

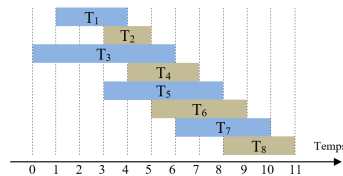
Programmation dynamique - Ordonnancement de tâches pondérées

Définition du problème - Weighted Interval Scheduling

On dispose d'une liste de tâches T_i pour $1 \leq i \leq n$.

Chaque tâche est définie par :

- d_i l'instant de début de la tâche T_i ;
- f_i l'instant de fin de la tâche T_i ;
- v_i la valeur associée à la tâche T_i .



Hypothèse : $f_1 \leq f_2 \leq \dots \leq f_n$ (si nécessaire, on trie la liste des tâches selon ce critère).

Critère de compatibilité entre deux tâches : deux tâches compatibles ne se chevauchent pas.

Objectif ✔ : déterminer le sous-ensemble de tâches compatibles pour lequel la valeur totale est maximum.

Applications

- Planning d'activités plus ou moins valorisées en maximisant la valorisation totale (une seule activité à la fois).
- Planification de réservation d'une ressource par différents utilisateurs en maximisant la durée d'utilisation (une seule réservation à la fois).

Bibliothèques

```
from _validation import *
from random import randint, seed
import numpy as np
import matplotlib.pyplot as plt
```

Analyse de l'exemple - Ebauche d'un algorithme

Si on aborde le problème avec une stratégie descendante (i.e. en partant de la tâche T_8), deux cas se présentent :

- soit T_8 fait partie de la solution optimale ;
- soit T_8 ne fait pas partie de la solution optimale.

Dans le premier cas, la tâche suivante à considérer est T_5 (T_6 et T_7 sont incompatibles avec T_8 car elles terminent après le début de T_8) et on itère le procédé avec T_5 .

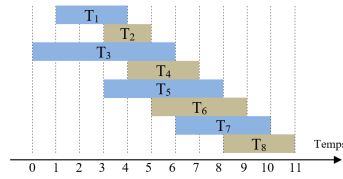
Dans le second cas, on itère ce même raisonnement avec T_7 (tâche précédent T_8).

L'analyse amène donc à définir un critère de compatibilité entre tâches.

Définition

On note $c(i)$ le nombre de tâches compatibles avec la tâche T_i .

$c(i)$ est donc le nombre de tâches se terminant avant que la tâche i ne commence ou encore $c(i)$ est le plus grand index $k < i$ tel que la tâche k termine avant que la tâche i ne débute i.e. $f_k \leq d_i$ puisque les tâches sont classées par ordre croissant de leur instant de fin.



 Sur l'énoncé papier de ce TP, remplir la ligne $c(i)$ dans le tableau ci-dessous (correspondant à l'exemple ci-dessus) en commençant par $i = 8$ conformément à la stratégie descendante.

i	1	2	3	4	5	6	7	8
$c(i)$								

Fonction $c(i)$: nombre de tâches compatibles avec la tâche T_i

 Compléter le code de la fonction `c(i)` (lignes 10, 13, 14 et 15) renvoyant le plus grand index $k < i$ tel que $f_k \leq d_i$. On considérera qu'une variable globale nommée `taches` a été définie sous la forme d'une liste de tuples de la forme : `taches = [(d1, f1, v1), (d2, f2, v2), ..., (dn, fn, vn)]` **supposée triée par valeurs f croissantes**.

```
def c(i):
    """
    Paramètres :
        i : tâche numéro i (int)

    Renvoie :
        k : plus grand index k < i tel que Tk soit compatible avec Ti (i.e fk <= di)
    """
    global taches
    di = 0 # Facultatif
    # Début de la tâche Ti
    # On cherche la plus grande valeur de k tq fk <= di donc boucle avec k croissant
    k = 0 # Initialisation de k
    while
        k
    return
```

Génération aléatoire des tâches et tri

On cherche à construire une liste de tâches (liste de tuples) de la forme :

`taches = [(0, 0, 0), (d1, f1, v1), (d2, f2, v2), ..., (dn, fn, vn)]` avec $d_i < f_i$ entiers (compris entre 0 et 24) et $1 \leq v \leq v_{max}$.

