TP 10

TAQUIN

Structures fournies

Deux modules sont fournis :

- **Heap** (fichiers `heap.ml` et `heap.mli`) pour une file de priorité min permettant l'opération DECREASE-PRIORITY ;
- **Vector** (fichiers `vector.ml` et `vector.mli`) pour des tableaux dynamiques. Ce module est surtout utilisé de manière interne par le module **Heap**, vous n'aurez à vous en servir que tout à la fin du sujet.

Dans les deux cas, une rapide lecture du fichier `.mli` devrait vous permettre d'utiliser le module sans difficulté. Il n'est pas vraiment nécessaire d'aller lire les `.ml`.

D'autre part, vous aurez besoin dans le sujet d'utiliser le module **Hashtbl** qui fournit des tables de hachage. On rappelle ci-dessous quelques fonctions utiles (`'a` est le type des clés et `'b` celui des valeurs).

- `Hashtbl.create : int -> ('a, 'b) Hashtbl.t` crée une table vide. L'entier fourni donne la capacité initiale mais n'a que peu d'importance (la table sera redimensionnée au besoin).
- `Hashtbl.mem : ('a, 'b) Hashtbl.t -> 'a -> bool` permet de tester si une clé est présente dans la table.
- `Hashtbl.add : ('a, 'b) Hashtbl.t -> 'a -> 'b -> unit` ajoute une association à la table.
- `Hashtbl.replace : ('a, 'b) Hashtbl.t -> 'a -> 'b -> unit` modifie une association existante, ou crée l'association si elle n'existait pas. On peut donc l'utiliser systématiquement à la place de `Hashtbl.add`.
- `Hashtbl.find : ('a, 'b) Hashtbl.t -> 'a -> 'b` renvoie la valeur associée à une clé, ou lève l'exception `Not_found` si la valeur n'est pas dans la table.
- `Hashtbl.find_opt : ('a, 'b) Hashtbl.t -> 'a -> 'b option` fait la même chose, mais utilise une option au lieu d'une exception.

Jeu du taquin

Le jeu de taquin est constitué d'une grille $n \times n$ dans laquelle sont disposés les entiers de 0 à $n^2 - 2$, une case étant laissée libre. Voici un état initial possible pour $n = 4$ (qui est la version classique) :

	0	1	2	3
0	2	3	1	6
1	14	5	8	4
2		12	7	9
3	10	13	11	0

On obtient un nouvel état du jeu en déplaçant dans la case libre le contenu de la case située au-dessus, à gauche, en dessous ou à droite, au choix. Si on déplace par exemple le contenu de la case située à droite de la case libre, c'est-à-dire 12, on obtient le nouvel état suivant :

2	3	1	6
14	5	8	4
12		7	9
10	13	11	0

Le but du jeu de taquin est de parvenir à l'état final suivant :

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	

Dans ce sujet, on s'intéresse à la résolution optimale du jeu du taquin, c'est-à-dire à déterminer une suite de déplacements légaux de longueur minimale permettant de passer d'une configuration initiale donnée à la configuration finale. Dans l'exemple ci-dessus, la solution optimale est de longueur 50 (à partir de l'état initial, ou 49 à partir du deuxième état représenté).

En OCaml, une position sera représentée par le type suivant :

```
type state = {
  grid : int array array;
  mutable i : int;
  mutable j : int;
  mutable h : int;
}
```

- On suppose qu'une constante globale `n` a été définie.
- `i` et `j` indique les coordonnées de la case libre.
- `grid` est une matrice $n \times n$ codant la grille. Le contenu de la case libre est arbitraire : si `s` est de type `state`, alors `s.grid.(s.i).(s.j)` peut avoir n'importe quelle valeur.
- `h` sera une heuristique estimant la distance de l'état actuel à l'état final, que nous définirons plus loin.

