ENSICAEN
ÉCOLE NATIONALE SUPÉRIEURE D'INGÉNIEURS DE CAEN
& CENTRE DE RECHERCHE

Année 2014/2015

Cahier de TD/TP d'Algorithmique Avancée

Christine Porquet

ENSICAEN – 1ère année
Spécialité Informatique

Contenu

Les fichiers à utiliser pour l'ensemble des TPs sont récupérables sur la plateforme pédagogique, cours « `Algorithmique avancée` », dans le dossier « `Fichiers pour les TPs` ».

FILES D'ATTENTE

Buts : Implantation des files d'attente en représentation chaînée

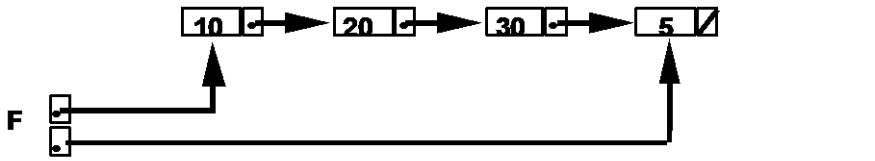
Durée : 1 semaine

Rappel du cours : le type abstrait "file d'attente" = structures de données + fonctions
On appelle *type abstrait* l'association **structure de données + fonctions** permettant de définir toutes les manipulations que l'on souhaite réaliser sur ce type de données.

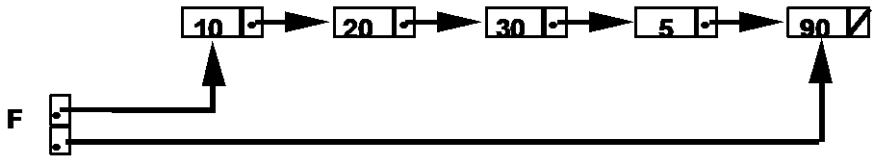
Les opérations que nous souhaitons réaliser sont les opérations caractéristiques des *files d'attente*:
- créer une file vide,
- tester si la file est vide,
- enfiler un élément (ie. l'ajouter en queue de file),
- défiler (ie. supprimer l'élément se trouvant en tête de file),
- concaténer 2 files (ie. les mettre bout à bout),
- afficher le contenu de la file.

Chacune de ces opérations sera définie sous forme de fonction C. Toutes ses opérations doivent être **O(1)** (sauf bien sûr l'affichage qui est **O(n)**).

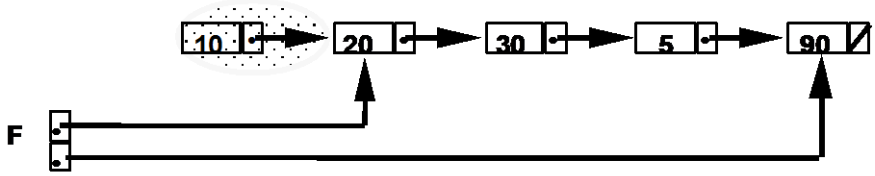
Vous avez le choix entre deux représentations chaînées des files :
- file « classique » avec deux pointeurs tête et queue,



