

Devoirs Surveillés Ecrits - Janvier 2010 – L3 et L'3
Eléments de correction

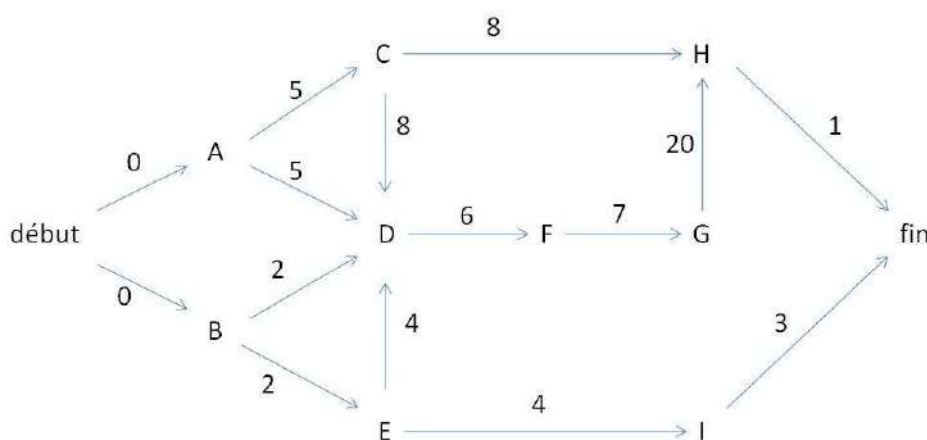
L3 -- Ordonnancement

Construisez (dessinez) le graphe, puis calculez les calendriers au plus tôt et au plus tard, ainsi que les marges libre et totale, pour le projet suivant :

Tâches	A	B	C	D	E	F	G	H	I
Durées	5	2	8	6	4	7	20	1	3
Contraintes	(aucune)	(aucune)	A	A B C E	B	D	F	C G	E

Les contraintes sont de type « fin-début » comme étudié en TD, c'est-à-dire que, par exemple, il faut que la tâche 'A' soit terminée pour que la tâche 'C' puisse commencer.

Pour le calcul du calendrier au plus tard, vous considérerez que la date au plus tard de la fin de projet est égale à sa date au plus tôt.

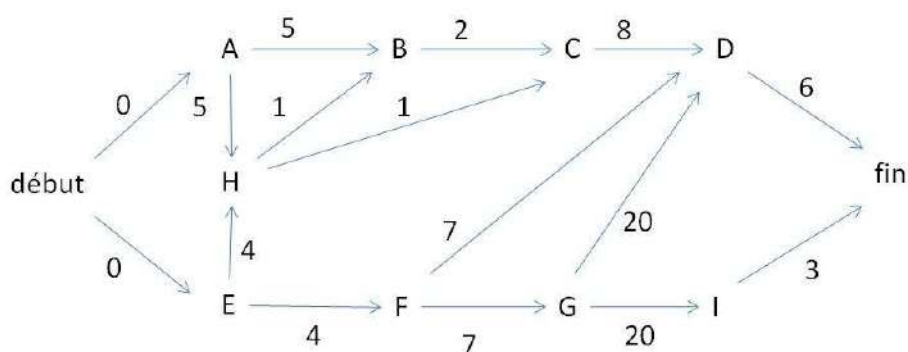


Tâches	début	A	B	C	D	E	F	G	H	I	fin
Dates au plus tôt	0	0	0	5	13	2	19	26	46	6	47
Dates au plus tard	0	0	7	5	13	9	19	26	46	44	47
Marges totales	0	0	7	0	0	7	0	0	0	38	0
Marges libres	0	0	0	0	0	0	0	0	0	38	/

L'3 -- Ordonnancement

[...énoncé similaire...]