1 Graphe du taquin

Une configuration du taquin se code naturellement comme une permutation de $[0 \dots 15]$ (où le 15 correspond à la case vide). On peut alors définir le graphe (non orienté) G du taquin comme suit :

- les sommets sont les éléments de $\mathcal{S}_{16}$;
- il y a une arête entre s et s' si et seulement si on peut passer de s à s' (en une étape) par l'un des quatre déplacements décrits plus haut.

On admettra que ce graphe possède exactement deux composantes connexes, contenant chacune la moitié des sommets [1].

▶ **Question 1** Quel est le nombre de sommets de ce graphe ? le nombre approximatif d'arêtes ? Est-il raisonnable de le stocker explicitement en mémoire ?

On code un déplacement par le type suivant, où `U`, par exemple, correspond à un déplacement de la **case libre** vers le haut :

```
type direction = U | D | L | R | No_move

let delta = function
  | U -> (-1, 0)
  | D -> (1, 0)
  | L -> (0, -1)
  | R -> (0, 1)
  | No_move -> assert false
```

Remarque

La valeur `No_move` ne sera utilisée qu'en fin de sujet.

1. Le prouver est un bon exercice. . .de mathématiques.

▸ **Question 2** Écrire une fonction `possible_moves` qui renvoie la liste des directions de déplacement légales à partir d'un certain état.

```
val possible_moves : state -> direction list
```

Pour orienter la recherche, on définit une heuristique h comme suit qui associe à chaque état du taquin un entier positif ou nul. Pour e un état et $v \in [0 \dots n^2 - 2]$, on note e^i_v la ligne de l'entier v dans e et e^j_v sa colonne. On pose alors :

$$h(e) = \sum_{v=0}^{n^2-2} \left| e^i_v - \lfloor v/n \rfloor \right| + \left| e^j_v - (v \bmod n) \right|.$$

▸ **Question 3** Montrer que l'heuristique h est admissible et cohérente.

Remarque

Cette question peut sembler difficile mais elle est en fait très simple, *une fois qu'on a compris ce que représentait* $h(e)$.

▸ **Question 4** Écrire une fonction `compute_h` qui prend en entrée un état, dans lequel le champ `h` a une valeur quelconque, et donne à ce champ la bonne valeur.

Remarque

On pourra, pour cette question et la suivante, utiliser la fonction `distance` fournie.

```
val compute_h : state -> unit
```

▸ **Question 5** Écrire une fonction `delta_h` qui prend en entrée un état e et une direction d et renvoie la différence $h(e') - h(e)$, où e' est l'état que l'on atteint à partir de e en effectuant le déplacement d. On ne fera que les calculs nécessaires (on évitera donc de recalculer toute la somme définissant h).

```
val delta_h : state -> direction -> int
```

▸ **Question 6** Écrire une fonction `apply` qui modifie un état en lui appliquant un déplacement, que l'on supposera légal.

```
val apply : state -> direction -> unit
```

On choisit de modifier l'état plutôt que d'en calculer un nouveau car cela nous sera utile en fin de sujet. Cependant, il sera souvent pratique de disposer d'une copie indépendante.

▸ **Question 7** Écrire une fonction `copy` qui prend un état et en renvoie une copie. On pourra utiliser la fonction `Array.copy`, mais attention : `grid` est un tableau de tableaux...

```
val copy : state -> state
```

2 Utilisation de $A^\star$

▸ **Question 8** Écrire une fonction `successors` qui prend en entrée un état et renvoie la liste de ses successeurs dans le graphe (ou de ses voisins, d'ailleurs, le graphe n'étant pas orienté).

```
val successors : state -> state list
```

▸ **Question 9** Nous avons souvent codé des arbres dans des tableaux de la manière suivante :

- $t[i] = i$ si et seulement si i est la racine de l'arbre ;
- $t[i] = j$ si j est le père de i.

