MPI* Info - TP9

Branch and bound

O. Caffier

1 Maxsat

```
type formule = int list list;;
```

Question 1.1 Donner la représentation de $\varphi = (x_1 \vee x_2 \vee \neg x_3) \wedge (x_3 \vee x_4) \wedge (\neg x_4 \vee x_2)$.
Corrigé :

```
let phi_ex = [ [1;2;-3] ; [3;4] ; [-4;2] ]
```

```
type valuation = bool array;;
```

Question 1.2 Ecrire une fonction `sat : formule -> valuation ->int` telle que `sat phi v` renvoie le nombre de clauses de `phi` satisfaites par la valuation `v`
Corrigé :

```
let sat (phi : formule) (v : valuation) : int =
  let res = ref 0 in
  List.iter (fun l -> if List.exists (fun i -> if i < 0 then not v.(-i) else v.(i) ) l then incr(res)) phi ;
  !res
```

Remarque : pour des soucis d'écriture (et de logique), je considère tout au long de l'exo la valuation de x_i comme la valeur de `v.(i)`

Question 1.3 Ecrire une fonction `insat : formule -> valuation -> int -> int` telle que `insat f valuation k` renvoie le nombre de clauses totales moins celles que l'on ne pourra pas satisfaire avec un prolongement de la valuation partielle définie par les `k` premières cases de `val` (les cases entre 0 et `k-1` de valuation sont les seules considérées c'est-à-dire les variables entre 1 et `k`).
Corrigé :

```
let insat (phi : formule) (v : valuation) (k : int) : int =
  let res = ref (List.length phi) in
  List.iter
    (fun l ->
       if List.for_all (fun i -> if i < 0 then -i<=k && v.(-i) else i <= k && not v.(i) ) l then decr(res))
    phi ;
  !res
```

Question 1.4 En déduire une implémentation de la résolution exacte du problème MAXSAT avec la méthode branch and bound utilisant l'heuristique `insat`. On écrira une fonction `maxsat : formule -> int -> int` qui renvoie le nombre optimal de clauses satisfaites par la formule. La fonction prendra en entrée le nombre de variables utilisées par la formule.
Corrigé :

```
let maxsat (phi : formule) (n : int) : int =
  let max = ref 0 in
  let v = Array.make (n+1) false in

  let rec aux (v_tmp : valuation) (k_tmp : int) : unit =
    match k_tmp with
    |k when k = n ->
        let max_tmp = sat phi v_tmp in if max_tmp > !max then max := max_tmp
    |_ ->
        if insat phi v_tmp k_tmp > !max then
          begin
            aux v_tmp (k_tmp +1) ;
            v_tmp.(k_tmp +1) <- true;
            aux v_tmp (k_tmp +1) ;
          end
  in
  aux v 0 ;
  !max
```

2 PVC

```
type graphe = int array array
type chemin = int list
```

Question 2.1 Donner la matrice d'adjacence du graphe de gauche de la figure suivante (le graphe de droite est un cycle hamiltonien de poids minimal et permettra de tester votre fonction)
Corrigé :

```
let g_ex =[|[|max_int;2;max_int;max_int;max_int;1;max_int;1|];
            [|2;max_int;1;max_int;1;max_int;max_int;max_int|];
            [|max_int;1;max_int;1;max_int;max_int;max_int;5|];
            [|max_int;max_int;1;max_int;2;max_int;1;max_int|];
            [|max_int;1;max_int;2;max_int;1;max_int;max_int|];
            [|1;max_int;max_int;max_int;1;max_int;2;max_int|];
            [|max_int;max_int;max_int;1;max_int;2;max_int;1|];
            [|1;max_int;5;max_int;max_int;max_int;1;max_int|]|]
```

Question 2.2 Supposons qu'on dispose d'un chemin partiel $\tilde{c}$ dont la longueur est strictement inférieure au nombre total de sommets. Donner un minorant simple du poids d'un chemin hamiltonien c qui prolonge $\tilde{c}$.
Corrigé :
On a

$$\boxed{p(\tilde{c}) \leq p(c)}$$

Question 2.3 Ecrire une fonction `supprimer : int->int list->int list` qui supprime un élément dont la valeur est passée en entrée de la liste.
Corrigé :

```
let rec supprimer (elt : int) (l : int list) : int list =
  match l with
  |[] -> []
  |x::xs when x = elt -> xs
  |x::xs -> x::(supprimer elt xs)
```

Question 2.4 Ecrire une fonction `poids_chemin : graphe -> chemin -> int` qui calcule le poids d'un chemin donné dans un graphe pondéré (attention à la gestion des arêtes de poids `max_int` et à ne pas dépasser la capacité mémoire).
Corrigé :

```
let rec poids_chemin (g : graphe) (c : chemin) : int =
  match c with
  |[] -> 0
  |x::[] -> 0
  |x::y::xs ->
      if g.(x).(y) = max_int then max_int
      else
        let tmp = poids_chemin g (y::xs) in if tmp = max_int then tmp else g.(x).(y) + tmp
```

Question 2.5 En déduire une fonction `pvc : graphe->chemin` en utilisant la méthode *branch and bound*.
Corrigé :

```
let rec map_min_filter heur f l =
  match l with
  |[]-> max_int,[]
  |h::t -> let x,c_aux = map_min_filter heur f t in
      if (heur(h) < x) then begin
        if fst(f h) < x then  f h
        else x,c_aux
      end
      else x,c_aux

let pvc (g : graphe) : chemin =
  let n = Array.length g in
  let chemin_init = List.init n (fun i-> i) in
  let rec aux (c_courant: int list) (sommets_rest: int list) : int * chemin =
    match sommets_rest with
    |[]-> (poids_chemin g c_courant), c_courant
    |h::t -> map_min_filter (fun y -> poids_chemin g c_courant) (fun y->(aux (y::c_courant) (supprimer y
        sommets_rest))) sommets_rest
  in
  snd(aux [] chemin_init)
```

3 Sac à dos