# Master 2 Informatique I2A

**Master DVL Master ITVL**

- Finance
- Histoire
- Géographie
- Informatique
- Mathématiques
- Sciences Pour l'Ingénieur
- Français Langue Étrangère
- Administration Économique et Sociale
- Diplôme d'accès aux études universitaires

**Master mention Informatique**

Parcours Informatique Avancé et Applications (I2A)

CTU
Centre de Télé-enseignement Universitaire
http://ctu.univ-fcomte.fr

## Filière Informatique

**VVI9MTGC**

Théorie des graphes et combinatoire

**Mr PHILIPPE - LAURENT**
*laurent.philippe@univ-fcomte.fr*

UNIVERSITÉ DE FRANCHE-COMTÉ
UBFC
UNIVERSITÉ BOURGOGNE FRANCHE-COMTÉ

UNIVERSITÉ DE FRANCHE-COMTÉ

Module INFO-TGC

L. Philippe

# Théorie des Graphes et Combinatoire

## Master Informatique Avancée et Applications

**Chapitre I - Graphes et Algorithmes : Exercices**

CTU Centre de Télé-enseignement Universitaire
UNIVERSITÉ DE FRANCHE-COMTÉ

Centre de télé enseignement
Filière Informatique
Domaine Universitaire de la Bouloie
25030 Besançon Cedex (France)

# 1 Modélisation sous forme de graphes

Le but de ces exercices est de proposer une modélisation sous la forme d'un graphe du problème posé avant d'en faire la résolution.

### Exercice 1 : Choix d'un itinéraire

Les durées des trajets entre deux villes sont indiquées comme suit :

| | |
|---|---|
| Bordeaux ↔ Nantes | 4h |
| Bordeaux ↔ Marseille | 9h |
| Bordeaux ↔ Lyon | 12h |
| Nantes ↔ Paris - Montparnasse | 2h |
| Nantes ↔ Lyon | 7h |
| Paris - Montparnasse ↔ Paris - Lyon | 1h (en bus) |
| Paris - Lyon ↔ Grenoble | 4h30 |
| Marseille ↔ Lyon | 2h30 |
| Marseille ↔ Grenoble | 4h30 |
| Lyon ↔ Grenoble | 1h30 |

**Question 1.1** : *Modéliser les distances entre ces villes sous la forme d'un graphe*

**Question 1.2** : *Comment faire pour aller le plus rapidement possible de Bordeaux à Grenoble ? de Marseille à Nantes ? de Lyon à Nantes ?*

### Exercice 2 : Loup, Chèvre et Chou

Dans une énigme bien connue, un passeur se trouve sur le bord d'un fleuve et doit faire passer de l'autre côté de la rive un loup, une chèvre et un chou mais il ne peut transporter qu'un seul client à la fois et ne peut laisser seuls ensemble ni le loup avec la chèvre, ni la chèvre avec le chou puisqu'ils mangerait l'autre.

**Question 2.1** : *Donner les différents états possibles pour le système.*

Dans l'état initial, le batelier, le loup, la chèvre et le choux sont sur la rive de départ.

**Question 2.2** : *Existe-t-il plusieurs solutions pour le passeur ?*

### Exercice 3 : Organisation d'une session d'examen

Les étudiants A, B, C, D, E, F, G et H doivent passer des examens, chacun dans des disciplines différentes données dans la table 1. Chaque examen occupe une demi-journée.

**Question 3.1** : *Compte-tenu de ces contraintes, on cherche à organiser la session d'examens de manière à ce que l'ensemble des examens soit terminé le plus tôt possible.*

### Exercice 4 : Planification de travaux

| Matière | Informatique | Gestion | Droit | Maths | Économie |
|---|---|---|---|---|---|
| Étudiants | A, B | C, D | C, E, F, G | A, E, F, H | B, F, G, H |

TABLE 1 – Matières à passer pour chaque étudiant

Pour rénover une maison, il est prévu de refaire l'installation électrique (3 jours), de réaménager (5 jours), de carreler la salle de bains (2 jours), de faire le parquet dans le séjour (6 jours), de repeindre les chambres (3 jours). La peinture et le carrelage ne devant être faits qu'après l'installation électrique.

**Question 4.1** : *Si le propriétaire décide de tout faire lui-même, dans quel ordre doit il procéder et quelle est la durée minimale des travaux ?*

**Question 4.2** : *Si la rénovation est faite par une entreprise et que chacune des tâches est accomplie par un employé différent, quelle est la durée minimale des travaux ?*

## Correction Exercice 1 : Choix d'un itinéraire

**Solution question 1.1 :** *Modéliser les distances entre ces villes sous la forme d'un graphe*

La figure 1 modélise les distances entre les villes sous la forme d'un graphe dont les arêtes sont valuées par la distance.

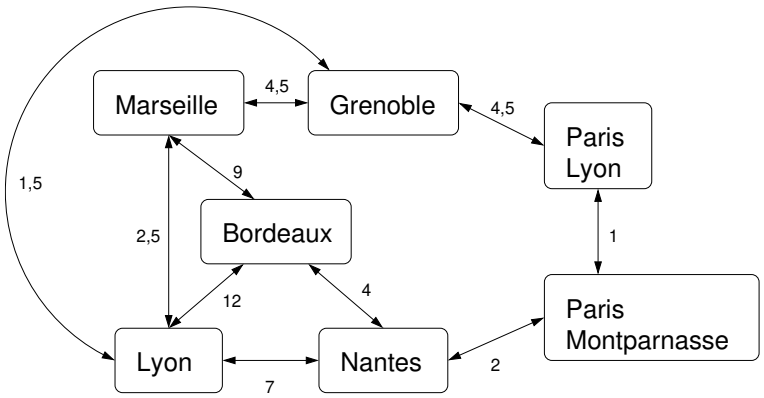



FIGURE 1 – Graphe des villes avec les distances entre elles

**Solution question 1.2 :** *Comment faire pour aller le plus rapidement possible de Bordeaux à Grenoble ? de Marseille à Nantes ? de Lyon à Nantes ?*

Pour aller le plus rapidement d'une ville à une autre, il s'agit de trouver le plus court chemin dans ce graphe pour un départ et une destination en explorant les chemins possibles :

Bordeaux → Grenoble :
- Bordeaux - Marseille - Grenoble : 13,5
- **Bordeaux - Nantes - Paris Montp. - Paris Lyon - Grenoble : 11,5**
- Bordeaux - Lyon - Grenoble : 13,5
- Bordeaux - Marseille - Lyon - Grenoble : 13
- Bordeaux - Nantes - Lyon - Grenoble : 12,5

Marseille → Nantes :
- **Marseille - Lyon - Nantes : 9,5**

Lyon → Nantes :
- **Lyon - Nantes : 7**

## Correction Exercice 2 : Loup, Chèvre et Chou

**Solution question 2.1 :** *Donner les différents états possibles pour le système.*

Il est possible d'identifier l'ensemble des états accessibles (ou états possibles) depuis l'état initial. La solution passe par un ensemble d'états intermédiaires avant d'atteindre l'état final. La liste des états est donnée par la figure 2.

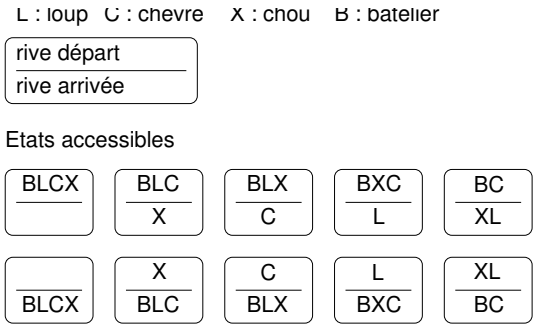



FIGURE 2 – liste des états accessibles depuis la position initiale

**Solution question 2.2** : *Existe-t-il plusieurs solutions pour le passeur ?*

On réalise le graphe suivant : les états possibles sont les sommets du graphe et les arêtes sont les transitions possibles d'un état possible à un autre état possible. Dans ce cas la solution est un des chemins qui permet de passer de l'état initial à l'état final. La figure 3 illustre ce graphe et donc les solutions possibles pour le batelier s'il veut faire passer la riverez au loup, à la chèvre et au chou. Sur ce graphe deux chemins différents sont clairement visibles : il existe donc bien plusieurs solutions pour le batelier : une fois la chèvre sur l'autre rive il peut transporter soit le choux soit le loup, qu'il laissera sur la rive en ramenant la chèvre.

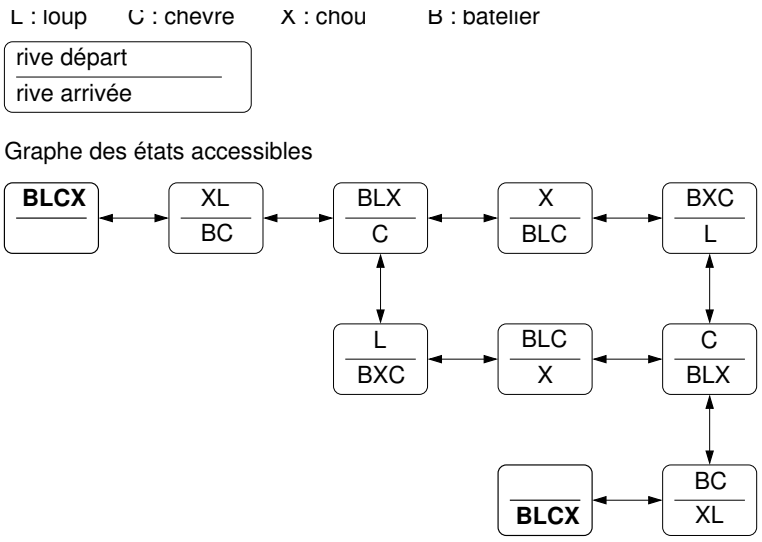



FIGURE 3 – liste des états accessibles depuis la position initiale

## Correction Exercice 3 : Organisation d'une session d'examen

**Solution question 3.1** : *Session d'examens la plus courte possible*

| | A | B | C | D | E | F | G | H |
|---|---|---|---|---|---|---|---|---|
| Informatique | X | X | | | | | | |
| Gestion | | | X | X | | | | |
| Droit | | | X | | X | X | X | |
| Maths | X | | | | X | X | | X |
| Économie | | X | | | | X | X | X |

TABLE 2 – Table des matières à passer par étudiant

Pour commencer nous construisons la table des matières à passer pour chacun des étudiants. Dans cette table nous pouvons remarquer tout d'abord que certaines matières n'ont pas de dépendance entre elles :

— Droit et Informatique ;
— Gestion et Maths ;
— Gestion et Économie ;
— Gestion et Informatique.

Nous pouvons voir ce problème sous un autre angle ; Il s'agit en fait d'un problème de coloration de graphe non orienté. Les sommets de ce graphe sont les matières que les étudiants doivent éventuellement repasser et les arêtes sont les relations entre les matières qui ont en commun des étudiants. Plus formellement on a le graphe $G(X, E)$ avec :

$$X = \{\text{matières de la session d'examen}\}$$

$$E = \{(x, y) \text{ avec } x \in X \text{et} y \in X | \text{certains étudiants repassent } x \text{ et } y\}$$

Le problème de coloration est qu'on ne peut pas donner la même couleur à deux sommets voisins. Il s'agit de faire une coloration valide, mais en plus de trouver le nombre minimum de couleur pour cela. En associant une couleur à une demi-journée d'examen nous pouvons voir que le résoudre le problème de la coloration nous permet bien de trouver le nombre minimal de demi-journées nécessaires et donc d'avoir le session d'examen la plus courte possible.

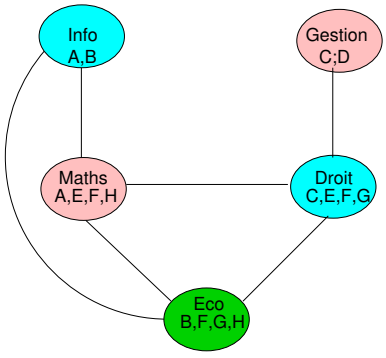



FIGURE 4 – Graphe modélisant l'organisation des examens

La figure 4 illustre une coloration possible du graphe $G$. Arbitrairement on peut en effet choisir la coloration : (Bleu ; Informatique et Droit), (Rose ; Gestion et Maths) et (Vert ; Économie).

En fait il y a 3! possibilités de coloration, toutes ne sont pas valides, et il suffit ici d'en choisir une. La solution donnée précédemment est optimale. La durée minimum de la session d'examen est de 3 demi-journées.

## Correction Exercice 4 : Planification de travaux

Pour répondre à cet exercice, on construit le graphe avec les contraintes de précédence : les tâches sont les sommets du graphe et les arcs sont les contraintes de précédence. Les valeurs des tâches sont associées aux sommets, mais pourrait aussi l'être aux arcs. Il est possible d'ajouter une tâche de début et une tâche de fin pour rendre le graphe connexe, comme sur la figure 5.

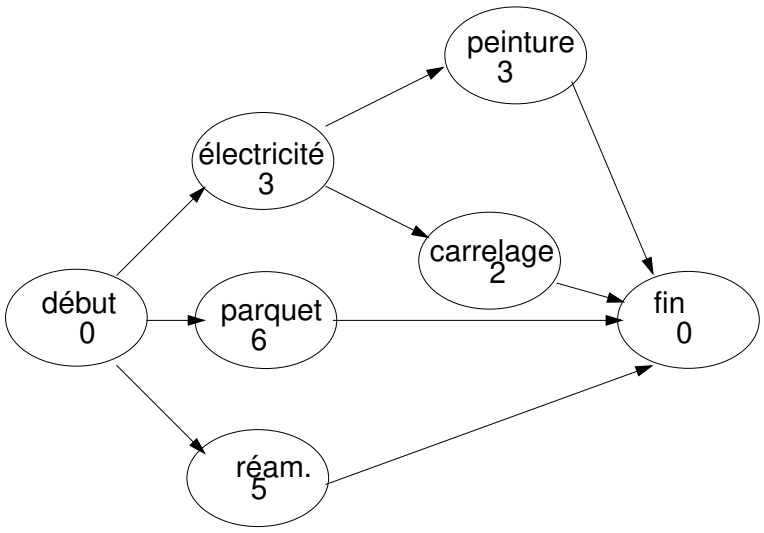



FIGURE 5 – Graphe des tâches à planifier

**Solution question 4.1** : *Si le propriétaire décide de tout faire lui-même, dans quel ordre doit il procéder et quelle est la durée minimale des travaux ?*

Si le propriétaire décide de tout faire lui-même, la seule contrainte à respecter est de faire l'électricité avant la peinture et le carrelage. Comme il ne peut faire qu'une seule chose à la fois la durée minimale des travaux est la somme des tâches, comme montré à la figure 6 soit 19 jours.


FIGURE 6 – Graphe des tâches exécutée seuls par le propriétaire

**Solution question 4.2** : *Si la rénovation est faite par une entreprise et que chacune des*

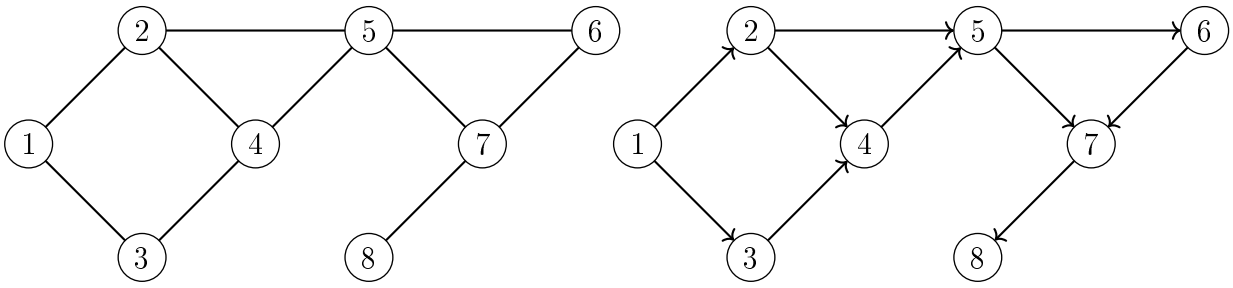
*tâches est accomplie par un employé différent, quelle est la durée minimale des travaux ?*

Il est possible, sur le graphe donné à la figure 5, de calculer la durée minimale du chantier en calculant la longueur de tous les chemins possibles et en prenant le plus grand. Ici nous voyons qu'ils y a deux chemins de longueur 5 et deux chemins de longueur 6. Le chantier durera donc au minimum 6 jours.