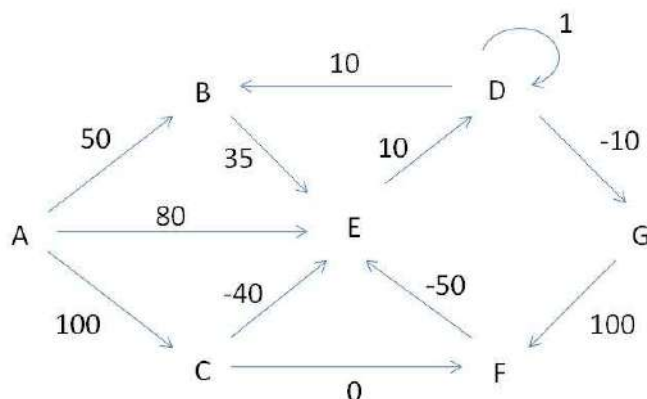
Tâches	A	B	C	D	E	F	G	H	I
Durées	5	2	8	6	4	7	20	1	3
Contraintes	(aucune)	A H	B H	C F G	(aucune)	E	F	A E	G



Tâches	début	A	B	C	D	E	F	G	H	I	fin
Dates au plus tôt	0	0	6	8	31	0	4	11	5	31	37
Dates au plus tard	0	15	21	23	31	0	4	11	20	34	37
Marges totales	0	15	15	15	0	0	0	0	15	3	0
Marges libres	0	0	0	15	0	0	0	0	0	3	/

L3 -- Recherche des chemins les plus courts – Algorithme de Bellman

Déroulez l'algorithme de Bellman sur le graphe ci-dessous, pour le calcul des chemins les plus courts partant de 'A' :



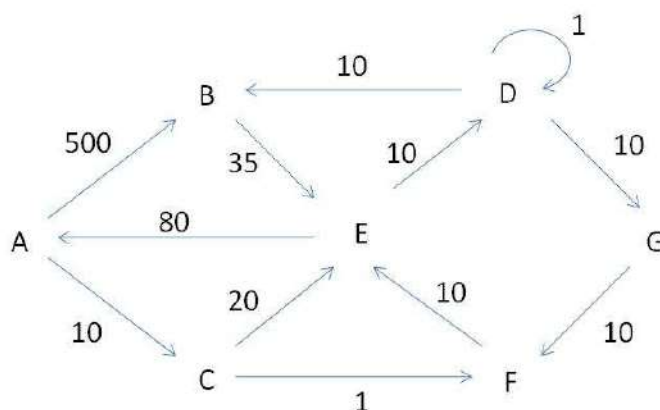
Remarque : seule la longueur des chemins vous est demandée.

Itération	Valeurs des plus courts chemins (π)						
K =	A	B	C	D	E	F	G
0 (initialisation)	0	∞	∞	∞	∞	∞	∞
1	0	50	100	∞	80	∞	∞
2	0	50	100	90	60	100	∞
3	0	50	100	70	50	100	80
4	0	50	100	60	50	100	60
5	0	50	100	60	50	100	50
6	0	50	100	60	50	100	50

Stabilité pour k=6

L'3 -- Recherche des chemins les plus courts – Algorithme de Dijkstra

Déroulez l'algorithme de Dijkstra sur le graphe ci-dessous, pour le calcul des chemins les plus courts partant de 'A' :



CC	A		B		C		D		E		F		G	
	π	Préd.	π	Préd.	π	Préd.	π	Préd.	π	Préd.	π	Préd.	π	Préd.
Init	0	0	500	A	10	A	∞		∞		∞		∞	
C	0	0	500	A	10	A	∞		30	C	11	C	∞	
F	0	0	500	A	10	A	∞		21	F	11	C	∞	
E	0	0	500	A	10	A	31	E	21	F	11	C	∞	
D	0	0	41	D	10	A	31	E	21	F	11	C	41	D
B/G	0	0	41	D	10	A	31	E	21	F	11	C	41	D
G/B	0	0	41	D	10	A	31	E	21	F	11	C	41	D

La colonne « CC » indique les sommets choisis et passés de « M » dans « CC » à chaque itération.

« B/G » signifie « B ou G » ; l'itération suivante prenant bien entendu l'autre des deux sommets.

L3 -- Algorithme

Ecrivez l'algorithme (pseudo-code) d'une fonction `int rang ( int pt_entree , int sommet )` qui calcule le rang d'un sommet 'sommet' en partant d'un point d'entrée 'pt_entree'.