enfiler 90

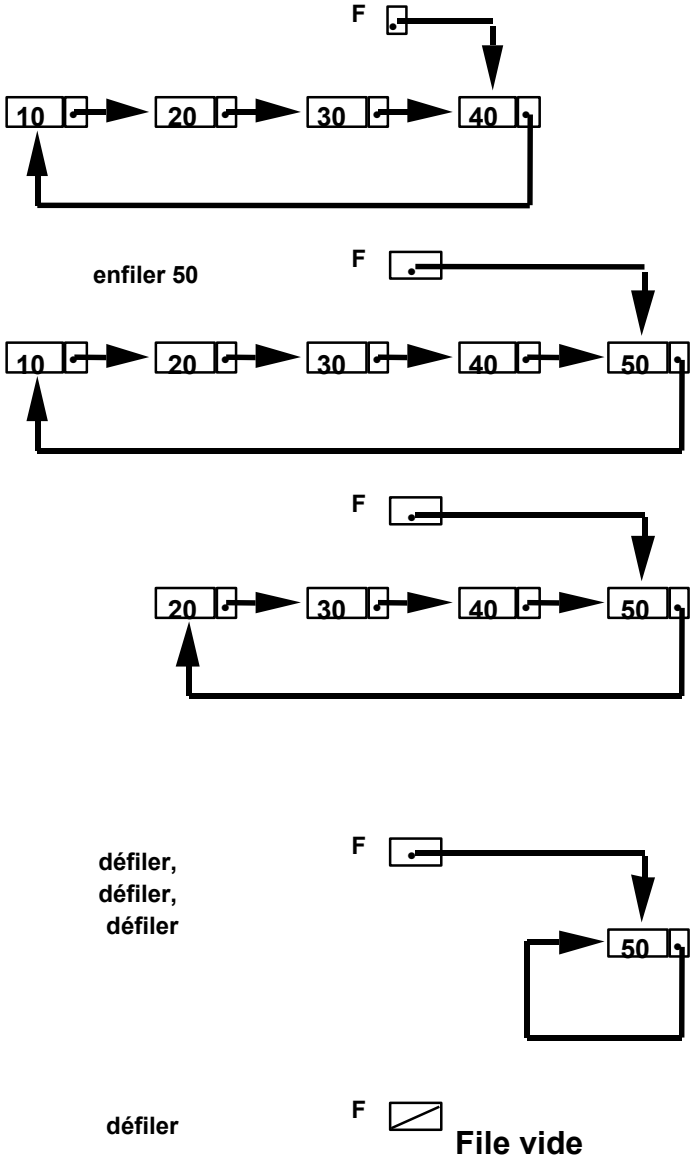


défiler



file vide

- file « circulaire » avec un unique pointeur de queue.



Travail à réaliser

Implémenter l'ensemble des fonctions de manipulations des files d'attente proposées. Leur bon fonctionnement devra avoir été soigneusement testé et validé. En effet, ces fonctions seront réutilisées dans les TPs suivants.

Facultatif

- copie de file d'attente : `File copie(File);`
- destruction de file d'attente : `void destruction(File *);`
  Il doit en résulter une file vide dont toutes les cellules de liste ont été désallouées.

PARCOURS D'ARBRE BINAIRE DE RECHERCHE

Buts : Manipuler les arbres binaires de recherche

Réutiliser le type abstrait « file d'attente »

Durée : 2 semaines

Fichier(s) à utiliser/modifier : `ABR_TP.c`

TRAVAIL À RÉALISER

Écrire un programme permettant de générer / afficher / parcourir des arbres binaires de recherche.

Les fonctions de parcours à programmer sont :

- les 3 fonctions récursives classiques (préfixe/ infixe / suffixe) ;
- une fonction récursive qui compte et retourne le ***nombre de feuilles*** de l'ABR ;
- une fonction récursive avec affichage en ordre préfixe ***et*** suffixe et comptage du ***nombre de de noeuds*** de l'ABR ;
- une fonction de *parcours en largeur* (utilisation du type abstrait « file d'attente ») avec comptage du ***nombre de nœuds*** de l'ABR.

```
Parcours_largeur (rac : pointeur sur la racine de l'ABR);

F = file_vide;
Enfiler(rac,F);
Tant que F est non vide faire
      p = Défiler(F); /* p, pointeur sur un noeud de l'ABR */
      Afficher(contenu(p));
      Si p a un fils gauche, Enfiler (gauche(p), F);
      Si p a un fils droit,  Enfiler (droit(p), F);
```

N.B. : Vous réutiliserez les fonctions sur les files d'attente du premier TP. Attention, au lieu de manipuler des files d'***entiers***, il s'agit maintenant manipuler des files de ***pointeurs sur des nœuds d'ABR***.

FACULTATIF : BORDURE D'ARBRE

La bordure d'un ABR est la liste de ses feuilles et peut se définir récursivement comme la ***concaténation*** des bordures des sous-arbres gauche et droit de l'arbre.

Ecrire une fonction qui construit et retourne la bordure d'un ABR sous forme d'une file et dont le prototype est :

```
File bordure(ABR);
```

