Sujet : ordonnancement de flux de tâches

1 Contexte

Les problèmes d'ordonnancement ont été largement étudiés tant leurs applications sont nombreuses, comme l'ordonnancement des processus au niveau d'un processeur, la planification de transports ferroviaires ou l'agencement de tâches dans une entreprise. Il existe de nombreuses variantes de ces problèmes, dont une grande partie est NP-difficile.

Dans ce TP, on s'intéresse au cas particulier de l'ordonnancement d'un flux de tâches. Pour illustrer ce problème, considérons l'atelier du père Noël où trois lutins doivent préparer les cadeaux. Chaque lutin est spécialisé dans une seule opération : Frimousse doit mettre le jouet dans un carton, Pimprenelle doit emballer le carton en un paquet-cadeau et Gribouille doit aller ranger le cadeau sur les étagères en attente de livraison. Selon le jouet, ces opérations peuvent prendre un temps différent ; cependant, elles doivent toujours être faites dans le même ordre : on ne peut pas emballer le carton s'il n'y a pas le jouet dedans, par exemple. De plus, un cadeau ne peut pas en doubler un autre : tous les lutins s'occuperont des différents cadeaux toujours dans le même ordre.

Les lutins cherchent à minimiser le temps total nécessaire pour préparer tous les cadeaux. Par exemple en considérant la répartition des temps de la figure 1, on peut envisager les ordonnancements donnés par les figures 2 et 3. Le premier ordonnancement a une durée totale de 37 minutes, et le deuxième ordonnancement a une durée totale de 33 minutes. Ce dernier est optimal.

		Jouet (indice)			
		Ballon (0)	Puzzle(1)	Peluche (2)	Camion (3)
Lutin	Frimousse	6	3	5	9
	Pimprenelle	8	7	7	6
	Gribouille	2	8	5	3

Figure 1: Répartition des temps en minutes pour chaque opération

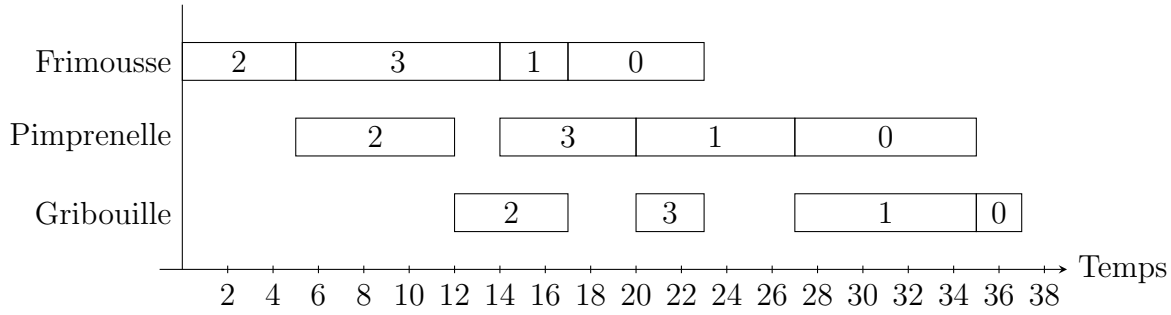

Figure 2: Représentation graphique de l'ordonnancement $\sigma_1 = (2, 3, 1, 0)$, de durée 37 minutes

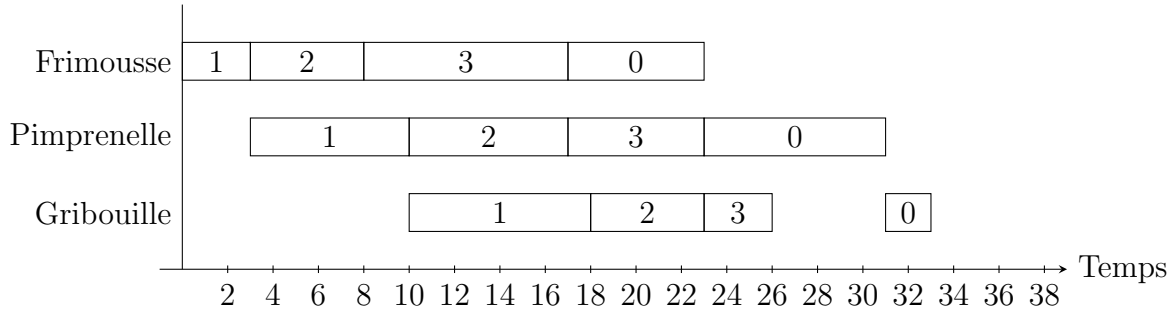

Figure 3: Représentation graphique de l'ordonnancement $\sigma_2 = (1, 2, 3, 0)$, de durée 33 minutes

