DS 5

Informatique MP2I

Julien Reichert

Questions de cours

Question de cours 1 : Que dire d'un tas (dont tous les éléments sont supposés différents deux à deux) qui est un ABR ? Et d'un tas (même hypothèse) qui est un arbre bicolore ?

Question de cours 2 : Donner une définition acceptable d'un arbre PATRICIA.

Question de cours 3 : Que dire de la matrice d'adjacence d'un graphe non orienté complet ?

Question de cours 4 : Définir la notion de graphe sans circuit. Définir la notion de composante fortement connexe. Que dire des composantes fortement connexes d'un graphe sans circuit ?

Question de cours 5 : Dessiner un exemple de graphe admettant un circuit eulérien et matérialiser le circuit de manière explicite. Un graphe admettant un circuit eulérien admet-il forcément un chemin eulérien si on ajoute un nouvel arc ? Et si on en retire un ? Dans les deux cas faire une preuve ou donner un contre-exemple.

Question de cours 6 : Écrire en OCaml un type permettant de représenter une structure persistante d'ABR. Écrire avec ce type une fonction prenant en entrée un ABR non vide (erreur sinon) et renvoyant un couple formé du minimum de l'ABR et de l'ABR sans ce minimum. Prouver la terminaison et la correction de la fonction.

Question de cours 7 : On considère une structure de trie utilisant uniquement les caractères `0` et `1`, de sorte que la représentation utilise des arbres binaires. On ajoutera toujours l'hypothèse qu'un nœud ne correspondant pas à un mot (donc ayant au moins un fils) aura toujours deux fils et que la racine ne correspondra jamais à un mot. Déterminer le nombre de formes d'arbres possibles avec trois mots sous cette restriction. Les tries ainsi obtenus seront assimilés à des codes. Déterminer combien d'entre eux sont préfixes et combien sont non ambigus.

Question de cours 8 : On donne ci-après une fonction réalisant l'algorithme de Floyd-Warshall. Adapter le code pour que la fonction retourne un chemin optimal en valeur de tout sommet à tout sommet.

```
let floyd_warshall graphe =
  let n = Array.length graphe in
    let dist = Array.make_matrix n n max_int in
      for i = 0 to n-1 do List.iter (fun (sommet, poids) -> dist.(i).(sommet) <- poids) graphe.(i) done;
      for i = 0 to n-1 do if dist.(i).(i) > 0 then dist.(i).(i) <- 0 done;
      for k = 0 to n-1 do
        for i = 0 to n-1 do
          for j = 0 to n-1 do
            if max dist.(i).(k) dist.(k).(j) < max_int
              then dist.(i).(j) <- min dist.(i).(j) (dist.(i).(k)+dist.(k).(j))
          done;
          if dist.(i).(i) < 0 then failwith "circuit de poids négatif détecté"
        done
      done; dist;;
```

Exercices

Sauf mention explicite du contraire, le langage est au choix **parmi OCaml et C** pour tous les exercices demandant d'écrire un programme. Les recommandations ne correspondent qu'au choix fait pour l'élaboration du corrigé. Cependant, **si un exercice est traité dans les deux langages, aucune des deux versions ne sera corrigée**. Pour changer d'avis alors qu'une partie sur un des langages a déjà été commencée au propre, cette partie sera à barrer de manière bien visible et sera ignorée.

Consigne : tout exercice prenant en argument ou renvoyant une « séquence » prendra en argument / renverra un tableau en C (ainsi que sa taille, comme d'habitude), mais en OCaml le choix est laissé entre tableau et liste.

On rappelle la documentation du module `Hashtbl` :

