Correction du DS 5

Julien Reichert

La correction des questions de cours étant dans le cours, elle ne sera pas donnée ici. De même, les exercices issus des TD et TP ont leur correction déjà publiée.

Exercices

Exercice 1 :

En pratique, les arbres binaires de recherche ayant les enfants d'un nœud d'un trie comme nœuds et les tries utilisant de tels arbres binaires de recherche, les deux types sont définis par une récursion mutuelle.

```
type tabr = V | N of tabr * atrie * tabr
and atrie = Noeud of char * bool * tabr

let recherche_mot trie chaine =
  let rec mouline_trie (Noeud(_, b, a)) indice =
    if indice = String.length chaine then b else
    let c = chaine.[indice] in
    let rec mouline_abr abr = match abr with
      | V -> false
      | N(g, Noeud(c2, b2, a2), d) ->
        if c = c2 then mouline_trie (Noeud(c2, b2, a2)) (indice+1)
        else if c < c2 then mouline_abr g
        else mouline_abr d
    in mouline_abr a
  in mouline_trie trie 0

let rec branche chaine indice =
  if indice = String.length chaine - 1 then Noeud(chaine.[indice], true, V)
  else Noeud(chaine.[indice], false, N(V, branche chaine (indice + 1), V))

let insere_mot trie chaine =
  let rec mouline_trie (Noeud(c1, b, a)) indice =
    if indice = String.length chaine then Noeud(c1, true, a) else
    (* On aurait pu tester b et déclencher une erreur si true *)
    let c = chaine.[indice] in
    let rec mouline_abr abr = match abr with
      | V -> N(V, branche chaine indice, V)
      | N(g, Noeud(c2, b2, a2), d) ->
        if c = c2 then N(g, mouline_trie (Noeud(c2, b2, a2)) (indice+1), d)
        else if c < c2 then N(mouline_abr g, Noeud(c2, b2, a2), d)
        else N(g, Noeud(c2, b2, a2), mouline_abr d)
    in Noeud(c1, b, mouline_abr a)
  in mouline_trie trie 0
```

Exercice 2 :

Les vérifications à faire sont sur le degré entrant et le degré sortant de chaque sommet qui doivent tous être égaux à un.

Le type est un peu spécial à définir, pour le plaisir d'avoir une difficulté supplémentaire. Le travail serait autrement plus simple sur une matrice d'adjacence... Pour information, si on voulait des chaînes de caractères de taille au plus dix pour les sommets, il aurait fallu écrire, de manière contre-intuitive, `char (*sommets)[11];` ou créer un type, de manière encore plus contre-intuitive `typedef char char_11[11];` pour utiliser `char_11* sommets`.

```
struct grapho { char** sommets ; int nb_sommets ; char*** arcs ; int nb_arcs ; };
typedef struct grapho go;

bool permutation(go g)
{
  if (g.nb_arcs != g.nb_sommets) return false; // Pas cher !
  bool* deja_source = malloc(g.nb_sommets * sizeof(bool));
  bool* deja_dest = malloc(g.nb_sommets * sizeof(bool));
  for (int i = 0 ; i < g.nb_sommets ; i += 1)
  {
    deja_source[i] = false;
    deja_dest[i] = false;
  }
  bool rep = true;
  for (int i = 0 ; i < g.nb_arcs && rep ; i += 1)
  {
    for (int j = 0 ; j < g.nb_sommets && rep ; j += 1)
    {
      if (strcmp(g.sommets[j], g.arcs[i][0]) == 0)
      {
        if (deja_source[j]) rep = false;
        deja_source[j] = true;
      }
      if (strcmp(g.sommets[j], g.arcs[i][1]) == 0)
      {
        if (deja_dest[j]) rep = false;
        deja_dest[j] = true;
      }
    }
  }
  free(deja_source);
  free(deja_dest);
  return rep;
}
```

Problème 1

Question P1.1 :

Un graphe non orienté non vide est un arbre si, et seulement si, deux des trois conditions suivantes (qui entraînent alors la troisième) sont vérifiées : le graphe est connexe, il n'a pas de cycle, et son nombre d'arêtes est un de moins que son nombre de sommets.

