```
(* PROGRAMME DE RECUIT SIMULÉ *)

(* V2 : cellules convexes *)


(* Plan du code :

I - Définition des types employés
II - Fonctions d'affichage graphique avec le module graphics
III - Fonctions de modification élémentaires d'une ville
       A. QUELQUES FONCTIONS PRÉLIMINAIRES
       B. FUSION DE DEUX CELLULES
       C. AJOUT D'UN SOMMET PAR MORCELEMENT D'UNE CELLULE
       D. SCINDER UNE CELLULE
       E. SUPPRIMER UN SOMMET
       F. DEPLACER UN SOMMET
IV - Fonction de coût
       A. TYPE POIDS
       B. CONVERSION VILLE-GRAPHE
       C. CALCUL DU COÛT
V - Fonction de recuit simulé
       A. MODIFICATION ÉLÉMENTAIRE ALÉATOIRE
       B. ÉTAT INITIAL
       C. NETTOYAGE
       D. FONCTION FINALE

En fin de code se trouvent des test permettant de tester la plupart des
fonctions (avec un affichage visuel lorsque ce c'est pertinant).

*)

(*------------------------------- I - TYPES : -------------------------------*)

(* CONVENTION : les cellules doivent être dans le sens HORAIRE *)

(* ces types n'ont pas besoin d'être définis, ils sont donnés à titre informatif
*)

type sommet_c = int * int ;;

type cellule = sommet_c list;;

type ville_basique = cellule list ;;
(* Les fonctions de modification élémentaire du III et les fonctions d'affichage
graphique recoivent le type ville_basique. *)

type ville = { ce : cellule list ; so : sommet_c list ; bord : sommet_c
list } ;;
(* Ce type sert dans le V pour les modifications aléatoires *)

(* les sommets des bords, listés dans 'bords' ne sont pas modifiables,
    'so' références tous les sommets de la ville,
    'ce' correspond au type ville_basique, soit la liste des cellules de la
ville. *)

(* ----------------------- II - MODULE GRAPHICS : -------------------------- *)

#load "graphics.cma";;

Graphics.open_graph "" ;;

Graphics.clear_graph();;

let dessine_cellule c =
```

```
    let a = 1 in (* agrandissement éventuel *)
    let (x0, y0) = List.hd c in
    let rec trace c = match c with
        | [(x,y)] -> Graphics.moveto (x*a) (y*a) ;
                     Graphics.lineto (x0*a) (y0*a)
        | (x1,y1)::(x2,y2)::q -> Graphics.moveto (x1*a) (y1*a) ;
                                 Graphics.lineto (x2*a) (y2*a) ;
                                 trace ((x2,y2)::q)
    in trace c ;;


let dessine_ville v =

    Graphics.open_graph "" ;
    Graphics.clear_graph() ;

    let rec aux v = match v with
        | [] -> ()
        | c::q -> dessine_cellule c ; aux q
    in aux v ;;


(* dessine un point de rayon r en (x,y) : *)

let point x y r =
   for i = (x-r) to (x+r) do
      for j = (y-r) to (y+r) do
         if ( ( (i-x)*(i-x) + (j-y)*(j-y) ) <= r*r ) then Graphics.plot i j
      done;
   done;
   ;;


(*dessine les cellules et les sommets (en gros), les sommets des bords en rouge
*)

let dessine_tout v =
    Graphics.clear_graph () ;
    dessine_ville v.ce ;
    let rec parcours l = match l with
        |[] -> ()
        | (x,y)::q -> point x y 3 ; parcours q
    in parcours v.so ;
    Graphics.set_color Graphics.red ;
    parcours v.bord ;
    Graphics.set_color Graphics.black ;;


dessine_tout depart2 ;;



(* ----------------- III - MODICICATIONS ÉLÉMENTAIRES ----------------------- *)


(* A. QUELQUES FONCTIONS PRÉLIMINAIRES : *)

(* conversions tableaux/liste : *)

(* Array.to_list

Array.of_list *)
```

```
(* Une fonction qui retire l'élément x de la liste l : *)

let rec retire x l = match l with (* retire une seule fois, la première
occurence de x dans l *)
    | [] -> []
    | a::q when a=x -> q
    | a::q -> a::(retire x q) ;;


(* Une fonction qui prend en entrée une cellule et renvoie une liste
correspondant à la cellule avec en plus le premier élément collé
à la fin : *)

let prolonge c =
    let t = List.hd c in List.rev (t::(List.rev c)) ;;
(* /!\ le résultat n'est plus une cellule *)


(* Une fonction qui renvoie le déterminant des deux vecteurs de a à b et de c à
d (a,b,c,d des points du plan à coordonnées entières) *)

let det a b c d =
    let (xa,ya) = (float_of_int (fst a), float_of_int (snd a))
    and (xb,yb) = (float_of_int (fst b), float_of_int (snd b))
    and (xc,yc) = (float_of_int (fst c), float_of_int (snd c))
    and (xd,yd) = (float_of_int (fst d), float_of_int (snd d))

    in let x1 = xb -. xa
    and x2 = xd -. xc
    and y1 = yb -. ya
    and y2 = yd -. yc

    in x1 *. y2 -. x2 *. y1 ;;


(* Une fonction qui teste si une cellule est convexe *)

let convexe c =
    let rec aux cel = match cel with
        | [] -> true
        | [x] -> true
        | [x;y] -> true
        | a::b::c::q when det b a b c >= 0. -> print_float (det b c b a) ; aux
(b::c::q)
        | _ -> false
    in aux (prolonge (prolonge c)) ;;



(* Une fonction qui pend en entrée deux segments (4 couples de coordonnées) et
retourne un booléen coorespondant à la valeur de "les
deux segments se coupent" : *)

(* rq : pour les segments qui se coupent en un sommet, on considèrera qu'ils ne
se coupent pas (inégalités strictes) *)

(* le raisonnement employé est présenté en Annexe 1 *)

let intersect a b d c =

    let (xa,ya) = (float_of_int (fst a), float_of_int (snd a))
    and (xb,yb) = (float_of_int (fst b), float_of_int (snd b))
    and (xc,yc) = (float_of_int (fst c), float_of_int (snd c))
    and (xd,yd) = (float_of_int (fst d), float_of_int (snd d))
```

```
    in
    let x1 = min xa xb and x2 = max xa xb
    and x3 = min xc xd and x4 = max xc xd
    and y1 = min ya yb and y2 = max ya yb
    and y3 = min yc yd and y4 = max yc yd
    in

    (* Cas d'un segment vertical et un horizontal : *)
        (* [a,b] vertical : *)
        if (xa=xb && yc=yd) then (xa<x4)&&(xa>x3)&&(yc<y2)&&(yc>y1)
        else begin

        (* [a,b] horizontal : *)
        if (xc=xd && ya=yb) then (ya<y4)&&(ya>y3)&&(xc<x2)&&(xc>x1)
        else begin

    (* Autres cas : *)
        (* [a,b] non vertical : *)
            if xa <> xb then

            begin
            let alpha1 = (yb -. ya) /. (xb -. xa)
            and alpha2 = (yd -. yc) /. (xd -. xc)
            in
            (* segments parallèles :*)
            if alpha1 = alpha2 then false

            (* Cas normal : *)
            else
                let beta1 = ya -. ( alpha1 *. xa )
                and beta2 = yc -. ( alpha2 *. xc )
                in
                let x = ( beta2 -. beta1 ) /. ( alpha1 -. alpha2 )

                in ( ( x > x1 ) && ( x > x3 ) && ( x < x2 ) && ( x < x4 ) )
            end

            else false (* si [a,b] vertical *)
        end
        end ;;



let intersection a b c d =
    (intersect a b c d)||(intersect c d a b) ;;




(* Une fonction qui indique si oui ou non le segment [a,b] coupe l'une des
arêtes au moins de la cellule c : *)

let coupe cel a b =
    let rec parcours c = match c with
        | [] -> false (* ce cas ne doit pas arriver *)
        | [x] -> false
        | x::y::q -> (intersection a b x y) || (parcours (y::q) )
    in let (x,y) = ( List.hd(List.rev cel) , List.hd(cel) )
    in (intersection x y a b ) || (parcours cel) ;;


(* Une fonction qui teste si le point p appartient à [a,b] *)
```

```
let passe_par p a b =
    let (x_a, y_a) = a and (x_b, y_b) = b and (x_p, y_p) = p in
    if ( x_p <= max x_a x_b ) && ( x_p >= min x_a x_b ) then
        (* conversion en flottants : *)
        let (xa,ya,xb,yb,xp,yp) = (float_of_int x_a,float_of_int y_a,float_of_int