La suite des exercices de cette partie concerne la conception d'algorithmes sur les graphes. Pour comprendre les algorithmes qui sont donnés ou valider ceux que vous avez conçus, il est recommandé de les dérouler sur des exemples. Cela peut se faire à la main, sur une feuille de papier, ou en programmant puis en testant sur un exemple. Pour le déroulement manuel, vous pouvez vous inspirez des figures du cours illustrant le parcours en largeur (figure 1.10) ou la recherche d'un plus cou r chemin (figure 1.13). Si vous souhaitez programmer, vous pouvez utiliser n'importe quel langage, avec les structures de données décrites dans le cours. A noter qu'il existe des bibliothèques implémentant les structures et fonctions de base sur les graphes. Je vous recommande la bibliothèque JGraphT[1] qui implémente les graphes sous la forme d'objets et propose la plupart des fonctions de manipulation des sommets et des arêtes. Quatre exemples simples - création d'un graphe, recherche en profondeur, import/export d'un graphe au format DOT et affichage - sont donnés en accompagnement pour vous faciliter de démarrage.

# 2 Manipulation des structures de graphes

Le but de ces exercices est de se familiariser avec la manipulation les graphes et leurs structures de données, passer d'une structure à une autre.




(a) Graphe $G$ non orienté (b) Graphe $G'$ orienté

FIGURE 7 – Graphes non orienté et orienté

**Exercice 5 : matrice d'adjacence vers** $FS/APS$

Soit un graphe orienté $G$ défini par sa matrice d'adjacence $A$.

**Question 5.1 :** *Écrire un algorithme qui permet de construire la file des successeurs $FS$ et le tableau des adresses des premiers successeurs $APS$ du graphe $G$.*

**Question 5.2 :** *Comment modifier l'algorithme pour qu'il fonctionne pour un graphe non orienté ?*

**Question 5.3 :** *Construire les matrices d'adjacence et dérouler l'algorithme sur les graphes $G$ et $G'$ donnés à la figure 7.*

1. `https://jgrapht.org/`

## Exercice 6 : $FS/APS$ vers matrice d'adjacence

Soit un graphe orienté $G$ défini par sa file des successeurs $FS$ et son tableau des adresses des premiers successeurs $APS$.

**Question 6.1 :** *Écrire un algorithme construisant la matrice d'adjacence $A$ du graphe $G$.*

**Question 6.2 :** *Comment modifier l'algorithme pour qu'il fonctionne pour un graphe non orienté ?*

**Question 6.3 :** *Construire les structures $APS$ et $FS$ et dérouler l'algorithme sur les graphes $G$ et $G'$ donnés à la figure 7.*

## Exercice 7 : construction de $G^{-1}$

À partir d'un graphe orienté $G$, on construit le graphe inverse $G^{-1}$ en inversant le sens de tous les arcs orientés de $G$. Nous disposons des tables $FS$ et $APS$ du graphe $G$.

**Question 7.1 :** *Écrire un algorithme, linéaire en nombre de sommets, qui calcule le tableau des prédécesseurs $FP$ (qui est aussi l'ensemble des suivants dans $G^{-1}$) et le tableau des adresses des premiers prédécesseurs $APP$ (qui est le tableau des adresses des premiers suivants dans $G^{-1}$).*

Il est conseillé de procéder en déterminant d'abord le nombre de prédécesseurs de chaque sommet en faisant un parcours linéaire de $FS$, puis en remplissant $APP$ et $FP$.

## Exercice 8 : Calcul de la fermeture transitive d'un graphe

Soit un graphe orienté $G$ défini par sa matrice d'adjacence $A$.

**Question 8.1 :** *Écrire un algorithme permettant de calculer la fermeture transitive du graphe $G$.*

**Question 8.2 :** *Comment modifier l'algorithme pour qu'il fonctionne pour un graphe non orienté ?*

**Correction Exercice 5** :

**Solution question 5.1** : *Écrire un algorithme qui permet de construire la file des successeurs $FS$ et le tableau des adresses des premiers successeurs $APS$ du graphe $G$.*

On rappelle que les structures $APS$ et $FS$ telles que définies dans le cours supposent que les sommets sont numérotés de 1 à $N$ puisque la valeur 0 sert de séparateur dans $FS$. La numéro de ligne $i$ dans la matrice $A$ correspond donc au sommet $i+1$.

L'algorithme 1 permet de créer les structures $APS$ et $FS$ d'un graphe $G$ à partir de la matrice d'adjacence $A$. Le principe est de parcourir la matrice ligne par ligne (lignes 2 à 9 de l'algorithme), sur chaque ligne (lignes 4 à 7 de l'algorithme) les successeurs du sommet sont les sommets adjacents, donc pour lesquels l'élément de la matrice vaut 1 (ligne 5). Ces successeurs sont ajoutés à la structure $FS$ (ligne 6) et l'indice de parcours *indiceCourant* augmenté. À la fin de la ligne du sommet courant (ligne 9), un séparateur est ajouté dans $FS$ puisqu'on passe au sommet suivant.

**Algorithme 1 :** Matrice d'adjacence $\rightarrow FS/APS$

**Données** : $N, M$ : nombres de sommets et d'arêtes
$A$ : matrice d'adjacence
*indiceCourant* : indice de parcours de $FS$ initialement à 0
**Résultat** : $FS[0..N+M-1]$ : tableau des successeurs des sommets
$APS[0..N-1]$ : tableau des adresses des premiers successeurs

```
1 début
2     pour i de 0 à N − 1 faire
3         APS[i] ← indiceCourant ; /* l'adresse du premier successeur du
              sommet i est la position courante du remplissage de FS    */
4         pour j de 0 à N − 1 faire
5             si A[i, j] = 1 alors
6                 FS[indiceCourant] ← j + 1
7                 indiceCourant ← indiceCourant + 1
8         FS[indiceCourant] ← 0 /* ajout du séparateur                  */
9         indiceCourant ← indiceCourant + 1
```

**Solution question 5.2** : *Comment modifier l'algorithme pour qu'il fonctionne pour un graphe non orienté ?*

Cet algorithme fonctionne que le graphe soit orienté ou non. Il n'y a donc pas besoin de le modifier.

**Solution question 5.3** : *Construire les matrices d'adjacence et dérouler l'algorithme sur les graphes $G$ et $G'$ donnés à la figure 7.*

La matrice $A$ donne la matrice d'adjacence du graphe $G$ et la matrice $A'$ donne la matrice d'adjacence du graphe $G'$.

$$A = \begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix} \qquad A' = \begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

**Correction Exercice 6** :

**Solution question 6.1** : *Écrire un algorithme construisant la matrice d'adjacence $A$ du graphe $G$.*

Comme précédemment les sommets sont nommés 1 à $N$ et les indices vont de 0 à $N-1$.

Une solution est donnée par l'algorithme 2. Celui-ci parcourt le tableau $FS$ pour remplir les lignes de la matrice (lignes 2 à 6). Lorsque la valeur du tableau est à 0 (ligne 3), il s'agit d'un séparateur entre sommets, on passe à la ligne de la matrice suivante (ligne 4)), sinon les numéros des sommets successeurs ($FS[parcours]$) donnent les éléments de la matrice à mettre à 1 (ligne 6).

**Algorithme 2 :** $FS/APS \rightarrow$ matrice d'adjacence

**Données** : $N, M$ : nombres de sommets et d'arêtes
$FS[0..N+M-1]$ : tableau des successeurs
$parcours$ : indice de parcours de $FS$, initialement à 0
$l$ : numéro de ligne dans la matrice $A$, initialement à 0
**Résultat** : $A[0..N-1, 0..N-1]$ : la matrice d'adjacence

```
1 début
2     pour parcours de 0 à N+M-1 faire
3         si FS[parcours] = 0 alors
4             l ← l+1 /* passage au sommet suivant                    */
5         sinon
6             A[l, FS[parcours]-1] ← 1 /* le -1 est du à la numérotation des
                 indices des tableaux à partir de 0                   */
```

**Solution question 6.2** : *Comment modifier l'algorithme pour qu'il fonctionne pour un graphe non orienté ?*

Le fait que le graphe soit orienté ou non ne modifie en rien la manière de fonctionner de l'algorithme. La seule chose qui doit être changée dans l'algorithme est la taille du tableau $FS$ qui passe de $N+M+1$ à $N+2M+1$ puisque chaque arête y est comptée deux fois.

**Solution question 6.3** : Construire les tableaux $APS$ et $FS$ et dérouler l'algorithme

sur les graphes $G$ et $G'$ donnés à la figure 7.

Les tables 3 et 4 donnent les tableaux $APS$ et $FS$ u graphe $G$.

| 0 | 2 | 6 | 9 | 13 | 17 | 20 | 24 |
|---|---|---|---|---|---|---|---|

TABLE 3 – Table $APS$ du graphe $G$

| 2 | 3 | 0 | 1 | 4 | 5 | 0 | 1 | 4 | 0 | 2 | 3 | 5 | 0 | 2 | 4 | ... |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ... | 6 | 7 | 0 | 5 | 7 | 0 | 5 | 6 | 8 | 0 | 7 | 0 | | | | |

TABLE 4 – Table $FS$ du graphe $G$

**Correction Exercice 7** :

**Solution question 7.1** : *Écrire un algorithme linéaire en nombre de sommets qui calcule le tableau des prédécesseurs $FP$ et le tableau des adresses des premiers prédécesseurs $APP$.*

Comme précédemment les sommets sont nommés 1 à $N$ et les indices vont de 0 à $N-1$.

L'algorithme 3 donne une solution à la construction des tables $APP$ et $FP$. Comme cela a été suggéré dans l'énoncé, l'algorithme commence par calculer le nombre de prédécesseurs pour chaque sommet (lignes 2 et 3) qu'il mémorise dans le tableau $NS$. Il construit ensuite une première version du tableau $APP$ sur la base de ce nombre de prédécesseurs (lignes 4 à 6) : pour le sommet $i$ l'adresse du premier prédécesseur sera l'adresse du premier prédécesseur du sommet précédent ($APP[i-1]$), plus le nombre de prédécesseurs de ce sommet, ($NS[i-1]$) qui vient d'être calculé, plus 1 pour le séparateur. L'algorithme remplit ensuite le tableau $FP$ (lignes 6 à 11) en parcourant le tableau $FS$ : si la valeur courante est nulle, alors il change de sommet (ligne 8), sinon il ajoute le sommet courant au tableau $FP$ à la première place déterminée par le tableau $APP$ (ligne 10) et incrémente la valeur du tableau $APP$ pour mémoriser le prochain prédécesseur à la case suivante. Noter ici que l'indice du tableau $APP$ est à chaque fois diminué de 1 pour tenir compte du décalage entre les numéros de sommets et les indices des tables. Pour finir l'algorithme reconstruit le tableau $APP$ (lignes 12 à 15) qui a été modifié.

**Algorithme 3 :** construction de $FP/APP$ ou calcul de $G^{-1}$

**Données** : $N, M$ : nombres de sommets et d'arêtes
$FS[0..N+M-1]$ : tableau des successeurs
$APS[0..N-1]$ : tableau des adresses des premiers successeurs
$NS[0..N-1]$ : nombres de prédécesseurs par sommet
$som$ : numéro du sommet courant, initialement à 1
**Résultat** : $FP[0..N+M-1]$ : tableau des prédécesseurs
$APP[0..N-1]$ : tableau des adresses des premiers prédécesseurs

```
1  début
2      pour i de 0 à N+M-1 faire
3          si FS[i] ≠ 0 alors NS[FS[i]-1]++
           /* remplissage de APP                                   */
4      APP[0] ← 0
5      pour i de 1 à N-1 faire  APP[i] ← APP[i-1] + NS[i-1] + 1
           /* remplissage de FP                                    */
6      pour i de 0 à N+M-1 faire FP[i] ← 0
7      pour i de 0 à N+M-1 faire
8          si FS[i] = 0 alors  som ← som + 1
9          sinon
10             FP[APP[FS[i]-1]] ← som
11             APP[FS[i]-1]++
           /* reconstruction de APP                                */
12     APP[0] ← 0/* Premier sommet                                 */
13     som ← 2
14     pour i de 0 à N+M-1 faire
15         si FP[i] = 0 alors
16             APP[som-1] ← i+1
17             som ← som + 1
```

## Correction Exercice 8 : Calcul de la fermeture transitive d'un graphe

**Solution question 8.1** : *Écrire un algorithme permettant de calculer la fermeture transitive d'un graphe.*

Comme nous l'avons vu dans le cours, la fermeture transitive d'un graphe est la contraction de toutes les relations du type $a\mathcal{R}b$ et $b\mathcal{R}c$ en $a\mathcal{R}c$. C'est-à-dire qu'un arc, ou une arête, est ajouté entre deux sommets $a$ et $c$ si il existe en chemin entre eux. Comme l'opération est répétée tant qu'il est possible d'ajouter des arcs, ou des arêtes, il y aura un arc ou une arête entre toute paire de sommets reliés par un chemin. Le calcul de la fermeture transitive $G^*$ d'un graphe $G$ permet donc de savoir s'il existe un chemin d'un sommet $x$ vers un sommet $y$ dans $G$ en examinant si le sommet $x$ a comme voisin $y$ dans $G^*$.

Si $M$ est la matrice d'adjacence du graphe $G$, $M^*$ est la matrice d'adjacence du graphe $G^*$, la fermeture transitive de $G$ se calcule mathématiquement comme suit :

$$M^* = M + M^2 + M^3 + \ldots + M^{n-1}$$

où $n$ est le nombre de sommets du graphe et $M^k$ est le produit booléen de la matrice $M$ $k$ fois par elle-même [2]. Dans ce cas $m^*_{x,y} = 1$ signifie qu'il existe un chemin qui relie $x$ à $y$ et $m^*_{x,y} = 0$ sinon.

La complexité associée à ce calcul est $O(n^4)$ ce qui est lourd mais il est heureusement possible de concevoir des algorithmes qui ne font pas le calcul des puissances successives et ont des complexités moins élevées. Il existe plusieurs approches pour réaliser ce calcul, nous donnons ici l'approche par compression de chemin et celle de Roy-Warshall.

L'approche par compression de chemin, donnée par l'algorithme 4, suit simplement de les arcs $a \to b$ et $b \to c$ pour ajouter $c$ dans le voisinage de $a$ autant que cela est nécessaire. Il s'agit quelque part de faire de la compression de chemin le long d'un parcours en largeur à partir de tous les sommets de $G$. Puisque le graphe peut avoir un diamètre maximal de $n-1$, il est nécessaire de faire cette compression $n-1$ fois. Ici, suivant le sens de découverte des sommets, certains chemins peuvent avoir déjà été compressés.

**Algorithme 4 :** Algorithme de fermeture transitive type *compression de chemin*

**Données** : $N$ : nombres de sommets et d'arêtes
$A$ : matrice d'adjacence
**Résultat** : $A*$ : matrice de la fermeture transitive

```
1 début
2     A* ← A
3     pour p de 1 à N − 1 faire
4         pour i de 0 à N − 1 faire
5             pour j de 0 à N − 1 faire
6                 si A*[i, j] = 1 alors
7                     pour k de 0 à N − 1 faire
8                         si A*[j, k] = 1 alors A*[i, k] ← 1
```

Cet algorithme a une complexité de $O(n^4)$, mais il est plus simple que l'algorithme d'addition des puissances de matrices. Ce dernier ne présente donc un intérêt que si nous disposons des puissances de la matrice $A$, par exemple pour mémoriser les chemins d'une certaine longueur.