On peut naturellement étendre cette définition à un `('a, 'a) Hashtbl.t`. Écrire une fonction `reconstruct` qui prend en entrée un dictionnaire codant un arbre et un nœud x de l'arbre, et renvoie un chemin de la racine à x, sous la forme d'une liste de nœuds.

```
val reconstruct : ('a, 'a) Hashtbl.t -> 'a -> 'a list
```

▸ **Question 10** Écrire une fonction `astar` prenant en entrée un état initial et calculant un chemin de longueur minimale vers l'état final à l'aide de l'algorithme A*. Cette fonction lèvera l'exception `No_path` si aucun chemin n'existe.

```
val astar : state -> state list
```

Remarque

Comme indiqué, on utilise un dictionnaire `parents` au lieu d'un tableau : il faudra faire de même pour les distances.

▸ **Question 11** Tester cette fonction sur les différents exemples fournis (tous ne seront pas forcément traitables en un temps raisonnable !) et compter le nombre d'états explorés dans chaque cas.

3 Algorithme IDA*

Dans le cas de l'exploration d'un graphe infini, ou en tout cas suffisamment grand pour ne pas pouvoir être stocké en mémoire, on peut se retrouver limité par la mémoire plus que par le temps. En effet, explore des centaines de millions de nœuds n'est pas forcément problématique sur une machine moderne, mais les stocker, et faire des tests d'appartenance à chaque étape, peut vite s'avérer prohibitif. On peut dans ce cas utiliser l'algorithme dit IDA*, qui est un hybride entre le *parcours en profondeur itéré* et l'algorithme A*.

3.1 Parcours en profondeur itéré

On considère l'algorithme suivant :

Algorithme 1 – Parcours en profondeur limité par une profondeur maximale m.

fonction DFS(m, e, p)
  si $p > m$ **alors**
    renvoyer FAUX
  si e est l'état final **alors**
    renvoyer VRAI
  pour x successeur de e **faire**
    si DFS(m, x, $p+1$) **alors**
      renvoyer VRAI
  renvoyer FAUX

Dans cet algorithme, e représente l'état actuel, p la profondeur actuelle (c'est-à-dire la longueur du chemin suivi de l'état initial au nœud actuel, longueur qui n'est pas nécessairement minimale) et m la profondeur maximale autorisée.

▸ **Question 12** Montrer que DFS(m, `init`, 0) renvoie VRAI si et seulement si le sommet final est à une distance inférieure ou égale à m de `init`.

Le parcours en profondeur itéré IDS consiste à effectuer des appels successifs à DFS(0, `init`, 0), DFS(1, `init`, 0), et ainsi de suite jusqu'à trouver un m pour lequel on obtient une réponse positive : ce m est alors la distance de `init` au sommet final.

▶ **Question 13** Déterminer la complexité en temps et en espace d'un parcours en profondeur itéré depuis un sommet initial situé à distance n du sommet final dans les deux cas suivants :

- le graphe contient exactement 1 sommet à distance k de init pour tout k ;
- le graphe contient exactement 2^k sommets à distance k de init pour tout k.

Quel pfut être l'intérêt d'effectuer un parcours en profondeur itéré plutôt qu'un parcours en largeur pour déterminer un plus court chemin ?

3.2 Algorithme IDA*

L'algorithme IDA* est obtenu en ajoutant à l'algorithme IDS une heuristique h admissible, et en effectuant les modifications suivantes :

- la borne ne concerne plus la profondeur p mais le coût estimé $h(e) + p$;
- si un parcours avec une borne de m a échoué, le parcours suivant se fait avec comme borne la plus petite valeur de $h(e) + p$ qui a dépassé m lors du parcours.

On va à nouveau parcourir plusieurs fois des fragments d'arbres, mais cette fois la croissance de ces fragments sera *orientée vers l'état final* (à condition que l'heuristique soit bonne).

