Gloutonner

G. Aldon - J. Germoni - J.-M. Mény

mars 2012

Algorithme glouton

Le principe de l'algorithme glouton (greedy algorithm) :
faire toujours un choix localement optimal dans l'espoir que ce choix mènera à une solution globalement optimale.

Le problème du gymnase

Dans un gymnase doivent se dérouler une série d'épreuves dans une journée de 24h.
Pour chaque épreuve, la durée et l'heure de début d_i sont imposées (et donc l'heure de fin f_i est imposée également).

Le problème du gymnase

Dans un gymnase doivent se dérouler une série d'épreuves dans une journée de 24h.
Pour chaque épreuve, la durée et l'heure de début d_i sont imposées (et donc l'heure de fin f_i est imposée également).

Un problème pour l'organisateur : certains horaires se chevauchent et il ne peut pas y avoir deux épreuves en même temps (ne peuvent donc se dérouler dans la journée que des épreuves dont les intervalles $[d_i; f_i]$ sont disjoints).

Illustration

La fonction python ci-dessous génère des épreuves au hasard :

Python

```
import random
def epreuves(n) :
    T=[ ]
    for k in range(n) :
        d=random.randint(0,23)
        f=random.randint(1,24)
        while f<=d :
            f=random.randint(1,24)
        T.append((d,f))
    return T
```

L=epreuves(10); print L donne par exemple [(7, 21), (1, 7), (20, 22), (13, 19), (21, 22), (0, 13), (7, 16), (6, 17), (11, 22), (14, 15)]

Choix des épreuves

On trie la liste suivant un critère à définir et on choisit ensuite dans l'ordre : la première épreuve, puis la première qu'il lui est compatible, puis la première compatible avec celles déjà choisies . . .

Illustration du choix

Python

```
def choix(T) :
    C=[T[0]]
    Tri(T)
    for k in range(1,len(T)) :
        compatible=True
        for j in range(0,len(C)) :
            if T[k][0]<=C[j][1] and C[j][0]<=T[k][1] :
                compatible=False
                break
        if compatible : C.append(T[k])
    return C
```

Illustration du choix

En supprimant, pour l'instant, l'instruction de tri, on peut obtenir :
L= [(3, 14), (14, 23), (19, 20), (0, 21), (3, 14), (21, 24), (0, 1), (20, 23), (21, 24), (12, 14)]
et M= [(3, 14), (19, 20), (21, 24), (0, 1)]

Choix d'un tri

Les organisateurs proposent quatre tris différents des épreuves (voir plus loin) :

- Pour chacun des choix, convaincre un auditeur peu rigoureux (par exemple un élève !) du caractère "optimisant" du choix.
- Quels choix sont les meilleurs ? Certains choix sont-ils optimaux ?

Choix un : ordre croissant des durées

Python

```
def triUn(T) :
    if T==[ ] : return [ ]
    pivot=T[0]
    return triUn([x for x in T[1:] if (x[1]-x[0]<=pivot[1]-pivot[0])])+[pivot]+triUn([x for x in T[1:] if (x[1]-x[0]>pivot[1]-pivot[0])])
```

Exemple de résultat :
Liste = [(15, 22), (2, 9), (9, 22), (9, 23), (23, 24), (11, 13), (8, 14), (6, 14), (9, 21), (11, 18)]
Après le tri :[(23, 24), (11, 13), (8, 14), (11, 18), (2, 9), (15, 22), (6, 14), (9, 21), (9, 22), (9, 23)]
Choix des épreuves : [(23, 24), (11, 13), (2, 9), (15, 22)]

Choix deux : ordre croissant des heures de début

Python

```
def triDeux(T) :
    if T==[ ] : return [ ]
    pivot=T[0]
    return triDeux([x for x in T[1:] if (x[0]<=pivot[0])])+[pivot]+triDeux([x for x in T[1:] if (x[0]>pivot[0])])
```

Exemple de résultat :
Liste = [(0, 23), (5, 20), (5, 8), (20, 23), (20, 24), (14, 16), (6, 17), (3, 5), (17, 23), (0, 8)]
Liste triée = [(0, 8), (0, 23), (3, 5), (5, 8), (5, 20), (6, 17), (14, 16), (17, 23), (20, 24), (20, 23)]
Choix des épreuves =[(0, 8), (14, 16), (17, 23)]

Choix trois : ordre croissant des incompatibilités

On définit tout d'abord une fonction retournant la liste des incompatibilités (où l'incompatibilité d'une épreuve $\mathcal{E}$ est le nombre d'épreuves dont l'intervalle a une intersection non vide avec celui de l'épreuve $\mathcal{E}$).