x_b,float_of_int y_b,float_of_int x_p,float_of_int y_p)

        in let alpha = ( ya -. yb ) /. ( xa -. xb )
        in let beta = ya -. alpha *. xa

        in yp = alpha *. xp +. beta

    else false ;;


(* Une fonction qui calcule le barycentre d'une cellule (relativement aux
sommets) : *)

let barycentre c =
    let rec sommes c = match c with
        | [] -> (0,0,0)
        | (x,y)::q -> let ( s_x, s_y , n ) = sommes q in ( s_x + x , s_y + y , n
+ 1 )
    in let ( s_x, s_y , n ) = sommes c in ( s_x / n , s_y / n ) ;;




(* B. FUSION DE DEUX CELLULES : *)

(* Princie : On vérifie d'abord que les deux cellules ont deux sommets en commun
(consécutifs) puis on fusionne en enlevant l'arrête
correspondant. *)

(* D'abord une fonction cherchant deux sommets consécutifs (= arrête) communs à
deux cellules,
renvoie aussi un booleen indiquant si les cellules sont effectivement
fusionnables (ie une telle arrête existe) : *)

let arrete_commune c1 c2 =
    (* Pour l'arrête entre le dernier et le premier sommet on colle la tête à la
fin avec prolonge *)
    (* rq : une arrête commune à deux cellules sera toujours dans un sens dans
une cellule,
     et dans le sens opposé dans l'autre : on renverse c2 *)
    let cel1 = prolonge c1 and cel2 = List.rev (prolonge c2) in
    let err = ((-1,-1),(-1,-1)) in (* valeur d'erreur *)
    let rec parcours c1 = match c1 with
        | [] -> (err,false)
        | [x] -> (err,false)
        | a::b::q -> let bo = ( ( List.mem a c2 ) && (List.mem b c2) )
                              in if bo then ((a,b),true)
                                 else parcours (b::q)
    in let rec verifie c2 a b = match c2 with (* vérifie que a et b sont
consécutifs dans c2 aussi *)
        | [] -> false
        | [x] -> false
        | x::y::q -> if (x=a && y=b) then true
                     else verifie (y::q) a b  (* ATTENTION, si les cellules ne
sont pas codées dans le même sens ça ne marche pas !*)
    in let ((a,b),bo1) = parcours cel1 in let bo2 = verifie cel2 a b
    in if (bo1 && bo2) then ((a,b),true)
```

```
    else (err, false) ;;


(* Une fonction qui extrait le morceau de la cellule c entre a inclu et b
exclu : *)

let decoupe c a b = (* /!\ a avant b dans la cellule, on suppose qu'on a demandé
quelque chose de possible *)
    let rec aux c bo = match c, bo with
        | [],_ -> []
        | x::q, false -> if x = a then a::(aux q true) (* bo indique si on a
dépassé a*)
                         else aux q false
        | x::q, true -> if x <> b then x::(aux q true)
                        else []  (* découpe de a inclu à b exclu, sinon [b]
au lieu de [] *)
    in aux (c@c) false ;;


(* Une fonction qui termine la fusion dans le cas où il y avait plusieurs arêtes
en commun entre les deux cellules : *)

(* on commence par repérer le motif à éliminer dans la cellule créée (le motif -
a-b-a- si a et b sont des sommets : *)

let aba cel =
    let rec aux c = match c with
        | l when (List.length l) < 3  -> (false, ((-1,-1),(-1,-1)) )
        | a::b::c::q when a=c -> (true, (a,b))
        | _::q -> aux q
    in aux (cel@cel) ;;



(* ensuite on le supprime en le remplaçant par -a- : *)

(* prend en entrée une liste de longueur paire et renvoie la première moitié de
la liste *)

let moitie l =
    let n = List.length l and t = Array.of_list l in
    Array.to_list (Array.sub t 0 (n/2) ) ;;

moitie [2 ; 2 ; 2 ; 4] ;;

let supr_aba cel a b =
    let rec aux c = match c with
        | l when (List.length l) < 3  -> l
        | x::y::z::q when (x=a)&&(z=a)&&(y=b) -> a::(aux q)
        | x::y::z::q -> x::(aux (y::z::q))
    in let c_2fois = aux (cel@cel)
    in moitie c_2fois ;;


[(1,2) ; (3,4) ; (5,6) ; (9,1) ; (7,7) ; (9,1) ; (6,6) ] ;;

supr_aba [(1,2) ; (3,4) ; (9,1) ; (7,7) ; (9,1) ; (5,6) ; (6,6)  ] (9,1)
(7,7) ;;


(* fonction de fusion des cellules : *)

let fusionne c1 c2 =
```

```
    let ((a,b),bo) = arrete_commune c1 c2

    in if not bo then failwith "cellules pas fusionnables"

    else let bout1 = decoupe c1 b a and bout2 = decoupe c2 a b
    in let cel = ref (bout1@bout2) and boo = ref true

    in while (!boo) do
        let (bo, (a,b)) = aba (!cel) in
        boo := (!boo)&&bo ;
        if bo then cel := supr_aba (!cel) a b
        done ;

    !cel ;;


(* fonction qui ajoute la nouvelle cellule à ville et retire les deux
anciennes : *)

let fusion ville c1 c2 =
    let rec retire1 v = match v with
        | [] -> []
        | x::q when ( ( x = c1 ) || ( x = c2 ) ) -> retire1 q
        | x::q -> x::(retire1 q)
    in (fusionne c1 c2)::(retire1 ville);;




(* C. AJOUT D'UN SOMMET PAR MORCELEMENT D'UNE CELLULE : *)

(* Principe :

- On veut découper la cellule en parts de gâteau en ajoutant des arêtes entre le
barycentre de la cellule en les sommets de la
  cellule
- Toutes les arêtes ainsi définies ne peuvent pas être tracées, car celles qui
créeraient une intersection avec un bord de la
  cellule initiale ne seraient pas correctes.
- On commence donc par repérer les sommets avec lesquels ont peut créer une
arête sans créer d'intersection : *)


(* Une fonction qui prend en entrée une cellule et renvoie une liste de 0 et de
1 de même longueur. Aux positions correspondant à
un 'bon' sommet on trouve un 1, aux autres un 0 :  *)

let liste_bons cel b = (* b le barycentre déjà calculé *)
    let rec aux c = match c with
        | [] -> []
        | s::q when (coupe cel s b) -> 0::(aux q)

        | _::q -> 1::(aux q)
    in aux cel ;;


(* Pour éviter de créer trop de triangles, on retire aléatoirement des sommets
'ok' pour en garder au plus 2. Ce choix est arbitraire *)

(* compte les occurences de x dans la liste l *)
let rec compte x l = match l with
    | [] -> 0
    | a::q when a=x -> 1 + (compte x q)
    | _::q -> compte x q ;;
```

```
(* donne la liste des positions des x dans t un tableau :*)
let posi x t =
    let n = Array.length t and pos = ref [] in
    for i = 0 to (n-1) do
        if t.(i) = x then pos := i::(!pos)
    done ;
    !pos ;;

(* Renvoie une liste correspondant à au plus 3 sommets 'ok' : *)

let liste_bis cel b =
    let l = liste_bons cel b in let n = compte 1 l in
    if n <= 3 then l
    else begin
        let t = Array.of_list l in
        for i = 0 to (n-5) do (* on garde 2 sommets ok, on sait que n >= 4 *)
            let pos = Array.of_list (posi 1 t) in
            let k = Random.int (n-i) in
            let p = pos.(k)
            in t.(p) <- 0
        done ;
        Array.to_list t
        end ;;


(* On utilise ensuite cette liste pour créer les nouvelles cellules *)

(* Il faut aussi s'assurer que le barycentre est bien situé à l'intérieur de la
cellule. Il s'agit d'un problème du type Point
in Polygone (PIP), on utilise le principe de l'algorithme "Ray casting" qui est
le suivant :

On trace le segment entre le barycentre et un point extérieur à la cellule (ici
(0,0)). On compte combien de fois ce segment
coupe la cellule. Si c'est un nombre pair, sur le trajet on est 'sorti' de la
cellule autant de fois qu'on y est 'entré', pour
finir à l'extérieur, donc le point de départ (le barycentre) est à l'extérieur
de la cellule, si c'est un nombre impair, par le
même raisonnement, on déduit que le barycentre est à l'intérieur de la cellule.

Cette méthode présente l'énorme avantage de fonctionner quelque soit la forme de
la cellule (si elle n'est pas convexe notamment).

