Résolution d'un taquin

Présentation du problème

Un taquin est un tableau rectangulaire rempli, à l'exception d'une case vide, de pièces marquées de signes (lettres, chiffres, morceaux de dessin genre puzzle, ...). On suppose ici que ces signes sont distincts. Le jeu consiste à faire passer ce tableau d'une disposition initiale à une disposition finale par une succession de déplacements horizontaux ou verticaux de pièces adjacentes vers la case vide. Par exemple, avec un taquin 3×3, passer de :

$$\textit{départ} = \begin{pmatrix} 8 & 2 & 3 \\ 1 & 6 & 4 \\ 7 & & 5 \end{pmatrix} \qquad \text{à} \qquad \textit{but} = \begin{pmatrix} & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{pmatrix}.$$

Il n'est pas toujours possible de passer d'une disposition quelconque à une autre ; en pratique on pose un problème de taquin en déplaçant des pièces à partir de la position de référence *but* et en proposant à un autre joueur de reconstituer *but*. On simulera donc le jeu réel en utilisant comme *départ* une position obtenue par perturbation aléatoire en N coups à partir de la position *but* : les mouvements autorisés étant réversibles, ceci garantit l'existence d'une solution en N coups au plus.

Algorithmes de résolution

Étant donné une position p des pièces, on appelle *successeurs de* p les positions p' que l'on peut obtenir à partir de p par un mouvement. Il y a entre 2 et 4 successeurs, selon l'emplacement de la case vide. Un *chemin* est un couple (p, ℓ) où p est une position du taquin et ℓ une liste des mouvements à effectuer pour passer de *départ* à p (le dernier mouvement à effectuer étant en tête de ℓ).

Recherche en profondeur d'abord : on construit récursivement tous les chemins possibles partant de la position *départ* jusqu'à trouver *but*. A un instant donné on dispose d'une liste L de chemins en cours d'examen ; on extrait de L le premier chemin (p, ℓ) et si $p \neq \textit{but}$ on ajoute en tête de L tous les chemins $(p', m :: \ell)$ déduits de p par un mouvement m tels que p' est une position non encore vue. Si l'on aboutit à $L = \varnothing$ alors il n'y a pas de solution.

Recherche en largeur d'abord : ici aussi on examine tous les chemins possibles partant de *départ*, mais l'examen est conduit par longueur de chemin croissante (on est alors certain de trouver un chemin de longueur minimale menant de *départ* à *but* s'il en existe). En pratique, cela revient à placer les chemins $(p', m :: \ell)$ déduits de (p, ℓ) en queue de la liste L des chemins à examiner.

Recherche en largeur par les deux bouts : ici on examine simultanément et par longueur croissante tous les chemins partant de *départ* et tous ceux partant de *but* jusqu'à trouver deux chemins (p, ℓ) issu de *départ* et (p', ℓ') issu du *but* tels que $p = p'$. Alors $(\textit{but}, \text{miroir}(\ell')@\ell)$ est un chemin de longueur minimale conduisant de *départ* à *but*.

Recherche heuristiquement ordonnée : si p et p' sont deux positions, on note $d(p, p')$ la somme des distances pour la norme numéro 1 entre chaque pièce dans p et la même pièce dans p' (y compris la case vide). Si p'' est un successeur de p alors on a $d(p'', p') \geqslant d(p, p') - 1$, donc $d(p, p')$ est un minorant du nombre de mouvements à effectuer pour passer de p à p'. L'algorithme de recherche heuristiquement ordonnée consiste à examiner en premier les chemins (p, ℓ) tels que $d(p, \textit{but})$ est minimale. En pratique, ceci revient à maintenir la liste L des chemins à examiner classée par valeurs croissantes de la distance au but. Remarquer que si l'on trouve un chemin menant de *départ* à *but* par cette méthode, il n'est pas forcément de longueur minimale.

