CHAPITRE XIII

MANIPULATION D'UNE SOURIS

Etant donné un événement (c'est un type prédéfini) e, il est possible d'accéder à :

e.keypressed	indique si oui ou non une touche a été frappée.
e.key	donne le code ascii de cette touche.
e.button	indique si oui ou non la souris a été cliquée.
e.mouse_x	
e.mouse_y	sont les coordonnées de la position de la souris.
mouse_pos () ;;	renvoie le couple (x, y)
button_down () ;;	renvoie un booléen.

1. Courbes de Béziers

Etant donnés des points de contrôles ou pôles P_0, P_1, ... P_n, l'idée est de construire une courbe qui soit "attirée" par ces pôles, sans nécessairement y passer. (Le problème de trouver la courbe passant vraiment par des points est résolu avec les polynôme de Lagrange) L'application évidente étant la possibilité en CAO de créér des formes à partir de points que l'utilisateur pourra déplacer à son gré dans un but technique ou artistique.
Ce problème a reçu une solution sous la forme de courbe de Bezier associée aux points P_0, P_1 .... P_n : c'est la courbe définie grâce au paramètre t par :

$$\overrightarrow{OM}(t) = \sum_{k=0}^{n} B_{k,n}(t).\overrightarrow{OP_k} \quad \text{où } B_{k,n}(t)= C_n^k t^k (1-t)^{n-k} \text{ sont les polynômes de Bernstein}$$

On vérifie que la somme des poids vaut 1, que cette courbe passe en M_0 et en M_n, elle est tangente en M_0 à M_0M_1, et en M_n à $M_{n-1}M_n$, par ailleurs, si n = 2, M_1 a une tangente parallèle à M_0M_2.
Construction récurrente, par un jeu de factorisations, on peut montrer que :

$$\overrightarrow{OM}(t) = B_{0,n-1}(t)[(1-t)\overrightarrow{OP_0}+ t\overrightarrow{OP_1}]+B_{1,n-1}(t)[(1-t)\overrightarrow{OP_1}+ t\overrightarrow{OP_2}] +...+ B_{n-1,n-1}(t)[(1-t)\overrightarrow{OP_{n-1}}+ t\overrightarrow{OP_n}]$$

On en déduit le résultat suivant que le point M_t relatif aux n+1 points de départ, est le même que celui relatif à la courbe de Bezier déterminée par les n points P' tels que:

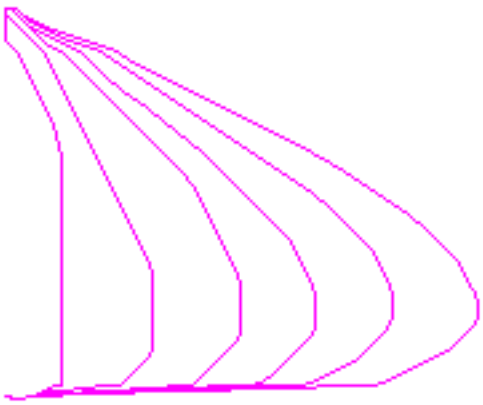
$$\overrightarrow{OP'_k}(t) = (1-t)\overrightarrow{OP_k} + t\overrightarrow{OP_{k+1}}$$

En réitérant cette propriété, on arrive à déduire M_t comme relatif à deux seuls points, puis pour finir, à un seul, c'est l'algorithme de tracé de De Casteljau ci-après.

Entrée des points P_0, ... , P_n
Pour t allant de 0 à 1 de h en h
Pour i allant de 0 à n faire $M_i \leftarrow P_i$ {recopie des points initiaux}
Pour i allant de 1 à n Pour j allant de 0 à n-i faire $M_j \leftarrow (1-t)M_j + tM_{j+1}$
Tracer M_0

Ainsi, en débutant, M_0, ... ,M_n, sont remplacés par d'autres points toujours apelés M_0, ..., M_{n-1}, pour i = 1, puis pour i = 2, ils le sont à leur tour par M_0, ... ,M_{n-2}. Ainsi arrivé à i = n les deux points M_0, M_1 sont remplacés par un M_0 unique qui est tracé. Cet algorithme est de complexité analogue, mais il permet d'épargner le calcul des polynômes de Bernstein.

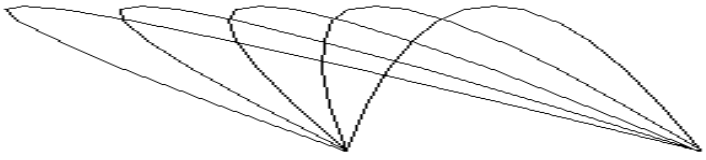
```
#open "graphics";;

let cadre () = set_color blue; fill_rect 0 0 480 280;  set_color white;
        fill_rect 4 4 471 271; set_line_width 1;;

let round x = if x >=. 0. then int_of_float(x +. 0.5) else -(int_of_float (-. x +. 0.5));;  (* arrondi *)

let bary u (a, b) v (c, d) =
        (round (u*.(float_of_int a) +. v*.(float_of_int c)),
        round(u*.(float_of_int b) +. v*.(float_of_int d))) ;;

let trace p c = set_color c; moveto (fst p.(0)) (snd p.(0));
        (* p est un vecteur de couples d'entiers, ce sont les pôles *)
        let t = ref 0. and n = vect_length p in
        while !t <. 1. do
                t := !t +. 0.001; let m = p in
                for i = 1 to n - 1 do for j = 0 to n - i - 1 do
                        vect_assign m j (bary (1.-. !t) m.(j) !t m.(j + 1)) done
        done;
        lineto (fst m.(0)) (snd m.(0)) done ;;
```



Comme démonstration, on fixe un point de départ et un point d'arrivée en faisant varier le point intermédiaire (une fois n'est pas coutume, on définit une constante "demo" :

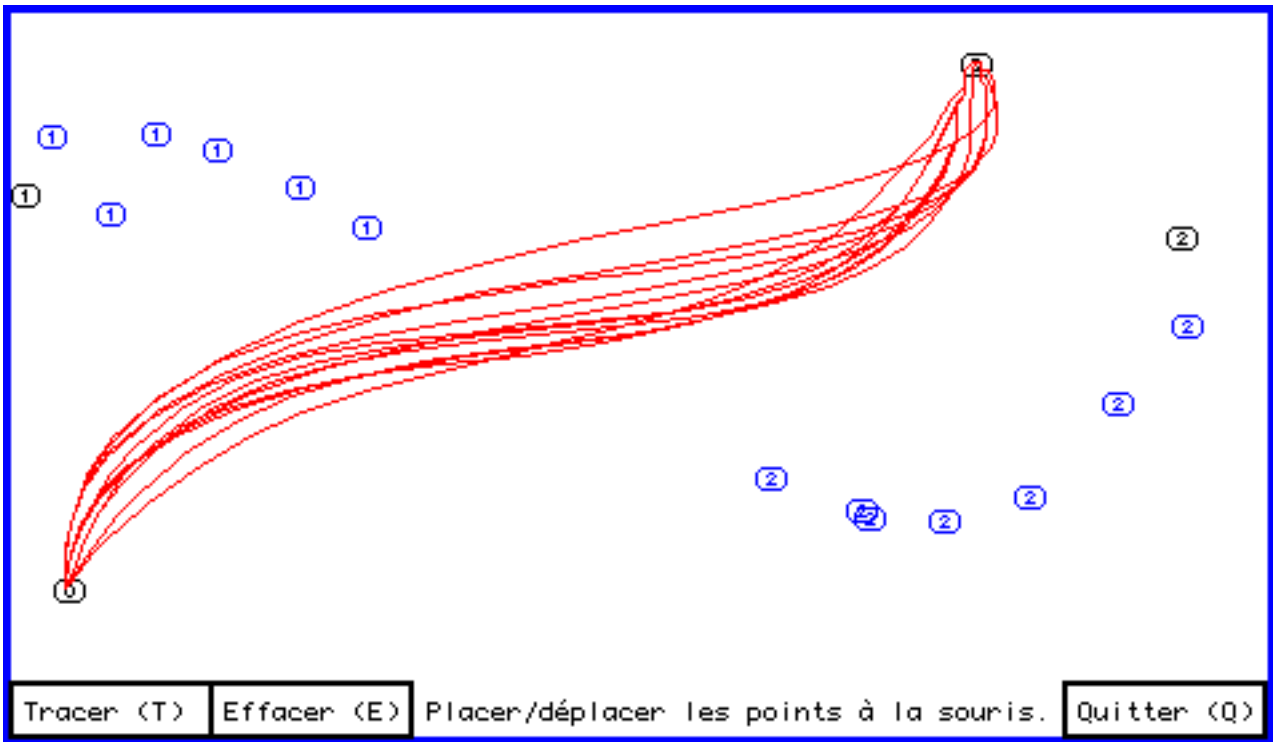
```
let demo = open_graph"";
        let poles = make_vect 3 (5, 5) in
        for i = 1 to 6 do
                poles.(0)<- (5,150);
                poles.(1) <- (60 * i, 0);
                trace poles magenta
        done;
        read_key(); close_graph();;
```

Ou bien, le point intermédiaire prenant les cinq positions successives ci-dessous :



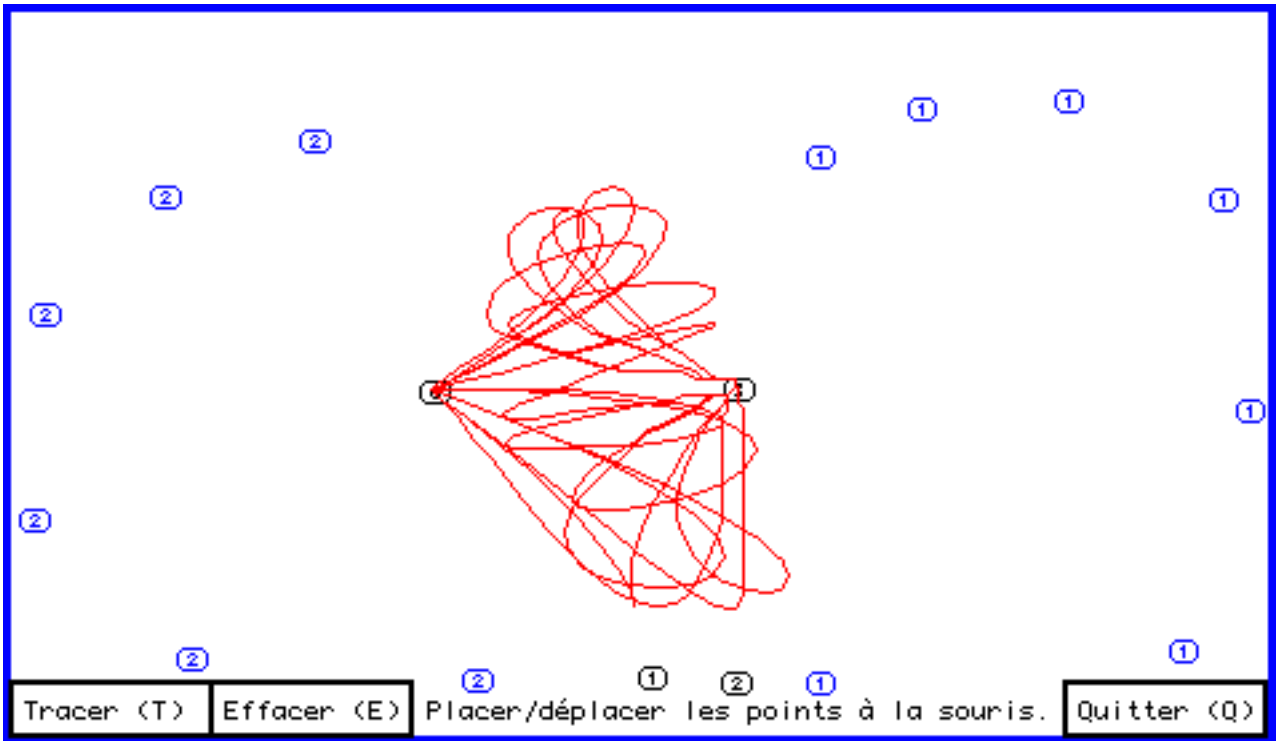
La fonction principale "trace" fait quelques lignes, l'interface pour l'utilisateur "beziers" en fait cinq fois plus. Cette interface consiste à tracer sur l'écran un cadre et des boîtes contenant des messages divers. Boîtes qui serviront à cliquer, afin de tracer la courbe, de l'effacer pour une autre expérimentation, ou de quitter l'application. Mais avant cela, l'utilisateur devra cliquer dans la fenêtre pour placer ses points de contrôles. Le rang du point courant est repéré par la variable p (12 points au maximum) et nx, ny sont ses coordonnées. La fonction "beziers" qui remplit ce rôle d'interface, ne renvoie rien et réalise une boucle infinie analysant les touches pressées ou le bouton de la souris. On ne sort de cette boucle infinie que par une exception.
Cette méthode sera reprise, sans grands changements dans quelques application suivante?

```
exception Exit;;

let quitter x y =  400 < x & y < 25

and tracer x y = x < 80  & y < 25

and effacer x y = 80 < x & x < 160 & y < 25;;

let boite x y l h c = set_color black; fill_rect x y l h; set_color white;
        fill_rect (x + 2) (y + 2) (l - 4) (h - 4) ;
        moveto (x + 6) (y + 6);
        set_color black; draw_string c;;

let maxi = 12;;

let place k u v c = moveto (u - 5) (v -  4); set_color c; draw_string ("("^(string_of_int k)^")");;

let deja x y tp n = if n < 1 then false else deja' x y (list_of_vect (sub_vect tp 0 n))
        where rec deja' x y = fun [] -> false
                        | ((u, v) :: ll) -> if abs (x - u) + abs(y - v) < 12 then true else deja' x y ll;;

let cherche x y V = let k = ref 0 in
        while abs ((fst V.(!k)) - x) + abs((snd V.(!k)) - y) >= 12 do incr k done;
        (!k, (fst V.(!k)), (snd V.(!k)));;

let beziers () = open_graph""; cadre();
        boite 3 3 77 22 "Tracer (T)";
        boite 78 3 77 22 "Effacer (E)";
        boite 398 3 77 22 "Quitter (Q)";
        moveto 160 9; draw_string "Placer/déplacer les points à la souris.";
        let p = ref 0 and P = ref (make_vect maxi (0, 0))  (* !p = nombre de points de contrôle *)
        in try while true do
                set_text_size 9; set_font "zeal"; (* pour avoir des chiffres entourés *)
                let e = wait_next_event [Button_down; Key_pressed] in
                if e.keypressed then match e.key with
                        | `t`|`T`-> trace (sub_vect !P 0 !p) red
                        | `q`|`Q`-> raise Exit
                        | `e`|`E`-> trace (sub_vect  !P 0 !p) white
                        | _ ->()
                else    if e.button then let (x, y) = mouse_pos() in
                        if effacer x y then  trace (sub_vect  !P 0 !p) white
                        else if tracer x y then trace  (sub_vect !P 0 !p ) red
                        else if quitter x y then raise Exit
                        else if deja x y !P !p then (let (k, u, v) = cherche x y !P in place k u v blue;
                                                while button_down() do   done;
                                                let nx, ny = mouse_pos() in !P.(k) <- (nx, ny);
                                                place k nx ny black)
                        else  if !p < maxi then (!P.(!p) <- (x, y); place !p x y black; incr p)
        done
with Exit -> close_graph();;
```



Ici, on place 4 points dont les points 0 et 3 sont fixes et on demande les tracés sans effacement des précédents, en faisant varier les deux points intermédiaires 1 et 2.

Ci-dessous 4 points dont les deux intermédiaires sont déplacés de façon à attirer la courbe et la faire boucler.