L'approche proposée par Roy et Warshall consiste elle à parcourir chaque sommet et d'ajouter des liens entre tous ses prédécesseurs et tous ses successeurs. Il s'agit également d'une compression de chemin, même s'il est moins facile de s'en persuader. Il faut imaginer que chaque examen d'un sommet conduit à réduire la distance de un entre les sommets, quelque soit l'ordre d'examen des sommets. Ainsi un graphe linéaire conduit à rendre à la

2. Puisque la valeur 1 dans la matrice d'adjacence traduit un lien entre les sommets considérés, les puissances d'une matrice d'adjacence ne sont donc composées que de valeurs 0 ou 1 qui traduisent l'existence d'un chemin dont la longueur est égale à la puissance.

fin du processus, tous les sommets à distance 1 les uns des autres. L'algorithme 5 donne cette solution avec la matrice d'adjacence $A$. Même si sa similitude avec l'algorithme 4 est notable, la logique de fonctionnement reste différente et la complexité finale est inférieure, en $O(n^3)$.

**Algorithme 5 :** Algorithme de fermeture transitive de *Roy-Warshall*

```
Données : N : nombres de sommets et d'arêtes
          A : matrice d'adjacence
Résultat : A* : matrice de la fermeture transitive
1 début
2     A* ← A
3     pour i de 0 à N − 1 faire
4         pour j de 0 à N − 1 faire
5             si A[j, i] = 1 alors
6                 pour k de 0 à N − 1 faire
7                     si A[i, k] = 1 alors A*[j, k] ← 1
```

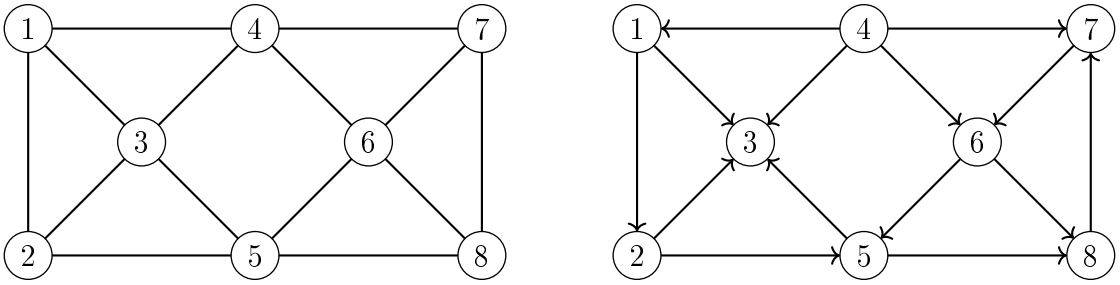
Pour finir, il est également possible d'utiliser des algorithmes de calcul de plus courts chemins, tels que celui de Floyd-Warshall donné dans le cours, pour trouver la fermeture transitive du graphe. Lorsque l'algorithme donne une distance entre deux sommets cela signifie qu'un chemin existe entre les deux. Au besoin, ces algorithmes peuvent être adaptés pour de rendre qu'une matrice d'adjacence classique. À noter que la complexité de ces algorithmes est comparable à celle de l'algorithme 5.

**Solution question 8.2** : *Comment modifier l'algorithme pour qu'il fonctionne pour un graphe non orienté ?*

Puisque l'algorithme utilise la matrice d'adjacence il est utilisable quelque soit le type de graphe : orienté ou non. À noter que la fermeture transitive d'un graphe orienté conduit à la formation de cliques dans les composantes connexes.

# 3 Parcours en largeur et parcours en profondeur

L'objectif des exercices qui suivent est de maîtriser manipulation des parcours de graphe en largeur et en profondeur.




(a) Graphe $G$ non orienté (b) Graphe $G'$ orienté

FIGURE 8 – Graphes non orienté et orienté

**Exercice 9 : Parcours en largeur**

Dans cet exercice nous nous intéressons aux parcours largeur d'un graphe.

**Question 9.1 :** *Donner un parcours en largeur de $G$ (figure 8a) et son arborescence associée.*

**Question 9.2 :** *Les permutations $L_1 = (3, 1, 2, 5, 6, 4, 7, 8)$ et $L_2 = (1, 2, 3, 4, 5, 6, 7, 8)$ sont-elles des parcours en largeur de $G$ ? Si oui, donner une arborescence associée. Est-elle unique ?*

**Question 9.3 :** *Donner un parcours en largeur pour $G'$ (figure 8b) ainsi que l'arborescence associée.*

**Question 9.4 :** *Donner un algorithme récursif générique*[3] *de parcours en largeur d'un graphe $G = (V, E)$ à partir du sommet $s$.*

**Question 9.5 :** *Dérouler l'algorithme sur le graphe de la figure 8a et vérifiez les résultats de la question 2*

**Exercice 10 : Parcours en profondeur**

**Question 10.1 :** *Donner un parcours en profondeur de $G$ (figure 8a) et son arborescence associée.*

**Question 10.2 :** $L_1 = (5, 3, 1, 2, 4, 6, 8, 7)$ *et* $L_2 = (5, 8, 7, 3, 4, 1, 2, 6)$ *sont-ils des parcours en profondeur ? Si oui, donner une arborescence associée. Est-elle unique ?*

3. Nous rappelons qu'un algorithme générique ne fait pas de supposition quant à la structure de données de mémorisation du graphe. Il se base alors sur des notations ensemblistes pour $G = (V, E)$.

**Question 10.3** : *Donner un parcours en profondeur pour $G'$ (figure 8b) ainsi que l'arborescence associée.*

**Question 10.4** : *En confrontant les définitions des parcours en largeur et en profondeur, dire pourquoi la structure de données contenant les sommets visités encore ouvert du parcours en profondeur est cette fois ci une pile.*

**Question 10.5** : *Donner l'état de la pile et de la liste des sommets au cours du parcours donné en 1.*

**Question 10.6** : *Donner un algorithme récursif de parcours en profondeur d'un graphe $G = (V, E)$ à partir du sommet $s$.*

**Question 10.7** : *Dérouler l'algorithme sur le graphe de la figure 8a et vérifiez les résultats de la question 2*

## Correction Exercice 9 : Parcours en largeur

**Solution question 9.1** : *Donner un parcours en largeur de $G$ (figure 8a) et son arborescence associée.*

$L = (8, 5, 6, 7, 2, 3, 4, 1)$ est un parcours en largeur. L'arborescence associée est donnée à la figure 9 ;

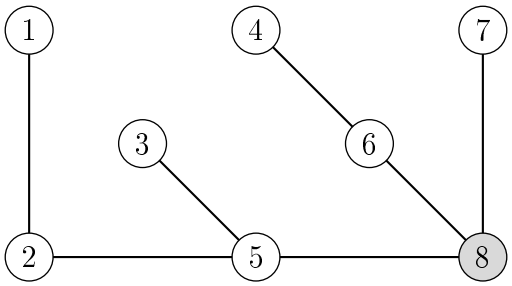



FIGURE 9 – Arborescence associée au parcours en largeur $L = (8, 5, 6, 7, 2, 3, 4, 1)$

**Solution question 9.2** : *Les permutations $L_1 = (3, 1, 2, 5, 6, 4, 7, 8)$ et $L_2 = (1, 2, 3, 4, 5, 6, 7, 8)$ sont-elles des parcours en largeur ? Si oui, donner une arborescence associée. Est-elle unique ?*

$L_1 = (3, 1, 2, 5, 6, 4, 7, 8)$ n'est pas un parcours en largeur car le sommet 6 n'est pas adjacent au premier sommet ouvert pour $L[3, 1, 2, 5]$ dans le graphe 8a.

$L_2$ est un parcours en largeur. La figure 10 montre l'arborescence associée. Cette arborescence est unique car les arêtes sont choisies en fonction du parcours, en fermant les sommets dans l'ordre du parcours qui, lui, est unique. Les arêtes traduisent la relation unique *père-fils* établie par le parcours.

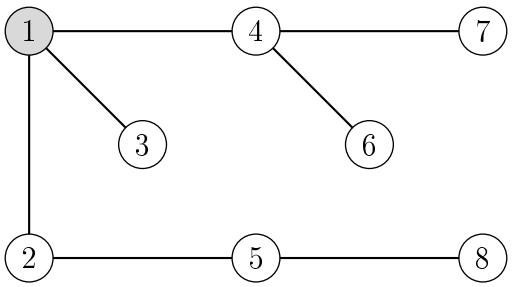



FIGURE 10 – Arborescence associée au parcours en largeur $L_2 = (1, 2, 3, 4, 5, 6, 7, 8)$

**Solution question 9.3** : *Donner un parcours en largeur pour $G'$ (figure 8b) ainsi que l'arborescence associée.*

$L = (4, 3, 6, 7, 1, 8, 5, 2)$ est un parcours en largeur du graphe 8b. La figure 11 donne l'arborescence associée à $L$.

**Solution question 9.4** : *Donner un algorithme récursif générique de parcours en largeur d'un graphe $G = (V, E)$ à partir du sommet $s$.*

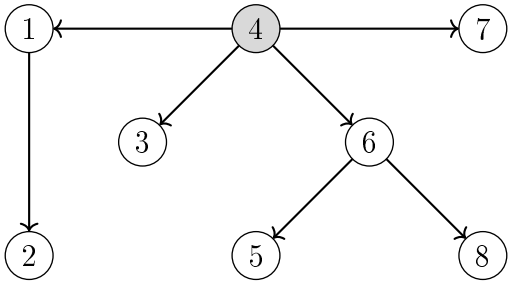



FIGURE 11 – Arborescence associée au parcours en largeur $L = (4, 3, 6, 7, 1, 8, 5, 2)$

L'algorithme 6 réalise un parcours en largeur en appelant récursivement la fonction *parcoursLargeurRec*.

**Algorithme 6 :** parcours en largeur d'un graphe $G(V, E)$ à partir du sommet $s$

**Données** : $f$ : file des sommets en cours **initialisé à** vide
$couleur[u] : u \in \{BLANC, GRIS\}$, **initialisé à** $BLANC$, $\forall u \in V$

```
1  Fonction parcoursLargeurRec(G, s) :
2  début
3      if couleur[s] = BLANC then
4          couleur[s] = GRIS
5          f.add(s)
6      if ¬f.empty() then
7          u ← f.poll()
8          pour chaque sommet v voisin de u faire
9              si couleur[v] = BLANC alors
10                 couleur[v] ← GRIS
11                 f.add(v)
12         parcoursLargeurRec(G, u)
13     else
14         retourner
```

Les lignes 3 à 5 permettent d'initialiser la file si nous sommes au premier appel de l'algorithme. Pour tous les appels successifs la couleur associée au sommet aura été mise à $GRIS$ avant que le sommet ne soit mis dans la file. Ensuite, si la file n'est pas vide, l'algorithme en prend le premier élément et met tous les voisins de ce sommet dans la file. Ainsi les sommets sont bien entrés dans la file dans l'ordre de leur niveau par rapport au sommet d'origine.

Dans le cas du parcours en largeur nous pouvons constater que la formulation récursive n'est pas plus concise que la formulation itérative, l'utilisation d'un objet file étant toujours indispensable au bon déroulement.

## Correction Exercice 10 : Parcours en profondeur

**Solution question 10.1** : *Donner un parcours en profondeur de G (figure 8a) et son arborescence associée.*

$L = (6, 5, 3, 4, 7, 8, 1, 2)$ est un parcours en profondeur pour le graphe 8a. L'arborescence associée est donnée à la figure 12 ;

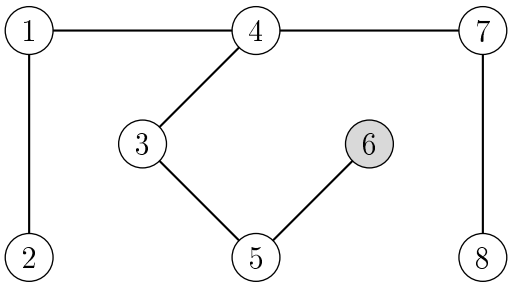



FIGURE 12 – Arborescence associée au parcours en profondeur $L = (6, 5, 3, 4, 7, 8, 1, 2)$

**Solution question 10.2** : $L_1 = (5, 3, 1, 2, 4, 6, 8, 7)$ *et* $L_2 = (5, 8, 7, 3, 4, 1, 2, 6)$ *sont-ils des parcours en profondeur ? Si oui, donner une arborescence associée. Est-elle unique ?*

$L_1 = (5, 3, 1, 2, 4, 6, 8, 7)$ est un parcours en profondeur. L'arborescence donnée à la figure 13 est unique car elle traduit la relation unique *père-fils* établie par le parcours.

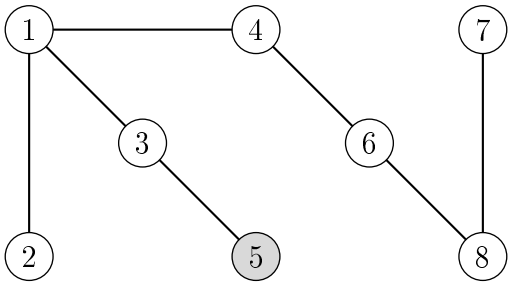



FIGURE 13 – Arborescence associée au parcours en profondeur $L_1 = (5, 3, 1, 2, 4, 6, 8, 7)$

$L_2 = (5, 8, 7, 3, 4, 1, 2, 6)$ n'est pas un parcours en profondeur de 8a car 3 n'est pas dans le voisinage de 7.

**Solution question 10.3** : *Donner un parcours en profondeur pour G′ (figure 8b) ainsi que l'arborescence associée.*

$L = (4, 1, 3, 2, 5, 8, 7, 6)$ est un parcours en profondeur de G' (figure 7b). L'arborescence associée est donnée à la figure 14 ;

**Solution question 10.4** : *En confrontant les définitions des parcours en largeur et en profondeur, dire pourquoi la structure de données contenant les sommets visités encore ouvert du parcours en profondeur est cette fois ci une pile.*

Dans une pile, l'élément retiré est le dernier élément arrivé sur la pile. Ceci correspond donc bien à la définition des ajouts des voisins des sommets déjà visités. En effet, le prochain sommet du parcours est un voisin du dernier sommet ouvert du parcours en cours, donc de la pile.

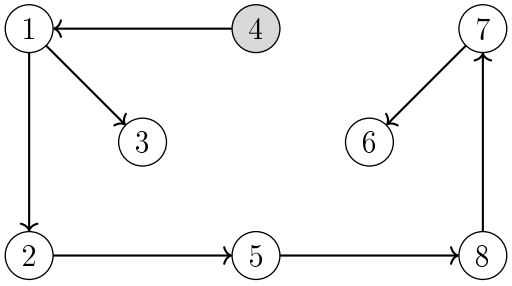



FIGURE 14 – Arborescence associée au parcours en profondeur $L_1 = (4, 1, 3, 2, 5, 8, 7, 6)$

**Solution question 10.5** : *Donner l'état de la pile et de la liste des sommets au cours du parcours donné en 1.*

L'état de la pile et de la liste des sommets, pour le parcours $L = (6, 5, 3, 4, 7, 8, 1, 2)$, est le suivant :

| P | L |
|---|---|
| 6 | 6 |
| 6,5 | 6,5 |
| 6,5,3 | 6,5,3 |
| 6,5,3,4 | 6,5,3,4 |
| 6,5,3,4,7 | 6,5,3,4,7 |
| 6,5,3,4,7,8 | 6,5,3,4,7,8 |
| 6,5,3,4,7 | 6,5,3,4,7,8 |
| 6,5,3,4 | 6,5,3,4,7,8 |
| 6,5,3,4,1 | 6,5,3,4,7,8,1 |
| 6,5,3,4,1,2 | 6,5,3,4,7,8,1,2 |
| 6,5,3,4,1 | 6,5,3,4,7,8,1,2 |
| 6,5,3,4 | 6,5,3,4,7,8,1,2 |
| 6,5,3 | 6,5,3,4,7,8,1,2 |
| 6,5 | 6,5,3,4,7,8,1,2 |
| 6 | 6,5,3,4,7,8,1,2 |
| - | 6,5,3,4,7,8,1,2 |

**Solution question 10.6** : *Donner un algorithme récursif de parcours en profondeur d'un graphe $G$ à partir du sommet $s$.*

L'algorithme récursif du parcours en profondeur est très concis. En effet les appels récursifs permettent de se passer de la pile puisque celle-ci est implicitement mise en place lors des appels imbriqués. Contrairement au parcours en largeur la récursivité simplifie grandement l'algorithme

**Algorithme 7 :** Algorithme de parcours en profondeur d'un graphe $G(V, E)$

**Données** : $couleur[u] : u \in \{BLANC, GRIS\}$, **initialisée à** $BLANC$, $\forall u \in V$

```
1 Fonction parcoursProfondeur(G, u)
2 début
3 |   couleur[u] ← GRIS
4 |   pour chaque sommet v voisin de u faire
5 |   |   si couleur[v] = BLANC alors  parcoursProfondeur(G, v)
```

**NB** : Attention la maîtrise des algorithmes de parcours est très importante car ceux-ci sont très souvent à la base d'algorithmes sur les graphes.