Question P1.2 :

Les conditions les plus faciles à remplir sont la première (application des parcours, faite en TP) et la troisième (simple comptage, mais attention au piège).

```
struct graphe_non_oriente { int nbs; int** liste_adj; };
typedef struct graphe_non_oriente gno;

bool connexe(gno g) // On ne refait pas de parcours ici, donc le parcours est compris dans la fonction.
{
  bool* visite = malloc(g.nbs * sizeof(bool)); // On n'appellera pas cette fonction si g.nbs vaut 0.
  for (int i = 1 ; i < g.nbs ; i += 1) visite[i] = false;
  visite[0] = true;
  int* file_limitee = malloc(g.nbs * sizeof(int));
  file_limitee[0] = 0;
  int indecr = 1;
  int indlec = 0;
  bool rep = true;
  while (indlec < indecr && indecr < g.nbs) // Astuce : indecr vaut g.nbs si c'est gagné
  {
    int s = file_limitee[indlec];
    indlec += 1;
    for (int j = 0 ; g.liste_adj[s][j] != -1 ; j += 1)
    {
      int t = g.liste_adj[s][j];
      if (!visite[t])
      {
        visite[t] = true;
        file_limitee[indecr] = t;
        indecr += 1;
      }
    }
  }
  free(visite);
  free(file_limitee);
  return indecr == g.nbs;
}

bool arbre(gno g)
{
  if (g.nbs < 2) return true;
  int nba = 0;
  for (int i = 0 ; i < g.nbs ; i += 1)
  {
    for (int j = 0 ; g.liste_adj[i][j] != -1 ; j += 1) nba += 1;
  }
  return connexe(g) && nba / 2 == g.nbs - 1;
}
```

Question P1.3 :

```
gno copie_graphe(gno g)
{
  int n = g.nbs;
  gno rep = { .nbs = n , .liste_adj = malloc(n * sizeof(int*)) };
  for (int i = 0 ; i < g.nbs ; i += 1)
  {
    rep.liste_adj = malloc(g.nbs * sizeof(int));
    for (int j = 0 ; j < g.nbs ; j += 1)
    {
      rep.liste_adj[i][j] = g.liste_adj[i][j];
      if (g.liste_adj[i][j] == -1) break;
    }
  }
  return rep;
}
```

Question P1.4 :

```
bool est_feuille(gno g, int i)
{
  return g.liste_adj[i][0] != -1 && g.liste_adj[i][1] == -1;
}
```

Question P1.5 :

```
void retirer_sommet(gno g, int i)
{
  for (int j = 0 ; g.liste_adj[i][j] != -1 ; j += 1)
  {
    int s = g.liste_adj[i][j];
    int ind = 0;
    while (g.liste_adj[s][ind] != i && g.liste_adj[s][ind] != -1) ind += 1;
    // -1 par pure sécurité, cas impossible normalement
    if (g.liste_adj[s][ind] == -1) exit(1); // Préparation du debug.
    while (g.liste_adj[s][ind+1] != -1)
    {
      g.liste_adj[s][ind] = g.liste_adj[s][ind+1];
      ind += 1;
    }
    g.liste_adj[s][ind] = -1;
  }
  g.liste_adj[i][0] = -1;
}
```

Question P1.6 :

Puisqu'on doit muter le graphe, on travaille sur une copie. Il faudra penser à la libérer.

```
void libere_graphe(gno g)
{
  for (int i = 0 ; i < g.nbs ; i += 1) free(g.liste_adj[i]);
  free(g.liste_adj);
}
```

```
int* codage_prufer(gno g)
{
  int n = g.nbs;
  int* rep = malloc((n-2) * sizeof(int));
  gno gbis = copie_graphe(g);
  for (int iter = 0 ; iter < n-2 ; iter += 1)
  {
    for (int i = 0 ; i < n ; i += 1)
    {
      if (est_feuille(g, i))
      {
        rep[iter] = gbis.liste_adj[i][0];
        retirer_sommet(gbis, i);
        break;
      }
    }
  }
  libere_graphe(gbis);
  return rep;
}
```

