ALGORITHMES GLOUTONS

1 Ordonnancement d'intervalles (interval scheduling)

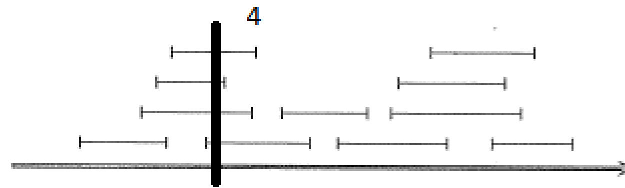

FIGURE 1 – Exemple d'ensemble de requêtes

Notez qu'un algorithme glouton procède comme suit : Étant données une solution (partielle) courante et un sous problème il effectue un choix arbitraire, mets la solution partielle et le sous problème à jour.

Dans le cas des intervales :

1. La solution partielle courante est un ensemble d'intervales selectionnés. par construction ceux ci sont sans conflit.
2. Le sous problème A est un ensemble d'intervales dit *Actifs* restant possibles, c est à dire n'intersectant pas ceux de la solution.
3. Une heuristique détermine quel intervale actif de A est choisi.

Un exemple naif en python est présenté ci dessous.

1. ligne 25-32 on initialise, l'ensemble de intervales actif est celui de tout les intervales et la solution est vide,
2. le while de la ligne 33 choisi et ajouter intervale tant qu'il reste des intervales actifs.
3. la fonction utilisée ligne 36 peut être remplacée par tout autre fonction choisssant un intervale actif : pex le plus court, un interval au hasard, le plus cher etc ...
4. la ligne 43 met à jour la solution en y ajoutant l'intervale choisi.
5. les lignes 48-53 mettent à jour la liste de intervales actifs, de fait ceux intersectant l'intervale choisi sont desactivés.

```
1    from random import randint
2
3  class Interval:
4      def __init__(self,d,f,v):
5          self.debut=d
6          self.fin=f
7          self.valeur=v
8
9      def Intersect(self, otherInt):
10         return( (self.debut <= otherInt.fin) and (self.fin >= otherInt.debut))
```

```
11
12      def __str__(self):
13          return ("[{},{}] val={}".format(self.debut,self.fin,self.valeur) )
14
15  def GetEarliest(Dict_of_Interval, Active):
16      earliest=+10**6
17      for mykey  in Active:
18          theInter= Dict_of_Interval[mykey]
19          if theInter.debut < earliest:
20              earliest= theInter.debut
21              bestid= mykey
22      return bestid
23
24  def GreedyEarliest(L):
25      Active=list(L.keys())
26      # Active est l'ensemble des Intervales actif,
27      # plus exactement Active contient les clefs/id des intervales actifs
28      # Au debut tout les intervales sont actif
29
30      Solution= []
31      # La solution courante est vide
32
33      while len(Active) >0 :
34      ## Tnat qu'il reste des intervals actifs
35
36          MonChoixGlouton = GetEarliest(L,Active)
37
38          print("k", MonChoixGlouton,  L[MonChoixGlouton])
39          # MonChoixGlouton contient la clef de l intervale choisi
40          #On ajoute l'intervale à la solution
41
42          Interval_choisi= L[MonChoixGlouton]
43          Solution += [MonChoixGlouton]
44
45          print ("interval numero {} =  {} est choisi ".format(MonChoixGlouton ,
                Interval_choisi))
46
47          ## Il faut maintenant supprimer les intervales actifs qui intersecte l'
                intervale choisit
48          Next=[]
49          for index in Active :
50              if not ( Interval_choisi.Intersect( L[index])):
51                  ## ici L[index] n est pas en conflit avec Interval_choisi
52                  Next+=[index]
53          Active=list(Next)
54      return(Solution)
```

Un petit generateur d'instances :

```
1
2
3   def RandomInstance(nint, largeur):
4       #Return a dict on nint interval in 0,largeur
5       Res={}
6       for interkey in range(nint):
7           deb =randint(0,largeur)
8           fin =randint(0,largeur)
9           if (fin < deb):
```

```
10            (deb,fin)=(fin,deb)
11        Res[interkey]= Interval(deb,fin, randint(0,5))
12    return(Res)
```

Remarque 1 *1. L'algorithme n est pas très efficace, il est écrit de facon générique de sorte que l'on puisse remplacer la fonction de choix par une autre.*

2. *Étant donné que l'on choisit à chaque étape l'intervale qui commence au plus tot, on pourrait au départ trier les intervales par date de début croissante et ensuite à chaque fois selectionner l'intervale Actif minimum. Avec un* tas *cette selection pourrait s'effectuer en* $\Theta(1)$*. De même on pourrait aussi simplement parcourir la liste des intervales en* conflit *avec l'intervale selectionné et supprimer ceux ci, etc ...*
3. *Pour modifier l'algorithme et utiliser l'intervale le plus court (smaller) ou celui qui se termine le plus tot il suffit de remplacer GetEarliest par une fonction analogue. Il suffit en fait de modifier la méthode qui compare deux intervale, dans GetEarliest on compare le début des intervales, mais on pourrait utiliser tout ordre sur les intervales.*
4. *Pour getlessconflict c'est un peu plus compliqué. La solution naive (mais lente) consiste à calculer à chaque fois le nombre de conflits.*