J'ajoute deux faits :

- Le segment ne coupe qu'une fois au plus une arête ( détecté par la fonction
'intersection')
- Le segment passe une fois au plus par un sommet ( détecté par la fonction
'passe_par' )

Le nombre d'intersections entre la cellule est le segment est donc la somme des
intersections aux arêtes et de celles aux sommets.

*)

(* La fonction qui teste si le point p est dans la cellule cel (PIP) : *)

let est_dans p cel =
    let rec compte c = match c with
        (* [] -> 0 pas atteint normalement *)
        | [s] when ( intersection (0,0) p s (List.hd cel) ) -> if (passe_par s
(0,0) p) then 2 else 1
        | [s] -> if (passe_par s (0,0) p) then 1 else 0
```

```
       | x::y::q when (intersection (0,0) p x y ) -> if (passe_par x (0,0) p)
then 2 + ( compte (y::q) )
                                                   else 1 + ( compte (y::q) )
       | x::q -> if (passe_par x (0,0) p) then 1 + ( compte q )
                 else compte q
   in let k = compte cel
   in if (k mod 2) = 0 then false
      else true ;;


(* La fonction finale de morcellement, elle renvoie aussi un booleen indiquant
si elle a effectivement pu effectuer une modification : *)


let morcelle ville cel =
(* /!\ La fonction construit les listes retournées, pour ne pas les retourner
une par une à la fin, on travaille dès le départ sur
List.rev cel. Attention, la fonction est adaptée à cette structure précise. *)

   let b = barycentre cel in

   if not (est_dans b cel) then (ville, false) else (* si le barycentre est à
l'exterieur de cel, on abandonne *)

   let rev = List.rev cel in
   let bons = liste_bis rev b in

   let rec compte1 l = match l with (* on compte k le nombre de 'bons' sommets
*)
       | [] -> 0
       | 1::q -> 1 + (compte1 q)
       | _::q -> compte1 q

   in let k = compte1 bons in

      if (k=1)||(k=0) then (ville, false) (* si on ne peut pas diviser la
cellule en plusieurs cellules, on ne modifie rien *)

       else

          let filles = Array.make k [] in (* On va créer k nouvelles cellules
filles, stockées dans le tableau filles *)

          let t = Array.of_list bons and c = Array.of_list rev in

          let m = ref 0 and n = List.length rev in (* m est l'indice qui va
nous permettre de parcourir 'filles' *)

          for i=0 to n-1 do

             if t.(i) = 0 (* si le sommet c.(i) n'est pas 'bon' on l'ajoute à
la cellule filles.(!m) et on passe au sommet suivant *)
                 then
                    filles.(!m) <- (c.(i))::(filles.(!m))
             else
                 begin
                 (* si le sommet c.(i) est 'bon', on l'ajoute à la cellule
actuelle, on passe à la cellule suivante,
                 on ajoute encore ce sommet à la cellule suivante (c'est son
premier sommet), puis on passe au sommet suivant.
                 Rq : l'ajout du barycentre à la cellule se fait plus tard. *)
                 filles.(!m) <- (c.(i))::(filles.(!m)) ;
                 if !m < k-1 then
                    begin
```

```
                    m := !m + 1 ;
                    filles.(!m) <- (c.(i))::(filles.(!m))
                    end
                end
            done ;

        let rec retire1 v = match v with
            | [] -> []
            | x::q when x = cel -> retire1 q
            | x::q -> x::(retire1 q)

        in let v = ref (retire1 ville)

        in for i=1 to k-2 do
            v := ( b::filles.(i) )::(!v)
        done ;

        (* 4 cas possibles, en fonction de si le premier et le dernier sommet
de rev sont 'bons' ou pas : *)

        match t.(0), t.(n-1) with
            | 1, 0 -> v := ( b::c.(0)::filles.(k-1) )::(!v) ;
                      v := ( b::filles.(0) )::(!v) ;
                      (!v,true)

            | 0, 0 -> v := ( b::(filles.(0)@filles.(k-1) ) )::(!v) ;
                      (!v, true)

            | 1, 1 -> v := ( b::filles.(0) )::(!v) ;
                      v := ( b::filles.(k-1) )::(!v) ;
                      v := [ c.(n-1) ; b ; c.(0) ]::(!v) ;
                      (!v,true)

            | 0, 1 -> v := ( b::filles.(k-1) )::(!v) ;
                      v := ( b::( filles.(0)@[c.(n-1)] ) )::(!v) ;
                      (!v,true)
    ;;



(* D. SCINDER UNE CELLULE *)

(* Une fonction qui scinde la cellule c en 2 en ajoutant l'arête [a,b], a et b
étant des sommets de c *)

let scinde ville c a b =

    if (coupe c a b) then (ville, false) (* on vérifie qu'on ne crée pas
d'intersection *)

    else  let rec retire1 v = match v with
        | [] -> []
        | x::q when x = c -> retire1 q
        | x::q -> x::(retire1 q)

    in let c1 = b::(decoupe c a b) and c2 = a::(decoupe c b a)

    in ( c1::c2::(retire1 ville) , true ) ;;



(* E. SUPPRIMER UN SOMMET *)
```

```
(* Principe : On fusionne toutes les cellules auxquelles le sommet appartient.
*)

(* Une première fonction liste les celulles de ville auxquelles s appartient :
*)

let rec touchent s ville = match ville with
    | [] -> []
    | c::q when (List.mem s c) -> c::(touchent s q)
    | _::q -> touchent s q ;;


(* Une fonction qui fusionne deux cellules consécutives touchant s : *)

let fusionne2 s v =
    let cel::l = touchent s v in
    let rec parcours l = match l with
        | [] -> failwith "fusionne2"
        | c::q -> let (_,b) = arrete_commune cel c in if b then fusion v c cel
                  else parcours q
    in parcours l ;;


(* La fonction finale qui supprime le sommet s dans la ville 'ville' : *)

let supprime s ville =
    let n = List.length (touchent s ville) in (* n-1 fusions à exécuter *)
    let rec aux v i = match i with
        | x when x <= 0 -> failwith "supprime"
        | 1 -> v
        | i -> aux (fusionne2 s v) (i-1)

    in let v = aux ville n in

    (* Il reste une seule arrête à supprimer, qui est deux fois dans la
cellule : *)

    let [c] = touchent s v in

    let rec enleve c = match c with (* enleve l'arrête avec s de c *)
        | [] -> []
        | x::q when x=s -> q
        | x::q -> x::(enleve q)

    in let cel = enleve c

    in let rec enleve2 v = match v with (* enleve la cellule avec s de ville *)
        | [] -> []
        | x::q when x=c -> q
        | x::q -> x::(enleve2 q)

    in cel::( enleve2 v) ;;




(* F. DEPLACER UN SOMMET *)

(* pas encore codée *)


(*------------------------ IV - FONCTION DE COÛT : -------------------------*)
```

```
(* A. TYPE POIDS : *)


type poids = Inf | P of float ;;

(* Somme : *)
let som a b = match a, b with
   | Inf, _ -> Inf
   | _, Inf -> Inf
   | (P a), (P b) -> P (a +. b) ;;

(* Minimum : *)
let mini a b = match a, b with
   | Inf, x -> x
   | x, Inf -> x
   | (P a), (P b) -> P (min a b) ;;

(* Test de a < b : *)
let inferieur a b = match a, b with
   | Inf, _ -> false
   | _, Inf -> true
   | (P a), (P b) -> a < b ;;



(* B. CONVERSION VILLE-GRAPHE : *)

(* On veut un graphe en matrice d'adjacence correspondant à la ville. *)

(* Fonction qui calcule la distance entre les points a et b : *)