Idée générale de calcul des rangs :

On fixe le rang du point de départ à 0. On calcule ensuite itérativement les rangs des autres sommets en appliquant le principe suivant : un sommet dont tous les prédécesseurs ont déjà un rang calculé peut se voir donner le rang max de ses prédécesseurs +1.

Pour l'exercice, on s'arrête dès que le rang de 'sommet' est trouvé, puisqu'on ne demande pas tous les rangs. Note : calculer tous les rangs ne donne pas un résultat faux, mais c'est simplement inutile.

Attention à la présence de circuits qui rend le calcul des rangs impossible (voir booléen modif).

```
int rang[] ← -1      rang des sommets, initialisé à '-1' (rang non calculé)
rang[pt_entree] ← 0  rang du sommet de départ de calcul du rang
```

```
int pred_sans_rang [] ← degre intérieur
                        Pour chaque sommet : nombre de prédécesseurs pour lesquels
                        le rang n'a pas encore été calculé
```

```
Initialisé au nombre de prédécesseurs
pour tout sommet S successeur de pt_entree faire
    pred_sans_rang[S] -- on décompte pt_entree des prédécesseurs sans rang
```

```
fin pour

int rang_courant ← 1    Prochaine valeur de rang à donner à un sommet
bool modif ← vrai      indique si une affectation de rang a été faite à l'itération précédente

tant que rang[sommet] = -1  on n'a pas le rang recherché
    et modif = vrai
faire
    modif ← faux
    pour tout sommet S faire
        si rang[S] = -1          rang non calculé
        et pred_sans_rang[S] = 0  tous les prédécesseurs ont un rang calculé
        alors
            rang[S] ← rang_courant
            modif ← vrai
            pour tout sommet Succ successeur de S faire
                pred_sans_rang[Succ] --
                On « supprime » S du nombre des prédécesseurs sans rang
            fin pour tout
        fin si
    fin pour tout
    Si on a « modif=faux » ici, alors aucun rang n'a été affecté lors de cette itération. Il y a donc
    un circuit et le calcul est impossible
    rang_courant ++
fin tant que
retourner rang[sommet]
    retourne le rang de sommet
    ou -1 si sommet n'est pas accessible depuis pt_entree
```

L'3 -- Algorithme

Ecrire l'algorithme (pseudo-code / méta-code) d'une fonction

void aller_retour (graphe G ; int S)

qui imprime l'ensemble des sommets X tels qu'il existe un chemin de S à X et un chemin de X à S.

G est bien entendu un graphe orienté.

Il s'agit ici de « chemins », et non pas d'arcs !

Une idée parmi d'autres :

- chercher l'ensemble des sommets X tels qu'il existe un chemin de S à X : les successeurs directs et indirects de S (SuccDirIndir) ;
- chercher l'ensemble des sommets X tels qu'il existe un chemin de X à S : les prédécesseurs directs et indirects de S (PredDirIndir) ;
- retourner l'intersection des deux ensembles.

Pour le calcul des ensembles, on utilise la technique du marquage vue en TD (exercice « bouteilles »). L'ensemble « à_traiter » contient les sommets non marqués ; un sommet traité et donc marqué est transféré dans PredDirIndir ou SuccDirIndir.

```
ensemble de sommets PredDirIndir ← vide
ensemble de sommets SuccDirIndir ← vide
```

Calcul des prédécesseurs directs et indirects

ensemble de sommets à_traiter $\leftarrow$ prédécesseurs directs de S

tant que à_traiter $\neq$ vide il existe des sommets « non marqués »

faire

sommet PDI $\leftarrow$ choix dans à_traiter en prendre un au hasard

PrédDirIndir $\leftarrow$ PrédDirIndir + PDI et le marquer

à_traiter $\leftarrow$ à_traiter - PDI

pour tout sommet Pred prédécesseur de PDI faire

si Pred $\notin$ à_traiter et Pred $\notin$ PrédDirIndir alors

à_traiter $\leftarrow$ à_traiter + Pred

fin si

fin pour

fin tant que

Calcul des successeurs directs et indirects

...similaire au code précédent

Impression du résultat

pour tout sommet som faire

si som $\in$ PrédDirIndir et som $\in$ SuccDirIndir alors

imprimer som

fin si

fin pour

L3 -- Modélisation

Vous voyagez en utilisant un véhicule électrique. Son autonomie est limitée, et vous la connaissez à chaque instant. Le réseau de transport est constitué d'endroits reliés par des voies de communication. La quantité d'énergie qui vous est nécessaire pour un « aller » est généralement différente de celle nécessaire pour le « retour ». Vous connaissez ces quantités.