Pour définir plus formellement le problème, on souhaite traiter des tâches sur des machines, sous les hypothèses suivantes :

- il y a n **tâches** (les cadeaux dans l'exemple, avec $n = 4$), d'indices compris entre 0 et $n - 1$;

- il y a m **machines** (les lutins dans l'exemple, avec $m = 3$), d'indices compris entre 0 et $m - 1$;
- une tâche $i \in \{0, \ldots, n-1\}$ est décomposée en m opérations ; pour traiter la tâche i, il faut traiter les m opérations, dans l'ordre (l'opération 0, puis l'opération 1, ...), en respectant la contrainte que l'opération j doit être traitée sur la machine j ;
- chaque opération a une durée définie à l'avance ; on notera $t_{i,j}$ la durée de l'opération $j \in \{0, \ldots, m-1\}$ de la tâche $i \in \{0, \ldots, n-1\}$; on supposera que les $t_{i,j}$ sont des entiers positifs ou nuls.
- une machine ne peut traiter qu'une seule opération à la fois. Lorsqu'une opération est commencée sur une machine, la machine doit terminer complètement cette opération avant de commencer à traiter une nouvelle opération ;
- une tâche ne peut pas en doubler une autre : cela signifie que sur chaque machine, les numéros de tâches des opérations traitées seront toujours dans le même ordre.

Étant donnée une instance du problème, un **ordonnancement** est une permutation σ de $\{0, \ldots, n-1\}$. Un ordonnancement $(\sigma(0), \sigma(1), \ldots, \sigma(n-1))$ s'interprète par le fait que la première tâche dont on exécutera l'opération sur chaque machine est celle d'indice $\sigma(0)$, la deuxième celle d'indice $\sigma(1)$, etc. Pour un ordonnancement σ, on définit les temps de **début** et de **fin** de l'opération j de la tâche i, notés respectivement $d_{i,j}$ et $f_{i,j}$, par :

- le temps de fin d'une opération est égal à son temps de début plus sa durée :

$$\text{pour } i \geqslant 0 \text{ et } j \geqslant 0, f_{i,j} = d_{i,j} + t_{i,j}$$

- pour commencer une opération d'une certaine tâche, il faut que l'opération précédente de la même tâche soit terminée, et que l'opération sur la même machine de la tâche précédente soit terminée :

$$\text{pour } i \geqslant 0 \text{ et } j \geqslant 0, d_{\sigma(i),j} = \max(f_{\sigma(i-1),j}, f_{\sigma(i),j-1})$$

On pose, par convention, $\sigma(-1) = -1$ et $f_{-1,j} = f_{i,-1} = 0$, pour que la formule précédente reste correcte pour $i = 0$ ou $j = 0$.

Par exemple, pour l'ordonnancement σ_1 de la figure 2, on aurait $d_{1,0} = 14$, $f_{1,0} = 17$, $d_{1,1} = 20$, $f_{1,1} = 27$, $d_{1,2} = 27$ et $f_{1,2} = 35$.

On cherche à minimiser le temps total des opérations, c'est-à-dire minimiser le temps de fin de la dernière opération sur la dernière machine : $f_{\sigma(n-1),m-1}$.

2 Cas à deux machines

Cette partie est à traiter en langage C.

Dans cette partie, on suppose qu'il n'y a que deux machines, c'est-à-dire que $m = 2$. Pour reprendre l'exemple précédent, la journée du 24 décembre, les cadeaux doivent être mis directement dans le sac du père Noël, donc seules Frimousse et Pimprenelle s'occupent des cadeaux. Avec les temps donnés par la figure 4, l'ordonnancement $(0, 1, 2, 3)$ aurait une durée totale de 24 minutes et l'ordonnancement $(1, 3, 2, 0)$ une durée optimale de 22 minutes.

		Jouet (indice)			
		Ballon (0)	Puzzle (1)	Peluche (2)	Camion (3)
Lutin	Frimousse	6	5	5	4
	Pimprenelle	2	4	3	4

Figure 4: Répartition des temps dans un cas à deux machines

2.1 Préliminaires

On représente une instance du problème par la donnée de l'entier `n` et de deux tableaux de taille `n` :

- `temps0` correspondant aux valeurs de $t_{i,0}$, pour $i \in \{0, \ldots, n-1\}$;
- `temps1` correspondant aux valeurs de $t_{i,1}$, pour $i \in \{0, \ldots, n-1\}$.