ARBRE N-AIRE DE MOTS

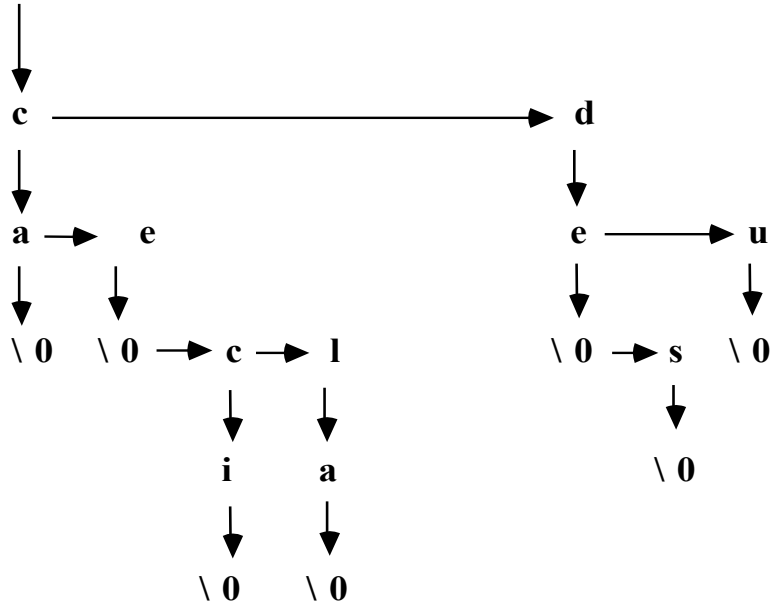
Buts : Etude et manipulation d'un arbre n-aire dans la représentation « fils-frère »

Durée : 3,5 semaines

Fichiers(s) à utiliser : `dico.ang, dico.fr`

Dans un dictionnaire de mots où figurent toutes les formes des mots, on a beaucoup de préfixes de mots communs (en particulier pour les conjugaisons des verbes). Représenter un tel dictionnaire sous forme d'arbre n-aire permet de ne stocker qu'une seule fois chaque préfixe de mot. La structure retenue est une *arborescence ordonnée* de mots dans laquelle chaque nœud contient une lettre d'un mot et deux pointeurs sur le fils gauche et le frère droit de cette lettre. Suivant le lien "frère", les lettres sont rangées par ordre *alphabétique*. De plus, pour indiquer les fins de mots, on utilise le caractère nul '\0'.

Exemple : L'ensemble de mots **{ ca, ce, ceci, cela, de, des, du }** est représenté par :



TRAVAIL À RÉALISER

- Définir une structure « nœud » avec 3 champs : le caractère qu'il contient et les 2 pointeurs sur le fils et le frère,

- Ecrire des fonctions *récursives* de ***recherche*** et d'***insertion*** d'un mot dans un arbre n-aire.

- Ecrire une fonction de ***chargement*** d'arbre n-aire à partir d'un fichier texte contenant un dictionnaire : on part d'un arbre vide et on insère (fonction ***insertion***) successivement chaque mot lu dans le fichier.

Une fois vos fonctions validées sur un tout petit dictionnaire, vous pourrez charger des données plus conséquentes : un dictionnaire français (`dico.fr`) et un dictionnaire anglais (`dico.ang`).

FACULTATIF

- *Facile* Evaluer la *complexité mémoire* de cette représentation. Il suffit de mettre en place un compteur d'allocation de nœuds. Discutez du résultat obtenu : Est-ce une méthode économe en mémoire ? Comparez avec la taille du fichier.

- *Assez facile* Ecrire une fonction d'*affichage* de l'ensemble des mots de l'arbre *par ordre alphabétique*. Adaptez-là ensuite de manière à en faire une fonction de *sauvegarde* de l'arbre sur fichier texte.

- *Difficile* Ecrire une fonction de suppression d'un mot de l'arbre. Attention : seules les lettres de la fin du mot qui ne sont pas partagées par d'autres mots doivent être supprimées.

- *Assez difficile* Ecrire une fonction de destruction de l'arbre avec désallocation de chacun des nœuds.

Début du fichier `dico.ang` *:*

```
45369
a
aarhus
aaron
ababa
aback
abaft
abandon
abandoned
abandoning
…
```

Début du fichier `dico.fr` *:*

```
87050
a
a-t-elle
a-t-il
a-t-on
abaissa
abaissable
abaissai
abaissaient
abaissais
…
```

PARCOURS DE GRAPHE EN PROFONDEUR ET EN LARGEUR

Buts : Manipuler des graphes non orientés non valués en représentation « matrice d'adjacence »

Réutiliser le type abstrait « file d'attente »

Durée : 2,5 semaines

Fichier à utiliser : `genere_graphe_0.c`

Représentation du graphe
Le graphe sera représenté sous forme de sa matrice d'adjacence (représentation baptisée M_ADJ en cours).

Format de stockage du graphe
Celle du cours, un fichier texte au format suivant :

```
6 7   Sur la première ligne, le nombre de sommets et le nombre d'arêtes du graphe.
0 1   Sur les lignes suivantes, une arête par ligne, donnée par ses sommets extrémités.
1 2   Les arêtes peuvent être données dans n'importe quel ordre.
2 3
3 4
4 5
5 0
1 4
```

TRAVAIL À RÉALISER

Écrire un programme permettant de parcourir des graphes non orientés non valués, connexes ou non.