let distance a b = (* retourne un type float *)
    let (x_a, y_a) = a and (x_b, y_b) = b in
    let (xa, ya, xb, yb) = (float_of_int (x_a),float_of_int (y_a),float_of_int
(x_b),float_of_int (y_b))
    in sqrt ( ( (xb -. xa) ** 2. ) +. ( (yb -. ya) ** 2. ) ) ;;


(* Fonction qui donne la position du sommet a dans le tableau de une ligne s *)

let position a s =
    let rec aux i = match i with
        | -1 -> failwith "position"
        | x when s.(x) = a -> x
        | _ -> aux (i-1)
    in aux ( (Array.length s) - 1 ) ;;


(* Fonction de conversion : *)

let graphvil v =
    (* on crée une liste des sommets de la ville en s'assurant que chaque sommet
n'y apparaît qu'une fois : *)
    (* Pour cela on cherche avec sommets_c dans chaque cellule les sommets non
encore listés par sommets_v : *)
    let rec sommets_c c sommets = match c with
        | [] -> []
        | x::q when List.mem x sommets -> sommets_c q sommets
        | x::q -> x::(sommets_c q (x::sommets))

    in let rec sommets_v v sommets = match v with
        | [] -> []
```

```
       | c::q -> let l = (sommets_c c sommets) in l@(sommets_v q (l@sommets) )

    in let sommets = Array.of_list ( sommets_v v [] ) in

    let n = Array.length sommets in

    let g = Array.make_matrix n n Inf in (* par défaut la valeur Inf indique que
deux sommets ne sont pas reliés *)

    (* On parcours les cellules de la ville et on ajoute pour chaque arête de la
ville l'arête équivalente dans la matrice : *)
    let ajoute_c c = (* pour une cellule donnée *)
        let s = List.hd c in
        let rec aux c = match c with
            | [] -> ()
            | [a] -> let i = position a sommets and j = position s sommets and d
= P (distance a s) in
                     g.(i).(j) <- d ;
                     g.(j).(i) <- d
            | a::b::q -> let i = position a sommets and j = position b sommets
and d = P (distance a b) in
                         g.(i).(j) <- d ;
                         g.(j).(i) <- d ;
                         aux (b::q)
        in aux c

    in let rec ajoute_v v = match v with (* pour toute la ville *)
        | [] -> ()
        | c::q -> ajoute_c c ; ajoute_v q

    in ajoute_v v ;

    g ;;

    (* rq : on peut retourner (g, sommets) si on veut les coordonnées des
sommets *)


(* C. CALCUL DU COÛT : *)

(* POIDS DES PLUS COURTS CHEMINS *)


(* renvoie les poids des plus courts chemins entre chaque points du graphe *)

let floydwarshall w =
   let n = Array.length w in
   let m = Array.make_matrix n n Inf in
   for i = 0 to n-1 do
      for j = 0 to n-1 do
         m.(i).(j) <- w.(i).(j)
      done
   done ;
   for k = 0 to n-1 do
      for i = 0 to n-1 do
         for j = 0 to n-1 do
            m.(i).(j) <- mini m.(i).(j) (som m.(i).(k) m.(k).(j))
         done
      done
   done ;
   m ;;
```

```
(* La fonction proposée ici (choix un peu arbitraire) :

- on additionne les poids de tous les (plus courts) chemins
- plus un modérateur : la distance totale en routes existantes -> longroute

rq : plus tard, éventuellement, on pourra cibler des points et leur donner plus
d'importance

*)


(* conversion poids -> flottant : *)

let float_of_poids x = if x = Inf then failwith "Un point n'est pas relié :
intersection vide"
                        else let (P a) = x in a ;;


(* Renvoie la somme de la longueur de toutes les routes (=arêtes) de la ville *)

let longroute v = (* /!\ renvoie le double de la longueur réelle car compte
toutes les arrêtes deux fois *)
    let long_c c =
        let s = List.hd c in
        let rec aux c = match c with
            | [] -> 0. (* normalement ça n'arrive pas *)
            | [a] -> let d = ( distance a s ) in d
            | a::b::q -> let d = ( distance a b ) in d +. aux (b::q)
        in aux c

    in let rec long_v v = match v with
        | [] -> 0.
        | c::q -> (long_c c) +. (long_v q)

    in long_v v ;;


(* Calcul du coût : *)

let cout_1 v =

    let g = graphvil v in let chemins = floydwarshall g in

    let n = Array.length g in

    let cout = ref 0. in

    for i=0 to (n-1) do
        for j=0 to (n-1) do
            cout := (!cout) +. ( float_of_poids (chemins.(i).(j)) )
        done
    done ;

    !cout +. ( (longroute v) /. 2. ) ;; (* on a donné le même poids aux deux
fonctions ici *)


(*------------------- V - FONCTION DE RECUIT SIMULÉ : ------------------------*)



(* A. UNE MODIFICATION ELEMENTAIRE : *)
```

```
(* Les fonctions suivantes prennent en compte la répétition du procédé jusqu'à
ce que ce qu'une modification ait effectivement eut
lieu. Elles ne renvoient pas de booleen mais directement la nouvelle ville *)

(* Radom.int n renvoie un entier aléatoire entre 0 et n-1 *)

(* Une fonction qui tire deux entiers aléatoires différents entre 0 et m-1 : *)

let indices_differents m =
   let limite = 1000 in (* on s'autorise 1000 tentatives avant d'arrêter *)
   let rec aux i j compteur = match i, j, compteur with
          | _, _, x when x > limite -> failwith "indices_différents"
          | i, j, _ when i=j -> aux (Random.int m) (Random.int m) m
          | _ -> (i,j)
   in aux (Random.int m) (Random.int m) m ;;



let indices_diff2 m = (* donne deux indices non consécutifs, ni
(premier,dernier), dans l'ordre croissant*)
   let limite = 1000 in
   let rec aux compteur = match compteur, indices_differents m with
      | x, _ when x > limite -> failwith "indices_diff2"
      | _, (i,j) when (i=j+1)||(j=i+1)||((i=0)&&(j=m-1))||((j=0)&&(i=m-1)) ->
aux (compteur + 1)
      | _, (i,j) -> (min i j, max i j)
   in aux 0 ;;



(* Renvoie un sommet aléatoire de ville qui n'est pas sur le bord : *)

(*
let sommet_int_al ville =
   let limite = 1000 in
   let sommets = ville.so and bor = ville.bord
   in let n = List.length sommets in let som = Array.of_list sommets in

   let rec aux s compteur = match (List.mem s bor), compteur with
      | _, x when x > limite -> failwith "sommet_int_al"
      | false, _ -> s
      | _ -> aux ( som.(Random.int n) ) (compteur + 1)

   in aux ( som.(Random.int n) ) 0 ;;
*)

let sommet_int_al ville =
   let limite = 1000 and n = List.length (ville.so) and sommets = Array.of_list
(ville.so) in
   let rec aux compteur = match compteur, Random.int n with
      | c, _ when c > limite -> failwith "sommet_int_al"
      | _, x when  ( List.mem (sommets.(x)) (ville.bord) ) -> aux (compteur +
1)
      | _, x -> (sommets).(x)
   in aux 0 ;;




(* Les 4 fonctions suivantes fonctionnent selon le même principe :
```

```
On tire un sommet ou une cellule au hasard dans la ville, on essaye d'appliquer
la modification,
on recommence jusqu'à ce que la modification fonctionne. *)

let scind_al ville =
    let b = ref false and nouv = ref ville in
    while not !b do
        let v = ville.ce in let v2 = Array.of_list v in let n = List.length v
        in let k = Random.int n in let c = v2.(k) in let c2 = Array.of_list c
        in let m = List.length c in let (i,j) = indices_diff2 m