Il s'agit donc ici de représenter le réseau de transport au moyen d'un graphe orienté et valué. Les sommets sont les « endroits » ; les arcs représentent les déplacements possibles d'un « endroit » à un autre ; les valeurs sont les quantités d'énergies dépensées.

Seuls certains de ces endroits (vous les connaissez, heureusement pour vous) disposent d'un mécanisme permettant de recharger la batterie de votre véhicule.

Il y a donc des sommets particuliers que vous connaissez.

A chaque étape, au moment de quitter l'endroit où vous êtes et reprendre votre chemin, vous voulez déterminer vers quels autres endroits vous pouvez ou ne pouvez pas vous diriger afin d'éviter la panne sèche.

Il s'agit donc, depuis l'endroit où l'on est, de déterminer la liste des endroits voisins vers lesquels on peut se diriger sans risque de tomber plus tard en panne d'énergie. Il ne s'agit pas de calculer un trajet complet jusqu'à une prochaine « station de recharge »...

Il faut donc, lorsqu'on décide de se déplacer vers un endroit donné, d'une part vérifier qu'on peut y aller en fonction de l'énergie dont on dispose, mais aussi s'assurer qu'à partir de cet endroit, il

existe un chemin menant à une « station de recharge » et dont la longueur est inférieure ou égale à la puissance d'énergie qui sera alors disponible.

Comment faire ?... en utilisant la théorie des graphes bien entendu !

On dispose donc :

- d'un graphe orienté et valué représentant le réseau de transport ;
- de la valeur E de l'énergie disponible à un instant donné ;
- du sommet ICI correspondant à l'endroit où l'on est ;
- de la liste des sommets 'SRi' correspondant aux endroits où une station de recharge est disponible.

Première approche :

On calcule les chemins les plus courts partant de ICI. On retient ceux menant aux sommets SRi dont la longueur est inférieure ou égale à E : ce sont les stations de recharge que l'on peut atteindre. Les successeurs de ICI dans ces chemins sont les solutions au problème.

Après chaque déplacement vers un lieu voisin, on diminue la valeur de E de celle de l'arc emprunté, et on recommence le calcul.

Lorsqu'on arrive à un des sommets SRi, on réinitialise E à sa valeur maximale.

Cette première méthode risque cependant de limiter les choix possibles car les algorithmes ne rendront qu'un seul « plus court chemin » pour chaque SRi, donc élimineront peut être des choix possibles quelques endroits voisins de celui où vous êtes. Ceci est dû au fait qu'entre le sommet ICI et un sommet SRi, il peut exister plusieurs chemins de longueur inférieure ou égale à E (et donc acceptables pour notre problème), mais le calcul ne donnera que le plus court d'entre eux.

Optimisation :

Nommons ACOTEj les successeurs de ICI tels que la valeur de l'arc (ICI, ACOTEj) est inférieure ou égale à E : ACOTEj sont les voisins accessibles en fonction de l'énergie disponible.

Si ACOTEj est un sommet où une recharge d'énergie est possible, alors c'est une solution au problème.

Dans le cas contraire :

On pose $E_j = E$ moins la valeur de l'arc (ICI, ACOTEj).

On calcule les chemins les plus courts partant de ACOTEj.

S'il existe un sommet SRi tel que la valeur du chemin le plus court de ACOTEj à SRi est inférieure ou égale à E_j , alors ACOTEj est une solution au problème ; sinon, ce n'est pas un endroit où il faut aller.

L'3 -- Modélisation

Vous disposez un réseau de communication composé de routeurs interconnectés ($R_1, R_2, \dots$). Vous connaissez les capacités (bande passante maximale) de chaque lien.