Algorithme 2 – Pseudo-code de l'algorithme IDA*.

```
fonction IDA*( )
    m ← h(e_0)
    minimum ← ∞
    fonction DFS*(m, e, p)
        c ← p + h(e)
        si c > m alors
            minimum ← min(c, minimum)
            renvoyer FAUX
        si e est l'état final alors
            renvoyer VRAI
        pour x successeur de e faire
            si DFS*(m, x, p + 1) alors
                renvoyer VRAI
        renvoyer FAUX
    tant que m ≠ ∞ faire
        min ← ∞
        si DFS*(m, e_0, p) alors
            renvoyer VRAI
        m ← min
    renvoyer FAUX
```

▶ **Question 14** Écrire une fonction `idastar_length` qui calcule la longueur minimale d'un chemin du sommet fourni jusqu'au sommet final. On n'utilisera qu'un seul état que l'on mutera au fur et à mesure du parcours (pas d'appels à la fonction `successors`, donc). La fonction renverra `None` si l'état est inaccessible (et qu'elle le détecte).

```
val idastar_length : state -> int option
```

Remarque

On vérifiera que la fonction traite correctement, et en un temps raisonnable, les exemples `ten`, `twenty` et `thirty`.

▶ **Question 15** Montrer que si l'heuristique h est admissible, la fonction `idastar_length` renvoie toujours la distance du sommet fourni au sommet final, à condition qu'un chemin existe.

▶ **Question 16** Apporter les modifications suivantes à cette fonction :

- on souhaite obtenir le chemin gagnant, sous la forme d'un `direction Vector.t;`
- on évitera de revenir immédiatement en arrière (on n'essaiera pas le coup `L` si le dernier coup du chemin actuel est `R`). La présence de `No_move` dans le type `direction` peut ici s'avérer utile.

```
val idastar : state -> direction Vector.t option
```

```
# print_idastar fifty;;
Length 50
Down Left Left Up Right Right Up Left Down Left Left Up Right
Right Right Down Left Left Left Down Right Right Up Left Left
Up Right Right Up Left Left Down Right Right Right Up Left Left
Down Right Right Down Down Left Up Left Left Up Right Down
```

Solutions

▸ **Question 1** Le graphe a $16! \simeq 2 \cdot 10^{13}$ sommets (donc environ 10^{13} si on se limite à la composante connexe de l'état final). Le degré des sommets varie entre 2 et 4, donc le nombre d'arêtes n'est pas très différent du nombre de sommets. À raison d'un octet par sommet ou arête, il faudrait déjà de l'ordre de 20 téra-octets pour stocker cela (et un octet ne suffit clairement pas, même 32 bits ne suffisent pas pour identifier un sommet de manière unique). Donc non, ça ne va pas tenir en mémoire…

▸ **Question 2** Inutile de chercher une solution compliquée :

```
let possible_moves state =
  let possible = ref [] in
  if state.i > 0 then possible := U :: !possible;
  if state.i < n - 1 then possible := D :: !possible;
  if state.j > 0 then possible := L :: !possible;
  if state.j < n - 1 then possible := R :: !possible;
  !possible
```

▸ **Question 3** Pour une valeur v donnée, la quantité $\left|e^i_v - \lfloor v/4 \rfloor\right|$ mesure l'écart vertical entre la position de la case portant le numéro v dans l'état e et sa position dans l'état final ; l'autre terme mesure l'écart horizontal. Ainsi :

- la somme vaut zéro si et seulement si e est l'état final ;
- un mouvement élémentaire déplace une case de une unité dans un axe, et modifie donc h de ± 1 ;
- on a donc $h(e) \leqslant 1 + h(e')$ si e et e' sont voisins, et l'heuristique est donc cohérente (les poids des arêtes valant 1) ;
- puisque $h(\mathtt{final}) = 0$, cela implique également que h est admissible.