Calculer le nombre de conflits

Compter le nombre de conflits en un point revient simplement à savoir combien d'intervales sont *ouverts* en ce point (ie ont commencé mais ne sont pas terminés). C'est presque comme compter des parenthèses ouvertes moins des parenthèses fermées. Il faut cependant savoir ce qu'est un *point*. Si il y a deux intervales $[1, 80]$ et $[60, 210]$ il y a just 4 points $1, 60, 80, 210$.

Pour le faire on commence donc par générer l'ensemble de toutes les extrémités d'intervales. On peut trier ces extrémités de gauche à droite et considérer que l'on a n extermités $e[0] = 0, e[1], e[2] ... e[n]$. On détermine alors deux quantités $debut[i]$ et $fin[i]$ qui sont respectivement le nombre d'intervales commencant en $e[i]$ et le nombre d'intervales se terminant en $t[i]$.

En pratique on va souvent confondre l extermité $e[i]$ et i, les extremités sont identifié par le point ù elle sont.

La *charge en* i peut être alors calculée inductivement ainsi :

$$charge[e[i]] = charge[e[i-1]] - fin[e[i-1]] + debut[i], i >= 1, charge[0] = s[0]$$

Autrement dit

— quand un intervale se termine la charge diminue pour les points (strictement) suivants.

— quand un intervale se commence la charge augment pour le point courant et les points suivants.

Ici un exemple de programme calculant le nombre de conflits en toute extremite. On peut bien sur utiliser un programme plus simple qui pour chaque extremité parcoure tout les intervales mais la complexité est supérieure. Dans le programme suivant les intervales ne sont parcourus que deux fois, une fois pour determiner les extremités, une seconde fois pour déterminer la charge.

Rappel : En Python $a.get(c, def)$ retourne $a[c]$ si la clef c existe dans a et def sinon.

```
1        def FindLoad(L):
2     # determine la charge de chaque extremite d'un ensemble d'intervale
3     debut={}
4     fin={}
5     Load={}
6     #debut[ex] et fin[ex] vont contenir le nombre d' intervales qui
7     #s arretent (debutent) en l extremite de nom ex
8     #le nom de l extremite est tout simplement le nombre (a priori entier) ou
          elle est situee
9
10    for index, inter in L.items():
11        ## Pour tout les intervales on incremente de 1
12        ##  la valeur de la clef de debut de son debut et idem pour la valeur de
              la clef de fin
13        debut[inter.debut]= debut.get(inter.debut,0)+1
14        fin[inter.fin]= fin.get(inter.fin,0)+1
15
16    #ceci est l ensemble des extremite
17    extremites= set(debut.keys() ) | set(fin.keys())
18    #on utilise un ensemble pour eviter d avoir une extremite deux fois
19
20    # attention ici extremites est normalement trie
21    # si tel n etait pas le cas on peut ajouter
22    # extremite= list(extremite)
23    # extremite.sort()
24    currentload=0
25    pred=-1
26    # pred est le nom de l extremite precedente
27
28    for ex in extremites:
29        currentload+= debut.get(ex,0) - fin.get(pred,0)
30        # on augmente la charge par le nombre d' intervales qui debutent en ex
31        # on diminue par nombre d' intervales qui s arretaient 'a l extremite
              precedente/
32        Load[ex]=currentload
33        print (" extremite {} , start {}, stop {} load {}" .format(ex, debut.get(
              ex,0),fin.get(pred,0) ,currentload))
34        pred=ex
```

Attention :

La charge d'un intervale n'est :

- Ni son degré (ie le nombre d'intervales avec lequel il est en conflit)
- Ni la charge de son extremité de début.
- Ni la charge de son extremité de fin.

La charge pour un intervale $[a, b]$ et le maximum de $charge[i]$ pour i une extremite située entre dans $[a, b]$.

Ici une facon peu efficace de determiner la charge des intervales, ce morceau de code vient à la suite du précédent. Ainsi L est un dictionnaire d'intervales et $ex, Load$ sont respectivement la liste des extremités et le dictionnaire des charges.

```
1       # on peut proceder comme pour les extremite
2     Iload={}
3     for interID, inter  in L.items():
```

```
4        interExtremites=[ ex for ex in extremites if ex >= inter.debut and ex <=
             inter.fin]
5        Iload[interID] = max ([ Load[x] for x in interExtremites])
6
7
8    for interID,inter in L.items():
9        print (inter, "load", Iload[interID])
10   return (Iload)
```