# 4 Coloration de graphes

## Exercice 11 : Planification de travaux

Soient $T = \{t_1, ..., t_7\}$ un ensemble de 7 travaux, et $M = \{m_1, ..., m_7\}$ un ensemble de 7 machines. Chaque travail $t_i$ utilise 3 machines et prend une journée à être réalisé. Pour ne pas avoir de travail inachevé en fin de journée, on décide que deux travaux $t_i$ et $t_j$ ne peuvent être exécutés dans une même journée que s'ils utilisent deux machines différentes. L'utilisation des machines par les travaux est donnée dans la table 5.

| Machine / Travail | $t_1$ | $t_2$ | $t_3$ | $t_4$ | $t_5$ | $t_6$ | $t_7$ |
|---|---|---|---|---|---|---|---|
| $m_1$ | X | X | | | | | |
| $m_2$ | | X | X | X | | | |
| $m_3$ | X | | X | | | | |
| $m_4$ | | X | | X | | X | |
| $m_5$ | X | | X | | X | | X |
| $m_6$ | | | | | X | X | X |
| $m_7$ | | | | X | X | X | X |

TABLE 5 – Utilisation des machines par les travaux

Nous cherchons à planifier les travaux $T = \{t_1, ..., t_7\}$ sur le moins de jours possibles.

**Question 11.1** : *Représenter les contraintes du problème à l'aide d'un graphe.*

**Question 11.2** : *Proposer une solution pour la réalisation de ces travaux.*

## Exercice 12 : Algorithme de coloration

La coloration des graphes est un domaine d'applications à part entière. A titre d'exemples on peut citer les cartes géographiques planaires, la visualisation, les tracés des circuits imprimés sur plusieurs faces, etc. Il s'agit de colorier les sommets d'un graphe avec un nombre minimum de couleurs. Le problème général est NP-complet, d'où l'idée d'utiliser des heuristiques. Plusieurs solutions sont envisagées, en voici deux :

1. traiter les sommets dans l'ordre des numéros apparaissant dans la structure de graphe,
2. traiter les sommets dans l'ordre décroissant de leurs degrés.

Dans les deux cas, on attribue à chaque sommet la plus petite couleur non utilisée par un de ses voisins.

**Question 12.1** : *Déterminer les deux colorations possibles du graphe figure 15 selon l'ordre des sommets et les degrés.*

**Question 12.2** : *Écrire un algorithme qui permet de colorier les sommets d'un graphe.*

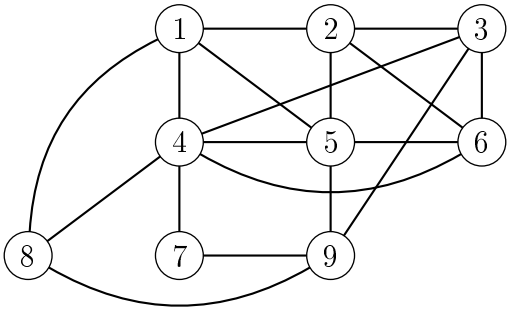



FIGURE 15 – Graphe à colorier

**Correction Exercice 11** :

**Solution question 11.1** : *Représenter les contraintes du problème à l'aide d'un graphe.*

Il est possible de représenter les contraintes de ce problème avec un graphe. Pour cela nous choisissons de représenter les travaux à réaliser par les sommets du graphe et la relation *"partage une machine"* par des arêtes. Ainsi deux sommets mis en relation par une arête ne peuvent par être exécutés dans la même journée. La figure 16 donne le graphe correspondant aux contraintes données par la table 5.

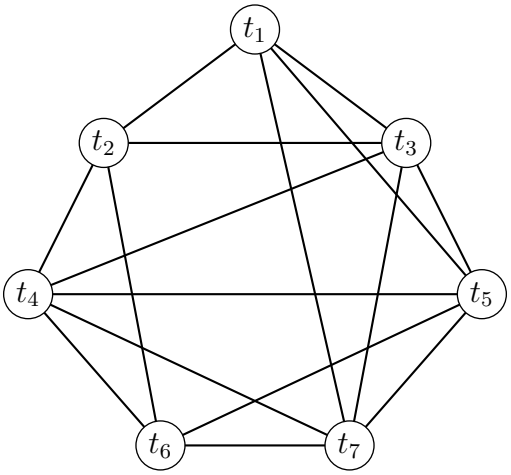



FIGURE 16 – Graphes des contraintes de partage des machines entre travaux

**Solution question 11.2** : *Proposer une solution pour la réalisation de ces travaux.*

Il est évident qu'en choisissant sept dates différentes pour les sept travaux, ceux-ci pourront être réalisés en disposant de toutes les machines dont ils ont besoin. Notre but est cependant de trouver une planification sur une période la plus courte tout en respectant les contraintes de non-partage. Pour, cela il est possible d'utiliser le graphe 16 en examinant les travaux dans un ordre arbitraire et en leur affectant une date en faisant attention à ne pas choisir une date déjà choisie par un voisin.

Pour être sûr de choisir une date compatible avec ses voisins, il est possible d'utiliser une coloration de graphe puisque, par définition, chaque couleur définit un ensemble de

sommets qui ne sont pas voisins. Nous associons ensuite une date différente à chaque couleur. La figure 17 montre une coloration du graphe 16 avec quatre couleurs. Il est alors possible de proposer une solution avec une date par couleur, par exemple : le premier jour nous exécutons les tâches rouges, donc $t1$ et $t_4$, le second jour la tâche verte $t_7$, le troisième jour les tâches bleues $t_3$ et $t_6$ et le quatrième jour les tâches jaunes $t_2$ et $t_5$.

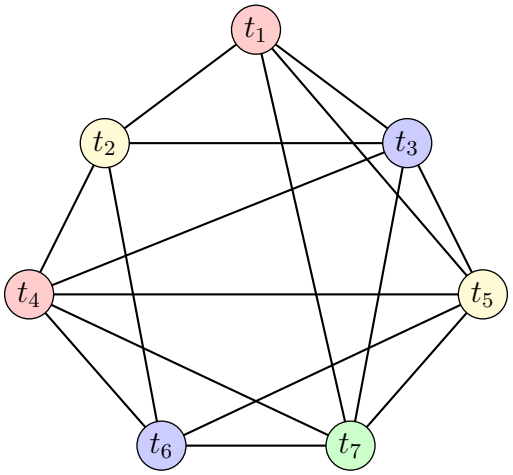



FIGURE 17 – Planification de sept tâches ayant des contraintes de partage de machines entre elles

## Correction Exercice 12 :

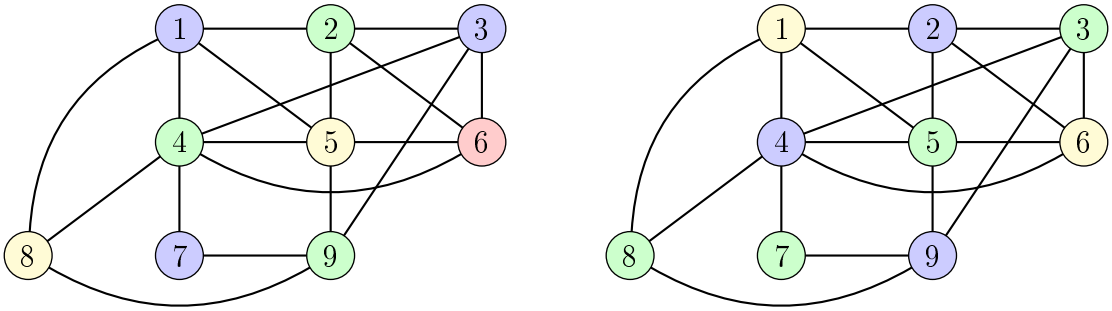
**Solution question 12.1 :** *Déterminer les deux colorations possibles du graphe figure 15 selon l'ordre des sommets et les degrés.*

La coloration du graphe 15 suivant l'ordre des numéros ou l'ordre décroissant du degré des sommets donne les résultats suivants :

1. 4 couleurs si on examine les sommets dans l'ordre croissant des numéros. Un exemple est donné à la figure 18a.
2. 3 couleurs si on examine les sommets dans l'ordre décroissant des degrés des sommets : 4, 5, 1, 2, 3, 6, 9, 8, 7. Un exemple est donné à la figure 18b.

**Solution question 12.2 :** *Écrire un algorithme qui permet de colorier les sommets d'un graphe.*

L'algorithme 8 donne une solution possible pour la coloration d'un graphe. Il ne s'agit pas de la coloration avec le plus petit nombre de couleur, mais simplement une coloration possible.




(a) Coloration avec les numéros de sommets (b) Coloration avec les degrés

FIGURE 18 – Graphe coloré

**Algorithme 8 :** Algorithme de la fonction $coloration(G) : k$

**Données** : $G$ : le graphe à colorier à $n$ sommets
$ppncl$ : plus petite couleur libre
$tabOccCoul[1..n]$ : tableau des couleurs utilisées par les voisins d'un sommet
**Résultat** : $couleur(u)$ : numéro de la couleur du sommet $u$, initialement à 0
$k$ : nombre de couleurs utilisées

```
1  début
2      k ← 0
3      pour chaque sommet s de G dans l'ordre décroissant du degré de s faire
4          tabOccCoul[1..n] ← 0
5          pour chaque sommet u voisin de s faire
6              if couleur(u) ≠ 0 then tabOccCoul[couleur(u)] + +
7          ppncl ← 1
8          tant que tabOccCoul[ppncl] ≠ 0 faire ppncl + +
9          couleur(s) ← ppncl
10         si k < ppncl alors kgetsppncl
```

# Master 2 Informatique I2A

**Master DVL Master ITVL**

- Finance
- Histoire
- Géographie
- Informatique
- Mathématiques
- Sciences Pour l'Ingénieur
- Français Langue Étrangère
- Administration Économique et Sociale
- Diplôme d'accès aux études universitaires

**Master mention Informatique**

Parcours Informatique Avancé et Applications (I2A)

CTU
Centre de Télé-enseignement Universitaire
http://ctu.univ-fcomte.fr

## Filière Informatique

**VVI9MTGC**

Théorie des graphes et combinatoire

**Mr PHILIPPE - LAURENT**
*laurent.philippe@univ-fcomte.fr*

UNIVERSITÉ DE FRANCHE-COMTÉ
UBFC
UNIVERSITÉ BOURGOGNE FRANCHE-COMTÉ

UNIVERSITÉ DE FRANCHE-COMTÉ

Module INFO-TGC

L. Philippe

# Théorie des Graphes et Combinatoire

## Master Informatique Avancée et Applications

**Chapitre II - Exercices - Programmation Linéaire**

CTU Centre de Télé-enseignement Universitaire
UNIVERSITÉ DE FRANCHE-COMTÉ

Centre de télé enseignement
Filière Informatique
Domaine Universitaire de la Bouloie
25030 Besançon Cedex (France)

# 1 Résolution de programmes linéaires

## Exercice 1 : Aliment pour bétail

On désire déterminer la composition, à coût minimal, d'un aliment pour bétail qui est obtenu en mélangeant au plus trois produits bruts : orge, arachide, et sésame. L'aliment ainsi conditionné doit comporter au moins 22% de protéines et 3,6% de graisses, pour se conformer aux exigences de la clientèle. On a indiqué dans le tableau 1 les pourcentages de protéines et de graisses contenues, respectivement, dans l'orge, les arachides et le sésame, ainsi que le coût par tonne des produits bruts.

| produit brut | orge | arachide | sésame | % requis |
|---|---|---|---|---|
| % de protéines | 12 | 52 | 42 | 22 |
| % de graisses | 2 | 2 | 10 | 3,6 |
| coût par tonne | 25 | 41 | 39 | |

TABLE 1 – Composition et coût des composants des aliments

On note $x_j$ $(j = 1, 2, 3)$ la fraction de tonne de produit brut $j$ contenu dans une tonne d'aliment.

**Question 1.1 :** *Formuler le problème sous la forme d'un programme linéaire en respectant d'abord la forme canonique et puis standard. Expliquer les différentes étapes de votre raisonnement.*

**Question 1.2 :** *Montrer qu'il est possible de réduire la dimension du problème. Le résoudre graphiquement. Justifier.*

**Question 1.3 :** *Faire la résolution du problème en utilisant l'algorithme du simplexe.*

## Exercice 2 : Algorithme du simplexe

Soit le programme linéaire suivant :

$$\begin{cases} \text{minimiser} & 2x_1 + 7x_2 & & \\ \text{avec les contraintes} & x_1 & = & 7 \\ & 3x_1 + x_2 & \geq & 24 \\ & x_2 & \geq & 0 \\ & x_3 & \leq & 0 \end{cases}$$

**Question 2.1 :** *Convertir ce programme linéaire dans sa forme standard.*

**Question 2.2 :** *Quelles sont les variables de base et les variables hors-base ?*

**Question 2.3 :** *Donner l'état des variables $N$, $B$, $A$, $b$, $c$ et $v$ utiles au bon déroulement de l'algorithme du simplexe.*

## Exercice 3 : Sites internet

Le directeur d'une entreprise de développement souhaite rentabiliser au mieux son chiffre d'affaire du mois. Il a 10 commandes de site internet à réaliser dans le mois sachant que chacun des sites peut prendre trois formes plus ou moins élaborées à partir d'un canevas initial sur lesquelles il fait intervenir un graphiste et un développeur. Le temps mensuel consacré à ces commandes est de 136 heures pour le graphiste et de 160 pour le développeur.

Dans la première forme demande 12 heures de travail au graphiste et 24 heures de travail au développeur. La seconde demande 16 heures au graphiste et 4 au développeur et la troisième 4 heures au graphiste et 8 heures au développeur. En fonction de la forme du site le prix varie. Le directeur facture ainsi 500 € pour la première forme, 300 € pour la deuxième et 200 € pour la troisième.