- `Hashtbl.create n` crée une table de hachage avec `n` places pour commencer, mais en adaptant si besoin (donc on devine `n` sans qu'il n'y ait de risque si l'estimation est mauvaise) ;
- `Hashtbl.add th cle valeur` ajoute une association à la table de hachage, en masquant une éventuelle clé déjà existante (l'autre valeur sera de nouveau accessible en cas de retrait de ce qui l'a masqué) ;
- `Hashtbl.find th cle` détermine la valeur associée à la clé dans la table de hachage, en déclenchant l'erreur `Not_found` si la clé est absente ;
- `Hashtbl.mem th cle` détermine si la clé est présente dans la table de hachage ;
- `Hashtbl.remove th cle` retire une occurrence de la clé dans la table de hachage s'il y en a une (sinon la fonction n'a pas d'effet) ;
- `Hashtbl.replace th cle valeur` remplace la valeur associée à la clé dans la table de hachage par une nouvelle valeur (une éventuelle valeur masquée n'est pas impactée) en ajoutant la clé si elle n'y était pas encore.
- `Hashtbl.find_opt th cle` agit comme la fonction `find`, mais retourne une option, donc si la clé est absente aucune exception n'est levée, c'est simplement le cas où `None` est renvoyé ;
- `Hashtbl.iter f th` appelle la fonction fournie, prenant des clés et des valeurs (dans cet ordre) en argument, à tous les éléments de la table de hachage, sans contrôle sur l'ordre, sachant que si des clés sont masquées par d'autres clés identiques, elles subiront aussi la fonction (et on sait que ce sera dans l'ordre inverse de leur apparition dans la table de hachage).

Exercice 1 [OCaml recommandé] : Créer un type `trie` permettant de représenter un trie. Écrire dans le même langage une fonction prenant en entrée un trie et un mot et déterminant si ce mot est un préfixe d'un mot d'un trie. On signale ou rappelle qu'un mot est un préfixe d'un autre si le second s'obtient en ajoutant des lettres à la fin du premier. Écrire aussi (toujours dans le même langage) une fonction déterminant si un objet de type `trie` est un trie valide, en commençant par écrire les conditions nécessaires et suffisantes à vérifier.

Exercice 2 : Créer un type permettant de représenter une structure modifiable d'arbre binaire (en C : d'entiers), sachant que chaque nœud a une information supplémentaire : la taille du sous-arbre enraciné en ce nœud. Écrire alors une fonction qui mute un arbre en ajoutant un nouveau nœud, nécessairement dans une feuille et en partant à gauche si, et seulement si, la taille du sous-arbre gauche est inférieure ou égale à celle du sous-arbre droit. Montrer qu'un tel arbre de taille une puissance de deux moins un est forcément complet.

Exercice 3 [OCaml recommandé] : On définit le diamètre d'un graphe non orienté connexe comme la longueur maximale d'une chaîne minimale, c'est-à-dire $\max_{s,t\in S} \mathrm{dist}(s,t)$ avec les notations intuitives. Écrire une fonction qui calcule le diamètre d'un graphe (erreur s'il n'est pas connexe). En donner la complexité.

Exercice 4 : Écrire un type permettant de représenter un graphe non orienté sans doubler les arcs d'un graphe orienté. En utilisant ce type, écrire une fonction prenant un graphe non orienté qu'on suppose complet et qui détermine s'il admet une chaîne hamiltonienne. Idem pour une chaîne eulérienne.

Exercice 5 : On considère une chaîne de caractères `journal` représentant un journal dont on veut découper des morceaux pour écrire une lettre anonyme en tant que chaîne de caractères `s`. On suppose que les morceaux découpés sont sur une même ligne et que le nombre d'exemplaires du journal est illimité. Écrire une fonction prenant en argument ces deux chaînes de caractères et qui détermine le nombre minimal de sous-chaînes de `journal` à extraire pour former `s`. Comme annoncé, ces sous-chaînes peuvent se chevaucher, voire être plusieurs fois les mêmes. Le nombre d'exemplaires du journal à utiliser n'est ni à minimiser ni à calculer.

Exercice 6 [OCaml recommandé] : On considère un réseau de transports `reseau` représenté en tant que tableau de séquences de chaînes de caractères, une séquence par ligne (numérotées à partir de zéro pour simplifier), précisant le nom des stations desservies par la ligne en question (peu importe l'ordre). Écrire une fonction prenant en argument un tel réseau et une séquence `lgn`, strictement croissante d'entiers naturels strictement inférieurs au nombre de lignes, et qui détermine la séquence des stations par lesquelles passent toutes les lignes de `lgn` et uniquement elles.