Autour de ce réseau, vous installez un certain nombre de points d'accès ($PE_1, PE_2, \dots$), chacun relié à un ou plusieurs routeurs. Vous connaissez aussi les capacités de ces liens point d'accès / routeur.

On suppose ici que sur chaque lien, les données peuvent circuler dans les 2 sens. De plus, il n'y a pas de bande passante maximale « montante » et « descendante » : la même est disponible dans les deux sens.

On peut donc imaginer un graphe non orienté, valué avec les bandes passantes maximales, dont les sommets sont les routeurs R_i et les points d'accès PE_j .

Vos clients sont de grandes entreprises disposant de plusieurs sites. Chaque client relie chacun de ses sites à un point d'accès.

Vous leur vendez des connexions de bout en bout d'un site à un autre. Un de vos clients va par exemple vouloir relier son site de Marseille avec son site de Brest, avec une bande passante minimale fixée en fonction de ses besoins.

Lorsqu'un client passe commande d'une connexion, vous devez vous assurer que la bande passante exigée est bien disponible, puis vous configurez votre réseau (suite des routeurs à traverser) de façon à satisfaire sa demande.

Comment pouvez-vous vous aider d'un graphe pour faire cela ?

Le problème est donc de configurer un routage au travers du réseau qui satisfasse cette bande passante, c'est-à-dire une chaîne sur laquelle toutes les arêtes (connexions PE/R ou R/R) disposent de cette bande passante. Il est donc nécessaire de disposer de la bande passante disponible (celle pas encore « vendue » à un client), et pas uniquement la bande passante maximale...

On considère donc finalement le graphe suivant :

Sommets = les R_i et PE_j

Arêtes = les connexions PE/R et R/R avec les bandes passantes disponibles

A l'initialisation du graphe, on associe aux arêtes les bandes passantes maximales.

Soit un client C qui demande une bande passante BP entre deux points d'accès PE-M(arseille) et PE-B(rest).

On cherche une chaîne de PE-A à PE-B pour laquelle toutes les arêtes ont une valeur au moins égale à B. Si une telle chaîne n'existe pas, la demande est refusée.

Pour chaque arête appartenant à la chaîne retenue, on diminue la valeur associée de la bande passante demandée et allouée au client. La bande passante vendue n'est ainsi plus disponible pour les clients futurs.

Lorsqu'un client résilie son contrat, on retrouve la chaîne utilisée et on ajoute B sur toutes les arêtes.

Remarque : Lorsque plusieurs chaînes satisfont une demande client, on peut en théorie prendre celle que l'on veut. Un bon administrateur réseau appliquera cependant une stratégie bien spécifique. Par exemple, on peut choisir la chaîne ayant la bande passante disponible la plus grande afin d'éviter le plus longtemps possible une congestion du réseau. Mais ceci est plus un problème d'administration réseau que de théorie des graphes...

L3 -- Structures de données

Voici deux structures de données permettant de représenter un graphe non valué.

1)

```
int ** MatAdj ;  
    // MatAdj[x][y] = 1 si x et y représentent deux sommets  
    //                  et (x,y) est un arc du graphe  
    // MatAdj[x][y] = 0 sinon
```

2)

```
int ** MatInc ;  
    // MatInc[x][a] = +1 si x represente un sommet,  
    //                               et a represente un arc dont x est  
    //                               l'extremite initiale  
    // MatInc[x][a] = -1 si x represente un sommet,  
    //                               et a represente un arc dont x est  
    //                               l'extremite terminale  
    // MatInc[x][a] = 0 sinon
```

Proposez les modifications nécessaires pour pouvoir représenter un graphe valué, à valeurs entières quelconques.

Attention au cas d'arc à valeur nulle, ainsi qu'au cas d'arc à valeur négative !!!

```
1)  
typedef struct {  
    bool existe ; // 'vrai' si l'arc existe, 'faux' sinon  
    int valeur ; // utilisé uniquement si existe=vrai  
} arc ;  
arc ** MatAdj ;
```

```
2)  
int * Valeurs ; // Valeurs[a] = valeur associée à l'arc 'a'  
int ** MatInc ; // Inchangé : cf énoncé
```

L'3 -- Structures de données

*Voici deux structures de données permettant de représenter un graphe non valué.
[SDD... cf ci-dessus]*

Proposez les modifications nécessaires pour pouvoir représenter un graphe valué pour lequel les valeurs sont toutes positives et non nulles.