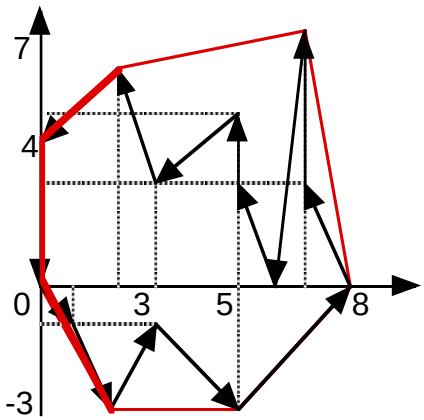
2. Enveloppe convexe d'un ensemble de points du plan

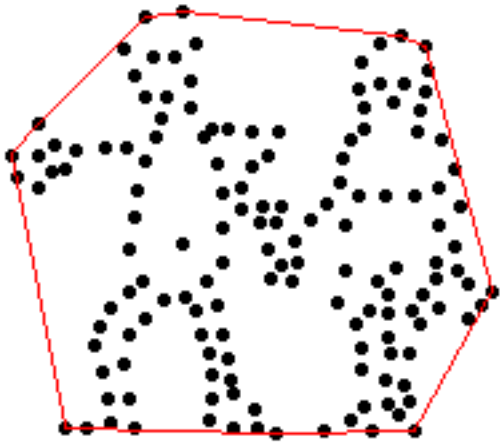
On reprend la structure de cycle ou liste circulaire (doublement chaînée) vue au chapitre 9. Un cycle est définit par une "cellule", le "même cycle" peut être abordé par la cellule voisine, c'est le rôle des fonctions avance et recule.
Fonctions de suppression de l'élément courant et d'insertion d'un objet x dans un cycle, fonctions de conversion d'un cycle en liste et réciproque.

```
type 'a cycle = Vide | Cel of 'a cellule
and 'a cellule = {val : 'a; mutable avant : 'a cycle; mutable apres : 'a cycle};;

let valeur = fun Vide -> failwith "cycle vide" | (Cel c) -> c.val;;

let avance = fun Vide -> Vide | (Cel c) -> c.avant;;

let recule = fun Vide -> Vide | (Cel c) -> c.apres;;

let changeavant nouv = fun Vide -> failwith "cycle vide" | (Cel c) -> (c.avant <- nouv);;

let changeapres nouv = fun Vide -> failwith "cycle vide" | (Cel c) -> (c.apres <- nouv);;

let seul = fun Vide -> false | c -> (c= avance c);;

let supprime = fun Vide -> failwith "cycle vide " | c -> if seul c then Vide
        else  (changeavant (avance c) (recule c); changeapres (recule c) (avance c); avance c);;

let insavant x cycle = let c = Cel {val = x; avant = avance cycle; apres = cycle} in
        if cycle = Vide then (changeavant c c; changeapres c c; c)
        else (changeavant c cycle; changeapres c (avance cycle); c) ;;

let insapres x cycle = let c = Cel {val = x; avant = cycle; apres = recule cycle} in
        if cycle = Vide then (changeavant c c; changeapres c c; c)
        else (changeavant c (recule cycle); changeapres c cycle; c) ;;

let taille = fun Vide -> 0 | cycle -> let rec compte n c = if c = cycle then n
        else compte (n+1) (avance c) in compte 1 (avance cycle);;

let cycletolist = fun Vide -> []
                | c -> let rec aux cycle res = if cycle = c then (valeur c) :: res
                                        else aux (recule cycle) ((valeur cycle) :: res)
                        in aux (recule c) [];;

let rec listocycle = fun [] -> Vide | (a :: q) -> insapres a  (listocycle q);;

let rec creer n = if n = 0 then Vide else insapres n (creer (n-1));;
```

```
cycletolist (listocycle [1;2;3;4;5]);; ☞ [1; 2; 3; 4; 5]
cycletolist (creer 7);; ☞ [7; 6; 5; 4; 3; 2; 1]
cycletolist(avance (creer 7));; ☞ [6; 5; 4; 3; 2; 1; 7]
cycletolist(recule (creer 7));; ☞ [1; 7; 6; 5; 4; 3; 2]
cycletolist (supprime(creer 7));; ☞ [6; 5; 4; 3; 2; 1]
```

Une partie convexe du plan est, par définition, une partie telle qu'elle doit contenir tous les segments dont les extrémités lui appartiennent.
L'enveloppe convexe de n points est le plus petit convexe contenant ces points et c'est en même temps l'ensemble des barycentres à coefficients positifs de ces n points. Pour n points, elle est entièrement déterminée comme l'intérieur d'un polygone formé par certains (les points extrèmaux) de ces points.





L'algorithme de Graham-Andrew consiste à parcourir le "cycle" des points jusqu'à ce qu'il n'y ait plus d'angle rentrant : chaque fois qu'on regarde trois points p, q, r dans le sens positif, si l'angle qu'ils forment est saillant, on avance, s'il est rentrant ou plat, on supprime q du cycle et on recule (car p peut à son tour être supprimé).

On fait un tour complet tant qu'il y a eu au moins une suppression. En supposant le cycle déjà formé, l'angle déterminé par les points pqr (dans le sens positif, sera dit rentrant, si le déterminant des vecteurs qp et qr est positif ou nul (points alignés), saillant sinon.

```
let det (x, y) (x', y') = x*y' - x'*y;;

let rentrant p q r = 0 <= det ((fst p) -  (fst q), (snd p) - (snd q)) ((fst r )-  (fst q), (snd r) - (snd q)) ;;
```

```
rentrant(0,2) (1,3) (2,0);; -☞ true
rentrant(0,2) (1,1) (2,0);; ☞ true
rentrant(0,2) (1,0) (2,0);;  ☞ false
```

```
let rendconvexe c =
        let f = valeur c in
        let rec aux m drap tour = let p = valeur m and q = valeur (avance m)
                and r = valeur (avance (avance m)) in
        if rentrant p q r then aux (recule (supprime (avance m))) true tour
        else if p = f & tour & not (drap) then c
        else aux (avance m) (if p = f then false else drap) true
        in aux  c false false;;
```

```
cycletolist (rendconvexe (listocycle [(8,0); (7, 3); (7, 7); (5, 2); (5, 5); (3, 3); (2, 6); (0, 4);
(0, 0); (1, -1); (2, -3); (3, -1); (5, -3); (6, 0)]));;
☞ [8, 0; 7, 7; 2, 6; 0, 4; 0, 0; 2, -3; 5, -3]
```

La difficulté est de former le cycle, une première solution consiste à prendre une origine intérieure (sur un segment joignant deux points) et à classer les points par angles polaires croissants.

Une seconde solution consiste à prendre deux points extrèmes P et P' (d'abscisses min et max), recalculer les abscisses suivant l'axe PP' et trier les points d'ordonnées positives par abscisses décroissantes, suivis des points d'abscisses négatives triés par abscisses croissantes.

La solution adoptée ci-dessous ne recalcule pas les abscisses mais sépare les points suivant les demi-plans définis par PP' (droite ax + by + c = 0 dans "sep"). Fonctions de tri par fusion :

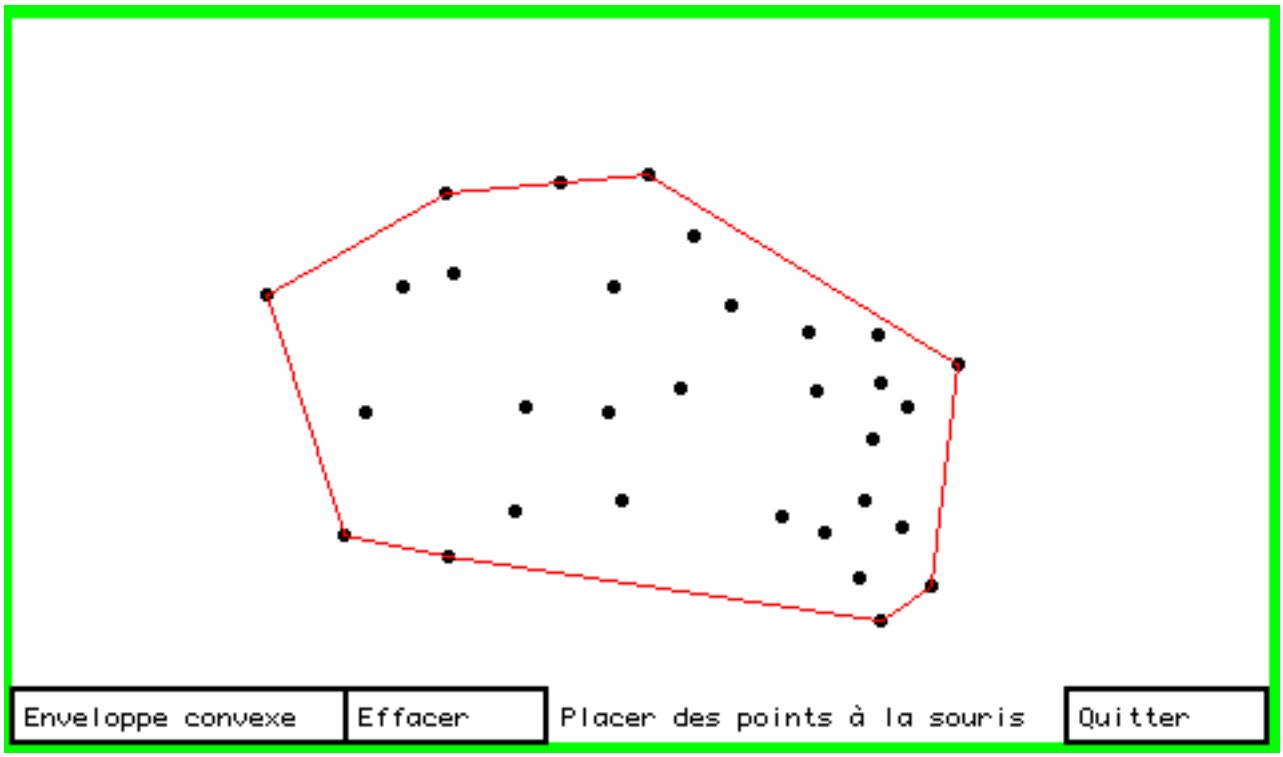
```
let rec separ l1 l2 = fun [] -> (l1, l2) | (a :: r) -> separ (a :: l2) l1 r;;
```

```
let rec fusion ord = fun
        | [] q -> q | q [] -> q
        | (x :: q) (y :: r) -> if ord x y then x :: (fusion ord q (y :: r))
                               else y :: (fusion ord (x :: q) r);;

let rec trifus ord = fun   | [] -> []
                           | [x] -> [x]
                           | q -> let (l1, l2) = separ [] [] q in fusion ord (trifus ord l1) (trifus ord l2);;

let rec sep a b c lp ln = fun [] -> lp, ln
        | ((x, y)::q) -> if 0< a * x + b * y + c then sep a b c  lp ((x, y) :: ln) q
                         else sep a b c ((x, y) :: lp) ln q;;

let prepare l = let lt = trifus (fun p q -> fst p > fst q) l
        in let (a, b) = hd lt and  (a', b') = hd (rev lt)
        in let lp, ln = sep (b - b') (a' - a) (a * b' - a' * b) [] [] lt
        in listocycle ((rev lp) @ ln);;
```

```
cycletolist (rendconvexe (prepare [(7, 7); (1, -1); (2, -3); (3, -1); (5, -3); (5, 5); (3, 3); (2, 6);
(0, 4);(8,0); (7, 3); (5, 2);  (0, 0); (6, 0)]));;
☞ [8, 0; 7, 7; 2, 6; 0, 4; 0, 0; 2, -3; 5, -3]
```

Petite interface pour placer les points à la souris :

```
#open "graphics";;

exception Exit;;

let cadre () = set_color green; fill_rect 0 0 480 280;
        set_color white; fill_rect 4 4 471 271; set_line_width 1;;

let quitter x y =  400 < x & y < 25

and tracer x y = x < 130  & y < 25

and effacer x y = 128 < x & x < 205 & y < 25;;

let boite x y l h c = set_color black; fill_rect x y l h; set_color white;
        fill_rect (x + 2) (y + 2) (l - 4) (h - 4) ;
        moveto (x + 6) (y + 6); set_color black; draw_string c;;

let colorier c = let rec trait cycle =  let (x, y) = valeur cycle in
        lineto x y; if c <> cycle then trait (avance cycle) in
        let (a, b) = valeur c in moveto a b; trait (avance c); c;;

let rec convexe () = open_graph""; cadre(); boite 3 3 127 22 "Enveloppe convexe";
        boite 128 3 77 22 "Effacer ";
        boite 398 3 77 22 "Quitter "; moveto 210 9;
        draw_string "Placer des points à la souris";
        let lp = ref [] in (* lp = liste des points*)
try while true do set_text_size 9;
        let e = wait_next_event [Button_down] in
        if e.button then let (x, y) = mouse_pos() in
                if effacer x y then convexe()
                else if tracer x y then (set_color  red;
                                lp:= cycletolist (colorier (rendconvexe (prepare !lp))) )
                else if quitter x y then raise Exit
                else (lp := (x, y) :: (!lp); set_color  black; fill_ellipse x y 2 2)
        done
with Exit -> close_graph();;
```


Enveloppe convexe
Effacer
Placer des points à la souris
Quitter

CHAPITRE XIV

EXEMPLES DE SYSTEMES MULTI-AGENTS

1. Evolution d'une opinion dans une population

On présente un modèle très simple où pour n agents notés a_i, relativement à un critère comme le goût pour telle activité, produit commercial ou opinion diverse, cet agent possède une opinion x_i dans l'intervalle [-1, 1]. Cette opinion est en fait un segment $s_i = [x_i - u_i, x_i + u_i]$ réalisant une fourchette où le coefficient u_i est un degré d'incertitude dans l'intervalle [0, 1], 0 signifie que a_i est sur de lui et $u_i = 1$ ne peut se produire que pour $x_i = 0$ c'est l'incertitude maximale. On définit le chevauchement h_{ij} entre deux agents comme la longueur de l'intersection de leurs segments.
$h_{ij} = \max(0, \min(x_i + u_i, x_j + u_j) - \max(x_i - u_i, x_j - u_j))$ est une fonction symétrique.

L'influence relative de a_i sur a_j est définie comme $f_{ij} = h_{ij} / u_i - 1$, c'est donc un coefficient non symétrique égal au minimum à -1 au cas où les segments sont disjoints, et au maximum à 1 au cas où le segment de a_i est inclus dans celui de a_j.
Règle de transition d'influence de a_i sur a_j : on pose μ entre 0.2 et 1.5 un coefficient positif destiné à pondérer cette influence.
On dit que si $f_{ij} > 0$ cette influence sera active, c'est à dire si $u_i < h_{ij}$. En ce cas l'agent a_j sera modifié de la façon suivante :

$x_j \leftarrow x_j + \mu f_{ij}(x_i - x_j)$ ce qui signifie que a_i déplace a_j vers lui (vers 1 ou vers -1)

$u_j \leftarrow u_j + \mu f_{ij}(u_i - u_j)$ ce qui signifie que si a_i est plus certain que a_j ($u_i < u_j$)
alors il lui réduit son incertitude.

Supposons que ce soit le cas, alors $f_{ji} < f_{ij}$ car $u_i < u_j$, si f_{ji} est encore positif, à son tour a_j influence a_i avec les mêmes règles mais cette modification est moins forte.
Si ces modifications sont faites de manière synchrone (en fait, on fait d'abord agir celui qui a la plus forte influence), au bout d'un certain nombre d'itérations, on souhaite regarder la trajectoire de n individus aléatoirement répartis au début entre -1 et 1.