Les fonctions de parcours à programmer sont :

- parcours en profondeur avec affichage des sommets en ordre préfixe ***et*** suffixe et comptage du ***nombre de nœuds*** de chaque composante connexe du graphe,
- parcours en largeur avec comptage du ***nombre de nœuds*** de chaque composante connexe du graphe.

Ces fonctions devront retourner un booléen indiquant si le graphe est connexe ou pas. En outre, elles devront fournir des informations sur les composantes connexes du graphe (nombre de composantes connexes / numéro de la composante connexe à laquelle appartient chaque sommet).

Pour le parcours en largeur, vous réutiliserez les fonctions sur les files d'attente du premier TP. Vous n'avez pas à en modifier le code. Vous inclurez directement le fichier d'en-tête correspondant.

Programmer et valider votre algorithme sur divers graphes (connexes ou non, petits ou gros). Pour générer des graphes au format demandé, vous pouvez utiliser le générateur de graphe `genere_graphe_0.c`.

FACULTATIF : BICOLORIAGE DE GRAPHE

Définition
On dit qu'un graphe non orienté G est *bicoloriable* s'il est possible de partager son ensemble S de sommets en deux sous-ensembles disjoints, l'ensemble B des sommets coloriés en bleu et l'ensemble J des sommets coloriés en jaune, de telle sorte qu'aucune arête de G ne relie deux sommets de même couleur (i.e. appartenant au même sous-ensemble).

Principe
On s'inspire du ***parcours en largeur*** : comme pour ce parcours, une file des sommets non encore examinés est utilisée. Au départ, on peut choisir n'importe quel sommet et le colorier, en bleu par exemple; si un sommet est colorié d'une couleur, il faudra alors colorier tous ses successeurs avec l'autre couleur.

➔ Ecrire une fonction qui détermine si un graphe G non orienté (connexe ou pas) est bicoloriable, et si oui, réalise un coloriage de sommets de G en `bleu/jaune` (ou toute autre paire de couleurs de votre choix). La fonction utilisera et retournera une variable booléenne `possible`, qui, si l'on détecte que deux sommets reliés sont de même couleur, passe à « faux ».

MARQUAGE TOPOLOGIQUE D'UN GRAPHE ORIENTÉ

Buts : Manipuler des graphes orientés non valués en représentation « listes d'adjacence »

Réutiliser le type abstrait « file d'attente »

Durée : 2 semaines

Fichier(s) à utiliser/modifier : `L_ADJ.c`

Définition

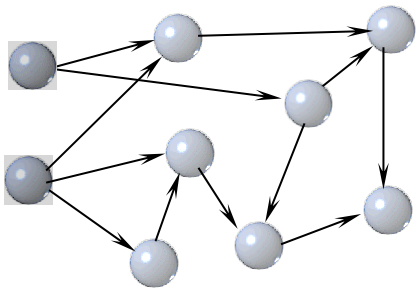
Un marquage topologique consiste en une numérotation $\{S_i\}$ des sommets du graphe de façon à respecter la propriété suivante de précédence entre les sommets :

$$\{S_i\} \text{ est un marquage topologique du graphe } G \Leftrightarrow S_j \text{ successeur de } S_i \Rightarrow i < j$$

Un algorithme de marquage topologique

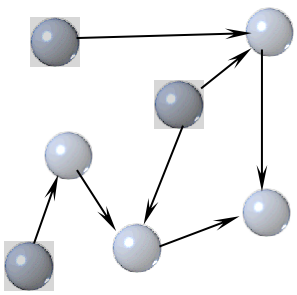
Cet algorithme assez simple de marquage topologique ne s'applique qu'à un graphe ayant plusieurs sommets sans prédécesseur. La propriété qui est à la base de cet algorithme est la suivante : les sommets ne possédant aucun prédécesseur doivent avoir les numéros les plus « bas » disponibles.

Soit un graphe G possédant 9 sommets (illustré ci-dessous). Les sommets sans prédécesseur sont repérés en foncé.



Dans le marquage topologique, ces deux sommets doivent donc être numérotés 0 et 1.

Une fois ces sommets numérotés, on les retire *virtuellement* du graphe, ainsi que tous les arcs dont ils sont à l'origine. Dans ce graphe réduit, apparaissent trois nouveaux sommets sans prédécesseurs (repérés en foncé).



