TD n°15 – Corrigé

Exercice 1 *Ordonnancement d'intervalles*

1. - *Montrer que les deux premières règles peuvent ne pas fournir une solution optimale. (Il suffit de trouver un exemple où le S' obtenu n'est pas le plus grand possible).*
   Heuristique 1 : un contre exemple est $\{[1,4];[2,3];[3,4]\}$. L'algorithme glouton choisit $\{[1,4]\}$ et ensuite il ne peut plus rien choisir. La solution optimale est $\{[2,3];[3,4]\}$.
   - Heuristique 2 : un contre exemple est $\{[3,5];[1,4];[4;7]\}$. L'algorithme glouton choisit $\{[3,5]\}$ et ensuite il ne peut plus rien choisir. La solution optimale est $\{[1,4];[4,7]\}$.

On note $G = (i_1, ..., i_m)$ avec $\forall j \in [|1,m|], 0 \leq i_j \leq n-1$ la famille d'intervalles choisis par un algorithme glouton selon la troisième règle. On les considère dans l'ordre de choix, c'est à dire ordonnés par horaire de fin croissant.

On considère un autre ensemble quelconque $H = (j_1, ..., j_p)$ d'intervalles de S deux à deux disjoints, avec $p \geq m$. Cela signifie que H est une solution qui est aussi bien ou meilleure que G

2. *Montrer que pour tout $1 \leq r \leq m$, $f_{i_r} \leq f_{j_r}$.*
   On montre par récurrence sur p, $\forall 1 \leq r \leq min(m,p)$, $f_{i_r} \leq f_{j_r}$.
   - Cas $p = 1$: on remarque que i_1 est nécessairement l'intervalle de date de fin la plus petite parmi tous les intervalles disponibles. Donc $f_{i_1} \leq f_{j_1}$.
   - On considère que la propriété est vraie pour tout ensemble H de p intervalles avec $p < n$. On veut montrer la propriété pour $p+1$. Soit $H = (j_1, ..., j_{p+1})$, alors $H' = (j_1, ..., j_p)$ vérifie l'hypothèse de récurrence, d'où $1 \leq r \leq min(m,p)$, $f_{i_r} \leq f_{j_r}$.
     Si $p \geq m$, alors on déduit immédiatement la propriété pour $p+1$.
     Sinon, on doit juste montrer que $f_{i_{p+1}} \leq f_{j_{p+1}}$. Puisque i_{p+1} a été choisi par l'algorithme glouton, on sait que $f_{i_{p+1}}$ est minimal parmi l'ensemble A des dates de fin des intervalles qui ne s'intersectent pas avec $i_1, ..., i_p$. On sait également, puisque $f_{j_p} < f_{j_{p+1}}$ et que j_p et j_{p+1} ne s'intersectent pas, que $f_{j_p} < d_{j_{p+1}}$, ce qui entraine que $f_{i_p} < d_{j_{p+1}}$ et donc que $\forall 1 \leq r \leq p$, $f_{i_r} < d_{j_{p+1}}$.
     Ainsi j_{p+1} est disjoint de $i_1, ..., i_p$, donc par minimalité de $f_{i_{p+1}}$, on a $f_{i_{p+1}} \leq f_{j_{p+1}}$. On a prouvé l'hérédité.
3. *En déduire que $m = p$, c'est à dire que H ne peut pas être meilleure que G. Conclure sur la 3ème règle.*
   D'après la preuve précédente, toute solution $H = (j_1, ..., j_p)$ pour $p \geq m$ vérifie pour tout $1 \leq r \leq m$, $f_{i_r} \leq f_{j_r}$.
   Si $p > m$, c'est à dire que la solution H est meilleure que celle donnée par le glouton, alors j_{m+1} est un intervalle disjoint des autres $j_1, ..., j_m$. En particulier j_{m+1} est un intervalle disjoint de j_m et $f_{j_m} < f_{j_{m+1}}$. Donc $f_{j_m} < d_{j_{m+1}}$.
   De plus, comme $f_{i_m} < f_{j_m}$, on a $f_{i_m} < d_{j_{m+1}}$. Comme les intervalles de G sont ordonnés par ordre de fin, on peut généraliser $\forall 1 \leq r \leq m$, $f_{i_r} < d_{j_{m+1}}$. Ainsi j_{m+1} est un intervalle disjoint de tous les i_r. Donc il devrait être ajouté par l'algorithme glouton. Contradiction.
   Donc on ne peut trouver une solution avec plus de m intervalles, l'algorithme glouton est optimal.

