TD XLIV

BACKTRACKING

1 Chemins hamiltoniens

Définition XLIV.1

Soit $G = (V, E)$ un graphe non-orienté à n sommets.

- Un *chemin hamiltonien* de G est un chemin élémentaire de longueur $n-1$. Autrement dit, c'est un chemin passant une et une seule fois par chaque sommet du graphe.
- Un *cycle hamiltonien* de G est un cycle élémentaire de longueur n. Autrement dit, c'est un cycle $x = x_0, x_1, \dots, x_{n-1}, x_n = x$ où tous les x_i sont distincts, à part x_0 et x_n.

Déterminer si un graphe possède un chemin, ou un cycle, hamiltonien est un problème « difficile » (NP-complet, nous le verrons l'année prochaine). Cependant, il se prête bien au *backtracking*, et avec la bonne heuristique on peut traiter des graphes relativement gros, hors cas pathologiques.

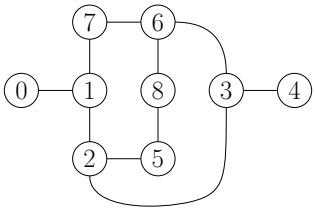

FIGURE XLIV.1 – Le graphe G_0.

▶ **Question 1** Combien de chemins hamiltoniens le graphe G_0 possède-t-il ? Combien de cycles hamiltoniens ?

Pour représenter les graphes, on utilisera le type suivant :

```
type graphe = {nb_sommets : int; voisins : int -> int list}
```

Remarque

On supposera toujours les sommets d'un graphe `g` numérotés de 0 à `g.nb_sommets - 1`.

▶ **Question 2** Écrire une fonction `hamiltonien_depuis` qui prend en entrée un graphe `g` et un indice de sommet (valide) `x0` et renvoie :

- `None` s'il n'existe pas de chemin hamiltonien dans G commençant au sommet x_0 ;
- `Some ordre`, où `ordre` est un `int array` de longueur n codant un chemin hamiltonien à partir de x_0, s'il en existe un. On codera le chemin de la façon suivante : si le chemin est $x_0, x_1, \dots, x_{n-1}$, alors la case x_i du tableau ordre contiendra l'entier i.

Remarque

On pourra :

- utiliser une exception `Trouve of int array`, qu'on lèvera dès que l'on trouve un chemin hamiltonien ;
- initialiser `ordre` à `[|-1; -1; ...; -1|]` et le remplir au fur et à mesure. Le tableau `ordre` fera alors double emploi, puisqu'il indiquera si le sommet `i` est, ou non, encore disponible (suivant que `ordre.(i)` vaut `-1` ou pas).

```
hamiltonien_depuis : graphe -> int -> int array option
```

Parcours du cavalier

Aux échecs, le cavalier est une pièce qui peut se déplacer de deux case suivant un axe et une suivant l'autre :

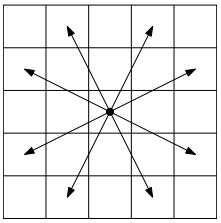

Figure XLIV.2 – Les huit déplacements possibles pour un cavalier.

Le problème du *parcours du cavalier* consiste à faire parcourir à un cavalier toutes les cases d'un échiquier $n \times m$ sans jamais repasser deux fois par la même. On peut éventuellement imposer de plus qu'on finisse sur la case dont on est parti.

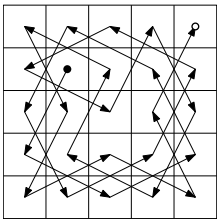

Figure XLIV.3 – Un parcours du cavalier sur un échiquier 5×5.

▶ **Question 3** Écrire une fonction `graphe_cavalier` prenant en entrée deux entiers n et m et codant le problème du cavalier sur un échiquier $n \times m$ sous forme d'un graphe, dans lequel on pourra chercher un chemin hamiltonien. On numérotera les cases du graphes de 0 à $m-1$ sur la première ligne, m à $2m-1$ sur la deuxième ligne et ainsi de suite.

```
graphe_cavalier : int -> int -> graphe
```

▶ **Question 4** Écrire une fonction `affiche_parcours_cavalier` prenant en entrée deux entiers n et m, ainsi qu'un couple (x, y) indiquant les coordonnées d'une case, et affichant le parcours sous la forme ci-dessous :

```
# affiche_parcours_cavalier 5 6 (0, 1);;
 11   0  17  24   9   6
 18  25  10   7  22  29
  1  12  23  16   5   8
 26  19  14   3  28  21
 13   2  27  20  15   4
```

Remarque
S'il n'existe pas de parcours depuis la case demandée, on l'indiquera par un message : c'est par exemple le cas dans un échiquier 5×5 si l'on part de la case $(0, 1)$.

```
affiche_parcours_cavalier : int -> int -> (int * int) -> unit
```

▶ **Question 5** Jusqu'à quelle valeur de n la recherche d'un parcours sur un échiquier $n \times n$ (à partir de la case $(0, 0)$, disons) prend-elle un temps raisonnable ?