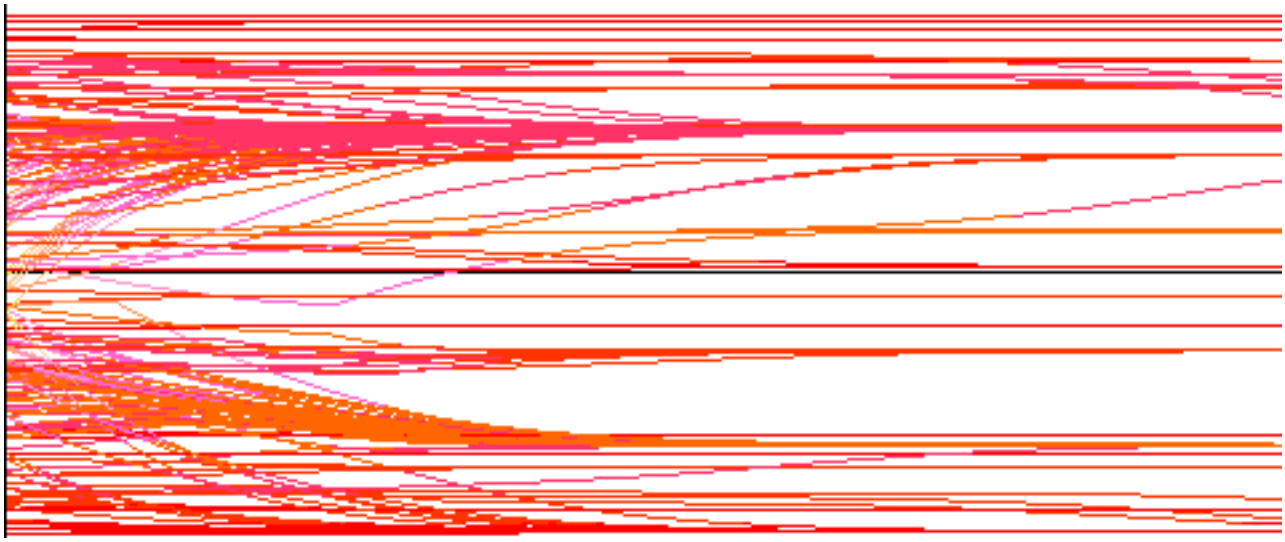
Leur incertitude sera aussi aléatoire, c'est ce que l'on fait en initialisant les vecteurs de position "pos" et d'incertitude "unc".

```
#open "graphics";; let np = 200;;
let pos = make_vect np 0. and unc = make_vect np 0. and pos' = make_vect np 0.
and unc' = make_vect np 0.;;

let round x = if x >=. 0. then int_of_float(x +. 0.5) else -(int_of_float (-. x +. 0.5));;  (* arrondi *)

let influence i j = max 0. (min (pos.(i) +. unc.(i)) (pos.(j) +. unc.(j))
                            -. (max (pos.(i) -. unc.(i)) (pos.(j) -. unc.(j)))) /. unc.(i) -. 1.;;
```

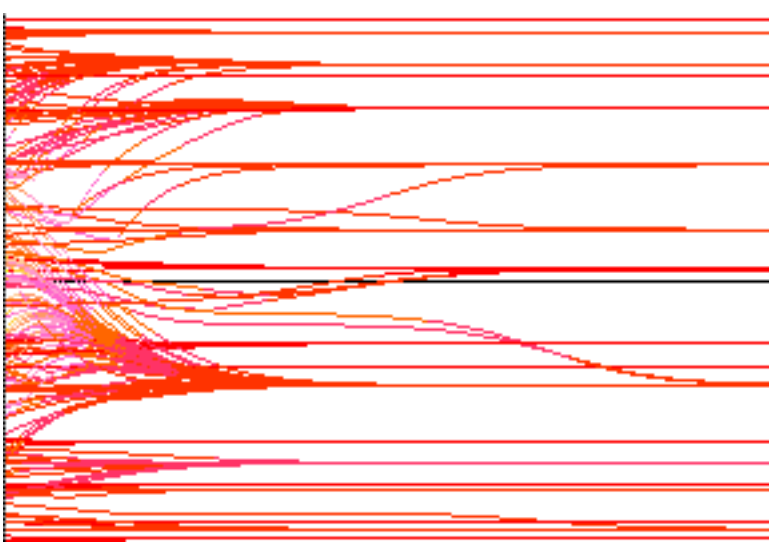
```
let modif i j mu = (* modif'' met les mise à jour de pos et unc dans pos' et unc' *)
        let f = influence i j
        and f' = influence j i
        and modif' t t' i j r mu = if r > 0. then t'.(j) <- t.(j) +. mu *. r *. (t.(i) -. t.(j)) in
        let modif'' i j r mu = (modif' pos pos' i j r mu; modif' unc unc' i j r mu) in
        if f > max 0. f' then (modif'' i j f mu ; if f' > 0.
                then modif'' j i f' mu)
                else if f' > max 0. f then (modif'' j i f' mu ; if f > 0. then modif'' i j f mu);;

let generation mu = for i = 0 to np-1 do for j = i+1 to np-1 do modif i j mu done done;;

let recopie t1 t2 = for i = 0 to vect_length t1 - 1 do t2.(i) <- t1.(i) done;;

let rec nth q n = if n = 0 then hd q else nth (tl q) (n-1) ;;

let iterer mu n = open_graph'' ''; moveto 0 0; lineto 0 200; moveto 0 100; lineto 480 100;
        (* let s = ref 0 in while not (key_pressed()) do incr s done; read_key(); random__init !s ; *)
        for i = 0 to np-1 do pos.(i) <- random__float 2. -. 1.;
                        unc.(i) <- random__float (1. -. abs_float pos.(i)) done;
        let ordo i = 100 + round (100.*.pos.(i)) in
                (* k = numéro d'itération = abscisse, n = nb d'itérations *)
        for k = 1 to n do generation mu; recopie pos' pos; recopie unc' unc;
                for i = 0 to np-1 do
                        set_color (nth [16711680; 16724736; 16730215; 16737792;
                                        16740022; 16747192; 16750950; 16765314;
                                        16767792; 16776960 ] (round (unc.(i)*.10.)));
                        moveto k (ordo i); lineto k (ordo i) done
        done; read_key(); close_graph();; (* iterer 0.02 480;; *)
```

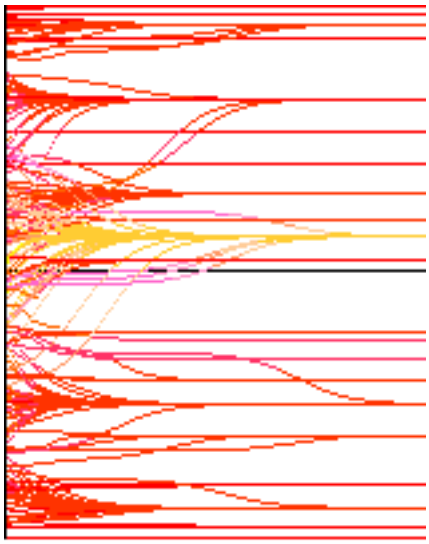
L'expérimentation montre que μ doit être très faible (de l'ordre de 0.05) pour que la convergence vers les positions les plus déterminées (u petit points rouges) ne soit pas trop rapide. Lorsque les données sont aléatoires, ce sont les plus déterminées qui constituent plusieurs valeurs d'adhérence (et pas seulement deux extrèmes) en attirant les moins déterminés (points les plus jaunes).

Voir : Deffuant G. Amblard F. Weisbuch G. Faure T. *How can extremism prevail ? A study based on the relative agreement interaction model,* Journal of Artificial Societies and Social Simulation 2002 <jass.soc.surrey.ac.uk>



μ = 0.02

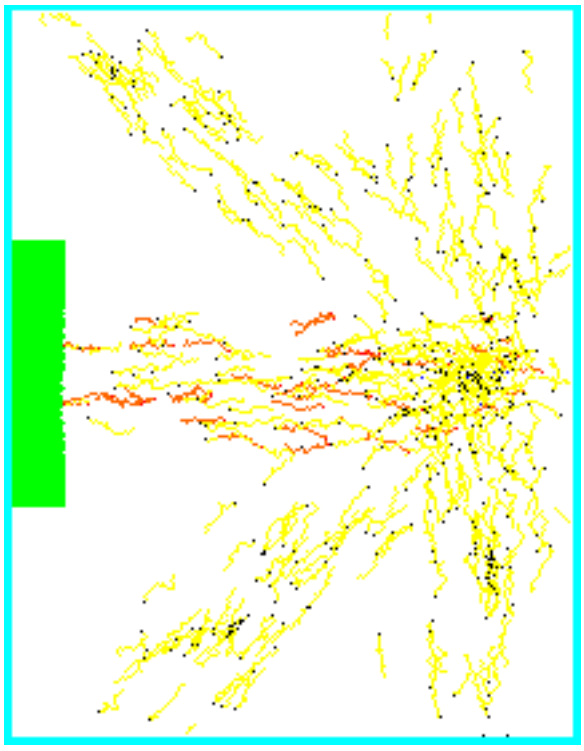




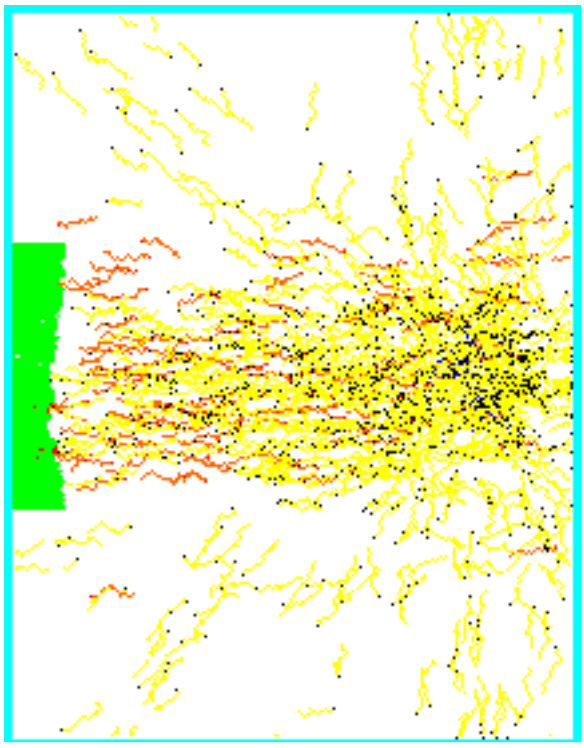
Exemples pour $\mu = 0.05$ et $\mu = 0.1$

2. Simulation de fourmillière

Le but est de simuler, en suivant pas à pas, une fourmillière réduite où le déplacement des fourmis à la recherche de nourriture (couleur verte) sera systématique. Chaque fourmi (couleur noire) sortant du nid pour une exploration laissera une trace de phéromone (jaune) sur les ns dernières cases parcourues. On simule ainsi l'évaporation. Lorsqu'elle atteint de la nourriture, elle s'empare d'une unité (qui peut être par exemple un pixel vert), change de couleur pour que l'on puisse étudier le processus et cherche à rentrer au nid. Durant son voyage de retour, la trace de phéromone sera légèrement différente (couleur orange) de façon à guider les autres fourmis exploratrices.



A gauche, début d'exploration, cas d'une seule source de nourriture.



L'exploitation bat son plein en lançant : sim 2 10000;;.

Initialisation

Le fond d'écran étant blanc, son bord bleu, on détermine la fourmillière par une zone carrée ou circulaire placée au centre invisible ou légèrement colorée. Des dépôts de nourriture (colorés en vert) sont disposés autour. Prévoir plusieurs dispositions gaussiennes : 2 zones à droite et à gauche, ou bien 4 zones aux extémités de la fenêtre, ou encore une diffusion uniforme ou par taches...

Un nombre nf de fourmi (colorées en noir) est distribué dans la fourmillière, chacune aura une direction propre parmi 8 directions. Ces directions sont déterminées initialement au hasard (dans la réalité, une fourmi décide de sa direction d'exploration et s'y tient plus ou moins grâce à une perception du champ magnétique terrestre). Les fourmis ne sont envoyées en exploration que graduellement, par exemple de façon croissante par rapport au nombre d'étapes de la simulation et à la quantité de nourriture rapportée.

Ordre de grandeur des constantes nf = 1000, probabilités définies plus loin : pdv = 0.4, pdg = pdd = 0.2, pga = pdr = 0.1, pag = pa = pad = 0, m = 1.25 ou plus, ns = 10, 20, 30...

Règles de transition

Pour chaque foumi, la case où elle se trouve et ses 8 voisines sont analysées, on nomme relativement à sa direction les cases voisines par DV, DG, GA, AG, A, AD, DR, DD (devant, devant-gauche, gauche, arrière-gauche, arrière, arrière-droite, droite, devant-droite).

1) Si la case où se trouve la fourmi, comporte de la nourriture, et que la fourmi n'en porte pas, elle reste sur cette case, prend une unité de nourriture, et inverse sa direction. Cette foumi sera signalée en rouge.

2) Si la fourmi est porteuse de nourriture et arrivée dans la fourmillière (celle-ci peut être représentée par un disque visible ou non), elle dépose cette nourriture et redevient noire après avoir traverser le nid. Elle indique ainsi d'où elle vient à un gplus grand nombre de ses soeurs et peut ensuite revenir sur ses pas. La nourriture, logée dans des depôts souterrains, apparaîtra en bleu.

3) Si la fourmi est à vide et qu'une case voisine comporte de la nourriture, elle s'y dirige. Si plusieurs cases voisines en comporte, celle qui est choisie est la première à partir de la direction de la fourmi, avec les règles de probabilité du 5.

4) Si un bord (cyan) est voisin, la fourmi revient sur sa trace et elle modifie sa direction soit en l'inversant, soit par exemple en lui ajoutant $\pi/4$.

5) Si une fourmi non porteuse rencontre une fourmi porteuse, elle modifie sa direction en prenant l'opposée de celle de la fourmi porteuse. C'est le seul moment où il y a communication entre deux fourmis.

6) Si pas de nourriture sur ces 9 cases ou bien que la fourmi est chargée et qu'une case voisine comporte de la nourriture, alors la fourmi se déplace sur une des 5 cases voisines suivant les probabilités DV : pdv, DG : pdg, DD : pdg, GA : pga, DR : pga et 0 pour les 3 cases arrières.
Toutefois, si l'une de ces cases est un bord, elle ne s'y rend pas et il se peut même suivant la position et la direction de la fourmi, que la case de déplacement soit obligatoirement, si elle est la seule ne faisant pas partie du bord AG, soit AD, ou bien que l'une des 2 soit tirée au sort si elle sont libres.
Au cas où une fourmi chargée passe sur de la nourriture, cette nourriture doit réapparaître l'étape suivante. Du point de vue pratique, on tire un réel aléatoire dans [0, 1], et si celui-ci est inférieur à pdv, c'est le déplacement DV qui sera choisi.
Sinon, en retirant un réel aléatoire de [0, 1], s'il est inférieur à pdg / 2*(pdg+pga) c'est DG qui est choisi etc. C'est pourquoi il peut être préférable de choisir des probabilités conditionnelles

comme pdg = pr(DG / ¬DV) = 1/3, pga = 0.25, pdr = 1/3, pdd = 1 si les cases sont examinées dans ce sens positif, ou bien pdv = 0.4, pdg = 1/3, pdd = 0.5, pga = 0.5, pdr = 1 si les tests se font dans cet ordre.

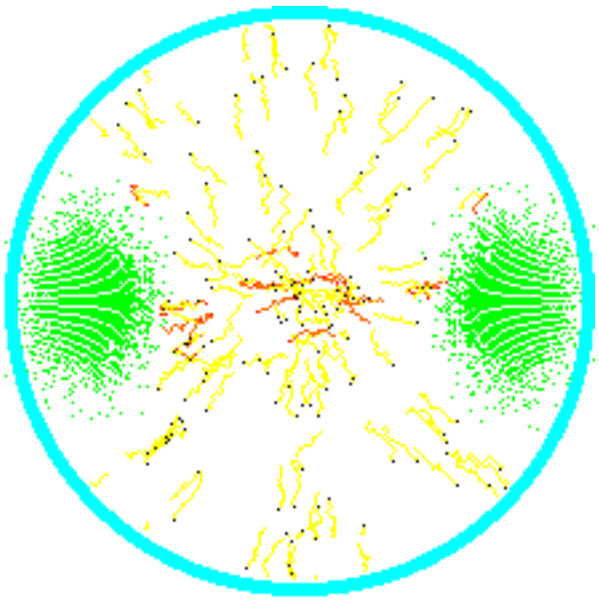
7) Si la case candidate au déplacement comporte de la phéromone d'exploration sa probabilité est multipliée par m, et si elle comporte de la phéromone de retour, par m^2.

Extensions : une stratégie plus simple consiste à suivre toute trace de phéromone voisine. Une autre à accorder sa direction avec les traces de phéromone de retour, si non chargée.
Une simulation plus fine consiste à représenter la trace par ns cases de couleurs dégradées de rouge au jaune par exemple, et la probabilité de suivre cette trace serait alors fonction de cette intensité.