Question P1.7 :

```
void ajouter_demi_arete(gno g, int s, int t) // On suppose que l'arête n'existe pas, etc.
{
  int i = 0;
  while (g.liste_adj[s][i] != -1) i += 1;
  g.liste_adj[s][i] = t;
  g.liste_adj[s][i+1] = -1;
}

gno decodage_prufer1(int* codage, int nm2)
{
  int n = nm2 + 2;
  gno rep = { .nbs = n, .liste_adj = malloc(n * sizeof(int*))};
  for (int i = 0 ; i < n ; i += 1)
  {
    rep.liste_adj[i] = malloc(n * sizeof(int));
    rep.liste_adj[i][0] = -1;
  }
  int* degres = malloc(n * sizeof(int));
  for (int i = 0 ; i < n ; i += 1) degres[i] = 1;
  for (int iter = 0 ; iter < nm2 ; iter += 1)
  {
    degres[codage[iter]] += 1;
  }
  for (int iter = 0 ; iter < nm2 ; iter += 1)
  {
    for (int i = 0 ; i < n ; i += 1)
    {
      if (degres[i] == 1)
      {
        ajouter_demi_arete(rep, codage[iter], i);
        ajouter_demi_arete(rep, i, codage[iter]);
        degres[i] -= 1;
        degres[codage[iter]] -= 1;
        break;
      }
    }
  }
  int arete_finale[2];
  int indice = 0;
  for (int i = 0 ; i < n ; i += 1)
  {
    if (degres[i] == 1)
    {
      arete_finale[indice] = i;
      indice += 1;
    }
  }
  ajouter_demi_arete(rep, arete_finale[0], arete_finale[1]);
  ajouter_demi_arete(rep, arete_finale[1], arete_finale[0]);
  free(degres);
  return rep;
}
```

Question P1.8 :

```
gno decodage_prufer2(int* codage, int nm2)
{
  int n = nm2 + 2;
  gno rep = { .nbs = n, .liste_adj = malloc(n * sizeof(int*))};
  for (int i = 0 ; i < n ; i += 1)
  {
    rep.liste_adj[i] = malloc(n * sizeof(int));
    rep.liste_adj[i][0] = -1;
  }
  int* indices = malloc(n * sizeof(int));
  for (int i = 0 ; i < n ; i += 1) indices[i] = i; // Au lieu de retirer, on remplacera par n+1
  for (int iter = 0 ; iter < nm2 ; iter += 1)
  {
    int x = codage[iter];
    for (int i = 0 ; i < n ; i += 1)
    {
      if (indices[i] != n+1) // Donc vaut i
      {
        bool absent = true;
        for (int j = i ; j < nm2 && absent ; j += 1)
        {
          if (codage[j] == i) absent = false;
        }
        if (absent)
        {
          ajouter_demi_arete(rep, x, i);
          ajouter_demi_arete(rep, i, x);
          indices[i] = n+1;
          break;
        }
      }
    }
  }
  int arete_finale[2];
  int indice = 0;
  for (int i = 0 ; i < n ; i += 1)
  {
    if (indices[i] == i)
    {
      arete_finale[indice] = i;
      indice += 1;
    }
  }
  ajouter_demi_arete(rep, arete_finale[0], arete_finale[1]);
  ajouter_demi_arete(rep, arete_finale[1], arete_finale[0]);
  free(indices);
  return rep;
}
```

Question P1.9 :

Dans les deux cas, on construit un graphe non orienté dont le nombre d'arêtes est un de moins que le nombre de sommets. Concernant une autre propriété nécessaire des arbres, on va la prouver en commençant par considérer que la renumérotation des sommets fait simplement une permutation sur les entiers qui se répercute sur le codage de Prüfer. Ainsi, on renumérote les sommets pour correspondre à l'ordre d'ajout des arcs (en tant que « `j` » dans l'algorithme). On se rend compte qu'on ne relie alors les sommets `j` qu'à des sommets strictement supérieurs, interdisant la création d'un cycle.