On va faire une implémentation en Ocaml

4. *Proposer un type Ocaml pour stocker chacun des intervalles i.*
   On propose `int*int list` : une liste des couples d_i, f_i.
5. *Écrire une fonction `disjoints` qui détermine si deux intervalles sont disjoints.*
   Écrire une fonction `disjoints` qui détermine si deux intervalles sont disjoints

```
let disjoints (d1,f1) (d2,f2) =
   (f1<d2) || (f2<d1);;
```

6. *Écrire une fonction qui trie une liste d'intervalles selon la troisième règle.*

```
let compare_fin (a,b) (c,d) = %On trie par la date de fin. Dans le doute l'ordre est total.
 if a = c && b = d then 0
 else if b > d || (b=d && a>c) then 1
 else -1;;

 let tri_glouton4 liste =
   List.sort compare_fin liste;;
```

7. *Écrire l'algorithme glouton complet.*

```
let glouton4 liste =
 let rec prochain liste prec = (*Entrées : la liste des intervalles encore possibles, rangé*)
 (*dans l'ordre de fin et prec le dernier intervalle choisi*)
 (*Cette fonction passe en revue les intervalles et sélectionne le prochain disjoint des précédents*)
```

```
(*On remarque qu'il suffit d'être disjoint DU précédent pour être disjoint de tous les précédents*)
  match liste with
  |[] -> []
  |h::q -> if disjoints prec h then h::(prochain liste h)
           else prochain q prec
  in
  (*On trie la liste dans le bon ordre et on sélectionne immédiatement le premier *)
  let h::q = tri_glouton4 liste in h :: (prochain q h);;
```

8. *En considérant que `List.sort` s'effectue en temps optimal, quelle est la complexité de cette méthode ?*
   On a un $O(n\log(n))$ pour le tri puis la fonction ne fait que parcourir la liste et lui appliquer des opérations élémentaires. Donc `prochain` s'effectue en $O(n)$, qui est dominé par le $O(n\log(n))$.