On itère en les retirant *virtuellement* du graphe, ainsi que tous les arcs dont ils sont à l'origine. On s'arrête quand le graphe est « vide ».

L'ordre dans lequel on a retiré les sommets du graphe constitue un marquage topologique de celui-ci.

Travail à réaliser

Programmer l'algorithme de marquage topologique sur un graphe en représentation « listes d'adjacence » (baptisée L_ADJ dans le cours).

Vous modifierez les déclarations du fichier `L_ADJ.c` pour y ajouter le nombre de prédécesseurs de chaque sommet.

Une fois construit le graphe en mémoire, une fonction construira une *file de sommets sans prédécesseurs*. Pour ce faire, vous réutiliserez les fonctions sur les files d'attente du premier TP. Vous n'avez pas à en modifier le code. Vous inclurez directement le fichier d'en-tête correspondant.

Expliquons ce que signifie « retirer virtuellement un sommet du graphe » :

> Cela consiste à examiner tous les arcs qui partent de ce sommet pour décrémenter le nombre de prédécesseurs des sommets extrémités de ces arcs. Si ce nombre devient nul, le sommet extrémité de l'arc est alors enfilé dans la file des sommets sans prédécesseur.

Votre fonction devra retourner un booléen indiquant si le marquage topologique est possible ou non.

Facultatif

Programmer le même algorithme mais en utilisant à la place la représentation « liste de listes » (baptisée LL_ADJ dans le cours) et en remplaçant la file des sommets sans prédécesseurs par une *pile*.

Vous modifierez les déclarations du fichier `LL_ADJ.c` pour y ajouter le nombre de prédécesseurs de chaque sommet.

Une fois construit le graphe en mémoire, la liste des sommets sera « brisée » pour devenir une *pile de sommets sans prédécesseurs*. Pour ce faire, vous devez modifier uniquement les chaînages des nœuds, sans les désallouer.

ORDONNANCEMENT DE TÂCHES : LA MÉTHODE MPM

Buts : Manipuler des graphes orientés non valués en représentation « listes d'adjacence »

Durée : 2 semaines

Introduction

La réalisation de grands travaux nécessite la surveillance et la coordination des différentes tâches pour éviter les pertes de temps souvent coûteuses. Les problèmes qui traitent de cette coordination sont appelés « problèmes d'ordonnancement ».

Le problème que nous allons aborder ne traite que de *contraintes de succession dans le temps :* une tâche ne peut commencer que lorsque certaines autres sont terminées.

La méthode de B. Roy (1960), appelée également méthode MPM permet notamment de connaître la *date au plus tôt* et la *date au plus tard* de la fin des travaux.

Présentation de la méthode MPM sur un exemple

Cette méthode utilise un graphe appelé « potentiel tâches ». Voici sur un exemple les grandes lignes de la mise en œuvre de cette méthode.

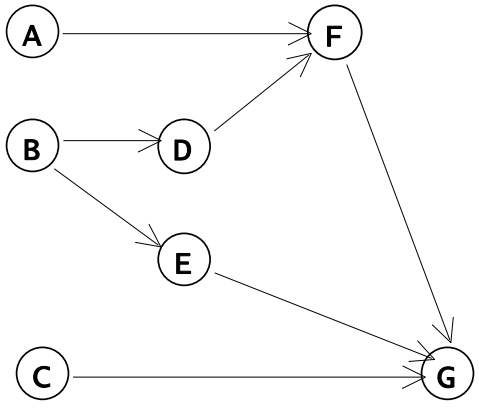
Première partie : date au plus tôt et date de fin des travaux

On doit exécuter 7 tâches A, B, C, D, E, F, G soumises aux contraintes suivantes :

tâche	durée des travaux (en semaines)	contraintes
A	6	
B	3	
C	6	
D	2	B doit être achevée
E	4	B doit être achevée
F	3	D et A doivent être achevées
G	1	F, E, C doivent être achevées

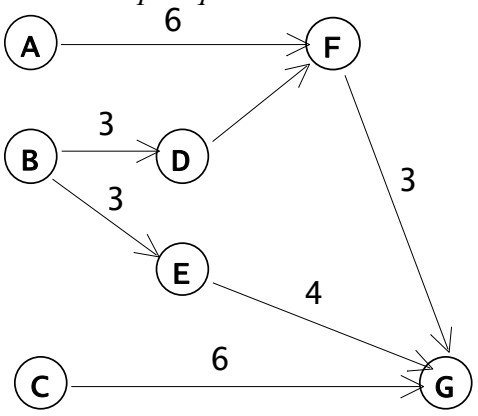
Étape 1 : expression des contraintes :

On commence par représenter dans un graphe orienté toutes les contraintes de précédence : on introduit un arc entre un sommet i et un sommet j si la tâche i doit être terminée avant le début des travaux de la tâche j.