Question P1.10 :

Puisque le décodage de Prüfer de n'importe quel tableau de $n-2$ valeurs entre 0 et $n-1$ permet de créer des arbres-graphes uniquement (et un seul arbre par code) et que tout arbre-graphe a un codage de Prüfer, le nombre d'arbres-graphes non orientés à n sommets est égal au nombre de tels tableaux, soit n^{n-2}.

Question P1.11 :

On note n le nombre de sommets du graphe. Toute mention ultérieure de $\mathcal{O}(n)$ provient du fait que la taille du graphe soit elle-même linéaire en n (le nombre d'arcs est supposé de l'ordre du nombre de sommets puisque c'est un arbre).

$\longrightarrow$ Codage :

`retirer_sommet` est en $\mathcal{O}(n)$, notamment si le sommet retiré a pour voisin un sommet de degré le nombre de sommets moins un et qu'il faut aller au bout de la liste d'adjacence.
Par conséquent, la boucle sur `i` est en $\mathcal{O}(n)$ car on teste tous les sommets et pour un seul d'entre eux on procède au retrait.
Finalement, la boucle sur `iter` est en $\mathcal{O}(n^2)$ et c'est la complexité du codage.

$\longrightarrow$ Décodage par le premier algorithme :

L'ajout d'une arête est en $\mathcal{O}(n)$ pour la même raison que le retrait d'un sommet l'était.
L'initialisation et la préparation des degrés sont en $\mathcal{O}(n)$.
La boucle sur `i` est en $\mathcal{O}(n)$ car le test n'est réalisé qu'une fois en raison du `break` et le corps du test est en temps linéaire.
La boucle sur `iter` est en $\mathcal{O}(n^2)$, et c'est la complexité du décodage.

$\longrightarrow$ Décodage par le deuxième algorithme : La complexité est la même pour des raisons analogues.

Question P1.12 :

Puisque 1, 2 et 3 sont des feuilles, on commence par mettre 4 dans le tableau trois fois (le voisin de chacune de ces feuilles). Par la suite, 4 devient une feuille et on termine le tableau par un 5, car par la suite il ne reste que deux sommets.

Question P1.13 :

On crée un graphe avec pour sommets les nombres de 0 à 4. Les sommets 0 à 2 sont de degré 2 et les sommets 3 et 4 sont de degré 1.

D'après le premier algorithme de décodage, on relie alors 3 (sommet de degré 1 de plus petit indice) et 1 (premier élément du tableau), puis 1 (sommet désormais de degré 1 de plus petit indice) et 2 (deuxième élément du tableau), puis 2 (sommet désormais de degré 1 de plus petit indice) et 0 (troisième élément du tableau), puis 0 et 4 en tant que derniers sommets de degré 1. Ce codage donne donc un arbre-graphe réduit à la chaîne 3 - 1 - 2 - 0 - 4 (on note qu'il y a une majorité de chaînes parmi les arbres-graphes à 5 sommets, à savoir 60 sur 125, car cela correspond à tous les codages de Prüfer avec trois valeurs différentes).

Problème 2

Question P2.1 :

Mouvements possibles :
- Le 2♢ de la 6^{e} colonne et les trois cartes au-dessus de lui sur le 3♢ de la 1ère colonne.
- Le 5♡ de la 1ère colonne et la carte au-dessus de lui sur le 6♡ de la 3^{e} colonne.
- L'A♡ de la 5^{e} colonne et les cinq cartes au-dessus de lui sur le 2♡ de la 4^{e} colonne.
- Le 6♡ de la 3^{e} colonne sur le 7♡ de la 7^{e} colonne.