Tâches aléatoires

✎ Compléter le code (lignes 5, 8, 11, 13 et 15) de la fonction `tirage(n, vmax)` qui renvoie une liste de n tuples de la forme (d, f, v) de trois entiers aléatoires tels que $0 \leq d < f \leq 24$ et $1 \leq v \leq v_{max}$.

```
def tirage(n, vmax):
    taches = [(0, 0, 0)]          # Initialisation
    # Tirages aléatoires pour Ti : di, fi et vi
    for i in range(n):
        di, fi =
        # En cas d'égalité di = fi, on renouvelle le tirage de fi
        while di == fi:
            fi =
        # On doit avoir fi > di
        if di > fi: # Ne pas renouveler de tirage, solution simple !
            di, fi =
        # Valeur de la tâche Ti avec 1 <= vi <= vmax
        vi =
        # Une tâche Ti est définie comme un tuple Ti = (di, fi, vi)
        taches.append(
        )
    return taches
```

1 Algorithme : récursivité descendante avec choix binaire

Rappel objectif : déterminer le sous-ensemble de tâches compatibles pour lequel la valeur totale est maximum.

Cette valeur maximale ou *optimale* recherchée est notée $opt(n)$ où n est le nombre total de tâches.

Notation

On note $opt(i)$ la valeur optimale (i.e. maximum ici) du problème restreint aux tâches T_1 à T_i .

On considère le sous-problème restreint aux tâches T_1 à T_i .

Deux cas se présentent pour le calcul de $opt(i)$.

1er cas : la valeur optimale est obtenue avec la tâche i

$\Rightarrow$ les tâches $n^{\circ}c(i+1)$, $c(i+2)$, sont incompatibles car *par définition* $c(i)$ est l'index de la dernière tâche compatible avec T_i .

$\Rightarrow$ la valeur optimale est donc $v_i +$ valeur optimale pour les tâches compatibles précédentes, c'est-à-dire $v_i + opt(c(i))$ où $c(i) < i$ est le nombre de tâches compatibles avec la tâche i .

2nd cas : la valeur optimale est obtenue sans la tâche i

$\Rightarrow$ la valeur optimale doit donc être recherchée pour l'ensemble de toutes les tâches précédentes (autrement dit, c'est la solution optimale du sous-problème restreint aux tâches T_1 à T_{i-1}); cette valeur est $opt(i-1)$.

Finalement : $opt(i) = \begin{cases} 0 & \text{si } i = 0 \\ \max(v_i + opt(c(i)), opt(i-1)) & \text{sinon} \end{cases}$

1.1 Fonction $opt(i)$ « naïve » récursive

✎ Ecrire la fonction $opt(i)$ à partir de cet algorithme

$$opt(i) = \begin{cases} 0 & \text{si } i = 0 \\ \max(v_i + opt(c(i)), opt(i-1)) & \text{sinon} \end{cases}$$

```
def opt(i):
    global taches
```

Questions

1. Cet algorithme possède-t-il la propriété de sous-structure optimale ?
2. Les sous-problèmes se chevauchent-ils ?

Réponses

1/
2/

2 fonction `opt_mem(i)` récursive descendante utilisant la mémorisation

Rappel de la stratégie :

- on utilise un dictionnaire de la forme $\{j : opt(j)\}$ avec $0 \leq j \leq i$ pour stocker les valeurs calculées pour chaque tâche T_i ;
- avant de calculer un terme, on vérifie qu'il n'existe pas déjà dans le dictionnaire ;
- à chaque nouveau calcul, la résultat est stocké dans le dictionnaire.

Remarque : cette fonction renvoie la valeur maximum correspondant à une sélection de tâches parmi toutes les tâches mais ne fournit pas la sélection en question.

$$opt(i) = \begin{cases} 0 & \text{si } i = 0 \\ \max(v_i + opt(c(i)), opt(i - 1)) & \text{sinon} \end{cases}$$

 Deux manières d'écrire le code :

- Fonction récursive encapsulée avec le dictionnaire utilisé par la mémorisation dans une fonction.
- Fonction récursive avec paramètre mutable (le dictionnaire).

 Méthode 1 - Encapsulation.  Ecrire une fonction `opt_mem(i)` récursive descendante utilisant la mémorisation.

```
def opt_mem(i):                                # Fonction "d'encapsulation"
    """
    Paramètres :
        i : tâche numéro i (int)

    Renvoie :
        la plus grande valeur pour la somme des vi pour les tâches compatibles
    """
    memo = {}                                  # Dictionnaire pour la fonction opt_mem_aux
    def opt_mem_aux(i):
        global taches
        # Si le resultat cherché est déjà présent, on utilise le dictionnaire
        if i in memo:
            return memo[i]
        # Sinon, on fait le calcul et on mémorise la valeur dans le dictionnaire
        memo[i] = max(taches[i] + opt_mem_aux(c[i]), opt_mem_aux(i-1))
        return memo[i]
    return opt_mem_aux(i)
```

 Méthode 2 - Fonction avec paramètre mutable.

```
def opt_mem2(i, memo = {0:0}):                 # Fonction avec paramètre mutable (dictionnaire memo)
```

3 Fonction `opt_asc(i)` itérative ascendante

 Compléter le code de la fonction `opt_asc(i)` utilisant un algorithme itératif ascendant avec stockage des valeurs dans un dictionnaire.