**Question 3.1 :** *Formuler le problème sous la forme d'un programme linéaire en respectant d'abord la forme canonique et puis standard. Expliquer les différentes étapes de votre raisonnement.*

**Question 3.2 :** *Montrer qu'il est possible de réduire la dimension du problème. Le résoudre graphiquement. Justifier.*

## Correction Exercice 1 : Aliment pour bétail

**Solution question 1.1 :** *Formuler le problème sous la forme d'un programme linéaire en respectant d'abord la forme canonique et puis standard. Expliquer les différentes étapes de votre raisonnement.*

$$\text{programme linéaire} \begin{cases} \text{minimiser} & 25x_1 + 41x_2 + 39x_3 & \\ \text{avec les contraintes} & 12x_1 + 52x_2 + 42x_3 & \geq 22 \\ & 2x_1 + 2x_2 + 10x_3 & \geq 3,6 \\ & x_1 + x_2 + x_3 & = 1 \\ & x_1, x_2, x_3 & \geq 0 \end{cases}$$

$$\text{forme canonique} \begin{cases} \text{maximiser} & -25x_1 - 41x_2 - 39x_3 & \\ \text{avec les contraintes} & -12x_1 - 52x_2 - 42x_3 & \leq -22 \\ & -2x_1 - 2x_2 - 10x_3 & \leq -3,6 \\ & x_1 + x_2 + x_3 & \leq 1 \\ & -x_1 - x_2 - x_3 & \leq -1 \\ & x_1, x_2, x_3 & \geq 0 \end{cases}$$

$$\text{forme standard} \begin{cases} z = & -25x_1 - 41x_2 - 39x_3 \\ x_4 = -22 + & 12x_1 + 52x_2 + 42x_3 \\ x_5 = -3,6 + & 2x_1 + 2x_2 + 10x_3 \\ x_6 = 1 - & x_1 - x_2 - x_3 \\ x_7 = -1 + & x_1 + x_2 + x_3 \\ & x_1, x_2, x_3, x_4, x_5, x_6, x_7 \geq 0 \end{cases}$$

**Solution question 1.2 :** *Montrer qu'il est possible de réduire la dimension du problème. Le résoudre graphiquement. Justifier.*

Il est possible de réduire le nombre de variables du programme linéaire précédent grâce à la 3ème équation. On tire $x_3$ de cette équation :

$$x_3 = 1 - x_1 - x_2$$

On remplace sa valeur dans le programme linéaire précédent.

$$\begin{cases} \text{minimiser} & 25x_1 + 41x_2 + 39(1 - x_1 - x_2) & \\ \text{avec les contraintes} & 12x_1 + 52x_2 + 42(1 - x_1 - x_2) & \geq 22 \\ & 2x_1 + 2x_2 + 10(1 - x_1 - x_2) & \geq 3,6 \\ & x_1 + x_2 + & x_3 = 1 \\ & x_1, x_2 & \geq 0 \end{cases}$$

On obtient le programme linéaire suivant à 2 variables :

$$\begin{cases} \text{minimiser} & - & 14x_1 & + & 2x_2 & & \\ \text{avec les contraintes} & - & 30x_1 & + & 10x_2 & \geq & -20 \\ & - & 8x_1 & - & 8x_2 & \geq & 6,4 \\ & & x_1, x_2 & & & \geq & 0 \end{cases}$$

Que l'on peut simplifier en :

$$\begin{cases} \text{minimiser} & - & 7x_1 & + & x_2 & & \\ \text{avec les contraintes} & - & 3x_1 & + & x_2 & \geq & -2 \\ & - & x_1 & - & x_2 & \geq & 0,8 \\ & & x_1, x_2 & & & \geq & 0 \end{cases}$$

Finalement nous mettons le programme sous forme canonique :

$$\begin{cases} \text{maximiser} & 7x_1 & - & x_2 & & \\ \text{avec les contraintes} & 3x_1 & - & x_2 & \leq & 2 \\ & x_1 & + & x_2 & \leq & 0,8 \\ & x_1, x_2 & & & \geq & 0 \end{cases}$$

La figure 1 illustre la résolution 2D de ce problème. Dans cette figure on pose $x_1 =$ `x` et $x_2 =$ `y`. Les solutions du problème sont contenues dans le simplexe définit par les 4 points `(0,0)`, $(\frac{2}{3}$`,0)`, `(0.7, 0.1)` et `(0, 0.8)`. Pour le programme 2D, la recherche de la solution optimale nous conduit à déplacer la courbe `y = 7x + b`. L'expression à maximiser a la plus grande valeur au niveau de l'intersection de la droite `3x - y - 2 = 0` avec la droite `x + y - 0.8 = 0`.

Ainsi la solution de ce problème est $x_1 = 0.7$, $x_2 = 0.1$ et $x_3 = 0.2$ avec une valeur de 29.4 pour la valeur minimale du prix de la tonne d'aliment ayant les propriétés recherchées.

**Solution question 1.3** : *Faire la résolution du problème en utilisant l'algorithme du simplexe.*

Pour répondre à cette questions nous repartons de la forme simplifiée du problème :

$$\begin{cases} \text{maximiser} & 7x_1 & - & x_2 & & \\ \text{avec les contraintes} & 3x_1 & - & x_2 & \leq & 2 \\ & x_1 & + & x_2 & \leq & 0,8 \\ & x_1, x_2 & & & \geq & 0 \end{cases}$$

Que nous passons sous forme standard :

$$\text{forme standard} \begin{cases} z & = & 7x_1 & - & x_2 & & \\ x_3 & = & 2 & - & 3x_1 & + & x_2 \\ x_4 & = & \frac{4}{5} & - & x_1 & - & x_2 \\ & & x_1, x_2, x_3, x_4 & & & \geq & 0 \end{cases}$$

À noter que nous avons mis la valuer 0.8 sous forme de fraction ($\frac{4}{5}$) pour des raisons de simplicité. Et les solutions de base sont :

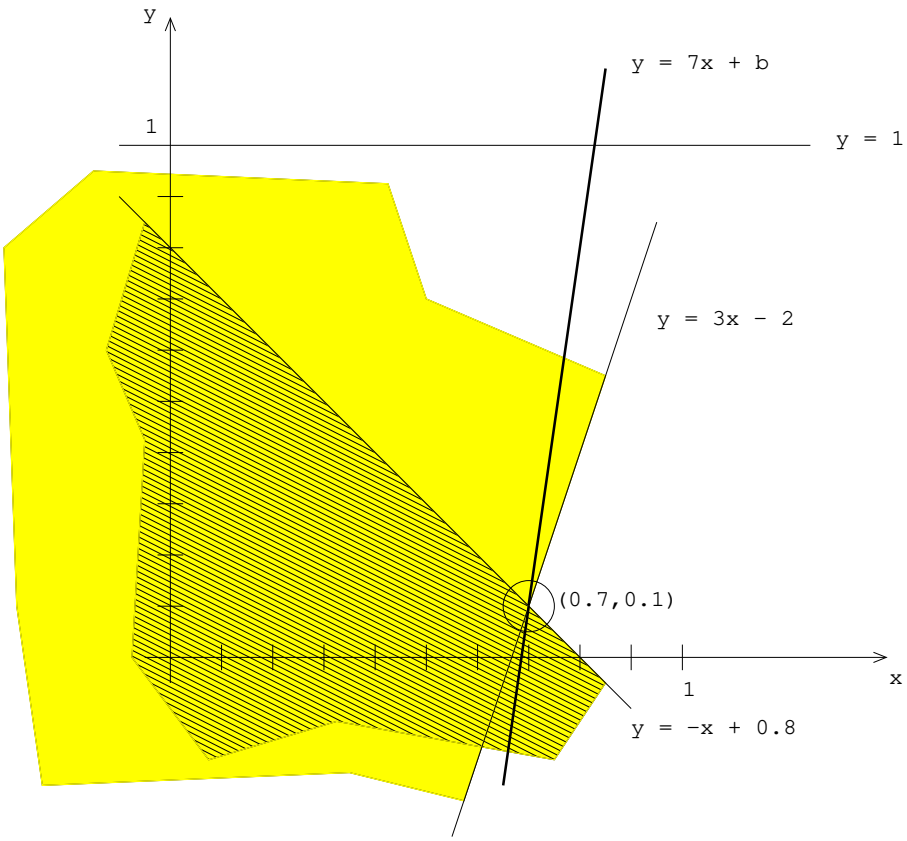



FIGURE 1 – résolution 2D du programme linéaire simplifié

$$(\overline{x_1}, \overline{x_2}, \overline{x_3}, \overline{x_4}) = (0, 0, 2, \frac{4}{5})$$

et la valeur de l'objectif $z = 0$.

La variable $x_1$ est la variable hors-base qui a le plus grand coefficient dans la fonction objectif, c'est elle dont la valeur est augmentée. Dans les contraintes nous avons :

$$\begin{array}{lcl} x_1 > \frac{2}{3} & \Rightarrow & x_3 < 0 \text{ contrainte la plus stricte} \\ x_1 > \frac{4}{5} & \Rightarrow & x_4 < 0 \end{array}$$

C'est la première contrainte qui est la plus stricte. Nous exprimons donc $x_1$ dans cette équation :

$$x_1 = \frac{2}{3} + \frac{1}{3}x_2 - \frac{1}{3}x_3$$

Ceci qui nous donne le programme suivant en remplaçant $x_1$ :

$$\begin{cases} z = 7(\frac{2}{3} + \frac{1}{3}x_2 - \frac{1}{3}x_3) - x_2 \\ x_1 = \frac{2}{3} + \frac{1}{3}x_2 - \frac{1}{3}x_3 \\ x_4 = \frac{4}{5} - (\frac{2}{3} + \frac{1}{3}x_2 - \frac{1}{3}x_3) - x_2 \end{cases}$$

Et se simplifie en :

$$\begin{cases} z = \frac{14}{3} + \frac{4}{3}x_2 - \frac{7}{3}x_3 \\ x_1 = \frac{2}{3} + \frac{1}{3}x_2 - \frac{1}{3}x_3 \\ x_4 = \frac{2}{15} - \frac{4}{3}x_2 + \frac{1}{3}x_3 \end{cases}$$

Pour ce nouveau programme, la solution avec les variables hors-base nulles est :

$$(\overline{x_1}, \overline{x_2}, \overline{x_3}, \overline{x_4}) = (\frac{2}{3}, 0, \frac{2}{15}, 0)$$

Et la valeur objectif $z = \frac{14}{3}$.

C'est au tour de la variable $x_2$ d'entrer en base. Dans les contraintes, la valeur de $x_2$ n'est limitée par la première donc nou prenons la deuxième :

$$x_2 = \frac{1}{10} + \frac{1}{4}x_3 - \frac{3}{4}x_4$$

Ceci qui nous donne le programme suivant en remplaçant $x_2$ :

$$\begin{cases} z = \frac{14}{3} + \frac{4}{3}(\frac{1}{10} + \frac{1}{4}x_3 - \frac{3}{4}x_4) - \frac{7}{3}x_3 \\ x_1 = \frac{2}{3} + \frac{1}{3}(\frac{1}{10} + \frac{1}{4}x_3 - \frac{3}{4}x_4) - \frac{1}{3}x_3 \\ x_2 = \frac{1}{10} + \frac{1}{4}x_3 - \frac{3}{4}x_4 \end{cases}$$

Et se simplifie en :

$$\begin{cases} z = \frac{24}{5} - 2x_3 - x_4 \\ x_1 = \frac{7}{10} - \frac{1}{4}x_3 - \frac{1}{4}x_4) \\ x_2 = \frac{1}{10} + \frac{1}{4}x_3 - \frac{3}{4}x_4 \end{cases}$$

Il n'y a plus de variable à coefficient positif dans l'expression de $z$, nous sommes donc arrivés au bout de la maximisation avec :

$$(\overline{x_1}, \overline{x_2}, \overline{x_3}, \overline{x_4}) = (\frac{7}{10}, \frac{1}{10}, 0, 0)$$

Et la valeur maximale du système simplifié est $z = \frac{24}{5} = 4.8$ pour $x_1 = \frac{7}{10}$ et $x_2 = \frac{1}{10}$.

Pour trouver la solution finale il faut calculer la valeur de $x_3$, à partir de l'équation :

$$x_3 = 1 - x_1 - x_2$$

Ce qui donne $x_3 = \frac{2}{10}$ et une valeur optimale de : $25x_1 + 41x_2 + 39x_3 = \frac{294}{10} = 29,4$

## Correction Exercice 2 : Algorithme du simplexe

**Solution question 2.1 :** *Convertir ce programme linéaire dans sa forme standard.*

Pour convertir le programme dans sa forme standard nous allons d'abord le mettre sous sa forme canonique.

La fonction objectif est une fonction linéaire à maximiser. Comme la fonction objectif de ce programme linéaire est une minimisation, il suffit de prendre l'opposé de cette fonction objectif pour rechercher ensuite à la maximiser.

Il faut d'autre part veiller à garantir les contraintes de positivité pour toutes les variables du problème linéaire. Seule $x_1$ et $x_3$ n'ont pas une telle contrainte. On remplace $x_1$ par $x'_1 - x''_1$. Pour $x_3$ le problème est un peu différent car cette variable n'apparaît dans aucune autre équation, ni dans la fonction objectif. On remplace donc la variable $x_3$ par $-x'_3$. De plus, la contrainte d'égalité est remplacée par deux inégalités (si $x = y$ ssi $x \leq y$ et $x \geq y$). D'où le problème linéaire équivalent :

$$\left\{\begin{array}{lrcrcrcr} \text{maximiser} & -2x'_1 & + & 2x''_1 & - & 7x_2 & & \\ \text{avec les contraintes} & x'_1 & - & x''_1 & & & \leq & 7 \\ & -x'_1 & + & x''_1 & & & \leq & -7 \\ & -3x'_1 & + & 3x''_1 & - & x_2 & \leq & -24 \\ & & & x'_1, x''_1, x_2, x'_3 & & & \geq & 0 \end{array}\right.$$

On renomme les variables $x'_1$, $x''_1$, $x_2$ et $x'_3$ par $x_1$, $x_2$, $x_3$ et $x_4$. Le programme linéaire donné dans l'énoncé est donc équivalent au programme linéaire suivant, sous forme canonique :

$$\left\{\begin{array}{lrcrcrcr} \text{maximiser} & -2x_1 & + & 2x_2 & - & 7x_3 & & \\ \text{avec les contraintes} & x_1 & - & x_2 & & & \leq & 7 \\ & -x_1 & + & x_2 & & & \leq & -7 \\ & -3x_1 & + & 3x_2 & - & x_3 & \leq & -24 \\ & & & x_1, x_2, x_3, x_4 & & & \geq & 0 \end{array}\right.$$

Nous pouvons maintenant convertir le programme linéaire dans sa forme standard, en ajoutant des variables d'écart $x_4$, $x_5$ et $x_6$ :

$$\left\{\begin{array}{llrcrcrcrcr} \text{maximiser} & & -2x_1 & + & 2x_2 & - & 7x_3 & & & \\ \text{avec les contraintes} & x_4 & = & 7 & - & x_1 & + & x_2 & & \\ & x_5 & = & -7 & + & x_1 & - & x_2 & & \\ & x_6 & = & -24 & + & 3x_1 & - & 3x_2 & + & x_3 \\ & & & x_1, x_2, x_3, x_4, x_5, x_6 & & & & & \geq & 0 \end{array}\right.$$

**Solution question 2.2 :** *Quelles sont les variables de base et les variables hors-base ?*

Les *variables de base* sont $x_4$, $x_5$ et $x_6$. Les *variables hors base* sont $x_1$, $x_2$ et $x_3$.

**Solution question 2.3 :** *Donner l'état des variables $N$, $B$, $A$, $b$, $c$ et $v$ utiles au bon déroulement de l'algorithme du simplexe.*

$$N = \left\{1, 2, 3\right\} \quad B = \left\{4, 5, 6\right\} \quad A = \begin{pmatrix} 1 & -1 & 0 \\ -1 & 1 & 0 \\ -3 & 3 & -1 \end{pmatrix} \quad b = \begin{pmatrix} 7 \\ -7 \\ -24 \end{pmatrix} \quad c = \begin{pmatrix} -2 \\ 2 \\ -7 \end{pmatrix}$$

A noter que ce programme linéaire n'est as fait pour être résolu. Il sert juste à travailler sur la reformulation des équations et l'introduction de nouvelles variables. En effet, pour le résoudre il fait regarder le programme initial qui nous donne déjà la solution pour $x_1$ ($x_1 = 7$). On peut alors remplacer $x_1$ dans la fonction objectif pour obtenir maximiser $14 + 7x_2$. Les seules contraintes qui existent sur $x_2$ sont $x2 \geq 24$ et $x2 \geq 0$ ce qui revient à dire que $x_2$ peut prendre n'importe quelle valeur supérieure à 24. Comme il n'est pas borné, la fonction de maximisation doit l'augmenter le plus possible donc lui donner une valeur infinie. Pour ce qui est de $x_3$, on ne sait pas quoi en faire puisqu'il n'intervient pas dans la fonction à maximiser. Il peut prendre n'importe quelle valeur pourvu qu'elle soit négative ($x_3 \leq 0$).