Question P2.2 :

Pour chaque colonne, on peut déplacer la seule carte disponible sur la carte du haut. Exception : s'il n'y a pas de carte, on peut déplacer n'importe quel roi. Cela fait au plus 19 déplacements aux équivalences près (un roi peut aller sur n'importe laquelle des quatre dernières colonnes sans que cela ne change rien quand elles sont vides, mais pour les trois premières colonnes l'existence de la pioche peut avoir un impact).

Question P2.3 :

C'est exactement la fonction Fisher-Yates du TD 2, à transcrire en OCaml :

```
let shuffle tab =
  let n = Array.length tab in
  for i = n-1 downto 1 do
    let j = Random.int (i+1) in
    let x = tab.(j) in tab.(j) <- tab.(i); tab.(i) <- x
  done
```

Question P2.4 :

```
type carte = { valeur : int ; couleur : int }
type jeu = { plateau : carte array array ; mutable pioche : carte array ; cachees : int array }

let configuration_depart () =
  let paquet = Array.init 52 (fun i -> { valeur = i / 13 ; couleur = i mod 4 }) in
  shuffle paquet;
  let pl = Array.init 7 (fun i -> Array.sub paquet (7 * i) 7) in
  { plateau = pl ; pioche = Array.sub paquet 49 3 ; cachees = [| 4; 4; 4; 4; 0; 0; 0 |] }
```

Question P2.5 :

```
let cherche_carte jeu k v c =
  let rec aux i j =
    if i = 7 then []
    else if i = k || j = Array.length jeu.plateau.(i) then aux (i+1) jeu.cachees.(i+1)
    else if jeu.plateau.(i).(j) = { valeur = v ; couleur = c }
         || jeu.plateau.(i).(j).valeur = v && v = 12 && j <> 0 then (i, j, k) :: aux i (j+1)
    else aux i (j+1)
  in aux 0 jeu.cachees.(0)

let deplacements_possibles jeu =
  let rec place_colonne k =
    if k = 7 then [] else
    if jeu.plateau.(k) = [||] then cherche_carte jeu k 12 4 @ place_colonne (k+1)
    else let { valeur = v ; couleur = c } = jeu.plateau.(k).(Array.length jeu.plateau.(k) - 1)
      in cherche_carte jeu k (v-1) c @ place_colonne (k+1)
    in place_colonne 0
```

Question P2.6 :

```
let pioche jeu =
  for i = 0 to 2 do
    jeu.plateau.(i) <- Array.append jeu.plateau.(i) [| jeu.pioche.(i) |]
  done;
  jeu.pioche <- [||]
```

Question P2.7 :

```
let copie_configuration jeu =
  let pl = Array.init 7 (fun i -> Array.copy jeu.plateau.(i)) in
  let pi = Array.copy jeu.pioche in
  let ca = Array.copy jeu.cachees in
  { plateau = pl ; pioche = pi ; cachees = ca }
```

Question P2.8 :

```
let colonne_ok jeu i =
  if Array.length jeu.plateau.(i) <> 13 || jeu.cachees.(i) > 0 then false
  else let indice = ref 0 in
    while !indice < 12 do
      if jeu.plateau.(i).(!indice).valeur = jeu.plateau.(i).(!indice+1).valeur - 1
      && jeu.plateau.(i).(!indice).couleur = jeu.plateau.(i).(!indice+1).couleur
      then incr indice else indice := 13
    done; !indice = 12

let fini jeu =
  let ok = ref 0 in
  for i = 0 to 6 do
    if colonne_ok jeu i then incr ok
  done; !ok = 4

let deplace config i j k =
  let cc = copie_configuration config in
  let reste = Array.length cc.plateau.(i) - j in
  cc.plateau.(k) <- Array.append cc.plateau.(k) (Array.sub cc.plateau.(i) j reste);
  cc.plateau.(i) <- Array.sub cc.plateau.(i) 0 j;
  if Array.length cc.plateau.(i) = cc.cachees.(i) && cc.cachees.(i) > 0
    then cc.cachees.(i) <- cc.cachees.(i) - 1;
  cc

let resout_jeu jeu =
  let rec mouline config =
    if fini config then true
    else let l = deplacements_possibles config in
    if l = [] && Array.length config.pioche = 3 then
    ( let cc = copie_configuration config in pioche cc; mouline cc )
    else List.exists (fun (i, j, k) -> mouline (deplace config i j k)) l
  in mouline jeu
```

