autre ordre topologique

Algorithme #1.

On étiquette les sommets lors du parcours en profondeur.

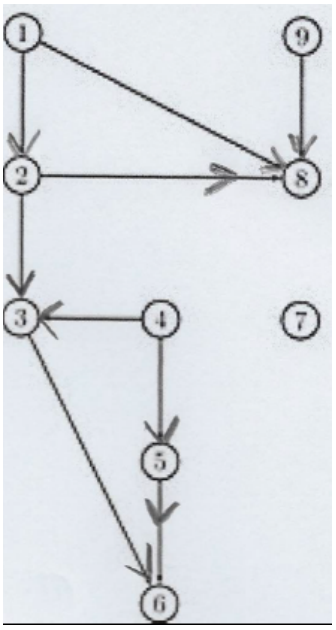
On constate que pour tout arc (u, v), on a : FIN[v] < FIN[u]

En effet :

- si v est noir alors FIN[v] < temps < FIN[u],
- si v est blanc alors DEB[u] = temps < DEB[v] < FIN[v] < FIN[u],
- enfin, v ne peut pas être gris car ça impliquerait l'existence d'un circuit.

→ Il suffit donc de trier les sommets par ordre de valeur de fin décroissante.

Exemple wikipedia



On commence par le sommet 1, on a alors pour ce point la date de début : (1,) , la date de fin n'étant pas encore connue. On a alors le choix de continuer soit vers 2 ou vers 8. Supposons que l'on aille vers 2. On aura alors pour 2 (2,), puis en 3 avec (3,) et enfin en 6 avec (4,).

Ici, 6 est un sommet dont ne part aucun arc. Il est donc impossible de descendre encore de niveau. On marque donc notre première date de fin pour 6 : (4 , 5). Puis on « remonte » en passant par le parent d'où l'on vient, c'est-à-dire 3.

Comme il n'y a pas d'autre arc partant du sommet 3, on marque notre deuxième date de fin pour 3 : (3 , 6) puis on « remonte » encore par le parent d'où l'on est venu, c'est-à-dire 2.

Du sommet 2, on continue en suivant l'arc qui n'a pas encore été visité (vers 8), d'où il est impossible de descendre davantage. On a donc pour 8 : (7 , 8) puis on remonte au sommet 2.

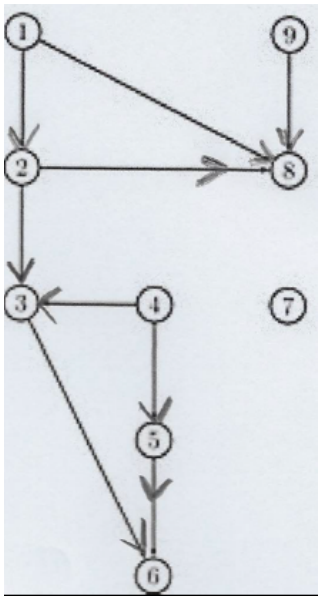
Cette fois, tous les arcs partant du sommet 2 ont été traités, donc on marque la date de fin pour 2 : (2 , 9) et on remonte vers le sommet 1, qui est dans la même situation, donc 1 : (1, 10).