## Correction Exercice 3 : Sites internet

**Solution question 3.1 :** *Formuler le problème sous la forme d'un programme linéaire en respectant d'abord la forme canonique et puis standard. Expliquer les différentes étapes de votre raisonnement.*

Formulation du problème sous la forme d'un programme linéaire :

$$\text{programme linéaire} \begin{cases} \text{maximiser} & 500x_1 + 300x_2 + 200x_3 & & \\ \text{avec les contraintes} & 12x_1 + 16x_2 + 4x_3 & \leq & 136 \\ & 24x_1 + 4x_2 + 8x_3 & \leq & 160 \\ & x_1 + x_2 + x_3 & = & 10 \\ & x_1, x_2, x_3 & \geq & 0 \end{cases}$$

$$\text{forme canonique} \begin{cases} \text{maximiser} & 500x_1 + 300x_2 + 200x_3 & & \\ \text{avec les contraintes} & 12x_1 + 16x_2 + 4x_3 & \leq & 136 \\ & 24x_1 + 4x_2 + 8x_3 & \leq & 160 \\ & x_1 + x_2 + x_3 & \leq & 10 \\ & -x_1 - x_2 - x_3 & \leq & -10 \\ & x_1, x_2, x_3 & \geq & 0 \end{cases}$$

$$\text{forme standard} \begin{cases} z = 500x_1 + 300x_2 + 200x_3 & & \\ x_4 = 136 - 12x_1 - 16x_2 - 4x_3 & & \\ x_5 = 160 - 24x_1 - 4x_2 - 8x_3 & & \\ x_6 = 10 - x_1 - x_2 - x_3 & & \\ x_7 = -10 + x_1 + x_2 + x_3 & & \\ x_1, x_2, x_3, x_4, x_5, x_6, x_7 & \geq & 0 \end{cases}$$

**Solution question 3.2 :** *Montrer qu'il est possible de réduire la dimension du problème. Le résoudre graphiquement. Justifier.*

Il est possible de réduire le nombre de variables du programme linéaire précédent grâce à la $3^{ieme}$ équation. $x_3$ est tiré de cette équation :

$$x_3 = 10 - x_1 - x_2$$

Puis il est remplacé dans les autres équations du programme linéaire précédent. Nous obtenons alors le programme linéaire suivant à 2 variables.

$$\begin{cases} \text{maximiser} & 300x_1 & + & 100x_2 & + & 2000 \\ \text{avec les contraintes} & 8x_1 & + & 12x_2 & \leq & 96 \\ & 16x_1 & - & 4x_2 & \leq & 80 \\ & & x_1, x_2 & & \geq & 0 \end{cases}$$

Qui peut se simplifier en :

$$\begin{cases} \text{maximiser} & 3x_1 & + & x_2 & + & 20 \\ \text{avec les contraintes} & 2x_1 & + & 3x_2 & \leq & 24 \\ & 4x_1 & - & x_2 & \leq & 20 \\ & & x_1, x_2 & & \geq & 0 \end{cases}$$

Il est alors possible de résoudre graphiquement, comme cela est illustré par la figure 2. Dans cette figure on pose $x_1 =$ `x` et $x_2 =$ `y` puis on trace les droites correspondant aux contraintes :

$$\begin{cases} y & = & 3/2x & + & 8 \\ y & = & 4x & - & 20 \end{cases}$$

Les solutions du problème sont contenues dans le simplexe définit par les 4 points `(0,0)`, `(8,0)`, `(6, 4)` et `(5, 0)` (zone hachurée sur la figure). La recherche de la solution optimale nous conduit à déplacer la courbe `y = -3x + b`. L'expression à maximiser a la plus grande valeur au niveau de l'intersection de la droite `2/3x + y - 8 = 0` avec la droite `4x - y - 20 = 0`, c'est à dire au point (6,4).

Ainsi la solution de ce problème est $x_1 = 6$, $x_2 = 4$ et $x_3 = 0$ avec une valeur de 4200 pour le chiffre d'affaires.

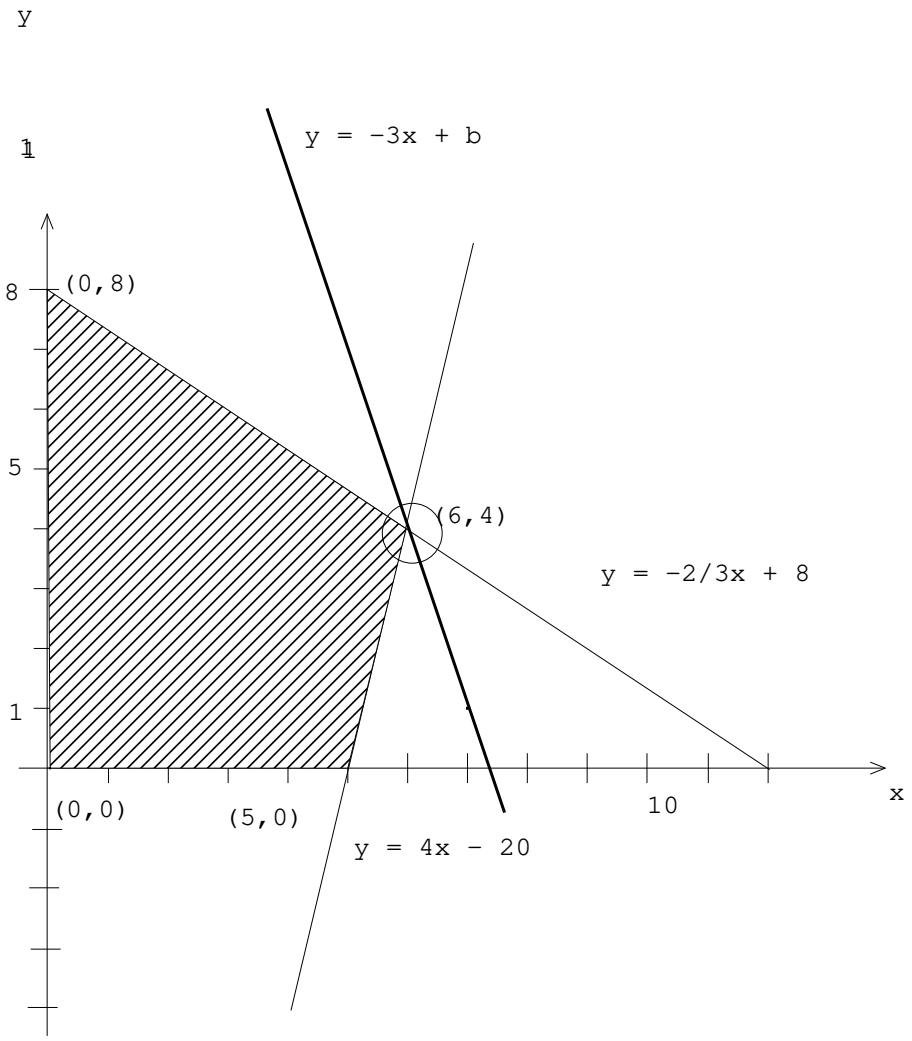



FIGURE 2 – résolution 2D du programme linéaire simplifié

# Master 2 Informatique I2A

**Master DVL Master ITVL**

- Finance
- Histoire
- Géographie
- Informatique
- Mathématiques
- Sciences Pour l'Ingénieur
- Français Langue Étrangère
- Administration Économique et Sociale
- Diplôme d'accès aux études universitaires

**Master mention Informatique**

Parcours Informatique Avancé et Applications (I2A)

CTU
Centre de Télé-enseignement Universitaire
http://ctu.univ-fcomte.fr

## Filière Informatique

**VVI9MTGC**

Théorie des graphes et combinatoire

**Mr PHILIPPE - LAURENT**
*laurent.philippe@univ-fcomte.fr*

UNIVERSITÉ DE FRANCHE-COMTÉ
UBFC
UNIVERSITÉ BOURGOGNE FRANCHE-COMTÉ

UNIVERSITÉ DE FRANCHE-COMTÉ

Module INFO-TGC

L. Philippe

# Théorie des Graphes et Combinatoire

## Master Informatique Avancée et Applications

**Chapitre III - Exercices - Programmation dynamique**

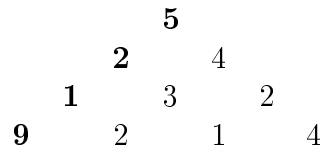
CTU Centre de Télé-enseignement Universitaire
UNIVERSITÉ DE FRANCHE-COMTÉ

Centre de télé enseignement
Filière Informatique
Domaine Universitaire de la Bouloie
25030 Besançon Cedex (France)

# 1 Construction de solutions de programmation dynamique

## Exercice 1 : Pyramide de nombres

Un problème simple et classique en programmation dynamique est la recherche du meilleur chemin dans une pyramide de nombres telle que celle donnée ci-dessous :




Le chemin part de la pointe de la pyramide et descend étage par étage. A chaque étage le chemin cumule les points des cases par lesquelles il passe et l'objectif est de trouver le chemin qui obtient le meilleur score après avoir traversé toute la pyramide.

Sur la figure le chemin donné en gras obtient le meilleur score (ici 17). Nous pouvons aussi constater que le fait de choisir la plus grande valeur à chaque étape ne permet pas de trouver la solution optimale puisqu'ici cette stratégie ne permet d'obtenir qu'un score de 14 avec le chemin 5-4-3-2.

La solution naïve de résolution de ce problème consiste à explorer, depuis le sommet, l'ensemble des chemins possibles. Il y en a 8 dans notre exemple mais le nombre croit très rapidement avec la hauteur de la pyramide : à chaque nouvel étage chacun des chemins calculés dispose de deux possibilités, donc crée deux nouveaux chemins. Le nombre total de chemin est alors de l'ordre de $2^n$. Nous nous proposons donc de trouver une solution de programmation dynamique à ce problème.

Le principe de la programmation dynamique repose sur la décomposition d'une solution en sous-solutions. Il faut alors montrer que la solution optimale peut être calculée à partir des sous-problèmes.

**Question 1.1** : *Donner la structure de la solution optimale en fonction des sous-problèmes.*

**Question 1.2** : *Donner une expression récursive de la solution.*

La solution récursive, si elle est calculé directement, engendre un grand nombre de calcul.

**Question 1.3** : *Comment diminuer le nombre de calculs ?*

**Question 1.4** : *Donner un algorithme de programmation dynamique qui permet de trouver la solution optimale de manière efficace.*

L'algorithme donné précédemment permet de trouver la plus grande valeur de chemin mais ne donne pas le chemin lui-même.

**Question 1.5** : *Comment, à partir des calculs réalisés, retrouver le chemin optimal ? Ce chemin contiendra la liste des indices dans leur ligne des points traversés, en commençant*

*par la base de la pyramide. Dans le cas de l'exemple, le résultat sera 1,1,1,1 puisque le meilleur chemin n'utilise que les premières valeurs de chaque ligne.*

Pour se convaincre de l'intérêt de la programmation dynamique il suffit de tester les temps d'exécution pour les deux algorithmes.

**Question 1.6** : *Écrire une classe* `Pyramid` *qui lit un fichier texte contenant une pyramide de nombres et deux méthodes,* `recursiveAlgo` *et* `progdynAlgo`*, qui implémentent les algorithmes conçus aux questions précédentes.*

Vous pourrez utiliser le fichier `pyramide1.txt` donné sur moodle pour effectuer vos tests.

**Question 1.7** : *En utilisant les fichiers* `pyramide15.txt` *et* `pyramide100.txt`*, calculer les temps d'exécution pour chacun des algorithmes.*

**Question 1.8** : *À partir de quelle taille de pyramide le calcul récursif devient-il trop long ?*

## Exercice 2 : Sac à dos

Nous avons introduit dans le cours le problème du sac à dos, un classique en programmation dynamique. Dans ce problème nous disposons d'un sac supportant un poids maximum que nous voulons remplir avec des objets caractérisés par un poids et une valeur. Chaque objet n'existe qu'en un seul exemplaire.

Le problème du sac à dos consiste à maximiser la valeur totale des objets emportés en respectant la contrainte de poids liée à la capacité du sac. Ce problème a été montré NP-Complet et, de ce fait, il n'existe pas d'algorithme polynomial capable de trouver la solution optimale. La solution systématique, qui consiste à explorer toutes les combinaisons possibles d'objets est combinatoire. Elle revient à construire un arbre où chaque sommet représente le choix possible d'un objet et possède deux branches, une où l'objet est choisi et l'autre où il ne l'est pas. De ce fait le nombre de solutions est de l'ordre de $2^n$.

L'objectif de cet exercice est de réaliser une programmation dynamique permettant de trouver la solution optimale de ce problème.

Pour illustrer le problème posé nous commençons par un exemple où nous disposons d'un sac de 10 kg. Les objets que nous pouvons emporter sont donnés dans la table 3.

| article | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| valeur | 3 | 5 | 8 | 10 |
| poids | 2 | 3 | 5 | 6 |

TABLE 1 – Tableau des valeurs et des poids des objets

**Question 2.1** : *Quelles sont les solutions possibles et leur valeur pour cet exemple.*

Nous cherchons maintenant une solution de programmation dynamique au problème général d'un sac à dos de poids maximum $P$ que nous voulons remplir à partir d'un ensemble $O$ d'objets $o_i$. Un objet est caractérisé par un poids $p_i$ et une valeur $v_i$.

Comme nous l'avons vu dans le cours une approche consiste à décomposer le problème

initial en considérant qu'un des objets $o_i$ est mis ou non dans le sac. Dans ce cas la valeur optimale du sac à dos est calculée à partir :

1. du cas où l'objet fait partie de la solution optimale, donc du même problème de sac à dos mais où on considère que l'objet est pris et il nous reste à résoudre le sous-cas avec un poids maximal $P = P - p_i$ diminué du poids de l'objet $o_i$ et un ensemble d'objets sans $o_i$ : $O = O - o_i$,
2. du cas où l'objet ne fait pas partie de la solution optimale, donc du même problème de sac à dos où le poids maximal reste à $P$ mais l'ensemble ne contient plus l'objet $o_i$ : $O = O - o_i$.

**Question 2.2** : *Caractériser la structure d'une solution optimale.*

**Question 2.3** : *Donner la relation de récurrence permettant d'ajouter un objet dans le sac à dos compte tenu des objets déjà rangés dans le sac.*

**Question 2.4** : *Dans quelle structure est il possible de stocker les résultats intermédiaires afin de ne jamais calculer deux fois la même chose pour cette résolution ascendante ? Comment la remplir ?*

**Question 2.5** : *Écrire l'algorithme de résolution du problème entier grâce à la programmation dynamique.*

**Question 2.6** : *Donner la complexité de cet algorithme.*

**Question 2.7** : *Programmer l'algorithme récursif et l'algorithme de programmation dynamique en Java pour l'exemple initial.*

**Question 2.8** : *A partir de la table des sous-problèmes, comment retrouver le contenu du sac à dos ?*

## Exercice 3 : Chaîne de montage

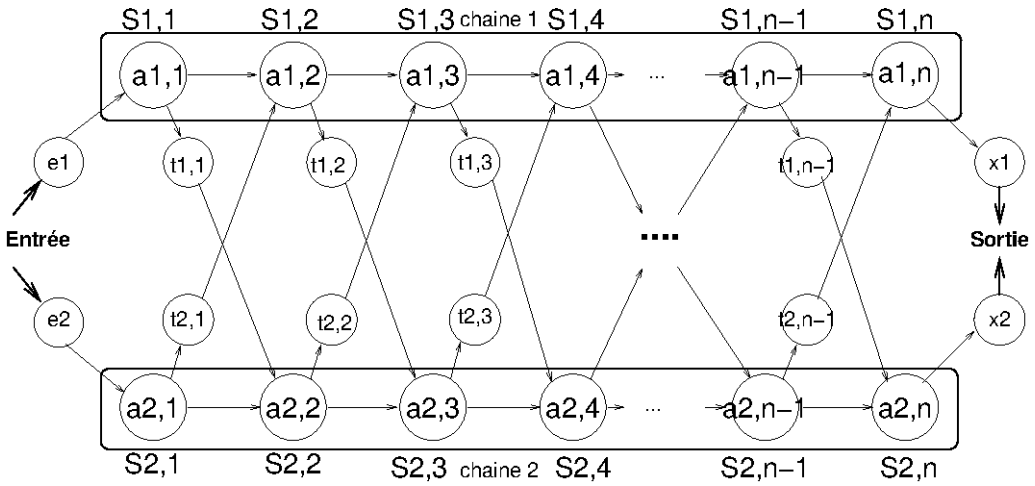



FIGURE 1 – L'atelier de fabrication

Une firme fabrique des automobiles dans un atelier qui a deux chaînes de montage (voir la figure 1). Un châssis arrive sur l'une des chaînes, puis passe par un certain nombre de postes où on lui ajoute des pièces ; une fois terminée, l'auto sort par l'autre extrémité de la chaîne. Chaque chaîne de montage a $n$ stations, numérotées $j = 1, 2, \ldots, n$. On représente le j$^{ieme}$ poste de la chaîne $i$ (avec $i = 1$ ou $i = 2$) par $S_{i,j}$. Le j$^{ieme}$ ($S_{1,j}$) poste de la chaîne 1 fait le même travail que le j$^{ieme}$ ($S_{2,j}$) poste de la chaîne 2.