▸ **Question 4**

```
let compute_h state =
  let h = ref 0 in
  for i = 0 to n - 1 do
    for j = 0 to n - 1 do
      if i <> state.i || j <> state.j then
        h := !h + distance i j state.grid.(i).(j)
    done
  done;
  state.h <- !h
```

▸ **Question 5** Une seule case a été modifiée. Attention, on déplace *la case libre* de (i, j) à $(i + \delta i, j + \delta j)$.

```
let delta_h state move =
  let (di, dj) = delta move in
  let i = state.i in
  let j = state.j in
  let x = state.grid.(i + di).(j + dj) in
  distance i j x - distance (i + di) (j + dj) x
```

▶ **Question 6**

```
let apply state move =
  let i = state.i in
  let j = state.j in
  let (di, dj) = delta move in
  let x = state.grid.(i + di).(j + dj) in
  state.h <- state.h + delta_h state move;
  state.grid.(i).(j) <- x;
  state.i <- i + di;
  state.j <- j + dj
```

▶ **Question 7** `Array.copy` fait une copie « en surface », il faut donc l'appeler sur chaque ligne de la grille. On peut le faire avec une boucle `for` ou directement avec un `Array.init`.

```
let copy state =
  {grid = Array.init n (fun i -> Array.copy state.grid.(i));
   i = state.i;
   j = state.j;
   h = state.h}
```

▶ **Question 8** Il faut faire bien attention à générer un état indépendant pour chaque déplacement : si les états sont aliasés, chaque appel à `apply` affectera tous les états.

```
let successors state =
  let rec aux moves =
    match moves with
    | [] -> []
    | m :: ms ->
      let s = copy state in
      apply s m;
      s :: aux ms in
  aux (possible_moves state)
```

▶ **Question 9** Il n'y a pas de différence fondamentale avec la version utilisant un tableau que nous avons écrite à plusieurs reprises :

```
let reconstruct parents x =
  let rec aux v path =
    let p = Hashtbl.find parents v in
    if p = v then v :: path
    else aux p (v :: path) in
  aux x []
```

▶ **Question 10** C'est essentiellement l'algorithme du cours, dans lequel on a remplacé les tableaux par des tables de hachage. Notons que pour déterminer si on a atteint l'état final, le test `x.h = 0` sera nettement plus efficace que `x = final` (qui demande de parcourir toute la grille).

```
exception No_path

let astar initial =
  let dist = Hashtbl.create 100 in
  let parents = Hashtbl.create 100 in
  Hashtbl.add parents initial initial;
  let q = Heap.create () in
  Heap.insert q (initial, initial.h);
  Hashtbl.add dist initial 0;
  let rec loop () =
    match Heap.extract_min q with
    | None -> raise No_path
    | Some (x, _) when x.h = 0 -> reconstruct parents x
    | Some (x, _) ->
      let dx = Hashtbl.find dist x in
      let process v =
        let dv = dx + 1 in
        match Hashtbl.find_opt dist v with
        | Some d when d <= dv -> ()
        | _ ->
          Hashtbl.replace dist v dv;
          Heap.insert_or_decrease q (v, dv + v.h);
          Hashtbl.replace parents v x in
      List.iter process (successors x);
      loop () in
  loop ()
```

▸ **Question 11** C'est instantané pour les états situés à distance 10, 20 et 30, et l'on explore environ 40, 1 200 et 12 000 états. Pour l'état a distance 40, cela prend quelques secondes et l'on explore environ 350 000 états. Pour l'état à distance 50, c'est long : trois minutes sur ma machine, en version compilée en `-O3`. On explore un peu moins de 5 millions d'états.

▸ **Question 12** On montre par récurrence sur $m - p$ la propriété suivante : « DFS(m, e, p) renvoie VRAI si et seulement si on peut passer de e à l'état final en au plus $m - p$ étapes. ».

Initialisation Si $m - p$ vaut 0, l'algorithme renvoie immédiatement VRAI si e est final, et FAUX sinon (après les appels nécessairement infructueux sur les successeurs).