Implémentation

```
type position = {
  table : int vect vect; (* matrice des pièces *)
  i0 : int;              (* coordonnées de la case vide *)
  j0 : int
};;
type mouvement = Nord | Sud | Est | Ouest;;
```

On représente une position du taquin en CAML par une structure constituée d'une matrice `table` de taille $n \times p$ à coefficients entiers, la case vide ayant le numéro 0 et les autres pièces les numéros 1 à $np - 1$, et des coordonnées i_0, j_0 de la case vide. Les mouvements sont codés de manière symbolique, `Nord` représente un déplacement de la case vide d'une position vers le haut (j_0 inchangé, i_0 diminue d'une unité). `Sud`, `Est` et `Ouest` représentent un déplacement de la case vide d'une position vers le bas ($i_0 \longleftarrow i_0 + 1$), la droite ($j_0 \longleftarrow j_0 + 1$) et vers la gauche ($j_0 \longleftarrow j_0 - 1$).

La fonction `but` ci-dessous retourne un taquin de taille $n \times p$ dans la position à atteindre et la fonction `affiche_position` ci-dessous affiche une position sous forme matricielle :

```
(* position à obtenir *)
let but n p =
  let m = make_matrix n p 0 in
  for i=0 to n-1 do for j=0 to p-1 do
    m.(i).(j) <- p*i + j
  done done;
  {table = m; i0 = 0; j0 = 0}
;;
```

```
(* affiche une position *)
let affiche_position t =
  let n = vect_length t.table
  and p = vect_length t.table.(0) in
  for i=0 to n-1 do
    for j=0 to p-1 do
      printf__printf " %2d" t.table.(i).(j)
    done;
    print_newline()
  done
;;
```

Écrire les fonctions CAML suivantes :

`possibles : position -> mouvement list` : retourne la liste des mouvements possibles à partir d'une position donnée ;

`déplace : position -> mouvement -> position` : retourne la position obtenue à partir d'une position donnée en appliquant un mouvement donné (supposé possible). De façon à conserver la position donnée en argument, on effectuera une copie préalable de la matrice `table` pour constituer la position résultat.

La fonction suivante retourne une position obtenue en effectuant k mouvements aléatoires à partir de la position *but* :

```
(* taquin résoluble en au plus k mouvements *)
let random_position n p k =
  let t = ref(but n p) in
  for i=1 to k do
    let mvts = ref(possibles !t) in
    for i=1 to random__int(list_length !mvts) do mvts := (tl !mvts) done;
    t := déplace !t (hd !mvts)
  done;
  !t
;;
```

Algorithmes de recherche

Programmer les algorithmes de recherche en profondeur, en largeur, en largeur par les deux bouts et heuristiquement ordonnée. On tiendra à jour, outre la ou les listes des chemins à explorer, l'ensemble des positions déjà vues (de façon à ne pas tourner en rond, c'est indispensable pour l'algorithme de recherche en profondeur et c'est une source d'économies pour les autres algorithmes). On utilisera pour cela la bibliothèque `set` de CAML qui implémente la structure d'ensemble sur un type totalement ordonné à l'aide d'arbres binaires de recherche :

`let vus = ref(set__empty eq__compare)`	initialise `!vus` à l'ensemble vide ;
`vus := set__add x !vus`	ajoute l'élément x à l'ensemble `!vus` ;
`set__mem x !vus`	retourne un booléen disant si x appartient à l'ensemble `!vus` ;
`list_length(set__elements !vus)`	retourne le cardinal de l'ensemble `!vus`.

Pour les algorithmes de recherche en largeur et en largeur par les deux bouts, on évitera les concaténations en queue de liste (coûteuses en temps de calcul) en accumulant dans une liste auxiliaire les nouveaux chemins à examiner et en remplaçant la liste principale par cette liste auxiliaire lorsque la liste principale est vide. Pour l'algorithme de recherche heuristiquement ordonnée on stockera dans la liste L des triplets (p, ℓ, d) où p est une position, ℓ une liste de mouvements et d la distance de p à *but* de façon à ne pas recalculer cette distance lors des insertions dans L (qui doit être maintenue triée par valeurs croissantes de d).

Comparer expérimentalement ces algorithmes de recherche en termes de nombre de positions examinées lors de la résolution d'un même problème. Pour la recherche en profondeur on prendra des taquins de très petite taille car il se peut que l'algorithme explore *toutes* les positions possibles avant de trouver une solution, et on peut montrer qu'il y a $\frac{1}{2}(np)!$ telles positions...