Un ordonnancement σ est donné par un tableau `sigma` de taille `n` contenant les valeurs $\sigma(0), \sigma(1), \ldots$

Question 1 Écrire une fonction `temps_total` qui détermine le temps total des opérations selon l'ordonnancement `sigma`.

```
int temps_total(int n, int* temps0, int* temps1, int* sigma)
```

2.2 Algorithme de Johnson

S'il existe une tâche dont l'opération est très courte sur la machine 0 et très longue sur la machine 1, il peut être intéressant de la programmer en premier dans l'ordonnancement : le temps court sur la machine 0 permettra de rendre active la machine 1 le plus vite possible, le temps long sur la machine 1 évitera des temps morts sur cette dernière lors de l'attente de la fin d'une opération sur la machine 0.

C'est cette idée qui est mise en œuvre dans l'algorithme de Johnson.

On dit qu'une suite de tâches d'indices $(0, \ldots, n-1)$ satisfait la **condition de Johnson** si et seulement si pour tout couple (i, j) vérifiant $0 \leqslant i < j < n$, on a :

$$\min(t_{i,0}, t_{j,1}) \leqslant \min(t_{i,1}, t_{j,0})$$

On admet que tout ordonnancement vérifiant cette condition est optimal pour le cas $m = 2$ machines.

On représente une tâche par le type de données :

```
struct Tache{
    int tm0;
    int tm1;
    int ind;
};
typedef struct Tache tache;
```

de telle sorte que si `tch` est un objet de type `tache` représentant la tâche d'indice i, alors `tch.tm0` (`tm` pour *temps machine*) vaut $t_{i,0}$, `tch.tm1` vaut $t_{i,1}$ et `tch.ind` vaut i.

Pour obtenir un ordonnancement vérifiant la condition de Johnson, on propose la méthode suivante :

- créer $I_0 = \{i \in [\![0, n-1]\!] \mid t_{i,0} < t_{i,1}\}$ et $I_1 = \{i \in [\![0, n-1]\!] \mid t_{i,0} \geqslant t_{i,1}\} = [\![0, n-1]\!] \setminus I_0$;
- trier I_0 par ordre croissant de $t_{i,0}$;
- trier I_1 par ordre décroissant de $t_{i,1}$;
- concaténer I_0 et I_1.

Question 2 Écrire une fonction `tri_taches` qui modifie en place un tableau de tâches pour qu'il respecte la condition de Johnson, en utilisant la méthode précédente.

```
void tri_taches(int n, tache* tab)
```

Question 3 En déduire une fonction `johnson` qui écrit un ordonnancement optimal dans le tableau `sigma` donné en argument.

```
void johnson(int n, int* temps1, int* temps2, int* sigma)
```

2.3 Approximation en cas de maintenance

Dans cette partie seulement, on s'intéresse à un cas particulier où la première des deux machines peut être mise en maintenance pendant les opérations (Frimousse prend une pause).

Dans la donnée du problème, on suppose connues deux valeurs entières x et y correspondant au début et à la fin de la maintenance respectivement. Aucune opération ne peut être continuée sur la machine 0 pendant l'intervalle de temps $[x, y]$. Cependant, l'opération qui était en cours de traitement au temps x peut continuer sans pénalité au temps y. Toutes les opérations qui devaient terminer à un temps $> x$ sur la machine 1 termineront alors à un temps décalé de $y - x$.

On admet que ce problème est NP-difficile.

Par exemple, considérons la répartition des temps de la figure 5, avec une pause pour Frimousse entre les temps $x = 10$ et $y = 18$. Pour l'ordonnancement $(0, 1, 3, 2)$ de la figure 6, on aurait une durée totale de 37 minutes. Pour l'ordonnancement $(3, 0, 2, 1)$ de la figure 7, on aurait une durée totale de 29 minutes, qui est optimale.