Hérédité À la lecture du code, on peut affirmer que l'algorithme renvoie VRAI si et seulement si e est final ou l'un des DFS$(m, x, p + 1)$ avec x successeur de e renvoie VRAI. Par hypothèse de récurrence, cette dernière condition équivaut à l'existence d'une suite « gagnante » de longueur au plus $m - p - 1$ depuis l'un des voisins de e, et donc d'une suite gagnante de longueur au plus $m - p$ depuis e.

En appliquant la propriété que l'on vient de prouver avec $e = \texttt{initial}$ et $p = 0$, on obtient que DFS$(m, \texttt{init}, 0)$ renvoie VRAI si et seulement si $\texttt{init}$ est à distance au plus m de l'état final.

▸ **Question 13** Dans les deux cas, la complexité en espace est en $O(n)$: la seule consommation est celle de la pile d'appel, dont la taille est celle du chemin actuel depuis le sommet initial. Comme on s'arrête dès qu'on trouve un chemin vers le sommet final, la longueur du chemin sera majoré par n.

Pour la complexité en temps :

- s'il y a exactement un sommet à distance k, l'appel DFS$(k, \texttt{init}, 0)$ va prendre un temps $O(k)$, et l'on fera successivement des appels pour $k = 0, \dots, n$. Au total, on obtient donc du $O(n^2)$.
- s'il y a 2^k nœuds à profondeur k, l'appel DFS$(k, \texttt{init}, 0)$ va prendre un temps $\sum_{p=0}^{k} 2^p \leqslant 2^{k+1}$ puisqu'on va explorer tous les états à profondeur au plus k. En sommant sur les appels pour $k = 0, \dots, n$, on obtient donc au total du $O\left(\sum_{k=0}^{n} 2^k\right) = O(2^n)$.

Il n'y a aucun intérêt dans le premier cas : un parcours en largeur se ferait en temps et en espace $O(n)$. En revanche, dans le deuxième cas, on obtient la même complexité en temps qu'un parcours en largeur classique (à une constante multiplicative près), mais une complexité en espace bien plus faible. Pour de très gros graphes, on sera souvent plus limité par l'espace que par le temps, d'où l'intérêt.

Remarque

En général, quand les complexités en temps et en espace sont les mêmes, c'est l'espace qui est le facteur limitant.

▸ **Question 14**

- Il n'est pas indispensable d'utiliser une exception, mais c'est sans doute le plus pratique.
- Puisqu'on va muter l'état, il faut travailler sur une copie.
- La partie cruciale va de la ligne 17 à la ligne 19 : on se déplace dans une direction, on tente le parcours depuis le nouvel état. En cas de succès, une exception sera levée, donc la ligne 19 n'est exécutée que si la tentative a échoué : il faut alors annuler le déplacement avant de continuer la recherche, en effectuant le déplacement opposé.

```
1  let opposite = function
2    | L -> R
3    | R -> L
4    | U -> D
5    | D -> U
6    | No_move -> No_move
7
8  let idastar_length initial =
9    let exception Found of int in
10   let state = copy initial in
11   let rec search depth bound =
12     if depth + state.h > bound then depth + state.h
13     else if state.h = 0 then raise (Found depth)
14     else
15       let minimum = ref max_int in
16       let make_move direction =
17           apply state direction;
18           minimum := min !minimum (search (depth + 1) bound);
19           apply state (opposite direction); in
20       List.iter make_move (possible_moves state);
21       !minimum in
22   let rec loop bound =
23     let m = search 0 bound in
24     if m = max_int then None
25     else loop m in
26   try
27     loop state.h
28   with
29   | Found depth -> Some depth
```