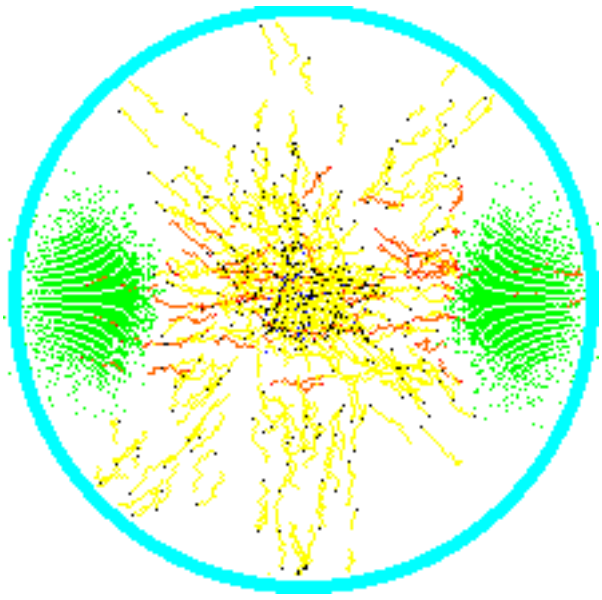
```
#open "graphics";; (* Ecran 0*480 sur 0*240 *)
let rec tete ll n = if ll = [] or n = 0 then [] else (hd ll):: (tete (tl ll) (n - 1));; (* n premiers éléments *)
let rec nth ll n = if n = 0 then hd ll else nth (tl ll) (n-1) and last = fun [x] -> x | (_::q) -> last q;;

let hasard ll = nth ll (random__int (list_length ll))
and has a b = a + random__int (b - a + 1)
and opp (x, y) = -x, -y;;

let gauss m s = let som = ref 0 in for i = 1 to 12 do som := !som + has (-100) 100 done;
        m + (s*(!som))/200;; (* donne un entier avec m et s entiers *)

let poisson m = poissonbis (exp (-. (float_of_int m))) 0 (random__float 1.)
        where rec poissonbis ex n x = if x <. ex then n
                else poissonbis ex (n+1) (x *. (random__float 1.));;

let round x = if x >=. 0. then int_of_float(x +. 0.5) else -(int_of_float (-. x +. 0.5));;  (* arrondi *)

let place x y c = set_color c; moveto x y; lineto x y
and couleur (x, y) = point_color x y;;

let centre i j = abs (240 - i) + abs(140 - j) < 30;;
        (* définit le grenier où la nourriture doit être ramenée *)

let cadre c = set_color white; fill_rect 0 0 (size_x ()) (size_y ()); set_color c;
        set_line_width 5; moveto 0 3; lineto (size_x ()-3) 3; lineto (size_x ()-3) (size_y ());
        lineto 0 (size_y ()); lineto 0 3; set_line_width 1;;

let rec qqch c = fun [] -> false | ((x, y)::v) -> if point_color x y = c then true else qqch c v;;

let rec colorer c = fun [] -> ()
        | [(x, y)] -> (if not (mem (point_color x y) [cyan; blue; green]) then place x y white)
        | ((x, y):: lp) -> (if point_color x y = white then place x y c; colorer c lp);;

type fourmi = {mutable pos : int*int; mutable porte : bool; mutable retour : bool;
                mutable avant : color; mutable dir : int*int; mutable route : (int*int) list};;
                (* type direction = N | NO | O | SO | S | SE | E | NE;; est devenu inutile *)

let proba = [0.4;0.33;0.5;0.5;1.;1.] and m = 1.5 and ns = 20 (* constantes *)
        and nf = 1000 (* nombre de fourmis *) and nn = ref 0 and orange = 16737792;;

let voisins i j d = map (fun (x, y) ->(x+i, y+j)) (voisins' d)
        (* donne la liste triée des voisins de (i, j) en direction d *)
where rec voisins' d =
        match d with (1, 1) -> [(1, 1); (0, 1); (1, 0); (-1, 1); (1, -1); (-1, 0); (0, -1); (-1, -1)]
                | (-1, 1) -> [(-1, 1); (0, 1); (-1, 0); (-1, -1); (1, 1); (1, 0); (0, -1); (1, -1)]
                | (1, 0) -> [(1, 0); (1, 1); (1, -1); (0, 1); (0, -1); (-1, 1); (-1, -1); (-1, 0)]
                | (0, 1) -> [(0, 1); (-1, 1); (1, 1); (-1, 0); (1, 0); (-1, -1); (1, -1); (0, -1)]
                | (x, y) -> map opp (voisins' (opp d));;
```

```
let tourne=fun (1,0)-> (1,1)
                | (1,1) -> (0,1)
                | (0,1) -> (-1,1)
                | (-1,1) - > (-1,0)
                | (-1,0) -> (-1,-1)
                | (-1,-1) -> (0,-1)
                | (0,-1) -> (1,-1)
                | (1,-1) -> (1,0);;

let suivant v = suivant' proba v (* donne la case suivante parmi la liste v des voisins *)
where rec suivant' lp = fun [] -> (0, 0)  (* ce premier cas inutile, évite les warnings *)
        | [(u, v)] -> (u, v)
        | ((u, v) :: lv) -> if random__float 1. <. (coef (point_color u v))*.(hd lp)
                        then (u, v) else suivant' (tl lp) lv  (* lv est la liste triée des cases voisines *)
and coef c = if c = yellow then m else if c = orange then m*.m else if c = cyan then 0. else 1. ;;

let rec vaprendre = fun [] -> (0, 0)
                        | ((u, v) :: lv) -> if point_color u v = green then (u, v) else vaprendre lv;;

let tf = make_vect nf {pos = (0, 0); porte = false; retour = false; avant = white;
        dir = (0, 0); route = []};;             (* tf est le tableau de toutes les fourmis *)

let dirporteuse v = let i = ref 0 in while not(mem tf.(!i).pos v) do incr i done; tf.(!i).dir;;
        (* seule recherche pénible *)

let init p = open_graph" ";
        for i = 0 to nf-1 do
                tf.(i) <- {pos = (gauss 240 8, gauss 140 8); porte = false;
                retour = false; avant = white; route = []; dir = (hasard [-1; 0; 1], hasard [-1; 0; 1])};
                if (tf.(i)).dir = (0, 0) then (tf.(i)).dir <- (0, hasard [-1; 1])
        done;
        cadre cyan ; set_color green;
        match p with
        1 -> (set_line_width 5; draw_circle 240 140 75;
                set_color cyan; draw_circle 240 140 80; set_line_width 1)
        | 2 -> (fill_rect 25 90 65 100; set_color cyan; fill_rect 280 0 25 300; fill_rect 0 0 70 300)
        | 3 -> (fill_ellipse 50 140 20 80; fill_ellipse 400 140 20 80; fill_ellipse 240 50 100 20)
        | 4 -> for i = 1 to 2000 do
                place (poisson 20) (gauss 140 40) green;
                place (480-poisson 20) (gauss 140 40) green done
        | _ -> (for i = 1 to 5000 do let r = float_of_int (gauss 80 10)
                                        and t = float_of_int (gauss 0 10) /.60. in
                place (240 + (round (r*.cos t ))) (140 + (round(r*.sin t ))) green;
                place (240 - (round (r*.cos t ))) (140 - (round(r*.sin t ))) green
                done;
                set_color cyan; set_line_width 5; draw_circle 240 140 110; set_line_width 1);;

let transition k = let f = tf.(k) in let (i, j) = f.pos in let v = voisins i j (f.dir) in place i j (f.avant);
        if f.avant = green & not f.porte
        then (f.avant<- red; place i j white; f.porte<- true; f.dir<- (opp f.dir))
        else if not(centre i j) & f.porte & f.retour
        then (place (gauss 240 9) (gauss 140 9) blue; f.porte<- false; f.dir<- opp f.dir;
                f.retour<- false; incr nn)
        else if qqch cyan v then (f.dir <- opp f.dir; f.pos <-last f.route;
                f.avant <-couleur (last f.route)) (* accélére *)
        else (let (x, y) = if not f.porte & qqch green v then vaprendre v else suivant v
                in f.pos <- (x, y);
                if (not f.porte) & (qqch red v) then f.dir <- (opp (dirporteuse v));
                if centre x y & f.porte then f.retour <- true;
                f.avant <-point_color x y;  f.route <- tete((i, j)::f.route) ns;
                place x y (if f.porte then red else black);
                colorer (if f.porte then orange else yellow) f.route);;
```

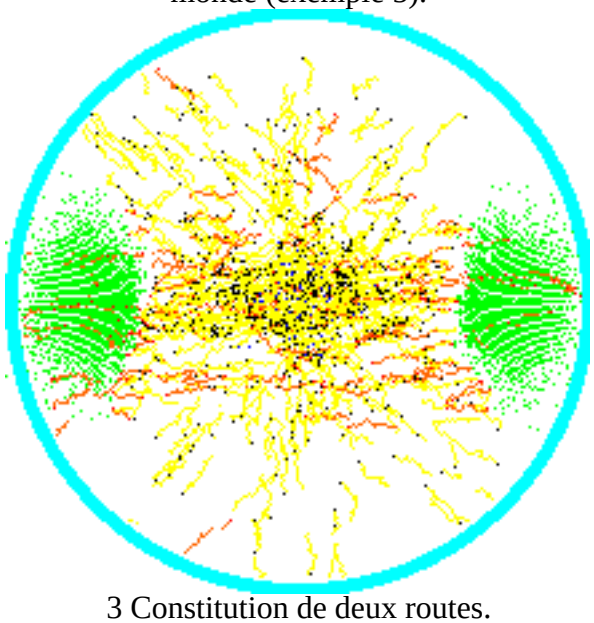
```
let sim ex ng = init ex; moveto 5 7;
        for n = 1 to ng do for k = 0 to min (nf-1) (n + !nn) do transition k done done;
        moveto 180 7; set_color black;
        draw_string ("Rendement " ^ (string_of_int ((1000000 * !nn) / (nf * ng))) ^
        " / 1000000 Appuyez sur une touche. "); read_key(); close_graph();;
```



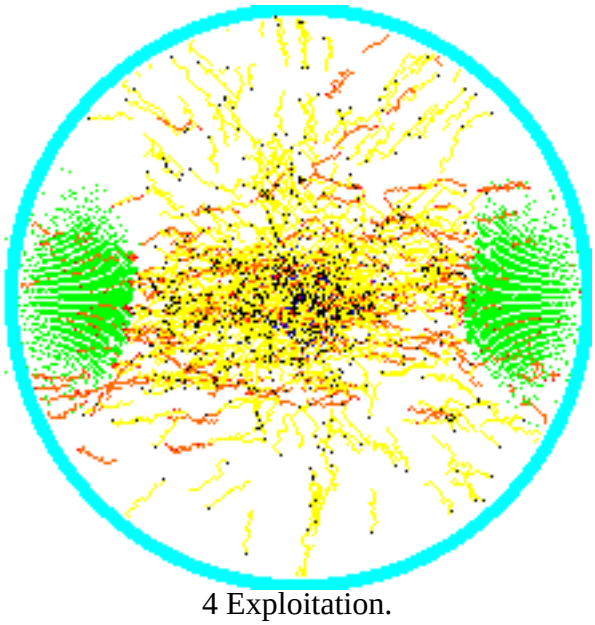
1 Départ de quelques exploratrices sur 1000 fourmis et premières découvertes de nourriture, les autres exploratrices arrivent aux limites du monde (exemple 5).



2 De plus en plus de fourmis sortent du nid, retour de fourmis bredouilles et début de communication.



3 Constitution de deux routes.



4 Exploitation.

Corbara B. *La cité des abeilles*, Gallimard 91
Fabre J.H. *La vie des insectes* (Un des plus anciens)
Maeterlinck *La vie des fourmis* (auteur également de la vie des termites et la vie des abeilles)
Von Frisch K. *Vie et moeurs des abeilles* (Premières éditions en 1927, en français en 1955)

3. Optimisation du voyageur par le modèle de l'élastique

Soient n villes x_i à parcourir en trouvant la boucle la plus courte ne passant pas deux fois par la même ville et n'en manquant aucune.
L'heuristique de l'élastique, qui n'est pas sans rapport avec celle des essaims de particules, consiste à partir d'un ensemble de m.n points y_k disposés en polygone régulier initialement centré au sein du nuage de points.
Le rapport m = 3 est expérimentalement assez bon, et le cercle initial doit être assez petit (ici r = 50 pixels) et plus ou moins centré sur le barycentre des villes. A chaque génération, les points y sont modifiés par une règle du type (les voisins de i sont i-1 et i+1 modulo m.n) :

$$y_j(t+1) = y_j(t) + \alpha\sum w_{ij}(x_i - y_j) + \beta k\sum_{j' \text{ voisin de } j} (y_{j'} - y_j).$$

Il s'agit donc d'un système d'agents qui s'atirent, cette attirance évoluant dans le temps, et qui sont attirés par des agents fixes contraignants.
La "température" k est un paramètre graduellement réduit à la manière du recuit simulé, initialement 0.2 et réduit de 1% à chaque génération par exemple.
On peut prendre $\alpha = 0.2$ et $\beta = 2$ par exemple.

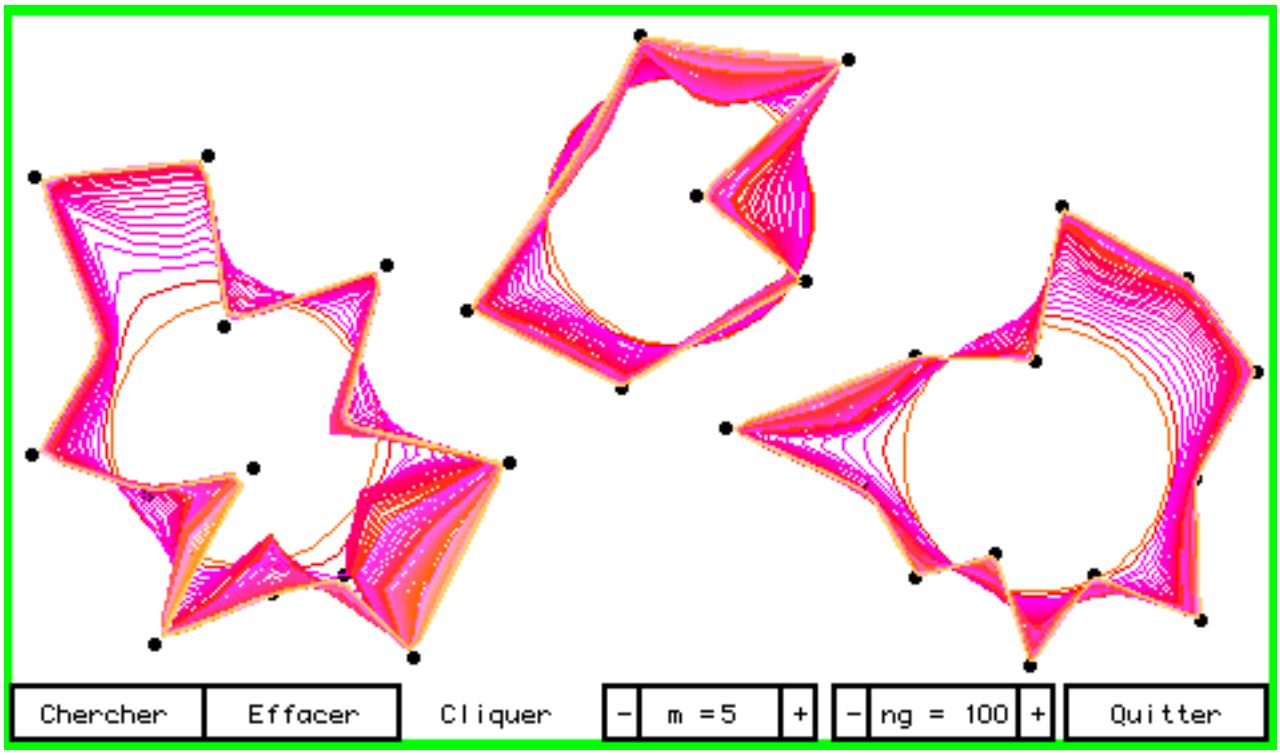
Les poids sont $w_{ij} = f(|x_i - y_j|) / \sum_p f(|x_i - y_p|)$ où f est une fonction gaussienne d'attraction définie par $f(x) = \exp(-x^2/2k^2)$.