        in let (vi,a) = scinde v c c2.(i) c2.(j)
        in if a then begin
            nouv := { ce = vi ; so = ville.so ; bord = ville.bord } ;
            b := true
            end
    done ;
    !nouv  ;;


let morcel_al ville =
    let v = ville.ce in let v2 = Array.of_list v in let n = List.length v in
    let b = ref false and nouv = ref v and cel = ref [] in
    while not !b do
        let k = Random.int n in let c = v2.(k)
        in let (vi, a) = morcelle v c in
        if a then begin
            nouv := vi ;
            cel := c ;
            b := a
        end
    done ;
    let bary = barycentre !cel in
    { ce = !nouv ; so = bary::(ville.so) ; bord = ville.bord } ;;



let fusion_al ville =
    let s = sommet_int_al ville in
    { ce = (fusionne2 s ville.ce) ; so = ville.so ; bord = ville.bord} ;; (* les
sommets supprimés sont enlevés dans le nettoyage, ainsi que les arêtes
résiduelles *)


let suppr_al ville =
    let s = sommet_int_al ville in
    { ce = (supprime s (ville.ce)) ; so = retire s (ville.so) ; bord =
ville.bord} ;;


(* + déplacer un sommet *)




(* fonction qui effectue modification aléatoire parmis les 4 précédentes : *)

let modif v =
    let n = Random.int 6 in match n with
        | 0 -> scind_al v
        | 1 -> morcel_al v
        | 2 -> fusion_al v
        | 3 -> suppr_al v
        | 4 -> fusion_al v
```

```
       | 5 -> suppr_al v ;;


(* B. ÉTAT INITIAL : *)

(* Je reçoie de Nathan (mon binôme) une liste de points sur le bord de la ville
actuelle et une liste de points sur le bord de la
ville future, les deux listes sont de même longueur et les points se
correspondent deux à deux. Pour créer la ville de départ, on
relie ces points par 4 pour former des cellules en forme de quadrilatères
recouvrant la couronne de la partie nouvelle de la ville. *)

let grille nouveaux_points anciens_points = (*sens anti-trigo *)
    if (List.length nouveaux_points) <> (List.length anciens_points) then
failwith "listes incorrectes";
    let v = [] in
    let rec remplit l1 l2 v debut = match l1, l2 with
        | [],[] -> v (*normalement les deux listes font la même longueur*)
        | [(x11,y11)], [(x21,y21)] -> let [(x12,y12);(x22,y22)] = debut in
[(x11,y11);(x12,y12);(x22,y22);(x21,y21)]::v
       | (x11,y11)::(x12,y12)::q1 , (x21,y21)::(x22,y22)::q2 -> remplit
((x12,y12)::q1) ((x22,y22)::q2) ([(x11,y11);(x12,y12);(x22,y22);(x21,y21)]::v)
debut
    in let debut = [List.hd nouveaux_points; List.hd anciens_points]
    in remplit nouveaux_points anciens_points v debut ;;


(* La ville ainsi obtenue ne possède que des sommets sur le bord. Certaines
modifications sont donc impossibles. On lui ajoute donc
k sommets intérieurs en appriquant morcel_al k fois avec la fonction suivante :
*)

let augmente ville k =
    let v = ref ville in
    for i=1 to k do
        v := morcel_al !v
    done ;
    !v ;;


(* C. FONCTION DE NETTOYAGE *)

(* On passe en revue toutes les cellules du graphe et on supprime les motifs aba
et
les sommets reliés à rien de la liste des sommets *)

(* Une fonction qui retire les éléments d'ue liste l1 à la liste l2 (la première
occurence de chaque élément) :*)

let rec retire_liste l1 l2 = match l1 with
    |[] -> l2
    |x::q -> retire_liste q (retire x l2) ;;

retire_liste [(1,1) ; (2,2) ; (3,3)] [(5,5); (8,8); (2,2) ;(0,0) ;(7,7) ; (3,3)]
;;


let nettoyage1 v = (* enleve les arêtes en trop et les sommets qui vont avec *)

    let enleve_aba c =
        let cel = ref c and boo = ref true and sommets_supprimes = ref []
        in while (!boo) do
            let (bo, (a,b)) = aba (!cel) in
```

```
          boo := (!boo)&&bo ;
          if bo then begin
              cel := supr_aba (!cel) a b ;
              sommets_supprimes := b::(!sommets_supprimes)
              end
       done ;
       (!cel,!sommets_supprimes)

   in  let rec parcours l = match l with
       | [] -> ([],[])
       | c::q -> let (cel,s)= enleve_aba c
                   and (q1,s1) = parcours q
                   in (cel::q1, s@s1)
   in let (l,sommets) = parcours (v.ce)
   in {ce = l ; so = retire_liste sommets (v.so) ; bord = v.bord} ;;



let nettoyage2 ville = (*ressence les sommets oubliés*)
   let l = ref ville.so in
   let rec parcours_c c = match c with
       | [] -> ()
       | x::q -> if not (List.mem x (!l)) then l := x::(!l) ; parcours_c q
   and parcours_v v = match v with
       | [] -> ()
       | c::q -> parcours_c c ; parcours_v q
   in parcours_v ville.ce ;
   {ce = ville.ce ; so = !l ; bord = ville.bord} ;;




(* Supprime les cellules de 1 ou 2 sommets *)

let nettoyage3 v =
   let rec aux l = match l with
       | [] -> []
       | c::q when ((List.length c) < 3 ) -> aux q
       | c::q -> c::(aux q)
   in {ce = aux (v.ce) ; so = v.so ; bord = v.bord } ;; (* le nettoyage4 enlève
les sommets correspondants, il doit être après le 3 *)




let nettoyage4 v = (* supprime les sommets isolés*)
   let rec cherche v s = match v with
       | [] -> false
       | c::q -> (List.mem s c)||(cherche q s)
   in let rec aux l = match l with
       | [] -> []
       | s::q -> if (cherche (v.ce) s) then s::(aux q)
                 else aux q
   in {ce = v.ce ; so = aux (v.so) ; bord = v.bord} ;;




(* D. FONCTION FINALE : *)
```

```
(* on suppose qu'on a :
    nouveaux_points
    anciens_points  (Nathan)*)


(* Renvoie true avec une probabilité de p , p entre 0 et 1 exclu : *)

let proba p =
    let x = Random.float 1. in
    (x < p ) ;;


(* Fonction de recuit simulé : *)

let recuit_simule anciens_points nouveaux_points =

    let depart = { ce = grille nouveaux_points anciens_points ;
                   so = nouveaux_points@anciens_points ;
                   bord = nouveaux_points@anciens_points }

    and nbr_tirages = 100 and lambda = 0.99 and t = ref 100. (* arbitraire *) in
    (* affichages & résultats : *)
    let tab = Array.make nbr_tirages (-1.) in
    (* --------- *)
    let ville = ref (augmente depart 20) in
    let e1 = ref (cout_1 (!ville).ce) in

    for i=0 to (nbr_tirages - 1) do

        ville := nettoyage1 (!ville) ;
        ville := nettoyage2 (!ville) ;
        ville := nettoyage3 (!ville) ;
        ville := nettoyage4 (!ville) ;

        dessine_tout (!ville) ;
        print_newline () ;
        print_int i ;
        print_newline () ;
        print_float !e1 ;

        let v = modif !ville in
        let e2 = cout_1 v.ce in

        tab.(i) <- !e1 ;

        if e2 < !e1 then begin
            ville := v ;
            t := lambda *. !t ;
            e1 := e2
            end

        else let b = proba ( exp ( -. ( e2 -. !e1 ) /. !t ) ) in
            if b then begin
                ville := v ;
                t := lambda *. !t ;
                e1 := e2
                end
            else t := lambda *. !t
    done ;
    (tab, !ville) ;;
```

```
(* Version partant d'une ville "depart" quelconque : *)

let recuit2 depart =

    let nbr_tirages = 100 and lambda = 0.99 and t = ref 100. (* arbitraire *) in