Heuristique

Pour améliorer la recherche, on propose d'utiliser l'heuristique suivante : on ordonne les enfants d'un nœud (dans l'arbre de recherche sous-jacent) par nombre croissant de voisins non visités. Autrement dit, si l'on se trouve sur un sommet du graphe et que l'on souhaite étendre le chemin, on commence par essayer le voisin ayant le moins de voisins non visités.

▶ **Question 6** Écrire une fonction `hamiltonien_opt_depuis` implémentant cette heuristique. Cette fonction aura comme effet secondaire d'afficher le nombre de nœuds de l'arbre de recherche explorés.

```
hamiltonien_opt_depuis : graphe -> int -> int array option
```

▶ **Question 7** Utiliser cette heuristique pour trouver un parcours du cavalier sur un échiquier 200×200 (*afficher* ce parcours n'est pas une très bonne idée...). Que constate-t-on ?

Remarque

Cette heuristique très simple permet de trouver un chemin hamiltonien en temps linéaire dans de nombreuses classes de graphes (possédant un tel chemin, bien sûr). Les graphes du cavalier sont l'une de ces classes...

2 Tableaux auto-référents

Un tableau t d'entiers de taille n (indices allant de 0 à $n-1$) est dit *auto-référent* si, pour chaque i entre 0 et $n-1$, le nombre d'occurrences de i dans t est égal à t_i. Par exemple :

indices	0	1	2	3
t	2	0	2	0
occurrences	2	0	2	0

▶ **Question 8** Écrire une fonction énumérant toutes les solutions de taille n.

```
auto_referents : int -> int array list
```

▶ **Question 9** Votre fonction marche-t-elle pour $n = 8$? 10 ? 20 ? 50 ? 80 ? Faire en sorte que ce soit le cas...

▶ **Question 10** Que peut-on conjecturer ? Démontrer votre conjecture.

Remarque

Je n'ai pas vraiment essayé, mais ça n'a pas l'air évident.

Solutions

▶ **Question 11** Il y a exactement un chemin hamiltonien (ou deux, si on le compte dans les deux sens), et pas de cycle hamiltonien. Le chemin est $0-1-7-6-8-5-2-3-4$.

▶ **Question 12**

```
exception Trouve of int array

let hamiltonien_depuis g x0 =
  let ordre = Array.make g.nb_sommets (-1) in
  let voisins_libres x =
    List.filter (fun y -> ordre.(y) = -1) (g.voisins x) in
  let rec explore k x =
    ordre.(x) <- k;
    if k = g.nb_sommets - 1 then raise (Trouve ordre);
    List.iter (explore (k + 1)) (voisins_libres x);
    ordre.(x) <- -1 in
  try
    explore 0 x0;
    None
  with
  | Trouve _ -> Some ordre
```

▶ **Question 13** Il faut s'organiser un peu si l'on veut éviter que le code ne devienne horrible...

```
let graphe_cavalier n m =
  let indice i j = i * m + j in
  let coord s = (s / m, s mod m) in
  let voisins s =
    let i, j = coord s in
    let ok x y =
      0 <= x && x < n && 0 <= y && y < m in
    let u = ref [] in
    let deltas = [(1, 2); (-1, 2); (1, -2); (-1, -2)] in
    let ajoute (dx, dy) =
      if ok (i + dx) (j + dy) then u := indice (i + dx) (j + dy) :: !u;
      if ok (i + dy) (j + dx) then u := indice (i + dy) (j + dx) :: !u in
    List.iter ajoute deltas;
    !u in
  {nb_sommets = n * m; voisins = voisins}
```

▶ **Question 14**

```
let affiche_parcours_cavalier n m (x, y) =
  let i = x * m + y in
  match hamiltonien_depuis (graphe_cavalier n m) i with
  | None -> Printf.printf "Pas de parcours.\n"
  | Some ordre ->
    for i = 0 to n - 1 do
      for j = 0 to m - 1 do
        Printf.printf "%3d " ordre.(i * m + j)
      done;
      print_newline ()
    done
```

▶ **Question 15** Pour $n = 6$ c'est instantané, pour $n = 7$ cela prend déjà plusieurs secondes. Pour $n = 8$ il faut s'armer de patience, et au delà c'est sans doute sans espoir.