Python

```
def incompatibilite(T) :
    A=[0 for i in range(len(T))]
    for k in range(len(T)-1) :
        for j in range(k+1,len(T)) :
            if T[k][0]<=T[j][1] and T[j][0]<=T[k][1] :
                A[k]+=1
                A[j]+=1
    return [(T[k],A[k]) for k in range(len(T))]
```

Choix trois : ordre croissant des incompatibilités

Et on trie :

Python

```
def triTrois(T) :
    J=incompatibilite(T)
    def tri(J) :
        if J==[] : return []
            pivot=J[0]
            return tri([x for x in J[1:] if (x[1]<=pivot[1])])+[pivot]+tri([x
for x in J[1:] if (x[1]>pivot[1])])
    J=tri(J)
    return [x[0] for x in J]
```

Choix trois : ordre croissant des incompatibilités

Exemple de résultat :

- Liste des épreuves : [(13, 17), (13, 17), (5, 16), (20, 22), (4, 16), (16, 21), (3, 11), (20, 24), (19, 20), (23, 24)]
- Ajout des degrés d'incompatibilité : [((13, 17), 4), ((13, 17), 4), ((5, 16), 5), ((20, 22), 3), ((4, 16), 5), ((16, 21), 7), ((3, 11), 2), ((20, 24), 4), ((19, 20), 3), ((23, 24), 1)]
- Tri : [(23, 24), (3, 11), (19, 20), (20, 22), (20, 24), (13, 17), (13, 17), (4, 16), (5, 16), (16, 21)]
- Choix des épreuves : [(23, 24), (3, 11), (19, 20), (13, 17)]

Choix quatre : ordre croissant des heures de fin

Python

```
def triQ(T) :
    if T==[ ] : return [ ]
    pivot=T[0]
    return triQ([x for x in T[1:] if (x[1]<=pivot[1])])+[pivot]+triQ([x
for x in T[1:] if (x[1]>pivot[1])])
```

Exemple de résultat :
Liste = [(4, 10), (12, 13), (4, 17), (6, 16), (15, 20), (22, 24), (8, 18), (4, 8), (3, 18), (9, 10)]
Liste triée = [(4, 8), (9, 10), (4, 10), (12, 13), (6, 16), (4, 17), (3, 18), (8, 18), (15, 20), (22, 24)]
Choix des épreuves =[(4, 8), (9, 10), (12, 13), (15, 20), (22, 24)]

Choix cinq : ordre décroissant des heures de début

Python

```
def triC(T) :
    if T==[ ] : return [ ]
    pivot=T[0]
    return triC([x for x in T[1:] if (x[0]>pivot[0])])+[pivot]+triC([x for
x in T[1:] if (x[0]<=pivot[0])])
```

Exemple de résultat :
Liste = [(12, 22), (17, 19), (12, 21), (8, 16), (10, 24), (20, 21), (14, 24), (6, 12), (11, 20), (3, 16)]
Liste triée = [(20, 21), (17, 19), (14, 24), (12, 22), (12, 21), (11, 20), (10, 24), (8, 16), (6, 12), (3, 16)]
Choix des épreuves =[(20, 21), (17, 19), (8, 16)]

Des présentations convaincantes ?

1. ordre croissant des durées :
2. ordre croissant des heures de début :
3. ordre croissant des incompatibilités :
4. ordre croissant des heures de fin :
5. ordre décroissant des heures de début :

.

Des présentations convaincantes ?