Exercice 2 *Partitionnement d'intervalles*

1. *Montrer que peu importe l'ordre choisi pour la première étape, cette méthode renvoie une partition telle que les intervalles d'une même partie sont disjoints.*
   On fait une récurrence sur i.
   Avant la boucle ($i = 1$) la seule partie déclarée est la partie 1 qui contient r_1. Cette partie est donc constituée d'intervalles disjoints.
   Supposons que la propriété soit vraie à la fin de la boucle i. On montre la propriété pour $i+1$. Pendant l'itération $i+1$, on a deux choix possibles :
   - Soit on met r_{i+1} dans une partie d déjà existante, uniquement si r_{i+1} est disjoint de tous les éléments de la partie d. Alors à la fin de l'itération $i+1$ la partie d n'a que des intervalles disjoints, et comme on a pas touché aux autres parties, par hypothèse de récurrence elles contiennent des intervalles disjoints.
   - Soit on met r_{i+1} dans une nouvelle partie d. Alors cette nouvelle partie ne contient que des intervalles disjoints, et par hypothèse de récurrence, toutes les autres parties vérifient la propriété.
2. *Exhiber un ordre de traitement à la première étape qui ne renvoie pas la solution optimale sur une entrée particulière.*
   On considère l'ensemble d'intervalles $\{]7;9[,]1;5[,]1;2[,]4;6[,]5;8[\}$.
   On va les considérer dans l'ordre dans lequel ils sont écrits.
   Alors l'algorithme crée les parties suivantes :
   $\{]7;9[\} \to \{]7;9[,]1;5[\} \to \{]7;9[,]1;5[\}$ et $\{]1;2[\} \to \{]7;9[,]1;5[\}$ et $\{]1;2[,]4;6[\} \to \{]7;9[,]1;5[\}$ et $\{]1;2[,]4;6[\}$ et $\{]5;8[\}$
   Or une solution optimale est $\{]5;8[,]1;5[\}$ et $\{]1;2[,]4;6[,]7;9[\}$.
3. *On suppose désormais que le tri des requêtes se fait par ordre de début croissant. Montrer que la solution renvoyée est optimale.*
   On considère que la solution renvoyée par la méthode gloutonne quand les intervalles sont triés par date de début croissante est une partition $\mathcal{P}$ en k classes. Pour $\alpha \in [|1,k|]$, on note n_α la taille de la classe α et pour tout $\beta \in [|1,n_\alpha|]$, on note i_α^β le β-ième intervalle de la classe α, quand on les prend par date de début croissante. De plus les classes sont numérotées dans l'ordre de leur création, donc si on considère les i_α^1, ils sont également rangés par ordre croissant de date de début.
   On suppose par l'absurde que ce n'est pas la solution optimale. Il existe donc une partition $\mathcal{P}'$ en $l < k$ ensembles.
   On considère l'intervalle i_{l+1}^1 qui le premier de la classe $l+1$. Lors de l'étude de cet intervalle, l'algorithme a crée une nouvelle classe. Cela signifie qu'il existe $i_1^{\beta_1}, ..., i_l^{\beta_l}$ des éléments de toutes les classes précédentes qui sont tous non disjoints de i_{l+1}^1 et qui commencent avant.
   La famille $i_{l+1}^1, i_1^{\beta_1}, ..., i_l^{\beta_l}$ contient $l+1$ éléments, donc par principe des tiroirs, deux de ces éléments sont dans la même classe de $\mathcal{P}'$. Puisque i_{l+1}^1 ne peut pas être dans la même classe qu'un autre élément de la famille, il existe $a < b \in [|1,l|]$ tels que $i_a^{\beta_a}$ et $i_b^{\beta_b}$ sont dans la même classe de $\mathcal{P}'$. En particulier ils doivent être disjoints, ce qui signifie que $d_a \le f_a \le d_b \le f_b$ **(1)** (pour simplifier les notations, d_a et f_a sont la date de début et la date de fin de $i_a^{\beta_a}$ et pareil avec b)
   De plus $i_a^{\beta_a}$ commence avant i_{l+1}^1 et ils sont non disjoints, ce qui nous indique $d_a \le d_{l+1}^1 \le f_a$ **(2)**. De même $d_b \le d_{l+1}^1 \le f_b$ **(3)**.
   Or, en combinant **(1)** et **(3)**, on obtient que $d_a \le f_a \le d_b \le d_{l+1}^1 \le f_b$, ce qui contredit **(2)**. C'est absurde.
   Donc la solution trouvée par l'algorithme glouton est optimale.
4. *Implémenter cet algorithme en Ocaml*

```
let compare_couples (a,b) (c,d) =
 if a = c && b = d then 0
 else if a > c || (a=c && b>d) then 1
 else -1;;

 let tri_glouton liste =
   List.sort compare_couples liste;;


 let partition_glouton liste =
   let disjoints_liste i l = //Fonction qui teste si l'intervalle i est disjoint de tous les éléments de l
     match l with
     |[] -> true
     |h::q -> (disjoints h i)&&(disjoints_liste i q)
   in
   let rec ppd i p d = match p with //Fonction qui cherche le plus petit d acceptable pour i
     |[] -> d //nouvelle catégorie
     |h::q when disjoint_liste h i -> d
     |h::q -> ppd i q (d+1)
   in
   let ajoute i p d = //Fonction qui ajoute i dans la partie d
     let rec aux i p d = match p with //auxiliaire qui parcourt p pour trouver la partie d
       |h::q when d=0 -> (i::h)::q
       |h::q -> h::(aux i q (d-1))
     in
     //Ici, pour garantir qu'on trouve le plus petit d la nouvelle partie doit etre placee a la fin
     if d=List.length p then List.rev([i]::(List.rev p))
     else aux i p d
   in
   let aux_part liste p = match liste with //Fonction qui parcourt la liste
     |[] -> p
     |h::q -> let d = ppd h p 0 in let p' = ajoute h p d in aux q p'
     in
   (*On trie la liste dans le bon ordre et démarre le glouton *)
   let aux_part (tri_glouton liste) [] in ;;
```