Exercice 7 : On considère une séquence de codes en tant qu'entiers compris entre `-2` et `2`. Le code `2` représente une victoire bonifiée, le code `1` une victoire simple, le code `0` n'existe pas, le code `-1` représente une défaite simple et le code `-2` une défaite critique. Un joueur est content si **à partir de tout élément de la séquence et jusqu'à la fin**, le nombre de victoires (bonifiées ou non) est strictement supérieur au nombre de défaites (idem), le nombre de victoires bonifiées est supérieur ou égal au nombre de défaites critiques **et** aucune série de défaites (resp. défaites critiques) n'est strictement plus longue que la plus longue série de victoires (resp. victoires bonifiées) ultérieure (en particulier on ne peut pas finir sur une défaite...). Écrire une fonction `content` déterminant si le joueur est content au vu des critères ainsi énoncés.

Exercice 8 [OCaml imposé] : On considère une structure d'arbre d'arité quelconque utilisant le type suivant : `type 'a arbre = N of 'a * 'a arbre list`. Écrire une fonction d'insertion pour cette structure selon les conditions suivantes : un argument particulier est une liste d'entiers et signale dans l'ordre quel enfant du nœud courant verra l'insertion se poursuivre (erreur s'il n'y en a pas autant), sachant que le dernier élément de la liste indique avant quel enfant actuel (à la fin si cet élément est égal au nombre d'enfants du nœud actuel) on insère le nouveau nœud (ayant l'étiquette en argument et une liste d'enfants vide). Écrire aussi une fonction de retrait d'un nœud ayant aussi une liste en argument supplémentaire. Si le nœud n'a pas une liste d'enfants vide, on déclenchera une erreur.

Par exemple, l'insertion de la valeur `4` dans l'arbre `N(0, [N(1, [N(5, [])]); N(2, []); N(3, [])])` donnera l'arbre `N(0, [N(1, [N(5, [])]); N(2, [N(4, [])]); N(3, [])])` si la liste en argument supplémentaire est `[1, 0]` et donnera l'arbre `N(0, [N(1, [N(5, [])]); N(2, []); N(3, []); N(4, [])])` si cette liste est `[3]`.

En utilisant la même liste en argument supplémentaire, le nœud nouvellement inséré est supprimé par la fonction de suppression. Mais il y aurait une erreur avec la liste `[0]` en argument supplémentaire depuis ces arbres.

Exercice 9 : On donne ci-après une fonction réalisant l'algorithme de Bellman-Ford. Adapter le code pour que la fonction ne considère à chaque tour de boucle que les sommets qui ont effectivement été améliorés.

Pour changer, les listes d'adjacence seront des tableaux de tableaux.

```
let bellman_ford graphe origine =
  let n = Array.length graphe in
  let dist = Array.make n max_int in dist.(origine) <- 0;
  for k = 1 to n do
    let nouv_dist = Array.copy dist in
    for sommet = 0 to n-1 do
      if dist.(sommet) < max_int then
        for isucc = 0 to Array.length graphe.(sommet)-1 do
          let succ, poids = graphe.(sommet).(isucc) in
          nouv_dist.(succ) <- min nouv_dist.(succ) (dist.(sommet) + poids)
        done
    done;
    if k < n then
      for recopiage = 0 to n-1 do
        dist.(recopiage) <- nouv_dist.(recopiage)
      done
    else
      for verif = 0 to n-1 do
        if nouv_dist.(verif) < dist.(verif) then failwith "Circuit de poids strictement négatif !"
      done
  done; dist;;
```

Exercice 10 : Déterminer ce que fait la fonction suivante.

```
int f(double** p, double* tab, int n)
{
  int rep = 0;
  for (int i = 0 ; i < n ; i += 1)
  {
    if (tab[i] > 0) rep += 1;
  }
  *p = malloc(rep * sizeof(double));
  int j = 0;
  for (int i = 0 ; i < n ; i += 1)
  {
    if (tab[i] > 0)
    {
      (*p)[j] = tab[i];
      j += 1;
    }
  }
  return rep;
}
```

Problème 1

Ce problème est à traiter intégralement dans un seul langage au choix selon les mêmes consignes que les exercices.