Les postes ont été installés à des époques différentes et avec des technologies différentes ; ainsi, le temps de montage varie d'un poste à l'autre, même quand il s'agit de postes au fonctionnement identique mais situés sur des chaînes différentes.

Le temps de montage au poste $S_{i,j}$ est $a_{i,j}$. Comme le montre la figure 1, un châssis arrive au poste 1 de l'une des chaînes, puis passe de poste en poste. On a de plus un temps d'arrivée $e_i$ pour le châssis qui entre sur la chaîne $i$, et un temps de sortie $x_i$ pour l'auto achevée qui sort de la chaîne $i$.

Normalement, une fois qu'un châssis arrive sur une chaîne de montage, il ne circule que sur cette chaîne et le temps de passage d'un poste à l'autre sur une même chaîne est négligeable. En cas d'urgence, toutefois, il se peut que l'on veuille accélérer le délai de fabrication d'une automobile. En pareil cas, le châssis transite toujours par les $n$ postes dans l'ordre, mais le chef d'atelier peut faire passer une auto partiellement construite d'une chaîne à l'autre, et ce après chaque poste. Le temps de transfert d'un châssis depuis la chaîne $i$ et après le poste $S_{i,j}$ est $t_{i,j}$, avec $i = 1, 2$ et $j = 1, 2, \ldots, n-1$ (car après le n$^{ieme}$ poste c'est fini).

Le problème consiste à déterminer quels sont les postes à sélectionner sur la chaîne 1 et sur la chaîne 2 pour minimiser le délai de transit d'une auto à travers l'atelier.

On note $f_i[j]$ le délai le plus court possible avec lequel le châssis va du point de départ à la sortie du poste $S_{i,j}$. On note $f^*$ le délai le plus court par lequel un châssis traverse tout l'atelier. De plus, on note $l_i[j]$ le numéro de la chaîne du poste $j-1$ appartenant au chemin optimal arrivant en $S_{i,j}$ et $l^*$ le numéro de la chaîne par laquelle sortent les voitures ayant empruntées le chemin optimal.

**Question 3.1** : *Caractérisation d'une solution optimale*

**Question 3.2** : *Exprimer $f^*$ en fonction de $f_1[n]$ et de $f_2[n]$.*

**Question 3.3** : *Donner les valeurs de $f_1[1]$ et de $f_2[1]$.*

**Question 3.4** : *Caractériser par une relation de récurrence les valeurs de $f_1[j]$ et de $f_2[j]$.*

**Question 3.5** : *Écriture des algorithmes*

**Question 3.6** : *Proposer un algorithme récursif et un algorithme itératif qui calcule les valeurs de $f_1[j]$ et $f_2[j]$ ainsi que les valeurs de $l_1[j]$ et $l_2[j]$ pour $j = 1, 2, \ldots, n$.*

**Question 3.7** : *Proposer un algorithme d'affichage du chemin optimal, d'abord dans le sens inverse (algorithme itératif simple) puis dans le bon sens, en utilisant le tableau $l_i[j]$.*

**Question 3.8** : *Programmation des solutions*

**Question 3.9** : *Écrire un programme C qui permette la mise en œuvre des solutions récursives et itératives présentées dans l'exercice précédent.*

**Question 3.10** : *Comparer les exécutions des deux solutions. Jusqu'à quelle taille de chaîne de montage êtes vous allé avec la version récursive ? Quel temps cela a pris pour le calcul ? Quelle a été la durée du même calcul avec la version itérative issue de la programmation dynamique ? Conclure.*

## Correction Exercice 1 : Pyramide de nombres

**Solution question 1.1** : *Donner la structure de la solution optimale en fonction des sous-problèmes.*

Les chemins partent de la pointe de la pyramide et arrivent à base, le niveau $n$, sans retour. La solution optimale est le chemin qui obtient le meilleur score. Pour le trouver il faut connaître tous les chemins qui arrivent en chacun des points de l'étage $n$. Nous devons donc calculer les scores des chemins en chaque point de la base $p_{i,n}$ avec $i = 1, \ldots, n$.

Pour chacun des points $p_{i,n}$, les chemins qui y arrivent sont passés par l'étage $n-1$. Plus précisément, ces chemins arrivent soit du nœud de droite, soit du nœud de gauche. Par exemple, sur l'étage 4 de la pyramide donnée en exemple, les chemins qui arrivent au point 2 proviennent tous de l'étage 3, soit de droite (le point 3), soit gauche (le point 1).

Nous pouvons donc définir la structure de la solution optimale en fonction des sous-problèmes. Pour cela nous définissons trois fonctions : $g(p)$ qui donne le plus grand score obtenu en passant par le point à gauche du point $p$ à l'étage $n-1$, $d(p)$ qui donne le plus grand score obtenu en passant par le point à droite du point $p$ à l'étage $n-1$ et $v(p)$ qui rend la valeur du point $p$. Le score optimal obtenu l'étage $n$ est le maximum des scores obtenus en chaque point. Ce qui s'écrit :

$$S^* = max_{i=1,\ldots,n}(v(p_{i,n}) + max(g(p_{i,n}), d(p_{i,n})))$$

**Solution question 1.2** : *Donner une expression récursive de la solution.*

A partir de cette formulation la définition récursive est relativement simple puisque l'expression précédente inclut déjà une expression de la valeur en $n$ à partir de la valeur en $n-1$. Il nous suffit donc d'exprimer le premier niveau de la récurrence, celui pour lequel la pyramide n'a qu'un étage. Lorsque la pyramide n'a qu'un étage, la valeur optimale est celle du seul point existant.

$$S^* = v(p_{1,1})$$

L'algorithme 1 donne une solution récursive à la résolution du problème de la pyramide de nombres. Il est décomposé entre une fonction qui calcule la valeur maximale qui peut être atteinte en un point et le programme principal qui cherche la valeur maximale sur la ligne de base de la pyramide.

**Algorithme 1 :** Algorithme récursif pour le problème de la pyramide de nombres

**Données** : $V$ : matrice contenant la liste de nombres
$n$ : nombre d'étages dans la matrice
**Résultat** : $S^*$ : valeur du plus grand chemin

```
1  fonction plusGrandChemin(l, c)
2  début
3      si c = 1 ∧ l = 1 alors retourner V(1, 1)
4      si c = 1 alors retourner plusGrandChemin(l − 1, c) + V(l, c)
5      si c = l alors retourner plusGrandChemin(l − 1, c − 1) + V(l, c)
6      retourner
       max(plusGrandChemin(l − 1, c − 1), plusGrandChemin(l − 1, c)) + V(l, c)
7  Programme principal
8  début
9      S* ← 0
10     pour c de 1 à n faire
11         S ← plusGrandChemin(n, c)
12         si S > S* alors S* ← S
```

Pour la réalisation de l'algorithme, un des problèmes auquel nous sommes confrontés est l'utilisation d'une structure de données pour stocker les valeurs contenues dans la pyramide. Nous choisissons de la stocker dans une matrice $V$ dont nous n'utilisons que le triangle inférieur. Attention, nous supposons ici, pour la lisibilité de l'algorithme, que la matrice est indicée de 1 à $n$, cela ne sera pas le cas lorsque nous écrirons le programme.

**Solution question 1.3** : *Comment diminuer le nombre de calculs ?*

Avec l'expression récursive nous avons une formulation du problème qui, plutôt que de chercher les chemins de manière descendante, les cherche de manière ascendante. Or il est aisé de voir que, pour un point de l'étage au dessus de l'étage courant, il se trouve une fois à droite et une fois à gauche dans les calculs de l'étage. De ce fait, nous pouvons diviser par deux le nombre de calculs à réaliser à chaque étage et obtenir une solution beaucoup plus efficace, à condition de mémoriser les résultats intermédiaires. Nous allons donc créer une nouvelle pyramide, de manière ascendante dans laquelle nous mémorisons, en chaque point la valeur maximale obtenue.

**Solution question 1.4** : *Donner un algorithme de programmation dynamique qui permet de trouver la solution optimale de manière efficace.*

L'algorithme 2 donne une solution de programmation dynamique efficace.

**Algorithme 2 :** Programmation dynamique pour la pyramide de nombres

**Données** : $V$ : matrice contenant la liste de nombres
$n$ : nombre d'étages dans la matrice
**Résultat** : $S^*$ : valeur du plus grand chemin

```
1  I : matrice contenant les résultats intermédiaires
2  début
3      I(1,1) = V(1,1)
4      pour l de 2 à n faire
           /* Extremites de la ligne                                  */
5          I(l,1) = I(l-1,1) + V(l,1)
6          I(l,l) = V(l-1,l-1) + V(l,l)
7          pour c de 2 à l-1 faire
8              I(l,c) = max(I(l-1,c-1), I(l-1,c)) + V(l,c)
9      S* ← 0
10     pour c de 1 à n faire
11         si I(n,c) > S* alors S* ← I(n,c)
```

Comme pour l'algorithme récursif, la matrice $V$ est utilisée pour stocker les valeurs de la pyramide. Les résultats intermédiaires sont stockés dans une matrice $I$, dont nous n'utilisons que le triangle inférieur.

**Solution question 1.5** : *Comment, à partir des calculs réalisés, retrouver le chemin optimal ? Ce chemin contiendra la liste des indices dans leur ligne des points traversés, en commençant par la base de la pyramide. Dans le cas de l'exemple, le résultat sera 1,1,1,1 puisque le meilleur chemin n'utilise que les premières valeurs de chaque ligne.*

Le chemin obtenant le meilleur score peut être retrouvé à partir de la matrice des résultats intermédiaires. A partir du point de la base qui obtient le meilleur score, il est possible de remonter dans la matrice $I$ en choisissant le meilleur des deux voisins (lorsqu'il y a deux voisins) et en mémorisant l'indice correspondant.

L'algorithme 3 permet de retrouver le chemin qui donne le meilleur score.

**Algorithme 3 :** Meilleur chemin dans la pyramide de nombres

**Données** : $I$ : matrice contenant les valeurs des plus grands chemins
$n$ : nombre d'étages dans la matrice
**Résultat** : $C^*$ : chemin optimal

```
1  début
2      best ← 1
3      pour c de 2 à n faire
4          si I(n, c) > I(n, best) alors best ← c
5      C* ← {best}
6      pour l de n à 2 faire
7          si (best = 0) ∨ (I(l − 1, best − 1) < I(l − 1, best)) alors
8              C* ← C* ∪ best
9          sinon
10             C* ← C* ∪ best − 1
11             best ← best − 1
```

**Solution question 1.6** : *Écrire une classe* `Pyramid` *qui lit un fichier texte contenant une pyramide de nombres et deux méthodes,* `recursiveAlgo` *et* `progdynAlgo`*, qui implémentent les algorithmes conçus aux questions précédentes.*

Le code source de la classe `Pyramid` est donné sur moodle.

**Solution question 1.7** : *En utilisant les fichiers* `pyramide15.txt` *et* `pyramide100.txt`*, calculer les temps d'exécution pour chacun des algorithmes.*

Le tableau 2 donne les temps en millisecondes de calcul sur différentes tailles de pyramides.

| taille matrice | 15 | 20 | 25 | 30 | 35 | 40 | 50 |
|---|---|---|---|---|---|---|---|
| tps récursif | 2 | 18 | 184 | 5522 | 205168 | $>$ 1h | $>$ 1h |
| tps dynamique | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

TABLE 2 – Performance comparée entre la programmation récursive et la programmation dynamique

**Solution question 1.8** : *À partir de quelle taille de pyramide le calcul récursif devient-il trop long ?*

Sur ma machine de travail, équipée d'un processeur Intel Core i5-10310U CPU à 1.70GHz et avec un JDK Oracle 12, une pyramide avec 35 lignes prend 251847 millisecondes soit 3 minutes et 25 secondes pour exécuter la recherche récursive alors que le temps pris en programmation dynamique n'atteint pas la milliseconde ! Le programme dynamique met moins d'1 milliseconde pour traiter une pyramide de 100 lignes alors qu'il est impossible de réaliser ce traitement en récursif.

## Correction Exercice 2 : Sac à dos

**Solution question 2.1** : *Quelles sont les solutions possibles et leur valeur pour cet exemple.*

Les solutions possibles sont les combinaisons d'objets qui ne dépassent pas le poids supporté par le sac à dos. Elle sont données dans la table 3.

| article | 1 | 2 | 3 | 4 | poids | valeur |
|---|---|---|---|---|---|---|
| solution 1 | x | - | - | - | 2 | 3 |
| solution 2 | - | x | - | - | 3 | 5 |
| solution 3 | - | - | x | - | 5 | 8 |
| solution 4 | - | - | - | x | 6 | 10 |
| solution 5 | x | x | - | - | 5 | 8 |
| solution 6 | x | - | x | - | 7 | 11 |
| solution 7 | x | - | - | x | 8 | 13 |
| solution 8 | - | x | x | - | 8 | 13 |
| solution 9 | - | x | - | x | 9 | 15 |
| **solution 10** | x | x | x | - | 10 | **16** |

TABLE 3 – Tableau des valeurs et des poids des objets

Cette table permet de mettre en évidence la nécessité d'explorer les solutions pour trouver la plus grande valeur possible.

**Solution question 2.2** : *Caractériser la structure d'une solution optimale.*

Le problème initial peut être décomposé en deux sous problèmes suivant que l'objet $o_i$ est pris ou non.

Le premier des sous problèmes, celui où l'objet est chargé dans le sac à dos, est caractérisé par un sac à dos le poids maximal $P - p_i$ et une liste d'objets $O - o_i$. Le second, celui où l'objet $o_i$ n'est par chargé dans le sac à dos, est caractérisé par un poids maximal $P$ et une liste d'objets $O - o_i$. La sous-structure optimale de ce problème repose donc sur la plus grande des valeurs optimales de ces deux sous-problèmes.

**Solution question 2.3** : *Donner la relation de récurrence permettant d'ajouter un objet dans le sac à dos compte tenu des objets déjà rangés dans le sac.*

Soit $V(P, i)$ la valeur optimale d'un chargement de poids maximal $P$ pour l'ensemble des $i$ premiers objets (ceux-ci peuvent être sélectionnés ou pas). La résolution du problème entier du sac à dos consiste à résoudre $V(P, n)$. La relation de récurrence permettant de savoir si l'objet $i$ peut être ajouté est la suivante :

$$V(P, i) = \max\{V(P, i-1), V(P - p_i, i-1) + v_i\}$$

**Solution question 2.4** : *Dans quelle structure est il possible de stocker les résultats intermédiaires afin de ne jamais calculer deux fois la même chose pour cette résolution ascendante ? Comment la remplir ?*

L'idée est de conserver les valeurs des chargements optimaux pour chaque poids et pour des séries des objets 1 à $i$, avec $i = 1 \dots n$. La taille de la table ainsi calculée est $P \times n$.