		Jouet (indice)			
		Ballon (0)	Puzzle (1)	Peluche (2)	Camion (3)
Lutin	Frimousse	6	7	2	3
	Pimprenelle	7	3	7	6

Figure 5: Répartition des temps dans un cas à deux machines

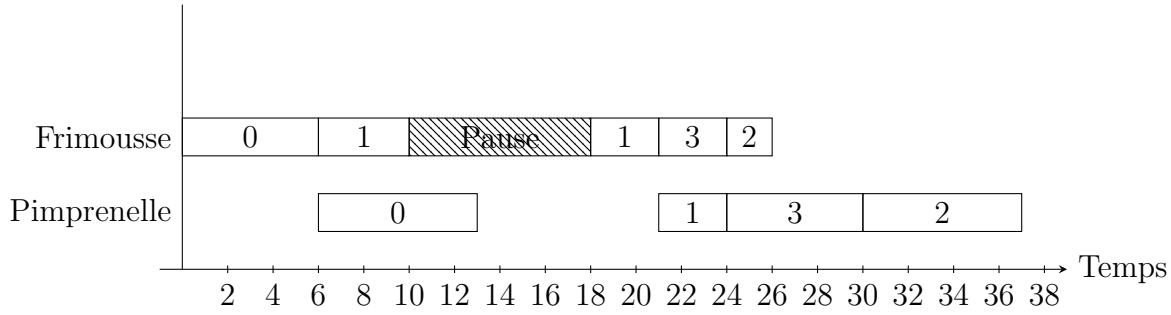

Figure 6: Représentation graphique de l'ordonnancement $(0, 1, 3, 2)$, de durée 37 minutes

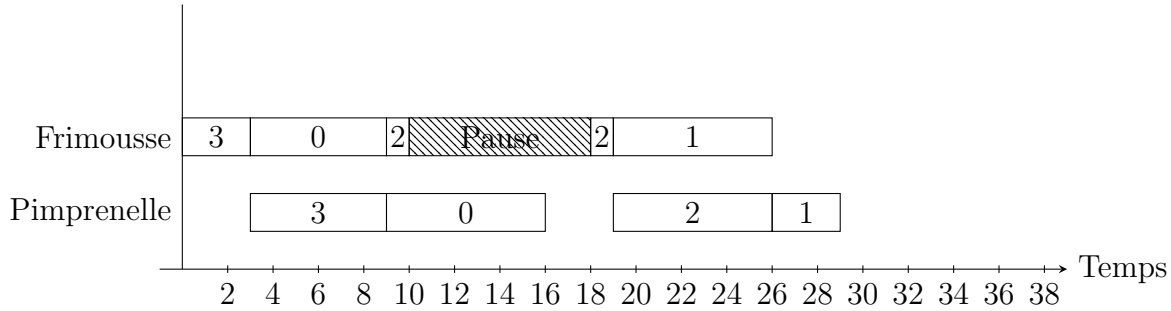

Figure 7: Représentation graphique de l'ordonnancement $(3, 0, 2, 1)$, de durée 29 minutes

Question 4 Adapter la fonction `temps_total` pour prendre en compte le temps de maintenance de la machine 1.

On se propose maintenant de construire une $\frac{3}{2}$-approximation du problème, en renvoyant l'ordonnancement ayant un temps minimal parmi les deux suivants :

- σ_1 un ordonnancement quelconque commençant par la tâche maximisant le temps de l'opération sur la deuxième machine ;
- σ_2 un ordonnancement correspondant aux tâches triées par ordre décroissant de $\dfrac{t_{i,1}}{t_{i,0}}$.

On admet qu'il s'agit effectivement d'une $\frac{3}{2}$-approximation.

Question 5 Écrire une fonction calculant une $\frac{3}{2}$-approximation du problème.

2.4 Solution exacte en programmation dynamique

On admet qu'il existe une solution optimale vérifiant la condition de Johnson.

Question 6 Écrire une fonction de programmation dynamique permettant de calculer le temps total minimal d'un ordonnancement. On pourra considérer $f(\ell, u_0, u_1, v_0, v_1)$ comme le temps total minimal des opérations lorsque :

- on ordonnance uniquement les tâches d'indices 0 à $\ell - 1$;
- la dernière tâche i dont l'opération sur la machine 0 termine avant x vérifie $f_{i,0} = u_0$ et $f_{i,1} \geqslant u_1$;
- la première tâche i' dont l'opération sur la machine 0 termine après x vérifie $d_{i',0} = v_0$ et $d_{i',1} = v_1$.

3 Cas général

Cette partie est à traiter en langage OCaml

On admet que pour $m > 2$, le problème est NP-difficile. On représente une instance du problème par la donnée d'une matrice `temps` de taille $n \times m$ telle que pour $i \in [\![0, n-1]\!]$ et $j \in [\![0, m-1]\!]$, `temps.(i).(j)` correspond à $t_{i,j}$. Comme dans la partie précédente, un ordonnancement est représenté par un tableau de taille n.