Pour la programmation, on va rester sur la structure de deux vecteurs, plutôt que des cycles, ce qui est quand même plus simple, d'où les fonctions simples sur les vecteurs. Toutes les coordonnées seront réelles, la distance est euclidienne, et, comme d'habitude, il faut arrondir les coordonnées afin de tracer des "point" et "trait".

Voir : Durbin R. Willshaw D. *An analogue approach to the travelling salesman problem using an elastic net method,* Nature n°326 p689-691, 1987

```
let rec  sqr x = x*.x
and dist (x, y) (x', y') = sqrt (sqr (x-.x') +. sqr (y-.y'));;

let round x = if x >=. 0. then int_of_float(x +. 0.5) else -(int_of_float (-. x +. 0.5));;  (* arrondi *)

let cycle i n = if i < 0 then i + n else if n <= i then i - n else i;; (* car modulo non exact si x < 0 *)

let vecteur p1 p0 = (fst p1 -. fst p0, snd p1 -. snd p0)

and somvect v v' = (fst v +. fst v', snd v +. snd v');;

let mul a v = (a*.(fst v), a*.snd v);; (* multiplication*)

let voisins i dim =  (cycle (i - 1) dim, cycle (i + 1) dim);;
        (* renvoie les indices des deux points voisins de l'élastique *)

let attire d k = exp (-. (sqr (d /. k)) /. 2.);;

#open "graphics";; (* Ecran 0*480 sur 0*240 *)
let orange = 16737792;;

let place x y c = set_color c; moveto x y; lineto x y ;;

let point (x, y) c = place (round x) (round y) c

and trait (x, y) = lineto (round x) (round y);;

let dessine  t c =  point t.(0) c; for i = 0 to vect_length t-1 do trait t.(i) done; trait t.(0);;
```

```
let transition k al bt n m tv el = (* al = alpha, bt = beta, n = nb de villes, m = 2.5 ou autre  *)
        let el' = make_vect (n*m) (0., 0.)
        and denom = make_vect n 0. in
                (* denom.(i) est le dénominateur du poids wij, la somme des poids est 1 *)
                for i = 0 to n - 1 do for p = 0 to n * m - 1 do
                        denom.(i) <-  denom.(i) +. attire (dist tv.(i) el.(p)) k done done;
        for j = 0 to n*m - 1 do let (v1, v2) = voisins j (n*m) and delta = ref (0., 0.) in
                for i = 0 to n - 1 do
                delta := somvect !delta
                                (mul ((attire (dist tv.(i) el.(j)) k) /. denom.(i)) (vecteur tv.(i) el.(j)))
                done;
                delta := somvect (mul  al  !delta )
                                (mul bt (somvect (vecteur el.(v1) el.(j)) (vecteur el.(v2) el.(j))) );
                el'.(j) <-  somvect el.(j) !delta
        done;
        el';; (* les mises à jour sont synchrones *)

let generations ki al bt m lv ng = (* réalise ng  générations enchaînées *)
        let n = list_length lv
        and tv = map_vect (fun (x, y) -> (float_of_int x, float_of_int y)) (vect_of_list lv) in
        let el = ref (make_vect (n * m) (0., 0.)) and a = 6.28 /.(float_of_int (n*m))
        and r = 50. and x = ref 0. and y = ref 0. in
                for i = 0 to n - 1 do x := !x +. fst tv.(i); y := !y +. snd tv.(i) done;
                x := !x  /.float_of_int n; y := !y  /. float_of_int n;
                for j = 0 to n * m - 1 do
                !el.(j) <-  (!x  +. r *. (cos ((float_of_int j) *. a)), !y  +. r *. (sin ((float_of_int j) *. a)))
                done;
        dessine !el orange; let k = ref ki in
        for g = 0 to ng do
                el :=  transition !k al bt n m tv !el; dessine !el (red + 500 * g);
                k := 0.99*.(!k)
        done;;

exception Exit;;
```

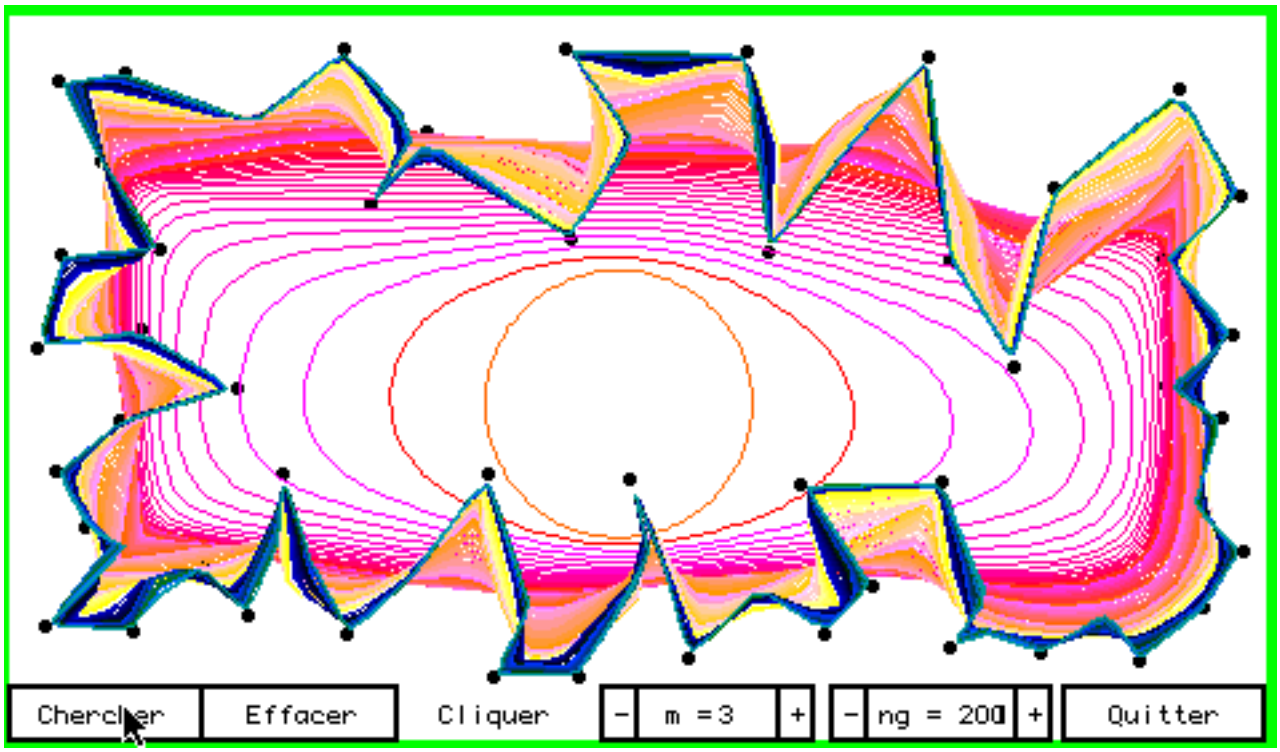


Trois exemples réunis dans la même fenêtre pour k = 20 initialement, $\alpha = 0.7$ et $\beta = 0.1$. Le nombre de points de l'élastique est 5 fois le nombre de villes et 10 fois pour le tracé du haut. On trace la succession des 100 élastiques à chaque fois.
Afin de réaliser, ces interfaces, on programme:

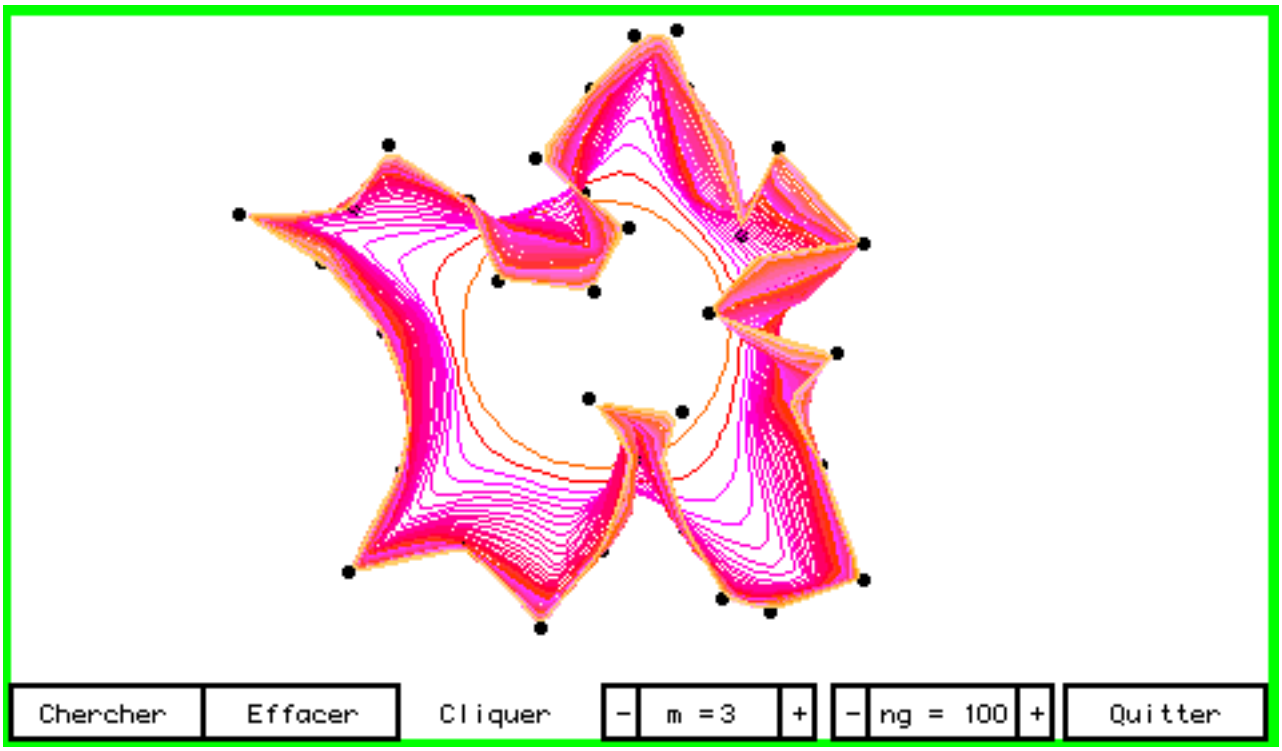
```
let cadre () = set_color green; fill_rect 0 0 480 280;  set_color white;
        fill_rect 4 4 471 271; set_line_width 1;;

let quitter x y =  400 < x & y < 25

and chercher x y = x < 75 & y < 25

and effacer x y = 75 < x & x < 148 & y < 25

and mmoins x y = 225 < x & x < 238 & y < 25

and mplus x y = 291 < x & x < 311 & y < 25

and gmoins x y = 311 < x & x < 324 & y < 25

and gplus x y = 380 < x & x < 395 & y < 25;;

let boite x y l h c = set_color black; fill_rect x y l h;
        set_color white; fill_rect (x + 2) (y + 2) (l - 4) (h - 4) ;
        moveto (x + 6) (y + 6); set_color black; draw_string c;;

let rec villes ki a b = open_graph""; cadre();
        boite 3 3 75 22 " Chercher ";
        boite 75 3 75 22 "  Effacer ";
        boite 225 3 15 22 "- ";
        boite 238 3 55 22 " m = ";
        boite 291 3 15 22 "+ ";
        boite 311 3 15 22 "- ";
        boite 324 3 58 22 "ng = ";
        boite 380 3 15 22 "+ ";
        boite 398 3 77 22 "  Quitter ";
        moveto 165 9; draw_string "Cliquer"; set_color black;
        let lp = ref [] and m = ref 5 and ng = ref 100 and drap = ref false in
                (* liste des points, m fourmis, ng générations *)
        try while true do moveto 270 9; set_color 0; draw_string (string_of_int !m);
                        moveto 360 9; draw_string (string_of_int !ng);
                        let e = wait_next_event [Button_down] in
                        if e.button then let (x, y) = mouse_pos() in
                        if effacer x y then villes ki a b
                        else if chercher x y then (drap := true; generations ki a b !m !lp !ng)
                        else if quitter x y then raise Exit
                        else if mmoins x y then (m := max 1 (!m - 1); set_color white;
                                fill_rect 270 5 15 18; set_color black)
                        else if mplus x y then (m := min 20 (!m + 1); set_color white;
                                fill_rect 270 5 15 18; set_color black)
                        else if gmoins x y then (ng := max 1 (!ng - 1); set_color white;
                                fill_rect 360 5 15 18; set_color black)
                        else if gplus x y then (ng := min 999 (!ng + 1); set_color white;
                                fill_rect 360 5 15 18; set_color black)
                        else (lp := (x, y) :: (!lp); set_color  black; fill_ellipse x y 2 2)
        done
with Exit -> close_graph();;
```

Comme la plupart des heuristiques d'intelligence artificielle, le choix des paramètres initiaux est très délicat.

Empiriquement on peut remarquer que si m est élevé l'élastique va être trop étoilé et une ville risque d'attirer plusieurs points de l'élastique, si m faible, il sera trop rond. Néanmoins, la méthode marche assez bien avec m aux environs de 2.5 pour un grand nombre de villes, et elle est très rapide. 60 villes et 200 générations k = 50. $\alpha = 0.7$ $\beta = 0.1$) :



Ci-dessous 35 villes de France :



4. L'optimisation par essaim de particules PSO (μ, α, β, γ)

L'optimisation par essaim de particules est motivée par la simulation de comportement social. Dans cette technique, pour t = 0, un ensemble P_0 de μ solutions $x_i(0)$ sont choisies au hasard dans le domaine de la fonction à minimiser f, chacune (particule) a une position $x_i(t)$ et une velocité $v_i(t)$.

Chaque génération t, la "fitness" f est évaluée à la position $x_i(t)$, soit g la meilleure de toutes. Pour chaque particule i, on regarde dans son voisinage (ou son groupe) pour obtenir le meilleur local "leader du groupe", et :

$v_i(t+1) = \alpha v_i(t) + \beta(n - x_i(t)) + \gamma(g - x_i(t))$ sera la nouvelle vitesse et :
$x_i(t+1) = x_i(t) + v_i(t+1)$ sera la nouvelle position (on a ici une sorte de tri-crossover).

Le problème des paramètres est réduit en prenant
$v_i(t+1) = \chi[\alpha v_i(t) + \beta\phi_1(n - x_i(t)) + \beta\phi_2(g - x_i(t))]$ où χ (coefficient de constriction dans]0,1], plus χ est grand, plus la convergence est lente) et ϕ_1, ϕ_2 pris au hasard entre 0 et 4.

Plusieurs options peuvent être prise pour définir les groupes, la plus simple est de prendre les nn = 5 plus proches voisins de la population de μ = 100 particules. C'est l'option "geom" de la fonction "pso". Une autre façon est de définir arbitrairement dès le départ comme les nn particules de plus proches indices voisins (indices définis cycliquement).

Une autre façon serait de prendre les voisins dans un rayon σ comme |b - a] / 5 par exemple si f : $[a, b]^{dim}$ -> R.

On s'attend par cette heuristique à obtenir des "essaims" autour des optima locaux, mais le problème, comme toutes ces heuristiques réside dans la mise au point des paramètres initiaux.

Les vitesses initiales sont choisies comme au maximum 10% de l'intervalle [a, b] et on reste dans $[a, b]^2$. Pour les coefficients α, β, γ, on favorise le comportement de groupe en prenant β plus grand que les autres.

Experimentalement, on peut dire grossièrement que plus α, est grand, plus grande est la dispersion des nuages et le mouvement dans l'espace. Mais la convergence vers un minimum local n'est pas enrayée par un faible γ et la formation de nuages locaux n'est pas non plus favorisée par un grand β. L'évolution montre que les nuages ont tendance à rester trop statiques autour de minima locaux.