    (* affichages & résultats : *)
    let tab = Array.make nbr_tirages (-1.) in
    (* --------- *)
    let ville = ref (augmente depart 1) in
    let e1 = ref (cout_1 (!ville).ce) in

    for i=0 to (nbr_tirages - 1) do

        ville := nettoyage1 (!ville) ;
        ville := nettoyage2 (!ville) ;
        ville := nettoyage3 (!ville) ;
        ville := nettoyage4 (!ville) ;

        dessine_tout (!ville) ;
        print_newline () ;
        print_int i ;
        print_newline () ;
        print_float !e1 ;

        let v = modif !ville in
        let e2 = cout_1 v.ce in

        tab.(i) <- !e1 ;

        if e2 < !e1 then begin
            ville := v ;
            t := lambda *. !t ;
            e1 := e2
            end

        else let b = proba ( exp ( -. ( e2 -. !e1 ) /. !t ) ) in
            if b then begin
                ville := v ;
                t := lambda *. !t ;
                e1 := e2
                end
            else t := lambda *. !t
    done ;
    (tab, !ville) ;;




(*-------------------------------------------------------------------------------*)
(*--------------------------- TESTS DIVERS --------------------------------------*)
(*-------------------------------------------------------------------------------*)



let ville_test = [[(1,1);(3,1);(5,3);(1,3)];
[(3,1);(6,1);(5,3)];
[(6,1);(8,5);(5,3)];
[(6,1);(10,1);(10,8);(8,5)];
[(1,3);(5,3);(3,7);(1,6)];
[(5,3);(8,5);(3,7)];
[(3,7);(8,5);(10,8);(8,10)];
[(1,6);(3,7);(8,10);(1,10)]] ;; (* /!\ codée dans le sens trigonométrique *)

dessine_ville ville_test ;;
```

```
intersection (4,3) (1,1) (5,1) (2,2) ;; (* vrai *)

intersection (4,3) (1,1) (1,4) (3,3) ;; (* faux *)

barycentre [(1,1);(3,1);(5,3);(1,3)] ;;


divise [(1,1);(3,1);(5,3);(1,3)] ;;
dessine_ville ( divise [(1,1);(3,1);(5,3);(1,3)] ) ;;

let cel = [ (1,4) ; (3,7) ; (6,4) ; (6,7) ; (10,4) ; (7,2) ; (4,4 ); (4,1) ] ;;

liste_bons cel (barycentre cel) ;;

liste_bis cel (barycentre cel) ;;

let (v, _) = ( morcelle [cel] cel ) in dessine_ville v ;;


(* test supprime : *)

dessine_ville ville_test ;;

Graphics.clear_graph();;

dessine_ville (supprime (5,3) ville_test) ;;

(* ---- *)

est_dans (5,3) [(1,1) ; (4,1) ; (4,4) ; (1,4)] ;;

intersection (7,6) (0,0) (4,1)  (4,4) ;;

est_dans (2,3) [(1,1) ; (4,1) ; (4,4) ; (1,4)] ;;

est_dans (barycentre cel) cel ;;

dessine_cellule [(1,1) ; (4,1) ; (4,4) ; (1,4)] ;;

dessine_cellule [(3,2) ; (3,2)]

dessine_cellule cel ;;

dessine_ville (divise cel) ;;

graphvil ville_test ;;

floydwarshall (graphvil ville_test) ;;

longroute ville_test ;;

cout_1 ville_test ;;

indices_differents 10 ;;


(* Test de grille : *)

let l1 = [ (3,4) ; (4,5) ; (6,6) ; (7,5) ; (8,4) ; (8,3) ; (7,2) ; (6,2) ; (5,2)
; (4,2) ; (3,3) ] ;;
let l2 = [ (2,6) ; (4,7) ; (6,7) ; (9,6) ; (10,3) ; (9,2) ; (7,1) ; (6,0) ;
(5,0) ; (3,0) ; (2,2) ] ;;
```

```
let l_1 = [ (300,400) ; (400,500) ; (600,600) ; (700,500) ; (800,400) ;
(800,300) ; (700,200) ; (600,200) ; (500,200) ; (400,200) ; (300,300) ] ;;
let l_2 = [ (200,600) ; (400,700) ; (600,700) ; (900,600) ; (1000,300) ;
(900,200) ; (700,100) ; (600,0) ; (500,0) ; (300,0) ; (200,200) ] ;;



let test_vi = grille l1 l2 ;;

dessine_ville test_vi ;;

(* ------ *)

let depart = { ce = grille l1 l2 ; so = l1@l2 ; bord = l1@l2 } ;;

dessine_ville depart.ce ;;

dessine_ville (augmente depart 100).ce ;;

let depart2 = { ce = grille l_1 l_2 ; so = l_1@l_2 ; bord = l_1@l_2 } ;;

recuit_simule l1 l2 ;;

recuit_simule l_1 l_2 ;;

(* Fonction coupe, un cas problématique : *)

let d2 = Array.of_list depart2.ce ;;

let c4 = d2.(4) ;;

liste_bons c4 (barycentre c4) ;;

let c_4 = Array.of_list c4 ;;

let b4 = barycentre c4 ;;

let n_ = List.length c4 ;;

let screbe = c_4.(0) ;;

coupe c4 screbe b4 ;;

liste_bons c4 b4 ;;

(* --------- *)

(* cellules non convexes : *)

[(0,0) ; (1,1) ; (0,2) ; (2,2) ; (3,1) ; (2,1)]
[ (1,0) ; (3,3) ; (0,7) ; (4,10) ; (7,5) ; (11,5) ; (11,8); (14, 8) ; (14,0)]

(* cellules convexes : *)

[ (0,2) ; (0,5) ; (3,7) ; (5,5) ; (5,3) ; (5,1) ; (2,0)]
[ (0,0) ; (0,5); (2,2)]



(* REMARQUES AUTRES : *)

[(3, 5), (6, 7), (7, 2)];; (* les listes pythons sont des uplets en caml *)
```

```
(* UN PREMIER TEST AVEC DES POINTS DE NATHAN : *)


let avant1 =
[(277, 846);(279, 849);(284, 849);(287, 847);(290, 845);(293, 843);(295, 840);
(298, 838);(301, 836);(303, 833);(306, 831);(309, 829);(312, 827);(313, 823);
(311, 820);(309, 817);(305, 816);(302, 814);(297, 814);(293, 813);(289, 812);
(284, 812);(280, 811);(276, 810);(273, 808);(272, 804);(274, 801);(276, 798);
(279, 796);(281, 793);(283, 790);(284, 786);(285, 782);(286, 778);(287, 774);
(287, 769);(287, 764);(289, 761);(291, 758);(293, 755);(294, 751);(294, 746);
(292, 743);(290, 740);(288, 737);(288, 732);(288, 727);(290, 724);(292, 721);
(293, 717);(295, 714);(295, 709);(295, 706);(295, 701);(296, 697);(298, 694);
(300, 691);(304, 690);(307, 692);(310, 694);(313, 696);(315, 699);(316, 703);
(318, 706);(319, 710);(321, 713);(325, 712);(329, 713);(333, 714);(336, 716);
(340, 717);(344, 716);(347, 714);(348, 710);(347, 706);(348, 702);(350, 699);
(352, 696);(356, 695);(360, 696);(361, 700);(360, 704);(358, 707);(355, 709);
(353, 712);(352, 716);(353, 720);(355, 723);(357, 726);(359, 729);(361, 732);
(364, 734);(366, 737);(369, 739);(373, 740);(378, 740);(383, 740);(385, 737);
(388, 735);(391, 733);(394, 731);(397, 731);(400, 733);(401, 737);(401, 740);
(401, 745);(399, 748);(396, 750);(392, 751);(390, 754);(391, 758);(393, 761);
(395, 764);(399, 765);(402, 763);(404, 760);(406, 757);(408, 754);(411, 752);
(414, 750);(417, 748);(418, 744);(420, 741);(420, 736);(419, 732);(416, 730);
(413, 728);(410, 726);(406, 725);(404, 722);(403, 718);(402, 714);(401, 710);
(398, 708);(396, 705);(393, 703);(390, 701);(388, 698);(385, 696);(383, 693);
(381, 690);(383, 687);(385, 684);(387, 681);(390, 679);(394, 678);(398, 677);
(402, 676);(405, 674);(408, 672);(409, 668);(410, 664);(410, 659);(412, 656);
(415, 654);(419, 653);(423, 654);(428, 654);(433, 654);(436, 652);(438, 649);
(440, 646);(440, 641);(439, 637);(439, 632);(439, 627);(439, 622);(440, 618);
(440, 613);(440, 608);(440, 603);(439, 599);(437, 596);(434, 594);(430, 595);
(426, 596);(424, 599);(421, 601);(418, 601);(414, 600);(413, 596);(414, 592);
(414, 587);(411, 585);(406, 585);(402, 584);(400, 581);(399, 577);(400, 573);
(400, 568);(398, 565);(396, 562);(394, 559);(392, 556);(390, 553);(390, 548);
(392, 545);(394, 542);(397, 540);(401, 539);(406, 539);(410, 540);(414, 541);
(419, 541);(423, 542);(427, 543);(430, 545);(434, 546);(439, 546);(443, 545);
(447, 544);(451, 543);(454, 541);(457, 539);(460, 537);(463, 535);(466, 533);
(469, 531);(472, 529);(473, 525);(470, 523);(465, 523);(461, 522);(460, 518);
(459, 514);(459, 509);(461, 506);(462, 502);(459, 500);(456, 502);(452, 503);
(448, 504);(444, 505);(440, 506);(437, 508);(433, 507);(429, 506);(427, 503);