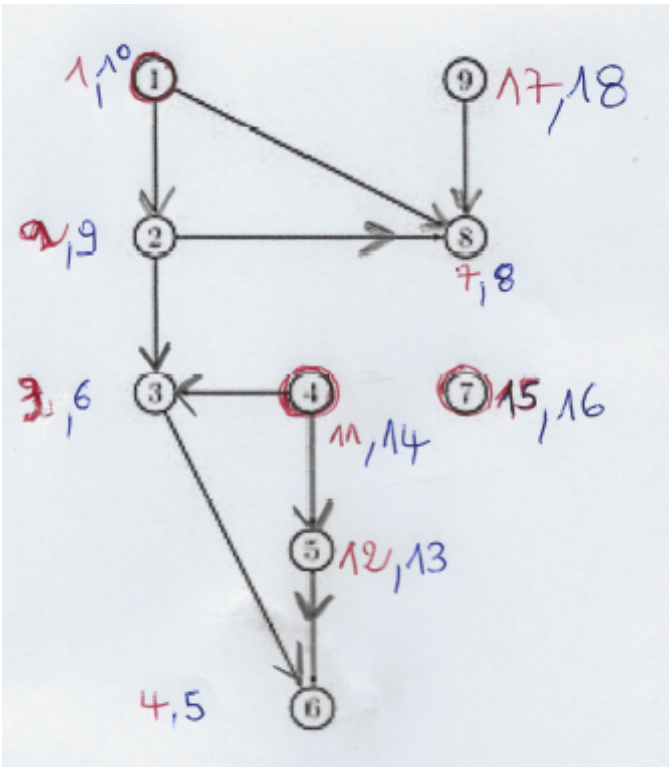
On répète ensuite ces opérations en commençant par un sommet qui n'a pas encore été visité, par exemple 4 : (11,) d'où on peut descendre en 5 : (12,), mais pas plus loin car 6 a déjà été visité. On a donc pour 5 : (12 , 13), puis après être remonté en 4 : (11, 14) car tous ses voisins ont été visités.

On recommence avec 7 : (15 , 16) et 9 : (17 , 18), qui n'ont aucun voisin non-traité.

→ on trie les sommets par date de fin décroissante pour obtenir l'ordre suivant :

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





Finalement, on trie les sommets par date de fin décroissante pour obtenir l'ordre suivant :

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

Proposer d'autres ordres topologiques ?

Algorithme #2.

On peut se passer de l'étiquetage des sommets par dates de début/fin. En effet, comme le tri topologique correspond aux sommets par ordre de valeur de fin décroissante, il suffit

a) de les empiler successivement quand ils deviennent noirs,

b) et de dépiler une fois tous les sommets parcourus.

```
Procédure TRI_TOPO_2 (G : graphe orienté sans cycle) ;
début
        AU PREALABLE : initialiser PRED et marquer sommets en blanc ;
        initStack(P) ;
        tantque il existe (au moins) un sommet s de couleur blanche faire
                DFS_rec_TER(G, s)
        fttque ;
        tantque non emptyStack(P)  faire
                 pop(t, P) ;
                écrire(t)
        fttque
fin

Procédure DFS_rec_TER(G : graphe, s : sommet source) ;
début
        marquer le sommet s en gris ;
        pour tout sommet t successeur de s faire
                si t est de couleur blanche
                        alors
                                PRED[t] := s ;
                                BFS_rec_TER (G, t)
        fpour
        marquer s en noir ;
        push(s, P)
fin
```

(3) Ordonnancement : Méthode Potentiels/Tâches

Bien que les nouvelles méthodes de conception de systèmes de production aient tendance à diminuer la taille de certains problèmes d'ordonnancement en divisant les systèmes en cellules flexibles élémentaires, la diversité, la complexité et l'importance dans le monde industriel des problèmes d'ordonnancement demeurent très grandes. Ainsi les réalisations importantes, tels que la construction d'un barrage, d'une centrale, d'un immeuble, d'un avion, le fonctionnement d'une chaîne de fabrication, le développement d'un très volumineux logiciel informatique, … demandent une surveillance constante et une parfaite coordination des différentes cellules de travail pour éviter des pertes de temps souvent très onéreuses.

Ordonnancer, c'est programmer dans le temps l'exécution d'une réalisation décomposable en tâches, en attribuant des ressources à ces tâches (en fixant en particulier leurs dates de début d'exécution) tout en respectant un ensemble de contraintes afin d'optimiser un ou plusieurs critères fixés.

Les problèmes réels paraissent a priori très complexes. Cependant, ils peuvent souvent être résolus de façon très satisfaisante. Ces problèmes sont distincts les uns des autres et ne peuvent pas être traités efficacement à l'aide d'un outil unique et standard. Les décideurs ignorent souvent l'origine des difficultés de leur résolution. Les outils théoriques, issus du monde de la recherche, permettent à l'heure actuelle de résoudre certains problèmes de grande taille. Les bonnes heuristiques, suffisantes le plus souvent, sont des « sous-produits » d'études théoriques fines.