Étape 2 : durée des tâches :

On décore les arcs de ce graphe en valuant les arcs avec la durée de la tâche à l'origine de l'arc :
par exemple: l'arc A→F porte la valeur 6 puisque la tâche A dure 6 (semaines).



D'une façon générale, dans ce qui suit,

- la notation (i,j) désignera l'arc i→j.
- la notation v_{ij} désignera la valeur de l'arc (i,j).

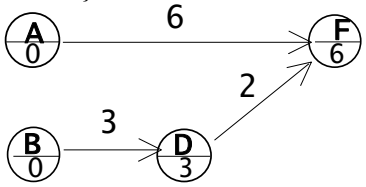
Ce graphe permet de calculer la date au plus tôt du début de chaque tâche :

par exemple :

- la tâche A, qui n'a pas de prédécesseur dans le graphe, peut commencer à l'instant 0.
- la tâche D, qui doit attendre que B soit terminée, ne peut commencer qu'en semaine 3 puisque B commence en semaine 0 et dure 3 semaine: 3= 0 + 3;
- la tâche F, doit attendre que A et D soient terminées
    - A commence en semaine 0 et dure 6 semaines: fin des travaux de A: 0 +6 = 6
    - D commence en semaine 3 et dure 2 semaines: fin des travaux de D: 3 + 2 = 5

Ainsi, les travaux de F ne peuvent commencer qu'en semaine 6.

On représente souvent cette situation de la façon suivante :



D'une façon générale, la notation $dtot_k$ désignera la date de début des travaux de la tâche k. On constate que :

$dtot_k$ = Max($dtot_i$+v_{ik}), où i est une tâche qui précède k

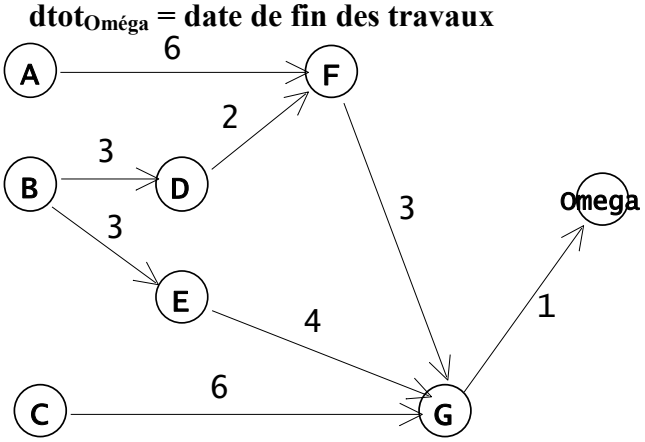
Étape 3 : date de fin des travaux : tâche Oméga

On constate que le graphe précédent ne permet pas de calculer la date de la fin des travaux : en effet, on connaît la date de début de la tâche G (=9), mais on ne connaît pas la date de la fin des travaux de G.

Pour uniformiser l'ensemble de la méthode, on crée une tâche fictive (toujours appelée Oméga).

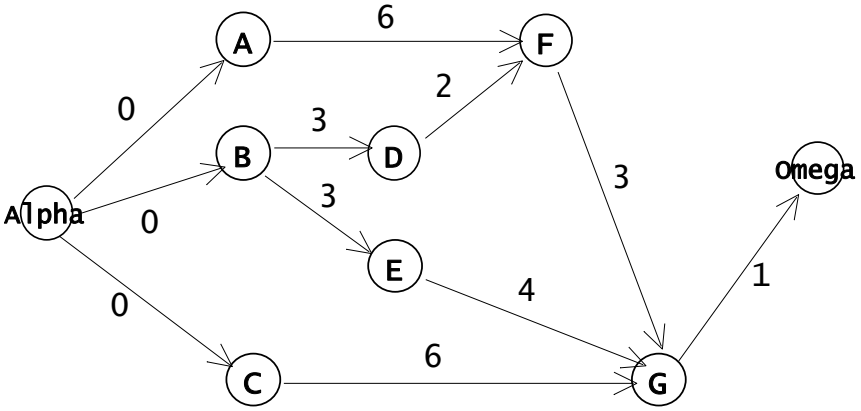
- Oméga indique la fin des travaux.
- Oméga a pour prédécesseur toutes les tâches qui n'ont pas de successeur ! (ici, seule G est concernée).