(426, 499);(424, 496);(422, 493);(418, 494);(414, 493);(411, 491);(409, 488);
(408, 484);(407, 480);(404, 478);(399, 478);(397, 481);(396, 485);(398, 488);
(399, 492);(401, 495);(402, 499);(403, 503);(401, 506);(398, 508);(394, 509);
(389, 509);(384, 509);(379, 509);(376, 507);(375, 503);(377, 500);(380, 498);
(383, 496);(384, 492);(386, 489);(387, 485);(388, 481);(390, 478);(392, 475);
(393, 471);(393, 466);(392, 462);(391, 458);(389, 455);(387, 452);(385, 449);
(381, 448);(378, 446);(375, 444);(371, 443);(367, 442);(363, 441);(360, 439);
(357, 437);(353, 436);(349, 435);(346, 433);(342, 432);(340, 429);(336, 428);
(333, 430);(331, 433);(329, 436);(327, 439);(325, 442);(324, 446);(323, 450);
(321, 453);(318, 455);(315, 457);(311, 456);(308, 454);(305, 452);(302, 450);
(299, 448);(295, 447);(293, 444);(291, 441);(288, 439);(286, 436);(284, 433);
(283, 429);(282, 425);(282, 420);(280, 417);(277, 415);(274, 413);(269, 413);
(264, 413);(261, 411);(258, 409);(255, 407);(252, 405);(248, 404);(243, 404);
(240, 402);(241, 398);(246, 398);(250, 399);(255, 399);(259, 398);(261, 395);
(263, 392);(264, 388);(265, 384);(266, 380);(267, 376)];;



let avant2 =
[(256, 318);(256, 313);(255, 309);(253, 306);(249, 305);(246, 307);(243, 309);
(241, 312);(240, 316);(239, 320);(240, 324);(241, 328);(243, 331);(246, 333);
(247, 337);(245, 340);(241, 341);(237, 342);(234, 344);(232, 347);(230, 350);
(227, 352);(223, 353);(220, 355);(217, 357);(216, 361);(217, 365);(219, 368);
(222, 370);(224, 373);(226, 376);(223, 378);(220, 378);(216, 377);(212, 376);
```

```
(207, 376);(202, 376);(200, 379);(199, 383);(197, 386);(194, 388);(190, 389);
(186, 390);(184, 393);(182, 396);(183, 400);(184, 404);(187, 406);(191, 407);
(195, 408);(199, 409);(202, 411);(205, 413);(207, 416);(208, 420);(208, 425);
(208, 430);(209, 434);(211, 437);(213, 440);(214, 444);(214, 447);(213, 451);
(213, 456);(212, 460);(211, 464);(210, 468);(209, 472);(209, 477);(208, 481);
(207, 485);(205, 488);(203, 491);(202, 495);(202, 498);(203, 502);(203, 507);
(201, 510);(198, 512);(194, 511);(192, 508);(192, 503);(192, 500);(191, 496);
(188, 494);(186, 491);(182, 490);(178, 489);(174, 490);(171, 492);(167, 493);
(164, 493);(162, 490);(161, 486);(158, 484);(155, 482);(150, 482);(146, 483);
(143, 485);(141, 488);(141, 493);(141, 498);(143, 501);(145, 504);(148, 506);
(151, 508);(155, 509);(159, 510);(163, 511);(165, 514);(167, 517);(169, 520);
(172, 522);(175, 524);(178, 526);(181, 528);(183, 531);(185, 534);(185, 539);
(186, 543);(186, 548);(187, 552);(189, 555);(190, 559);(190, 564);(191, 568);
(193, 571);(195, 574);(197, 577);(199, 580);(199, 585);(197, 588);(194, 590);
(191, 592);(188, 594);(187, 598);(185, 601);(183, 604);(182, 608);(181, 612);
(179, 615);(179, 620);(180, 624);(178, 625);(174, 624);(170, 625);(167, 627);
(166, 631);(166, 636);(167, 640);(168, 644);(169, 648);(171, 651);(172, 655);
(173, 659);(173, 664);(171, 667);(170, 671);(167, 673);(165, 676);(162, 678);
(159, 680);(158, 684);(158, 689);(158, 694);(159, 698);(159, 703);(158, 707);
(157, 711);(155, 714);(154, 718);(151, 720);(147, 721);(143, 720);(139, 721);
(135, 722);(134, 726);(134, 731);(136, 734);(140, 735);(144, 736);(149, 736);
(153, 737);(158, 737);(162, 736);(166, 737);(169, 739);(169, 744);(166, 746);
(162, 747);(158, 746);(154, 745);(150, 746);(148, 749);(146, 752);(145, 756);
(146, 760);(149, 762);(152, 764);(154, 767);(158, 768);(161, 770);(164, 772);
(168, 773);(172, 772);(175, 770);(179, 769);(182, 767);(186, 766);(189, 764);
(192, 762);(197, 762);(200, 764);(200, 769);(199, 773);(200, 777);(200, 782);
(199, 786);(197, 789);(195, 792);(192, 794);(189, 796);(185, 797);(181, 798);
(178, 798);(177, 794);(175, 791);(173, 788);(171, 785);(167, 784);(163, 785);
(159, 786);(157, 789);(154, 791);(153, 795);(152, 799);(149, 801);(145, 800);
(143, 797);(142, 793);(144, 790);(145, 786);(145, 781);(142, 779);(137, 779);
(133, 780);(129, 781);(125, 782);(121, 783);(118, 785);(114, 786);(110, 785);
(106, 786);(103, 788);(102, 792);(101, 796);(104, 798);(108, 799);(111, 797);
(115, 796);(119, 797);(122, 799);(126, 800);(130, 801);(133, 803);(134, 807);
(134, 812);(134, 817);(135, 821);(137, 824);(139, 827);(140, 831);(141, 835);
(143, 838);(147, 839);(152, 839);(156, 838);(159, 836);(162, 834);(166, 833);
(169, 831);(173, 830);(177, 831);(180, 833);(183, 835);(186, 837);(188, 840);
(191, 842);(194, 844);(198, 843);(201, 841);(202, 837);(204, 834);(204, 829);
(205, 825);(208, 823);(211, 821);(215, 820);(220, 820);(224, 821);(227, 823);
(229, 826);(231, 829);(233, 832);(235, 835);(239, 836);(244, 836);(248, 835);
(250, 832);(250, 827);(249, 823);(247, 820);(247, 815);(251, 814);(254, 816);
(256, 819);(256, 824);(258, 827);(261, 829);(263, 832);(266, 834);(269, 836);
(271, 839);(273, 842);(268, 372);(270, 369);(273, 367);(275, 364);(277, 361);
(277, 356);(278, 352);(279, 348);(280, 344);(280, 339);(279, 335);(277, 332);
(274, 330);(270, 329);(265, 329);(262, 327);(259, 325);(257, 322)] ;;


let liste_avant = avant1@avant2 ;;


let apres1 =
[(394, 703);(393, 700);(390, 699);(388, 697);(386, 695);(383, 694);(380, 693);
(379, 690);(378, 687);(380, 685);(381, 682);(383, 680);(386, 679);(388, 677);
(392, 677);(395, 676);(398, 675);(401, 674);(403, 672);(405, 670);(406, 667);
(406, 663);(407, 660);(408, 657);(410, 655);(412, 653);(415, 652);(418, 651);
(421, 650);(425, 650);(428, 649);(431, 648);(434, 647);(436, 645);(437, 642);
(438, 639);(438, 635);(437, 632);(436, 629);(435, 626);(435, 622);(436, 619);
(437, 616);(438, 613);(438, 609);(439, 606);(440, 603);(439, 600);(438, 597);
(436, 595);(434, 593);(430, 593);(427, 594);(425, 596);(422, 597);(420, 599);
(416, 599);(414, 597);(413, 594);(414, 591);(415, 588);(415, 584);(413, 582);
(411, 580);(407, 580);(404, 579);(401, 578);(399, 576);(398, 573);(398, 569);
(397, 566);(396, 563);(395, 560);(393, 558);(391, 556);(390, 553);(389, 550);
(388, 547);(389, 544);(390, 541);(391, 538);(393, 536);(396, 535);(400, 535);
(403, 536);(407, 536);(410, 537);(413, 538);(416, 539);(419, 540);(421, 542);
(423, 544);(425, 546);(428, 547);(432, 547);(435, 546);(438, 545);(441, 544);
```

```
(444, 543);(447, 542);(450, 541);(453, 540);(456, 539);(459, 538);(461, 536);
(463, 534);(465, 532);(467, 530);(469, 528);(470, 525);(470, 521);(468, 519);
(466, 517);(462, 517);(460, 515);(458, 513);(457, 510);(456, 507);(455, 504);
(453, 502);(451, 500);(447, 500);(444, 501);(441, 502);(439, 504);(436, 505);
(434, 507);(431, 508);(427, 508);(424, 507);(420, 507);(416, 507);(413, 508);
(410, 509);(408, 511);(405, 512);(402, 513);(399, 514);(396, 515);(393, 516);
(390, 515);(386, 515);(383, 514);(380, 513);(376, 513);(372, 513);(369, 512);
(367, 510);(366, 507);(365, 504);(366, 501);(367, 498);(369, 496);(372, 495);
(374, 493);(376, 491);(378, 489);(380, 487);(382, 485);(383, 482);(385, 480);
(386, 477);(387, 474);(388, 471);(390, 469);(390, 465);(391, 462);(390, 459);
(389, 456);(388, 453);(387, 450);(386, 447);(384, 445);(381, 444);(378, 443);