Il est possible de réduire le nombre de colonnes en divisant tous les encombrements, y compris celui du sac, par le *pgcd* des encombrements des objets.

Comme la récurrence le montre, chaque valeur dépend des valeurs des chargements calculées pour des problèmes de taille plus petite. Le calcul de ces valeurs se fait donc de manière ascendante et non descendante afin de profiter des calculs précédents pour le calcul optimal courant. Le calcul descendant intuitivement déduit de la formule de récurrence introduit trop de combinatoire dans le calcul de $V(P, i)$. En effet beaucoup de calculs seraient faits de nombreuses fois. Le principe de la programmation dynamique est par contre de les éviter grâce à la tabulation des valeurs optimales des sous problèmes.

**Solution question 2.5** : *Écrire l'algorithme de résolution du problème entier grâce à la programmation dynamique.*

L'algorithme 4 utilise une approche ascendante pour le remplissage du tableau des valeurs des chargements optimaux pour tous les sous problèmes.

**Algorithme 4 :** Programmation dynamique pour le problème du sac à dos

**Données** : $v_i$ : valeur de l'objet $i$ avec $i = 1..n$
$p_i$ : poids de l'objet $i$ avec $i = 1..n$
$P$ : poids disponible dans le sac
**Résultat** : V : valeur optimale du sac à dos avec les $n$ objets pour un poids $P$

```
1  début
2      p : poids courant pour lequel on cherche un chargement optimal
3      i : numéro de l'objet courant (initialisé à 1)
4      V(i,p) : valeur optimale pour le poids p avec des objets 1 à i
       /* Initialisation de la table                                  */
5      pour p de 0 à P faire  V(0,p) ← 0
6      pour i de 0 à n faire  V(i,0) ← 0
7      pour i de 1 à n faire
8          pour p de 1 à P faire
9              si p >= p_i alors
10                 V(i,p) ← max(V(i-1,p), V(i-1,p-p_i) + v_i)
11             sinon  V(i,p) ← V(i-1,p)
12     V ← V(n,P)
```

**Solution question 2.6** : *Donner la complexité de cet algorithme.*

La complexité de l'algorithme de programmation dynamique permettant la résolution de la variante entière du problème du sac à dos est $O(n \times P)$. En effet, pour le calcul de la valeur optimale du chargement du sac acceptant un poids $P$ avec $n$ objets, il faut remplir, pour tout poids entier de 1 à $P$, la valeur optimale du chargement obtenue avec la série des objets 1 à $i$ pour tout $i$ de 1 à $n$.

Dans le cas initial nous avons choisi un poids $P$ qui est relativement petit. Le problème considéré nécessite donc une place mémoire, pour la table $V$, qui reste raisonnable. Mais si nous considérons un sac avec un poids supportable de l'ordre du millier de kilos avec un grand nombre d'objets alors la taille de la table $V$ peut rapidement devenir

conséquente. Il est éventuellement possible alors de réduire d'espace des poids visités en divisant les poids par leur *pgcd* et en ne commençant l'exploration que depuis le plus petit des poids. La complexité dans ce cas est de $O(n \times (\frac{P}{pgcd_{(p_{1..n})}} - \min(p_{1..n})) \sim O(n \times P)$.

**NB** : Attention cette complexité de ne signifie en rien que la solution soit polynomiale. Le problème initial étant $NP$-Complet, trouver une solution polynomiale reviendrait à dire que $P = NP$. La question que nous sommes alors en droit de nous poser est pourquoi cette complexité n'est pas polynomiale ? En fait, la théorie de la complexité étudie les problème par rapport à la taille des entrées et non par rapport à leur valeur. Ici la complexité est dite *pseudo-polynomiale*. Ceci signifie que le problème n'est pas $NP$-Complet au sens dur, mais au sens faible. En effet, pour des tailles de problème relativement petite, le nombre d'opérations est faible, donc la solution est calculable. Le terme *pseudo* vient du fait que dans la complexité dépend d'une *valeur* codée en binaire et non en unaire. Du point de vue de l'étude de la complexité de ce problème, le poids $P$ du sac n'est pas linéaire mais exponentiel, car lorsque sa représentation croît linéairement, sa valeur croît de manière exponentielle. Par contre le nombre d'objets est une valeur unitaire dont la représentation est logarithmique. La taille du problème initial n'est pas la valeur des nombres qui composent la complexité, mais la taille de leur représentation. On comprend alors qu'en doublant la taille de la représentation du volume du sac, alors la valeur du nombre représenté croît de manière exponentielle. A l'inverse, lorsque nombre d'objets croît linéairement, sa représentation en mémoire croît de manière logarithmique. Lorsqu'on considère un problème comme un tri avec une complexité en $O(n^2)$, cette complexité est bien polynomiale car ce que représente $n$ est un nombre de cases (éléments unaires) et doubler la taille du problème revient donc bien à doubler le nombre de case. Pour le tri, $n$ n'est pas un nombre en tant que valeur, mais un nombre de cases. Pour plus de détail, il est possible de trouver des compléments d'information sur Wikipedia [1].

**Solution question 2.7** : *Programmer l'algorithme récursif et l'algorithme de programmation dynamique en Java pour l'exemple initial.*

Le code source du programme Java est donné dans le fichier KnapSac.java

**Solution question 2.8** : *A partir de la table des sous-problèmes, comment retrouver le contenu du sac à dos ?*

A l'issue de l'exécution de la programmation dynamique la table $V[i][p]$ a les valeurs données par la table 4.

Le contenu du sac, en fonction de l'ensemble des objets $o_i$ et pour un sac où $P = 10$, est obtenu en parcourant la table 4 à partir de de la case (4,10). Si la valeur immédiatement au dessus est la même c'est que l'objet n'a pas été pris et on continue à partir de cette case puisque le poids disponible ne change pas. Si par contre la case immédiatement au dessus est différente c'est que l'objet a été pris et il faut se déplacer dans la ligne pour retrouver le volume diminué du poids de l'objet. On obtient l'ensemble des objets lorsqu'on arrive sur la ligne $o_i = 0$.

1. `http://en.wikipedia.org/wiki/Pseudo-polynomial_time`

| $o_i$ \ $P$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 |
| 2 | 0 | 0 | 3 | 5 | 5 | 8 | 8 | 8 | 8 | 8 | 8 |
| 3 | 0 | 0 | 3 | 5 | 5 | 8 | 8 | 11 | 13 | 13 | 16 |
| 4 | 0 | 0 | 3 | 5 | 5 | 8 | 10 | 11 | 13 | 15 | 16 |

TABLE 4 – Table des valeurs optimales du sac à dos

## Correction Exercice 3 : Chaîne de montage

**Solution question 3.1 :** *Exprimer $f^*$ en fonction de $f_1[n]$ et de $f_2[n]$.*

$f^* = \min(f_1[n] + x_1, f_2[n] + x_2)$

**Solution question 3.2 :** *Donner les valeurs de $f_1[1]$ et de $f_2[1]$.*

$$f_i[1] : \begin{cases} f_1[1] &= e_1 + a_{1,1} \\ f_2[1] &= e_2 + a_{2,1} \end{cases}$$

**Solution question 3.3 :** *Caractériser par une relation de récurrence les valeurs de $f_1[j]$ et de $f_2[j]$.*

$$f_1[j] = \begin{cases} e_1 + a_{1,1} & \text{si } j = 1, \\ \min(f_1[j-1] + a_{1,j}, f_2[j-1] + t_{2,j-1} + a_{1,j}) & \text{si } j \geq 2 \end{cases}$$

$$f_2[j] = \begin{cases} e_2 + a_{2,1} & \text{si } j = 1, \\ \min(f_2[j-1] + a_{2,j}, f_1[j-1] + t_{1,j-1} + a_{2,j}) & \text{si } j \geq 2 \end{cases}$$

**Solution question 3.4 :** *Proposer un algorithme récursif et un algorithme itératif qui calculent les valeurs de $f_1[j]$ et $f_2[j]$ ainsi que les valeurs de $l_1[j]$ et $l_2[j]$ pour $j = 1, 2, \ldots, n$.*

L'algorithme récursif est divisé en deux parties. Il y a la partie correspondant au lancement du calcul donné par l'algorithme 5. Les fonctions de calcul des valeurs optimales sur chacune des chaînes sont données par l'algorithme 6.

L'algorithme itératif, résultat de la programmation dynamique grâce à une programmation ascendante est donné par l'algorithme 7. Il retrace le cheminement exact de la solution optimale à l'intérieur de la chaîne de montage.

**Solution question 3.5 :** *Proposer un algorithme d'affichage du chemin optimal, d'abord dans le sens inverse (algorithme itératif simple) puis dans le bon sens, en utilisant le tableau $l_i[j]$.*

L'algorithme 8 permet l'affichage d'un cheminement optimal.

**Solution question 3.6 :** *Programmation des solutions*

**Solution question 3.7 :** *Écrire un programme C qui permette la mise en œuvre des*

**Algorithme 5 :** Recherche récursive du plus court cheminement dans les chaînes

**Données** : $n$ : la taille de la chaîne de montage
$a_{i,j}$ : temps de montage à $S_{i,j}$, $i = 1..2$ ,$j = 1..n$
$e_1$, $e_2$ : temps d'arrivée sur les chaînes 1 et 2
$x_1$, $x_2$ : temps de sortie des chaînes 1 et 2
$t_{1,j}$ : temps de transfert des chaînes 1 à 2 entre $S_{1,j}$ et $S_{2,j+1}$, $j = 1..n-1$
$t_{2,j}$ : temps de transfert des chaînes 2 à 1 entre $S_{2,j}$ et $S_{1,j+1}$, $j = 1..n-1$
**Résultat** : $l_{i,j}$ : les numéros de chaînes précédent l'arrivée en $(i, j)$
$f^*$ : plus court temps de traversée des chaînes de montage
$l^*$ : le numéro de chaîne de sortie du plus court cheminement

```
1 début
2     f_1 ← func1(a, e_1, e_2, t, n, l) + x_1
3     f_2 ← func2(a, e_1, e_2, t, n, l) + x_2
4     si f_1 ≤ f_2 alors
5         l* ← 1
6         f* ← f_1
7     sinon
8         l* ← 2
9         f* ← f_2
```

*solutions récursives et itératives présentées dans l'exercice précédent.*

Le programme C permettant d'exécuter la calcul de la solution optimale dans sa version récursive ou itérative est donnée sur la page de téléchargement de cette feuille d'exercices. Il suffit ensuite de compiler ce code avec un compilateur comme `gcc`, puis de lancer le programme. Lancé sans paramètre, le programme affiche les paramètres à utiliser.

**Solution question 3.8 :** *Comparer les exécutions des deux solutions. Jusqu'à quelle taille de chaîne de montage êtes vous allé avec la version récursive ? Quel temps cela a pris pour le calcul ? Quelle a été la durée du même calcul avec la version itérative issue de la programmation dynamique ? Conclure.*

**Algorithme 6 :** Fonctions $func1$ et $func2$

**Données** : $a_{i,j}$ : temps de montage au poste $S_{i,j}$, $i = 1..2$ ,$j = 1..n$
$e_1$, $e_2$ : temps d'arrivée sur les chaînes 1 et 2
$t_{1,j}$ : temps de transfert des chaînes 1 à 2 entre $S_{1,j}$ et $S_{2,j+1}$, $j = 1..n-1$
$t_{2,j}$ : temps de transfert des chaînes 2 à 1 entre $S_{2,j}$ et $S_{1,j+1}$, $j = 1..n-1$
$j$ : l'étage courant
**Résultat** : $l_{i,j}$ : chaîne de sortie du sous chemin optimal, $i = 1..2$ ,$j = 1..n$
$f^*$ : plus court temps de traversée des chaînes de montage

```
1  fonction func1(a, e1, e2, t, j, l)
2  début
3      si j = 1 alors f* ← e1 + a1,1
4      sinon
5          f1 ← func1(a, e1, e2, t, j − 1, l)
6          f2 ← func2(a, e1, e2, t, j − 1, l)
7          si f1 ≤ f2 + t2,j−1 alors
8              l1,j ← 1
9              f* ← f1 + a1,j
10         sinon
11             l1,j ← 2
12             f* ← f2 + t2,j−1 + a1,j
13 fonction func2(a, e1, e2, t, j, l)
14 début
15     si j = 1 alors f* ← e2 + a2,1
16     sinon
17         f1 ← func1(a, e1, e2, t, j − 1, l)
18         f2 ← func2(a, e1, e2, t, j − 1, l)
19         si f2 ≤ f1 + t1,j−1 alors
20             l2,j ← 2
21             f* ← f2 + a2,j
22         sinon
23             l2,j ← 1
24             f* ← f1 + t1,j−1 + a2,j
```

**Algorithme 7 :** Recherche itérative du plus court cheminement dans les chaînes

**Données** : $a[1..n]$ : temps de montage au poste $S_{i,j}$
$e_1$, $e_2$ : temps d'arrivée sur les chaînes 1 et 2
$x_1$, $x_2$ : temps de sortie des chaînes 1 et 2
$t_{1,j}$ temps de transfert des chaînes 1 à 2 entre $S_{1,j}$ et $S_{2,j+1}$, $j = 1..n-1$
$t_{2,j}$ : temps de transfert des chaînes 2 à 1 entre $S_{2,j}$ et $S_{1,j+1}$, $j = 1..n-1$
**Résultat** : $f^*$ : plus court temps de traversée des chaînes de montage
$l^*$ : le numéro de chaîne de sortie du plus court cheminement

```
1  début
2      f_1[1] ← e_1 + a_{1,1}
3      f_2[1] ← e_2 + a_{2,1}
4      pour j de 2 à n faire
5          si f_1[j−1] + a_{1,j} ≤ f_2[j−1] + t_{2,j−1} + a_{1,j} alors
6              f_1[j] ← f_1[j−1] + a_{1,j}
7              l_1[j] ← 1
8          sinon
9              f_1[j] ← f_2[j−1] + t_{2,j−1} + a_{1,j}
10             l_1[j] ← 2
11         si f_2[j−1] + a_{2,j} ≤ f_1[j−1] + t_{1,j−1} + a_{2,j} alors
12             f_2[j] ← f_2[j−1] + a_{2,j}
13             l_2[j] ← 2
14         sinon
15             f_2[j] ← f_1[j−1] + t_{1,j−1} + a_{2,j}
16             l_2[j] ← 1
17     si f_1[n] + x_1 ≤ f_2[n] + x_2 alors
18         f* ← f_1[n] + x_1
19         l* ← 1
20     sinon
21         f* ← f_2[n] + x_2
22         l* ← 2
```

**Algorithme 8 :** Affichage d'un cheminement optimal

**Données** : $l_i[j = 1..n]$ : le numéro de chaîne de sortie du plus court cheminement à l'étape $j$ avant la réalisation de la tâche $S_{i,j}$

```
1  début
2      i ← l*
3      afficher "chaîne" i "poste" n
4      pour j de n à 2 faire
5          i ← l_i[j]
6          afficher "chaîne" i "poste" j − 1
```

# Master 2 Informatique I2A

**Master DVL Master ITVL**

- Finance
- Histoire
- Géographie
- Informatique
- Mathématiques
- Sciences Pour l'Ingénieur
- Français Langue Étrangère
- Administration Économique et Sociale
- Diplôme d'accès aux études universitaires

**Master mention Informatique**

Parcours Informatique Avancé et Applications (I2A)

CTU
Centre de Télé-enseignement Universitaire
http://ctu.univ-fcomte.fr

## Filière Informatique

**VVI9MTGC**

Théorie des graphes et combinatoire

**Mr PHILIPPE - LAURENT**
*laurent.philippe@univ-fcomte.fr*

UNIVERSITÉ DE FRANCHE-COMTÉ
UBFC
UNIVERSITÉ BOURGOGNE FRANCHE-COMTÉ