Il suffit ensuite d'appliquer la même méthode (*générale*, pour cette tâche comme pour les autres) : la date de fin des travaux est égale à la date de début des travaux de Oméga :



Étape 4 : par souci d'uniformisation : tâche Alpha

Par symétrie avec la tâche Oméga, on crée une tâche Alpha qui indique le début des travaux (t_{Alpha}=0). Cette tâche de durée 0 précède toutes les tâches qui n'ont pas de prédécesseur dans le graphe précédent*: (ici, il s'agit de A, B, et C)*



Bilan : Calcul de la date au plus tôt de chacune des tâches et la date de la fin des travaux :

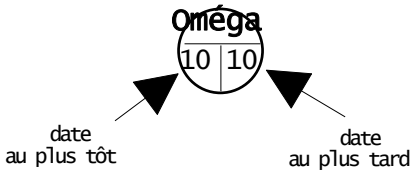
- *Effectuer un* ***marquage topologique*** *du graphe (y compris les sommets Alpha et Oméga).*
- *La date au plus tôt d'Alpha est égale à 0*
- *Prendre l'un après l'autre chaque sommet du graphe* ***dans l'ordre topologique*** *et calculer sa date au plus tôt selon la formule :* **$dtot_k = Max(dtot_i+v_{ik})$, où i est une tâche qui précède k**

Deuxième partie : date au plus tard, marge et tâches critiques

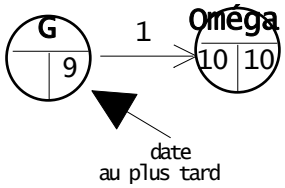
La date au plus tard d'une tâche est la date où une tâche peut démarrer (avec éventuellement un certain retard – qu'on détermine ici) **sans retarder la date de fin des travaux**.

En menant la méthode jusqu'à terme dans l'exemple précédent, on s'aperçoit que la date au plus tôt du début de la tâche Oméga aura lieu la semaine 10 : 10 correspond donc à la durée totale des travaux.

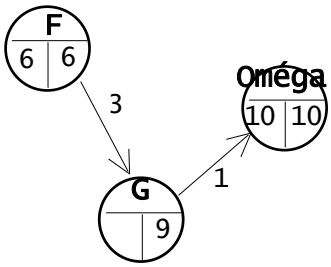
Si on veut que l'ensemble des travaux ne prenne aucun retard, il faut que la tâche Oméga se termine **au plus tard** la semaine 10 (qui est la fin supposée des travaux). Ce que l'on représente de la façon suivante :



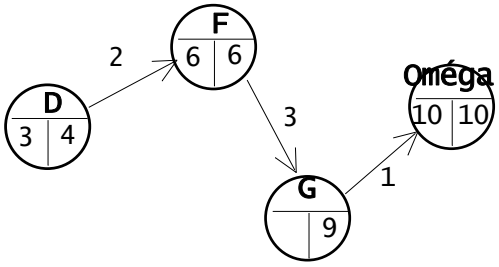
La tâche G durant 1 semaine, elle ne peut pas commencer après la semaine 10 (si on souhaite que la durée totale des travaux soit de 10 semaines). On complète le diagramme de la façon suivante :



De même pour F :

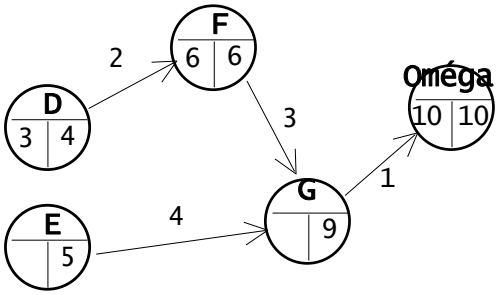


La tâche D dure 2 semaines, elle peut donc commencer en semaine 4 (c'est à dire 6 – 2), sans remettre en cause la fin de la tâche F.

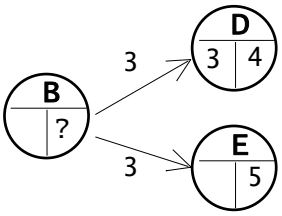


On voit ici que la tâche D dispose d'une « **marge** » de 1 semaine : la tâche D peut commencer avec une semaine de retard sans remettre en cause la durée globale des travaux. Par contre, la tâche F ne peut prendre aucun retard, on dit qu'elle est « **critique** ».

Pour la tâche E, la date au plus tard est 5.