On définit quelques petites fonctions servant identiquement aux heuristiques suivantes :

```
let nl () = print_newline();;
let rec nth q n = if n = 0 then hd q else nth (tl q) (n - 1) and last = fun [x] -> x | (_ :: q) -> last q;;

let hasard q = nth q (random__int (list_length q))
and has a b = a + random__int (b - a + 1)
and has_float a b = a +. random__float (b -. a);;

let rec reste p q = if p < 0 then reste (p+q) q else if p < q then p else reste (p - q) q;; (* modulo *)
let round x = if x >=. 0. then int_of_float(x +. 0.5) else -(int_of_float (-. x +. 0.5)) (* arrondi *)
and sqr x = x *. x;;

let rec tete q n = if q = [] or n = 0 then [] else (hd q):: (tete (tl q) (n - 1)) (* les n premiers de q *)
and queue q n = if q = [] or n = 0 then q else queue (tl q) (n - 1);; (* tout sauf les n premiers de q *)

#open "graphics";;
let efface m = set_color white; fill_rect 0 0 m m
        (* petits dessins dans [0, m]2, prendre m = 100 à 200 *)
and cadre m = set_color blue; set_line_width 2;
        moveto 0 0; lineto m 0; lineto m m; lineto 0 m; lineto 0 0;;
```

```
let dilat m x a b = round (m *. (x -. a) /. (b -. a));; (* exprime le réel x de [a, b] dans 0..m *)
let contract lc a b = contract' ((b -. a) /. 10.) a lc  (* codage liste de chiffres -> réel *)
where rec contract' k r = fun [] -> r | (x :: q) -> contract' (k /. 10.) (r +. (float_of_int x) *. k) q;;
```

```
contract [3;3;3;3;3;3;3;3;3;3] 7. 10.;; ☞ 7.99999999999
contract [3;3;3;3;3;3;3;3;3;3;3] 7. 10.;; ☞ 8.0
```

```
let nf = 100 and ng = ref 0 and nv = ref 0 and dim = 2;;
(* Les variables globales sont : nf est l'effectif de la population, ng, le nombre de générations et nv
le nombre d'évaluation de la fonction f à minimiser en dim 2 *)

type particule = {mutable score : float; mutable pos : float vect; mutable vit : float vect;
                  mutable dd : float};;

let tp = make_vect (2*nf) {score = 0.; pos = make_vect dim 0.; vit = make_vect dim 0.; dd = 0.};;
        (* tp = tableau de la population et dd = distance *)

let init a b = let v = make_vect dim 0. in for j = 0 to dim - 1 do v.(j) <- has_float a b done; v;;
        (* vecteur aléatoire dans le domaine [a, b]^dim *)

let mul a t = let r = make_vect dim 0. in for i = 0 to dim - 1 do r.(i) <- a *. t.(i) done; r
        (* opérations vectorielles *)
and som t t' = let r = make_vect dim 0. in for i = 0 to dim - 1 do r.(i) <-  t.(i) +. t'.(i) done; r
and dif t t' = let r = make_vect dim 0. in for i = 0 to dim - 1 do r.(i) <-  t.(i) -. t'.(i) done; r;;

let triv t n =
(* tri-bulle et réévaluation obligatoire après déf du type particule (même champ "score") *)
        let drap = ref true and k = ref (n - 1) in
        while !drap (* & 0 < !k *) do drap := false; decr k;
                for i = 0 to !k do if t.(i + 1).score <. t.(i).score then (ech i (i + 1); drap := true) done
        done
        where ech i j = let z = t.(i) in t.(i) <- t.(j); t.(j) <- z;;

let tore v a b = let w = make_vect dim 0. in  for i = 0 to dim - 1 do w.(i) <-  aux v.(i) done; w
        (* vecteur ramené à [a, b] *)
        where rec aux x = if x < a then aux (b -. a +. x) else if x > b then aux (a +. x -. b) else x;;

let meilleur t n = let m = ref 0 in for i = 1 to n - 1 do if t.(i).score < t.(!m).score then m := i done;
        !m;; (* renvoie l'indice du meilleur du tableau t de particules *)

let dist x y = let d = ref 0. in for i = 0 to dim - 1 do d := !d +. abs_float (x.pos.(i) -. y.pos.(i)) done;
        !d;;

let rec ins x q nr nn = if q = [] then [x]
        else if nr > nn then q
        else if x.dd < (hd q).dd then x :: q
        else (hd q) :: (ins x (tl q) (nr + 1) nn);;

let voisins i nn = let r = ref [] in
        for k = 0 to nf - 1 do
        if k <> i then (tp.(k).dd <- dist tp.(k) tp.(i); r := ins tp.(k) !r 0 nn) done;
        tete !r nn;; (* donne la liste des nn particules les plus proches *)
let groupe i nn = let r = ref [] in for k = 1 to nn do r := tp.(reste (i + k - 1 -  nn / 2) nf) :: !r done; !r;;
        (* nn indices voisins *)

let voir m = nl(); print_string ("ng=" ^ (string_of_int !ng)
                        ^ " nv= " ^ (string_of_int !nv) ^ " Meilleur = ");
        print_float tp.(m).score; print_string " pour (";
        for i = 0 to dim - 2 do print_float tp.(m).pos.(i) ; print_string ", " done;
        print_float tp.(m).pos.(dim-1); print_char `)`; let s = ref 0. in s := 0.;
        for i = 0 to nf - 1 do s := !s+.tp.(i).score done;
        print_string " moyenne = "; print_float (!s/.(float_of_int nf));; (* m =  indice du meilleur *)
```

```
let point m p w a b =
        let (x, y, u, v) = (dilat m p.(0) a b,
                            dilat m p.(1) a b,
                            round (m *. w.(0) /. (b -. a)),
                            round (m *. w.(1) /. (b -. a)))
        in set_color red; fill_rect x y 1 1 ;;
        (* On pourrait accessoirement faire un peu plus gros avec :
        fill_rect (x - 1) (y - 1) 3 3; set_color black; moveto x y; lineto (x + u) (y + v); *)

let graph m a b = efface (round m); cadre (round m);
        for i = 0 to nf - 1 do point m tp.(i).pos tp.(i).vit a b done;;
```

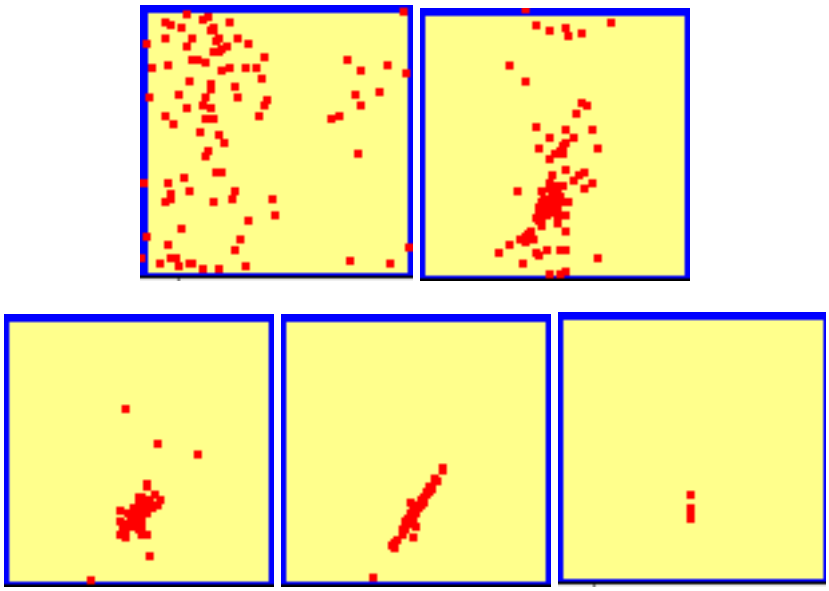
On rajoute un éclaircissement de la population, déjà vu au chapitre sur l'évolution artificielle. Avec un coefficient prox = 40 dans les premiers tests, qui remplace les doubles par des nouveaux et retrie, donc prend du temps. Mais "elim" est une option presque indispensable si on veut éviter la convergence prématurée.

```
let elim f a b eps =
        for i = nf - 1 downto 1 do
        if dist tp.(i) tp.(i-1) < eps
        then (incr nv; let n = init a b in
                if tp.(i).score < tp.(i-1).score
                then tp.(i-1) <-  {score = f n; pos = n; vit = init 0. 1.; dd = 0.}
                else tp.(i) <-  {score = f n; pos = n; vit = init 0. 1.; dd = 0.})
        done;;

let pso max eps f a b al bt khi nn geom visu =
        if visu then (open_graph" "; while not key_pressed () do print_int (random__int 9) done);
        nv := 0; for i = 0 to nf - 1 do
                tp.(i) <- {score = 0.; pos = init a b; vit = init ((a -. b) /. 10.) ((b -. a) /. 10.); dd = 0.};
                tp.(i).score <-  f tp.(i).pos done; (* initialisation et calcul des valeurs *)
        nv := !nv + nf;
        let m = ref (meilleur tp nf) in ng := 0; if visu then (voir !m; graph 100. a b);
        while !nv < max & eps <. tp.(!m).score do
                (*  nouvelle popu entre nf et 2nf car on veut modif synchrone *)
                for i = 0 to nf - 1 do let n = meilleur (vect_of_list
                                        (if geom then voisins i nn else groupe i nn)) nn in
                                let v = (mul khi (som (mul al tp.(i).vit)
                                        (som (mul (bt*.(has_float 0. 4.)) (dif tp.(n).pos tp.(i).pos))
                                        (mul (bt*.(has_float 0. 4.)) (dif tp.(!m).pos tp.(i).pos)))))
                                in (* v est la nouvelle vitesse de tp.(i) *)
                tp.(i+nf) <- {score = 0.; pos = tore (som tp.(i).pos v) a b; vit = v; dd = 0.};
                tp.(i + nf).score <- f tp.(i + nf).pos
                done;
                nv := !nv + nf; incr ng; for i = 0 to nf -1 do tp.(i) <- tp.(i + nf)
        done;
        m := meilleur tp nf; (* nouvelle popu au début *)
        if visu then (voir !m; graph 100. a b) done; if visu then (nl (); close_graph ());
        !nv;; (* afin de comparer des algorithmes divers, on renvoie le nombre d'évaluations de f
*)
```

Pour la fonction "tripod" déjà vue :

$x, y \in [-100, 100]$, $f(x, y) =$ if $y < 0$ then $|x| + |y + 50|$
else if $x < 0$ then $1 + |x + 50| + |y - 50|$
else $2 + |x - 50| + |y - 50|$

```
On lancera : pso 10000 1e-04 tripod (-100.) 100. 0.3 0.6 0.6 5 false true;;
```



Pour la fonction "tripod" qui a 3 minima, PSO a b k avec nn = 5 voisins arbitrairement définis pour les générations 3, 6, 11, 16, 35, on représente à chaque fois le carré $[-100, 100]^2$.

Angeline P. *Using selection to improve particle swarm optimization*, Proc. of the IEEE Int. conf. on Evolutionary Computation, 1998 (aussi http://ics.yediteps.edu.tr/~eozcam/pso/)
Eberhart R.C. Kennedy J. *A new optimizer using particles swarm theory*, Proc. 6-th symposium on micro-machine and human science, IEEE 1995 (http://ics.yeditepe.edu.tr/•eozcan/pso/

5. Self organizing migration algorithm SOMA(μ, r, step, mass, ml, χ)

L'idée principale de cette heuristique d'optimisation est de définir également des groupes où chaque individu va faire des petits pas vers son "leader" . une marche aléatoire (x_1, ...x_{ml}) est crée par chaque individu x_0 vers son "leader" m, de telle sorte que $x_i x_{i+1}$ est colinéaire avec x_0m. Ce petit "step" est appelé "migration loop" car une sorte de mutation en évolution classique.

1) P_0 est un esemble de μ points pris au hasard dans le domaine de f à minimiser (habituellement $[a, b]^{dim}$), le pire cas serait avec μ = 2 et l'initialisation des classes se fait :
2) Le meilleur individu m est choisi comme centre du premier groupe et les adhérents à ce groupe sont simplement les individus proches {x / d(x, m) < r} définis grâce à un rayon r qui determine un domaine d'attraction du leader. Du reste des individus, est de nouveau choisi le meilleur et un autre groupe est formé en fonction du même rayon r. Ceci est répété de telle sorte que le nombre de groupes est variable mais inférieur à μ. Il est possible d'éviter qu'un individu crée seul son propre groupe par des considérations sur μ, r et le domaine $[a, b]^{dim}$.
3) Chaque particule x commence des sauts vers son leader avec une distance fixe "step" (il y a donc plusieurs positions où la fonction f est à chaque fois évaluée). Le paramètre "mass" signifie que si par exemple mass = 1, le "voyage" ou "migration loop" s'arrête sur la première position derrière le leader, si mass = 2 à la seconde etc. Le nombre de sauts est borné par le paramètre ml et la position suivante qui sera retenue pour la particule est la meilleure de la successsion.
4) si non(condition d'arrêt) aller en 2 (les groupes sont donc redéfinis)
5) fin

Au point 3, il est possible de dispenser le leader du groupe de sauts, mais on observe alors une stagnation de ces leaders et il est beaucoup plus intéressant qu'ils en fassent eux-mêmes

aléatoirement. Nous avons testé différentes valeurs de "mass" avant de le fixer à 5. Le problème général est celui de fixer empiriquement les paramètres de l'algorithme.
L'expérimentation montre que les groupes ne doivent pas être trop petits, par exemple, une classe avec un unique élément pourrait être stationnaire et une classe avec seulement deux éléments restraindrait l'exploration à un segment trop petit de la droite qu'ils définissent.
Si nous souhaitons avoir une connaissance empirique des paramètres, nous pouvons supposer, que μ individus sont répartis uniformément dans l'espace $[a, b]^{dim}$ et considérer qu'en moyenne 5 individus par classe font $\mu / 5$ classes par exemple.
Ainsi si ν est l'effectif moyen des classes, et relativement à la distance de Hamming, si nous considerons que les μ/ν classes de diamètre 2r doivent recouvrir l'espace, nous avons une estimation de r à partir de $\mu/\nu = (b - a)^{dim}$.
En fait, dans les expériences $\mu = 100$ et une valeur raisonnable ν serait au moins 5, r doit être au moins 1 pour $[-5, 5]^2$, et au moins 7.5 dans $[-10, 10]^{10}$.

Expérimentalement, nous observons la formation de nuages pour la paramètre "step" aux alentours de r / 10. Pour être plus précis, dans une application pratique, si r = 2 dans $[-5, 5]^2$, alors nous choisissons step = 0.1. Mais pour éviter la stagnation "step" peut être aussi décroissant, et nous avons testé avec succès suivant une "température" comme $step(t) = step_0.e^{-0.05t}$, afin de l'abaisser à chaque génération.
Un autre point de discussion est la perturbation apportée par la possibilité de changer "step" ou de choisir quelques nouveaux individus aléatoires avec une probabilité χ durant la "migration loop". L'essai monte qu'une assez forte perturbation grâce à $\chi = 0.5$ donne de meilleurs résultats que sans perturbation.