Question 7 Écrire une fonction `matrice_alea` qui crée une matrice d'entiers de dimensions $n \times m$ contenant des temps aléatoires compris entre 0 inclus et `tmax` exclu. On rappelle que `Random.int tmax` permet de renvoyer un entier choisi aléatoirement et uniformément entre 0 inclus et `tmax` exclu.

```
matrice_alea (n : int) (m : int) (tmax : int) : int array array
```

Question 8 Écrire une fonction `temps_total` calculant le temps total d'un ordonnancement.

```
temps_total (temps : int array array) (sigma : int array) : int
```

3.1 Une approche gloutonne

Question 9 Programmer une approche gloutonne pour résoudre le problème. Illustrer sa non-optimalité.

3.2 Algorithme *Branch and Bound*

On se propose d'implémenter une approche par séparation et évaluation (*Branch and Bound*) pour résoudre le problème. Pour cela, on considère des solutions partielles correspondant à des permutations de parties de $\{0, \ldots, n-1\}$. Par exemple, pour $n = 6$, la solution partielle $(2, 0, 3)$ signifie qu'on ordonnance les tâches 2, 0 et 3 uniquement. Une solution partielle sera complétée en rajoutant les tâches manquantes **après** celles qui sont présentes.

Question 10 Programmer une résolution du problème par retour sur trace (*backtracking*). On n'utilisera aucune heuristique de branchement ou d'évaluation pour cette fonction.

Pour améliorer l'exploration par retour sur trace, on se donne une heuristique calculée pour une solution partielle $\tilde{\sigma}$:

- pour chaque indice i n'apparaissant pas dans $\tilde{\sigma}$:
  - créer $\tilde{\sigma}_i$ en rajoutant la tâche i à $\tilde{\sigma}$;
  - déterminer le temps de fin de l'ordonnancement partiel $\tilde{\sigma}_i$;
  - ajouter la somme des temps des opérations **sur la dernière machine** des tâches n'apparaissant pas dans $\tilde{\sigma}_i$;
  - poser $B_i(\tilde{\sigma})$ cette valeur ;
- traiter récursivement les enfants $\tilde{\sigma}_i$ du nœud courant, par ordre croissant des $B_i(\tilde{\sigma})$, tant que cette valeur est strictement inférieure au plus petit majorant connu du temps total d'ordonnancement de toutes les tâches.

Question 11 Justifier que la plus petite valeur des $B_i(\tilde{\sigma})$ donne une heuristique admissible, c'est-à-dire qui minore la valeur minimale du temps total qu'il est possible d'obtenir en complétant la solution partielle.

Question 12 Programmer une résolution du problème par séparation et évaluation. Comparer à l'approche précédente.

4 Audit de code

Dans cette partie de l'épreuve, il est demandé au candidat d'étudier un fichier source qui comporte des erreurs ou des maladresses, de qualité de code ou de fonctionnement, et il est demandé d'auditer ce fichier, c'est-à-dire :

- de comprendre et d'être capable d'expliquer le fonctionnement du code à l'oral ;
- de proposer des corrections en réécrivant certaines parties afin de corriger les erreurs ou maladresses éventuelles et de rendre le code plus clair, notamment dans une optique pédagogique ;
- de proposer des améliorations de la complexité en temps ou en mémoire, ou de la sûreté du code.

Le temps indicatif de préparation de cette partie est d'une heure. Les fichiers sources sont `~/audit1.py` et `~/audit2.c` respectivement.

4.1 Code en Python

```
def aux(a, b):
    c = []
    i = j = 0, n1, n2 = len(a), len(b)
    for _ in range(n1 + n2):
        if a[i] <= b[j]:
            c.append(a[i])
            i += 1
        else:
            c.append(b[j])
    return c
```

```
def tri_merge(L):
    n = len(L)
    if n < 1:
        return L
    a, b = L[0::n // 2], L[n // 2::n]
    return aux(tri_merge(a), tri_merge(b))
```

4.2 Code en C

```
#include <stdlib.h>
#include <stdio.h>
#include <assert.h>
recherche(int tab[],int n,int x){
    int a=0,b=n-1;
    while(a<b){
        int c=(a+b)/2;
        if(tab[c]==x)
        return 1;
        if(tab[c]<x)
        a=c;
        else
        b=c;
    }
    return 0;
}
main(){
    int*tab=malloc(10*sizeof(int));
    tab[0]=2;tab[1]=4;tab[2]=5;tab[3]=7;tab[4]=11;
    tab[5]=12;tab[6]=15;tab[7]=20;tab[8]=22;tab[9]=28;
    assert(recherche(tab,10,15));
    assert(!recherche(tab,10,0));
    printf("Tout marche !");
}
```