On qualifie de « central » un problème d'ordonnancement sans contraintes de ressources ou pour lequel les ressources sont toujours en quantité suffisante quel que soit l'ordonnancement. L'objectif de cette section est d'étudier différentes méthodes de résolution de tels problèmes.

Formulation du problème.

Soit un ensemble de n *tâches* T = {t_1, t_2, … , t_n} où chaque tâche t_i est de *durée* d_i, et soit C l'ensemble des *contraintes de précédence* portant sur T.

On note :

- T_i la *date effective de début* de la tâche t_i
- a_i la *date de début au plus tôt* de t_i
- b_i la *date de début au plus tard* de t_i

On souhaite ***minimiser la durée de l'ordonnancement***.

On considère le graphe orienté G = (S, A) muni de la valuation w tq :

- S = T u {Deb, Fin} où Deb est une tâche fictive de début et Fin est une tâche fictive de fin
- l'arc (t_i, t_j) appartient à A ssi la tâche t_i précède la tâche t_j
- la valuation de l'arc (t_i, t_j) est w(t_i, t_j) = d_i

Soit plc(t_i, t_j) le poids du plus long chemin entre les sommets t_i et t_j dans le graphe G.
Alors :

- la durée minimale de l'ordonnancement : plc(Deb, Fin)
- a_i (date de début au plus tôt de t_i): plc(Deb, t_i)
- b_i (date de début au plus tard de t_i) : plc(Deb, Fin) - plc(t_i, Fin)

Une tâche t_i est critique ssi a_i = b_i.

Toutes les tâches critiques appartiennent au plus long chemin de Deb à Fin

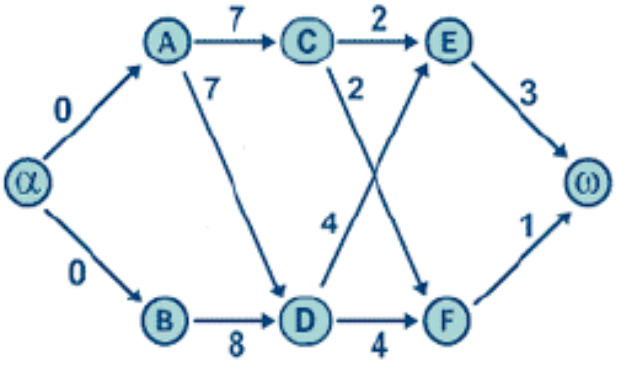
Exemple : résoudre le problème central ci-dessous

	Durée	Prédécesseurs
A	7	//
B	8	//
C	2	A
D	4	A et C
E	3	C et D
F	1	C et D

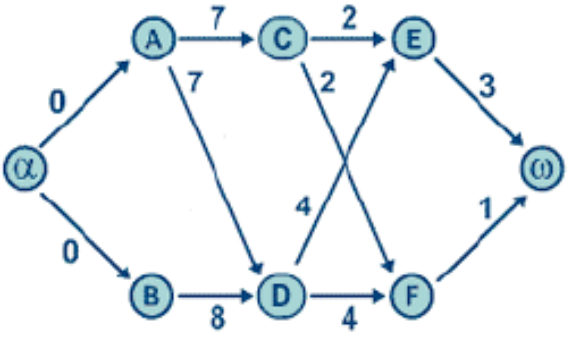
typo : préd. de D sont A et B (pas C)