Voir : Zelinka I. Lampinen J. *SOMA : self organizing migration algorithm*, Mendel Proc. p177-187, 2000

Pour la programmation, ce sont les mêmes fonctions diverses que précédemment, plus :

```
let rec ret x = fun [] -> [] | (a :: q) -> (if x = a then q else a :: (ret x q));;
        (* ne retire que la première occurrence de x *)

let rec aplatir = fun [] -> [] | ([] :: q) -> aplatir q | ((a :: q)::m) -> a :: aplatir (q :: m);;

let aff t t' n = for i = 0 to n-1 do t.(i) <- t'.(i) done;; (* copie d'un tableau dans un autre *)

let best li = best' (tl li) (hd li) where rec
        best' q m = if q = [] then m else best' (tl q) (if (hd q).score < m.score then hd q else m);;

let partition li r = part li li (best li) [] [] where rec (* lr = liste restante, li = liste d'individus *)
        part lr q m gm res =
        (* donne une liste de listes d'individus (boule de rayon r) débutant par leur leader *)
        if lr = [] then (m :: (ret m gm)) :: res (* res = [[l0, a1,...], [l1, b1, b2...], [l2, c1...], ....] *)
        else if q = [] then part lr lr (best lr) [] ((m :: (ret m gm)) :: res)
        else if dist (hd q) m < r then part (ret (hd q) lr) (tl q) m ((hd q) :: gm) res
        else part lr (tl q) m gm res;;

let travel f a b ind m step mass khi =
        let d, s, sm, p, pm = dist m ind, ref ind.score, ref ind.score, ref ind.pos, ref ind.pos
        in let u = mul (step/.d) (dif m.pos ind.pos) in
        for i = 1 to round (d/.step) + mass do
                p := if random__float 1. < khi then init a b else som !p u; s := f !p; incr nv;
                if s < sm then (sm := !s; pm := !p) done;
        {score = !sm; pos = !pm; vit = init 0. 1.; dd = 0.};; (* renvoie le meilleur du parcours *)
```

```
let modif f a b cl step mass khi = modif' cl (hd cl) []
        (*  hd cl renvoie une classe où chacun a voyagé vers son leader *)
        where rec modif' q m res =
                if q = [] then res
                else modif' (tl q) m ((travel f a b (hd q) m step mass khi) :: res);;

let soma max eps f a b r step mass khi visu =
        (*  li = liste d'individus, lc = liste des classes, m = meilleur individu *)
        if visu then (open_graph " "; while not key_pressed () do print_int (random__int 9) done);
        nv := 0;
        for i = 0 to nf -1  do
                tp.(i) <- {score = 0.; pos = init a b; vit = init 0. 1.; dd = 0.}; tp.(i).score <-  f tp.(i).pos
        done;
        nv := !nv + nf; ng := 0;
        let li = ref (list_of_vect (sub_vect tp 0 nf))
        in let m = ref (best !li) and lc = ref [] in
        while !nv < max & eps <. !m.score do
                incr ng;
                lc := map (fun c -> modif f a b c (step *. exp (-0.01*.(float_of_int !ng))) mass khi)
                        (partition !li r);
                li := aplatir !lc;
                m := best (!li);
                if visu then (aff tp (vect_of_list !li) nf; voir (meilleur tp nf); graph 100. a b);
                elim f a b step;
                li := (list_of_vect (sub_vect tp 0 nf))
        done;
        if visu then (nl (); close_graph ()); !nv;;
        (* le step est decru exponentiellemnt au fil des générations *)
```

Toujours pour la même fonction "tripod", on lancera par exemple :

```
soma 50000 1e-06 tripod (-100.) 100. 20. 2. 5 0.5 true;;
```

6. L'algorithme macro-évolutionnaire MGA (μ, ρ, τ)

Cette heuristique est inspirée de la dynamique des espèces et leur diversification. Les solutions candidates dans l'espace de recherche de la fonction f sont appelées espèces et sont reliées dans un réseau de poids.

Les liens entre individus determinent le nouvel état de la génération suivante : vivant ou éteint pour chacun d'entre eux, w_{ij} mesure l'influence de x_j sur x_i en temps réel, une influence négative en termes de minimisation, signifiera une survivance.

1 Une population de μ individus est aléatoirement choisie.
2 Une matrice de "connectivité" W est mise à jour grâce $w_{ij} = (f(x_i) - f(x_j)) / \| x_i - x_j \|$
3 A chaque génération t, si l'entrée $\sum_{j = 1...\mu} w_{ij}(t) \leq 0$, alors $x_i(t+1) = x_i(t)$, (en ce cas, x_i est meilleur que les autres en moyenne pondérée, x_i est dite vivante) sinon, x_i est supprimé et remplacé avec une probabilité τ par un nouvel individu aléatoire, (colonisation). Dans l'autre cas, une sorte de crossover continu est appliqué, en choisissant un individu aléatoire x_k dans la population comme un point attractif, et $x_i(t+1)$ devient $x_i(t+1) = x_k(t) + \rho\lambda(x_k(t) - x_i(t))$ avec λ un nombre aléatoire dans [-1, 1].
4 La "température" τ peut être fixée ou décroissante de façon à réduire l'exploration en faveur de l'exploitation.

   Comme pour le recuit simulé, une façon simple est de suivre $\tau(t) = 1 - t / ng_{max}$, ainsi τ abaissé à chaque génération.
5 aller en 2

Notons que les definitions des poids favorise l'élimination des voisins de solutions et maintient les différentes "solutions" à des distances importantes. Dans leur travail [Marin, Solé 99] ne trouvent pas une influence significative pour la constante ρ. Celle-ci peut être fixée à 0.5. Ils observent une amélioration par rapport aux AG, en particulier pour la fonction de Griewank (leurs meilleurs résultats étant obtenus avec un τ linéairement décroissant durant 500 générations).

Malheureusement, cette heuristique est très lente à cause d'une complexité en μ^2 et très peu d'individus sont remplacés à chaque génération, même dans le cas d'une forte probabilité de migration (0.5 à 0.8), le nuage des points qui se dessine autour des "leaders" est trop petit pour être capable par croisements, de poursuivre l'exploration.

Cette remarque sera aussi vérifiée pour SOMA, la meilleure solution est souvent très proche de la solution optimale, mais le fait de fixer mutation et croisement avec un voisin, entraîne une stagnation. Cependant, comme pour PSO, différents nuages de points sont maintenus durant toute l'évolution.

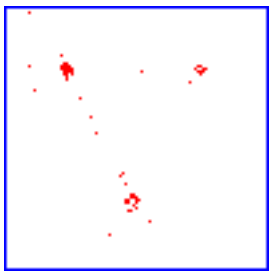
Voir : Marìn J. Solé R.V. *Macroevolutionary algorithms : a new optimization method on fitness landscapes,* IEEE Transactions on Evolutionary Computation, vol 3, n°4, p272-286, 1999

Les petites fonctions sont exactement celles de PSO.

```
let w = make_matrix nf nf 0.;;
let entree i = let s = ref 0. in
        for j = 0 to nf - 1 do s := !s +. w.(i).(j) done;
        !s;;

let mga max eps f a b pm visu = (* !m= indice du meilleur individu, tri inutile *)
        if visu then (open_graph" "; while not key_pressed () do print_int (random__int 9) done);
        nv := 0;
        for i = 0 to nf - 1 do
                tp.(i) <- {score = 0.; pos = init a b; vit = init 0. 1.; dd = 0.}; tp.(i).score <-  f tp.(i).pos
        done;
        nv := !nv + nf; ng := 0; let m = ref (meilleur tp nf) in
        while !nv < max & eps <. tp.(!m).score do
        (* w matrice antisymétrique des poids mise à jour à chq génération *)
                for i = 0 to nf - 1 do for j = 0 to nf - 1 do
                        if i <> j then w.(i).(j) <- (tp.(i).score -. tp.(j).score) /. (dist tp.(i) tp.(j))
                done done;
                for i = 0 to nf - 1 do
                        if entree i > 0.
                        then ( incr nv; tp.(i).pos <- if random__float 1. < pm then init a b
                                        else (let k = random__int nf in
                                        som tp.(k).pos (mul (has_float (-0.5) 0.5)
                                        (dif tp.(k).pos tp.(i).pos))); tp.(i).score <- f tp.(i).pos)
        done;
        incr ng; m := meilleur tp nf; if visu then (voir !m; graph 100. a b) done;
if visu then (nl (); close_graph ());
!nv;;
```

Toujours avec la même fonction tripod sur le carré $[-100, 100]^2$ avec p = 0.3, le dessin montre des lignes d'individus entre les minima locaux, dues à ce crossover spécial entre les bonnes solutions. Image de la population à la génération 35, trop de concentration. :

```
mga 50000 1e-06 tripod (-100.) 100. 0.3 true;;
```



7. L'évolution différentielle DE (μ, p_C, χ)

Dans cette heuristique, chaque individu peut être changé grâce à une sorte de "tetra-crossover" qui apporte une large perturbation. P est aléatoirement initialisée, et à chaque génération, pour un individu qui est un vecteur x, dont les composantes sont x_1, ..., x_{dim}, un numéro de composante k et trois autres individus x, y et z sont aléatoirement choisis tels que x, y, z, t soient distincts, alors, la mise à jour de x est x', vecteur dont les composantes sont :

$x'_j = t_j + \chi(y_j - z_j)$ pour j = k, mais aussi pour $j \neq k$ avec une probabilité p_C, sinon $x'_j = x_j$.

La mise à jour des générations est élitiste en remplaçant x par x' chaque fois que ce dernier est meilleur, c'est à dire f(x') < f(x). Dans l'autre cas, c'est x qui est conservé. Cet algorithme, initialement prévu pour opérer sur des variables continues a été étendu à des variables discrètes ou continues.

Les résultats ci-dessous sont assez bons (de plus, aucun tri de la population n'est nécessaire). Il est possible d'observer que la population se groupe de façon rectiligne autour des optimums (si p_C est bas) et se groupe très rapidement vers l'optimum global.
Des changements de p_C montrent que les meilleurs résultats sont obtenus avec 0.5, sans différence pertinente entre 0.4 et 0.6.

Mêmes petites fonctions que pour PSO :

```
let de max eps f a b pc khi visu = (* !m= indice du meilleur individu, tri inutile *)
        if visu then (open_graph" ";
        while not key_pressed () do print_int (random__int 9) done); nv := 0;
        for i = 0 to nf - 1 do
                tp.(i) <- {score = 0.; pos = init a b; vit = init 0. 1.; dd = 0.}; tp.(i).score <-  f tp.(i).pos
        done;
        nv := !nv + nf; ng := 0; let m = ref (meilleur tp nf) in
        while !nv < max & eps <. tp.(!m).score do
                (* le vecteur tp.(i) a pour fils x calculé grâce aux tp d'indices u, v, w *)
                for i = 0 to nf - 1 do
                        let u, v, w, x = random__int nf, random__int nf,
                                random__int nf, make_vect dim 0. in
                        for j = 0 to dim - 1 do
                        x.(j) <-  if random__float 1. < pc
                                then tp.(w).pos.(j)+.khi*.(tp.(u).pos.(j)-.tp.(v).pos.(j))
                                else tp.(i).pos.(j)
                        done;
                        let k = random__int dim (* facultatif *)
                        in x.(k) <- tp.(w).pos.(k) +. khi *. (tp.(u).pos.(k) -. tp.(v).pos.(k));
                        let s = f x in if s < tp.(i).score then (tp.(i).score <-  s; tp.(i).pos <- x)
                done;
                nv := !nv + nf; incr ng; m := meilleur tp nf;
                if visu then (voir !m; graph 100. a b)
        done;
        if visu then (nl (); close_graph ()); !nv;;
```

Le paramètre χ est recommandé est 1. Notons que χ pourrait être plus grand que 1 ce qui pourrait faire sortir les individus de l'espace de recherche et d'autre part, les résultats que nous avons trouvé sont moins bons avec de plus petites valeurs pour ce paramètre.
Les expériences montrent des résultats similaires en variant p_C, aussi, celui-ci a été fixé à 0.5.

De plus, on obtient de meilleurs résultats avec une modification appelée "DE homogène", procédure identique mis à part que chaque composante est modifiée grâce à la probabilité p_C, le choix de k devenant inutile.On lance par exemple :

```
de 500 1e-06 parab (-5.) 5. 0.5 0.5 true;;
de 50000 1e-06 ros (-2.) 2. 0.5 0.5 true;;
```

$\mu = 100$	MGA	PSO géom	PSO social	SOMA r=(b-a)/5	SOMA (b-a)/10	DE hom	DE comp	ESAO 33%	ESAO 33% elim
De Jong dim 3	4880	3030	2060	2854	3251	2775	2450	765	853
Parabola dim 3	5261	6545	8790	6081	5304	7630	6775	1310	2183
... ... dim 10	max	max	max	max	max	46265	49265	12880	11204
Tripod dim 2	max	14210	12685	28405	max	16120	11525	9800	11697
Rosenbrok dim 2	22658	3855	4575	32558	46059	20240	10185	20645	15939
Rastrigin dim 2	23550	32705	33225	16552	15245	10815	8395	1525	2546
... ... dim 10	max	max	max	max	max	max	max	3320	2465
Griewank dim 2	47856	95295	85725	16522	18470	9710	7110	5475	3070
... ... dim 10	max	max	max	max	max	max	max	13826	3914

Comparaisons du nombre moyen d'évaluations de 9 fonctions f à minimiser (moyenne sur 20 runs) pour atteindre un seuil de 10^{-4}. Les différentes techniques sont grossièrement classées. Un bon algorithme génétique élitiste avec élimination reste meilleur que toutes ces heuristiques.

Voir : Price K. Storn R. Differential evolution, Dr Dobb's Journal 1997

Storn P. Price K. *Differential evolution a simple and efficient adaptative scheme for global optimization over continuous spaces,* Global Optimization vol 11 p341-359, 1997 (www.icsi.berkeley.edu/~storn)

Storn R. *On the usage of differential evolution for function optimization,* NAFIPS p519-523, 1996

8. Optimisation du voyageur par colonie de fourmis AS(n, m, α, β, ρ, Q)

Le problème du voyageur de commerce est représenté par un graphe dont les n noeuds représentent les villes (il faut au moins n = 30).
Les arcs sont pondérés par les distances d_{ij} entre ces villes ($\eta_{ij} = 1 / d_{ij}$ est apellée la visibilité entre les noeuds i et j) et par une autre pondération $\tau_{ij}(t)$ qui représente la trace de phéromone laissée par les fourmis. Soient $b_i(t)$ le nombre d'agents au noeud i à l'instant t et $m = \sum b_i(t)$ le nombre total de fourmis.
AS pour "Ant system" lorsque la mise à jour des traces est synchrone, une autre façon de voir est le "système de colonies de fourmis" où la mise à jour est immédiate, donc asynchrone.

On prendra $\alpha = 1$, $\beta = 7$, $\rho = 0.5$ ou mieux 0.25, Q = 100 ou plus (1 - ρ = évaporation).

On appelle génération une succession de n-1 étapes où chaque fourmi aura visité n villes, initialement par exemple si m = 2n, $b_i(t) = 2$ (2 fourmis par villes, c'est à dire m = 2n, semble empiriquement optimal) et $\tau_{ij}(0)$ aléatoire entre 0 et 1 ou bien initialisé par le réel n/L (en ce cas Q=1) avec L longueur de visite des n villes en prenant toujours le plus proche voisin.

Chaque agent k possède une liste "tabou" TL_k des villes déjà visitées et se meut de la façon suivante grâce à la probabilité qu'il passe du noeud i à j :

$$p_{ij}(t) = \text{si } j \in TL_k(t) \text{ alors 0 sinon } (\tau_{ij}(t))^{\alpha}.(\eta_{ij})^{\beta} / \sum_{h \notin TLk(t)} (\tau_{ih}(t))^{\alpha}.(\eta_{ih})^{\beta}$$

Pour chaque ville i et chaque fourmi k présente en i, si $TL_k(t)$ n'est pas complète, on choisit la ville j aléatoirement parmi celles de plus forte probabilité, j est empilé dans $TL_k(t)$ pour former $TL_k(t+1)$, et la fourmi k passe en j.
Les options de mise à jour sont (par ordre d'efficacité) : $\tau_{ij}(t+1) = \rho.\tau_{ij}(t) + \sum_k$ passe de i à j Δ_{ij}

1 Ant-density model : Δ_{ij} est toujours la même quantité Q
2 Ant-quantity model : $\Delta_{ij} = Q / d_{ij}$ d'autant plus grande que la distance est courte.
3 Ant-cycle model : Δ_{ij} n'est mis à jour qu'après un cycle de n étapes par Q / L_k où L_k est la longueur du tour accompli par l'agent k.
4) ou bien par Q / L* où L* est la longueur du meilleur agent de la génération antérieure (empiriquement meilleur d'après [Boryczka 98]).

En fait toutes ces options de mise à jour sont discutables car on mélange des grandeurs non homogènes, nous avons pris la première avec Q = 1. Le règlage des paramètres de l'algorithme n'est pas non plus aisé, par exemple, l'évaporation peut être entre 0.25 et 0.75 sans grandes différences.