(375, 442);(372, 441);(368, 441);(365, 440);(362, 439);(360, 437);(357, 436);
(354, 435);(352, 433);(349, 432);(346, 431);(343, 430);(341, 428);(339, 426);
(336, 425);(333, 426);(331, 428);(329, 430);(327, 432);(325, 434);(324, 437);
(322, 439);(321, 442);(320, 445);(319, 448);(318, 451);(316, 453);(314, 455);
(311, 456);(309, 454);(306, 453);(304, 451);(302, 449);(300, 447);(298, 445);
(295, 444);(293, 442);(291, 440);(288, 439);(284, 439);(282, 437);(280, 435);
(280, 431);(280, 427);(280, 423);(280, 419);(279, 416);(277, 414);(275, 412);
(272, 411);(269, 410);(266, 411);(263, 410);(259, 410);(257, 408);(255, 406);
(252, 405);(249, 404);(246, 403);(242, 403);(238, 403);(236, 401);(235, 398);
(235, 394);(237, 392);(240, 391);(243, 392);(246, 393);(249, 394);(253, 394);
(256, 393);(259, 392);(260, 389);(261, 386);(262, 383);(262, 379);(263, 376);
(264, 373);(265, 370);(266, 367);(267, 364);(269, 362);(271, 360);(272, 357);
(273, 354);(274, 351);(274, 347);(275, 344);(275, 340);(275, 336);(274, 333);
(272, 331);(269, 330);(266, 329);(263, 328);(259, 328);(256, 327);(253, 326);
(252, 323);(252, 319);(251, 316);(251, 312);(250, 309);(248, 307);(244, 307);
(242, 309);(240, 311);(239, 314);(238, 317);(239, 320);(240, 323);(242, 325);
(245, 326);(247, 328);(247, 332);(247, 336);(245, 338);(242, 339);(239, 340);
(236, 341);(234, 343);(231, 344);(230, 347);(228, 349);(226, 351);(223, 352);
(220, 353);(217, 354);(215, 356);(214, 359);(214, 363);(216, 365);(218, 367);
(220, 369);(222, 371);(221, 374);(219, 376);(216, 375);(213, 374);(210, 373);
(206, 373);(203, 374);(199, 374);(197, 376);(194, 377);(191, 378);(188, 379);
(185, 380);(182, 381);(180, 383);(179, 386);(177, 388);(177, 392);(178, 395);
(179, 398);(181, 400);(184, 401);(186, 403);(189, 404);(191, 406);(194, 407);
(197, 408);(200, 409);(203, 410);(205, 412);(206, 415)] ;;


let apres2 =
[(207, 418);(206, 421);(206, 425);(207, 428);(208, 431);(209, 434);(211, 436);
(212, 439);(213, 442);(212, 445);(211, 448);(211, 452);(210, 455);(209, 458);
(208, 461);(208, 465);(207, 468);(207, 472);(207, 476);(207, 480);(206, 483);
(204, 485);(203, 488);(202, 491);(201, 494);(202, 497);(203, 500);(203, 504);
(202, 507);(200, 509);(198, 511);(196, 513);(192, 513);(190, 511);(189, 508);
(188, 505);(188, 501);(189, 498);(188, 495);(187, 492);(185, 490);(182, 489);
(180, 487);(176, 487);(172, 487);(170, 489);(167, 490);(166, 493);(165, 496);
(163, 498);(160, 499);(159, 496);(159, 492);(159, 488);(157, 486);(155, 484);
(153, 482);(150, 481);(146, 481);(143, 482);(142, 485);(141, 488);(140, 491);
(141, 494);(141, 498);(143, 500);(145, 502);(148, 503);(150, 505);(153, 506);
(156, 507);(160, 507);(161, 510);(163, 512);(166, 513);(167, 516);(169, 518);
(172, 519);(174, 521);(177, 522);(179, 524);(182, 525);(184, 527);(185, 530);
(185, 534);(185, 538);(185, 542);(186, 545);(186, 549);(187, 552);(188, 555);
(188, 559);(189, 562);(189, 566);(191, 568);(193, 570);(195, 572);(197, 574);
(199, 576);(199, 580);(198, 583);(196, 585);(194, 587);(191, 588);(189, 590);
(187, 592);(186, 595);(184, 597);(183, 600);(182, 603);(180, 605);(179, 608);
(177, 610);(176, 613);(176, 617);(177, 620);(178, 623);(176, 625);(172, 625);
(170, 625);(167, 626);(165, 628);(163, 630);(162, 633);(162, 637);(163, 640);
(164, 643);(165, 646);(166, 649);(167, 652);(167, 656);(167, 660);(167, 664);
(166, 667);(165, 670);(163, 672);(161, 674);(159, 676);(157, 678);(156, 681);
(156, 685);(156, 689);(157, 692);(157, 696);(157, 700);(156, 703);(155, 706);
(154, 709);(153, 712);(152, 715);(150, 717);(148, 719);(144, 719);(141, 718);
(137, 718);(134, 719);(132, 721);(131, 724);(132, 727);(133, 730);(135, 732);
(139, 732);(143, 732);(146, 733);(149, 734);(152, 735);(155, 736);(159, 736);
(163, 736);(165, 738);(167, 740);(166, 743);(164, 745);(161, 746);(158, 747);
(155, 748);(153, 750);(151, 752);(151, 756);(152, 759);(153, 762);(155, 764);
```

```
(158, 765);(159, 768);(161, 770);(162, 773);(164, 775);(165, 778);(167, 780);
(171, 780);(174, 779);(176, 777);(178, 775);(180, 773);(181, 770);(183, 768);
(184, 765);(186, 763);(187, 760);(189, 758);(191, 756);(194, 755);(198, 755);
(201, 754);(204, 753);(206, 751);(207, 748);(207, 744);(208, 741);(206, 739);
(204, 737);(203, 734);(201, 732);(200, 729);(201, 726);(204, 725);(207, 724);
(211, 724);(213, 726);(214, 729);(213, 732);(213, 736);(214, 739);(216, 741);
(218, 743);(220, 745);(222, 747);(224, 749);(225, 752);(224, 755);(222, 757);
(220, 759);(219, 762);(218, 765);(219, 768);(219, 772);(220, 775);(219, 778);
(218, 781);(217, 784);(215, 786);(212, 787);(208, 787);(205, 786);(202, 787);
(199, 788);(198, 791);(196, 793);(194, 795);(192, 797);(190, 799);(188, 801);
(186, 803);(184, 805);(182, 807);(180, 809);(179, 812);(177, 814);(176, 817);
(175, 820);(176, 823);(177, 826);(178, 829);(180, 831);(182, 833);(184, 835);
(187, 836);(191, 836);(193, 834);(195, 832);(196, 829);(196, 825);(197, 822);
(198, 819);(200, 817);(202, 815);(205, 814);(209, 814);(213, 814);(217, 814);
(220, 813);(223, 812);(225, 810);(227, 808);(229, 806);(231, 804);(234, 803);
(238, 803);(242, 803);(245, 804);(248, 805);(250, 807);(253, 808);(255, 810);
(257, 812);(259, 814);(260, 817);(260, 821);(260, 825);(261, 828);(263, 830);
(264, 833);(266, 835);(267, 838);(268, 841);(270, 843);(271, 846);(273, 848);
(274, 851);(276, 853);(277, 856);(279, 858);(283, 858);(286, 857);(288, 855);
(290, 853);(292, 851);(294, 849);(294, 845);(295, 842);(296, 839);(297, 836);
(399, 758);(395, 758);(392, 757);(391, 754);(391, 750);(393, 748);(394, 745);
(396, 743);(398, 741);(399, 738);(400, 735);(401, 732);(400, 729);(400, 725);
(400, 721);(399, 718);(399, 714);(398, 711);(397, 708);(396, 705)] ;;

let liste_apres = apres1@apres2 ;;


(*true*)
intersection (1,1) (4,4) (1,4) (4,1) ;;
intersection (4,4) (1,1) (4,1) (1,4) ;;
intersection (0,1) (4,1) (1,0) (1,4) ;;
intersection (1,0) (1,4) (0,1) (4,1) ;;
intersection (3,5) (4,2) (1,1) (5,5) ;;


(*false *)
intersection (2,1) (2,5) (3,1) (3,3) ;;



let commencement = [ avant1@apres2 ; avant2@apres1] ;;

dessine_ville (grille liste_avant liste_apres) ;;

dessine_ville commencement ;;



let undepar = { ce = [liste_avant] ; so = liste_avant ; bord = liste_avant } ;;


let villeTest n =
    let rec multiplie l = match l with
        | [] -> []
        | (x,y)::q -> ( (n*x + 100), (n*y) )::(multiplie q)
     in let rec aux v = match v with
        | [] -> []
        | l::q -> (List.rev (multiplie l))::(aux q)
    in let cellules = aux ville_test

    in let rec fusion ll = match ll with (* liste de liste vers liste *)
        | [] -> []
        | l::q -> l@(fusion q)
    in let sommets = fusion cellules (* /!\ tous les sommets seront en double *)
```

```
    in let sommets_bords = multiplie [(1, 1); (1, 3); (3, 1); (6, 1); (10, 1);
(10, 8); (1, 6); (8, 10); (8, 10); (1, 10)]
    in { ce = cellules ; so = sommets ; bord = sommets_bords } ;;


recuit2 (villeTest 60) ;;
```