▸ **Question 15**

Montrons que si `search 0 m` renvoie m' (sans lever d'exception), alors $d(\mathrm{initial}, \mathrm{final}) \geqslant m'$. Par l'absurde, supposons qu'il existe un chemin $\mathrm{initial} = e_0, \dots, e_k = \mathrm{final}$ avec $k < m'$. Puisque la fonction n'a pas levé d'exception, ce chemin n'a pas été intégralement exploré : soit e_p le dernier état exploré lors de l'appel `search 0 m`. L'appel récursif `search p m` qui a été effectué avec `state = e_p` s'est arrêté en renvoyant $p + h(e_p)$ (puisqu'on n'a pas exploré e_{p+1}), donc $m' \leqslant p + h(e_p)$ (puisque l'appel racine renvoie le minimum des valeurs de retour de tous les appels récursifs). Or $d(e_p, \mathrm{final}) \leqslant k - p$ et h est admissible, donc $p + h(e_p) \leqslant p + k - p = k$. Ainsi, on a $m' \leqslant k$, ce qui est absurde.

On a maintenant l'invariant suivant dans la boucle principale : « $d(\mathrm{initial}, \mathrm{final}) \geqslant \mathrm{bound}$ ».

- C'est vrai au départ, puisqu'on part de $\mathrm{bound} = h(\mathrm{initial})$ et que l'heuristique est admissible.
- Cela reste vrai ensuite, puisque l'on ne fait un nouveau tour de boucle avec m' que si `search` n'a pas levé d'exception, et que dans ce cas on vient de montrer que $d(\mathrm{initial}, \mathrm{final}) \geqslant m'$.
- L'exception n'est levé que si l'on trouve un chemin vers l'état final de longueur $\leqslant \mathrm{bound}$, qui est donc forcément optimal d'après ce qui précède.

▸ **Question 16** Ici on utilise simplement le vecteur comme une pile : on aurait pu prendre une `ref list` ou une `Stack.t`.

```
let idastar initial =
  let counter = ref 0 in
  let state = copy initial in
  let exception Found in
  let path = Vector.create () in
  let rec search depth bound last_move =
    incr counter;
    if depth + state.h > bound then depth + state.h
    else if state.h = 0 then raise Found
    else
      let minimum = ref max_int in
      let make_move direction =
        if direction <> opposite last_move then (
          Vector.push path direction;
          apply state direction;
          minimum := min !minimum (search (depth + 1) bound direction);
          apply state (opposite direction);
          ignore (Vector.pop path)
        ) in
      List.iter make_move (possible_moves state);
      !minimum in
  let rec loop bound =
    let m = search 0 bound No_move in
    if m = max_int then None
    else loop m in
  try
    loop state.h
  with
  | Found ->
    Printf.printf "%d node expansions\n" !counter;
    Some path
```

En compilant en `-O3`, on trouve la solution optimale de longueur 50 en un peu moins d'une seconde, et en effectuant 13 millions d'appels à `search` (la plupart des nœuds sont bien sûr explorés plusieurs fois, donc le nombre de nœuds distincts explorés est plus faible). On voit la différence avec A* : on explore plus de nœuds (d'un facteur environ 2.5 ici), mais on consomme beaucoup moins de mémoire. On pourrait sans doute rendre A* aussi efficace que IDA* pour l'état à distance 50 en modifiant notre manière de le programmer, mais on serait bloqué par la mémoire pour un état à distance 80 de l'état final (80 se trouvant être la distance maximale dans le cas du taquin pour $n = 4$). Pour IDA*, trouver une telle solution serait long, mais possible.

Pour l'état `sixty_four`, situé à distance 64 de l'origine, IDA* permet de trouver la solution en un peu plus d'une minute, avec environ un milliard d'appels à `search`. Une recherche avec A* a échoué par manque de mémoire au bout de 7 minutes (elle consommait environ 8 giga-octets au moment où le système a dit stop).

Remarque

Il pourrait être intéressant de se limiter aux chemins élémentaires : cela nécessiterait de garder en mémoire les états rencontrés sur le chemin actuel depuis la racine, et de ne pas faire l'appel récursif s'il concerne l'un des états de ce chemin. Ce n'est pas complètement évident à faire de manière efficace.