Question P2.9 :

Le pseudo-variant pour la récursion de `mouline` est le triplet (nombre de cartes de la pioche, 4 - nombre de rois en fond de colonne, 48 - nombre de cartes placées directement sur la carte de même couleur et de valeur un de plus). En effet, chaque appel imbriqué à cette fonction se fait sur une nouvelle configuration où la pioche a été utilisée (et sinon le nombre de cartes de la pioche n'augmente pas) ou un roi a été placé depuis une colonne mais pas au fond vers le fond d'une autre colonne (d'où il ne peut plus être redéplacé), ou une carte supplémentairie a été placée sur la carte de même couleur et de valeur un de plus, sans que d'autres telles connexions ne soient perdues même dans les autres scénarios.

On constate que la fonction écrite ne mémorise pas les configurations rencontrées et pourtant la preuve de terminaison a pu être rédigée. Il en aurait été autrement en autorisant les rois à passer d'une colonne à une autre (ce qui peut être crucial en anticipant l'arrivée des trois cartes de la pioche)...

Question P2.10 :

```
let recherche plateau v c =
  let rec mouline i j =
    if i = 7 then None
    else if j = Array.length plateau.(i) - 1 then mouline (i+1) 0
    else if plateau.(i).(j) = { valeur = v ; couleur = c } then Some (i, j) else mouline i (j+1)
  in mouline 0 0

let blocage2 plateau =
  let rec verifie i j =
    if i = 7 then false
    else if j = Array.length plateau.(i) - 1 then verifie (i+1) 0
    else let { valeur = v ; couleur = c } = plateau.(i).(j) in
    let { valeur = v2 ; couleur = c1 } = plateau.(i).(j+1) in
      match recherche plateau (v2+1) c1 with
      | None -> verifie i (j+1)
      | Some (l, c) -> plateau.(l).(c+1) = { valeur = v-1 ; couleur = c } || verifie i (j+1)
  in verifie 0 0
```

Question P2.11 :

Le degré sortant de chaque sommet étant zéro ou un, on va faire une représentation sous forme de liste d'adjacence et remplacer les cartes par des entiers en prenant la réciproque de la fonction qui a engendré une carte à partir d'un entier dans la question P2.4.

```
let carte_to_int { valeur = v ; couleur = c } = 13 * v + c

let graphe_dependance plateau =
  let rep = Array.make 52 [] in
  let plateau_int = Array.init 7 (fun i -> Array.init 7 (fun j -> carte_to_int plateau.(i).(j))) in
  for i = 0 to 7 do
    for j = 0 to Array.length plateau_int.(i) - 2 do
      rep.(plateau_int.(i).(j)) <- [plateau_int.(i).(j+1)]
    done
  done; rep
```

Question P2.12 :

Il s'agit d'une recherche de circuit dans le graphe. Ceci étant, avec un degré sortant d'un au plus, la recherche de circuit va être remplacée par des parcours simplifiés en suivant depuis chaque sommet l'unique chemin jusqu'à un blocage.

```
exception Circuit

let clairement_impossible plateau =
  let g = graphe_dependance plateau in
  let deja_vu = Array.make 52 false in
  let rec parcours i deja_vu_bis =
    match g.(i) with
    | [] -> for i = 0 to 51 do if deja_vu_bis.(i) then deja_vu.(i) <- true done
    | [j] -> if deja_vu_bis.(j) then raise Circuit;
      deja_vu_bis.(j) <- true; parcours j deja_vu_bis
    | _ -> failwith "Graphe anormal"
  in
  try
    for i = 0 to 51 do
      if not deja_vu.(i) then
        begin
          deja_vu.(i) <- true;
          parcours i (Array.make 52 false)
        end
    done;
    false
  with Circuit -> true
```