▶ **Question 16** Une version relativement simple mais assez peu efficace :

```
let hamiltonien_opt_depuis g x0 =
  let ordre = Array.make g.nb_sommets (-1) in
  let voisins_libres x =
    List.filter (fun y -> ordre.(y) = -1) (g.voisins x) in
  let nb_voisins_libres x =
    List.length (voisins_libres x) in
  let compteur = ref 0 in
  let rec explore k x =
    incr compteur;
    ordre.(x) <- k;
    if k = g.nb_sommets - 1 then raise (Trouve ordre);
    let voisins_tries =
      List.sort
        (fun y z -> nb_voisins_libres y - nb_voisins_libres z)
        (voisins_libres x) in
    List.iter (explore (k + 1)) voisins_tries;
    ordre.(x) <- -1 in
  try
    explore 0 x0;
    Printf.printf "%d\n%!" !compteur;
    None
  with
  | Trouve _ ->
    Printf.printf "%d\n%!" !compteur;
    Some ordre
```

Le problème est qu'un appel `nb_voisins_libres x` est en $O(\deg x)$, et qu'on fait beaucoup de tels appels. On pourrait assez facilement corriger cela, en maintenant à jour un tableau indiquant pour chaque sommet son « degré actuel » (son nombre de voisins libres), mais cette version suffira largement.

▶ **Question 17** On constate deux choses :

- l'appel termine presque instantanément ;
- on explore 40 000 nœuds, ce qui correspond exactement à la longueur du chemin ; autrement dit, on ne fait en réalité jamais de *backtracking*. Autrement dit, l'heuristique semble parfaite.

Remarque
Elle l'est (dans ce cas particulier), mais c'est difficile à montrer.

▶ **Question 18** On reprend le code générique vu en cours :

```
type 'a reponse =
  | Refus
  | Accepte of 'a
  | Partiel of 'a

type 'a probleme =
  {accepte : 'a -> 'a reponse;
   enfants : 'a -> 'a list;
   initiale : 'a}

let enumere probleme =
  let rec backtrack candidat =
    match probleme.accepte candidat with
    | Refus -> []
    | Accepte solution -> [solution]
    | Partiel c' ->
      let rec aux enfants =
        match enfants with
        | [] -> []
        | e :: es -> backtrack e @ aux es in
      aux (probleme.enfants c') in
  backtrack probleme.initiale
```

On définit ensuite le problème :

```
let occurrences t n =
  let occs = Array.make n 0 in
  Array.iter (fun x -> occs.(x) <- occs.(x) + 1) t;
  occs

let enfants_auto n t =
  let f i = Array.append t [| i |] in
  List.init n f

(* Première version en pure force brute (pas d'élagage). *)

let accepte_auto n t =
  if Array.length t = n then
    let occs = occurrences t n in
    if occs = t then Accepte t
    else Refus
  else Partiel t

let autoreferent_brute n =
  {accepte = accepte_auto n;
   enfants = enfants_auto n ;
   initiale = [| |]}
```

Cette version fonctionne pour de toutes petites valeurs de n (instantané pour $n = 5$, un peu long pour $n = 8$, sans espoir pour $n = 15$).

▶ **Question 19** Pour accélérer la recherche, il faut élaguer l'arbre (repérer le plus rapidement possible qu'on se trouve dans une branche ne pouvant donner de solution). On utilise les remarques suivantes :

- à la fin, la somme doit faire n, donc si elle dépasse n à un moment, c'est perdu ;
- c'est également perdu si l'on est sûr qu'elle dépassera n plus tard ;

- s'il y a déjà plus d'occurrences d'une valeur i que la valeur de t_i, c'est également perdu (cette valeur ne changera plus, le nombre d'occurrences ne pourra que croître) ;
- inversement, si par exemple t_3 vaut 5 et qu'il n'y a qu'un seul 3 dans le tableau pour l'instant, alors il faut réserver 4 cases futures pour y mettre des 3. Si ce n'est pas possible, c'est perdu ;

On pourrait encore clairement améliorer, mais cela suffit pour se convaincre qu'il existe une unique solution (sauf pour certaines petites valeurs de n). Je vous laisse le prouver.

```
let accepte_auto_bis n t =
  let exception Echec in
  let k = Array.length t in
  if n = k then accepte_auto n t
  else
    try
      let somme = Array.fold_left (+) 0 t in
      if somme > n then raise Echec;
      let occs = occurrences t n in
      if k > 0 && somme + (n - k) - (t.(0) - occs.(0)) > n then raise Echec;
      let dispo = ref (n - k) in
      for i = 0 to k - 1 do
        dispo := !dispo - t.(i) + occs.(i);
        if occs.(i) > t.(i) || !dispo < 0 then raise Echec;
      done;
      Partiel t
    with
      Echec -> Refus
```

On obtient en une poignée de secondes :

```
# auto_referents_opt 70;;
- : int array list =
[[|66; 2; 1; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0;
   0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0;
   0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0;
   0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 1; 0; 0; 0|]]
```