Chemins dans $\mathbb{Z}^2$

On appelle « chemin dans $\mathbb{Z}^2$ » toute suite $\Gamma = (A_0, \ldots, A_n)$ de points du plan à coordonnées entières telle que l'on passe de chaque point au suivant par une translation dont le vecteur appartient à l'ensemble

$$\{N = (0,1), S = (0,-1), E = (1,0), W = (-1,0)\} \qquad \text{(Nord, Sud, Est et West).}$$

Le « mot » de Γ est la suite des vecteurs joignant deux points successifs : $\gamma = (\overrightarrow{A_0A_1}, \overrightarrow{A_1A_2}, \ldots, \overrightarrow{A_{n-1}A_n})$. On dit que :

- Γ a des points multiples s'il existe i, j distincts tels que $A_i = A_j$;
- Γ est fermé si $A_0 = A_n$;
- Γ est simple si $A_0 = A_n$ et si les points A_0, A_1,… , A_{n-1} sont distincts.

L'objet de ce TP est d'étudier quelques algorithmes sur les chemins : dire si un chemin comporte des points multiples, supprimer les boucles éventuelles en conservant les extrémités d'un chemin, remplir la surface du plan délimitée par un chemin simple. Un chemin Γ sera représenté en Caml soit par la liste des points (couples d'entiers) soit par le mot γ associé, le point initial étant pris par convention égal à $(0,0)$.

Pour pouvoir visualiser les chemins sur l'écran graphique il faut utiliser le système interactif `camlgraph` contenant les fonctions de base d'affichage graphique en plus du compilateur-interpréteur Caml standard (répondre `camllight -lang fr camlgraph` lorsque Xemacs demande quel interpréteur Caml il faut lancer).

Les fonctions graphiques les plus courament utilisées sont :

`open_graph : string -> unit`
: ouvre la fenêtre graphique. L'argument de type string permet de choisir les dimensions de cette fenêtre, donner une chaine vide pour ouvrir une fenêtre avec les dimensions par défaut.

`close_graph : unit -> unit`
: referme la fenêtre graphique.

`size_x : unit -> unit`
: retourne la largeur de la fenêtre graphique.

`size_y : unit -> unit`
: retourne la hauteur de la fenêtre graphique.

`clear_graph : unit -> unit`
: efface la fenêtre graphique.

`set_color : color -> unit`
: détermine la couleur des tracés suivants. Les couleurs prédéfinies sont `black`, `white`, `red`, `green`, `blue`, `yellow`, `cyan`, `magenta`.

`moveto : int -> int -> unit`
: déplace le crayon au point indiqué (abscisse, ordonnée) sans effectuer de tracé.

`lineto : int -> int -> unit`
: trace un segment depuis la position courante du crayon vers le point indiqué. Le crayon est ensuite plaçé en ce point.

`draw_circle : int -> int -> int -> unit`
: trace un cercle de centre et de rayon donnés (abscisse, ordonnée, rayon). Le crayon n'est pas déplacé.

`fill_circle : int-> int -> int -> unit`
: trace un disque de centre et de rayon donnés. Le crayon n'est pas déplacé.

`fill_rect : int -> int -> int -> int -> unit`
: trace un rectangle plein d'origine et de dimensions données (abscisse, ordonnée, largeur, hauteur). Le crayon n'est pas déplacé.

On entrera les définitions suivantes :

```
#open "graphics";;
open_graph "";;
type direction = N | S | E | W;;

                    (* +--------------------+
```

```
                        |  Chemin aléatoire  |
                        +--------------------+ *)

(* chemin comportant n fois N, s fois S, e fois E et w fois W *)
let rec random_chemin (n,s,e,w) =
  let t = n+s+e+w in
  if t = 0 then []
  else begin
    let x = random__int(t) in
    if      x < n     then N :: random_chemin(n-1,s,e,w)
    else if x-n < s   then S :: random_chemin(n,s-1,e,w)
    else if x-n-s < e then E :: random_chemin(n,s,e-1,w)
    else                   W :: random_chemin(n,s,e,w-1)
  end
;;
                         (* +-------------+
                            |  Affichage  |
                            +-------------+ *)

let pas = 20;;
let rayon1 = 5;;
let rayon2 = 3;;

let dessine points =

  (* origine = centre de la fenêtre *)
  let x0 = size_x()/2 and y0 = size_y()/2 in
  moveto x0 y0;
  fill_circle x0 y0 rayon1;

  (* tracé des segments et des points *)
  let rec trace = function
    | []           -> ()
    | (x,y)::suite ->
        let r = if suite = [] then rayon1 else rayon2 in
        fill_circle (x0 + x*pas) (y0 + y*pas) r;
        lineto (x0 + x*pas) (y0 + y*pas);
        trace(suite)
  in trace(points)
;;

(* noircit un rectangle de diagonale [(x1,y1),(x2,y2)] *)
let noircit (x1,y1) (x2,y2) =
  let x0 = size_x()/2       and y0 = size_y()/2       in
  let x3 = min x1 x2        and y3 = min y1 y2        in
  let dx = (max x1 x2) - x3 and dy = (max y1 y2) - y3 in
  fill_rect (x0 + x3*pas) (y0 + y3*pas) (dx*pas) (dy*pas)
;;
```

Le rôle de `#open "graphics"` est de rendre acessibles les fonctions graphiques (elles sont chargées en mémoire lorsqu'on lance `camlgraph` mais leurs noms ne sont connus de l'interpréteur qu'après ce `#open "graphics"`).