Ici, les valeurs sont strictement positives. Les réponses L3 restent valables, mais sont plus compliquées que nécessaire.

Il suffit ici :

- pour la matrice d'adjacente MatAdj de remplacer le '1' par la valeur de l'arc ;
- pour la matrice d'incidence MatInc de remplacer '+1' par la valeur de l'arc, et '-1' par l'inverse de la valeur.

L3 -- Questions de cours / QCM

1. Quelle matrice peut-on utiliser pour le calcul du degré d'un sommet ?
 - a - Matrice d'adjacence SxS
 - b - Matrice d'incidente SxA
 - c - Les deux
 - d - Aucune
2. Un graphe non orienté est un couple $G=(S,A)$ où S définit un ensemble de sommets et A un ensemble de paires de sommets appelées arcs.
 - a - Vrai
 - b - Faux
3. Si x est l'extrémité initiale d'un arc dont y est l'extrémité finale, alors y est le :

- a - prédécesseur de x
 - b - successeur de x
4. Si on note $CFC(a)$ la composante fortement connexe contenant le sommet a dans un graphe, alors l'expression « $CFC(x) \subset CFC(a)$ »
- a - est vrai pour tout sommet x appartenant à $CFC(a)$
 - b - est fausse pour tout sommet x appartenant à $CFC(a)$
 - c - la réponse est un peu plus complexe que cela...
5. Dans un arbre, il peut exister plusieurs chaînes entre deux sommets donnés.
- a - Vrai
 - b - Faux
6. $G'=\{S',A'\}$ est le graphe partiel de $G=\{S,A\}$ engendré par A' si :
- a $S' \subset S$ et $A' \subset A$
 - b $S' \subseteq S$ et $A' \subseteq A$
 - c $S' \subset S$ et $A' \subseteq A$
 - d $S' = S$ et $A' \subset A$
 - e $S' = S$ et $A' \subseteq A$
7. Un point d'entrée peut appartenir à un circuit si un nombre impair d'arcs lui sont incidents
- a- Vrai
 - b- Faux
 - c - Ça dépend...
8. Dans un graphe valué, un circuit absorbant est un circuit de coût :
- a Négatif
 - b Nul
 - c négatif ou nul
9. Si un graphe est sans circuit, alors les valeurs de la diagonale de sa matrice d'adjacence sont :
- a des '1'
 - b des '0'
 - c des '0' ou des '1'
10. Un graphe est dit symétrique si et seulement si :
- a pour tout arc $x \rightarrow y$, il existe un arc $y \rightarrow x$
 - b pour tout chemin de x à y, il existe un chemin de y à x
 - c aucune réponse n'est correcte

L'3 -- QCM

NB : Seules les questions différentes du devoir L3 sont reprises ici.

4. Soit un graphe défini par S l'ensemble de ses sommets et A l'ensemble de ses arcs. Une matrice d'incidence représente une fonction définie entre les ensembles
- a S et A
 - b S et S
 - c A et A
5. Dans un arbre, il peut exister plusieurs chaînes entre deux sommets donnés.
- a Vrai
 - b Faux
 - c cela dépend de la structure de l'arbre

6. Un graphe non orienté est simple si et seulement si il est sans boucle
a Vrai
→b Faux
c on ne peut rien dire car les deux notions sont indépendantes
8. Un point d'entrée peut appartenir à un circuit si un nombre pair d'arcs lui sont incidents
a Vrai
→b Faux
c vrai s'il n'y a pas de boucle parmi les arcs
9. Dans un graphe valué, un circuit absorbant est un circuit de coût :
→a strictement négatif
b Nul
c négatif ou nul
10. Si $G'=\{S',A'\}$ est le graphe partiel de $G=\{S,A\}$ engendré par A' , alors tous les sommets de S' sont une extrémité d'un arc de A'
a Vrai
→b Faux