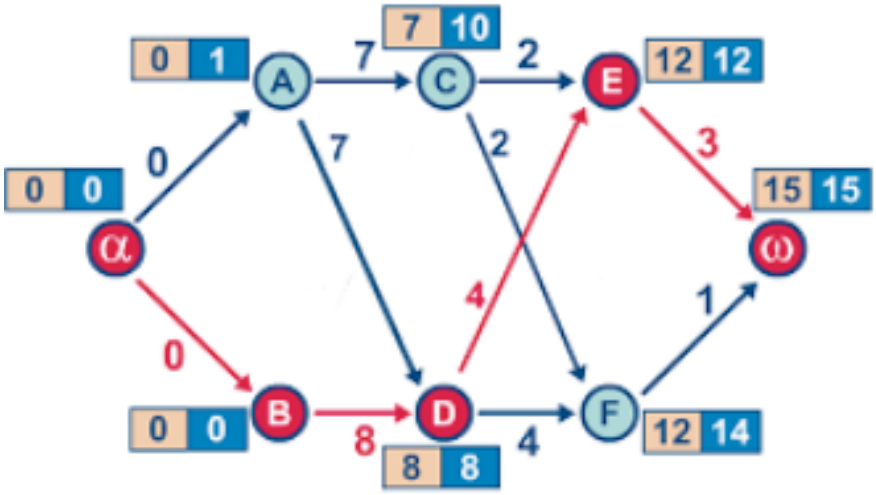
1) Représenter le problème sous forme d'un graphe potentiels/tâches
2) Pour chaque tâche, calculer sa date de début au plus tôt et sa date de début au plus tard
3) Quel est le chemin critique ?
4) Représenter le déroulement du projet sous forme d'un diagramme de GANTT

Graphe potentiels/tâches



Durée minimale Ordonnancement et dates début/fin au plus tôt

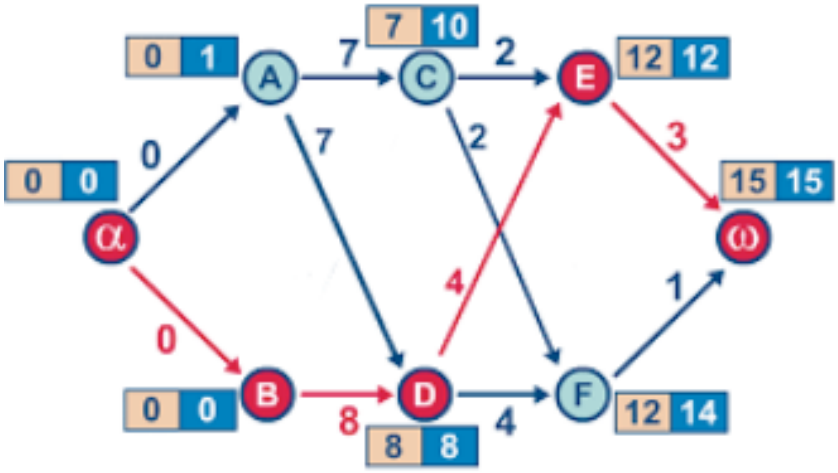




Durée minimale 15 unités de temps

Chemin critique → Alpha-B-D-E-Omega

Diagramme de GANTT



Graphe Potentiel – Tâches (en rouge le chemin critique)

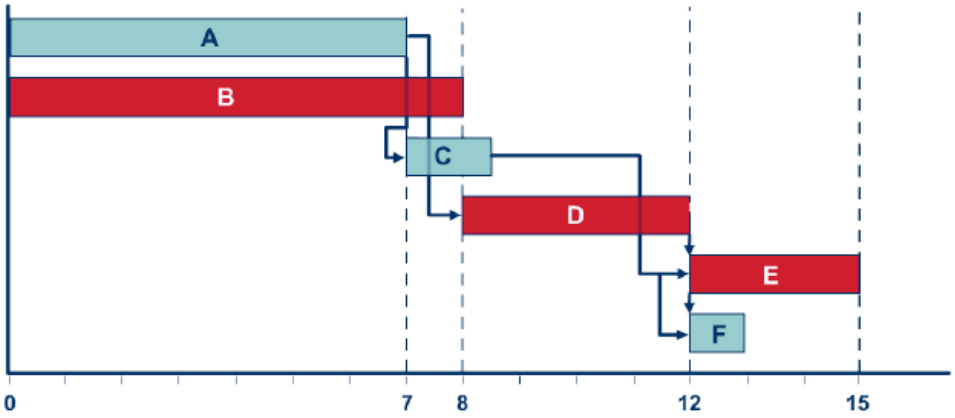


Diagramme de GANTT

- Chaque tâche est représentée par une barre de longueur proportionnelle à sa durée.
- Les contraintes de précédence entre tâches sont matérialisées par des flèches.
- Le diagramme de GANTT montre qu'il est possible de déplacer les tâches A, C et F tout en respectant les contraintes de précédence et sans retarder la fin des travaux.