`random_chemin` tire au hasard un mot de chemin, γ, comportant n fois Nord, s fois Sud, e fois Est et w fois West. On obtient un mot de chemin fermé si (et seulement si) $n = s$ et $e = w$.

`dessine` trace dans la fenêtre graphique un chemin Γ (liste de points) passé en argument. Les quantités `pas`, `rayon1` et `rayon2` déterminent l'unité d'échelle et la taille des disques matérialisant les points du chemin.

`noircit` trace un rectangle dont on donne deux sommets opposés en tenant compte du facteur d'échelle `pas`.

1) Conversion mot $\longrightarrow$ liste de points.
Écrire une fonction `points : (int*int) -> direction list -> (int*int) list` qui calcule les points d'un chemin dont l'origine et le mot sont passés en argument. Tirer des mots de chemins au hasard et les faire afficher (effacer la fenêtre graphique entre deux tracés).

2) Détection et élimination des boucles.

a) Écrire une fonction `multiples : (int * int) list -> bool` qui teste si un chemin contient des points multiples. On utilisera la fonction standard : `mem : 'a -> 'a list -> bool` qui dit si une liste contient un élément donné en premier argument.

b) Pour supprimer les boucles dans le chemin $\Gamma = (A_0, \ldots, A_n)$ on utilise l'algorithme suivant :

1. Constituer la liste de couples $\ell = ((A_1, A_0), (A_2, A_1), \ldots, (A_n, A_{n-1}))$. *Cette liste sera utilisée comme « table de prédecesseurs » grâce à la fonction standard* `assoc` *:* `assoc` M ℓ *renvoie pour un point* M *le premier point* N *tel que le couple* (M, N) *appartient à* ℓ*, c'est à dire le premier prédecesseur de* M *dans* Γ *(si* M *n'a pas de prédecesseur,* `assoc` *déclenche une erreur).*

2. Constituer de proche en proche le chemin sans boucle $\Gamma' = (A'_0, \ldots, A'_p)$ *associé à* Γ *à l'aide des relations :*

$$A'_p = A_n, \quad A'_i = \text{premier prédecesseur}\,(A'_{i+1}), \quad A'_0 = A_0.$$

Programmer cet algorithme. On écrira une fonction `sans_boucle : (int*int) list -> (int*int) list` qui calcule Γ' à partir de Γ.

Remarque : la complexité asymptotique de `sans_boucle` est $O(n^2)$ car le temps d'une recherche effectuée par `assoc` est linéaire en la taille de la liste d'association. On pourrait améliorer cette complexité en utilisant une structure de données plus efficace qu'une liste d'association, par exemple une table de hachage ou un arbre binaire de recherche (structures dont le fonctionnement ne relève pas du cours de Maths-Sup).

La fonction suivante retourne un chemin simple aléatoire d'au moins n segments :

```
let rec random_boucle(n) =
  let ch = random_chemin (n,n,n,n-1)    in
  let pt = sans_boucle(points (0,0) ch) in
  if list_length(pt) >= n then pt @ [(0,0)] else random_boucle(n)
;;
```

On tire un chemin non fermé au hasard, on élimine les boucles et on le ferme avec un segment supplémentaire. Si le chemin obtenu est trop court alors on recommence (cela peut boucler indéfiniment mais la théorie des probabilités montre que cette éventualité a une probabilité nulle ; en pratique si n n'est pas trop grand on obtient un chemin convenable assez rapidement).

3) Remplissage
Soit Γ un chemin simple ; on veut « noircir » la région bornée délimitée par Γ (en fait la remplir avec la dernière couleur sélectionnée par `set_color`). Voici un algorithme possible pour ce faire :

1. Déterminer les ordonnées minimale et maximale, y_1 *et* y_2*, des points de* Γ*.*

2. Pour $y = y_1$*,* $y_1 + 1$*, …,* $y_2 - 1$ *faire :*
2.1. Déterminer les abscisses (x_i) *des segments verticaux de* Γ *coupant la droite horizontale d'ordonnée* $y + \frac{1}{2}$ *et classer ces abscisses par valeur croissantes. On obtient une liste* $(x_0, x_1, \ldots, x_{2n-1})$*.*
2.2 Noircir tous les rectangles $[x_{2i}, x_{2i+1}] \times [y, y + 1]$*.*
fin pour

On se convaincra intuitivement que le nombre d'abscisses trouvées est forcément pair et que les zones noircies sont les bonnes. Demander à votre professeur de mathématiques une justification rigoureuse.

Programmer cet algorithme. Le tri d'une liste ℓ d'entiers est obtenu par l'expression : `sort__sort (prefix <=)` ℓ.