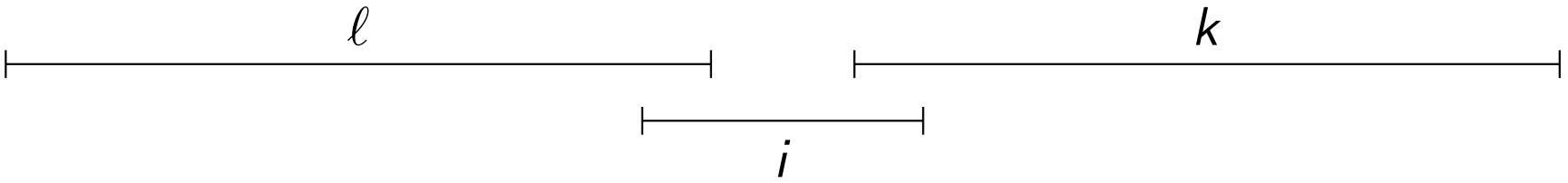
1. ordre croissant des durées : en choisissant de caser d'abord l'épreuve la plus courte, on laisse un maximum de temps disponible pour les autres épreuves.
2. ordre croissant des heures de début : en commençant le plus tôt possible, on gaspille le moins de temps possible.
3. ordre croissant des incompatibilités : en retardant les épreuves qui en gênent beaucoup d'autres, on se laisse le plus possible de compatibilités.
4. ordre croissant des heures de fin : en terminant au plus tôt la première épreuve, on laisse après elle un maximum de temps.
5. ordre décroissant des heures de début : en retardant le plus possible le début de la dernière épreuve, on laisse un maximum de temps avant elle.

.

Des heuristiques gloutonnes qui fonctionnent

Ordre croissant des durées : non optimal.

Contre-exemple :

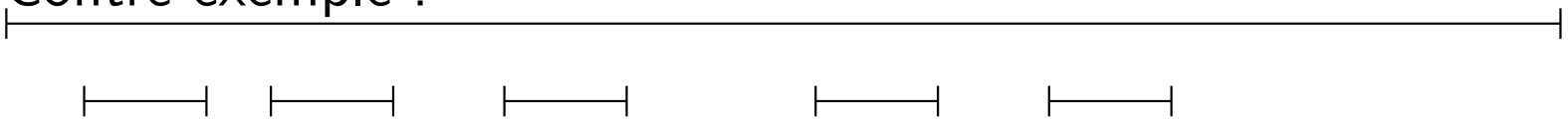


L'algorithme choisit i : une épreuve alors que deux sont possibles.

Des heuristiques gloutonnes qui fonctionnent parfois

Ordre croissant des heures de début : non optimal.

Contre-exemple :

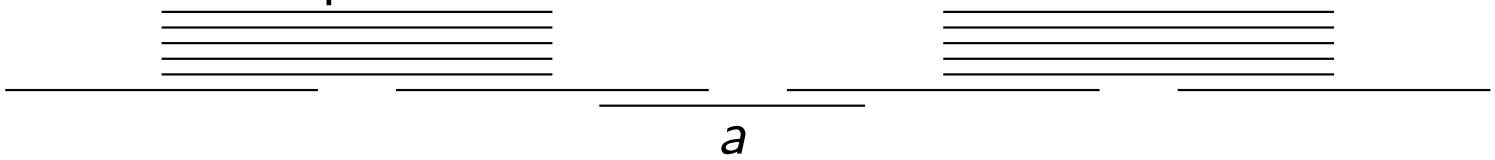


L'algorithme choisit une épreuve au lieu de plusieurs.

Des heuristiques gloutonnes qui fonctionnent parfois

Ordre croissant des incompatibilités : non optimal.

Contre-exemple :



L'algorithme choisit a et trois événements seront choisis au total. Alors que l'on peut clairement en choisir 4.

Des heuristiques gloutonnes qui fonctionnent parfois

Ordre croissant des heures de fin : optimal.

On note f la date de fin la plus petite.
Soit $\Gamma = \{f_1, f_2, \ldots, f_k\}$ un ensemble de dates de fin d'épreuves correspondant à une solution optimale avec $f_1 < f_2 < \cdots < f_k$.

Si $f \neq f_1$, on remplace une épreuve correspondant à f_1 par une épreuve correspondant à f, cela est possible puisque $f \leq f_1 < f_2 < \ldots$. L'ensemble $\Gamma' = \{f, f_2, \ldots, f_k\}$ est donc également optimal (même nombre d'épreuves).

Des heuristiques gloutonnes qui fonctionnent parfois

On note ensuite f' la date de fin la plus petite parmi les dates de fin d'épreuves compatibles avec f (c'est à dire les épreuves de date de début $> f$).

Si $f_2 \neq f'$, on remplace une épreuve correspondant à f_2 par une épreuve correspondant à f', cela est possible puisque f' compatible avec f et $f' \leq f_2 < \ldots$. L'ensemble $\Gamma'' = \{f, f', \ldots, f_k\}$ est donc également optimal (même nombre d'épreuves).

...

Des heuristiques gloutonnes qui fonctionnent parfois

Ordre décroissant des heures de début : optimal.
Même raisonnement.