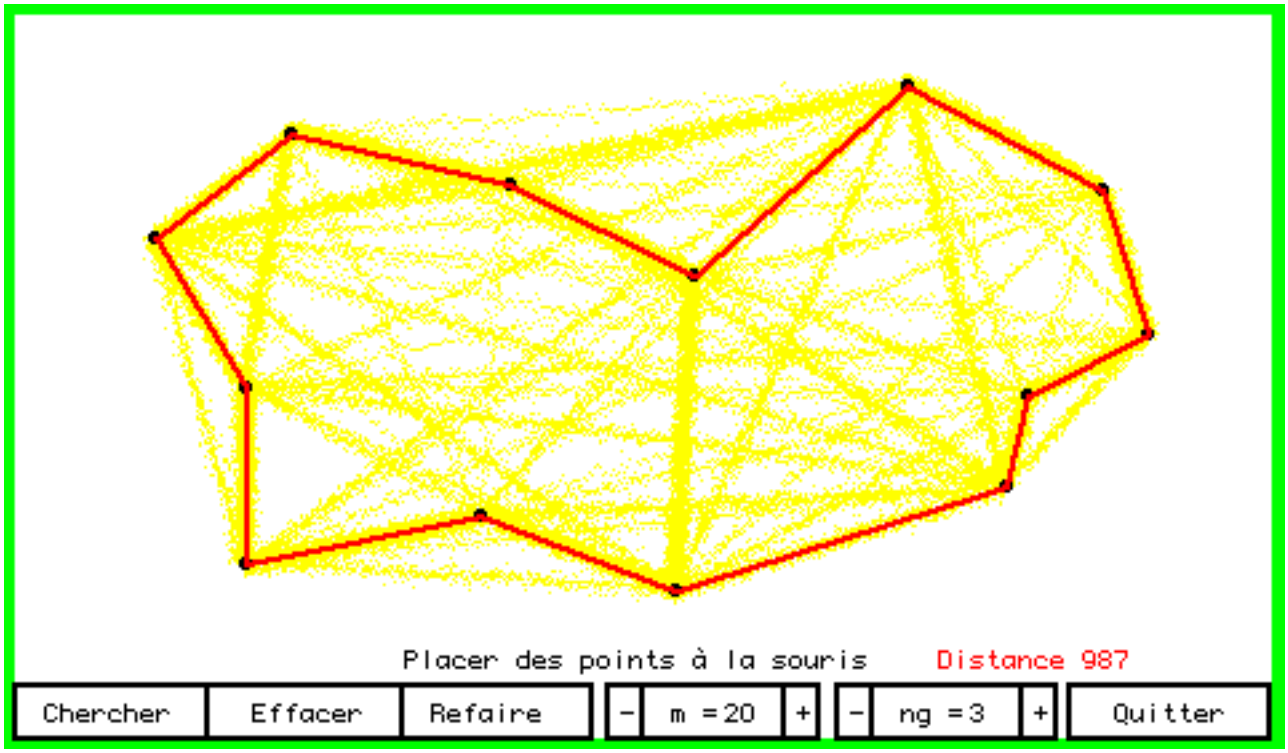
Après n-1 étapes, (une génération), la meilleure liste Tabou (plus petite distance cumulée) est mémorisée et une autre génération est reconduite.

Une fourmi sera ici un article comprenant un numéro de ville et une liste taboue contenant toutes les villes traversées, y compris le retour, donc n+1 éléments pour n villes.

```
let rec  sqr x = x *. x  and dist (x, y) (x', y') = sqrt (sqr (float_of_int (x - x')) +. sqr (float_of_int (y - y')));;
let rec expo a n = if n = 0 then 1. else a *. (expo a (n - 1)) and maxf x y = if x <. y then y else x;;
let round x = if x >=. 0. then int_of_float(x +. 0.5) else -(int_of_float (-. x +. 0.5));;  (* arrondi *)
let rec nth q n = if n = 0 then hd q else nth (tl q) (n - 1) and last = fun [x] -> x | (_ :: q) -> last q;;

let rec complete n lt = if n < 0 then true else if mem n lt then complete (n - 1) lt else false;;
        (* teste si les villes de 0 à n sont dans la liste lt *)

let hasard q = nth q (random__int (list_length q))
and has a b = a + random__int (b - a + 1)  ;;

let gauss m s = let som = ref 0 in
        for i = 1 to 12 do som := !som + has (-100) 100 done;
        m + (s * (!som)) / 200;; (* donne un entier avec m et s entiers *)

type fourmi = {mutable pos : int; (* numéro de ville *) mutable tabou : int list};;

#open "graphics";; (* Ecran 0*480 sur 0*240 *)
let orange = 16737792;;

let place x y c = set_color c; moveto x y; lineto x y ;;

let point (x, y) c = place x y c and trait (x, y) = lineto x y;;

let rec zigzag (x, y) (x', y') c =
        (* simule une marche de fourmi de couleur c entre 2 points *)
        if x' < x then zigzag (x', y') (x, y) c
        else if abs (y - y') < x' - x
                then for u = x to x' do
                place u (gauss (y + (u - x) * (y' - y) / (x' - x)) 2) c done
        else for  v = min y y' to max y y' do place  (gauss (x + (v - y) * (x' - x) / (y' - y)) 2) v c done;;
```

```
let rec dessine  tv c = fun [_] -> () | [] -> () | (i :: j :: q) -> (point tv.(i) c; trait tv.(j);
        dessine  tv c (j :: q));;
        (* dessine est bien sûr plus rapide que "zigzag" mais d'un moins bel effet *)

let rec circuit  tv c = fun [_] -> () | [] -> () | (i :: j :: q) -> (zigzag tv.(i)  tv.(j) c; circuit  tv c (j :: q));;

let choix n prob =
        (* renvoie un numero de ville suivant sa proba, calcul avec proba cumulées *)
        let k = ref 0 and fin = ref false and p = ref 0. in
        while not (!fin) & !k < n do
                p := !p +. prob.(!k);
                if prob.(!k) = 0. then incr k
                else ( if random__float 1. < !p then fin := true else incr k)
        done;
        !k;;

let initv lv  = let n = list_length lv and tv = vect_of_list lv
        (* tdv matrice symétrique de diagonale nulle *)
        in let tdv = make_matrix n n 0.
        and tau = make_matrix n n 0. in
        for i = 0 to n-1 do for j = 0 to n-1 do
                tdv.(i).(j) <- dist tv.(i) tv.(j);
                tau.(i).(j) <- random__float 1.
        done done;
        n, tv, tdv, tau;;
        (* renvoie le quadruplet de tout ce qui concerne la répartition des n villes *)

let initf m n tv =  (* place fourmis sur villes aléatoires *) let tf = make_vect m {pos = 0; tabou = []}
        in for k = 0 to m-1 do let v = random__int n in tf.(k) <-  {pos = v; tabou = [v]} done;
        tf;;

let rec longueur acc td = fun [_] -> acc | [] -> acc | (i :: j :: q) -> longueur (acc +. td.(i).(j)) td (j :: q);;
        (* longueur d'un circuit à l'aide du tableau des distances td *)

let longopt m tf td =  (* renvoie la liste des num de villes de la fourmi optimale et sa distance *)
        let f = ref 0 and opt = ref (longueur 0.  td (tf.(0).tabou)) in
        for k = 1 to m - 1 do
                let nl = longueur 0. td (tf.(k).tabou) in
                if nl < !opt then (opt := nl; f := k) done;
        tf.(!f).tabou, !opt;;


let transition ro bt n m opt tf tau tv  tdv =
        let prob = make_vect n 0.
        and  temp = make_matrix n n 0. in
        for k = 0 to m - 1 do (* k = numéro de fourmi à la ville i  opt=long optimale antérieure *)
                let i = tf.(k).pos in let d = ref 0. in (* calcul du dénominateur d des proba *)
                for j = 0 to n-1 do
                        if not (mem j tf.(k).tabou) then d := !d +. tau.(i).(j) /. (expo (tdv.(i).(j)) bt)
                done;
                let vc = if complete (n-1) tf.(k).tabou then last (tf.(k).tabou)  (* vc = ville choisie *)
                else (for j = 0 to n-1 do
                        (* calcul des proba pour la fourmi k, puis son mouv. de i vers vc : *)
                        prob.(j) <- if mem j tf.(k).tabou then 0.
                                else (tau.(i).(j)) /.(expo (tdv.(i).(j)) bt) /. !d done;
                        choix n prob) in temp.(i).(vc) <- 1.+.temp.(i).(vc);
                        tf.(k) <- {pos = vc; tabou = vc :: tf.(k).tabou}  done;
                        (* puis mise à jour des pheromones : *)
                        for i = 0 to n - 1 do for j = 0 to n - 1 do
                                tau.(i).(j) <- maxf 0.01 (ro *. tau.(i).(j) +.temp.(i).(j))  done
        done;;
        (* les mises à jour sont synchrones *)
```

```
let generations ro bt m lv ng =
        (* réalise l'option 4 pour un nombre ng de générations enchaînées *)
        let n, tv, tdv, tau = initv lv in let opt = ref 0. in
        for i = 0 to n - 2 do opt := !opt +. tdv.(i).(i + 1) done;
        opt := !opt +. tdv.(n - 1).(0);
        let solg = ref [] and optg = ref !opt in
        for g = 1 to ng do
                let tf = initf m n tv in
                for t = 1 to n do transition ro bt n m !opt tf tau tv tdv
                done; (* n transitions *)
                for k = 0 to m - 1 do circuit tv yellow (* ou encore : orange + (k * yellow / m) *)
                tf.(k).tabou done;
                let best = longopt m tf tdv in opt := snd best;
                if !opt < !optg then (solg := fst best; optg := !opt); set_color black;
                for v = 0 to n - 1 do let (x, y) = tv.(v) in fill_ellipse x y 2 2 done
        done;
        set_line_width 2; dessine tv red !solg; set_line_width 1;
        set_color white; fill_rect 350 28 100 10;
        set_color red; moveto 350 29;
        draw_string ("Distance " ^ (string_of_int (round !optg)));;
```

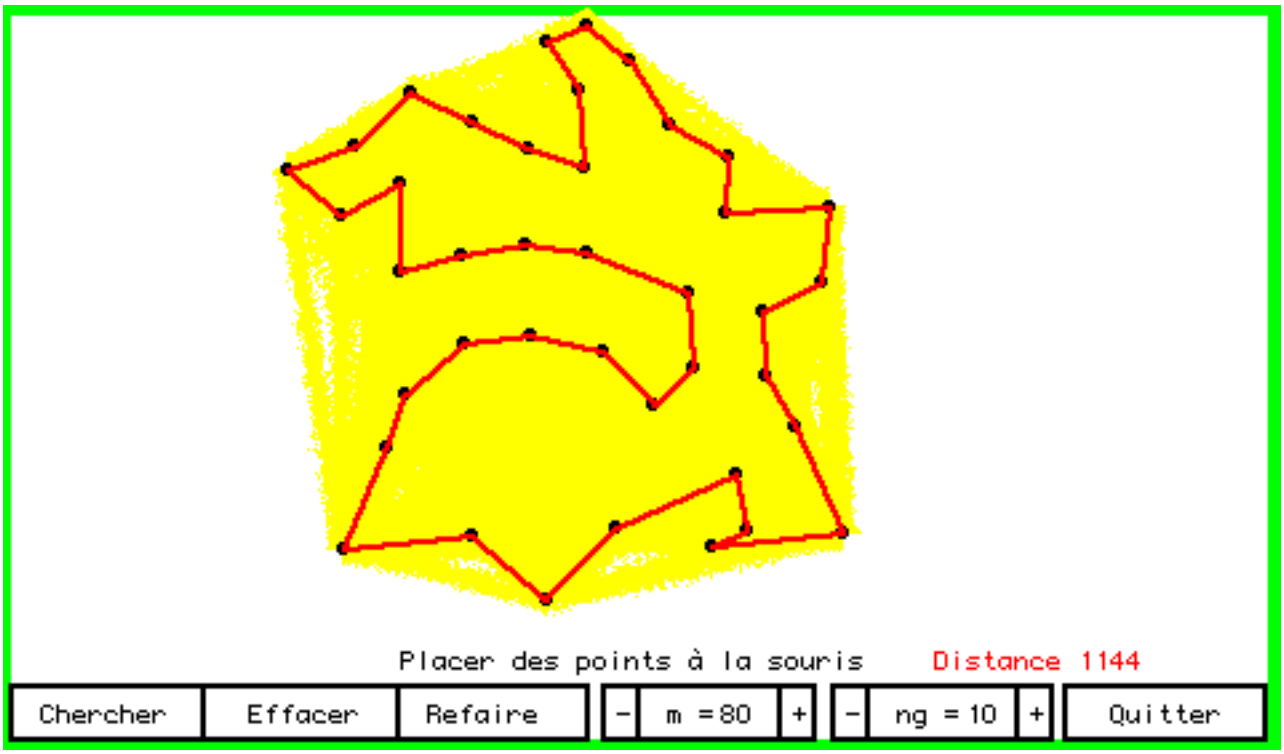


Exemple facile avec 13 villes, 20 fourmis, 3 générations.

L'interface graphique permettant de placer les villes à la souris et montrant l'évolution des traces de phéromone, est réalisée avec les "boutons" suivants, leur placement à l'écran avec "boite" et la grande fonction "villes" dont l'essentiel, en fait, est d'appeler "generations". Le dessin en lui-même peut être fait par exemple par coloration plus ou moins accentuée ou, comme ici, segments plus ou moins larges.

```
exception Exit;;

let cadre () = set_color green; fill_rect 0 0 480 280;  set_color white;
        fill_rect 4 4 471 271; set_line_width 1;;
```

```
let quitter x y =  400 < x & y < 25

and chercher x y = x < 75 & y < 25

and effacer x y = 75 < x & x < 148 & y < 25

and refaire  x y = 148 < x & x < 225 & y < 25

and mmoins x y = 225 < x & x < 238 & y < 25

and mplus x y = 291 < x & x < 311 & y < 25

and gmoins x y = 311 < x & x < 324 & y < 25

and gplus x y = 380 < x & x < 395 & y < 25;;


let boite x y l h c = set_color black; fill_rect x y l h; set_color white;
        fill_rect (x + 2) (y + 2) (l - 4) (h - 4) ;
        moveto (x+6) (y+6); set_color black; draw_string c;;

let rec villes () = open_graph""; cadre(); boite 3 3 75 22 " Chercher ";
        boite 75 3 75 22 "  Effacer ";
        boite 148 3 73 22 " Refaire"; boite 225 3 15 22 "- "; boite 238 3 55 22 " m = ";
        boite 291 3 15 22 "+ ";  boite 311 3 15 22 "- "; boite 324 3 58 22 " ng = ";
        boite 380 3 15 22 "+ "; boite 398 3 77 22 "  Quitter ";
        moveto 150 29; draw_string "Placer des points à la souris";
        let lp = ref [] and m = ref 15 and ng = ref  5 and drap = ref false
        in      (* liste des points, m fourmis, ng générations *)
try while true do moveto 270 9; set_color black; draw_string (string_of_int !m);
                                moveto 362 9; draw_string (string_of_int !ng);
        let e = wait_next_event [Button_down] in
        if e.button then let (x, y) = mouse_pos() in
                if effacer x y then villes()
                else if chercher x y then (drap := true; generations 0.5 7 !m !lp !ng)
                else if quitter x y then raise Exit
                else if refaire x y  & !drap then generations 0.5 7 !m !lp !ng
                else if mmoins x y then
                (m := max 1 (!m - 1); set_color white; fill_rect 270 5 15 18; set_color black)
                else if mplus x y then
                        (m := min 99 (!m + 1); set_color white; fill_rect 270 5 15 18; set_color 0)
                else if gmoins x y then
                        (ng := max 1 (!ng - 1); set_color white; fill_rect 360 5 15 18; set_color 0)
                else if gplus x y then
                        (ng := min 99 (!ng + 1); set_color white; fill_rect 360 5 15 18;
                        set_color black)
                else (lp := (x, y) :: (!lp); set_color  black; fill_ellipse x y 2 2)
        done
with Exit -> close_graph();;
```



Exemple avec 35 villes et 70 fourmis, ce n'est pas l'optimum global

Boryczka M. *Some aspects of ant systems for the travelling salesman problem,* Fund. Informaticae 35 p197-209 1998

Colorni A. Dorigo M. Maniezzo U. *Distributed optimization by ant colonies,* European Conference on Artificial Life p134-142, 1991

Colorni A. Dorigo M. Maniezzo U. *An investigation of some properties of an ant algorithm,* PPSN p509-520, 1992

Dorigo M. Gambardella L.M. *A study of some properties of ant-Q,* PPSN p656-665 Springer Verlag 1996

Dorigo M. Gambardella L.M. *Ant colonies for the traveling salesman,* rapport IRIDIA Univ. libre de Bruxelles 1997

Goss S. Beckers R. Deneubourg J.L. Aron S. Pasteels J.M. *How trail laying and trail following can solve foraging problems for ant colonies,* Behavioural mechanisms of food selection, NATO ASI series G20, Springer Verlag 1990

SOMMAIRE