On considère une structure de tableau associatif pour associer des clés d'un certain type `T1` et des valeurs d'un certain type `T2`. La structure est formée d'un dictionnaire indexé par des `T1` et contenant des couples (entier, `T2`), d'une file de priorité (tas binaire presque complet stocké dans un tableau redimensionnable) contenant les clés de type `T1` associées à une priorité qui matérialise leur ordre d'apparition et d'un entier qui est le nombre d'insertions qui ont été faites. L'entier qui est l'information supplémentaire dans le dictionnaire correspond à l'indice dans le tableau implémentant la file de priorité.

Pour ajouter un élément, on insère la clé dans la file de priorité en lui donnant comme priorité le nombre d'insertions déjà faites (nombre qui est alors incrémenté), on relève la position dans le tableau où cette insertion se fait (la dernière...), et on procède à l'insertion dans le dictionnaire avec cette information en plus. On veillera au préalable à interdire les doublons.

Pour retirer un élément, on modifie artificiellement sa priorité pour pouvoir procéder à une extraction (il est aussi permis de passer outre la structure, tant que cela fonctionne et que c'est bien expliqué), ce qui forcera à reporter la mise à jour des informations afin qu'elles demeurent cohérentes, puis on procède au retrait dans le dictionnaire.

La consultation et la modification d'une valeur ne posent pas de problème en soi (la création non plus par ailleurs).

Une opération élémentaire supplémentaire est rendue possible par cette structure : le parcours de l'ensemble des clés dans l'ordre chronologique de leur ajout parmi celles qui sont encore dans le dictionnaire. Cette opération retournera une séquence de couples (clé, valeur).

Exercice du problème 1 : Réaliser la structure associée.

Attention : Si l'exercice est fait en OCaml, le module `Hashtbl` est interdit !

Problème 2

Ce problème est à traiter intégralement en OCaml.

On dispose d'un tableau de chaînes de caractères supposées **toutes non vides** et différentes deux à deux (inutile de le vérifier) et on souhaite obtenir un tableau non ambigu, représentant les chaînes dans le même ordre mais de manière raccourcie.

Par exemple, le tableau des jours de la semaine en français se raccourcit en `"L", "Ma", "Me", "J", "V", "S", "D"`, une deuxième lettre étant nécessaire pour mardi et mercredi.

Nous allons utiliser un trie avec une information supplémentaire au niveau de chaque nœud, à savoir le nombre de mots dans le sous-arbre enraciné en ce nœud (le nœud y compris).

Consigne importante : **Inutile de chercher à minimiser la somme des tailles totales par des raffinements, les raccourcissements sont indépendants. En particulier, la non ambigüité ne doit pas provenir d'un raisonnement par élimination mais être intrinsèque à la chaîne.**

Question P1 : Écrire un type adéquat pour de tels tries. Il sera utilisé pendant tout le problème.

Question P2 : Écrire les opérations élémentaires de création, insertion et test d'appartenance pour cette structure de trie. **On n'écrira pas la suppression qui n'est pas utile ici !**

Question P3 : Écrire une fonction prenant en argument un trie et un mot dont on admet qu'il y appartient (inutile de le vérifier) et retournant le plus petit préfixe non ambigu de ce mot pour l'état actuel du trie.

Question P4 : Écrire une fonction prenant en argument un tableau de chaînes de caractères et retournant le tableau non ambigu dont les tailles des chaînes sont toutes les plus petites possible.

Question P5 : La solution est-elle unique ? Justifier.

Étendons le travail : l'argument sera désormais un tableau de couples de chaînes (deux mots qui peuvent par exemple représenter une identité), on cherche alors à raccourcir chacun des mots pour que la taille totale soit la plus courte possible dans chaque mot. Un couple ne peut apparaître qu'une fois dans le tableau.

La même consigne importante s'applique.

Question P6 : Suggérer une structure de données d'appui pour ce nouveau problème (attention, ce ne sera pas forcément un trie). En particulier, discuter d'autres possibilités plus simples mais incorrectes ou de complexité rédhibitoire. Par ailleurs, serait-ce grave si un même mot était répété plusieurs fois (en tant que premier élément de plusieurs couples, par exemple) ?

Question P7 : Écrire une fonction résolvant le problème posé dans le nouveau contexte.

À tout hasard, on signale que le module `Hashtbl` *est autorisé cette fois...*

Question P8 : La solution est-elle à présent unique ? Justifier.