Rappel de la stratégie : la fonction remplit un dictionnaire de la forme $\{j : opt(j)\}$.

```
def opt_asc(i):  
    """  
    Paramètres :  
        i : tâche numéro i (int)  
  
    Renvoie :  
        la plus grande valeur pour la somme des vi pour les tâches compatibles  
    """  
    # Initialisation du dictionnaire  
    memo = {}  
    # Construction ascendante du dictionnaire  
    for k in range(1, i+1):  
        memo[k] = 0  
    # Dernière valeur = valeur cherchée  
    return memo[i]
```

Remarque : ces algorithmes permettent d'obtenir la valeur maximale mais pas les tâches sélectionnées pour atteindre cette valeur.

4 Tâches sélectionnées

La fonction `taches_selectionnees` détermine l'ordonnancement des tâches à effectuer, elle renvoie la liste des index des tâches choisies dans la liste des tâches, par ordre croissant.

La première partie de l'algorithme repose sur celui de la fonction `opt_asc` dans lequel on mémorise dans `second` un dictionnaire, nommé `dico_choix`, la décision concernant la tâche k (optimise la valeur ou non) : `dico_choix[k] = True` ou `False`.

On pourrait penser qu'il suffit de sélectionner les tâches associées à la valeur `True` dans le dictionnaire mais ce raisonnement est incorrect car on ne sait pas encore à l'étape i si une étape ultérieure ne conduit pas à une valeur supérieure (autrement dit, une tâche sélectionnée à l'étape i peut être abandonnée au cours des évaluations ultérieures si elle devient moins intéressante).

Dans la seconde partie, l'algorithme parcourt donc le tableau à partir de la fin, en sautant de choix optimal en choix optimal.

$$opt(i) = \begin{cases} 0 & \text{si } i = 0 \\ \max(v_i + opt(c(i)), opt(i-1)) & \text{sinon} \end{cases}$$

 Compléter les lignes 16, 17, 19, 20, 24, 25, 28 et 30 du script ci-dessous.

```
def taches_selectionnees(i):  
    """  
    Paramètres :  
        i : tâche numéro i (int)  
  
    Renvoie :  
        L : liste des index des tâches choisies dans la liste des tâches, par ordre croissant  
    """  
    # ----- Première partie (basée sur opt_asc)  
    memo = {0:0}          # Dictionnaire pour le calcul de la valeur optimale  
    dico_choix = {}       # Dictionnaire pour la mémorisation des décisions  
    # Remplissage des dictionnaires  
    for k in range(1, i+1):  
        s = taches[k][2] + memo[c(k)]  
        if s > memo[k-1]:      # La tâche k est sélectionnée  
            memo[k] =  
            dico_choix[k] =  
        else:                  # La tâche k n'est pas sélectionnée  
            memo[k] =  
            dico_choix[k] =  
  
    # ----- Seconde partie (parcours des choix optimaux à partir du dernier)  
    L = []                  # Initialisation de la liste des tâches optimales  
    k =                     # Initialisation boucle while : index de la dernière tâche optimale  
    while k > 0:             # Parcours du dictionnaire des choix  
        if dico_choix[k] == : # La tâche k est sélectionnée  
            L.append(k)      # Ajout de la à la liste des tâches choisies  
            k =              # SAUT à la tâche compatible précédente  
        else:                # La tâche k n'est pas sélectionnée  
            k                 # On consulte la tâche précédente  
    L.reverse()              # Car la liste L est remplie de la dernière tâche à la première  
    return L
```

Remarques

1. Il est possible de n'utiliser qu'un seul dictionnaire de la forme $\{k : (opt(k), \text{True ou False})\}$.
2. Pour éviter `L.reverse()`, au lieu de `L.append(k)` faire `L = [k] + L` (ajout par l'avant).
3. Pour comprendre le fonctionnement, il peut être intéressant de visualiser `dico_choix` :
 - soit instruction `print(dico_choix)` juste avant `return` ;
 - soit `return L, dico_choix`.