De la tâche B, dépendent les travaux de D et de E.



Pour que la tâche D commence au plus tard la semaine 4, il faut que la tâche B commence au plus tard la semaine (4 – 3) c'est à dire la semaine 1.

Pour que la tâche E commence au plus tard la semaine 5, il faut que la tâche B commence au plus tard la semaine (5 – 3) c'est à dire la semaine 2.

La première condition est la plus contraignante : **Min [(4 – 3), (5 – 3)] = (4 – 3)**. Il faut donc que la tâche B commence au plus tard la semaine 1.

D'une façon générale, la notation $dtard_k$ désignera la date au plus tard de la tâche k. On constate que :

$$\mathbf{dtard_k = Min(dtard_i - v_{ki})}\text{, où i est une tâche qui suit k}$$

Bilan : Calcul de la date au plus tard de chacune des tâches :

- *La date au plus tard d'Oméga est égale à sa date au plus tôt.*
- *Prendre l'un après l'autre chaque sommet du graphe* ***dans l'ordre topologique inverse*** *et calculer sa date au plus tard selon la formule :* $\mathbf{dtard_k = Min(dtard_i - v_{ki})}$**, où i est une tâche qui suit k**

Travail à réaliser

Programmer la méthode MPM sur le graphe de l'exemple, ainsi que sur un autre graphe de votre choix. En sortie, votre programme doit fournir les dates au plus tôt, dates au plus tard et marges de toutes les tâches, ainsi que la liste des tâches critiques et bien évidemment, la date de fin des travaux.

Vous utiliserez la représentation `L_ADJ`. Notez que le graphe est *non valué* car la durée des tâches est associée aux sommets et pas aux arcs. Notez également que vous avez besoin ***des listes de successeurs et des listes de prédécesseurs***.

Vous avez *libre choix* du format du fichier. Notez qu'à partir de ce fichier (appelons-le `mpm.txt`), vous devez :

1. réaliser le marquage topologique suivant la méthode du TP précédent.
2. mettre en œuvre la méthode MPM.

FACULTATIF

Il est beaucoup plus « professionnel » que les tâches fictives Alpha et Oméga soient ajoutées automatiquement dans le graphe. L'utilisateur du programme n'a en effet pas à prendre en charge les artifices utilisés dans les algorithmes (cf. exemples 2 & 3).

Exemple 1 :

```
9 11
1 6
6 7
7 8
2 4
2 5
4 6
5 7
3 7
0 1
0 2
0 3
1 6 A
2 3 B
3 6 C
4 2 D
5 4 E
6 3 F
7 1 G
0 0 Alpha
8 0 Oméga
# On a choisi de mettre des commentaires à la fin du fichier pour ne pas avoir à
# les traiter lors de la lecture des données.
# Format choisi : sur la première ligne, nombre de sommes, nombres d'arcs
# puis les 11 arcs traduisant les précédences, fournis dans n'importe quel
# ordre : origine, extrémité
# enfin les infos associées aux 9 sommets :
# numéro du sommet durée de la tâche, nom de la tâche
```

Exemple 2 :

```
# nom du fichier : btp.txt
# Un chantier de construction d'un bâtiment
9 sommets représentant les tâches
1 1 pose_portes_et_fenêtres_extérieur
2 2 pose_tuiles
3 3 montage_mur
4 1 fondation
5 1 carrelage
6 3 électricité
7 2 charpente
8 2 plomberie
```

```
9 2 cloisons_et_plâtre
13 arcs traduisant les précédences
4 3
3 7
3 1
7 2
1 8
1 6
2 8
2 6
6 9
6 5
5 9
8 5
8 9
```

Exemple 3 :

```
# nom du fichier : recette.txt
# La recette des pâtes à la Carbonara
14 sommets représentant les tâches
1 20 Acheter les ingrédients
2 4 Porter l'eau à ébullition
3 2 Couper le jambon en dés
4 2 Eplucher l'ail
5 4 Faire chauffer la poêle
6 2 Emincer l'ail
7 8 Mettre les pâtes à cuire
8 1 Faire fondre le beurre
9 1 Egoutter les pâtes
10 5 Ajouter le jambon et l'ail
11 1 Poser un jaune d'œuf sur les pâtes
12 2 Ajouter la crème
13 1 Mélanger le tout
14 1 Servir aussitôt
16 arcs traduisant les précédences
1 2
1 3
1 4
1 5
2 7
7 9
9 11
11 13
3 10
4 6
6 10
5 8
8 10
10 12
12 13
13 14
```